Operating system support for warehouse-scale computing

Malte Schwarzkopf

University of Cambridge
Computer Laboratory
St John’s College

October 2015

This dissertation is submitted for the degree of Doctor of Philosophy
Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation is not substantially the same as any that I have submitted, or, is being concurrently submitted for a degree or diploma or other qualification at the University of Cambridge or any other University or similar institution.

I further state that no substantial part of my dissertation has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University of similar institution.

This dissertation does not exceed the regulation length of 60,000 words, including tables and footnotes.
Operating system support for warehouse-scale computing

Malte Schwarzkopf

Summary

Computer technology currently pursues two divergent trends: ever smaller mobile devices bring computing to new everyday contexts, and ever larger-scale data centres remotely back the applications running on these devices.

These data centres pose challenges to systems software and the operating system (OS): common OS abstractions are fundamentally scoped to a single machine, while data centre applications treat thousands of machines as a “warehouse-scale computer” (WSC). I argue that making the operating system explicitly aware of their distributed operation can result in significant benefits.

This dissertation presents the design of a novel distributed operating system for warehouse-scale computers. I focus on two key OS components: the OS abstractions for interaction between the OS kernel and user-space programs, and the scheduling of work within and across machines.

First, I argue that WSCs motivate a revisit of the 1980s concept of a distributed operating system. I introduce and justify six design principles for a distributed WSC OS. “Translucent” primitives combine transparency and control: they free users from being explicitly aware of object locality and properties, but expose sufficient optional hints to facilitate optimisations. A novel distributed capability scheme ensures fine-grained isolation, with cryptographic capabilities being treated as data and segregated capabilities used as temporary handles. Multi-phase I/O with kernel-managed buffers improves scalability and security at the system call level, but also permits the implementation of diverse application-level consistency policies.

Second, I present the DIOS operating system, a realisation of these design principles. The DIOS system call API is centred around distributed objects, globally resolvable names, and translucent references that carry context-sensitive object meta-data. I illustrate how these concepts are used to build distributed applications, and describe an incrementally deployable DIOS prototype.

Third, I present the Firmament cluster scheduler, which generalises prior work on scheduling via a minimum-cost flow optimisation. Firmament can flexibly express many scheduling policies using pluggable cost models; it makes accurate, high-quality placement decisions based on fine-grained information about tasks and resources; and it scales to very large clusters by optimising the flow network incrementally.

Finally, I demonstrate that the DIOS prototype achieves good performance in micro-benchmarks and on a data-intensive MapReduce application, and that it offers improved cross-machine isolation between users’ applications. In two case studies, I show that Firmament supports policies that reduce co-location interference between tasks and that exploit flexibility in the workload to improve the energy efficiency of a heterogeneous cluster. Moreover, Firmament scales the minimum-cost flow optimisation to very large clusters while still making rapid decisions.
Acknowledgements

“I find Cambridge an asylum, in every sense of the word.”
— attributed to A. E. Housman [Ric41, p. 100].

My foremost gratitude extends to my advisor, Steve Hand, for his help and support over the course of the past six years. Steve’s enthusiasm, encouragement, patience, and high standards have impacted my journey into systems research as much as they have shaped my thesis. Steve also took the time to comment on countless drafts of this work and regularly talked to me about it at length, even as he himself moved between jobs and continents.

Likewise, I am grateful to Ian Leslie, my second advisor, who gave insightful feedback on drafts of this document, and gave me the space and time to finish it to my satisfaction. In the same vein, I am also indebted to Frans Kaashoek and Nickolai Zeldovich for their seemingly infinite patience during my longer-than-expected “final stretch” prior to joining the MIT PDOS group.

Other current and former members of the Systems Research Group have also supported me in various ways. I am grateful to Ionel Gog, Richard Mortier, Martin Maas, Derek Murray, Frank McSherry, Jon Crowcroft, and Tim Harris for comments that have much improved the clarity of this dissertation. Moreover, I owe thanks to Robert Watson for our discussions on security and capabilities, and for “adopting” me into the MRC\(^2\) project for two years; and to Anil Madhavapeddy, Andrew Moore, and Richard Mortier, who assisted with equipment and feedback at key moments.

Ionel Gog and Matthew Grosvenor deserve huge credit and gratitude for our close collaboration and camaraderie over the years. Our shared spirit of intellectual curiosity and deep rigour, as well as the humour to go with it, embodies everything that makes systems research enjoyable.

I have also been privileged to work with several enthusiastic undergraduate and master’s students: Adam Gleave, Gustaf Helgesson, Matthew Huxtable, and Andrew Scull all completed projects that impacted my research, and I thank them for their excellent contributions.

The ideas for both systems presented in this dissertation go back to my internship at Google, where I worked with Andy Konwinski, John Wilkes, and Michael Abd-El-Malek. I thank them for the insights I gained in our collaboration on Omega and in the Borgmaster team, which made me realise the opportunities for innovative systems software to support the massive compute clusters deployed at internet companies.

In addition to those already mentioned above, I am grateful to my other friends in the systems research community – especially Allen Clement, Natacha Crooks, Arjun Narayan, Simon Peter, Justine Sherry, and Andrew Warfield – as well as my family, whose support, opinions and counsel have impacted my work in many ways, and who I value immensely.

Finally and above all, I thank Julia Netter, who has accompanied this journey – from its very beginning to the final proofreading of this document – with wit, insight, and loving support.
# Contents

1 Introduction 17  
1.1 Background ................................................. 19  
1.2 Why build a new *operating system*? ...................... 20  
1.3 Contributions .............................................. 22  
1.4 Dissertation outline ...................................... 24  
1.5 Related publications ...................................... 27  

2 Background 29  
2.1 Operating systems .......................................... 30  
2.2 Warehouse-scale computers ................................ 38  
2.3 Cluster scheduling ......................................... 52  

3 Designing a WSC operating system 59  
3.1 Design principles ........................................... 60  
3.2 Distributed objects ......................................... 62  
3.3 Scalable, translucent, and uniform abstractions ........... 64  
3.4 Narrow system call API ..................................... 66  
3.5 Identifier and handle capabilities ......................... 68  
3.6 Flat persistent object store ................................ 73  
3.7 Request-based I/O with kernel-managed buffers .......... 74  
3.8 Summary ..................................................... 82  

4 D1OS: an operating system for WSCs 83  
4.1 Abstractions and concepts .................................. 84  
4.2 Objects ..................................................... 88
## CONTENTS

4.3 Names ................................................................. 90  
4.4 Groups ............................................................... 97  
4.5 References ........................................................... 99  
4.6 System call API ....................................................... 103  
4.7 Distributed coordination ............................................ 106  
4.8 Scalability ............................................................. 108  
4.9 Prototype implementation ........................................... 109  
4.10 Summary ............................................................... 114  
5 Firmament: a WSC scheduler .......................................... 117  
5.1 Background .......................................................... 118  
5.2 Scheduling as a flow network ........................................ 121  
5.3 Implementation ....................................................... 128  
5.4 Scheduling policies .................................................... 133  
5.5 Cost models ........................................................... 138  
5.6 Scalability ............................................................. 155  
5.7 Summary ............................................................... 159  
6 Evaluation ............................................................... 161  
6.1 Experimental setup ................................................... 161  
6.2 DIOS ................................................................. 163  
6.3 Firmament ............................................................. 174  
6.4 Summary and outlook ............................................... 192  
7 Conclusions and future work ........................................... 193  
7.1 Extending DIOS ....................................................... 194  
7.2 Evolving Firmament ................................................ 197  
7.3 Summary ............................................................... 198  
Bibliography ............................................................. 199  
A Additional background material ...................................... 236  
A.1 Operating system specialisation ................................... 236  
A.2 Additional workload interference experiments .................. 242  
A.3 CPI and IPMA distributions in a Google WSC ................... 248
# List of Figures

1.1 Outline of topics that have influenced my research. .................................. 26  
2.1 The Google infrastructure stack. .......................................................... 39  
2.2 The Facebook infrastructure stack. ...................................................... 40  
2.3 Machine types and configurations in a Google WSC. .............................. 44  
2.4 Micro-benchmarks on heterogeneous machine types. .............................. 45  
2.5 Micro-architectural topologies of the systems used in my experiments. ...... 46  
2.6 WSC application interference on different CPU platforms. ....................... 48  
2.7 WSC application interference of batch workloads on a shared cluster. ...... 49  
2.8 Comparison of different cluster scheduler architectures. ......................... 54  
3.1 Results of exploratory system call tracing on common WSC applications. . 67  
3.2 Sequence diagrams of read and write I/O requests. ............................... 75  
3.3 Example I/O requests with different concurrent access semantics. ............ 77  
3.4 Stages of an I/O request. ................................................................. 78  
3.5 Handling an I/O request to a remote object. ........................................ 79  
3.6 Buffer ownership state transitions in the DIOS I/O model. ....................... 81  
4.1 Schematic example of different DIOS abstractions. ............................... 85  
4.2 DIOS MapReduce application example structure. ................................. 86  
4.3 DIOS web-service application example structure. ................................. 87  
4.4 A DIOS object and its relations to names and references. ....................... 88  
4.5 Illustration of the DIOS name resolution process. ................................ 95  
4.6 Structure of a DIOS reference. ......................................................... 99  
4.7 Re-order matrix of DIOS system calls. ............................................... 107
LIST OF FIGURES

4.8 DIOS prototype structure by example. .................................. 112
5.1 Example scheduling flow network. ........................................ 122
5.2 Example flow network with capacities. ................................. 124
5.3 Example flow network with arc cost scopes. ......................... 125
5.4 Equivalence class aggregator example. ................................. 127
5.5 Examples of different Firmament multi-coordinator setups. ........ 129
5.6 High-level structure of the Firmament coordinator. .................. 130
5.7 Example Firmament flow network with machine topology information. 131
5.8 Representation of gang scheduling requests in the flow network. ... 136
5.9 Example flow network for the Quincy cost model. .................... 139
5.10 Example flow network for the Whare-Map cost model. .............. 141
5.11 Example flow network for the CoCo cost model. ...................... 145
5.12 Overview of the energy-aware scheduling setup. ..................... 151
5.13 Example flow network for the Green cost model. .................... 153
6.1 Object creation latency micro-benchmark. ............................. 164
6.2 DIOS task spawn performance micro-benchmark. ...................... 166
6.3 DIOS shared-memory I/O performance. ................................. 167
6.4 DIOS remote I/O request throughput. ................................ 167
6.5 MapReduce WordCount application performance in DIOS. ........... 170
6.6 Workloads normalised runtime in Firmament cluster mix experiment. 177
6.7 Firmament’s Whare-Map cost model learning good mappings. ....... 179
6.8 Task wait time with different schedulers in cluster mix experiment. 180
6.9 CPI distribution for different schedulers in cluster mix experiment. 181
6.10 Energy consumption for different schedulers and cost models. ...... 184
6.11 Energy savings obtained by auto-scaling service jobs. ............... 185
6.12 Firmament scalability experiments. .................................. 190
A.1 Interference between SPEC CPU2006 workload pairs. ............... 243
A.2 Co-located WSC application cycle counts on different CPU platforms. 244
A.3 Co-located WSC application cache misses on different CPU platforms. 245
LIST OF FIGURES

A.4  $n$-way interference of SPEC CPU2006 benchmarks. .......................... 246
A.5  CPI and IPMA distributions in the 2011 Google trace. ...................... 249
B.1  DIOS prototype integration with Linux. .......................................... 267
C.1  Example of the minimum-cost, maximum-flow optimisation problem. .... 269
C.2  Example flow network with capacities. .......................................... 273
C.3  Example flow network for the Quincy cost model. ......................... 275
# List of Tables

2.1 Classic distributed OSes and their properties. ................................ 33  
2.2 Machine heterogeneity micro-benchmark workloads. ...................... 44  
2.3 Machine configurations used in the heterogeneity micro-benchmarks. .......... 45  
2.4 WSC applications used in pairwise interference experiments. .............. 47  
2.5 WSC batch workloads used in scale-up interference experiment. .......... 50  
2.6 WSC cluster schedulers and their key design goals ........................ 53  

3.1 Previous systems’ distributed capability schemes compared to DIOS. .... 70  
3.2 Concurrent access semantics choices for I/O requests. ...................... 76  

4.1 Object types supported in the DIOS prototype. ............................ 89  
4.2 A task’s capabilities depending its on relationship with a group. ........... 97  
4.3 Example DIOS reference attributes and their properties. ................. 102  
4.4 Prototype DIOS system call API. ........................................ 104  
4.5 Changes made to the Linux host kernel for the DIOS prototype. .......... 110  
4.6 Binary types supported in the DIOS prototype. .......................... 113  

5.1 Comparison of cluster scheduling, CPU scheduling, Quincy, and Firmament. . . 119  
5.2 Common general costs used in Firmament’s flow network. .................. 126  
5.3 Task profiling metrics tracked by the Firmament coordinator. ............. 132  
5.4 Cost parameters in the Whare-Map cost models and their roles. ........... 142  
5.5 Summary of cost models and their features. ............................. 154  
5.6 Complexities of minimum-cost, maximum-flow solver algorithms. ......... 156  

6.1 Evaluation cluster and machine specifications. ................................ 162  
6.2 DIOS security properties compared to other isolation techniques. .......... 172
<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3</td>
<td>Synthetic workloads used in cluster mix experiments.</td>
<td>175</td>
</tr>
<tr>
<td>6.4</td>
<td>Machine specifications for energy-aware scheduling experiments.</td>
<td>182</td>
</tr>
<tr>
<td>6.5</td>
<td>Machine power consumption at different CPU load levels.</td>
<td>182</td>
</tr>
<tr>
<td>6.6</td>
<td>Performance constraint satisfaction in the Green cost model.</td>
<td>186</td>
</tr>
<tr>
<td>6.7</td>
<td>Firmament’s flexibility to support existing schedulers’ policies.</td>
<td>187</td>
</tr>
<tr>
<td>C.1</td>
<td>Edge capacity parameters for different arc types in the Quincy scheduler.</td>
<td>273</td>
</tr>
<tr>
<td>C.2</td>
<td>Cost parameters in the Quincy cost model and their roles.</td>
<td>275</td>
</tr>
<tr>
<td>C.3</td>
<td>Parameter values used in the Quincy cost model.</td>
<td>276</td>
</tr>
</tbody>
</table>
Listings

4.1 DIOS object creation example. .................................................. 93
4.2 DIOS name resolution example. .................................................. 96
4.3 DIOS reference picking example. .................................................. 101
4.4 Example of using DIOS select(2) to multiplex synchronous I/O. .......... 105
5.1 Excerpt of Firmament’s CoCo cost model resource fitting code. ............ 147
6.1 User-provided code in the DIOS MapReduce WordCount implementation. .. 169
7.1 Distributed computation of fib(10) in Rust on DIOS. .......................... 195
B.1 Example DIOS I/O request for exclusive access to a blob. .................... 264
B.2 Example DIOS I/O request buffer re-use in a MapReduce reducer. .......... 265
C.1 Firmament cost model API. ......................................................... 279
Chapter 1

Introduction

“Go back to thinking about and building systems. Narrowness is irrelevant; breadth is relevant: it’s the essence of system. Work on how systems behave and work, not just how they compare. Concentrate on interfaces and architecture, not just engineering. Be courageous. Try different things; experiment.” — Rob Pike, “Systems Software Research is Irrelevant” [Pik00, sl. 20].

Operating systems have been an essential part of our computing infrastructure for decades: they provide crucial abstractions, enable safe and portable use of hardware resources, support multi-programming, and enforce isolation between programs, users, and machine resources.

Since the late 1970s, computer architecture has kept moving from large “mainframes” towards ever smaller, more affordable machines and devices. Today, however, we witness the return of mainframe-like, large-scale computer installations: “warehouse-scale” data centres composed of thousands of commodity computers [BCH13, pp. 1–5]. These installations are required to support applications that either cannot function on a single machine due to their large resource demands, or which require distribution to ensure service availability and fault-tolerance.

Modern operating systems, however, are designed for singular devices, such as smartphones, laptops, and individual servers. Hence, existing commodity operating systems are ill-matched to large data centres: their abstractions assume a “one machine, several users” model, rather than the “many machines, many users” reality of a WSC. Distributed and local operations use fundamentally different abstractions, and OS architectures assume that off-chassis communication is comparatively slow, costly, and rare.

To make matters worse, the distributed infrastructure of a data centre is by necessity complex, shared, and highly multi-programmed. The sensitivity of the data processed demands strong isolation and controlled information flow, while component failures and complex multi-dimensional interactions between applications threaten availability and fault-tolerance. Nevertheless, system users and programmers demand to be shielded from this complexity: the most
successful distributed systems are often those which are fully transparent\textsuperscript{1} and hide the details of distributed communication and coordination.

In other words, we must offer the simplest possible abstractions to programmers, while constructing far larger and more complex distributed systems than ever before, handling unprecedentedly sensitive personal data – all without any help from the operating system.

Indeed, as its scope is limited to the local machine, the OS fails to fulfil two of its key responsibilities: first, it loses its ability to make good resource allocation decisions on the basis of global knowledge, and second, the OS relegates responsibility for isolation to user-space runtimes and applications themselves.

In my research for this dissertation, I have developed the concepts and design for a distributed data centre operating system. In this OS, each machine kernel is explicitly part of a larger whole: it is able to address, describe, and interact with remote objects and tasks. I highlight the benefits of this approach by focusing on two key parts of the OS:

1. \textit{The abstractions used within the kernel and exposed to user-space applications}. Better abstractions make distributed systems more secure, more efficient, and easier to build.

2. \textit{The scheduling of work to compute resources}. Combining the global placement of work across the WSC with its machine-level impact makes the “warehouse-scale computer” more efficient, increases utilisation, and saves energy.

Based on my design, I constructed D\textsc{ios}, a prototype distributed operating system for modern data centres, and the Firmament cluster scheduler. I use D\textsc{ios} and Firmament to investigate the following thesis:

Explicit operating system support for distributed operation offers benefits for the efficiency and security of distributed systems in “warehouse-scale” data centres.

Introducing primitives for distributed operation at the system call level, and combining high-level cluster state and fine-grained per-machine information in the long-term scheduler, realises a distributed operating system. This operating system executes prototypical distributed applications at comparable performance to existing commodity operating systems and “middleware”, while granting significant opportunities for further improvement.

In the next section, I summarise why commodity operating systems are insufficient to meet the demands of modern data centre systems (§1.1). I then explain why designing a new operating system – as opposed to application-level “middleware” – helps to address these problems (§1.2). Following, I state the contributions described in this dissertation (§1.3) and outline its overall structure (§1.4). Finally, I list prior publications of parts of the work described and related projects that have impacted it (§1.5).

\textsuperscript{1}Contrary to common usage, “transparent” in distributed systems literature means that implementation details are \textit{hidden} from the user; I stick to this somewhat idiosyncratic definition in this dissertation.
CHAPTER 1. INTRODUCTION

1.1 Background

Many of today’s pre-eminent operating system paradigms and abstractions can be traced back to the first versions of Unix in the early 1970s. Unix was developed for the then-emergent “minicomputers” [RT74], rather than the mainframes that large operations favoured at the time. It succeeded in the workstation and server markets, inspiring widely-used contemporary systems such as GNU/Linux, the BSD family, and Windows NT. Mainframes and their specialised operating systems, by contrast, fell out of favour outside specialised domains.

Consequently, the 1990s popularised a “client-server” architecture in which specialised, server machines dedicated to an application handle requests from many clients. Each server could typically hold all necessary state to serve requests, and each request’s processing needs did not exceed the server’s capacity.

However, as computing devices have become ever smaller, cheaper, and tightly connected to the internet, applications’ server-side back-ends now rely on access to enormous repositories of data, or on computations that exceed the abilities of the local device or a single server.

Such back-end services run in the large-scale data centres operated by internet companies like Google, Amazon, Yahoo!, Facebook, Twitter, or Microsoft. The data centres consist of thousands or tens of thousands of individual server machines. However, unlike classic client-server architectures, request processing no longer happens within a single machine. Instead, it can involve several distributed systems that extend over hundreds or thousands of machines.

Individual machines merely contribute resources to a large pool in this environment. They may join, leave or fail at any time without affecting the overall system, much like hot-swappable processing units in a mainframe or nodes in a peer-to-peer system. Moreover, many different applications and users share the data centre’s resources. To the programmer, however, the data centre is often conceptually abstracted as a single machine of very large capacity, akin to a time-shared mainframe, sometimes called a “warehouse-scale computer” (WSC) [BCH13, pp. 2–5]. In other words, large-scale, mainframe-like “machines” are making a return.

One might expect the operating system in this environment to be quite unlike the Unix-derived server operating systems of past decades. But not so: the individual machines in a WSC today run standard desktop and server operating systems – typically variants of Linux, BSD, or Windows.

This choice, however convenient for backwards-compatibility, gives rise to significant challenges:

1. **Lack of specialisation**: commodity operating systems are designed to support a broad range of use cases, from interactive desktops to highly-loaded servers. Many of the abstractions and techniques used are compromises for generality, rather than a good fit for a known use case.
CHAPTER 1. INTRODUCTION

This leads to inefficiency when a specialised approach could utilise contextual information (e.g. an object-level buffer cache) or offer better security (e.g. mandatory distributed compartmentalisation or data-flow tracking).

2. **Complex mapping of disjoint abstractions**: abstractions for state and data differ between the local OS and the distributed system (e.g. file descriptors, sockets, and memory mappings vs. UUIDs, RPC callbacks, and resilient distributed datasets [ZCD+12]).

This requires extensive translation between abstractions, which impacts scalability, reduces performance, complicates tracing and debugging, and introduces bugs.

3. **Segregated scheduling**: machine-level scheduling decisions (in the OS kernel) are decoupled from global task scheduling (in a cluster-scheduler).

This results in poor control over work placement as the different scheduling levels fail to exchange information to avoid inefficiencies (e.g. negative co-location interference).

4. **Poor access restrictions**: the WSC is a multi-tenant environment and users may accidentally or deliberately run malicious code. WSC operators use virtualisation techniques such as containers and virtual machines to restrict access and contain attacks. These techniques are coarse-grained, and hence must be augmented by application-level security.

This complicates isolation across applications (e.g. restrict leakage of data that a program may legitimately access, such as its inputs), and delegation of work to restricted-privilege helpers (e.g. limiting access to infrastructure systems, such as a key-value store).

Getting the operating system involved can help address all of these challenges. Moreover, a custom OS can complement the efficiency gains that custom-built physical data centre infrastructure, custom machine chassis, rack, and cooling equipment gain their operators [BCH13, pp. 47–65; WSK*09]. Indeed, Google already customize the kernel extensively [WC09]; my work is a logical extension of this customisation.

1.2 Why build a new operating system?

I have argued that the use of commodity OSes in WSC environments comes with a number of drawbacks. A question remains, however, as to whether these challenges necessarily have to be addressed in the OS, especially since current distributed systems are implemented as user-space applications (viz. “middleware”).

The preference for user-space middleware came after a decade of experimentation with distributed OS-level primitives (which I discuss in §2.1.2). Most of these did not survive outside academic research, which might suggest that a new distributed operating system runs the risk
of failing to offer any utility beyond intellectual gratification. Moreover, a huge body of existing code targets well-established abstractions – any new proposals run the risk of breaking compatibility with these.

Indeed, others have suggested solving similar problems in other parts of systems infrastructure:

- Zaharia et al. note the need for operating system functionality that supports data centre applications [ZHK+11]. Their solution is a cluster manager, viz. Mesos [HKZ+11], that runs as user-space middleware, but fulfils the classical operating system tasks of resource arbitration and scheduling.

- Maas et al. envision a holistic language runtime in user-space that transcends machine boundaries and takes the place of the operating system [MAH+14].

- Projects such as Mirage [MMR+13] and OSV [KLC+14] build specialised single-application operating systems that run inside virtual machines atop a hypervisor and enhance safety or performance by specialising the virtualised OS.

Why should we nevertheless look into pushing distributed abstractions back into the operating system, i.e. the kernel?

A key reason why the distributed OS concept is timely again is that distributed operation is a necessity in WSCs, rather than – as in the 1980s – an option useful only to special workloads. Additionally, there are other advantages in targeting the operating system itself:

1. The operating system sets the ultimate rules by which all programs have to abide: in the absence of bugs, its abstractions are impossible to bypass (even maliciously).

2. An OS virtualises resources and may thus present a different, yet internally consistent, view of the system to each program.

3. The privileged operating system has an elevated, omniscient view of resource allocation at all levels of granularity, and can hence make better, globally optimal decisions than individual programs.

4. Application programmers rely on key primitives provided by the operating system: the primitives shape their conceptual thinking.

Indeed, an increasing number of research efforts attempt to shake up the established OS abstractions and re-think the role of the operating system:

- Corey makes OS abstractions scalable to many CPU cores by using per-core data structures by default, with all sharing being explicit and application-driven [BCC+08]. Corey focuses on single-machine scalability, but similar selective sharing approaches can ensure scalability in distributed systems.
fos is a single system image (SSI) OS for many-core machines and cloud deployments, based on message passing between “factored” servers that offer OS services [WGB+10]. Each core runs a simple micro-kernel and OS services consist of multiple spatially scheduled instances, making replication and fine-grained locality explicit.

Barrelfish [BBD+09] is a new OS designed for heterogeneous and potentially non-cache-coherent many-core systems. Based on the premise that scalable operating systems must apply distributed systems techniques [BPS+09], it performs inter-process communication over a range of different channels, including across machines [HGZ11].

The Akaros operating system reduces the transparency of virtualisation, maximising the exposition of system information to applications [RKZ+11]. It provides gang-scheduled multi-core processes (MCPs) that allow applications to enact their own scheduling policies in order to improve the overall efficiency of the system.

Tesselation gives Quality-of-Service (QoS) guarantees to applications using space-time partitioning [CEH+13] and, like Akaros, performs two-level scheduling. Its resource partitioning along with continuous statistics monitoring counters interference between co-located applications at the OS level.

nonkernel [BPA+13], Arrakis [PLZ+14], and IX [BPK+14] remove the OS from the critical path of I/O-intensive operations (“data plane”), and permit applications to interact directly with hardware for improved scalability and performance.

The Andromeda design of “a massively distributed operating system [...] for the commodity cloud” [VKS15] envisages a fully transparent distributed OS based on a minimal pico-kernel and with migratable “fibril” tasks that communicate via unidirectional channels.

My work specialises the OS to a specific domain (warehouse-scale computers). In doing so, it draws on many of the above, as well as on historic distributed operating systems (§2.1.2). DIOS, my distributed OS, is a single system image operating system (like fos and Andromeda), emphasises scalable abstractions (like Corey and Barrelfish), and externalises policy to applications (like Akaros and Tesselation).

1.3 Contributions

This dissertation describes three principal contributions:

1. My first contribution is an analysis of the requirements and a design for a distributed operating system for warehouse-scale computers. I formulate six key properties that such an OS should have, and six principles that help attaining them. From those principles, I
derive core system design decisions: (i) a focus on distributed objects; (ii) the exposition of translucent abstractions; (iii) the use of capability-based protection; (iv) a deliberately narrow system call API; (v) durable storage in a flat object store; and (vi) multi-phase I/O requests with OS-managed buffers. I also introduce the notions of identifier and handle capabilities and show that their combination is sufficient to protect and track objects within a machine and across the distributed system.

2. My second contribution is the D1os operating system, in which I apply these concepts to construct an OS kernel. I have based D1os on typed objects, isolated via namespaces and capabilities, and devised a system call API for user applications to interact with local and remote objects. My D1os namespace and capability design supports low-overhead isolation of distributed applications on shared infrastructure, and, unlike kernel namespace virtualisation in traditional OSes, extends across machines. I also describe how I support translucency in D1os: this enables transparency, while yet exposing sufficient information for application-level optimisation to the user. Moreover, I based the system call API on scalable design principles, and added support for efficient I/O with configurable concurrent access semantics. Finally, I implemented a backwards-compatible D1os prototype based on Linux.

3. My third contribution is the Firmament cluster scheduler. I generalise the approach taken in Quincy [IPC⁺09], which models the scheduling problem as a minimum-cost optimisation over a flow network, and show that it can express most desirable scheduling policy features. With Firmament, I demonstrate that an accurate, flexible and scalable scheduler is possible. I have designed Firmament to act as a centralised or distributed scheduler, and to combine detailed task profiling information with machine architecture data into accurate, high-quality decisions. I have also given Firmament a flexible core to support pluggable scheduling policies. Finally, I demonstrate that – contrary to common prior belief – the flow network optimisation approach is scalable even to large clusters.

All models, architectures, and algorithms described are results of my own work. However, colleagues and students in the Computer Laboratory have at times assisted me in implementing, extending, and evaluating individual components of the prototypes. Wherever this is the case, I have added annotations highlighting the contributions made by others.

In particular, Matthew Grosvenor and I sketched an initial version of the D1os system call API together, and Andrew Scull contributed to the implementation of D1os, adding, in particular, the ELF branding for D1os binaries (§4.9.3) and porting the Rust runtime to D1os (§7.1.1) as part of his Part II project in the Computer Science Tripos [Scu15].

Ionel Gog implemented the flowlessly minimum-cost, maximum-flow solver for Firmament (§5.6.3); additionally Gustaf Helgesson implemented and evaluated Firmament’s Green cost model (§5.5.4) under my supervision during his MPhil in Advanced Computer Science. Finally,
CHAPTER 1. INTRODUCTION

Adam Gleave added support for an incremental solver using the relaxation algorithm (§5.6.3) in his Part II project under my supervision [Gle15].

1.4 Dissertation outline

The following chapters of this dissertation are structured as follows:

Chapter 2 introduces the three key topics of my dissertation: operating systems, “warehouse-scale computers” (WSCs), and cluster scheduling. I first trace the historical evolution of operating systems, explain how distributed OSes were investigated in the 1980s, and discuss why a revisit is timely. I then outline the hardware and software structure of a WSC using real-world examples, and illustrate three challenges faced by WSC operators: hardware heterogeneity, task co-location interference, and energy efficient operation. Finally, I give an overview of the state of the art in cluster scheduling.

Chapter 3 discusses the requirements that a distributed WSC operating system must satisfy, outlines a high-level design for such an OS, and relates it to prior work. I introduce the notion of translucent abstractions, which enable the system to be transparent, yet expose sufficient information to implement application-level policies. I also show how storable and communicable identifier capabilities and contextual handle capabilities are sufficient to enact object-level protection. Finally, I sketch an I/O model for a WSC OS: clearly delineated I/O requests manage concurrent access without global synchronisation, and the OS tracks buffer ownership.

Chapter 4 introduces the DIOS operating system design by considering its abstractions and the interfaces offered to user-level distributed applications. I explain how names and references instantiate identifier and handle capabilities, and how they are tracked in the distributed OS. I outline the DIOS system call API, explain how machines coordinate with each other, and discuss the scalability of the DIOS abstractions. I describe the implementation of a DIOS prototype as a Linux kernel extension, and explain how applications can incrementally migrate to DIOS.

Chapter 5 describes the Firmament cluster scheduler. I show how Firmament generalises the Quincy scheduler [IPC+09], how it collects detailed task-level and machine-level information, and how this information is made available to pluggable cost models that can flexibly express many scheduling policies. I describe three cost models for Firmament, which, respectively, avoid co-location interference; respect a multi-dimensional resource model; and improve energy efficiency in heterogeneous clusters. Finally, I discuss how the underlying minimum-cost, maximum-flow optimisation problem can be solved incrementally in order to scale the flow network optimisation approach to a large WSC.
Chapter 6 evaluates DIOS and Firmament. I use micro-benchmarks of OS-level primitives and an example application to show that the DIOS prototype runs distributed applications at comparable performance to a legacy OS. I also compare and contrast the security properties of the DIOS abstractions with widely-used isolation techniques. My evaluation of Firmament investigates (i) the effectiveness of three cost models that reduce interference between tasks and improve energy efficiency, using real-world test-bed deployments; (ii) Firmament’s ability to flexibly express different scheduling policies; and (iii) its scalability to large clusters, using a simulation of a Google WSC workload.

Chapter 7 points out directions for future work and concludes my dissertation. In particular, I focus on the accessibility of DIOS abstractions for the programmer, changes to OS kernel structure motivated by DIOS, and on techniques to further improve its security. I also discuss how Firmament might be extended to cover more heterogeneous systems.

Relations between topics. As my dissertation explores concepts situated at the intersection of computer architecture, operating systems, and distributed systems, its scope is by necessity quite broad. In Figure 1.1, I relate the topics covered: a directed edge indicates that a topic has influenced another, and the vertical ordering corresponds to a rough historical timeline as well as a causality chain.
Figure 1.1: An outline of topics that have influenced the research described in this dissertation. A line from topic $x$ to topic $y$ indicates that $x$ influenced $y$. 
1.5 Related publications

Although the work described in this dissertation is unpublished, parts of it have previously been presented in peer-reviewed workshop publications:


I have also authored or co-authored the following publications, which have impacted the work presented in this dissertation, but did not directly contribute to its contents:


Chapter 2

Background

“Whereof what’s past is prologue, what to come
In yours and my discharge.”

Modern computers rely on an extensive software stack to execute user programs. Warehouse-scale data centre “machines” are no exception to this, but have an even more extensive software stack that also includes distributed systems software. Roughly, this stack can be divided into three types of software:

**Operating systems** interface between machine hardware and higher-level, hardware-independent software layers. Operating systems enforce isolation, arbitrate machine resources, and perform privileged operations on behalf of the other two categories of software.

**Distributed infrastructure systems** are user-space applications that run on many or on all machines in the WSC. They offer services that are distributed versions of classic OS functionality, such as data storage, scheduling, or coordination of computations.

**User applications** implement the functionality desired by end-users, relying on the services and abstractions offered by the distributed infrastructure systems. They are managed by a dedicated infrastructure system, the “job master” or “cluster manager”.

In this chapter, I survey the core concepts, implementation techniques, and specific realisations of these three types of software. My aim in doing so is two-fold: first, I aim to furnish the reader with an understanding of the historical evolution that led to today’s widely-accepted ways of building distributed systems; second, I offer an insight into their shortcomings in the face of warehouse-scale computing in modern-day data centres.

In Section 2.1, I trace the historical evolution of operating systems, focusing especially on the distributed operating systems of the 1980s. I explain why some of their key ideas are applicable to modern warehouse-scale data centres.
Following, Section 2.2 describes the user-space software in today’s WSCs (their “workload”), and how WSCs are different to prior distributed systems’ environments: they are of unprecedented scale, have significant heterogeneity in hardware and workloads, and run at high utilisation. As a result of the high utilisation, co-location interference between task sharing resources is common, and I show that it significantly degrades performance.

Finally, I focus on one particularly crucial distributed infrastructure system: the WSC’s cluster scheduler. Section 2.3 discusses the design goals for a cluster scheduler and surveys the extensive work of recent years in this area.

### 2.1 Operating systems

Operating systems (OSes) are ubiquitous today. They provide hardware abstraction, multiprogramming, time-sharing of resources, and safe isolation between programs and users. However, this notion of an operating system evolved over the course of six decades, and keeps evolving.

**Before operating systems.** Early mechanical and electric computing machines – with the exception of Babbage’s hypothetical Analytical Engine – could not execute arbitrary computations, and hence had no need for an operating system.

Early stored-program computers typically had some minimal, hard-wired bootstrap code as a primitive “boot loader”: it set up the I/O devices – a reader and puncher for paper tape or Hollerith cards – and initialised the memory. In his 1946 report on the Automatic Compute Engine (ACE), Turing refers to “a certain invariable initial order” that would cause a punched card with “sufficient orders to ‘get us started’” to be loaded [Tur46, p. 37]. Wilkes’s description of the early EDSAC similarly mentions hard-wired “initial orders” [Wil85, pp. 143–4], which Wheeler extended with some “co-ordinating orders” for subroutine relocation, akin to a primitive program loader, as early as August 1949 [Wil85, pp. 144–5, 147].

Such hard-wired bootstrap code, however, is more akin to the Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI) in modern machines than to an OS. Once started up, the early machines were entirely under the control of a single user program – either until it terminated, or until the machine was forcibly reset.

#### 2.1.1 From one program to many

The operational efficiency of expensive early machines was often poor, leading their owners to seek ways to increase utilisation. One source of inefficiency was the hand-over of the computer between different users; another was that I/O operations required manual insertion and removal of cards or tape.
Early supervisor systems. Time-consuming manual I/O setup became an even greater overhead as complex manual assembling, linking and loading of multiple sub-routines became the commonplace [Ros69, p. 39]. In the mid-1950s, monitor (or supervisor) programs were conceived to automate the job set-up process, and to amortise it across multiple user jobs where possible. The monitor started a user job and expected the job to return control to it after completion.¹

The General Motors Input/Output System, developed for the IBM 701 in 1954 [LM88; p. 32, citing Ryc83] and its successor GM-IAA I/O for the IBM 704 are early examples of such systems.² By 1959, the SHARE operating system (SOS) for the IBM 709 included a resident supervisor that could manage four different classes of jobs [BB59].

Even with automated job management, early computers were almost always I/O-bound due to the slow peripherals available. Overlapping I/O and computation using separate processors addressed this inefficiency. The extra “I/O processors” [CV65] were co-processors similar to modern special-purpose processors in hard disk controllers or network cards.³

Multi-programming, which permits other jobs to be executed while I/O is performed, increased efficiency further, but required a notification mechanism that indicates the completion of I/O, interrupts the current computation and causes the processor to jump into supervisor code. Interrupts first appeared in the Atlas computer [KPH61], and permitted more fine-grained overlapping between jobs waiting for I/O. In addition, they also introduced a way for the supervisor to regain control from a user program that did not return to it voluntarily.

Fine-grained interleaving of different jobs improved computer utilisation, but programmers still had to submit their jobs to a queue and wait. The introduction of time-shared operation addressed this problem by allowing interactive and seemingly concurrent execution of programs.

Time-sharing. Interrupts improved machine utilisation by interleaving computation from different jobs. However, the same mechanism also enabled interactive machine use: when multiple programs are loaded at the same time, they can time-share the CPU by handing control over via a context switch. If context switches store and re-load execution state, programs need not be aware that others are running concurrently. Timer interrupts that regularly cause the CPU to enter the OS kernel – which may decide to run another program – help multiplexing the CPU fairly.

Time-sharing was first introduced in the Compatible Time-Sharing System (CTSS) [CMD62]. Early systems like CTSS and the Titan supervisor [HLN68] operated as a combined batch super-

---

¹Anecdotes claim that the idea of writing a “supervisory program” was first suggested at an informal meeting in Herb Grosch’s hotel room at the 1953 Eastern Joint Computer Conference [LM88, pp. 31–2; citing Ste64].

²MIT, however, also lays a claim to having developed the first operating system [Ros86]: the “director tape” for the Whirlwind I computer enabled automatic execution of jobs without operator intervention by mid-1954 [Pro54, p. 7]. Unlike the GM systems, however, the director tape did not have permanently resident supervisory code in the machine’s memory, but instead ran a read procedure off drum memory, processing paper tape in a spare mechanical reader [Hel54].

³In the early days, however, the I/O processors could even be entire “computers” of their own [Rad68, p. 6].
visor and interactive OS. Remote terminals could interactively access the computer, but when it was idle, the processor ran traditional batch jobs (e.g. the “background facility” via a FORTRAN Monitor Systems (FMS) instance in CTSS [Cre81, p. 484]).

Time-sharing gave rise to the operating system’s role in isolating both the kernel from the user processes, and user processes from each other while sharing physical resources.

*Dual-mode operation* is a simple mechanism to separate privileged and unprivileged code. The Titan computer’s OS, for example, had a privileged “extracode control” mode [HLN68], which could only be entered via specific entry points, and a “main control” for unprivileged code.

Other mechanisms for isolating different unprivileged processes from each other include *segments*, *virtual memory*, and *rings of protection*. Multics [CV65] introduced segments: protected blobs of data that were named and addressed, and which had explicit access control limits [DD68; BCD72]. Segments were automatically moved between different storage levels and could be private or shared [DN65, pp. 223–5]. In addition, Multics also introduced multiple, fine-grained “rings” of protection in which code executed [CV65].

Unix, whose derivates are still widely used today, is arguably the most successful early-day time-sharing OS. Unix was created for PDP-series minicomputers and did not initially support multi-programming [Rit79], but later added support for it via virtual memory. The other key isolation mechanism in Unix is the pervasive file abstraction with its owner and group properties, as well as read, write, and execute permissions [RT74], which allows many users to share a single file system namespace.

### 2.1.2 From one machine to many

Early computing was confined to a single general-purpose machine with remote “dumb” terminals. However, even in the mid-1960s, Fuchel and Heller experimented with an extended core store (ECS) shared between two CDC 6600 machines. Their concept of an “ECS based operating system” anticipated process migration between the connected machines via swapping to the ECS, as well as buffered I/O and tolerance of faults in the store [FH67].

However, few institutions ran more than one computer until the late 1970s, when “microcomputers” and personal workstations became available at lower cost and reliable, affordable local-area networking technologies appeared. At this point, it was first feasible to have a substantial number of connected computers. The opportunities of pooling resources led to the development of numerous *distributed operating systems* (see Table 2.1).

---

4 Multics conceptually supported 64 rings, but the implementation only had eight, which due to restrictions in the underlying Honeywell 645 hardware were implemented partially in software; Schroeder and Saltzer demonstrated a full hardware implementation in 1972 [SS72]. However, Wilkes observed in 1993 that rings usually collapse to dual-mode operation [Wil94, p. 12].
Early distributed OSes. HYDRA [WCC+74] for the C.mmp machine [WB72] was one of the earliest distributed operating systems. The C.mmp consisted of sixteen PDP-11 machines with a shared clock and shared memory, connected via a cross-bar. Despite its rather SMP-like structure of the C.mmp, HYDRA pioneered several key distributed principles, such as distributed objects, fault tolerance, and capability-based protection.

The subsequent Medusa OS targeted the C*m* architecture, which connected 50 “compute modules” in clusters of ten. As a result of the high communication cost, locality and restricted sharing were key to Medusa’s performance [OSS80, §1]. Medusa is one of the earliest examples of a distributed OS centred around message-passing. It exposed a three-class object abstraction to the user, consisting of (i) pages, (ii) pipes and semaphores, and (iii) files, “task forces” and descriptor lists. Due to hardware limitations, Medusa only provided modest protection: for access to an object, a descriptor that could reside in a protected descriptor list was required [OSS80].

Message-passing systems. The Aleph OS for the Rochester Intelligent Gateway (RIG), Accent, and Mach were three related distributed OSes developed in the early 1980s [Ras86]. All of them were based on message-passing via ports (process message queues).

Aleph’s design was simple: ports were unprotected, global identifiers were treated as data, and
messaging was not fully transparent. Its use-case was to offer a small network of minicomputers and act as a “gateway” to big, time-shared machines [LGF+82].

Accent [RR81] re-aligned RIG’s principles around virtual memory, full transparency, and an object-style programming model. Accent had an on-disk “page store” for durable storage, with the kernel bringing pages into memory in response to messages. To optimise transmission of large messages, Accent used copy-on-write remapping into the host address space; messages across machine boundaries were supported by lazy retrieval of remote memory. The Accent programmer, however, did not typically send messages explicitly. Instead, a procedure-call interface was exposed via stubs generated via the Matchmaker IDL [JR86]. Similar IDL concepts exist in many later distributed and micro-kernel operating systems.

Mach took Accent’s principles and extended them with multi-processor support and Unix compatibility [Ras86, §5]. It introduced threads, which share a task’s (≡ process’s) ports, allowed user-space shared memory synchronisation, and supported external user-space pagers. Mach was an influential micro-kernel architecture (see §A.1.1), but was ultimately hampered by the high cost of its fully transparent messaging abstractions in an environment where communication was significantly more expensive than computation.

Other distributed OSes offered interfaces closer to traditional system call APIs. For example, both LOCUS [WPE+83] and VAXClusters [KLS86] created distributed systems from then-common VAX machines, extending VAX/VMS and Unix paradigms with support for distribution. LOCUS supported nested transactions on replicated files in a shared store, with the OS at each replica locally enforcing mutual exclusion. LOCUS internally locates files via a distributed naming catalog, and access to them was fully transparent (i.e. their physical location was never exposed). Moreover, LOCUS supported dynamic cluster membership and remained available during network partitions, using a reconciliation protocol to merge divergent state on repair. VAXClusters, by contrast, combined several VAX machines and dedicated storage nodes into a single security domain. Decisions were taken by quorum voting and the system’s minority partition did not remain available when partitioned. Unlike in LOCUS, the manipulation of files in the distributed store relied on a distributed lock manager.

**RPC-based systems.** The late 1980s saw several large distributed OS projects – e.g. V [Che88], Sprite [OCD+88], and Amoeba [MRT+90] – that used a remote procedure call (RPC) model and did not expose message-passing directly. However, their approaches differed [DKO+91]: Sprite targeted a model of per-user workstations, using process migration to harness idle resources (as in V). It was based on a shared file system (like LOCUS), emphasised Unix compatibility (like Mach), and did not expose special features to support distributed applications. Amoeba, by contrast, assumed a centralised pool of processing shared between all users, was based on distributed objects and capabilities (as in Chorus [RAA+91]). It had a micro-kernel architecture with system-wide user-space servers and fully transparent remote operations. Of all classic distributed OSes, Amoeba likely comes closest to matching the requirements of a modern WSC.
Plan 9. In the early 1990s, Plan 9 constituted a final OS with kernel-level support for transparent distributed operation [PPD+95]. Plan 9 targeted networked end-user workstations, and advocates an even more radically file-centred design than Unix, allowing for remote resources to be mounted into a per-process namespace. Consequently, local applications could transparently interact with remote processes, devices, and other resources via namespace mounts [PPT+93].

Plan 9 was never widely adopted, but some of its key concepts have subsequently appeared in operating systems: the directory-structured procfs in Linux is an example of control hooks exposed as files; kernel namespaces as used by containers (§A.1.2) are an example of per-process namespaces; and JSON-based REST APIs are similar to Plan 9’s textual messaging.

Why did distributed OSes fail to be adopted? As distributed operating systems were developed over 30 years ago, and yet few of their features are present in modern OSes, one might wonder why they did not gain adoption. I believe that there are three key reasons why distributed operating systems “failed”:

1. They did not address a pressing workload need. Few workloads in the 1980s actually required the resources of multiple machines, and the complexity of distributed operation rarely made them worthwhile. Classic distributed OSes may have been an instance of a hammer looking for a nail: technology just about made them feasible, but no pressing need motivated them.

2. Single-machine performance gains trumped parallelism. Workloads that could in principle exploit parallelism for performance were often better off running on a large, expensive, time-shared machine; for desktop workloads, the effort of parallelisation was hardly worth the gain, as faster workstations soon became available.

3. The disparity between compute and network speed favoured local computation. Even with improving networking technologies, local compute speed still vastly exceeded cross-machine communication speed. In fact, this gap widened towards the end of the 1980s: clock speeds increased rapidly, but network latency reduced only slowly, making remote messaging increasingly expensive.

However, all of these conditions have materially changed in the context of WSCs:

1. Workloads already require distribution. WSC workloads fundamentally require distribution for scale or fault tolerance (§2.2.1), so distribution over multiple machines is a necessity rather than an option.

2. Single-machine performance no longer improves rapidly. Workloads can no longer rely on machines getting faster. Moreover, request-based and data-parallel workloads require network and storage bandwidth that exceeds a single machine’s resources.
3. **Network performance increases relative to compute speed.** The trend of compute speed outscaling network speeds in the 1980s and 1990s has reversed: network bandwidth still increases and comes close to DRAM bandwidth, and network latencies are falling after constancy [ROS+11]. Single-threaded compute speed, on the other hand, stagnates.

Hence, it seems worthwhile to revisit distributed operating systems in the context of WSCs. Indeed, I draw on many classic distributed OS ideas – e.g. distributed object abstractions, capabilities, transparent operation, and universal I/O interfaces – in my work.

### 2.1.3 From generality to specialisation

After the distributed operating systems of the 1980s, OS research branched out in several directions, many of them specialising OSes to particular use-cases or hardware environments.

I briefly summarise several relevant trends and highlight their impact on my work in the following. Appendix A.1 covers the related work in more detail.

**Micro-kernels (§A.1.1)** are minimal OS kernels with the goal of modularity, allowing most OS services to be implemented as specialised user-space servers. To support these, they support fast, message-based inter-process communication (IPC). The message-passing architecture of many micro-kernels makes them a natural fit for distributed operation across machines, and some (e.g. Mach [ABB+86]) have supported cross-machine operation.

Like micro-kernels, my work aims to expose only the minimal abstractions required for distributed, data-intensive WSC applications. It also leaves it to user-space applications (similar to micro-kernel’s servers) to define their own policies (e.g. the consistency level in a distributed key-value store). However, unlike micro-kernels, I do not restrict the OS interface to simple message-passing.

**Multi-tenant isolation (§A.1.2)** is important to safely share the infrastructure of WSCs. Depending on their mutual trust, programs may either share a kernel, root file system, network and PID namespace, or come with their own.

Many isolation techniques are based on virtualisation: underlying resources are partitioned and exposed as virtual resources towards a higher-level entity in the stack. The virtualised entity might be the whole machine (via a hypervisor such as Xen [BDF+03]), kernel-level OS state (via kernel namespaces, e.g. Linux containers [Men07]), or I/O devices (via hardware-assisted I/O virtualisation, e.g. in Arrakis [PLZ+14]). Such virtualisation specialises the OS to a multi-user environment with particular sharing and trust assumptions; approaches differ in their isolation granularity.

My work draws on the kernel namespace approach to isolate users’ applications, but extends the namespace scope across machines, and supplements them with fine-grained distributed capabilities.
Multi-core OSes (§A.1.3) specialise the OS for scalable operation on many-core machines, or for increasingly common heterogeneous multi-core hardware. The former can be done by adapting the OS to allow only explicit sharing [BCC+08], by applying selective patches to improve lock scalability [BCM+10a], or by changing the OS abstractions to be inherently amenable to scalable implementation [HMC+12; PSZ+12; CKZ+13]; the latter typically relies on message-passing [BBD+09; NHM+09] or process migration [BSA+15].

WSCs are composed of thousands of highly utilised many-core machines, and multi-core scalability is important to their efficient use. My work aims to base its new OS abstractions on inherently scalable principles, and my scheduling work explicitly takes heterogeneity – both across machines and within a single machine – into account.

Specialisation (§A.1.4) entails OS kernel modification for a particular specialised environment, or tailors the whole OS to an application. This can be done by making the OS kernel itself extensible, as in VINO [ESG+94] and SPIN [BSP+95], or by developing a special-purpose OS like Akaros [RKZ+11]. Alternatively, OS functionality can be combined with the application into a library OS. This allows the application (or its libraries) to specialise OS functionality (e.g. thread scheduling, I/O strategies, and storage layout) to specific needs, while the host OS kernel merely serves as a resource-allocating “control plane”. This approach was pioneered in the Exokernel [EKO95] and Nemesis [LMB+96], but more recently revisited by modern library OSes such as Drawbridge [PBH+11], Bascule [BLF+13], IX [BPK+14], and Arrakis [PLZ+14], and, more radically, in unikernels like Mirage [MMR+13] and OSv [KLC+14].

My work is situated somewhere between a specialised OS and a library OS approach: the OS kernel offers explicit support for distributed operation, but the distributed systems’ policies are implemented by user-space application logic using the OS primitives.

In the remainder of this dissertation, I occasionally refer to specific research efforts that happened as part of these wider trends.

2.1.4 Summary

Operating systems evolved in concert with hardware and workloads (§2.1.1), and continue to do so (§2.1.3). Machine utilisation, and its associated efficiency gains, have always been drivers of innovation, from time-sharing to virtualisation and multi-core scalability. Modern WSCs likewise require high utilisation for efficiency.

Distributed operating systems, which were en vogue in the 1980s, did not offer sufficient utility at the time to achieve wide deployment (§2.1.2). However, their ideas are ripe for a revisit now: workloads require distribution; transparency is in demand; and a global view allows the OS to make better decisions.

Based on these insights, I state a set of goals for a distributed WSC OS.
A distributed operating system for a warehouse-scale data centre environment must:

1. expose the minimal OS abstractions required to support distributed applications at competitive performance, drawing on classic distributed operating systems (§2.1.2) and micro-kernels (§A.1.1);

2. offer isolation between users and applications at least as strong as that of widely-used virtualisation and kernel namespace (container) approaches (§A.1.2); and

3. ensure scalability to many-core machines and large clusters by basing the OS abstractions on scalable design principles (§A.1.3).

My work addresses these challenges in the design and implementation of D\textsc{ios}, a specialised “warehouse-scale” operating system, which I describe in Chapters 3 and 4. I return to the challenges and review my solutions to them in the evaluation chapter (§6.2.4).

In the next section, I describe the hardware and workloads that a WSC OS must deal with.

## 2.2 Warehouse-scale computers

Many modern applications are dependent on distributed systems running in “warehouse-scale computers”, i.e. large clusters in data centres [BCH13].

Such clusters are shared by multiple applications and users to increase utilisation [HKZ+11; VPK+15]: they run many independent tasks – applications instantiated as processes, containers, or virtual machines – that belong to different applications. Catering well to the resulting “workload mix” is key to the efficient operation of a WSC.

In this section, I describe the typical workload in a WSC (§2.2.1), outline how heterogeneity in its constituent hardware matters (§2.2.2), and how high utilisation comes at the cost of interference that degrades workload performance (§2.2.3). Finally, I discuss why cost-effective WSC operation requires energy-efficient clusters (§2.2.4).

### 2.2.1 Diverse workloads

WSCs exist to support novel workloads that require very large amounts of compute and storage resources. Distribution over many machines is required either to keep up with a large number of user requests, to perform parallel processing in a timely manner, or to be able to tolerate faults without disruption to service.

In general, there are two workload categories: infrastructure systems (§2.2.1.1) and user applications that process data or expose them to remote end-users (§2.2.1.2).
In addition, WSC workloads can also often be divided into batch and service work (as e.g. at Google [SKA+13]). This division is orthogonal to the dichotomy of infrastructure systems and applications described above, although most infrastructure systems are service workloads.

**Service workloads** run continuously and offer functionality either directly to end-users or to applications that they interact with. They only terminate due to failure, human or cluster scheduler intervention. A distributed key-value store is an example of a service workload.

**Batch workloads** are finite data processing jobs that start, perform some work, and terminate when completed. An example of a batch workload is a regularly executed log crawling and transformation pipeline.

Empirically, the majority of jobs and tasks are typically in batch jobs, but the majority of cluster resources over time are devoted to service jobs [SKA+13, §2].

### 2.2.1.1 Distributed infrastructure systems

Infrastructure systems are key to the operation of WSC applications. They often serve the same purpose as OS services traditionally implemented in the kernel. For example, they offer coordination, storage and file systems, and process-like abstractions, that higher-level applications build upon. As a result, new “stacks” of mutually dependent infrastructure systems have been
created. Figures 2.1 and 2.2 illustrate this using the known components of the WSC infrastructure software stacks at Google and Facebook.

Broadly speaking, the infrastructure stacks typically consist of coordination and cluster management services, storage services, and parallel data processing frameworks.

**Coordination and cluster management.** Since many infrastructure services in a WSC are co-dependent, they require an authoritative source of liveness and location information in order to coordinate.

This coordination authority is usually implemented as a reliable, consistent distributed key-value store. This store records the locations of master processes (leaders), offers distributed locking, and enables service discovery by tracking the location of service tasks. Google’s Chubby service [Bur06] is based on Paxos [CGR07]; Yahoo!’s Zookeeper [HKJ+10], which is based on Zab [RJ08], and etcd, which is based on Raft [OO14], are popular open-source coordination services. They all use distributed consensus algorithms that trade raw performance for reliability in the face of failures.

A cluster manager, by contrast, manages services’ and applications’ tasks and arbitrates resources between them. This entails tracking machine liveness, starting, monitoring, and terminating tasks, and using a cluster scheduler to decide on task placements. Mesos [HKZ+11] and Google’s Borg [VPK+15] and Omega [SKA+13] are such cluster managers.

**Data storage.** WSCs store huge amounts of data, but depending on access frequency and structure, different infrastructure systems are used for this purpose.
Block storage either comes in the form of unstructured stores, or as hierarchical file systems akin to NFS or local file systems. Facebook’s Haystack [BKL*10] and f4 [MLR*14] are unstructured stores which store and replicate binary large objects (blobs) of different popularity. By contrast, GFS [GGL03] and its successor, Colossus, at Google, the Hadoop Distributed File System (HDFS) at Facebook and Yahoo!, and TidyFS [FHI*11] and Flat Datacenter Storage (FDS) [NEF*12] at Microsoft are hierarchical distributed file systems.

