Title
Hardware and Software Support for Managed-Language Workloads in Data Centers

Author
Maas, Martin Christoph

Publication Date
2017

Peer reviewed|Thesis/dissertation

Hardware and Software Support for Managed-Language Workloads in Data Centers

by

Martin Christoph Maas

A thesis submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Krste Asanović, Chair
Professor John Kubiatowicz
Professor Randy Katz
Professor Joshua Bloom

Fall 2017

Hardware and Software Support for Managed-Language Workloads in Data Centers

Abstract

Hardware and Software Support for
Managed-Language Workloads in Data Centers

by

Martin Christoph Maas

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Krste Asanović, Chair

An increasing number of workloads are moving to cloud data centers, including large-scale machine learning, big data analytics and back-ends for the Internet of Things. Many of these workloads are written in managed languages such as Java, Python or Scala. The performance and efficiency of managed-language workloads are therefore crucial in terms of hardware cost, energy efficiency and quality of service for these data centers.

While managed-language issues such as garbage collection (GC) and JIT compilation have seen a significant amount of research on single-node deployments, data center workloads run across a large number of independent language virtual machines and face new systems challenges that were not previously addressed. At the same time, there has been a large amount of work on specialized systems software and custom hardware for data centers, but most of this work does not fundamentally address managed languages and does not modify the language runtime system, effectively treating it as a black box.

In this thesis, we argue that we can substantially improve the performance, efficiency and responsiveness of managed applications in cloud data centers by treating the language runtime system as a fundamental part of the data center stack and co-designing it with both the software systems layer and the hardware layer. In particular, we argue that the cloud operators' full control over the software and hardware stack enables them to co-design these different layers to a degree that would be difficult to achieve in other settings. To support this thesis, we investigate two examples of co-designing the language runtime system with the remainder of the stack, spanning both the hardware and software layers.

On the software side, we show how to better support distributed managed-language applications through a "Holistic" Language Runtime System, which treats the runtimes underpinning a distributed application as a distributed system itself. We first introduce the concept of a Holistic Runtime System. We then present *Taurus*, a prototype implementation of such a system, based on the OpenJDK Hotspot JVM. By applying Taurus to two representative real-world workloads, we show that it is effective both in reducing the overall runtime and resource consumption, as well as improving long tail-latencies.

On the hardware side, we describe how custom data center SoCs provide an opportunity to revisit the old idea of hardware support for garbage collection. We first show that garbage collection is a suitable workload to be offloaded from the CPU to data-parallel accelerators, by demonstrating how integrated GPUs can be used to perform garbage collection for applications running on the CPU. We then generalize these ideas into a custom hardware accelerator for garbage collection that performs GC more efficiently than running the operation on a traditional CPU. We show this design in the context of a stop-the-world garbage collector, and describe how it could be extended to a fully concurrent, pause-free GC.

Finally, we discuss how hardware-software research on managed languages requires new research infrastructure to achieve a higher degree of realism and industry adoption. We then present the foundation of a new research platform for this type of work, using open-source hardware based on the free and open RISC-V ISA combined with the Jikes Research Virtual Machine. Using this research infrastructure, we evaluate the performance and efficiency of our proposed hardware-assisted garbage collector design.

Contents

Acknowledgments

This thesis would not have been possible without the support of everyone who contributed directly or indirectly to its creation.

I owe a tremendous amount of gratitude to my advisors, Krste Asanović and John Kubiatowicz. Throughout my PhD, they provided me with the support, insight and advice that enabled me not only to pursue a wide range of exciting projects, but also work within the context of a larger research vision that was embodied by large-scale projects such as the ParLab, ASPIRE, FireBox and RISC-V. I especially want to thank them for the trust they placed in me by giving me a large amount of academic freedom, and for their support through technical conversations, funding, and advice. I could not have imagined two better advisors, and I owe them much of the credit for making my time in graduate school not only productive but also enormously enjoyable.

The work presented in this thesis is based on several co-authored papers. I therefore want to thank all of my co-authors: Krste Asanović, Tim Harris, Anthony Joseph, John Kubiatowicz, Jeffrey Morlan and Philip Reames. In recognition of the fact that all of the authors contributed to this work, the thesis is written in first person plural.

This thesis is primarily based on three papers: "GPUs as an Opportunity for Offloading Garbage Collection" (presented at the *International Symposium on Memory Management* in June 2012), "Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications" (presented at the *International Conference on Architectural Support for Programming Languages and Operating Systems* in April 2016) and an unpublished manuscript under submission at the time of writing. The thesis also incorporates several workshop papers published between 2014 and 2017. Using the material in the context of this thesis has been approved by UC Berkeley and all co-authors, and the material has been incorporated into a larger argument (details can be found at the end of the document). I now want to highlight the contributions of several people who have had a large impact on this work, and on my research in general.

I want to thank Tim Harris, with whom I have collaborated on a large portion of the work presented in this thesis, specifically the work around *Holistic Runtime Systems*. Tim has been a mentor since I was an undergraduate student at the University of Cambridge and worked with him on my Part II Dissertation. After I left Cambridge, we worked together during two internships at Oracle Labs and continued this collaboration after my return to Berkeley. Tim has had a tremendous impact on my research career and I want to thank him for all his support and guidance throughout the years.

