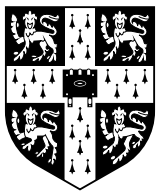
Operating system support for warehouse-scale computing

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 that is being concurrently submitted for a degree, diploma, or other qualification at the University of Cambridge, or any other University or 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

Modern applications are increasingly backed by large-scale data centres. Systems software in these data centre environments, however, faces substantial challenges: the lack of uniform resource abstractions makes sharing and resource management inefficient, infrastructure software lacks end-to-end access control mechanisms, and work placement ignores the effects of hardware heterogeneity and workload interference.

In this dissertation, I argue that uniform, clean-slate operating system (OS) abstractions designed to support distributed systems can make data centres more efficient and secure. I present a novel *distributed operating system for data centres*, focusing on two OS components: the abstractions for resource naming, management and protection, and the scheduling of work to compute resources.

First, I introduce a reference model for a decentralised, distributed data centre OS, based on pervasive distributed objects and inspired by concepts in classic 1980s distributed OSes. *Translucent abstractions* free users from having to understand implementation details, but enable introspection for performance optimisation. Fine-grained access control is supported by combining storable, communicable *identifier capabilities*, and context-dependent, ephemeral *handle capabilities*. Finally, multi-phase I/O requests implement optimistically concurrent access to objects while supporting diverse application-level consistency policies.

Second, I present the DIOS operating system, an implementation of my model as an extension to Linux. 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 support distributed applications, and evaluate the performance of DIOS in micro-benchmarks and a data-intensive MapReduce application. I find that it offers improved, fine-grained isolation of resources, while permitting flexible sharing.

Third, I present the Firmament cluster scheduler, which generalises prior work on scheduling via minimum-cost flow optimisation. Firmament can flexibly express many scheduling policies using pluggable cost models; it makes high-quality placement decisions based on fine-grained information about tasks and resources; and it scales the flow-based scheduling approach to very large clusters. In two case studies, I show that Firmament supports policies that reduce co-location interference between tasks and that it successfully exploits flexibility in the workload to improve the energy efficiency of a heterogeneous cluster. Moreover, my evaluation shows that Firmament scales the minimum-cost flow optimisation to clusters of tens of thousands of machines while still making sub-second placement 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 for his 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 and Andrew Moore, who assisted with equipment and feedback at key moments.

Ionel Gog and Matthew Grosvenor deserve credit and gratitude for our close collaboration and FN07 camaraderie over the years. Our shared spirit of intellectual curiosity, rigorous experimentation, and lighthearted humour embodied what makes systems research enjoyable to me.

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, I thank Julia Netter, who accompanied my PhD journey – from its very beginning to the final proofreading of this document – with wit, insight, and loving support.

Contents

List of Figures

List of Tables

Listings

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].

Since the late 1970s, computers have moved from large "mainframes" towards ever smaller, more affordable machines and devices. Today, however, we witness the return of mainframe-like, large-scale computer data centres: "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 resource demands, or which require distribution to ensure service availability and fault-tolerance.

The software infrastructure of a data centre is by necessity complex, shared, and highly multi-programmed. The sensitivity of the data processed demands strong isolation between applications and users, and controlled sharing of information between them. Component failures and complex multi-dimensional interactions between applications threaten availability of the system and make performance unpredictable. Nevertheless, system users and programmers demand to be shielded from this complexity: the most successful distributed systems are often those which hide the details of distributed communication and coordination.

Resource management, sharing, and isolation are traditionally the responsibility of an *operating system*. Operating systems have been an essential part of 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.

Today's widely-deployed operating systems are designed for smartphones, laptops, and individual servers, but do not have any native abstractions for distributed operation over clusters of machines in a data centre. As their scope is limited to the local machine, such OSes fail to fulfil two key OS responsibilities: first, they are unable to name, allocate, and manage resources

outside the local machine, and second, they cannot enforce isolation between applications and users across machines. Instead, they leave it to distributed systems software to provide this functionality.

Hence, a new breed of distributed infrastructure "middleware" has emerged to provide "OS-like" functionality for data centres: cluster managers, distributed file systems, and parallel programming frameworks help users build distributed applications by providing resource management services to them. However, such software currently fails to provide the same uniform, general, and powerful abstractions that traditional single-machine OSes have long offered.

In my research for this dissertation, I have developed a reference model for a *distributed data centre operating system* and prototype implementations of several of its components. In this OS, each machine is part of a larger whole: it is able to address, describe, and interact with remote data and resources. I focus on two parts of the OS to highlight the benefits of my approach:

1. *The abstractions for resource naming, access, and management.* Better abstractions make distributed systems more secure, more efficient, and easier to build.

2. *The scheduling of work to compute resources.* Globally optimising placement of work across the cluster while considering its machine-level impact allows for more deterministic performance, increases utilisation, and saves energy.

Based on my model, I constructed two prototypes: the DIOS extensions to Linux, and the Firmament cluster scheduler. I use them to investigate the following thesis:

> Better operating system support for distributed operation can improve the efficiency and security of distributed systems in "warehouse-scale" data centres.
>
> A clean-slate distributed data centre OS, based on uniform, translucent OS primitives and capability-based access control, implements efficient and secure resource management, while running prototypical distributed applications at performance comparable to, or better than, current OSes and data centre "middleware".
>
> A cluster scheduler that combines distributed cluster state and fine-grained local machine information additionally improves performance determinism and energy efficiency of the data centre, and is sufficiently flexible to extend to other policies.

In the next section, I explain the background of today's commodity operating systems and data centre "middleware", outline why a new distributed data centre OS can improve upon both, and point out related recent research efforts (§1.1). Following, I state my contributions described in this dissertation (§1.2) and outline its overall structure (§1.3). Finally, I list prior publications of the work described and related projects that have impacted it (§1.4).

1.1 Background

As computing devices become ever smaller, cheaper, and more ubiquitously 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. Request processing in these data centres involves multiple distributed systems that extend over thousands of individual server machines.

Individual machines in these data centres merely contribute resources to a large pool: they may join, leave, or fail at any time. While many different applications and users share the data centre's machines, it is often conceptually abstracted as a single machine of very large capacity to the programmer. This notion is somewhat akin to that of a time-shared mainframe, and referred to as a "warehouse-scale computer" (WSC) [BCH13, pp. 2–5].

Within a single machine, the functionality of resource virtualisation, sharing, naming, and management is provided by its operating system. Indeed, the local abstractions for this purpose have a long history: many of today's pre-eminent operating system paradigms and abstractions originated with 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, and inspired widely-used contemporary systems such as GNU/Linux, the BSD family, and Windows. Unlike mainframe operating systems, which were designed to deal with large, shared installations with multiple independent components that might fail, these OSes focus solely on managing a single machine with shared memory and I/O hardware.

One might expect the operating system of a "warehouse-scale computer" to be more similar to a mainframe OS than to the single-machine operating systems. However, the individual machines locally run standard server operating systems – typically variants of Linux, BSD, or Windows. Instead of extending these OSes, a new class of systems software for data centres has emerged: distributed infrastructure systems built to offer distributed variants of operating system services to distributed applications. This "middleware" forms the WSC OS and includes systems that, individually, provide distributed coordination, distributed file systems, distributed process and resource management, and distributed parallel programming frameworks. Indeed, industrial package management and deployment systems – e.g. Mesosphere DC/OS[1] – are advertised as a "data centre operating system", since they provide abstractions on top of which application programmers build their distributed applications.

However, the use of commodity OSes combined with several, individually single-purpose distributed "middleware" components has drawbacks:

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

[1] See `https://dcos.io/`; accessed 10/05/2016.

and between different distributed systems (e.g. HDFS ACLs [SKR+10] *vs.* key-value store identifiers [CDG+06] *vs.* cluster-level access control [VPK+15]).

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

2. **Lack of specialisation:** commodity operating systems are designed to support use cases from interactive desktops to highly-loaded servers. Many of their abstractions and techniques used are compromises for generality, rather than a good fit for the specific use case of a data centre.

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

3. **Poor access restrictions:** the data centre is a multi-tenant environment and users may accidentally or deliberately run malicious code. Operators currently use virtualisation techniques, such as containers, and virtual machines to restrict access and contain attacks, but these techniques are coarse-grained and make delegation difficult.

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

4. **Segregated scheduling:** machine-level scheduling decisions (in the local OS) 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).

Improving the distributed operating system and its abstractions can help address these drawbacks, for example by introducing more uniform resource management and access control. Consequently, it might prove insightful to consider what a clean-slate OS for a data centre should look like.

Indeed, others have made similar observations:

- Zaharia *et al.* note the need for "operating system" abstractions in the data centre, required to support a growing set of distributed applications [ZHK+11].
- Maas *et al.* envision a holistic language runtime that transcends machine boundaries and takes the place of the operating system [MAH+14].

However, the idea of a distributed OS with uniform abstractions is not itself new. In the 1980s, researchers experimented with distributed OS-level primitives, but these failed to see adoption outside academic research, and consequently the current, "middleware-based" data centre OSes

evolved. Why should we nevertheless look into better distributed operating system abstractions, rather than sticking with what we have?

A key reason why the distributed OS concept is timely again is that distributed operation is a *necessity* in warehouse-scale data centres, rather than – as in the 1980s – an option useful only to some workloads. Additionally, there are several advantages to a coherent distributed OS over an ad-hoc combination of heterogeneous middleware systems:

1. The operating system's abstractions set the rules by which ultimately all applications 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 application.
3. The OS abstractions are uniform across applications, since they are designed to be application-agnostic primitives, and they are available to *any* application.
4. The privileged OS has an elevated, omniscient view of resource allocation, and can hence make better decisions than individual programs can on their own.

A new, clean-slate data centre OS might complement the efficiency gains that custom-built physical data centre infrastructure, custom machine chassis, rack, and cooling equipment already grant to data centre operators [BCH13, pp. 47–65; WSK+09]. Indeed, Google already customise Linux extensively [WC09]; the effort involved in building a new OS is likely acceptable to such large organisations.

Indeed, research efforts increasingly 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 performs two-level scheduling (like Akaros). 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 proposes a new distributed OS model, specialised to the domain of warehouse-scale data centres. In doing so, it draws on many of the above, as well as on historic distributed operating systems (§2.2.1). DIOS, my prototype implementation, 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).

1.2 Contributions

This dissertation describes three principal contributions:

1. My first contribution is a **model for a decentralised, distributed operating system** for data centres, based on uniform, clean-slate OS abstractions that improve the efficiency and security of distributed applications. This decentralised data centre OS model is built around pervasive *distributed objects*, and relies on storable and communicable *identifiers* to name and discover resources. Once discovered, resource management relies on *translucent handles* that form delegatable capabilities, which can be introspected upon to improve application performance and fault tolerance. To allow flexible concurrent access to objects without implicit synchronisation, the model supports *transaction-like I/O requests*.

2. My second contribution is the **DIOS prototype**, which implements the decentralised data centre OS model by extending Linux. DIOS is based on typed objects, which are named,

accessed, and managed via distributed capabilities. DIOS implements the decentralised data centre OS model's identifiers as globally unique *names* resolved via a distributed kernel name service, while *references* implement translucent handles as segregated capabilities. Moreover, DIOS has a new system call API for interaction with local and remote objects based on scalable design principles, and supports I/O requests with flexible concurrent access semantics. Finally, I describe how DIOS integrates with Linux to achieve backwards-compatibility with existing applications.

3. My third contribution is the **Firmament cluster scheduler**. I generalise the approach taken by Quincy [IPC+09], which models the scheduling problem as a minimum-cost optimisation over a flow network. I show that this generalisation allows the flow-optimisation approach to flexibly express desirable scheduling policies not supported by Quincy. I also demonstrate that – contrary to common prior belief – the flow-optimisation approach is scalable to warehouse-scale clusters. Firmament is implemented as a cluster manager and tracks detailed task profiling and machine architecture information. In three case studies, I implemented Firmament scheduling policies that consider machine heterogeneity, avoid task co-location interference, and increase data centre energy efficiency.

All models, algorithms, and implementations described are results of my own work, and I built the DIOS and Firmament implementations from scratch. However, colleagues and students in the Computer Laboratory have at times assisted me in extending and evaluating specific prototype components.

In particular, Matthew Grosvenor and I sketched an initial version of the DIOS system call API together, and Andrew Scull contributed the ELF branding for DIOS binaries (§4.10.3) and ported the Rust runtime to DIOS (§9.1.1) during his Part II project in the Computer Science Tripos [Scu15]. Ionel Gog implemented the `flowlessly` minimum-cost, maximum-flow solver for Firmament (§6.4.3), and Gustaf Helgesson implemented and evaluated Firmament's Green cost model (§7.4) under my supervision during his MPhil in Advanced Computer Science. Finally, Adam Gleave investigated incremental minimum-cost, maximum-flow solvers and the relaxation algorithm (§6.4.3) in his Part II project under my supervision [Gle15].

1.3 Dissertation outline

This dissertation is structured as follows:

Chapter 2 surveys background and related work for three areas covered by my work: "warehouse-scale" data centres, operating systems, and cluster scheduling. I explain the software infrastructure of modern data centres, and show that hardware heterogeneity and task co-location interference hamper performance predictability. I then consider operating systems, survey classic distributed OSes and the current data centre software, and highlight

key deficiencies with the latter. Finally, I give an overview of the state of the art in cluster scheduling.

Chapter 3 introduces a reference model for resource naming and management in a decentralised, distributed data centre operating system. I enumerate the requirements this model must satisfy, explain its foundation in distributed objects, introduce the notion of *translucent* abstractions, and show how storable and communicable *identifier capabilities* and contextual *handle capabilities* are sufficient to enact fine-grained access control. Finally, I explain how the model stores objects persistently, and how *I/O requests* enable concurrent I/O without implicit synchronisation.

Chapter 4 introduces DIOS, a prototype implementation of my model as an extension to Linux. I describe the key primitives and interfaces of DIOS: *names*, *references*, *groups*, and *tasks*. I outline the DIOS system call API, explain how machines coordinate with each other, discuss the scalability of the DIOS abstractions, and finally explain how the seamless integration with Linux enables incremental migration to DIOS.

Chapter 5 evaluates DIOS using micro-benchmarks of OS-level primitives and an example application. I show that DIOS runs distributed applications at comparable performance to current systems and compare and contrast the security properties of the DIOS abstractions with widely-used isolation techniques.

Chapter 6 describes my Firmament cluster scheduler. I show how Firmament generalises the Quincy scheduler [IPC+09] and how its pluggable *cost models* flexibly express scheduling policies. 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 large data centres, and finally describe how Firmament is implemented as a cluster manager.

Chapter 7 describes three cost models for Firmament that I implemented as case studies: the first avoids co-location interference by monitoring micro-architectural performance counters, the second also respects a multi-dimensional resource model, and the third improves energy efficiency in heterogeneous clusters.

Chapter 8 evaluates Firmament, investigating the effectiveness of its scheduling decisions using real-world test-bed deployments, Firmament's ability to flexibly express different scheduling policies, and its scalability to large clusters and a simulated Google workload.

Chapter 9 points out directions for future work and concludes my dissertation. In particular, I focus on the accessibility of DIOS abstractions for the programmer, on deeper local OS kernel changes motivated by DIOS, and on techniques to further improve security. I also discuss how Firmament might be extended to cover more heterogeneous systems.

1.4 Related publications

Parts of the work described in this dissertation have been covered in peer-reviewed publications:

[GSG+16] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. "Firmament: fast, centralized cluster scheduling at scale". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. To appear. Savannah, Georgia, USA, Nov. 2016.

[SGH13] Malte Schwarzkopf, Matthew P. Grosvenor, and Steven Hand. "New Wine in Old Skins: The Case for Distributed Operating Systems in the Data Center". In: *Proceedings of the 4th Asia-Pacific Systems Workshop (APSYS)*. Singapore, July 2013, 9:1–9:7.

[SH12] Malte Schwarzkopf and Steven Hand. "A down-to-earth look at the cloud host OS". in: *Proceedings of the 1st Workshop on Hot Topics in Cloud Data Processing (HotCDP)*. Bern, Switzerland, Apr. 2012, 3:1–3:5.

[SMH11] Malte Schwarzkopf, Derek G. Murray, and Steven Hand. "Condensing the cloud: running CIEL on many-core". In: *Proceedings of the 1st Workshop on Systems for Future Multicore Architectures (SFMA)*. Salzburg, Austria, Apr. 2011.

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:

[GSG+15] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. "Queues don't matter when you can JUMP them!" In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, California, USA, May 2015.

[GSC+15] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. "Musketeer: all for one, one for all in data processing systems". In: *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*. Bordeaux, France, Apr. 2015.

[SKA+13] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: flexible, scalable schedulers for large compute clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 351–364.

[SMH12] Malte Schwarzkopf, Derek G. Murray, and Steven Hand. "The Seven Deadly Sins of Cloud Computing Research". In: *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. Boston, Massachusetts, USA, June 2012, pp. 1–6.

[MSS+11] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. "CIEL: a universal execution engine for distributed data-flow computing". In: *Proceedings of the 8th USENIX Symposium on Networked System Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Mar. 2011, pp. 113–126.

Chapter 2

Background

Many modern applications – directly or indirectly – depend on distributed systems running in large-scale compute clusters situated in data centres. These data centres conceptually form "warehouse-scale computers", since the details of their operation are abstracted away from both the end-user and the application programmer [BCH13].

Like most modern computers, warehouse-scale data centre "machines" have an extensive systems software stack, though one that uniquely includes *distributed* systems software. Roughly, it contains three types of software:

Local operating system kernels interface between local machine hardware and higher-level, hardware-independent software. Operating system kernels enforce isolation, arbitrate machine resources, and locally perform privileged operations on behalf of other software.

Distributed infrastructure systems are user-space applications that run on many or on all machines in the WSC, and which collectively form the "operating system" of the data centre. Many of their services are distributed versions of classic OS functionality, such as data storage, scheduling, or coordination of computations.

User applications implement the functionality exposed to end-users of the data centre, 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 how this distributed software stack is realised in current data centres, and motivate the opportunity for a more uniform, secure and efficient design of two key components, resource management and the cluster scheduler.

In Section 2.1, I describe the make-up of today's warehouse-scale data centres. I first describe the roles of typical distributed infrastructure systems and the applications executed atop them (the "workload"). I then discuss several challenges posed by the data centre environment: unprecedented scale, a need for high utilisation in the face of hardware and workload heterogeneity, and consequent performance variance due to co-location interference between applications.

Following, Section 2.2 considers the role of the *operating system* in a WSC-style data centre. I compare the software that currently functions as the operating system of a WSC to classic distributed operating systems of the 1980s, and explain why revisiting some of their key ideas can benefit modern warehouse-scale data centres.

Finally, I focus on the *cluster scheduler*, a particularly crucial distributed infrastructure component in current data centre "operating systems". Section 2.3 discusses the design goals for a cluster scheduler and surveys the extensive work of recent years in this area, highlighting the challenges that my work addresses.

2.1 Warehouse-scale computers

To increase cluster utilisation, multiple applications and users typically share the cluster that constitutes a "warehouse-scale computer" [HKZ+11; VPK+15]. The WSC cluster runs many independent *tasks* – instantiated as processes, containers, or virtual machines – that belong to different applications. Executing the resulting "workload mix" efficiently and isolating applications from each other is key to the efficient and secure operation of a warehouse-scale data centre.

In this section, I describe the typical workload in a warehouse-scale data centre (§2.1.1), outline how heterogeneity in its constituent hardware matters (§2.1.2), and how high utilisation comes at the cost of interference that degrades workload performance (§2.1.3).

2.1.1 Workloads

WSCs exist to support workloads that require 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 end-user applications (e.g. mobile or web front-ends).

In general, the "workload" of such a data centre falls into two categories: *infrastructure systems* (§2.1.1.1) that provide essential services and *user applications* that process data or expose them to remote end-users (§2.1.1.2).

In addition, workloads can also often be divided into *batch* and *service* work (as, e.g., at Google [SKA+13]). This division is orthogonal to the split into infrastructure systems and applications, although most infrastructure systems are service workloads.

Service workloads run continuously, offering functionality either directly to end-users or to end-user-facing applications. They only terminate due to failure, or human or cluster scheduler intervention. A distributed key-value store is an example of a service workload.

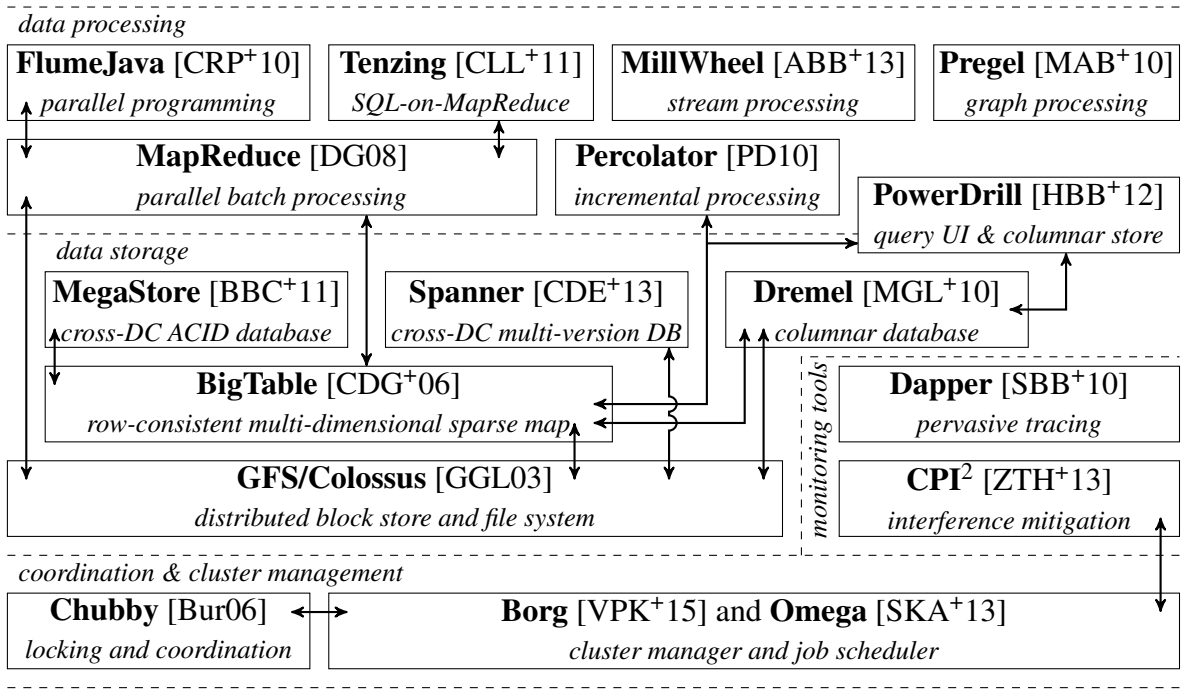


Figure 2.1: The Google infrastructure stack. I omit the F1 database [SOE+12], the back-end of which was superseded by Spanner, and unknown or unpublished front-end serving systems. Arrows indicate data exchange and dependencies between systems; "layering" does *not* imply a dependency or relation.

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].

Classification into the batch and service categories does not *per se* imply a priority order. However, service workloads are more likely to have high priorities, since their operation is essential to serving end-user requests and to keeping other applications operational.

It is also worth noting that most service jobs (and most infrastructure systems) are request-oriented, online transaction processing (OLTP) type workloads (even though they need not explicitly use transactions). Batch workloads, by contrast, are often online analytical processing (OLAP) workloads, have less rigid request structure, and tolerate higher response latencies.

2.1.1.1 Distributed infrastructure systems

Infrastructure systems are key to the operation of data centre 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

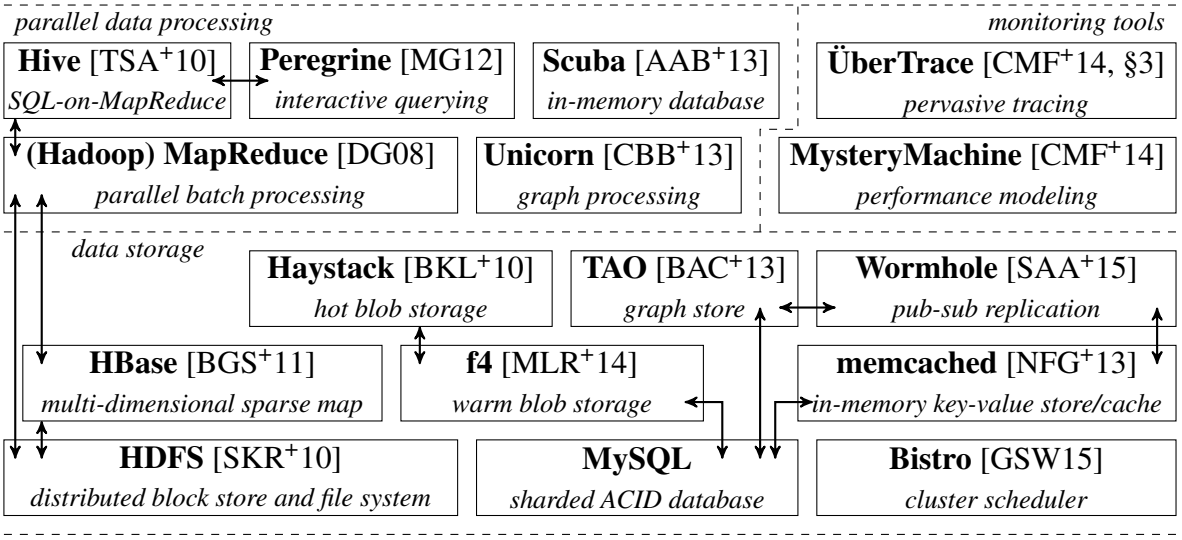


Figure 2.2: The Facebook infrastructure stack. No single cluster manager for a shared substrate analogous to Google's Borg and Omega is known; it is unclear if schedulers like Bistro deal with all workloads. Arrows indicate data exchange and dependencies between systems; "layering" does *not* imply a dependency or relation.

been created. Figures 2.1 and 2.2 illustrate this using the known components of the distributed 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. Many data centre infrastructure services are co-dependent and require an authoritative source of configuration information 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] for this purpose is based on the Paxos consensus algorithm [CGR07]. Yahoo!'s Zookeeper [HKJ+10], which is based on Zab [RJ08], and `etcd`, which is based on Raft [OO14], are similar open-source coordination services. All of these use distributed consensus algorithms that trade raw performance for reliability in the face of failures.

A *cluster manager*, by contrast, manages service and application 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. Task scheduling decisions are made as part of the cluster manager, although not all deployments use a *single*, unified cluster manager. Instead, some data centres are partitioned into single-purpose sub-clusters, each with an independent cluster scheduler. I will review these and other scheduler architectures in Section 2.3.1.

Data storage. Warehouse-scale data centres store huge amounts of data, but use different infrastructure systems for this purpose, depending on the data access frequency and structure.

Block storage either comes in the form of unstructured stores, or as hierarchical file systems akin to networked 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, data centres run sharded, replicated key-value stores that make 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. Some analytics 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 graph.

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, S4 at Yahoo! [NRN+10], and Storm [TTS+14] and its successor, Heron [KBF+15], at Twitter. 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 pervasively used communication libraries, 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 Facebook's Mystery Machine [CMF+14]), or to detect performance anomalies such as negative interference between co-located tasks (e.g. in Google's CPI^2 [ZTH+13]).

2.1.1.2 Applications and user jobs

Applications form the "business logic" of the data centre: they serve end user requests, analyse data to derive insights, or support other productivity tasks.

For example, Facebook's web server instances respond to end 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 local OS kernel, 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 latency bounds as part of a service level objective (SLO).

3. Application developers use high-level languages [MAH+14], and rely on higher-level interfaces than used for constructing infrastructure systems; as a result, application code is ignorant of the details of machines, coordination, and parallelisation.

Since the applications are merely consumers of APIs provided by the infrastructure systems, changes to the underlying operating system – either in the local OS kernel, or in the distributed infrastructure components – can be entirely invisible to the application programmers. Section 2.2.5 and later chapters will return to this observation.

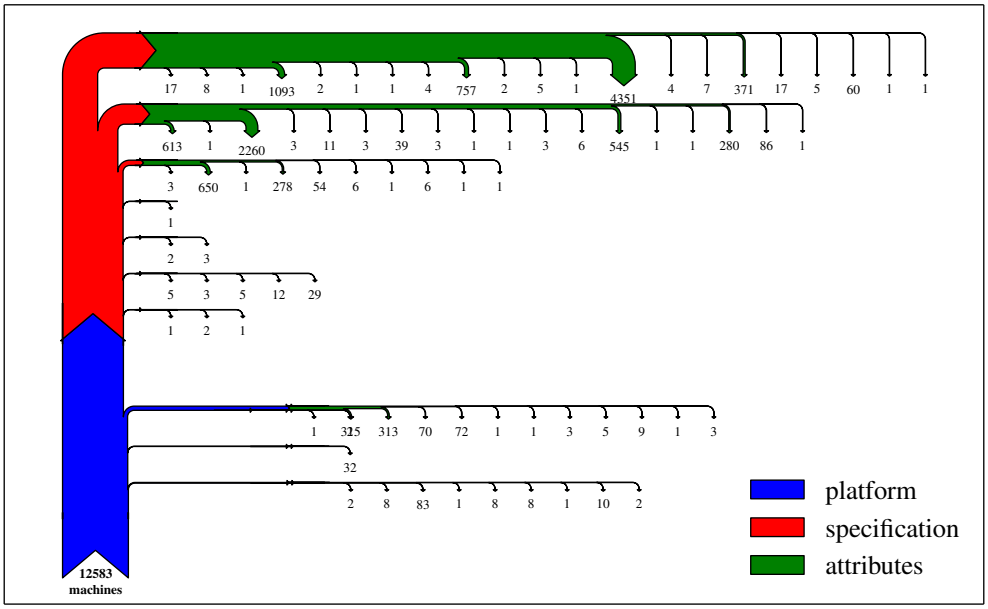


Figure 2.3: Sankey diagram of machine types and configurations in the public mid-2011 Google cluster trace [RWH11]. The cluster is heterogeneous: the 12,583 machines becoming increasingly fragmented when differentiated by platform, specification, and attributes.[1]

2.1.2 Hardware heterogeneity

Warehouse-scale data centres are typically composed of machines purchased in bulk, since this allows for economies of scale in purchasing. However, in practice, the machine mix is heterogeneous due to rolling hardware upgrades and deliberate diversification.

For example, the cluster in the public 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. 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 data centres, 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].

[1] The 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*, the latter being the number of cores and the total RAM [RWH11, p. 5].

[2] `http://www.coker.com.au/bonnie++/`; accessed on 05/01/2015.

[3] `https://iperf.fr/`; accessed on 05/01/2015.

[4] `init/calibrate.c` in the Linux kernel source; see `http://lxr.free-electrons.com/source/init/calibrate.c?v=3.14`; accessed on 05/01/2015.

Benchmark	Description	Reference
`hmmer`	Hidden Markov model gene database search (integer)	[Hen06, p. 5]
`gromacs`	Molecular dynamics simulation (floating point)	[Hen06, p. 11]
STREAM	DRAM memory throughput benchmark.	[McC95]
NUMA-STREAM	Multi-threaded version of STREAM for machines with multiple memory controllers.	[Ber11]
`bonnie-rd`	Disk read throughput measurement from the `bonnie++` benchmarking tool.	See footn.[2]
`bonnie-wr`	Disk write throughput measurement from the `bonnie++` benchmarking tool.	(see above)
iperf-cpu	CPU load while running `iperf` in TCP mode, saturating a 1 GBit/s link.	See footn.[3]
BogoMips	BogoMips number reported by Linux kernel.	See footn.[4]

Table 2.1: Machine heterogeneity micro-benchmark workloads.

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

Table 2.2: Machine configurations used in the experiments in Figure 2.4. All machines run Ubuntu 14.04 with Linux 3.13.

Impact of heterogeneous hardware. To illustrate this impact, 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 2.1) on a set of otherwise idle machines (Table 2.2). All machines are post-2009 designs representative of machines found in contemporary data centres assuming a five-year depreciation cycle.

Figure 2.4 shows the results, normalised to the oldest machine type (**A**). For the single-threaded, 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 also roughly correlated with the BogoMips measure reported by the Linux kernel.

The 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, al-

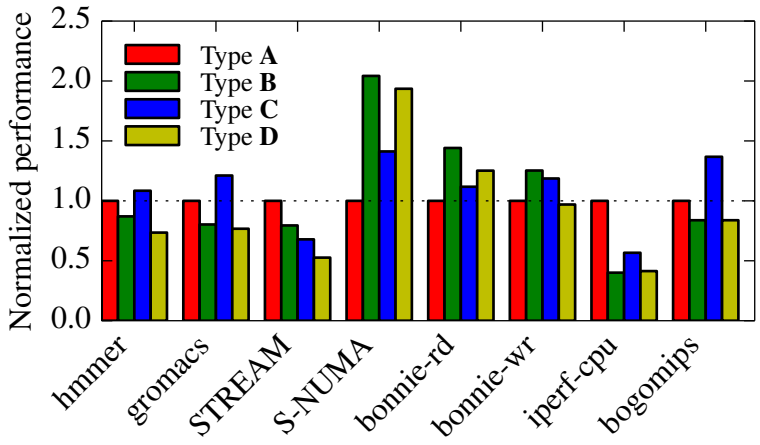


Figure 2.4: Normalised performance of the micro-benchmarks on the heterogeneous machine types listed in Table 2.2. Higher is better, and substantial variance of ±50–100% exists across machine types.

though 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) hard drives.

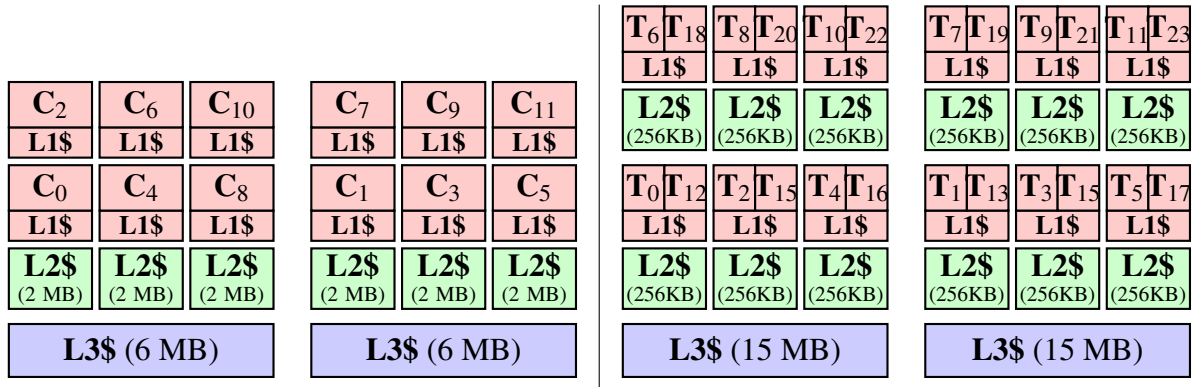
For network I/O with `iperf`, all machines saturate the link. However, type **A** machines see the lowest CPU load while doing so, which may be due to hardware offloading features present in the NICs of the type **A** machines, which are not available on other machines' NICs.

2.1.3 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 over-subscribed by the machine architecture – for example, NIC network bandwidth does not typically support all CPUs doing network I/O. Such oversubscription is a result of physical constraints, hardware cost, and typical server workloads. In the following, I illustrate the effects of contention on various hardware resources 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.1.1, I show that on both type **B** and type **C** machines, the runtime of SPEC CPU2006 benchmarks suffers degradations of up to 2.3× even when only two tasks are co-located on a machine.



(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. $\mathbf{C}_i$ are physical CPU cores, while $\mathbf{T}_i$ denote hyper-threads; first-level caches (**L1$**) are shown in red, second-level caches (**L2$**) in green, and third-level caches (**L3$**) in blue.

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

Table 2.3: Data centre applications used in pairwise interference experiments (Figure 2.6).

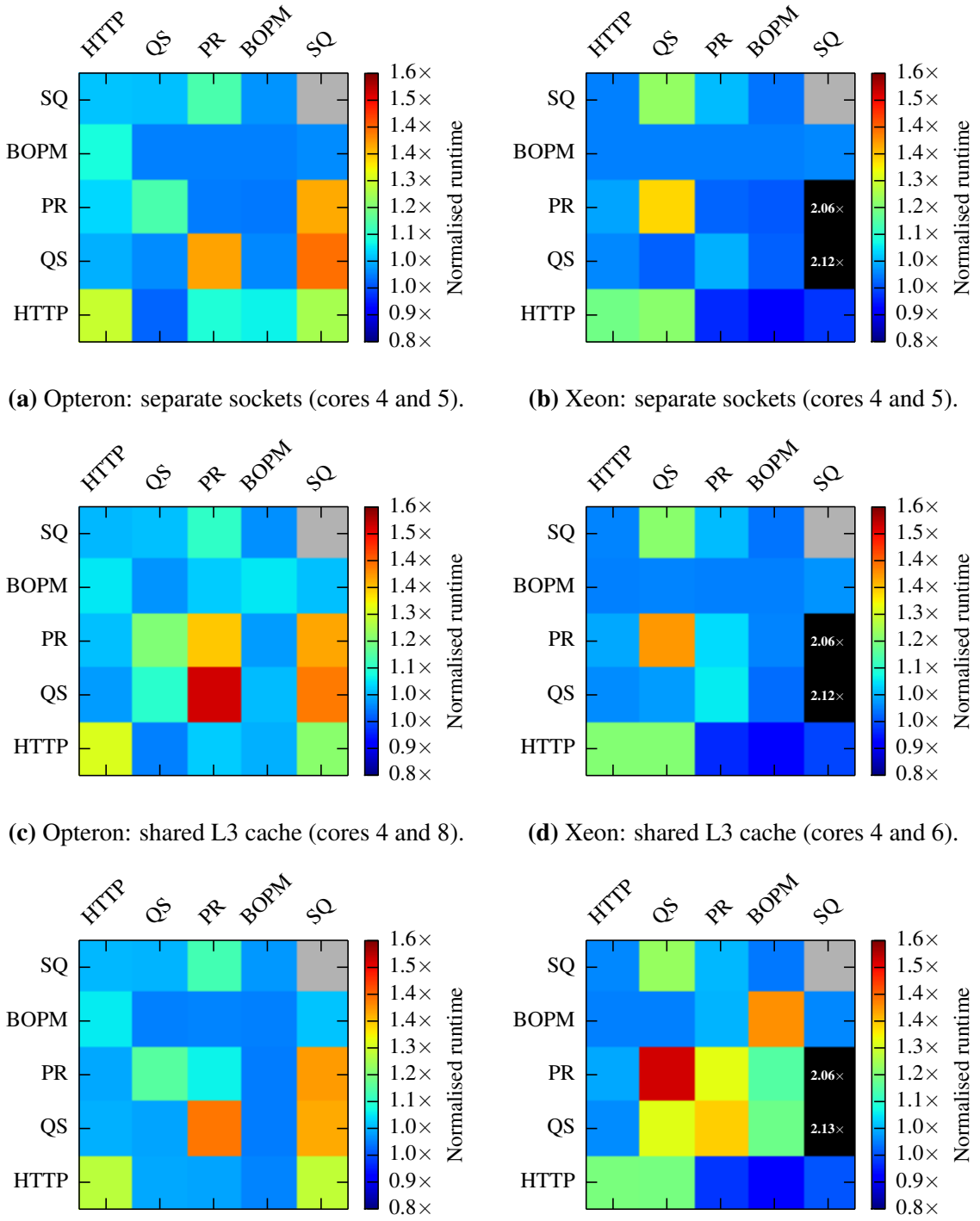
Benchmarks like SPEC CPU2006, however, use highly-optimised compute kernels that typically have good cache affinity, and sharing consequently has an especially severe effect. As Ferdman *et al.* found, most data centre applications are not tuned for cache affinity [FAK+12].

I thus repeat the same co-location experiment with a set of data centre applications. I run the applications shown in Table 2.3 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). To isolate contention and make the setup comparable to the SPEC CPU2006 experiments, I run each application on a single core only.[5]

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× 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.1.2 illustrate that many of the degradations are co-incident with increased cache miss counts. In some cases, however, interference is present even when

[5]The Spark query application is an exception: it has extra runtime threads (e.g. the JVM's garbage collection threads) and I allow it to use multiple cores.



(a) Opteron: separate sockets (cores 4 and 5).

(b) Xeon: separate sockets (cores 4 and 5).

(c) Opteron: shared L3 cache (cores 4 and 8).

(d) Xeon: shared L3 cache (cores 4 and 6).

(e) Opteron: sharing L2 and L3 (cores 4 and 6).

(f) Xeon: adjacent hyperthreads (cores 4 and 16).

Figure 2.6: Co-location interference between different data centre 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; grey ones indicate that the benchmark failed. See Appendix A.1.2 for corresponding heatmaps of cycle and cache miss counts.

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. non-scalable locks or kernel data structures) are contended.

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× degradation, compared to 1.4× 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.

***n*-way interference.** I have so far considered pairs of workloads on otherwise idle machines. In practice, many-core machines in a data centre 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$^{\text{th}}$ percentile [ZTH+13, Fig. 1(a)].

I hence investigated how *n*-way co-location (for *n* CPU cores) affects data centre application workloads. Figure 2.7 shows the normalised runtime of seven different batch processing workloads on a 28-machine cluster.[6] Most of the workloads are implemented using Naiad [MMI+13] (see Table 2.4), 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×, and the highly iterative and synchronisation-bound strongly connected components (SCC) and PageRank workloads degrade by up to 3×.

This experiment showed that interference on machine and cluster resources can have a substantial effect on the end-to-end job runtime of realistic 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× when their user-space runtimes make poor placement decisions on busy machines [HMM14]. Similarly, Mars *et al.* found a 35% degradation in application-level performance for Google workloads in different processes when sharing a machine [MTH+11].

[6] I later use the same cluster in evaluation experiments; see §5.1 and §8.1 for configuration details.
[7] `http://harthur.github.io/kittydar/`; accessed 07/01/2015.

Workload	Application	Share of cluster
Image analysis (cat detection)	kittydar[7]	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 2.4: Batch *(top)* and Service *(bottom)* workloads used in the scale-up interference experiment shown in Figure 2.7.

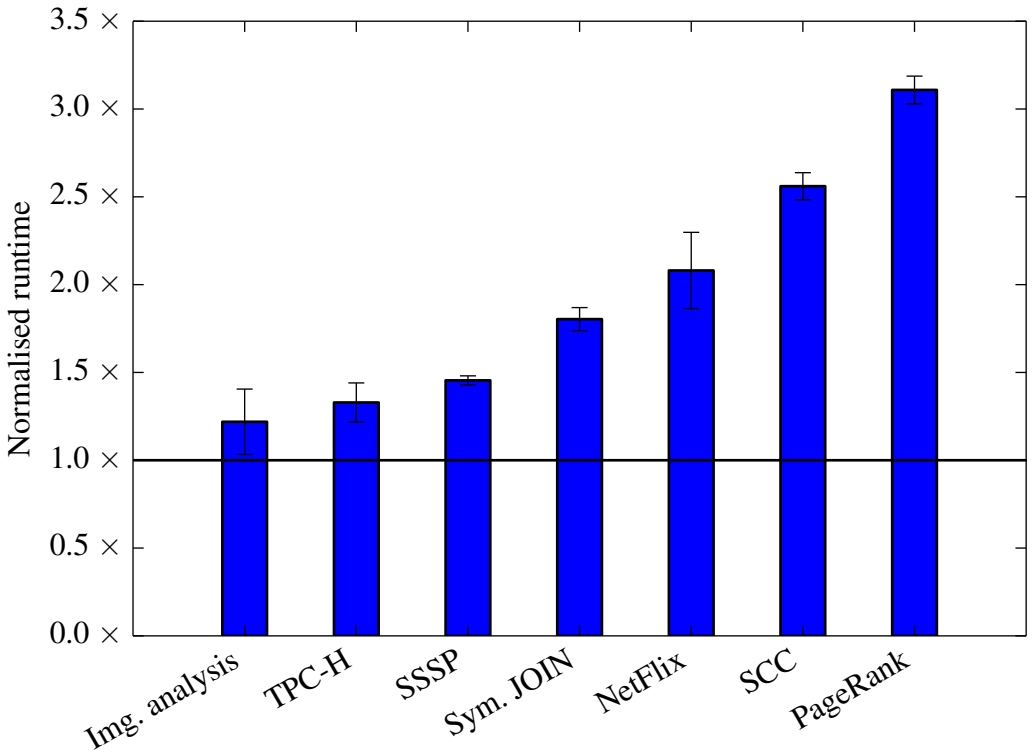


Figure 2.7: Normalised runtime of seven workloads on a 28-machine cluster, scheduled using a random first fit placement policy. All results are normalised to the isolated job runtime on an otherwise idle cluster.

Zhang *et al.* analysed the impact of co-location interference on production services at Google and saw up to 10× worst-case degradation in application performance [ZTH+13]. Leverich and Kozyrakis match these findings with an in-depth analysis of the impact of background batch workloads on a latency-sensitive key-value store [LK14]. Finally, Delimitrou and Kozyrakis studied the interference of a range of data centre workloads on dedicated machines and EC2 VMs, finding interference of up to 2× [DK13].

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

2.1.4 Summary

Warehouse-scale data centres are a new environment in which distributed systems software runs over thousands of interconnected machines. Their workloads are heterogeneous: they vary in nature, resource requirements, and in their impact on machine and cluster resources.

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

To adequately support common data centre workloads, we must:

1. develop mechanisms to deal with hardware heterogeneity in the cluster and use it to match workloads to machines well-suited to executing them (§2.1.2); and

2. avoid co-location interference between data centre workloads, and offer predictable performance by separating those workloads that interfere (§2.1.3).

In the next section, I look at how data centre workloads relate to the operating system, with particular focus on the abstractions used within distributed infrastructure systems. Following, Section 2.3 will survey cluster scheduling, which can help address the above challenges.

2.2 Operating systems

The examples in Section 2.1.1 illustrated the workload in a typical large-scale data centre. While their specifics differ, all distributed applications executed rely on underlying systems software, both local and distributed. In this section, I consider the nature and role of the operating system (OS) in a data centre. Operating systems provide hardware abstraction, multiprogramming, sharing of resources, and safe isolation between programs and users. This notion of an operating system evolved over the course of six decades, and has been primarily driven by the goal of *resource sharing* for efficiency.

Origins. While early machines had only equivalent of a primitive bootloader – e.g. "a certain invariable initial order" in Turing's ACE [Tur46, p. 37], and hard-wired "initial orders" on EDSAC [Wil85, pp. 143–4] – it soon became evident that having systems software to run multiple users' programs was essential to efficient machine use [Ros69, p. 39].

Consequently, *monitor* (or *supervisor*) programs were conceived to automate the process of running multiple jobs in sequence ("batch processing").[8] Such batch monitors enabled *multiprogramming*, but did not support interactive system use. *Time-sharing* of a machine enables such use by having several programs share a processor and execute seemingly concurrently, but requires a mechanism to interrupt the current computation and jump into supervisor code [KPH61]. Early time-shared systems like CTSS [CMD62] and the Titan supervisor [HLN68] operated as a combined batch supervisor and interactive OS: remote terminals interactively accessed the computer, but the processor ran traditional batch jobs (e.g. the "background facility" in CTSS [Cre81, p. 484]) when idle. This is somewhat similar to the resource sharing between batch and service jobs in data centres (cf. §2.1.1), which aims to reclaim spare service jobs resources for opportunistic use by batch jobs [VPK+15, §5.2; SKA+13, §2.1; BEL+14, §3.5].

Time sharing gave rise to many of the protection and isolation mechanisms used in today's operating systems: segments [BCD72] and virtual memory [DD68], rings of protection [CV65; SS72], and a single file system namespace with ACLs [RT74]. However, all of these abstractions focused on safely sharing *a single machine* between multiple programs and users. In the following, I explain how distribution over multiple machines came about.

2.2.1 Classic distributed operating systems

Early computing was confined to a single general-purpose machine with remote "dumb" terminals. However, research into distributed operation across machines started as early as the mid-1960s, when Fuchel and Heller shared an extended core store (ECS) between two CDC 6600 machines. Their "ECS based operating system" enabled process migration between the connected machines via swapping to the ECS, and suggested buffered I/O and tolerance of faults [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. When it was first feasible to own a substantial number of connected computers, the resource pooling opportunities triggered the development of *distributed operating systems* (see Table 2.5).

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

[8] Anecdotes 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]. MIT, 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 other supervisor systems, the director tape did not have *permanently resident* supervisory code in memory, however, but followed instructions on a paper tape in a spare mechanical reader [Hel54].

Distributed OS	Target environment	Architecture	Use case
HYDRA [WCC+74]	C.mmp machine	shared-memory with objects	parallel processing
Medusa [OSS80]	Cm* machine	message-passing	parallel processing
RIG/Aleph [LGF+82]	heterogeneous (Xerox Alto workstations)	message-passing	intelligent campus computing gateway
Accent [RR81]	homogeneous (PERQ workstations)	message-passing with objects	distrib. sensor network, personal computing
LOCUS [WPE+83]	homogeneous (VAX/705 machines)	message-passing	fault-tolerant time-sharing
VAXClusters [KLS86]	homogeneous (VAX machines)	message-passing	networked storage and compute
Mach [ABB+86]	heterogeneous (various workstations)	message-passing with objects	parallel and desktop computing
V [Che88]	homogeneous (MC68k workstations)	message-based RPC	desktop computing
Sprite [OCD+88]	heterogeneous (Sun workstations)	message-based RPC	desktop computing
Amoeba [MRT+90]	heterogenerous (SPARC, x86, Sun-3)	object-based RPC	desktop and parallel computing
Chorus/QNX [RAA+91]	heterogeneous (x86, 68k, SPARC)	async. messages, object-based RPC	low-level platform for networked applications
Plan 9 [PPD+95]	heterogeneous (workstations)	namespaces	desktop computing

Table 2.5: Classic distributed operating systems and their properties.

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 OS principles, such as using distributed objects as a primary abstraction, tolerance of machine and interconnect faults, and capability-based protection.

Medusa, another early distributed OS from CMU, targeted the Cm* architecture, which connected 50 "compute modules" in clusters of ten. As a result of the high communication cost between modules, Medusa emphasised locality and restricted sharing [OSS80, §1]. Medusa is one of the earliest examples of a distributed OS centred around explicit message-passing, and 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 and coarse-grained protection: to access an object, a *descriptor* capability, which 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, i.e. users had to understand the location of a message's

recipient. 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. Accent programmers, used a higher-level procedure-call interface based on stubs generated via the Matchmaker IDL [JR86]. Similar IDL concepts exist in many later distributed and micro-kernel operating systems, and indeed in today's data centre RPC libraries such as Protocol Buffers.[9]

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 influential as a micro-kernel architecture, but as a distributed OS, it 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 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 located files via a distributed naming catalog, and access to them was fully transparent. 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 and V targeted a model of per-user workstations, using process migration to harness idle resources. Sprite 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.

[9] `https://developers.google.com/protocol-buffers/`; accessed 21/02/2016.

Plan 9. Plan 9 was developed as a conceptual successor to Unix in the early 1990s, with a more pervasive file abstraction and kernel-level support for transparent distributed operation [PPD+95]. Plan 9 targeted networked end-user workstations, and advocates a file-centric approach that allows 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 [Men07; Mer14] 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? Distributed operating systems were developed over 30 years ago, and yet few of their features are present in modern OSes. I believe that there are three key reasons why distributed operating systems failed to see wide adoption:

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 feasible technology, without any pressing use case.

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 modern data centres:

1. *Workloads already require distribution.* Data centre workloads fundamentally require distribution for scale or fault tolerance (§2.1.1), and distribution over many 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.

Local OS component	Data centre OS equivalent	Example
Kernel/TCB	Cluster manager	Borg [VPK+15], Mesos [HKZ+11]
File system	Distributed file system	GFS [GGL03], HDFS [SKR+10]
Buffer cache	Distributed in-memory cache	Tachyon [LGZ+14], memcached [NFG+13]
Raw block store	Unstructured distributed store	f4 [MLR+14]
Database	Structured distributed store	Spanner [CDE+13], HBase [BGS+11]
Threading library	Parallel data processing system	Dryad [IBY+07], Spark [ZCD+12]
Synchronization library, configuration (e.g., `/etc`)	Coordination service	Chubby [Bur06], Zookeeper [HKJ+10]
Debugger, tracing system	Distributed tracing framework	Dapper [SBB+10], Finagle [Eri13]

Table 2.6: Many classic local operating system components have equivalents in the current distributed infrastructure systems that make up a *de-facto* data centre operating system.

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 again falling [ROS+11].

As a result, data centre operators have developed distributed infrastructure systems (§2.1.1.1) that meets these needs. I discuss their role in a conceptual "data centre OS" in the next section.

2.2.2 Data centre operating systems

The workloads seen in warehouse-scale data centres (§2.1.1) require cross-machine coordination, resource management, data sharing, and authentication. Purpose-built *distributed infrastructure systems* (§2.1.1.1) serve to provide these facilities to higher-level user applications (§2.1.1.2). User applications depend on the distributed infrastructure systems just like traditional applications depend on the local operating system: they assume that distributed infrastructure services are always available, and rely on them for crucial functionality (e.g. persistent storage). Hence, distributed infrastructure systems function as a *data centre operating system*.

Table 2.6 lists the distributed infrastructure equivalents of traditional, single-machine OS functionality. For example, the cluster manager forms the distributed equivalent of a privileged kernel, as it starts, schedules, and kills tasks. The cluster manager also performs admission control, and partitions hardware resources between different tasks – another traditional kernel responsibility. Likewise, hierarchical distributed file systems are scalable equivalents of traditional local and networked file systems, designed to expose a similar storage abstraction.

However, the list is by necessity a somewhat crude approximation. Some distributed infrastructure does not have a direct local OS equivalent: unstructured storage systems such as blob stores and key-value stores might at best be seen as equivalents of database backends that bypass the kernel file system or build atop it.

In other cases, the analogies are more tenuous: for example, parallel data processing systems in some ways are the distributed equivalent of multi-threading libraries, but often have higher-level programming models and built-in fault-tolerance mechanisms. Likewise, coordination services used for distributed locking, resource and service discovery have some local predecessors (e.g., the UNIX `/etc` file system, and the Windows registry), but must achieve distributed consensus using algorithms like Paxos [CGR07] or Raft [OO14] to offer a similarly consistent view across machines.

Given these similarities, one might wonder why these systems were not built upon abstractions from a classic distributed OS. There are two reasons why developers devised their own abstractions and built new systems from scratch:

1. *Flexibility:* Data centre stacks often have several systems for related ends – e.g. the different caching systems in the Facebook stack (§2.1.1.1). These systems evolved in response to acute business needs, rather than well-known, long-term use cases. Building systems from scratch enabled rapid evolution independent of the much slower standardisation of widely used, general abstractions or OS extensions.

2. *Unavailability:* classic distributed operating systems are no longer deployable, and their abstractions have not survived in, or been adopted into, current OSes. As such, no widely-deployed abstractions for building distributed systems were available.

This organic evolution of data centre OS software has yielded stacks of systems that work well for their specific use cases and deployments (see §2.1.1). However, it is not without its downsides, and gives rise to several problems and challenges, which I discuss next.

2.2.3 Problems and challenges

Unlike single-machine operating systems and classic distributed OSes, the collection of infrastructure services that make up the data centre OS is somewhat ad-hoc in nature, and the distributed OS components lack uniformity and composability.

Table 2.7 illustrates this by comparing current data centre infrastructure software to classic distributed OSes.[10] At a high level, it is evident that classic distributed operating systems strove for *uniformity* in their abstractions, expecting them to be used by a wide variety of applications.

[10] My survey here relies on widely-used open-source systems (Mesos, HDFS, and Spark) because it is straightforward to determine their precise features (unlike with similar, proprietary systems).

	"Data centre OS" (§2.2.2)			**Typical classic distributed OS** (§2.2.1)
	Mesos [HKZ+11]	HDFS [SKR+10]	Spark [ZCD+12]	
Data sharing	container volume mapping	files via wire protocol	RDDs within a job	uniform object notion
Data caching	✗	✗	only within a job	uniform notion of cached objects
Controller architecture	single active master	single active NameNode	single master	no single point of control
User/task isolation	processes, containers	req. multiple NameNodes	multiple deployments	built-in/via capabilities
Resource naming	internal identifiers	file paths, block IDs	internal identifiers	uniform GUID scheme
Res. existence deniability	✗	limited (`-r`/`-x` permission)	✗	often via capabilities
Access control	custom ACL	coarse-grained file ACL	✗	uniform capabilities
Audit of resource access	✗	via request audit log	✗	✗ (typically)

Table 2.7: Comparison of current distributed infrastructure systems that function as a "data centre OS" and classic distributed operating systems.