For more structured data, WSCs run sharded, replicated key-value stores with varying trade-offs between consistency and performance. BigTable [CDG*06] implements a three-dimensional map indexed by a row, column, and timestamp on top of GFS and offers per-row consistency; Facebook uses HBase over HDFS [HBD*14] in a similar way to store users’ messages [BGS*11]. Other data stores are closer to traditional databases and offer transactions with ACID guarantees: examples are Google’s Megastore [BBC*11] over BigTable, and Spanner [CDE*13]. In some cases, classic sharded and replicated databases are used, too: Facebook, for example, uses MySQL for structured long-term storage.

For expedited access by request-serving applications, data are often cached in ephemeral stores. These stores can be generic key-value stores – like memcached, which is used as the in-memory serving tier at Facebook [NFG*13] – or specifically designed for particular use-cases. Google’s Dremel [MGL*10] and PowerDrill [HBB*12], for example, store data in columnar form to enable fast aggregation queries, while Facebook’s Tao [BAC*13] is a cache for graph-structured data with locality.

**Parallel data processing.** WSC applications often need to process very large data sets in a timely manner. In order to expose an accessible programming interface to non-expert application programmers, parallel data processing frameworks hide challenging aspects of distributed programming. Examples of the details abstracted include fault tolerance, scheduling, and message-based communication.

MapReduce [DG08] is a widely-used abstraction for such transparent distributed parallelism. Its relative simplicity – the user only has to implement a map() and a reduce() function – makes it an appealing abstraction. Other frameworks are more expressive: for example, Dryad [IBY*07] at Microsoft models the computation as a data-flow DAG.

Even higher-level abstractions are deployed on top of the data processing frameworks in order to make them accessible to lay users: common examples are domain-specific languages, such as the SQL-like Tenzing [CLL*11] at Google, and Hive [TSJ*09] at Facebook, or language integration (e.g. FlumeJava at Google [CRP*10] and DryadLINQ at Microsoft [YIF*08]), or interactive UIs like Facebook’s Scuba [AAB*13]. For some applications, purpose-built systems perform specialised processing: for example, Percolator at Google was built specifically for fast incremental updates to the web search index in BigTable [PD10]. Likewise, streaming data is processed with special stream processing frameworks such as MillWheel [ABB*13] at Google and S4 at Yahoo! [NRN*10]. Graph structured data is processed using systems which
let users express computations in a “vertex-centric” way, with Unicorn [CBB+13] at Facebook and Pregel [MAB+10] at Google being well-known examples.

**Monitoring and tracing.** The complex interactions between the aforementioned infrastructure systems require bespoke performance tracing and debugging tools, since events from many different machines and contexts must be correlated.

Such tools either hook into common libraries widely used in the WSC, as in Google’s Dapper [SBB+10] or Twitter’s Finagle [Eri13], or leverage common identifiers to construct a cross-system request trace, as in Facebook’s ÜberTrace [CMF+14, §3]. The large corpus of tracing data available enables statistical analysis to derive causal relationships (e.g. in Mystery Machine [CMF+14]), or to detect performance anomalies such as negative interference between co-located tasks (e.g. in CPI2 [ZTH+13]).

**Observations.** WSC infrastructure systems often implement the distributed equivalents of traditional OS kernel functionality. For example, coordination systems and cluster managers implement synchronisation and task scheduling, distributed storage systems implement file systems, and parallel data processing systems implement the distributed equivalent of threading.

There are two reasons why this functionality was implemented in user-space systems:

1. No modern OS used in WSCs has native support for distributed operation. While historic distributed OSes had such features (§2.1.2), they are no longer present in current OSes.

2. WSC stacks often comprise of different systems for similar purposes: consider, e.g. the different caching systems in the Facebook stack. These systems evolved in response to acute business needs, rather than well-known use cases for which optimised OS primitives have been devised over time. As a result, independent, rapid evolution was a priority.

However, there are merits to offering OS-level abstractions to build distributed infrastructure systems upon: as I highlighted in the introduction (§1.2), the privileged nature of the OS grants it broad, global knowledge and the ability to impose abstractions with strong isolation properties, as well as the potential to improve efficiency and simplify systems programming.

**2.2.1.2 Applications and user jobs**

Applications form the “business logic” of the WSC: they serve user requests, analyse data to derive insights, or support other productivity tasks.

For example, Facebook’s web server instances respond to user requests by aggregating elements from the TAO, memcached, Haystack and f4 storage systems into a response. At the same time, Hive queries run MapReduce jobs that analyse the same data to collect information on user
behaviour, and long-running MapReduce jobs move data between storage systems. Similar setups exist in other companies.

Such applications and user jobs differ from infrastructure services in three ways:

1. Applications generally rely on libraries that interact with the infrastructure systems, rather than interfacing directly with the OS, as most infrastructure systems do.

2. High performance and low latency are important to some applications (e.g. serving front-ends), but not to others (e.g. batch jobs), while almost all infrastructure services are subject to strict service level objectives (SLOs).

3. Application developers demand higher-level interfaces than the expert engineers who build infrastructure systems, and application code is usually ignorant of the details of machines, coordination, and parallelisation. In addition, applications are often implemented in high-level languages [MAH+14].

Crucially, the applications themselves are oblivious as to how the primitives they rely on are supplied: it does not matter whether they come from “thick” infrastructure services on commodity OSes, or “thin” infrastructure services on a WSC OS, or directly from the WSC OS. This grants the opportunity to evolve the infrastructure stack without necessarily requiring application programmers to change their habits.

2.2.2 Hardware heterogeneity

While WSCs are constructed from commodity machines purchased in bulk, their constituent hardware is more heterogeneous than one might expect, often due to rolling hardware upgrades and deliberate diversification.

For example, the WSC in the public cluster trace released by Google in 2011 [RWH11; Wil11], consists of around 12,550 machines, which cover three different machine platforms and ten different machine specifications.5 Once other distinguishing attributes of a machine – such as “kernel version, clock speed, presence of an external IP address”, or “whether [the] machine runs a GFS chunkserver” [RWH11, p. 5] – are considered, the number of unique combinations grows to 34. Of these combinations, eight apply to more than 100 machines, and thirteen apply to ten or more machines (Figure 2.3). Anecdotal evidence from other WSCs, such as Amazon’s EC2 infrastructure, confirms that this heterogeneity generalises [OZN+12]. Moreover, it matters for performance: Google have observed varying performance characteristics of identical workloads on different machine platforms [TMV+11].

5The trace documentation defines machine platform as the combination of “microarchitecture and chipset version” [RWH11, p. 5]; in addition, I regard the specification of a machine to refer to its platform and its capacity,
Figure 2.3: Sankey diagram of Machine types and configurations in the public mid-2011 Google cluster trace [RWH11]. The WSC is highly heterogeneous.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>hmmr</td>
<td>Hidden Markov model gene database search (integer)</td>
<td>[Hen06, p. 5]</td>
</tr>
<tr>
<td>gromacs</td>
<td>Molecular dynamics simulation (floating point)</td>
<td>[Hen06, p. 11]</td>
</tr>
<tr>
<td>STREAM</td>
<td>DRAM memory throughput benchmark.</td>
<td>[McC95]</td>
</tr>
<tr>
<td>NUMA-STREAM</td>
<td>Multi-threaded version of STREAM for machines with multiple memory controllers.</td>
<td>[Ber11]</td>
</tr>
<tr>
<td>bonnie-rd</td>
<td>Disk read throughput measurement from the bonnie++ benchmarking tool.</td>
<td>See footn.6</td>
</tr>
<tr>
<td>bonnie-wr</td>
<td>Disk write throughput measurement from the bonnie++ benchmarking tool.</td>
<td>(see above)</td>
</tr>
<tr>
<td>iperf-cpu</td>
<td>CPU load while running iperf in TCP mode, saturating a 1 GBit/s link.</td>
<td>See footn.7</td>
</tr>
<tr>
<td>BogoMips</td>
<td>BogoMips number reported by Linux kernel.</td>
<td>See footn.8</td>
</tr>
</tbody>
</table>

Table 2.2: Machine heterogeneity micro-benchmark workloads.

**Impact.** To illustrate the impact of heterogeneity, I ran a set of simple micro-benchmarks measuring integer and floating point operation throughput, memory access bandwidth, disk I/O bandwidth, and network I/O cost (Table 2.2) on a set of otherwise idle machines (Table 2.3). All machines are post-2009 designs representative of machines found in today’s WSCs assuming a five-year depreciation cycle.

Figure 2.4 shows the results, normalised to the oldest machine type (A). For the single-threaded,
CHAPTER 2. BACKGROUND

<table>
<thead>
<tr>
<th>Type</th>
<th>CPUs</th>
<th>Microarchitecture</th>
<th>GHz</th>
<th>Cores</th>
<th>Threads</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Intel Xeon E5520</td>
<td>Gainestown (2009)</td>
<td>2.26</td>
<td>4</td>
<td>8</td>
<td>12 GB DDR3-1066</td>
</tr>
<tr>
<td>B</td>
<td>Intel Xeon E5-2420</td>
<td>Sandy Bridge (2012)</td>
<td>1.90</td>
<td>12</td>
<td>24</td>
<td>64 GB DDR3-1333</td>
</tr>
<tr>
<td>C</td>
<td>AMD Opteron 4234</td>
<td>Valencia (2011)</td>
<td>3.10</td>
<td>12</td>
<td>12</td>
<td>64 GB DDR3-1600</td>
</tr>
<tr>
<td>D</td>
<td>AMD Opteron 6168</td>
<td>Magny Cours (2010)</td>
<td>1.90</td>
<td>48</td>
<td>48</td>
<td>64 GB DDR3-1333</td>
</tr>
</tbody>
</table>

Table 2.3: Machine configurations used in the experiments in Figure 2.4. All machines run Ubuntu 14.04 with Linux 3.13.

Figure 2.4: Normalised performance of the micro-benchmarks on the heterogeneous machine types listed in Table 2.3; higher is better.

compute-bound SPEC CPU2006 benchmarks hmmer and gromacs, machines with a faster CPU clock speed (types A and C) exceed the performance of the lower-clocked ones (types B and D). As one might expect, CPU performance is roughly correlated with the BogoMips measure reported by the Linux kernel.

The default, single-threaded STREAM memory-access benchmark [McC95], however, is limited to the bandwidth of a single memory controller. Machine type A (the only single-socket system tested) outperforms all more recent machine types. This could be due to the overhead of cache coherence protocols on NUMA machines.

In the multithreaded STREAM-NUMA, multiple memory controllers easily outperform type A machines by up to 2×. Type D outperforms the newer Valencia-based type C since type D machines have four instead of two memory controllers. Yet, the highest overall throughput is attained by the dual-controller QPI-based Sandy Bridge Xeon machines (type B).

Storage and networking benchmarks are more dependent on the peripherals than on the CPU, although architecture and clock speed also have an impact. When reading from disk (bonnie-rd) and writing to it (bonnie-wr), newer machines match type A, or outperform it by 20–40%, even though type A machines have high-throughput Serial-Attached-SCSI (SAS) harddrives. For network I/O with iperf, however, type A machines see the lowest CPU load. This may be a consequence of the offloading features present in the Intel NICs of the type A machines, which are not available on other machines’ NICs.
CHAPTER 2. BACKGROUND

2.2.3 Co-location interference

Even on homogeneous machines, workload performance can vary significantly when multiple workloads are co-located. Specifically, workloads often contend for shared hardware or software resources. Contention may be direct, e.g. for access to a hard disk, a network interface, or a lock; or it may be indirect, for example via cache evictions.

Some hardware resources in commodity servers are provisioned for peak load (e.g. CPUs), while others are systematically over-subscribed by the machine architecture (e.g. network I/O bandwidth). Oversubscription is a result of physical constraints, hardware cost, and typical server workloads. In a WSC environment, however, machines experience continuous high load and resources are routinely highly contended. In the following, I illustrate the effects of such contention using both highly cache-sensitive micro-benchmarks and parallel data processing workloads.

**Pairwise interference.** Co-location interference can easily be measured using common benchmarks such as SPEC CPU2006. In Appendix A.2.1, I show that on both type B and type C machines, the runtime of SPEC CPU2006 benchmarks suffers degradations of up to $2.3 \times$ even when only two tasks are co-located on a machine.

Benchmarks like SPEC CPU2006, however, use highly-optimised compute kernels that typically have good cache affinity, and sharing consequently has an especially severe effect. As Ferdman et al. found, most WSC applications are not tuned for cache affinity [FAK+12].

I thus repeat the same co-location experiment with a set of WSC applications. I run the applications shown in Table 2.4 in different combinations and pinned to different cores on a 12-core Opteron 4234 (“Valencia”, Figure 2.5a), and also on a 12-core Intel Xeon E5-2420 (“Sandy Bridge”, Figure 2.5b). The applications are executed single-threadedly in order to make the

<table>
<thead>
<tr>
<th>L1$</th>
<th>L1$</th>
<th>L1$</th>
<th>L1$</th>
<th>L1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>C6</td>
<td>C10</td>
<td>C7</td>
<td>C9</td>
</tr>
<tr>
<td>L1S</td>
<td>L1S</td>
<td>L1S</td>
<td>L1S</td>
<td>L1S</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>L2$</th>
<th>L2$</th>
<th>L2$</th>
<th>L2$</th>
<th>L2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(2 MB)</td>
<td>(2 MB)</td>
<td>(2 MB)</td>
<td>(2 MB)</td>
<td>(2 MB)</td>
</tr>
</tbody>
</table>

| L3$ (6 MB) | L3$ (6 MB) | L3$ (15 MB) | L3$ (15 MB) |

(a) AMD Opteron 4234 (“Valencia”). (b) Intel Xeon E5-2420 (“Sandy Bridge”).

Figure 2.5: Micro-architectural topologies of the systems used in the co-location experiments. \( C_i \) are physical CPU cores, while \( T_i \) denote hyper-threads; first-level caches (L1$) are shown in red, second-level caches (L2$) in green, and third-level caches (L3$) in blue.
CHAPTER 2. BACKGROUND

<table>
<thead>
<tr>
<th>Key</th>
<th>Application</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>HTTP</td>
<td>HTTP serving</td>
<td>nginx serving a static page.</td>
<td>Network-bound</td>
</tr>
<tr>
<td>QS</td>
<td>QuickSort</td>
<td>Sort a large set of integers using qsort.</td>
<td>I/O-bound</td>
</tr>
<tr>
<td>PR</td>
<td>PageRank</td>
<td>GraphChi PageRank on LiveJournal dataset.</td>
<td>Memory-bound</td>
</tr>
<tr>
<td>BOPM</td>
<td>BOPM</td>
<td>Binomial options pricing model.</td>
<td>CPU-bound</td>
</tr>
<tr>
<td>SQ</td>
<td>Spark queries</td>
<td>JOIN and SELECT queries on web log.</td>
<td>Memory-bound</td>
</tr>
</tbody>
</table>

Table 2.4: WSC applications used in pairwise interference experiments (Figure 2.6).

setup comparable to the SPEC CPU2006 benchmarks.\(^9\)

Figure 2.6 shows the normalised runtime for different co-locations. It is evident that interference occurs, as workload runtime degrades by up to $2.13 \times$ compared to running alone. As with SPEC CPU2006, the frequency and magnitude of interference increases as additional levels of the memory hierarchy are shared – consider, for example, the difference between the performance of PageRank and QuickSort on the Opteron in Figure 2.6a (no caches shared) and Figure 2.6c (shared L3 cache).

Additional figures in Appendix A.2.2 illustrate that many of the degradations are co-incident with increased cache miss counts. In some cases, however, interference is present even when there is no correlation with cache misses, or when the applications run on separate sockets. Such interference may occur for several reasons:

1. applications may contend for other shared machine resources, such as the disk or the network interface; or

2. operating system abstractions (e.g. locks or kernel data structures) are contended, or do not scale to concurrent use (cf. §A.1.3).

The experiments also show that the two machines behave differently under contention: on the Xeon, the Spark query application interferes much more severely with QuickSort and PageRank (over $2 \times$ degradation, compared to $1.4 \times$ on the Opteron). The Xeon, however, is less sensitive towards contention for the shared L3 cache as a consequence of its larger size (15 MB compared to 6 MB on the Opteron). Likewise, applications using adjacent hyperthreads on the Xeon (sharing a small 256 KB L2 cache) experience strong interference, but suffer less when sharing an L2 cache (2 MB) on the Opteron.

\textit{n-way interference.} I have so far considered pairs of workloads on otherwise idle machines. In practice, many-core machines in a WSC run more than two tasks at a time: production clusters at Google run around eight tasks per machine in the median, and around 25 tasks in the 90\textsuperscript{th} percentile [ZTH\(^{+}13\), Fig. 1(a)].

\(^9\)The Spark query application, however, comes with extra runtime threads (e.g. the JVM’s garbage collection threads) and I allow it to use multiple cores.
Figure 2.6: Co-location interference between different WSC applications on an AMD Opteron 4234 (left column) and Intel Xeon E5-2420 (right column). All runtimes are for the x-axis benchmark in the presence of the y-axis benchmark, normalised to the former’s isolated runtime on an otherwise idle machine. Black squares indicate results exceeding the scale; gray ones indicate that the benchmark failed. See Appendix A.2.2 for corresponding heatmaps of cycle and cache miss counts.
I hence investigated how \( n \)-way co-location (for \( n \) CPU cores) affects WSC application workloads. Figure 2.7 shows the normalised runtime of seven different batch processing workloads on a 28-machine cluster.\(^{10}\) Most of the workloads are implemented using Naiad [MMI+13] (see Table 2.5), and the cluster runs at 80–90\% task slot utilisation. As in many cluster schedulers, work is assigned by a simple random first fit algorithm. As the scheduler occasionally makes poor decisions, workloads end up interfering. However, some suffer worse than others: the highly compute-bound image classification task only degrades by about 20\% on average, while I/O-bound NetFlix degrades by 2.1\times, and the highly iterative and synchronisation-bound strongly connected components (SCC) and PageRank workloads degrade by up to 3\times.

This experiment used realistic WSC workloads in a real cluster environment and shows that interference on machine (and cluster) resources can have a major effect on the end-to-end job runtime of batch jobs.

**Related studies.** My observations corroborate the findings reported in related work. For example, Tang *et al.* demonstrated performance variability of 20\% due to thread placement in multi-threaded workloads [TMV+11]. Harris *et al.* found that such workloads even degrade by up to 3.5\times when their user-space runtimes make poor placement decisions on busy ma-
Workload | Application | Share of cluster
--- | --- | ---
Image analysis (cat detection) | kittydar\textsuperscript{11} | 9% (30 tasks)
PageRank on LiveJournal graph | Naiad | 6% (20 tasks)
Strongly connected components | Naiad | 3% (10 tasks)
TPC-H query 17 | Naiad | 8% (28 tasks)
Single-source shortest path | Naiad | 9% (30 tasks)
Netflix movie recommendation | Naiad | 8% (28 tasks)
Symmetric join | Naiad | 2% (7 tasks)
HTTP server | nginx | 13% (45 tasks)
HTTP clients | ab | 13% (45 tasks)
In-memory key-value store | memcached | 13% (45 tasks)
Key-value store clients | memaslap | 13% (45 tasks)

Table 2.5: Batch (top) and Service (bottom) workloads used in the scale-up interference experiment shown in Figure 2.7.

My experiments and the abundant related work show that interference is a key problem for WSCs, and suggests that better scheduling at multiple levels (within a machine and across machines) is needed to address it.

2.2.4 Energy efficiency

Power contributes significantly to the cost of running data centres [BCH13, pp. 12, 87–89], and data centres are responsible for a sizable fraction of global CO\textsubscript{2} emissions [RRT\textsuperscript{+}08]. Hence, the energy footprint of WSCs in data centres matters for economic and ecological reasons.

While the machine types in a WSC may differ (see §2.2.2), all are typically x86 server machines. As some WSC workloads do not actually require such powerful machines [AFK\textsuperscript{+}09; GLL14], using other architectures may yield economies in energy consumption and cost. The increasing need to consider such trade-offs is illustrated by Urs Hölzle’s response to the question whether Google “would switch to [the] Power [architecture]”: “the answer is absolutely”, he said, and “even for a single machine generation” [Pri15].\textsuperscript{12}

\textsuperscript{12}This is notable, since scale-out parallelism at the cost of single-threaded performance was of no interest to Google in 2008 [Höl10]. While Power8 and modern ARM server processors are considerably more powerful than the “wimpy” cores of 2008, they are still unlikely to match x86 single-threaded performance.
Upcoming server-class machines based on low-power designs may therefore further increase hardware heterogeneity in WSCs. Indeed, mixing different machine types can deliver “the best of both worlds”: power-efficient architectures are sufficient for I/O-intensive workloads and those which are not time-critical, while high performance hardware supports workloads that require it. Consequently, several researchers use heterogeneous hardware, smart scheduling, or dynamic power control to reduce WSC energy footprints.

Guevara et al. simulate the effect of mixing different processors in a WSC using a SPEC CPU2006 workload. Their market-based scheduler requires users to specify their jobs’ resource requirements and reschedules tasks every ten minutes [GLL14].

Other efforts focus on scheduling workloads according to the availability of green energy: Aksanli et al. perform admission control on a batch and service job mix according to predicted availability of green energy [AVZ+11]. GreenHadoop takes a similar approach based on the current electricity supply, prices, and weather conditions, and makes use of flexible deadlines in batch jobs to maximise the use of green energy [GLN+12]. BEEMR partitions a MapReduce cluster into an “always-on” partition for interactive analytics and a partition under dynamic power-management for batch jobs that only powers machines up on demand [CAB+12].

In my work, I give evidence that a heterogeneous datacenter can work well, provided that workloads are scheduled correctly.

### 2.2.5 Summary

WSCs are a new environment somewhere between a multi-tenant server co-location facility and a single-authority super-computer. Their workloads are heterogeneous: they vary in nature, resource requirements, and in their impact on machine and cluster resources.

Moreover, WSCs’ hardware is far less homogeneous than one might expect, and the interactions between workloads sharing the infrastructure can have significant impact on performance.

Again, this gives rise to a set of goals.

To adequately support common workloads in a WSC environment, we must:

1. develop OS abstractions suited to WSC workloads, and especially infrastructure systems, matching functionality in the OS kernel to distributed systems software (§2.2.1);
2. develop mechanisms to deal with hardware heterogeneity in the cluster and use it to match workloads to machines well-suited to executing them (§2.2.2);
3. avoid co-location interference between WSC workloads, and offer predictable performance by separating those workloads that interfere (§2.2.3); and
4. improve energy efficiency by smartly assigning work on upcoming mixed-architecture clusters (§2.2.4).

My work consequently focuses on finding appropriate OS abstractions to support WSC workloads in D10s (Chapters 3 and 4), and on maximising the efficiency of the shared WSC by scheduling work appropriately using the Firmament scheduler (Chapter 5).

2.3 Cluster scheduling

Many of the challenges outlined in the previous section can be addressed by appropriately scheduling tasks in the WSC cluster. These decisions are the responsibility of the cluster scheduler.

Scheduling work to compute resources is, of course, not a new problem. Extensive prior work on CPU scheduling exists, but OS CPU schedulers are different from cluster schedulers: they are invoked for brief periods of time during context switches, and block a user-space process while making their decision. A cluster scheduler, by contrast, runs continuously alongside the cluster workload; its scheduling decisions last for a longer time; and it has more diverse and complex design goals than a single-machine CPU scheduler.

In this section, I outline the design goals of existing cluster schedulers and how they meet them. Table 2.6 summarises the core design goals of each system.

2.3.1 Scheduler architecture

Some existing schedulers fundamentally differ in their architecture: the degree to which decisions are made in a centralised or distributed fashion. Figure 2.8 illustrates the approaches that I discuss.

Most early WSC schedulers are monolithic: they have a simple, centralised architecture and process all decisions via the same logic. Typically, a monolithic scheduler runs on a dedicated machine or as part of a cluster manager. The advantage of this approach is its relative simplicity: all state is held in one place, and there is only a single decision-making entity (Figure 2.8a). Scheduler fault-tolerance can be implemented via primary/backup fail-over, or by restarting the scheduler from a previously saved checkpoint.

Recent work, however, has introduced distributed cluster scheduler architectures, albeit with varying motivations:

**Resource sharing between specialised frameworks.** WSCs run many infrastructure systems and applications concurrently (§2.2.1), which requires cluster resources to be partitioned
### Table 2.6: WSC cluster schedulers and their design goals. A ✓ indicates that the property is a design goal, a ✗ indicates that it is not a goal and unsupported. Ticks in parentheses indicate that the system can support the goal via its APIs, but does have built-in support.

<table>
<thead>
<tr>
<th>System [Reference]</th>
<th>Target workload</th>
<th>Distributed</th>
<th>Data locality</th>
<th>Fairness</th>
<th>Soft constraints</th>
<th>Hard constraints</th>
<th>Avoid interference</th>
<th>Auto-scaling</th>
<th>Heterog. machines</th>
</tr>
</thead>
<tbody>
<tr>
<td>HFS [HFS]</td>
<td>MapReduce tasks</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>√</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>LATE [ZKJ+08]</td>
<td>MapReduce tasks</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Quincy [IPC+09]</td>
<td>Dryad tasks</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Delay Sched. [ZBS+10]</td>
<td>Hadoop tasks</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Mesos [HKZ+11]</td>
<td>Framework tasks</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>CIEL [Mur11, §4.3]</td>
<td>CIEL tasks</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Jockey [FBK+12]</td>
<td>SCOPE tasks</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>alsched [TCG+12]</td>
<td>Binaries (sim.)</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>tetrisched [TZK+13]</td>
<td>Binaries</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Where-Map [MT13]</td>
<td>Binaries</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>YARN [VMD+13]</td>
<td>YARN tasks</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Omega [SKA+13]</td>
<td>Google tasks</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Sparrow [OWZ+13]</td>
<td>Shark queries</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>H-DRF [BCF+13]</td>
<td>Hadoop v2 tasks</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Choosy [GZS+13]</td>
<td>Framework tasks</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Quasar [DK14]</td>
<td>Mixed binaries</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Apollo [BEL+14]</td>
<td>SCOPE tasks</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>KMN [VPA+14]</td>
<td>Spark tasks</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Tarcil [DSK15]</td>
<td>Mixed binaries</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Hawk [DDK+15]</td>
<td>Binaries (sim.)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Mercury [KRC+15]</td>
<td>YARN containers</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Bistro [GSW15]</td>
<td>Facebook tasks</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Firmament (Ch. 5)</td>
<td>D1OS tasks, binaries</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Two-level schedulers therefore have a resource manager to allocate resources and application-level schedulers to assign application-level tasks within these allocations. The resource manager is simpler than a monolithic scheduler as it is oblivious to application semantics and scheduling policies. The application schedulers, by contrast, apply application-specific scheduling logic to place tasks, but only see their allocated resources (Figure 2.8b).

Yahoo!’s Hadoop-on-Demand (HoD) was an early two-level scheduler. It combined the TORQUE resource manager [CDG+05] and the Maui HPC scheduler [JSC01], to allocate users’ clusters from a shared pool [VMD+13, §2.1]. Subsequently, Mesos [HKZ+11] and...
YARN [VMD+13, §2.2ff.] devised principled two-level architectures, using offer-driven (Mesos) or request-driven (YARN) resource multiplexing.

**Engineering complexity.** The diverse needs of cluster workloads make it challenging for large organisations to manage and evolve a single scheduler code base [SKA+13, §1].

Google’s Omega cluster manager thus introduced a partially distributed, shared state, distributed logic scheduler architecture. Omega supports multiple co-existent schedulers within the same WSC. The schedulers may be based on different implementations and run distributedly [SKA+13, §3.4], each dealing with a fraction of the workload. However, unlike two-level schedulers, all schedulers contain weakly consistent replicas of the full shared cluster state. They mutate the cluster state by issuing optimistically-concurrent transactions against it (Figure 2.8c). Transactions may yield a successful task placement, or fail, necessitating a re-try. The Apollo scheduler at Microsoft adopts a similar approach [BEL+14].

**Scalability to very short tasks.** Some analytics workloads process interactive queries that must return information in a matter of seconds. To facilitate this, work is broken down into a large number of very short-lived tasks. Such small task granularities increase utilisation and combat straggler tasks’ effects on job completion time [OPR+13].

The short tasks only exist for sub-second durations, and thus the scheduling overhead – either from a centralised scheduler, or from transactions against shared state – can be significant. The Sparrow fully distributed scheduler addresses this problem by entirely eschewing shared state and coordination [OWZ+13]. Instead, cluster machines pull tasks directly from schedulers in response to probes (Figure 2.8d).

Different solutions are appropriate for different environments, and none of these architectures is necessarily “better” than the others. The Firmament scheduler, which I introduce in Chapter 5, therefore supports both centralised and distributed modes of operation.
2.3.2 Data locality

Locality of reference is key to many engineering optimisations in computer systems, notably the efficacy of caching mechanisms in microprocessors. Distributed systems also have notions of “locality”: if an input to a task is not present on a machine, it must be fetched via the network, which necessarily incurs latency and increases network utilisation. To avoid this cost, cluster schedulers aim to increase the number of tasks that operate on local data.

Google’s MapReduce preferentially schedules map tasks on machines that have relevant GFS input chunks available locally or across the same leaf switch [DG08, §3.4], and other systems adopted similar optimisations [IBY+07; HKZ+11; MSS+11]. Research efforts further refined the concept: delay scheduling holds off starting tasks in the hope of task churn leading to better locations becoming available; Scarlett increases replication of popular input data to increase disk locality [AAK+11]; and Quincy weighs the locality benefit of moving already-running tasks against the cost of restarting them [IPC+09].

While early work focused on disk locality, the performance gap between local and remote disk accesses is increasingly marginal [AGS+11]. Other locality notions have emerged, however: many data processing frameworks [ZCD+12] and storage systems [ORS+11; LGZ+14] cache data in RAM. As remote DRAM access over the network is slower than local access, the locality of in-memory objects is important.

Additionally, notions of locality exist within machines: remote memory access in NUMA systems is costly, and locality of PCIe devices matters for high-performance network access. Information about such fine-grained locality inside a machine is normally only available to the OS kernel’s scheduler, not to the cluster scheduler – a limitation that my work addresses.

2.3.3 Constraints

Not all resources are necessarily equally suitable for a particular WSC application: availability of special hardware features (such as flash-based storage, or a general-purpose GPU accelerator), software compatibility constraints (e.g. a specific kernel version), and co-location preferences (proximity to a crucial service, or distance from applications that negatively interfere) all contribute to the goodness of an assignment.

Scheduling is therefore sometimes subject to placement constraints. Such constraints are very common in practice: for example, 50% of Google WSC tasks have some form of simple placement constraint related to machine properties, and 13% of production workloads have complex constraints [SCH+11, §1].

There are three general types of constraints:

**Soft constraints** specify a “preferential” placement, indicating that a task benefits from the presence of a certain property. For example, an I/O-intensive workload such as log crawl-
ing might have a soft constraint for machines with a flash storage. The scheduler may choose to ignore a soft constraint and proceed to schedule the task anyway.

**Hard constraints** by contrast, *must* be met by the scheduler. A task with a hard constraint cannot be scheduled until a placement that satisfies its requirements is found. In the flash storage example, this constraint would be appropriate if the task cannot execute using a slower storage device – e.g. a fast, persistent log backing a distributed transaction system. Likewise, an application that requires a specific hardware accelerator would use a hard constraint.

**Complex constraints** can be hard or soft in nature, but are difficult for the scheduler to deal with. *Combinatorial constraints*, which depend not only on machine properties, but also on the other tasks running on the machine and on other concurrent placement decisions, are a prominent example of complex constraints. In the aforementioned flash storage example, a combinatorial constraint might indicate that only one task using the flash device may run on the machine at a time.

Constraints reduce the number of possible placements available to a given task and therefore lead to increased scheduling delays [MHC10; SCH11; ZHB11].

Many schedulers support constraints, but there is little consensus on the types supported. For example, Sparrow and Choosy support only hard constraints and use them as filters on possible assignments [GZS13; OWZ13]. Quincy, on the other hand, supports soft constraints via per-task placement preferences, but does not support hard constraints, as tasks have a wildcard “fallback” [IPC09, §4.2]. Quasar supports soft high-level performance constraints, but relies on the scheduler’s profiling and performance prediction mechanisms to satisfy them via corrective action [DK14].

By contrast, YARN’s resource manager supports both soft and hard constraints [VMD13, §3.2], and alsched [TCG12] also supports both, in addition to complex combinatorial constraints. However, tetrisched subsequently argued that support for soft constraints is sufficient and offers attractive benefits [TZK13].

Support for constraints is a trade-off between high-level scheduling policy expressiveness, scheduler complexity, and job wait time. Different cluster schedulers make different choices owing to differences in workloads and operating environments. My Firmament scheduler allows this choice to be made on a per-policy basis.

### 2.3.4 Fairness

Most WSCs are operated by a single authority, but nevertheless run a wide range of workloads from different organisational units, teams, or external customers [SKA13, §1]. These users may behave antagonistically in order to increase their share of the cluster resources. Consequently, cluster schedulers allocate shares and aim to provide *fairness*. 
CHAPTER 2. BACKGROUND

Some systems rely on task churn to converge towards users’ fair shares, resources are offered to users according to their fair shares, but running tasks are not preempted if the allocation becomes unfair. The Hadoop Fair Scheduler (HFS), which fairly shares a MapReduce cluster by splitting it into “job pools” [HFS], and Delay Scheduling [ZBS+10] are such churn-based approaches. The Sparrow distributed scheduler uses a similar approach: tasks experience weighted fair queuing at each worker, and fair shares of the cluster emerge as tasks are serviced at different rates [OWZ+13, §4.2].

Quincy [IPC+09], by contrast, preempts running tasks to enforce fair shares. It models the scheduling problem as a flow network optimisation, and enforces the updated fair shares whenever its solver runs. To guarantee progress, Quincy does not preempt tasks once they have been running for a certain time; hence, temporary unfairness is still possible [IPC+09, §4.3].

Some policies support fair shares over multiple resource dimensions: for example, Dominant Resource Fairness (DRF) offers multi-dimensional max-min fairness by ensuring that each user receives at least her fair share in all dimensions [GZH+11]. DRF has proven properties that incentivise users to share resources and to honestly state their demands [GZH+11, §3]. DRF variants also exist for fair allocation with regard to placement constraints (Constrained Max-Min-Fairness (CMMF) in Choosy [GZS+13]) and hierarchical allocation delegation (in H-DRF [BCF+13]).

While strong fairness is appealing, it is unclear how useful it is in a single-authority WSC. Anecdotally, many production systems rely on out-of-bands mechanisms to ensure approximately fair sharing [VPK+15, §2.5]. Furthermore, even though the scheduler may allocate fair resource shares, heterogeneity and interference (§2.2.2) can lead to significant differences in seemingly identical resource allocations. Firmament supports a basic notion of fairness, and can model more complex ones as part of its scheduling policies.

2.3.5 Dynamic resource adjustment

Most cluster scheduling systems assume that all jobs’ tasks either have uniform resource requirements (as, e.g. with fixed-size MapReduce worker “slots”), or that users specify resource requirements at job submission time (e.g. in Borg [VPK+15, §2.3], YARN [VMD+13, §3.2] and Mesos [HKZ+11, §1]).

Some cluster schedulers, however, support dynamic adjustment of resource allocations. This is beneficial to harness spare resources, to satisfy job deadlines, or to cope with varying load.

Omega’s MapReduce scheduler opportunistically allocates extra resources to increase the degree of parallelism when possible [SKA+13, §5]; Apollo likewise launches additional opportunistic tasks within jobs if their allocations are not fully utilised [BEL+14, §3.5].

Jockey [FBK+12] dynamically increases the resource allocation of a SCOPE job if it runs a risk of missing its deadline, and decreases it if there is headroom. Similarly, Quasar [DK14] automatically “right-sizes” resource allocations and chooses the best available resources, based
on co-location and machine types; it grows the resource allocation until the user-specified performance constraints are met. Finally, Borg’s “resource reclamation” mechanism dynamically reduces tasks’ resource requests to an envelope around their actual usage in order to reduce reservation slack and improve utilisation [CCB+14; VPK+15, §5.5].

These examples highlight that resource allocations can be dynamically adjusted by the scheduler. Most commonly, however, it is left to applications to introspect on their performance and request extra resources when necessary.

2.3.6 Summary

As I already discussed in Section 2.2.5, good scheduling decisions are essential to making efficient use of a WSC’s resources. There is a vast amount of existing work on cluster scheduling, but few existing schedulers meet all needs of increasingly diverse workloads.

In my work, I develop a WSC scheduler with the following goals.

A flexible WSC scheduler must:

1. have a modular, versatile architecture that allows for different degrees of centralisation and distribution of scheduling decisions (§2.3.1);

2. integrate machine-level information and cluster-level information to make scheduling decisions based on fine-grained profiling and locality information (§2.3.2);

3. allow flexible, deployment-specific scheduling policies to be used atop the scheduler, and support soft, hard, and complex constraints as part of them (§2.3.3);

4. derive high-quality scheduling assignments; and

5. make rapid decisions when required to do so.

Furthermore, other features are desirable, but not the core focus of my work. The scheduler should ideally:

6. allow for fairness between users to be enforced if desired, or for resource quotas and priorities to be applied (§2.3.4); and

7. support scheduling policies that perform dynamic resource adjustment and scale allocations as a function of cluster load or workload deadlines (§2.3.5).

My Firmament scheduler (Chapter 5) supports these goals, and the evaluation chapter reviews how well it meets them (§6.3.4).
Chapter 3

Designing a WSC operating system

"Those are my principles, and if you don’t like them...
well, I have others."
— attributed to Groucho Marx [Sha06, p. 498]

In the previous chapters, I have highlighted several challenges that arise from the use of traditional operating systems in warehouse-scale computing (WSC) environments. In this chapter, I discuss the principles that I believe a special-purpose operating system for WSCs should adopt to address these challenges. I also present the high-level design of DIOS, a new OS implemented according to said principles.

As noted in Chapter 1, much of the functionality I describe may also be supplied via libraries or “middleware” in user-space. Yet, there is still a case for supplying it from the kernel. The OS kernel is both privileged and omniscient. It is privileged because it has the ultimate authority over all system resources and may side-step or change otherwise immutable isolation barriers. It is also omniscient, because it has a global view of the system. Consequently, the kernel can make decisions that take into account the overall state of resources combined with application-specific information.

In general, an operating system kernel:

(i) names and locates system objects (I/O targets, devices, and programs),
(ii) allocates and manages hardware resources, virtualising them where necessary, and
(iii) effects privileged operations on the behalf of user applications, while isolating them from each other.

— [SGG08, pp. 3–6; TW06, pp. 4–6; LM88, pp. 2–10].

A distributed WSC OS is no exception to this: it must expose abstractions to user applications that support these ends. However, it must also operate distributedly across hundreds or thousands of nodes, coordinate them reliably, and deal with the inevitable faults. On the upside,
however, the distributed WSC OS has access to significantly more information about the state of the distributed system than a traditional OS.

In this chapter, I outline the key design principles for a distributed WSC OS (§3.1) and then focus on individual areas in which a WSC OS differs from traditional approaches:

1. the use of **distributed objects** as a central building block (§3.2);
2. the need to provide **scalable, translucent, and uniform APIs** and abstractions (§3.3);
3. the sufficiency of a **narrow, specialised system call API** for applications (§3.4);
4. the utility of **capability-based protection** in a distributed system with failures, (§3.5);
5. the ability to simplify the OS by relying on a **flat storage layer** (§3.6); and
6. the opportunity to support **OS-managed distributed I/O** using a request-based I/O model with additional safety through kernel-managed buffers (§3.7).

I illustrate my explanations with reference to the DIOS operating system, which I describe in Chapter 4.

### 3.1 Design principles

Based on the goals identified in Chapter 2, a distributed operating system for a modern WSC must have the following properties:

1. No single component of the system can hold state whose loss would disable the system.
2. All synchronisation is explicit; no OS operation or primitive *implicitly* (i.e. without user awareness) synchronises across WSC nodes.
3. Applications and libraries are able to make and implement policies of their own choosing and the OS mandates minimal mechanism.
4. Tasks are strongly isolated from each other, data-flow invariants can be enforced between them, and their I/O targets can be restricted.
5. Heterogeneity within the WSC is represented in the OS and can be exploited by the OS and applications alike.
6. Co-location interference between tasks is avoided as far as possible.

In the following, I outline six principles that help supporting these properties.
**Principle 1: fully distributed architecture.** Replication of state helps to enable scalable operation and tolerance of component faults; the WSC OS must therefore never rely on singular centralised state.

All data stored in the system must be replicable, no central authority must be required to maintain meta-data or permissions, and recovery from failures must be possible by other parts of the WSC taking over the functionality of failed ones.

*This principle is supported by the use of a distributed object model (§3.2) and distributed capability-based security (§3.5).*

**Principle 2: no implicit synchronisation.** As synchronisation in a distributed system is expensive, it is best avoided. The WSC OS must thus *inform* users of concurrent access to data, but should not prevent it by default. User-space infrastructure may construct synchronised abstractions on top of these notifications.

To achieve this property, the WSC OS must strive to base its abstractions on scalable design principles, and make no assumptions of mutual exclusion.

*This principle is supported by scalable abstractions (§3.3) and a request-based I/O system with built-in consistency meta-data (§3.7).*

**Principle 3: externalised policy.** Distributed applications are complex and heterogeneous, and many policy decisions are application-specific. A WSC operating system must strike the right balance between specificity of mechanism and flexibility in policies.

Sufficient information must be exposed to user-space for applications to enact common distributed systems policies – such as where to locate data, or whether to maintain strongly consistent replicas – of their choice.

*This principle is supported by the distributed object model (§3.2), a narrow system call API (§3.4), and a flat storage system (§3.6) suitable as a basis for further application-specific abstractions.*

**Principle 4: translucent abstractions.** Despite the inherent complexity of the distributed environment it represents, the abstractions exposed by the WSC OS should be accessible to users. Transparency can help, but also hides information and complicates optimisation.

A distributed WSC OS should thus be *translucent*: offering simple, transparent abstractions by default, yet exposing sufficient information to users to allow optimisation of their applications.¹

*This principle is supported by the use of transparent and uniform abstractions (§3.3) in combination with capabilities carrying additional meta-data (§3.5).*

¹The term “translucency”, meaning “transparency with exposed meta-data”, also sees occasional – though rare – use in programming language literature, e.g. by Rozas [Roz93].
Principle 5: **strong isolation and data-flow enforcement.** A WSC is fundamentally a shared environment, in which different users’ and jobs’ tasks must be isolated from each other both for data confidentiality and predictable performance.

Therefore, the WSC OS must be able to restrict tasks’ access to data and resources via managed namespaces, sandbox code effectively, and explicitly track all communication and data access. This isolation, unlike in a traditional OS, must also extend across machines.

*This principle is supported by a capability model built on identifier capabilities and handle capabilities (§3.5), and a restriction to a narrow API (§3.4).*

Principle 6: **incremental adoption path.** Even though a new WSC OS brings benefits, porting software to it is time-consuming. Hence, an incremental migration to running increasingly larger portions of a WSC’s workload within the new OS must be feasible. While not all of the benefits of the new OS might initially be attainable, each successive migration step ought to offer further improvement over the previous state.

*This principle is supported by the backwards-compatible host kernel extension method, which I describe in Section 4.9.*

In the following sections, I will discuss how these design principles can be realised with reference to my DIOS OS kernel (Chapter 4). The Firmament scheduler (Chapter 5) addresses the remaining desirable properties: it captures heterogeneity and avoids co-location interference.

### 3.2 Distributed objects

Objects are a convenient abstraction both for programming and structured data storage. While the definitions differ between these areas, their lowest common denominator is perhaps that an object is a collection of related, structured state. Objects are created and destroyed atomically, and often have unique identifiers.

An object abstraction has several advantages for distributed systems:

1. Objects impose structure on otherwise unstructured binary data and delineate scopes for updates at different consistency levels. For example, Amazon’s Dynamo key-value store holds versioned objects that can diverge [DHJ+07, §4.4], while Google’s BigTable store guarantees consistent updates within a row object, but not across row objects [CDG+06, §2].

2. Object-level replication enables fault tolerance. An object encapsulates all state required to act as an independent, replicated entity. For example, distributed key-value stores such as Cassandra [LM10, §5.2] replicate keys and their corresponding objects across fault
domains to achieve reliability. Data processing systems’ object abstractions – e.g. Spark’s resilient distributed datasets (RDDS) [ZCD12] and CIEL’s data objects [MSS11] – offer fault tolerance based on replication and deterministic replay.

3. Objects simplify distributed programming. They lend themselves a communicating actor model: for instance, Sapphire expresses complex distributed systems concisely by using transparent interactions between objects with modular “deployment managers” [ZSA14, §8.1].

4. Dependencies between objects enable data-flow computation models that lend themselves to automatic parallelisation. For example, Dryad [IBY07] and Naiad [MMI13] are based on data-flow of records between stateful vertices, while CIEL schedules dynamically generated tasks based on their dependencies on input objects [MSS11, §3.1].

These, combined with the requirements of a fully distributed system (Principle 1, §3.1) and support for extensible user-level policies on top of the abstraction (Principle 3, §3.1), make a distributed object model a good fit for a WSC operating system.

DIOS adopts a distributed object model at its core, but uses a deliberately weak notion of an “object”. It only assumes that:

(i) an object is either a passive blob (a sequence of bytes), a streaming communication endpoint (a FIFO channel), an active task (a process), or a group (a namespace);
(ii) an object’s instances (≡ replicas) may be used interchangeably;
(iii) an object is created and deleted, but not necessarily updated, atomically;
(iv) object meta-data that applications may wish to inspect to make policy decisions are carried by the handles used to interact with an object.

Specifically, DIOS – unlike many previous distributed OSes (§2.1.2) – does not:

(i) assume any (specific) language-level integration of its object notion;
(ii) impose any requirements on the structure of objects, e.g. storing references to other objects in a particular way, or coupling code and data;
(iii) enforce a specific consistency level for object replicas; or
(iv) guarantee the continued availability of a live object in the presence of failures.

This object notion is practical for an OS as it makes minimal assumptions about the implementation of user applications. In the following, I briefly contrast it with the object notions in prior systems and indicate where I have taken inspiration from them.

Related approaches. Classic distributed operating systems research coincided with the development of object-oriented programming languages. As a result, many previous distributed OSes were object-based:
HYDRA [WCC+74] was an early “distributed” OS with a flat name space of uniquely identified “objects”, which represented procedures (data and code), name spaces, and processes. To interact with an object, its identifier and a kernel-validated capability (“reference”) were required. This is similar to the separation between identifier and handle capabilities in my work (§3.5).

Eden’s “Ejects” (Eden objects) interacted using messages (“invocations”) authenticated via capabilities [LLA+81]. To ease programming, Ejects were mobile and invocation-transparent, i.e. the user had no knowledge of an Eject’s location, and the Eject could be checkpointed for suspension and replication. Contrary to initial plans, Eden was ultimately implemented in user mode rather than in the kernel [ABL+85].

Argus [LCJ+87] emphasised fault-tolerant object abstractions (using “guardians” and handlers), but assumed language integration (like Eden and unlike my work). In the Clouds OS [DLA88], passive, persistent objects corresponding to virtual address spaces stored code and data, and active “threads” entered and left the objects to run computations. Unlike Eden, Argus, and my work, Clouds supported atomic actions and cross-object transactions.

The V operating system [Che88] was based on message-oriented communication between objects and object servers. Each object had an “object manager”, which could itself be replicated. Objects also had associated identifiers in a three-level hierarchy (human-readable names, object identifiers and entity identifiers). Like my work, V emphasized object replication and a uniform I/O interface (§3.7).

In Amoeba [MRT+90], typed objects were managed explicitly by user-space servers. To perform an operation on an object, the user first had to present a capability for the object to the managing machine. Amoeba left the responsibility for synchronisation and parallel programming abstractions atop the distributed object model to user-level infrastructure (e.g. the Orca language [BKT92]), similar to my work (§3.4).

Although these systems operated at a much smaller scale than a WSC; yet, many of their concepts apply to the WSC environment, too. Object-based data-flow computing models, such as CIEL’s object model based on typed references [Mur11, §4], are another, more recent, influence.

### 3.3 Scalable, translucent, and uniform abstractions

The abstractions offered by a distributed WSC OS must satisfy the requirements of large-scale distributed systems:

1. They must be *scalable*, so that the system can run across many machines and serve a large number of jobs. For example, different replicas of an object should be concurrently accessible without synchronisation.
2. They need to be translucent, so that user code can transparently interact with other parts of the system, but also has sufficient information to optimise application performance when required. For example, the access to a remote object should not require implementing the wire protocol, but expose the information that the object is remote.

3. Abstractions for different mechanisms should be easy to comprehend, and, as far as possible, uniform. For example, since the location of an object is not typically known in advance, the abstractions for accessing local and remote objects should be similar.

**Scalability** can be derived from the scalable principles developed for single-machine operating systems on many-core architectures (cf. §A.1.3): like a scalable multi-threaded program, a scalable distributed system must exploit asynchrony and coordination-free sharing, and avoid global atomic operations. The OS system call API should respect these principles and encourage commutative implementations where possible [CKZ+13].

DIOS ensures scalability by building on these insights: it avoids hierarchical structures (e.g. via a flat name space) and enables concurrency by reducing dependencies between operations (e.g. by generating identifiers independently rather than using shared counters). This supports the avoidance of implicit synchronisation (Principle 2, §3.1). Section 4.8 discusses the scalability of the DIOS system call API in detail.

**Translucency** serves to simplify the system for the user without restricting their freedom to optimise. It emerges from making abstractions transparent by default, but opaque on request: this obviates the need to understand the detailed low-level operation of a distributed system without making it impossible to introspect on.

Distributed systems commonly support transparency – consider the remote procedure call (RPC) concept [BN84], Mach’s transparent message-based object access via ports [SGG08, app. B.5], or the automated dependency tracking in Dryad [IBY*07] and CIEL [MSS*11]. However, fully transparent interaction has the disadvantage of unpredictable performance. For example, every operation in distributed shared memory systems (e.g. Mungi [Hei98]) and transparent message-passing distributed object systems (e.g. Mach [ABB*86] and MIKE [CNT*93]) may require communication with a remote machine and the consequent latency. Yet, transparency means that the application cannot detect this in advance.

In DIOS, OS abstractions themselves are transparent, but object handles carry contextual metadata that expose detailed information about the object. The application may choose to ignore this information (and forgo predictable performance) or may use it to optimise its choices (e.g. choosing a more proximate object replica). This translucent approach (Principle 4, §3.1) helps externalise policy decisions to user applications (Principle 3, §3.1); Section 4.5 describes the implementation in DIOS.

**Uniformity** of abstractions simplifies a system, but is only sometimes available in current operating systems. For example, the BSD sockets API offers transparency over several streaming
and datagram-based network protocols, but interaction with local files or IPC between processes uses completely different abstractions.

An object-based WSC OS should have uniform abstractions as every interaction target is an object. Objects should also be described in a uniform way: identifiers are globally meaningful descriptors for logically equivalent object instances, while handles to concrete physical objects are valid only within a specific context. Handles may only be transferred across context boundaries with appropriate transformation by the kernel. I further explain the uniform identifier and handle capabilities in Section 3.5 and describe a uniform I/O model in Section 3.7. Sections 4.3–4.6 explain their realisation in DIOS via object names, references, and the system call API.

### 3.4 Narrow system call API

Uniform and translucent abstractions reduce the number of operations DIOS must expose via its system call API. This benefits scalability and safety: analysing a smaller API for its scalability is a more tractable undertaking, and validating the safety of abstractions is simplified by having fewer of them.

The recent resurgence of interest in library operating systems, which run a full OS kernel API atop a typically much smaller host ABI, testifies to the timeliness of this observation. Drawbridge, for example, defines a deliberately narrow ABI of 36 calls to virtualise an underlying host kernel towards the library OS [PBH+11]. Likewise, the Embassies project devised a deliberately minimal “client execution interface” (CEI) for “pico-datacenters” in web browsers, consisting of only 30 CEI calls [HPD13, §3.1]. By contrast, Linux offers 326 system calls on the x86-64 architecture.²

The key question is whether a small set of operations is sufficient to support the “data plane” of common WSC applications efficiently: if so, we can get away with a narrow API. In the following, I present an exploratory study intended to offer some insight into this question.

**Experiment.** To assess how many of the system calls are actually used – commonly or at all – by typical WSC applications, I investigated four applications: Hadoop MapReduce, the Redis key-value store, the Zookeeper coordination service, and the GraphChi graph computation framework. Each application was benchmarked using a typical workload, and its system call invocations were monitored using `strace`.

Figure 3.1a compares the total numbers of distinct Linux system calls invoked by the different applications. For Hadoop, I measured the four different system components into which a typical Hadoop cluster is decomposed: (i), the MapReduce JobTracker (“Hadoop-JT”), which coordinates job execution; (ii), the MapReduce TaskTracker (“Hadoop-TT”), which represents

²As of kernel version 3.14.
a worker node; (iii), the HDFS NameNode (“Hadoop-NN”), managing HDFS meta-data; and (iv), the HDFS DataNode (“Hadoop-DN”), that reads and writes bulk data.

Hadoop, running atop the Java Virtual Machine (JVM), uses the largest number of distinct system calls at 85; the likewise Java-based ZooKeeper, however, only uses 28, while C/C++-based Redis and GraphChi use 19 and 20 distinct system calls, respectively. This suggests that the system call pattern does not primarily depend on the compiler or runtime, but rather that it is an inherent property of data-centric applications. The set of distinct calls shrinks further if we consider only commonly invoked system calls which either (i) account for more than 1% of the calls made, or (ii) occur at least once per application-level request. Indeed, five or fewer “common” system calls exist for all applications apart from Hadoop (Figure 3.1a).

In Figure 3.1b, I group the system calls by category. Perhaps unsurprisingly, the dominant categories are I/O, synchronisation, and resource management, with I/O system calls dominating as we might expect in data-intensive WSC applications.

Figure 3.1c shows the relative frequency of each system call that contributes more than 1% of the total invocations. 35–50% of the total system call invocations are \texttt{read(2)} or \texttt{write(2)}; the next most frequent calls are those relating to Linux’s futex (“fast user-space mutex”) syn-
chronisation mechanism. All in all, a total of eleven system calls cover 99% of the invocations made. Compared to the 326 Linux system calls, this is a small subset, suggesting that the breadth of the Linux system call API is not required on the “data-plane” of these WSC applications.

However, while these results hint that a narrow system call API may be sufficient for many WSC applications, one ought to be cautious. Many of the extra system calls offered in Linux exist solely for backwards compatibility and are rarely used, but others, such as `ioctl(2)`\(^3\) and `fnctl(2)` have highly overloaded semantics and may have completely different effects despite only appearing once in this study.

Yet, a small number of system calls are clearly responsible for the majority of the kernel interaction in WSC applications. This suggests two conclusions for the design of D\(\text{iOS}\):

1. If D\(\text{iOS}\) implements efficient, scalable equivalents of the small set of common system calls, it ought to attain competitive performance in most cases. Other, rarer system calls may need to be emulated using multiple calls or less optimised invocations, but should have negligible impact on overall performance.

2. Restricting the system call API reduces the number of implementations that require auditing. As rarely used code paths are known to beget vulnerabilities, this should increase the operating system’s security.

Consequently, the D\(\text{iOS}\) system call API is narrow: it offers only thirteen system calls to user-space applications. However, it is worth noting that the WSC OS system call API does not have to suit every end-user’s needs: it must only expose the lower-level facilities needed by infrastructure systems (e.g. MapReduce, or a key-value store) and libraries (e.g. an RPC library), while leaving them to implement higher-level abstractions (Principle 3, §3.1).

### 3.5 Identifier and handle capabilities

A WSC operating system must ensure that tasks belonging to different jobs, users, or even commercial entities, can safely share the WSC infrastructure. To achieve this, it must ensure three properties:

- **Isolation** between users and tasks, and their data and resources, unless explicitly shared.
- **Deniability** of the existence of data inaccessible to a task or user.
- **Auditability** of any sharing or communication of data, and of granting of access to them.

\(^3\)I use the conventional Unix manual page notation, `name(section)` throughout this dissertation.
Furthermore, a WSC OS must be able to express useful application-level sharing policies. For example, access to web log files containing sensitive information may be restricted to tasks within specific analysis jobs, but these tasks might need to delegate work to helper tasks (e.g. a decompression routine) without leaking sensitive data outside the trusted job.