I want to thank Philip Reames for working together on the GPU-based garbage collector from Chapter 6, as well as six years of great discussions about managed runtimes, compilers and garbage collection. Philip and I worked closely on all aspects of the GPU-based GC, and the work presented in Chapter 6 should be seen as the result of a close collaboration. I also want to thank Jeffrey Morlan for his contributions to the project, and for implementing the reference graph prototype in JikesRVM (Section 6.4.3). Finally, I want to thank Anthony Joseph for advising us on this project. Working with Anthony, Jeffrey and Philip was a

tremendous amount of fun, especially as this was my first research project at UC Berkeley when I was starting out as a new PhD student.

My work on the hardware-assisted garbage collector presented in Chapter 7 started in 2012, shortly after the conclusion of the GPU-based garbage collector. Throughout the project, I was inspired by conversations with several researchers. I want to especially thank Mario Wolczko, who gave a large amount of feedback over the years (including on this thesis), and shared his experiences and insights from working on earlier hardware-assisted GC schemes at Sun and Oracle. Our conversations played a major role in shaping my views on the trade-offs between different garbage collector designs and helped me understand the design requirements for garbage collectors in production settings. I also want to thank Steve Blackburn, Kathryn McKinley and Gil Tene for feedback at various stages of the project. Last but not least, I want to thank Ben Keller for working together on a precursor of this work during a class project for CS252 in Spring 2014, which kick-started this effort.

The research methodology presented in Chapter 8 and the hardware-assisted GC prototype from Chapter 9 would not have been possible without a wide range of infrastructure developed at the Berkeley Architecture Research Group (BAR). This includes the RISC-V instruction set, software ecosystem and open-source hardware built around Rocket Chip. I want to specifically thank Christopher Celio, Henry Cook, Palmer Dabbelt, Yunsup Lee, Albert Ou and Andrew Waterman for building large parts of this infrastructure.

I want to thank Jonathan Bachrach, Adam Izraelevitz, Jack Koenig, Jim Lawson, Richard Lin, Chick Markley, Stephen Twigg and the entire Chisel team for their work, as well as David Biancolin, Sagar Karandikar, Donggyu Kim and Howard Mao for their work on the MIDAS and FireSim infrastructure that enabled my evaluation. I also want to thank Edward Wang for his work on the Hammer synthesis scripts that I used to drive Synopsys Design Compiler, and Colin Schmidt for his help synthesizing Rocket Chip. Finally, I want to thank all my colleagues at BAR for an amazing time in graduate school, including Rimas Avizienis, Scott Beamer, Sarah Bird, Ben Keller, Albert Magyar, Nathan Pemberton, Zhangxi Tan and Brian Zimmer. I could not have imagined a better group to be a part of, and had some of the best years of my life working in our group. A very special thanks is owed to Eric Love for being a close confidant ever since our first days at Berkeley, and Mohit Tiwari for his mentorship and collaboration while working on my Master's Thesis on secure processors.

Thanks are also owed to the ParLab and ASPIRE staff who have supported me throughout my entire time at Berkeley: Ria Briggs, Tami Chouteau and Roxana Infante. I also want to thank Kostadin Ilov, who has spent late nights and weekends in the lab to help me with the research infrastructure. Finally, conversations with a wide range of faculty in the lab have substantially shaped my views, and I therefore want to thank Jonathan Bachrach, Eric Brewer, Randy Katz, Kurt Keutzer, Bora Nicolić, Dave Patterson, Vladimir Stojanovic and John Wawrzynek for everything they taught me throughout my years at Berkeley.

I want to thank my dissertation committee – Krste Asanović, John Kubiatowicz, Randy Katz and Joshua Bloom – for their comments on this thesis, which has helped to improve it tremendously. I also want to thank them for their feedback during my qualifying exam, which has helped shape the direction of this work.

I also want to thank everyone else who has provided feedback to this work throughout the years, including Michael Armbrust, Peter Bailis, Juan Colmenares, Daniel Goodman, Kim Keeton, Peter Kessler, Kay Ousterhout, Margo Seltzer, David Sheffield, Mario Wolczko and Reynold Xin. I also want to thank the anonymous reviewers of ISMM'12, WRSC'14, HotOS'15, SOSP'15, ASPLOS'16, ASBD'16, HotOS'17 and CARRV'17.

Finally, I want to thank my family – my parents Georg and Susanne, and my siblings Kirsten, Lukas and Jonas – for all their support throughout my years, and their encouragement to pursue a doctorate degree. And last but not least, I want to thank my wonderful wife Yucy for her support, advice and patience through all the years I spent in graduate school. Without her continued support, none of this would have been possible.