The systems in the data centre OS, however, each have their own approaches, APIs, and abstractions. In the following, I discuss the effects of this difference on the efficiency and security of data centre infrastructure.

2.2.3.1 Efficiency

Different distributed infrastructure components use and expose different representations for the data they store and process: consider, for example, an HDFS file *vs.* a Spark RDD *vs.* a value in BigTable's three-dimensional map [CDG+06]. All these hold arbitrary binary data, but it is impossible to convert from one representation into another without copying the data.

This not only leads to unnecessary copies, but also *complicates sharing data* between different systems: there is no distributed equivalent of a shared read-only memory mapping, for example, of a set of photos between a web server, a distributed file system, and an analytics job. Indeed, as a result of this, data centre infrastructure has ended up with multiple, uncoordinated and conflicting implementations of data caching: the local machine OS buffer cache, key-value stores like memcached [NFG+13], and file system caching overlays like Tachyon [LGZ+14].

The existing distributed systems' abstractions also often lack *support for introspection*. Without knowing system-specific API details, it is difficult or impossible to find out the location of data, their persistence, or where they are replicated. Yet, this information can be crucial to cross-system optimisation. For example, it can be more efficient to read from a data set cached in

memory on another machine than to read from the local disk, but writes may need replicating in different failure domains.

Finally, most infrastructure systems that manage resources in current data centres have a *centralised controller*: examples are the BorgMaster in Borg, the NameNode in HDFS, and the Spark master in Spark. "High-availability" extensions that replicate the controller's state across multiple backup instances handle controller failure [VPK+15, §3.1; MesosDocs15; WQY+09; SparkDocs15], but do not manage resources and their meta-data in a distributed way.

Therefore, the efficiency of a data centre OS depends on having more uniform, introspection-enabled, and distributed resource management abstractions. Indeed, future data centre hardware trends – such as the expected wide deployment of RDMA [ORS+11; DNN+15] – will likely only exacerbate this need.

2.2.3.2 Security

Within a single machine, OS kernels isolate different processes using virtual address spaces, and kernel namespaces. Isolation and access control in distributed systems, however, are left to the applications.

In a data centre, *isolation* between different applications' tasks and different users is crucial. Typically, it is implemented by the cluster manager compartmentalising application tasks into virtual machines (VMs) or OS-level containers. These isolation mechanisms have the advantage that root file systems, software packages, and running processes are isolated between different users, applications, and tasks. However, distributed storage and data processing systems only support such isolation by running entirely separate deployments (e.g. Spark), or separate meta-data servers (HDFS), which conflicts with the efficiency requirements, since these separate deployments cannot share any data.

Another key aspect of data centre security is protection of resources, i.e. deciding whether a user can discover, access, and interact with them. Access control protection is traditionally based either 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.

Current distributed systems pick and mix these approaches: for example, web applications use cryptographic identifiers as part of URLs to identify data such as photos or messages, and such

identifiers are also used as lookup keys in key-value stores. By contrast, distributed file systems such as HDFS, rely on traditional POSIX-style access control, i.e. user and group ownership, and read/write/execute permissions.

Resource naming in distributed applications is often capability-based, as in the example of cryptographic identifiers. Such capabilities are useful because they can be stored and communicated as plain data (e.g. as part of a URL), but are coarse-grained: for example, delegation of a restricted version of the capability is not usually possible. Moreover, resource naming schemes are different across systems, making it impossible to reference a resource independently of the system managing it.

Naming is also intimately related to *deniability of resource existence*: if a user obtains the key to a data item in a key-value store, they can discover that it exists. Most existing systems do not have namespaces, and hence cannot compartmentalise resource discovery or selectively deny resource existence, no matter whether they are capability-based or ACL-based. For example, memcached namespaces must be implemented using key prefixes or suffixes, and HDFS directory listing can only be denied to the extent of UNIX file system permissions' expressivity.

The current *access control* schemes are not only too coarse-grained, but also lack uniformity: systems have their own custom notions of subjects (e.g. users/groups in HDFS, users and frameworks in Mesos), authentication methods, and restrictable properties. Since each system implements access control from scratch and often incompletely, end-to-end security and isolation of multiple users accessing the same data centre OS components are generally poor.

Finally, data centre operators need to *audit* the attempts to access resources. Unfortunately, current systems either do not support auditing at all, or merely have support for bespoke audit logs (e.g. in HDFS).

Classic distributed operating systems solved many of these problems by relying on capability protection, primarily for two reasons. First, while both ACLs and capabilities can be managed distributedly, capabilities lend themselves to distributed use as they require no authentication of a subject using them. Second, 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 a fully distributed environment.

Clearly, current data centre infrastructure would benefit from more fine-grained protection and access control implemented using uniform approaches. Therefore, good security and fine-grained protection in a data centre environment require a pervasive, fine-grained capability-based access control.

2.2.3.3 Observations

This high-level survey of the current data centre systems software acting as an OS suggests that there is substantial potential for improvement. Indeed, others have made similar observations

and also called for more uniform, reusable, and composable abstractions and mechanisms in a data centre OS [ZHK+11].

My key insight is that classic distributed OSes already solved many of the above issues – albeit in a somewhat different context – and that, consequently, we can and should learn from them. As a result, we may be able to identify new common distributed OS abstractions that provide current data centre OS components with more uniform, efficient, and secure resource management and access control.

The next section looks at whether we can feasibly hope to support existing data centre applications' needs with only a small number of clean-slate abstractions.

2.2.4 Feasibility of new abstractions

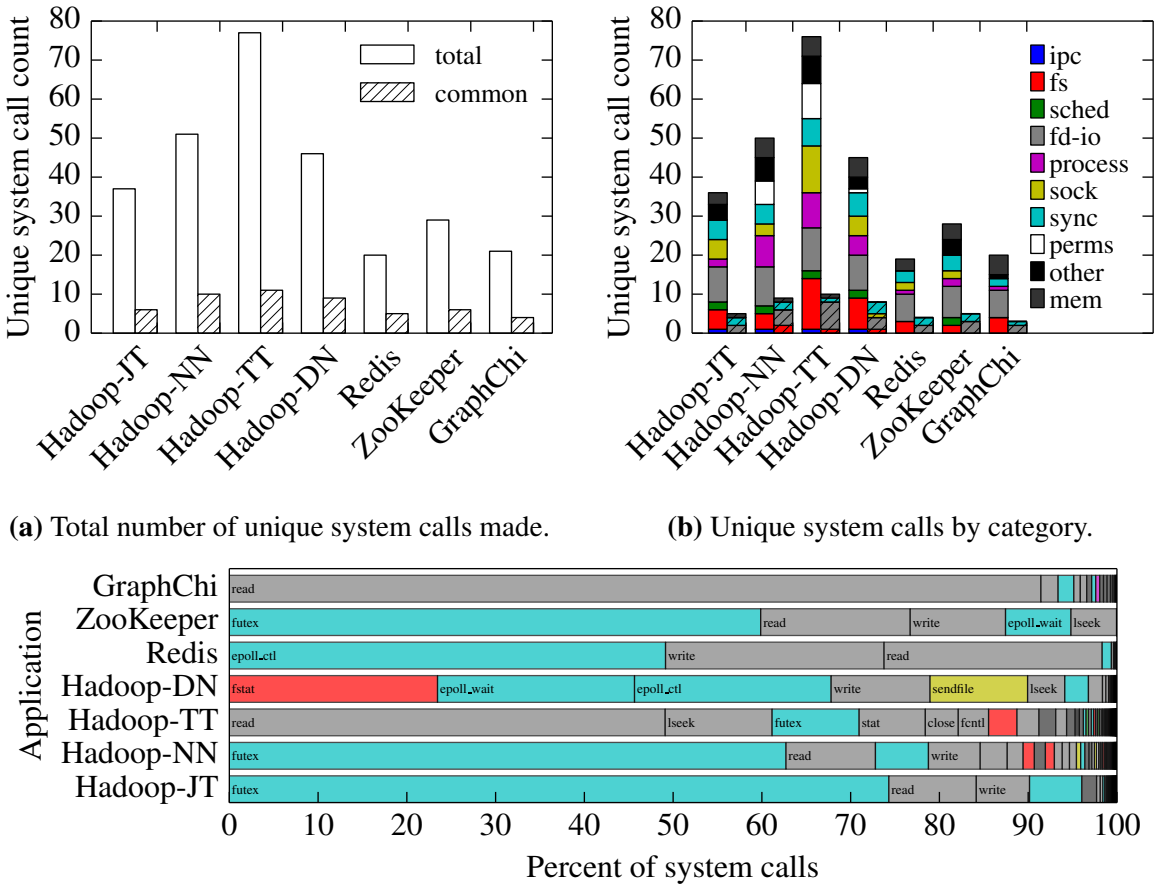
The idea of new resource management and access control abstractions for a data centre OS necessarily raises the questions of complexity: how many such abstractions will be required to support common applications, and can we move applications to *only* use the new abstractions? The latter is interesting from a security perspective: applications that can only use the new abstractions cannot possibly bypass their access control mechanism.

The recent resurgence of interest in *library operating systems*, which run a full OS kernel API atop a typically much smaller host ABI, suggests that even a small set of abstractions is sufficient to support a wide variety of applications. 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.[11]

The key question is whether a small set of operations can be sufficient to support data centre applications – or even only their performance-sensitive "data plane" – efficiently. If so, a clean-slate model for a data centre OS seems feasible. In the following, I present an exploratory study that offers some insight into this question.

Experiment. I use Linux system calls as a (crude) proxy for OS functionality invoked by typical data centre 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`. This does, of course, fail to capture the higher-level, distributed operations invoked, but it gives an indication of what minimum local OS functionality a clean-slate model would have to provide.

[11] As of kernel version 3.14.



(a) Total number of unique system calls made.

(b) Unique system calls by category.

(c) Relative system call invocation frequency. Colours correspond to the categories in Figure 2.8b.

Figure 2.8: Results of exploratory system call tracing on common data centre applications.

Figure 2.8a compares the total numbers of distinct Linux system calls invoked by the different applications. For Hadoop, I measured the four different system components of a typical Hadoop cluster: *(i)*, the MapReduce JobTracker ("Hadoop-JT"), which coordinates job execution; *(ii)*, the MapReduce TaskTracker ("Hadoop-TT"), which represents a worker node; *(iii)*, the HDFS NameNode ("Hadoop-NN"), which manages 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 breadth of OS functionality used 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 considering 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 (i.e. on the "data-plane"). Indeed, five or fewer "common" system calls exist for all applications apart from Hadoop (Figure 2.8a).

In Figure 2.8b, I group the system calls by category. Perhaps unsurprisingly, the dominant cat-

egories are I/O, synchronisation, and resource management, with I/O system calls dominating as we might expect in distributed, data-intensive applications.

Figure 2.8c 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 `read(2)` or `write(2)`; the next most frequent calls are those relating to Linux's futex ("fast user-space mutex") synchronisation mechanism. All in all, a total of eleven system calls cover 99% of the invocations made. This is a small subset of the 326 Linux system calls, suggesting that the full breadth of the Linux system call API is not required on the "data-plane" of these data centre applications.

Observations. While these results hint that many data centre applications only use a small set of local OS abstractions, one ought to be cautious. Many of the extra system calls offered in Linux exist solely for backwards compatibility and are rarely used, while others may only be used on rare code paths (e.g. for error handling), and even others like `ioctl(2)`[12] and `fnctl(2)` have highly overloaded semantics.

Nevertheless, these results are encouraging: they suggest that it ought to be feasible to build a new distributed data centre OS whose abstractions are sufficient to support common distributed applications without requiring recourse or access to "legacy" OS abstractions in the common case. This has the dual advantages of *(i)* making it possible to evaluate the performance of a clean-slate "data-plane" built upon the new abstractions, and *(ii)* allowing access to legacy OS facilities to be withdrawn, preventing potential security exploits and side-channels (e.g. via `pipe(2)`).

2.2.5 Summary

In this section, I have explained how operating system evolution has been driven by the quest for higher resource utilisation via sharing of underlying hardware. I surveyed the classic distributed OSes of the 1980s, which failed to see adoption, possibly because they were ahead of the workloads of their time (§2.2.1).

I then found that modern data centres *de-facto* have distributed OS composed of multiple distributed infrastructure systems in use today (§2.2.2). However, this approach has several downsides, and faces efficiency and security challenges (§2.2.3):

1. I observed that the distributed infrastructure systems lack the uniform abstractions of classic distributed OSes.

2. As a result, the efficiency of the data centre OS overall suffers: data must be copied and transformed across systems' representations, are duplicated when shared, and systems cannot easily take advantage of new hardware paradigms.

[12] I use the conventional Unix manual page notation, `name(section)` throughout this dissertation.

3. The lack of a single access control scheme reduces the end-to-end security of the data centre OS: systems have their own notion of authentication, use access control primitives of variable granularity, and selective delegation is often impossible.

Consequently, it seems useful to consider ideas from classic distributed operating systems in the context of a data centre OS. Indeed, I argued with the aid of an exploratory study that the narrow and uniform requirements of data centre applications make it feasible to devise a new set of distributed OS abstractions that fully support the "data-plane" of such applications (§2.2.4).

In the next section, I look at a specific part of the data centre OS: the cluster manager's scheduler, which decides where to place both application tasks in the shared cluster infrastructure.

2.3 Cluster scheduling

As noted in the previous section, the *cluster scheduler* is a crucial part of the infrastructure systems that function as the data centre OS today. Moreover, the challenges posed by hardware heterogeneity and co-location interference (§2.1.2–2.1.3) can be addressed, or at least mitigated, by good cluster-level scheduling.

Scheduling work (such as parallel tasks) to compute resources (such as machines) 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 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 2.8 summarises the core design goals of each system.

2.3.1 Scheduler architecture

Existing cluster schedulers differ in their *architecture*: the degree to which decisions are made in a centralised or distributed fashion. Figure 2.9 illustrates the approaches that I discuss.

Most early cluster 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 2.9a). Scheduler fault-tolerance can be implemented via primary/backup fail-over, or by restarting the scheduler from a previously saved checkpoint.

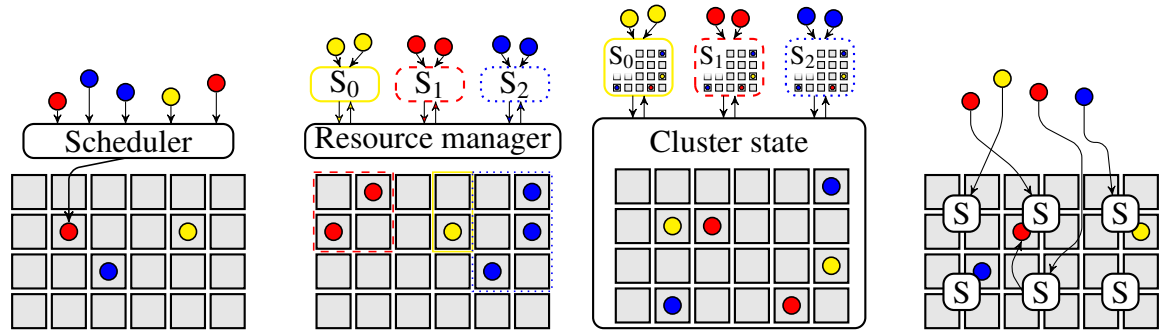
Recent work, however, has introduced distributed cluster scheduler architectures, albeit with varying motivations:

System [Reference]	Target workload	Distributed	Data locality	Fairness	Soft constraints	Hard constraints	Avoid interference	Auto-scaling	Heterog. machines
HFS [HFS]	MapReduce tasks	✗	✓	✓	✗	✗	✗	✗	✗
LATE [ZKJ+08]	MapReduce tasks	✗	✗	✗	✗	✗	✗	✗	(✓)
Quincy [IPC+09]	Dryad tasks	✗	✓	✓	✓	✗	✗	✗	✗
Delay Sched. [ZBS+10]	Hadoop tasks	✗	✓	✓	✗	✗	✗	✗	✗
Mesos [HKZ+11]	Framework tasks	✗	✓	✓	✗	✗	✗	✗	✗
CIEL [Mur11, §4.3]	CIEL tasks	✗	✓	✗	✗	✗	✗	✗	✗
Jockey [FBK+12]	SCOPE tasks	✗	✗	✓	✗	✗	✗	✓	✗
alsched [TCG+12]	Binaries (sim.)	✗	✓	✗	✓	✓	✗	✗	✓
tetrisched [TZP+16]	Binaries	✗	✓	(✓)	✓	✓	✗	✗	✓
Whare-Map [MT13]	Binaries	✗	✗	✗	✗	✗	✓	✗	✓
YARN [VMD+13]	YARN tasks	✗	✓	(✓)	✗	✗	✗	✗	✗
Omega [SKA+13]	Google tasks	✓	(✓)	✗	✓	✓	✗	(✓)	✗
Sparrow [OWZ+13]	Shark queries	✓	✓	✓	✗	✓	✗	✗	✗
H-DRF [BCF+13]	Hadoop v2 tasks	✗	✗	✓	✗	✗	✗	✗	✗
Choosy [GZS+13]	Framework tasks	✗	✓	✓	✓	✓	✗	✗	✗
Paragon [DK13]	Mixed binaries	✗	✗	✗	✗	✗	✓	✗	✓
Quasar [DK14]	Mixed binaries	✗	✗	✗	(✓)	✗	✓	✓	✓
Apollo [BEL+14]	SCOPE tasks	✓	✓	✓	✗	✗	✗	✗	✓
KMN [VPA+14]	Spark tasks	✗	✓	✗	(✓)	✗	✗	✗	✗
Tarcil [DSK15]	Mixed binaries	✓	✓	✗	✓	✗	✓	✗	✓
Hawk [DDK+15]	Binaries (sim.)	✓	✓	✗	✗	✗	✗	✗	✗
Mercury [KRC+15]	YARN containers	✓	✓	✓	(✓)	✗	✗	✗	✗
Bistro [GSW15]	Facebook tasks	(✓)	✓	(✓)	✗	✓	✓	✗	✗

Table 2.8: 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.

Resource sharing between specialised frameworks. Data centres run many infrastructure systems and applications concurrently (§2.1.1), which requires cluster resources to be partitioned across users and systems. Many infrastructure systems also perform their own application-level task scheduling (e.g. MapReduce assigns map and reduce tasks). **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 2.9b).

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



(a) Centralised, monolithic scheduler.

(b) Two-level partitioned scheduler.

(c) Shared-state scheduler.

(d) Fully distributed scheduler.

Figure 2.9: Comparison of different cluster scheduler architectures. Grey boxes represent cluster machines, and coloured circles correspond to different applications' tasks.

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, multiplexing resources via offers (Mesos) or requests (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 cluster. 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 2.9c). Transactions may yield a successful task placement, or fail, necessitating a re-try. Microsoft's Apollo takes this model a step further and only detects and resolves conflicts in worker-side queues on cluster machines [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 2.9d).

Different solutions are appropriate for different environments, and none of these architectures is necessarily "better" than the others.

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].

Early work optimised for disk locality, since reading data from a local disk was faster than transferring it over the network. Data centre network bandwidth has increased, however, making differences between local and remote disks increasingly marginal [AGS+11]. Locality is still important, however: many data processing frameworks [ZCD+12] and storage systems [ORS+11; LGZ+14] cache data in RAM, making the locality of in-memory objects an important concern. Moreover, recent machine-learning workloads have locality constraints on GPUs [RYC+13; MAP+15; TZP+16], and require careful placement to ensure sufficient network bandwidth between their tasks. Even disaggregated storage architectures based on NVMe flash devices [KKT+16], which have no local data on each machine, require careful, workload-aware placement in locations with sufficient CPU and network capacity [NSW+15].

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 only to the local kernel CPU scheduler, not to the cluster scheduler.

2.3.3 Constraints

Not all resources are necessarily equally suitable for a particular 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's Borg 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 crawling 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 (or made available vi preemption). 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 [MHC+10; SCH+11; 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 [GZS+13; OWZ+13]. 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" [IPC+09, §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 [VMD+13, §3.2], and alsched [TCG+12] also supports both, in addition to complex combinatorial constraints. However, tetrisched subsequently argued that support for soft constraints is sufficient and offers attractive benefits [TZP+16].

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.

2.3.4 Fairness

Most warehouse-scale data centres are operated by a single authority, but nevertheless run a wide range of workloads from different organisational units, teams, or external customers [SKA+13, §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*.

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 queueing 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 data centre. 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 (§2.1.2) can lead to significant differences in seemingly identical resource allocations.

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 2.1.4, good scheduling decisions are essential to the efficient use of cluster resources. This section has surveyed many existing cluster schedulers. I started by looking at their architecture (§2.3.1), and then discussed several desirable features: support for locality within the cluster (§2.3.2), placement constraints (§2.3.3), fairness (§2.3.4), and dynamic resource adjustment (§2.3.5).

Few existing schedulers, however, address the machine heterogeneity and workload interference challenges highlighted earlier. Yet, in order to achieve deterministic workload performance at high cluster utilisation, a cluster scheduler must:

1. integrate machine-level information and cluster-level information to make scheduling decisions based on fine-grained task profiling, machine type, and locality information;
2. avoid negative interference between workloads that share hardware resources by co-locating only those tasks that work well together; and
3. flexibly allow workload-specific scheduling policies to be expressed, so that the scheduler can be tailored towards the desired use case.

In Chapters 6–8, I will discuss Firmament, a new scheduler that supports these goals.

Chapter 3

A decentralised data centre OS model

In the previous chapters, I have highlighted the need for a more uniform, clean-slate foundation for a data centre OS, replacing the ad-hoc abstractions built as part of distributed infrastructure systems. I argued that such a replacement has the potential to make critical infrastructure more secure and efficient.

In this chapter, I introduce a reference model for resource naming and management in a *decentralised, data centre-scale operating system*. This reference model informs the implementation of a prototype, DIOS, which I will discuss in Chapter 4.

I begin my description of the decentralised data centre OS model by defining key terms and concepts upon which the model relies (§3.1). Based on the challenges identified in Section 2.2.3, I then state efficiency goals and security requirements for the data centre OS model (§3.2).

I then describe the six key aspects of the decentralised data centre OS model:

1. The core abstractions of **distributed objects**, the building blocks which allow the model to support the construction of diverse distributed applications (§3.3).
2. **Resource naming**, which uses globally unique, namespaced identifiers that act as capabilities for object discovery, and which can both be exposed to end-users and stored as serialised data (§3.4).
3. Runtime **resource management** and addressing, supported via translucent handles that expose context-dependent object meta-data and act as delegatable capabilities (§3.5).
4. How **persistent storage** of data is achieved in the model via a flat object store, which flexibly supports different application-specific storage systems (§3.6).
5. **Concurrent access** to distributed objects, which the model supports via a transaction-like I/O requests, which support different application-level concurrency models and consistency levels (§3.7).

Finally, I summarise the chapter (§3.8), before I move on to discuss my prototype implementation of this model, DIOS, in the next chapter.

3.1 Definitions and concepts

Systems software research often relies on analogies between concepts to relate them to each other, and to position new ones. For example, my survey of existing data centre infrastructure systems that function as an "operating system" for the data centre (§2.2.2) relied on loose analogies of these systems with classic OS functionality.

To specify my data centre OS model without ambiguity and contradiction, I need to be more precise. Hence, I assume the terms and definitions listed in the following.

Human users. There are three different categories of human users who directly or indirectly interact with the data centre OS:

1. An **operator** is a core systems developer or system administrator who has privileged access to the cluster infrastructure, and who controls the admission and access control policies applied by the data centre OS. The cluster manager developers are also operators.

2. An **infrastructure user** of the data centre OS is a developer who deploys application workloads (§2.1.1.2) on the shared cluster infrastructure. Infrastructure users are subjects of access control and security mechanisms.

3. Finally, an **end user** is a user of an application or web service that is backed by the data centre application workloads. End users may store their data in the data centre, and interact with applications via their APIs, but they do not have direct access to the infrastructure and cannot deploy their own applications.[1]

Systems software. The key pieces of operator-provided systems software that infrastructure users interact with are:

- The data centre **operating system** (OS), which is the complete set of privileged code, system services, runtimes and libraries required to support infrastructure users' applications. Crucially, the OS contains all such software that is not ordinarily provided by an infrastructure user, and which is ubiquitously available throughout the data centre.

- The **trusted computing base** (TCB) consists of those parts of the operating system that are critical to maintaining its security, and whose compromise may jeopardise access control. The TCB consists of all machines' local OS kernel and the cluster management software, but does *not* include other infrastructure services (e.g. a distributed storage system).

[1] While "cloud computing" may enable any end user to deploy applications in the data centre, they become an *infrastructure user* when doing so.

- The local OS **kernel** is the privileged code that runs on each machine to initialise, manage and perform privileged operations on its hardware. It can access any physical memory, and starts and terminates local programs. The kernel is part of the TCB, since compromising it allows bypassing access controls on (at least) local data.
- The **cluster manager** does not run inside the kernel, but is nevertheless part of the TCB, since it starts and stops application tasks within the cluster infrastructure, furnishes them with inputs and outputs, and isolates them from other tasks. One or more instances of the cluster manager run within the data centre, deployed and configured by operators, and handling infrastructure users' requests to deploy applications.

Information hiding. When system designers conceive abstractions, they face a choice between exposing and hiding information. Different approaches can be characterised as follows:

- An abstraction or operation is **transparent** if it is entirely invisible to the user, i.e. the infrastructure user does not need to be aware of what is happening and the specifics of how it is implemented, and indeed has no way of finding out.
- By contrast, the opposite approach is **opaque**: an abstraction or operation is opaque if it does *not* hide any of the underlying details from the infrastructure user, or even requires them to be specified explicitly.
- A **translucent** approach takes the middle ground, hiding implementation details from the infrastructure user by default, but allowing for introspection. In other words, translucent abstractions or operations *may* be treated as transparent or as opaque at the infrastructure user's choosing.

I define any other terms and concepts specific to my model when they are first introduced.

3.2 Requirements

My survey of existing data centre OS software revealed two challenges that are poorly addressed by existing systems (§2.2.3):

1. *Efficient sharing of resources* between tasks that are part of different jobs, distributed infrastructure systems, and applications. I found that data are being copied unnecessarily, and that crucial information is often hidden even though systems share data.
2. *Fine-grained protection and access control* are lacking, and the existing mechanisms are coarse, fragmented, and implemented as part of distributed infrastructure systems that each invent their own, bespoke notion of protection. This makes it difficult to securely delegate access to resources,

In the following, I concretise these challenges into a set of goals and requirements for the decentralised data centre OS model.

3.2.1 Efficiency requirements

To make the decentralised data centre OS as efficient as possible, the model must meet two high-level goals:

1. It must supply uniform abstractions that applications can be built upon without having to unnecessarily transform data or resource representations.
2. It must expose sufficient information for efficient implementations of applications, and avoid mandating abstractions that can be costly.

In other words, we need an efficient and expressive, but yet minimal common denominator over many applications, meeting three requirements.

1. The model and its abstractions must be *scalable*, so that the OS can run across many machines and serve a large number of applications. For example, different replicas of a resource should be concurrently accessible without synchronisation.
2. Abstractions should be *uniform* across applications, and easy to comprehend. For example, since the location of a resource may not be known in advance, the abstractions for accessing local and remote resources should be identical.
3. The abstractions should be *introspectible*, allowing applications to obtain information required for efficient operation. For example, the abstraction for a remote resource should expose the fact that it is remote, but access should not require implementing a wire protocol.

Since both data centre machines and the network may fail, my model is *decentralised*: at the level of OS abstractions, it is a **fully distributed architecture**. The OS itself must not rely on singular centralised state, and must allow replication of state, which aids scalability and fault tolerance. Specifically, all data stored in the system must be replicable, and no central authority must be required to maintain meta-data or permissions.

Note that this does *not* mean that all applications or infrastructure systems built atop the OS abstractions must be fully decentralised. Indeed, most infrastructure systems atop this model would likely still use a centralised controller; however, the OS abstractions should not themselves require any centralised control or depend on a single point of failure.

Scalability of the data centre OS abstractions and interfaces can be derived from the scalable principles developed for single-machine operating systems on many-core architectures [BCC+08;

BCM+10a; CKZ13]: like a scalable multi-threaded program, a scalable distributed system must exploit asynchrony and coordination-free sharing, and avoid global atomic operations. OS abstractions should encourage commutative implementations where possible [CKZ+13].

More generally, since synchronisation in a distributed system is expensive, the model must **avoid implicit synchronisation**, i.e. synchronisation that the infrastructure user is potentially unaware of. The data centre OS should *inform* applications of concurrent access, but should not prevent it by default or itself enforce synchronisation or mutual exclusion. Infrastructure systems and applications may, however, construct synchronised abstractions on top of the concurrent access notifications exposed by the OS.

Distributed systems commonly use *transparency* to hide implementation details – consider, for example, 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, full transparency has the disadvantage of potentially unpredictable performance, since no *introspection* is possible. 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 see high latency, but the application cannot detect in advance whether it will.

Translucency serves to offer the simplicity of transparency without restricting applications' freedom to introspect: it makes 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 the details.[2] The OS abstractions in my model should therefore be translucent, but also carry contextual meta-data that expose detailed information about resources for introspection. The application may choose to ignore this information (and forgo predictable performance) or may use it to optimise (e.g. choose a more proximate replica).

3.2.2 Security requirements

As noted in Section 2.2.3, the security of current distributed infrastructure systems leaves much to be desired: it is fragmented, enforced selectively, and often insufficiently fine-grained.

My decentralised data centre OS model aims to improve upon this. In the following, I review the threat models considered, and the security principals that the model must support.

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

1. **Isolation** between different infrastructure users and different tasks, and their data and resources, unless explicitly shared.

[2] The term "translucency", meaning "transparency with exposed meta-data", also sees occasional use in programming language literature, e.g. by Rozas [Roz93].

2. **Deniability** of the existence of resources inaccessible to a task or infrastructure user.

3. **Auditability** of any sharing or communication of data or resources, and of access to them.

Unlike in a traditional OS, isolation and sharing must also extend across machines.

For example, the existence of a specific end-user email may only be visible to serving and analytics jobs that are authorised to use it. However, a task in an analytics job might need to delegate work to helper tasks (e.g. a decompression routine, or a virus scanner) without leaking sensitive data outside the trusted job, and the fact that it does should be recorded in an audit log.

3.2.2.1 Definitions

The security principals and resources in my model are defined as follows:

- **Subjects** (*principals*) are infrastructure users, or non-human service accounts.
- **Objects** (*resources*) are items of volatile or persistent data, hardware resources, tasks, and other OS-level abstractions (e.g. IPC endpoints, timers).

As I already noted in Section 3.1, the trusted computing base (TCB) – which may access any object, and which furnishes subjects with their rights – contains the local machine kernels and the cluster manager. Additionally, the identifier resolution mechanism, which I discuss in Section 3.4, is also part of the TCB.

3.2.2.2 Threat model

My model aims to defend the data centre infrastructure and its users against threats that arise for two reasons: *deliberate malice* and *accidental mistakes*. More specifically, there are four threats that my model seeks to address:

Internal malice occurs when an evil, or subverted, principal attempts to gain access to a resource that ought not to be available to it. For example, a malicious infrastructure user may run a specially crafted MapReduce job that leaks its inputs to a third party. My model's goal is to compartmentalise resources to restrict infrastructure users' access to *only* those resources that their jobs need and legitimately access, minimising exposure. This requires more fine-grained access control than current systems provide.

External compromise happens when an external party obtains unauthorised access, typically by exploiting and impersonating an existing principal. If subverting a valid principal, this threat becomes identical to *internal malice*. For example, a malicious end-user might exploit a vulnerability in a decompression routine to gain access to the infrastructure and subvert the identity of the invoking task. My model's goal is to make such compromise

less likely by restricting the abstractions accessible to distributed applications, and to contain the compromise as much as possible if it does happen.

Accidental exposure of private resources can occur without malice, due to bugs or incorrectly configured permissions. For example, the permissions of end-user data in a distributed file system might be insufficiently restrictive, allowing a buggy analytics job to accidentally expose more data than intended. My model's goal is to reduce the exposure in this situation by restricting tasks' access to the minimal resources required.

Hardware failures typically occur without malicious intent, but lead to sudden unavailability of resources that may have security or liveness implications. For example, a network partition can make an authentication server temporarily unreachable. My model's goal is to support access control that remains secure and available even under failures, as long as the target resource remains available.

However, several types of threats are out of the scope of my work, and must be mitigated using other mechanisms:

Compromise of the TCB may occur if an evil subject or an external attacker successfully exploits a vulnerability in the OS kernel, the name service, or in the cluster manager. Encrypted memory enclaves might help reduce application exposure to such attacks [BPH14; CD16].

Hardware and physical access exploits can circumvent access control and potentially expose private information. It is difficult for an OS to defend against such threats in the absence of end-to-end encryption and hardware support for oblivious computation [MLS+13; LHM+15].

Information Flow Control (IFC) involves the enforcement of policies on whether information can be transferred between different subjects, including indirectly. While my model restricts accidental exposure of information, and reduces the attack surface via fine-grained access control, IFC techniques such as taint tracking (e.g. in HiStar [ZBK+06] and DStar [ZBM08]) are required to enforce such policies.

3.2.3 Other requirements

There are several additional requirements the decentralised data centre OS model must meet, but which are not directly related to efficiency or security.

Distributed applications are complex, and many policy decisions are application-specific. A data centre operating system must thus balance specifying mechanism against maintaining the flexibility to support **externalised policies** specified by applications. To achieve this, sufficient information must be exposed to applications to enact common distributed systems policies –

such as where to locate data, or whether to maintain strongly consistent replicas – of their choice.

Incremental adoption. Even though a new data centre OS model has many benefits, porting software to it will be time-consuming. Hence, an incremental adoption must be feasible, with increasingly larger portions of data centre workloads using the new paradigms, while legacy applications continue to work. While not all of the benefits of the model might initially be attainable, incremental migration should increase efficiency and security.

Next, I explain how a distributed object abstraction helps my model meet these and the previously outlined requirements.

3.3 Distributed objects

Objects are a convenient abstraction commonly used both for programming and structured data storage. Their lowest common denominator definition 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 data and delineate scopes for update consistency. 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) [ZCD+12] and CIEL's data objects [MSS+11] – offer fault tolerance based on replication and deterministic replay.

3. Objects simplify distributed programming, as they lend themselves to a communicating actor model in which actors maintain private state and exchange messages to effect state changes. For instance, Sapphire can concisely express complex distributed systems via transparently interacting objects in combination with modular "deployment managers" [ZSA+14, §8.1].

4. Dependencies between objects enable data-flow computation models that lend themselves to automatic parallelisation. For example, Dryad [IBY+07] and Naiad [MMI+13] are based on data-flow of records between stateful vertices, while CIEL schedules dynamically generated tasks based on their dependencies on input objects [MSS+11, §3.1].

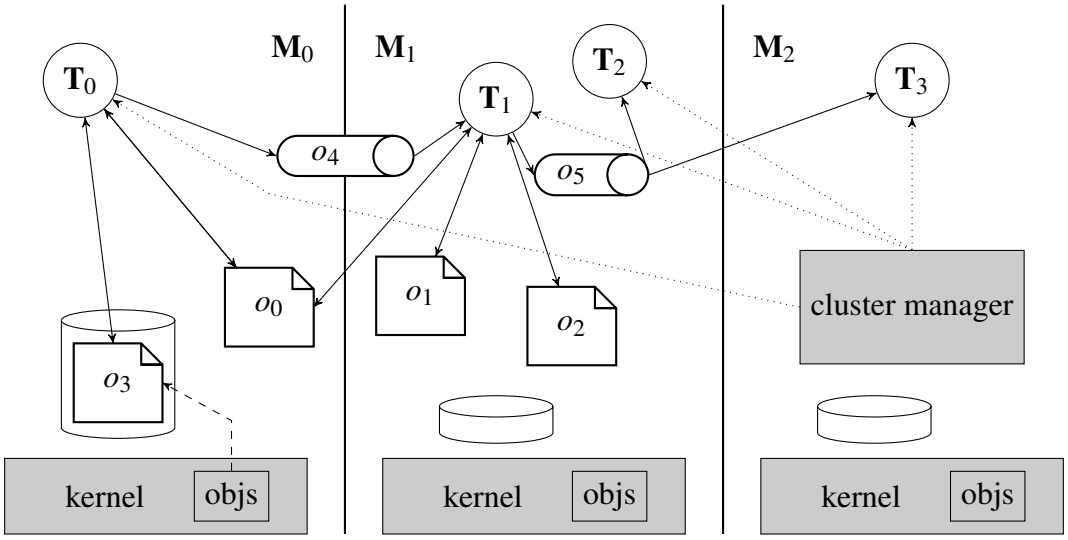


Figure 3.1: Schematic overview of the decentralised data centre OS model. Three task objects, $\mathbf{T}_0$–$\mathbf{T}_2$, are shown as circles; four physical blob objects are represented as documents; and two streaming communication objects as pipes. The shaded parts of the system are part of the TCB. Solid arrows indicate data flow, dotted arrows object creation, and dashed arrows a persistent reference.

These, combined with the requirements of a fully distributed architecture and support for extensible application-level policies, make a distributed object model a good fit for a data centre operating system.

My model is centred around two separate object notions:

- A *logical object* is named by a single, globally unique identifier, but can have many replicated instances that are interchangeable for application purposes.
- A *physical object*, by contrast, is a single, specific object instance with clearly defined location, and a target on which applications can perform operations or I/O. Each physical object is part of exactly one logical object.

Otherwise, the distributed object model is based on a deliberately broad definition and only assumes that:

(i) each logical object has a globally unique identifier;
(ii) a logical object's replicated instances ($\equiv$ corresponding physical objects) may be used interchangeably and carry the same globally unique identifier;
(iii) each logical object has a type, which is either a passive *blob* (a sequence of bytes), a *streaming communication endpoint* (a FIFO channel), a *task* (a process), or a *group* (a namespace), and its physical object instances are entities of said type;
(iv) a physical object is created and deleted, but not necessarily updated, atomically;
(v) physical object handles expose meta-data that applications can inspect to make policy decisions.

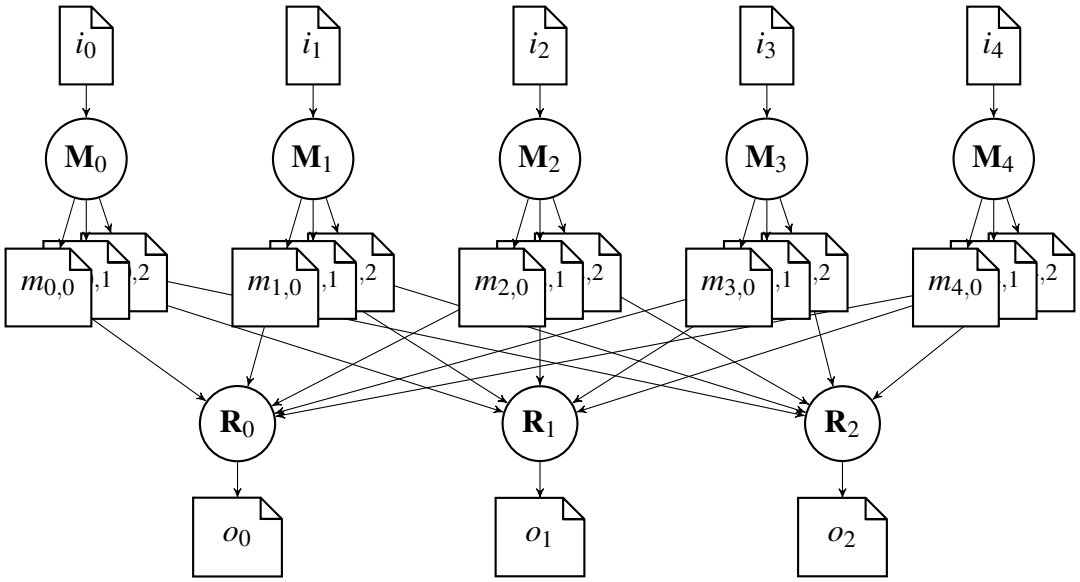


Figure 3.2: Distributed MapReduce (five map tasks, three reduce tasks): a deterministic data-flow batch computation. In this example, five logical input objects ("splits"), i_0–i_4, are transformed into three logical output objects, o_0–o_2, via physical intermediate objects $m_{i,o}$. The intermediate and output objects are created by a MapReduce controller task (not shown), which also spawns the tasks via the cluster manager.

Specifically, my model – unlike the object notions of many classic distributed OSes (§2.2.1) – **does not**:

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

This object notion is practical for a data centre OS as it makes minimal assumptions about the application implementations.

Figure 3.1 illustrates the object concept with an example of four task objects ($\mathbf{T}_0$–$\mathbf{T}_3$) that access private and shared blobs (o_0–o_3) and communicate via unidirectional channels (o_4–o_5). In the following, I show how two typical data centre applications are expressed in my object model.

Examples. Consider *(i)* a distributed MapReduce [DG08] implementation, which represents a typical deterministic data-flow application, and *(ii)* an event-driven HTTP server with an in-memory key-value store back-end, which is typical of an end-user-facing service application.

In the **MapReduce** framework (Figure 3.2), the i^{th} *map task* ($\mathbf{M}_i$) applies a map function of the form `map(key, value)` $\rightarrow \{\langle$`key2, value2`$\rangle\}$ to all records in its input object (i_i). All map tasks run in parallel. The items in each resulting list are subsequently hash-partitioned on `key2` and the resulting intermediate sets of key-value pairs (stored in intermediate physical object $m_{i,j}$) are provided as input to the parallel reduce tasks ($\mathbf{R}_j$ taking $m_{k,j}$ for all k). These apply

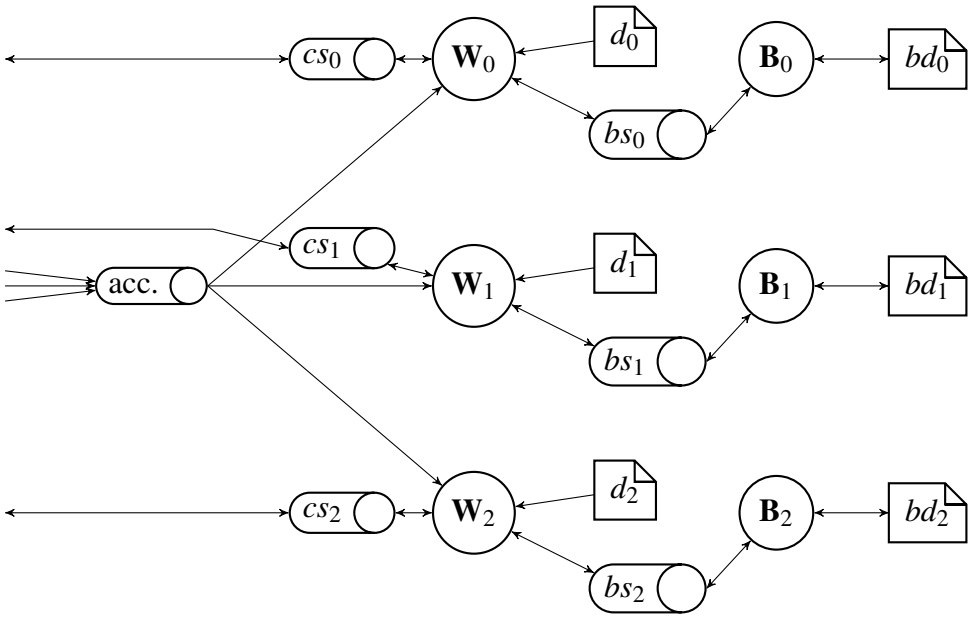


Figure 3.3: Multi-process HTTP server with back-end: the user-facing service is implemented with three front-end worker tasks ($\mathbf{W}_0$–$\mathbf{W}_2$) that share a physical acceptor object (*acc.*) which supplies physical client connection objects (cs_0–cs_1). These serve responses composed from static data (d_i) and dynamic data, bd_i (e.g. from a key-value store), obtained by communicating with three back-end tasks ($\mathbf{B}_0$–$\mathbf{B}_2$) via physical stream objects (bs_0–bs_1).

a reduce function, `reduce(key2, {values})` $\rightarrow$ $\{$`out-values`$\}$, storing the final values in an output object o_j. A controller task manages the map and reduce tasks and monitors them for fault tolerance, re-executing any failed tasks.

The **HTTP server** (Figure 3.3), 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 tasks ($\mathbf{W}_i$) 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 ($\mathbf{B}_i$) over shared memory or the network. Fault tolerance can be implemented by the cluster manager restarting any failed worker tasks.

3.4 Resource naming

Each resource in the decentralised data centre OS model is represented by a logical object. This logical object must be *named*, and my model consequently assigns it a globally unique identifier. Unique identifiers are required for two reasons:

1. The operating system itself must be have a way of unambiguously referring to a particular logical object.

2. Applications must have identifiers for logical objects in order to bootstrap their access to them, and to record and communicate their existence.

In keeping with the definition of my model's logical object notion in the previous section, the identifier refers to all physical objects for a logical object, which must be *interchangeable* (i.e. usable with equivalent semantics) from the application's perspective.

For example, consider a strongly consistent distributed key-value store that requires replication of writes for fault tolerance: its tasks store a value by writing it to multiple physical objects, which in combination form a logical object representing the stored value. To fit the decentralised, distributed data centre OS model's object semantics, the key-value store must maintain the physical objects' interchangeability after serving a write. To achieve this, it must either ensure that *all* physical objects are updated atomically (via application-level locking or transactions), or that the updated value is stored in a new logical object (with a new identifier), and that the mapping in the key-value store is atomically updated to refer to the new logical object.

It is worth noting that physical objects also have unique identifiers, but these are ephemeral: as I explain in Section 3.5, they are only locally valid within a task context, cannot be stored persistently, and are only partially exposed to applications.

Properties. Logical object identifiers, however, are exposed to applications and can be arbitrarily stored and communicated. Hence, they must have three key properties:

1. In order to be suitable for persistent storage and for communication over the network as plain (binary or text) data, the identifiers must be *serialisable*.

2. To make them usable anywhere in the data centre, and to decouple them from physical objects, identifiers are *location-independent* and do not encode the physical location of a resource (unlike, e.g. a `hostname:port` combination).

3. To restrict the impact of leakage, translation of identifiers into physical objects must be limited. Hence, logical object identifiers are *namespaced*: they only form a data centre OS capability for resource identification within a namespace, even though applications manage provenance.

Generation. To meet the requirement of a fully distributed architecture, identifiers need to be generated from local information only. Moreover, for security purposes, they must be unforgeable and unguessable. In many cases, simply picking uniformly random identifiers from a large space is sufficient to meet these needs, and also yields an unambiguous identifier that can be stored as data. This is a good match for the semantics of many existing data centre applications: for example, unforgeable identifiers are already used in URLs referring to photos and other content objects in many web applications.

However, many data centre applications – 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]). The data centre OS model therefore also supports deterministic generation of identifiers using a hash function that combines the chain of previous identifiers to generate new ones.

Resolution. Logical object identifiers exist in order to be translated into sets of interchangeable physical objects that applications can work with. This translation from a location-independent identifier to physical objects that translucently expose location information is called *resolution*, and is the responsibility of a *name service* that forms part of the data centre OS model.

This name service can be implemented by the OS itself, or may itself be delegated to an infrastructure application, but either way forms part of the data centre OS TCB. This comes because the name service knows about all logical and physical objects, and about the mappings between them. This makes it critical to the security goals of resource isolation, deniability of resource existence, and auditability of resource access (see §2.2.3.2 and §3.2.2).

Resource identifiers, when presented to the name service, form *capabilities* for name resolution: the knowledge of a logical object identifier (i.e. being able to name a resource) authorises the presenting task to obtain the corresponding set of physical objects. Importantly, such global, coarse-grained "identifier capabilities" can be treated as data (i.e. stored and communicated) due to their cryptographic generation: possession of the bit pattern is sufficient to authorise identifier resolution via the name service, independent of how the bit pattern was obtained. An identifier capability is atomic, i.e. it cannot be subdivided or refined (e.g. into a capability only to a subset of physical objects).

However, as presented thus far, identifier capabilities have a dangerous security flaw: leaking an identifier's bit pattern enables an attacker to obtain access to the physical objects. To prevent this attack, identifiers are part of one or more namespaces, and are only resolvable within these. The combination of namespaces and identifiers is a form of *split capability* [KRB+03]: given the identifier and access to a namespace that maps it, the holder can discover the corresponding physical objects. Either part on its own, however, is useless.

This restriction grants access only to tasks which have access to an appropriate namespace. Namespaces are themselves represented as objects: they can be created, have identifiers, and can be shared between tasks (albeit subject to some restrictions).

However, even identifiers and namespaces on their own are insufficient to fully implement capability-based protection in the decentralised, distributed data centre OS model. This is for

System	Capability store location	Secured by	Capabilities as data	Location transparent	Refinement possible
Eden [LLA+81]	user-space + kernel	MMU	✗†	✓	✓
Accent [RR81]	user-space + kernel	MMU	✓	✓	✗
Mach [ABB+86]	user-space + kernel	MMU	✓	✓	✗
Amoeba [MRT+90]	user-space	cryptography	✓	✓	✓
Plan 9 [CGP+02]‡	kernel	cryptography	✗	✗	✗
Barrelfish [BBD+09]	kernel (monitor)	MMU	✗	✗	✓
Web URLs	user-space	cryptography	✓	✓	✗
Macaroons [BPE+14]	user-space	cryptography	✓	✓	✓
Identifiers (§3.4)	application (user-space)	cryptography	✓	✓	✗
Handles (§3.5)	user-space + kernel	MMU + crypto.	✗	✗	✓

† 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.

Table 3.1: Previous systems' distributed capability schemes compared to the decentralised data centre OS model.

two reasons: first, identifiers are atomic and do not express fine-grained permissions (e.g. read-/write access); and second, they do not carry sufficient information to form a translucent abstraction (as required for efficiency, cf. §3.2.1). Indeed, it is not possible for a single capability to be application-managed, storeable, communicable, delegatable, and translucent, since these properties are in some cases mutually exclusive (e.g. arbitrary persistent storage and translucency). Instead, my model supports another type of capability used as a context-specific handle for resource management, which I describe in the following section.

Related approaches. Several previous distributed systems made use of identifiers as capabilities (see Table 3.1).

In Eden, capabilities identified "Ejects" [LLA+81], which were distributed objects. Eden's capabilities were location-independent, but could not be treated as data, since – being originally designed to rely on hardware support via "Access Descriptors" in the Intel iAPX 432 [Lev84, §9.3.2] – they were implemented as two-part segregated capabilities, with the Eden kernel holding private meta-data required for their use [ABL+85, p. 50].

The Amoeba distributed operating system used sparse cryptographic capabilities that were managed entirely by user-space server processes [TMV86]. Consequently, capabilities could be treated as data and exchanged over the network. Amoeba capabilities were also location-independent and could be refined. However, Amoeba relied on unforgeable hardware source addresses being network messages for security.

Many modern web applications use cryptographically derived URLs as capabilities.[3] Sharing such a URL amounts to delegating (copying) the capability, although the capability cannot 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.

3.5 Resource management

In addition to identifiers for logical objects, the model also requires an abstraction that implements a secure *handle* to a physical object. This handle must uniquely identify a physical object, carry sufficient information for the holding task to interact with the object (irrespective of its location), and allow introspection via translucent meta-data for efficiency.

Handles are the targets for resource management operations (e.g. sharing, deletion) and I/O on physical objects' data. Unlike identifiers, they carry permissions and other meta-data (e.g. a location, a persistency level, etc.). The values of some of the meta-data, such as locality, depend on the observer. Thus, a handle is owned by a specific task and its meta-data specific to the owner task and the context of the physical object.

Consequently, handles are ephemeral and only persist as long as their owner task: they cannot be stored persistently. For initially unreferenced physical objects (e.g. on persistent storage), this means that handles must be created by resolving their identifiers, as only these can be stored as data.

To ensure that handles can only be generated by identifier resolution, they must guarantee unforgeability; i.e. it must not be possible for an application to gain access to a physical object by means of synthesising a fake handle. Handles therefore form local, context-sensitive **handle capabilities**, which cannot be treated as data, and which are a type of *segregated capability* between the TCB and the application. Each handle has a public part that is exposed to the application, and a TCB-private counterpart that is accessible only to the local OS kernel.

Delegation. The data centre OS must sometimes be able to generate restricted versions of a capability, e.g. removing permissions. Unlike identifiers, handle capabilities can be refined by the TCB. This is useful, for example, when a helper task needs restricted access to only a subset of the permissions available to the parent task. Such *refinement*[4] of capabilities is achieved by the model supporting *delegation* of handles.

Delegation involves creating a new, possibly more restricted, copy of a handle via the TCB. The new handle may be owned by the same task, or by a different task, as the original handle. It

[3] Consider, for example, the "share via link" functionality in Google Docs (`http://docs.google.com`) or Doodle (`http://doodle.com`).

[4] Sometimes also called "reification" or "minting" in capability literature.

is the TCB's responsibility to adapt the handle to the target context, to create the new handle's kernel and application parts, and to notify the target context of the delegation.

When handle capabilities are delegated across machines, they must be sent over the data centre interconnect. This requires unforgeability of delegated capabilities across machines, which is non-trivial to attain in 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, but can be defended against by ensuring the freshness of delegated handle capabilities.

Authentication protocols such as Kerberos [SNS88] or the Needham-Schroeder-Lowe authentication protocol [Low95] solve this problem by authenticating the communicating parties, but require a logically centralised authentication server to maintain shared secrets (Kerberos) or keys (Needham-Schroeder-Lowe). Such centralisation would violate the principle of full distribution in the decentralised data centre OS.

Since data centres are operated by a single authority, some attacks – such as compromises of switch firmware – although possible, are rather unlikely. In addition, most data centre operators encrypt even internal data centre traffic [EFF13], which helps defend against snooping. For the purpose of my model, I consider such threats to be out of scope and assume an uncompromised TCB and interconnect. The secure delegation of capabilities across untrusted commodity interconnects is, however, an interesting area for future work (see §9.1.3).

Resource management summary. Together, identifier and handle capabilities enforce mandatory access control. An identifier capability must be resolved to one or more local, context-sensitive handle capabilities before an application can interact with a logical object. While identifier capabilities refer to logical objects, handle capabilities help manage physical objects and effect I/O on them. However, handle capabilities are only valid within their task context, while identifier capabilities are valid within any task that has access to their namespace.

This split also controls capabilities' provenance: in my model, the resolution of identifier capabilities respects namespaces, which restricts the set of tasks that can use them, but does not restrict their communication. Handle capabilities, by contrast, can only be transferred to other tasks via delegation through the TCB, but can be refined, and can be delegated to any task that the owner task can name.

Related approaches. As with identifier capabilities, my notion of handle capabilities is similar to capability concepts in several prior systems (Table 3.1).

Traditional capability schemes typically relied 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. By contrast, Mach [ABB+86], EROS [SSF99], seL4 [KEH+09], and Barrelfish [BBD+09] rely on memory protection hardware to separate capability space and data space.

Accent [RR81] and Mach [ABB+86] used capabilities to control access to "ports" (IPC endpoints), between which message-based IPC could communicate capabilities. Capabilities granted access to ports and were location-independent, but could not be refined. In Mach, the 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 Barrelfish OS uses a distributed capability scheme based on the segregated seL4 capabilities [SEL4RM, §2.2, §2.5], which hierarchically refines capabilities [Nev12, pp. 29–30]. When capabilities are shared, they must be serialised and sent via a secure channel [BFTN0, p. 10]. Since Barrelfish does not have a transparent object abstraction, capabilities are not location-independent.

3.6 Persistent storage

Data centre storage systems are often more reminiscent of the flat data stores in early operating systems than of today's complex hierarchical file systems such as 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 only supports appending writes on existing files, rather than random write access [SKR+10, §3.A]. If needed, directory services are implemented via a meta-data controller, while the actual data are stored as flat blocks (e.g. 64 MB chunks in GFS and HDFS, 8 MB "tracts" in FDS).

The decentralised data centre OS model therefore has the opportunity to simplify the storage subsystem. Simplifying the storage stack can make it less error-prone and easier to scale, and may also improve I/O performance if the abstractions are a better match for data centre storage.