OS kernels today implement process isolation within a machine using e.g. virtual address spaces and kernel namespaces. Access control in distributed systems, however, is left to the applications. Access control implementations are typically based on Access Control Lists (ACLs) or capabilities [SS75; Bac98, p. 61].

**Access Control Lists** store a list of subjects (e.g. users) and their access permissions with each object. This approach is used in most commodity operating systems (e.g. POSIX file system permissions).

**Capabilities** carry a set of permissions that can be invoked by presenting them to an object [DV66]. In other words, the possession of a capability *intrinsically* authorises the holder to invoke the permissions [Lev84]. Common examples of capabilities are cryptographic keys and unforgeable URLs authorising access to web resources.

As a WSC OS must scale to thousands of machines, centralised access control relying on a single credential repository is not viable (cf. Principle 1, §3.1). While both ACLs and capabilities can be managed distributedly, capabilities particularly lend themselves to distributed use as they require no authentication of a subject. Moreover, capabilities map well to the data-flow abstractions common in distributed systems. Their traditional drawbacks – complex programming models and difficulty of revocation – matter less in the WSC environment, since other mechanisms must address the same difficulties. (Consider, for example, the difficulty of revoking an ACL entry during a network partition).

**Requirements.** Traditional capability schemes often rely on machine hardware to protect their capabilities. For example, the CAP Computer [WN79], the iAPX 432 [Lev84, §9.3.2], and CHERI [WWC+14] use custom hardware or hardware extensions, while Mach [ABB+86], EROS [SSF99], seL4 [KEH+09], and Barrelfish [BBD+09] rely on memory protection hardware to separate capability space and data space.

In a distributed OS, capabilities must *i*) be able to cross machine boundaries, and *ii*) use an existing shared interconnect. This makes hardware protection on its own insufficient. Instead, it must be possible to exchange *capabilities as data* via ordinary interconnect messages.

Moreover, the objects protected by capabilities are frequently remote and may be mobile. Distributed systems avoid embedding assumptions about object locality by using transparent abstractions (§3.3). Thus, capabilities must be *location independent*: using pointers as capabilities, for example, is incompatible with this requirement.
Finally, the WSC OS must be able to generate restricted versions of a capability. This is useful, for example, when a helper task needs restricted access to only a subset of the capabilities available to the parent task. Such delegation of restricted capabilities requires support for the refinement\(^4\) of capabilities.

In the following, I discuss prior distributed capability schemes’ support for these features. I then explain how a split into identifier and handle capabilities satisfies all three requirements of capabilities as data, location independence, and refinement.

**Related approaches.** Several previous distributed systems have made use of different types of capabilities (Table 3.1):

- In Eden, capabilities authenticated “invocations” on “Ejects” [LLA+81]. Initially envisaging to use hardware capabilities (“Access Descriptors”) in the Intel iAPX 432 [Lev84, §9.3.2], Eden’s capabilities were eventually implemented in software, stored in user virtual memory, and mirrored in the (Unix user-space) “kernel” [ABL+85, p. 50]. Ejects were distributed and capabilities were location-independent, i.e. if an Eject was moved or copied, existing capabilities continued to work.

- Accent [RR81] and Mach [ABB+86] used capabilities to control access to “ports” (IPC endpoints). Both used message-based IPC between different OS subsystems and user-space applications, granting location independence. Capabilities – which granted access to ports, but could not be refined – could be sent as data within messages. In Mach, the **Table 3.1:** Previous systems’ distributed capability schemes compared to DIOS.

<table>
<thead>
<tr>
<th>System</th>
<th>Capability store location</th>
<th>Secured by</th>
<th>Capabilities as data</th>
<th>Location transparent</th>
<th>Refinement possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eden [LLA+81]</td>
<td>user-space + kernel</td>
<td>MMU</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Accent [RR81]</td>
<td>user-space + kernel</td>
<td>MMU</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Mach [ABB+86]</td>
<td>user-space + kernel</td>
<td>MMU</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Amoeba [MRT+90]</td>
<td>user-space</td>
<td>cryptography</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Plan 9 [CGP+02](†)</td>
<td>kernel</td>
<td>cryptography</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Barrellfish [BBD+09]</td>
<td>kernel (monitor)</td>
<td>MMU</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Web URLs</td>
<td>user-space</td>
<td>cryptography</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>Macaroons [BPE+14]</td>
<td>user-space</td>
<td>cryptography</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DIOS names (§4.3)</td>
<td>user-space + kernel</td>
<td>cryptography</td>
<td>✓</td>
<td>✓</td>
<td>X</td>
</tr>
<tr>
<td>DIOS references (§4.5)</td>
<td>user-space + kernel</td>
<td>MMU + crypto.</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
</tbody>
</table>

\(†\) Capabilities in Eden have corresponding kernel meta-data, so they cannot be treated as ordinary data.

\(‡\) Plan 9 uses capabilities only locally to empower its factotum server to authenticate changes of user ID.

\(^4\) Sometimes also called “reification” or “minting” in capability literature.
sending of port capabilities in messages had “move” semantics: the sender lost access to
the port when delegating the capability [SGG08, app. B.5].

- The Amoeba distributed operating system used sparse cryptographic capabilities that
  were managed entirely by user-space server processes [TMV86]. Consequently, capa-
  bilities could be treated as ordinary data and exchanged over the network. To protect
capabilities against forgery, Amoeba used either hardware support for one-way functions
or a cryptographic scheme. The latter relied on unforgeable, hardware-enforced source
addresses being added to network messages.

- Barrelfish uses a distributed capability scheme based on the segregated seL4 capabili-
ties [SEL4RM, §2.2, §2.5], which arranges capabilities in a tree of “CNodes” within each
kernel’s “mapping database” (MDB) [Nev12, pp. 29–30]. When capabilities are shared
across cores or machines, they must be serialised and sent via a secure channel [BFTN0,
p. 10]. Barrelfish, unlike the previous systems, does not use a transparent distributed ob-
ject abstraction at the OS level. Hence, capabilities are location-specific, referring to, for
example, a specific region of memory.

- Many web applications use cryptographically derived URLs as capabilities. Sharing
  such a URL amounts to delegating (copying) the capability, although the URL itself can-
  not be refined further by recipients. Likewise, web cookies and Macaroons [BPE+14] are
effectively distributed capabilities. All of these capabilities are location-independent, and
  can be stored and communicated as raw data.

Capabilities for a WSC OS. Some prior capability schemes (e.g. Amoeba’s), meet all three
requirements set out above. However, it is not desirable for the same capability type to meet
all requirements: a refineable capability type that cannot be treated as data is useful, as it can
carry context-dependent information and cannot easily be leaked. Hence, a combination of two
distributed capability types is the best fit for a WSC OS:

1. Global, coarse-grained identifier capabilities are generated cryptographically and can be
treated as data (i.e. stored and communicated). Identifier capabilities are atomic, i.e. they
cannot be subdivided or refined, but they can be resolved into handle capabilities.

2. Local, context-sensitive handle capabilities that cannot be treated as data, and which
are segregated between the kernel and user-space. Handle capabilities can be refined via
kernel mediation, and unlike identifiers carry context-dependent, read-only information.

Together, identifier and handle capabilities enforce mandatory access control (Principle 5, §3.1).
An identifier capability must be resolved to a local, context-sensitive handle capability and can-

\[^5\text{Consider, for example, the “share via link” functionality in Google Docs (http://docs.google.com) or Doodle (http://doodle.com).}\]
not be passed directly as an argument to system calls (with one exception, viz. identifier resolution). By contrast, handle capabilities are only valid within their context, usually a particular task.

This split also controls capabilities’ provenance: in DIOS, the resolution of identifier capabilities respects resolution scopes (namespaces) and generates new handle capabilities (§4.3.3), while handle capabilities can only be transferred to other tasks via the kernel (§4.5.2).

**Cross-machine delegation.** When capability-based protection is extended across machines, delegation involves sending a capability over the interconnect. This requires unforgeability of capabilities to be guaranteed across machines. Cryptographic generation of sparse capabilities ensures that they are unguessable, but does not address other issues specific to a distributed system:

- **Man-in-the-middle attacks** can occur when capabilities are intercepted on the shared interconnect: for example, a compromised switch might further delegate a copy of an observed capability to a colluding host by impersonating the delegation protocol.

- **Replay attacks** involve the recording and later re-use of a capability. While it is possible to ensure the freshness of handle capabilities, identifier capabilities can be snooped and replayed easily.

Authentication protocols such as Kerberos [SNS88] or the Needham-Schroeder-Lowe authentication protocol [Low95] solve this problem, but require a logically centralised authentication server to maintain shared secrets (Kerberos) or keys (Needham-Schroeder-Lowe), and thus partially violate the principle of full distribution in the WSC OS (Principle 1, §3.1).

Since WSCs are operated by a single authority, stronger assumptions about the interconnect integrity can often be made: a compromise of switch firmware, although possible, is rather unlikely. In addition, most WSC operators encrypt even internal data centre traffic [EFF13], which helps defend against snooping.

For the purpose of DIOS, I consider such threats to be out of scope and assume an uncompromised kernel and interconnect. The secure delegation of capabilities across untrusted commodity interconnects is, however, an interesting area for future work (see §7.1.3).

**Summary.** The capability scheme described satisfies the three goals of isolation, deniability, and auditability:

1. Unless identifier capabilities’ data are leaked or handle capabilities wrongfully delegated, *isolation* is ensured as a result of capabilities’ unforgeability. Identifier capabilities are unguessable as they are based on cryptography, and handle capabilities are unforgeable as they are segregated (i.e. have a kernel counter-part). Since identifier capabilities must
be resolved and handle capabilities can only be delegated via the kernel, leakage is con-
tainable if it does happen.

2. The resolution requirement for identifier capabilities enforces *deniability*: when resolving
an identifier capability, the kernel can deny the existence of any objects corresponding to
the capability by returning an empty set of handle capabilities. This is indistinguishable
from the case in which no object in fact exists.

3. *Auditability* is guaranteed because the kernel creates all handle capabilities. When re-
solving an identifier capability, the request must pass an audit hook in the kernel name resolution code. Likewise, delegation of a handle capability passes the kernel audit hook before creating a new capability. Since all system calls other than name resolution require one or more handle capabilities, auditability is maintained by adding audit hooks to the system call handlers.

### 3.6 Flat persistent object store

The storage systems in WSCs are often more reminiscent of the flat data stores in early operating systems than of the complex hierarchical file systems offered today (e.g. ext4, NTFS, or NFS).

Storage systems such as BigTable [CDG+06] and Dynamo [DHJ+07] are flat key-value stores representing a distributed map, while others, like FDS [NEF+12], Haystack [BKL+10] and f4 [MLR+14], are distributed blob stores. While hierarchical distributed file systems exist (e.g. GFS [GGL03], HDFS [SKR+10], and TidyFS [FHI+11]), they have far more restricted semantics than “legacy” file systems. For example, HDFS supports only appending writes on existing files, rather than random write access [SKR+10, §3.A]. If needed, directory services are implemented user-space systems, while the actual data are stored as flat blocks (e.g. 64 MB chunks in GFS and HDFS, 8 MB “tracts” in FDS).

A WSC OS therefore has the opportunity to simplify the storage subsystem of the OS kernel. This can both simplify the storage stack and improve performance if the abstractions are a better match for WSC systems.

The ideal storage interface for a WSC is a minimal one: the OS offers unique names and a flat object store abstraction, with hierarchical abstractions being implemented on top of this by individual infrastructure applications. This approach has the benefit of maximal flexibility for the user-space infrastructure code implementing storage systems, as it externalises policy decisions (Principle 3, §3.1). For example, both BigTable-style distributed maps and a GFS-style distributed file system can be implemented on top of a simple object store, with protection relying on OS-enforced capabilities (§3.5).

Objects within the store may be replicated, and replicas can have different levels of persistence. For example, some replicas may exist only in volatile memory, some on durable storage, and
some might be memory-mapped with a persistent backing copy. Properties of this nature can be indicated as part of the handle capabilities’ meta-data.

The implementation of the persistent object store in DIOS is similar to the object stores in Opal [CLF94, §4.4] and CIEL [Mur11, §4.5]: persistent objects are stored on disk, and their meta-data is maintained in a durable log or an underlying local file system.

**Related approaches.** A flat object store model is similar to the Multics segment model, an early OS storage abstraction. In Multics, segments were named blocks of memory which could be backed by persistent storage or another device [VCG65, pp. 207–8]. Multics transparently abstracted the segments’ actual storage location and moved segments automatically and transparently between different locations [DN65]. By contrast, “translucent” handle capabilities allow introspection on the storage location of objects in the WSC OS (Principle 4, §3.1), and the OS does not automatically move them.

Many later single address space OSes, such as Grasshopper [DBF94], Opal [CLF94], and Mungi [Hei98], also supported persistent segments or objects, and recent high-performance network attached storage (NAS) solutions adopt similar object-centric models [CWM+14, §4].

### 3.7 Request-based I/O with kernel-managed buffers

Most WSC applications are I/O-intensive: they either communicate with other applications, or process large amounts of streaming data (from the network) or archival data (from durable storage). Oftentimes, different applications are combined into data processing pipelines or multi-component services.

In the WSC OS, these workloads frequently involve access to remote objects. Moreover, objects may be shared between tasks, and depending on the object and the applications’ consistency requirements, certain concurrent accesses might be acceptable. The WSC OS should not dictate a specific concurrent access semantic, but instead externalise the policy to the application (Principle 3, §3.1). Transaction-like **I/O requests** (§3.7.1) are a flexible and low-overhead way of tracking concurrent accesses and notifying applications when they take place.

I also argue that a WSC OS kernel should allocate and manage I/O buffers for applications, rather than allowing user-space code to specify arbitrary buffers (§3.7.2). This helps support the I/O request system and defends against several classes of exploits related to I/O buffers.

#### 3.7.1 I/O requests

An **I/O request** delineates the time between the acquisition of I/O resources (i.e. a buffer for reading or writing) and the completion of the intended operation (viz. reading and processing data, or writing them to the buffer). An I/O request is either a **read request** or a **write request**.
CHAPTER 3. DESIGNING A WSC OPERATING SYSTEM

When acquiring an I/O request to begin with, the application specifies the desired concurrent access semantics for the object and receives an indication of whether they can be provided at this point. When committing an I/O request, the application finalises the request and asks the OS to validate whether the concurrent access semantics did indeed hold for its duration.

Hence, I/O requests are conceptually similar to optimistically concurrent transactions (as e.g. in Farm [DNN+15]). However, I/O requests do not offer transactional semantics (e.g. ACID). Each I/O request is also specific to an individual object, i.e. it cannot offer multi-object consistency semantics, which must be implemented by the application if desired.

Figure 3.2 illustrates how I/O requests proceed, and how the flow of data and control differs between read and write requests.

Read requests receive their data in the acquire stage, or fail to acquire if the requested concurrent access semantics cannot be satisfied by the object in its current state (e.g. because other tasks already have outstanding I/O requests on it). The validity of the data read is checked at the commit stage, which may fail if the desired concurrent access semantics were violated during the request.

Write requests initially receive a working buffer on acquire. Depending on the type of the target object, this buffer may or may not already contain existing object data. The task then writes to the buffer, and the commit stage indicates if the concurrent access semantics held during the request.

For write requests, the effect of a failed commit depends on whether the object exposes raw buffers (e.g. with a raw shared memory object), shadow copies (e.g. a double-buffered shared memory object), or appends writes sequentially (as with streams). With sequentially applied
writes or shadow copies, the changes are easily discarded. In the case of an exposed raw buffer, however, they have already been applied at the commit point. In this case, the request is considered invalid and the data may potentially be corrupted; the application must handle this appropriately.

I/O requests are optimistically concurrent: in other words, the assumption is that they eventually succeed after potential back-offs and re-tries. There are no implicit OS-level per-request locks (Principle 2, §3.1) that can enforce progress, because they would require costly and unscalable distributed locking protocols. However, the acquire and commit points of I/O requests offer “hooks” for constructing application-specific and multi-object mechanisms. For example, a distributed mutex lock can be built atop I/O requests (see example below).

**Concurrent access semantics options.** The I/O request model can offer many different concurrent access semantics, but DIOS restricts itself to the relatively simple options shown in Table 3.2. The four levels are exclusive access (EXCL), multiple-reader, single-writer (MRSW), single-reader, multiple-writer (SRMW), and unrestricted (NONE).⁶

To enforce these concurrent access semantics, DIOS associates four atomic counters with each object:

1. the number of active read requests on the object, \( a_r \);
2. the number of active write requests on the object, \( a_w \);
3. the read version number, \( v_r \); and
4. the write version number, \( v_w \).

\( a_r \) and \( a_w \) are incremented when the acquire for a new I/O request is handled, and decremented once the commit stage has finished; \( v_r \) and \( v_w \) are incremented on successful commit of a read or write request, respectively. The counter updates and the application of changes on a successful write request are performed under local mutual exclusion. This might be improved upon using well-known techniques for scalable counters [ELL+07; CKZ13, §3.1] and deferred updates (e.g. lazily reconciling copy-on-write pages).

In addition to the high-level indication as to whether an I/O request succeeded or failed, DIOS

<table>
<thead>
<tr>
<th>Level</th>
<th>Readers</th>
<th>Writers</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXCL</td>
<td>1 (either)</td>
<td></td>
<td>Mutual exclusion (sequential consistency).</td>
</tr>
<tr>
<td>MRSW</td>
<td>*</td>
<td>1</td>
<td>Multiple-reader, single-writer (cannot corrupt data).</td>
</tr>
<tr>
<td>SRMW</td>
<td>1</td>
<td>*</td>
<td>Single-reader, multiple-writer (cannot read stale data).</td>
</tr>
<tr>
<td>NONE</td>
<td>*</td>
<td>*</td>
<td>No consistency guarantee.</td>
</tr>
</tbody>
</table>

Table 3.2: The four concurrent access semantics supported by I/O requests based on simple read/write versions and reader/writer counters.
also exposes the “raw” version counters via handle capabilities. This enables custom consistency policies spanning multiple I/O requests to be implemented on top.

The version counter mechanism is useful, but it does not impose a “happens-before” relationship on requests issued by different tasks. It could be adapted to use Lamport clocks [Lam78], which enforce a partial ordering on events in a distributed system. Once a happens-before relationship is established, multi-replica consistency schemes such as version clocks (cf. Dynamo [DHJ+07]) and causal consistency [BGH+13] can be implemented.

Examples. To illustrate I/O requests, consider the example of two tasks ($T_A$ and $T_B$) both accessing a blob object in shared memory. Assume that both tasks initiate a read-only I/O request, with $T_A$ specifying EXCL as its desired access semantic, while $T_B$ specifies MRSW. Figure 3.3 shows several different ways in which the requests may interact. Since $T_A$’s request asks for exclusive access semantics, it fails in all situations apart from the disjoint setting in Figure 3.3a. Note that if $T_A$’s request was a write, $T_B$ would fail on acquire in Figure 3.3b.

The I/O request system can also be used to implement synchronisation primitives across tasks. Consider a global mutex: by having a “lock” object that stores a value representing locked or unlocked states, we can use I/O requests to enforce mutual exclusion. To obtain the lock, an application first performs a read I/O request with exclusive access semantics. If this request succeeds and the mutex is unlocked, the application records $v_r$ and $v_w$, and initiates an exclusive write request to set the value to locked. If the version numbers $r'_w$ and $v'_w$ returned from acquiring this request are equal to the $v_r$ and $v_w$ recorded earlier, atomicity is guaranteed, and if they differ, the lock acquisition fails. Thus, if the write request succeeds, indicating that no other concurrent writes happened, the mutex is acquired.

Buffer management. I/O requests work with data in buffers. To initiate an I/O request, the application acquires a buffer and asks the kernel to set up request state. Subsequently, the

---

**Figure 3.3:** Two read requests with different concurrent access semantics attempt to access object $o$: $T_A$ requests exclusive access and $T_B$ requests MRSW semantics. Depending on how the requests align, $T_A$ may fail at the acquire or commit stages.
application performs its I/O on the buffer and eventually commits the request. At this point, the buffer could be de-allocated if the request succeeded. However, if the request fails, or further I/O must be done on the same object, releasing the buffer is wasteful. Ideally, the buffer should be reusable for further I/O.

Therefore, I extend the simple acquire–commit sequence with a release step: after attempting to commit, the task may either (i) initiate a new I/O request on the same buffer by repeating the acquire step; (ii) attempt to commit the same buffer again; or (iii) release the buffer back to the kernel (Figure 3.4).

In other words, an I/O request must acquire and commit at least once, but may subsequently commit any number of times. It may also start a new request using the same buffer by repeating the acquire stage; finally, it must release the buffer exactly once. I discuss buffer management in more detail in Section 3.7.2, where I also explain how buffers can be moved between objects.

Cross-machine requests. When I/O requests cross machine boundaries, they may need to use a separate, internal I/O mechanism to move the data (consider, for example, remote access to a shared memory buffer). Implementing this requires the participating WSC OS kernels to coordinate.

The approach taken in D10S is similar to the implementation of remote I/O in classic distributed operating systems (e.g. V [Che88]): a remote kernel thread acts as an I/O proxy for the originator task, reading data or applying writes received. Figure 3.5 illustrates this with an example. The task on machine $M_0$ starts a write I/O request on object $o$ by making an acquire system call. The kernel coordinates with the kernel on $M_1$ (via the D10S Coordination Protocol, §4.7), first issuing an RPC for $M_1$’s kernel to check the version and request counters for $o$. If the request can be admitted according to the desired concurrent access semantics, the kernels establish a connection via a transport object (e.g. a TCP connection) and a buffer is returned to user-space on $M_0$. The application deposits its data in the buffer or modifies its contents, and invokes the commit system call. $M_0$’s kernel now uses the transport object to send the buffer to $M_1$, where an I/O handler thread in the kernel checks the request’s validity and applies the write to $o$, synchronising with local or other remote I/O requests as required.

Related approaches. Mechanisms akin to I/O requests exist in prior distributed systems, although they typically have stronger transactional semantics. For example, the Locus distributed OS had network transparency [WPE*83], and supported nested transactions on replicated files [MMP83].
Figure 3.5: Handling write I/O request for a remote object: the task on $M_0$ first acquires (ACQ_WR) via RPC to $M_1$ and receives a buffer for local I/O; it then attempts to commit the request (COMM_WR), the data are sent via the transport object, a remote kernel thread handles the request, validates it, and applies the changes from the buffer.

The Clouds OS supported “atomic actions” with transactional semantics based on per-thread and per-object consistency levels [DLA88, §5]. The notion of threads that go through phases of different consistency semantics is similar to the I/O request notion I have described, but – unlike I/O requests – assumes a particular threading model and object-level consistency labels [CD89].

Distributed shared memory (DSM) systems often had similar semantics. For example, Munin was based on release consistency (allowing temporary inconsistency of objects) and, like I/O requests, could express multiple consistency levels [CBZ91]. My I/O request system could be further extended with flags similar to those employed by Munin.

Finally, Software Transactional Memory (STM) [ST97] enforces transactional semantics for memory accesses, typically with the goal of simplifying concurrent programming. I/O requests are more coarse-grained than STM and do not have transactional semantics, but operate over in-memory and durable objects alike.

### 3.7.2 Buffer management

When an application performs I/O, the data reside in an in-memory buffer; a kernel device driver drains or populates this buffer. Traditional OSes allow the user to choose the buffer, specifying a memory address and a length to I/O system calls. For example, the BSD socket calls take pointers to user-space buffers that are either populated with data (e.g. `recv(2)`), or hold the data to be sent (e.g. `send(2)`), and streaming file I/O (e.g. `read(2)`, `fread(3)`) likewise operates on user-provided buffers.

While it is flexible, there are several downsides to this approach:

1. Many security vulnerabilities relate to arbitrary memory buffers being passed to the operating system: for example, “buffer overrun” attacks are notoriously difficult to protect against in a system call API [DKK08, §5; SB09].
2. As buffers are identified only by pointers, their ownership is unclear: outside a managed environment (such as a language runtime), no unambiguous and binding way of passing ownership exists, which begets memory leaks and concurrency bugs.

3. OS kernels often copy the data in buffers passed to system calls, rather than using it directly, with a noticeable performance impact on data-intensive applications (no zero-copy I/O). BSD sockets are a key example of a copy-based API, and several recent endeavours improve network throughput by removing these data copies [Riz12; MWH14].

A WSC OS based on distributed objects can take a different approach. I/O buffers for applications can be allocated by the OS kernel and associated with a specific object handle that is tracked by the kernel. This approach works especially well as data-flow and object ownership are well-defined, and I/O requests offer a natural buffer management cycle.

If buffer lifetime is connected to I/O requests and their acquire-commit-release cycle, several optimisations are required to obtain acceptable performance:

1. Enabling buffer reuse to reduce the number of system calls required to perform a sequence of I/O operations that use the same buffer many times (e.g. making minor modifications to a large buffer). Repeatedly acquiring and committing an existing buffer without releasing it implements this optimisation.

2. Allowing buffer ownership transfer in order to facilitate pipelined processing. For example, to efficiently read data from one object and write them to another, it makes sense to simply transfer ownership of the buffer, rather than copying it.

3. For buffers that are tied to a particular object (e.g. as part of DMA-able memory), temporary borrowing allows ownership transfer as long as the buffer is eventually returned to the original owner.

Flags specified with I/O request calls can indicate buffer reuse (use), permanent transfer (take), and temporary borrowing (borrow) of buffers. Figure 3.6 shows the possible transitions for a buffer initially created for object $a$ with respect to itself and another object, $b$. Some transitions, for example between objects in different locations that do not share memory, may require the buffer contents to be copied by the kernel, however.

The explicit management of I/O buffers by the OS kernel has several advantages:

**Fixed buffer locations and extents** make it simple to check the buffer validity for I/O system calls.

**Clear ownership semantics** avoid concurrent use or modification of buffers: a buffer can only be used for I/O on the owning object and by the task that originally received it, and buffers share their fate with the owning object.
CHAPTER 3. DESIGNING A WSC OPERATING SYSTEM

Invalid Owned by a

Owned by a

Borrowed by b

use

release

take

release

take

Invalid

Owned by b

use

release

create

Figure 3.6: Buffer ownership state transition diagram for an I/O request on object $a$. Invalid and no-op transitions (e.g. *take* on an already owned buffer) are not shown.

**Amenability to zero-copy operation** is granted by the ability to move buffer ownership between objects without having to copy the buffer.

I do not specifically explore the zero-copy I/O facilities in *Dios*; other work based on similar APIs does (e.g. CamIO [GSM+13]). In addition, buffers in this model are atomic for simplicity: while offsets into them can be used, it is not possible to split or merge disjoint buffers without copying them. However, such functionality could be added, for example based on the APIs for sliced windows in IO-Lite [PDZ00], or chunked aggregates in Cosh [BHK+14].

**Related approaches.** Various prior I/O systems have similarities with the approach described above. For example, the Uniform I/O (UIO) [Che87] interface in the V distributed operating system [Che88] offered a unified I/O API consisting of compulsory, optional, and specialised functionality for each object. UIO supported blob-type and streaming I/O, and offered request-specific and object-specific concurrency semantics [Che87, §2.6].

Fast buffers (“fbufs”) remapped pages across protection domains on a single machine to reduce copying in pipelined network traffic processing [DP93]. The Fbufs ownership transfer model is similar to mine. Scout [MP96] extends fbufs to an explicit OS-level notion of I/O “paths”, which fuse layers together according to transformation rules in order to harness whole-system optimisation opportunities. As the WSC OS knows the communicating end-points, its kernel can likewise optimise the underlying mechanism for each I/O request.

The OS-managed buffer model in *Dios* is also similar to that in IO-Lite [PDZ00]. Like fbufs and Scout, IO-Lite facilitates zero-copy I/O where possible, but optimises buffer sharing across the OS kernel subsystems in order to maximise zero-copy opportunities.

Finally, the aforementioned Cosh [BHK+14] tracks ownership of “aggregates” (buffers) across coherence domains in heterogeneous multi-core systems, offering strong and weak “move” and “copy” semantics, and copying the data if unavoidable.
More generally, ownership tracking is an increasingly common notion: for example, the rvalue move semantics in C++11 facilitate memory ownership transfer at the language level, and the Rust language explicitly tracks memory ownership at compile time. It supports lending “boxed” memory outside the allocating scope [RustDoc15], and automatically releases boxed memory when it is no longer required.

### 3.8 Summary

This chapter investigated how a new operating system for a “warehouse-scale computer” should work, and how it can address the challenges highlighted in Chapter 2.

I stated six core principles for a WSC OS (§3.1) and showed how an OS meeting these principles differs from current systems. Specifically, I argued that a good WSC OS design,

1. revisits the concept of distributed objects in the context of modern WSCs (§3.2);
2. is based on scalable, translucent, and uniform abstractions (§3.3);
3. has a narrow system call API that focuses on I/O and synchronisation (§3.4);
4. uses capability-based protection, but has separate plain-data identifier capabilities for naming objects and context-sensitive handle capabilities for operating on them (§3.5);
5. simplifies the storage stack by exposing only a flat object store at the OS level (§3.6); and
6. is based on I/O requests with kernel-managed buffers (§3.7).

Some of these decisions are more unconventional than others, but all of them diverge from the way currently-deployed operating systems work.

To validate their practicality and utility, I created D10S, a proof-of-concept WSC OS based on this design. I explain the concrete abstractions and implementation techniques used by D10S in more detail in the following chapter.
Chapter 4

DIOS: an operating system for WSCs

“My guideline in the morass of estimating complexity is that compilers are three times as bad as normal batch application programs, and operating systems are three times as bad as compilers.”
— Fred Brooks, “The mythical man-month” [Bro75, p. 92].

DIOS is a WSC operating system built around the principles outlined in the previous chapter. This chapter describes its key abstractions, how they interact, and how they support WSC applications. I also describe a prototype implementation of DIOS in the Linux kernel.

Section 4.1 gives an overview of the topics discussed in the subsequent sections:

**Objects** (§4.2) form the core abstraction of DIOS. They abstract a blob of data, a streaming transport, a group, or an active task, they are interacted with via I/O calls, and they can be stored durably.

**Names** (§4.3) identify objects. They are freely communicable, flat pieces of binary data that act as identifier capabilities to locate object instances.

**Groups** (§4.4) represent name resolution scopes (namespaces) in DIOS and limit the visibility of names. Groups are objects themselves; they have names and can be accessed via references.

**References** (§4.5) constitute context-dependent object handle capabilities for specific object instances and expose informational meta-data to user-space, thus allowing applications to implement a variety of policies. They can be delegated only with kernel mediation, making it possible to sandbox applications and contain their outputs.

After introducing these concepts, Section 4.6 gives an overview of the initial DIOS system call API and presents examples of how user-space libraries and applications use it.
As a distributed operating system, DIOS relies on coordination across machines. In Section 4.7, I introduce the DIOS Coordination Protocol (DCP) and discuss what properties it requires of an underlying transport.

Finally, I describe the proof-of-concept implementation of a DIOS prototype (§4.9). The prototype consists of a kernel implementation of the DIOS abstractions as an extension to the Linux kernel. I also discuss how incremental migration to DIOS is enabled by the prototype’s support for combining legacy and DIOS abstractions.

DIOS can spawn tasks on any machine in the WSC, but does not come with a cluster-level scheduler itself. Instead, it relies on the Firmament scheduler, which is the subject of Chapter 5.

4.1 Abstractions and concepts

In this section, I describe the core abstractions in DIOS and the concepts they embody. Figure 4.1 illustrates them using a simple example.

DIOS has five key OS-level abstractions: objects, names, references, groups and tasks.

Object: a self-contained entity, for example a blob of data on disk, a blob of private or shared memory, a task, a streaming point-to-point connection, or another uniquely identifiable collection of data or state.

Objects can be replicated, giving rise to multiple physical object instances that make up a single logical object (although not all object types support such replication). An object instance is always managed as a unit: it is created, replicated and destroyed atomically. DIOS deliberately prescribes little about objects’ behaviour and, unlike many prior distributed OSes, assumes no language integration (cf. §3.2).

Each logical object has a permanent, unique name, is a member of one or more groups and can be referred to by many references. In Figure 4.1, physical object instances are depicted as sheets of paper, while all instances pointed to by the same name make up the logical object. In the following, I use “object” as synonymous with a particular physical object instance (as opposed to logical object).

Name: a globally unique identifier for a logical object (i.e., all instances of an object). Names are fixed-width, flat binary values and resolve to objects. DIOS requires that objects of the same name can be used in lieu of each other, i.e. they exhibit identical application semantics when used interchangeably.¹

A name is an identifier capability: it is globally valid, but its usefulness is subject to group membership (§3.5). Holding a name and having the matching group membership

¹Note that this notion of the objects being used in lieu of each other does not prescribe any particular I/O consistency semantics and need not imply bit-by-bit identity. I discuss this in detail in Section 4.2 and Appendix B.2.
conveys the authority to obtain the set of object instances that corresponds to the name. Names are also merely identifiers: they cannot serve as handle capabilities for I/O or as system call arguments (other than for name resolution). They can, however, be stored and communicated as plain data, unlike handle capabilities (references). Names are generated such that they are unforgeable by brute-force.

In Figure 4.1, a name is depicted by $N_o$ and maps to all of $r_o^0$–$r_o^2$, across tasks and machines.

**Reference:** a locally scoped, context-sensitive handle capability for a specific physical object instance. Many references may refer to the same object instance, potentially from different task contexts (see Figure 4.1). Likewise, the name corresponds to all object instances. References contain both internal information visible only to the DiOS kernel and public attributes that are exposed to user-space applications and used to implement translucency. References are how user code accesses objects in DiOS: all I/O happens via system calls that take references as arguments. References carry access control permissions, and act as capabilities: the possession of a reference enables the holder to perform certain operations on the object instance referred to. Consequently, a reference implementation must guarantee unforgeability; i.e. it must not be possible for an application to gain access to an object by means of synthesising a fake reference.

**Group:** a special type of object that is used to restrict name resolution scope. A task may only resolve names in those groups that it is a member of; group memberships are inherited from the parent task.

**Task:** a running or suspended DiOS process. Tasks run in their own address space, although they may have shared memory mappings. The DiOS design is, in principle, compatible with threads sharing an address space, but the current prototype only supports single-threaded tasks. Each task has its own, private set of references, and is a member of a fixed set of groups.
Figure 4.2: Distributed MapReduce (five map tasks, three reduce tasks): a deterministic data-flow batch computation. In this example, five input objects ("splits"), $i_0$–$i_4$, are transformed into three output objects, $o_0$–$o_2$, via intermediate objects $m_{i,o}$. The intermediate and output objects, and all tasks, are created by a MapReduce controller task (not shown).

In the following sections, I discuss these abstractions and their realisation in DIOS. Before moving on, however, I introduce some shorthand notation and two running application examples used in the following sections.

**Notation.** For brevity, the following short-hand notations are used for abstractions, mappings and data structures:

- $\Theta_o$ is the set of all instances of logical object $o$.
- $O_i$ denotes the structure describing a physical object instance $i$.
- $N_o$ denotes the name of a logical object $o$.
- $K_i$ denotes the kernel part of a reference to physical object instance $i$.
- $R_i$ denotes the user-level part of a reference to physical object instance $i$.
- $T_t$ denotes a task $t$ of type $T$.
- $\Gamma$ stands for a $k \mapsto v$ lookup table.
- $\{}$ stands for an unordered set.

In figures illustrating DIOS applications, data objects are represented by paper leaves, while task objects are shown as circles. Solid arrows indicate directional data-flow via references; dotted arrows indicate name resolution and dashed arrows indicate object creation.

**Examples.** As abstractions and APIs are sometimes best explained by example, I refer to two applications for illustration in this chapter.

The applications are (i) a distributed MapReduce [DG08] implementation – a typical determi-
Figure 4.3: Event-driven multi-process HTTP server with back-end: a user-facing service, implemented with three front-end worker tasks ($W_0$–$W_2$) that share an acceptor object (acc.) which supplies client connection objects ($cs_0$–$cs_1$), to which it serves responses composed from static data ($d_i$) and dynamic data, $bd_i$ (e.g. from a key-value store), obtained by communicating with three back-end tasks ($B_0$–$B_2$) via stream objects ($bs_0$–$bs_1$).

In the MapReduce framework (Figure 4.2), the $i^{th}$ map task ($M_i$) applies a map function of the form $\text{map}(\text{key}, \text{value}) \rightarrow \{\langle \text{key2}, \text{value2} \rangle\}$ to all records in the input object ($i$). All map tasks run in parallel. The items in each resulting list are subsequently hash-partitioned on $\text{key2}$ and the resulting intermediate sets of key-value pairs ($m_{i,j}$) are provided as input to the parallel reduce tasks ($R_j$ taking $m_{k,j}$ for all $k$). These apply a reduce function, $\text{reduce}(\text{key2}, \{\text{values}\}) \rightarrow \{\text{out-values}\}$, storing the final values in an output object $o_j$. A controller task manages the other tasks and monitors them for fault tolerance, re-executing any failed tasks.

The HTTP server (Figure 4.3), by contrast, is a non-deterministic, event-driven application. The design is similar to the event-driven “reactor pattern” in the widely-used nginx web server: multiple worker processes ($W_i$) all poll a TCP acceptor object (acc) to accept client connections. The client connections are materialised as TCP stream objects ($cs_j$), which the handling worker process performs I/O on. In the back-end, the worker processes may interface with either static content in the form of blob objects (in memory or on disk) or dynamic content from a key-value store via references to stream objects ($bs_i$) that communicate with the back-end tasks ($B_i$) over shared memory or the network. Fault tolerance can be implemented by the scheduler task restarting any failed worker tasks.
CHAPTER 4. DIOS: AN OPERATING SYSTEM FOR WSCS

Figure 4.4: A DIOS object and its relations to names and references.

4.2 Objects

DIOS manages distributed objects over many nodes (see §3.2). This entails creating and managing the physical objects, but also making them and their meta-data available across many shared-nothing WSC machines. In this section, I describe how objects are created, managed, and destroyed, and how DIOS represents them to user-space applications. My initial discussion focuses primarily on high-level concepts and omits the details of cross-machine coordination; Section 4.7 describes the communication protocol employed to implement operations across machines.

An object is represented by the structure shown in Figure 4.4. Each object has a name, a type, and type-specific data. The data may refer directly to a storage location, or to a type-specific state structure, such as a process control block (PCB) for a task, or the connection state for a network connection.

DIOS allocates an object structure when the object is first created, and it persists in memory until the object is deleted or the holding node crashes. For durable objects, the structure is also serialised to the object store alongside the object data, so that it can later be restored.

Granularity. DIOS data objects can hold any amount of data, but the per-object overheads of around 100 bytes make them inefficient at very small object sizes, such as a single integer or a short string. Hence, data-holding objects typically correspond to aggregates such as a table row, a user record, or the whole buffer of a request response of several kilobytes. In the web server example, each a key-value record in the back-end store might represent, for example, a user profile.

Types. DIOS supports four general object categories: blobs, streams, tasks, and groups. Blobs support random access, while streams offer ordered access to data, but do not support rewinding.

The specific set of types supported in the current DIOS prototype is listed in Table 4.1. This is by no means an exhaustive list: other object types are likely necessary to support specific functionality atop DIOS. A durable on-disk log, for example, would be a key addition required for some applications (e.g. a distributed file system).
CHAPTER 4. DIOS: AN OPERATING SYSTEM FOR WSCS

<table>
<thead>
<tr>
<th>Type name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DURABLE_BLOB</td>
<td>Memory-mapped file backed by durable storage.</td>
</tr>
<tr>
<td>PRIVMEM_BLOB</td>
<td>Private memory allocation.</td>
</tr>
<tr>
<td>SHMEM_BLOB</td>
<td>Shared memory allocation.</td>
</tr>
<tr>
<td>SHMEM_STREAM</td>
<td>Shared memory pipe/ring with FIFO semantics.</td>
</tr>
<tr>
<td>UDP_STREAM</td>
<td>Stream of UDP datagrams.</td>
</tr>
<tr>
<td>TCP_STREAM</td>
<td>TCP 1-to-1 connection.</td>
</tr>
<tr>
<td>TCP_ACCEPT_STREAM</td>
<td>TCP acceptor, a stream of connection references.</td>
</tr>
<tr>
<td>CONSOLE</td>
<td>Local system console for debugging.</td>
</tr>
<tr>
<td>GROUP</td>
<td>DIOS name resolution group (scope).</td>
</tr>
<tr>
<td>TASK</td>
<td>Active task (suspended or running).</td>
</tr>
<tr>
<td>TIMER</td>
<td>Current system time.</td>
</tr>
</tbody>
</table>

Table 4.1: Object types supported in the DIOS prototype.

Life cycle. Every physical object instance also has a life cycle type, which governs when it is considered ready for destruction. There are three such types:

- **Immortal** objects are never destroyed, even if they do not have any references pointing to them and no live node is able to resolve their name. An example of this is a task’s self-object, or a special console object.

- **One-off deletion** amounts to destruction as soon as any reference to the object with delete permission has the delete operation invoked on it. The actual deletion may occur asynchronously and may not be immediately effective, but once it has been issued, it cannot be stopped. A durable on-disk object with one-off deletion might have a deletion-privileged reference to it held by the distributed file system.

- **Reference-counted** objects maintain a reference count, incremented on reference creation in response to name resolution or reference delegation, and decremented when a reference is deleted. Once the reference count reaches zero, the object is deleted. Most DIOS objects are reference-counted, and the most common cause of object deletion is the exit of the final task holding a reference to an object.

In both one-off deletion and reference-counted deletion, the object is first “zombified” – that is, its I/O operations are atomically disabled, so that concurrent access fails safely – and then destroyed. Note that, however, other instances of the same logical object may continue to exist, since they have their own object meta-data (and possibly even a different life cycle type).

Persistence. Objects may be persistent, meaning that they are stored on durable storage and survive DIOS restarts. For such objects, the object structure is reinstated from serialised storage meta-data and runtime information as part of the DIOS bootup process. For example, an on-disk blob stored on consecutive raw blocks requires its name, owner, type, the location of the data, and the object’s length to be preserved in addition to the raw data.
When a machine running DIOS starts up, it synthesises some default objects and initialises object structures for those objects stored on local persistent storage.\footnote{In the DIOS prototype, these objects are files in a well-known directory, with object structures being created for them at bootup; a custom block storage system with a durable index would avoid the host file system.}

To make persistent objects available to tasks, references must be created for them. However, references are ephemeral and only persist as long as the task that owns them: after bootup, this creates a chicken-and-egg problem. This problem is solved by tasks’ ability to resolve \textit{names}, which are identifier capabilities that can be stored as data. In practice, this enables bootstrapping in one of two ways:

1. Another machine in the WSC may create a task on the machine that has the appropriate information (a name and a group membership) to acquire a reference to a persistent (but currently unreferenced) object via name resolution (see §4.3.3).

2. A special, privileged task (akin to \textit{init}) may be given a list of names for all persistent objects on the machine and a set of bootstrap references to them. It may distribute the names as data, and the references via delegation. In practice, the cluster manager or scheduler (§2.2.1.1) is well-suited to the role of this special task, since in DIOS it also has the power to create tasks with arbitrary names and group memberships (as I will discuss in §4.4).

The next section explains DIOS names, which identify logical objects in more detail.

### 4.3 Names

DIOS assigns every logical object a globally unique identifier, its \textit{name} ($N_o$ for a logical object $o$). Multiple instances (replicas) of the logical object may exist, but must maintain the invariant that they can be used \textit{in lieu} of each other, i.e. they are interchangeable without breaking applications’ expected semantics. If this is not the case, the replicas are different objects and must have different names.\footnote{DIOS does not automatically enforce this. Applications must instead themselves ensure that object replicas remain interchangeable. This allows for considerable flexibility, as applications are free to choose their own semantics for interchangeable use.}

For example, a strongly consistent distributed key-value store may require replication of writes for fault tolerance. Its tasks hold multiple references to replicated instances of each logical object representing a value stored. To fit DIOS object semantics, the key-value store must maintain the instances’ interchangeability after serving a write: it must either ensure that all instances are updated atomically (via application-level locking or transactions), or that the updated value is stored in a new logical object (with a new name) whose instances supersede the old ones.
Format. DIOS names are flat 256-bit binary values, similar to Universally Unique Identifiers (UUIDs), first introduced in the Apollo Network Computing System [DLM*88, §2.4]. This ensures that names are both globally meaningful, and that they can be stored and passed as data within the WSC (or even outside it, as with URL-based capabilities; cf. §3.5).

Names must be unforgeable, since they are resolution capabilities, permitting – in combination with an appropriate group membership – translation of names into sets of references. Unforgeability is ensured by the statistical properties of their cryptographic generation.

In addition to the cryptographic property of names, which protects against brute-forcing, name resolution power is also confined by the resolution scopes imposed by groups.

Scoping. Without resolution scopes, leaking a name would have disastrous consequences: any task that obtains access to the name could gain access to the corresponding objects.

To implement scoping, DIOS must either maintain a centralised notion of scopes or enforce them distributedly by combining the user-visible name with another piece of information. A centralised, fine-grained access control scheme would contradict Principle 1 (§3.1), and DIOS hence uses distributedly maintained groups to scope names.

For this purpose, DIOS administers two types of names for each logical object:

1. A single external name, which is the user-visible name of the object, denoted by $N_o$. It is returned to user-space when an object is created, can be stored and communicated, and it is provided as an argument to name resolution requests.

2. A set of internal names, one for each group in which the logical object exists, each denoted by $N_{g,o}$ for group $g$. Each internal name maps to the same object instances. Internal names are never made available to user-space applications, but are useless even if they were leaked by accident.

Each object is created in a set of groups and its internal names combine its external, user-visible name with the names of its groups. On resolution, the external name is combined with one or more groups to generate internal names used for for resolution. This grants access to the object only to tasks which can furnish an appropriate group membership, since only the internal names generated in combination with the correct groups yield name resolution matches.

The set of groups for an object is fixed during its lifetime, i.e. it is not possible to revoke group membership. The only way to remove an object from a group is to create a new object outside the group, copy the data and delete the object. I describe groups and their implementation in more detail in Section 4.4.

In the following, I describe how names are generated (§4.3.1), how they are stored and managed in the distributed machines’ kernels (§4.3.2), and how they are resolved into references (§4.3.3).
4.3.1 Generation

DIOS supports several ways of generating names, each with different properties. *Deterministic* name generation guarantees that an identical execution of tasks always yields the same sequence of names; *random* name generation generates uniquely random, one-off names, and *special* names are used for OS handles to well-known resources.

Other name generation schemes are conceivable: for example, the 256-bit string could be composed of multiple concatenated hashes or include a locality element.

While an object’s external name can be created in different ways, internal names are *always* generated deterministically. Each internal name is generated by computing a hash of the external name and the name of a group $g$:

$$\text{internal name}_g = \mathcal{H}(\text{external name} \parallel \text{name}_g).$$

This way of generating an internal name ensures that even if an internal name is ever leaked to user-space, it cannot be used to resolve objects.

**Deterministic names.** Many WSC applications, and especially those used for parallel data processing, express deterministic computations. They leverage the determinism to, for example, simplify the programming model and support replay-based fault tolerance (e.g. in MapReduce, CIEL [MSS+11] and Spark [ZCD+12]). Determinism is also useful to schedule computations with data dependencies and enables memoisation optimisations (as in CIEL [Mur11, §4.2.1] and Tachyon [LGZ+14, §3.2]). DIOS supports these needs via deterministic name generation.

To generate names deterministically, DIOS applies a one-way hash function to the name of the generating task object and a deterministically generated identifier. The identifier must come from a chain of identifiers which is deterministic in the generating task’s name. A simple example of such an identifier is a monotonic counter of objects created or tasks spawned (similar to the name generation scheme in CIEL [MSS+11, §5.1]).

To ensure unforgeability, the hash function used must have a sufficiently wide co-domain and a uniform value distribution. If collisions are extremely improbable, brute-forcing of names without knowledge of a previous name in the chain is impossible. The DIOS prototype uses the SHA-256 hash function.

**Random names.** If objects are generated non-deterministically, or deterministic naming is not required, DIOS generates names randomly. The name is a random pattern of 256 bits, sourced from a kernel entropy source or a hardware random number generator.

---

4 One exception exists: the cluster manager can start tasks of arbitrary names to seed the name chain.

5 The application may, of course, use the same hash function as DIOS to generate deterministic names in user-space. This may be acceptable; as an alternative, the kernel could use salts. This would, however, require persistent storage of the salt for later deterministic re-generation a the name.
CHAPTER 4. DIOS: AN OPERATING SYSTEM FOR WSCS

```c
int create_shmem_fifo(dios_name_t* name, uint64_t host_id) {
    dios_ref_t* ref = NULL;
    int result = -1;
    dios_flags_t flags = D_CREATE_NAME_DETERMINISTIC;

    /* non-zero host ID indicates a remote object creation */
    if (host_id != 0)
        flags |= D_CREATE_REMOTE;

    result = dios_create(flags, D_OBJ_SHMEM_STREAM, NULL, name, &ref, host_id);
    if (result < 0)
        perror("failed to create SHMEM_STREAM object");
    return 0;
}
```

Listing 4.1: Creating a shared memory stream object in the MapReduce example: a deterministic name is stored in the user-space allocated memory pointed to by `name`.

As with deterministic names, code have a dependency on a random name, but must, of course, obtain the name in the first place. Consequently, a dependency on a randomly-named object only makes sense if the object (and its name) already exists.

**Special names.** DIOS tasks have access to some default resources. For example, tasks may need access to a console or a timer, which are represented by special objects synthesised by the DIOS kernel. Such objects have well-known names composed of a 16-bit identifier in the least significant bits of an otherwise all-zero name.

This type of name is “special” because, unlike other DIOS name types, the names are well-known (i.e. forgeable) and refer to specific objects. Notably, the object instances mapped by a special name depend on the context in which it is used: resolving the console name does not return all consoles in the entire WSC.

**Example.** Listing 4.1 shows an example of how a deterministic name is created as part of an object creation in the MapReduce application. Note that the memory holding the name is allocated by user-space, and may be on the stack or within a DIOS memory object’s data.

### 4.3.2 Storage

The DIOS kernel must keep track of names and the live objects they identify. It does this by storing mappings between names and object instances in a *name table*. Each node in the WSC has a system-wide kernel name table.

The name table maps names to object structures, rather than directly to references. Since references are context-dependent structures, they are generated dynamically in response to resolution.
requests, and take the identity and location of the resolving task into account.

Hence, a name table, denoted by $\Gamma_N$, maps an internal name $N^g_o$ for object $o$ in group $g$ to a set of object structures for a set of known local instances of $o$:

$$N^g_o \mapsto \{O_r \mid r \in \Theta_o\}$$

That is, the internal name $N^g_o$ identifies a set of object structures that represent local instances of object $o$.

Additional instances may exist in other machines’ name tables. By default, the local name table is consulted first when DIOS resolves a name; the resolution may afterwards proceed to consult other machines’ name tables.

An alternative design would maintain a per-task name table, as opposed to a shared per-machine one. This might increase scalability, but requires either (i) that shared names are proactively inserted into all per-task name tables, or (ii) that name resolutions query all name tables. Depending on the degree of sharing, this may negate the scalability benefits.

The name table on each machine is updated on object creation and deletion. Access to remote name tables happens via the DIOS Coordination Protocol (see §4.7).\(^6\)

**Bootstrap.** A machine’s name table must be bootstrapped at startup. On bootup, the kernel populates the name table with well-known special names and their corresponding objects, as well as the names for all objects stored on durable storage on the node. Other name mappings are added incrementally at system runtime.

Applications boot-strap their access to objects either by using a name which is hard-coded into the application code (akin to a hard-coded file name in legacy applications), by receiving a name as data from another task, or by reading it from a persistent object’s data.

A name in itself is not useful for I/O, however: an application must first resolve it to a set of handle capabilities (references) before it may interact with the named object (§3.5). In the next section, I describe how this resolution proceeds.

### 4.3.3 Resolution

While a DIOS name describes a unique logical object, this object may have multiple physical instances\(^7\) within the WSC. A name resolution request returns references for a set of existing instances up to a maximum number specified.

---

\(^6\)It is possible to cache entries for remote objects in local name tables to improve resolution performance, but this requires the name tables to be eagerly or lazily updated in response to remote changes.

\(^7\)As before, these instances must be *interchangeable* at application-level. This can, but does not have to, imply bit-by-bit data identity.
Figure 4.5: To resolve external name $N_o$ for $o$, it is hashed with the task’s groups $a$, $d$, and $f$ to generate internal names $N_o^a$, $N_o^d$, and $N_o^f$; the object is located in the name table under its internal name $N_o^d$, an appropriate reference is generated and returned to user-space. Note that all of the task’s groups are queried here, rather than a specific one.

Therefore, the process of resolving a DIOS name amounts to evaluating a mapping function $r$:

$$r(N_o,k) \mapsto \emptyset \| \{R_0, \ldots, R_k\}.$$

In other words, an external name either resolves to an empty set or a set of $k$ newly generated references corresponding to $k$ available physical object instances. Although DIOS endeavours to return references for all existing replicas, this is not guaranteed: only references for reachable replicas are returned.

The notion of a reachable object is subject to three conditions:

1. The resolving task, $T$, must be permitted to resolve names;
2. an object named by one of the internal names generated from combining external name $N_o$ with one of $T$’s groups must exist in a name table on some machine;
3. for a remote object, the machine that holds it must be alive (i.e. able to generate a response), and neither the DCP message to this machine, nor the response may be lost.

When asked to resolve an external name $N_o$, the DIOS kernel proceeds as follows (Figure 4.5):

1. It generates the corresponding internal name, $N_o^g$, for the specified group $g$, or for each group that the calling task is a member of, by computing a hash of $N_o$ and $g$ (§4.3.1).
2. It uses each internal name $N_o^g$ to index into the local name table, returning any matching local object instances.
3. If no match is found, or if insufficiently many entries are found, further object instances are sought on remote machines. If so,
   (a) a lookup message containing the internal names of interest is broadcast to all nodes (via the DCP, §4.7);

---

8Flags allow the caller to choose local/distributed resolution only, if desired.
Listing 4.2: Name resolution example: looking up at most `MAX_REFS` references to replicas of the first input and checking how many were found.

(b) any machine that holds instances of the logical object named by one of the $N^G$ sends a unicast response to the enquiring machine kernel.

4. For any matching objects found, the references are generated and returned to user-space. If no match is found, an empty reference set is returned.

This procedure does not specify any priority ordering between instances except for the default priority for local object instances. Instead, responses are accumulated first-come, first-served until either sufficiently many instances have been found or the request times out. If priority ordering is desired, the DIOS kernel can contact machines in-order with unicast requests.

Listing 4.2 shows an example of the user’s view of name resolution: the task retrieves a name from its input description, asks the kernel to resolve up to `MAX_REFS` replicas of it, and checks the number of references returned.

References returned from a name resolution are freshly created, offer full access permissions (read, write, execute, delegate, and delete) and specify the “widest” value for all restrictable reference properties (see §4.5.1). Consequently, any task that successfully resolves a name has full access to all corresponding object instances that exist in its groups.

A task can create a more restricted version of a reference by delegating it with appropriate restrictions either to itself or to another task (see §4.5.2). However, references to group objects are an exception: they are always returned read-only from name resolution and cannot be delegated. Hence, a task can also have access an object created in a group that it is not a member of if a reference is explicitly delegated to it.

Reference delegation is preferable over name resolution as it supports the principle of least privilege: Name resolution should only be performed by logical “owners” of objects.\(^9\)

Next, I discuss groups, which delineate name resolution scopes in DIOS.

---

9The ownership is “logical” because the owner does not have to be the specific task that created the object in the first place; it must only have a shared group membership with the creator.
Table 4.2: Relations between a task and group $g$ and the corresponding capabilities a task has with regard to the group $g$.  († with exception of the cluster manager)

<table>
<thead>
<tr>
<th>Task relationship to $g$</th>
<th>Resolve names in $g$</th>
<th>Create objects in $g$</th>
<th>Grant membership of $g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Member</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Created $g$</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Resolved $g$’s name</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

4.4 Groups

Name resolution scopes in DIOS are defined by groups. Groups are themselves DIOS objects and accordingly have unique names. However, they contain no data other than a copy of their name (for introspection), and cannot be written to.

Each group object is itself created within a group. The name of a group (represented by a group object) can hence only be resolved by tasks which are members of the group in which it was originally created. At the root of this hierarchy, a special “null” group exists, which is only accessible to the cluster manager for bootstrapping.