Chapter 1

Introduction

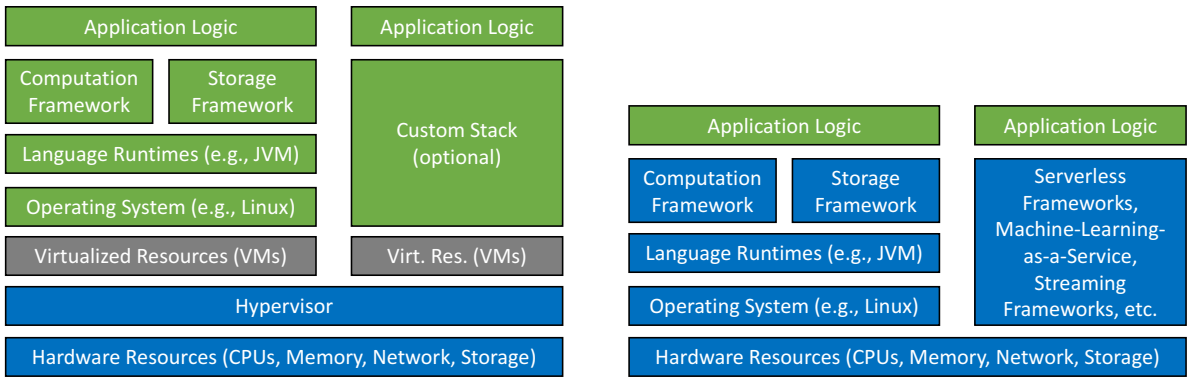
This chapter describes current trends in cloud data centers and how these trends change the role of the language runtime system in the data center stack. We first highlight challenges that current and future language runtime systems face in this setting. We then introduce the research contributions of this thesis, and how they address these challenges.

1.1 Trends in Cloud Data Centers

An increasing number of server workloads are moving to the public cloud. Cisco's Global Cloud Index estimates that by 2020, 92% of data center workloads will run in cloud data centers [52], and the market size for public cloud resources is estimated to increase to an annual $162B [57]. Much of this growth comes from the major cloud vendors (*Amazon Web Services*, *Microsoft Azure* and *Google Cloud Platform*), with increases of year-over-year sales ranging from 43% to 93% [139] in 2016-2017. Further, surveys among businesses [1] indicate that cloud adoption is seen as one of the most important shifts by business leaders.

As its adoption is growing, the public cloud's deployment model is moving from an Infrastructure-as-a-Service (IaaS) model to a Platform-as-a-Service (PaaS) model. With IaaS, customers buy cloud resources from the cloud operator, such as compute (i.e., virtual CPUs and machine VMs), network bandwidth or storage. They then deploy their own software stack, such as computation frameworks, storage layers or application-level logic (Figure 1.1). In the PaaS model, the cloud operator provides high-level services such as databases, machine-learning frameworks or speech recognition, and customers access these services through high-level APIs [55]. This model is currently being adopted by all major cloud providers, including Amazon (e.g., *Amazon Polly* [7], *Amazon Aurora* [5]), Google (e.g., *BigQuery* [32], *Machine Learning Engine* [182], *Cloud Speech API* [205]) and Microsoft (e.g., *Speech API [34]*, *Face API* [70]).

This trend decouples the application from the underlying infrastructure and brings a unique opportunity for cloud operators to replace any part of the stack, including hardware, operating system, and language runtime system. Emerging data center designs are already taking



(a) Infrastructure as a Service (IaaS) **(b)** Platform as a Service (PaaS)

Figure 1.1: *Comparing Infrastructure as a Service (IaaS) and Platform as a Service (PaaS).* Blue indicates parts of the data center stack managed by the cloud operator, green parts are managed by the customer. With IaaS, customers rent hardware resources from the cloud operator and run their own software stack. In the PaaS model, the cloud operator runs applications directly and sells services such as machine-learning resources or databases.

advantage of this, through custom hardware [46, 174] and a shift to resource disaggregation in server racks [77]. One example is TensorFlow [2]: While users can rent TensorFlow resources from Google and program against a high-level Python API, computation can run on CPUs, GPUs or custom hardware called *Tensor Processing Unit* (TPUs [125]).

In the PaaS setting, application developers increasingly focus on high-level functionality such as application logic, processing pipelines and mathematical models, while performance-critical components such as machine-learning infrastructure or data stores are managed by the cloud operator. This is especially true with the emergence of *serverless* frameworks such as Amazon Lambda [19], Azure Functions [20] or Google AppEngine [84], where cloud customers implement their application logic as high-level functions in languages such as Python, and the cloud operator deploys and scales these applications transparently.

As a result of these trends, applications are increasingly written in high-level *managed languages* such as Java, JavaScript, Python, or Scala, while the underlying services are provided by the cloud platform. This shift gives the cloud provider the freedom to implement these services and frameworks using any language (including C/C++, Go, or Rust), co-design them with the platform they are running on, or even deploy custom hardware.

1.2 Role of the Managed-Language Runtime System

The trend towards PaaS elevates the role of the language runtime system. It now becomes the component that connects applications to the services they are using, is responsible for reliably executing a large number of potentially latency-sensitive serverless functions, and is targeting new hardware such as resource-disaggregated systems or custom hardware accelerators.

At the same time, managed runtime systems have not fundamentally changed over the past 10 years, although they were originally designed for very different scenarios. It is therefore unsurprising that problems have been reported in connection with managed languages in the cloud setting. These problems range from performance overheads and unpredictability from garbage collection (GC) [80, 150] to memory bloat [164] and long startup times [143].

Meanwhile, the PaaS model also decouples the application from the underlying data center stack, including the language runtime system, operating system, distributed system infrastructure and hardware. This new flexibility provides an opportunity to rethink the language runtime system and how it interacts with the rest of the data center stack: As long as the language-level programming interface remains unchanged, the cloud operator can replace any layer underneath it. In particular, they can now co-design the language runtime system with the rest of the software stack, and even the data center hardware.