The storage interface for the decentralised, distributed data centre OS model is therefore a minimal one: uniquely identified objects are stored in a flat persistent object store. This approach externalises the implementation of hierarchical (or other) storage abstractions to higher-level distributed applications, rather than layering them atop an already-hierarchical baseline storage abstraction. For example, both BigTable-style distributed maps and a GFS-style distributed file system can be implemented on top of a simple object store. Protection and access control across

both of these higher-level abstractions can then rely on the uniform capability abstractions provided by the data centre OS (§3.4–3.5).

This storage abstraction also allows common optimisations such as caching to be easily expressed. Logical objects in the store may be replicated using multiple physical objects, and the replicas (≡ physical objects) may have different persistence levels. For example, some physical objects may be cached copies in volatile memory, while others are on durable storage, and finally, some might be memory-mapped with a persistent backing copy. The translucent meta-data of the physical objects' respective handle capabilities exposes the information as to whether a physical object is in the durable store or not.

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 introspecting the storage location of physical objects in my model.

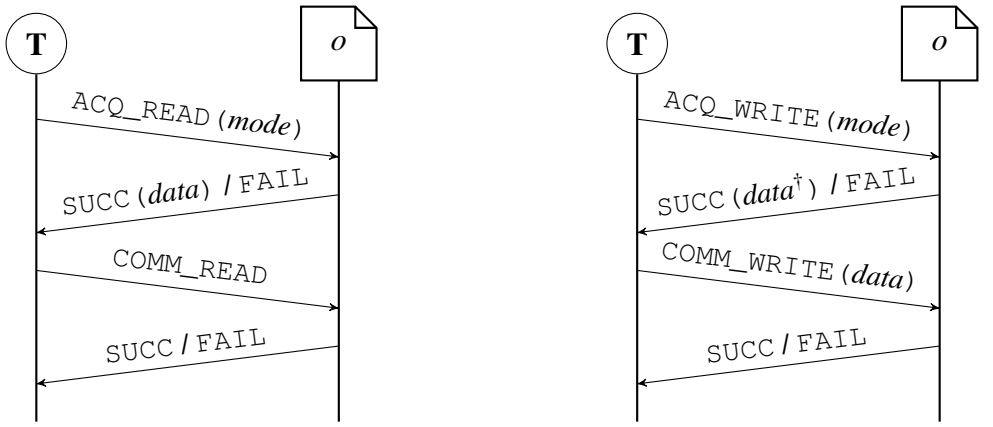
Many later single address space OSes, such as Grasshopper [DBF+94], Opal [CLF+94], and Mungi [Hei98], also supported persistent segments or objects. In Opal, objects are also held in a pervasive, flat store [CLF+94, §4.4]. More recently, distributed data processing systems such as CIEL [Mur11, §4.5], and high-performance network attached storage (NAS) systems have adopted similar object-centric storage models [CWM+14, §4].

3.7 Concurrent access

Many data centre applications are I/O-intensive and concurrent: they communicate with other applications, process large amounts of streaming data from the network, or process static data from durable storage in parallel using many tasks. Oftentimes, different applications are combined, and may share data with each other.

The data centre OS model therefore needs to provide a way for workloads to access remote physical objects, and a way to share physical objects between tasks. Depending on the physical object and the applications' consistency requirements, certain concurrent accesses might be acceptable. Since application-level consistency guarantees vary substantially across different systems, my model must not dictate a specific concurrent access semantic, however, but instead allows applications to define their own policies.

The decentralised, distributed data centre OS model therefore uses transaction-like **I/O requests**, which are a flexible and low-overhead way of tracking concurrent access and notifying applications when it occurs. 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.*



(a) Read request.

(† optional, only some object types.)

(b) Write request.

Figure 3.4: Sequence diagrams of (a) read and (b) write requests: the caller first acquires (ACQ), then commits (COMM). The arrows are calls from the task (**T**) to the local kernel on the machine maintaining the physical object *o*, and their returns.

reading and processing data, or writing them to the buffer). An I/O request is either a *read request* or a *write request*.

An I/O request begins by acquiring a request: the application specifies the desired operation and concurrent access semantics for the target physical object. If successful, the request is in the *acquired* stage, during which the application performs I/O. Once the application finalises its I/O, it commits the request and asks the OS to validate whether the concurrent access semantics did indeed hold for its duration. If successful, the request is *committed*; otherwise, it is aborted.

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

Figure 3.4 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 physical 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 receive a working buffer on *acquire*. Depending on the type of the target physical 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 physical 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, changes have already been applied at the commit point. In this case, the data may potentially have been corrupted by the invalid request, and the OS must either invalidate the physical object for all tasks that access it, or the application must implement its own recovery procedure.

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 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 by having them race to acquire a shared object for exclusive access, setting the value to "locked" on success.

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].

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.

3.8 Summary

This chapter introduced the decentralised, distributed data centre OS model, a clean-slate reference model for uniform, efficient, and secure data centre resource naming and management.

After defining key terms and concepts (§3.1), I distilled the shortcomings of current infrastructure systems observed in Section 2.2.3 into a set of requirements for my model (§3.2).

I then explained how the model's core abstraction of distributed objects (§3.3) expresses distributed applications, and how resource naming (§3.4) and management (§3.5) are realised via distributed capabilities.

My model achieves **efficiency** by making its resource handles *translucent*: applications can introspect on the handles to physical objects to choose between different interchangeable physical objects that correspond to the same logical object. Such choice enables performance optimisations that would not have been possible in a transparent distributed system.

Moreover, the model implements uniform and pervasive capability-based **security and access control**, satisfying the three goals of isolation, deniability, and auditability:

1. *Isolation* is ensured by unforgeability of capabilities: identifier capabilities are unguessable as they are random or based on cryptographic hashes, and handle capabilities are unforgeable as they are segregated (i.e. have a kernel counter-part). Since identifier capabilities are namespaced and handle capabilities can only be delegated via the TCB, leaks are containable if they do happen.

2. Having to resolve identifier capabilities grants *deniability* of resource existence: when resolving an identifier capability, the name service can deny the existence of physical objects corresponding to a logical object identifier, and this is indistinguishable from the case in which no physical objects in fact exist.

3. *Auditability* is guaranteed because the TCB resolves identifier capabilities and creates all handle capabilities. When resolving an identifier capability, the name service can log the operation, and delegation of a handle capability must likewise pass through the TCB and can be recorded.

Finally, I explained how objects are stored persistently in a flat object store (§3.6), and how my model uses optimistically concurrent, transaction-like I/O requests to flexibly express concurrent access for I/O to objects (§3.7).

In the next chapter, I present DIOS, a prototype implementation of my model as an extension to Linux.

Chapter 4

DIOS: a distributed operating system for data centres

DIOS is a prototype implementation of the data centre OS model outlined in the previous chapter, developed as an extension module for Linux. This chapter describes the key abstractions in DIOS, how they implement the model's concepts, how they interact, and how they support applications.

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

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

Names (§4.3) identify logical objects. They are freely communicable, flat pieces of binary data that realise the model's concept of identifier capabilities, used to locate physical objects.

Groups (§4.4) implement namespaces, limiting the resolution scope of DIOS names. Groups are objects themselves; they have names and can be accessed via references.

References (§4.5) constitute context-dependent physical object handle capabilities for specific physical objects and expose informational meta-data to user-space, thus allowing applications to introspect them. They can be delegated, but only with kernel mediation.

After introducing these concepts, Section 4.6 gives an overview of the initial DIOS system call API. Section 4.7 explains how DIOS implements concurrent access via I/O requests and presents examples of how user-space libraries and applications use them.

As a distributed operating system, DIOS relies on coordination across machines. In Section 4.8, I introduce the DIOS Coordination Protocol (DCP) and discuss its implementation.

Finally, I discuss the scalability of the DIOS abstractions in Section 4.9, and describe how DIOS integrates with the Linux kernel in Section 4.10. I also discuss how incremental migration to DIOS is enabled by its support for combining legacy Linux and new data centre OS abstractions.

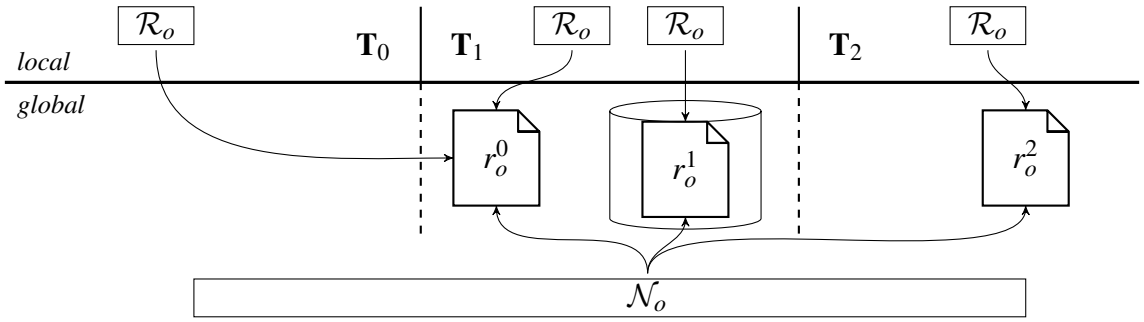


Figure 4.1: Example of references ($\mathcal{R}_o$) to logical object o's in-memory and durable (cylinder) physical objects (r_o^i), described by a single name for the logical object ($\mathcal{N}_o$), and distributed over three tasks ($\mathbf{T}_0$–$\mathbf{T}_2$).

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*, *groups*, *references*, 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.

The DIOS object notion follows the decentralised data centre OS object model (§3.3). *Logical objects* can be replicated, giving rise to multiple *physical objects* (although not all object types are replicable). A physical object is always managed as a unit: it is created, replicated and destroyed atomically.

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 objects are depicted as sheets of paper, while all physical objects pointed to by the same name make up the logical object.

Name: a globally unique identifier for a logical object (i.e., one or more physical objects). Names are fixed-width, flat binary values and resolve to physical objects. The model requires that all physical objects for a logical object are interchangeable, i.e. they exhibit identical application semantics.[1]

A name is an *identifier capability*: it is globally unique, unforgeable except by brute-force, and can be resolved by any task it shares a group membership with. 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).

[1]This notion of the physical objects being interchangeable 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.

In Figure 4.1, a name is depicted by $\mathcal{N}_o$ and maps to three physical objects, r_o^0–r_o^2, across tasks and machines.

Reference: a locally scoped, context-sensitive handle capability for a specific physical object. Many references may refer to the same physical object, potentially from different task contexts (see Figure 4.1, which depicts references as boxes in local scopes, with arrows pointing to the physical objects they refer to).

References contain both *internal* information visible only to the OS and *public* attributes that are exposed to applications and which implement translucency. All DIOS 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 physical object referred to.

Group: a special type of object that is used to restrict name resolution scope. A task can only resolve names in those groups that it can access. Group memberships are selectively inherited from the parent task to child tasks spawned by it.

Task: a special type of object that represents a running or suspended DIOS process. Each task has its own, private set of references, and is a member of a set of groups which is fixed at creation time. Tasks run in their own address space, although they may share memory mappings with other tasks. My model is compatible with multiple tasks sharing an address space, but the current DIOS implementation only supports tasks in separate virtual address spaces, since it tracks references and group memberships per-process, not per-thread.

In the following sections, I discuss these abstractions and their realisation in DIOS. In keeping with the notation used in Figure 4.1, I represent DIOS concepts as follows:

$\mathcal{O}_i$	denotes the structure describing a physical object i.
$\mathcal{N}_o$	denotes the name of a logical object o.
$\mathcal{R}_i$	denotes a reference to a physical object i.
$\mathbf{T}_t$	denotes a task t of type $\mathbf{T}$.

As before, figures illustrating DIOS abstractions represent physical blob objects as documents, physical streaming objects as pipes, and task objects as circles.

4.2 Objects

DIOS manages distributed objects over many machines, which entails creating and managing the physical objects, but also making them and their meta-data available across many shared-nothing data centre machines. In this section, I describe how objects are created, managed, and

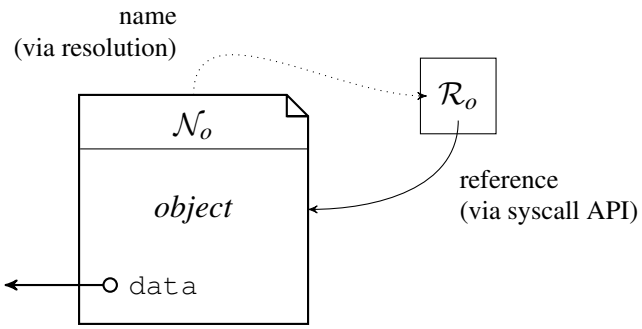


Figure 4.2: A physical DIOS object and its relations to names and references.

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

Table 4.1: Object types currently supported in DIOS.

destroyed, and how DIOS represents them to applications. My initial discussion focuses primarily on high-level concepts and omits the details of cross-machine coordination; Section 4.8 describes the communication protocol for operations across machines.

A physical object is represented by the structure shown in Figure 4.2. Each physical object has a name, a type, and type-specific data. The type-specific 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 a physical object structure when the physical object is first created, and it persists in memory until the physical object is deleted or the holding node crashes. For durable physical objects, the structure is serialised to the object store alongside the object data, so that it can later be restored.

Granularity. Physical DIOS data objects can hold any amount of data, but per-object overheads of around 100 bytes make them inefficient at fine granularities 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 a request response of several kilobytes. In the web server example, each 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 types supported in the current DIOS implementation are 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 logical streaming object type, for example, would be a key addition required for some applications (e.g. a distributed file system).

Life cycle. Every physical object has a *deletion mode*, which governs when it is considered ready for destruction. There are three such types:

- *Immortal* objects persist even if they do not have any references pointing to them and if their name is not resolvable. Examples of immortal objects include a task's self-object (which is destroyed implicitly with the task) and the special console object (which lasts until machine shutdown).

- *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. Only when the reference count reaches zero, the physical object is deleted. Most DIOS objects are reference-counted, and the most common cause of physical object deletion is the exit of the final task holding a reference to a physical object.

- *Immediate deletion* amounts to destruction as soon as the delete operation is invoked on *any* deletion-privileged reference to the physical object. All other references to the physical object are invalidated, and the reference count forced to zero. The actual object removal may occur asynchronously, but once deleted, the object cannot be resurrected. A durable on-disk object with one-off deletion might have a deletion-privileged reference to it held by the distributed file system.

In both reference-counted and one-off deletion, the physical object is first "zombified" – that is, its I/O operations are atomically disabled, so that concurrent access fails safely – and then destroyed. Other physical objects for the same logical object may continue to exist, since they have their own physical object meta-data (and possibly even a different deletion mode).

Persistence. Objects may be *persistent*, in which case they are stored on durable storage and survive DIOS restarts. For such physical 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 physical 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 physical objects stored on local persistent storage.[2] This enables bootstrapping references to these objects in one of two ways:

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

2. A special, privileged task (akin to `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 (§2.1.1.1) is well-suited to the role of this special task, since it is part of the model's TCB already.

The next section explains DIOS names in more detail.

4.3 Names

DIOS assigns every logical object a globally unique identifier, its *name* ($\mathcal{N}_o$ for a logical object o). Multiple physical objects may exist for the logical object, but must maintain the invariant that they are interchangeable without breaking applications' expected semantics. If this is not the case, the physical objects do not correspond to the same logical object.

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 data centre or even outside it.

Names must be unforgeable, since they are *resolution capabilities*, permitting – in combination with an appropriate group membership – translation of names into sets of references (see §4.3.3). Unforgeability is ensured by the statistical properties of their cryptographic generation.

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

Scoping. Without namespaces, leaking a DIOS 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

[2] In the current implementation, 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.

centralised namespace repository would contradict the model's goal of a fully-distributed architecture at the OS level. Hence, DIOS 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 logical object, denoted by $\mathcal{N}_o$. It is returned 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 of which the logical object is a member, each denoted by $\mathcal{N}_o^g$ for group g. Each internal name maps to the same physical object. Internal names are never made available to applications, but do not constitute secret information: even if leaked by accident, they are useless to applications.

The internal names are used as identifiers within DIOS, while the external names are merely identifier capabilities that, when combined with another capability (*viz.* a group membership), yield the identifier under which the OS internally tracks the logical object. This is a split capability approach [KRB+03]; I describe this in more detail in Section 4.4.

In the following, I describe how names are generated (§4.3.1), how they are stored (§4.3.2), and how they are resolved into references (§4.3.3).

4.3.1 Generation

DIOS, in accordance with my model supports three ways of generating names (§3.4): *deterministic* name generation allows stable sequences of names to be generated when replaying a sequence of tasks; *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 a logical 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:

$$\textit{internal name}_g = \mathcal{H}(\textit{external name} \parallel \textit{external name of } g).$$

This way of generating an internal name ensures that even if an internal name is ever leaked, it cannot be used to resolve physical objects. This is because a resolution attempt would combine it with a group name, yielding *another*, invalid, internal name.

```
1  int create_shmem_fifo(dios_name_t* name, uint64_t host_id) {
2    dios_ref_t* ref = NULL;
3    int result = -1;
4    dios_flags_t flags = D_CREATE_NAME_DETERMINISTIC;
5
6    /* non-zero host ID indicates a remote object creation */
7    if (host_id != 0)
8      flags |= D_CREATE_REMOTE;
9
10   result = dios_create(flags, D_OBJ_SHMEM_STREAM, NULL, name, &ref, host_id);
11   if (result < 0)
12     perror("failed to create SHMEM_STREAM object");
13   return 0;
14 }
```

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`.

Deterministic names. To generate names deterministically, DIOS applies the SHA-256 hash function to the name of the generating task object and an additional, deterministically generated identifier. This 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 name generation in CIEL [MSS+11, §5.1]).[3]

Random names. If objects deterministic naming is not required or possible, 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. Random names are useful for temporary objects, or in tasks that do not need to support deterministic replay fault tolerance. The name for a randomly-named logical object is only known once it exists; unlike a deterministic name, it cannot be predicted.

Special names. DIOS tasks have access to default resources such as a console or a timer. Such objects have well-known names composed of a 16-bit identifier in the least significant bits of an otherwise all-zero name. Unlike other DIOS name types, special resource names are well-known (i.e. forgeable) and refer to specific objects. Notably, the physical objects for a logical object with a special name depend on the context in which it is used: resolving the console name does not return all consoles in the entire data centre.

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 the application, and may be on the stack or within another physical DIOS object.

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

4.3.2 Storage

DIOS keeps track of names and the live physical objects they identify by storing mappings between names and physical objects in a *name table*. Each machine in the data centre has a system-wide name table, held within the TCB (i.e. the kernel).

A **name table** maps an internal name $\mathcal{N}_o^g$ for a logical object o in group g to a set of object structures, $\{\mathcal{O}_0, ..., \mathcal{O}_n\}$, for a set of n known physical objects for o. Each machine's name table holds entries for all local physical objects, and is updated on object creation and deletion. Additional physical objects for the same name may exist in other machines' name tables, and the local name table may cache mappings for remote physical objects to reduce resolution latency, at the cost of additional coordination on object deletion.

The name table maps names to object structures, rather than to references, since references are context-dependent handles. References are generated dynamically during name resolution using the object structure information, and are specific to the resolving task and its location.

Implementation. An alternative implementation would maintain a per-task name table, as opposed to a shared per-machine one. This appears to increase scalability, but does not in fact improve it: per-task name tables require either *(i)* that shared names are proactively inserted into all per-task name tables (reducing the scalability of object creation), or *(ii)* for name resolution to query *all* per-task name tables (reducing resolution scalability), or *(iii)* that names are no longer global identifiers and only explicitly shared (which goes counter to the model). Hence, DIOS uses a single per-machine name table. This does not have any security implications, because the name table is held entirely in the TCB and never directly exposed to applications.

Bootstrap. On bootup, the kernel populates the name table with well-known special names and their corresponding physical objects, as well as the names for all physical objects stored on durable storage on the node. Other name mappings are added incrementally at system runtime.

Applications bootstrap their access to physical 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 (§3.4). 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 correspond to multiple physical objects within the data centre. A name resolution request returns references for a set of existing physical objects up to a maximum number specified.

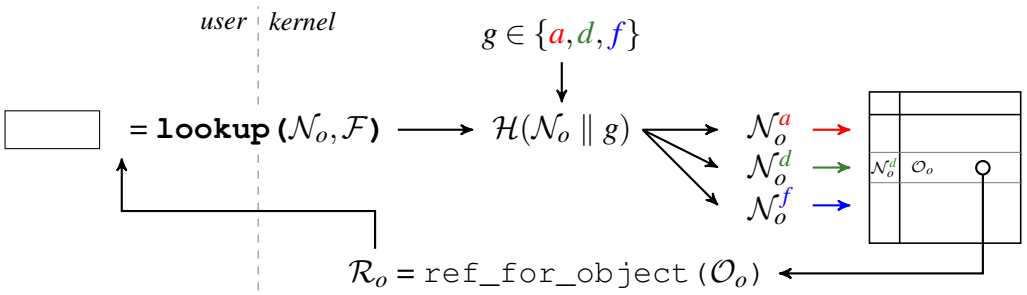


Figure 4.3: To resolve external name $\mathcal{N}_o$ for o, it is hashed with the task's groups a, d, and f to generate internal names $\mathcal{N}_o^a$, $\mathcal{N}_o^d$, and $\mathcal{N}_o^f$; the physical object is located in the name table under its internal name $\mathcal{N}_o^d$, an appropriate reference is generated based on the object structure $\mathcal{O}_o$ and returned. Note that all groups are queried here, rather than a specific one.

Therefore, resolving a DIOS name amounts to translating an external name into either an empty set, or a set $\{\mathcal{R}_0, ..., \mathcal{R}_k\}$ of k newly generated references corresponding to k physical objects. Although DIOS endeavours to return references for *all* existing replicas (up to an upper limit specified), this is not guaranteed: only references for *reachable* physical objects are returned. *Reachability* is subject to two conditions:

1. a physical object identified by one of the internal names generated from combining external name $\mathcal{N}_o$ with one of **T**'s groups must exist in a name table on some machine;
2. for a remote object, the machine that holds it must be alive (i.e. able to generate a response), and neither the name resolution request, nor the response may be lost.

In terms of the CAP theorem [Bre00], this is an "AP" system, i.e. name resolution preserves availability and partition tolerance, at the expense of consistency, since unreachable objects are not returned. This makes sense for a decentralised data centre OS, which must remain available and partition-tolerant at all times.

The name resolution is the responsibility of the **name service**, which is part of the TCB in the decentralised data centre OS model. DIOS implements the name service as a local kernel service that communicates with other machine kernels, but it could also be implemented as a user-space server (although remaining part of the TCB).

When resolving a name, the DIOS name service always consults the local name table first, but may proceed to consult other machines' name tables as well. Access to remote name tables happens via the DIOS Coordination Protocol (see §4.8). The resolution for an external name $\mathcal{N}_o$ proceeds as follows (Figure 4.3):

1. DIOS generates the corresponding internal name, $\mathcal{N}_o^g$ for the specified group g, or for each group that the calling task is a member of, by computing a hash of $\mathcal{N}_o$ and g (§4.3.1).
2. It uses each internal name $\mathcal{N}_o^g$ to index into the local name table, returning any matching local physical objects.

```
dios_name_t* name = task_info->input_names[0];
dios_ref_t refs[MAX_REFS];
int ref_count = MAX_REFS;

if (dios_lookup(name, &refs, &ref_count, D_NONE) <= 0) {
  /* failed */
  printf("error: name lookup failed; got %ld references\n", ref_count);
} else {
  /* success */
  printf("got %ld references for the first input!\n", ref_count);
}
```

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.

3. If no match is found, or if insufficiently many entries are found and the resolution is not explicitly local-only, further physical objects are sought on remote machines. If so,
   (a) a lookup message containing the internal names of interest is broadcast to all live machines (via the DCP, §4.8);
   (b) any machine that holds physical objects for the logical object named by one of the internal names sends a unicast response to the enquiring machine kernel.
4. For any matching objects found, references are generated and returned. If no match is found, an empty reference set is returned.

This procedure does not specify any ordering between physical objects except for giving priority to local objects. Instead, the first responses to arrive are returned; if fewer physical objects than the requested number are found before timeout, references to those are returned. For priority ordering between remote objects, DIOS could 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` physical objects for 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 physical objects that exist in its groups.

Name resolution should therefore only be available to logical "owners" of objects.[4] Reference delegation, by contrast, supports the principle of least privilege. In Section 4.5.3, I explain how a task creates a more restricted version of a reference by delegating it either to itself or to another task. First, however, I discuss groups, which delineate namespaces in DIOS.

[4] An "owner" of an object is a task that is trusted with full access to it (of which there might be several); it is not necessarily the task that created the object in the first place.

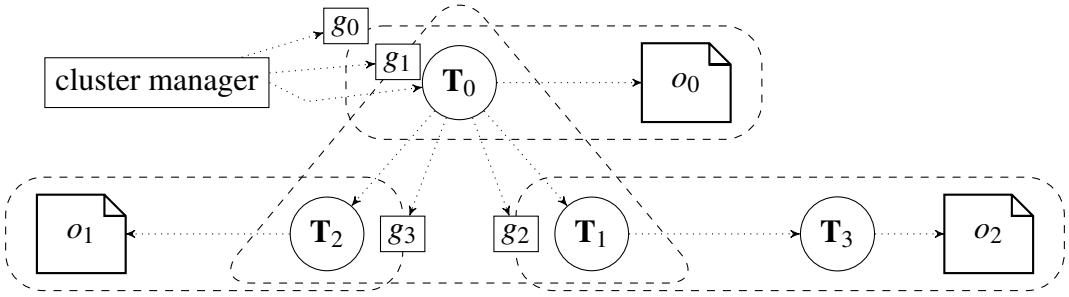


Figure 4.4: DIOS groups form a non-transitive hierarchy: each group is represented by a group object (dashed boxes with square labels), which itself is a member of one or more groups in which it is created. Dotted arrows indicate object creation. Here, the cluster manager initially creates g_0, g_1, and $\mathbf{T}_0$, which creates other tasks, groups, and blob objects.

4.4 Groups

Identifier namespaces in DIOS are defined by *groups*. Groups are a specialised instantiation of the more general concept of split capabilities [KRB+03], as I outlined in Section 3.4.

Each logical object in DIOS is a member of one or more groups. Each task – which is also an object, and thus a member of one or more groups – can resolve names in the groups it is a member of. In split capability terms, the name being resolved is the "handle" part of the split capability, while the group membership forms the "key" part [KRB+03, §2].

Representation. Groups are themselves represented as DIOS objects. Accordingly a group also has a unique DIOS name.[5] However, a group object contains no data other than a read-only copy of its name (for introspection), and cannot be written to.

Group objects are used in two situations:

1. When a task creates a new object, it passes references to one or more group objects to indicate the new object's group membership. Any group object accessible to the task can be used.

2. When a task spawns a new task, a set of references to group objects that specify the child task's group memberships can be passed. Only group objects that represent groups of which the parent task is a member, or group objects for groups that the parent task created, may be passed.

Each group object is itself created within one or more groups. The name of the new group object can therefore only be resolved by tasks which are members of an *owning* group (one in which the group object was originally created). Consequently, groups form a non-transitive hierarchy: each group has at least one "parent" group, but need not be a member of that parent group's

[5] Group names are DIOS names, rather than human-readable group identifiers to ensure uniqueness in the absence of centralised name management.

Task relationship to g	Resolve names in g	Create objects in g	Grant membership of g to child task
Member	✓	✓	✓
Created g	✗	✓	✓
Resolved g's name	✗	✓	✗†

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)

own parent group or groups (see Figure 4.4). At the root of the hierarchy, a special "null" group that implicitly contains all groups exists. This group is only accessible to the cluster manager, and is used for bootstrapping.

Bootstrap. Since group membership can only be inherited from parent to child task, DIOS ultimately needs a way of bootstrapping group memberships for tasks. Specifically, the original group memberships for the root of a tree of tasks must come from within the TCB. In DIOS, the cluster manager injects initial group memberships when starting a new job. Tasks within the job can delegate membership of these initial groups (which might, for example, contain the input data objects for the job), or create new groups internal to the job (for new objects) and create subtasks that are members of these. Task group membership, however, is fixed at task creation time and cannot be modified dynamically.

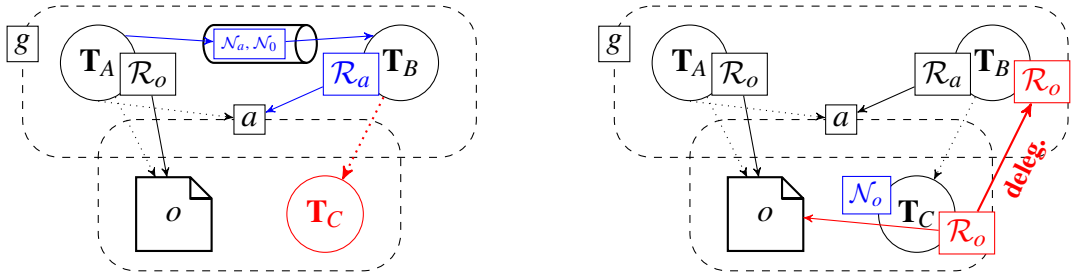
When a task starts, it receives a set of default references, including ones to group objects for the groups it is a member of. When new task is created without any inherited group memberships, DIOS assigns it a new, otherwise empty group. In this situation, the task is limited to resolving names in this single, initially empty, group.

Revocation. It is not possible to revoke group membership of a logical object or a task in DIOS. This comes because revocation would require either a centralised repository of group memberships, or global synchronisation across all machines that hold physical objects for the logical object in order to remove internal name mappings from their name tables.[6] Accordingly, 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 original object (although the deletion is eventually consistent).

Group object restrictions. Compared to normal DIOS objects, group objects are somewhat restricted for security reasons. Permissible uses of a group object depend on how the task acquired access to it and on its own group memberships.

In general, references to group objects – independent of how they were obtained – *cannot* be delegated to other tasks. Supporting delegation of group object references would allow a task to

[6] An eventually consistent approach could be used, but would not guarantee that the membership has actually been revoked when the operation returns.



(a) $\mathbf{T}_B$ receives names for o and a from $\mathbf{T}_A$ (blue), resolves $\mathcal{N}_a$ to a group object reference, and creates a child task, $\mathbf{T}_C$ in a (red).

(b) $\mathbf{T}_C$ receives name $\mathcal{N}_o$ from $\mathbf{T}_B$ (blue), resolves it to a reference to o, and delegates the reference back to its parent, $\mathbf{T}_B$ (red).

Figure 4.5: Illustration of the "access-via-child" vulnerability. By preventing tasks from granting membership of a group whose name they resolved, the red operations are prohibited and $\mathbf{T}_B$ cannot gain access to physical objects in group a via its child, $\mathbf{T}_C$.

grant access to its groups to *any* task that it can communicate with, risking accidental exposure. Table 4.2 explains what other operations are permitted for a task with different relationships to a group g.

First, a task can *resolve names* to physical objects only in those groups that it is a member of. A task's group memberships are immutable and tracked in the kernel. Whenever the task requests a name resolution, it specifies a group object reference to use; if this reference does not refer to a group that the task is a member of, name resolutions fails.

Second, a task can *create objects* using a reference to any group object it has access to. 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. This is useful because it allows for temporary data exchange groups to be established between tasks. For example, a MapReduce controller task might create a group g for all outputs, and communicate its name to the reduce tasks. The reduce tasks can now each create their output objects inside that group without being able to resolve the output object names for any *other* task. There are no security implications of this design decision, because the controller and reduce tasks must still share at least one group membership: the reduce tasks can only resolve g's name if g is created inside a group that they are members of.

Finally, a task can *grant membership* of both the groups it is a member of, and of those that it created, to a child task at creation time. Moreover, a task *cannot* grant membership of a group for which it merely obtained a reference via name resolution. This restriction is necessary to defend against an "access-via-child" vulnerability. Consider the following example (Figure 4.5):

1. Two tasks, $\mathbf{T}_A$ and $\mathbf{T}_B$, are both members of group g.
2. $\mathbf{T}_A$ creates a new group, a, in g and hands its name to $\mathbf{T}_B$. $\mathbf{T}_B$ can now resolve the name into a reference to the group object for a.
3. $\mathbf{T}_B$ is not a member of a and hence cannot resolve names in a.

4. Assume that $\mathbf{T}_B$ was to be allowed to grant membership of a to a child task $\mathbf{T}_C$ that it creates.

5. $\mathbf{T}_C$ can now resolve a name in a that was handed to it by $\mathbf{T}_B$ (since $\mathbf{T}_C$ is a member of a) and subsequently delegate a resulting reference back to $\mathbf{T}_B$.

This effectively grants $\mathbf{T}_B$ the same power as $\mathbf{T}_C$. Hence, $\mathbf{T}_B$'s abilities are equivalent to those of a member of a via collusion with its own child, $\mathbf{T}_C$. Restricting child task membership to the parent task's group memberships and those groups that it created solves this problem.

Groups limit the scope of name resolution, but DIOS tasks have another way of restricting access to objects, implemented via references. In the next section, I describe references in detail.

4.5 References

While names are globally meaningful identifiers for objects, references are *handles* that are only valid in a particular context. Unlike names, they cannot be stored as data or communicated directly, but they allow interaction with the object they describe (§3.5).

The closest functional analogy for references in POSIX terms are file descriptors (FDs). Both FDs and references are handles for objects and associated kernel state, but FDs can only refer to local not remote objects. Moreover, FDs merely serve as plain identifiers, while references carry additional information that makes objects translucent to applications.

Mach's [ABB+86] ports have some similarities with DIOS references: they are location-transparent handles for local or remote objects, owned by a Mach "task". However, unlike DIOS references, they are fully transparent rather than translucent, and only support streaming message-passing semantics with an active task at the far end handling incoming messages, while DIOS references can refer to passive objects.

A DIOS reference is a segregated capability that consists of two parts: *(i)* privileged information only available within the TCB, and *(ii)* public information that is exposed to the task. This division is required to track protected, privileged state only visible to the TCB (such as the logical object's internal names), while maintaining translucency. Figure 4.6 illustrates this structure with reference to an example.

The **privileged 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), and a pointer to the object structure. In DIOS, 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 contains read-only meta-data about the object. This information includes properties that an application might

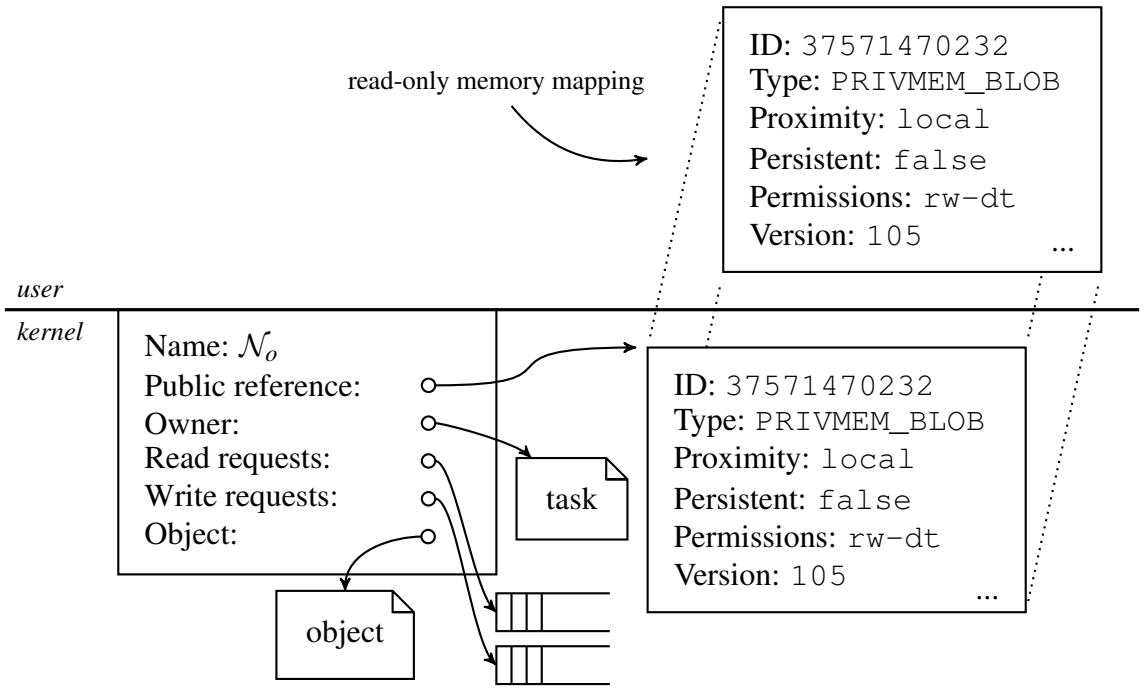


Figure 4.6: Structure of a DIOS reference illustrated by the example of a physical in-memory data object in, e.g. the web service back-end in Figure 3.3.

consider when choosing between references to alternative physical objects for the same logical object (e.g. proximity and persistence), or when checking the consistency of an I/O request on the physical object 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 update the public information without writing to an application-allocated buffer.

4.5.1 Attributes

DIOS maps the public part of a reference ($\mathcal{R}_o$ for object o) into the task's virtual address space (Figure 4.6) and returns a pointer to this structure from the reference-creating system call. Thus, the public attributes are visible to the application task holding the reference, but cannot be modified by it.

References' public attributes enable applications to implement their own policies in the distributed system while still using DIOS abstractions. Consider an example: a MapReduce application's map tasks may run 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 shows an example implementation of a `pick_input_ref()` function for the MapReduce application: here, the application policy is to choose in-memory inputs over on-disk inputs regardless of their location, but it prefers more proximate inputs when they are otherwise equal.

```
dios_ref_t* pick_input_ref(dios_ref_t** refs, int count) {
  dios_ref_t* best_ref = refs[0]; /* default: first reference */

  for (int i = 1; i < count; ++i) {
    /* always prefer in-memory over on-disk objects */
    if (best_ref->type == DURABLE_BLOB && refs[i]->type == SHMEM_BLOB)
      best_ref = refs[i];
    /* prefer more proximate objects */
    else if (closer_than(refs[i]->proximity, best_ref->proximity))
      best_ref = refs[i];
  }

  return best_ref;
}
```

Listing 4.3: Example of using DIOS reference attributes to choose the best input for a MapReduce map task: here, in-memory objects are always preferred over on-disk objects, no matter if local or remote; and more proximate objects are preferred at equal persistence.

In practice, the choice between different references is typically made in libraries rather than in application code written by the infrastructure user. The DIOS standard library, for example, offers several default reference pickers, which can choose the most proximate reference, or the one with the fewest overlapping fault domains.

Since a reference's public attributes are read-only, the task cannot directly modify them. The only way to mutate reference attributes is to *delegate* the reference, which creates a new reference with modified attributes. A task can either delegate the new reference back to itself, or delegate it to another task. Some reference attributes, however, are inherent features of the underlying object (e.g. its type), and cannot be changed on delegation. Others are mutable, but cannot be modified arbitrarily. One such example is the set of permissions on a reference: a task may remove permissions when delegating a reference, but cannot add them.

Next, I describe how tasks first acquire references, followed by how they can be delegated.

4.5.2 Acquisition

A task may acquire a reference in one of three ways. Each of these ways always creates a *fresh* reference, even if the task already holds a reference to the same object. Applications 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 physical object.

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 ac-

Attribute	On delegation... automatically adapted	mutable by user	copy on mutation
Type	✗	✗	–
Group	✗	✗	–
Read version (v_r)	✗	✗	–
Write version (v_w)	✗	✗	–
Active readers (a_r)	✓	✗	–
Active writers (a_w)	✓	✗	–
R/W buffer size	✓	✗	–
Permissions	✗	✓, restrict only	✗
Concurrent access mode	✗	✓†	if incompatible
I/O multiplexing mode	✗	✓†	if incompatible
Persistence	✗	✓†	✓
Proximity	✓	✓†	✓

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

quired data to be converted into references to physical objects. I covered name resolution in Section 4.3.3.

Third, the task may receive a reference from another task through *delegation*. This is useful in order to grant selective access to an object. I describe reference delegation next.

4.5.3 Delegation

As I explained in Section 4.3.3, resolving an object's name yields a fully privileged reference to it. DIOS tasks also often need to grant *selective* access, i.e. access with restricted permissions, to objects. DIOS therefore supports another, restricted way of sharing an object between tasks even in the absence of the group membership required to resolve its name.

To enable such sharing, references must be able to traverse task boundaries. This process, which involves the DIOS TCB, is called *delegation*. A delegation of $\mathcal{R}_o$ from $\mathbf{T}_A$ to $\mathbf{T}_B$ makes DIOS generate a new reference that refers to the same physical object as $\mathcal{R}_o$. The public and private parts of the new, delegated reference, $\mathcal{R}_d$, are identical to those of $\mathcal{R}_o$, except for transformation specific to $\mathbf{T}_B$'s context.

The transformation is important: since references are handles that carry context-dependent information, a delegated reference cannot merely be copied into the target task's 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 $\mathbf{T}_B$ 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 to change these attributes. In some cases, changing an attribute has the side-effect of creating a new physical object 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 mutated by the user on delegation. Further extensions of this set, or even support for application-defined attribute labels, are likely to be useful for applications to take full advantage of "translucent" objects.

Notifying the recipient task. When a task delegates a reference to another task, DIOS inserts it into the receiving task's reference table, but the target task itself is initially unaware of the new reference. Since applications identify a reference to object o by its public part, $\mathcal{R}_o$, the recipient task cannot perform any system calls on the new reference until it has been informed about its public part's location.

However, each DIOS task has access to a special default reference to its own task object, the "self-reference". This reference acts as a stream of event notifications, which can supply pointers to newly delegated references. Tasks can choose to ignore these events, or have to poll the self-reference for delegated references.[7]

4.6 System call API

The DIOS system call API defines how applications interact with the local TCB. 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 API, which consists of thirteen system calls that cover I/O, object, and task management. These thirteen calls are sufficient to implement key data centre applications efficiently and with strong compartmentalisation guarantees.

The DIOS 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 4.10.3.

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

Blocking and synchrony. Handling a DIOS system call can require communication across the data centre interconnect, and can take a long time compared to a local system call (on the

[7] A callback notification mechanism akin to conventional Unix signals, where a task registers a callback to be invoked on incoming reference delegation, would avoid the need to poll. However, it would require another abstraction to be added to DIOS.

System call	§	Description
$\langle \mathcal{N}_o, \mathcal{R}_o \rangle$ **dios_create** $(S_G, \mathcal{P}_{args}, \texttt{host})$	B.1.1	Creates a new logical object o in the groups in S_G with type-specific arguments in $\mathcal{P}_{args}$, with a physical object on an optional host; returns logical object name $\mathcal{N}_o$ and reference $\mathcal{R}_o$ to the physical object.
$\{\mathcal{R}_o^0, \ldots, \mathcal{R}_o^k\}$ **dios_lookup** $(\mathcal{N}_o, \mathcal{R}_g)$	B.1.2	Attempts to find all reachable physical objects of the logical object named by $\mathcal{N}_o$ in group g (optional, all task groups if unspecified).
`bool` **dios_copy** $(\mathcal{R}_o, \mathcal{R}_\mathbf{T}, \mathcal{P}_{transform})$	B.1.3	Delegates reference $\mathcal{R}_o$ by copying it into the reference table of task **T** referenced by $\mathcal{R}_\mathbf{T}$ and transforming it as requested in $\mathcal{P}_{transform}$; returns success indication.
`bool` **dios_delete** $(\mathcal{R}_o)$	B.1.4	Removes reference $\mathcal{R}_o$ and invokes deletion handler for physical object o's deletion type; returns indication of success.
$\mathcal{R}_t$ **dios_run** $(\mathcal{R}_{bin}, S_G, \mathcal{P}_{info})$	B.1.5	Runs object referenced by $\mathcal{R}_{bin}$ as member of groups in S_G; returns reference $\mathcal{R}_t$ to running task. $\mathcal{P}_{info}$ is a task-describing structure containing input and output references, arguments, and an optional host.
`bool` **dios_pause** $(\mathcal{R}_\mathbf{T})$	B.1.6	Pauses task **T** referenced by $\mathcal{R}_\mathbf{T}$; returns `true` if now paused.
`bool` **dios_resume** $(\mathcal{R}_\mathbf{T})$	B.1.6	Resumes paused task **T** referenced by $\mathcal{R}_\mathbf{T}$; returns `true` on success.
$\langle \mathcal{P}, \texttt{size} \rangle$ **dios_acquire_read** $(\mathcal{R}_o, \texttt{size}, \texttt{sem})$	B.1.7	Initiates a read I/O request of `size` (optional) with semantics `sem` on the physical object referenced by $\mathcal{R}_o$; returns a read buffer.
`bool` **dios_commit_read** $(\mathcal{R}_o, \langle \mathcal{P}, \texttt{size} \rangle)$	B.1.8	Checks validity of read I/O request of `size` on $\mathcal{R}_o$; returns `true` if valid.
`bool` **dios_release_read** $(\mathcal{R}_o, \langle \mathcal{P}, \texttt{size} \rangle)$	B.1.9	Gives up read buffer for $\mathcal{R}_o$ and returns it to the kernel.
$\langle \mathcal{P}, \texttt{size} \rangle$ **dios_acquire_write** $(\mathcal{R}_o, \texttt{size}, \texttt{sem})$	B.1.10	Initiates write I/O request of `size` with semantics `sem` on the physical object referenced by $\mathcal{R}_o$; returns a write buffer.
`bool` **dios_commit_write** $(\mathcal{R}_o, \langle \mathcal{P}, \texttt{size} \rangle)$	B.1.11	Checks validity of completed write I/O request of `size` on $\mathcal{R}_o$; returns `true` if valid.
`bool` **dios_release_write** $(\mathcal{R}_o, \langle \mathcal{P}, \texttt{size} \rangle)$	B.1.12	Gives up write buffer for $\mathcal{R}_o$ and returns it to the kernel.
$\mathcal{R}_o^i$ **dios_select** $(\{\mathcal{R}_0, \ldots, \mathcal{R}_k\}, \texttt{mode})$	B.1.13	Returns the first reference $\mathcal{R}_i$ out of the k references in $\{\mathcal{R}_0, \ldots, \mathcal{R}_k\}$ to become ready for I/O in `mode` (read/write).

Table 4.4: DIOS system call API. All syscalls are blocking and take a set of call-specific flags, $\mathcal{F}$, in addition to the arguments listed. S_G is shorthand for a set of group object references $\{\mathcal{R}_{g0}, \ldots, \mathcal{R}_{gn}\}$, and $\langle \mathcal{P}, \texttt{size} \rangle$ stands for a buffer of `size` pointed to by $\mathcal{P}$.

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

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

  /* Get input data from ref returned by dios_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: `dios_select(2)` blocks and returns the reference to the first mapper output stream object that has data available.

order of hundreds of microseconds, as opposed to tens). All DIOS I/O is synchronous, and its system calls are therefore blocking. Blocking the calling task while waiting for a remote operation allows other tasks to run in the meantime, and thus increases the effective resource utilisation.

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 `dios_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 that backs pointers passed through the system call API (with the exception of allocating calls like `brk(2)` and `mmap(2)`). DIOS takes a slightly different approach: 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 `dios_acquire_read(2)`/`dios_acquire_write(2)` calls (for I/O buffers). This allows the OS to validate the integrity of the pointers passed to the kernel, and simplifies the implementation of zero-copy I/O requests.

Handling failures Failures of components, machines, or network links are a common occurrence in a data centre. DIOS must therefore accommodate such failures: it must either handle them itself, or indicate to applications that they occurred.

In accordance with my model, DIOS leaves handling failures to applications, since applications' policies for different types of failures can differ substantially. It indicates detectable failures by returning an appropriate error codes from system calls, and it is the application's responsibility to check for such errors (as it is with legacy system calls). For example, if the remote physical object that a reference refers to has become inaccessible due to a network or machine failure, DIOS returns an `EHOSTUNREACH` error, rather than transparently substituting another physical object.

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 are retried a configurable number of times before failing to avoid heavy-handed handling of temporary outages.

A detailed description of each system call's arguments and semantics can be found in Appendix B.1. In the next section, I describe the implementation of DIOS I/O requests.

4.7 I/O requests

In Section 3.7, I described how the decentralised data centre OS model uses *I/O requests* to support concurrent access to local and remote physical objects.

The I/O request model can offer many different concurrent access semantics, but DIOS restricts itself to the relatively simple options shown in Table 4.5. The four levels are exclusive access (`EXCL`), multiple-reader, single-writer (`MRSW`), single-reader, multiple-writer (`SRMW`), and unrestricted (`ANY`).[8]

To enforce these concurrent access semantics, DIOS keeps four atomic counters for each object:

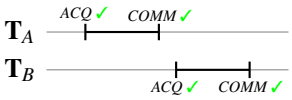
1. the number of active read requests on the object, a_r;
2. the number of active write requests on the object, a_w;
3. the read version number, v_r; and
4. the write version number, v_w.

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 committing a write request are performed under local mutual exclusion. If necessary, the scalability of this step could be further improved using well-known techniques for scalable counters [ELL+07; CKZ13, §3.1] and deferred updates (e.g. lazily reconciling copy-on-write pages).

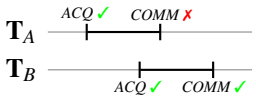
[8] `SRMW` semantics, while somewhat unusual, are useful e.g. for an application that reads and decodes variable-length requests from a buffer and multiplexes them to handlers that write the responses to predefined offsets.

Level	Readers	Writers	Meaning
EXCL	1 (either)		Mutual exclusion (sequential consistency).
MRSW	*	1	Multiple-reader, single-writer (cannot corrupt data).
SRMW	1	*	Single-reader, multiple-writer (cannot read stale data).
ANY	*	*	Any access allowed, no consistency guarantee.

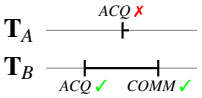
Table 4.5: The four concurrent access semantics supported by I/O requests based on simple read/write versions and reader/writer counters.



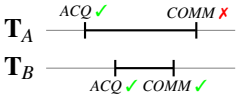
(a) Disjoint requests both succeed.



(b) $\mathbf{T}_A$ fails on commit as $a_r = 2$.



(c) $\mathbf{T}_A$ fails on acquire as $a_r = 1$.



(d) $\mathbf{T}_A$ fails as $\mathbf{T}_B$ has increased v_r.

Figure 4.7: Two read requests with different concurrent access semantics attempt to access object o: $\mathbf{T}_A$ requests exclusive access and $\mathbf{T}_B$ requests MRSW semantics. Depending on how the requests align, $\mathbf{T}_A$ may fail at the acquire or commit stages.

In addition to the high-level indication as to whether an I/O request succeeded or failed, DIOS also exposes these "raw" version counters via reference attributes. This enables custom consistency policies spanning multiple I/O requests to be implemented at the application level.

Examples. To illustrate I/O requests, consider the example of two tasks ($\mathbf{T}_A$ and $\mathbf{T}_B$) both accessing a blob object in shared memory. Assume that both tasks initiate a read-only I/O request, with $\mathbf{T}_A$ specifying `EXCL` as its desired access semantic, while $\mathbf{T}_B$ specifies `MRSW`. Figure 4.7 shows several different ways in which the requests may execute. Since $\mathbf{T}_A$'s request asks for exclusive access semantics, it fails in all situations apart from the disjoint schedule in Figure 4.7a. Note that if $\mathbf{T}_A$'s request was a write, $\mathbf{T}_B$ would fail on acquire in Figure 4.7b.

Listing 4.5 shows another example: the implementation of an update operation in a key-value store (an in-memory blob) under exclusive access. Here, `dios_commit_write(2)` indicates if any concurrent reads or writes have taken place between the `dios_acquire_write(2)` call and the commit. If so, the code re-acquires and keeps trying until a valid write is completed. This implementation starves writers under high contention, but a higher-level lock primitive can be implemented using I/O requests to ensure progress.

Higher-level abstractions. The I/O request system can also be used to implement higher-level concurrent access abstractions, such as synchronisation primitives across tasks, e.g. for a coordination service like Chubby [Bur06] or Zookeeper [HKJ+10].

For example, a global mutex can be implemented via I/O requests on a "lock" object that stores a

```
1  void set_value_excl(dios_ref_t* val_ref, char* new_value, int val_size) {
2    dios_iovec_t* iov = NULL;
3    int ret = 0;
4    /* Set up write with buffer */
5    dios_acquire_write(val_ref, D_ACQUIRE_WRITE_IOV_CREATE, &iov);
6    do {
7      ret = dios_acquire_write(D_ACQUIRE_WRITE_IOV_REUSE,
8                               val_ref, val_size, &iov, D_IO_EXCL);
9      /* Try again if we failed to acquire at EXCL */
10     if (ret < 0)
11       continue;
12     /* Modify data ... */
13     memcpy(iov->buf, new_value, val_size);
14     /* Check for concurrent modification */
15     ret = dios_commit_write(D_COMMIT_WRITE_IOV_USE, val_ref,
16                             val_size, iov);
17     if (ret == 0)
18       /* Write was valid under EXCL concurrent access, done */
19       break;
20   } while (true);
21   /* De-allocate buffer */
22   dios_release_write(D_NONE, val_ref, &iov);
23 }
```

Listing 4.5: 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.

value representing `locked` or `unlocked` states. To obtain the lock, an application first performs a read I/O request with exclusive access semantics. If this request succeeds and the mutex value is set to `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.

Some more complex, multi-object abstractions require "happens-before" relationships on requests issued by different tasks. I/O requests on their own do not provide such an ordering relationship because they are decentralised. However, an application can implement e.g. Lamport clocks [Lam78] via I/O requests, and thus enforce a partial ordering on events. 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.

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 application performs its I/O on the buffer and eventually *commits* the request. At this point, the

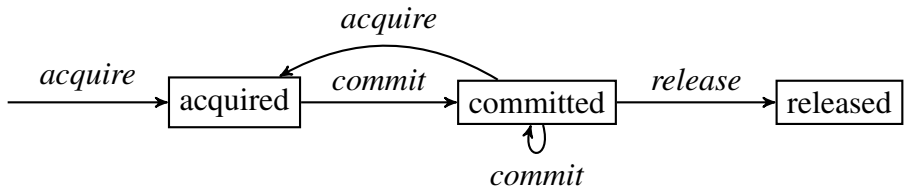


Figure 4.8: Stages of a DIOS I/O request and transitions between them.

```
[...]
/* Read chunk size prefix */
ret = dios_acquire_read(size_buf ? D_ACQUIRE_READ_IOV_REUSE : D_NONE,
                        input_ref, sizeof(uint64_t), &size_buf, D_ANY);
if (ret < 0)
  return ret;
/* Got an incoming chunk size */
uint64_t read_size = *(uint64_t*)size_buf->buf;
ret = dios_commit_read(D_NONE, input_ref, size_buf);
if (ret < 0)
  return ret;
/* Can't re-use data buffer if it's too small */
if (read_size > data_buf->len) {
  dios_release_read(D_NONE, input_ref, data_buf);
  data_buf = NULL;
}
/* Get the actual data chunk */
ret = dios_acquire_read(
  data_buf ? D_ACQUIRE_READ_IOV_REUSE : D_ACQUIRE_READ_IOV_CREATE,
  input_ref, read_size, &data_buf, D_ANY);
if (ret < 0)
  return ret;
/* Process input data from mapper */
[...]
```

Listing 4.6: 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.

buffer could be de-allocated if the request succeeded. However, if further I/O must be done on the same object, or if the request failed, releasing the buffer would be wasteful. Instead, the buffer should be reuseable for further I/O until explicitly released.

Therefore, I extend the simple acquire–commit sequence with an additional *release* step and several extra transitions (Figure 4.8). 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. In other words, a DIOS 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.

Listing 4.6 shows a practical example of this buffer re-use in a MapReduce reducer's input processing code, where buffers are re-used across I/O requests to amortise setup costs.

On a single machine, the DIOS I/O request implementation is straightforward. Across machines, however, additional communication is required to coordinate the requests and to move data into the appropriate buffers. In the next section, I discuss how DIOS implements this communication.

4.8 Distributed coordination

A physical DIOS object exists on a particular machine, but can be interacted with across machine boundaries. Hence, DIOS must coordinate operations and ship data between machines. Two types of communication exist: *coordination* and *data transmission*. Examples of coordination include 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 exceeds that of coordination traffic.

Both types of communication are implemented and executed by the machine kernels in DIOS. Involvement of at least one kernel is required for coordination, since messages must pass through the TCB for authentication and access control; however, alternative designs that only involve the source or destination kernel are conceivable.