Each task is a member of at least one group. This group is the task’s primary group: the group in which objects are created by default. The task may also be a member of zero or more secondary groups, in which it can also create objects.

Table 4.2 shows the actions a task can take on a group depending on its relationship with it.

First, a task can resolve names to object instances in any of the groups that it is a member of. A task’s group memberships are tracked in the kernel and cannot be modified by user-space code. Whenever the task requests a name resolution, it specifies a group to use (or falls back to the default, primary group). However, a task cannot resolve names in groups it is not a member of.

Second, a task can create objects both in a group it is a member of, but also in a group of which it is not a member, as long as it has access to a reference to the group object for this group. A task obtains access to such a group object reference either by (i) having created the group object itself, or (ii) by having resolved its (group) name.

Finally, a task can grant membership of both its own groups and those groups that it created to any child task. It cannot, however, grant membership of a group to whose object it merely has access via a reference obtained by resolving its name. This restriction is necessary to defend against an “access-via-child” vulnerability. Consider the following example:

1. Two tasks, $T_A$ and $T_B$, are both members of group $g$.
2. $T_A$ creates a new group, $a$, in $g$ and hands its name to $T_B$. The latter can now resolve the name into a reference to the group object for $a$.

10The names are DIOS names, rather than human-readable group identifiers, in order to ensure uniqueness and avoid name clashes in the absence of centralised name management.

11There is an exception for the cluster manager, as it must be able to bootstrap tasks that access data in the WSC with appropriate group memberships.
3. $T_B$ is not a member of $a$ and hence cannot resolve names in $a$.

4. Assume that $T_B$ was to be allowed to grant membership of $a$ to a child task $T_C$ that it creates.

5. $T_C$ can now resolve a name in $a$ that was handed to it by $T_B$ (since $T_C$ is a member of $a$) and subsequently delegate a resulting reference back to $T_B$.

This effectively grants $T_B$ the same power as $T_C$. Hence, $T_B$’s abilities are equivalent to those of a member of $a$ via collusion with its own child, $T_C$. Restricting child task membership to the parent task’s group memberships and those groups that it created solves this problem.

It is worth noting that a single task can create recursively “nested” groups, since creation of an object (which may itself be a group object) in a group does not require membership (so it is e.g. possible to create $f$ in $g$ without being a member of $g$). This allows a task to create groups on behalf of another task without subsequently having access to their objects.

Importantly, references to group objects – independent of whether they were obtained via membership, creation or name resolution – cannot be delegated to other tasks. Supporting delegation of group object references would allow a task to grant access to its groups to any task that it can communicate with, no matter whether this task has any credentials to access the group. By contrast, when a group name is passed, the destination task must be a member of the named group’s parent group in order to access the group object (via name resolution).

In summary, group objects are used in two situations:

1. When a task creates a new object, it can pass a reference to a group object, indicating the group for the new object. The DIOS kernel combines the new object’s external name with the group object’s name to generate its internal name (see §4.3.1). The group object reference passed must correspond to a group of which the creating task is a member, a group which it created, or a group whose name it successfully resolved.

2. When a task spawns a new task, a set of references to group objects specifying group memberships for the child task can be passed. Only group objects representing groups that the parent task is a member of, or which it directly created, may be passed.

Each task receives default references to the group objects representing the groups that it is a member of. When new task is created without any inherited group memberships, DIOS assigns it a new, otherwise empty group. In this situation, the task is limited to resolving names in its primary (and only) group.

Groups limit the scope of name resolution, but DIOS tasks have another way of restricting access to objects, implemented via references, which are the DIOS implementation of handle capabilities (cf. §3.5). In the next section, I describe references in detail.
4.5 References

While names are globally meaningful identifiers for objects, references constitute handles that are only valid in a particular task context, but which allow interaction with the object they describe. Unlike names, they cannot be stored as data or communicated directly.

A DIOS reference is a segregated capability that consists of two parts: (i) privileged information only available to the DIOS kernel, and (ii), public information that is exposed to the task’s user-space. This division is required to track the necessary kernel state while also maintaining translucency (§3.3). Figure 4.6 illustrates this structure with reference to an example.

The privileged in-kernel information includes the logical object name, as well as pointers to the public information, to the owning task object, to lists of outstanding read and write I/O requests (§3.7.1), and a pointer to the object structure. This part of the reference is stored in kernel memory (bottom half of Figure 4.6).

In addition, the reference also contains public, translucent information, which consists of read-only meta-data about the object. This information includes properties that a user-space application might consider when choosing between references to alternative instances of the same logical object (e.g. proximity and persistence), or when checking the consistency of an I/O request that was applied to the object instance referenced (e.g. the version number). This public part of a reference is stored in virtual memory that is mapped read-only into the task’s virtual address space. This permits the kernel to revoke the reference by unmapping or otherwise invalidating it at any time.
Reference acquisition. A task may acquire a reference in one of several ways.

First, each task is provided with an initial set of references at startup, including default references (a self-reference to the task object and a console reference), as well as pre-specified inputs and outputs. This is useful to execute tasks whose reference set is fixed at execution time.

Second, a task may resolve a name to one or more references. This allows dynamically acquired data to be converted into references to objects. I covered name resolution in Section 4.3.3.

Third, the task may receive a reference from another task by virtue of delegation. This is useful in order to dynamically grant selective access to objects to a task. I describe reference delegation in detail in Section 4.5.2.

It is worth noting that each of these options creates a fresh reference, even if the task already holds a reference to the same object. User-space code can, however, inspect the object ID (a random 64-bit integer) exposed as part of the public reference attributes to detect if multiple references refer to the same object.

Related concepts. The closest functional analogy for references in POSIX terms are file descriptors (FDs). Both FDs and references are handles for kernel state, but while FDs merely serve as identifiers, references carry additional information that makes objects translucent to user-space programs.

Mach’s ports also have similarities with DIOS references: they are location-transparent handles owned by a particular Mach “task” (shared state for a set of threads). However, unlike DIOS references, they are fully transparent and do not expose additional information about the remote end. Moreover, they only support streaming message-passing semantics, and require the far end to explicitly handle incoming messages on a port.

4.5.1 Attributes

The public part of a reference is made available by mapping the public reference structure ($R_o$ for object $o$) into the user-space virtual address space (Figure 4.6) and returning a pointer to this structure from the reference-creating system call. Thus, the public attributes are visible to the user-space application holding the reference, but cannot be modified by it.

A reference’s public attributes are the key DIOS feature that enables applications to implement their own policies in the distributed system while still using DIOS abstractions.

Consider an example: the MapReduce application’s map tasks may, depending on circumstances, be scheduled on machines where their input data are already in memory, on disk, or not available at all. The map tasks prefer to work on local data rather than a remote copy. If the data are already in memory (e.g. because another job with the same input is running), the application prefers to re-use the in-memory copies.
Listing 4.3: Example of using DIOS reference attributes to choose the best input for a MapReduce map task: in-memory objects are always preferred over on-disk objects, no matter if local or remote; and more proximate objects are preferred.

Listing 4.3 shows an example implementation of a pick_input_ref() function for the MapReduce application: in this case, the application policy is to choose in-memory inputs over on-disk inputs regardless of their location, but it also prefers more proximate inputs when they are otherwise equal.

In practice, the choice between different references is likely to be made in libraries and infrastructure systems, rather than application code. The DIOS standard library (dlibc), for example, offers several default reference pickers, which can choose the most proximate reference, or the one with the least overlapping fault domains.

Some reference attributes are inherent features of the underlying object (e.g. its type), while others may be changed by user code. One example of such a mutable attribute is the permission mask: a task may, for example, want to remove write permissions from a reference.

Since a reference’s public attributes are read-only, the task must delegate the reference to create a new reference with changed attributes – either by delegating it back to itself, or by delegating it to another task. I describe reference delegation in the next section.

4.5.2 Delegation

DIOS tasks can share an object with each other even if they are not in the same groups as the object. This is necessary in order to grant selective access to objects: not being a member of the object’s group makes it impossible for a task to obtain a fully privileged reference to an object by resolving its name.

To enable such sharing, references must be able to traverse task boundaries. This process, which involves the DIOS kernel, is called delegation. A delegation of $R_o$ from $T_A$ to $T_B$ involves the
### On delegation...

<table>
<thead>
<tr>
<th>Attribute</th>
<th>automatically adapted</th>
<th>mutable by user</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Group</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Read version ((v_r))</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Write version ((v_w))</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Active readers ((a_r))</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Active writers ((a_w))</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>R/W buffer size</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Permissions</td>
<td>✓</td>
<td>✓†</td>
</tr>
<tr>
<td>Concurrent access mode</td>
<td>x</td>
<td>✓†</td>
</tr>
<tr>
<td>I/O multiplexing mode</td>
<td>x</td>
<td>✓†</td>
</tr>
<tr>
<td>Persistence</td>
<td>x</td>
<td>✓†</td>
</tr>
<tr>
<td>Proximity</td>
<td>✓</td>
<td>✓†</td>
</tr>
</tbody>
</table>

Table 4.3: Examples of DIOS reference attributes and their fungibility properties. Properties with a ✓† can be changed by the user on delegation, but this results in a new object instance, copying the object’s data.

DIOS kernel generating a new reference that refers to the same object as \(R_o\). The public and private parts of the new, delegated reference, \(R_d\) and \(K_d\), are identical to \(R_o\) and \(K_o\), except for transformation specific to \(T_B\)’s context.

The transformation is important: since references are handles that carry context-dependent information, a delegated reference cannot merely be communicated to the new task and inserted into its reference table. Instead, the delegated reference must be adapted appropriately when exchanged between tasks.

As part of this transformation, context-dependent reference attributes are changed to reflect the view of its target object from the perspective of the receiving task. For example, the “proximity” attribute is adapted to “remote” if \(T_B\) is on a different machine to the object \(o\).

Some attributes can also be explicitly transformed by the user when delegating a reference: restricting permissions is a key example. The user can pass appropriate flags to the reference delegation system call to change these attributes. In some cases, changing an attribute may have the side-effect of creating a new object instance and copying the object’s data (if applicable). For example, if the “persistence” attribute on a reference to an ephemeral in-memory object is changed to “durable” on delegation, a durable copy of the object is made.

Table 4.3 shows a selection of attributes that the public part of a reference exposes, and indicates whether they are static, automatically adapted or can be changed by the user on delegation. Further extensions of this set of attributes are possible, and likely required; this set merely serves to illustrate the concept of “translucent” objects.

**Notifying the recipient task.** When a task delegates a reference, the kernel inserts it into the receiving task’s reference table, but the task itself is initially unaware of the new reference it has
received. In fact, since user-space applications identify a reference to object \( o \) by its public part, \( \mathcal{R}_o \), user-space code in the recipient task cannot perform any system calls on the new reference until it has been informed about its public part’s location in memory.

This information can be conveyed in two ways:

1. A notification mechanism akin to conventional Unix-style signals can be used: a task registers a callback to be invoked when a reference is delegated to it, with the new reference as its argument.

2. The task can alternatively obtain the references by accessing a special default reference which is always available, such as the tasks “self-reference”. This reference can act as a stream of reference pointers, allowing the task’s main loop to poll for delegated references.

While the DIOS prototype assumes the second option, a signal-style notification system provides a more elegant solution, as it avoids the task logic having to poll the task’s self-reference and handle I/O on it specially.

### 4.6 System call API

The system call API defines how user programs interact with the OS kernel, which privileged operations they can invoke and how information is shared between the kernel and user-space. The DIOS system call API is centred around the name, group and reference abstractions described in Sections 4.3–4.5.

Table 4.4 shows the current DIOS system call API, consisting of thirteen system calls that cover I/O, object, and task management. These thirteen calls are sufficient to implement key data-intensive WSC applications efficiently and with strong compartmentalisation guarantees (as I show in §6.2.3).

This API is similar to subsets of the ABIs in Drawbridge [PBH+11, §4], Bascule [BLF+13, §3.1], and Embassies [HPD13, Fig. 2]. However, it is not necessarily complete or universal: further extensions may be required for specific use cases. Alternatively, DIOS system calls can be combined with legacy Linux system calls in the same program, as I discuss in Section 4.9.3.

In the following, I discuss general properties and semantics of the DIOS system calls.

**Blocking and synchrony.** All DIOS system calls are by default blocking and I/O is synchronous. Handling a DIOS system call can require communication across the WSC interconnect, and can take a long time compared to a local system call (on order of hundreds of microseconds, as opposed to tens). However, blocking the calling task while waiting for a remote operation allows other tasks to run in the meantime.
<table>
<thead>
<tr>
<th>System call</th>
<th>§</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(N_o, R_o)$ <strong>create</strong> $(R_g, P_{args}, H_{id})$</td>
<td>B.1.1</td>
<td>Creates a new object $o$ in group $g$ (optional, primary if unspecified) with type-specific arguments in $P_{args}$ on host $H_{id}$ (optional; default local); and returns a name $N_o$ for it and a reference $R_o$ to it.</td>
</tr>
<tr>
<td>${R^0_o, \ldots, R^k_o}$ <strong>lookup</strong> $(N_o, R_g)$</td>
<td>B.1.2</td>
<td>Attempts to find all reachable instances of the object named by $N_o$ in group $g$ (optional, all task groups if unspecified).</td>
</tr>
<tr>
<td>bool <strong>copy</strong> $({R^0_o, \ldots, R^k_o}, R_T)$</td>
<td>B.1.3</td>
<td>Delegates references $R_i$ by copying them into the reference table of task $t$ referenced by $R_T$ and transforming as requested; returns success indication.</td>
</tr>
<tr>
<td>bool <strong>delete</strong> $(R_o)$</td>
<td>B.1.4</td>
<td>Removes reference $R_o$ and invokes deletion handler for object life cycle type; returns indication of success.</td>
</tr>
<tr>
<td>$R_i$ <strong>run</strong> $(R_{bin}, S_G, R_{pg}, P_{info})$</td>
<td>B.1.5</td>
<td>Runs object referenced by $R_{bin}$ as member of groups in $S_G$ with primary group $pg$; returns reference $R_i$ to running task. $P_{info}$ is a task-describing structure containing input and output references, arguments and an optional host $H_{id}$.</td>
</tr>
<tr>
<td>bool <strong>pause</strong> $(R_i)$</td>
<td>B.1.6</td>
<td>Pauses task $T$ referenced by $R_i$; returns true if now paused.</td>
</tr>
<tr>
<td>bool <strong>resume</strong> $(R_i)$</td>
<td>B.1.7</td>
<td>Resumes paused task $T$ referenced by $R_i$; returns true if successful.</td>
</tr>
<tr>
<td>$(P, size)$ <strong>acquire_read</strong> $(R_o, size)$</td>
<td>B.1.8</td>
<td>Informs kernel of read of size (optional) on object referenced by $R_o$; returns read buffer pointer $P$ and size of data read.</td>
</tr>
<tr>
<td>bool <strong>commit_read</strong> $(R_o, (P, size))$</td>
<td>B.1.9</td>
<td>Checks validity of completed read of size on $R_o$; returns true if valid.</td>
</tr>
<tr>
<td>bool <strong>release_read</strong> $(R_o, (P, size))$</td>
<td>B.1.10</td>
<td>Gives up read buffer for $R_o$ and returns it to the kernel.</td>
</tr>
<tr>
<td>$(P, size)$ <strong>acquire_write</strong> $(R_o, size)$</td>
<td>B.1.11</td>
<td>Informs kernel of write of size on $R_o$; returns pointer to write buffer $P$.</td>
</tr>
<tr>
<td>bool <strong>commit_write</strong> $(R_o, (P, size))$</td>
<td>B.1.12</td>
<td>Checks validity of completed write of size on $R_o$; returns true if valid.</td>
</tr>
<tr>
<td>bool <strong>release_write</strong> $(R_o, (P, size))$</td>
<td>B.1.13</td>
<td>Gives up write buffer for $R_o$ and returns it to the kernel.</td>
</tr>
<tr>
<td>$R^i_o$ <strong>select</strong> $({R^0_o, \ldots, R^k_o}, mode)$</td>
<td>B.1.14</td>
<td>Returns the first reference $R_i$ out of the $k$ references in $(R^0_o, \ldots, R^k_o)$ to become ready for I/O in mode (read/write).</td>
</tr>
</tbody>
</table>

Table 4.4: DIOS system call API. All syscalls are blocking and take a set of call-specific flags, $F$, in addition to the arguments listed.
CHAPTER 4. DIOS: AN OPERATING SYSTEM FOR WSCS

[...]

/* Pull data from the mappers */
dios_ref_t* selected_ref = NULL;
int num_inputs_done = 0, ret = 0;

do {
    ret = dios_select(D_NONE, boot_info.input_refs, boot_info.input_count,
                      &selected_ref);
    if (ret || !selected_ref)
        return ret;

    /* Get input data from ref returned by select(2) */
    ret = reduce_read_input(selected_ref);
    if (ret == -EEOF)
        num_inputs_done++;
    else if (ret != -EAGAIN)
        return ret;
} while (num_inputs_done < cfg_.num_mappers_);

Listing 4.4: Input multiplexing logic in the MapReduce example: select(2) blocks and returns the reference to the first mapper output stream object that has data available.

Non-blocking variants of some system calls could be added to cater to applications that cannot tolerate blocking. In addition, asynchronous abstractions can be implemented in user-space programs using the select(2) system call to multiplex I/O over multiple references, as illustrated in Listing 4.4.

Memory allocation. POSIX-like system call APIs typically expect user-space to allocate all memory backing pointers passed through the system call API (allocating calls like brk(2) and mmap(2) being notable exceptions). DIOS instead takes the approach that object names and initialisation arguments (both of which are themselves data) must be backed by existing user-space memory, while all other pointer arguments (I/O buffers and references) must refer to kernel-allocated, ownership-tracked structures. These must have previously been passed to user-space from either (i) a reference-creating system call (for references), or (ii) the buffer-creating acquire_read(2)/acquire_write(2) calls (for I/O buffers). This allows the OS to validate the integrity of the pointers passed to the kernel.

A detailed description of each system call’s arguments and semantics can be found in Appendix B.1, and Appendix B.2 discusses how DIOS realises I/O requests using code examples.

4.6.1 Handling failures

Failures of components, machines, or network links are a common occurrence in any WSC. DIOS must therefore accommodate such failures: it must either handle them itself, or indicate
CHAPTER 4. DIOS: AN OPERATING SYSTEM FOR WSCS

... to user-space applications that they occurred.

In general, DIOS indicates a detectable failure in the distributed system by returning an appropriate error code from a system call, and it is the user-space application’s responsibility to check for such errors (as it is with legacy system calls). For example, if the remote object that a reference refers to has become inaccessible due to a network or machine failure, DIOS returns an **EHOSTUNREACH** error.

Some errors, however, can be recovered from without the user-space application having to be involved or aware: for example, RPCs involved in remote operations might be retried a configurable number of times before failing.

I describe the implementation of DIOS control RPCs – on which the system calls operating on remote objects rely – and their timeout mechanism in more detail in the next section.

### 4.7 Distributed coordination

A DIOS object exists on one machine, but can be interacted with across machine boundaries. Hence, DIOS must frequently coordinate operations and ship data between machines. This requires a means of communication between the DIOS kernels on different machines.

Two types of communication exist: **coordination** and **data transmission**. Coordination involves the creation and deletion of names and references, the delegation of references, task creation, and reference meta-data updates – all of which are typical “control plane” operations. Data transmission, by contrast, involves the exchange of bulk data for I/O requests (“data plane”), and its volume usually far exceeds that of coordination traffic.

**Coordination.** DIOS implements operations that coordinate different kernels via the DIOS **Coordination Protocol (DCP)**. The DCP is a binary RPC protocol that exchanges protocol buffers over reliable datagrams. Its messages are time-critical and should experience low end-to-end low latency.

Many DCP messages are sent directly to the appropriate machine, but name resolution requires a broadcast primitive. This can be realised via an existing broadcast primitive of the underlying network layer (e.g. UDP broadcast), but such primitives do not typically offer reliable delivery.

One way of attaining both reliable broadcast and a hard upper bound on the in-network latency of for a datagram is to use a QJUMP-style network [GSG*15]. QJUMP avoids any active ahead-of-time coordination, and as well as offering guaranteed delivery and a hard upper bound on latency for low-rate traffic.\footnote{Solutions with centralised coordination, such as Fastpass [POB*14], could also be employed to the same effect, but they introduce the need to “meta-coordinate” the coordination traffic.} QJUMP supports tens of thousands of messages per second at the scale of thousands of machines; since DCP name resolution messages are only a few bytes in...
size and can be batched, this is likely to be sufficient even at high load. QJUMP also allows for aggressive RPC timeouts: a guaranteed maximum round-trip time of several hundred microseconds in non-failure cases allows quick responses (which reduces latency) and failure detection (which reduces the cost of handling failures).

In the DIOS prototype, the DCP is implemented using a mixture of best-effort UDP broadcast and Reliable Datagram Sockets (RDS) [RDS15]. RPC messages are encoded using Protocol Buffers and handled by a DCP kernel thread on each machine. In Section 6.2.1.3, I show that DCP RPCs usually achieve round-trip times under 200 microseconds; other work has shown that tens of microseconds are feasible using commodity hardware [ORS+11].

Unlike reliable delivery, ordering of DCP RPCs is not required for correct operation. DIOS does not impose any ordering semantics in the system call API (§3.3). Re-ordering can occur if two system calls race within a machine or across the WSC interconnect.

---

**Figure 4.7:** Re-order correctness matrix for DIOS system calls. The colour of a cell indicates why a race does not affect correctness, while the symbol indicates whether the calls always commute, or – if not – which observable differences in outcome may occur.
Figure 4.7 illustrates the effect of pairwise system call re-ordering on observability (of the re-ordering) and correctness (of the outcome). Many system calls are idempotent, independent or can always be re-ordered without observable effect, i.e. they are always commutative. Some do – under certain circumstances, e.g. when invoked on the same reference – produce observably different results when re-ordered (i.e. the are not always commutative), but all such outcomes are permissible under the DIOS API semantics.

This lack of ordered messaging semantics differentiates the DCP from group communication protocols employed in other distributed operating systems: for example, all messages to a group in Amoeba are sequenced by a regularly elected leader [KT91].

**Data transmission.** The DCP is only used for small coordination messages. Bulk data transmission for remote I/O is the responsibility of transport objects, which I introduced in Section 3.7.1. Transport objects are local proxy objects that offer the same I/O semantics as the target object, but which forward any I/O requests on a reference to the remote machine on which the actual object referred to resides.

In the DIOS prototype, transport objects are supported by an underlying TCP connection to the kernel on the machine that holds the object. In response to DCP messages, I/O data is sent via this TCP connection. Alternative designs are conceivable: for example, RDMA might be employed for transport objects if available.

### 4.8 Scalability

As a distributed operating system, DIOS is premised upon a very large number of concurrently executing tasks, both local to a machine and across many machines in the WSC.

The DIOS design – unlike legacy POSIX abstractions – makes scalability a first class design goal, and actively attempts to address limitations observed in prior work [CKZ+13, §4]. In the following, I informally explain how DIOS concepts support this goal.

1. The system call API avoids serialising API constructs and allows for non-deterministic identifier choices to be made by the kernel: for example, the flat names and randomly chosen reference IDs are taken from unordered sets of identifiers and are independent of each other (unlike, say, monotonically increasing POSIX FD numbers).

2. The design avoids shared data structures: for example, the per-task reference tables are private, and both they and the shared name table are hash tables that lend themselves to scalable concurrent access via fine-grained locking [Cle14, p. 14].

---

13In the prototype, I use one connection per object; to scale to more objects, one could multiplex multiple object’s I/O data onto one TCP connection between each pair of hosts. This would result in fewer or equally many TCP connections as existing WSCs must support, but comes at the cost of reduced multi-core scalability within a machine. A high-performance datagram-oriented protocol that can be implemented scalably [Cle14, p. 40] might side-step this issue, however.
3. **Weak ordering** is permissible for many DIOS system calls: I discussed the re-ordering properties of DIOS system calls in the previous section, and showed that calls are always idempotent, are necessarily invoked on different arguments, or that a race returns an appropriate failure. Even data-dependent I/O requests may be re-ordered and overlapped if the application-level consistency policy permits it.

4. Resources are **released asynchronously**: for example, the deletion of objects, references, and names can take place asynchronously, as user-space code cannot immediately reuse an identifier (a common problem with POSIX FDs). Kernel-allocated I/O buffers (§3.7.2) also enable asynchronous release: the resources may be torn down after `release_read(2)` and `release_write(2)` return without any risk of the buffer being immediately re-used.

5. Some of the DIOS abstractions **decompose compound operations** compared to legacy system calls: for example, spawning a task via `run(2)` does not have the compound `fork(2)/exec(2)` semantics that the Linux system call API requires, but is more akin to the (scalable) `posix_spawn(2)`.

In the DIOS prototype, the practical scalability of operations is affected by Linux kernel implementation scalability. However, the above principles are conducive to commutative implementations, and I discuss in Section 7.1.2 how this might be attempted in future work.

### 4.9 Prototype implementation

For maximum flexibility, the DIOS kernel would be implemented from scratch. A clean-slate OS kernel might reduce the scalability bottlenecks present in many current kernels (cf. §A.1.3). However, an entirely new OS kernel forces a complete departure from legacy abstractions and loses backward-compatibility. While backward-compatibility is not *per se* a design goal for DIOS, support for legacy applications enables incremental migration. Hence, and in order to keep the implementation effort manageable, I chose to make DIOS deployable as a pluggable extension to existing operating systems.

This section describes the DIOS prototype, implemented as a host kernel extension module. To make the module portable, it interfaces to the host kernel via the DIOS Adaptation Layer (DAL). DIOS currently runs on a Linux host kernel, but I hope to adapt it for other host kernels in the future (§7.1.2).

In Section 4.9.1, I describe the necessary changes to the host kernel itself. Using the Linux kernel as an example, I show that only minimal changes are required. Following, I describe the implementation of the DAL and the DIOS core module in Section 4.9.2. Integration of the DIOS prototype with a host kernel allows incremental migration of existing applications, and I discuss this in Section 4.9.3.
CHAPTER 4. DIOS: AN OPERATING SYSTEM FOR WSCS

<table>
<thead>
<tr>
<th>Location</th>
<th>Linux source file</th>
<th>Change</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ELF binary loader</td>
<td>fs/binfmt_elf.c</td>
<td>+13 –0</td>
<td>Support ELF brands for DIOS binaries.</td>
</tr>
<tr>
<td>Process execution handler</td>
<td>fs/exec.c</td>
<td>+9 –0</td>
<td>Add initialisation code for DIOS tasks.</td>
</tr>
<tr>
<td>Process exit handler</td>
<td>kernel/exit.c</td>
<td>+8 –0</td>
<td>Add teardown code for DIOS tasks.</td>
</tr>
<tr>
<td>Process control block</td>
<td>sched.h</td>
<td>+10 –0</td>
<td>Add DIOS fields to process meta-data.</td>
</tr>
<tr>
<td>System call macros</td>
<td>syscalls.h</td>
<td>+17 –1</td>
<td>Add restrictions on legacy/DIOS syscalls.</td>
</tr>
<tr>
<td>System call table</td>
<td>syscall_64.tbl</td>
<td>+14 –0</td>
<td>Add thirteen new system calls.</td>
</tr>
<tr>
<td>ELF binary brands</td>
<td>uapi/linux/elf.h</td>
<td>+16 –9</td>
<td>Add new DIOS ELF brands.</td>
</tr>
<tr>
<td>System call handlers</td>
<td>new (dios/*)</td>
<td>+401 –0</td>
<td>Forward system calls to module.</td>
</tr>
</tbody>
</table>

Table 4.5: Changes made to the Linux host kernel for the DIOS prototype.

4.9.1 Host kernel changes

The host kernel must implement generic operating system functionality that DIOS relies on (via the DAL), as well as drivers for machine hardware, bootup code, a network stack, and low-level memory management code. All of this can be re-used from stable, mature host kernels, with DIOS building its abstractions on top.

A limited set of changes typically have to be made to a host kernel to deploy DIOS with it:

- **System call handlers** for the new DIOS system calls must be added. These must be part of the core kernel in most operating systems (e.g. Linux does not allow loadable kernel modules to rewrite the syscall table).

- **Process control block extensions** are required to differentiate DIOS tasks from legacy processes at runtime, in order to limit each to the correct set of system calls.

- **Process entry and exit code** must be amended to call into the initialisation and destruction routines for DIOS tasks, which handle DIOS-specific state such as the reference table.

- **The ELF binary loader** must be modified to recognise DIOS binaries and initialise them appropriately.

In the Linux prototype, the necessary changes amount to a patch changing about 500 lines (≈ 0.01% of the non-driver kernel source).\(^\text{14}\) Table 4.5 lists the key changes and their impact.

In addition to the above, the host kernel should provide a means of reliably delivering messages to remote WSC nodes. This can, however, also be implemented within the DAL by establishing

\(^{14}\text{Based on the DIOS patch for Linux kernel v3.14.}\)
reliable channels over an unreliable messaging protocol supported by the host kernel network stack.

In the Linux prototype, I use existing kernel support for Reliable Datagram Sockets (RDS) for DCP coordination. RDS supports Infiniband and TCP transports; my experiments using the latter transport over 10G Ethernet (§6.2).

4.9.2 DIOS modules

The DIOS abstractions are not inherently tied to any particular host kernel. In order to maintain portability to different host kernels, I implemented DIOS in two modules:

1. The **DIOS Core Module** (DCM), which contains the core DIOS logic (e.g. name and reference tables, system call handlers), but never invokes any host kernel functionality directly.

2. The **DIOS Adaptation Layer** (DAL), which indirects OS-independent requests for kernel functionality (e.g. starting a new process, installing memory mappings) to the invocations specific to the host kernel.

**DIOS Core Module.** The DCM implements the DIOS abstractions and OS functionality, including the name and reference tables, management and transformation code for names and references, an implementation of the DCP, and handlers for the DIOS system calls. It consists of about 6,900 lines of C code, and relies on the DAL for invoking host kernel functionality.

**DIOS Adaptation Layer.** The DAL indirects the following functionality, which must either be supported by a host OS kernel, or implemented in the DAL itself:

- **Process management:** creation and execution of user-space processes, management of DIOS-specific per-process information.

- **Memory management:** allocation of physical and virtual memory, mapping of kernel memory into user-space virtual address spaces.

- **Network access:** kernel network stack with support for UDP and TCP.

- **Block and character I/O:** block device access, writing to the console.

- **Data structures:** linked list, FIFO queue, hash table.

- **Locking and concurrency:** spin locks, reader-writer semaphores, atomic counters.

- **Concurrency:** kernel worker threads, wait queues.
Since the DAL is an adaptation layer, in most cases it simply wraps an appropriate host kernel function call. Hence, it is fairly compact: the implementation for the Linux kernel consists of about 3,000 lines of C code.

**Example.** Figure 4.8 illustrates the division of functionality between the kernel patch, the DAL module, and the DIOS Core Module using the example of a `lookup(2)` system call invocation. The system call is first invoked in a user-space task and handled by a shim system call handler implemented as part of the kernel patch. This shim handler proxies the system call invocation to the DIOS core module’s system call handling code, which in turn may rely on functionality offered by the DAL module. Once the DCM’s handling code has finished, the result is returned to user-space via the shim handler.

### 4.9.3 Binary types

Research operating systems sometimes translate conventional APIs (typically a subset of POSIX) to their new paradigms via a compatibility layer. The Unix compatibility layers of some new kernels developed from scratch take this approach – for example, in Mach [ABB+86, §8] or L4 [HHL+97] – but it usually comes with some overhead.

The DIOS prototype, by contrast, natively runs legacy applications side-by-side with DIOS applications. However, when doing so, DIOS must differentiate between process types in order to *(i)* initialise, manage, and destruct DIOS-specific state when needed, and *(ii)* restrict the availability of DIOS abstractions and system calls to applications that are permitted to use them.

DIOS can be configured for different degrees of restriction:
### Table 4.6:

<table>
<thead>
<tr>
<th>Binary type</th>
<th>ELF brand</th>
<th>System calls</th>
<th>Start via</th>
<th>Access pure DIOS obj.</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIOS</td>
<td>Pure</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0xD1 0x05</td>
<td>X</td>
<td></td>
<td>run(2)</td>
</tr>
<tr>
<td>Limbo</td>
<td>0xD1 0x11</td>
<td>✓</td>
<td></td>
<td>run(2)</td>
</tr>
<tr>
<td>Legacy</td>
<td>0xD1 0x13</td>
<td>✓</td>
<td>✓</td>
<td>legacy</td>
</tr>
<tr>
<td>Legacy ELF</td>
<td>any other</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
</tbody>
</table>

(† ✓ if in restricted mode.)

**Table 4.6:** DIOS supports four binary types, which have access to different system call APIs depending on the execution mode, which must be invoked differently and which may have limited access to objects.

- **In liberal mode,** it is possible for applications to exist in a “limbo” state, where they can access both DIOS system calls (and thus interact with DIOS objects) and invoke legacy OS facilities.

- **In restricted mode,** applications must commit to either DIOS abstractions or legacy abstractions and cannot mix them in the same process.

The mode choice is global to the machine, but a “branding” applied to each binary allows for different degrees of per-process restrictions. Table 4.6 shows the different types of binaries supported.

**Pure DIOS binaries** must use DIOS objects and system calls to access data and interact. They have the most aggressive restrictions applied to them, but, in return, are guaranteed to access only objects for which they have legitimately acquired a reference (see §4.5). Legacy abstractions (and thus possible side-channels) are unavailable.

A good example of a pure DIOS application is a MapReduce worker task, which takes user-provided lambdas (map() and reduce()) and applies them on inputs to deterministically generate outputs. The pure DIOS restrictions ensure that the user-provided code cannot leak data via legacy abstractions.

**DIOS limbo binaries** have the same access to DIOS objects and system calls as pure DIOS binaries, but they can also make legacy system calls. Unlike pure DIOS binaries, however, they cannot access objects with a “pure-only” attribute set.

An example limbo application is the Firmament scheduler (Chapter 5), which may use both legacy and DIOS abstractions: the former for monitoring tasks and the latter (e.g. run(2)) to start DIOS tasks.

**DIOS legacy binaries** are identical to DIOS limbo binaries, except that they must be executed using legacy host kernel means (e.g. exec(2)), rather than DIOS’s run(2) system call. This is useful when a legacy process without access to DIOS facilities must start a DIOS-enabled process.
One use case for DiOS legacy mode is the dizibox shell, a port of BusyBox, which is launched by init (a legacy process). However, dizibox can only execute pure and limbo DiOS binaries in liberal mode.

Non-DiOS legacy binaries execute on the machine as if it did not run DiOS at all. They can only use legacy host kernel abstractions, and the DiOS namespaces and objects are unavailable to them.

Most legacy utilities run on a WSC machine fall into this category, common examples being system utilities such as mount or init that perform purely local tasks.

In restricted mode, DiOS limbo binaries are treated as pure DiOS binaries (losing their ability to access legacy abstractions) and DiOS legacy binaries are treated as legacy binaries (losing their ability to use DiOS abstractions).

To inform the kernel of the type of a given binary on execution, DiOS modifies the Executable and Linking Format (ELF) header [ELF-64], and sets the fields specifying the OS ABI targeted by the binary (EI_OSABI) and the ABI version (EI_OSABIVERSION) to custom values.\(^{15}\)

The ability to run different binary types at the same time enables an incremental migration path from legacy abstractions to using DiOS ones. Appendix B.3 explains the migration process from a legacy binary, first to scheduling via Firmament, and then to using the DiOS abstractions, in detail.

### 4.10 Summary

In this chapter, I introduced the DiOS operating system design, which embodies the principles set out in Chapter 3. I also discussed a prototype implementation of it.

DiOS is a modern take on the old idea of a distributed operating system, and its design is specifically geared towards data-intensive applications on “warehouse-scale” compute clusters.

I described the key abstractions on which DiOS is built: distributed objects, which encapsulate state and data (§4.2); names, which serve as identifier capabilities (§4.3); groups, which are distributed namespaces (§4.4); and references, which realise handle capabilities (§4.5).

I also gave an overview of the current DiOS system call API (§4.6), explained how it realises I/O requests, implements distributed operation via the DCP and transport objects (§4.7), and how it embodies scalable design principles (§4.8).

Finally, I outlined how I have implemented a prototype of DiOS by extending the Linux kernel, taking care to make the implementation both portable to other host kernels in the future and compatible with legacy applications (§4.9).

\(^{15}\)This approach is inspired by FreeBSD’s Linux binary compatibility layer, which requires “branding” of Linux binaries for the ELF loader to set up appropriate system call traps [FBSD-HB, ch. 11.3].
The next chapter focuses on the Firmament cluster scheduler; I then evaluate both DIOS and Firmament in Chapter 6.
Chapter 5

Firmament: an accurate, flexible and scalable scheduler

Over thir heads a chrystal Firmament,
Whereon a Saphir Throne, inlaid with pure
Amber, and colours of the showrie Arch."

The efficient scheduling of tasks onto compute and I/O resources in the warehouse-scale computer is a key challenge for its operating system, and accordingly for D10S.

In modern WSCs, scheduling of work to compute resource happens at two disjoint levels:

1. the cluster scheduler places coarse-grained tasks onto machines, subject to resource availability, and maintains state about machines’ load and liveness; and

2. the kernel CPU scheduler on each node decides on the order in which threads and processes are executed, and reacts to fine-grained events by moving processes between different states, CPU cores, and priority levels.

The time horizons for which these schedulers make their decisions are very different: cluster schedulers’ decisions are in effect for several seconds, minutes, or even hours, while kernel CPU schedulers operate at millisecond timescales.

In this chapter, I describe the Firmament cluster scheduler. Firmament integrates information from both of the scheduling domains mentioned – cluster-level work placement and local-machine CPU scheduling – to improve overall task placement.

I first offer some background (§5.1), and briefly contrast Firmament with domain-restricted scheduling approaches in CPU and cluster scheduling. I then introduce the Quincy scheduler [IPC+09], whose optimisation-based approach Firmament extends.
Like Quincy, Firmament models the scheduling problem as a minimum-cost, maximum-flow optimisation over a flow network (§5.2). This approach balances multiple mutually dependent scheduling goals to arrive at a globally optimal schedule according to the costs.

Firmament can be used as a centralised or distributed scheduler. Each scheduler instance collects fine-grained information about machine resources and running tasks (§5.3).

Using this information, Firmament can express many scheduling policies via its flow network. I outline how data locality, fairness, and placement constraints are represented, and point out some limitations of the flow network approach (§5.4).

The minimum-cost, maximum-flow optimisation returns an optimal result for the costs specified in the flow network. These costs are defined by a cost model. Firmament supports customisable cost models, and I discuss four examples in this chapter (§5.5):

1. The original Quincy cost model, which trades off performance benefits of data locality against task preemption by expressing rack and machine affinities (§5.5.1).
2. A basic co-location interference model based on a per-task instructions per second (IPS) metric, as used in the Whare-Map system [MT13] (§5.5.2).
3. The coordinated co-location cost model that I developed for Firmament, which allows the cluster-level scheduler to consider all shared resources within a machine and places tasks directly on the most appropriate CPUs (§5.5.3).
4. A case study of a more unusual policy, an energy-aware cost model that optimises the placement of tasks over a highly heterogeneous cluster, using low-power machines to save energy when permissible under performance constraints (§5.5.4).

After describing Firmament, I discuss its algorithmic core, the minimum-cost, maximum-flow optimisation. I explain the prevalent algorithms and discuss how Firmament scales to large WSCs by solving the optimisation problem incrementally (§5.6).

Finally, I summarise the chapter (§5.7), before evaluating Firmament in the next chapter.

5.1 Background

I have already discussed the historic evolution of cluster schedulers and the key goals of recent systems in Section 2.3. In this section, I show that cluster schedulers can take some useful lessons from an ancestor, the decades-old domain of CPU scheduling (§5.1.1).

I then explain the design of the Quincy scheduler – which Firmament generalises – and the approach of modelling the scheduling problem as a flow network optimisation (§5.1.2).
### 5.1.1 Cluster and CPU scheduling

If we view the data centre as a computer, an obvious question is whether we can treat scheduling just as we would in a very large multi-core machine. There are indeed many commonalities, but also some key differences. Table 5.1 is an attempt at summarising them; I highlight the key dimensions and their impact on Firmament in the following.

**Decision scopes:** cluster schedulers make decisions over many machines, aggregating hundreds of thousands of cores, while CPU schedulers have a myopic view of a single machine’s cores. The latter therefore cannot notice imbalance across machines.

**Decision time horizons:** while even the fastest cluster schedulers take tens of milliseconds to

<table>
<thead>
<tr>
<th>Feature</th>
<th>Cluster schedulers ($\S 2.3$)</th>
<th>CPU schedulers</th>
<th>Quincy [IPC+09]</th>
<th>Firmament ($\S 5.2$ et seq.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scheduler architecture</td>
<td>centralised, distributed</td>
<td>centralised, per-CPU</td>
<td>centralised</td>
<td>supports either</td>
</tr>
<tr>
<td>Operating scale</td>
<td>100–10,000 machines</td>
<td>hundreds of cores</td>
<td>≤2,500 machines</td>
<td>≥10,000 machines</td>
</tr>
<tr>
<td>Workload types</td>
<td>batch, service</td>
<td>batch, interactive, real-time</td>
<td>batch</td>
<td>batch, service</td>
</tr>
<tr>
<td>Task length assumption</td>
<td>varies (scheduler-specific)</td>
<td>very short to infinite</td>
<td>≥seconds</td>
<td>short to infinite</td>
</tr>
<tr>
<td>Fairness</td>
<td>varying notions</td>
<td>varying notions</td>
<td>fair share of tasks</td>
<td>fair share of tasks</td>
</tr>
<tr>
<td>Constraints</td>
<td>soft, hard</td>
<td>affinity, pinning</td>
<td>X</td>
<td>soft, hard</td>
</tr>
<tr>
<td>Admission control</td>
<td>✓, multi-dim.</td>
<td>✓, ✓</td>
<td>✓, ✓</td>
<td>✓, ✓</td>
</tr>
<tr>
<td>Preemption</td>
<td>common</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Explicit priorities</td>
<td>common</td>
<td>✓</td>
<td>X</td>
<td>✓, implicit</td>
</tr>
<tr>
<td>Dynamic priorities</td>
<td>fairly rare</td>
<td>✓</td>
<td>✓, implicit</td>
<td>✓, implicit</td>
</tr>
<tr>
<td>Global view of WSC</td>
<td>✓</td>
<td>X</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Application-specific policies</td>
<td>some</td>
<td>X, except user level schedulers</td>
<td>X</td>
<td>supported</td>
</tr>
<tr>
<td>Heterogeneity awareness</td>
<td>rare</td>
<td>X</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Micro-architecture awareness</td>
<td>X</td>
<td>some</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Interference awareness</td>
<td>rare</td>
<td>rare, heuristic-based</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Data locality awareness</td>
<td>✓, common</td>
<td>rare</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Deadline support</td>
<td>rare</td>
<td>rare (RT only)</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Gang scheduling</td>
<td>rare</td>
<td>✓, common</td>
<td>X</td>
<td>✓</td>
</tr>
<tr>
<td>Automatic resource scaling</td>
<td>rare</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

**Table 5.1:** Similarities and differences between cluster task scheduling, CPU scheduling, Quincy, and Firmament along different feature dimensions.
CHAPTER 5. FIRMAMENT: A WSC SCHEDULER

make their decisions [OWZ+13], CPU schedulers make fast decisions on context switches, taking on the order of microseconds.

**Priority notions:** cluster schedulers often derive ad-hoc task priorities based on fairness regimes and application-specific knowledge, while notions of static and dynamic priorities are well-established in CPU scheduling.

**Locality and constraints:** while process pinning and custom affinities are rarely used in CPU schedulers (mostly in HPC environments), cluster schedulers routinely rely on placement constraints for locality preferences and improved task performance.

**Micro-architecture awareness:** typically, cluster schedulers do not take machines’ micro-architecture (e.g. shared caches and NUMA) into account when they make placement decisions. Some modern SMP CPU schedulers, however, have heuristics that schedule processes to avoid pessimal sharing [Lam13].

**Interference awareness:** cluster schedulers move work across machines, and some consider negative interference on shared resources [MT13; ZTH+13]; CPU schedulers, by contrast, can only work with the local set of processes, and have little leeway to avoid interference.

There are some other minor differences: CPU schedulers can assume shared kernel state in memory even if they make short-term decisions local to a CPU, and thus have a centralised component; they more often support gang scheduling; and real-time (RT) schedulers have fine-grained deadline support, while cluster schedulers only sometimes support coarse-grained deadlines for batch workloads.

Many of these differences, however, boil down to one key difference: in cluster schedulers, application-specific knowledge is commonly available, while CPU schedulers, designed for general-purpose workloads, do not have access to it. Some researchers have recently combined application-specific knowledge with CPU scheduling on a single machine (e.g. in Callisto [HMM14]); Firmament combines them across machines and uses detailed per-machine and per-process information (such as that used by CPU schedulers) in the cluster scheduler.

At its core, Firmament expresses the scheduling problem as a minimum-cost optimisation over a flow network, which was introduced in Quincy [IPC+09].

### 5.1.2 The Quincy scheduler

Isard et al. developed the Quincy scheduler [IPC+09] to coordinate Dryad [IBY+07] data processing clusters. Quincy takes an unusual approach compared to other contemporary schedulers: instead of servicing task queues according to a heuristic, it models the scheduling problem as a constraint-based optimisation over a flow network.
Traditional cluster schedulers typically service multiple work queues according to a pre-defined policy. This design is similar to the multi-level feedback queue (MLFQ) architecture of single-machine CPU scheduler [AA14, ch. 8; PS85, pp. 127–9]. It has the benefits of conceptual simplicity and low scheduling overhead. However, the lack of clear prioritisation and the increasingly complex set of properties and trade-offs that ought to be considered in WSC-wide scheduling makes an effective queue-based abstraction challenging to build.

Consider, for example, the three-way relationship between data locality, fair resource sharing, and scheduling delay (i.e. a task’s wait time in queues): a task may benefit from better locality if it waits for longer (increasing scheduling delay), or if it preempts a running task (reducing fairness). Alternatively, it may run sooner (reducing wait time), but in a worse location (reducing locality). Heuristically identifying the ideal trade-off over all tasks is difficult.

Furthermore, a queue-based scheduler may choose assignments that satisfy its local heuristics, but which do not correspond to an overall optimum solution.

The key insight behind Quincy’s approach is (Isard et al. [IPC+09, §4]):

 [...] that there is a quantifiable cost to every scheduling decision. There is a data transfer cost incurred by running a task on a particular computer; and there is also a cost in wasted time to killing a task that has already started to execute. If we can at least approximately express these costs in the same units (for example if we can make a statement such as “copying 1GB of data across a rack’s local switch costs the same as killing a vertex that has been executing for 10 seconds”) then we can seek an algorithm to try to minimise the total cost of our scheduling assignments.

Based on this premise, modelling the scheduling problem as an optimisation on a graph makes sense, as the optimiser is able to consider all jobs, tasks, and machines at once, and may consider the impact of a broad range of different choices for each of them. If the costs are set correctly, the resulting schedule is globally optimal – a property that heuristic-driven, queue-based schedulers cannot be guaranteed to achieve.

Firmament follows the same basic approach as Quincy, but generalises it: as Table 5.1 shows, Firmament supports more scheduling policies than Quincy, adds several advanced features based on recent research, and uses information traditionally unavailable to cluster schedulers.

### 5.2 Scheduling as a flow network

The approach used at the core of Firmament is to model the scheduling problem as an optimisation over a flow network (§5.1.2).

This section explains how Firmament constructs the flow network and introduces the entities in it (§5.2.1). Further, I explain how arc capacities (§5.2.2) and costs (§5.2.3) are assigned.
Figure 5.1: Example flow network as generated by Firmament for a four-node cluster consisting of two racks with two machines each. Flow is generated at task vertices ($T_{j,i}$) and routed to the sink ($S$) via either machine vertices ($M_m$) or per-job unscheduled aggregators ($U_j$). The dashed line is a task preference arc ($T_{0,2}$ prefers $M_1$), while the dotted line corresponds to $T_{0,1}$ running on $M_3$.

Finally, I introduce the notion of aggregated vertices based on equivalence classes: a key building block of Firmament’s flow network, which differs from the original Quincy approach and enables support for additional scheduling policies (§5.2.4).

5.2.1 Network structure

Firmament, like Quincy, models the scheduling problem as a flow network. It routes flow from task vertices to a sink via a path composed of directed arcs, and uses the traversal of flow through machine vertices to indicate task assignment. Each arc in the flow network has an associated cost. Hence, minimising the overall cost of routing flow through this network corresponds to the policy-optimal schedule.\footnote{“Policy-optimal” means that the solution is optimal for the given scheduling policy; it does not preclude the existence of other, more optimal scheduling policies.} I discuss the details of common optimisation algorithms for this problem in Section 5.6; in the following, I describe the construction of the flow network itself, and explain how it expresses the cluster task scheduling problem.

The core component of the flow network is a tree of WSC resources (the resource topology) that corresponds to the physical real-world data centre topology. For example, machine vertices are usually subordinate to rack vertices, which in turn descend from a common cluster aggregator vertex. Figure 5.1 shows an example flow network for a WSC consisting of four machines...
(M₀–M₃) distributed over two racks (R₀–R₁). The cluster aggregator vertex is labelled as X.²

In addition to vertices for the WSC resources, a Firmament flow network also contains a vertex for each task in the WSC, independent of whether it is currently scheduled or not. These task vertices are flow sources and each generate one unit of flow. In the example, two jobs with three and two tasks are present (T₀₀–T₀₂ and T₁₀–T₁₁).

The flow generated by task vertices is eventually drained by a sink vertex (S). To reach the sink vertex, the flow passes through the resource topology and ultimately reaches a leaf vertex (in the example, a machine). Each leaf vertex has an arc to the sink vertex.

Since there may not be sufficient resources available to execute all runnable tasks, some tasks may need to wait in an unscheduled state until resources become available. Since these tasks also generate flow, the network must somehow route their flow to the sink. This role is performed by the per-job unscheduled aggregator vertices (U₀–U₁). One such vertex exists for each job, and all unscheduled aggregators are connected to the sink vertex.

Only the leaf vertices of the resource topology and the unscheduled aggregator vertices are connected to the sink. This enforces the invariant that every task must either be scheduled or unscheduled, since it can only be in one of the following states:

1. routing its flow to through the resource topology, from whence it is routed through a leaf vertex where the task is scheduled; or
2. routing its flow directly through a leaf vertex where the task is scheduled; or
3. routing its flow through an unscheduled aggregator, so that the task remains unscheduled.

By contrast with Quincy, Firmament extends the resource topology with additional information. The leaf vertices do not have to be machines: instead, the micro-architectural topology inside the machines – e.g. NUMA layouts and shared caches – can be automatically extracted and added to the flow network. I explain this extension in detail in Section 5.3.2.

The quality of scheduling decisions resulting from a given flow network depends on the capacities and costs assigned to the arcs connecting the aforementioned vertices. In the following sections, I explain how these arc parameters are determined.

### 5.2.2 Capacity assignment

Each arc in a flow network has a capacity bounded by the minimum and maximum capacity, i.e. within \([cap_{\text{min}}, cap_{\text{max}}]\). In Firmament, as in Quincy, \(cap_{\text{min}}\) is generally zero, while the value of \(cap_{\text{max}}\) depends on the type of vertices connected by the arc and on the cost model.³

²The exposition here matches Quincy’s flow network structure, but in Firmament, the specific structure is not fixed: as I show later, the X, Rᵢ, and Mₘ vertices are merely examples of aggregating equivalence classes.

³For simplicity, “the capacity” refers to the maximum capacity value in the following.
Figure 5.2: The flow network in Figure 5.1 with example capacities on the arcs.

Figure C.2 illustrates a typical capacity assignment. Outgoing arcs from tasks have unit capacity, and since each machine (M₀ – M₁) can run a maximum of K tasks, the arcs from machines to the sink have capacity K. Arcs from the cluster aggregator (X) to rack aggregators (R₀ – R₁) have capacity rK for r machines in each rack. Finally, the capacities on the arcs from jobs’ unscheduled aggregators (U₀ – U₁) to the sink are set to the difference between the upper bound on the number of tasks to run for job j (F_j) and the lower bound (E_j).

Appendix C.2 explains the capacity assignment in more detail and explains why the $F_j - E_j$ capacity on unscheduled aggregators’ arcs to the sink makes sense. Firmament generally assigns the same capacities as Quincy, but some cost models customise them.

While the arc capacities establish how tasks can route flow through the network and enforce fairness constraints, the costs describe how preferable possible scheduling assignments are. Some costs are general and always assigned in the same way, but others are configurable, allowing different scheduling policies to be expressed. I explain the general cost terms in the next section, and describe four specific cost models configuring others in Section 5.5.

### 5.2.3 Cost assignment

The assignment of costs in Firmament follows several general rules. The cost on an arc expresses how much it costs to schedule any task that can send flow on this arc on any of the machines reachable through the arc. Depending on where the arc is located in the flow network, this notion can be restricted in useful ways (Figure 5.3):
Figure 5.3: The example scheduling flow network in Figure 5.1 with annotations highlighting the cost scopes for each arc type, and zero cost terms.

**Task-specific** arcs can only receive flow from a single task. Their cost expresses factors *specific to the nature of the task*, e.g. the amount of input data that the task fetches.

*Example:* the arcs $T_{j,i} \rightarrow X$ are task-specific arcs.

**Aggregator-specific** arcs originate at an aggregator and point to another aggregator. Their cost quantifies the cost of *any* task aggregated by the source aggregator running on *any* resource reachable from (≡ aggregated by) the destination aggregator.

*Example:* the arcs $X \rightarrow R_r$ are aggregator-specific arcs, since both the cluster aggregator, $X$, and the rack-level vertices, $R_r$, constitute aggregators.

**Resource-specific** arcs point only to a leaf in the resource topology, i.e. a vertex representing a machine or another processing unit directly connected to the sink. Since such an arc can only route flow to the sink via the resource, it carries a cost *specific to running on this resource* (e.g. current existing load).

*Example:* the arcs $R_r \rightarrow M_m$ are resource-specific arcs, because the machines’ vertices are leaves in this flow network.

The use of aggregator vertices is beneficial as it reduces the number of arcs required from multiplicative to additive in the number of source and target entities (a property that Firmament’s equivalence classes exploit, as I show in §5.2.4). However, aggregation also reduces the specificity of the costs, as costs specific to an individual entity (task or machine) cannot be expressed on any arcs on the far side of an aggregator.
Parameter | Edge | Meaning
--- | --- | ---
v\_j\_i | T\_j\_i \rightarrow U\_j | Cost of leaving T\_j\_i unscheduled.
α\_j\_i | T\_j\_i \rightarrow X | Cost of scheduling in any location (wildcard). \(^{(†)}\)
γ\_j\_i\_m | T\_j\_i \rightarrow M\_m | Cost of scheduling or continuing to run on machine M\_m.
ω | – | Wait time factor.
v\_j\_i | – | Total unscheduled time for task.
θ\_j\_i | – | Total running time for task.

\(^{(†)}\) not used in some cost models

Table 5.2: General cost terms in Firmament’s flow network, which exist independently of the cost model chosen. Cost models assign specific values to these terms in different ways.

Some arc costs are key to the operation of Firmament. I explain those in the following and list them in Table 5.2. The cost models presented in Section 5.5 may add additional arcs and cost expressions, and use specific expressions to compute the values for the cost terms in Table 5.2.

**1:1 task–resource mappings.** If an arc points directly from a task to a schedulable entity (in this case, a machine), the cost associated with the arc is denoted by γ\_j\_i\_m for the \textit{i}th task of job \textit{j} and machine \textit{m}. This cost is specific both to the task (T\_j\_i) and the machine (M\_m). All running tasks have a direct arc to the resource they run on, which carries a cost of γ\_j\_i\_m. In many cost models, γ\_j\_i\_m is discounted by a multiple of a task’s current runtime, θ\_j\_i, both to control hysteresis and to ensure that finite-duration tasks (e.g. batch tasks) eventually finish.

**Wildcard mappings.** By contrast, if an arc points to the cluster aggregator, then the cost on the arc expresses the cost of running the task on any subordinate resource. This is denoted by α\_j\_i, and typically expresses a worst-case cost.