Despite these opportunities, most work on managed runtimes has only investigated the language runtime system in isolation, without taking into account the data center environment. In contrast, there exists little research that spans the boundary to other parts of the data center stack, such as the distributed systems layer or the hardware. Instead, most research on managed runtime systems considers the hardware and operating system as fixed. The reverse is true as well: most systems and hardware research considers the language runtime as part of the application, essentially treating it as a black box.

1.3 Summary of Contributions

In this thesis, we argue that the rapidly evolving data center environment introduces both an opportunity and a necessity to look at the managed runtime system not in isolation, but to co-design it with the hardware and the systems software layers.

We first discuss a general approach to redesigning the language runtime system for future cloud data centers. We then demonstrate the necessity of working across the boundary of the language runtime system through two examples that both address a specific problem: the pause times and overheads introduced by the garbage collector, which is an integral part of any managed runtime system. Specifically, this thesis makes the following contributions:

1. **Holistic Runtime Systems:** We introduce the concept of a *Holistic Language Runtime System* to better support managed data center applications. A Holistic Runtime System is a distributed language runtime system that treats the runtimes underlying a distributed application as a distributed system itself. We argue that by bridging the barrier between the systems and the language-runtime layers, we can address managed-language problems in data centers in a universal and flexible way.

   After describing Holistic Runtimes in their general form, we present Taurus, a prototype Holistic Runtime System based on the OpenJDK Hotspot JVM. Taurus can coordinate managed-language workloads across machines in a data center and introduces a custom Domain-Specific Language (DSL) to describe coordination strategies. We evaluate

Taurus's performance to show that it scales to several hundred machines without introducing substantial performance overheads.

Using Taurus, we show how a Holistic Runtime System can be used to address the problem of garbage collection pauses by coordinating between different runtime system instances. We demonstrate the effectiveness of this approach by applying it to two real-world workloads. We then provide a nomenclature to generalize these strategies, to show how they may extend to other workloads.

2. **Offloading Garbage Collection to GPUs and Custom Hardware:** We present a second cross-layer approach to address garbage collection problems in data-center applications, by offloading GC from the CPU to other hardware. We first show how to offload garbage collection to integrated GPUs. We then introduce a hardware-assisted collector design that offloads GC to custom hardware accelerators close to memory.

3. **Methodology for Hardware-Software Research on Managed Runtimes:** Hardware support for managed runtimes has traditionally been an area that is difficult to evaluate, due to the lack of available research infrastructure. To address this problem, we ported the Jikes Research Virtual Machine [4] to RISC-V [225], an open instruction set with associated open-source hardware [18]. This creates the foundation of a new open-source evaluation platform for managed-language research.

   Using this new simulation infrastructure, we perform an initial full-stack evaluation of our hardware-assisted garbage collector design, to demonstrate its effectiveness and understand the design trade-offs.

1.4 Thesis Organization

We first provide an overview of background material related to this thesis, specifically on data centers, managed runtime systems and garbage collectors (Chapter 2). We then discuss challenges of managed runtime systems in data centers and introduce the concept of a Holistic Runtime System to address these challenges (Chapter 3). Next, we present and evaluate a prototype of such a Holistic Runtime System (Chapter 4) and apply it to the problem of garbage collection pauses, to show its effectiveness (Chapter 5).

In the second part of the thesis, we show a different strategy to address the same garbage collection problem, by offloading garbage collection itself onto different hardware. We first show how to offload GC to an integrated GPU (Chapter 6) and then generalize this approach to a custom hardware accelerator (Chapter 7). As this kind of hardware is difficult to evaluate using traditional research infrastructure, we next present a platform for hardware-software co-design of managed language workloads, based on the Jikes Research Virtual Machine and RISC-V (Chapter 8). Finally, we use this infrastructure for a preliminary evaluation of our hardware-assisted garbage collector (Chapter 9). We conclude with a discussion of future work (Chapter 10), and how this research fits into future cloud data centers (Chapter 11).

Chapter 2

Background

This chapter provides an overview of data center architectures and their relation to managed-language runtime systems. We discuss the structure of these runtime systems, their different components and how they are used in cloud data centers. Finally, we provide a survey of research on one of these components, the garbage collector.

2.1 Cloud Data Centers

Cloud data centers as they exist today have evolved in three stages [17]. The data centers that powered the Internet revolution of the 1990s were built from off-the-shelf servers and other components. These servers either ran proprietary applications such as Google Search, hosted websites or were available for rent. The structure of these data centers resembled traditional networks of workstations [68], connected with off-the-shelf network hardware.

In the early 2000s, as these data centers grew, companies such as Google started designing their own servers [87]. Components such as CPUs were still off-the-shelf, but the system design (and increasingly other parts, such as switches [201]) became custom. This trend led to the OpenCompute [213] project which provides an open standard for these system designs. Meanwhile, we also saw the ascent of modern IaaS, where customers can rent resources (typically in the form of virtual machines) and elastically adjust their resource allocation to meet their requirements. This model was notably different from the prevailing model of renting a fixed number of servers and gave customers more flexibility, as well as enabling dynamic sharing of data center resources between customers. This reduced prices and made it feasible for companies to move their existing infrastructure into the cloud.

In recent years, both of these trends have continued, and we are seeing a move towards an increasing amount of fully custom hardware, with completely custom components such as Google's TPUs and Microsoft's FPGA-based data center infrastructure. With this shift, we have also seen cloud operators sharing resources more efficiently and selling increasingly high-level services such as storage or machine learning in a PaaS setting.