Coordination. DIOS implements operations that coordinate different kernels via the DIOS *Coordination Protocol (DCP)*. The DCP is a binary RPC protocol over reliable datagrams, implemented using Reliable Datagram Sockets (RDS) [RDS15]. RPC messages are encoded using Protocol Buffers and handled by a DCP kernel thread on each machine.

DCP RPCs are time-critical and their synchronous execution requires low end-to-end low latency. In Section 5.2.3, I show that DCP RPCs usually achieve round-trip times under 200 microseconds; other work has shown that RPC latencies of tens of microseconds are feasible using commodity hardware [ORS+11].

Most DCP messages are sent directly to the destination machine. However, since DIOS implements a distributed name service (see §4.3.3), it requires a broadcast primitive. While an existing network layer broadcast primitive could be used (e.g. UDP broadcast), such primitives do not typically offer reliable delivery. Instead, DIOS uses a best-effort peer registration protocol over broadcast UDP, and implements broadcast lookups by sending coordination messages to all live peers. The downside of this approach is that it sends p messages for p live peers, duplicating the content of each message. To reduce this overhead, DIOS could use UDP broadcast

	release_write	commit_write	acquire_write	release_read	commit_read	acquire_read	select	pause/resume	copy	run	delete	lookup	create
create	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	(∗)	✓
lookup	✓	(∗)	(∗)	✓	(∗)	(∗)	✓	(∗)	(∗)	✓	(∗)	✓	
delete	✓	(§)		✓	(§)		(‡)		(†)	(†)	(†)		
run	✓	(§)	✓	✓	✓	✓	✓	✓	✓	✓			
copy	✓	(§)	(∗)	✓	(∗)	(∗)	✓	✓	✓				
pause/resume	✓	✓	✓	✓	✓	✓	(‡)	✓					
select	✓	(‡)	✓	✓	(‡)	✓	(‡)						
acquire_read	✓	(§)	(§)	✓	(§)	(§)							
commit_read	✓	(§)	(§)	✓	(§)								
release_read	✓	(§)	(§)	✓									
acquire_write	✓	(§)	(§)										
commit_write	✓	(§)											
release_write	✓												

	Calls are always idempotent.
	Race has no effect; arguments and returns of calls always necessarily disjunct.
	Race okay; appropriate failure indication returned if operation impossible.
✓	Always commutative: re-ordering produces no observable effect.
(∗)	May return a stale or incomplete reference set, or out of date reference attributes.
(†)	Logical object deletion not guaranteed even if `delete(2)` on last physical object.
(‡)	`dios_select(2)` outcome may vary non-deterministically.
(§)	Application must handle I/O concurrency semantics.

Figure 4.9: 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.

over a QJUMP network, which guarantees reliable delivery and a hard upper bound on latency in the absence of hardware failures [GSG+15].[9]

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. Re-ordering can occur if two system calls race within a machine or across the data centre network.

Figure 4.9 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. they are *not* always commutative), but all such outcomes

[9] Centralised coordination, e.g. as in Fastpass [POB+14], could also be employed, but requires ahead-of-time coordination for sending coordination messages; QJUMP requires no ahead-of-time coordination.

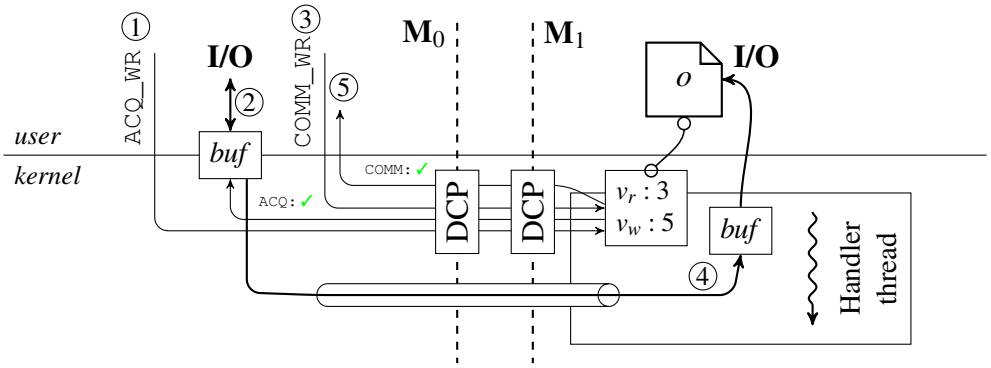


Figure 4.10: Handling write I/O request for a remote object: the task on $\mathbf{M}_0$ first acquires (`ACQ_WR`) via RPC to $\mathbf{M}_1$ and receives a buffer for local I/O; it then attempts to commit the request (`COMM_WR`), the data are sent via the transport object, a remote kernel thread handles the request, validates it, and applies the changes from the buffer.

are permissible under the DIOS API semantics, and must be handled by the application.

This lack of ordered messaging semantics differentiates the DCP from group communication protocols employed in classic 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*. Transport objects are local physical proxy objects for a remote physical target object, which match the target object's I/O semantics. This approach is similar to the implementation of remote I/O in classic distributed operating systems (e.g. in V [Che88]): a remote kernel thread acts as an I/O proxy for the originator task, reading data or applying writes received.

DIOS transport objects are supported by an underlying TCP connection to the kernel on the machine that holds the physical object. In response to DCP messages, I/O data are sent via this TCP connection.[10] The transport object abstraction, however, also supports other means of communication: for example, an RDMA mapping might be used if RDMA is available.

Remote I/O example. Figure 4.10 illustrates DIOS remote I/O with an example. The task on machine $\mathbf{M}_0$ starts a write I/O request on physical object o by making an *acquire* system call. The kernel coordinates with the kernel on $\mathbf{M}_1$ (via the DCP), first issuing an RPC for $\mathbf{M}_1$'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 or an RDMA mapping) and a buffer is returned to user-space

[10]The current implementation uses per-object connections; to scale to more objects, it would multiplex multiple objects' I/O data onto one TCP connection between each pair of hosts. This would require at most as many TCP connections as existing data centres must support. However, connection sharing 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 solve this issue.

on $\mathbf{M}_0$. The application deposits its data in the buffer or modifies its contents, and invokes the *commit* system call. $\mathbf{M}_0$'s kernel now uses the transport object to send the buffer to $\mathbf{M}_1$, 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.

4.9 Scalability

Scalability is a design goal for DIOS, as it runs many tasks across many machines in a data centre. DIOS also aims to address some of the inherent scalability limitations of legacy, single-machine POSIX abstractions observed in prior work [CKZ+13, §4]. In the following, I informally explain how DIOS abstractions support scalability by design.

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 unscalable data structures**: for example, the reference tables are private to each task, 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].

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 either *(i)* are always idempotent, *(ii)* are necessarily invoked on different arguments, or *(iii)* that a race appropriately fails one of the racing calls. 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 (references, unlike POSIX FDs, do not re-use IDs). Kernel-allocated I/O buffers (§4.7) also enable asynchronous release: the resources may be torn down after `dios_release_read(2)` and `dios_release_write(2)` return without any risk of the buffer being re-used in user code.

5. Some of the DIOS abstractions **decompose compound operations** compared to legacy system calls: for example, spawning a task via `dios_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)`.

The practical scalability of the DIOS implementation is affected by the underlying Linux kernel's scalability. However, the above principles are conducive to commutative implementations of operations even if they do not commute in the Linux kernel today. I discuss in Section 9.1.2 hows this might be attempted in future work.

4.10 Linux integration

I developed DIOS as an extension module for the Linux kernel in order to support legacy applications, enable incremental migration, and to keep the implementation effort manageable.

This section describes the DIOS implementation, which consists of a portable core module and an integration module for the Linux host kernel. By adding an appropriate integration module, DIOS could support host kernels other than Linux (see §9.1.2).

In Section 4.10.1, I describe the necessary changes to the core Linux kernel, and show that only minimal changes are required. Following, I describe the implementation of the DIOS modules in Section 4.10.2. The integration of DIOS with Linux allows incremental migration of existing applications, and I discuss this in Section 4.10.3.

4.10.1 Linux kernel changes

The host kernel – Linux, in my implementation – must supply generic kernel functionality that DIOS requires: drivers for machine hardware, bootup code, a network stack, and low-level memory management code. Since this code is not DIOS-specific, it makes sense to re-use stable, mature kernel code, and build the DIOS abstractions on top.

To deploy DIOS, a small number of changes to the core Linux kernel code were required:

System call handlers for the new DIOS system calls had to be added. These require a kernel patch because 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 had to 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 had to be modified to recognise DIOS binaries and initialise them appropriately.

The necessary changes amount to a patch changing about 500 lines ($\approx 0.01\%$ of the non-driver kernel source).[11] Table 4.6 lists the changes and their impact.

In addition to the above, DIOS requires a means of reliably delivering messages to remote machines (§4.8). I implemented this by establishing reliable channels via messaging protocols already supported in the Linux kernel: I use existing kernel support for Reliable Datagram Sockets (RDS) for DIOS coordination messages. RDS supports Infiniband and TCP transports; my experiments use the TCP transport over 10G Ethernet.

[11] Based on the DIOS patch for Linux kernel v3.14.

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

Table 4.6: Changes made to the Linux host kernel for DIOS.

4.10.2 DIOS modules

The DIOS abstractions are not inherently tied to any particular host kernel, even though my implementation uses Linux. DIOS consists of two modules:

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

2. The **DIOS Adaptation Layer** (DAL), which indirects calls into the host kernel (e.g. starting a new process, installing memory mappings) to the matching Linux symbols.

DIOS Core Module. The DCM implements the core of DIOS, including the name and reference tables, management and transformation code for names and references, coordination via 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 Linux kernel functionality.

DIOS Adaptation Layer. The DAL indirects DIOS invocations of host kernel functionality to Linux-specific symbols. Such functionality includes the creation and execution of user-space processes (for DIOS tasks), allocation of physical and virtual memory, and mapping of kernel memory into user-space virtual address spaces (for DIOS references). The DAL also interfaces with the Linux kernel network stack for remote object access and DCP coordination, and indirects other I/O on block devices and the console. Finally, it provides access to Linux concurrency mechanisms such as worker threads and wait queues, as well as architecture-specific implementations of locks, semaphores, and atomics.

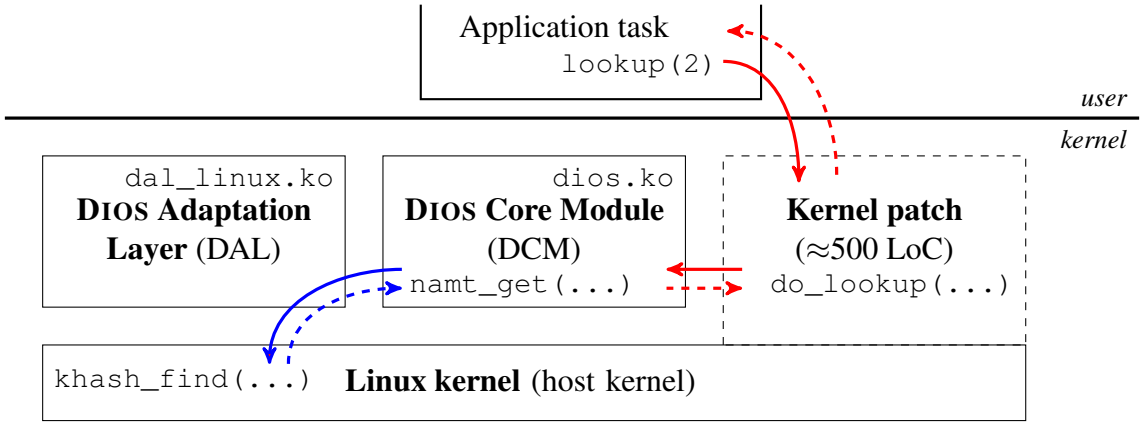


Figure 4.11: Structure of DIOS implementation, showing the constituent kernel modules using the example of handling a `lookup(2)` system call: the call (red arrows) is forwarded by the kernel patch to the DCM (`dios.ko`), which uses the DAL (`dal_linux.ko`) to invoke Linux kernel functionality (blue arrows). Dashed arrows show the return path.

Since the DAL is an adaptation layer, it typically wraps an appropriate Linux kernel function, but in some cases maintains extra state (e.g. for work queues). It is fairly compact and consists of about 3,000 lines of C code.

Example. Figure 4.11 illustrates the roles of the kernel patch, the DAL module, and the DIOS Core Module using the example of a `dios_lookup(2)` system call invocation. The system call is first invoked in a user-space task and handled by a shim system call handler in the Linux kernel patch. This handler proxies the system call invocation to the DIOS core module, which performs a name table lookup (`namt_get()`), which in turn relies on the DAL indirecting access to the Linux kernel hash table data structure (`khash_find()`). Finally, the result is returned to user-space via the shim handler.

4.10.3 Incremental deployment

Unlike research operating systems that support conventional APIs (e.g. subsets of POSIX) via compatibility layers – such as Mach [ABB+86, §8] or L4 [HHL+97] – the Linux integration of DIOS allows native execution of legacy applications side-by-side with DIOS applications. This enables DIOS to be deployed while applications are incrementally ported to it.

In such a mixed environment, DIOS must however differentiate between process types in order to *(i)* initialise, manage, and destruct DIOS-specific state when needed, and *(ii)* restrict the availability of DIOS objects and system calls to applications that are permitted to use them.

DIOS can be configured for different degrees of restriction:

- In **liberal mode**, applications can exist in a "limbo" state that allows them to mix DIOS objects and legacy OS facilities (e.g. pipes and FDs).

Binary type		ELF brand ABI	ELF brand version	System calls: *liberal mode* legacy	System calls: *liberal mode* DIOS	System calls: *restricted mode* legacy	System calls: *restricted mode* DIOS	Start via	Access pure DIOS obj.
DIOS	Pure	`0xD1`	`0x05`	✗	✓	✗	✓	`run(2)`	✓
	Limbo	`0xD1`	`0x11`	✓	✓	✗	✓	`run(2)`	✓†
	Legacy	`0xD1`	`0x13`	✓	✓	✓	✗	legacy	✗
Legacy ELF		any other		✓	✗	✓	✗	legacy	✗

(✓†: only in restricted mode.)

Table 4.7: 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 **restricted mode**, applications must either use only DIOS abstractions, or only legacy ones, and cannot mix them inside the same process.

The choice between liberal and restricted mode is global, but a "branding" on binaries permits different degrees of per-process restrictions. Table 4.7 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. The must, however, be launched via `dios_run(2)`.

An example limbo application is the Firmament scheduler (Chapter 6), which may use both legacy and DIOS abstractions: the former for monitoring tasks and the latter (i.e. `dios_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 the `dios_run(2)` system call. This is useful when a legacy process without access to DIOS facilities must start a DIOS-enabled process.

A key use case for DIOS legacy mode is bootstrapping DIOS at startup: for example, the `dizibox` shell or a local DIOS init process are launched by a legacy process, `init`. This works in liberal mode; in restricted mode, `init` itself must be a DIOS binary.

Non-DIOS legacy binaries execute as if DIOS did not exist. They can only use legacy host kernel abstractions, and the DIOS namespaces and objects are unavailable to them.

Most legacy utilities fall into this category; examples are 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.[12]

The ability to run different binary types at the same time enables an incremental migration path from legacy abstractions to using DIOS ones.

4.11 Summary

In this chapter, I introduced DIOS, a prototype implementation of the decentralised data centre OS model introduced in Chapter 3.

After an overview (§4.1), I described the key abstractions in DIOS and their implementations in detail. Distributed objects encapsulate state and data (§4.2); names serve as identifiers (§4.3); groups implement distributed namespaces (§4.4); and references realise the model's context-sensitive handles (§4.5).

I also gave an overview of the DIOS system call API (§4.6), explained how it realises I/O requests (§4.7), implements distributed operation via the DCP and transport objects (§4.8), and how it embodies scalable design principles (§4.9).

Finally, I outlined how DIOS extends the Linux kernel such that it maintains backwards compatibility with legacy applications and supports incremental migration (§4.10).

In the next chapter, I evaluate DIOS in a testbed deployment.

[12] 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].

Chapter 5

DIOS evaluation

In this chapter, I evaluate DIOS, my implementation of the decentralised data centre OS model. Since the design and implementation of a new operating system even with the most basic functionality is a large, complex, and time-consuming undertaking, the evaluation of DIOS must be taken with a pinch of salt: its functionality is limited, its implementation is unoptimised, and its software stack is less mature than those it is being compared against. Better performance can almost certainly be attained. Its experimental nature can, however, also work in favour of DIOS: as a research OS, it is leaner and may occasionally perform better than the more featureful systems I compare against.

In the following, I aim to answer three questions:

1. Do the DIOS abstractions add any undue overheads over "classic" POSIX abstractions?
2. Is the model behind DIOS sufficiently powerful and performant to support typical data centre applications, e.g. parallel MapReduce-style data processing?
3. What are the qualitative security benefits of the capability-based resource management specified in the decentralised data centre OS model (and implemented in DIOS), and how does it compare to state-of-the-art isolation techniques?

In a set of micro-benchmarks, I test individual DIOS features and system calls to address the first question (§5.2). 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 (§5.3). Finally, I analyse how system security can be improved using DIOS (§5.4), answering the third question.

5.1 Experimental setup

I deployed DIOS on a 28-machine testbed cluster in the Computer Laboratory model data centre. All machines are Dell R320 servers with an Intel Xeon E5-2430Lv2 six-core processor with

twelve hyperthreads. Their CPU clock frequency is 2.4 GHz, all machines have 64 GB of PC3-12800 RAM, and they are connected by a leaf-spine topology 10G network with a 320 Gbit/s core bandwidth. While hardly "warehouse-scale", this testbed constitutes a realistic and complex networked environment for a real-world deployment of DIOS.

Implementation limitations. Before I describe my experiments, I point out a number of limitations of the current DIOS implementation. 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. Persistent object storage (§3.6) is implemented as a special directory within the legacy Linux kernel file system.
2. DIOS does not currently have an implementation of group-based namespaces (§4.4). Adding them, however, will not affect performance, since the implementation merely requires indirection of DIOS name table lookups.
3. Reference delegation (§4.5.3) is only partially implemented; my experiments rely on the more expensive name lookup instead.
4. The I/O request API (§4.7) is implemented, but there is only limited support for the different concurrent access semantics.
5. The DCP (§4.8) uses synchronous RPCs implemented over RDS. They time out on failure, but no recovery action is taken when machines fail.

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.

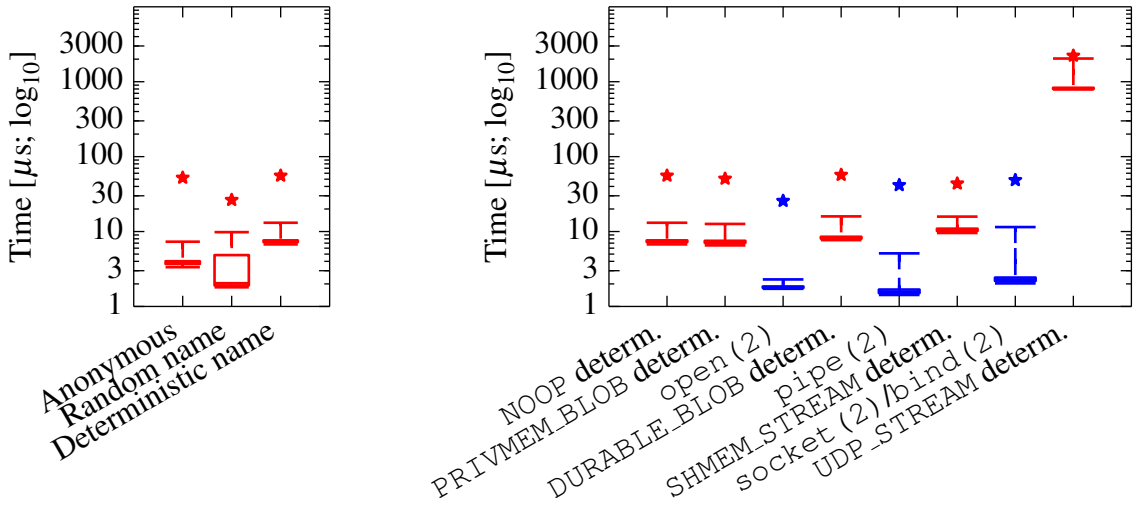
5.2 Performance micro-benchmarks

The invocation of privileged operating system functionality via system calls is often 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 abstractions via micro-benchmarks. I consider object creation (§5.2.1), task creation (§5.2.2), and the performance of I/O via DIOS objects (§5.2.3).

5.2.1 Object creation

DIOS programs create objects to allocate memory, store data, and to open communication channels. In the first experiment, I thus consider the performance of the `dios_create(2)` system call.



(a) Latency by DIOS name type.

(b) Latency by object type, compared to POSIX mechanisms.

Figure 5.1: Micro-benchmark of 1024 object creations: per-entity creation latency for DIOS objects (red) is competitive with traditional Linux entities (blue). Boxes delineate 25th, 50th, and 75th percentiles, whiskers correspond to the 1st and 99th percentile latencies, and the star corresponds to the maximum value observed.

As explained in Section 4.3, DIOS objects can be anonymous, can have random names, or use deterministic naming. The name generation methods have different overheads: random name generation must source 32 random bytes, while deterministic name generation computes several SHA-256 hashes. Figure 5.1a illustrates the impact on object creation latency: deterministic name creation is the most expensive, at 7.5µs in the median, while random names are cheaper, but see higher variance, than anonymous objects, which take 4µs in the median to create.

Figure 5.1b 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 object abstraction adds a 4–5× overhead over legacy Linux system calls.[1]

However, this overhead is somewhat overstated: the 10µs creation time for DIOS objects is

[1]The high creation overhead for `UDP_STREAM` objects (around 800µs) is a consequence of allocating large internal buffers, and of synchronously binding and connecting the underlying socket.

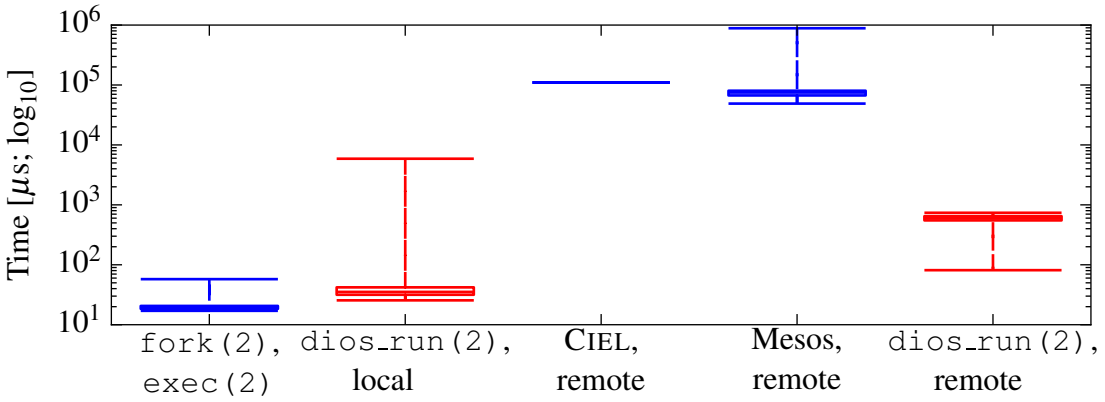


Figure 5.2: Task spawn experiments with `dios_run(2)` and existing OS and distributed systems facilities. Each experiment spawns 5,000 tasks that each run for 100ms in waves of twelve. Boxes delineate 25th, 50th, and 75th percentiles, whiskers correspond to the 1st and 99th percentile latencies. Value for CIEL according to Murray [Mur11, p. 132].

dominated by the deterministic name generation (cf. Figure 5.1a). With anonymous objects, the results are closer to the Linux system calls at an overhead of about 2× overhead, which can be attributed to lack of critical path optimisation and the cost of indirecting calls via the Linux kernel patch into the DIOS core module (cf. §4.10.2).

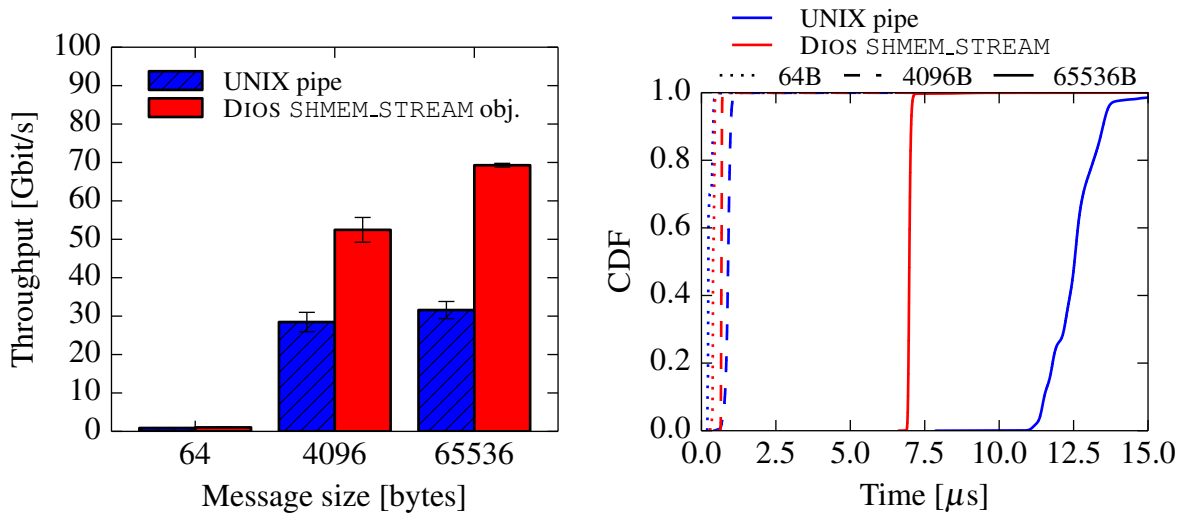
5.2.2 Task spawn overhead

Many data centre applications (e.g. a web server) use long-lived tasks. 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 implemented as application-level requests, rather than cluster-level tasks since the task creation overhead is large. DIOS has the opportunity to reduce this overhead, and thus to permit more applications to use fine-grained tasks that better compartmentalise the application for improved security.

I therefore 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 `dios_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 5.2, I compare these two 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 the Mesos cluster manager [HKZ+11].

The cost of a purely local DIOS task spawn is similar to that of a Linux process creation –



(a) Throughput.

(b) Read latency.

Figure 5.3: Performance of streaming shared-memory I/O mechanisms in DIOS and Linux equivalent. All experiments sequentially read and write one million messages; average and standard deviation are calculated over ten runs.

an unsurprising outcome, since the work performed is largely identical.[2] However, creating a remote DIOS task takes 100–300μs, which is significantly than similar task creations in CIEL and Mesos, which take hundreds of milliseconds.[3] This outcome is not necessarily surprising: CIEL and Mesos are not specifically optimised for fast task spawning (although neither is DIOS), and process all task spawn requests at a single, centralised master. DIOS, by contrast, allows direct, decentralised task spawns by direct interaction with the kernel on a remote machine. This saves network round trips and removes two kernel/user-space transitions compared to the centralised, user-space controllers for task creation in CIEL and Mesos.

5.2.3 I/O performance

Good I/O performance is crucial to many data-intensive applications, and I thus measure the I/O performance obtained by DIOS objects. To do so, I run two DIOS tasks situated *(i)* on the same machine; and *(ii)* 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 5.3a 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 higher throughput (by up to 87.5%) than the pipe transport does. This is perhaps a little surprising, but can be explained

[2]The 99$^{\text{th}}$ percentile outlier for DIOS `run(2)` comes because DIOS uses the Linux kernel's User Mode Helper (UMH): the process creation indirects via two kernel threads, which are subject to scheduling delays.

[3]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.

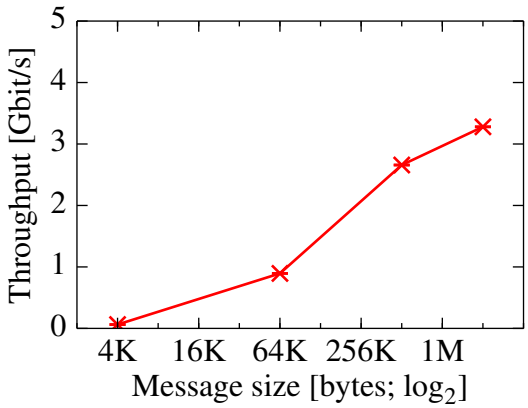


Figure 5.4: Average synchronous, unpipelined I/O throughput for one million reads via a TCP transport object as a function of the read size. Each I/O request takes 300–400μs.

by *(i)* by DIOS avoiding locks to restrict concurrent access, and *(ii)* by DIOS re-using the same buffer for subsequent requests (the `ACQUIRE_IOV_REUSE` options to `dios_acquire_read(2)` and `dios_acquire_write(2)`), which reduces the necessary dynamic memory allocations. The one-way message latency (Figure 5.3b) follows a similar pattern: the DIOS `SHMEM_STREAM` object's read latencies are significantly lower than those of the pipe.

I also consider **remote I/O**. Figure 5.4 shows the throughput measured when a single task makes synchronous, unpipelined 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 300–400μs. At 4 KB reads, the throughput is around 16 MBit/s, while large 2 MB reads reach around 3.3 GBit/s. This relatively low throughput is a consequence of the transport object being idle while the synchronous, small (≈40 bytes) DCP RPCs are processed.

In addition to optimising the current, naïve DCP and TCP transport implementations in DIOS, there are several other ways to improve upon this:

1. Non-blocking, batched, or asynchronous I/O requests would allow DIOS to achieve higher throughput, as would request pipelining in the benchmark application. High-throughput systems that serve small requests typically use similar techniques (e.g. Facebook's use of "multigets" in memcached [NFG+13]).

2. 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" (forgoing validation) or "commit-and-acquire" (immediately starting a new request on commit), the number of DPC RPCs per I/O request can be halved, although this would increase the API complexity.

3. Hardware offload and kernel bypass techniques can be applied for the DCP traffic and

would accelerate the coordination RPCs, as there is no need for DCP messages to traverse the entire network stack. Existing systems already achieve request latencies around 20μs 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 matches the performance of state-of-the-art systems.

5.3 Application benchmark

The processing of huge data volumes is a classic "scale-out" analytics workload common in data centres. 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 §2.1.1.1). 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 and compares it against two state-of-the-art systems built atop standard Linux abstractions: the Metis multi-threaded, single-machine MapReduce framework [MMK10], and Spark [ZCD+12] v1.4.0.[4]

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 computes the number of occurrences of each word in a dataset. However, the implementation is sufficiently generic to support any computation that fits the MapReduce paradigm. Listing 5.1 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 datasets:

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.[5]

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.

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

[4]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].

[5]The original Metis dataset had no line breaks; since Spark partitions map inputs by newline characters, I reformatted the dataset to ten words per line.

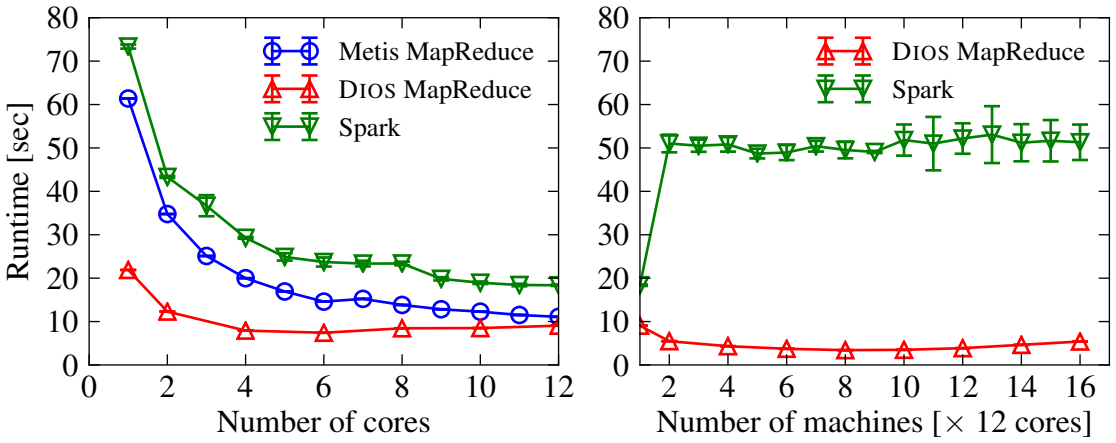
```
#include "dmr.h"
#include "dmr_map.h"
#include "dmr_reduce.h"

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;
}
```

Listing 5.1: User-provided code in the DIOS MapReduce WordCount implementation.

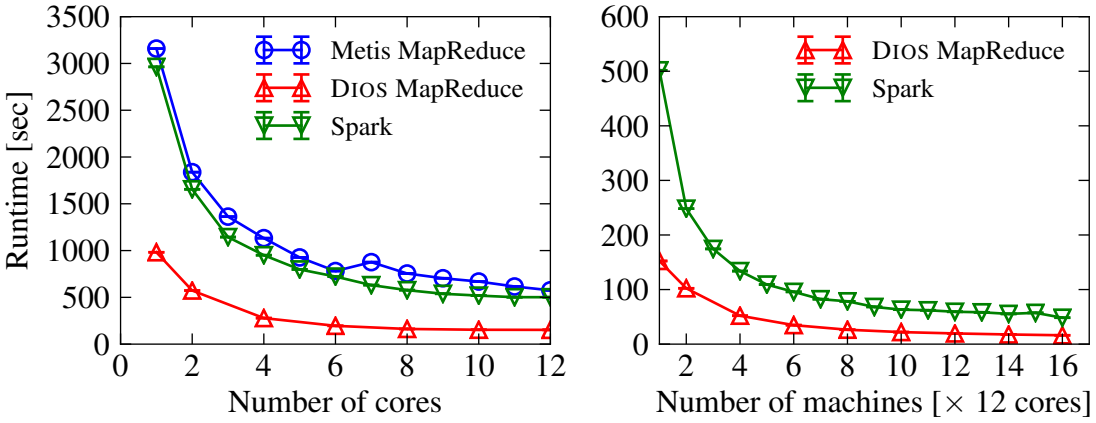
dataset (24 MB of output) and less than 40 seconds for the Wikipedia dataset (3.7 GB of output) even if written entirely on a single machine.

In Figure 5.5a, I show the job runtime on the Metis dataset 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 due to a single straggler task in the execution.[6] DIOS MapReduce continues to (asymptotically) scale

[6]The other Spark tasks finish roughly at the same time as the DIOS MapReduce job.



(a) Metis dataset (300 MB, 1M keys). The lowest runtime is achieved by DIOS MapReduce on eight machines (3.4s).



(b) Wikipedia dataset (21 GB, 111M keys). Note the change of *y*-axis scale between the graphs. The lowest runtime is achieved by DIOS MapReduce on sixteen machines (16.2s).

Figure 5.5: MapReduce WordCount application performance in DIOS, scaling to multiple cores on one machine, and to a cluster of 16 machines. Values are averages over five runs, and error bars represent standard deviations.

up to eight machines (96 tasks, handling ≈3 MB each), after which the coordination overheads start to dominate and runtime degrades slightly. This illustrates that DIOS supports fine-grained parallel distributed processing even for small datasets.

Figure 5.5b shows the same results for the larger Wikipedia input dataset. 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 up to 3× faster than Spark on the Wikipedia dataset. 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 application-level fault toler-

ance 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. The decentralised data centre OS model and its DIOS implementation are sufficiently expressive to efficiently implement parallel data analytics.
2. DIOS scales to hundreds of parallel tasks, and offers parallel speedups even for fine-grained processing, and matches or outperforms current data processing systems' runtime for this workload.

These performance results are an encouraging sign that DIOS may offer efficiency gains for data centre software. In the following, I consider additional qualitative benefits of using DIOS.

5.4 Security

Security and isolation in the decentralised data centre OS model, and thus in DIOS, rely on distributed capabilities (§3.4–3.5). In the following, I qualitatively assess the resulting security properties of the model by *(i)* comparing it with alternative isolation techniques used in data centres, and *(ii)* sketching a plausible attack that its capability-based protection mitigates.

5.4.1 Comparison with alternative approaches

The primary security goal in a data centre is to isolate independent, possibly mutually distrusting applications atop shared hardware and software (§3.2.2). Table 5.1 lists existing, commonly deployed isolation mechanisms and compares their features.

Approaches differ in their overheads (e.g. containers' lightweight namespaces *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 infrastructure user (e.g. specialised, custom kernels in unikernels *vs.* a shared underlying kernel). In general, higher specialisation effort on the infrastructure user's part 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 across different compartments on different machines. Permitting an application to establish network connections is tantamount to enabling it to send *any* of the data accessible to it to any other task in the data centre, unless application-level authentication and restrictions are implemented. Even if such restrictions are present, they are not uniform across different applications (see §2.2.3).

Approach	Overhead	Per-task kernel	Isolated distr. namespaces	Separate filesystem	Fine-grained capabilities	Resource sharing
Shared machine (processes)	none	✗	✗	✗	✗	✓
Virtual machines (VMs)	high	✓	✗	✓	✗	✗†
Specialised OS/stack (unikernels)	medium	✓	✗	✓	✗	✗†
Kernel namespaces (containers)	low	✗	✗	✓	(✓)	✓
Capability systems (e.g. Capsicum)	low	✗	✗	(✓)	✓	✓
Decentr. data centre OS model (DIOS)	low	✗	✓	✓	✓	✓

Table 5.1: DIOS security properties compare favourably with other inter-task isolation techniques deployed in data centres. († possible with hypervisor support)

The decentralised data centre OS model, by contrast, supports both *isolated distributed namespaces*, and *fine-grained capabilities* using split identifier/namespace capabilities and translucent handle capabilities. Hence, it supports selective sharing of objects (via delegated handles), and compartmentalises tasks such that they can only access (and leak data to) a specific, narrow set of objects. DIOS implements this compartmentalisation using names, groups, and references. By contrast with VMs and unikernels, however, DIOS does not run a full OS kernel per task (and hence requires all applications to use the same kernel), and by contrast with containers, it cannot isolate legacy applications from each other.

5.4.2 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 `map()` and `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), but the job may also access other information accessible to the same principal (e.g. email access logs). The evil MapReduce job's user-provided `map()` function can "siphon" this additional information into a network connection to another, harmless-looking task in another job started by the same principal.

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 data centre – 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]. Other distributed infrastructure

systems see the same limitations, or even do not support access control at all (see §2.2.3.2). This only enforces fairly coarse-grained isolation: *any* file and directory accessible to the authenticated user is accessible to *all* jobs running as this infrastructure 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 DIOS. If the MapReduce tasks are implemented as pure DIOS binaries, DIOS restricts them to only access their predefined input and output objects. These input and output objects are determined by the infrastructure user, but can be matched against a security policy (e.g. "no job can touch both click logs and end-user emails without approval") by the cluster manager. The cluster manager creates the DIOS MapReduce job controller task and grants it membership of groups containing the job input and output objects. The job controller task does not run any code provided by the infrastructure user, but resolves the job's input names, creates intermediate objects and output objects, and delegates references to these objects to the mapper and reducer tasks.

Since the mapper and reducer tasks share no group memberships with the controller task or with each other, the network communication between distributed file system, mapper and reducer tasks is restricted to those streams and intermediate objects permissible under the MapReduce programming model. When using DIOS objects, 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 DIOS 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.

5.4.3 Limitations

Like most operating systems, DIOS offers no protection against *TCB compromises* that enable an attacker to execute arbitrary code inside the TCB (§3.2.2.2). This could happen due to an exploitable kernel or cluster manager bug. However, commonly deployed data centre isolation

[7]The 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 DIOS can deny creation of these object types to untrusted tasks.

solutions face the same issues: Linux containers do not protect against a kernel compromise, and VMs do not protect against a hypervisor compromise. However, DIOS reduces the attack surface by offering only a minimal API to pure DIOS tasks (see §4.10.3).

DIOS is also not currently resilient to *man-in-the-middle attacks* on the data centre interconnect. If an attacker manages to compromise an intermediary (usually a data-link layer switch or a router), the integrity of RPCs and coordination messages can be compromised. Section 9.1.3 discusses possible solutions to this problem.

5.5 Summary

In the preceding sections, I used DIOS to evaluate my decentralised data centre OS model. I set out a set of requirements for such a novel distributed OS model in Section 3.2, and DIOS demonstrates that a practical implementation of this model is feasible.

I have demonstrated that DIOS, my implementation of the decentralised distributed data centre OS model, is practical and achieves efficiency and security benefits over the state-of-the-art:

1. The DIOS abstractions are competitive with traditional POSIX abstractions in micro-benchmarks, adding either only minor extra overhead or outperforming them, and enable low-overhead distributed operations (§5.2).

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 (§5.3).

3. The decentralised distributed data centre OS model's resource management abstractions offer isolation across machines that improves security over current approaches such as local containers (§5.4).

Of course, the performance results presented are impacted by implementation choices in Linux, which are sometimes at odds with the goals of DIOS. In Section 9.1.2, I discuss deeper changes to the Linux kernel that would benefit DIOS.

Chapter 6

Flexible and scalable scheduling with Firmament

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 DIOS.

In modern data centres, scheduling of work to compute resources 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 (§6.1), 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 generalises.

Like Quincy, Firmament models the scheduling problem as a minimum-cost, maximum-flow optimisation over a flow network (§6.2). This approach balances multiple mutually dependent scheduling goals to arrive at a globally optimal schedule according to the costs.

Moreover, Firmament can express many scheduling policies via its flow network using this information – unlike Quincy, which only supports a single policy. I outline how data local-

ity, fairness, and placement constraints are represented, and point out limitations of the flow network approach (§6.3).

After describing how Firmament expresses different policies, I discuss its algorithmic core, the minimum-cost, maximum-flow optimisation. While Firmament primarily targets OLAP-style analytic workloads and long-running services (cf. §2.1.1), it also scales the flow-based scheduling approach to workloads that include short, transaction processing-style tasks in large data centres (§6.4).

I then discuss the Firmament implementation, focusing on its architecture, and how it collects fine-grained information about machine resources and running tasks (§6.5).

Finally, I summarise the chapter (§6.6). In the next chapter, I will I discuss four examples of customisable *cost models* for Firmament, define the flow network structure and the costs assigned to the network's arcs to define different scheduling policies.

6.1 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 (§6.1.1).

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 (§6.1.2).

6.1.1 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 6.1 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 *between* machines.

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

Feature	Cluster schedulers (§2.3)	CPU schedulers	Quincy [IPC+09]	Firmament (§6.2 et seq.)
Scheduler architecture	centralised, distributed	centralised, per-CPU	centralised	supports either
Operating scale	100–10,000 machines	hundreds of cores	≤2,500 machines	≥10,000 machines
Workload types	batch, service	batch, interactive, real-time	batch	batch, service
Task length assumption	varies (scheduler-specific)	very short to infinite	≥seconds	short to infinite
Fairness	varying notions	varying notions	fair share of tasks	fair share of tasks
Constraints	soft, hard	affinity, pinning	✗	soft, hard
Admission control	✓, multi-dim.	✓	✓	✓
Preemption	common	✓	✓	✓
Explicit priorities	common	✓	✗	✓
Dynamic priorities	fairly rare	✓	✓, implicit	✓, implicit
Global view of WSC	✓	✗	✓	✓
Application-specific policies	some	✗, except user level schedulers	✗	supported
Heterogeneity awareness	rare	✗	✗	✓
Micro-architecture awareness	✗	some	✗	✓
Interference awareness	rare	rare, heuristic-based	✗	✓
Data locality awareness	✓, common	rare	✓	✓
Deadline support	rare	rare (RT only)	✗	✓
Gang scheduling	rare	✓, common	✗	✓
Automatic resource scaling	rare	✗	✗	✗

Table 6.1: Similarities and differences between cluster task scheduling, CPU scheduling, Quincy, and Firmament along different feature dimensions.

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].

6.1.2 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 a single-machine CPU scheduler [AA14, ch. 8; PS85, pp. 127–9], and enjoys 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 cluster-wide scheduling make 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.

Google's Borg scheduler uses a similar cost-based scoring approach [VPK+15, §3.2]. In Quincy, however, the optimiser considers *all* jobs, tasks, and machines simultaneously, and thus many different choices and their impact on the cluster. If the costs are set correctly, the resulting schedule is globally optimal – a property that heuristic-driven, queue-based schedulers cannot guarantee.

Quincy maps the scheduling problem to a graph – a flow network – and optimises this graph. Firmament follows the same basic approach, but generalises it: as Table 6.1 shows, Firmament supports more scheduling policies than Quincy, adds several advanced features, and utilises detailed information traditionally unavailable to cluster schedulers.

6.2 Scheduling as a flow network

At its core, Firmament models also the scheduling problem as an optimisation over a flow network (§6.1.2), but generalises the Quincy approach.

This section explains how Firmament constructs the flow network (§6.2.1), and how it assigns arc capacities (§6.2.2) and costs (§6.2.3). In each case, I explain how Firmament's approach differs from Quincy's. Finally, I introduce *equivalence classes*, aggregation vertices in Firmament's flow network, which enable generalisation over Quincy and support for additional scheduling policies (§6.2.4).

6.2.1 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 models task assignments as flow through machine vertices. 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.[1] I discuss the details of common optimisation algorithms for this problem in Section 6.4. 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 data centre resources (the *resource topology*) that corresponds to the physical real-world data centre topology. For example, *machine*

[1] "Policy-optimal" means that the solution is optimal for the given scheduling policy; it does not preclude the existence other, more optimal scheduling policies.

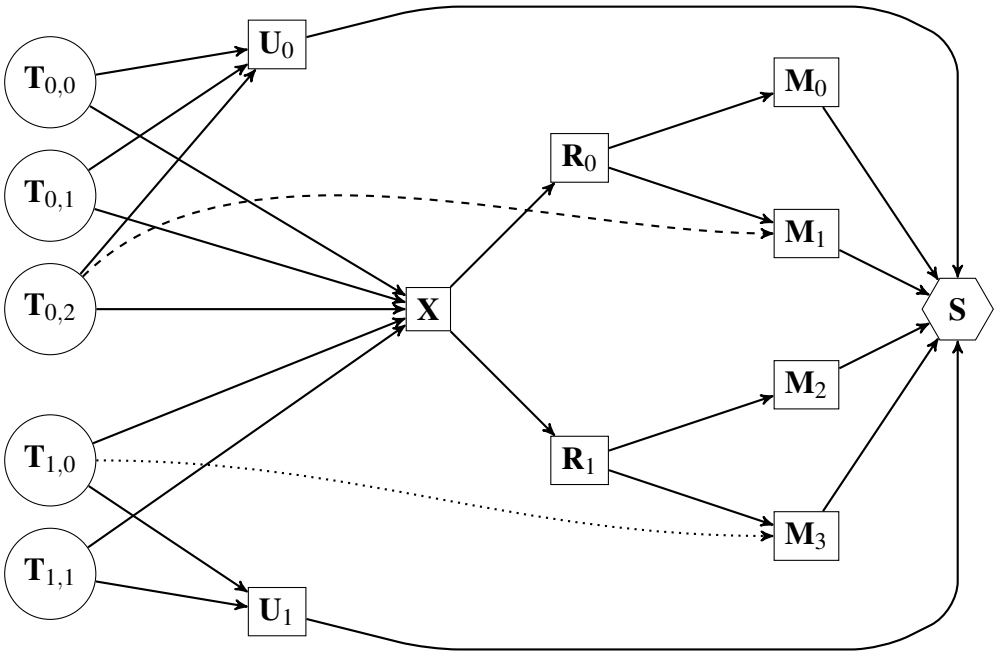


Figure 6.1: Example flow network as generated by Firmament for a four-node cluster consisting of two racks with two machines each. Flow is generated at task vertices ($\mathbf{T}_{j,i}$) and routed to the sink ($\mathbf{S}$) via either machine vertices ($\mathbf{M}_m$) or per-job unscheduled aggregators ($\mathbf{U}_j$). The dashed line is a task preference arc ($\mathbf{T}_{0,2}$ prefers $\mathbf{M}_1$), while the dotted line corresponds to $\mathbf{T}_{0,1}$ running on $\mathbf{M}_3$.

vertices are usually subordinate to *rack* vertices, which in turn descend from a common *cluster aggregator* vertex. Figure 6.1 shows an example flow network for a date centre consisting of four machines ($\mathbf{M}_0$–$\mathbf{M}_3$) distributed over two racks ($\mathbf{R}_0$–$\mathbf{R}_1$). The cluster aggregator vertex is labelled as $\mathbf{X}$.[2]

In addition to vertices for the data centre resources, the flow network also contains a vertex for each task, independent of whether it is currently scheduled or not. These *task* vertices are flow sources and each generate one unit of flow (as in Quincy). In the example, two jobs with three and two tasks are present ($\mathbf{T}_{0,0}$–$\mathbf{T}_{0,2}$ and $\mathbf{T}_{1,0}$–$\mathbf{T}_{1,1}$).

The flow generated by task vertices is eventually drained by a sink vertex ($\mathbf{S}$). 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 be insufficient 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 ($\mathbf{U}_0$–$\mathbf{U}_1$). As in Quincy, one such vertex exists for each job, and all unscheduled aggregators are connected to the sink.

Only the leaf vertices of the resource topology and the unscheduled aggregator vertices are

[2]The exposition here matches Quincy's flow network structure, but in Firmament, the specific structure is not fixed: as I show later, the $\mathbf{X}$, $\mathbf{R}_r$, and $\mathbf{M}_m$ vertices are merely examples of aggregating *equivalence classes*.

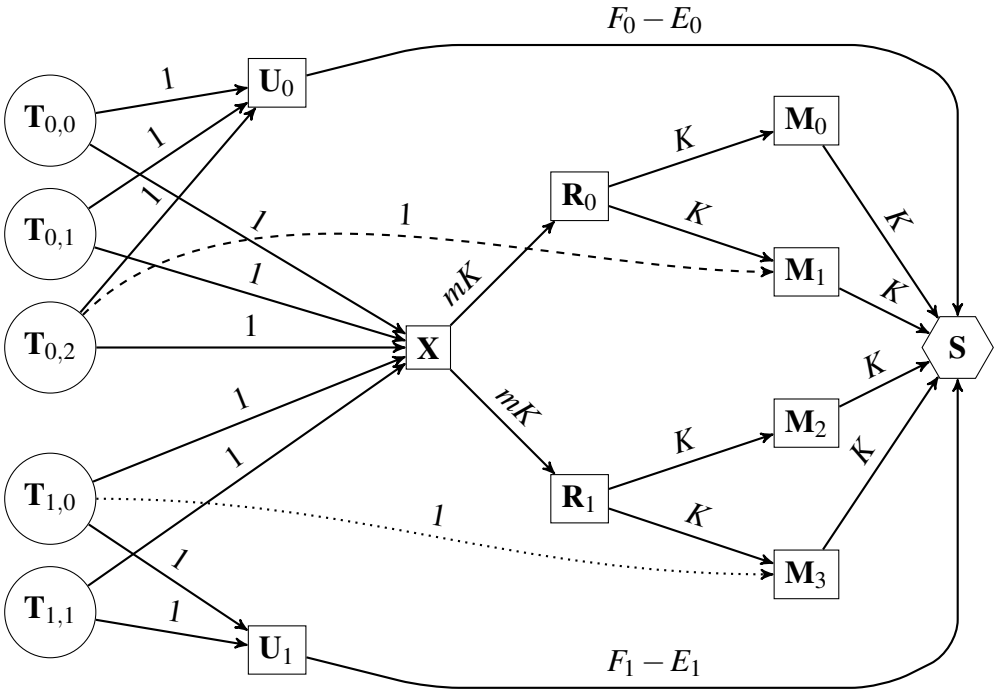


Figure 6.2: The flow network in Figure 6.1 with example capacities on the arcs.

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 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.

However, unlike 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 added to the flow network. I explain this extension in detail in Section 6.5.2.

The quality of scheduling decisions made by optimising a given flow network depends on the capacities and costs assigned to its arcs. In the following sections, I explain how these arc parameters are determined.

6.2.2 Capacity assignment

Each arc in a flow network has a *capacity* for flow within an interval $[cap_{\min}, cap_{\max}]$. In Firmament, as in Quincy, $cap_{\min}$ is generally zero, while the value of $cap_{\max}$ depends on the type of vertices connected by the arc and on the cost model.[3]

[3]For simplicity, "the capacity" refers to the maximum capacity value in the following.

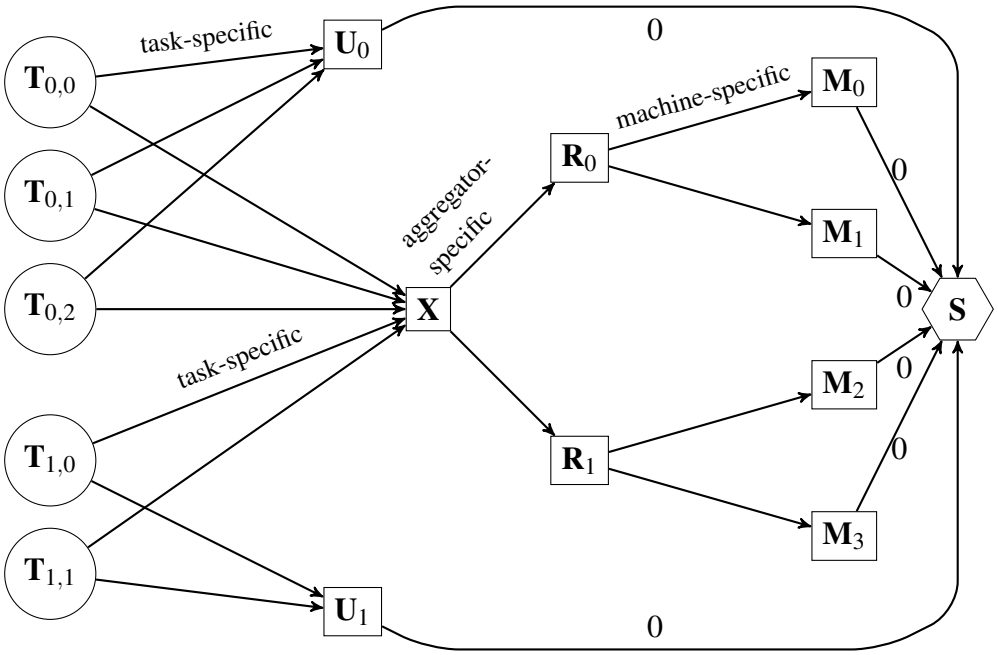


Figure 6.3: The example scheduling flow network in Figure 6.1 with annotations highlighting the cost scopes for each arc type, and zero cost terms.

Figure 6.2 illustrates a typical capacity assignment. Outgoing arcs from tasks have unit capacity, and since each machine ($\mathbf{M}_0$–$\mathbf{M}_1$) can run a maximum of K tasks, the arcs from machines to the sink have capacity K. Arcs from the cluster aggregator ($\mathbf{X}$) to rack aggregators ($\mathbf{R}_0$–$\mathbf{R}_1$) have capacity rK for r machines in each rack. Finally, the capacities on the arcs from jobs' unscheduled aggregators ($\mathbf{U}_0$–$\mathbf{U}_1$) 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, but others are configurable, and thus allow Firmament to express different scheduling policies. I explain the general costs in the next section, and describe how four Firmament cost models configure others in Chapter 7.

6.2.3 Cost assignment

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.

To extend Quincy, which assigns ad-hoc costs for its singular policy, I developed the notion of cost "scopes", which depend on where an arc is in the flow network (Figure 6.3).

Parameter	Edge	Meaning
v_i^j	$\mathbf{T}_{j,i} \to \mathbf{U}_j$	Cost of leaving $\mathbf{T}_{j,i}$ unscheduled.
α_i^j	$\mathbf{T}_{j,i} \to \mathbf{X}$	Cost of scheduling in any location (usually: worst-case cost).
$\gamma_{i,m}^j$	$\mathbf{T}_{j,i} \to \mathbf{M}_m$	Cost of scheduling or continuing to run on machine $\mathbf{M}_m$.
ω	–	Wait time factor.
ν_i^j	–	Total unscheduled time for task.
θ_i^j	–	Total running time for task.

Table 6.2: 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.

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 $\mathbf{T}_{j,i} \to \mathbf{X}$ 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 ($\equiv$ aggregated by) the destination aggregator.

Example: the arcs $\mathbf{X} \to \mathbf{R}_r$ are aggregator-specific arcs, since both the cluster aggregator, $\mathbf{X}$, and the rack-level vertices, $\mathbf{R}_r$, 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 $\mathbf{R}_r \to \mathbf{M}_m$ 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 §6.2.4). 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.