**Unscheduled mappings.** If insufficient resources are available, a task may remain unscheduled. The cost of this option is denoted by v\_j\_i, which is applied to the arc to the unscheduled aggregator vertex for job \textit{j} (U\_j). In order to ensure progress and to reduce scheduling delay, the cost on this arc grows as a function of the task’s waiting time: v\_j\_i denotes the aggregate number of seconds the task has been waiting in unscheduled state. This encompasses both the initial wait before the first assignment to a machine and any further wait times incurred when the task was preempted. The value is scaled by a constant \textit{wait time factor}, ω, which increases the cost of keeping the tasks waiting for longer (assuming ω > 1).

**Mapping changes due to preemption.** Once a task is scheduled on a machine, the scheduler may decide to \textit{preempt} it by routing its flow through the unscheduled aggregator vertex on a subsequent scheduling iteration. It may also choose to \textit{migrate} the task by routing its flow through a different machine vertex; this implies a preemption on the original machine. Some
cost models do not allow preemption, however – in these cases, all arcs apart from the 1:1 mapping to the resource where the task is scheduled are removed (effectively “pinning” it).

Many tasks have similar characteristics, and likewise many resources can be treated as similar by the scheduler. This enables Firmament to make use of aggregators for tasks and machines, which in turn simplifies the construction of higher-level cost models. The following section introduces equivalence classes, which aggregate vertices in the flow network.

### 5.2.4 Equivalence classes

Firmament classifies both tasks and resources (e.g. machines, CPU cores) into *equivalence classes*. These contain elements that are expected to behave comparably, all other factors being equal, and which may thus be treated as fungible for scheduling. This notion is similar to the equivalence classes in alsched [TCG+12].

Equivalence classes aggregate parts of the flow network and allow properties of aggregates to be expressed. For example, tasks in task equivalence class $c_a$ may work particularly well on machines in equivalence class $c_b$. Thus, we create a vertex that all tasks in $c_a$ are connected to, and connect it to a vertex that has arcs to all machines in $c_b$. An arc with a low cost between these vertices can now express the desired property (Figure 5.4).

![Figure 5.4: Preference of tasks in $c_a$ for machines in $c_b$ expressed via equivalence class aggregators ($A_{c_a}$ and $A_{c_b}$). Dashed lines delineate equivalence classes.](image)

The use of equivalence class aggregators significantly reduces the number of arcs required to express desirable properties: it goes from $O(nm)$ to $O(n + m)$ for a pair of equivalence classes of sizes $n$ and $m$. Thus, the use of equivalence classes improves overall scheduler scalability. I discuss this further in Sections 5.5.2 and 5.5.3.

**Tasks.** The equivalence classes of a task are determined by the cost model, and can depend on factors such as the task’s job name, its binary, its inputs and the task that created it. Typically, a deterministic hash function combines this information to generate a task equivalence class identifier.

A task’s equivalence classes can also have different levels of specificity: for example, all tasks of a job form an equivalence class, all tasks running the same binary are members of an equivalence
class, and all tasks running the same binary with the same arguments on the same inputs are part of a (narrowly defined) equivalence class.

When a cost model uses multiple task equivalence classes, the cost on their outgoing arcs should be inversely proportional to the specificity of the equivalence class. This gives priority to the more specific equivalence classes, which likely yield better matches as their members’ behaviour is less variable. For example, we might expect all tasks in a job to perform similar work and thus have similar characteristics. However, if the job consists of heterogeneous tasks or the tasks’ input data is skewed, a more specific equivalence class (e.g. based on the task binary, arguments and inputs) has more predictive power.

**Resources.** Equivalence class can also aggregate over resources. Most commonly, they aggregate machines, but other resource can be aggregated similarly. In fact, the cluster aggregator (X) and rack aggregators (R_r) in Quincy are implemented as resource equivalence classes in Firmament.

A machine equivalence class is usually a function of the machine’s architecture. Firmament generates the machine equivalence class identifier by hashing machine meta-data, e.g. the machine’s micro-architectural topology (§5.3.2).

### 5.3 Firmament implementation

I implemented Firmament as a cluster manager in approximately 22,000 lines of C++, of which about 7,000 relate to the scheduler. The remainder implement task execution and management, health monitoring and machine management, functionality similar to that found in Mesos [HKZ+11] and Borg [VPK+15].

Unlike most previous systems, Firmament can operate both as a centralised or as a distributed scheduler. It uses an optimistically concurrent shared-state approach akin to Omega [SKA+13] to allow multiple distributed schedulers to make parallel decisions. When multiple schedulers are used, they form a delegation hierarchy. I describe the high-level architecture of Firmament and the possible configurations in Section 5.3.1.

Firmament extracts each machine’s micro-architectural topology and represents it as part of the flow network to permit scheduling decisions at CPU granularity. Section 5.3.2 explains how the topology information is obtained and used.

In order to make good scheduling decisions at fine granularity, Firmament collects information about workloads through automatic profiling. Section 5.3.3 describes the methods used to extract the information, and how it is stored and aggregated.
Figure 5.5: Contrasting (a) single, (b) delegating, and (c) fully distributed Firmament deployment setups; showing Coordinators and Schedulers.

5.3.1 Multi-scheduler architecture

Firmament supports a variety of scheduling setups, including ones with multiple concurrent and distributed schedulers, and ones in which schedulers form hierarchies.

Each machine in the WSC runs a user-space Firmament coordinator process. Coordinators may be arranged in parent–child relationships, where parents can schedule tasks on their children and delegate the tasks to them for execution. As each coordinator runs a scheduler, their nested hierarchy forms a tree of schedulers.

However, the flip side of permitting delegation is that coordinators’ schedulers must sometimes operate with partial knowledge, as their local state may be stale. Any task delegation to a remote coordinator is optimistically concurrent with other scheduling decisions: it may fail when racing with the target or another coordinator. This could, at worst, lead to many failed delegations, but it cannot lead to deadlock: one of the racing coordinators always succeeds. This approach is similar to the way in which multiple schedulers are supported in Omega [SKA+13], Apollo [BEL+14], and Tarcil [DSK15].

This design affords significant flexibility in configuring the scheduling paradigm:

(a) **Centralised scheduling** can be implemented by having a single “master coordinator” with all machine coordinators as its children (Figure 5.5a). All jobs are dispatched to this master coordinator, which delegates tasks to the machines for execution. The child coordinators use a no-op scheduler that accepts no jobs. This setup is identical to traditional centralised cluster schedulers.

(b) **Hierarchical distributed scheduling** has coordinators arranged in a tree, for example with per-machine, per-rack, and master coordinators (Figure 5.5b). Jobs can be submitted to any of these coordinators, which either schedule them directly to local resources, or on resources attached to a subordinate coordinator.

(c) **Fully distributed scheduling** is possible if jobs are dispatched to individual machine coordinators (Figure 5.5c). This can be used to implement policies akin to Sparrow’s fully distributed operation [OWZ+13].
Figure 5.6: High-level structure of the Firmament coordinator.

Figure 5.6 shows the high-level architecture of the Firmament coordinator and how it interacts with an executing task. In the following sections, I explain several of the key coordinator modules in detail.

5.3.2 Machine topology extraction

Each coordinator is host to a resource topology, which includes the resource topologies of any child coordinators. Resources are either (i) schedulable processing units (i.e. CPU threads), or (ii) aggregates that correspond to some degree of sharing between nested elements (e.g. CPU caches, memory controllers, shared machine-level devices, or shared rack-level network connections).

On startup, a Firmament coordinator must bootstrap its resource topology. It may find itself in either of two situations:

1. It has directly attached physical resources that it may schedule. This is necessarily the case for any (useful) leaf in the tree of coordinators. In this situation, the coordinator discovers the resource topology and forwards it alongside a registration message to its parent coordinator (if any).

2. It is a “virtual” node and has no directly attached machine resources. However, it may have (or subsequently discover) children who register their resources with it.

In the former case, information about the machine resources must be extracted from the OS. Firmament uses the portable hwloc library to discover local resources, which in Linux are obtained from sysfs [BCM+10b]. The information returned includes the setup and sizes of any shared CPU caches, as well as information regarding NUMA and simultaneous multi-threading (SMT).

---

4This can also be a DAG if the children are able to multi-cast messages to several parents.
Finally, after the resource discovery is completed – or when a new child coordinator’s registration request is received – the part of the flow network that corresponds to the coordinator’s resource topology is updated to include any new resources. Failures of child coordinators are handled by removing the appropriate subtree from the local resource topology.

Compared to Quincy, Firmament thus ends up with a larger flow network and represents more fine-grained information about the WSC machines’ resources in it. This explicit representation of machine resources not only allows Firmament to make more fine-grained placement decisions, it also offers a way of representing heterogeneity in machine types and architectures to the scheduler.

Figure 5.7 shows the example flow network from Figure 5.1 extended with the topology information extracted by Firmament. This extra detail, however, does not come for free: it increases the number of nodes and edges in the flow network and necessitates judicious aggregation of subtrees via equivalence classes (§5.2.4) in order to keep the number of arcs from tasks to resources bounded.

5.3.3 Task profiling

The Firmament coordinator on each node in the WSC is responsible for running, monitoring and profiling tasks. In particular, using the mechanisms described in the previous section, it acquires statistical information about tasks’ performance and resource needs. This information is stored
Table 5.3: Metrics tracked by the Firmament coordinator for each task running on the local machine. Additional per-machine metrics such as available and used resources are also tracked.

The information is acquired from two principal sources:

1. Operating system resource accounting information, exposed, for example, via procfs in Linux. This includes information about the aggregate RAM and CPU time consumed, yield and preemption counts, and other information that accrues as a result of OS choices.

2. CPU hardware performance counters, which track low-level and micro-architectural events. This includes metrics such as the last-level cache miss count, the frequency of stalls due to DRAM access or I/O, and the number of instructions per memory access.

The precise list of metrics tracked in the current Firmament implementation is shown in Table 5.3.

Metrics can be measured either by sampling them continuously at intervals during task execution, or by retrieving a single summary at the end of the task’s execution. The latter approach is useful to determine aggregate metrics over the task’s lifetime (e.g. its total number of CPU migrations), but works less well for long-running service tasks which do not per se terminate at some completion point, since it may take a very long time until the metrics are reported for such tasks.

All information collected is stored in the coordinator’s knowledge base and forwarded to other coordinators as appropriate. Subordinate coordinators forward new information to their parent coordinator, although they may aggregate it for a while in order to send a batched update.

The collection of such information is not entirely novel: CPU performance counter information has been used for WSC scheduling before. For example, CPI\(^2\) at Google implements reactive
task migration based on sampling the cycle and instruction counters. Using the cycles-per-instruction (CPI) metric, CPI\(^2\) detects negative interference between co-located tasks [ZTH\(^+\)13]. Likewise, Mars et al. have used performance counter information on the frequency of last-level cache (LLC) misses to avoid interference on a single machine via the CAER runtime [MVH\(^+\)10].

Firmament routinely collects performance counter information, and aggregates the information across many machines. This allows Firmament to build a profile specific to the task, the machine type it executes on, and other co-located tasks. As many tasks often perform the same work in parallel in a WSC environment, Firmament uses its equivalence classes to combine their performance metrics and rapidly attain an accurate profile: task profiles are constructed in the knowledge base using the performance information received for all tasks in the same equivalence class. Since the cost model controls which equivalence classes tasks are assigned to, this allows flexible collection of metrics along different dimensions (e.g. task and machine type combinations).

5.4 Scheduling policies

Firmament’s approach of optimising a flow network affords the opportunity for different scheduling policies to be implemented by changing the flow network’s structure and its cost and capacity parameters. While Quincy supports a single policy based on data locality and preemption, my work generalises the flow network optimisation approach to other policies. In this section, I explain how three key policy elements – placement constraints, fairness, and gang scheduling – can be modelled and composed.

Like most cluster schedulers, Firmament supports data locality preferences and both job-level and task-level constraints. Section 5.4.1 shows how they are encoded in the flow network.

Global fairness across jobs and users is important in multi-tenant environments (cf. §2.3.4), although less crucial in single-authority WSCs [SKA\(^+\)13, §3.4]. In Section 5.4.2, I explain the notions of fairness that Firmament can support.

When tasks operate in tight synchronisation, they may require gang scheduling in order to avoid wasting resources. I show in Section 5.4.3 that Firmament can support both strict (all tasks) and relaxed (\(k\)-out-of-\(n\) tasks) notions of gang scheduling.

Although powerful, the flow network optimisation approach is not without limitations. Section 5.4.4 discusses policy elements that are challenging to accommodate, specifically: combinatorial constraints, and global invariants.

5.4.1 Data locality and constraints

Accessing input data locally on a machine, whether in memory or on persistent storage, is often advantageous to task performance. Data locality is therefore a key part in the original Quincy
scheduling policy [IPC+09, §2.3]. Specifically, locality drives placement preferences in Quincy, as direct preference arcs are added from each task to machines with local data. Some of the cost models for Firmament described in Section 5.5 support similar notions of locality preferences.

While Quincy considers locality preferences only, the approach is generalisable: Firmament can also express other placement constraints (§2.3.3). Constraints can be expressed at different granularities by pointing arcs to the relevant aggregator or resource vertices in the flow network. Different types of constraints are expressed in different ways:

**Soft constraints** use the same mechanism as locality preferences: they are expressed by adding preference arcs with attractive cost values from a task to the relevant location. Most cost models for Firmament make extensive use of these constraints.

**Hard constraints** must ensure that no placement violating the constraint is possible. There are two ways to achieve this, depending on whether the constraint expresses a positive or a negative affinity:

1. **Affinity constraint:** remove the task’s “wildcard” arc to the cluster aggregator vertex, \(X\), and add arcs from the task to permissible locations (or aggregators thereof). The task may now only schedule either in a suitable location, or not at all.

2. **Anti-affinity constraint:** ensure that all paths to locations that violate the constraint have arcs of cost greater than the maximum cost of leaving the task unscheduled forever.\(^5\) This works well if a location needs to be made unavailable to a class of tasks (e.g. batch tasks). Other tasks (e.g. service tasks) can still schedule there via other arcs, or if their unscheduled cost exceeds the anti-affinity cost.

In either case, use of hard constraints on a job breaks any progress guarantee to that job: tasks may never schedule if suitable locations fail to become available, since a wildcard aggregator such as \(X\) cannot be used.

**Complex constraints** involve inter-task dependencies. Since the flow network’s arc costs cannot be dependent on each other, these constraints can only be applied reactively. To use them, a job “drip-feeds” tasks into the system, making one task at a time eligible for scheduling. Each time, the task’s complex constraints are adjusted based on previous decisions. However, the scheduler may decide to invalidate a premise (≡ a prior assignment), which may require re-visiting it in the following round. As a consequence, progress is not always guaranteed.

Such reactive, multi-round scheduling increases the scheduling delay for tasks with complex constraints. However, this also tends to be the case in current environments: at Google, for example, tasks with complex constraints take up to \(10\times\) longer to schedule [SCH+11].

---

\(^5\)This only works if there is a ceiling on the growth of \(\omega^j_i\), which is true for practical implementations.
Generally, the higher the degree of aggregation of the target vertex of a constraint, the more likely it is to be respected and the less the scheduler runtime is affected by it. This is a consequence of the fact that fine-grained constraints introduce more additional arcs, and the scheduler runtime is proportional to the number of arcs in the flow network (see §5.6.2). Equivalence classes (§5.2.4) allow constraints to be applied to entire sets of tasks or resources.

Two common, but not entirely straightforward “constraints” in cluster scheduling are (i) global fairness guarantees across jobs sharing a cluster (cf. §2.3.4), and (ii) gang-scheduling of all of a job’s tasks. In the next section, I explain how Firmament enforces fairness guarantees using the flow network; Section 5.4.3 looks at gang scheduling.

5.4.2 Fairness

Firmament supports notions of fairness that partition WSC resources according to fractional shares of the number of running tasks.

Like Quincy, Firmament does not offer strict fairness (where fair shares are maintained at all times), but rather puts a bound on the maximum effect unfairness may have on job completion time. Specifically, the maximum slowdown experienced due to unfairness is proportional to the number of jobs that the cluster is shared with; in other words:

“For a job taking \( t \) time running in isolation on an idle cluster, it takes no more than \( Jt \) time to complete on a cluster running \( J \) concurrent jobs.” [IPC\(^*\)09, §1, §2.4, §5.2]

This notion of fairness is enforced by a combination of admission control and adapting the maximum and minimum number of running tasks for each job (\( F_j \) and \( E_j \) for job \( j \), respectively; cf. §5.2.2).

Each job’s fair share of the WSC is represented by its share of the schedulable leaf resources in the flow network, e.g. its share of machines. I denote the number of tasks allowed per fair share for job \( j \) as \( A_j \). It can be enforced by setting \( E_j = A_j = F_j \), which forces the scheduler to work towards the fair share. To reach this goal, the scheduler may preempt other tasks and start new tasks.

Like most schedulers, Firmament can experience transient periods of unfairness. Typically, schedulers rely on task churn to converge towards fair shares [GZH\(^+\)11; HKZ\(^+\)11; OWZ\(^+\)13; BEL\(^+\)14], and may additionally use preemption to accelerate convergence. Firmament takes the same approach; however, unfairness may still occur if there are no spare resources in the WSC and there are no tasks eligible for preemption.

Max-min fair policies. The computation of the fair shares (\( A_j \)) can follow a max-min fair allocation, but the flow network optimisation does not guarantee that preferred resources (e.g.
those resources pointed to by a task’s preference arcs) are split in a max-min fair way. As noted by Ghodsi et al. [GZS+13, §8], this can lead to unfairness especially if resources are allocated in multiple dimensions (e.g. CPU and RAM allocations), as in DRF [GZH+11; BCF+13] and Choosy [GZS+13].

However, Firmament’s flow network can approximate multi-dimensional max-min fairness. To achieve this, any increase of $A_j$ – granting additional tasks to job $j$ – is subject to a condition: extra tasks are only granted if, across all dimensions, the maximum demands of any waiting task in $j$ can be satisfied without violating max-min fairness. The maximum demands must be used because any waiting task may potentially schedule. Firmament must hence assume that the worst possible task in every dimension is chosen in order to maintain strict max-min fairness. This approximation is more coarse-grained than DRF-style max-min fairness, however: as the threshold for allowing another task is based on an artificial union of the worst-case demands, we might miss opportunities to schedule some (or even all) of a job’s tasks without violating the fair share.

Fortunately, heavy-handed enforcement of complex fair sharing policies is not typically required in WSCs (unlike in multi-tenant “cloud” environments): anecdotally, Google “[does] not care about fairness” and instead uses quotas and out-of-band policies to steer users’ behaviour [Wil14, 18:20–19:00]. Indeed, there are use cases in which deliberate unfairness is sometimes welcome: one key example is the ability of more important workloads (e.g. service jobs) to preempt “best effort” jobs even if this violates their fair share of the cluster.

5.4.3 Gang scheduling

Some WSC jobs cannot make progress unless all their tasks are running (for example, a synchronised iterative graph computation), while others can begin processing even as tasks are scheduled incrementally (e.g. a MapReduce job).

Firmament cannot easily express gang-scheduling constraints because they involve dependent costs (cf. §5.4.1): the cost of a feasible assignment is infinite if even one of the other tasks in the gang remains unscheduled.
However, the flow network does offer another way of implementing gang scheduling: we can leverage the capacities on arcs to force a group of tasks to schedule.

Figure 5.8a shows how a gang of $n$ tasks is expressed:

1. A new gang aggregator vertex (GA) is added and all tasks in the gang are connected to it. All other outgoing arcs from the tasks are removed.\(^6\)

2. The gang aggregator is connected to a single aggregator that connects to the prior destinations of the tasks’ outgoing arcs. These destinations are typically aggregators themselves (a task equivalence class aggregator, $TA_t$, or the cluster aggregator, X).

3. The lower bound on the capacity of the arc connecting the gang aggregator and the downstream aggregator is set to $n$, as is the upper bound.

Since the lower bound forces the new arc to have a flow of exactly $n$, it constrains the acceptable solutions to the minimum-cost, maximum-flow problem. Namely, only solutions in which all $n$ tasks are scheduled are possible – thus enforcing the gang-scheduling invariant.\(^7\)

With a small modification, this scheme can also support relaxed gang scheduling policies, e.g. scheduling at least $k$ out of $n$ tasks (as in the KMN scheduler [VPA*14]). To achieve this, Firmament sets the lower bound on the arc capacity of the gang aggregator’s outgoing arc to $k$ and leaves the upper bound at $n$ (Figure 5.8b).

There is one downside to this way of implementing gang scheduling. Due to the lower bound on capacity, gang-scheduled tasks must be scheduled, independent of their cost. While they nevertheless schedule in the best place available, this placement may come at the expense of preempting other tasks. Gang-scheduling must therefore be used with a degree of care: it can be limited to high-priority jobs, or gang-scheduled jobs can be subject to admission control before being added to the flow network.

### 5.4.4 Limitations

Firmament’s global perspective on the scheduling problem grants allows it to weigh many concerns against each other, and flexibly supports many scheduling policies. However, the formulation as a minimum-cost, maximum-flow optimisation over a flow network also restricts Firmament’s ability to directly express some policies:

**Combinatorial constraints.** Each unit of flow in the flow network is subject to the same costs, independent of its origin, and each decision to route flow is made independently of other

---

\(^6\)The removal of other arcs is not strictly required; however, they are guaranteed to receive no flow.

\(^7\)However, if the gang-scheduled tasks exceed the available resources, the solver fails to find a solution at all. However, gang scheduling is rare [SKA*13, §5.2] and the problem is not isolated to Firmament: shared-state schedulers do not guarantee progress when gang scheduling is supported, either.
concurrent decisions. As a result, dependent costs cannot be expressed using straightforward arcs: costs cannot be conditional.

**Global invariants.** Arc capacities cannot enforce invariants expressed in terms other than task counts; if an arc has capacity, *any* task that can reach it can send flow through it, even if the assignment violates a global invariant.

**Multi-dimensional capacities.** Capacities in the flow network are single-dimensional integers and each unit of flow is atomic. This makes multi-dimensional resource models difficult to model directly:

Both limitations can, however, be mitigated using two techniques (either on their own, or in combination):

1. **Reactive multi-round scheduling** expresses dependent costs by adapting the costs in the flow network in successive optimisation rounds in response to earlier decisions. Tasks with dependent constraints can only be scheduled one at a time, and must thus be added to the flow network in sequence (see §5.4.1).

2. **Admission control** only adds schedulable tasks to the flow network if suitable resources are available, and ensures that they can *only* route flow through suitable resources (often by having no arc to the cluster aggregator, $X$).

Combinatorial constraints (e.g. “no two tasks from this job can share a machine”) can generally be supported via reactive multi-round scheduling, multi-dimensional capacities (e.g. CPU, RAM, disk I/O bandwidth) can be respected via admission control, and global invariants (e.g. fair sharing of rack uplink bandwidth allocations) can typically be expressed via a combination of the two. Appendix C.4 explains these limitations and the solutions to them in more detail.

Another limitation is the performance of the minimum-cost, maximum-flow solver: its algorithmic complexity can hamper the scalability of the flow network approach. However, Firmament addresses this by *incrementally* optimising the flow network, and I discuss this in Section 5.6.3.

## 5.5 Cost models

Firmament supports a standardised interface for custom, configurable cost models. A *cost model* describes a scheduling policy by assigning concrete cost values to the different arcs in the flow network (see Appendix C.5 for the cost model API).

In the following, I describe three different cost models that I have implemented for Firmament. Section 6.3 later compares their performance on a range of workloads.

I first briefly introduce the original Quincy cost model (§5.5.1; details in Appendix C.3). I then explore the power of Firmament’s customisable cost models via three case studies:
1. a cost model based on the Whare-Map [MT13] system for exploiting the benefits of resource heterogeneity and avoiding co-location interference (§5.5.2);

2. the coordinated co-location (CoCo) cost model, which integrates the distributed cluster scheduler and local CPU affinity by scheduling tasks directly to hardware threads (§5.5.3); and

3. the Green cost model, an energy-aware scheduling approach which optimises task placement in a highly heterogeneous cluster with respect to performance constraints and live power monitoring data (§5.5.4).

Other cost models are, of course, also conceivable. In Section 5.5.5, I summarise and contrast the properties of the four cost models described, and Section 6.3.2 summarises how existing schedulers’ policies can be translated into Firmament cost models.

### 5.5.1 Quincy cost model

Quincy does not support customisable cost models, but instead assigns costs that express a specific trade-off between data locality, task wait time, and wasted work due to preemption of running tasks.

Figure 5.9 shows flow network structure and cost terms in Quincy. The cost to the unscheduled aggregator, $v^j_i$, is proportional to the task’s wait time, while the cost to the unschedule aggregator, $\alpha^j_i$, is set to a cost proportional to the data transfer required at the worst possible locality in the
cluster. Additionally, a preference arc to a rack aggregator \( R_l \) has cost \( \rho^j_{l,l} \) proportional to the worst-case data transfer within the rack, and an arc to a machine \( M_m \) has cost \( \gamma^j_{l,m} \) proportional to the data transfer required when scheduling on this machine.

In Appendix C.3, I explain the exact cost terms and how their values are determined. Firmament simply models the cluster aggregator (\( X \)) and the rack aggregators (\( R_l \)) as resource equivalence classes, and there are no task equivalence classes in the Quincy cost model.

### 5.5.2 Whare-Map cost model

The Whare-Map system by Mars and Tang avoids negative interference and exploits machine heterogeneity in WSCs [MT13]. In essence, Whare-Map builds a matrix of performance scores for each combination of a task, machine type and potentially interfering tasks. It then applies a stochastic hill-climbing approach to find good assignments.

Some of the high-level goals for Whare-Map and Firmament are similar: both try to avoid co-location interference, and both aim to improve utilisation by using heterogeneous WSC resources as efficiently as possible. In the following, I demonstrate that Firmament can express all of Whare-Map’s scoring policies via a cost model.

Whare-Map bases its notion of cost on performance scores attributed to a task in different environments. The scoring metric used by Mars and Tang is instructions-per-second (IPS), measured by hardware performance counters [MT13, §5.3].

Unlike other approaches that require a-priori profiling (e.g. Paragon [DK13] and Quasar [DK14]), Whare-Map can build its scoring matrix incrementally as tasks run and information is obtained. This yields no benefit for tasks that only run once, but can be useful in WSC environments where the majority of work is recurring [RTM+10; ZTH+13].

Whare-Map’s scoring information for each task type can be maintained at different levels of granularity. In Firmament, I use task and machine equivalence classes (§5.2.4) to aggregate information for similar tasks and machines.

Whare-Map has four scoring policies:

1. **Whare-C**, which is based on co-location interference only, but ignorant to heterogeneous machines;

2. **Whare-Cs**, which takes into account co-location interference for each specific machine type;

3. **Whare-M**, which uses machine type affinities, but does not consider co-location interference at all; and

---

\[ \text{The same work was previously described in Mars’s 2012 PhD thesis under the name SmartyMap [Mar12].} \]
4. **Whare-MCs**, which takes all of the above into account (machine-specific co-location interference and machine type affinity).

Mars and Tang found Whare-M and Whare-MCs to be most effective for their workloads, and I only consider these variants in the following. However, they subsume Whare-C and Whare-Cs as degenerate cases of Whare-MCs.

Figure 5.10 shows an example Firmament flow network for both Whare-Map cost models.

**Whare-M.** For Whare-M, the scoring function maps each task to its affinity for different machine types. This is implemented by linking each task to a set of machine aggregator vertices in the flow network, each of which represents one machine type (MA$_{c_m}$ for machine type $c_m$ in Figure 5.10).

The aggregator vertices are in turn connected to all machines of the type represented. For $m$ machines, $n$ machine types, and $t$ tasks, this approach requires $n$ vertices and $mt + m$ arcs. The number of distinct machine types in the WSC, $n$, is usually on the order of a few dozen (cf. §2.2.2). Thus, $t \gg m \gg n$, making this solution preferable over a naïve approach of each task having an arc to each machine, which requires $mt$ arcs but adds no more information.

To further reduce the number of arcs, it makes sense to add aggregators for tasks of the same equivalence class (TA$_{c_t}$ for EC $c_t$). For $u$ different task equivalence classes, using aggregators requires $n + n$ vertices and $tu + un + m$ arcs, with $t \gg m \gg u \geq n$. Since $tu \ll mt$, this approach scales better than using machine-type aggregators only.
### Cost Parameters in the Whare-Map Cost Models and Their Roles

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Edge</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v^j_i$</td>
<td>$T_{j,i} \rightarrow U_j$</td>
<td>Cost of leaving $T_{j,i}$ unscheduled.</td>
</tr>
<tr>
<td>$\alpha^j_i$</td>
<td>$T_{j,i} \rightarrow X$</td>
<td>Cost of scheduling in the worst possible location.</td>
</tr>
<tr>
<td>$\gamma_{i,m}$</td>
<td>$T_{j,i} \rightarrow M_m$</td>
<td>Cost of continuing to run on machine $M_m$.</td>
</tr>
<tr>
<td>$\Psi(c_t, c_m)$</td>
<td>$\text{TA}<em>{c_t} \rightarrow \text{MA}</em>{c_m}$</td>
<td>Cost of running task of type $c_t$ on a machine of type $c_m$.</td>
</tr>
<tr>
<td>$\Xi(c_t, L_m, c_m)$</td>
<td>$\text{TA}_{c_t} \rightarrow M_m$</td>
<td>Cost of running task of type $c_t$ with co-located tasks in $L_m$.</td>
</tr>
</tbody>
</table>

Table 5.4: Cost parameters in the Whare-Map cost models and their roles.

I denote the score for a task category $c_t$ on a machine of type $c_m$ by the function $\Psi(c_t, c_m)$. In the flow network, $\Psi(c_t, c_m)$ is associated with an arc between a task aggregator ($\text{TA}_{c_t}$) and a machine type aggregator ($\text{MA}_{c_m}$), as shown in Figure 5.10.

Since Whare-M relies only on information about tasks’ performance on different machine types, it requires only limited profiling information. However, the collected profiling information for each machine type $m$ can be noisy, since tasks’ performance variation due to the machine type may be diluted by co-location interference from other tasks.\(^9\)

### Extension to Whare-MCs

The Whare-MCs policy takes interference from co-located tasks into account. It extends the scoring function with a component dependent on the tasks that already run on a candidate machine. I refer to the set of equivalence classes of co-located tasks as $L_m$, and express the co-location score for a task in equivalence class $c_t$ via the function $\Xi(c_t, L_m, c_m)$.

This extension requires additional arcs from $\text{TA}_{c_t}$ to individual machines, since the co-location interference is a property of the workload mix scheduled on a specific machine. I therefore add arcs from each task aggregator to machines with preferable co-location conditions. The number of outgoing arcs from task aggregator $\text{TA}_{c_t}$ is equal to the minimum of the number of incoming arcs into $\text{TA}_{c_t}$ and the number of machines with suitable resources.

The machine type aggregators and the arcs connecting the task type aggregators to them (at $\Psi(c_t, c_m)$ cost) are still present in Whare-MCs. However, since the profiling data for $\Psi(c_t, c_m)$ is less specific as it averages over all profiled co-locations, the lowest cost values of $\Xi(c_t, L_m, c_m)$ are likely better, and the highest values worse, than $\Psi(c_t, c_m)$:

\[
\min_{L_m}(\Xi(c_t, L_m, c_m)) \leq \Psi(c_t, c_m) \leq \max_{L_m}(\Xi(c_t, L_m, c_m))
\]

As a result, it is more attractive for tasks to schedule via the co-location-aware arcs than via those pointing to the machine type aggregators. If insufficiently many good co-location options are available, however, tasks still schedule on the best machine type available in preference to scheduling via the cluster aggregator vertex, $X$.

---

\(^9\)Profiling tasks in isolation is impractical as it reduces utilisation; biasing the score in favour of those from otherwise idle environments might work, but such environments are rare in practice.
CHAPTER 5. FIRMAMENT: A WSC SCHEDULER

Cost values. Since the flow optimisation requires costs to increase as placements become less desirable, the IPS metric used in Whare-Map, in which greater values are better, must be inverted. Consequently, I convert IPS to seconds-per-instruction by dividing the task runtime by the instruction count obtained from performance counters (cf. §5.3.3). As the available solvers require integral costs, I further normalise this metric to picoseconds-per-instruction (psPI).

I have already explained the definitions of \( \Psi(c_t, c_m) \) and \( \Xi(c_t, L_m, c_m) \) and the arcs on which they are applied. The costs assigned to other arcs are similar to those in the Quincy cost model, albeit re-written in terms of the Whare-Map scores instead of data transfer costs:

- \( v^j_i \) is the cost of leaving the \( i^{th} \) task in job \( j \) unscheduled. It is proportional to the wait time \( v^j_i \), and lower-bounded by the average Whare-Map score, with \( T_{j,i} \in c_t \):

\[
v^j_i = \begin{cases} 
\max(v^j_i, \Psi(c_t, c_m)) & \text{for Whare-M} \\
\max(v^j_i, \Xi(c_t, L_m, c_m)) & \text{for Whare-MCs}
\end{cases}
\]

- \( \alpha^j_i \) is the cost of scheduling via the cluster aggregator and is set to the cost of scheduling in the worst possible location. In Whare-Map, this is the machine equivalence class least suited towards task \( T_{j,i} \) (Whare-M) or, the machine where the task will experience the worst possible interference (Whare-MCs). In other words, for \( T_{j,i} \in c_t \):

\[
\alpha^j_i = \begin{cases} 
\max_{c_m}(\Psi(c_t, c_m)) & \text{for Whare-M} \\
\max_{L_m,c_m}(\Xi(c_t, L_m, c_m), \Psi(c_t, c_m)) & \text{for Whare-MCs}
\end{cases}
\]

- \( \gamma^j_i \) is the cost of running \( T_{j,i} \) on a particular machine, or continuing to run there if already scheduled. As in Quincy, the cost of continuing to run in a location is discounted by the total cumulative runtime of \( T_{j,i} \).

Possible extensions. The Whare-Map cost models, as described, implement the behaviour of the published Whare-Map system. The key difference is that, instead of using the stochastic hill-climbing approach, my implementation is based on the minimum-cost, maximum-flow optimisation. This results in higher-quality assignments, since they are no longer approximate.

There are a number of conceivable improvements to this approach:

Model machine load. The co-location-aware arcs (TA\(_{c_m} \rightarrow M_m\)) already implicitly reflect the load on their target machines (as higher load likely leads to more interference and thus a higher cost). However, the arcs from the machine type aggregators to the machines (MA\(_{c_m} \rightarrow M_m\)) have a zero cost.

\( ^{10} \)For modern gigahertz-clocked CPUs, this value ranges from approximately 250 (IPC = 1, e.g. simple integer arithmetic on a 4 GHz CPU) to 100,000 (IPC = \( \frac{1}{400} \), e.g. remote DRAM-bound work).
In order to bias scheduling in preference of machines with lower load, the arcs from machine type aggregators to machines could be assigned costs proportional to the machines’ load.

Express co-location at CPU granularity. Whare-Map considers co-location sets on a machine granularity ($L_m$). However, Firmament routinely extracts the micro-architectural topology of each machine ($\S 5.3.2$).

One could imagine extending the Whare-Map cost model described to manage CPUs rather than machines: instead of scoring co-location set $L_m$ in $\Xi(c_t, L_m, c_m)$, the score would refer to the set of shared machine resources at the arc’s target (e.g. the tasks running on CPU cores sharing caches with the targeted one).

I do not, however, explore these improvements in the context of Whare-Map here. Instead, I next describe another cost model developed for Firmament: the coordinated co-location model, which incorporates these ideas.

5.5.3 Coordinated co-location (CoCo) cost model

The Whare-Map cost model described in the previous section addresses co-location interference and machine heterogeneity. However, it comes with a number of limitations:

1. It does not provide for scheduling dimensions other than machine type affinity and co-location interference.
2. It only supports a very coarse-grained, machine-level notion of co-location interference.
3. It does not model tasks’ resource demands and the available resources on machines; machine load is only implicitly considered as part of the co-location scores.
4. It does not afford the flexibility to assign a higher weight to some scheduling dimensions than to others.

To address these limitations, I developed the Coordinated Co-location (CoCo) cost model for Firmament.

The key insight in the CoCo cost model is that costs can be modelled as multi-dimensional cost vectors, and that this, in combination with task equivalence class aggregators ($\S 5.2.4$), offers an efficient way of expressing tasks’ multi-dimensional resource requirements.

In summary, the CoCo cost model offers the following properties:

1. **Strict priorities**: higher priority tasks are always scheduled in preference to lower priority ones.
2. **Strict resource fit**: tasks only schedule on machines that have sufficient available resources to accommodate them.

3. **Balanced load**: tasks preferentially schedule on lower-loaded machines, i.e. a task does not schedule on a highly-loaded machine when an otherwise equivalent lower-loaded one is available.

4. **Low average wait time**: the longer a task waits to schedule, the more likely it is to be assigned a suboptimal placement instead of waiting further.

CoCo achieves these properties by a combination of admission control, smart cost assignment, and efficient updates to the flow network, as I explain in the following.

**Admission control.** To meet the strict resource fit property, CoCo must match tasks to resources such that no task can schedule on a machine with insufficient available resources. This is more complex than it might seem.

First of all, if a task does not fit on all machines, we must remove the arc to the cluster aggregator, X. A naive approach might then add arcs to all suitable machines – however, this requires...
$O(t \times m)$ task-specific arcs. Consider the case of a task that fits on all machines but one: it would end up with $m - 1$ preference arcs. We could cap the number of preference arcs per task, but would lose optimality as a consequence, since the discarded arcs restrict the possible solutions.

Instead, I change the structure of the flow network to accommodate CoCo (see Figure 5.11). All tasks already have arcs to task aggregators representing their task equivalence classes (e.g. TA$_{c_0}$ and TA$_{c_1}$). From these aggregators, CoCo adds outgoing arcs to maximally aggregated subtrees of the resource topology. In other words, each arc points to the largest possible aggregate of resources on which tasks in the equivalence class definitely fit. Several such aggregates may exist, and each may include multiple machines – consider, for example, a rack in which a task fits on all machines (e.g. TA$_{c_1}$ on R$_1$).

For small tasks that fit in many places, these aggregates are large and require few arcs, while large and “picky” tasks receive a small set of highly specific arcs. In addition, CoCo caps the number of outgoing arcs at each task aggregator to the $n$ cheapest ones, where $n$ is the number of incoming task arcs: no more than $n$ tasks can schedule via this aggregator anyway, so this does not compromise optimality.

**Efficient admission control.** The above approach has an attractive invariant: no path from a task to the sink crosses any machine where the task is unable to fit. To maintain the correct set of arcs, however, the scheduler must on each iteration (and for each task equivalence class) reconsider all machines whose resource load has changed. This computation can be simplified by tracking some state in the flow network: for each resource vertex, CoCo maintains current minimum and maximum available resources across its children. This is implemented as a simple breadth-first traversal in Firmament (see Listing 5.1), which discovers suitable subtrees in worst-case $C \times O(N)$ for $C$ task equivalence classes and $N$ leaves in the flow network.

An even faster approach would use a pre-constructed two-dimensional segment tree for worst-case $C \times O(\log^2 N)$ traversal time; constructing the segment tree has a one-off cost of $O(N \log N)$, and each update of a machine in response to changes costs at most $O(\log^2 N)$.

**Resource overcommit.** Whether a task from an equivalence class “fits” under a resource aggregate – i.e., the result of TaskFitsUnderResourceAggregate() in Listing 5.1 – depends on three factors: (i) the resource request for tasks in this equivalence class; (ii) the current reserved resources and actual load of the resources in the aggregate; and (iii) the workload type represented by the task equivalence class:

- For service jobs, an arc is added unless the task’s resource request exceeds the reserved machine resources in any dimension. This is very conservative: actual resource usage is typically far lower than the reservation – e.g. by 30–40% at Google [RTG+12, §5.1].
CHAPTER 5. FIRMAMENT: A WSC SCHEDULER

147

vector<ResourceID_t>* CocoCostModel::GetMaximallyAggregatedSubtrees(
    const EqvClass_t task_ec, const ResourceTopologyNode* start_res) {
    vector<ResourceID_t>* subtree_heads = new vector<ResourceID_t>();
    queue<const ResourceTopologyNode*> to_visit;
    to_visit.push(start_res);

    // Breadth-first traversal with early termination conditions
    while (!to_visit.empty()) {
        ResourceTopologyNode* res_node = to_visit.front();
        to_visit.pop();

        // Check task fit
        TaskFitIndication_t task_fit =
            TaskFitsUnderResourceAggregate(task_ec, res_node);
        if (task_fit == TASK_ALWAYS_FITS) {
            // We fit under all subordinate resources, so put an arc here and
            // stop exploring the subtree.
            subtree_heads->push_back(res_node->uuid());
            continue;
        } else if (task_fit == TASK_NEVER_FITS) {
            // We don’t fit into *any* subordinate resources, so give up
            // on this subtree.
            continue;
        }

        // We fit at least in some dimensions, so we may have suitable
        // resources here -- let’s continue exploring the subtree.
        for (auto rtnd_iter = res_node->children().pointer_begin();
             rtnd_iter != res_node->children().pointer_end();
             ++rtnd_iter) {
            to_visit.push(*rtnd_iter);
        }
    }
    return subtree_heads;
}

Listing 5.1: Simplified excerpt of the CoCo resource fitting code: a breadth-first traversal of the resource topology yields maximally aggregated subtrees under which tasks in equivalence class task_ec definitely fit.

• For batch jobs, an arc is added unless the task’s resource request exceeds the used machine resources in any dimension. This allows spare resources on machines to be used by best-effort batch tasks. However, if the service tasks’ usage increases, batch tasks are killed to free up resources.

If a task fits under an aggregate, an arc between TA_t and the resource aggregate is added, and its cost is set as described in the following.
Expressing multi-dimensional costs. Each cost in CoCo is expressed internally as an eight-dimensional vector:

\[ a(v, w) = A = \begin{bmatrix} \text{Priority} \\ \text{CPU cost} \\ \text{Memory cost} \\ \text{Network cost} \\ \text{Disk I/O cost} \\ \text{Machine type cost} \\ \text{Interference cost} \\ \text{Data locality cost} \end{bmatrix} \]

The vectors are flattened to an integral cost (required by the solver) via a weighted inner product, i.e. \( a(v, w) = w_0A_0 + \cdots + w_7A_7 \).

The cost values in the different dimensions depend on the type of arc that the cost vector is associated with:

1. On task-specific arcs (any outgoing arc from a task vertex, i.e. \( T_{j,i} \rightarrow * \)), the priority dimension is set to the task’s priority, while the CPU, memory, network, and I/O dimensions indicate the task’s resource request, i.e. the resources it needs to run. The remaining dimensions (machine type, interference, and data locality cost) are set to zero.

2. On resource-specific arcs, CPU, memory, network, and I/O costs are set to an indication of the resource’s load in each dimension. For higher load, the cost increases. Machine type and interference costs are set according to the resource’s machine type and any existing co-running tasks. Priority and data locality costs are set to zero.

3. On an unscheduled arc (\( T_{j,i} \rightarrow U_j \)), the costs in the resource capacity dimensions are set to the maximum value, while the cost in the interference and machine type dimension is set to one.

The values in each dimension are normalised into the range \([0, \Omega)\), where \( \Omega \) is a fixed maximum cost value. For resource requests and load, the value is first normalised to the largest quantity of the resource available in any machine, and then multiplied by \( \Omega \).

Co-location interference. Expressing the cost of co-location interference requires a different approach to resource requirements, since interference cannot be quantified as a fraction of a total capacity. Moreover, the interference experienced by a task increases as a function of sharing resources with more (or more aggressive) neighbours.

CoCo relies on interference classes to model the interaction of tasks when sharing resources. In the Firmament prototype, tasks are currently manually classified by the user, but automatic classification is possible.\(^{11}\)

\(^{11}\)For example, the “animal” taxonomy proposed by Xie and Loh for cache partitioning [XL08] can be used to automatically classify tasks based on the profiling data collected (§5.3.3).
Each leaf node of the resource topology supplies a penalty score for each interference class, with low penalty scores corresponding to strong affinity for a neighbour of that class. For example, “devil” tasks (strongly interfering) have a low affinity – and a high penalty score – for interference-sensitive neighbours (“rabbits” and “sheep”), but a medium affinity for further “devils”.

Each leaf’s penalty score is used as its resource-specific cost, and is propagated upwards through the resource topology in two ways:

1. Each vertex stores the normalised cumulative penalty score of its children for each class. This allows CoCo to propagate class-specific interference scores for resource aggregates; as a result, different task equivalence classes see different interference scores when they add arcs to an aggregate.

2. Each arc within the resource topology carries an integral cost proportional to its childrens’ penalty scores, with the impact of an individual leaf’s score decaying as it propagates upwards. The cost for an arc that does not directly connect a leaf is given by:

\[ cost(u, w) = e^{-i/t} \times \frac{\sum_{v \in \text{children}(w)} cost(w, v)}{\text{num. children of } w}, \]

where \( i \) is the number of idle leaf resources below the current one, and \( t \) is the total number of subordinate resources. The super-linear \( e^{-i/t} \) scale factor ensures that dense clusters of idle resources are preferred to sparse collections of idle resources.

The arc cost enables CoCo to steer an incoming task’s flow towards the least interfering location within a resource aggregate.

The cost specific to the task equivalence class (on the \( \text{TA}_c \rightarrow \text{resource aggregate} \) arc) emphasises the interference that a task must expect within an aggregate. The costs on arcs within the resource aggregate, by contrast, are proportional to load and interference below each resource. Their combination has a useful effect: the cheapest path through a resource aggregate corresponds to the most preferable task placement.

**Priorities.** The priority dimension in the cost vector is inversely proportional to a task’s priority: the lower the value, the higher is the priority of the task. In order to maintain strict priorities, the cost model must ensure that the priority dimension always dominates. This is achieved by scaling the priority component by \( d \times \Omega \), where \( d \) is the number of dimensions (eight) and adding this value to the cost when flattening a cost vector.

**Preemption.** The coordinated co-location model supports preemption, which may come about in two flavours:
1. **Priority preemption** occurs because a task of higher priority (i.e., lower cost in the priority dimension) was placed on a resource.

2. **Optimisation preemption** is a result of the flow network optimisation deciding to stop a task in order to run another one.

Service jobs cannot be preempted by batch jobs, since batch job’s lower priority always dominates their cost.

A task may also *migrate* if it is preempted, but is immediately re-scheduled on another resource. Since CoCo represents each CPU core in the flow network, it can migrate a task from one CPU core to another one on the same machine at low cost.

**Scheduling in waves.** Firmament’s flow network optimisation approach allows many scheduling assignments to be made at the same time. This is problematic for CoCo, because the solver may place tasks in such a way that their resource requirements *conflict*. For example, $M_1$ in Figure 5.11 may have four idle CPU cores, but may only have sufficient I/O capacity for *either* a task from $TA_{c_0}$ or a task from $TA_{c_1}$. With two incoming arcs, however, nothing stops both $T_{0,0}$ and $T_{1,1}$ from scheduling there.

To prevent this situation from arising, CoCo introduces two restrictions:

1. **Selective arc creation:** arcs to resource aggregates are only added until the cumulative resource requirements for the possible incoming tasks exceed the available resource capacity.

2. **Unit capacities:** the capacity on all arcs to resource aggregates, within the resource topology and from its leaves to the sink is set to one.$^{12}$

As a result, tasks schedule in “waves”: each task aggregator can only place one task in each resource aggregate per scheduling iteration. This can slow down the placement of large jobs’ tasks, but avoids unwanted resource overcommit and co-location interference, as costs are updated after each placement.

**Summary.** The CoCo cost model is the most elaborate cost model developed for Firmament, and makes full use of its capabilities, modelling both tasks’ multi-dimensional resource requirements and their mutual interaction when co-located.

In Section 6.3.1.1, I evaluate CoCo on a 28-machine cluster with a heterogeneous workload, and find that it significantly reduces variance in task runtime due to co-location interference.

---

$^{12}$This implies that each leaf can only run one task, which is the case in CoCo. However, multiple tasks per leaf can be supported by adding “virtual” per-task leaves if required.
5.5.4 Green cost model

Firmament can also support cost models based on more unusual inputs, such as power consumption in the Green cost model.

For this cost model, I extended Firmament to collect live power usage statistics, and to assign tasks such that the overall energy efficiency of a heterogeneous WSC (§2.2.4) is maximised.

With the Green cost model, Firmament runs as a closed-loop feedback scheduler, placing tasks in accordance with current power measurements. As a result, it offers:

1. Dynamic, energy-aware provisioning and migration of service tasks as a function of current load. The best available combination of machines is chosen such that SLAs can be met at current load, and energy efficiency is maximised.

2. Energy-aware scheduling for batch jobs, such that slower, but more energy-efficient machines are used to run batch jobs which are neither time-critical nor have a sufficiently proximate completion deadline. If the deadline is close, or the current progress indicates that a task will fail to meet it in its current location, it is automatically migrated.

Figure 5.12 gives an overview of the Green cost model’s operation. In this example, all machines (bottom, grey) run Firmament coordinators, with a single master coordinator as their parent.

---

The energy-aware scheduling case study and necessary extensions to Firmament were carried out by Gustaf Helgesson for his MPhil ACS research project under my supervision [Hel14].
I assume that client requests are handled by application-specific request load balancers that redirect them to service job tasks on different machines. This is a reasonable assumption: multi-layer load-balancing is a common setup in WSC data centres.

Job submission. Batch jobs are submitted directly to the master coordinator and their tasks are scheduled and run to completion. Once the final task in a job exits, the job completes. Service jobs are submitted in the same way, but typically run indefinitely. Their number of tasks is automatically scaled according to the current load seen by the application-specific load balancer. If it is necessary to commission additional tasks in order to meet the relevant service level objective (SLO) – for example, a 99\textsuperscript{th}-\%ile request latency for a given throughput – the load balancer launches additional tasks in the service job. Conversely, if surplus capacity is available, the load balancer may terminate running tasks.

Energy statistics collection. All machines in the WSC are continuously monitored for their full-system power consumption, measured at the power outlet. The power samples obtained are forwarded to the master coordinator. The reported power consumption includes energy used by CPU, DRAM, peripheral and storage devices, as well as cooling and power supply losses. When multiple tasks share a machine, I divide the overall power consumed between them according to the number of CPU cores utilised by each task. While this is somewhat crude, it works well: in exploratory experiments, I found that power consumption is highly linear in the number of cores utilised.

Energy consumption information is recorded in a task’s profile in the coordinator knowledge base. Samples are categorised by the relevant machine equivalence class. Over time, this allows Firmament to build a statistical profile of the task’s energy cost on this platform.

Costs. All costs assigned are in the range $[0, \Omega]$, with $\Omega$ being a large, fixed constant. Tasks always have costs on their arc to an unscheduled aggregator ($T_{j,i} \rightarrow U_j$), expressed by $\Gamma$ for service tasks and $\gamma$ for batch tasks. They also have arcs to aggregator vertices for specific machine type equivalence classes ($\text{MA}_{c_m}$ in Figure 5.13), which carry a cost for running on a machine of this class (service: $\rho(c_m)$, batch: $\delta(c_m)$). Once a task is running, it has a direct arc to its machine $M_{k}$ with cost $\phi(k)$ (service) or $\theta(k)$ (batch) in addition to other arcs.

A new service task $T_{j,i}$ is connected to aggregators for each machine equivalence class $c_m$ ($\text{MA}_{c_m}$) at a cost $\rho(k)$ proportional to the per-request energy cost on these machines. The cost of leaving the task unscheduled ($\Gamma$) is set to maximum value $\Omega$. Service tasks are thus always scheduled most expeditiously, and never preempted (although they may migrate). Once a service

\footnote{Power consumption attributable to individual tasks could also be measured directly, using, for example, the RAPL interface available in recent Intel CPUs [HDV*12].}
Figure 5.13: Firmament flow network with the Green cost model. One batch and one service job are shown, with running, new service, and new batch tasks. Running tasks also have arcs to machine aggregators corresponding to migration options, which are not shown here.

A task is running, the cost for maintaining this assignment in future scheduling iterations ($\phi(k)$) is equal to $\rho(k)$ with a discount applied to control hysteresis.

Unlike service tasks, the start times of batch jobs are flexible, subject only to their deadlines. A batch task is considered schedulable on a machine if that machine is expected to be able to meet its completion deadline. The maximum cost for a batch task is $\omega (\ll \Omega)$. Costs are assigned as follows:

**Batch task $T_{j,i}$ schedulable on $M_k$ of type $c_m$.** $\delta(c_m)$, the cost from the task to aggregator $MA_c$ is proportional to the energy cost of running $T_{j,i}$ on a machine in class $c_m$. When $T_{j,i}$ is running on $M_k$, the cost $\theta(k)$ is inversely proportional to the task’s completion percentage. The cost of leaving the task unscheduled, $\gamma$, is set slightly above the cheapest machine’s cost.

**Batch task $T_{j,i}$ schedulable on some machines, but not $M_k$.** $T_{j,i}$ and $M_k$ are connected via an “unschedulable” machine type aggregator. The cost from this aggregator to $M_k$ is $\omega$, as is the cost to $X$; only arcs to preferred machine type aggregates can thus be used.

**Batch task $T_{j,i}$ only schedulable on $\leq \varepsilon$ machines.** The task is considered high priority for scheduling. $\gamma$ is set to a high multiplier of the best machine’s cost, giving the task precedence over other tasks.

**Batch task $T_{j,i}$ not schedulable on any machine.** While we expect to miss the deadline, Firmament still attempts to run the task as soon as possible. Edges to the best machines for
\( T_{j,i} \) are assigned cost \( \delta(c_m) \); all other machines are connected at cost \( \omega \) via the unschedulable aggregator.

As in other cost models, tasks may occasionally preempt others. The likelihood of a task being preempted is proportional to its suitability for the resource it is running on (if it is low, a better task is likely to appear) and how close to predicted completion it is (the further the task proceeds, the less likely it is to be preempted). A further refinement of the model would introduce discounted arcs to nearby locations, reflecting the cheaper migration due to, e.g. fast restart on local or checkpointed state.

**Discussion.** The Green cost model is, first and foremost, a demonstration of Firmament’s flexibility. It is a stand-alone cost model, but could be extended with support for dimensions other than energy, such as co-location interference, machine load and data locality, as in the Whare-Map and CoCo cost models.

In Section 6.3.1.2, I show that the energy-aware cost model saves up to 45% of the above-idle energy consumed in a test deployment. Even without dynamic power management, this effects an 18% reduction in overall energy consumption.

### 5.5.5 Summary and other cost models

Firmament can support many different cost models. I have outlined four, which Table 5.5 summarises. The CoCo cost model supports the most extensive feature set, but is also the most complex cost model.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Quincy ($§5.5.1$)</th>
<th>Whare-Map ($§5.5.2$)</th>
<th>CoCo ($§5.5.3$)</th>
<th>Green ($§5.5.4$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preemption</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Data locality</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Co-location interference</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Multi-dimensional load</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Per-core scheduling</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Explicit priorities</td>
<td>x</td>
<td>x</td>
<td>✓</td>
<td>x</td>
</tr>
<tr>
<td>Energy-awareness</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 5.5:** Cost models supported in Firmament and their features.

These cost models do not exhaust the possibilities of Firmament: many other recent cluster schedulers can also be modelled. For example, Apollo [BEL+14] uses a combination of initialisation (data fetch) time, expected scheduling delay (according to a wait-time matrix), and expected runtime, and Section 6.3.2 outlines how it can be implemented atop Firmament.

There are approaches that cannot easily be expressed in Firmament, however. For example, tetrisch [TZK+13] can express complex constraints ($§5.4.4$) as algebraic expressions that
do not map to Firmament cost models. Likewise, as I mentioned before, some global multi-
dimensional max-min fairness invariants cannot be expressed (§5.4.2).

5.6 Scalability

Firmament’s suitability for large-scale WSCs depends on the efficacy and performance of the
underlying flow network optimisation. The minimum-cost, maximum-flow optimisation prob-
lem is a computationally intensive problem, and solving it for large graphs requires non-trivial
optimisation algorithms. Appendix C.1.1 offers a reference explanation of the problem; this
section assumes basic familiarity with its terms.

In the following, I survey minimum-cost, maximum-flow algorithms and discuss their known
complexity bounds (§5.6.1). I then show that the flow networks generated for the scheduling
problem in practice see much better runtimes than the worst-case bounds suggest, but still scale
poorly to very large clusters (§5.6.2). However, it turns out that scalability can be much im-
proved by Firmament relying on solvers which incrementally solve the problem if only minor
changes to the flow network have occurred (§5.6.3).

In Section 6.3.3, I show that this incremental solving technique allows Firmament to scale to
very large clusters at sub-second decision times.