2.1.1 Data Center Overview

Modern data centers consist of up to 100–200,000 servers organized into racks. Each rack contains a top-of-rack switch, and these switches are connected through a high-speed data center network [192]. While cloud operators try to keep data centers as homogeneous as possible, there will typically be several different SKUs within the same data center, as old servers are repurposed and replaced with new machines [186].

Machines are shared between different workloads and users, both internal and external. The partitioning mechanisms differ between data centers and companies, but the most common strategies use virtual machines or containers. Workloads are assigned to machines through a cluster scheduler, such as Borg [221], Omega [198] or Mesos [102]. There has been a large amount of research on cluster scheduling, and schedulers use approaches that are centralized, decentralized, reservation-based or can respond to measurements and feedback from the application (e.g., to identify interference between jobs).

An important challenge in data centers is failure tolerance. Due to their scale, components fail constantly, and applications have to address this through replication, transparent failure recovery and hedging requests. This is typically handled at the software level. Services such as parallel computation and storage are provided by parallel frameworks such as Hadoop [226], Spark [239] or Cassandra [134], which handle failure transparently and are used as building blocks to implement higher-level applications.

While cluster schedulers and frameworks are data-center specific software components, the nodes themselves typically run unmodified operating systems such as Linux, and use conventional language runtime systems such as the JVM. The OS and runtime system are mostly treated as black boxes by the cluster scheduler. Research proposals such as data center operating systems (e.g., DiOS [197]) or Multikernels [29] are intended to lead to OSs that are better-suited for data centers, but are not yet deployed widely.

2.1.2 Data Center Workloads

Data centers run a large amount of code, which can be broadly classified into (1) fully-custom application code, (2) data center frameworks such as Hadoop or Spark and (3) maintenance and support code. While this encompasses a large range of very different workloads, the literature [17, 221] often divides them into two categories:

- **Batch workloads**: These are long-running workloads such as big data analytics, indexing or OLAP processing. As these workloads run for a long time and across a large number of machines, their efficiency is the primary concern.
- **Latency-sensitive workloads**: These are workloads such as user-facing web front-ends or back-ends for real-time systems such as autonomous vehicles, personal assistants or augmented-reality applications. These workloads typically have strict latency requirements that can range from microseconds to several hundred milliseconds. The primary goal is to meet these deadlines, with efficiency as a secondary concern.

Language	Runtime Systems	Examples	TIOBE
Java	OpenJDK, IBM JRE	Netflix, Cassandra, Hadoop, Tomcat	13.0%
C++	N/A	MapReduce, TensorFlow	5.6%
C#	CLR, Mono	Microsoft	4.2%
PHP/Hack	Zend, HHVM	Wordpress, Facebook	2.3%
JavaScript	V8, node.js	Uber, Netflix	2.1%
Ruby	Rubinius, JRuby, MRI	Github, Airbnb, Ruby on Rails	2.0%
Go	Go Runtime	CockroachDB, Kubernetes, Docker, etcd	1.6%
Python	CPython, PyPy	Django, TensorFlow (frontend)	1.6%
Scala	OpenJDK, IBM JRE	Twitter, Spark	0.9%

Table 2.1: *Classifying Workloads in Data Centers by Language.* We list a subset of implementations and projects for each of these languages. Except for C++, all of these languages are managed languages (Go is a hybrid as it does not have a JIT compiler). TIOBE is an index that measures the popularity of programming languages (higher means better).

While this characterization still holds true, the distinction between these categories is starting to blur. The general trend is that *both* types of workloads are operating on ever-increasing data sets, with ever-decreasing response time requirements. As a result, future applications such as autonomous vehicles, real-time security and cloud robotics may require processing on large data sets and millisecond-level response times simultaneously. A particularly important emerging scenario is machine learning for real-time systems: Traditionally, model training is a batch workload while inference is latency-sensitive. With the emergence of real-time machine learning, we are starting to see workloads that require both.

One important factor is that these workloads are not typically monolithic applications but composed of a large number of different components, frameworks and services. For example, a typical Amazon.com request requires 150 different service calls [63]. This has important implications for latency-sensitive workloads: the latency of a request is determined by all of its sub-requests, and as the number of services that are being called grows, the probability that any one of them will delay the request is growing as well, leading to *tail-latency* problems where a certain fraction of requests misses their deadlines. This is a major problem in data centers and techniques to address tail-latency can mitigate the effect [62, 140].

Not only are applications composed of a range of frameworks and services, they are also written in different languages. As a result, data centers are running a range of different runtimes, and workloads are often composed of components running in different language environments. Table 2.1 shows an overview of programming languages used in data centers, together with companies and projects that are using them. We also show the TIOBE index from August 2017 [214], which is a metric indicating their popularity (not limited to the cloud setting). This shows that managed languages play an important role in data centers, are being used as the main language by a number of major companies (e.g., Facebook, Twitter), and run a number of important frameworks (e.g., Cassandra, Spark, Hadoop).