Some general notions of cost are key to the min-cost flow scheduling approach. I explain them in the following and list them in Table 6.2; the terms here are largely identical to Quincy's. The cost models presented in Chapter 7 add additional arcs and cost expressions, and use specific expressions to compute values for the cost terms in Table 6.2.

1:1 task–resource mappings. If an arc points directly from a task to a schedulable entity (e.g. a machine), the cost associated with the arc is denoted by $\gamma_{i,m}^j$ for the i^{th} task of job j and machine m. This cost is specific both to the task ($\mathbf{T}_{j,i}$) and the machine ($\mathbf{M}_m$). All running tasks

have a direct arc to the resource they run on, which carries a cost of $\gamma^j_{i,m}$. In many cost models, $\gamma^j_{i,m}$ is discounted by a multiple of a task's current runtime, θ^j_i, both to control hysteresis and to ensure that finite 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, and applies to the task's arc to the unscheduled aggregator vertex for job j ($\mathbf{U}_j$). 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 starting up and any further wait times incurred when the task was preempted. The value is scaled by a constant *wait time factor*, ω, which increases the cost of keeping the tasks waiting for longer ($\omega > 1$).

Mapping changes due to preemption. Once a task is scheduled on a machine, the scheduler may subsequently decide to *preempt* it by routing its flow through the unscheduled aggregator vertex. It may also choose to *migrate* the task by routing its flow through a different machine vertex; this implies a preemption on the original machine. However, Firmament cost models can disallow preemption – in this case, all arcs apart from the 1:1 mapping to the resource a task is scheduled on are removed, thus "pinning" the task.

Many tasks have similar characteristics, and many resources can likewise be treated as similar by the scheduler. Firmament relies on this insight to generalise Quincy to many scheduling policies: the following section introduces equivalence classes, which are custom aggregators that enable the construction of higher-level cost models.

6.2.4 Equivalence classes

Unlike Quincy, Firmament classifies both tasks and resources (e.g. machines, CPU cores) into *equivalence classes*. An equivalence class contains 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 therefore allow properties of aggregates to be expressed concisely. For example, tasks in task equivalence class c_t may work particularly well on machines in equivalence class c_m. Thus, Firmament creates a vertex that all tasks in c_t connect to, and connects it to a

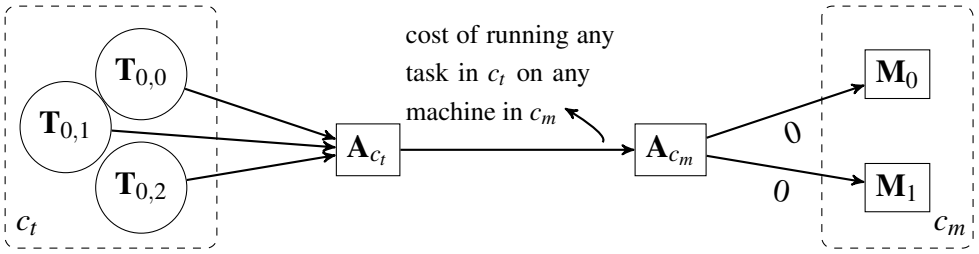


Figure 6.4: Preference of tasks in c_t for machines in c_m expressed via equivalence class aggregators ($\mathbf{A}_{c_t}$ and $\mathbf{A}_{c_m}$). Dashed lines delineate equivalence classes.

vertex that has arcs to all machines in c_m. An arc with a low cost between these vertices now expresses the desired property (Figure 6.4).

The use of equivalence class aggregators reduces the number of arcs required to express such properties 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 7.2 and 7.3.

Tasks. A task can have multiple equivalence classes, with 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 using 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.

Typically, the cost model uses a deterministic hash function to combine information such as the task's job, its binary and arguments, its inputs, and its parent task, to yield a set of task equivalence class identifiers.

Resources. Equivalence classes can also aggregate over resources. Most commonly, they aggregate machines, but other resources can be aggregated similarly. In fact, the cluster aggregator ($\mathbf{X}$) and rack aggregators ($\mathbf{R}_r$) in Quincy are implemented as resource equivalence classes in Firmament.

A machine equivalence class is usually a function of the machine's architecture. Some Firmament cost models, for example, generate the machine equivalence class identifier by hashing machine meta-data, e.g. the machine's micro-architectural topology (§6.5.2).

6.3 Scheduling policies

While Quincy supports only a single policy based on data locality and preemption, my work generalises its approach to other policies. Firmament achieves this by changing the flow network's structure and its cost and capacity parameters. In this section, I explain how three key policy elements – placement constraints, fairness, and gang scheduling – can be modelled, and how this makes Firmament more expressive than Quincy.

Like most cluster schedulers, Firmament supports placement preferences and both job-level and task-level constraints. Section 6.3.1 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 data centres [SKA+13, §3.4]. In Section 6.3.2, I explain the notions of fairness that Firmament supports.

When tasks operate in tight synchronisation, they may require gang scheduling in order to avoid wasting resources. I show in Section 6.3.3 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 6.3.4 discusses policy elements that are challenging to accommodate, specifically: combinatorial constraints, and global invariants.

6.3.1 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 Firmament's cost models described in Chapter 7 have similar notions of placement preferences.

Quincy considers locality preferences only, but 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 a 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.[4] 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 appear, 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, Firmament "drip-feeds" a job's tasks into the flow network one at a time. Each time, the task's complex constraints are adjusted based on previous decisions. However, the scheduler may decide to invalidate a premise ($\equiv$ a prior assignment), which may require re-visiting it in the following round. As a consequence, progress is not guaranteed.

Such reactive, multi-round scheduling increases the scheduling delay for tasks with complex constraints. This tends to be the case in current clusters, too: at Google, for example, tasks with complex constraints take up to $10\times$ longer to schedule [SCH+11].

Generally, the higher the out-degree of a constraint's target vertex, the more likely the constraint is respected and the less it affects the scheduler runtime. This is the case because fine-grained constraints introduce additional arcs, and the min-cost flow optimisation runtime is typically proportional to the number of arcs in the flow network (see §6.4.2). Equivalence classes (§6.2.4) 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 all tasks in a job. In the next section, I explain how Firmament enforces fairness guarantees using the flow network; Section 6.3.3 looks at gang scheduling.

6.3.2 Fairness

Like Quincy, Firmament supports notions of fairness that partition the total number of running tasks between users. These notions of fairness are 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. §6.2.2 and Appendix C.2). The number of tasks allowed

[4] This only works if there is a ceiling on the growth of ωv_i^j, which is true for practical implementations.

per fair share for job j is A_j. By setting $E_j = A_j = F_j$, fair shares are strictly enforced; other configurations with $E_j < A_j < F_j$ give the scheduler more freedom to work towards the fair share over time and while respecting costs.

In the latter case, Firmament (like Quincy) experiences transient periods of unfairness. While many schedulers rely on task churn to converge to fair shares [GZH+11; HKZ+11; OWZ+13; BEL+14], Quincy and Firmament also preempt tasks. However, unfairness may still occur if there are no tasks eligible for preemption.

In the following, I show how Firmament extends Quincy's original notion of bounded unfairness [IPC+09, §1, §2.4, §5.2] to support max-min fair assignments.

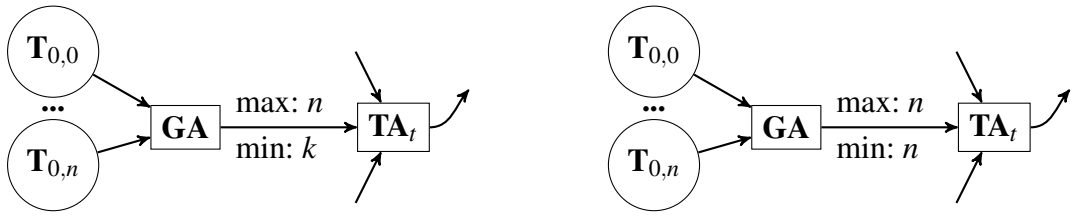
Max-min fair policies. The computation of the fair shares (A_j) can follow a max-min fair allocation, but the flow network optimisation does not guarantee that *preferred* resources (e.g. those resources pointed to by a task's preference arcs) are split in a max-min fair way. As noted by Ghodsi *et al.* [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 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, Firmament might miss opportunities to schedule tasks without violating the fair share.

Fortunately, heavy-handed enforcement of complex fair sharing policies is not typically required in data centres (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.

6.3.3 Gang scheduling

Some 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).



(a) Relaxed, k-of-n.

(b) Strict, gang-of-n.

Figure 6.5: Gang scheduling requests are expressed in the flow network by a gang aggregator (**GA**) and an extra arc with appropriate lower and upper bounds on its flow capacity.

Min-cost flow-optimisation schedulers cannot trivially express gang-scheduling constraints because they require dependent costs (cf. §6.3.1): the cost of a feasible assignment is infinite if even one of the other tasks in the gang remains unscheduled.

However, the flow network can implement gang scheduling by using arc *capacities* to force a group of tasks to schedule. This approach supports both relaxed gang scheduling policies, e.g. one that requires at least k out of n tasks to be placed (as in KMN [VPA+14]), and a strict one which requires *all* tasks in a gang to schedule.

Figure 6.5 shows how this is works, both for relaxed, k-of-n gang scheduling (Figure 6.5a) and strict, all-n gang scheduling (Figure 6.5b):

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.[5]

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, $\mathbf{TA}_t$, 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 $0 < k \leq n$ (relaxed) or n (strict), and the upper bound to n.

Since the lower bound forces the new arc to have a flow of at least k, it constrains the acceptable solutions to the minimum-cost, maximum-flow problem. Namely, only solutions in which *at least* k tasks are scheduled are possible. If $k = n$, this enforces strict gang-scheduling of all n tasks.[6]

There is one downside to this implementation of 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 whose cost to schedule is lower. Gang-scheduling must therefore be used with a degree of care. It can be limited to high-priority jobs, or gang-scheduled decisions can be subject to admission control to avoid excessive disruption.

[5] The removal of other arcs is not strictly required; however, they are guaranteed to receive no flow.

[6] However, if the gang-scheduled tasks exceed the available resources, the solver will fail to find a solution.

6.3.4 Limitations

Firmament's global perspective on the scheduling problem allows it to weigh multiple concerns against each other, and to flexibly support 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 concurrent decisions. As a result, dependent costs cannot be expressed using straightforward arcs: costs cannot be conditional. For example, it is not possible to prevent tasks from the same job from scheduling on the same machine.

Global invariants. Arc capacities can only enforce bounds on the number of tasks that can schedule through them. If an arc has capacity, *any* task that can reach it can send flow through it, even if the assignment violates a global invariant that is inexpressible via task counts. For example, it is not possible to fairly divide aggregate network bandwidth using arc capacities.

Multi-dimensional capacities. Capacities in the flow network are single-dimensional integers and each unit of flow is atomic. This makes multi-dimensional resource requests (e.g. containing CPU, RAM, and I/O bandwidth requests) difficult to model directly.

All three limitations can, however, be mitigated using two techniques:

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 §6.3.1).

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 with sufficient capacity.

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.

6.4 Scalability

Firmament's suitability for large data centres depends on the performance of the underlying min-cost flow optimisation. The minimum-cost, maximum-flow optimisation problem is a computationally intensive problem, and solving it for large graphs requires non-trivial optimisation algorithms. I explain the problem for reference in Appendix C.1.1; this section assumes basic familiarity with the relevant terminology.

In the following, I survey minimum-cost, maximum-flow algorithms and discuss their known complexity bounds (§6.4.1). I then show that, in practice, the scheduling problem sees much better runtimes than the worst-case bounds suggest, but that the algorithm used by Quincy fails to scale to large clusters. However, it turns out that scalability can be much improved by using a seemingly inefficient algorithm in the common case (§6.4.2). Firmament's min-cost flow solver automatically uses the fastest algorithm, and incrementally optimises the solution if only minor changes to the flow network have occurred (§6.4.3).

In Section 8.4, I show that these techniques allow Firmament to scale to very large clusters at sub-second decision times.

6.4.1 Algorithms and solvers

Naïve algorithms for solving the minimum-cost, maximum-flow problem have exponential complexity, but several algorithms with polynomial and strongly polynomial complexity exist. 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 is beyond the scope of this dissertation, I give a high-level overview of recent algorithms in the following.[7]

Table 6.3 lists recent key algorithms and their worst-case time complexities. In practice, algorithm performance depends both on the flow network structure and the arc costs.

Cycle cancelling. Originally due to Klein [Kle67] and of exponential complexity, cycle cancelling algorithms are the simplest minimum-cost, maximum-flow algorithms. They start from a feasible flow (obtained via a maximum-flow computation), and use the residual network to iteratively cancel negative cost cycles by sending flow along them. Cancelling these cycles removes arcs from the residual network; once it is depleted, the optimal solution has been found. A strongly polynomial minimum-mean cycle cancelling algorithm exists [GT89], but it is not competitive with other algorithms 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

[7] A more extensive overview can be found in §1.3 and §3.2–3.5 of Adam Gleave's Part II dissertation, written under my supervision [Gle15].

Algorithm	Worst-case complexity	Reference
Minimum mean-cost cycles (Tardos, 1985)	$O(V^2 \text{polylog}(E))$	[Tar85]
Relaxation (Bertsekas and Tseng, 1988)	$O(E^3 C U^2)$	[BT88a]
Cycle cancelling (Goldberg and Tarjan, 1989)	$O(VE^2CU)$	[GT89]
Successive approximation (Goldberg and Tarjan, 1990)	$O(VE\log(VC)\log(\frac{V^2}{E}))$	[GT90]
Premultiplier network simplex (Orlin, 1997)	$O(\min(VE\log(VC), VE^2\log(V)))$	[Orl97]
Scaling push-relabel (Goldberg, 1997)	$O(VE\min(\log(VC), E\log(V))\log(\frac{V^2}{E}))$	[Gol97]

Table 6.3: 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, U is the maximum capacity, and C the maximum cost.

a polynomial worst-case bound. 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 (ε-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. Appendix C.1.2 explains the algorithm in more detail. In practice, cost-scaling performs consistently across many different flow networks because its successive refinement avoids exploring partial solutions that turn out to be infeasible.

Relaxation. The relaxation algorithm is based on Lagrangian relaxation [AMO93, ch. 16], and works on the dual version of minimum-cost, maximum-flow. It decouples improvements in feasibility from cost reductions, and prioritises reducing cost over improving feasibility. Its worst-case time complexity is cubic in the number of edges, substantially worse than most other algorithms. However, it turns out to work well for many Firmament flow networks (§6.4.3).

6.4.2 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.

Scalability was not a primary concern for Quincy, which targeted clusters of hundreds of machines. The authors evaluated it on a 243-node cluster and found an optimisation runtime of a few milliseconds [IPC+09, §6]. In a simulated 2,500-machine deployment, they found that the optimisation still completes in "a little over a second" [IPC+09, §6.5]. Industrial data centres, 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,[8] I found that the solver runtime is super-linearly proportional to the number of arcs in the flow network. This is far better than the theoretical worst case of $O(VE \min(\log(VC), E\log(V))\log(\frac{V^2}{E}))$, but nevertheless poses a noticeable overhead at large scale, especially for workloads of many short tasks. Short tasks can occur e.g. when online transaction processing (OLTP) or interactive data analytics workloads are scheduled. Upwards of 10,000 machines, scheduling with `cs2` and using the Quincy cost model and a Google workload trace [RTG+12] takes over 4 seconds in the median, and up to 70 seconds in the worst case.

There are several ways in which scalability could, in principle, be improved:

1. **Partition the problem** by having multiple Firmament coordinators arranged in a tree (§6.5.1), each responsible for scheduling a smaller subset of the overall data centre (e.g. a few racks each). However, a partitioned approach loses the global optimality of placements, as each subordinate scheduler can only schedule tasks on 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 must routinely work with imperfect data, it may not be necessary to find the optimal task assignments – an assignment "close enough" to the optimal solution may be sufficient.

3. Compute an **incremental solution** by re-using the prior solution and solver state. In a large data centre, only a small fraction of tasks and machines change state 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 accumulated changes shrinks as the solver completes faster, which in turn reduces the work required in the next iteration.

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 gradually, but rather by leaps [Gle15, §4.4; GSG+16, §5.1]. However, scalability can indeed be much improved by using the relaxation algorithm instead of cost-scaling, and by solving the problem incrementally.

[8] `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`.

6.4.3 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 fairly small. Instead of running the solver from scratch each time, maintaining state across runs and starting from a previous solution can accelerate it substantially.

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 changes to the flow network created by these events reduce to three types of change:

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 has 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 task assignments 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 three ways of optimising the flow network incrementally:

1. An incremental version of Goldberg's cost-scaling push-relabel algorithm [Gol97] using the `flowlessly` solver.[9] 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 often outperforms cost-scaling in practice on the flow networks generated by Firmament, although it can suffer from pathological runtime in edge cases that involve highly-contended vertices. `flowlessly` side-steps these issues by running an instance of incremental cost-scaling alongside relaxation, and by picking the solution of the first algorithm to complete.

[9] The `flowlessly` solver was implemented by Ionel Gog and includes both a from-scratch and an incremental cost-scaling push-relabel algorithm implementation.

3. Finally, Firmament also supports a modified version of the relaxation-based RELAXIV solver [BT88b; BT94], which performs incremental relaxation.[10]

Incremental cost-scaling yields about a 2× runtime reduction compared to running from scratch. The incremental relaxation achieved higher speedups (up to 14×) in exploratory experiments, but in subsequent work, I have found that most of the speedup is due to the relaxation algorithm being a good fit for Firmament's flow networks and likewise applies to running relaxation from scratch [GSG+16].

Nevertheless, the combination of relaxation and incremental cost-scaling results in tangible solver runtime reductions: in Chapter 8.4, I show that Firmament achieves sub-second scheduling latency for a Google-scale data centre and workload.

6.5 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 implements 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 6.5.1.

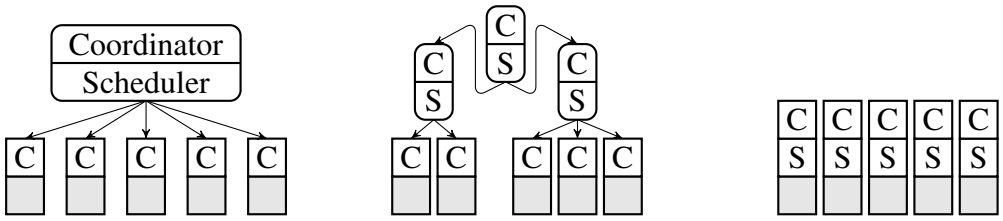
Firmament extracts each machine's micro-architectural topology and represents it as part of the flow network to permit scheduling decisions at CPU granularity. Section 6.5.2 explains how the topology information is obtained and used.

In order to make good scheduling decisions at fine granularity, Firmament collects information about workloads through automatic profiling. Section 6.5.3 describes the methods used to extract the information, and how it is stored and aggregated.

6.5.1 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.

[10] The modifications to RELAXIV 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].



(a) Centralised. **(b)** Hierarchical. **(c)** Distributed.

Figure 6.6: Contrasting (a) single, (b) delegating, and (c) fully distributed Firmament deployment setups; showing **C**oordinators and **S**chedulers.

Each machine runs a user-space Firmament **coordinator** process. Coordinators may be arranged in parent–child relationships, where parents schedule tasks on their children by *delegating* the tasks. As each coordinator runs a scheduler, their nested hierarchy forms a tree of schedulers.[11]

However, the flip side of permitting delegation is that coordinators' schedulers sometimes make decisions based on stale local state. Any task delegation to a remote coordinator is optimistically concurrent with other decisions, and may fail when racing with the target or with another coordinator. Omega [SKA+13], Apollo [BEL+14], and Tarcil [DSK15] take similar approaches, which could, at worst, lead to many failed delegations. However, Firmament's coordinators cannot deadlock: one of the racing coordinators always succeeds, and others back off.

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 6.6a). 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 6.6b). Job submissions can be load-balanced over these coordinators, which either schedule tasks directly to local resources, or on resources attached to subordinate coordinators.

(c) **Fully distributed scheduling** is possible if jobs are dispatched to individual machine coordinators (Figure 6.6c). This can be used to implement policies akin to Sparrow's fully distributed operation [OWZ+13].

Figure 6.7 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.

[11] This can also be a DAG if the children multi-cast each message to several parents.

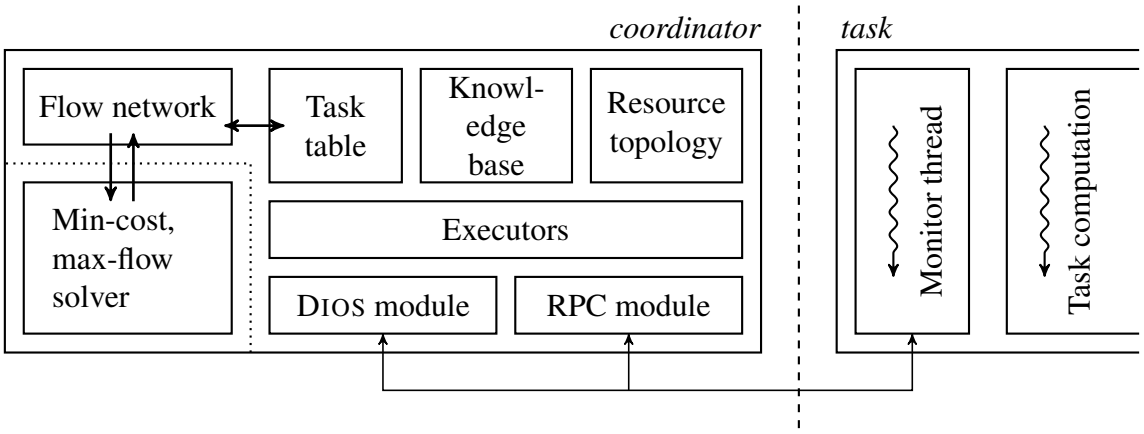


Figure 6.7: High-level structure of the Firmament coordinator.

6.5.2 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 uplinks).

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 shared CPU caches, as well as NUMA and simultaneous multi-threading (SMT) topologies.

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 data centre machines' resources in it. This explicit representation of machine resources not only allows Firmament to make more fine-grained placement

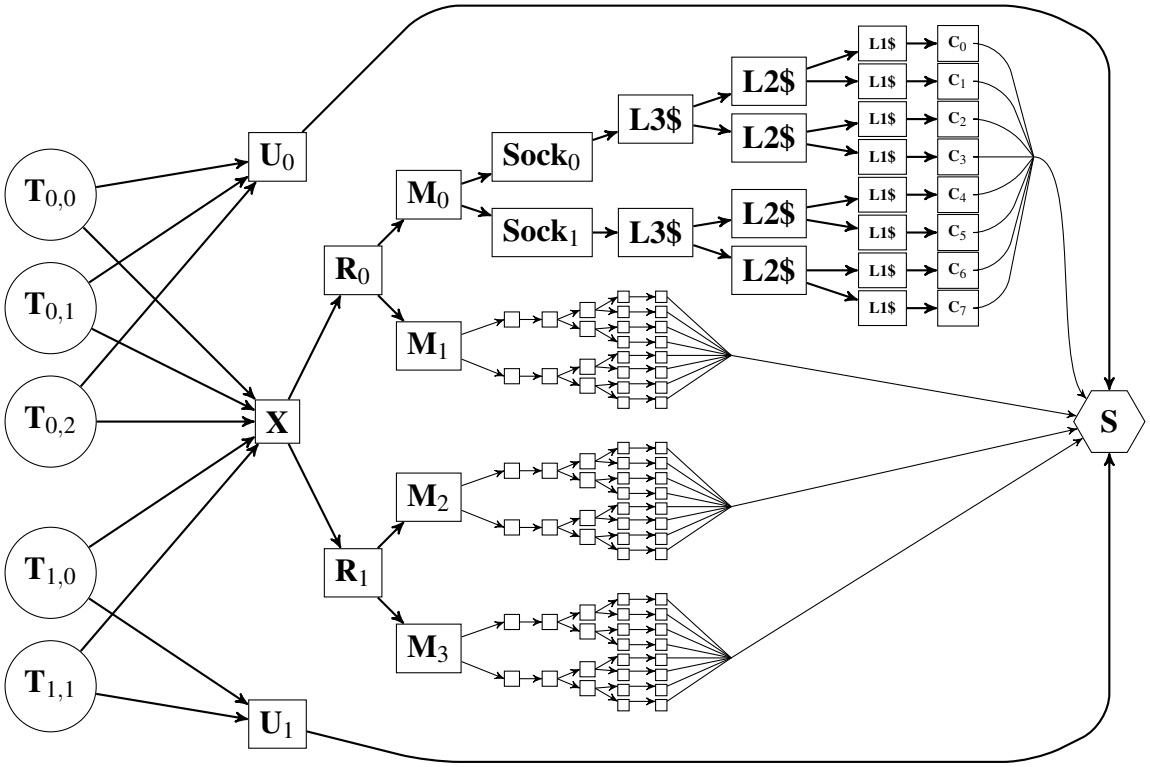


Figure 6.8: Example Firmament flow network (cf. Figure 6.1) with added machine topology for four eight-core machines. Preference arcs and running task arcs are not shown.

decisions, it also offers a way of representing heterogeneity in machine types and architectures to the scheduler.

Figure 6.8 shows the example flow network from Figure 6.1 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 (§6.2.4) in order to restrict the number of arcs from tasks to resources.

6.5.3 Task profiling

The Firmament coordinator on each machine in the cluster 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 in the **knowledge base** coordinator module, where cost models may access it to improve future decisions.

Firmament acquires this information 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.

Metric	Type	Source
Virtual memory size	sampled	`procfs`
Real memory size	sampled	`procfs`
Time runnable and scheduled	sampled	`procfs`
Time runnable and waiting	sampled	`procfs`
Total runtime	aggregate	timer
Total cycles	aggregate	HW counter
Total instructions	aggregate	HW counter
LLC references	aggregate	HW counter
LLC misses	aggregate	HW counter
Cycles per instruction (CPI)	aggregate	calculated
Memory accesses per instruction (MAI)	aggregate	calculated
Instructions per memory access (IPMA)	aggregate	calculated

Table 6.4: Metrics tracked by the Firmament coordinator for each task running on the local machine, in addition to used and available resources.

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.

Table 6.4 lists the precise set of metrics tracked in the current Firmament implementation. Metrics are measured either by continuous sampling during task execution, or by retrieving an aggregate summary when a task completes. The latter approach is useful to determine aggregate metrics over the task's lifetime (e.g. its total CPU migration count), but works less well for long-running service tasks, since it may take a very long time until the metrics are reported for such tasks. Either collection strategy, however, only adds negligible overhead to tasks' runtime.

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 cluster 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 [MVH+10].

Firmament similarly collects performance counter information, but also aggregates it across many machines. This allows Firmament to build a profile specific to each task, its combination with a machine type, and the set of co-located tasks. As many tasks often perform similar work in parallel, Firmament uses its equivalence classes to combine their performance metrics. Since the cost model controls which equivalence classes a task is part of, metrics can be flexibly aggregated along different dimensions in this way. Based on the collective performance metrics received for all tasks in an equivalence class, Firmament rapidly establishes an accurate profile.

6.6 Summary

In this chapter, I contrasted cluster scheduling for data centres with single machine CPU scheduling, and illustrated how my Firmament cluster scheduler draws on both (§6.1).

Firmament is unusual because it maps the scheduling problem to a minimum-cost, maximum flow optimisation – an approach only taken in one prior system, Quincy (§6.2). Firmament generalises the flow network optimisation approach by customising network structure, and hence expresses many scheduling policies (§6.3). Moreover, Firmament substantially improves the scalability of the flow network optimisation approach to scheduling: it relies on multiple minimum-cost, maximum-flow algorithms, and applies optimisations to improve their performance (§6.4).

Finally, I described the Firmament architecture, and how it obtains the fine-grained profiling information and makes it available to scheduling policies (§6.5).

In the following, Chapter 7 demonstrates Firmament's flexibility via four different cost models, covering a wide range of policies that Firmament can express.

In Chapter 8, I evaluate Firmament using both real clusters and a trace from a Google data centre, and find that it compares favourably to other systems and scales well.

Chapter 7

Firmament case studies

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. I first briefly introduce the original Quincy cost model (§7.1; 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 (§7.2);
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 (§7.3); 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 (§7.4).

In Section 7.5, I summarise and contrast the properties of the four cost models described.

Chapter 8 will evaluate these cost models for a range of workloads and clusters. Other cost models are, of course, also conceivable, and Section 8.3 will sketch how existing schedulers' policies can be translated into Firmament cost models.

7.1 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.

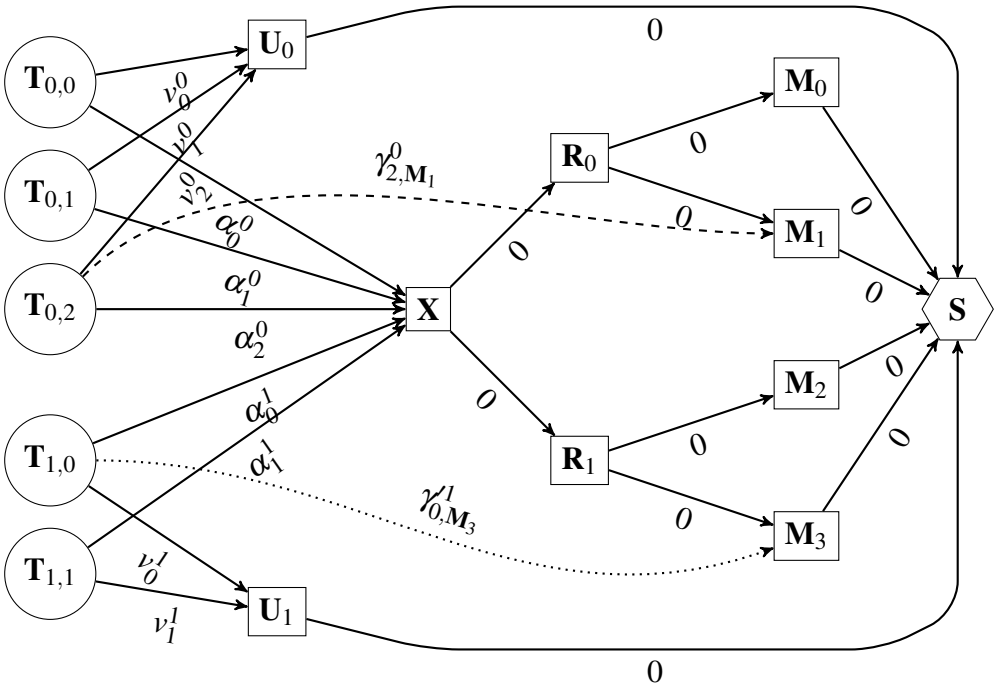


Figure 7.1: The example flow network in Figure 6.1 with costs according to the Quincy cost model added to the arcs. The capacities are assigned as in Figure 6.2.

Figure 7.1 shows the flow network structure and cost terms in Quincy. The cost to the unscheduled aggregator, v_i^j is proportional to the task's wait time, while the cost to the unscheduled aggregator, α_i^j, 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 $\mathbf{R}_l$ has cost $\rho_{i,l}^j$ proportional to the worst-case data transfer within the rack, and an arc to a machine $\mathbf{M}_m$ has cost $\gamma_{i,m}^j$ 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 ($\mathbf{X}$) and the rack aggregators ($\mathbf{R}_l$) as resource equivalence classes, and there are no task equivalence classes in the Quincy cost model.

7.2 Whare-Map cost model

The *Whare-Map* system by Mars and Tang avoids negative interference and exploits machine heterogeneity in WSCs [MT13].[1] 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 data centre 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.

[1] The same work was previously described in Mars's 2012 PhD thesis under the name SmartyMap [Mar12].

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 data centre 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 (§6.2.4) 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 machine type;
3. **Whare-M**, which uses machine type affinities, but does not consider co-location interference at all; and
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.

7.2.1 Flow network structure

Figure 7.2 shows an example Firmament flow network for both Whare-Map cost models.

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 ($\mathbf{MA}_{c_m}$ for machine type c_m in Figure 7.2).

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 $nt + m$ arcs. The number of distinct machine types in the cluster, n, is usually on the order of a few dozen (cf. §2.1.2). 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 ($\mathbf{TA}_{c_t}$ for equivalence class c_t). For u different task equivalence classes, using

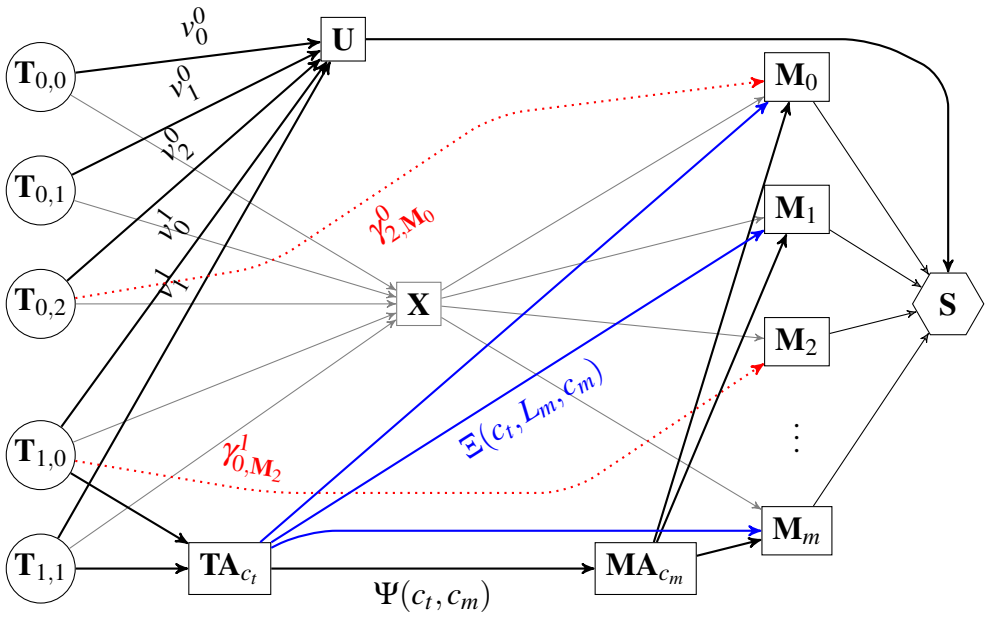


Figure 7.2: Firmament flow network with the Whare-Map cost models. Blue arcs are only present in Whare-MCs, while black ones are in both Whare-M and Whare-MCs. Red, dotted arcs correspond to already running tasks. Note that Whare-Map has no rack aggregators and this graph uses a single unscheduled aggregator (**U**), rather than per-job ones.

Parameter	Edge	Meaning
v_i^j	$\mathbf{T}_{j,i} \to \mathbf{U}_j$	Cost of leaving $\mathbf{T}_{j,i}$ unscheduled.
α_i^j	$\mathbf{T}_{j,i} \to \mathbf{X}$	Cost of scheduling in the worst possible location.
$\gamma_{i,m}^j$	$\mathbf{T}_{j,i} \to \mathbf{M}_m$	Cost of continuing to run on machine $\mathbf{M}_m$.
$\Psi(c_t, c_m)$	$\mathbf{TA}_{c_t} \to \mathbf{MA}_{c_m}$	Cost of running task of type c_t on a machine of type c_m.
$\Xi(c_t, L_m, c_m)$	$\mathbf{TA}_{c_t} \to \mathbf{M}_m$	Cost of running task of type c_t with co-located tasks in L_m.

Table 7.1: Cost parameters in the Whare-Map cost models and their roles.

aggregators requires $u+n$ vertices and $O(t+un+m)$ arcs, with $t \gg m \gg u \geq n$. Since $t+un \ll nt$, this approach scales better than using machine-type aggregators only.

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 ($\mathbf{TA}_{c_t}$) and a machine type aggregator ($\mathbf{MA}_{c_m}$), as shown in Figure 7.2.

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.[2]

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 ma-

[2]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.

chine. 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 $\mathbf{TA}_{c_t}$ 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 $\mathbf{TA}_{c_t}$ is equal to the minimum of the number of incoming arcs into $\mathbf{TA}_{c_t}$ 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, $\mathbf{X}$.

7.2.2 Cost assignment

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. §6.5.3). As the available solvers require integral costs, I further normalise this metric to picoseconds-per-instruction (psPI).[3]

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_i^j is the cost of leaving the i^{th} task in job j unscheduled. It is proportional to the wait time ν_i^j, and lower-bounded by the average Whare-Map score, with $\mathbf{T}_{j,i} \in c_t$:

$$v_i^j = \begin{cases} \max(\nu_i^j, \overline{\Psi(c_t, c_m)}) & \text{for Whare-M} \\ \max(\nu_i^j, \overline{\Xi(c_t, L_m, c_m)}) & \text{for Whare-MCs} \end{cases}$$

- α_i^j 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

[3] 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 = 1/400, e.g. remote DRAM-bound work).

suited towards task $\mathbf{T}_{j,i}$ (Whare-M) or, the machine where the task will experience the worst possible interference (Whare-MCs). In other words, for $\mathbf{T}_{j,i} \in c_t$:

$$\alpha_i^j = \begin{cases} \max_{c_m}(\Psi(c_t, c_m)) & \text{for Whare-M} \\ \max_{L_m, c_m}(\Xi(c_t, L_m, c_m), \Psi(c_t, c_m)) & \text{for Whare-MCs} \end{cases}$$

- γ_i^j is the cost of running $\mathbf{T}_{j,i}$ 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 $\mathbf{T}_{j,i}$.

7.2.3 Summary

The Whare-Map cost models implement the policies of the published system [MT13], with one key difference: instead of using the approximate stochastic hill-climbing approach, the Firmament version is based on an exact minimum-cost flow optimisation. However, several improvements to the cost model are conceivable; I summarise them below.

Model machine load. The co-location-aware arcs ($\mathbf{TA}_{c_m} \to \mathbf{M}_m$) implicitly reflect the load on their target machines (since load-proportional interference increases the cost), but the arcs from machine type aggregators to machines ($\mathbf{MA}_{c_m} \to \mathbf{M}_m$) have a zero cost.

In order to bias scheduling towards 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 at machine granularity (L_m). However, Firmament routinely extracts the micro-architectural topology of each machine (§6.5.2).

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 be specific to the tasks running on CPU cores that share caches with the target.

My next Firmament cost model, the coordinated co-location model, incorporates these ideas in a more advanced cost model that also supports resource reservations.

7.3 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 (§6.2.4), 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.

7.3.1 Admission control

To meet the strict resource fit property, CoCo must match tasks to resources such that *no task can be scheduled 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 naïve approach would instead add arcs to all suitable machines, but 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. Capping the number of preference arcs per task would solve this, but loses optimality since the missing arcs restrict the possible solutions.

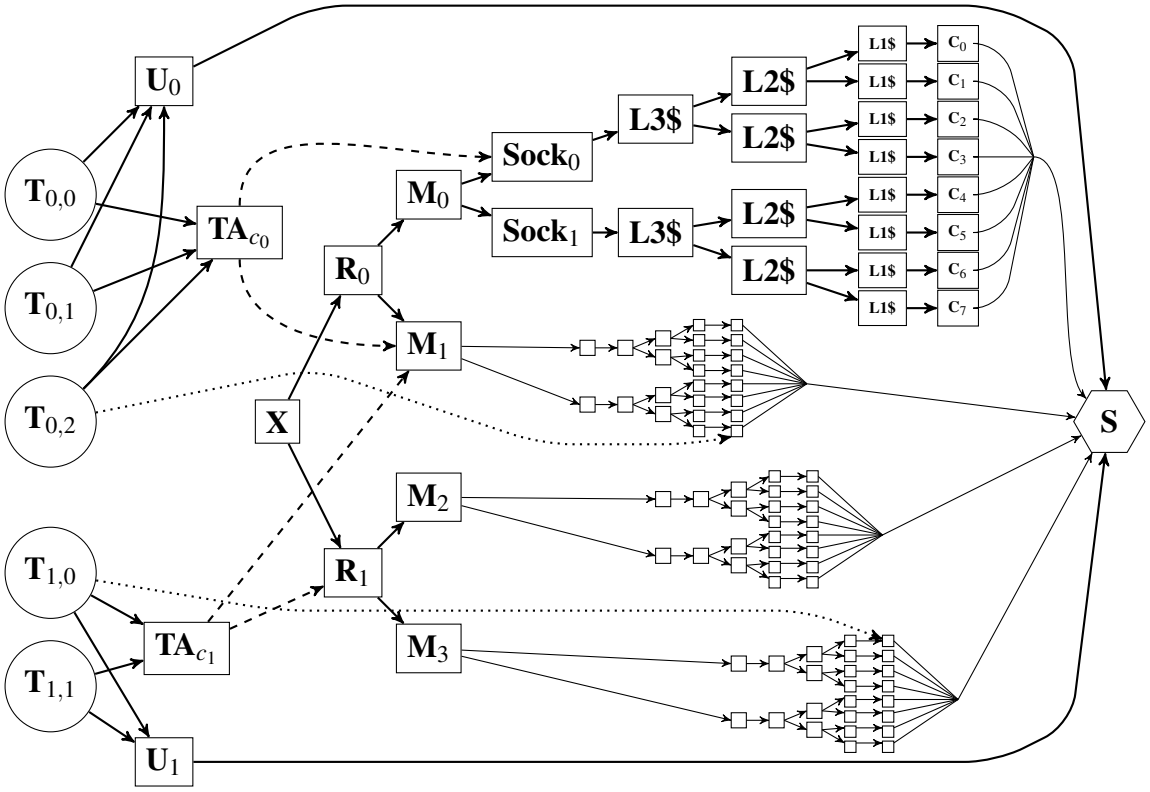


Figure 7.3: Firmament flow network for the CoCo cost model. Each job's tasks have one equivalence class, represented by task aggregator $\mathbf{TA}_j$. Dotted arcs indicate tasks already running ($\mathbf{T}_{0,2}$ and $\mathbf{T}_{1,0}$), while dashed arcs point to maximally aggregated subtrees of the resource topology where tasks fit: job 0's tasks fit under socket 0 on $\mathbf{M}_0$ and on $\mathbf{M}_1$, and job 1's tasks fit anywhere in rack $\mathbf{R}_1$ and on $\mathbf{M}_1$.

Instead, I change the structure of the flow network to accommodate CoCo (see Figure 7.3). Each task receives arcs to its task equivalence class aggregators (e.g. $\mathbf{TA}_{c_0}$ and $\mathbf{TA}_{c_1}$). From these aggregators, CoCo adds outgoing arcs to *maximally aggregated subtrees* of the resource topology, which represent locations in which the tasks in the equivalence class definitely fit. Several such subtrees may exist, and each may include multiple machines – consider, for example, a rack in which a task fits on all machines (e.g. $\mathbf{TA}_{c_1}$ on $\mathbf{R}_1$).