5.6.1 Algorithms and solvers

Naïve algorithms for solving the minimum-cost, maximum-flow problem have exponential
complexity. However, a number of algorithms with polynomial and strongly polynomial com-
plexity have been developed. Goldberg [Gol87, p. 41] and Orlin [Orl93] provide concise
overviews of the state-of-the-art algorithms as of the early 1990s. While a detailed discussion
of all recent approaches to the minimum-cost, maximum-flow problem is beyond the scope of
this dissertation, I give a high-level overview of recent algorithms in the following.15

Table 5.6 lists recent key algorithms and their worst-case time complexities. In practice, the
best algorithm depends both on the flow network structure and the costs within it.

**Cycle cancelling.** Originally due to Klein [Kle67] and of exponential complexity, cycle can-
celling algorithms are the simplest minimum-cost, maximum-flow algorithms. They start from
a feasible flow (obtained via a maximum-flow computation), and using the residual network,
they iteratively cancel negative cost cycles by sending flow along them. Cancelling these cy-
cles removes arcs from the residual network; once it is depleted, the optimal solution has been
found. Goldberg and Tarjan devised a strongly polynomial minimum-mean cycle cancelling

---

15A more extensive discussion of the algorithms can be found in Section 1.3 and Sections 3.2–3.5 of Adam
Gleave’s Cambridge Part II dissertation, written under my supervision [Gle15].
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Worst-case complexity</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum mean-cost cycles</td>
<td>$O(V^2 \text{polylog}(E))$</td>
<td>[Tar85]</td>
</tr>
<tr>
<td>(Tardos, 1985)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cycle cancelling</td>
<td>$O(V^2E^3\log(V))$</td>
<td>[GT89]</td>
</tr>
<tr>
<td>(Goldberg and Tarjan, 1989)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Successive approximation</td>
<td>$O(VE\log(VC)\log(V^2/E))$</td>
<td>[GT90]</td>
</tr>
<tr>
<td>(Goldberg and Tarjan, 1990)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Premultiplier network simplex</td>
<td>$O(\min(VE\log(VC),VE^2\log(V)))$</td>
<td>[Orl97]</td>
</tr>
<tr>
<td>(Orlin, 1997)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scaling push-relabel</td>
<td>$O(VE\min(\log(VC),E\log(V))\log(V^2/E))$</td>
<td>[Gol97]</td>
</tr>
<tr>
<td>(Goldberg, 1997)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5.6: Worst-case time complexities of different algorithms for the minimum-cost, maximum-flow problem. $V$ is the number of vertices, $E$ the number of arcs, and $C$ the maximum cost.

The algorithm [GT89], but solvers based on it are not competitive for Firmament’s flow networks in practice.

**Network simplex.** Like cycle cancelling, network simplex is a class of primal algorithms for minimum-cost, maximum-flow. Orlin’s premultiplier network simplex variant [Orl97] achieves a polynomial worst-case bound, and the MCFZIB network simplex solver [Löb96] is competitive with cost-scaling algorithms for dense graphs, but falls short on large, sparse graphs similar to Firmament’s flow networks [FM06, pp. 13, 17].

**Cost-scaling push-relabel.** The cost-scaling family of minimum-cost, maximum-flow algorithms is based on work by Goldberg, Kharitonov, and Tarjan [GT90; GK93; Gol97]. Cost-scaling maintains a feasible flow with a bounded deviation from the minimum-cost solution ($\epsilon$-optimality), and successively refines the solution. Refining involves pushing flow from vertices with excess to their neighbours, and relabelling vertices with new prices once no more flow can be pushed. The performance of cost-scaling is much improved by heuristics that reduce the operations required on each iteration. Appendix C.1.2 explains the algorithm in more detail. In practice, cost scaling is the fastest algorithm for solving Firmament’s flow networks when running from scratch.

**Relaxation.** The relaxation algorithm (not listed in Table 5.6) is a worst-case exponential algorithm derived from successive shortest path algorithms, which based on Lagrangian relaxation [AMO93, ch. 16] applied to the primal version of the minimum-cost, maximum-flow problem. It is not competitive with cost-scaling when running from scratch, but achieves excellent performance when incrementally re-optimising a flow network (§5.6.3)
5.6.2 Scalability of the scheduling problem

The time complexity of the minimum-cost, maximum-flow optimisation is proportional to the size of the flow network generated by Firmament, which in turn is proportional to both workload and cluster size.

Quincy originally targeted clusters of hundreds of machines: the authors evaluated it on a 243-node cluster and found the optimisation runtime to be a few milliseconds [IPC+09, §6]. They also investigated its performance in a simulated 2,500-machine deployment and found that the optimisation still completes in “a little over a second” [IPC+09, §6.5]. Industrial WSCs, however, often have tens of thousands of machines running thousands of concurrent jobs.

In an experiment similar to the Quincy scale-up simulation, using the cost-scaling cs2 solver,\(^\text{16}\) I found that the solver runtime is linearly proportional to the number of arcs in the flow network, which is far better than the theoretical worst case of \(O(VE \min(\log(VC), E \log(V)) \log(V^2))\).

Nevertheless, Firmament has a noticeable overhead at WSC scale: upwards of 10,000 machines, scheduling with cs2 takes over 10 seconds. At a scale of 30,000 eight-core machines running about 400,000 tasks – about twice the size of a 2011 Google workload [RTG+12] – the solver takes 90 seconds to complete.

There are several ways in which this limited scalability might be improved upon:

1. **Partition the problem** by having multiple Firmament coordinators arranged in a tree (§5.3.1), each responsible for scheduling a smaller subset of the overall WSC (e.g. a few racks each). While efficient, a partitioned approach loses the global optimisation property that Firmament offers, as each of the subordinate schedulers can only see, and schedule, a part of the cluster.

2. Use **approximate solutions**, rather than running the minimum-cost, maximum-flow optimisation all the way to the end. Since Firmament routinely works with imperfect data, it may not be necessary to find the optimal solution – a task assignment that is “close enough” to the optimal solution may often be sufficient.

3. Compute an **incremental solution** by re-using the prior solution and solver state. In a large WSC, only a comparatively small fraction of tasks and machines experiences a state change in between scheduler iterations (even if they take tens of seconds). Much of the work from a prior iteration can therefore be reused, which might speed up the solver. One appealing property of this approach is that the number of changes shrinks as the solver completes faster – potentially allowing the solver to complete even sooner.

An investigation into approximate solutions showed that this approach fails to offer meaningful improvements for the problems generated by Firmament, since optimality is not reached

\(^{16}\text{cs2 was freely available for academic research and evaluation at http://www.igsystems.com until late 2014; it is now on Github at https://github.com/iveney/cs2.}\)
gradually, but rather in leaps [Gle15, §4.4]. However, scalability can be much improved by incrementally solving the problem, and I discuss this approach in the next section.

5.6.3 Incremental minimum-cost, maximum-flow optimisation

By default, minimum-cost, maximum-flow solvers expect to be given a complete flow network and solve it from scratch. When scheduling tasks on a large cluster, however, the number of changes to the network between runs of the solver is usually insignificant compared to the size of the cluster. Instead of running the cost solver from scratch each time, it can speed up significantly by maintaining state across runs and starting from a previous solution.

Firmament collects relevant events (e.g. task arrivals, machine failures, etc.) while the solver runs, and applies them to the flow network before running the solver again. The possible situations to handle reduce to three types of flow network changes:

1. **Excess is created** at a vertex. This happens when a “downstream” vertex, or an arc on a path that previously carried flow, is removed (e.g. due to a machine failure), or when more supply is added (e.g. due to the arrival of another task).

2. **Capacity is added** to an arc. This can either happen because a new leaf resource has appeared (e.g. a new machine being added), or because a new arc connects two previously disjoint vertices (e.g. a new arc between aggregators).

3. An arc’s **cost is changed**. Typically, this happens because the load on a resource has changed, a task has waited or run for longer, or the relative goodness of a scheduling assignment changed due to other assignments. This is the most common change.

Arc capacity can also be reduced without leading to excess (e.g. if an idle machine fails), but this has no impact on the result unless the cost changes: if the solver previously did not route flow through the edge, it will not do so after a capacity reduction either.

Firmament supports two ways of optimising the flow network incrementally:

1. An incremental version of Goldberg’s cost-scaling push-relabel algorithm [Gol97] using the **flowlessly** solver. This solver first applies the “global price update” heuristic on all vertices to adjust their prices and then re-runs the cost-scaling algorithm. As most vertices’ prices are already correct or close to correct, the number of “push” and “relabel” operations is greatly reduced.

2. A hybrid approach that combines cost-scaling push-relabel with the relaxation algorithm by Bertsekas and Tseng [BT88a]. The relaxation algorithm is slower than cost-scaling at

---

17The **flowlessly** solver was implemented by Ionel Gog and includes both a from-scratch and an incremental cost-scaling push-relabel algorithm implementation.
solving the problem from scratch, but much faster when re-optimizing a network incrementally. Firmament supports a modified version of the RELAX IV solver [BT88b; BT94] that incrementally re-optimises the relaxation problem.\(^{18}\)

In practice, the incremental cost-scaling approach only yields about a $2 \times$ improvement, while the incremental relaxation approach achieves much higher speedups (up to $14 \times$), since the algorithm does not need to continuously restore global invariants.

The incremental approach results in tangible solver runtime reductions: in Section 6.3, I show that the incremental solver runs in sub-second time on a Google-scale WSC.

### 5.7 Summary

In this chapter, I contrasted cluster scheduling for WSCs with the far older domain of single machine CPU scheduling, and illustrated how my Firmament cluster scheduler draws on both (§5.1).

Firmament is unusual because it maps the scheduling problem to a minimum-cost, maximum flow optimisation – an approach only taken once in prior work (§5.2). However, Firmament extends the flow network optimisation approach in several ways, improving both the scheduler architecture and the information available for decisions (§5.3). As a result, Firmament is highly flexible and allows most desirable scheduling policies to be expressed (§5.4). I demonstrated this flexibility by considering four different cost models, illustrating a range of policies that Firmament can express (§5.5).

Finally, I outlined the theory behind the prevalent solvers for the minimum-cost, maximum-flow problem and described how the scalability challenges imposed by WSC scheduling are addressed by incremental solvers in Firmament (§5.6).

In Section 6.3, I evaluate Firmament using both real clusters and a trace from a Google WSC, finding that it compares favourably to the alternatives and scales well.

\(^{18}\)The modifications to RELAX IV were made by Adam Gleave as part of his Cambridge Part II individual project and are described in detail in his dissertation [Gle15, §3.7].
Chapter 6

Evaluation

“The first principle is that you must not fool yourself
– and you are the easiest person to fool.
So you have to be very careful about that.
After you’ve not fooled yourself, it’s easy not to fool other scientists.”
— Richard Feynman, “Cargo Cult Science” [FL10, p. 343].

In this chapter, I evaluate DIOS and Firmament with a variety of performance benchmarks. I run controlled experiments that stress key OS primitives, and measure the quantitative performance of application prototypes. I also use qualitative measures to assess programmability and security aspects of the DIOS design, and of Firmament’s flexibility.

The design and implementation of a new operating system even with most basic functionality is a large, complex, and time-consuming undertaking. The evaluation of the DIOS prototype must therefore be taken with a pinch of salt: its functionality is limited, its implementation is unoptimised, and the software stack is less mature than those in other OSes. Better performance can almost certainly be attained with more engineering. Its experimental nature can, however, also work in favour of DIOS: as a research OS, it leaner and may occasionally perform better than widely-deployed OSes that have more extensive feature sets.

That fact that I consider systems for warehouse-scale further muddies the waters: testing my systems in a real-world data centre with thousands of machines and real workloads is clearly outside the realm of possibility of academic research. Hence, the results presented in the following should be viewed as indicative validations of the principles behind DIOS and Firmament, rather than a conclusive evaluation of their performance.

6.1 Experimental setup

All experiments described in this chapter were carried out on one of two local testbeds in the Computer Laboratory. While hardly operating at “warehouse-scale” of thousands of machines,
CHAPTER 6. EVALUATION

<table>
<thead>
<tr>
<th>Type</th>
<th>Machine</th>
<th>Architecture</th>
<th>Cores</th>
<th>Thr.</th>
<th>Clock</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 x</td>
<td>A</td>
<td>GW GR380</td>
<td>Intel Xeon E5520</td>
<td>4</td>
<td>8</td>
<td>2.26 GHz</td>
</tr>
<tr>
<td>2 x</td>
<td>B</td>
<td>H8SGL-F</td>
<td>AMD Opteron 6168</td>
<td>12</td>
<td>12</td>
<td>1.9 GHz</td>
</tr>
<tr>
<td>2 x</td>
<td>C</td>
<td>Dell R420</td>
<td>Intel Xeon E5-2420</td>
<td>12</td>
<td>24</td>
<td>1.9 GHz</td>
</tr>
<tr>
<td>1 x</td>
<td>D</td>
<td>Dell R415</td>
<td>AMD Opteron 4234</td>
<td>12</td>
<td>12</td>
<td>3.1 GHz</td>
</tr>
<tr>
<td>1 x</td>
<td>E</td>
<td>SM AS1042</td>
<td>AMD Opteron 6168</td>
<td>48</td>
<td>48</td>
<td>1.9 GHz</td>
</tr>
</tbody>
</table>

(a) Heterogeneous SRG test cluster.

<table>
<thead>
<tr>
<th>Type</th>
<th>Machine</th>
<th>Architecture</th>
<th>Cores</th>
<th>Thr.</th>
<th>Clock</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>28 x</td>
<td>M</td>
<td>Dell R320</td>
<td>Intel Xeon E5-2430Lv2</td>
<td>6</td>
<td>12</td>
<td>2.4 GHz</td>
</tr>
</tbody>
</table>

(b) Computer Laboratory model data centre.

Table 6.1: Specifications of the machines in the two evaluation clusters.

The testbeds constitute sufficiently realistic and complex networked environments to represent a small real-world deployment of D\textsuperscript{I}OS and Firmament. They differ in their scale and heterogeneity:

**The heterogeneous SRG cluster** is a small ten-machine cluster composed of a mixture of Intel and AMD x86-64 machines. The machines have CPUs of various generations, ranging from four 2009 Intel “Gainestown” Xeons to a 2012 “Sandy Bridge” Xeon, and cover a range of clock frequencies and memory subsystem architectures (Table 6.1a). They are connected by via a two-switch 1G network and a single-switch 10G network.

**The homogeneous model data centre** is a recent 80-machine installation at the Computer Laboratory consisting of Dell R320 servers with identical specifications (Table 6.1b). The machines are connected by 1G and 10G networks, the latter in a leaf-spine topology with a 320 Gbit/s core interconnect. I use sub-clusters of up to 28 machines across two racks for my evaluation.\(^1\)

All machines run Ubuntu 14.04 (Trusty Tahr) with Linux kernel v3.14, with the D\textsuperscript{I}OS module enabled as needed, and all experiments use the 10 Gbit/s network.

The SRG test cluster, unlike the model data centre, exhibits similar heterogeneity to a real-world WSC (cf. §2.2.2).

I also use a third ad-hoc test cluster for the Green cost model case study with Firmament. This cluster consists of two ARM-based machines and a subset of the SRG test cluster machines (further details in §6.3.1.2).

\(^{1}\)28 machines are certainly small compared to an industrial WSC, but a scale of tens of machines is reportedly representative of many commercial customers’ “big data analytics” setups [ORR\textsuperscript{*}15, §2.4].
6.2 DIOS

I deployed the DIOS prototype on the model data centre testbed described in the previous section. In the following, I aim to answer the following questions using this deployment:

1. Do the DIOS operating system abstractions add any undue overheads over the cost of “classic” POSIX abstractions?

2. Is the design behind DIOS sufficiently performant to support typical WSC applications, e.g. parallel MapReduce-style data processing?

3. What benefits for security does the split identifier/handle capability scheme have, and how do they compare to other isolation techniques?

In a set of micro-benchmarks, I test individual DIOS features and system calls to address the first question (§6.2.1). I then address the second question by investigating the application-level performance of MapReduce on DIOS, comparing it to state-of-the-art single-machine and distributed implementations (§6.2.2). Finally, I analyse how system security can be improved using DIOS (§6.2.3), answering the third question.

Prototype limitations. Before I describe my experiments, I should point out a number of limitations of the DIOS prototype evaluated in the following. All of these limitations are consequences of missing features that can be supported with additional implementation work, but which are not critical to validating my hypotheses.

1. The flat, durable blob store (§3.6) is implemented as a special directory on a legacy ext4 file system.

2. The I/O request API (§3.7) is implemented, but there is only limited support for the different concurrent access semantics.

3. The prototype does not currently have support for group-based name spaces (§4.4). Implementing it will add no performance penalty, however, since it merely requires DIOS to store different names in the name table.

4. Reference delegation (§4.5.2) is only partially implemented; my experiments rely on the more expensive name lookup instead.

5. The DCP (§4.7) uses synchronous RPCs implemented over RDS, which times out on failure. However, no recovery action is taken on machine failure.

I believe that none of these limitations impact the validity of the results I present in the following, but I nevertheless intend to address them in future work.
### 6.2.1 Performance micro-benchmarks

The invocation of operating system functionality via system calls is typically on the critical path of an application. Hence, DiOS system calls ought to be fast.

In my first set of experiments, I measure the performance of individual DiOS kernel abstractions via micro-benchmarks. I consider object creation (§6.2.1.1), task creation (§6.2.1.2), and the performance of I/O via DiOS objects (§6.2.1.3).

#### 6.2.1.1 Object creation

Object creation is frequently invoked in DiOS programs, since they use it to allocate memory, create files and communication channels. In the first experiment, therefore I consider the performance of the `create(2)` system call.

As explained in Section 4.3, DiOS objects can be anonymous, or can have randomly or deterministically generated names. The different name generation algorithms have varying overheads: random name generation must source 32 random bytes, while deterministic name generation computes several SHA-256 hashes. Figure 6.1a illustrates the impact on object creation latency: deterministic name creation is the most expensive, at 7.5\(\mu s\) in the median, while random names are cheaper, and anonymous objects are the cheapest at 4\(\mu s\) in the median. In all cases, the 99\(^{th}\) percentile latency is around 10\(\mu s\).

Figure 6.1b shows the distribution of latencies for objects of different types. I consider five DiOS object types:
(i) a **NOOP** object, which is a special type of object that has no data, designed to measure only the overheads of object meta-data management;

(ii) a **PRIVMEM_BLOB** object, which establishes a private memory mapping;

(iii) a **DURABLE_BLOB** object, which represents an on-disk object;

(iv) a **SHMEM_STREAM** object, which allows for unidirectional shared-memory messaging; and

(v) a **UDP_STREAM** object, which represents a UDP network channel.

Where possible, I compare the creation latency to the latency of a roughly equivalent POSIX system call in Linux. For example, creation of a **DURABLE_BLOB** is comparable to **open(2)** on a file with **O_CREAT** set, while creating a **UDP_STREAM** object is equivalent to creating a UDP socket via **socket(2)** and **bind(2)**.

The results indicate that the DIOS prototype’s object-based abstraction has some overhead over the legacy Linux system calls. However, the 10\(\mu\)s creation time for DIOS objects is dominated by the deterministic name generation (cf. Figure 6.1a); with anonymous objects, the results are closer to the Linux system calls.\(^2\)

### 6.2.1.2 Task spawn overhead

Many WSC applications (e.g. a web server) use long-lived tasks that outlive the median OS process. Others, however, perform only small amounts of work on distributed data, and last for a sub-second duration [OPR+13]. Such short tasks have thus far been uncommon in WSC environments, as the task creation overhead is typically large, but DIOS has the opportunity to reduce this overhead.

In the following, I measure the cost of spawning a new DIOS task. This involves a process creation, and the setup work of furnishing the new task with a reference table and a default set of references. I run a job of 5,000 synthetic tasks that perform no work: they merely spin for 100ms and exit. I consider both a local spawn (using the **SPAWN_LOCAL** flag to **run(2)**) and a remote spawn on another machine. Waves of twelve tasks are executed in parallel, saturating all CPUs on a single machine in the local spawn case; when doing remote spawns, I round-robin the tasks across twelve machines.

In Figure 6.2, I compare these different cases and relate them to the overhead of a local Linux process creation (via the **fork(2)** and **exec(2)** legacy system calls), and the user-space task creation in two contemporary distributed systems: CIEL, a task-parallel compute framework [MSS+11], and Mesos, a Borg-like cluster manager [HKZ+11].

The cost of a purely local DIOS task spawn is similar to that of a Linux process creation – an unsurprising outcome, since the work performed is largely identical.\(^3\) However, creating a

---

\(^2\)The high creation overhead for **UDP_STREAM** objects (around 800\(\mu\)s) is a consequence of allocating large internal buffers, and of synchronously binding and connecting the underlying socket.

\(^3\)The 99th percentile outlier for **DIOS run(2)** comes because the prototype uses the Linux kernel’s User Mode Helper (UMH): the process creation indirections via two kernel threads, which are subject to scheduling delays.
remote DIOS task is significantly faster (at 100-300µs) than similar task creations in CIEL and Mesos, which take hundreds of milliseconds.\(^4\) This outcome is not necessarily surprising: CIEL and Mesos are not specifically optimised for fast task spawning (although neither is the DIOS prototype). However, it illustrates that having OS support for remote task creation can speed up distributed task creation.

### 6.2.1.3 I/O performance

Good I/O performance is crucial to many data-intensive WSC applications. I thus measure the I/O performance obtained via DIOS references. To do so, I run two DIOS tasks situated on the same machine or on different machines in a producer-consumer setup. I measure both the latency for a unidirectional message and the throughput at various message sizes.

Figure 6.3a compares the throughput for shared memory communication via a Linux pipe between parent and child process to the throughput of the same communication via a DIOS SHMEM_STREAM object. At all message sizes, the DIOS object achieves a somewhat higher throughput (by up to 87.5%) than the pipe transport does. This is perhaps a little surprising, but can be explained by (i) the DIOS implementation being leaner and not using any locks to restrict concurrent access, and (ii) DIOS re-using the same buffer for subsequent requests (the ACQUIRE_IOV_REUSE options to acquire_read(2) and acquire_write(2)), which reduces the necessary dynamic memory allocations. The one-way message latency (Figure 6.3b) follows a similar pattern: SHMEM_STREAM object’s read latencies are significantly lower than those of the pipe.

---

\(^4\)Note that while CIEL’s values include the time to run its cluster scheduler, those for Mesos do not: the measurement is taken between when the Mesos resource offer is accepted and when the task is reported as running.
I also consider remote I/O. Figure 6.4 shows the throughput measured when a single task makes I/O requests to a remote SHMEM_STREAM object, using an underlying TCP transport object for data transmission. The I/O request mechanism comes with some overheads: two DCP message round-trips, one to acquire and one to commit, are required. As a result, each I/O request has a latency of 300–400\(\mu\)s. At 4 KB reads, the throughput is around 16 MBit/s, while large 2 MB reads reach around 3.3 GBit/s. These relatively low results are a consequence of the transport object being idle while the (synchronous) DCP RPCs take place.

In addition to optimising the current prototype’s naïve DCP and TCP transport implementations, there are several other ways to improve upon this:

- I/O requests currently require two round-trips, even if validation never fails or if requests
run back-to-back. By supporting *compound calls*, such as “acquire-and-commit” (forgo-
ing validation) or “commit-and-acquire” (immediately starting a new request on commit),
the number of DPC RPCs per I/O request can be halved.

- Non-blocking, batched, or asynchronous I/O requests would also allow DIOS to achieve
  higher throughput. Most high-throughput systems that serve small requests rely on such
techniques (e.g. Facebook’s use of multigets in memcached [NFG+13]).

- Hardware offload and kernel bypass techniques can be applied for the DCP traffic: there
  is no need for its messages to traverse the entire network stack. Existing systems already
  achieve request latencies around 20\(\mu s\) using such techniques [LHA+14; MWH14], and
  Rumble *et al.* argue that \(\leq 10\mu s\) RPCs will be feasible in the near future [ROS+11].

Nevertheless, as I show in the next section, a distributed MapReduce application using the DIOS
I/O abstractions already outperforms state-of-the-art systems.

### 6.2.2 Application benchmark

The processing of huge data volumes is a classic “scale-out” workload that WSCs are con-
structed for. MapReduce [DG08] is a popular programming model for parallel data analytics
jobs, since it alleviates the programmer from having to manage concurrency and distribution
of work (see §2.2.1.1). To investigate the application-level performance of DIOS in its target
environment, I implemented MapReduce as a pure DIOS program. This section evaluates its
performance on the DIOS prototype and compares it against two state-of-the-art systems built
atop standard Linux abstractions: the Metis multi-threaded, single-machine MapReduce frame-
work [MMK10], and Spark [ZCD+12] v1.4.0.\(^5\)

In the experiment, I use MapReduce to process a large, distributed corpus of text. The test job
used in the following is the widely-used MapReduce “WordCount” benchmark, which com-
putes the number of occurrences of each word in a data set. However, the implementation is
sufficiently generic to support any computation that fits the MapReduce paradigm. Listing 6.1
shows the user-supplied `map()` and `reduce()` functions for WordCount.

The inputs are initially stored on disks across the cluster machines. I use two input data sets:

1. The synthetic **Metis dataset** consists of 51.2 million five-letter words (1 million unique
words) with a balanced distribution, and amounts to 300 MB.\(^6\)

2. A 2010 dump of all English-language **Wikipedia articles’ text**, post-processed to remove
all words longer than 1024 characters. This data set consists of 2.8 billion words (111
million unique words) and amounts to 21 GB.

---

\(^5\)The Hadoop MapReduce framework is a closer match to the paradigm implemented by DIOS MapReduce and
Metis than Spark, but its performance is known to be unrepresentative of state-of-the-art systems [SMH12, §3].

\(^6\)The original Metis data set had no line breaks; since Spark partitions map inputs by newline characters, I
reformatted the data set to ten words per line.
CHAPTER 6. EVALUATION

Listing 6.1: User-provided code in the DIOS MapReduce WordCount implementation.

```c
#include "dmr.h"
#include "dmr_map.h"
#include "dmr_reduce.h"

int64_t map(char* key, uint64_t klen, char* value, uint64_t vlen) {
    uint64_t i = 0, offset = 0, start = 0, end = 0, num = 0;

    while (offset < vlen) {
        i = offset;
        /* skip spaces */
        while (i < vlen && whitespace(value[i]))
            ++i;
        /* find end of word */
        start = i;
        for (; i < vlen && !whitespace(value[i]); ++i)
            value[i] = toupper(value[i]);
        end = i;
        /* emit (word, 1) tuple from mapper */
        map_emit(&value[start], end - start, (char*)1ULL, sizeof(uint64_t));
        offset = end + 1;
        num++;
    }
    return num;
}

int64_t reduce(const char* key, uint64_t klen, char** value, uint64_t vcount) {
    uint64_t sum = 0;

    /* add up the mappers’ counts for this key */
    for (uint64_t i = 0; i < e->count; ++i)
        sum += (uint64_t)e->values[i];

    /* emit (word, count) tuple from reducer */
    reduce_emit(key, klen, sum, sizeof(uint64_t));
    return 1;
}
```

While the Metis dataset is not a particularly taxing workload, it illustrates the fixed overheads of each framework, allows exploration of the scale-up behaviour within a single machine, and measures the frameworks’ ability to support fine-grained parallelism.

Unlike the competing systems, DIOS MapReduce does not write its output to durable storage in these experiments. However, writing the results would add less than a second for the Metis dataset (24 MB of output) and less than 40 seconds for the Wikipedia dataset (3.7 GB of output) even if written entirely on a single machine.

In Figure 6.5a, I show the job runtime on the Metis data set as a function of the number of
parallel cores and machines employed. It is evident that DIOS MapReduce offers competitive performance with Metis when scaling to multiple cores on a single machine. When scaling out to the cluster, Spark slows down from about 20s (12 cores, single machine) to about 50s, presumably since its overheads dominate for this small data set. DIOS MapReduce, however, continues to scale up to eight machines (albeit at only asymptotic benefit). This illustrates that DIOS can support fine-grained parallel distributed processing even for small data sets.

Figure 6.5b shows the same results for the larger Wikipedia input data set. Scaling to multiple machines has a greater impact here as parallel I/O and computation in the map phase can be exploited. Again, DIOS MapReduce performs well: it outperforms the competing systems both on a single machine and across machines, and scales up to 16 machines (192 tasks).

DIOS MapReduce is over $10 \times$ faster than Spark on the Metis dataset, and up to $3 \times$ faster on
CHAPTER 6. EVALUATION

the Wikipedia one. However, this result should not be overrated: while the features of DIOS MapReduce are on-par with Metis, it is a far less complex system than Spark. For example, Spark stores its shuffle data to disk [ORR+15, §3.2] for fault tolerance, while DIOS MapReduce does not support fault tolerance at the moment. Moreover, Spark runs in a JVM, which necessarily incurs overheads, but offers a more accessible programming interface to users.

Nevertheless, these experiments demonstrate two points:

1. Explicit support for data-intensive distributed computations at the OS level, as supported by DIOS, results in a high-performance platform that is at least competitive with widely used state-of-the-art systems.

2. The DIOS prototype scales to hundreds of parallel tasks, and offers parallel speedups even at fine-grained granularities for which current user-space distributed data processing systems fail to do so.

These performance results are an encouraging sign that DIOS may offer efficiency gains for WSCs. In the following, I consider additional qualitative benefits of using DIOS.

6.2.3 Security

Security and isolation in DIOS rely on distributed lightweight namespaces and capabilities (§3.5). In the following, I qualitatively assess the resulting security properties of DIOS by (i) comparing it with alternative isolation techniques used in WSCs, and (ii) sketching a plausible attack that its capability-based protection mitigates.

6.2.3.1 Comparison with alternative approaches

The primary security goal in a WSC is to isolate independent, possibly mutually distrusting applications atop shared hardware and software. Table 6.2 illustrates the features offered by common isolation mechanisms.

Approaches differ in their overheads (e.g. containers’ low cost vs. VMs high memory overhead), the granularity at which resources can be shared (e.g. capability schemes’ fine-grained sharing vs. containers’ and VMs’ coarse-grained shared volumes), and the customisation opportunities afforded to the user (e.g. specialised, custom kernels in unikernels vs. a shared underlying kernel). In general, higher specialisation effort permits tighter compartmentalisation, but restricts sharing.

All of these approaches are single-machine solutions, however: their namespaces and protection domains end at the machine boundary. Across machines, they rely on BSD sockets and standard network protocols, which make it difficult to restrict interaction. Permitting and application to establish network connections is tantamount to enabling it to send any of the data accessible
CHAPTER 6. EVALUATION

172

Table 6.2: DIOS security properties compare favourably with other inter-task isolation techniques deployed in WSCs. († possible with hypervisor support)

<table>
<thead>
<tr>
<th>Approach</th>
<th>Overhead</th>
<th>Per-task kernel</th>
<th>Isolated distr. namespaces</th>
<th>Separate filesystem</th>
<th>Fine-grained capabilities</th>
<th>Resource sharing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared machine (processes)</td>
<td>none</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Virtual machines (VMs)</td>
<td>high</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Specialised OS/stack (unikernels)</td>
<td>medium</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Kernel namespaces (containers)</td>
<td>low</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Capability systems (e.g. Capsicum)</td>
<td>low</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>DIOS</td>
<td>low</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

to it to any other task in the WSC, unless application-level restriction and authentication are implemented.

DIOS, by contrast, supports both isolated distributed namespaces via its groups, and fine-grained capabilities using names and references. As a result of this, it supports selective sharing of objects (via references), and compartmentalises tasks such that they can only access (and leak data to) a specific, narrow set of objects. However, DIOS does not by default have an OS kernel per task, or support for isolating non-DIOS kernels and legacy applications.

6.2.3.2 Case study: the evil MapReduce job

Consider the example of a MapReduce computation, which reads inputs from a distributed file system and executes user-provided lambdas for \texttt{map()} and \texttt{reduce()} on the input.

**Threat.** An “evil” MapReduce job crafted by a nosey developer might not merely access the inputs it is supposed to process (e.g. click logs for ads). Instead, the job also accesses other information accessible to the same identity (e.g. email access logs). The evil MapReduce job then “siphons” the additional information for accounts of interest into a network connection to another harmless-looking task commissioned by the user.

**State-of-the-art.** Only inbound network access to containers or VMs is typically filtered: outgoing connections can be established to any machine and port in the WSC – in part since the locations of other tasks are not known ahead of time.

In the Hadoop Distributed File System (HDFS), access control is based on user and group IDs derived from host OS or Kerberos identities [HDFSDocs15]. This only enforces fairly
coarse-grained isolation: any file and directory accessible to the authenticated user is accessible to all jobs running as this user.

Finally, since the map() and reduce() lambdas are user-specified code, they may have arbitrary effects, including the invocation of any external binary or library available in the task’s root file systems.

As a result, the above threat is hard to mitigate in state-of-the-art systems: distributed file system permissions are insufficiently granular to issue per-job credentials (capabilities); outgoing network connections are insufficiently restricted; and the invocation of arbitrary standard binaries (such as nc for network transmission) is possible.

**Mitigation in DİOS.** If the MapReduce tasks are implemented as pure DİOS binaries, DİOS can restrict them to only access their predefined input and output objects. In DİOS MapReduce, the job controller task – which does not run any user-provided code – resolves the job’s input names, creates intermediate objects and output objects, and delegates references to these objects to the map and reduce tasks.

Moreover, the network communication between distributed file system, mapper and reducer tasks is restricted to those channels permissible under the MapReduce programming model. When using DİOS objects only, remote I/O is transparent, i.e. the task does not know the host or port identities used by the underlying transport objects. Establishing an arbitrary network connection is not possible.⁷

Finally, if the MapReduce logic needs to invoke a helper binary, it may do so, but it must spawn a DİOS task for it. The only executable references available to the MapReduce task are those which were delegated to it by the job controller. Hence, the set of helper binaries is restricted, and the helper subtask inherits the restrictions of the MapReduce task.

### 6.2.3.3 Limitations

Like most operating systems with a monolithic kernel, DİOS offers little protection against kernel exploits that enable an attacker to execute arbitrary code in supervisor mode. Of course, commonly deployed OSes and kernel namespace solutions like Linux containers also fail to contain a kernel-level compromise. However, as discussed in Section 3.4, DİOS reduces the attack surface somewhat by offering only a minimal API to pure DİOS tasks (see §4.9.3).

DİOS is also not currently resilient to man-in-the-middle attacks on the WSC interconnect. If an attacker manages to compromise an intermediary node (usually a data-link layer switch or a router), the integrity of RPCs and coordination messages can be compromised. Section 7.1.3 discusses possible solutions to this problem.

---

⁷The reader may observe that nothing stops a task from creating its own network stream object and connecting it to a subtask. This is correct – however, the permissible subtasks can be constrained (see next paragraph), or the DİOS kernel can deny creation of these object types to untrusted tasks.
6.2.4 Summary

In the preceding sections, I used the DIOS prototype with the Linux host kernel to evaluate my WSC OS design. I set out a set of goals for such a novel distributed OS for WSCs in Section 2.1.4, and briefly summarise below how DIOS has successfully met them.

Results

I have demonstrated that DIOS meets the goals of a distributed OS for a WSC (§2.1.4):

1. The DIOS abstractions are competitive with traditional POSIX abstractions in micro-benchmarks, adding little extra overhead when operating locally, and enabling low-overhead distributed operations (§6.2.1).

2. A typical data-intensive application, MapReduce, exhibits competitive performance both with state-of-the-art single-machine and distributed implementations when run on DIOS (§6.2.2).

3. The DIOS abstractions and its group-based distributed namespaces offer isolation that improves over container-based solutions (§6.2.3).

Of course, the performance results presented are influenced by the architecture of the host kernel, i.e. Linux, whose implementation choices are sometimes at odds with the goals of DIOS. In Section 7.1.2, I discuss changes to kernel structure that would benefit DIOS.

6.3 Firmament

I now turn to evaluating the Firmament cluster scheduler, which I described in Chapter 5. Firmament is a general cluster scheduler for WSCs, and thus targets a broad set of applications. Unlike some other schedulers (see §2.3), Firmament is not limited to a specific parallel programming model or workload type.

In my evaluation, I use the cluster testbeds (§6.1) and a Google cluster trace [RTG+12] to answer the following questions:

1. What benefits do the placement decisions made by different Firmament cost models have for user applications? (§6.3.1)

2. How flexibly does Firmament’s generalised notion of scheduling as a flow network optimisation adapt to different scheduling policies? (§6.3.2)
CHAPTER 6. EVALUATION

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>SRG test cluster</th>
<th>Model DC</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpu_spin</td>
<td>Spin CPU for 60s.</td>
<td>4 × 10</td>
<td>8 × 10</td>
</tr>
<tr>
<td>mem_stream, L3-fit</td>
<td>Swap words in 1M array.</td>
<td>4 × 10</td>
<td>8 × 10</td>
</tr>
<tr>
<td>mem_stream, &gt;LLC</td>
<td>Swap words in 50M array.</td>
<td>4 × 10</td>
<td>8 × 10</td>
</tr>
<tr>
<td>io_stream, read</td>
<td>fio asynchronous read of 4 GB.</td>
<td>1 × 5</td>
<td>2 × 7</td>
</tr>
<tr>
<td>io_stream, write</td>
<td>fio asynchronous write of 4 GB.</td>
<td>1 × 5</td>
<td>2 × 7</td>
</tr>
</tbody>
</table>

Table 6.3: Synthetic workloads used in cluster mix experiments.

3. How well does Firmament scale to large clusters with thousands or tens of thousands of machines? (§6.3.3)

My evaluation only touches upon a subset of Firmament’s features and of the scheduling policies it can support in order to answer the above questions. In future work, I intend to use Firmament as a platform for exploring other concerns in WSC scheduling, some of which I outline in Chapter 7.

6.3.1 Decision quality

Firmament’s use of minimum-cost optimisation over a flow network is motivated by its ability to find high quality assignments for a scheduling policy specified as a cost model. As I explained in Section 5.2.1, the assignments found are \textit{policy-optimal} for the given cost model.\footnote{This does \textit{not} imply general optimality: a better cost model may lead to better assignments.}

Consequently, Firmament’s practical usefulness depends on how good these cost models are. In the following, I evaluate Firmament in two scenarios that correspond to the new cost models described in Section 5.5, and measure the quality of its decisions.

6.3.1.1 Case study: avoiding co-location interference

The proactive avoidance of workload interference (§2.2.3) was a key motivating use case for Firmament. To this end, I implemented two interference-aware cost models: a Whare-Map cost model (§5.5.2) and the CoCo cost model (§5.5.3), and I evaluate the reduction in co-location interference when using these cost models.

I use a set of five synthetic workloads, specifically designed to stress different machine resources: CPU, memory, and disk I/O (Table 6.3). These workloads constitute extremes and thus allow me to approximate an upper bound on the possible gain from an interference-aware scheduler. In the experiments, Firmament does not initially have any information about the workloads. However, the workloads are repetitive: completing jobs are resubmitted at most
10 seconds after they finish. Hence, Firmament over time acquires task profile information for each equivalence class (cf. §5.3.3).

The target cluster utilisation is around 80% of the CPU threads; if jobs take a long time to complete due to stragglers, the utilisation can at times drop below this target. I dimensioned the I/O-bound jobs such that in an optimal assignment, their tasks can run free of interference (i.e. there are as many disk-bound jobs as machines).

I compare against two baselines: (i) Firmament with a queue-based, “random first fit” scheduler instead of its usual flow optimisation scheduler, and (ii) the Mesos cluster manager. Comparing against the queue-based approach quantifies the impact of the flow scheduler and cost model while using the same underlying cluster manager (viz. Firmament). Mesos, on the other hand, is a widely-deployed production cluster manager which supports multi-dimensional resource requirements [GZH11], but does not explicitly consider co-location interference.⁹

Whare-Map cost model. The Whare-Map cost model’s scores are based on instructions-per-second (IPS) data collected at task runtime (converted to picoseconds per instruction, psPI, for use with Firmament; see §5.5.2). Zhang et al. showed that IPS are strongly correlated with the application-level performance of Google workloads [ZTH13, §3]. In the experiment, the Whare-Map cost model builds up a set of IPS scores for each combination of task equivalence class and machine equivalence class over time. As a result, scheduling decisions are initially random and improve over time.

CoCo cost model. The Whare-Map model only indirectly models machine load: a task running in a more contended environment achieves a lower IPS value. However, it may nevertheless accidentally co-locate tasks that overwhelm a machine’s resources.

The CoCo cost model, unlike Whare-Map, explicitly considers resource load and per-task resource requests. Consequently, it requires information about workloads’ resource requirements and potential interference between them. In the experiment, I specify appropriate resource requirements for each task on submission and assign it to one of four interference classes (§5.5.3).

Metric. The goal of this experiment is to measure performance unpredictability due to interference. I quantify it using the normalised task runtime relative to the ideal runtime. To obtain the ideal, I first executed a single task of each workload on an idle machine of each type (Table 6.1) without using a cluster scheduler. The best machine type’s average runtime over ten executions is the ideal runtime. In other words, a normalised task runtime of 1.0× means that the task completed as fast as on the most suitable, idle machine.

⁹As discussed in §2.3.1, Mesos is a two-level scheduling system. In this experiment, I use the simple shell executor “framework” on top of the Mesos resource manager. In the Mesos paradigm, a scheduler framework would implement co-location-aware decisions; however, none of the existing top-level frameworks do.
Figure 6.6: Runtime of the synthetic workloads from Table 6.3 for a 1-hour experiment, normalised to the best runtime on an idle machine (without using a cluster scheduler). Boxes around the median value correspond to 25th and 75th percentiles, whiskers are 1st and 99th percentiles, and the star represents the maximum outlier.
CHAPTER 6. EVALUATION

Results. Figure 6.6 shows the distribution of per-task runtimes as a box-and-whisker plot for the heterogeneous SRG test cluster (Figure 6.6a) and the homogeneous model data centre (Figure 6.6b). There are five workloads, and I show a cluster of four boxes for each workload, corresponding to the two baselines, queue-based Firmament and Mesos, the WhareMap cost model, and the CoCo cost model. Lower results and tighter distributions are better.

Firmament’s cost models improve normalised task runtime over the queue-based baseline in almost all cases. This is unsurprising, as the queue-based baseline neither models load nor interference. However, Firmament’s cost models also always match or outperform Mesos, which is load-aware.

In the following, I discuss these high-level results in more detail, focusing on the Whare-Map and CoCo cost models. I address three questions of

(i) how quickly the Whare-Map cost model’s self-tuning discovers good assignments;
(ii) why the CoCo cost model outperforms the Whare-Map one in most cases; and
(iii) whether Firmament’s interference-aware cost models lead to more efficient use of the underlying WSC hardware.

Self-tuning in the Whare-Map cost model. To answer the first question, it is important to understand what the Whare-Map cost model’s IPS scores mean:

1. A faster CPU or faster I/O hardware increase instruction throughput, and thus yield a higher IPS score. As a result, the Whare-Map cost model discovers affinities between tasks and machine types.

2. Co-location interference – for shared caches and for other resources – reduces instruction throughput, and thus yields a lower IPS score. As a result, the Whare-Map cost model discovers sets of tasks that fit well together, and ones that do not.

Figure 6.7 illustrates Whare-Map’s self-tuning for the “mem_stream, L3-fit” workload. The timeline shows the average normalised per-task runtime for each minute of a one-hour experiment, and compares timelines for the queue-based baseline, the Whare-Map cost model, and the CoCo cost model. In the first minute of the experiment, assignments are random and the variance is high. However, both average normalised runtime and variance for Whare-Map drop steeply once the first wave of tasks finishes and the cost model acquires information about their performance. By contrast, CoCo and the baseline approach neither accumulate knowledge over time, nor take micro-architectural counters into account, and see continuously high normalised runtimes and variance.

Whare-Map vs. CoCo. I now turn to answering the second question, which is why CoCo usually outperforms the Whare-Map cost model. Even though the latter self-tunes, tasks still frequently experience degradation in their normalised runtime. This happens because the Whare-


Figure 6.7: Average runtime of `mem_stream` (L3-fit) tasks in each minute of a one-hour experiment on the heterogeneous SRG cluster in the queue-based baseline and with the different Firmament cost models; Error bars are ±σ. Whare-MCs quickly discovers good mappings at the start of the experiment and continuously applies them when possible.

Map cost model relies only on IPS scores: it has no notion of machines’ multi-dimensional resource capacities, and does not model interference explicitly.

Consider, for example, the Whare-Map cost model’s performance for the `io_stream` workloads in Figure 6.6. Due to their fast disks, type A machines, have attractive IPS scores when running a single task, and the cost model hence perceives them as a good match. When another `io_stream` job arrives, the scheduler consequently – in a single scheduling round – assigns many of its tasks to the “good match” machines, leading to overcommit. This is especially problematic for the 24-core (type C) and 48-core (type E) machines, which can run many tasks.

The CoCo cost model avoids overcommit by using admission control to ensure that tasks fit before they schedule, and schedules tasks in “waves” to avoid overcommit due to independent concurrent placement of interfering tasks (§5.5.3). The benefits are evident in Figure 6.6b: only one `io_stream` task fits on each machine, and hence their normalised runtime is close to ideal.

On the heterogeneous cluster (Figure 6.6a), normalised runtime varies due to the machines’
CHAPTER 6. EVALUATION

Figure 6.8: Per-task wait time for different workloads in the cluster mix experiment on the heterogeneous SRG cluster. Boxes around the median value correspond to 25th and 75th percentiles, whiskers are 1st and 99th percentiles, and the star represents the maximum.

heterogeneous disks and CPU speed, rather than due to interference.

However, the Whare-Map cost model does have advantages: unlike CoCo, it requires no information from the user, and its IPS score accurately captures micro-architectural performance and interference via shared caches. For example, the L3-fitting mem_stream workload sees the tightest runtime distribution using Whare-Map on the heterogeneous SRG cluster (Figure 6.6a).

Moreover, CoCo’s conservative admission control requires tasks to wait much longer than with the Whare-Map cost model. Figure 6.8 shows task wait time distributions for the same workloads and setups as shown in Figure 6.6. The median wait time for tasks in CoCo is around 10–15 seconds, while the Whare-Map cost model places tasks within 200ms in the median.

**Hardware utilisation.** Finally, I now answer the third question – whether Firmament facilitates more efficient use of the same hardware. Figure 6.9 illustrates this using the cumulative distribution of average cycles-per-instruction (CPI) values for all tasks in the experiment. A cumulative distribution situated further to the left indicates that fewer cycles are required to execute each instruction, and thus that the hardware is utilised more efficiently. Firmament’s interference-aware cost models have CPI distributions that are almost always as good or better than the baseline distribution. The Whare-Map cost model, in particular, yields a significantly improved distribution as it avoids cache misses by optimising IPS scores.
Figure 6.9: Distribution of the per-task average CPI for the synthetic workloads in Table 6.3 over a 1-hour experiment on the heterogeneous SRG test cluster. Less is better, and Firmament utilises the hardware more efficiently than the baseline scheduler.

Discussion. My experiments have shown that Firmament cost models can be used to implement scheduling policies that effectively mitigate co-location interference. However, there are several plausible improvements over the Whare-Map and CoCo policies as implemented here:

1. Whare-Map’s ability to learn good mappings can be combined with CoCo’s resource reservations and interference scoring. Such an integration would allow the scheduler to learn avoiding co-locations not adequately covered by CoCo’s interference scoring, while still supporting resource reservations and load balancing.

2. CoCo’s wait times can be reduced by admitting tasks in priority order, rather than adding arcs to resource aggregates in random order. Priorities based on current wait time would help schedule stragglers sooner.\(^{10}\)

I also compared my CPI distributions in Figure 6.9, to CPI distributions for Google workloads in the 2011 cluster trace [RWH11], as well as to instructions-per-memory-access (IPMA) distributions. The detailed results are in Appendix A.3, but the key take-away is that Google’s workloads experience much higher CPI (suggesting higher load or more interference) and lower IPMA than the synthetic workloads I used (suggesting large working sets, low cache affinity, or high interference).

\(^{10}\)However, this conflicts with CoCo’s strict priority goal: stragglers would be scheduled in preference to higher-priority tasks that have not waited as long. A viable policy might admit tasks in priority order first and by wait time second, or to treat batch and service tasks differently.
Table 6.4: Specifications of the machines used in energy-aware scheduling experiments with Firmament.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Architecture, core count × clock speed</th>
<th>RAM</th>
<th>Network</th>
<th>Disk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pandaboard</td>
<td>ARM Cortex-A9, 2 × 1.0 GHz</td>
<td>1 GB</td>
<td>100 Mbit/s</td>
<td>SSD (USB 2.0)</td>
</tr>
<tr>
<td>Wandboard</td>
<td>ARM Cortex-A9, 4 × 1.0 GHz</td>
<td>2 GB</td>
<td>1 Gbit/s</td>
<td>SSD (S-ATA)</td>
</tr>
<tr>
<td>Dell R420</td>
<td>Intel Xeon E5-2420, 1 × 1.9 GHz</td>
<td>64 GB</td>
<td>10 Gbit/s</td>
<td>SSD (S-ATA)</td>
</tr>
<tr>
<td>Dell R415</td>
<td>AMD Opteron 4234, 12 × 3.1 GHz</td>
<td>64 GB</td>
<td>10 Gbit/s</td>
<td>HDD (S-ATA)</td>
</tr>
<tr>
<td></td>
<td>Intel Itanium 2 9015, 8 × 1.4 GHz</td>
<td>16 GB</td>
<td>1 Gbit/s</td>
<td>HDD (S-ATA)</td>
</tr>
</tbody>
</table>

Table 6.5: Machine power consumption at different CPU load levels.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Idle power</th>
<th>Full-load power</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pandaboard</td>
<td>6.95 W</td>
<td>7.58 W</td>
</tr>
<tr>
<td>Wandboard</td>
<td>9.55 W</td>
<td>12.25 W</td>
</tr>
<tr>
<td>Dell R420</td>
<td>87.10 W</td>
<td>175.04 W</td>
</tr>
<tr>
<td>Dell R415</td>
<td>70.60 W</td>
<td>238.02 W</td>
</tr>
<tr>
<td>Itanium 2</td>
<td>544.94 W</td>
<td>657.81 W</td>
</tr>
</tbody>
</table>

Interference-aware scheduling addresses an immediate need in today’s production WSCs: improving utilisation and performance determinism by using the existing hardware more efficiently. However, Firmament can also support more exotic use cases. In the following, I describe the Green cost model, which improves energy efficiency on a heterogeneous-ISA cluster.

6.3.1.2 Case study: energy-aware scheduling

In Section 5.5.4, I described the Green cost model for heterogeneous clusters. It enables Firmament to balance the energy efficiency benefits of using low-power architectures (e.g. ARM-based servers) against the consequent drop in performance. If workloads’ high-level performance constraints – such service job latency and throughput SLOs, or batch job deadlines – have flexible slack, the Green cost model automatically uses the most energy-efficient combination of machines that still meets the constraints.

I evaluate the practical energy savings attained in a heterogeneous cluster using a mixed batch and service WSC workload.\(^{11}\)

Cluster setup. The test cluster for this experiment consists of five heterogeneous machines, listed in Table 6.4. Two ARMv7-based SoCs represent upcoming many-core ARM servers: a dual-core Pandaboard\(^{12}\) and a quad-core Wandboard.\(^{13}\) Future ARM-based server products will feature higher clock speeds, larger numbers of cores, and increased memory capacity [AMD14],

\(^{11}\) The experiments presented here were run in collaboration with Gustaf Helgesson under my supervision; an earlier version of Figure 6.10 appears in Gustaf’s MPhil thesis [Hel14, Fig. 6.2].


\(^{13}\) http://www.wandboard.org/; accessed 03/07/2014.
but their relative energy efficiency and single-threaded performance compared to x86 servers is likely to be similar.

The cluster also includes two x86-64 servers with different CPU architectures (AMD “Valencia” and Intel “Sandy Bridge”) and clock frequencies (3.1 GHz and 1.9 GHz). Finally, for additional heterogeneity, the cluster also contains an IA-64-based Itanium 2 machine.\textsuperscript{14} All machines run Linux, although the x86 machines run Ubuntu 14.04 with kernel 3.13.0, the ARM-based machines run Arch Linux ARM with kernel 3.13.0, and the Itanium machine runs Debian 7.5 with kernel 3.2.0-4-mckinley.

**Power monitoring.** I use a fixed-core transformer measurement device to monitor the power consumption at the PSU of each cluster machine. This device samples the total root mean squared (RMS) current for each connected machine every three seconds and sends it to Firmament. The device measures power at the socket and reports whole-system power consumption, including CPU, DRAM, peripheral, and PSU components.\textsuperscript{15}

When multiple tasks share a machine, I divide the overall power consumption between them according to the number of CPU cores used. While this is somewhat crude, it works well in practice as power consumption is highly linear in the number of cores utilised.

**Workloads.** The experiment workload is a mix of batch jobs and service jobs. The batch jobs run typical data centre MapReduce workloads (WordCount and joining two datasets) and file transfers. Batch jobs are issued such that, on average, ten jobs run at any given time. Each job’s deadline is set as a randomly sampled factor of \((2, 20)\) times its runtime on the fastest machine for the job.

As a service workload, I run an HTTP server serving static web pages. Clients connect to a load-balancing HTTP proxy (HAProxy v1.4.25) which forwards connections to service tasks running the nginx web server (v1.6). The load-balancing proxy uses weighted round-robin load balancing, with weights corresponding to the typical throughput offered by each machine. The Green cost model automatically scales the number of web server tasks depending on the current load: additional tasks are launched when throughput exceeds 55% of the estimated current capacity, and scaled down when it falls below 15%.

**Metrics.** My experiments quantify energy efficiency by measuring the *above-idle energy consumption*. The “above-idle energy” is the energy (in Joule) consumed while running a workload, in addition to the baseline energy consumed by the idle machine. This metric makes the

\textsuperscript{14}The Itanium is neither energy-efficient nor power-proportional – it merely serves to test the scheduler’s performance on a wider trade-off space.

\textsuperscript{15}The device was calibrated against a third-party device and its readings compared to measurements from the Running Average Power Limit (RAPL) power measurement interface available on the Intel machine [HDV\textsuperscript{+}12]. The values observed agreed within a maximum deviation of 3–4 W, but were usually within ±1 W.
assumption that machines in a WSC are always powered on, i.e. the idle energy cost is incurred no matter what scheduling decisions Firmament makes. This is in contrast with efforts to perform dynamic power management (DPM) of data centre machines, such as BEEMR [CAB+12]. I use two different metrics to measure performance for service jobs and batch jobs. For service jobs, the high-level goal is for the service to meet the load it experiences (measured by throughput). Batch jobs, on the other hand, have the goal of completing by a deadline, and deadline satisfaction is their metric of effectiveness.\footnote{Even when meeting the deadline, it is of course preferable – i.e. more efficient – for a batch job to complete the same work in less time or using fewer resources.}

**Energy savings.** For evaluation, I compare four different approaches:

\begin{itemize}
\item[(i)] randomised task assignment over the x86 machines only (i.e. a homogeneous cluster);
\item[(ii)] randomised task assignment over the entire heterogeneous, mixed-ISA cluster;
\item[(iii)] task assignment according to a performance-oriented, but energy-oblivious Firmament cost model; and
\item[(iv)] task assignment according to the energy-aware Green Firmament cost model.
\end{itemize}

The first approach represents the state-of-the-art baseline of using a traditional, x86-only WSC. One would expect this configuration to perform well, but to expend more energy than a heterogeneous setup. The randomised approach over the entire heterogeneous cluster (option (ii))
corresponds to a naïve application of current schedulers to a heterogeneous setting, and likely makes some pathological placement decisions.

By contrast, the Green and maximum-performance cost models use Firmament’s flow network to express costs. In the maximum-performance cost model, the cost of running a task on a machine is proportional to its ability to complete the task quickly (in the case of batch tasks) or to deliver high throughput (for service jobs). This model should perform at least as well as a homogeneous x86 cluster, but may expend extra energy when using the non-x86 machines in addition to the x86 ones. Finally, the Green cost model (§5.5.4) aims to complete tasks in the most efficient way possible, subject to their performance constraints. This approach should yield the lowest energy consumption overall, while ideally still meeting the performance constraints.

In Figure 6.10, I show the timeline of above-idle energy consumed during an 800-second experiment. As expected, the Green cost-model yields the lowest above-idle energy consumption. All other approaches also perform as expected: the maximum-performance cost model uses slightly more energy than random assignment over a homogeneous x86 setup, while randomised assignment over the heterogeneous cluster comes second in terms of energy consumed. However, with randomised assignment, the experiment took 2× longer to complete (not shown in Figure 6.10), while all other setups completed within ±1.5% of the Green cost model’s runtime.