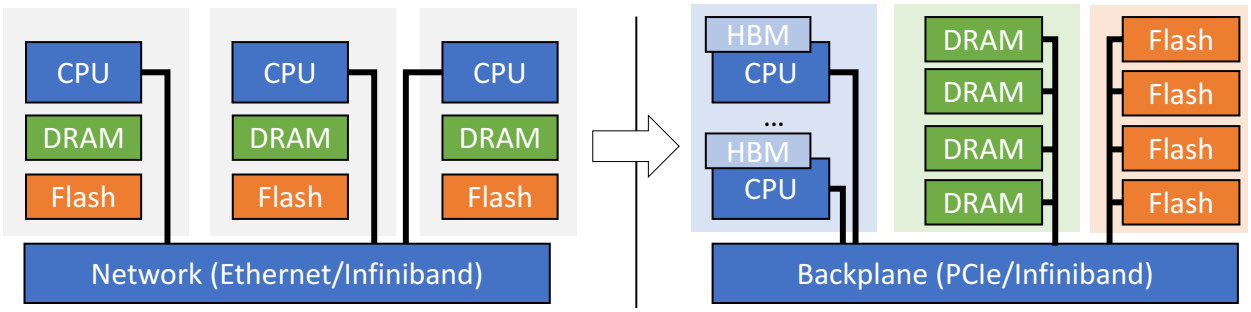


Figure 2.1: *Resource Disaggregation.* The left side shows a conventional server system composed of individual servers with CPUs, memory and storage. In a resource-disaggregated system (right side), resources are instead managed as pools of compute (with high-bandwidth memory), memory and storage, connected by a high-bandwidth, low-latency backplane.

Managed languages are used in two different kinds of scenarios: as application-level languages and to build software infrastructure. The two cases are very different: By improving the infrastructure, we can affect a large fraction of cycles and improve data center efficiency. By improving the application-level, we can improve individual application issues, such as long tail-latencies. As such, both scenarios are important and warrant investigation.

2.1.3 Data Center Trends

With the substantial growth of data centers over the past years, there have been an increasing number of proposals to fundamentally rethink how they are designed and programmed. While there are competing views, several common trends have emerged:

Resource Disaggregation

Facebook [156], HPE [72], Huawei [110], Intel [117] and UC Berkeley [17] have proposed rack-scale system designs where resources are *disaggregated*. Instead of deploying individual servers with a certain amount of compute, memory and storage, all resources in a rack (i.e., storage, memory, accelerators and compute System-on-Chips with a small amount of stacked high-bandwidth memory) are managed in separate pools and connected through a high-bandwidth, low-latency backplane such as PCIe or Infiniband (Figure 2.1).

Compared to a traditional deployment, this reduces the number of different system configurations: Instead of managing several types of nodes with varying combinations of resources to fit the requirements of different workloads, a disaggregated system can allocate exactly the right resource mix to each application. Disaggregation also makes it possible to scale resources independently and does not require keeping idle nodes powered on to retain access to their memory or storage. Finally, resource-disaggregated systems may be more predictable than traditional server deployments, at the cost of increased data movement.

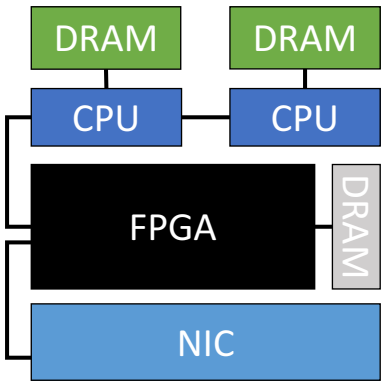


Figure 2.2: *FPGAs in the Data Center.* This deployment model resembles Microsoft's [46]. The FPGA is connected directly to the network, allowing it to handle network requests directly and communicate with other FPGAs to perform large-scale computations.

Hardware Accelerators

Recent work has shown that hardware accelerators can substantially improve certain cloud workloads, and major companies including Amazon [6], Baidu [174], Google [2] and Microsoft [46] are currently adopting them in production. These accelerators come in two flavors: custom ASICs with limited programmability (such as Google's TPU [125] or the DianNao-line of chips [47]) or fully programmable FPGAs (such as Microsoft's Catapult [187]).

The deployment model also differs: while accelerators can be managed as peripherals (similar to GPUs) or pooled as a disaggregated resource, Microsoft recently proposed connecting FPGAs directly to the network (Figure 2.2), which allows the FPGA to handle network requests and dispatch work to the CPU [46].

Serverless Deployment Model

Data center applications are often designed as micro-services that communicate with each other through service-level APIs. The services can be stateful or stateless and are often backed by infrastructure provided by the cloud operator, such as data stores or distributed computation frameworks.

Traditionally, these services were deployed as long-running server instances running within virtual machines or containers. However, there has been a recent shift towards a serverless deployment model, where customers implement their services as high-level functions and the cloud operator provides an orchestration framework that transparently scales and schedules these services as they see fit (e.g., Google AppEngine [84] or AWS Lambda [19]). Container-based orchestration frameworks such as Kubernetes [41] and library operating systems such as Mirage [152] have made it easier to deploy these services in a lightweight way.

2.1.4 Data Center Challenges

While data centers have seen a large amount of attention by industry and the academic community, there are a range of open or ongoing research problems. While this research includes diverse areas such as security or cloud economics, a large portion of projects falls into improving data centers along three categories:

Resource Efficiency

US data centers accounted for 70 billion KWh, or 2% of the United State's energy consumption in 2014 [100]. In 2016, Microsoft, Amazon and Google reported combined capital expenditures of $30B, a large part of which was likely invested into data centers. According to one estimate, the cost of ownership of a 50,000-server data center amounts to $5M per month [91]. With companies such as Google reportedly operating 2-3 million servers each [85], this would amount to $2.4-3.6B per year.