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

```
vector<ResourceID_t>* CocoCostModel::GetMaximallyAggregatedSubtrees(
    EquivClass_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 7.1: 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.

minimum and maximum available resources across its children. This is implemented as a simple breadth-first traversal in Firmament (see Listing 7.1), 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 7.1 – 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].
- 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 $\mathbf{TA}_{c_t}$ and the resource aggregate is added, and its cost is set as described in the following.

7.3.2 Cost assignment

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. $\mathbf{T}_{j,i} \to *$), 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** ($\mathbf{T}_{j,i} \rightarrow \mathbf{U}_j$), 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 Ω 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 Ω.

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.[4]

Each leaf of the resource topology supplies a penalty score for each interference class. A low penalty score implies a 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 the penalty scores of its children, 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:

$$cost(u,w) = e^{\frac{t-i}{t}} \times \frac{\sum_{v \in children(w)} cost(w,v)}{num.\ children\ of\ w},$$

   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 $e^{\frac{t-i}{t}}$ 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.

[4]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 (§6.5.3).

The cost specific to the task equivalence class (on the $\mathbf{TA}_{c_t} \rightarrow$ *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 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.

The CoCo cost model supports *priority preemption*, which occurs if a task of higher priority (i.e. lower cost in the priority dimension) displaces another task with lower priority. However, service jobs cannot be preempted by batch jobs, since batch job's lower priority always dominates their cost.

7.3.3 Dynamic flow network changes

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, $\mathbf{M}_1$ in Figure 7.3 may have four idle CPU cores, but may only have sufficient I/O capacity for *either* a task from $\mathbf{TA}_{c_0}$ or a task from $\mathbf{TA}_{c_1}$. With two incoming arcs, however, nothing stops both $\mathbf{T}_{0,0}$ and $\mathbf{T}_{1,1}$ 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.[5]

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.

[5]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.

7.3.4 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 8.2.1, 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.

7.4 Green cost model[6]

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 cluster 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 7.4 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.

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 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

[6]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].

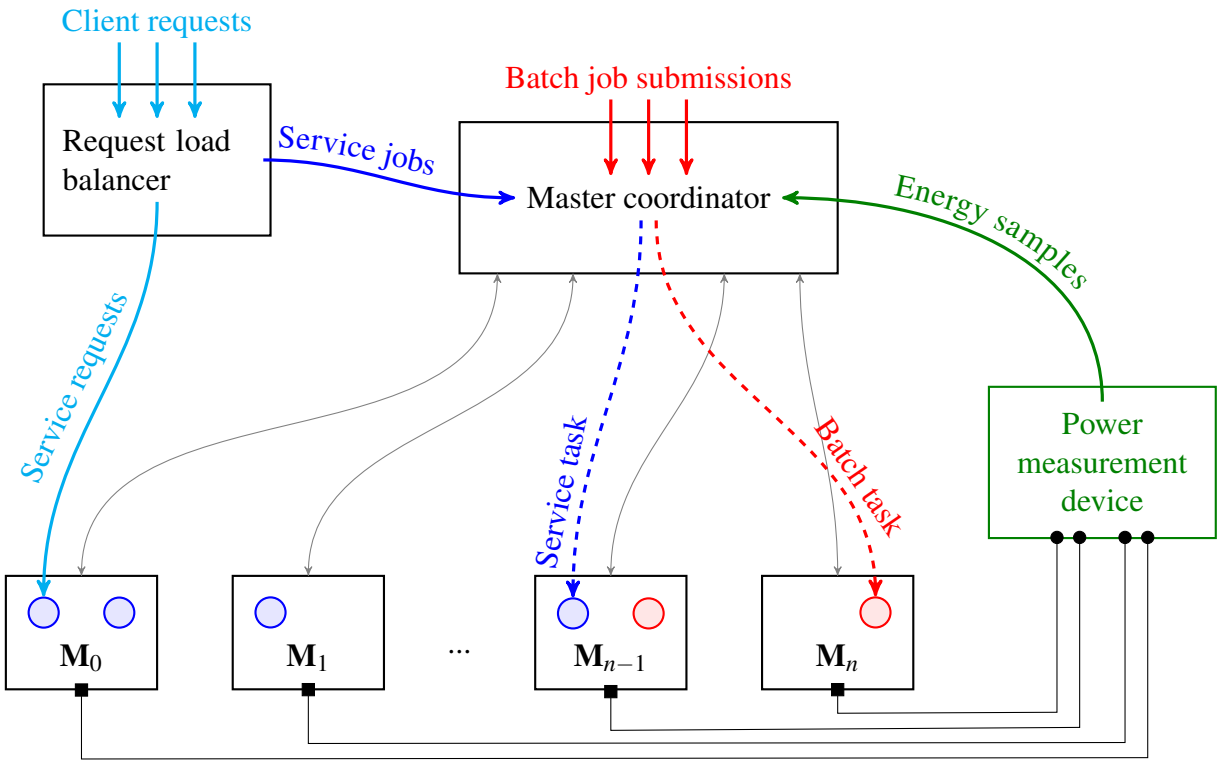


Figure 7.4: Overview of the energy-aware scheduling setup.

balancer. If it is necessary to commission additional tasks in order to meet the relevant service level objective (SLO) – for example, a 99$^{\text{th}}$ percentile 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 cluster 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 linear in the number of cores utilised.[7]

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.

[7]Power consumption attributable to individual tasks could also be measured directly, using, for example, the RAPL interface available in recent Intel CPUs [HDV+12].

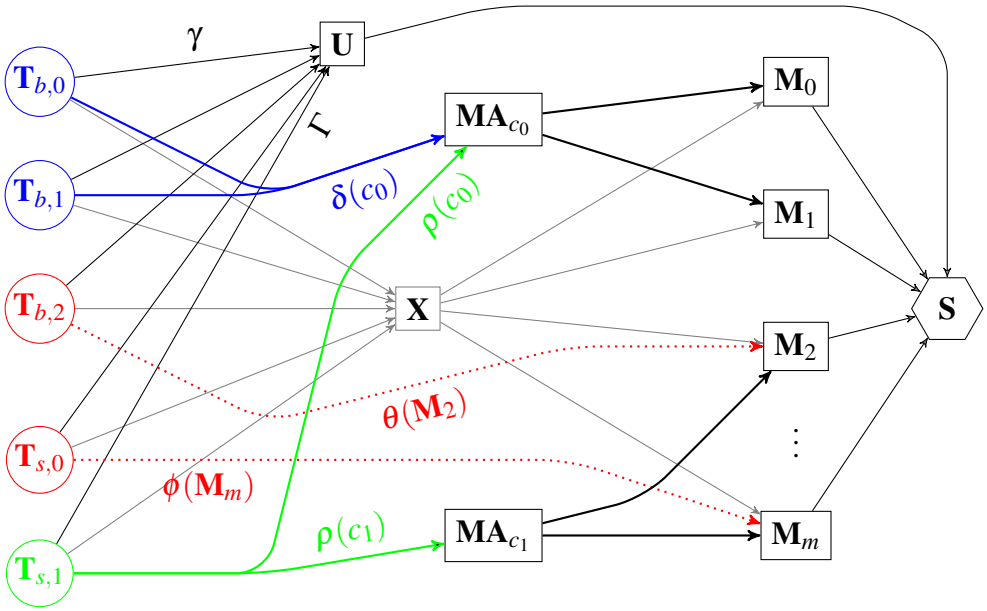


Figure 7.5: Firmament flow network with the Green cost model. One *b*atch and one *s*ervice job are shown, with running, new service, and new batch tasks. Running tasks also have arcs to machine aggregators corresponding to migration options, which are not shown here.

7.4.1 Cost assignment

All costs assigned are in the range $[0, \Omega]$, with Ω being a large, fixed constant. Tasks always have costs on their arc to an unscheduled aggregator ($\mathbf{T}_{j,i} \rightarrow \mathbf{U}_j$), expressed by Γ for service tasks and γ for batch tasks. They also have arcs to aggregator vertices for specific machine type equivalence classes ($\mathbf{MA}_{c_m}$ in Figure 7.5), 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 $\mathbf{M}_k$ with cost $\phi(k)$ (service) or $\theta(k)$ (batch) in addition to other arcs.

A new service task $\mathbf{T}_{j,i}$ is connected to aggregators for each machine equivalence class c_m ($\mathbf{MA}_{c_m}$) at a cost $\rho(k)$ proportional to the per-request energy cost on these machines. The cost of leaving the task unscheduled (Γ) is set to maximum value Ω. Service tasks are thus always scheduled most expediently, and never preempted (although they may migrate). Once a service 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 *schedulable* on a machine if that machine is expected to meet its completion deadline. The maximum cost for a batch task is ω ($\ll \Omega$), and costs are assigned as follows:

Batch task $\mathbf{T}_{j,i}$ schedulable on $\mathbf{M}_k$ of type c_m. $\delta(c_m)$, the cost from the task to aggregator $\mathbf{MA}_t$ is proportional to the energy cost of running $\mathbf{T}_{j,i}$ on a machine in class c_m. When $\mathbf{T}_{j,i}$ is running on $\mathbf{M}_k$, the cost $\theta(k)$ is inversely proportional to the task's completion percentage. The cost of leaving the task unscheduled, γ, is set 15% above the cheapest machine's cost.

Batch task $\mathbf{T}_{j,i}$ schedulable on some machines, but not $\mathbf{M}_k$. $\mathbf{T}_{j,i}$ and $\mathbf{M}_k$ are connected via an "unschedulable" machine type aggregator. The cost from this aggregator to $\mathbf{M}_k$ is ω, as is the cost to $\mathbf{X}$; only arcs to preferred machine type aggregates can thus be used.

Batch task $\mathbf{T}_{j,i}$ only schedulable on $\leq \varepsilon$ machines. The task is considered high priority for scheduling. γ is set to 20× the best machine's cost, giving the task precedence over other tasks.

Batch task $\mathbf{T}_{j,i}$ 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 $\mathbf{T}_{j,i}$ are assigned cost $\delta(c_m)$; all other machines are connected at cost ω 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, e.g. reflecting the cheaper migration enable by fast restart on local or checkpointed state.

7.4.2 Summary

The Green cost model primarily serves to demonstrate Firmament's flexibility. It is a standalone 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 8.2.2, 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.

7.5 Summary

Firmament can support many different cost models. I have outlined four, which Table 7.2 summarises. The CoCo cost model supports the most extensive feature set, but is also the most complex cost model.

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, and expected runtime that Firmament's cost models can express.

Feature	Quincy (§7.1)	Whare-Map (§7.2)	CoCo (§7.3)	Green (§7.4)
Preemption	✓	✓	✓	✓
Data locality	✓	✗	✓	✗
Co-location interference	✗	✓	✓	✗
Multi-dimensional load	✗	✗	✓	✗
Per-core scheduling	✗	✗	✓	✗
Explicit priorities	✗	✗	✓	✗
Energy-awareness	✗	✗	✗	✓

Table 7.2: Cost models supported in Firmament and their features.

There are approaches that cannot easily be expressed in Firmament, however. For example, tetrisched [TZP+16] can express complex constraints (§6.3.4) as algebraic expressions that do not map to Firmament cost models. Likewise, some global multi-dimensional max-min fairness invariants can only be approximated (§6.3.2).

Section 8.3 evaluates in detail which existing schedulers' policies Firmament can support.

Chapter 8

Firmament evaluation

I now turn to evaluating the Firmament cluster scheduler, which I described in Chapters 6 and 7. Firmament is a general cluster scheduler and thus targets a broad range of workloads. 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 three local cluster testbeds (§8.1) 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? (§8.2)
2. How flexibly does Firmament's generalised notion of scheduling as a flow network optimisation adapt to different scheduling policies? (§8.3)
3. How well does Firmament scale to large clusters with thousands or tens of thousands of machines? (§8.4)

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 cluster scheduling, some of which I outline in Chapter 9.

8.1 Experimental setup

All experiments described in this chapter were carried out on one of three local testbeds in the Computer Laboratory. 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

	Type	Machine	Architecture	Cores	Thr.	Clock	RAM
4×	**A**	GW GR380	Intel Xeon E5520	4	8	2.26 GHz	12 GB PC3-8500
2×	**B**	H8SGL-F	AMD Opteron 6168	12	12	1.9 GHz	32 GB PC3-10666
2×	**C**	Dell R420	Intel Xeon E5-2420	12	24	1.9 GHz	64 GB PC3-10666
1×	**D**	Dell R415	AMD Opteron 4234	12	12	3.1 GHz	64 GB PC3-12800
1×	**E**	SM AS1042	AMD Opteron 6168	48	48	1.9 GHz	64 GB PC3-10666

(a) Heterogeneous SRG test cluster.

	Type	Machine	Architecture	Cores	Thr.	Clock	RAM
28×	**M**	Dell R320	Intel Xeon E5-2430Lv2	6	12	2.4 GHz	64 GB PC3-12800

(b) Computer Laboratory model data centre.

Table 8.1: Specifications of the machines in the two x86 evaluation clusters.

from four 2009 Intel "Gainestown" Xeons to a 2012 "Sandy Bridge" Xeon, and cover a range of clock frequencies and memory subsystem architectures (Table 8.1a). 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 8.1b). 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]

The energy-efficiency testbed is an ad-hoc test cluster for the Green cost model case study (§7.4). This cluster consists of two ARM-based machines and a subset of the SRG test cluster machines connected by mixed 100M and 1G Ethernet (further details in §8.2.2).

All x86-based machines run Ubuntu 14.04 (Trusty Tahr) with Linux kernel v3.13.

The SRG test cluster, unlike the model data centre, exhibits heterogeneity similar to a real-world data centre (cf. §2.1.2).

8.2 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 explained in Section 6.2.1, the assignments found are *policy-optimal* for the given cost model.[2]

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 Chapter 7, and measure the quality of its decisions.

[1] 28 machines are certainly small compared to an industrial cluster, but a scale of tens of machines is reportedly representative of many commercial customers' "big data analytics" setups [ORR+15, §2.4].

[2] This does *not* imply general optimality: a better cost model may lead to better assignments.

Benchmark	Description	Workload (jobs × tasks) SRG test cluster	Model DC
`cpu_spin`	Spin CPU for 60s.	4×10	8×10
`mem_stream`, L3-fit	Swap words in 1M array.	4×10	8×10
`mem_stream`, >LLC	Swap words in 50M array.	4×10	8×10
`io_stream`, read	`fio` asynchronous read of 4 GB.	1×5	2×7
`io_stream`, write	`fio` asynchronous write of 4 GB.	1×5	2×7
Total tasks (% of CPU cores utilized)		130 (79.3%)	268 (79.8%)

Table 8.2: Synthetic workloads used in cluster mix experiments.

8.2.1 Case study: avoiding co-location interference

The proactive avoidance of workload interference (§2.1.3) was a key motivating use case for Firmament. To this end, I implemented two interference-aware cost models: a Whare-Map cost model (§7.2) and the CoCo cost model (§7.3), 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 8.2). 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: completed jobs are resubmitted at most ten seconds after they finish. Hence, Firmament over time acquires task profile information for each equivalence class (cf. §6.5.3), similar to real-world cluster managers [ZTH+13].

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 [GZH+11], but does not explicitly consider co-location interference.[3]

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 §7.2). Zhang *et al.* showed that IPS are strongly correlated with

[3] As 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 resource manager. The Mesos paradigm allows a scheduler framework to be aware of co-location within its resource offers; none of the existing top-level frameworks support this, however.

the application-level performance of Google workloads [ZTH+13, §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 (§7.3).

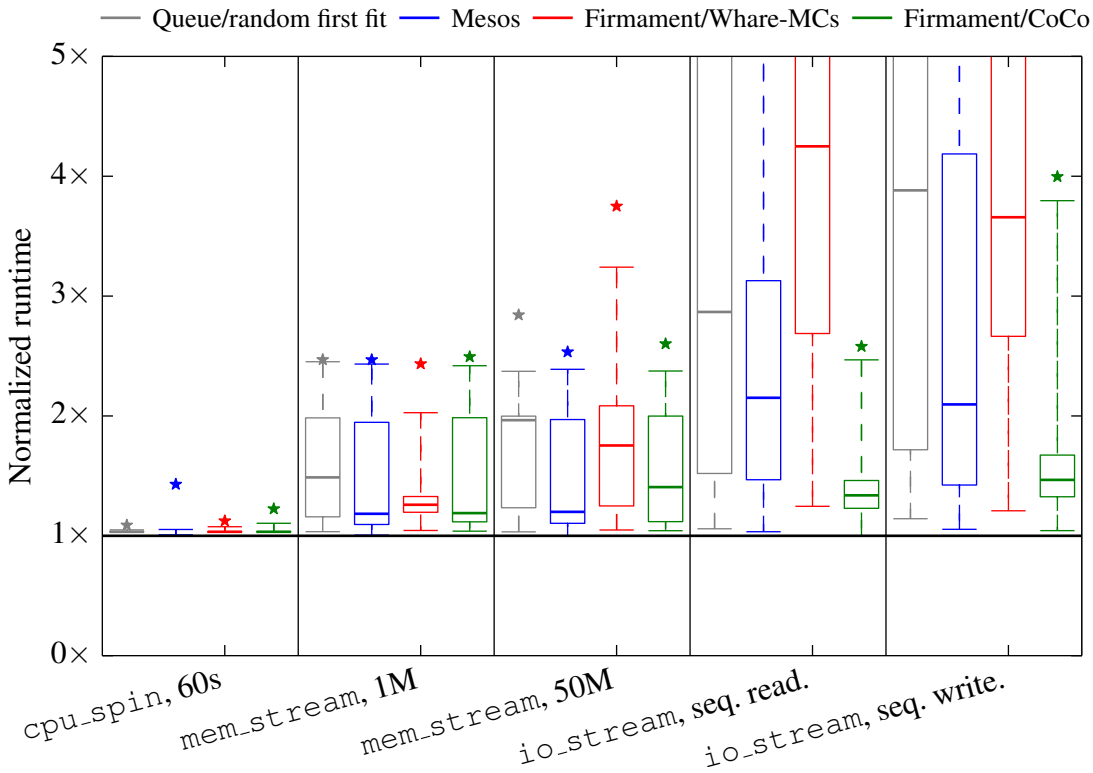
Metric. The goal of this experiment is to measure performance unpredictability due to interference. I quantify it using the *normalised task runtime* relative to the ideal runtime. To obtain the ideal, I first executed a single task of each workload on an idle machine of each type (Table 8.1) without 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.

Results. Figure 8.1 shows the distribution of per-task runtimes as a box-and-whisker plot for the heterogeneous SRG test cluster (Figure 8.1a) and the homogeneous model data centre (Figure 8.1b). 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.

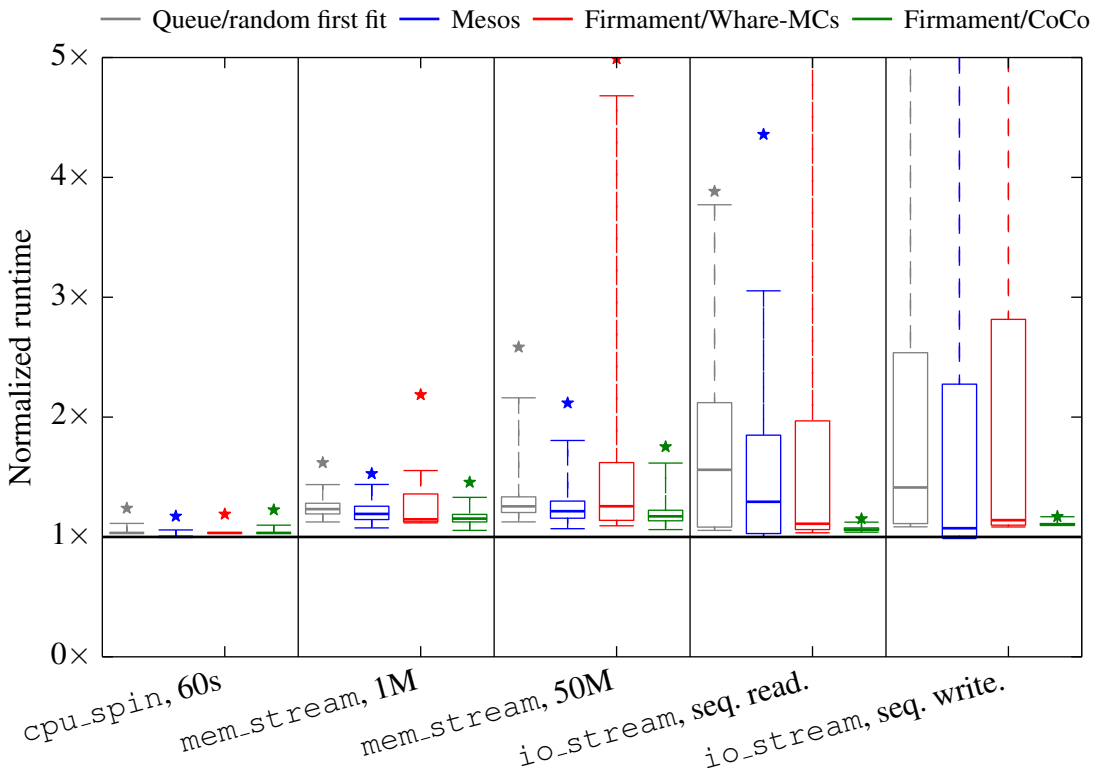
Firmament's cost models improve normalised task runtime over the queue-based baseline in almost all cases. This is unsurprising, as the queue-based baseline neither models load nor interference. However, Firmament's cost models also always match or outperform Mesos, which is load-aware.

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:

(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 cluster hardware.



(a) Heterogeneous SRG test cluster (10 machines, 140 cores).



(b) Homogeneous model data centre (28 machines, 336 cores).

Figure 8.1: Runtime of the synthetic workloads from Table 8.2 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 25$^{\text{th}}$ and 75$^{\text{th}}$ percentiles, whiskers are 1$^{\text{st}}$ and 99$^{\text{th}}$ percentiles, and the star represents the maximum outlier.

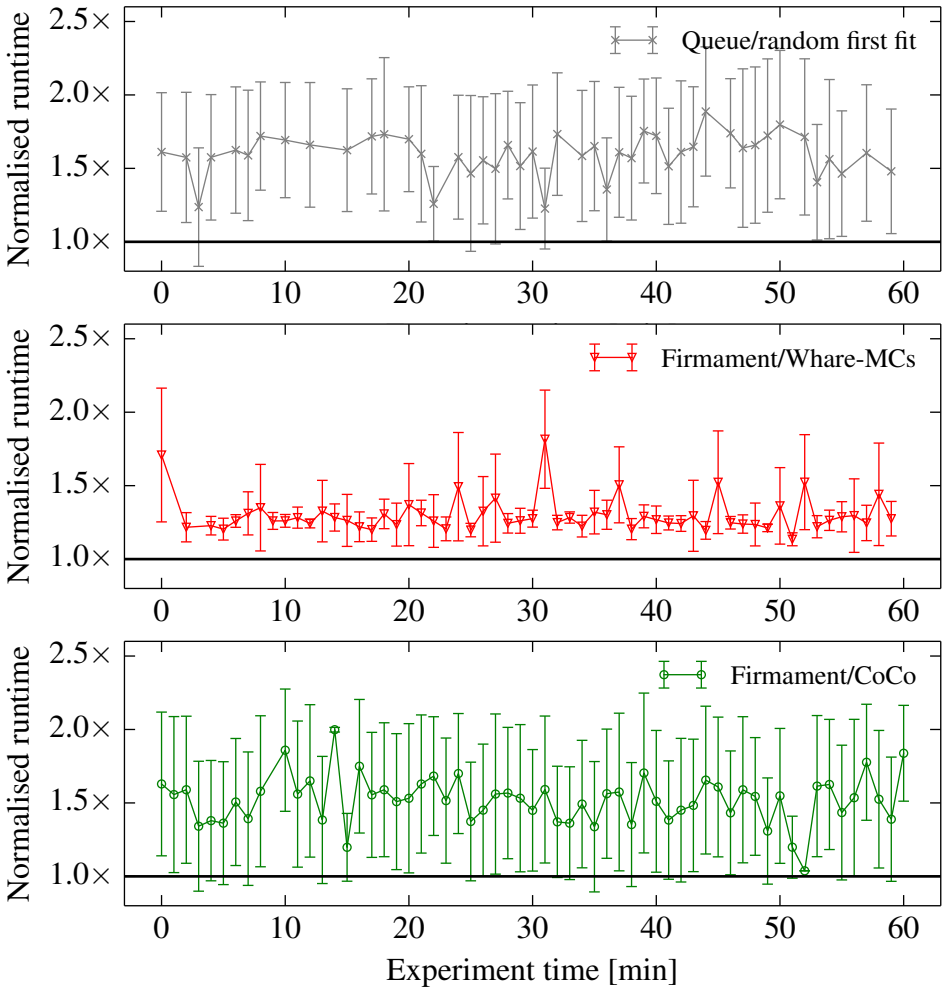


Figure 8.2: Average runtime of `mem_stream` (L3-fit) tasks in each minute of a one-hour experiment on the heterogeneous SRG cluster in the queue-based baseline and with the different Firmament cost models (error bars: standard deviation). Whare-MCs quickly discovers good mappings at the start of the experiment and applies them when possible.

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 8.2 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-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 8.1. 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 (§7.3). The benefits are evident in Figure 8.1b: only one `io_stream` task fits on each machine, and hence their normalised runtime is close to ideal. On the heterogeneous cluster (Figure 8.1a), normalised runtime varies due to the machines' 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 8.1a).

Moreover, CoCo's conservative admission control requires tasks to wait much longer than with the Whare-Map cost model. Figure 8.3 shows task wait time distributions for the same workloads and setups as shown in Figure 8.1. 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 8.4 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

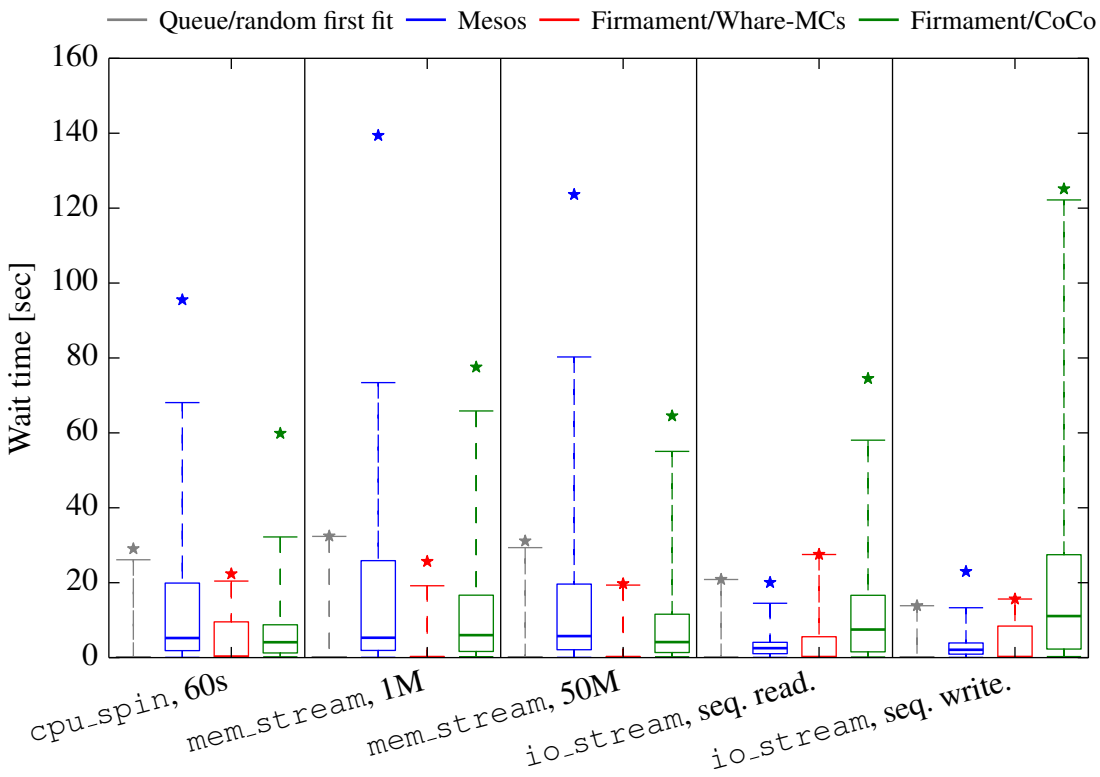


Figure 8.3: Per-task wait time for different workloads in the cluster mix experiment on the heterogeneous SRG cluster. Boxes around the median value correspond to 25th and 75th percentiles, whiskers are 1st and 99th percentiles, and the star represents the maximum.

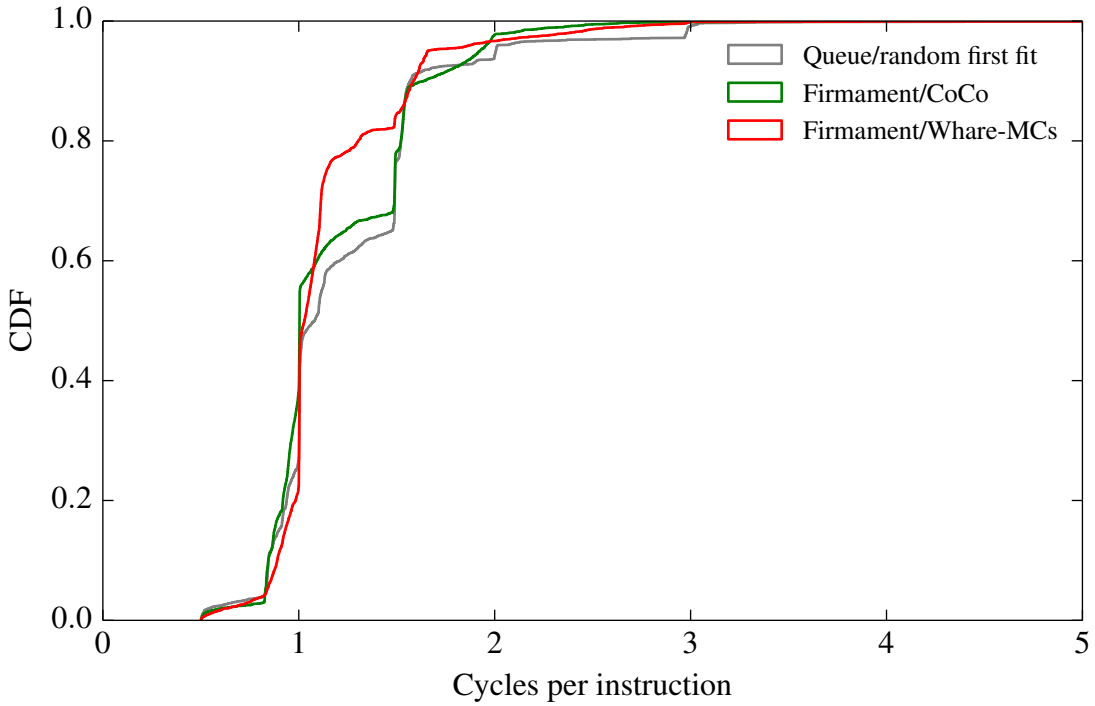


Figure 8.4: Distribution of the per-task average CPI for the synthetic workloads in Table 8.2 over a 1-hour experiment on the heterogeneous SRG test cluster. Less is better, and Firmament utilises the hardware more efficiently than the baseline scheduler.

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.

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 make scheduler learn to avoid 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.[4]

I also compared my CPI distributions in Figure 8.4, 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.2, but the key take-away is that Google's workloads experience much higher CPI (suggesting higher load or more interference) and lower IPMA (suggesting large working sets, low cache affinity, or high interference) than the synthetic workloads I used.

Interference-aware scheduling addresses an immediate need in today's production data centres: improving utilisation and performance determinism by using the existing hardware more efficiently. However, Firmament also supports more exotic use cases: next, I describe the Green cost model, which improves the energy efficiency of a heterogeneous-ISA cluster.

8.2.2 Case study: energy-aware scheduling

In Section 7.4, 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 workload.[5]

[4] 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.

[5] The experiments presented here were run in collaboration with Gustaf Helgesson under my supervision; an earlier version of Figure 8.5 appears in Gustaf's MPhil thesis [Hel14, Fig. 6.2].

Machine	Architecture, core count × clock speed	RAM	Network	Disk
Pandaboard	ARM Cortex-A9, 2× 1.0 GHz	1 GB	100 Mbit/s	SSD (USB 2.0)
Wandboard	ARM Cortex-A9, 4× 1.0 GHz	2 GB	1 Gbit/s	SSD (S-ATA)
Dell R420	Intel Xeon E5-2420, 1× 1.9 GHz	64 GB	10 Gbit/s	HDD (S-ATA)
Dell R415	AMD Opteron 4234, 12× 3.1 GHz	64 GB	10 Gbit/s	HDD (S-ATA)
Itanium	Intel Itanium 2 9015, 8× 1.4 GHz	16 GB	1 Gbit/s	HDD (S-ATA)

Table 8.3: Specifications of the machines used in energy-aware scheduling experiments with Firmament.

Machine	Idle power	Full-load power
Pandaboard	6.95 W	7.58 W
Wandboard	9.55 W	12.25 W
Dell R420	87.10 W	175.04 W
Dell R415	70.60 W	238.02 W
Itanium 2	544.94 W	657.81 W

Table 8.4: Machine power consumption at different CPU load levels.

Cluster setup. The test cluster for this experiment consists of five heterogeneous machines, listed in Table 8.3. Two ARMv7-based SoCs represent upcoming many-core ARM servers: a dual-core Pandaboard[6] and a quad-core Wandboard.[7] Future ARM-based server products will feature higher clock speeds, larger numbers of cores, and increased memory capacity [AMD14], 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.[8] 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 3.10.17, 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 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. Since the measurement is taken at the socket, it covers whole-system power consumption including CPU, DRAM, peripheral, and PSU components.[9] When multiple tasks share a machine, I divide the total power consumption according to number of CPU cores they use (see §7.4).

[6] `http://www.pandaboard.org/`; accessed 03/07/2014.

[7] `http://www.wandboard.org/`; accessed 03/07/2014.

[8] The Itanium is neither energy-efficient nor power-proportional – it merely serves to test the scheduler's performance on a wider trade-off space.

[9] 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+12]. The values observed agreed within a maximum deviation of 3–4 W, but were usually within ±1 W.

Workloads. The experiment workload is a mix of batch jobs and service jobs. The batch jobs run typical MapReduce workloads (WordCount and joining two datasets) and file transfers. Batch jobs are issued such that, on average, ten jobs run at any 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 Joules) consumed while running a workload, in addition to the baseline energy consumed by the idle machine. This metric makes the assumption that machines in a data centre are always powered on, i.e. the idle energy cost is incurred no matter what scheduling decisions Firmament makes. This is in contrast to efforts that perform dynamic power management (DPM) of data centre machines [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.[10]

Energy savings. For evaluation, I compare four different approaches:

(i) randomised task assignment over the x86 machines only (i.e. a *homogeneous* cluster);

(ii) randomised task assignment over the entire heterogeneous, mixed-ISA cluster;

(iii) task assignment according to a performance-oriented, but energy-oblivious Firmament cost model; and

(iv) task assignment according to the energy-aware Green Firmament cost model.

The first approach represents the state-of-the-art baseline of using a traditional, x86-only cluster. 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.

[10] 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.

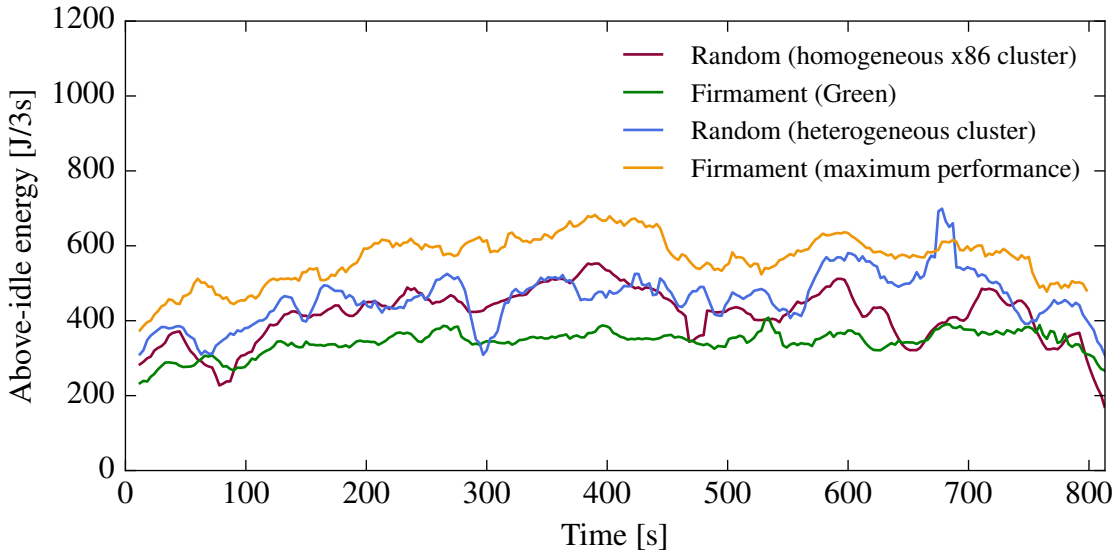
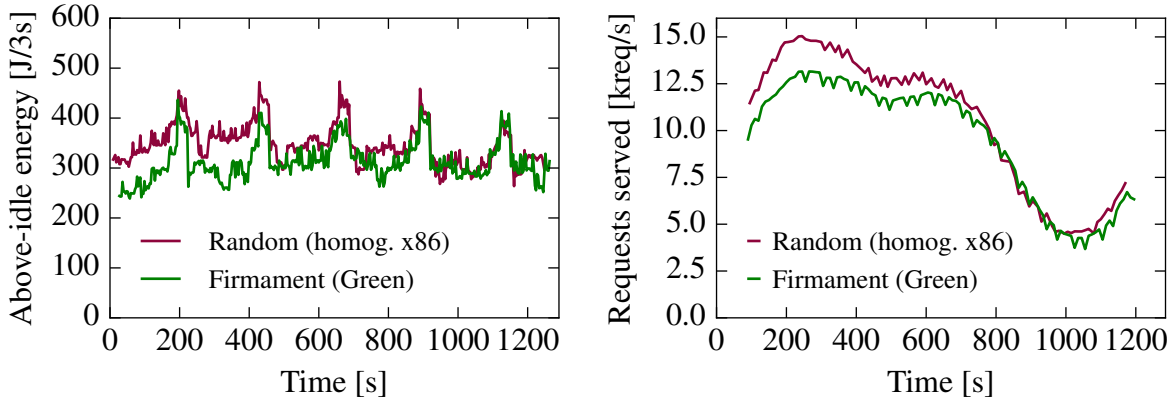


Figure 8.5: Active energy consumed by different setups for a mixed batch/service workload experiment. Values shown are running averages of five data points over the median of three runs. The *x*-axis is cut off at the completion (±10s) of all setups apart from the random heterogeneous one, which takes substantially longer at ≈ 1800s.



(a) Active energy over time.

(b) Requests served over time.

Figure 8.6: Auto-scaled service jobs using Firmament's energy-aware cost model compared to using a homogeneous x86 cluster without auto-scaling. The energy-aware cost model *(a)* offers a 12% reduction in above-idle energy, while *(b)* serving the same load pattern. Values are running averages over ten data points.

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 (for 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 also using the non-x86 machines. Finally, the Green cost model (§7.4) aims to complete tasks in the most *efficient* way possible, which should yield the lowest overall energy consumption but still meet the jobs' performance constraints.

Cost model	*Avg. web server throughput [req./s]*	*Missed batch deadlines*
Random homogeneous	52,553	0
Random heterogeneous	51,049	5
Maximum performance	53,324	0
Green cost model	52,871	0

Table 8.5: Service throughput and batch deadline satisfaction in the Green cost model.

In Figure 8.5, 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 8.5), 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; if the energy-inefficient IA-64 machine was replaced with an x86 machine, gains would increase 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 8.6a shows the 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 8.6b). 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 meet their performance constraints. I therefore measure the total request throughput for service jobs and the number of missed batch job deadlines in the experiment shown in Figure 8.5. As the values for each setup are shown in Table 8.5 illustrate, all cost models apart from random assignment to heterogeneous machines see comparable request throughput and meet all deadlines.

8.3 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 8.6 summarises Firmament's support for the policies implemented by other existing schedulers (see Table 2.8 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. §6.3.4).

Finally, some systems – Mesos, YARN, and Omega – are themselves flexible scheduler platforms with configurable policies. Firmament can support all existing high-level scheduling policies for these systems that I am aware of, although more challenging ones are 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.[11]

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.

[11] 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 ($\simeq$ Apollo's PNs) would have to register with multiple parent coordinators ($\simeq$ Apollo's JMs).

System [Reference]	Implementable	Admission control required	Multi-round scheduling req.	Implementation summary
HFS [HFS]	✓	✗	✗	Enforce fairness using unscheduled aggregator demand and arc capacities (§6.3.2).
LATE [ZKJ+08]	✓	✗	✗	Model using v_i^j, α_i^j and preference arcs.
Quincy [IPC+09]	✓	✗	✗	*Cost model described in §7.1.*
Delay Sched. [ZBS+10]	✓	✗	✗	Use v_i^j to induce delay; drop v_i^j after expiry.
Mesos [HKZ+11]	depends			Multi-policy two-level scheduler platform.
CIEL [Mur11, §4.3]	✓	✗	✗	Use preference arcs for Sweetheart references and locality (§6.3.1).
Jockey [FBK+12]	(✓)	✗	✓	Scale number of tasks in cost model; model deadlines via cost on arc to unscheduled aggregator.
alsched [TCG+12]	✓	✗	✓	Soft/hard constraints via preferences, combinatorial via multi-round scheduling (§6.3.1).
tetrisched [TZP+16]	✓	✓	✓	Soft constraints via preferences, combinatorial ones via multi-round scheduling (§6.3.1).
Whare-Map [MT13]	✓	✗	✗	*Cost model described in §7.2.*
YARN [VMD+13]	depends			Multi-policy two-level scheduler platform.
Omega [SKA+13]	depends			Multi-policy shared-state scheduler platform.
Sparrow [OWZ+13]	✓	✗	✗	Use distributed coordinators that each run a scheduler; optionally have multiple parent coordinators.
H-DRF [BCF+13]	✓	✓	✓	Only admit tasks if H-DRF constraints satisfied.
Choosy [GZS+13]	✓	✓	✓	Only admit tasks if CMMF satisfied.
Paragon [DK13]	✓	✓	✗	Use profiling results to determine costs on arcs to machine equivalence classes (cf. §7.2).
Quasar [DK14]	(✓)	✓	✓	As in Paragon, but also scale resource requests in cost model in response to model.
Apollo [BEL+14]	✓	✓	✓	*Cost model described in this section.*
KMN [VPA+14]	✓	✗	✗	Force k of n tasks to schedule via gang scheduling (§6.3.3), increase cost for additional $m = n - k$.
Tarcil [DSK15]	✓	✓	✗	Use distributed coordinators' schedulers for short tasks, and a top-level coordinator for long ones.
Hawk [DDK+15]	✓	✓	✗	Use arcs from equivalence class aggregator to machines to express work stealing cost.
Mercury [KRC+15]	✓	✓	✗	As Hawk, but guaranteed tasks preempt queueable ones on delegation conflicts (via higher priority).
Bistro [GSW15]	✓	✓	✓	Map resource forest onto resource topology and use resource model akin to CoCo (§7.3).
CoCo (co-location-aware)	✓	✓	✗	*Cost model described in §7.3.*
Green (energy-aware)	✓	✗	✗	*Cost model described in §7.4.*

Table 8.6: 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.

8.4 Scalability

In Section 6.2, 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 using the relaxation algorithm, and by solving the minimum-cost, maximum-flow problem incrementally. In this section, I evaluate the reduction in scheduler decision time attained by this approach.[12] I compare exploratory experiments with incremental relaxation to solving the optimisation problem from scratch each time (as Quincy's `cs2` solver does).

In the experiment, I use Firmament on a Google 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:

1. As the precise nature of the machines in the Google cluster is unknown, I synthesise the topology of a 24-core machine (equivalent to type **C** in Table 8.1a) for each machine.

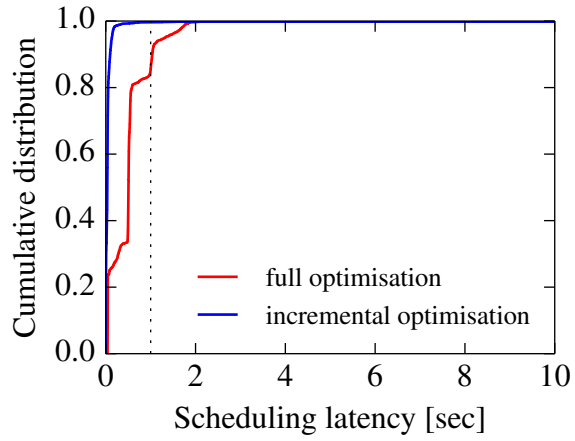
2. To be able to use Quincy's locality preferences in the flow network, I simulate a GFS-like distributed file system, and assign random inputs to each task. The distributed file system contains 1.2 billion 64 MB blocks (≈75 PB), with file sizes sampled from a distribution of HDFS file sizes at Facebook [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,[13] and I measure the time taken by different solvers to complete the minimum-cost, maximum-flow optimisation. I use two solvers: Goldberg's `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 RELAXIV [BT94] that runs an incremental optimisation.

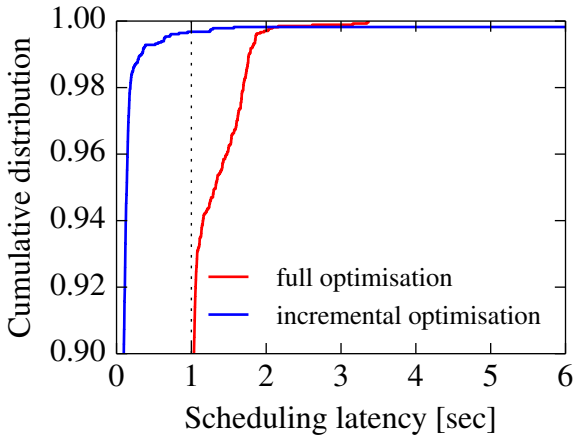
Figure 8.7 shows the results for both a medium-sized cluster of 3,000 machines – sub-sampling about a quarter of the Google trace events – and for the full Google cluster. In 2009, Quincy took "a little over a second" to schedule 100 jobs on a cluster of 2,500 quad-core machines [IPC+09, §6.5]; by contrast, my simulated 3,000-machine cluster is much larger: it 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 (§6.5.2). As Figures 8.7a and 8.7b show, Firmament's average decision time is 515ms when running Goldberg's `cs2` solver from scratch, with the 90$^{\text{th}}$ percentile at around 1s. This is comparable to the 2009 result for Quincy. The incremental relaxation solver, however, completes in 44ms on average and takes 250ms in the 90$^{\text{th}}$ percentile.

[12] 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.

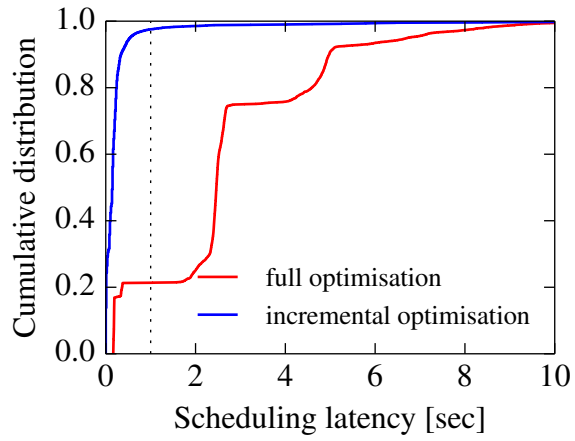
[13] This is a *simulation* and not a *replay* of the trace since different scheduling decisions are made: unlike in the trace, each machine runs at most 24 tasks (one per CPU thread) and the simulated DFS locality is hypothetical. However, as the primary metric of interest is the solvers' decision time, this reduction in fidelity is inconsequential.



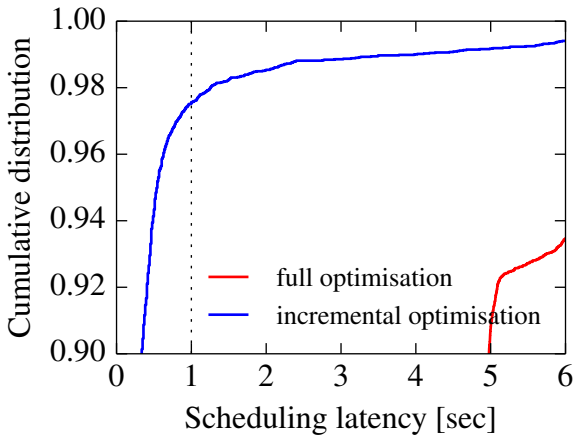
(a) 3,000 machines, entire distribution.



(b) 3,000 machines, 90th percentile.



(c) Google cluster, entire distribution.



(d) Google cluster, 90th percentile.

Figure 8.7: Cumulative distributions of the min-cost solver runtime over one hour of the workload on: *(a)*, *(b)*, a medium cluster representing a third of the Google workload, and *(c)*, *(d)*, the full 12,550-machine Google cluster. The incremental solver is an order of magnitude faster than the full optimisation, and runs in sub-second time at Google scale.

The results for the full-scale Google cluster show a similar trend: the average decision time is 265ms, the 90th percentile is at 345ms, and 97.4% of decisions are made in under a second. By comparison, the average decision time running `cs2` from scratch is 2.86s, over 10× longer than with the incremental relaxation solver. Moreover, tail of the decision time distribution is substantially improved: in the 90th percentile, decisions take 0.35s (`cs2`: 4.98s), and the 99th percentile is at 3.98s (`cs2`: 9.25s). This difference is primarily a consequence of the properties of the algorithms used by the relaxation-based RELAXIV and the cost-scaling `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 pushes the flow towards the sink, at the cost of having to back-track in the case of contention. Such contention occurs when the cluster runs at very high utilisation or in overload [GSG+16], but the Google workload trace does not encounter such pathological cases on the simulated cluster.

This experiment shows that Firmament can achieve sub-second scheduling latency even on large clusters that Quincy's cost-scaling `cs2` solver does not scale to. Indeed, these decision times are competitive with those achieved by distributed schedulers such as Sparrow [OWZ+13] or Tarcil [DSK15], despite Firmament running as a fully centralised scheduler in this experiment.

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 maintain the policy-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 of this approach is that it can easily lose optimality unless it is carefully implemented.

Only simple extensions to the Firmament implementation would be required to support these approaches. However, in light of the fact that even a relatively unoptimised minimum-cost, maximum-flow solver based on relaxation and incremental optimisation achieves decision times comparable to the fastest state-of-the-art distributed schedulers, neither option is likely to be required for scalability alone.

In further work since the experiments presented here,[14] I have shown that even pathological cases for the relaxation algorithm – such as high fan-in in the flow network, or an overloaded cluster – can be handled by falling back onto incremental cost-scaling [GSG+16]. This follow-on work also examines both the Google workload and Firmament's scalability to very short, sub-second tasks in more detail.

[14] I collaborated with Ionel Gog for the extensions to this work, which are outside the scope of this dissertation, but which appear in the OSDI 2016 paper on Firmament [GSG+16].

8.5 Summary

In Chapter 2, I enumerated several goals for improved data centre scheduling (§2.1.4 and §2.3.6). In this section, I evaluated how Firmament meets all of these goals.

With the experiments in this section, I have shown that:

1. Firmament *avoids co-location interference* between workloads and *matches workloads to appropriate machines*, addressing challenges 2 and 3 in Section 2.1.4. Using the Whare-Map and CoCo cost models, Firmament reduces workload slowdowns due to suboptimal assignments by 2–4× in the median (§8.2.1).

2. The Green cost model for Firmament yields *improved energy efficiency* on a heterogeneous ARM/x86 cluster, reducing overall energy consumption for the given workload by 18% (§8.2.2).

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 (§8.3).

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 (§8.4).

Moreover, Firmament meets all the goals for a cluster scheduler identified in Section 2.3 (Table 2.8). 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.

Chapter 9

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 abstracted as "warehouse-scale computers" that support large distributed systems. Such 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, as I observed in **Chapter 2**, current systems software for data centre environments is faced with several challenges: hardware heterogeneity and task co-location interference affect performance determinism, and the distributed infrastructure systems "middleware" developed to provide operating system functionality across machines lacks uniformity and security in its abstractions for resource management. To address these challenges, I have proposed novel approaches to building data centre operating systems and schedulers.

- In **Chapter 3**, I introduced a high-level model for a new, clean-slate *decentralised, distributed data centre operating system* that builds upon both classic distributed OS literature and recent innovations in data centre infrastructure systems. I illustrated the key abstractions and principles required for the OS to securely and efficiently support modern workloads, focusing especially on resource naming and management, implemented via a pervasive object abstraction and a capability system based on *identifier* and *handle* capabilities.

- Subsequently, in **Chapter 4**, I presented DIOS, a prototype implementation of this model in Linux. 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 physical objects. A new system call API around these abstractions allows DIOS to support scalable distributed applications with lightweight, rigid isolation between different tasks and with flexibly adjustable distribution transparency.

- **Chapter 5** evaluated DIOS, and found that despite being an unoptimised prototype, it offers comparable performance with existing systems in micro-benchmarks and for a MapReduce workload. I also qualitatively evaluated the security benefits via compartmentalisation that fine-grained capabilities in DIOS enable, and how they improve over the state-of-the-art in distributed infrastructure systems.

- *Scheduling* in a distributed data centre OS was the focus of **Chapter 6**. I showed that modelling the scheduling problem as a flow network allows for the expression of novel policies and is 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.

- In **Chapter 7**, I discussed three case studies of scheduling policies implemented for Firmament: *(i)* an implementation of the Whare-Map interference-aware scheduling policy, *(ii)* the coordinated co-location model (CoCo), which simultaneously optimises for high resource utilisation and minimal interference between tasks, and *(iii)* the Green cost model, which reduces power consumption by using power-efficient machines.

- Finally, **Chapter 8** evaluated Firmament, and found that it improves the quality scheduling decisions over state-of-the-art systems since it understands machine heterogeneity and avoids task co-location interference where possible. Moreover, I found that Firmament flexibly supports many different scheduling policies, and that its underlying minimum-cost, maximum-flow optimisation can scale to large data centres.

The work presented in these chapters collectively serves to prove the hypothesis introduced in Chapter 1. First, a new, uniform approach to distributed resource management in a clean-slate data centre OS yields efficiency and security benefits for applications. With DIOS, I have demonstrated that my decentralised data centre OS model can be implemented and deployed as an extension to the widely-used Linux operating system, and that practical distributed systems applications can be implemented atop it. Second, the Firmament scheduler helped me prove that the integration of fine-grained, machine-level information and global, cluster-level information successfully addresses the challenges of machine heterogeneity and task co-location interference in data centres. Moreover, it showed that the flow network approach to scheduling is not only highly expressive, but also scales to large, warehouse-scale clusters.

There is, however, ample opportunity for future work extending DIOS and Firmament.

9.1 DIOS and data centre operating systems

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