The aggregate above-idle energy saved by the Green cost model is 45% of the above-idle energy used in the homogeneous x86 setup. When taking into account machines’ baseline energy, this reduction amounts to a 6.2% saving of total energy consumed; excluding the energy-inefficient IA-64 machine increases the energy savings to 18% of the total energy.

I also measure the energy savings contributed by Firmament’s auto-scaling of service jobs. For this purpose, I run an experiment with a web server service job exposed to varying client load according to a diurnal pattern observed at Google [BCH13, p. 26]. Figure 6.11a shows the
<table>
<thead>
<tr>
<th>Cost model</th>
<th>Avg. web server throughput [req./s]</th>
<th>Missed batch deadlines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random homogeneous</td>
<td>52,553</td>
<td>0</td>
</tr>
<tr>
<td>Random heterogeneous</td>
<td>51,049</td>
<td>5</td>
</tr>
<tr>
<td>Maximum performance</td>
<td>53,324</td>
<td>0</td>
</tr>
<tr>
<td>Energy-aware</td>
<td>52,871</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.6: Service job performance and batch deadline satisfaction for the different schedulers and cost models.

above-idle energy expended on the service job over the time of the experiment. In the Green cost model, between one and four service tasks run on different machines, while in the homogeneous x86 case, four tasks continuously run on the two x86 machines. Auto-scaling combined with the energy-aware cost model reduces the above-idle energy consumption by 17.9% over the fixed setup while serving the same load pattern (Figure 6.11b). This indicates that Firmament successfully utilises the ARM machines at times of low load and that scaling the web server tasks according to load yields energy savings.

Performance constraints. The energy savings observed are only meaningful if jobs still manage to meet their performance constraints. I therefore measure the total request throughput for service jobs in the experiment shown in Figure 6.10 and the number of missed deadlines for batch jobs. The respective values for each cost model are shown in Table 6.6. All cost models apart from random assignment to heterogeneous machines attain the same request throughput and manage to meet all deadlines.

6.3.2 Flexibility

In the previous sections, I evaluated the utility of three cost models that cover two specific use cases for Firmament. However, Firmament can support many other scheduling policies expressed as pluggable cost models.

Table 6.7 summarises Firmament’s support for the policies implemented by other existing schedulers (see Table 2.6 in §2.3), and briefly indicates how they would be implemented.

Many simple policies (e.g. LATE and delay scheduling) can be implemented by simply adapting the cost terms or values. Others are more complex and rely on appropriate admission control, i.e. they only connect task nodes to the flow network once specific conditions hold. These include cost models based on multi-dimensional resource models and those with complex fairness notions (e.g. H-DRF and Choosy). Policies that express co-dependent decisions (e.g. via combinatorial constraints, or dynamic workload scaling) may require multi-round scheduling, in which Firmament places only one task at a time and recomputes the costs for others afterwards (cf. §5.4.4).

Finally, some systems – Mesos, YARN, and Omega – are themselves flexible scheduler platforms that can support variable policies. Firmament can support all existing high-level schedul-
CHAPTER 6. EVALUATION

<table>
<thead>
<tr>
<th>System [Reference]</th>
<th>Implementable</th>
<th>Admission control required</th>
<th>Multi-round scheduling req.</th>
<th>Implementation summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>HFS [HFS]</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Enforce fairness using unscheduled aggregator demand and arc capacities (§5.4.2).</td>
</tr>
<tr>
<td>LATE [ZKJ*08]</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Model using $v^I_j$, $\alpha^I_j$ and preference arcs.</td>
</tr>
<tr>
<td>Quincy [IPC*09]</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Cost model described in §5.5.1.</td>
</tr>
<tr>
<td>Delay Sched. [ZBS*10]</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Use $v^I_j$ to induce delay; drop $v^I_j$ after expiry.</td>
</tr>
<tr>
<td>CIEL [Mur11, §4.3]</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Use preference arcs for Sweetheart references and locality (§5.4.1).</td>
</tr>
<tr>
<td>Jockey [FBK*12]</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>Scale number of tasks in cost model; model deadlines via cost on arc to unscheduled aggregator.</td>
</tr>
<tr>
<td>alsched [TCG*12]</td>
<td>✓</td>
<td>x</td>
<td>✓</td>
<td>Soft/hard constraints via preferences, combinatorial via multi-round scheduling (§5.4.1).</td>
</tr>
<tr>
<td>tetrisched [TZK*13]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Soft constraints via preferences, combinatorial ones via multi-round scheduling (§5.4.1).</td>
</tr>
<tr>
<td>Whare-Map [MT13]</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Cost model described in §5.5.2.</td>
</tr>
<tr>
<td>Sparrow [OWZ*13]</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Use distributed coordinators that each run a scheduler; optionally have multiple parent coordinators.</td>
</tr>
<tr>
<td>Choosy [GZS*13]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Only admit tasks if CMMF satisfied.</td>
</tr>
<tr>
<td>Paragon [DK13]</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>Use profiling results to determine costs on arcs to machine equivalence classes (cf. §5.5.2).</td>
</tr>
<tr>
<td>Quasar [DK14]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>As in Paragon, but also scale resource requests in cost model in response to model.</td>
</tr>
<tr>
<td>Apollo [BEL*14]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Cost model described in this section.</td>
</tr>
<tr>
<td>KMN [VPA*14]</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Force $k$ of $n$ tasks to schedule via gang scheduling (§5.4.3), increase cost for additional $m = n - k$.</td>
</tr>
<tr>
<td>Tarcil [DSK15]</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>Use distributed coordinators’ schedulers for short tasks, and a top-level coordinator for long ones.</td>
</tr>
<tr>
<td>Hawk [DDK*15]</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>Use arcs from equivalence class aggregator to machines to express work stealing cost.</td>
</tr>
<tr>
<td>Mercury [KRC*15]</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>As Hawk, but guaranteed tasks preempt queueable ones on delegation conflicts (via higher priority).</td>
</tr>
<tr>
<td>Bistro [GSW15]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Map resource forest onto resource topology and use resource model akin to CoCo (§5.5.3).</td>
</tr>
<tr>
<td>CoCo (co-location-aware)</td>
<td>✓</td>
<td>✓</td>
<td>x</td>
<td>Cost model described in §5.5.3.</td>
</tr>
<tr>
<td>Green (energy-aware)</td>
<td>✓</td>
<td>x</td>
<td>x</td>
<td>Cost model described in §5.5.4.</td>
</tr>
</tbody>
</table>

Table 6.7: Firmament can flexibly support the policies implemented in many existing schedulers. A tick in parentheses, (✓), indicates that the cost model must modify the workload, i.e. it acts as both a job submission client and as a policy to the scheduler.
ing policies for these systems that I am aware of, although more challenging ones may be conceivable.

**Example: Apollo.** As a concrete example of how an existing complex scheduler would be mapped onto Firmament, consider the Apollo scheduler [BEL+14]. Apollo is an Omega-like shared-state scheduler, in which Job Managers (JMs) enqueue tasks for execution at Process Nodes (PNs) based on the PNs’ predicted resource availability (a “wait time matrix”) which is aggregated by a Resource Monitor (RM). The wait time matrix is computed from information about the runtime of prior runs of similar tasks – in Firmament, this information is available in the coordinator’s knowledge base.

Each Apollo JM combines the wait time for a task with its expected I/O time, which is estimated from the input data size, and its expected runtime. It then computes the estimated completion time $E$ as $E = I + W + R$, where $I$ is the I/O time, $W$ is the expected wait time at the machine, and $R$ is the task’s predicted runtime. In Firmament, the JM would add arcs with a cost proportional to $E$ from the task equivalence class aggregator to the machines in its candidate set.\(^{17}\)

To decide on the best matches of tasks to machines, Apollo employs “a variant of the stable matching algorithm [GS62]” [BEL+14, p. 290] for each “batch” (similar to a Firmament equivalence class), sorts the results by quality (wait time), and dispatches tasks until out of capacity. Firmament replaces this matching and subsequent dispatch with its minimum-cost, maximum-flow optimisation, processing all batches at the same time. The resulting assignment quality is no worse than with Apollo’s approach, as long as only one task is assigned to each machine in each iteration. This restriction already exists in Apollo [BEL+14, p. 291]; wave-based assignment (as in CoCo) enables it on Firmament.

### 6.3.3 Scalability

In Section 5.2, I discussed how Firmament’s scalability is impacted by the scalability of the underlying minimum-cost, maximum-flow solver. I explained how scalability can be improved by solving the minimum-cost, maximum-flow problem *incrementally*. In this section, I evaluate the reduction in scheduler decision time attained by this approach.\(^{18}\) I compare the incremental approach to solving the optimisation problem from scratch each time (as Quincy does).

In the experiment, I use Firmament on a Google WSC’s workload, simulating a subset or the whole of the 12,550 machine “cell” in the 2011 public cluster trace [RWH11]. I extend the public cluster trace in two ways:

\(^{17}\)Matching the exact Apollo semantics would require two minor extensions to the current Firmament prototype: first, a task wait queue would have to be associated with each resource, and second, the per-machine coordinators (≃ Apollo’s PNs) would have to register with multiple parent coordinators (≃ Apollo’s JMs).

\(^{18}\)The experiments presented here were first conducted by Adam Gleave for his Part II individual project under my supervision in the academic year 2014/15 [Gle15]; I re-analysed the results for exposition here.
1. As the precise nature of the machines in the WSC is unknown, I synthesise the topology of a 24-core machine (equivalent to type C in Table 6.1a) for each machine.

2. To be able to use Quincy’s locality preferences in the flow network, I simulate a GFS-like distributed file system, and assign random inputs to each task. The distributed file system contains 1.2 billion 64 MB blocks (≈75 PB), with file sizes sampled from a distribution of HDFS file sizes at Facebook [CAK12, Fig. 1, 3] and clamped to [64 MB, 20 GB].

In the experiment, Firmament uses the Quincy cost model and simulates one hour of the Google workload,\(^{19}\) and I measure the time taken by different solvers to complete the minimum-cost, maximum-flow optimisation. I use two solvers: Goldberg’s \(\text{cs2}\) based on the cost-scaling push-relabel algorithm [Gol97], which runs a full optimisation on every iteration, and a modified version of Frangioni et al.’s implementation [FGB11] of RELAX IV [BT94] that runs an incremental optimisation.

Figure 6.12 shows the results for both a medium-sized cluster of 3,000 machines – sub-sampling about a quarter of the Google trace events – and for the full Google WSC. In 2009, Quincy took “a little over a second” to schedule 100 jobs on a cluster of 2,500 quad-core machines [IPC+09, §6.5]; by contrast, my simulated 3,000-machine cluster uses 24-core machines, runs about 500 jobs with about 40,000 tasks, and adds an additional 198,000 vertices over Quincy (66 per machine) for the resource topology (§5.3.2). As Figures 6.12a and 6.12b show, Firmament’s average decision time is 515ms when running Goldberg’s \(\text{cs2}\) solver from scratch, with the 90\(^{th}\) percentile at around 1s. This is comparable to the 2009 result for Quincy. The incremental solver, however, completes in 44ms on average and has a 90\(^{th}\) percentile runtime of under 250ms.\(^{20}\)

The results for the full-scale Google WSC show a similar trend: the average decision time is 265ms, the 90\(^{th}\) percentile is at 345ms, and 97.4\% of decisions are made in under a second. By comparison, the average decision time running \(\text{cs2}\) from scratch is 2.86s, over 10\(\times\) longer than with the incremental solver. Moreover, the distribution of full optimisation decision times has a long tail: in the 90\(^{th}\) percentile, decisions take 4.98s (incremental: 0.35s), and the 99\(^{th}\) percentile is at 9.25s (incremental: 3.98s). This difference is a consequence of the different algorithmic approaches taken by the incremental re-optimising RELAX IV solver and the cost-scaling \(\text{cs2}\): cost scaling is based on amortised costs, but requires the solver to visit all nodes in the flow network several times, while the relaxation algorithm quickly identifies the small number of paths that have changed and focuses on optimising those (cf. §5.6.1).

This experiment shows that an incremental solver offers significantly reduced decision times in the common case of a single job arrival. Indeed, the incremental solver decision times are

\(^{19}\)This is a simulation and not a replay of the trace since different scheduling decisions are made: unlike in the trace, each machine runs at most 24 tasks (one per CPU thread) and the simulated DFS locality is hypothetical. However, as the primary metric of interest is the solvers’ decision time, this reduction in fidelity is inconsequential.

\(^{20}\)The first run on a pre-populated cluster at the start of the simulation is an exception to this: the incremental solver’s algorithm takes significantly longer than \(\text{cs2}\)’s cost-scaling here. However, Firmament can use a threshold on the number of changes to decide which solver to use.
competitive with those achieved by distributed schedulers such as Sparrow [OWZ+13] or Tar-
cil [DSK15], despite Firmament being used as a fully centralised scheduler here.

In addition to incrementally solving the min-cost, max-flow optimisation, there are several other ways in which Firmament’s decisions could be accelerated:

1. Multiple solvers can run in parallel on subsets of the workload. Since tasks are the most numerous entity in the flow network, it may make sense to shard jobs or tasks across schedulers, which each solve a smaller flow network based on a replica of the cluster state. If two schedulers both place a task on the same resource, a conflict ensues and one scheduler must retry. I first suggested and evaluated such optimistically concurrent decisions in Omega, showing that optimistic concurrency between dozens of schedulers works well even for Google workloads [SKA+13]. In Firmament this approach would maintains the optimality of decisions when operating on consistent cluster state.
2. A **hierarchical delegation** of the scheduling workload can reduce the size of the flow networks that each solver must optimise. For example, the top-level scheduler could make coarse-grained assignments over racks only, and leave it to another scheduler in each rack to optimise a flow network corresponding to this rack only, and similarly within machines. The drawback with this approach is that it can easily lose optimality unless it is carefully implemented.

The Firmament implementation supports the first option (optimistically concurrent solvers), and only simple extensions are required to support hierarchical delegation. However, in light of the fact that even a relatively unoptimised incremental minimum-cost, maximum-flow solver achieves decision times comparable to the fastest state-of-the-art distributed schedulers, neither option is likely to be required for scalability alone.

### 6.3.4 Summary

In Chapter 2, I enumerated several goals for improved scheduling in a WSC OS (§2.2.5), and for an accurate, flexible, and scalable cluster scheduler (§2.3.6). In this section, I evaluated how Firmament meets all of these goals.

### Results

With the experiments in this section, I showed that:

1. Firmament **avoids co-location interference** between workloads and **matches workloads to appropriate machines**, addressing challenges 2 and 3 in Section 2.2.5. Using the Whare-Map and CoCo cost models, Firmament reduces workload slowdowns due to suboptimal assignments by $2\sim 4\times$ in the median (§6.3.1.1).

2. The Green cost model for Firmament yields **improved energy efficiency** on a heterogeneous ARM/x86 WSC, reducing overall energy consumption for the given workload by 18%, and addressing challenge 4 in Section 2.2.5 (§6.3.1.2).

3. Firmament constitutes a **highly flexible platform** for scheduler development: in addition to the three cost models evaluated in detail, the policies of most existing cluster schedulers can be implemented (§6.3.2). This addresses challenges 1–3 in Section 2.3.6.

4. Despite generating high-quality solutions using a computationally intensive flow network optimisation, Firmament exhibits **good scalability** and makes **rapid scheduling decisions**. Using an incremental minimum-cost, maximum-flow solver, Firmament achieves sub-second decision times that are competitive with fast distributed schedulers (§6.3.3), addressing challenges 4 and 5 in Section 2.3.6.
Moreover, Firmament meets – to the extent of possibility – all the goals for a cluster scheduler identified in Section 2.3 (Table 2.6). In my experience, its flexible cost model API (Appendix C.5) makes it relatively straightforward to implement different scheduling policies: after the initial implementation and testing with Whare-Map, implementing the CoCo cost model took only about one week of work.

6.4 Summary and outlook

In this chapter, I have evaluated the D10S operating system and the Firmament cluster scheduler, which are prototype realisations of the concepts described in this dissertation. The results allow me to conclude that:

1. Explicit operating system support for distributed operation in a WSC is feasible.
   The D10S design and prototype implementation illustrate that the principles introduced in Chapter 3 suffice to build a specialised, distributed OS for a “warehouse-scale computer” that meets the goals set out in Chapter 2.

2. My WSC OS design supports non-trivial distributed systems at competitive performance.
   MapReduce-style parallel data processing on D10S achieves better performance than current state-of-the-art frameworks running on legacy operating systems.

   My cost models for Firmament yield high-quality placement decisions for goals as varied as interference avoidance and energy efficiency, and the generalised flow network scheduling approach flexibly extends to other policies.

4. Contrary to prior belief, scheduling via flow networks can scale.
   Using an incremental minimum-cost, maximum-flow solver, Firmament scales to large clusters with tens of thousands of machines while making rapid, sub-second scheduling decisions.

As research prototypes, D10S and Firmament probably raise as many new questions as they answered. In the final chapter, I discuss opportunities for extending the prototypes, and suggest directions for further research.
Chapter 7

Conclusions and future work

End-user applications are changing: they now routinely access enormous repositories of information stored in remote data centres. These data centres are often “warehouse-scale computers” that support large distributed systems. These systems require transparent distribution, scalable parallel processing and fault tolerant execution. The data centre environment is unique in scale, in the demands of its workloads and its high resource utilisation, and its importance is only likely to increase in the future.

However, the most privileged and crucial part of the systems software stack in such WSCs – the operating system – is entirely unaware of its part in a larger distributed installation. Motivated by both this unawareness and the mismatch between legacy operating systems (which cater to general workloads) and WSCs’ distributed systems workloads (which have specific needs), I have proposed novel approaches to building operating systems and schedulers for WSCs.

• In Chapter 3, I stated the high-level principles for a new distributed operating system for WSCs and illustrated how they differ from approaches taken in prior forays into distributed operating systems in the context of LANs of personal workstations. Using the design of DIOS as a guiding example, I illustrated how these principles can shape the abstractions exposed by a modern OS kernel.

• Subsequently, in Chapter 4, I presented the details of DIOS and its prototype implementation. I explained how DIOS uses names to identify logical objects, relies on groups to delineate namespaces, and employs references as information-rich handles for translucent interaction with objects. A new system call API around these abstractions makes it possible for DIOS users to implement scalable distributed applications with lightweight, yet rigid isolation and the flexibility to distribute the application as transparently as required.

• Chapter 5 focused on scheduling in the distributed WSC OS. I showed that modelling the scheduling problem as a flow network is both able to express novel policies and sufficiently flexible to cover many existing cluster scheduling policies. I presented the Firmament scheduler, a generalisation of Quincy [IPC+09], which combines the flow network
scheduling approach with additional information, a new scheduler architecture, and a more scalable incremental minimum-cost, maximum-flow solver.

I discussed three case studies of scheduling policies implemented for Firmament: (i) an implementation of the Whare-Map interference-aware scheduling policy, (ii) the co-ordinated co-location model (CoCo), which simultaneously optimises for high resource utilisation and minimal interference between tasks, and (iii) the Green cost model, which minimises the energy footprint of a WSC workload by using power-efficient machines.

The work presented in these chapters collectively serves to prove my hypothesis introduced in Chapter 1. First, explicit operating system-level support for distributed operation in modern warehouse-scale data centres is feasible, and applications can draw efficiency and security benefits from it. With DIOS, I have demonstrated that new abstractions facilitate an OS design on which practical distributed systems applications can be implemented. Second, the Firmament scheduler proves two points: (i) that the integration of fine-grained, machine-level information and global, cluster-level information yields better scheduling decisions, and (ii) that the flow network approach to scheduling is not only highly expressive, but can be made scalable and performant.

There is, however, ample opportunity for future work extending DIOS and Firmament.

### 7.1 Extending DIOS

DIOS has demonstrated that a modern distributed OS is both feasible and interesting. However, it is very much an initial step in the direction of making the operating system more aware of its role in a distributed system. To make DIOS a viable alternative to conventional OSes, more work is required. In the following, I discuss three possible avenues for future research.

#### 7.1.1 High-level language support

Writing programs directly against the DIOS system call API is rather complex. Most software indirectly calls into the operating system via a standard library (e.g. libc), and DIOS comes with a libc-like standard library (dlibc). However, its facilities are rather low-level compared to what common WSC applications need. By integrating DIOS abstractions with higher-level programming languages, users can draw on its benefits without having to implement their own low-level resource management.

Rust, for example, is a new, memory-safe “systems programming language” [Rust14] that supports functional, imperative, object-oriented, and concurrent-actor styles of programming. It does not have dangling or null pointers, and does not support implicit sharing, but instead statically tracks memory allocation ownership.
By extending the Rust runtime with an indirection layer for DIOS, Rust programs can be supported on DIOS, allowing distributed applications to be written concisely. To this end, Andrew Scull and I have adapted the “native” Rust runtime to work on DIOS.\(^1\) The ported runtime supports all core language features, and allows unboxed closures in a Rust program to execute in separate DIOS tasks. Tasks can communicate using Rust channels that are implemented over a DIOS stream object (e.g. a shared memory FIFO, or a network connection).

Listing 7.1 shows the implementation of a Rust program that uses DIOS tasks to compute the 10\(^{th}\) number of the Fibonacci sequence. This 24-line implementation is far simpler than the a C implementation against the DIOS system call API, which comes to 280 lines, or a Rust implementation using POSIX pipes and processes, which comes to 50 lines.

In preliminary benchmarks, the Rust implementation of \(\text{fib}(10)\) (265 tasks) had less than 10% overhead over a baseline implementation in C against the DIOS system call API. Given that Rust is a new language and still under active development, and considering the unoptimised nature of DIOS and the runtime port, this is an encouraging result.

\(^1\)The Rust runtime port was principally completed by Andrew Scull for his Part II individual project at Cambridge under my supervision in the academic year 2014/15 [Scu15].
An interesting next step would be to study which higher-level paradigms a programming language or standard library should expose in order to make the construction of distributed applications on D1OS as accessible as possible. Approaches such as composable component design from distributed objects in Sapphire [ZSA+14], the use of future, filter, and service abstractions in Finagle [Eri13] and the transparent in-memory object caching of Tachyon [LGZ+14] give some indication of directions that might be fruitful to explore.

7.1.2 Changing kernel structure

In my work on D1OS, I have so far focused especially the OS abstractions at the boundary between the kernel and user-space. However, the specific requirements of WSCs motivate further, deeper changes to OS construction.

**OS-level Quality-of-Service (QoS) enforcement.** Firmament’s scheduling policies allow the WSC scheduler to avoid negative co-location interference between tasks by evading interfering placement choices. This does not address the root cause of the problem, however: the fact that current hardware and OS kernels offer poor performance isolation between user-level processes, containers, or VMs sharing resources.

Against the backdrop of the OS kernel changes that I have proposed, it seems timely to revisit work on OS-level QoS enforcement. For example, performance isolation in the VM system – such self-paging in Nemesis [Han99] – and better performance isolation under shared concurrent access to I/O devices would benefit a WSC OS. Additionally, new hardware features – such as Intel’s Cache Allocation Technology (CAT), which explicitly partitions CPU caches between processes – will require OS and cluster-level support (as, e.g. in Heracles [LCG+15]).

**Scalable APIs.** In Section 4.8, I argued that the D1OS system call API is designed to be more scalable than a legacy POSIX API. While the API design targets scalability as a first-class principle, the scalability of a given kernel implementation backing it is a different matter entirely. Specifically, the reliance on Linux kernel code in the D1OS prototype restricts the practical scalability attainable.

Using a host OS kernel explicitly designed for scalability – such as Barrelish [BBD+09] or sv6 [CKZ+13, §6] – instead of Linux might improve the practical scalability of the D1OS abstractions. Barrelish in particular is an attractive target, since D1OS does not require implicit shared memory or cache coherence (Principles 1 and 2, §3.1).

To explore some of these questions, I hope to port D1OS to other host kernels. For example, a port of D1OS to sv6 would allow the scalability of the D1OS system call API to be evaluated independently of Linux kernel implementation choices; a port to Barrelish would investigate how well-suited the D1OS abstractions might be for use on non-cache-coherent systems.
7.1.3 Further security improvements

I have shown that the D10S capability system offers a degree of inter-task isolation that is at least as good, and usually better, than existing and widely-used kernel namespace virtualisation. Further improvements are possible, and might follow three different avenues:

**Mapping D10S capabilities to hardware capabilities.** The identifier and handle capabilities in D10S are implemented entirely in software. Handles (references) are valid within an entire address space, as they are pointers into virtual memory. Moreover, MMU-based bounds checking and overrun protection are coarse-grained, and references and I/O buffers (D10S `iovecs`) can be adjacent in memory. The use of guard pages or similar mechanisms can mitigate, but not eliminate, protection violations within a task address space. However, some use cases (e.g. holding private keys in an infrastructure tasks’ memory) necessitate more fine-grained protection and compartmentalisation. Fine-grained hardware-software capability models like CHERI [WWC+14] could help with this. CHERI’s object capability support [WWN+15, p. III.D] maps well onto the D10S object model, and would allow fine-grained compartmentalisation of objects even within the task address space. In this way, D10S could in the future isolate network threads from objects holding secrets in the same task, for example.

**Applying information flow control.** One key advantage of D10S is its use of consistent OS-level abstractions (names, references and objects) throughout a distributed system, rather than layering disparate abstractions. Information flow control (IFC) monitors, and reasons about, how information is exposed to different components of a system. Many IFC systems – e.g. HiStar [ZBK+06], and its distributed DStar [ZBM08] variant – are based on label propagation. Since D10S makes the communication between tasks explicit via shared objects, it may simplify the tracking and enforcement necessary for strict IFC.

**Using more advanced cryptography for delegation.** D10S currently relies on transport-level data encryption for secure capability transfer across machine boundaries. More expressive cryptographic schemes could be utilised to give stronger guarantees: for example, Macaroons can have attached attestations that authenticate them (possibly via a third party), but are still communicable on untrusted channels [BPE+14]. The capability delegation mechanisms in D10S could be adapted to use Macaroons to gain these benefits.

I hope to investigate these directions in the future.

7.2 Evolving Firmament

Firmament is a flexible platform for developing schedulers specifically optimised for a given use case, but it can also serve as a level playing field for comparing existing schedulers’ policies.
This is timely: despite dozens of different schedulers existing (cf. §2.3), the comparison of different scheduling policies for a given workload has received little attention in research, largely – I suspect – for reasons of practicality. I hope to extend my analysis in Section 6.3.2 with an implementation of several additional policies atop Firmament, and a comparative analysis of their relative merits on real and simulated workloads.

Moreover, even though Firmament supports all key features currently demanded of cluster schedulers, new requirements keep emerging. For example, future WSCs may contain even more fundamentally heterogeneous compute resources: Firebox [MAH+14, §2.1; Asa14] and HP’s “Machine” project [Edg14] expect future WSCs to be constructed from heterogeneous custom systems-on-chip designs (SoCs) with large shared memories, which will require careful scheduling. Other efforts accelerate computations using FPGAs (e.g. Bing’s Catapult [PCC+14], and the Dandelion compiler [RYC+13]), which likewise complicates the scheduling problem.

Firmament already detects heterogeneous machine types, resource load and tasks’ affinities for specific co-locations, and makes them available to scheduling policies. It would be interesting to extend it to take into account the additional dimensions afforded by such specialised hardware.

### 7.3 Summary

As applications increasingly rely on back-end services operated in large-scale “cloud” data centres, operating systems must evolve. In this dissertation, I have made the case for explicit OS-level support for distributed systems, and for a deeper integration of distributed systems with the OS in data centres abstracted as “warehouse-scale computers”.

With DiOS and Firmament, I have developed a prototype platform that can make such data-intensive, inherently distributed computing environments more efficient, safer and easier to use. I hope to further explore the implications and practical utility of both systems in my future research.
Bibliography


Lars Bergstrom. “Measuring NUMA effects with the STREAM benchmark”. In: *CoRR* abs/1103.3225 (2011) (cited on page 44).


[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ian Horn, Naty Leiser, and Grzegorz Czajkowski. “Pregel: A System for Large-scale Graph Processing”. In: Proceedings of the 2010 ACM SIGMOD


<table>
<thead>
<tr>
<th>Reference</th>
<th>Authors</th>
<th>Title</th>
<th>Conference/Proceedings</th>
<th>Pages</th>
<th>Notes</th>
</tr>
</thead>
</table>


Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, et al. “F1: The Fault-tolerant Distributed RDBMS Supporting Google’s Ad Business”. In: *Proceedings of the 2012 ACM SIG-


Appendix A

Additional background material

A.1 Operating system specialisation

Early operating systems were specially developed for particular hardware and user needs, but OSes soon became general-purpose. For example, Unix and its derivatives are used across dozens of hardware architectures, and in environments ranging from mainframes to servers, desktops and embedded devices. Even the distributed operating systems described in Section 2.1.2 were intended for general day-to-day desktop workloads on user workstations.

Such generality is not always beneficial: in specific domains and when the workload is well-known, specialised systems can offer significant benefit. In the following, I discuss several research directions that specialise OSes to particular use-cases or hardware environments.

**Micro-kernels** specialise OS kernels for fast inter-process communication (IPC), in order for a minimal privileged kernel to support most OS services as (possibly likewise specialised) user-space servers (§A.1.1).

**Multi-tenant isolation** uses techniques such as hypervisors, kernel namespaces or hardware extensions to specialise the OS to a multi-user environment with particular trust assumptions and workload needs (§A.1.2).

**Multi-core OSes** specialise the OS for scalable operation on many-core machines, or towards increasingly common heterogeneous multi-core hardware (§A.1.3).

**Full-system specialisation** entails OS kernel modification for a particular specialised environment, tailoring the whole OS to an application or environment (§A.1.4).

My work – to a varying extent – draws on all of these avenues of research, and I point out key influences.
A.1.1 Micro-kernels

As operating systems gained more functionality and an increasing number of device drivers, the volume of kernel code running in privileged mode kept increasing. In the late 1980s, researchers proposed the concept of a *micro-kernel* to reduce this amount of privileged code. Many of the later distributed operating systems were based on micro-kernels, most notably Mach [ABB+86].

However, most 1980s micro-kernels – including Mach – suffered from poor performance. The L4 micro-kernel set out to change this by radically restricting the primitives available in the kernel to basic IPC and protection facilities and supporting very rapid context-switches [Lie95]. It is the most widely-used micro-kernel today, although it is deployed mostly in embedded systems, where portability is secondary to safety and performance.

In the 1990s, several single address space operating systems (SASOS) based on micro-kernels were developed. Mungi [Hei98] implemented distributed shared virtual memory (DSVM) over several machines, with isolation relying on the use of memory addresses as capabilities. Neme-sis [LMB+96], on the other hand, targeted multimedia applications on a single machine, and separated address translation from protection, allowing many “execution domains” to share protection settings within a single address space.

Micro-kernels are – in principle – a natural fit for distributed operation across machines due to their message-passing architecture, and some (e.g. Mach and Mungi) support cross-machine operation. Perhaps paradoxically, however, micro-kernels might be responsible for distributed systems software being firmly confined to user-space today: after the micro-kernels of the 1980s and 1990s pushed distributed system components into user-space, there was no compelling performance advantage to bringing them back into the kernel, unlike with other OS subsystems.

Like micro-kernels, my work aims to expose *only* the abstractions required for user-space applications to implement their own policy (e.g. a distributed key-value store’s consistency level). However, unlike many micro-kernels, I do not restrict myself to simple message-passing abstractions as the OS interface.

A.1.2 Multi-tenant isolation

The isolation of user-space applications is important in shared systems, and process isolation is sometimes insufficient. Hence, more specialised approaches are common in multi-tenant environments: full OS virtualisation grants maximum flexibility and requires minimal mutual trust (at the cost of overheads), kernel namespaces and containers trust a shared kernel (with reduced overheads), and I/O hardware virtualisation trusts the “control plane” to set up isolated “data plane” domains (within which no overheads exist).

**Full OS virtualisation.** The idea of virtualising an execution environment – typically machine hardware – arose fairly early in the evolution of general-purpose computing. Early key
use cases included the ability to emulate old machines on new ones, permitting legacy programs to continue operating after system upgrades, as well as the isolated sharing of mainframe resources [LM88, pp. 591–2; TW06, pp. 46–8].

In the early 2000s, Xen [BDF+03] addressed drawbacks with the prevalent commercial virtualisation software by introducing *paravirtualization*: the running of slightly modified guest operating systems to avoid expensive translation on every memory access. Xen triggered attempts to use virtualisation to other, specialised ends: for a small hypervisor-based trusted-computing base (TCB) [SK10], virtual machine migration [CFH+05; ASG10; DPP+10], coarse-grained compartmentalisation using virtual machines [PJ10; RW10], and nested virtualisation for security and portability [BDD+10; ZCC+11; WJW12].

Virtualisation’s promised economic gains due to server consolidation and consequent increased utilisation are tightly connected to “cloud computing”. Cloud computing uses ephemeral, mobile virtual machines hosted at a provider who offers computation resources as chargeable “utilities”. Since a large provider achieves better economies of scale, this approach can be cheaper than running on-premises infrastructure.

**Kernel namespaces.** Full virtualisation allows users to run their own OS kernel inside a VM environment. Yet, this is not always required: in many cases, isolation between users’ *application stacks* is key, but all users can share the same OS kernel.

This can be achieved by partitioning namespaces. In traditional time-shared, multi-programmed OSes, all users share namespaces such as the file system, the process IDs, and open network ports. As a result, users can observe each others’ programs, they may make conflicting requests for resources, and they cannot be trusted with superuser privileges.

*Containers* instead rely on the kernel to expose entirely separate namespaces for different containers. Within the kernel, the clashing user-level identifiers (e.g. “PID 1”) are remapped to unique system-level identifiers (e.g. “PID abc:1” for container “abc”).

The notion of such lightweight isolation originates with resource containers [BDM99]. Resource containers account and limit *resources* on a per-application basis, including resource consumption in multiple processes and in the OS kernel. To achieve this, an application concept had to be defined, effectively introducing namespaces to the OS.

BSD jails [KW00] extended this notion with mechanisms for *security* containment via namespaces, allowing users to exert superuser privileges inside a jail. Solaris Zones [PT04] and the VServer¹ and LXC² extensions for Linux subsequently brought similar designs to other OSes, and combined the security isolation of jails with the resource isolation of resource containers.

Containers have recently gained in popularity for WSC environments due to their low memory and runtime overheads. Container support in Linux [Men07] was much advanced by Google’s

[^1]: [http://linux-vserver.org](http://linux-vserver.org); accessed 05/01/2015.
[^2]: [http://linuxcontainers.org](http://linuxcontainers.org); accessed 05/01/2015.
use of containers with the Borg cluster manager [VPK+15, §6.2]. Consequently, recent stacks that package applications as containers and manage them are increasingly popular: examples include Docker [Mer14] and CoreOS’s Rocket.³

I/O device virtualisation. Containers isolate user-space applications atop a shared OS kernel, but do not allow the user to supply privileged, kernel-mode code.⁴ I/O hardware virtualisation allows users to access privileged instructions and I/O hardware, but, unlike full virtualisation, does not require them to supply a full OS kernel.

For example, NoHype proposed [KSR+10] and implemented [SKL+11] a hypervisor-less system that uses hardware virtualisation extensions to statically partition a machine. Likewise, Dune [BBM+12] allows user processes to invoke privileged operations safely. It uses the VT-x hardware virtualisation extensions in recent Intel processors to run a Dune-enabled process in a VMX ring 0 non-root context. This grants access to ring 0 hardware features such as exceptions, virtual memory (page tables), TLB entry management, and segmentation. The IX OS uses Dune to run privileged “data plane” code with direct access to NIC queues, while keeping different applications safely isolated [BPK+14].

Arrakis [PLZ+14] is similar: it runs several library OS instances with direct access to I/O hardware, but requires no privileged (hypervisor) mediation beyond initial “control plane” setup. Like IX, Arrakis uses I/O devices’ virtualisation support, but runs applications as isolated library OS instances (similar to containers), rather than using virtualisation extensions to support an existing Linux user-space.

My work draws on the ideas in kernel namespaces to isolate users’ applications while sharing an omniscient OS kernel. Hardware-assisted virtualisation techniques can be used to improve performance and strengthen isolation, although they are not the focus of my work.

A.1.3 Multi-core operating systems

A different type of specialisation in OSes was required when CPU designers faced heat dissipation problems with deeply pipelined, super-scalar processors in the early 2000s. Instead of the clock frequency increasing, the number of cores per processor started increasing, making the scalability of OS abstractions a key concern in high-utilisation environments.

Scalability. In early work, Boyd-Wickizer et al. showed that Linux spin locks, file descriptor tables, and page tables scale poorly to many cores [BCC+08]. Hence, they argue that kernel state ought to be private rather than shared by default, and that all sharing should be explicit.

⁴Some hybrid schemes that combine containers with “micro-VMs” exist – e.g. Clear Linux (http://clearlinux.org; accessed 02/06/2015) and Hyper (http://hyper.sh; accessed 02/06/2015).
The Corey OS realises this via explicitly shared *address ranges, kernel cores* and *shares*, and a new, scalable system call API. Subsequent research, however, found that selective patches to existing kernels can be sufficient to attain scalability [BCM*10a].

Some operating system abstractions inherently scale poorly “by design”. For example, network socket operations (e.g. `accept(2)` ) access global data structures, and packet sending and receipt often take place on different cores [PSZ+12, §2]. Addressing these issues requires modifying socket semantics, as in e.g. the “affinity-accept” design [PSZ+12, §3.2] or MegaPipe’s “lightweight sockets” [HMC+12].

This example highlights that both the OS implementation and the OS *abstractions* must be scalable. Clements *et al.* showed that some OS interface designs are *inherently* conducive to good scalability: a scalable implementation must exist if two OS API calls commute, i.e. the outcome is identical regardless of how the calls are ordered [CKZ+13].

**Heterogeneous multi-core architectures.** Classic SMPs are homogeneous, but processors increasingly have heterogeneous CPU cores that operate at different speeds, lack cache coherency or use different instruction set architectures (ISAs). For example, the ARM big.LITTLE platform combines different ARM cores [Gre11]; the nVidia Tegra K1 combines GPU cores with a 64-bit ARM CPU [BBT+15]; and the Intel Xeon Phi augments an x86 system with additional highly parallel x86 cores [JR13, pp. 6–9].

Without cache-coherency or shared memory, OS designs may need to abandon the idea of a single kernel controlling of the whole system. Helios [NHM+09] and Barrelfish [BBD+09] instead take a message-passing approach: as in micro-kernel IPC, communication happens via message-passing – between processes (“dispatchers”) in Barrelfish, or kernels on heterogeneous cores (“satellite kernels”) in Helios. As a result of this multi-kernel architecture, Barrelfish can dynamically boot and shutdown cores [ZGK+14].

By contrast, Popcorn Linux [BSA+15] extends a traditional single-system image kernel with support for heterogeneous multi-core systems. Processes can dynamically migrate across ISAs in order to exploit specific features in specific computation phases.

In a heterogeneous multi-core system, workload performance often depends on correct placement and scheduling decisions. In Barrelfish, Peter investigated a “phase-locked” variant of gang-scheduling multiple dispatchers to avoid skew in synchronisation [Pet12, pp. 64–129]. Other projects focus on choosing the right processor type: “bias scheduling” uses dynamic measurements of internal and external stalls to derive core type affinities [KRH10], while Popcorn Linux requires the user to explicitly initiate migrations [BSA+15, §4].

WSCs are composed of thousands of many-core machines, and hence my work carefully ensures that the new OS abstractions developed are inherently scalable. Moreover, my scheduling work explicitly takes heterogeneity – both across machines and within a single machine – into account when placing workloads in the WSC.
A.1.4 Full-system specialisation

Some efforts also specialise the OS implementation to specific application workloads. One way of achieving this is to dynamically extend the OS code. This approach was popular in mid-1990s extensible OSes that allow applications to replace core OS functionality with bespoke variants. Examples of such systems include VINO [ESG+94], which allows code from a trusted compiler to be “grafted” dynamically into the kernel, and SPIN [BSP+95], which relies on type-safety in Modula-3 to guarantee extension behaviour.

Specialisation can also be achieved by moving OS functionality into the application and exposing interfaces closer to “bare-metal”. Library operating systems require only a minimal resource allocation and isolation substrate from the host, within which the library OS applies its own policies for, e.g., memory allocation, scheduling, and compartmentalisation. The Exokernel originally pioneered this use of self-managed resources allocated via “secure bindings” [EKO95].

Drawbridge [PBH+11] is a recent revisit of the library OS concept, with an application-binary interface (ABI) that converts complex Windows applications into library OSes. Bascule [BLF+13] extends Drawbridge with dynamically loadable OS extensions via recursive interposition of ABI calls between the library OS and the host. Such extensions enable low-overhead tracing, file system remapping, application checkpointing, and architecture emulation, and can be composed. While Drawbridge applications cannot communicate via standard IPC mechanisms, Graphene [TAB+14] further extends the Drawbridge ABI to communicating multi-process Linux applications, parts of which run in different library OS instances.

Other library operating systems use a hypervisor rather than a host OS: Mirage builds library OSes from type-safe OCaml modules and runs them either as UNIX processes or as single-process VMs atop Xen [MMR+13]. OSv, like Mirage, builds the library OS on top of a hypervisor, but supports legacy applications via a subset of libc [KLC+14]. Both Mirage and OSv show that tangible performance gains are attainable when using a specialised library OS on a thin hypervisor.

Library OSes were originally motivated by performance limitations in traditional OSes [And93], and similar concerns have recently resurfaced. Specialised OSes that use hardware virtualisation to give applications direct access to hardware (e.g. the nonkernel [BPA+13], Arrakis [PLZ+14] and IX [BPK+14]; see §A.1.2) typically run applications structured as library OSes.

If the workload is uniform, a special-purpose OS can have appealing benefits. For example, vNUMA offers distributed shared memory over multiple machines to run multi-threaded legacy applications on a cluster [CH09], while dOS [BHC+10] and DDOS [HBC+13] are fully deterministic OSes for repeatable, race-free computation. fos [WGB+10] and Tessellation [CEH+13] target multi-core hardware and specialise the OS for spatio-temporal resource allocation and give QoS guarantees to applications that can self-manage resources.
The most relevant specialised OSes to my work are Akaros [RKZ+11], CoreOS\textsuperscript{5} and Clear Linux,\textsuperscript{6} which aim to improve OS efficiency within data centres. Akaros is a new OS written from scratch, but CoreOS and Clear Linux customise the Linux kernel and user-space to reduce memory footprint and efficiently execute applications isolated in containers.

### A.2 Additional workload interference experiments

#### A.2.1 Pairwise SPEC CPU2006 interference experiments

I run two SPEC CPU2006 workloads on a 12-core AMD Opteron 4234 (“Valencia” microarchitecture, see Figure 2.5a), and assign them to CPUs such that they share different parts of the memory hierarchy. Dynamic CPU frequency scaling techniques that temporarily increase the clock rate beyond the P0 ACPI state are disabled.\textsuperscript{7} I pin benchmarks to cores (using cgroups, as used in Linux containers), and pin their memory on the local NUMA node.\textsuperscript{8}

I normalise the runtime of a measured workload under co-location to its ideal runtime on an otherwise idle machine: in other words, the result is 1.0 in the absence of any interference. Figure A.1 visualises the results as a heat map: warmer colours indicate a higher degree of interference. The colour for each entry is the makespan of the workload on the $x$-axis as a result of co-location with the workload on the $y$-axis, and workloads are ordered roughly by their frequency of memory accesses according to Merkel et al. [MSB10].

While some workloads degrade slightly even without shared caches (e.g. due to shared persistent storage, lock contention, and cache coherency traffic) in Figure A.1a, the interference increases dramatically when caches are shared.\textsuperscript{9} As Figure A.1b shows, sharing a level-2 cache causes almost all workloads to suffer when co-located with a memory-intensive workload. The worst-case degradation (off the scale in Figure A.1b) is $2.3 \times$. This is unsurprising: a workload with a large working set ends up frequently evicting cache lines of a co-located workload, even if its working set fits into the cache.
Figure A.1: Co-location heatmap on an AMD Opteron 4234: normalized runtime of x-axis workload in the presence of y-axis workload. Black squares indicate that the normalized runtime exceeded the scale; gray ones correspond to values below 0.8×.

(a) Cores 4 and 5: sharing only main memory, persistent storage and OS resources.

(b) Cores 4 and 6: sharing an L2 cache; scale capped at 1.52×.
Figure A.2: Normalised cycle counts for co-located WSC applications on the AMD Opteron 4234 (left column) and Intel Xeon E5-2420 (right column). All results are for the x-axis benchmark, normalised to its mean isolated cycle count on an otherwise idle machine. Black squares indicate normalized cycle counts exceeding the scale; gray indicates that no results are available.
Figure A.3: Normalised cache miss counts for co-located WSC applications on the AMD Opteron 4234 (left column) and Intel Xeon E5-2420 (right column). All results are for the x-axis benchmark, normalised to its mean isolated cache miss count on an otherwise idle machine. Black squares indicate normalized cache miss counts exceeding the scale; gray indicates that no results are available.
Figure A.4: Interference between multiple instances of SPEC CPU2006 benchmarks on AMD Opteron 4234 (top) and Intel Xeon E5-2420 (bottom). Values are averages and standard deviations over the normalised runtimes of ten executions of each experiment.
A.2.2 Pairwise application co-location: additional metrics

A.2.3 $n$-way SPEC CPU2006 interference experiments

In the following, I co-locate between two and twelve instances of the same SPEC CPU2006 benchmark on a machine and investigate the impact of different co-locations.\(^\text{10}\) Figure A.4 shows the benchmark runtime – normalised to the runtime on an otherwise idle machine – as they are co-located on the AMD Opteron 4234 (top) and the Intel Xeon E5-2420 (bottom). Again, the benchmarks are ordered roughly from compute-bound to memory-bound according to Merkel et al. [MSB10].

Unsurprisingly, increased machine utilisation leads to an increase in normalised runtime for those benchmarks that frequently access memory. By contrast, the compute-bound benchmarks do not suffer much as a result of co-location. In the worst cases, a $4–5 \times$ slowdown occurs.

The general trend is the same for both machines, but there are some differences:

1. Sharing an L2 cache on the Opteron 4234 impacts all benchmarks by 20–50%, even the compute-bound ones. By contrast, sharing the Intel Xeon’s much smaller L2 cache between hyperthreads has no effect. This suggests that the effect observed is not due to cache interference, but rather due to one of two other shared resources on the AMD Valencia’s “Bulldozer” micro-architecture:

   a. The clustered integer core design shares a floating point unit (FPU) between adjacent cores, akin to the Alpha 21264’s four-way integer execution [Kes99]. Many SPEC CPU2006 benchmarks make heavy use of the FPU.

   b. The shared, two-way set-associative, 64 KB $L1$ instruction cache of adjacent cores, which may cause additional instruction cache misses. By contrast, the Xeon has a dedicated, 4-way set-associative 32 KB $L1$ instruction cache per core.

2. Co-location on adjacent “hyper-threads” on the Intel Xeon does not induce additional interference over dedicating a core to each benchmark. This result contradicts prior work using SPEC CPU2000 and earlier-generation Intel processors [Bul05, pp. 44–56].

---


\(^{6}\)http://clearlinux.org; accessed 19/05/2015.

\(^{7}\)This includes “dynamic overclocking” techniques (AMD “TurboCORE” and Intel “TurboBoost”), since the non-deterministic clock speed that depends on the thermal state of the processor masks the effects of cache sharing.

\(^{8}\)Benchmark runtime increases by $\approx 33–50\%$ when working on remote memory.

\(^{9}\)In both cases, some workloads (e.g. omnetpp and xalancbmk) improve under co-location: I suspect that this is an experimental artefact related to storage access or prefetching: other investigations of SPEC CPU2006 behaviour under co-location do not observe this.

\(^{10}\)Running multiple instances of the same workload on a machine is not as naïve as it may seem: many cluster schedulers optimise for data locality, i.e. they attempt to place computations near their input data [IPC\(^*\)09; ZBS\(^*\)10]. If the input data to a large job is replicated to a handful of machines only, several tasks of the same job may end up on the same machine. Indeed, Google uses an explicit exclusivity constraint to avoid this for critical jobs [RWH11, p. 9].
3. Worst-case degradation and variability across experiments are higher on the Opteron than on the Xeon machine, possibly due to different cache sizes.

These experiments demonstrate that higher machine utilization leads to increased co-location interference, especially for memory-intensive workloads.

### A.3 CPI and IPMA distributions in a Google WSC

Figure A.5 shows that the synthetic workloads that I used in Section 6.3.1.1 (Table 6.3) are situated towards the lower end of the CPI distribution of Google workloads. The 99th percentile CPI observed in my experiments, at around three cycles per instruction, is similar to the average CPI at Google. This suggests that Google either over-commits machines to a greater extent, or that the Borg scheduler fails to adequately mitigate co-location interference (or both).

Of my workloads, three (the two `io_stream` workloads, and the out-of-LLC `mem_stream`) have similar levels of memory-intensity as common Google tasks do (IPMA values below 2,000), and the other two (`cpu_spin` and L3-fitting `mem_stream`) have much higher IPMA values that correspond to the 99th percentile for Google’s batch and service jobs.\textsuperscript{11} This suggests that a larger fraction of tasks are I/O-intensive or memory-intensive tasks with large working sets, which is hardly surprising: real-world I/O is likely more bursty than the `fio` benchmark in my workload (which saturates the entire disk bandwidth). Hence, real-world workloads likely permit co-location of I/O-bound tasks, which only increases the need for interference-avoiding solutions such as Firmament.

\textsuperscript{11}With the information available in the Google trace, it is difficult to say if Google workloads usually have poor cache-affinity (which decreases IPMA), or whether the workloads are inherently memory-intensive due to large working sets (same effect), or if the Borg scheduler fails to avoid interference in the cache hierarchy (again, having the same effect).
APPENDIX A. ADDITIONAL BACKGROUND MATERIAL

Figure A.5: Performance counter data from Google workloads in the 2011 trace, based on $n = 42,130,761$ total tasks over 30 days. The boundaries of the shaded area correspond to the distributions of the per-task minimum and maximum observed values; the thick line represents the distribution of averages.

(a) Batch workloads, cycles per instruction (CPI). Valid $n = 39,193,353$.

(b) Service workloads, cycles per instruction (CPI). Valid $n = 75,227$.

(c) Batch workloads, instructions per memory access (IPMA). $99.99^{\text{th}}$ percentile at 34,482.76; valid $n = 36,320,401$.

(d) Service workloads, instructions per memory access (IPMA). $99.99^{\text{th}}$ percentile at 9,684.85; valid $n = 72,678$. 

*Note:* The graphs show cumulative distribution functions (CDFs) for CPI and IPMA, with the shaded areas indicating the range of observed values and the thick line representing the average values. The data points are derived from 42,130,761 total tasks over 30 days, with valid counts of 39,193,353 for batch workloads and 75,227 for service workloads. The $99.99^{\text{th}}$ percentile for batch workloads is 34,482.76, and for service workloads, it is 9,684.85. The average values are also indicated on the graphs.
Appendix B

Additional DIOS material

B.1 DIOS system call API

B.1.1 create(2)

The create(2) system call is used to generate entirely new DIOS objects (and, consequently, names). It is the only system call that can generate objects and names; every name must once have been returned by a create(2) invocation.

create(2) typically takes two arguments: a reference to the group in which the new object is to be created, and a flag indicating the object type. Additionally, an optional host ID, optional and type-specific flags and parameters can be passed, and a flag controls whether the object is named or anonymous (§4.3.1).

The return value is a tuple consisting of a fresh name (unless an anonymous object is created) and a reference to the new object. If the object is not anonymous, the new name is copied into a user-space buffer, while the reference is mapped read-only into the user-space task’s address space and a pointer returned.

\[
\text{create}(\mathcal{R}_g, \mathcal{P}_{\text{args}}, \mathcal{H}_{\text{id}}, \mathcal{F}) \rightarrow (\mathcal{N}_o, \mathcal{R}_o)
\]

The create(2) system call, called from task T, takes an optional reference to a group object \(g\), \(\mathcal{R}_g\), indicating the group in which the new object is to be created. If no group reference is passed, the object is created in T’s primary group. create(2) returns a tuple consisting of the new object \(o\)’s external name, \(\mathcal{N}_o\), and a reference to the object, \(\mathcal{R}_o\).

The kernel also computes an internal name, \(\mathcal{N}_o^i\), for the new object and maps it to the object structure via the local name table, \(\Gamma_N\). Finally, the reference mapping \(\mathcal{R}_o \mapsto \mathcal{K}_o\) is inserted into the task’s reference table, \(\Gamma_R^T\).
APPENDIX B. ADDITIONAL DIOS MATERIAL

The reference returned grants full permissions to the task creating the object, and the reference type and other parameters are set according to the initialisation flags in $\mathcal{F}$.

Note that $\text{create}(2)$ does not necessarily allocate any I/O buffers or object state. The tuple returned only contains the identifier and handle for the object; buffers to interact with it and other state are typically set up lazily on the first I/O request. In other words, $\text{acquire\_read}(2)$ and $\text{acquire\_write}(2)$ must be employed after $\text{create}(2)$ to effect any I/O on the object.

An interesting bootstrapping challenge arises from the fact that names are stored in user-provided memory (unlike references). The names of the initial objects in a program must therefore be stored in static memory or on the stack; a typical DIOS “pattern” is to allocate a private memory object on startup that ends up storing further objects’ names.

B.1.2 lookup(2)

The ability to locate objects by resolving their names to locations is an important part of I/O DIOS. I described the name resolution process in Section 4.3.3; it is implemented by the $\text{lookup}(2)$ system call.

$\text{lookup}(2)$ takes a name as its argument and returns a set of references to reachable object instances corresponding to this name. Reachable in this context means

(a) that the object is named (i.e. not anonymous),
(b) that the object is in a group that the calling task is a member of,
(c) that the object instance in question is either local, or
(d) if the instance is remote, that it exists on a machine which can receive DCP messages from the local machine and responds to them (i.e. it has not failed, and no network partition separates it from the task’s local machine).

All reachable object instances found then have references generated for them. The reference generation applies similar transformation to a reference delegation (see §4.5.2) to adapt the reference attributes for the local context. The resulting new references are inserted into the local task’s reference table.

In other words, $\text{lookup}(2)$ is defined as follows:

\[
\text{lookup}(\mathcal{N}_o, \mathcal{F}) \rightarrow \{ \mathcal{R}_o \mid o \in \text{reachable}(\mathcal{N}_o, G_T) \}
\]

\[
\text{lookup}(\mathcal{N}_o, \mathcal{R}_g, \mathcal{F}) \rightarrow \{ \mathcal{R}_o \mid o \in \text{reachable}(\mathcal{N}_o, g) \}
\]

The $\text{lookup}(2)$ system call invoked by task $T$ takes an external name, $\mathcal{N}_o$, as its argument and returns a set of references corresponding either (i) to all reachable objects $o$ named by $\mathcal{N}_o$ in all of $T$’s groups $g \in G_T$; or (ii) to all reachable objects $o$ in group $g$, if an explicit $\mathcal{R}_g$ is specified.
To do so, the DIOS kernel computes either (i) the internal names for \( N_0 \) in each group \( g \) that \( T \) is a member of, or (ii) the internal name for \( N_0 \) in \( g \), respectively, by computing \( N_g^0 = \mathcal{H}(N_0 \parallel g) \), and looks up \( N_g^0 \) in the name table, \( \Gamma_N \).

As implied in above definitions, \texttt{lookup(2)} makes no guarantee to return all instances of the object corresponding to the name specified in the argument, but only reachable ones. In other words, name resolution for remote objects is best-effort, although a reliable interconnect can help increase the confidence of comprehensive results being returned.

Additionally, \texttt{lookup(2)} merely guarantees to return references for all instances which were reachable at the point of handling the system call. This may include stale information: the instances found may have been deleted or new ones created when the system call returns (see §4.7).

### B.1.3 copy (2)

As explained in Section 4.5.2, references can be delegated to other tasks. This is facilitated by the \texttt{copy(2)} system call.

\texttt{copy(2)} takes two arguments: a set of one or more references to delegate, and a reference to the target task that they are to be delegated to. The second reference must refer to a task; if it does not, the \texttt{copy(2)} invocation fails.

The return value indicates whether the delegation succeeded. Note that the delegating task does not itself gain access to the delegated reference, unless it is delegating to itself.

\[
\text{copy} \left( \{R_0, \ldots, R_k\}, R_{T_d}, F \right) \rightarrow \text{bool}
\]

The \texttt{copy(2)} system call, when invoked by task \( T_c \), takes a set of references, \( \{R_0, \ldots, R_k\} \), and delegates transformed copies of them to the task \( T_d \) referred to by \( R_{T_d} \).