Given this cost, making efficient use of the available hardware resources is crucial, as even 3% percent of performance inefficiency can translate to $100M per year. Improving efficiency has focused on a range of different areas:

- Work on the software stack, including improved cluster scheduling [64, 65, 155, 221], better data center frameworks [82, 239] and improved resource allocation algorithms [81]. Much of this work focuses on batch workloads, but there has been work on reconciling latency-sensitive workloads with high server utilization as well [138].

- More efficient data-center hardware or making better use of the existing hardware [144]. This includes data-center specific optimizations to processors or networking hardware, such as being able to deliver network packets directly to the processor cache. New processor designs also often include new features for specific data center workloads such as machine learning [60].

- Custom hardware accelerators, such as Google's TPU [125] or Microsoft's Catapult platform [187] and Brainwave [51]. These accelerators can potentially execute important workloads much more efficiently than a traditional CPU. For example, Google reported that without the TPU, they would have to double their data center resources [125].

- Improving the system-level design, cooling and rack setup. This includes the introduction of rack-scale systems such as the HP Moonshot [107] and resource-disaggregated systems such as The Machine [72].

In the future, the resource efficiency challenge will be exacerbated by the ever growing scale of cloud data centers and workloads with new characteristics moving into the cloud.

Tail Latency

Predictability of latency-sensitive workloads in data centers is a challenge, and has been widely investigated [62, 140]. Most of these problems are the result of unpredictable pauses or errors in the system that cause a request to miss its deadline (rather than hardware failures). These unpredictable delays can occur in any level of the data center stack, including the hardware, the systems software, the runtime system or the application itself.

Strategies to address these problems fall into two categories: improvements that make components of the system more predictable [127, 173] and strategies that help applications tolerate existing variability (such as hedging requests [62] or detecting interference [240]). Achieving acceptable tail-latencies typically requires a combination of both of these strategies, as building a 100% predictable system is infeasible and tolerating a large amount of unpredictability comes at the cost of increased resource utilization and reduced efficiency.

Tail-latency challenges are becoming more important as time-scales are moving from millisecond to microsecond granularity [28] and latency-sensitive workloads operate on ever-larger data sets. They are also exacerbated by an increasing amount of composition of different services, since a request's latency is determined by its slowest component.

Programmability

An important driver of cloud adoption has been a reduced barrier to entry, and a wide range of companies are aiming to make the cloud easier to program. This work ranges from cloud deployment managers [135] to backup and migration solutions, to high-level frameworks for writing and deploying web applications [2, 21, 84].

There is also an increasing number of tools to obtain more insights into data center applications, including profiling and tracing for distributed workloads [151], understanding consistency guarantees [71] and using continuous deployment in cloud data centers [194]. Finally, there is also work on verification, including generating provable code from high-level specifications [97] and making distributed applications easier to reason about [227].

Despite this progress, we are seeing an emerging programmability challenge through new deployment models such as accelerators, FPGAs and disaggregated hardware. It is an open question how to program for these platforms. At the same time, there has been no consensus on the ideal model for serverless computation.

2.1.5 Summary

These challenges affect all layers of the data center stack, including the hardware, systems layer and language runtime system. In particular, the language runtime system plays a particularly important role, as it connects the application to the underlying platform. Yet, most existing research is limited to one level of abstraction. In this thesis, we are making the case that these problems can be better addressed by working across multiple layers and co-optimizing the language runtime system with the remainder of the stack.

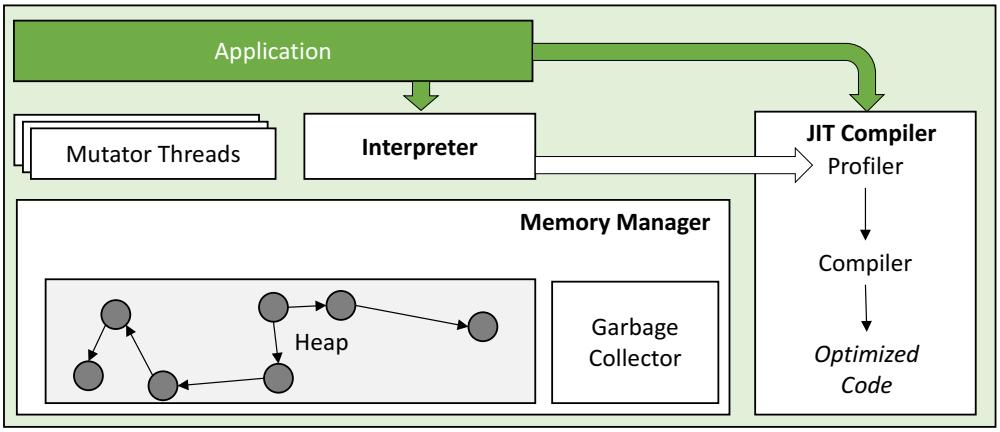


Figure 2.3: *Components of a Managed Runtime.* The heap consists of objects connected by references/pointers, and mutator threads are the threads belonging to the application.