```
1  use std::os::dios;
2
3  fn fib(n: u32) -> u32 {
4      fn fib_task(chan: dios::Channel) {
5          let n = chan.recv().unwrap();
6          chan.send(fib(n)).unwrap();
7      }
8
9      match n {
10         0  => 0,
11         1  => 1,
12         _  => {
13             let fib1 = dios::spawn_task(fib_task).unwrap();
14             fib1.send(n-2).unwrap();
15             let fib2 = dios::spawn_task(fib_task).unwrap();
16             fib2.send(n-1).unwrap();
17             fib1.recv().unwrap() + fib2.recv().unwrap()
18         }
19     }
20 }
21
22 fn main() {
23     println!("fib(10) = {}", fib(10));
24 }
```

Listing 9.1: Distributed `fib(10)` implementation in Rust on top of DIOS. The corresponding C implementation has 280 lines (10× more verbose). Lines 13 and 15 spawn the `fib_task` closure as new DIOS tasks.

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.

9.1.1 High-level language support

Writing programs directly against the DIOS system call API is rather complex. Most software indirects 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 common data centre application needs. By integrating DIOS objects and abstractions with higher-level programming languages, users might be able to 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.

Rust programs can run on DIOS by extending the Rust runtime and standard library with an

indirection layer for DIOS. 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). This allows distributed applications to be expressed concisely.

Listing 9.1 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 a C implementation against the DIOS system call API, which comes to 280 lines, and more concise than a Rust implementation using POSIX pipes and processes, which comes to 50 lines and does not support distribution across machines.

In preliminary benchmarks, the Rust implementation of $fib(10)$ (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 that the runtime port is unoptimised, this is an encouraging result.

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 DIOS 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.

9.1.2 Changing kernel structure

In my work on DIOS, I have so far focused on the OS abstractions exposed to applications. However, the specific requirements of a shared data centre environment motivate further, deeper changes even to local OS construction.

OS-level Quality-of-Service (QoS) enforcement. Firmament's scheduling policies allow the cluster scheduler to avoid negative co-location interference between tasks by evading interfering placements (§8.2.1). 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.

Hence, 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 distributed data centre OS. Additionally, new hardware features – such as Intel's Cache Allocation Technology (CAT),

[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].

which explicitly partitions CPU caches between processes – will require OS and cluster-level support (as, e.g. in Heracles [LCG+15]).

Implementation scalability. In Section 4.9, I argued that the DIOS 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 reliance on Linux kernel code in DIOS restricts the practical *implementation* scalability attained.

It would be interesting to implement the decentralised data centre OS model using approaches explicitly designed for scalability, such as a multikernel approach (as in Barrelfish [BBD+09]) or virtual memory and file systems based on the scalable commutativity rule (as in sv6 [CKZ+13, §6]). The multikernel model, in particular, is an attractive target, since the decentralised data centre OS model does not require implicit shared memory or cache coherence.

To explore some of these questions, the DIOS core module could be adapted to work atop systems that use these approaches. For example, an implementation using sv6 would allow the scalability of the DIOS system call API to be evaluated independently of Linux implementation choices; a port to a multikernel would require the DCP to be used for communication even within a (potentially non-cache-coherent) machine.

9.1.3 Further security improvements

I have argued that the DIOS 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 distributed capabilities to hardware capabilities. The identifier and handle capabilities in DIOS 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 can be adjacent in memory. The use of guard pages or similar mechanisms can mitigate protection violations *within* a task address space, but does not eliminate it. However, some use cases (e.g. holding private keys in task 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 DIOS object model, and would allow fine-grained compartmentalisation of objects even within the task address space. In this way, DIOS could for example isolate network threads and ensure that they cannot access sensitive objects available to the same task.

Applying information flow control. One key advantage of DIOS is its use of uniform OS-level abstractions (names, references and objects) throughout a distributed system, rather

than layering and combining disparate abstractions. This may enable pervasive information flow control (IFC) in the data centre, since all communication between tasks in DIOS must happen explicitly via shared objects. IFC monitors and reasons about how information is exposed to different components of a system, and many IFC systems – e.g. HiStar [ZBK+06], and its distributed DStar [ZBM08] variant – are based on label propagation, which could be added to DIOS with moderate effort.

Using more advanced cryptography for distributed capabilities. DIOS currently requires independent transport-level data encryption to securely transfer capabilities across machine boundaries. More expressive cryptographic schemes could add authentication directly to the capabilities: for example, Macaroons [BPE+14] can have attached attestations that authenticate them (including via a third party), and are communicable on untrusted channels. The capability delegation mechanisms in DIOS could be adapted to use Macaroons to gain these benefits.

I hope to investigate these directions in the future.

9.2 Firmament and cluster scheduling

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 the existence of dozens of different cluster schedulers (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 8.3 with an implementation of several additional policies atop Firmament, and to perform a comparative analysis of their relative merits on real and simulated workloads.

Moreover, even though Firmament supports all key features of current cluster schedulers, new requirements keep emerging. For example, future data centres may contain even more fundamentally heterogeneous compute resources: Firebox [MAH+14, §2.1; Asa14] and HP's "Machine" project [Edg14] expect future data centres to be based on 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.

9.3 Summary

As applications increasingly rely on back-end services operated in large-scale "cloud" data centres, systems software must better support these new environments. In this dissertation, I have made the case for a new model of resource management and for a new approach to scheduling as part of a data centre operating system.

With DIOS and Firmament, I have developed prototype platforms that make 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

[AA14] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Edition 0.80. Arpaci-Dusseau Books, May 2014 (cited on page 136).

[AAB+13] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, et al. "Scuba: Diving into Data at Facebook". In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pp. 1057–1067 (cited on pages 32–33).

[AAK+11] Ganesh Ananthanarayanan, Sameer Agarwal, Srikanth Kandula, Albert Greenberg, Ion Stoica, Duke Harlan, and Ed Harris. "Scarlett: Coping with Skewed Content Popularity in MapReduce Clusters". In: *Proceedings of the 6th European Conference on Computer Systems (EuroSys)*. Salzburg, Austria, Apr. 2011, pp. 287–300 (cited on page 58).

[ABB+13] Tyler Akidau, Alex Balikov, Kaya Bekiroğlu, Slava Chernyak, Josh Haberman, Reuven Lax, Sam McVeety, et al. "MillWheel: Fault-tolerant Stream Processing at Internet Scale". In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pp. 1033–1044 (cited on pages 31, 34).

[ABB+86] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. "Mach: A New Kernel Foundation for UNIX Development". In: *Proceedings of the 1986 USENIX Summer Conference*. Atlanta, Georgia, USA, June 1986, pp. 93–113 (cited on pages 44, 67, 76, 79, 99, 116).

[ABL+85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe. "The Eden System: A Technical Review". In: *IEEE Transactions on Software Engineering* 11.1 (Jan. 1985), pp. 43–59 (cited on page 76).

[AGS+11] Ganesh Ananthanarayanan, Ali Ghodsi, Scott Shenker, and Ion Stoica. "Disk-locality in Datacenter Computing Considered Irrelevant". In: *Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. Napa, California, USA, May 2011, pp. 12–17 (cited on page 58).

[AMD14] Advanced Micro Devices, Inc. *AMD to Accelerate the ARM Server Ecosystem with the First ARM-based CPU and Development Platform from a Server Processor Vendor*. AMD press release; accessed on 08/08/2014. Jan. 2014. URL: http://www.amd.com/en-us/press-releases/Pages/amd-to-accelerate-2014jan28.aspx (cited on page 186).

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice Hall, 1993 (cited on page 150).

[Asa14] Krste Asanović. *FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers*. Feb. 2014. URL: https://www.usenix.org/conference/fast14/technical-sessions/presentation/keynote (cited on page 202).

[BAC+13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, et al. "TAO: Facebook's Distributed Data Store for the Social Graph". In: *Proceedings of the USENIX Annual Technical Conference*. San Jose, California, USA, June 2013, pp. 49–60 (cited on pages 32–33).

[Bac98] Jean Bacon. *Concurrent Systems: Operating Systems, Database and Distributed Systems–an Integrated Approach*. Second edition. International computer science series. Addison-Wesley, 1998 (cited on page 50).

[BBC+11] Jason Baker, Chris Bond, James C. Corbett, J.J. Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, et al. "Megastore: Providing Scalable, Highly Available Storage for Interactive Services". In: *Proceedings of the 5th Biennial Conference on Innovative Data Systems Research (CIDR)*. Asilomar, California, USA, Jan. 2011, pp. 223–234 (cited on pages 31, 33).

[BBD+09] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, et al. "The Multikernel: A New OS Architecture for Scalable Multicore Systems". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. Big Sky, Montana, USA, Oct. 2009, pp. 29–44 (cited on pages 23, 76, 79, 201).

[BCC+08] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, M. Frans Kaashoek, Robert Morris, Aleksey Pesterev, et al. "Corey: An Operating System for Many Cores". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. San Diego, California, USA, Dec. 2008, pp. 43–57 (cited on pages 23, 66).

[BCD72] André Bensoussan, Charles T. Clingen, and Robert C. Daley. "The Multics Virtual Memory: Concepts and Design". In: *Communications of the ACM* 15.5 (May 1972), pp. 308–318 (cited on page 43).

[BCF+13] Arka A. Bhattacharya, David Culler, Eric Friedman, Ali Ghodsi, Scott Shenker, and Ion Stoica. "Hierarchical Scheduling for Diverse Datacenter Workloads". In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. Santa Clara, California, Oct. 2013, 4:1–4:15 (cited on pages 56, 60, 146, 191).

[BCH13] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition". In: *Synthesis Lectures on Computer Architecture* 8.3 (July 2013), pp. 1–154 (cited on pages 19, 21, 23, 29, 189).

[BCM+10a] Silas Boyd-Wickizer, Austin T. Clements, Yandong Mao, Aleksey Pesterev, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. "An Analysis of Linux Scalability to Many Cores". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, Oct. 2010, pp. 1–8 (cited on page 67).

[BCM+10b] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, et al. "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications". In: *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)*. Pisa, Italy, Feb. 2010 (cited on page 155).

[BEL+14] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, et al. "Apollo: Scalable and Coordinated Scheduling for Cloud-Scale Computing". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 285–300 (cited on pages 43, 56–57, 61, 146, 154, 174, 190–191).

[Ber11] Lars Bergstrom. "Measuring NUMA effects with the STREAM benchmark". In: *CoRR* abs/1103.3225 (2011) (cited on page 36).

[BFTN0] Team Barrelfish. *Barrelfish Architecture Overview*. Technical report. Barrelfish Technical Note 000; version of 04.12.2013. ETH Zürich, Dec. 2013 (cited on page 79).

[BGH+13] Peter Bailis, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. "Bolt-on Causal Consistency". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. New York, New York, USA, 2013, pp. 761–772 (cited on page 108).

[BGS+11] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, et al. "Apache Hadoop Goes Realtime at Facebook". In: *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Athens, Greece, June 2011, pp. 1071–1080 (cited on pages 32–33, 47).

[BKL+10] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. "Finding a Needle in Haystack: Facebook's Photo Storage". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, Oct. 2010, pp. 1–8 (cited on pages 32–33, 79).

[BLF+13] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, et al. "Composing OS Extensions Safely and Efficiently with Bascule". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 239–252 (cited on page 103).

[BN84] Andrew D. Birrell and Bruce Jay Nelson. "Implementing Remote Procedure Calls". In: *ACM Transactions on Computer Systems* 2.1 (Feb. 1984), pp. 39–59 (cited on page 67).

[BPA+13] Muli Ben-Yehuda, Omer Peleg, Orna Agmon Ben-Yehuda, Igor Smolyar, and Dan Tsafrir. "The nonkernel: A Kernel Designed for the Cloud". In: *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSYS)*. Singapore, July 2013 (cited on page 24).

[BPE+14] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. "Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud". In: *Proceedings of the 2014 Network and Distributed System Security Symposium (NDSS)*. Feb. 2014 (cited on pages 76–77, 202).

[BPH14] Andrew Baumann, Marcus Peinado, and Galen Hunt. "Shielding Applications from an Untrusted Cloud with Haven". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, 2014, pp. 267–283 (cited on page 69).

[BPK+14] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. "IX: A Protected Dataplane Operating System for High Throughput and Low Latency". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 49–65 (cited on page 24).

[BPS+09] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. "Your computer is already a distributed system. Why isn't your OS?" In: *Proceedings of the 12th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. Monte Verità, Switzerland, May 2009 (cited on page 23).

[Bre00] Eric A. Brewer. "Towards Robust Distributed Systems (Abstract)". In: *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. Portland, Oregon, USA, 2000, pp. 7– (cited on page 94).

[BT88a] Dimitri P. Bertsekas and Paul Tseng. "Relaxation Methods for Minimum Cost Ordinary and Generalized Network Flow Problems". In: *Operations Research* 36.1 (Feb. 1988), pp. 93–114 (cited on pages 150, 152).

[BT88b] Dimitri P. Bertsekas and Paul Tseng. "The Relax codes for linear minimum cost network flow problems". In: *Annals of Operations Research* 13.1 (Dec. 1988), pp. 125–190 (cited on page 153).

[BT94] Dimitri P. Bertsekas and Paul Tseng. *RELAX-IV: A faster version of the RELAX code for solving minimum cost flow problems*. Technical report LIDS-P-2276. Massachusetts Institute of Technology, Nov. 1994 (cited on pages 153, 192).

[Bul05] James R. Bulpin. "Operating system support for simultaneous multithreaded processors". PhD thesis. University of Cambridge Computer Laboratory, Feb. 2005 (cited on page 239).

[Bur06] Mike Burrows. "The Chubby Lock Service for Loosely-coupled Distributed Systems". In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, Washington, USA, Nov. 2006, pp. 335–350 (cited on pages 31–32, 47, 107).

[CAB+12] Yanpei Chen, Sara Alspaugh, Dhruba Borthakur, and Randy Katz. "Energy Efficiency for Large-scale MapReduce Workloads with Significant Interactive Analysis". In: *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*. Bern, Switzerland, Apr. 2012, pp. 43–56 (cited on page 187).

[CAK12] Yanpei Chen, Sara Alspaugh, and Randy Katz. "Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads". In: *Proceedings of the VLDB Endowment* 5.12 (Aug. 2012), pp. 1802–1813 (cited on page 192).

[CBB+13] Michael Curtiss, Iain Becker, Tudor Bosman, Sergey Doroshenko, Lucian Grijincu, Tom Jackson, Sandhya Kunnatur, et al. "Unicorn: A System for Searching the Social Graph". In: *Proceedings of the VLDB Endowment* 6.11 (Aug. 2013), pp. 1150–1161 (cited on pages 32, 34).

[CBZ91] John B. Carter, John K. Bennett, and Willy Zwaenepoel. "Implementation and Performance of Munin". In: *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP)*. Pacific Grove, California, USA, Oct. 1991, pp. 152–164 (cited on page 82).

[CCB+14] Marcus Carvalho, Walfredo Cirne, Francisco Brasileiro, and John Wilkes. "Long-term SLOs for Reclaimed Cloud Computing Resources". In: *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*. Seattle, Washington, USA, Nov. 2014, 20:1–20:13 (cited on page 61).

[CD16] Victor Costan and Srinivas Devadas. *Intel SGX Explained*. Cryptology ePrint Archive, Report 2016/086. `http://eprint.iacr.org/2016/086`. Jan. 2016 (cited on page 69).

[CD89] Raymond C. Chen and Partha Dasgupta. "Linking consistency with object/thread semantics: an approach to robust computation". In: *Proceedings of the 9th International Conference on Distributed Computing Systems (ICDCS)*. Newport Beach, California, USA, June 1989, pp. 121–128 (cited on page 82).

[CDE+13] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, et al. "Spanner: Google's Globally Distributed Database". In: *ACM Transactions on Computer Systems* 31.3 (Aug. 2013), 8:1–8:22 (cited on pages 31, 33, 47).

[CDG+05] Nicholas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounie, Pierre Neyron, et al. "A batch scheduler with high level components". In: *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*. Cardiff, Wales, United Kingdom, May 2005, pp. 776–783 (cited on page 56).

[CDG+06] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, et al. "Bigtable: A Distributed Storage System for Structured Data". In: *Proceedings of the 7th USENIX Symposium on Operating System Design and Implementation (OSDI)*. Seattle, Washington, USA, Nov. 2006 (cited on pages 22, 31, 33, 49, 70, 79).

[CEH+13] Juan A. Colmenares, Gage Eads, Steven Hofmeyr, Sarah Bird, Miquel Moretó, David Chou, Brian Gluzman, et al. "Tessellation: Refactoring the OS Around Explicit Resource Containers with Continuous Adaptation". In: *Proceedings of the 50th Annual Design Automation Conference (DAC)*. Austin, Texas, USA, June 2013, 76:1–76:10 (cited on page 24).

[CGP+02] Russ Cox, Eric Grosse, Rob Pike, David L. Presotto, and Sean Quinlan. "Security in Plan 9". In: *Proceedings of the 11th USENIX Security Symposium (USENIX Security)*. San Francisco, California, USA, Aug. 2002, pp. 3–16 (cited on page 76).

[CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. "Paxos Made Live: An Engineering Perspective". In: *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. Portland, Oregon, USA, Aug. 2007, pp. 398–407 (cited on pages 32, 48).

[Che88] David Cheriton. "The V distributed system". In: *Communications of the ACM* 31.3 (Mar. 1988), pp. 314–333 (cited on pages 44–45, 112).

[CKZ+13] Austin T. Clements, M. Frans Kaashoek, Nickolai Zeldovich, Robert T. Morris, and Eddie Kohler. "The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Nemacolin Woodlands, Pennsylvania, USA, Nov. 2013, pp. 1–17 (cited on pages 67, 113, 201).

[CKZ13] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. "RadixVM: Scalable Address Spaces for Multithreaded Applications". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 211–224 (cited on pages 67, 106).

[Cle14] Austin T. Clements. "The scalable commutativity rule: Designing scalable software for multicore processors". PhD thesis. Massachusetts Institute of Technology, June 2014 (cited on pages 112–113).

[CLF+94] Jeffrey S. Chase, Henry M. Levy, Michael J. Feeley, and Edward D. Lazowska. "Sharing and Protection in a Single-address-space Operating System". In: *ACM Transactions on Computer Systems* 12.4 (Nov. 1994), pp. 271–307 (cited on page 80).

[CLL+11] Biswapesh Chattopadhyay, Liang Lin, Weiran Liu, Sagar Mittal, Prathyusha Aragonda, Vera Lychagina, Younghee Kwon, et al. "Tenzing: A SQL Implementation On The MapReduce Framework". In: *Proceedings of the 37th International Conference on Very Large Data Bases (VLDB)*. Seattle, Washington, USA, Aug. 2011, pp. 1318–1327 (cited on pages 31, 33).

[CMD62] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. "An Experimental Time-sharing System". In: *Proceedings of the 1962 Spring Joint Computer Conference (SJCC)*. San Francisco, California, USA, May 1962, pp. 335–344 (cited on page 43).

[CMF+14] Michael Chow, David Meisner, Jason Flinn, Daniel Peek, and Thomas F. Wenisch. "The Mystery Machine: End-to-end Performance Analysis of Large-scale Internet Services". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 217–231 (cited on pages 32, 34).

[CNT+93] Miguel Castro, Nuno Neves, Pedro Trancoso, and Pedro Sousa. "MIKE: A Distributed Object-oriented Programming Platform on Top of the Mach Microkernel". In: *Proceedings of the 3rd Conference on USENIX MACH III*. Santa Fe, New Mexico, USA, Apr. 1993, pp. 6–26 (cited on page 67).

[Cre81] Robert J. Creasy. "The Origin of the VM/370 Time-Sharing System". In: *IBM Journal of Research and Development* 25.5 (Sept. 1981), pp. 483–490 (cited on page 43).

[CRP+10] Craig Chambers, Ashish Raniwala, Frances Perry, Stephen Adams, Robert R. Henry, Robert Bradshaw, and Nathan Weizenbaum. "FlumeJava: Easy, Efficient Data-parallel Pipelines". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Toronto, Ontario, Canada, June 2010, pp. 363–375 (cited on pages 31, 33).

[CV65] Fernando J. Corbató and Victor A. Vyssotsky. "Introduction and overview of the Multics system". In: *Proceedings of the 4th AFIPS Fall Joint Computer Conference (FJCC)*. Las Vegas, Nevada, USA, Nov. 1965, pp. 185–196 (cited on page 43).

[CWM+14] Brendan Cully, Jake Wires, Dutch Meyer, Kevin Jamieson, Keir Fraser, Tim Deegan, Daniel Stodden, et al. "Strata: High-Performance Scalable Storage on Virtualized Non-volatile Memory". In: *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, California, USA, Feb. 2014, pp. 17–31 (cited on page 80).

[DBF+94] Alan Dearle, Rex di Bona, James Farrow, Frans Henskens, Anders Lindström, John Rosenberg, and Francis Vaughan. "Grasshopper: An Orthogonally Persistent Operating System". In: *Computing Systems* 7.3 (June 1994), pp. 289–312 (cited on page 80).

[DD68] Robert C. Daley and Jack B. Dennis. "Virtual Memory, Processes, and Sharing in MULTICS". In: *Communications of the ACM* 11.5 (May 1968), pp. 306–312 (cited on page 43).

[DDK+15] Pamela Delgado, Florin Dinu, Anne-Marie Kermarrec, and Willy Zwaenepoel. "Hawk: Hybrid Datacenter Scheduling". In: *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA, July 2015, pp. 499–510 (cited on pages 56, 191).

[DG08] Jeffrey Dean and Sanjay Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters". In: *Communications of the ACM* 51.1 (Jan. 2008), pp. 107–113 (cited on pages 31–33, 58, 72, 125).

[DHJ+07] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*. Stevenson, Washington, USA, Oct. 2007, pp. 205–220 (cited on pages 70, 79, 108).

[DHR02] Jason Duell, Paul Hargrove, and Eric Roman. *The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart*. Technical report LBNL-54941. Lawrence Berkeley National Laboratory, Dec. 2002 (cited on page 252).

[DK13] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Houston, Texas, USA, Mar. 2013, pp. 77–88 (cited on pages 41, 56, 161, 191).

[DK14] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Salt Lake City, Utah, USA, Mar. 2014 (cited on pages 56, 59, 61, 161, 191).

[DKO+91] Fred Douglis, M. Frans Kaashoek, John K. Ousterhout, and Andrew S. Tanenbaum. "A Comparison of Two Distributed Systems: Amoeba and Sprite". In: *ACM Transactions on Computing Systems* 4 (1991), pp. 353–384 (cited on page 45).

[DLA88] Partha Dasgupta, Richard J. LeBlanc Jr., and William F. Appelbe. "The Clouds distributed operating system: functional description, implementation details and related work". In: *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*. San Jose, California, USA, June 1988, pp. 2–9 (cited on page 82).

[DLM+88] Terence H. Dineen, Paul J. Leach, Nathaniel W. Mishkin, Joseph N. Pato, and Geoffrey L. Wyant. "The network computing architecture and system: an environment for developing distributed applications". In: *Proceedings of the 33rd IEEE Computer Society International Conference (CompCon)*. Available at `http://jim.rees.org/apollo-archive/papers/ncs.pdf.gz`; accessed 02/11/2014. Feb. 1988, pp. 296–299 (cited on page 90).

[DN65] Robert C. Daley and Peter G. Neumann. "A General-purpose File System for Secondary Storage". In: *Proceedings of the 4th AFIPS Fall Joint Computer Conference (FJCC)*. Las Vegas, Nevada, USA, Nov. 1965, pp. 213–229 (cited on page 80).

[DNN+15] Aleksandar Dragojević, Dushyanth Narayanan, Edmund B. Nightingale, Matthew Renzelmann, Alex Shamis, Anirudh Badam, and Miguel Castro. "No Compromises: Distributed Transactions with Consistency, Availability, and Performance". In: *Proceedings of the 25th Symposium on Operating Systems Principles (SOSP)*. Monterey, California, 2015, pp. 54–70 (cited on pages 50, 81).

[DSK15] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. "Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters". In: *Proceedings of the 6th ACM Symposium on Cloud Computing (SoCC)*. Kohala Coast, Hawaii, USA, Aug. 2015, pp. 97–110 (cited on pages 56, 154, 191, 194).

[DV66] Jack B. Dennis and Earl C. Van Horn. "Programming Semantics for Multiprogrammed Computations". In: *Communications of the ACM* 9.3 (Mar. 1966), pp. 143–155 (cited on page 50).

[Edg14] Jake Edge. *A look at The Machine*. Article in Linux Weekly News (LWN); accessed 30/08/2015. Feb. 2014. URL: https://lwn.net/Articles/655437/ (cited on page 202).

[EFF13] Electronic Frontier Foundation (EFF). *EFF's Encrypt The Web Report*. Accessed on 12/01/2015. Nov. 2013. URL: https://www.eff.org/encrypt-the-web-report (cited on page 78).

[EIS75] Shimon Even, Alon Itai, and Adi Shamir. "On the complexity of time table and multi-commodity flow problems". In: *Proceedings of the 16th Annual Symposium on Foundations of Computer Science (FOCS)*. Berkeley, California, USA, Oct. 1975, pp. 184–193 (cited on page 269).

[ELF-64] AT&T, The Santa Cruz Operation, Inc., Caldera International and The SCO Group. "Chapter 4 – Object Files". In: *System V Application Binary Interface*. Snapshot of 31/12/2012, http://www.sco.com/developers/gabi/2012-12-31/contents.html (cited on page 118).

[ELL+07] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. "SNZI: Scalable NonZero Indicators". In: *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. Portland, Oregon, USA, Aug. 2007, pp. 13–22 (cited on page 106).

[Eri13] Marius Eriksen. "Your Server As a Function". In: *Proceedings of the 7th Workshop on Programming Languages and Operating Systems (PLOS)*. Farmington, Pennsylvania, USA, Nov. 2013, 5:1–5:7 (cited on pages 34, 47, 200).

[FAK+12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, et al. "Clearing the clouds: a study of emerging scale-out workloads on modern hardware". In: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. London, England, United Kingdom, Mar. 2012, pp. 37–48 (cited on page 38).

[FBK+12] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. "Jockey: Guaranteed Job Latency in Data Parallel Clusters". In: *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys)*. Bern, Switzerland, Apr. 2012, pp. 99–112 (cited on pages 56, 61, 191).

[FBSD-HB] The FreeBSD Documentation Project. *FreeBSD Handbook*. Revision 45698; accessed on 05/01/2015. 2014. URL: `https://www.freebsd.org/doc/handbook/index.html` (cited on page 118).

[FGB11] Antonio Frangioni, Claudio Gentile, and Dimitri P. Bertsekas. *MCFClass RelaxIV, version 1.80*. Accessed on 06/07/2015. Feb. 2011. URL: `http://www.di.unipi.it/optimize/Software/MCF.html#RelaxIV` (cited on page 192).

[FH67] Kurt Fuchel and Sidney Heller. "Consideration in the Design of a Multiple Computer System with Extended Core Storage". In: *Proceedings of the 1st ACM Symposium on Operating System Principles (SOSP)*. Gatlinburg, Tennessee, USA, Oct. 1967, pp. 17.1–17.13 (cited on page 43).

[FHI+11] Dennis Fetterly, Maya Haridasan, Michael Isard, and Swaminathan Sundararaman. "TidyFS: A simple and small distributed filesystem". In: *Proceedings of the USENIX Annual Technical Conference*. Portland, Oregon, USA, June 2011, pp. 34–48 (cited on pages 33, 79, 265).

[FM06] Antonio Frangioni and Antonio Manca. "A Computational Study of Cost Reoptimization for Min-Cost Flow Problems". In: *INFORMS Journal on Computing* 18.1 (2006), pp. 61–70 (cited on page 150).

[GGL03] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. "The Google File System". In: *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP)*. Bolton Landing, NY, USA, Oct. 2003, pp. 29–43 (cited on pages 31, 33, 47, 79, 265).

[GK93] Andrew V. Goldberg and Michael Kharitonov. "On Implementing Scaling Push-Relabel Algorithms for the Minimum-Cost Flow Problem". In: *Network Flows and Matching: First DIMACS Implementation Challenge*. Edited by D.S. Johnson and C.C. McGeoch. DIMACS series in discrete mathemat-

ics and theoretical computer science. American Mathematical Society, 1993 (cited on page 150).

[Gle15] Adam Gleave. "Fast and accurate cluster scheduling using flow networks". Computer Science Tripos Part II Dissertation. University of Cambridge Computer Laboratory, May 2015 (cited on pages 25, 149, 151, 153, 192).

[GLG+12] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. "PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs". In: *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Hollywood, California, USA, Oct. 2012, pp. 17–30 (cited on page 266).

[Gol08] Andrew V. Goldberg. "The Partial Augment-Relabel Algorithm for the Maximum Flow Problem". In: *Algorithms*. Edited by Dan Halperin and Kurt Mehlhorn. Volume 5193. Lecture Notes in Computer Science. Springer, 2008, pp. 466–477 (cited on page 263).

[Gol87] Andrew V. Goldberg. "Efficient Graph Algorithms for Sequential and Parallel Computers". Published as MIT/LCS/TR-374. PhD thesis. Massachusetts Institute of Technology, Feb. 1987 (cited on page 149).

[Gol97] Andrew V. Goldberg. "An Efficient Implementation of a Scaling Minimum-Cost Flow Algorithm". In: *Journal of Algorithms* 22.1 (1997), pp. 1–29 (cited on pages 150, 152, 192, 261–263).

[GS62] David Gale and Lloyd S. Shapley. "College Admissions and the Stability of Marriage". In: *The American Mathematical Monthly* 69.1 (Jan. 1962), pp. 9–15 (cited on page 190).

[GSC+15] Ionel Gog, Malte Schwarzkopf, Natacha Crooks, Matthew P. Grosvenor, Allen Clement, and Steven Hand. "Musketeer: all for one, one for all in data processing systems". In: *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*. Bordeaux, France, Apr. 2015 (cited on page 27).

[GSG+15] Matthew P. Grosvenor, Malte Schwarzkopf, Ionel Gog, Robert N. M. Watson, Andrew W. Moore, Steven Hand, and Jon Crowcroft. "Queues don't matter when you can JUMP them!" In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, California, USA, May 2015 (cited on pages 27, 111).

[GSG+16] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. "Firmament: fast, centralized cluster scheduling at scale". In: *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. To appear. Savannah, Georgia, USA, Nov. 2016 (cited on pages 27, 151, 153, 193–194).

[GSW15] Andrey Goder, Alexey Spiridonov, and Yin Wang. "Bistro: Scheduling Data-Parallel Jobs Against Live Production Systems". In: *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA, July 2015, pp. 459–471 (cited on pages 32, 56, 191).

[GT14] Andrew V. Goldberg and Robert E. Tarjan. "Efficient Maximum Flow Algorithms". In: *Communications of the ACM* 57.8 (Aug. 2014), pp. 82–89 (cited on page 263).

[GT89] Andrew V. Goldberg and Robert E. Tarjan. "Finding Minimum-cost Circulations by Canceling Negative Cycles". In: *Journal of the ACM* 36.4 (Oct. 1989), pp. 873–886 (cited on pages 149–150).

[GT90] Andrew V. Goldberg and Robert E. Tarjan. "Finding Minimum-Cost Circulations by Successive Approximation". In: *Mathematics of Operations Research* 15.3 (Aug. 1990), pp. 430–466 (cited on page 150).

[GZH+11] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. "Dominant resource fairness: fair allocation of multiple resource types". In: *Proceedings of the 8th USENIX Symposium on Networked System Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Mar. 2011 (cited on pages 60, 146, 179).

[GZS+13] Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. "Choosy: max-min fair sharing for datacenter jobs with constraints". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 365–378 (cited on pages 56, 59–60, 146, 191).

[Han99] Steven M. Hand. "Self-Paging in the Nemesis Operating System". In: *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. New Orleans, Louisiana, USA, Feb. 1999, pp. 73–86 (cited on page 200).

[HBB+12] Alexander Hall, Olaf Bachmann, Robert Büssow, Silviu Gănceanu, and Marc Nunkesser. "Processing a Trillion Cells Per Mouse Click". In: *Proceedings of the VLDB Endowment* 5.11 (July 2012), pp. 1436–1446 (cited on pages 31, 33).

[HBD+14] Tyler Harter, Dhruba Borthakur, Siying Dong, Amitanand Aiyer, Liyin Tang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Analysis of HDFS Under HBase: A Facebook Messages Case Study". In: *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST)*. Santa Clara, California, USA, Feb. 2014, pp. 199–212 (cited on page 33).

[HDFSDocs15] Apache Software Foundation. *HDFS Permissions Guide*. Hadoop project online documentation for v2.7.1. 2015. URL: http://hadoop.apache.org/docs/r2.7.1/hadoop-project-dist/hadoop-hdfs/HdfsPermissionsGuide.html (cited on page 129).

[HDV+12] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. "Measuring Energy Consumption for Short Code Paths Using RAPL". In: *SIGMETRICS Performance Evaluation Review* 40.3 (Jan. 2012), pp. 13–17 (cited on pages 172, 186).

[Hei98] Heiser, G. and Elphinstone, K. and Vochteloo, J. and Russell, S. and Liedtke, J. "The Mungi single-address-space operating system". In: *Software: Practice and Experience* 28.9 (July 1998), pp. 901–928 (cited on pages 67, 80).

[Hel14] Gustaf Helgesson. "Energy-Aware Scheduling in Heterogeneous Data Centers". Master's thesis. University of Cambridge Computer Laboratory, June 2014 (cited on pages 171, 185).

[Hel54] Frank C. Helwig. *Memorandum DCL-24: Director Tapes*. Technical report. Massachusetts Institute of Technology, Digital Computer Laboratory, Nov. 1954 (cited on page 43).

[Hen06] John L. Henning. "SPEC CPU2006 Benchmark Descriptions". In: *SIGARCH Computer Architecture News* 34.4 (Sept. 2006), pp. 1–17 (cited on page 36).

[HFS] Apache Hadoop. *Hadoop Fair Scheduler*. Accessed 13/03/2014. URL: http://hadoop.apache.org/common/docs/stable1/fair_scheduler.html (cited on pages 56, 60, 191).

[HGZ11] Jonas Hauenstein, David Gerhard, and Gerd Zellweger. *Ethernet Message Passing for Barrelfish*. Technical report. ETH Zürich, July 2011 (cited on page 23).

[HHL+97] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. "The Performance of μ-kernel-based Systems". In: *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. Saint Malo, France, Oct. 1997, pp. 66–77 (cited on page 116).

[HKJ+10] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. "ZooKeeper: Wait-free Coordination for Internet-scale Systems". In: *Proceedings of the USENIX Annual Technical Conference*. Boston, Massachusetts, USA, June 2010, pp. 149–158 (cited on pages 32, 47, 107).

[HKZ+11] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D. Joseph, Randy Katz, Scott Shenker, et al. "Mesos: A platform for fine-grained resource sharing in the data center". In: *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation (NSDI)*.

Boston, Massachusetts, USA, Mar. 2011, pp. 295–308 (cited on pages 30, 32, 47, 49, 56–58, 60, 122, 146, 153, 191).

[HLN68] David F. Hartley, Barry Landy, and Roger M. Needham. "The structure of a multiprogramming supervisor". In: *The Computer Journal* 11.3 (1968), pp. 247–255 (cited on page 43).

[HMM14] Tim Harris, Martin Maas, and Virendra Marathe. "Callisto: co-scheduling parallel runtime systems". In: *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*. Amsterdam, Netherlands, Apr. 2014 (cited on pages 40, 136).

[HPD13] Jon Howell, Bryan Parno, and John R. Douceur. "Embassies: Radically Refactoring the Web." In: *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Lombard, Illinois, USA, Apr. 2013, pp. 529–545 (cited on pages 52, 103).

[IBY+07] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. "Dryad: Distributed Data-parallel Programs from Sequential Building Blocks". In: *Proceedings of the 2nd ACM SIGOPS European Conference on Computer Systems (EuroSys)*. Lisbon, Portugal, Mar. 2007, pp. 59–72 (cited on pages 33, 47, 58, 67, 70, 136).

[IPC+09] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. "Quincy: fair scheduling for distributed computing clusters". In: *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*. Big Sky, Montana, USA, Oct. 2009, pp. 261–276 (cited on pages 25–26, 56, 58–60, 133, 135–136, 144, 146, 151, 191–192, 198, 239, 263–265, 267, 269).

[JR86] Michael B. Jones and Richard F. Rashid. "Mach and Matchmaker: Kernel and Language Support for Object-oriented Distributed Systems". In: *Proceedings of the 1986 Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA)*. Portland, Oregon, USA, Nov. 1986, pp. 67–77 (cited on page 45).

[JSC01] David Jackson, Quinn Snell, and Mark Clement. "Core Algorithms of the Maui Scheduler". English. In: *Job Scheduling Strategies for Parallel Processing*. Edited by Dror G. Feitelson and Larry Rudolph. Volume 2221. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, pp. 87–102 (cited on page 56).

[KBF+15] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M. Patel, et al. "Twitter Heron: Stream Processing at Scale". In: *Proceedings of the 2015 ACM SIGMOD Interna-*

tional Conference on Management of Data (SIGMOD). Melbourne, Victoria, Australia, 2015, pp. 239–250 (cited on page 34).

[KEH+09] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP)*. Big Sky, Montana, USA, Oct. 2009, pp. 207–220 (cited on page 79).

[Kes99] Richard E. Kessler. "The Alpha 21264 microprocessor". In: *IEEE Micro* 19.2 (Mar. 1999), pp. 24–36 (cited on page 239).

[KK12] Zoltán Király and P. Kovács. "Efficient implementations of minimum-cost flow algorithms". In: *CoRR* abs/1207.6381 (2012) (cited on page 263).

[KKT+16] Ana Klimovic, Christos Kozyrakis, Eno Thereska, Binu John, and Sanjeev Kumar. "Flash Storage Disaggregation". In: *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*. London, United Kingdom, 2016, 29:1–29:15 (cited on page 58).

[Kle67] Morton Klein. "A Primal Method for Minimal Cost Flows with Applications to the Assignment and Transportation Problems". In: *Management Science* 14.3 (1967), pp. 205–220 (cited on page 149).

[KLS86] Nancy P. Kronenberg, Henry M. Levy, and William D. Strecker. "VAXcluster: A Closely-coupled Distributed System". In: *ACM Transactions on Computer Systems* 4.2 (May 1986), pp. 130–146 (cited on pages 44–45).

[KPH61] Tom Kilburn, R. Bruce Payne, and David J. Howarth. "The Atlas Supervisor". In: *Proceedings of the 1961 Eastern Joint Computer Conference (EJCC)*. Washington D.C., USA, Dec. 1961, pp. 279–294 (cited on page 43).

[KRB+03] Alan H. Karp, J. Rozas Guillermo, Arindam Banerji, and Rajiv Gupta. "Using split capabilities for access control". In: *IEEE Software* 20.1 (Jan. 2003), pp. 42–49 (cited on pages 75, 91, 96).

[KRC+15] Konstantinos Karanasos, Sriram Rao, Carlo Curino, Chris Douglas, Kishore Chaliparambil, Giovanni Matteo Fumarola, Solom Heddaya, et al. "Mercury: Hybrid Centralized and Distributed Scheduling in Large Shared Clusters". In: *Proceedings of the USENIX Annual Technical Conference*. Santa Clara, California, USA, July 2015, pp. 485–497 (cited on pages 56, 191).

[KRT94] Valerie King, S. Rao, and Robert E. Tarjan. "A Faster Deterministic Maximum Flow Algorithm". In: *Journal of Algorithms* 17.3 (1994), pp. 447–474 (cited on page 263).

[KT91] M. Frans Kaashoek and Andrew S. Tanenbaum. "Group communication in the Amoeba distributed operating system". In: *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)*. May 1991, pp. 222–230 (cited on page 112).

[Lam13] Christoph Lameter. "NUMA (Non-Uniform Memory Access): An Overview". In: *ACM Queue* 11.7 (July 2013), 40:40–40:51 (cited on page 135).

[Lam78] Leslie Lamport. "Time, Clocks, and the Ordering of Events in a Distributed System". In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565 (cited on page 108).

[LCG+15] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. "Heracles: Improving Resource Efficiency at Scale". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA)*. Portland, Oregon, USA, June 2015, pp. 450–462 (cited on page 201).

[Lev84] Henry M. Levy. *Capability-based Computer Systems*. Digital Press, 1984 (cited on pages 50, 76, 79).

[LGF+82] Keith A. Lantz, Klaus D. Gradischnig, Jerome A. Feldman, and Richard F. Rashid. "Rochester's intelligent gateway". In: *Computer* 15.10 (Dec. 1982), pp. 54–68 (cited on pages 44–45).

[LGZ+14] Haoyuan Li, Ali Ghodsi, Matei Zaharia, Scott Shenker, and Ion Stoica. "Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks". In: *Proceedings of the 5th ACM Symposium on Cloud Computing (SoCC)*. Seattle, Washington, USA, 2014, 6:1–6:15 (cited on pages 47, 49, 58, 75, 200, 266).

[LHA+14] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. "MICA: A Holistic Approach to Fast In-memory Key-value Storage". In: *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Seattle, Washington, USA, Apr. 2014, pp. 429–444 (cited on page 125).

[LHM+15] Chang Liu, Austin Harris, Martin Maas, Michael Hicks, Mohit Tiwari, and Elaine Shi. "GhostRider: A Hardware-Software System for Memory Trace Oblivious Computation". In: *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Istanbul, Turkey, 2015, pp. 87–101 (cited on page 69).

[LK14] Jacob Leverich and Christos Kozyrakis. "Reconciling High Server Utilization and Sub-millisecond Quality-of-Service". In: *Proceedings of the 9th ACM European Conference on Computer Systems (EuroSys)*. Amsterdam, The Netherlands, Apr. 2014 (cited on page 41).

[LLA+81] Edward D. Lazowska, Henry M. Levy, Guy T. Almes, Michael J. Fischer, Robert J. Fowler, and Stephen C. Vestal. "The Architecture of the Eden System". In: *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*. Pacific Grove, California, USA, Dec. 1981, pp. 148–159 (cited on page 76).

[LM10] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Operating Systems Review* 44.2 (Apr. 2010), pp. 35–40 (cited on page 70).

[LM88] Malcolm G. Lane and James D. Mooney. *A practical approach to operating systems*. Boyd & Fraser Publishing Company, 1988 (cited on page 43).

[Löb96] Andreas Löbel. *Solving Large-Scale Real-World Minimum-Cost Flow Problems by a Network Simplex Method*. Technical report SC-96-07. Zentrum für Informationstechnik Berlin (ZIB), Feb. 1996 (cited on page 150).

[Low95] Gavin Lowe. "An attack on the Needham-Schroeder public-key authentication protocol". In: *Information Processing Letters* 56.3 (1995), pp. 131–133 (cited on page 78).

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. "Pregel: A System for Large-scale Graph Processing". In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Indianapolis, Indiana, USA, June 2010, pp. 135–146 (cited on pages 31, 34).

[MAH+14] Martin Maas, Krste Asanović, Tim Harris, and John Kubiatowicz. "The Case for the Holistic Language Runtime System". In: *Proceedings of the 4th Workshop on Systems for Future Multicore Architectures (SFMA)*. Amsterdam, The Netherlands, Apr. 2014, pp. 7–14 (cited on pages 22, 34, 202).

[MAP+15] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Whitepaper available at `http://tensorflow.org`. 2015 (cited on page 58).

[Mar12] Jason Mars. "Rethinking the Architecture of Warehouse-Scale Computers". PhD thesis. University of Virginia, May 2012 (cited on page 160).

[McC95] John D. McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* (Dec. 1995), pp. 19–25 (cited on page 36).

[Men07] Paul B. Menage. "Adding generic process containers to the Linux kernel". In: *Proceedings of the Ottawa Linux Symposium (OLS)*. Volume 2. June 2007, pp. 45–57 (cited on page 46).

[Mer14] Dirk Merkel. "Docker: Lightweight Linux Containers for Consistent Development and Deployment". In: *Linux Journal* 2014.239 (Mar. 2014) (cited on page 46).

[MesosDocs15] Apache Software Foundation. *Mesos High-Availability Mode*. Mesos project online documentation; accessed 20/09/2015. URL: `http://mesos.apache.org/documentation/latest/high-availability/` (cited on page 50).

[MG12] Raghotham Murthy and Rajat Goel. "Peregrine: Low-latency Queries on Hive Warehouse Data". In: *XRDS: Crossroad, ACM Magazine for Students* 19.1 (Sept. 2012), pp. 40–43 (cited on page 32).

[MGL+10] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. "Dremel: Interactive Analysis of Web-scale Datasets". In: *Proceedings of the VLDB Endowment* 3.1-2 (Sept. 2010), pp. 330–339 (cited on pages 31, 33).

[MHC+10] Asit K. Mishra, Joseph L. Hellerstein, Walfredo Cirne, and Chita R. Das. "Towards Characterizing Cloud Backend Workloads: Insights from Google Compute Clusters". In: *SIGMETRICS Performance Evalution Review* 37.4 (Mar. 2010), pp. 34–41 (cited on page 59).

[MLR+14] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, et al. "f4: Facebook's Warm BLOB Storage System". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 383–398 (cited on pages 32–33, 47, 79).

[MLS+13] Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, et al. "PHANTOM: Practical Oblivious Computation in a Secure Processor". In: *Proceedings of the 2013 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. Berlin, Germany, 2013, pp. 311–324 (cited on page 69).

[MMI+13] Derek G. Murray, Frank McSherry, Rebecca Isaacs, Michael Isard, Paul Barham, and Martın Abadi. "Naiad: A Timely Dataflow System". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Nemacolin Woodlands, Pennsylvania, USA, Nov. 2013, pp. 439–455 (cited on pages 40, 70, 266).

[MMK10] Yandong Mao, Robert Morris, and M. Frans Kaashoek. *Optimizing MapReduce for multicore architectures*. Technical report MIT-CSAIL-TR-2010-020. Massachusetts Institute of Technology, Computer Science and Artificial Intelligence Laboratory, 2010 (cited on page 125).

[MMP83] Erik T. Mueller, Johanna D. Moore, and Gerald J. Popek. "A Nested Transaction Mechanism for LOCUS". In: *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*. Bretton Woods, New Hampshire, USA, Oct. 1983, pp. 71–89 (cited on page 82).

[MRT+90] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robert van Renesse, and Hans van Staveren. "Amoeba: a distributed operating system for the 1990s". In: *Computer* 23.5 (May 1990), pp. 44–53 (cited on pages 44–45, 76).

[MSB10] Andreas Merkel, Jan Stoess, and Frank Bellosa. "Resource-conscious Scheduling for Energy Efficiency on Multicore Processors". In: *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*. Paris, France, Apr. 2010, pp. 153–166 (cited on pages 237, 239).

[MSS+11] Derek G. Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. "CIEL: a universal execution engine for distributed data-flow computing". In: *Proceedings of the 8th USENIX Symposium on Networked System Design and Implementation (NSDI)*. Boston, Massachusetts, USA, Mar. 2011, pp. 113–126 (cited on pages 28, 58, 67, 70, 75, 92, 122).

[MT13] Jason Mars and Lingjia Tang. "Whare-map: Heterogeneity in "Homogeneous" Warehouse-scale Computers". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*. Tel-Aviv, Israel, June 2013, pp. 619–630 (cited on pages 56, 135, 159–161, 164, 191).

[MTH+11] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations". In: *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Porto Allegre, Brazil, Dec. 2011, pp. 248–259 (cited on page 40).

[Mur11] Derek G. Murray. "A distributed execution engine supporting data-dependent control flow". PhD thesis. University of Cambridge Computer Laboratory, July 2011 (cited on pages 56, 75, 80, 122, 191).

[MVH+10] Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. "Contention Aware Execution: Online Contention Detection and Response". In: *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Toronto, Ontario, Canada, Apr. 2010, pp. 257–265 (cited on page 157).

[MWH14] Ilias Marinos, Robert N.M. Watson, and Mark Handley. "Network Stack Specialization for Performance". In: *Proceedings of the ACM SIGCOMM 2014 Conference (SIGCOMM)*. Chicago, Illinois, USA, Aug. 2014, pp. 175–186 (cited on page 125).

[NEF+12] Edmund B. Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. "Flat Datacenter Storage". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Hollywood, California, USA, Oct. 2012, pp. 1–15 (cited on pages 33, 79, 265).

[Nev12] Mark Nevill. "An Evaluation of Capabilities for a Multikernel". Master's thesis. ETH Zürich, May 2012 (cited on page 79).

[NFG+13] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, et al. "Scaling Memcache at Facebook". In: *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. Lombard, Illinois, USA, Apr. 2013, pp. 385–398 (cited on pages 32–33, 47, 49, 124).

[NRN+10] Leonardo Neumeyer, Bruce Robbins, Anish Nair, and Anand Kesari. "S4: Distributed Stream Computing Platform". In: *Proceedings of the 2010 IEEE International Conference on Data Mining Workshops (ICDMW)*. Sydney, Australia, Dec. 2010, pp. 170–177 (cited on page 34).

[NSW+15] Mihir Nanavati, Malte Schwarzkopf, Jake Wires, and Andrew Warfield. "Non-volatile Storage". In: *Communications of the ACM* 59.1 (Dec. 2015), pp. 56–63 (cited on page 58).

[OCD+88] John K. Ousterhout, Andrew R. Cherenson, Fred Douglis, Michael N. Nelson, and Brent B. Welch. "The Sprite network operating system". In: *Computer* 21.2 (Feb. 1988), pp. 23–36 (cited on pages 44–45).

[OO14] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *Proceedings of the USENIX Annual Technical Conference*. Philadelphia, Pennsylvania, USA, June 2014, pp. 305–319 (cited on pages 32, 48).

[OPR+13] Kay Ousterhout, Aurojit Panda, Joshua Rosen, Shivaram Venkataraman, Reynold Xin, Sylvia Ratnasamy, Scott Shenker, et al. "The case for tiny tasks in compute clusters". In: *Proceedings of the 14th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. Santa Ana Pueblo, New Mexico, USA, May 2013 (cited on pages 57, 122).

[Orl13] James B. Orlin. "Max Flows in O(nm) Time, or Better". In: *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*. Palo Alto, California, USA, June 2013, pp. 765–774 (cited on page 263).

[Orl93] James B. Orlin. "A Faster Strongly Polynomial Minimum Cost Flow Algorithm". In: *Operations Research* 41.2 (Apr. 1993), pp. 338–350 (cited on page 149).

[Orl97] James B. Orlin. "A polynomial time primal network simplex algorithm for minimum cost flows". In: *Mathematical Programming* 78.2 (Aug. 1997), pp. 109–129 (cited on pages 149–150).

[ORR+15] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. "Making Sense of Performance in Data Analytics Frameworks". In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, California, USA, May 2015, pp. 293–307 (cited on pages 127, 178).

[ORS+11] Diego Ongaro, Stephen M. Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. "Fast Crash Recovery in RAMCloud". In: *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*. Cascais, Portugal, Oct. 2011, pp. 29–41 (cited on pages 50, 58, 110).

[OSS80] John K. Ousterhout, Donald A. Scelza, and Pradeep S. Sindhu. "Medusa: An Experiment in Distributed Operating System Structure". In: *Communications of the ACM* 23.2 (Feb. 1980), pp. 92–105 (cited on page 44).

[OWZ+13] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. "Sparrow: Distributed, Low Latency Scheduling". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Nemacolin Woodlands, Pennsylvania, USA, Nov. 2013, pp. 69–84 (cited on pages 56–57, 59–60, 134, 146, 154, 191, 194).

[OZN+12] Zhonghong Ou, Hao Zhuang, Jukka K. Nurminen, Antti Ylä-Jääski, and Pan Hui. "Exploiting Hardware Heterogeneity Within the Same Instance Type of Amazon EC2". In: *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. Boston, Massachusetts, USA, June 2012 (cited on page 35).

[PBH+11] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. "Rethinking the Library OS from the Top Down". In: *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Newport Beach, California, USA, Mar. 2011, pp. 291–304 (cited on pages 52, 103).

[PCC+14] Andrew Putnam, Adrian Caulfield, Eric Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, et al. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". In: *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*. Minneapolis, Minnesota, USA, June 2014 (cited on page 202).

[PD10] Daniel Peng and Frank Dabek. "Large-scale Incremental Processing Using Distributed Transactions and Notifications". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. Vancouver, British Columbia, Canada, Oct. 2010, pp. 1–15 (cited on pages 31, 34).

[Pik00] Rob Pike. *Systems Software Research is Irrelevant*. Slide deck; accessed on 01/01/2015. Feb. 2000. URL: `http://herpolhode.com/rob/utah2000.pdf` (cited on page 19).

[PLZ+14] Simon Peter, Jialin Li, Irene Zhang, Dan R. K. Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, et al. "Arrakis: The Operating System is the Control Plane". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 1–16 (cited on page 24).

[POB+14] Jonathan Perry, Amy Ousterhout, Hari Balakrishnan, Devavrat Shah, and Hans Fugal. "Fastpass: A Zero-Queue Datacenter Network Architecture". In: *Proceedings of the ACM SIGCOMM 2014 Conference (SIGCOMM)*. Chicago, Illinois, USA, Aug. 2014 (cited on page 111).

[PPD+95] Rob Pike, Dave Presottó, Sean Dorward, Bob Flandrena, Key Thompson, Howard Trickey, and Phil Winterbottom. *Plan 9 from Bell Labs*. Technical report. AT&T Bell Laboratories, Murray Hill, NJ. 1995 (cited on pages 44, 46).

[PPT+93] Rob Pike, Dave Presottó, Ken Thompson, Howard Trickey, and Phil Winterbottom. "The Use of Name Spaces in Plan 9". In: *SIGOPS Operating Systems Review* 27.2 (Apr. 1993), pp. 72–76 (cited on page 46).