The newly created references are inserted into \( T_d \)'s reference table, and a message notifying \( T_d \) of the delegation is enqueued to be read from its self-reference.

copy (2) is a complex and powerful system call. It checks the validity of the reference delegation requested, transforms the references into new references as appropriate for the target context, and communicates the delegation to the target task and its managing kernel. Transformations are described in Section 4.5.2, and may involve copying the object data.

Tasks can delegate references to themselves in order to create transformed versions of them; in practice, this usually happens in order to pass the references to a different task, however, and direct delegation is preferable.

There are several possible future extension to the \texttt{copy(2)} system call:
1. It can be used to dynamically add objects to groups by permitting the target reference to refer to a group object. Instead of inserting a new reference into a task’s reference table, this invocation on a reference to object \( o, \mathcal{R}_o \), with external name \( \mathcal{N}_o \), would insert an internal name \( \mathcal{N}_g^o \) for the target group \( g \) in the name tables. In order to maintain security, however, only objects from groups that the calling task is a member of can be copied.

2. “Move semantics” for reference delegation might be supported, allowing a reference to be delegated to another task with the caller losing access to it.

These extensions are not currently implemented in the DIOS prototype, but are in principle compatible with the current API.

### B.1.4 delete(2)

While \texttt{create(2)} is responsible for creating names and references, there must also be a way of destroying them when no longer required. Note that the deletion of handles (references), identifiers (names), and objects are separate concerns.

The DIOS \texttt{delete(2)} system call is used to remove references from the current task’s reference table, i.e. deleting the reference. Deletion of a reference only mandatorily destroys the handle, however: what happens to the object referred to depends on its life cycle type (§4.2):

- A reference-counted object has its reference count decremented and is destructed once it reaches zero; at this point, it may either be reclaimed immediately or continue to exist in an orphaned state for asynchronous garbage-collection.

- If the object is one-off deleted, the object is remove atomically, and any other references to it become invalid and system calls on them fail.

- For an immortal object, nothing happens.

The object’s name is deleted from the name table once the object or the last reference to it is removed.

Under ordinary circumstances (i.e. in the absence of failures), \texttt{delete(2)} always returns true if a valid reference \( \mathcal{R}_o \) is passed.

Hence, the \texttt{delete(2)} system call is defined as follows:

\[
delete(\mathcal{R}_o, \mathcal{F}) \rightarrow \text{bool}
\]

The \texttt{delete(2)} system call takes a reference, \( \mathcal{R}_o \), to an object \( o \), and deletes it from the calling task \( T \)'s reference table, \( \Gamma_R^T \).
A deletion handler specific to object o’s life cycle type is invoked before returning, and any outstanding I/O requests on the reference being deleted are implicitly aborted (i.e. terminated without commit).

Finally, the delete(2) call returns a success indication.

Since delete(2) directly mutates the state of an existing object and affects the validate of names and references for it, race conditions with other system calls are subtle. Consider, for example, delete(2) racing with an invocation of copy(2): if the deletion is handled first, the delegation effected by copy(2) will fail as the reference no longer exists. However, if the deletion is handled first, it succeeds and the reference is subsequently deleted.

In other words, deletion of a reference does not guarantee that delegated copies of this reference do not continue to be made after the call to delete(2) is made (although this is guaranteed after delete(2) returns). A consequence of these semantics is that running lookup(\(N_o\)) to obtain the set of references corresponding to \(N_o\) and then invoking delete(\(R^i_o\)) on each reference \(R^i_o\) returned does not guarantee that all instances described by these references will be deleted.

As the deletion always affects the task-local reference table only, no other delete(2) call on the same reference can race in the network; local races within the task are idempotent, although the second call to be serviced may return false, indicating a now-invalid reference was passed.

Finally, a task may delete its self-reference using delete(2). This is a special case: it has the effect of terminating the task (and thus does affect the object). However, the task is not completely removed until all other references to it are destroyed. It remains in a “zombie” state (no longer executing and unable to communicate) until this is the case.

Finally, an exiting task implicitly invokes delete(2) on all of the references in its reference table. Since the table is destroyed immediately after, this is only consequential as other nodes may need to be notified of the reduced reference count for the objects.

### B.1.5 run(2)

A task can invoke the run(2) system call in order to spawn a further task. The executable argument must be a reference to a durable blob or a blob of executable memory. If the reference does not have the executable permission set, run(2) fails. The new task is a child of the current task.

Since a task’s group memberships are determined at creation time, the run(2) system call also specifies the group memberships to be inherited by the newly created child.

Two arguments to run(2) control this:
1. A set of references to group objects representing groups of which the child task is to be granted membership. Any group object available to the parent task which was not obtained by group name resolution can be a member of this set. This argument is optional; if it is not specified, an empty set is assumed.

2. A reference to the group object representing the primary group for the child task. This argument is optional; if it is not specified, a new group object is generated for the primary group. If specified, the group object must be a member of the set of groups whose membership the child task inherits (see above).

If none of the flags is specified, default behaviour is to give membership of a single, newly generated group to the spawned task.

The \texttt{run(2)} system call returns a reference to the newly spawned task object, and is defined as follows:

\[
\text{run}(R_{\text{bin}}, S_G, R_{pg}, P_{\text{info}}, F) \rightarrow R_{\text{child}}
\]

The \texttt{run()} system call, when invoked by parent task \(T_P\), takes a reference to an executable object (a binary), \(R_{\text{bin}}\), a set of references to group objects, \(S_G\), and a reference to a single group object, \(R_{pg}\), as arguments. It causes a child task, \(T_C\), to be created.

A reference to the new task object for \(T_C\), \(R_{\text{child}}\) is returned.

The child task’s set of group memberships, \(G_C\), is set to \(S_G\), which is defined as:

\[
S_G = \{ R_g \mid R_g \in \Gamma^p_R \text{ and } (g \in G_P \text{ or } g \in C_P) \},
\]

where \(G_P\) stands for the parent task’s group memberships and \(C_P\) for the groups created by the parent task.

The child task’s primary group, \(P_C\), is set as follows:

\[
P_C = \begin{cases} 
\text{create\_group()} & \text{if } R_{pg} \text{ not specified;} \\
\text{the } g \in S_G \text{ that } R_{pg} \text{ refers to} & \text{otherwise.}
\end{cases}
\]

where \texttt{create\_group()} brings a new group object into existence and makes it available to the new task.

Note that as a result of these semantics, the child task can resolve either:

1. a subset of the names that the parent can resolve (if only groups that the parent task is a member of are passed);

2. a disjoint set of names compared to those that the parent resolve (if no groups or only groups created by the parent are passed);
3. a superset of the names that the parent can resolve (if all groups that the parent task is a member of are passed and all groups created by the parent task are passed);
4. a partially overlapping set of names with those that the parent can resolve (if a mixture of inherited and created groups are passed).


dios tasks may be placed by the long-term cluster task scheduler, explicitly spawned on a specific remote machine, or constrained to running locally on the same machine as their parent. Hence, run(2) can have two different effects depending on the flags specified:

1. If it is invoked with the SPAWN_LOCAL flag set, the new task is started on the same machine. The Dios kernel creates a new local task and inserts it into the local task scheduler’s runnable set.
2. If the SPAWN_LOCAL flag is not present, and a specific host ID is set in the Pinfo structure, a DCP message is sent to spawn the task on the host specified.
3. Finally, if SPAWN_LOCAL is not set, and no explicit host ID is specified, message is sent to the cluster scheduler (Firmament), which will place the task and execute it by invoking run(2) with appropriate arguments.

Either way, the parent task is blocked until the child task is running unless the NON_BLOCKING flag is set.

The newly created task starts out with the default set of initial references and an otherwise empty reference table. It does not share any of its parent’s references unless they are explicitly delegated to it subsequently.

**B.1.6 pause(2) and resume(2)**

When a Dios application needs to suspend itself or another task, it can use the pause(2) system call. Likewise, resume(2) can be used to continue executing a suspended task.

**pause(2).** The argument to pause(2) must be a reference to a task object; if it is not, an error is returned. Dios changes the referenced task’s state to “suspended” and deschedules it if it is running.

\[
\text{pause}(R_T, F) \rightarrow \text{bool}
\]

The pause(2) system call pauses the execution of a task T. It takes a reference to T’s task object, \( R_T \), as its argument. The kernel notifies the Dios kernel on the machine running T to suspend its execution. If T is already suspended or it has exited, the call has no effect.

If T was suspended successfully, the return value is true; otherwise, it is false.
Invoking \texttt{pause(2)} on a task’s self-reference ($R_{self}$) is a special case. It has the effect of \textit{yielding} the processor with immediate effect. If the \texttt{STAY\_RUNNABLE} flag is set, the task stays runnable. Otherwise, it is suspended until \texttt{resume(2)} is invoked on it by another task.

\texttt{resume(2)}. A suspended task $T$ can be continued by any other task invoking the \texttt{resume(2)} system call with $R_T$ as its argument. \texttt{resume(2)} is the counterpart to \texttt{pause(2)} and its operation is analogous. The only difference is that a task cannot invoke \texttt{resume(2)} on a reference to itself.

In practice, \texttt{pause(2)} and \texttt{resume(2)} calls are primarily used by the scheduler. However, in combination with an extended \texttt{copy(2)} system call, they could also support task migration: a task would be paused, copied, the original deleted and the new copy resumed.\footnote{The current DiOS prototype does not yet support copying of task objects, but well-known process migration techniques such as checkpoint-restart (implemented e.g. by BLCR [DHR02]) can be used to add this facility.}

**B.1.7 acquire\textunderscore read(2)**

The \texttt{acquire\textunderscore read(2)} system call initiates a read-only I/O request. It takes a reference to an object $o$ as its argument and returns an I/O vector (a buffer of a defined length) for I/O on this reference or an error indication.

When invoked, \texttt{acquire\textunderscore read(2)} attempts to read the specified number of bytes (or as many bytes of data as possible, if \texttt{size} is zero) from the object $o$ referred to by $R_o$.

\texttt{acquire\textunderscore read(2)} is a \textit{buffer-acquiring} system call: when it returns, the kernel supplies the user application with a read buffer containing object data. Several buffer management options are available by passing appropriate flags:

1. \texttt{ACQUIRE\_IOV\_CREATE} (default): creates a buffer for the object. When reading, this buffer holds data; when writing, it may hold data for blobs, but not for streams.

2. \texttt{ACQUIRE\_IOV\_TAKE}: accepts an existing buffer as an argument, acquires ownership of it and uses it; the buffer is deleted on release.

3. \texttt{ACQUIRE\_IOV\_BORROW}: accepts an existing buffer and holds it until it is released, but does not acquire ownership. When \texttt{release\textunderscore read(2)}/\texttt{release\textunderscore write(2)} is called, the buffer is returned to the original owner, but not destroyed.

4. \texttt{ACQUIRE\_IOV\_REUSE}: reuses an existing, already associated buffer to be passed and records the start of a new I/O request on it (e.g. by copying a current version number).

5. \texttt{ACQUIRE\_IOV\_NONE}: does not associate a new or existing buffer but sets up reference I/O state. Subsequent calls may create, move, or borrow buffers.
The second option (ACQUIRE_IOV.Take) is useful for zero-copy I/O, as it allows moving buffers obtained from an earlier acquire operation on a different reference. The fourth option (ACQUIRE_IOV_REUSE) allows the caller to use a buffer multiple times, which significantly reduces the number of memory mappings required compared to a full acquire-use-release cycle.

All buffers are allocated by the kernel: this permits different underlying mechanisms to expose data in different ways. For example, a shared memory area may expose copy-on-write pages in the buffer, while a high-performance network transport (e.g. DPDK, netmap [Riz12]) accessing a remote object may expose NIC buffers directly. On the other hand, a double-buffered shared memory ring between tasks may copy the data into a new temporary buffer to enable receipt of additional data.

In other words, the acquire_read(2) system call can be specified as follows:

\[
\text{acquire_read}(R_o, \text{size}, F) \rightarrow (P, \text{size})
\]

The acquire_read(2) system call attempts to initiate a read-only I/O request on object \(o\) referred to by \(R_o\). The optional size parameter specifies how the amount of data that must be available before the call returns; if it is unset, all available data is read.

If the desired concurrent access semantic (specified via flags) can be satisfied, the I/O request is registered by incrementing the object’s active read request counter (§3.7.1).

If a new buffer is instantiated (i.e. the ACQUIRE_READ_IOV_CREATE flag is set), it is mapped into the calling task’s virtual address space.

acquire_read(2) returns a tuple \((P, \text{size})\) that contains a pointer to the buffer \(P\) and an indication of its length \(\text{size}\).

The underlying I/O mechanism can be based either on a streaming abstraction (as in most network transports, or FIFO IPC between tasks) or on a fixed-size blob abstraction (§4.2). In the blob case, the buffer returned typically corresponds to the entire object or a subset thereof, although an optional argument can specify an offset. For streams, the offset argument ends up discarding data until the offset is reached.

Any data read from the buffer returned by acquire_read(2), however, are not definitely valid until commit_read(2) has been called to complete the I/O request. Only if commit_read(2) indicates that the read was valid under the read consistency level of \(R_o\), the data can be treated as valid under the specified concurrent access semantics. If an application does not depend on consistency, it may use data from the read buffer directly; if it does depend on the integrity of the data read, it must copy the buffer and call commit_read(2) before using it.
B.1.8 \texttt{commit\_read(2)}

A DIOS I/O request is not valid until it has been committed. On read, a successful commit confirms that the data supplied at the acquire stage are still valid.

For read-only I/O requests, this is the role of the \texttt{commit\_read(2)} system call. It validates the I/O request and informs the user-space application by returning an error if the data read have been affected by another concurrent I/O request (usually a write) under the request’s concurrent access semantics.

By default, \texttt{commit\_read(2)} expects to be given a buffer already associated with the reference \(R_o\) passed, but flags can customise this:

1. COMMIT\_IOV\_USE (default): accepts an associated buffer and commits it; the buffer remains associated and ready for re-use.

2. COMMIT\_IOV\_MOVE: accepts an existing buffer from a different reference, takes ownership of it, and commits it before returning.

In the second case, the commit’s return value indicates the validity of the buffer with regards to the original reference that it was acquired on. No indication of the validity of the buffer is given with regard to the target reference – an acquire-commit cycle must be completed if this is needed.

The \texttt{commit\_read(2)} system call is thus defined as follows:

\[
\texttt{commit\_read}(R_o, \langle P, \text{size} \rangle, F) \rightarrow \text{bool}
\]

The \texttt{commit\_read(2)} system call takes a buffer and attempts to commit it, returning a validity indication.

If the commit succeeds, the data read were definitely valid, and the object’s read version counter is atomically incremented. If the commit fails, the read version counter is not incremented.

In either case, the object’s active reader count is atomically decremented.

The precise semantics of a failed commit depend on the underlying object. Some underlying I/O mechanisms may not guarantee the integrity of a buffer while it is shared with the user-space application: consider, for example a shared memory area that can be written to by another task. While failed commits indicate such a concurrent access in excess of the permissible semantics, the application may have read corrupt data. It is the application’s responsibility to ensure that this failure can either be recovered from, or that synchronisation is employed such that this situation cannot occur.
B.1.9 release_read(2)

Since buffers are a limited resource, they must eventually be returned to the kernel for reuse. The release_read(2) system call cleans up and tears down any related state for a read buffer. If the buffer is borrowed, it is returned to its previous owner; if it is owned, it is de-allocated.

release_read(2) normally returns true; only invalid arguments can lead to an error. The system call is thus defined as follows:

\[
\text{release_read}(\mathcal{R}_o, \langle P, \text{size} \rangle, \mathcal{F}) \rightarrow \text{bool}
\]

The release_read(2) system call returns an active buffer to the kernel.

The buffer at \(P\) is invalidated and unmapped from user-space virtual memory.

release_read(2) can typically be handled entirely locally, even if the object is remote, since the buffer contains local state only. After release_read(2) is called, the buffer passed is no longer valid for user-space access, even though it may be unmapped asynchronously. The kernel may, however, re-use the buffer immediately, returning it from another buffer-acquiring system call.

B.1.10 acquire_write(2)

Like acquire_read(2), the acquire_write(2) system call is a buffer-supplying call. However, instead of returning a buffer containing data available for reading, it returns a – blank or pre-populated – buffer for writing. The application then copies its data into the buffer, or generates them directly in the buffer. Finally, commit_write(2) is called to finalise the output request and check its validity.

The size of the buffer is requested by the user is passed as an optional argument, the default being \(\mathcal{R}_o\)’s write_buf_size attribute (if set). The system call may return a larger buffer than requested; for example, it may page-align the buffer for easier mapping to user-space.

The definition of the acquire_write(2) system call is similar to that of acquire_read(2):

\[
\text{acquire_write}(\mathcal{R}_o, \text{size}, \mathcal{F}) \rightarrow \langle P, \text{size} \rangle
\]

The acquire_write(2) system call attempts to initiate a write I/O request on object \(o\) referred to by \(\mathcal{R}_o\). The optional size parameter specifies the amount of data that will be written; if it is unset, a default-sized buffer is returned.

If the desired concurrent access semantic (specified via flags) can be satisfied, the I/O request is registered by incrementing the object’s active write request counter (§3.7.1).
If a new buffer is instantiated (i.e. the ACQUIRE_IOV_CREATE flag is set), it is mapped into the calling task’s virtual address space.

acquire_write(2) returns a tuple \( \langle P, \text{size} \rangle \) that contains a pointer to the buffer \( P \) and an indication of its length \( \text{size} \).

After acquire_write(2) returns, the write buffer may be mutated by the user-space application; it may also read from the buffer, although the buffer’s initial contents are dependent on the object type.

Depending on the reference’s write consistency level, other concurrent modifications to the buffer may be visible to the calling task, and may overwrite changes made by it.

B.1.11 commit_write(2)

As with read I/O requests, write requests are not valid until a successful commit confirms that the desired concurrent access semantics held for the duration of the I/O request.

By default, commit_write(2) expects to be given a buffer already associated with the reference \( R_o \) passed, but as with commit_read(2), this can be customised. In addition to the COMMIT_IOV_USE and COMMIT_IOV_MOVE flags, commit_write(2) also supports a temporary borrowing flag:

- COMMIT_IOV_BORROW: accepts an existing buffer and borrows it for the duration of the commit only, returning it to the owner reference afterwards.

This is useful in order to quickly write a buffer to multiple objects (e.g. network streams) without having any intention of re-using it with any of them.

As with acquire_read(2), the specification of commit_write(2) is similar to its read equivalent, commit_read(2):

\[
\text{commit_write}(R_o, \langle P, \text{size} \rangle, \mathcal{F}) \to \text{bool}
\]

The commit_write(2) system call takes a buffer and attempts to commit it, returning a validity indication.

If the commit succeeds, the data in the buffer at \([P, P + \text{size}]\) were definitely written to the object under the desired concurrent access semantics, and the object’s write version counter is atomically incremented. If the commit fails, the write version counter is not incremented, and the write state of the data depends on the object type.

In either case, the object’s active writer count is atomically decremented.
As with reads, the precise semantics of a failed write commit depend on the underlying object type. Some underlying I/O mechanisms (e.g. a shared memory area) may see parallel writes of buffer while it is shared with the user-space application. While failed commits indicate such a concurrent access in excess of the permissible semantics, writes to the buffer may be affected the object even though the commit failed.\footnote{Techniques like shadow copies and copy-on-write paging can be employed to avoid this in the object-level I/O implementation, but DIOS does not mandate them in the API.} As with reading, it is the application’s responsibility to tolerate this failure or perform synchronisation such that this situation cannot occur.

\subsection*{B.1.12 release_write(2)}

Once a user-space program has completed and committed the output written into a write buffer (supplied by \texttt{acquire_write(2)}), the buffer must eventually be returned to the kernel when it is no longer needed.

As with \texttt{release_read(2)}, this operation is defined as follows:

\begin{verbatim}
release_write(\mathcal{R}_0,(\mathcal{P}, \text{size}), \mathcal{F}) \rightarrow \text{bool}
\end{verbatim}

The \texttt{release_write(2)} system call returns an active buffer to the kernel.

The buffer at \(\mathcal{P}\) is invalidated and unmapped from user-space virtual memory.

In most cases, a \texttt{release_write(2)} system call – unlike the acquire and commit calls – can be handled entirely locally; only an implicit commit (\texttt{RELEASE_IOV_COMMIT}) may require a remote operation.

\subsection*{B.1.13 select(2)}

When an application performs I/O on multiple references, it may need to determine which reference to service next. This functionality is typically implemented using \texttt{select(2)} loops in conventional OSes.\footnote{Alternative systems with slightly different semantics – e.g. Linux’s \texttt{epoll} notifications – are also used.}

The DIOS \texttt{select(2)} system call implements synchronous parallel waiting on multiple references, returning the reference that becomes available first. It is defined as follows:

\begin{verbatim}
select(\{\mathcal{R}_0, \ldots, \mathcal{R}_k\}, \text{mode}, \mathcal{F}) \rightarrow \mathcal{R}_i
\end{verbatim}

The \texttt{select(2)} system call returns the first reference that has data available for I/O of \texttt{mode} (read, write) out of the set of references, \(\{\mathcal{R}_0, \ldots, \mathcal{R}_k\}\), passed to it. The caller is blocked until one of the references in the set becomes ready for I/O.
The most common use of `select(2)` involves asynchronous servicing of multiple stream-type references, either for reading or writing. The abstraction is useful, as it avoids blocking the caller when no data are available, and because it is more efficient than polling.

When a reference passed refers to a remote object, a DCP message is sent to the remote kernel to inform it of the reference being in a selector. When the object becomes ready for I/O, the remote kernel sends a notification to all kernels that have references to the object in selectors.\(^4\)

Finally, it is worth noting that `select(2)` can easily be extended to support timed waiting by passing a reference to a timer object which becomes ready after a certain time has elapsed.

### B.2 Dìos I/O requests

Six of the Dìos system calls are concerned with I/O requests on objects, as described in Section 3.7.1. To perform I/O on an object, the caller must first acquire a buffer (represented as a tuple of a pointer and a length) via the `acquire_read(2)` or `acquire_write(2)` system call. The buffer may then be committed one or more times (using `commit_read(2)` or `commit_write(2)`), and is eventually released via `release_read(2)` or `release_write(2)`.

Each reference \(R_o\) has collections of associated buffers for reading and writing, with each buffer either:

1. owned by \(R_o\), or
2. borrowed by \(R_o\) from another reference.

There are categories of data-holding objects (§4.2), which have different I/O semantics:

1. blob objects (e.g. private/shared memory blobs, durable blobs), and
2. stream objects (e.g. shared memory and network streams).

When a buffer is acquired on a blob, an appropriate memory representation of the blob is exposed to user-space (similar to memory-mapped I/O in classic OSes); how this representation maps onto the underlying object is type-specific.

By contrast, acquiring a buffer on a stream results in a buffer that represents a chunk of data read from, or written to, the stream in FIFO order.

---

\(^4\)If references to the object exist in multiple selectors, all are notified; the relevant tasks may then start I/O requests that race for the data unless they synchronise otherwise.
void set_value_excl(dios_ref_t* val_ref, char* new_value, int val_size) {
    dios_iovec_t* iov = NULL;
    int ret = 0;
    /* Set up write with buffer */
    dios_acquire_write(val_ref, D_ACQUIRE_WRITE_IOV_CREATE, &iov);
    do {
        ret = dios_acquire_write(D_ACQUIRE_WRITE_IOV_REUSE | D_IO_EXCL,
                                val_ref, val_size, &iov);
        /* Try again if we failed to acquire at EXCL */
        if (ret < 0)
            continue;
        /* Modify data ... */
        memcpy(iov->buf, new_value, val_size);
        /* Check for concurrent modification */
        ret = dios_commit_write(D_COMMIT_WRITE_IOV_USE, val_ref,
                                val_size, iov);
        if (ret == 0)
            /* Write was valid under EXCL concurrent access, done */
            break;
    } while (true);
    /* De-allocate buffer */
    dios_release_write(D_NONE, val_ref, &iov);
}

Listing B.1: Setting a backend key-value store entry’s value via a DIOS I/O request with exclusive (EXCL) concurrent access: modification of val_ref is retried until it applies without any readers or writers present.

B.2.1 Consistency

The purpose of the validity indication returned from a DIOS I/O request’s commit is to indicate to the caller whether any invalidating concurrent accesses to the object has taken place since it acquired the buffer. When an I/O request attempts to commit, the version numbers and concurrent request counts recorded at acquire time are compared to the current values for the object. If the values have not changed in excess of the deviation permissible under the request’s concurrent access semantic, the request atomically commits. Otherwise, a failure indication is returned.

As I outlined in Section 3.7.1, an I/O request specifies a desired concurrent access semantic on acquire. Additionally, outstanding I/O request counts and read/write version numbers can be exposed as public reference attributes (§4.5.1), which allows custom multi-object consistency policies to be implemented in user-space.

It is important to note that concurrent access semantics in DIOS only specify which types of concurrent accesses will yield invalid I/O requests. They do not make it impossible for such invalid accesses to take place, or for outstanding, uncommitted I/O requests to see the consistency of their buffers violated in excess of their concurrent access semantic. DIOS only guarantees
Listing B.2: Excerpt from the reducer’s input processing logic in the MapReduce example: the size buffer (size_buf) and the data buffer (data_buf) are re-used to amortise the cost of buffer allocation and memory mapping.

that such a violation is detected when the I/O request attempts to commit. For example, changes to a shared in-memory blob’s buffers may be immediately visible to other local readers even if they have requested exclusive access, although their I/O requests will fail. Remote I/O requests, which use separate buffers, do not observe changes to their buffers, but likewise fail to commit.

More elaborate multiple-access and consistency schemes than the four simple ones described in Section 3.7.1 are conceivable, and likely required for some applications. Many consistency policies (e.g. multi-object transactions) can be built on top of the primitives provided; others may require additional concurrent access semantics to be added to DIOS, but this is not the focus of my work.

B.2.2 Examples

To illustrate the operation of I/O requests, consider two examples:

1. Listing B.1 shows an implementation of an update to the value object in a key-value store (an in-memory blob) under exclusive access. Here, `commit_write(2)` indicates if any
concurrent reads or writes have taken place between the acquire_write(2) call and the commit. If so, the code re-acquires and keeps trying until a valid write is completed.\footnote{This implementation starves writers under high contention; in practice, a separate “lock object” might be used at application level to ensure that the writer can eventually proceed.}

2. Listing B.2 shows part of the MapReduce reducer’s input processing logic and illustrates how buffers are re-used across I/O requests to amortise setup costs.

The cost of an I/O request is non-negligible: two system calls that may trigger RPCs to a remote machine, and one local system call (the release). However, a traditional read(2)/sendmsg(2) or read(2)/write(2) cycle via a socket FD has comparable costs; and DIOS often allows amortising the costs of the acquire and release stages by re-using buffers, as illustrated in the example in Listing B.2.

## B.3 Incremental migration to DIOS

Many research operating systems require “all-or-nothing” adoption: either an application buys into the new OS’s paradigms and associated benefits, or it cannot run. As I explained in the previous section, DIOS offers more flexibility by supporting DIOS limbo and legacy binaries, which retain access to legacy OS features.

This section explains how DIOS permits applications to incrementally move to using DIOS and Firmament. Figure B.1 illustrates these different states.

**Legacy applications** are unmodified and linked against standard libraries (e.g. libc). These applications may not use DIOS abstractions or system calls, nor are they scheduled by the Firmament cluster scheduler (described in Chapter 5).

**TaskLib-enabled applications** are identical to legacy ones, except that they are scheduled via Firmament. They interact with Firmament via the TaskLib library, which runs a monitor thread that makes user-space RPCs. TaskLib is either linked with the application at compile time or dynamically linked at load time (e.g. via Linux’s LD_PRELOAD mechanism injecting an ELF c\texttt{tor}). Dynamic linking allows even unmodified legacy applications to be scheduled by Firmament.

**Limbo DIOS applications** are those which developers have started porting to DIOS, but which still require access to legacy system calls despite already using DIOS objects and system calls for some of their functionality. To deploy such applications on DIOS, their binaries must be branded with the limbo brands (§4.9.3). Limbo applications may use TaskLib to interact with Firmament.
Figure B.1: Schematic of DIOS integration with legacy infrastructure. Solid arrows indicate system calls, dotted arrows indicate RPCs.

**Pure DIOS applications** are compiled for the DIOS system call API. They may use only DIOS system calls and abstractions. Their ELF brand is set to the pure DIOS brand, and legacy abstractions are unavailable. They must interact with Firmament via a TCP_STREAM object of their own.

With this approach, an application can be incrementally ported by moving through these categories and increasingly embraces DIOS abstractions.

As part of this evolution, the benefits of DIOS are increasingly applicable, especially as tasks are increasingly restricted in the objects they can access. However, strong guarantees can only be made once the application is deployed in the form of pure DIOS binaries, since bypassing DIOS is still possible as long as legacy abstractions and system calls are available.
Appendix C

Additional Firmament material

C.1 Minimum-cost, maximum-flow optimisation

C.1.1 The min-cost, max-flow problem

The minimum-cost, maximum-flow problem is an optimisation problem on a flow network.\(^1\) Intuitively, it aims to find the cheapest way to move a given ingress volume of material to an egress destination such that \((i)\) throughput is maximised, and \((ii)\) the cost is minimised. An apt real-world example is the scheduling of goods distribution over a road network: a maximal number of goods ought to be delivered per day in the most economical way possible.

A flow network is typically expressed as a directed graph \((G)\) with cost-weighted arcs \((E)\) and vertices \((V)\), the latter optionally having a supply of flow (“sources”) or a demand for absorbing it (“sinks”).

The optimisation goal is the juxtaposition of two existing problems: finding the maximum flow from sources to sinks (the maximum flow problem) and the minimum-cost (≡ shortest) path from the sources to the sinks (the shortest path problem). Figure C.1 shows an example flow network with cost and capacity annotations, and the minimum-cost, maximum-flow solution.

A flow corresponds to a set of paths (which may include cycles) through the network. In other words, the flow is a global property of the network. By contrast, the flow \(f_e\) on an arc \(e = (v, w)\) is a local property of this arc, and must satisfy two properties:

\[
1. \quad f_e = f(v, w) \geq c_{\text{min}}(v, w) \quad \text{and} \quad f(u, w) \leq c_{\text{max}}(v, w), \quad \text{where} \quad c_{\text{min}} \quad \text{and} \quad c_{\text{max}} \quad \text{denote} \quad \text{the lower and upper limits on the flow capacity of the arc (the capacity constraint), i.e. each arc's flow must be within the permissible range bounded by its capacities); \quad \text{and}
\]

\(^1\)Minimum-cost maximum-flow is related, but not identical to the “minimum-cost flow problem”. The latter only aims to find a minimum-cost flow, rather than the maximal minimum-cost flow. The literature, however, frequently uses the two terms synonymously.
Figure C.1: Minimum-cost, maximum-flow optimisation example: vertex A is the source, vertex F is the sink; arcs are labeled as “cost, flow/capacity”. Arc width is proportional to the flow carried, and the minimum-cost, maximum-flow solution at cost 95 is highlighted in red.

2. \( f_e = f(v, w) = -f(w, v) \) (the anti-symmetry constraint), i.e. any positive flow is matched by a negative flow in the opposite direction.

If only these two properties are satisfied, the flow on the arc is a pseudoflow. If, in addition, it also satisfies flow conservation, it is a circulation:

\[
\sum_{(v, w) \in E} f(v, w) = \sum_{(v, w) \in E} f(w, v) \quad \text{for each vertex } v \in V \text{ that is not a source or sink},
\]

i.e. flow coming into a vertex must leave it again, since only sources can generate flow and only sinks can drain it.

Finally, the overall graph \( G \) must drain all flow generated at sources via one or more sinks:

\[
\sum_{w \in V} f(s, w) = d \quad \text{and} \quad \sum_{w \in V} f(w, t) = d, \text{where } s \text{ is the source vertex and } t \text{ is the sink vertex}.^2
\]

This is the required flow constraint: flow generated at the source must be drained by a sink – in other words, flow cannot vanish inside the network (other than via a sink).

The total cost of a given circulation \( C \) is equal to

\[
a(C) = \sum_{(v, w) \in E} f(v, w) \times a(v, w),
\]

where \( a(v, w) \in \mathbb{R} \) denotes the cost on arc \( (v, w) \).\(^3\) In other words, the total cost equates to the sum of, for each arc, the per-arc cost multiplied by the units of flow across the arc.

---

\(^2\)While a flow network can have multiple sources and sinks, it can always be transformed into one with a single source and sink by adding two vertices \( s \) and \( t \) that generate and drain the aggregate flow. Connecting \( s \) to all sources and all sinks to \( t \) with appropriate capacities preserves the original network.

\(^3\)Real-valued costs are possible in theory, but most efficient solvers use integer-valued costs.
C.1.2 The cost-scaling push-relabel algorithm

In the following, I summarise Goldberg’s cost-scaling minimum-cost, maximum-flow optimisation algorithm [Gol97], which forms the basis of the cs2 and flowlessly solvers for Firmament.

Definitions. In addition to the terms already defined in the previous section, several others are relevant to the cost-scaling push-relabel algorithm:

The excess at a vertex is the difference between its incoming flow and its outgoing flow. The vertex cannot yet be part of a circulation if the excess is non-zero, as this violates conservation of flows.

Residual capacity describes the remaining capacity of an arc after subtracting existing flow through it: residual\((v, w) = c_{\text{max}}(v, w) - f(v, w)\). Consequently, the residual graph is the set of arcs whose residual capacity is non-zero (i.e. that still have “spare” capacity).

ε-optimality at intermediate stages is an integer measure of the maximum factor by which the total cost of the current flow in the network is greater than the optimal minimum-cost flow. In other words, an ε-optimality of two means that the current solution is at most twice as expensive as the best solution; an ε-optimality of one indicates that the optimal solution has been found.

The price is a per-vertex quantity that is used to hold state of a partial solution in several algorithms. Specifically, the notion of a minimum-cost circulation depends on the price assignments: for a price-extended cost function \(a_p(v, w) = a(v, w) + \text{price}(v) - \text{price}(w)\), a minimum-cost circulation exists if (and only if) there exists a function price such that \(\forall (v, w). a_p(v, w) < -\epsilon \Rightarrow f(v, w) = c_{\text{max}}(v, w)\). In other words, price assigns prices to vertices in such a way that a price less than \(-\epsilon\) corresponds to each arc saturated with flow (i.e. with no residual capacity).

An admissible arc is an arc whose reduce cost is less than \(-\epsilon\). An arc becomes admissible either due to change in ε or due to a change in price at either vertex.

Algorithm. I show a simplified version of the cost-scaling approach in Algorithm C.1. Intuitively, the core iteration of the algorithm can be understood in terms of a network of commercial trade activity for a specific commodity (e.g. oil): each vertex represents a trader who buys and sells the commodity at a specified price, while the sinks represent the ultimate consumer. Naturally, it is in each trader’s interest to maximise her turnover (≡ flow) at minimal cost.
1. **Initialisation:**
   (a) set $\varepsilon = \alpha$, such that $\alpha$ is larger than the maximum cost of an arc in the flow network,
   and
   (b) initialise $Q$ as an empty FIFO queue.

2. **Iteration:** while $\varepsilon > 0$, if the flow network is not $\varepsilon$-optimal (i.e., there exists excess at
   some $v \in V$ or at least one arc has a reduce cost $<-\varepsilon$),
   (a) for each vertex $v \in V$,
      i. send the maximum flow possible on all admissible outgoing arcs.
   (b) for each vertex $v \in V$,
      i. append to $Q$ if $v$ has any excess.
   (c) while $Q$ is not empty, pull $v$ from its head,
      i. if $v$ has any outgoing admissible arcs, *push* flow along those arcs until either:
         A. all excess has been drained; or
         B. no more admissible outgoing arcs exist.
      ii. if $v$ still has excess and there are *no* admissible arcs left, *relabel* $v$:
         A. reduce the price of $v$ by $\varepsilon$,
         B. make admissible any arcs whose reduce cost is now negative,
         C. repeat from 2(c)i.
   (d) once no vertex has excess and there are no admissible arcs with a reduce cost $\leq -\varepsilon$,
      i. divide $\varepsilon$ by $\alpha$, and
      ii. repeat from 2a.

3. **Termination:** return the $\varepsilon$-optimal flow.

**Algorithm C.1:** Simplified outline of Goldberg’s cost-scaling push-relabel algorithm for minimum-cost, maximum-flow optimisation [Gol97].

Excess stock at a particular trader motivates the trader to attempt to sell her stock onwards,\(^4\) which she does as long as her business partners (≡ neighbours) are willing to buy at the asking price (step 2(c)i in Algorithm C.1).

When no business parter is willing to buy at the asking price, but further excess stock remains, the trader must reduce her price in order to keep selling (step 2(c)ii). Once she does so, her business partners may become interested again. As a side-effect, however, the trader also reduces her buying price, and thus becomes a less attractive customer to her own suppliers, thereby reducing her expected future stocking levels.

In practice, this algorithm is too slow unless it is improved by the application of heuristics that reduce the number of push and relabel operations. For example, the cs2 solver relies

---

\(^4\)In this fictive trade network, no trader ever has an interest in *storing* stock: she either consumes it herself, or is willing to sell it onwards at *any* price. It follows that no profit is ever made in this capitalist venture!
on arc fixing, which only rarely considers arcs that are unlikely to change; price refinement, which decays the $\varepsilon$-optimality criterion more rapidly; push lookahead, which invokes an early “relabel” operation to avoid flow becoming “stuck”; and price update, which updates prices at many vertices in one go [Gol97].

**Future algorithmic improvements.** Progress on the minimum-cost, maximum-flow optimisation problem is still being made, often based on improvements to solvers for the underlying maximum-flow problem [GT14]. For example, Király and Kovács in 2012 showed that Goldberg’s improvements to the push-relabel algorithm for maximum flow [Gol08] also apply to the minimum-cost, maximum-flow problem [KK12].

A recent maximum-flow algorithm by Orlin has worst-case $O(VE)$ time complexity [Orl13], improving on a prior bound of $O(VE \log_{\varepsilon/\text{avg}}(V))$ by King et al. [KRT94]. Further improvements to the leading minimum-cost, maximum-flow solvers may follow from this result.

### C.2 Flow scheduling capacity assignment details

Each arc in a flow network has a capacity within a range $[\text{cap}_{\min}, \text{cap}_{\max}]$ bounded by the minimum and maximum capacity. In Firmament, as in Quincy, $\text{cap}_{\min}$ is generally zero, while the value of $\text{cap}_{\max}$ depends on the type of vertices connected by the arc and the cost model.\(^5\) Table C.1 lists common capacity assignments for combinations of vertices.

Each task vertex ($T_i$) generates a single unit of flow. Thus, all arcs exiting from a task vertex have unit capacity, independent of whether they point to the cluster aggregator ($X$), an unscheduled aggregator ($U_j$), or a rack or machine vertex ($R_k$, $M_l$).

Similarly, at the far end of the network, each arc from a machine vertex ($M_l$) to the sink ($S$) has a fixed capacity $K$.\(^6\) This corresponds to the number of tasks that may be scheduled on each machine in the WSC. Accordingly, the capacity of the arc from a rack vertex ($R_k$) to each of its subordinate machine vertices might be set to $K$, and the capacity of the arc from the cluster aggregator to the rack vertex might be set to $mK$, where $m$ is the number of machines in the rack.

Any excess flow that corresponds to unscheduled tasks (i.e. generated flow $\geq rmK$, where $r$ is the number of racks) must be drained via the unscheduled aggregator vertices. Firmament, like Quincy, specifies a minimum number of tasks that must be scheduled for a job $j$ at all times ($E_j$). It also specifies a maximum number that may be scheduled concurrently ($F_j$). Clearly, $0 \leq E_j \leq F_j \leq N_j$, where $N_j$ is the total number of tasks in job $j$. Enforcing $E_j \geq 1$ guarantees

---

\(^5\)For simplicity, I use “the capacity” to refer to the maximum capacity value in the following.

\(^6\)Quincy actually sets $K = 1$ [IPC’09, app., p.275], although the possibility of sharing machines is discussed in the paper [IPC’09, §8]. A modern WSC composed of many-core machines requires $K > 1$. 
APPENDIX C. ADDITIONAL FIRMAMENT MATERIAL

Figure C.2: The flow network in Figure 5.1 with example capacities on the arcs.

<table>
<thead>
<tr>
<th>Edge</th>
<th>Capacity</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_i \rightarrow M_l$</td>
<td>1</td>
<td>Specific machine preference.</td>
</tr>
<tr>
<td>$T_i \rightarrow R_k$</td>
<td>1</td>
<td>Rack preference.</td>
</tr>
<tr>
<td>$T_i \rightarrow X$</td>
<td>1</td>
<td>Wildcard, may schedule anywhere.</td>
</tr>
<tr>
<td>$T_i \rightarrow U_j$</td>
<td>1</td>
<td>Possibility of remaining unscheduled.</td>
</tr>
<tr>
<td>$R_k \rightarrow M_l$</td>
<td>$K$</td>
<td>For $K$ tasks on the machine.</td>
</tr>
<tr>
<td>$X \rightarrow R_k$</td>
<td>$mK$</td>
<td>For $m$ machines in the rack.</td>
</tr>
<tr>
<td>$M_l \rightarrow S$</td>
<td>$K$</td>
<td>For $K$ tasks on the machine.</td>
</tr>
<tr>
<td>$U_j \rightarrow S$</td>
<td>$F_j - E_j$</td>
<td>For $1 \leq E_j \leq F_j \leq N_j$; $N_j$ is the number of tasks in job $j$.</td>
</tr>
</tbody>
</table>

Table C.1: Edge capacity parameters for different arc types in the Quincy scheduler.

starvation freedom by ensuring that at least one task from each job is always running [IPC*09, app., pp. 275–276].

To ensure that the upper bound $F_j$ is maintained, each unscheduled aggregator vertex also generates a flow of $F_j - N_j$. Since this is negative (as $F_j \leq N_j$), this actually amounts to draining a flow of $|F_j - N_j|$. This makes sense, because:

- $N_j - F_j$ tasks cannot schedule, as they would exceed the upper bound $F_j$;
- $F_j - E_j$ tasks may schedule, but may also remain unscheduled by reaching the sink via the unscheduled aggregator; and
- $E_j$ tasks must schedule by reaching the sink through machine vertices.

By draining the flow for $N_j - F_j$ tasks at $U_j$, and with the sink’s demand set to $-\sum_j(F_j)$, all possible solutions to the optimisation satisfy this demand. Any solution that drains all flow generated in the network must route at least $N_j - F_j$ flow through $U_j$, since there is no other
way for this flow to be drained (as it has been deducted from the demand at the sink vertex).

In order to enforce the lower bound on the number of tasks scheduled ($E_j$), the unscheduled aggregator vertices must ensure that it is never possible for more than $N_j - E_j$ tasks to send or drain flow through them. The only other way for flow to reach the sink is through the resource topology, so this limitation forces $E_j$ tasks to be scheduled. The limit is enforced by setting the capacity of $U_j$’s outgoing arc to the sink to $F_j - E_j$. At most $F_j$ flow can remain after the local demand of $F_j - N_j$ is satisfied, but $E_j$ flow must be forced to drain via other ways, giving $F_j - E_j$ remaining flow to drain via the sink.

The upper ($F_j$) and lower ($E_j$) bounds on the number of runnable tasks can be varied to enforce fairness policies (cf. §2.3.4). I elaborate on this in Section 5.4.2.

While the capacities are used to establish possible scheduling states of tasks (as well as to enforce fairness constraints), the costs associated with arcs in the flow network are used to describe how preferable a possible scheduling assignment is. Some cost terms are general and always assigned in the same way, but others are configurable, allowing different scheduling policies to be expressed. I explain the general cost terms in the next section, and describe four specific cost models configuring others in Section 5.5.

### C.3 Quincy cost model details

The original Quincy paper describes a cost model based on co-optimisation of data locality, preemption cost, and task scheduling delay [IPC+09, §4.2]. These dimensions are mutually dependent: good data locality reduces runtime, but may require either waiting for a suitable machine to become available (increasing wait time) or preempting an already running task (wasting work that needs to be re-done).

Quincy assumes that the tasks’ input data reside in a distributed filesystem on the same cluster, i.e. remote data can be fetched from any machine (as in GFS [GGL03], TidyFS [FHI+11] and FDS [NEF+12]), but incorporates data locality into the cost model.

Table C.2 gives an overview of the costs Quincy assigns to different arcs, as well as other cost parameters not specific to arcs. The relations between these parameters and the cost expressions assigned to arcs are summarised in Table C.3. The costs are expanded in terms of the expected time required to transfer remote input data, a task’s cumulative wait time, and, if applicable, the time for which a task has already been running once it is scheduled.

In all cases that involve placement of a task in a location where it is not currently running, the cost is calculated by multiplying the cost of transferring remote input data across the top-of-rack switch ($\psi$) and the cluster aggregation switch ($\xi$) with the amount of remote data required. $\mathcal{R}(T_{i,j})$ indicates the number of bytes that must be copied from machines within the same rack; $\mathcal{X}(T_{i,j})$ is the number of bytes that must be pulled in from other racks for task $T_{j,i}$. 
Table C.2: Cost parameters in the Quincy cost model and their roles.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Edge</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$v_j^i$</td>
<td>$T_{j,i} \rightarrow U_j$</td>
<td>Cost of leaving $T_{j,i}$ unscheduled.</td>
</tr>
<tr>
<td>$\alpha_j^i$</td>
<td>$T_{j,i} \rightarrow X$</td>
<td>Cost of scheduling in the worst possible location.</td>
</tr>
<tr>
<td>$\rho_j^i,l$</td>
<td>$T_{j,i} \rightarrow R_l$</td>
<td>Cost of scheduling on worst machine in rack.</td>
</tr>
<tr>
<td>$\gamma_j^i,m$</td>
<td>$T_{j,i} \rightarrow M_m$</td>
<td>Cost of scheduling or continuing to run on machine $M_m$.</td>
</tr>
<tr>
<td>$\psi$</td>
<td>–</td>
<td>Cost of transferring 1 GB across ToR switch.</td>
</tr>
<tr>
<td>$\xi$</td>
<td>–</td>
<td>Cost of transferring 1 GB across aggregation switch.</td>
</tr>
<tr>
<td>$d^*$</td>
<td>–</td>
<td>Data transfer cost for given locality (data term).</td>
</tr>
<tr>
<td>$p^*$</td>
<td>–</td>
<td>Opportunity cost of preemption (preemption term).</td>
</tr>
</tbody>
</table>

Figure C.3: The example flow network in Figure 5.1 with costs according to the Quincy cost model added to the arcs. The capacities are assigned as in Figure C.2.

As the scheduler runs, it may choose to route flow for a running task through a different machine to the one that the task was originally assigned to. This choice carries not only the cost of the new assignment (as described above), but also an opportunity cost of preempting (i.e., terminating) the already-running task and potentially wasting work.\(^7\)

The opportunity cost of preempting a task is expressed by the preemption term, $p^*$. This is set to $\theta^j_i$, the number of seconds for which task $T_{j,i}$ has already been running anywhere, whether on the current machine or as part of another prior assignment. In other words, $p^*$ grows with time as the task runs. The preemption term is applied as a discount to the cost of continuing to run in the same location: effectively, continuing to run a task running becomes increasingly

---

\(^7\)The exact amount of work wasted depends on whether a preempted task runs from the beginning once it is assigned to a machine again. Many distributed applications use checkpointing or similar techniques to save partial results of a task, especially in batch computations [LGZ'14, §4; MMI'13, §3.4; GLG'12, §7.5].
参数 | 值
---|---
$v_j^i$ | $\omega v_j^i$
$\alpha_j^i$ | $\psi \rho_j^X(T_{j,i}) + \xi \lambda_j^X(T_{j,i})$
$\rho_{i,l}^j$ | $\psi \rho_{i,l}^X(T_{j,i}) + \xi \lambda_{i,l}^X(T_{j,i})$
$\gamma_{i,m}^j$ | $\begin{cases} d^* & \text{if not running} \\ d^* - p^* & \text{if running on } m \end{cases}$
$d^*$ | $\psi \rho_{m}^X(T_{j,i}) + \xi \lambda_m^X(T_{j,i})$
$p^*$ | $\theta_i^j$

Table C.3: 参数值用于Quincy成本模型。

attractive over time and thus carries a reduced cost. Any better assignment that might lead to a preemption or a migration to a different machine must offer an advantage of more than $-p^*$ over the current assignment’s cost.

The Quincy cost model is powerful and yields good assignments in a wide range of practical settings [IPC+09, §6]. However, it suffers from a number of limitations:

1. It does not consider interference between tasks due to sharing machine resources. While the $K$ parameter in the flow network capacities allows for co-location (see §5.2.1), all co-location opportunities are treated equally.

2. It assumes that machines are homogeneous and that a task’s runtime on a machine only depends on input data locality. As I showed in Section 2.2.3, this is rarely the case in WSCs.

In the following sections, I describe two cost models that I developed for Firmament to address these limitations.

### C.4 Details of flow scheduling limitations

In Section 5.4.4, I outlined features of scheduling policies that are difficult to express directly using Firmament’s flow network optimisation.

In the following, I explain the challenging policies in more detail using examples, and sketch how the techniques of multi-round scheduling and admission control allow Firmament to model them nevertheless.

#### C.4.0.1 Combinatorial constraints and global invariants

The flow network representation is versatile, but it is not a panacea: there are desirable scheduling properties that it cannot easily express.
Combinatorial constraints are constraints that have mutual dependencies (also sometimes referred to as “correlated constraints”). Firmament assumes that the decision to place a task on a resource (by routing flow through it) is independent of the other decisions (placements, pre-emptions, or migrations) made in the same scheduling round.

As a result, it is challenging to express mutually dependent scheduling constraints in Firmament (e.g. “if \( T_0 \) goes here, \( T_1 \) must go there”, where both \( T_0 \) and \( T_1 \) are both unscheduled). For example, this includes the following types of constraints:

**Co-scheduling constraints**: “tasks of this job must run on the same rack/machine” or “tasks of this job must never share a rack/machine”.

**Distance constraints**: “this pair of tasks must have no more than two switch hops between them”.

**Conditional preemption**: “if this task is placed, another task must be preempted as a result”.

**\( n \)-choose-\( k \) constraints**: “at least \( k \) out of \( n \) tasks of this job must be scheduled at the same time, or none must be”.

Many of these can be handled by reactive multi-round scheduling (§5.4.1) and the \( n \)-choose-\( k \) constraint can be modelled using Firmament’s support for relaxed gang scheduling (§5.4.3).

Few other schedulers support generalised combinatorial constraints. One exception is the work by Tumanov et al. (alsched [TCG+12] and tetriscched [TZK+13]), which models scheduling as a Mixed Integer Linear Problem (MILP).

**Global invariants** are difficult to express if they require dependent assignments, as these amount to combinatorial constraints. On common global invariant is a fairness policy.

Some fairness policies can be expressed with the aid of the unscheduled aggregator vertices enforcing bounds on the number of runnable tasks (§5.4.2), but others cannot. For example, a policy that guarantees equal shares of preferable co-location assignments across users can only be enforced reactively, and one that guarantees fair shares of cross-rack bandwidth can only be enforced by admission control.

There are other global invariants that amount to dependent decisions. For example, Google’s scheduler supports a per-job “different machine” invariant [RWH11, p. 9], which ensures that no two jobs in a task share a machine. “No more than two web search tasks may share a machine” or “no two tasks with large inputs may start in a rack at the same time” are difficult invariants for Firmament to enforce in a single scheduling round. However, Firmament can support this, and the other invariants mentioned, via reactive multi-round scheduling.

Likewise, a strict global priority order for preemption (i.e. “no higher-priority task ever gets preempted by a lower-priority task”) can only be expressed with carefully bounded dynamic cost adjustments. If, for example, costs are increased based on wait time (as done in Quincy), a long-waiting low-priority task may – in the absence of a bound – end up preempting a higher-priority task once its cost is sufficiently high.
C.4.0.2 Multiple scheduling dimensions

Multi-dimensional resource models are common in practical WSC environments [RTG+12; SKA+13]. In the flow network approach, each unit of flow atomically represents a task. Resources’ arcs to the sink have an integer flow capacity that regulates the number of tasks that may schedule on a leaf resource. As observed in the Quincy paper [IPC+09, §8], this approach does not take into account the multi-dimensionality of tasks’ resource requirements.

Ideally, “capacity”-type resource dimensions – such as the free memory on a machine, or shares of disk or network bandwidth – would be expressed directly via flow network capacities. However, this would require routing flow in multiple dimensions, which is impossible with unambiguous results when using the minimum-cost, maximum-flow optimisation. Another approach – multi-commodity minimum-cost, maximum-flow optimisation – supports multiple “commodities” flowing from sources to sinks at the same time, but still assumes one-dimensional arc capacities. Extending the problem to multiple capacity dimensions would involve tracking vectors of flow across each arc.

Such tracking of flow vectors is unlikely to become feasible within reasonable time: even simple multi-commodity flow problems without cost minimisation are NP-complete [EIS75, §4] and solving vectorised capacities would require solving the (also NP-complete) bin packing problem. However, it might be possible to simplify the problem with good heuristics in the specific domain of task scheduling.

There is a way to enable multi-dimensional resource models in Firmament: admission control. Tasks that do not fit sufficiently in all resource dimensions can either (i) be rejected before being added to the flow network, or (ii) have their scheduling opportunities restricted by removing the option of scheduling via the wildcard aggregator (X). The CoCo cost model (§5.5.3) uses the latter approach.

C.5 Firmament cost model API

Listing C.1 shows the current Firmament cost model API.
```cpp
class CostModelInterface {
    virtual Cost_t TaskToUnscheduledAggCost(TaskID_t task_id) = 0;
    virtual Cost_t UnscheduledAggToSinkCost(JobID_t job_id) = 0;
    virtual Cost_t TaskToResourceNodeCost(TaskID_t task_id, 
        ResourceID_t resource_id) = 0;
    virtual Cost_t ResourceNodeToResourceNodeCost(ResourceID_t source, 
        ResourceID_t destination) = 0;
    virtual Cost_t LeafResourceNodeToSinkCost(ResourceID_t resource_id) = 0;
    virtual Cost_t TaskContinuationCost(TaskID_t task_id) = 0;
    virtual Cost_t TaskPreemptionCost(TaskID_t task_id) = 0;
    virtual Cost_t TaskToEquivClassAggregator(TaskID_t task_id, 
        EquivClass_t tec) = 0;

    virtual pair<Cost_t, int64_t> EquivClassToResourceNode( 
        EquivClass_t ec, 
        ResourceID_t res_id) = 0;
    virtual Cost_t EquivClassToEquivClass(EquivClass_t tec1, 
        EquivClass_t tec2) = 0;

    /**
     * Methods to determine equivalence classes.
     */
    virtual vector<EquivClass_t>* GetTaskEquivClasses(TaskID_t task_id) = 0;
    virtual vector<EquivClass_t>* GetResourceEquivClasses( 
        ResourceID_t res_id) = 0;
    virtual vector<ResourceID_t>* GetOutgoingEquivClassPrefArcs( 
        EquivClass_t tec) = 0;
    virtual vector<TaskID_t>* GetIncomingEquivClassPrefArcs( 
        EquivClass_t tec) = 0;
    virtual vector<ResourceID_t>* GetTaskPreferenceArcs(TaskID_t task_id) = 0;
    virtual pair<vector<EquivClass_t>*, vector<EquivClass_t>*> GetEquivClassToEquivClassesArcs(EquivClass_t tec) = 0;

    /**
     * Machine and task management.
     */
    virtual void AddMachine(ResourceTopologyNodeDescriptor* rtnd_ptr) = 0;
    virtual void AddTask(TaskID_t task_id) = 0;
    virtual void RemoveMachine(ResourceID_t res_id) = 0;
    virtual void RemoveTask(TaskID_t task_id) = 0;

    virtual FlowGraphNode* GatherStats(FlowGraphNode* accumulator, 
        FlowGraphNode* other) = 0;
    virtual void PrepareStats(FlowGraphNode* accumulator) { }
    virtual FlowGraphNode* UpdateStats(FlowGraphNode* accumulator, 
        FlowGraphNode* other) = 0;

    /**
     * Debug information
     */
    virtual const string DebugInfo() const;
    virtual const string DebugInfoCSV() const;
}