[Pro54] Project Whirlwind. *Summary Report No. 39, Third Quarter 1954*. Technical report. Massachusetts Institute of Technology, Digital Computer Laboratory, Oct. 1954 (cited on page 43).

[PS85] James L. Peterson and Abraham Silberschatz. *Operating Systems Concepts*. Second edition. Addison-Wesley, 1985 (cited on page 136).

[RAA+91] Marc Rozier, Vadim Abrossimov, François Armand, Ivan Boule, Michael Gien, Marc Guillemont, Frédéric Herrmann, et al. *Overview of the CHORUS Distributed Operating Systems*. Technical report CS/TR-90-25.1. Chorus systèmes, Feb. 1991 (cited on pages 44–45).

[Ras86] Richard F. Rashid. "From RIG to Accent to Mach: The Evolution of a Network Operating System". In: *Proceedings of 1986 ACM Fall Joint Computer Conference (FJCC)*. Dallas, Texas, USA, Nov. 1986, pp. 1128–1137 (cited on pages 44–45).

[RDS15] Oracle, Inc. *RDS Wire Specification 3.1*. Online documentation, Document Version 1.0.1; accessed 07/08/2015. Apr. 2015. URL: https://oss.oracle.com/projects/rds/dist/documentation/rds-3.1-spec.html (cited on page 110).

[Ric41] Grant Richards. *Housman 1897–1936*. Oxford, United Kingdom: Oxford University Press, 1941 (cited on page 6).

[Riz12] Luigi Rizzo. "netmap: A Novel Framework for Fast Packet I/O." In: *Proceedings of the USENIX Annual Technical Conference*. Boston, Massachusetts, USA, June 2012, pp. 101–112 (cited on page 253).

[RJ08] Benjamin Reed and Flavio P. Junqueira. "A Simple Totally Ordered Broadcast Protocol". In: *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*. Yorktown Heights, New York, USA, Dec. 2008, 2:1–2:6 (cited on page 32).

[RKZ+11] Barret Rhoden, Kevin Klues, David Zhu, and Eric Brewer. "Improving Pernode Efficiency in the Datacenter with New OS Abstractions". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*. Cascais, Portugal, Oct. 2011, 25:1–25:8 (cited on page 24).

[ROS+11] Stephen M. Rumble, Diego Ongaro, Ryan Stutsman, Mendel Rosenblum, and John K. Ousterhout. "It's Time for Low Latency". In: *Proceedings of the 13th USENIX Workshop on Hot Topics in Operating Systems (HotOS)*. Napa, California, USA, May 2011, pp. 11–15 (cited on pages 47, 125).

[Ros69] Robert F. Rosin. "Supervisory and Monitor Systems". In: *ACM Computing Surveys* 1.1 (Mar. 1969), pp. 37–54 (cited on page 42).

[Ros86] Douglas Ross. "A personal view of the personal work station: some firsts in the Fifties". In: *Proceedings of the ACM Conference on the history of personal workstations (HPW)*. Palo Alto, California, USA, Jan. 1986, pp. 19–48 (cited on page 43).

[Roz93] Guillermo J. Rozas. *Translucent Procedures, Abstraction without Opacity*. Technical report No. 1427. MIT AI Laboratory, Oct. 1993 (cited on page 67).

[RR81] Richard F. Rashid and George G. Robertson. "Accent: A Communication Oriented Network Operating System Kernel". In: *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP)*. Pacific Grove, California, USA, Dec. 1981, pp. 64–75 (cited on pages 44–45, 76, 79).

[RT74] Dennis M. Ritchie and Ken Thompson. "The UNIX Time-sharing System". In: *Communications of the ACM* 17.7 (July 1974), pp. 365–375 (cited on pages 21, 43).

[RTG+12] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. "Heterogeneity and dynamicity of clouds at scale: Google trace analysis". In: *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*. San Jose, California, Oct. 2012, 7:1–7:13 (cited on pages 151, 168, 177, 269).

[RTM+10] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. "Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers". In: *IEEE Micro* 30.4 (July 2010), pp. 65–79 (cited on page 161).

[Rust14] *The Rust Programming Language*. Accessed on 09/12/2014. 2014. URL: `http://www.rust-lang.org` (cited on page 199).

[RWH11] Charles Reiss, John Wilkes, and Joseph L. Hellerstein. *Google cluster-usage traces: format + schema*. Technical report. Revised 2012.03.20. Posted at URL `http://code.google.com/p/googleclusterdata/wiki/TraceVersion2`. Mountain View, California, USA: Google Inc., Nov. 2011 (cited on pages 35, 185, 192, 239, 268).

[RYC+13] Christopher J. Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. "Dandelion: A Compiler and Runtime for Heterogeneous Systems". In: *Proceedings of the 24th ACM Symposium on Operating Systems Principles (SOSP)*. Nemacolin Woodlands, Pennsylvania, USA, Nov. 2013, pp. 49–68 (cited on pages 58, 202).

[SAA+15] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, et al. "Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services". In: *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. Oakland, California, USA, May 2015, pp. 351–366 (cited on page 32).

[SBB+10] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, et al. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Technical report. Google, Inc., 2010 (cited on pages 31, 34, 47).

[SCH+11] Bikash Sharma, Victor Chudnovsky, Joseph L. Hellerstein, Rasekh Rifaat, and Chita R. Das. "Modeling and synthesizing task placement constraints in Google compute clusters". In: *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC)*. Cascais, Portugal, Oct. 2011, 3:1–3:14 (cited on pages 59, 145).

[Scu15] Andrew M. Scull. "Hephaestus: a Rust runtime for a distributed operating system". Computer Science Tripos Part II Dissertation. University of Cambridge Computer Laboratory, May 2015 (cited on pages 25, 200).

[SEL4RM] Philip Derrin, Dhammika Elkaduwe, and Kevin Elphinstone. *seL4 Reference Manual*. Technical report. version sel4@nicta.com.au-devel/l4cap-haskell–0.4-patch-304. NICTA (cited on page 79).

[SGG08] Abraham Silberschatz, Peter B. Galvin, and Greg Gange. *Operating Systems Concepts*. Eight edition. John Wiley & sons, July 2008 (cited on pages 67, 79).

[SGH13] Malte Schwarzkopf, Matthew P. Grosvenor, and Steven Hand. "New Wine in Old Skins: The Case for Distributed Operating Systems in the Data Center". In: *Proceedings of the 4th Asia-Pacific Systems Workshop (APSYS)*. Singapore, July 2013, 9:1–9:7 (cited on page 27).

[SH12] Malte Schwarzkopf and Steven Hand. "A down-to-earth look at the cloud host OS". In: *Proceedings of the 1st Workshop on Hot Topics in Cloud Data Processing (HotCDP)*. Bern, Switzerland, Apr. 2012, 3:1–3:5 (cited on page 27).

[SKA+13] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. "Omega: flexible, scalable schedulers for large compute clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 351–364 (cited on pages 27, 30–32, 43, 56–57, 60–61, 144, 153–154, 191, 194, 269).

[SKR+10] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. "The Hadoop Distributed File System". In: *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST)*. Incline Village, Nevada, USA, May 2010, pp. 1–10 (cited on pages 22, 32, 47, 49, 79).

[SMH11] Malte Schwarzkopf, Derek G. Murray, and Steven Hand. "Condensing the cloud: running CIEL on many-core". In: *Proceedings of the 1st Workshop on Systems for Future Multicore Architectures (SFMA)*. Salzburg, Austria, Apr. 2011 (cited on page 27).

[SMH12] Malte Schwarzkopf, Derek G. Murray, and Steven Hand. "The Seven Deadly Sins of Cloud Computing Research". In: *Proceedings of the 4th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. Boston, Massachusetts, USA, June 2012, pp. 1–6 (cited on pages 27, 125).

[SNS88] Jennifer G. Steiner, B. Clifford Neuman, and Jeffrey I. Schiller. "Kerberos: An Authentication Service for Open Network Systems." In: *Proceedings of the USENIX Winter Conference*. Dallas, Texas, USA, Feb. 1988, pp. 191–202 (cited on page 78).

[SOE+12] 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-*

MOD International Conference on Management of Data (SIGMOD). Scottsdale, Arizona, USA, May 2012, pp. 777–778 (cited on page 31).

[SparkDocs15] Apache Software Foundation. *Spark Standalone Mode – High Availability*. Spark project online documentation for v1.5.1. 2015. URL: `http://spark.apache.org/docs/1.5.1/spark-standalone.html#high-availability` (cited on page 50).

[SS72] Michael D. Schroeder and Jerome H. Saltzer. "A Hardware Architecture for Implementing Protection Rings". In: *Communications of the ACM* 15.3 (Mar. 1972), pp. 157–170 (cited on page 43).

[SS75] Jerome H. Saltzer and Michael D. Schroeder. "The protection of information in computer systems". In: *Proceedings of the IEEE* 63.9 (Sept. 1975), pp. 1278–1308 (cited on page 50).

[SSF99] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. "EROS: A Fast Capability System". In: *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP)*. Charleston, South Carolina, USA, Dec. 1999, pp. 170–185 (cited on page 79).

[ST97] Nir Shavit and Dan Touitou. "Software transactional memory". In: *Distributed Computing* 10.2 (1997), pp. 99–116 (cited on page 82).

[Ste64] Thomas B. Steel Jr. "Operating Systems". In: *Datamation* 10.5 (May 1964), pp. 26–28 (cited on page 43).

[Tar85] Éva Tardos. "A strongly polynomial minimum cost circulation algorithm". In: *Combinatorica* 5.3 (1985), pp. 247–255 (cited on page 150).

[TCG+12] Alexey Tumanov, James Cipar, Gregory R. Ganger, and Michael A. Kozuch. "Alsched: Algebraic Scheduling of Mixed Workloads in Heterogeneous Clouds". In: *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC)*. San Jose, California, Oct. 2012, 25:1–25:7 (cited on pages 56, 59, 142, 191, 268).

[TMV+11] Lingjia Tang, Jason Mars, Neil Vachharajani, Robert Hundt, and Mary Lou Soffa. "The impact of memory subsystem resource sharing on datacenter applications". In: *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA)*. San Jose, California, USA, June 2011, pp. 283–294 (cited on pages 35, 40).

[TMV86] Andrew S. Tanenbaum, Sape J Mullender, and Robbert Van Renesse. "Using sparse capabilities in a distributed operating system". In: *Proceedings of the 6th International Conference on Distributed Computing Systems (ICDCS)*. Cambridge, Massachusetts, USA, May 1986, pp. 558–563 (cited on page 76).

[TSA+10] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, et al. “Data Warehousing and Analytics Infrastructure at Facebook”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Indianapolis, Indiana, USA, June 2010, pp. 1013–1020 (cited on page 32).

[TSJ+09] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, et al. “Hive: A Warehousing Solution over a Map-reduce Framework”. In: *Proceedings of the VLDB Endowment* 2.2 (Aug. 2009), pp. 1626–1629 (cited on page 33).

[TTS+14] Ankit Toshniwal, Siddarth Taneja, Amit Shukla, Karthik Ramasamy, Jignesh M. Patel, Sanjeev Kulkarni, Jason Jackson, et al. “Storm @Twitter”. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. Snowbird, Utah, USA, 2014, pp. 147–156 (cited on page 34).

[Tur46] Alan M. Turing. “Proposal for Development in the Mathematics Division of an Automatic Computing Engine”. In: *A. M. Turing’s ACE report of 1946 and other papers*. Edited by Brian E. Carpenter and R. W. Doran. Volume 10. Charles Babbage Institute reprint series for the History of Computing. MIT Press, 1986 (cited on page 42).

[TZP+16] Alexey Tumanov, Timothy Zhu, Jun Woo Park, Michael A. Kozuch, Mor Harchol-Balter, and Gregory R. Ganger. “TetriSched: Global Rescheduling with Adaptive Plan-ahead in Dynamic Heterogeneous Clusters”. In: *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*. London, England, United Kingdom, 2016, 35:1–35:16 (cited on pages 56, 58–59, 175, 191, 268).

[VCG65] Victor A. Vyssotsky, Fernando J. Corbató, and Robert M. Graham. “Structure of the Multics Supervisor”. In: *Proceedings of the 4th AFIPS Fall Joint Computer Conference (FJCC)*. Las Vegas, Nevada, Nov. 1965, pp. 203–212 (cited on page 80).

[VKS15] Nikos Vasilakis, Ben Karel, and Jonathan M. Smith. “From Lone Dwarfs to Giant Superclusters: Rethinking Operating System Abstractions for the Cloud”. In: *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS)*. Kartause Ittingen, Switzerland, May 2015 (cited on page 24).

[VMD+13] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, et al. “Apache Hadoop YARN: Yet Another Resource Negotiator”. In: *Proceedings of the 4th Annual Symposium on Cloud Computing (SoCC)*. Santa Clara, California, Oct. 2013, 5:1–5:16 (cited on pages 56–57, 59–60, 191).

[VPA+14] Shivaram Venkataraman, Aurojit Panda, Ganesh Ananthanarayanan, Michael J. Franklin, and Ion Stoica. "The Power of Choice in Data-Aware Cluster Scheduling". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 301–316 (cited on pages 56, 147, 191).

[VPK+15] Abhishek Verma, Luis David Pedrosa, Madhukar Korupolu, David Oppenheimer, and John Wilkes. "Large scale cluster management at Google". In: *Proceedings of the 10th ACM European Conference on Computer Systems (EuroSys)*. Bordeaux, France, Apr. 2015 (cited on pages 22, 30–32, 43, 47, 50, 60–61, 137, 153).

[WB72] William A. Wulf and C. Gordon Bell. "C.mmp: A Multi-mini-processor". In: *Proceedings of the 1972 Fall Joint Computer Conference (AFIPS)*. Anaheim, California, USA, Dec. 1972, pp. 765–777 (cited on page 43).

[WC09] Mike Waychinson and Jonathan Corbet. *KS2009: How Google uses Linux*. Accessed 13/03/2014. 2009. URL: `http://lwn.net/Articles/357658/` (cited on page 23).

[WCC+74] William A. Wulf, Ellis Cohen, William Corwin, Anita Jones, Roy Levin, C. Pierson, and Fred Pollack. "HYDRA: The Kernel of a Multiprocessor Operating System". In: *Communications of the ACM* 17.6 (June 1974), pp. 337–345 (cited on pages 43–44).

[WGB+10] David Wentzlaff, Charles Gruenwald III, Nathan Beckmann, Kevin Modzelewski, Adam Belay, Lamia Youseff, Jason Miller, et al. "An Operating System for Multicore and Clouds: Mechanisms and Implementation". In: *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC)*. Indianapolis, Indiana, USA, June 2010, pp. 3–14 (cited on page 23).

[Wil11] John Wilkes. *More Google cluster data*. Google reseach blog; accessed 26/07/-2014. Nov. 2011. URL: `http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html` (cited on page 35).

[Wil14] John Wilkes. "Cluster Management at Google [Video]". YouTube video of talk at MesosCon 2014. `https://www.youtu.be/watch?v=VQAAkO5B5Hg`; accessed on 21/11/2014. Aug. 2014 (cited on page 146).

[Wil85] Maurice Vincent Wilkes. *Memoirs of a Computer Pioneer*. Series in the History of Computing. MIT Press, 1985 (cited on page 42).

[WN79] Maurice V. Wilkes and Roger M. Needham. *The Cambridge CAP computer and Its Operating System*. Operating and Programming Systems. Amsterdam, The Netherlands: North-Holland Publishing Co., 1979 (cited on page 79).

[WPE+83] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. "The LOCUS Distributed Operating System". In: *Proceedings of the 9th ACM Symposium on Operating Systems Principles (SOSP)*. Bretton Woods, New Hampshire, USA, Oct. 1983, pp. 49–70 (cited on pages 44–45, 82).

[WQY+09] Feng Wang, Jie Qiu, Jie Yang, Bo Dong, Xinhui Li, and Ying Li. "Hadoop High Availability Through Metadata Replication". In: *Proceedings of the 1st International Workshop on Cloud Data Management (CloudDB)*. Hong Kong, China, 2009, pp. 37–44 (cited on page 50).

[WSK+09] William Whitted, Montgomery Sykora, Ken Krieger, Benchiao Jai, William Hamburgen, Jimmy Clidaras, Donald L. Beaty, et al. *Data center uninterruptible power distribution architecture*. US Patent 7,560,831. July 2009 (cited on page 23).

[WWC+14] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, et al. "The CHERI capability model: Revisiting RISC in an age of risk". In: *Proceedings of the 41st International Symposium on Computer Architecture (ISCA)*. Minneapolis, Minnesota, USA, June 2014 (cited on pages 79, 201).

[WWN+15] Robert N. M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, et al. "CHERI: A hybrid capability-system architecture for scalable software compartmentalization". In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. San Jose, California, USA, May 2015 (cited on page 201).

[XL08] Yuejian Xie and Gabriel H. Loh. "Dynamic classification of program memory behaviors in CMPs". In: *Proceedings of 2nd Workshop on Chip Multiprocessors Memory Systems and Interconnects (CMP-MSI)*. Beijing, China, June 2008 (cited on page 169).

[YIF+08] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Úlfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey. "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language". In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, California, USA, Dec. 2008 (cited on page 33).

[ZBK+06] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. "Making Information Flow Explicit in HiStar". In: *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Seattle, Washington, USA, Nov. 2006, pp. 263–278 (cited on pages 69, 202).

[ZBM08] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Maziéres. "Securing Distributed Systems with Information Flow Control". In: *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. San Francisco, California, USA, Apr. 2008, pp. 293–308 (cited on pages 69, 202).

[ZBS+10] Matei Zaharia, Dhruba Borthakur, Joydeep Sen Sarma, Khaled Elmeleegy, Scott Shenker, and Ion Stoica. "Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling". In: *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*. Paris, France, Apr. 2010, pp. 265–278 (cited on pages 56, 60, 191, 239).

[ZCD+12] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, et al. "Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. San Jose, California, USA, Apr. 2012, pp. 15–28 (cited on pages 21, 47, 49, 58, 70, 75, 125).

[ZHB11] Qi Zhang, Joseph L. Hellerstein, and Raouf Boutaba. "Characterizing task usage shapes in Google's compute clusters". In: *Proceedings of the 5th Workshop on Large Scale Distributed Systems and Middleware (LADIS)*. Seattle, Washington, USA, Sept. 2011 (cited on page 59).

[ZHK+11] Matei Zaharia, Benjamin Hindman, Andy Konwinski, Ali Ghodsi, Anthony D. Joesph, Randy Katz, Scott Shenker, et al. "The Datacenter Needs an Operating System". In: *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing (HotCloud)*. Portland, Oregon, USA, June 2011, pp. 17–21 (cited on pages 22, 52).

[ZKJ+08] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy H. Katz, and Ion Stoica. "Improving MapReduce Performance in Heterogeneous Environments". In: *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. San Diego, California, USA, Dec. 2008, pp. 29–42 (cited on pages 56, 191).

[ZSA+14] Irene Zhang, Adriana Szekeres, Dana Van Aken, Isaac Ackerman, Steven D. Gribble, Arvind Krishnamurthy, and Henry M. Levy. "Customizable and Extensible Deployment for Mobile/Cloud Applications". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. Broomfield, Colorado, USA, Oct. 2014, pp. 97–112 (cited on pages 70, 200).

[ZTH+13] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. "CPI^2: CPU Performance Isolation for Shared Compute Clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys)*. Prague, Czech Republic, Apr. 2013, pp. 379–391 (cited on pages 31, 34, 40–41, 135, 157, 161, 179–180).

Appendix A

Additional background material

A.1 Additional workload interference experiments

A.1.1 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.[1] I pin benchmarks to cores (using `cgroups`, as used in Linux containers), and pin their memory on the local NUMA node.[2]

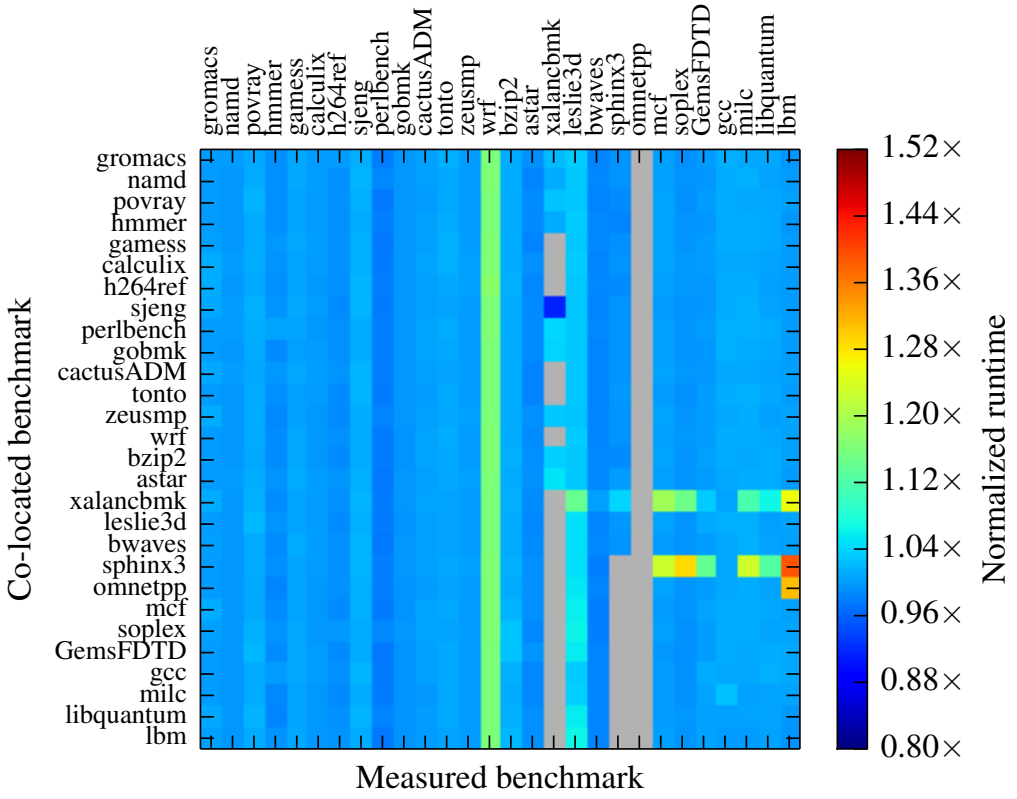
I normalise the runtime of a measured workload under co-location to its ideal runtime on an otherwise idle machine: in other words, the result is 1.0 in the absence of any interference. Figure A.1 visualises the results as a heat map: warmer colours indicate a higher degree of interference. The colour 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.[3] 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×. This is unsurprising: a workload with a large working sets ends up frequently evicting cache lines of a co-located workload, even if its working set fits into the cache.

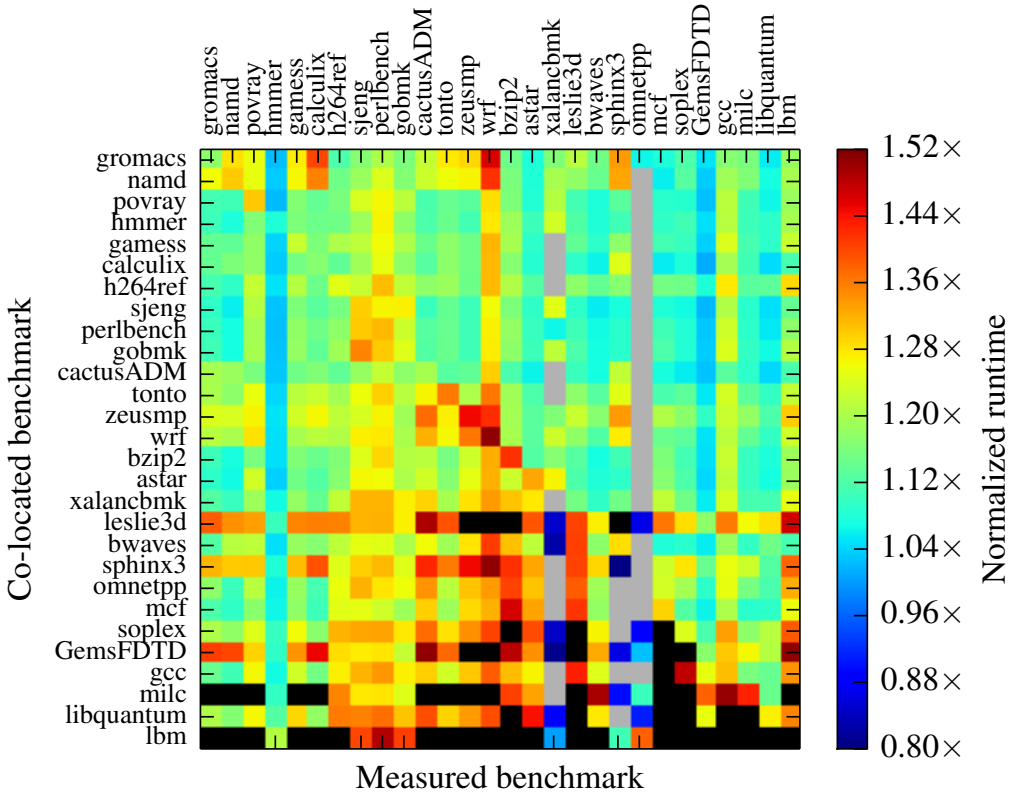
[1] This includes "dynamic overclocking" techniques (AMD "TurboCORE" and Intel "TurboBoost"), since the non-deterministic clock speed that depends on the thermal state of the processor masks the effects of cache sharing.

[2] Benchmark runtime increases by ≈33–50% when working on remote memory.

[3] 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.



(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.1: Co-location heatmap on an AMD Opteron 4234: normalised runtime of *x*-axis workload in the presence of *y*-axis workload. Black squares indicate that the normalised runtime exceeded the scale; grey ones correspond to values below 0.8×.

A.1.2 Pairwise application co-location: additional metrics

A.1.3 *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.[4] 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× 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.

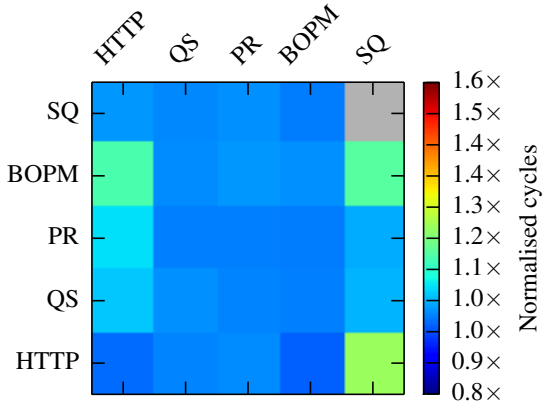
   (b) The shared, two-way set-associative, 64 KB *L1 instruction cache* of adjacent cores, which may cause additional instruction cache misses. By contrast, the Xeon has a dedicated, 4-way set-associative 32 KB L1 instruction cache per core.

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].

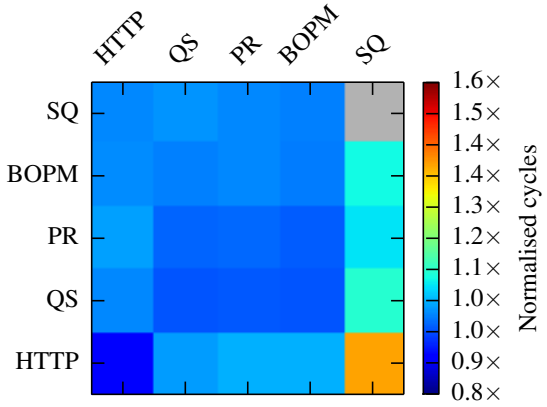
3. Worst-case degradation and variability across experiments are higher on the Opteron than on the Xeon machine, possibly due to different cache sizes.

These experiments demonstrate that higher machine utilisation leads to increased co-location interference, especially for memory-intensive workloads.

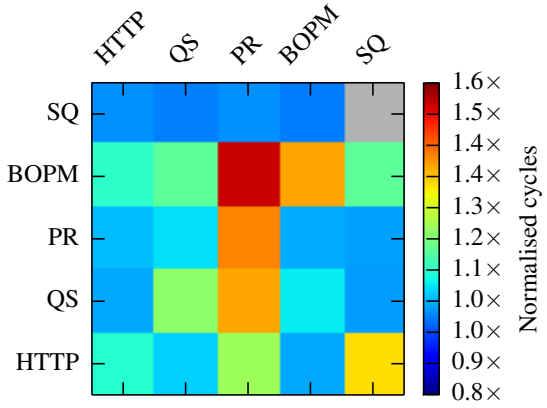
[4] Running multiple instances of the *same* workload on a machine is not as naïve as it may seem: many cluster schedulers optimise for data locality, i.e. they attempt to place computations near their input data [IPC+09; ZBS+10]. If the input data to a large job is replicated to a handful of machines 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].



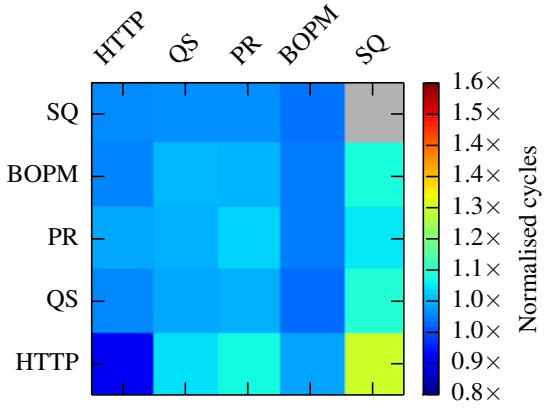
(a) Opteron: separate sockets (cores 4 and 5).



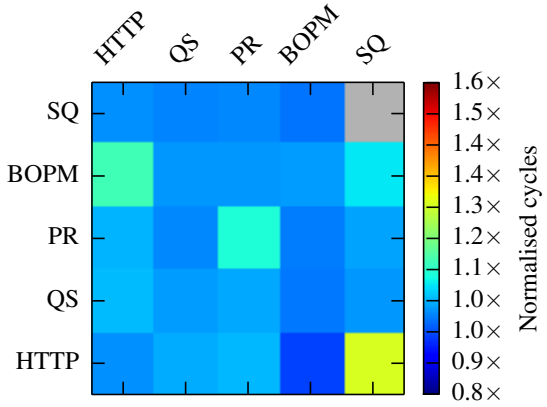
(b) Xeon: separate sockets (cores 4 and 5).



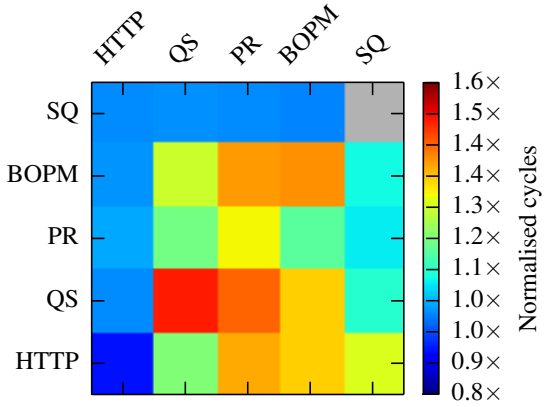
(c) Opteron: shared L3 cache (cores 4 and 8).



(d) Xeon: shared L3 cache (cores 4 and 6).

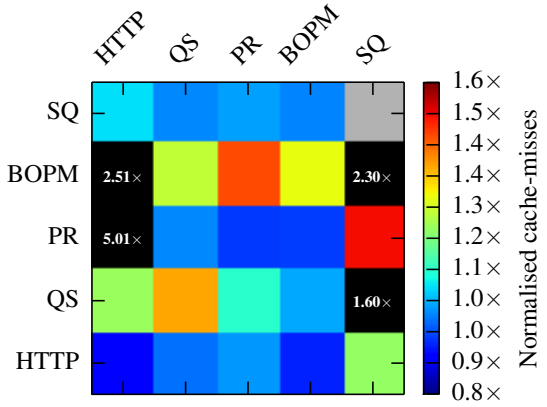


(e) Opteron: sharing L2 and L3 (cores 4 and 6).

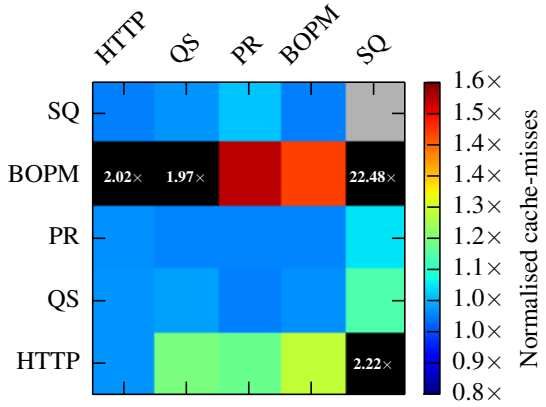


(f) Xeon: adjacent hyperthreads (cores 4 and 16).

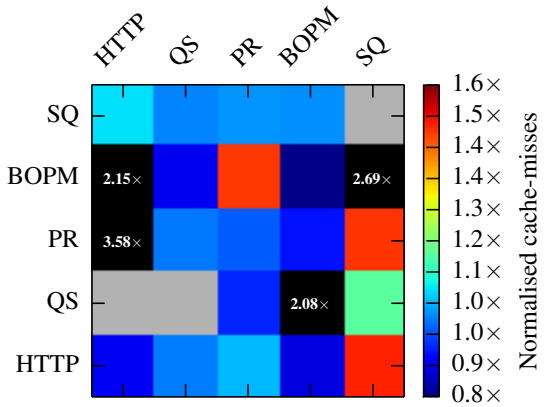
Figure A.2: Normalised cycle counts for co-located WSC applications on the AMD Opteron 4234 *(left column)* and Intel Xeon E5-2420 *(right column)*. All results are for the *x*-axis benchmark, normalised to its mean isolated cycle count on an otherwise idle machine. Black squares indicate normalised cycle counts exceeding the scale; grey indicates that no results are available.



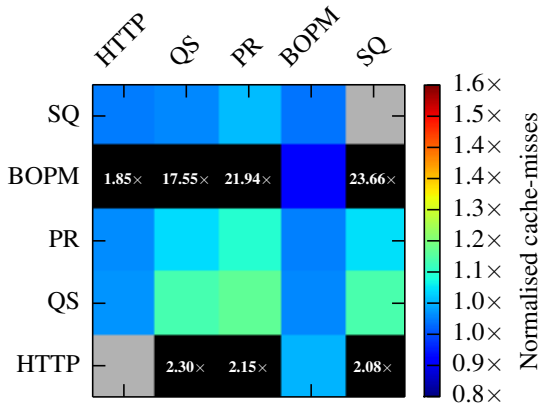
(a) Opteron: separate sockets (cores 4 and 5).



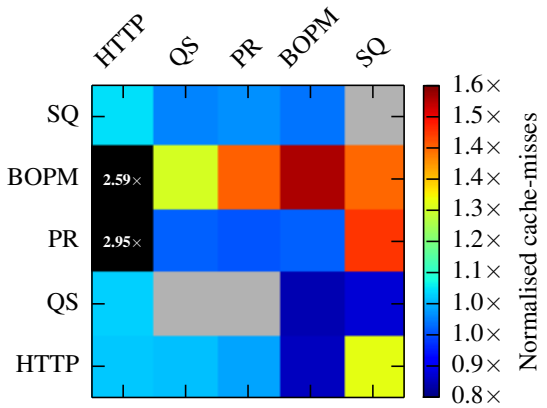
(b) Xeon: separate sockets (cores 4 and 5).



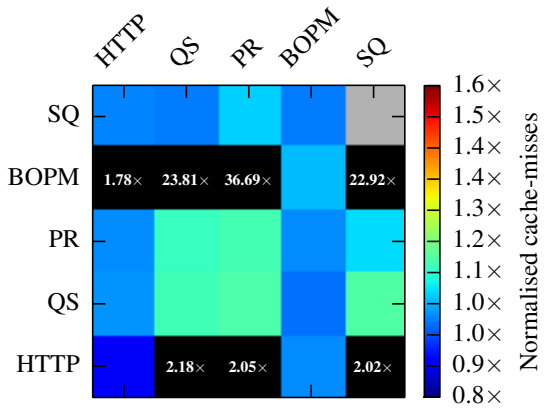
(c) Opteron: shared L3 cache (cores 4 and 8).



(d) Xeon: shared L3 cache (cores 4 and 6).



(e) Opteron: sharing L2 and L3 (cores 4 and 6).



(f) Xeon: adjacent hyperthreads (cores 4 and 16).

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 normalised cache miss counts exceeding the scale; grey indicates that no results are available.

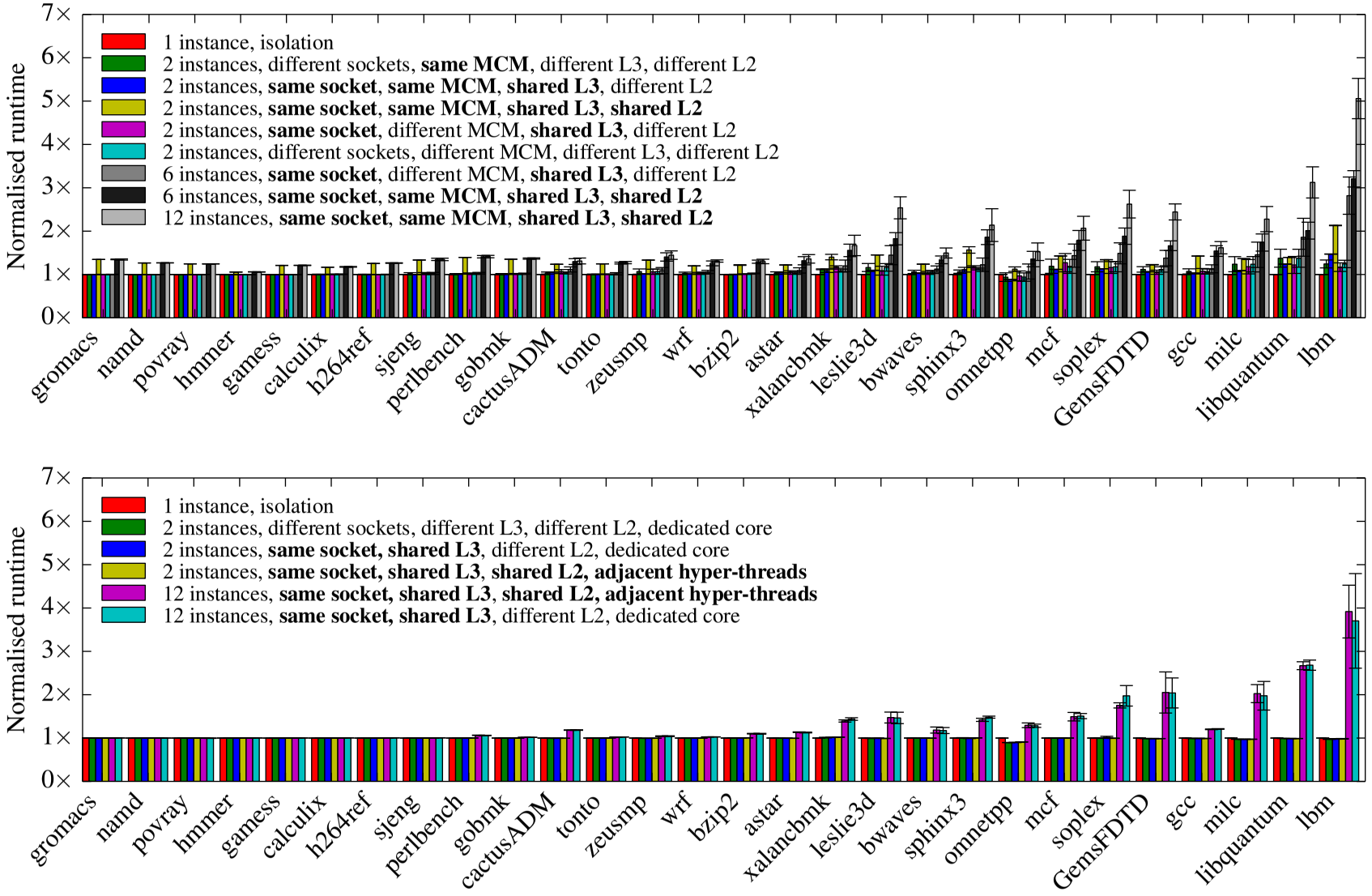


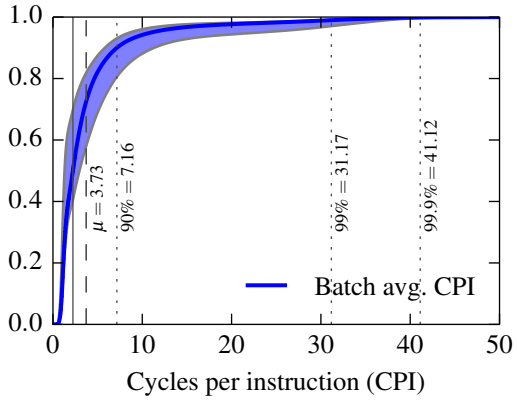
Figure A.4: Interference between multiple instances of SPEC CPU2006 benchmarks on AMD Opteron 4234 *(top)* and Intel Xeon E5-2420 *(bottom)*. Values are averages and standard deviations over the normalised runtimes of ten executions of each experiment.

A.2 CPI and IPMA distributions in a Google cluster

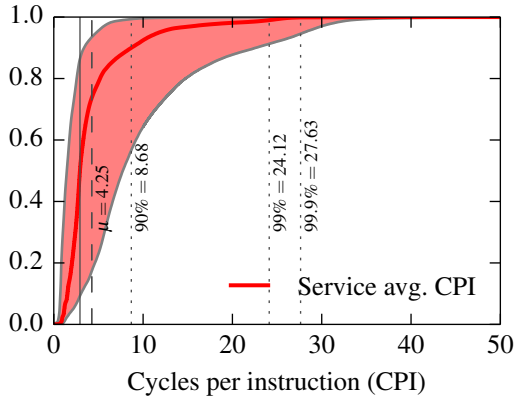
Figure A.5 shows that the synthetic workloads that I used in Section 8.2.1 (Table 8.2) are situated towards the lower end of the CPI distribution of Google workloads. The 99th percentile CPI observed in my experiments, at around three cycles per instruction, is similar to the *average* CPI at Google. 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.[5] 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.

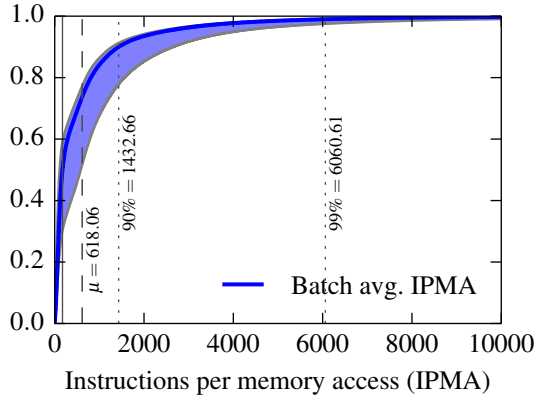
[5] With the information available in the Google trace, it is difficult to say if Google workloads usually have poor cache-affinity (which decreases IPMA), or whether the workloads are inherently memory-intensive due to large working sets (same effect), or if the Borg scheduler fails to avoid interference in the cache hierarchy (again, having the same effect).



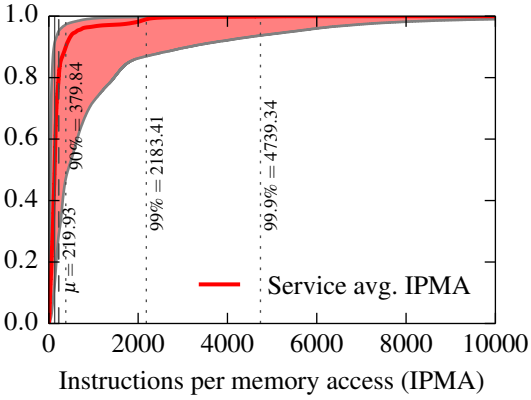
(a) Batch workloads, cycles per instruction (CPI). Valid $n = 39,193,353$.



(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 $n = 72,678$.

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.

Appendix B

Additional DIOS material

B.1 DIOS system call API

B.1.1 `dios_create(2)`

The `dios_create(2)` system call is used to generate an entirely new logical DIOS object (and, consequently, a name and an initial physical object). It is the only system call that can generate logical objects and names; every name must once have been returned by a `dios_create(2)` invocation.

`dios_create(2)` typically takes two arguments: a set of references to physical group objects (S_G) for groups in which the new logical 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 logical object is named or anonymous (§4.3.1).

The return value is a tuple consisting of a fresh name (unless an anonymous logical object is created) and a reference to the new, initial physical object. If the logical 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.

`dios_create`$(S_G, \mathcal{P}_{args}, \texttt{host}, \mathcal{F}) \rightarrow \langle \mathcal{N}_o, \mathcal{R}_o \rangle$

The `dios_create(2)` system call, called from task **T**, takes an optional set of references to physical group objects, S_G, indicating the groups in which the new logical object is to be created. `dios_create(2)` returns a tuple consisting of the new logical object o's external name, $\mathcal{N}_o$, and a reference to the new physical object, $\mathcal{R}_o$.

The kernel also computes an internal name, $\mathcal{N}_o^i$, for the new logical object and maps it to the object structure via the local name table. Finally, the reference $\mathcal{R}_o$ is inserted into the task's reference table.

The reference returned grants full permissions to the task creating the logical object, and the reference type and other parameters are set according to the initialisation flags in $\mathcal{F}$.

Note that `dios_create(2)` does *not* necessarily allocate any I/O buffers or object state. The tuple returned only contains the identifier for the logical object and a handle for the physical object; buffers to interact with it and other state are typically set up lazily on the first I/O request. In other words, `dios_acquire_read(2)` and `dios_acquire_write(2)` must be employed after `dios_create(2)` to effect any I/O on the physical object.

An interesting bootstrapping challenge arises from the fact that names are stored in user-provided memory (unlike references). The names of the initial logical 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 blob on startup that ends up storing further logical objects' names.

B.1.2 `dios_lookup(2)`

The ability to locate physical 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 `dios_lookup(2)` system call.

`dios_lookup(2)` takes a name as its argument and returns a set of references to reachable physical objects corresponding to this name. *Reachable* in this context means:

(a) that the corresponding logical object is named (i.e. not anonymous),
(b) that the logical object is in a group that the calling task is a member of,
(c) that the physical object in question is either local, or
(d) if the physical object 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 physical objects found then have references generated for them. The reference generation applies similar transformation to a reference delegation (see §4.5.3) 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, `dios_lookup(2)` is defined as follows:

$$\textbf{dios\_lookup}(\mathcal{N}_o, \mathcal{F}) \;\rightarrow\; \{\mathcal{R}_o \mid o \in \mathit{reachable}(\mathcal{N}_o, G_{\mathbf{T}})\}$$

$$\textbf{dios\_lookup}(\mathcal{N}_o, \mathcal{R}_g, \mathcal{F}) \;\rightarrow\; \{\mathcal{R}_o \mid o \in \mathit{reachable}(\mathcal{N}_o, g)\}$$

The `dios_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 physical objects for logical object o named by $\mathcal{N}_o$ in all of **T**'s groups $g \in G_{\mathbf{T}}$; or *(ii)* to all reachable physical objects for logical object o in group g, if an explicit $\mathcal{R}_g$ is specified.

To do so, the kernel computes either *(i)* the internal names for $\mathcal{N}_o$ in each group g that $\mathbf{T}$ is a member of, or *(ii)* the internal name for $\mathcal{N}_o$ in g, respectively, by computing $\mathcal{N}_o^g = \mathcal{H}(\mathcal{N}_o \parallel g)$, and looks up $\mathcal{N}_o^g$ in the name table.

As implied in above definitions, `dios_lookup(2)` makes no guarantee to return *all* physical objects for the logical object corresponding to the name specified in the argument, but only reachable ones. In other words, name resolution for remote physical objects is best-effort, although a reliable interconnect can help increase the confidence of comprehensive results being returned.

Additionally, `dios_lookup(2)` merely guarantees to return references for all physical which were reachable at the point of handling the system call. This may include stale information: the physical objects found may already have been deleted or new ones created when the system call returns (see §4.8).

B.1.3 `dios_copy(2)`

As explained in Section 4.5.3, references can be *delegated* to other tasks. This is facilitated by the `dios_copy(2)` system call.

`dios_copy(2)` takes three arguments: a reference to delegate, a reference to the target task that it is to be delegated to, and a specification of the desired transformations. The second reference must refer to a task-type physical object; if it does not, the `dios_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.

`dios_copy(`$\mathcal{R}_o, \mathcal{R}_{\mathbf{T}_d}, \mathcal{P}_{transform}, \mathcal{F}$**`)`** $\rightarrow$ `bool`

The `dios_copy(2)` system call, when invoked by task $\mathbf{T}$, takes a reference, $\mathcal{R}_o$, and a specification of desired transformations (pointed to by $\mathcal{P}_{transform}$). It delegates a transformed copy of $\mathcal{R}_o$ to the task $\mathbf{T}_d$ referred to by $\mathcal{R}_{\mathbf{T}_d}$.

The newly created reference is inserted into $\mathbf{T}_d$'s reference table, and a message notifying $\mathbf{T}_d$ of the delegation is enqueued to be read from its self-reference.

`dios_copy(2)` is a complex and powerful system call. It checks the validity of the reference delegation requested, transforms the reference into a new reference as appropriate for the target context, and communicates the delegation to the target task and its managing kernel. Transformations are described in Section 4.5.3, and may involve copying the physical 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 `dios_copy(2)` system call:

1. It can be used to dynamically add logical objects to groups by permitting the target reference to refer to a physical group object. Instead of inserting a new reference into a task's reference table, this invocation on a reference to physical object o, $\mathcal{R}_o$, with external name $\mathcal{N}_o$, would insert an internal name $\mathcal{N}_o^g$ for the target group g in the name tables. In order to maintain security, however, only logical 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 `dios_delete(2)`

While `dios_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 *physical objects* are separate concerns.

The DIOS `dios_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 physical object referred to depends on its deletion mode (§4.2):

- A *reference-counted* physical 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 physical object is *one-off deleted*, it is removed atomically, and any other references to it become invalid and system calls on them fail.
- For an *immortal* physical object, nothing happens (other than the reference being dropped).

The corresponding logical object's name is deleted from the name table once the last physical object for it is removed.

Under ordinary circumstances (i.e. in the absence of failures), `dios_delete(2)` always returns true if a valid reference $\mathcal{R}_o$ is passed.

Hence, the `dios_delete(2)` system call is defined as follows:

dios_delete($\mathcal{R}_o$, $\mathcal{F}$) $\rightarrow$ bool

The dios_delete(2) system call takes a reference, $\mathcal{R}_o$, to a physical object o, and deletes it from the calling task **T**'s reference table.

A deletion handler specific to physical object o's deletion mode 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 dios_delete(2) call returns a success indication.

Since dios_delete(2) directly mutates the state of an existing physical object and affects the validity of names and references for it, race conditions with other system calls are subtle. Consider, for example, dios_delete(2) racing with an invocation of dios_copy(2): if the deletion is handled first, the delegation effected by dios_copy(2) will fail as the reference no longer exists. However, if the delegation 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 dios_delete(2) is made (although this is guaranteed after dios_delete(2) *returns*). A consequence of these semantics is that running dios_lookup($\mathcal{N}_o$) to obtain the set of references corresponding to $\mathcal{N}_o$'s physical objects and then invoking dios_delete($\mathcal{R}_o^i$) on each reference $\mathcal{R}_o^i$ returned does *not* guarantee that all physical objects described by these references will be deleted.

As the deletion always affects the task-local reference table only, no other dios_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 dios_delete(2). This is a special case: it has the effect of terminating the task (and thus *does* affect the logical task 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 dios_delete(2) on all of the references in its reference table. Since the table is destroyed immediately after, this is only consequential as other machines may need to be notified of the reduced reference count for the physical objects.

B.1.5 dios_run(2)

A task can invoke the dios_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, `dios_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 `dios_run(2)` system call also specifies the group memberships to be inherited by the newly created child.

`dios_run(2)` takes a set of references to physical group objects representing groups of which the child task is to be granted membership. Any physical 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, and the default behaviour is to give membership of a single, newly generated group to the spawned task.

The `dios_run(2)` system call returns a reference to the newly spawned physical task object, and is defined as follows:

`dios_run`$(\mathcal{R}_{bin}, S_G, \mathcal{P}_{info}, \mathcal{F}) \rightarrow \mathcal{R}_{\mathbf{T}_C}$

The `dios_run(2)` system call, when invoked by parent task $\mathbf{T}_P$, takes a reference to an executable physical object (a binary), $\mathcal{R}_{bin}$, and a set of references to physical group objects, S_G, as arguments. It causes a child task, $\mathbf{T}_C$, to be created.

A reference to the new physical task object for $\mathbf{T}_C$, $\mathcal{R}_{\mathbf{T}_C}$ is returned.

The child task's set of group memberships, G_C, is set to S_G, which is defined as:

$$S_G = \{\mathcal{R}_g \mid g \in G_P \text{ or } g \in C_P\},$$

where G_P stands for the parent task's group memberships and C_P for the groups created by the parent task.

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, `dios_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 kernel creates a new local task and sets it runnable.

2. If the `SPAWN_LOCAL` flag is *not* present, and a specific host ID is set in the $\mathcal{P}_{info}$ 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 manager, which will place the task and execute it by invoking `dios_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 `dios_pause(2)` and `dios_resume(2)`

When a DIOS application needs to suspend itself or another task, it can use the `dios_pause(2)` system call. Likewise, `dios_resume(2)` can be used to continue executing a suspended task.

`dios_pause(2).` The argument to `dios_pause(2)` must be a reference to a physical 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.

> **`dios_pause(`**$\mathcal{R}_\mathbf{T}, \mathcal{F}$**`)`** $\rightarrow$ `bool`
>
> The `dios_pause(2)` system call pauses the execution of a task **T**. It takes a reference to **T**'s physical task object, $\mathcal{R}_\mathbf{T}$, as its argument. The kernel notifies the 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 `dios_pause(2)` on a task's self-reference ($\mathcal{R}_{self}$) is a special case. It has the effect of *yielding* the processor with immediate effect. If the `STAY_RUNNABLE` flag is set, the task stays runnable. Otherwise, it is suspended until `dios_resume(2)` is invoked on it by another task.

`dios_resume(2).` A suspended task **T** can be continued by any other task invoking the `dios_resume(2)` system call with $\mathcal{R}_\mathbf{T}$ as its argument. `dios_resume(2)` is the counterpart

to `dios_pause(2)` and its operation is analogous. The only difference is that a task cannot invoke `dios_resume(2)` on a reference to itself.

In practice, `dios_pause(2)` and `dios_resume(2)` calls are primarily used by the cluster scheduler. However, in combination with an extended `dios_copy(2)` system call, they could also support task migration: a task would be paused, copied, the original deleted and the new copy resumed.[1]

B.1.7 `dios_acquire_read(2)`

The `dios_acquire_read(2)` system call initiates a read-only I/O request. It takes a reference to a physical 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, `dios_acquire_read(2)` attempts to read the specified number of bytes (or as many bytes of data as possible, if `size` is zero) from the physical object *o* referred to by $\mathcal{R}_o$.

`dios_acquire_read(2)` is a *buffer-acquiring* system call: when it returns, the kernel supplies the user application with a read buffer containing physical object data. Several buffer management options are available by passing appropriate flags:

1. `ACQUIRE_IOV_CREATE` *(default)*: creates a buffer for the reference. When reading, this buffer holds data; when writing, it may hold data for blobs, but not for streams.
2. `ACQUIRE_IOV_TAKE`: accepts an existing buffer as an argument, acquires ownership of it and uses it; the buffer is deleted on release.
3. `ACQUIRE_IOV_BORROW`: accepts an existing buffer and holds it until it is released, but does not acquire ownership. When `dios_release_read(2)`/`dios_release_write(2)` is called, the buffer is returned to the original owner, but not destroyed.
4. `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. `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.

[1] The current DIOS prototype does not yet support copying of physical 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.

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 physical 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 `dios_acquire_read(2)` system call can be specified as follows:

> **`dios_acquire_read(`**$\mathcal{R}_o$, `size`, `sem`, $\mathcal{F}$**`)`** $\rightarrow \langle \mathcal{P}, \texttt{size} \rangle$
>
> The `dios_acquire_read(2)` system call attempts to initiate a read-only I/O request on physical object o referred to by $\mathcal{R}_o$. 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. The `sem` parameter specifies the desired concurrent access semantics.
>
> If the desired concurrent access semantic can be satisfied, the I/O request is registered by incrementing the physical object's active read request counter (§4.7).
>
> 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.
>
> `dios_acquire_read(2)` returns a tuple $\langle \mathcal{P}, \texttt{size} \rangle$ that contains a pointer to the buffer ($\mathcal{P}$) and an indication of its length (`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 physical 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 `dios_acquire_read(2)`, however, are not definitely valid until `dios_commit_read(2)` has been called to complete the I/O request. Only if `dios_commit_read(2)` indicates that the read was valid under the concurrent access semantics specified in `sem`, the data can be treated as valid. 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 `dios_commit_read(2)` before using it.

B.1.8 `dios_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 `dios_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, `dios_commit_read(2)` expects to be given a buffer already associated with the reference $\mathcal{R}_o$ 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 `dios_commit_read(2)` system call is thus defined as follows:

`dios_commit_read(`$\mathcal{R}_o$, $\langle\mathcal{P}$, `size`$\rangle$, $\mathcal{F}$**`)`** $\rightarrow$ `bool`

The `dios_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 physical object's read version counter is atomically incremented. If the commit fails, the read version counter is not incremented.

In either case, the physical object's active reader count is atomically decremented.

The precise semantics of a failed commit depend on the underlying physical 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 `dios_release_read(2)`

Since buffers are a limited resource, they must eventually be returned to the kernel for reuse. The `dios_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.

`dios_release_read(2)` normally returns `true`; only invalid arguments can lead to an error. The system call is thus defined as follows:

`dios_release_read(`$\mathcal{R}_o$, $\langle\mathcal{P}$, `size`$\rangle$, $\mathcal{F}$**`)`** $\rightarrow$ `bool`

The `dios_release_read(2)` system call returns an active buffer to the kernel.

The buffer at $\mathcal{P}$ is invalidated and unmapped from user-space virtual memory.

`dios_release_read(2)` can typically be handled entirely locally, even if the physical object is remote, since the buffer contains local state only. After `dios_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 `dios_acquire_write(2)`

Like `dios_acquire_read(2)`, `dios_acquire_write(2)` 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, `dios_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 `dios_acquire_write(2)` system call is similar to the definition of `dios_acquire_read(2)`:

`dios_acquire_write(`$\mathcal{R}_o$, `size`, `sem`, $\mathcal{F}$**`)`** $\rightarrow \langle\mathcal{P}$, `size`$\rangle$

The `dios_acquire_write(2)` system call attempts to initiate a write I/O request on physical 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 the `sem` parameter) can be satisfied, the I/O request is registered by incrementing the physical object's active write request counter (§4.7).

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.

`dios_acquire_write(2)` returns a tuple $\langle\mathcal{P}$, `size`$\rangle$ that contains a pointer to the buffer ($\mathcal{P}$) and an indication of its length (`size`).

After `dios_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 physical object's 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 `dios_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, `dios_commit_write(2)` expects to be given a buffer already associated with the reference $\mathcal{R}_o$ passed, but as with `dios_commit_read(2)`, this can be customised. In addition to the `COMMIT_IOV_USE` and `COMMIT_IOV_MOVE` flags, `dios_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 physical objects (e.g. network streams) without having any intention of re-using it with any of them.

As with `dios_acquire_read(2)`, the specification of `dios_commit_write(2)` is similar to its read equivalent, `dios_commit_read(2)`:

`dios_commit_write(`$\mathcal{R}_o, \langle\mathcal{P}$, `size`$\rangle$, $\mathcal{F}$**`)`** $\rightarrow$ `bool`

The `dios_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 $[\mathcal{P},\ \mathcal{P} +$ `size`$]$ were definitely written to the physical object under the desired concurrent access semantics, and the physical 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 physical object's type.

In either case, the physical object's active writer count is atomically decremented.

As with reads, the precise semantics of a failed write commit depend on the underlying physical object's 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 affect

the physical object even though the commit failed.[2] As with reading, it is the application's responsibility to tolerate this failure or perform synchronisation such that this situation cannot occur.

B.1.12 `dios_release_write(2)`

Once a user-space program has completed and committed the output written into a write buffer (supplied by `dios_acquire_write(2)`), the buffer must eventually be returned to the kernel when it is no longer needed.

As with `dios_release_read(2)`, this operation is defined as follows:

> **`dios_release_write(`**$\mathcal{R}_o, \langle \mathcal{P}$, `size`$\rangle$, $\mathcal{F}$**`)`** $\rightarrow$ `bool`
>
> The `dios_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 `dios_release_write(2)` system call – unlike the acquire and commit calls – can be handled entirely locally; only an implicit commit (`RELEASE_IOV_COMMIT`) may require a remote operation.

B.1.13 `dios_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 `dios_select(2)` loops in conventional OSes.[3]

The DIOS `dios_select(2)` system call implements synchronous parallel waiting on multiple references, returning the reference that becomes available first. It is defined as follows:

> **`dios_select(`**$\{\mathcal{R}_0, \ldots, \mathcal{R}_k\}$, `mode`, $\mathcal{F}$**`)`** $\rightarrow \mathcal{R}_i$
>
> The `dios_select(2)` system call returns the first reference that has data available for I/O of `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 `dios_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.

[2] 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.

[3] Alternative systems with slightly different semantics – e.g. Linux's `epoll` notifications – are also used.

When a reference passed refers to a remote physical object, a DCP message is sent to the remote kernel to inform it of the reference being in a selector. When the physical object becomes ready for I/O, the remote kernel sends a notification to all kernels that have references to the physical object in selectors.[4]

Finally, it is worth noting that `dios_select(2)` can easily be extended to support timed waiting by passing a reference to a physical timer object which becomes ready after a certain time has elapsed.

[4] If references to the physical 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.

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 ($\equiv$ 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. $f_e = f(v,w) \geq c_{\min}(v,w)$ and $f(u,w) \leq c_{\max}(v,w)$, where $c_{\min}$ and $c_{\max}$ denote 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); 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.

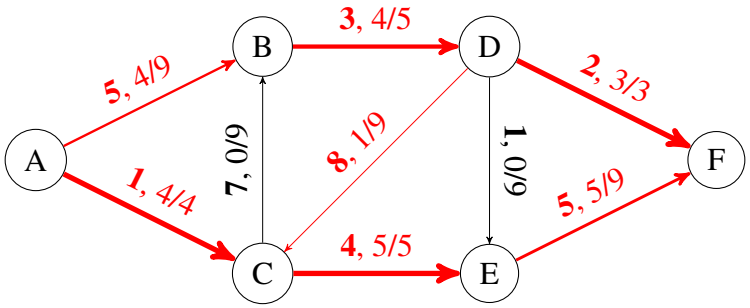


Figure C.1: Minimum-cost, maximum-flow optimisation example: vertex A is the source, vertex F is the sink; arcs are labelled as "**cost**, flow/capacity". Arc width is proportional to the flow carried, and the minimum-cost, maximum-flow solution at cost 95 is highlighted in red.

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*:

$$\sum_{(v,w)\in E} f(v,w) = \sum_{(v,w)\in E} f(w,v) \text{ for each vertex } v \in V \text{ that is not a source or sink,}$$

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:

$$\sum_{w\in V} f(s,w) = d \text{ and } \sum_{w\in V} f(w,t) = d, \text{where } s \text{ is the source vertex and } t \text{ is the sink vertex.}^{2}$$

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: $residual(v,w) = c_{\max}(v,w) - f(v,w)$. 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 $a_p(v,w) = a(v,w) + price(v) - price(w)$, a minimum-cost circulation exists if (and only if) there exists a function *price* such that $\forall (v,w).a_p(v,w) < -\varepsilon \Rightarrow f(v,w) = c_{\max}(v,w)$. In other words, *price* assigns prices to vertices in such a way that a price less than $-\varepsilon$ 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 $-\varepsilon$. 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 ($\equiv$ flow) at minimal cost.

1. **Initialisation**:
   (a) set $\varepsilon = \alpha$, such that α 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 ε-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 ε,
         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 ε by α, and
      ii. repeat from 2a.
3. **Termination**: return the ε-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 ($\equiv$ neighbours) are willing to buy at the asking price (step 2(c)i in Algorithm C.1).

When no business partner 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 ε-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 $O(VE \log_{E/V \log V}(V))$ 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 $[cap_{\min}, cap_{\max}]$ bounded by the minimum and maximum capacity. In Firmament, as in Quincy, $cap_{\min}$ is generally zero, while the value of $cap_{\max}$ 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 ($\mathbf{T}_i$) 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 ($\mathbf{U}_j$), or a rack or machine vertex ($\mathbf{R}_k$, $\mathbf{M}_l$).

Similarly, at the far end of the network, each arc from a machine vertex ($\mathbf{M}_l$) 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 ($\mathbf{R}_k$) 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 $K > 1$.

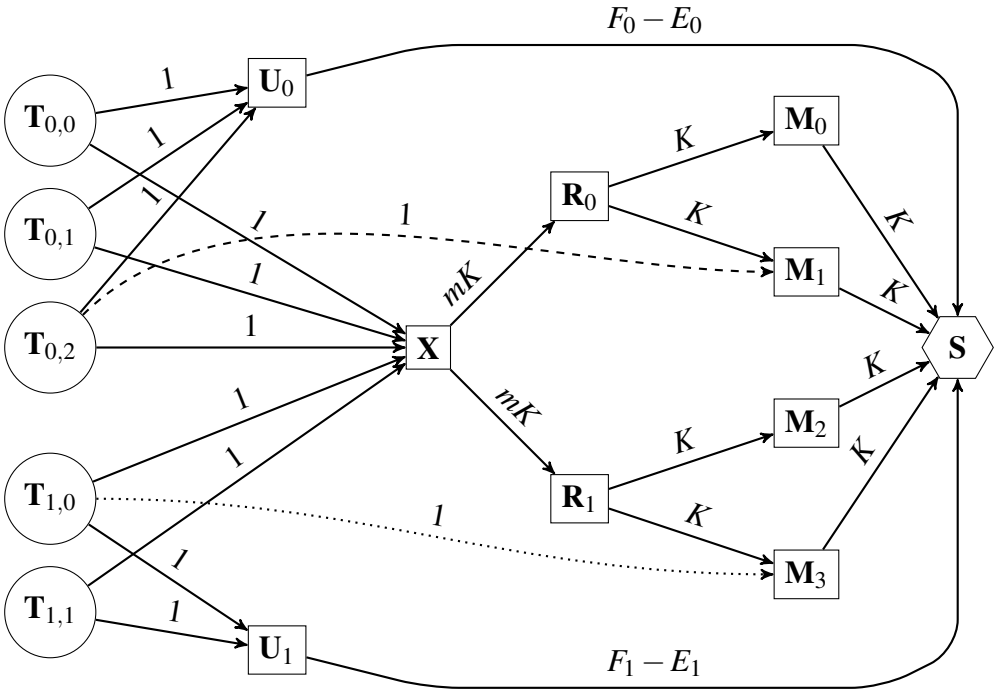


Figure C.2: The flow network in Figure 6.1 with example capacities on the arcs, repeated from Figure 6.2.

Edge	Capacity	Notes
$\mathbf{T}_i \to \mathbf{M}_l$	1	Specific machine preference.
$\mathbf{T}_i \to \mathbf{R}_k$	1	Rack preference.
$\mathbf{T}_i \to \mathbf{X}$	1	Wildcard, may schedule anywhere.
$\mathbf{T}_i \to \mathbf{U}_j$	1	Possibility of remaining unscheduled.
$\mathbf{R}_k \to \mathbf{M}_l$	K	For K tasks on the machine.
$\mathbf{X} \to \mathbf{R}_k$	mK	For m machines in the rack.
$\mathbf{M}_l \to \mathbf{S}$	K	For K tasks on the machine.
$\mathbf{U}_j \to \mathbf{S}$	$F_j - E_j$	For $1 \leq E_j \leq F_j \leq N_j$; N_j is the number of tasks in job j.

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:

- $N_j - F_j$ ($\equiv |F_j - N_j|$) 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 $\mathbf{U}_j$, 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 $\mathbf{U}_j$, 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 $\mathbf{U}_j$'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 6.3.2.

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 7.

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 (ψ) and the cluster aggregation switch (ξ) with the amount of remote data required. $\mathcal{R}(\mathbf{T}_{i,j})$ indicates the number of bytes that must be copied from machines within the same rack; $\mathcal{X}(\mathbf{T}_{i,j})$ is the number of bytes that must be pulled in from other racks for task $\mathbf{T}_{j,i}$.

Parameter	Edge	Meaning
v_i^j	$\mathbf{T}_{j,i} \to \mathbf{U}_j$	Cost of leaving $\mathbf{T}_{j,i}$ unscheduled.
α_i^j	$\mathbf{T}_{j,i} \to \mathbf{X}$	Cost of scheduling in the worst possible location.
$\rho_{i,l}^j$	$\mathbf{T}_{j,i} \to \mathbf{R}_l$	Cost of scheduling on worst machine in rack.
$\gamma_{i,m}^j$	$\mathbf{T}_{j,i} \to \mathbf{M}_m$	Cost of scheduling or continuing to run on machine $\mathbf{M}_m$.
ψ	–	Cost of transferring 1 GB across ToR switch.
ξ	–	Cost of transferring 1 GB across aggregation switch.
d^*	–	Data transfer cost for given locality (*data term*).
p^*	–	Opportunity cost of preemption (*preemption term*).

Table C.2: Cost parameters in the Quincy cost model and their roles.

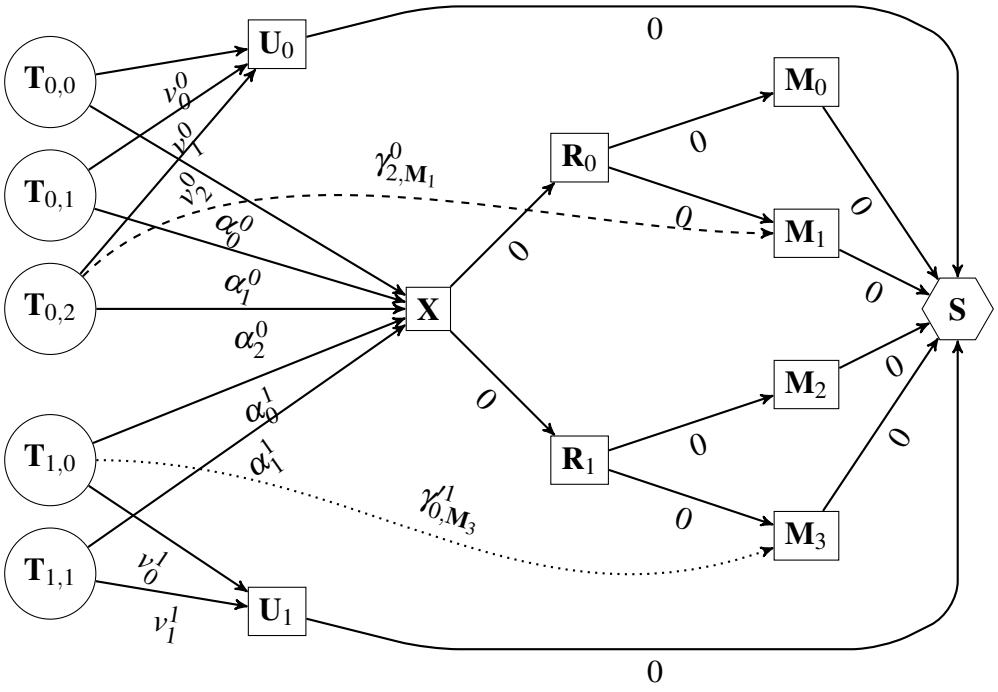


Figure C.3: The example flow network in Figure 6.1 with costs according to the Quincy cost model added to the arcs. The capacities are assigned as in Figure C.2.

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 θ_i^j, the number of seconds for which task $\mathbf{T}_{j,i}$ 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].

Parameter	Value
v_i^j	ωv_i^j
α_i^j	$\psi \mathcal{R}^X(\mathbf{T}_{j,i}) + \xi \mathcal{X}^X(\mathbf{T}_{j,i})$
$\rho_{i,l}^j$	$\psi \mathcal{R}_l^R(\mathbf{T}_{j,i}) + \xi \mathcal{X}_l^R(\mathbf{T}_{j,i})$
$\gamma_{i,m}^j$	$\begin{cases} d^* & \text{if not running} \\ d^* - p^* & \text{if running on } m \end{cases}$
d^*	$\psi \mathcal{R}_m^C(\mathbf{T}_{j,i}) + \xi \mathcal{X}_m^C(\mathbf{T}_{j,i})$
p^*	θ_i^j

Table C.3: Parameter values used in the Quincy cost model.

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 §6.2.1), 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 2.1.3, 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 Flow scheduling limitation details

In Section 6.3.4, 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.4.1 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, preemptions, 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 $\mathbf{T}_0$ goes here, $\mathbf{T}_1$ must go there", where both $\mathbf{T}_0$ and $\mathbf{T}_1$ 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 (§6.3.1) and the n-choose-k constraint can be modelled using Firmament's support for relaxed gang scheduling (§6.3.3).

Few other schedulers support generalised combinatorial constraints. One exception is the work by Tumanov *et al.* (alsched [TCG+12] and tetrisched [TZP+16]), 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 (§6.3.2), 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.4.2 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 (§7.3) uses the latter approach.

C.5 Firmament cost model API

Listing C.1 shows the current Firmament cost model API.

```
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(
      const ResourceDescriptor& source,
      const ResourceDescriptor& 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 TaskToEquivClassAggregatorCost(TaskID_t task_id,
                                                EquivClass_t tec) = 0;
  virtual pair<Cost_t, int64_t> EquivClassToResourceNodeCost(
      EquivClass_t ec,
      ResourceID_t res_id) = 0;
  virtual Cost_t EquivClassToEquivClassCost(EquivClass_t tec1,
                                            EquivClass_t tec2) = 0;
  /**
   * Methods to determine equivalence classes.
   */
  virtual vector<EquivClass_t>* GetTaskEquivClasses(TaskID_t task_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;
}
```

Listing C.1: The Firmament cost model API header.