2.2 Managed-Language Runtime Systems

To understand how we can improve the managed runtime system in the data center context, we first have to understand how these systems work, and how managed workloads differ from traditional (or "native") server workloads. While the term *managed runtime* originated at Microsoft and was originally used to specifically describe workloads targeting the Common Language Runtime [89], it is nowadays used as a general term describing language runtimes with the following properties:

1. They run within a virtual machine abstraction (often called a *language VM* to distinguish it from *machine VMs* running in hypervisors). This means that they do not run machine code but instead execute bytecodes targeting an abstract machine model.

2. They deploy some form of dynamic compilation. In most cases, code is either interpreted or just-in-time (JIT) compiled. However, there are managed runtime systems that compile code ahead-of-time and enable dynamic optimization out-of-band.

3. They feature automatic memory management (i.e., garbage collection).

The vast majority of modern languages are managed, including Java, Python, JavaScript, Scala, C#, Ruby, PHP and R. The exceptions include C/C++ and Rust. Go can be seen as a hybrid, since it arguably satisfies properties 1 and 3, but is statically compiled.

Managed runtime systems typically have a much higher degree of complexity than runtimes associated with native languages. In fact, much of the functionality of the language runtime system replicates that within the operating system and compiler, a property that has been pointed out repeatedly [106]. While language runtime systems differ significantly from one another, they typically contain the following components and functionality (Figure 2.3):

Interpreter and/or JIT Compiler: Some managed languages (such as Java or C#) start out with a traditional compiler which produces bytecode that is then loaded by the managed runtime system and executed. Other runtimes (such as Python and JavaScript engines) start from executable code in the target language, parse the code and then execute the resulting program. Both scenarios have in common that once the target code has been translated into bytecodes, it is either interpreted or dynamically compiled to machine code.

Most high-performance managed runtimes feature a tiered compilation system where code is initially executed by an interpreter that profiles the code, determines frequently used sections of the code (known as "hot" functions) and then compiles these functions to high-performance code using a profile-driven JIT compiler.

This approach enables dynamic optimizations such as trace-based inlining. These techniques are crucial for performance of object-oriented languages with a large number of dynamic function calls (e.g., for accessor methods). This approach also enables other optimization techniques such as dynamically picking the right representation for data structures [54], pre-tenuring of objects [95], or translating exceptions into explicit control flow [168].

Memory Management: Since managed runtime systems provide automatic memory management, the runtime system is responsible for tracking liveness of objects and recycling parts of memory that become unreachable. This is the responsibility of the system's garbage collector, which we describe in more detail below. In addition, the memory manager also contains a memory allocator which is responsible for making memory available to the application, and may also be responsible for relocating data structures.

Deoptimization: As JIT compilers transform the original program into a version that differs from the abstract virtual machine's model (e.g., due to inlining), debugging application-level code would be complicated for the programmer. The system therefore needs a mechanism to break at a specific point in the program and translate the optimized program state back to what it would look like in the VM abstraction. This is called dynamic deoptimization [104] and requires a large amount of metadata and machinery in the language runtime system, which is then used by debuggers and for exception handling.

Safepoints: Many operations in managed runtimes require so-called *safepoints* (or *yield points*), which are points throughout the execution where a thread checks whether it needs to stall. This is necessary for triggering garbage-collection pauses, phase shifts in the memory manager, biased locking, deoptimization and on-stack replacement of methods that have been optimized by the JIT. Global safepoints (where the runtime system needs to wait for every thread to reach a safepoint) can become a source of tail latency, and the wait time depends on how frequently safepoints occur. For example, some Java VMs introduces safepoints at every function prologue, epilogue and at every back-edge in the control flow graph.

Native-Call Interface: Since the abstract language VM model means that not all system-level functionality can be implemented in the runtime system itself, managed runtimes typically allow calling into native code through a mechanism such as JNI (Java) and PInvoke (C#). While semantics and implementations differ, implementations of native calls in managed runtimes often make use of `libffi` [142], a popular open-source library that abstracts away the details of the system's underlying calling convention.

Class Library: Managed runtime systems typically provides a standard library of functionality for I/O, OS interactions and common data structures. This is necessary to interact with the outside world from within the VM abstraction, without resorting to native calls.

2.2.1 Managed-Language Challenges

While managed runtime systems are often associated with an increase in programmer productivity, several challenges have been reported in connection with them. These challenges are not unique to the data center setting and apply to a wide range of application scenarios, including mobile and desktop workloads. We now review some of these challenges, and survey research that addresses them.

Managed-Runtime Overheads: A major concern regarding managed runtime systems has been their performance and energy overhead relative to native workloads (i.e., workloads that do not run within a managed environment, such as those written in C/C++).

These overheads stem from different sources. First, the higher level of the language abstraction necessitates features such as bounds checks or dynamic type checking that are not required in a native language and introduce overheads. While this caused major overheads in early implementations of managed languages, many of these overheads can be addressed effectively through speculative execution in trace-based JIT compilers. In fact, the higher-level language can even have advantages: For example, a trace-based JIT can inline functions that a native compiler can not, and strong typing available in languages such as Java can provide better alias analysis [207] and other optimizations. As such, today's managed workloads can even be faster than native implementations in practice [36].

At the same time, managed languages often introduce overheads in terms of memory utilization, through additional information that needs to be stored in object headers. These overheads can be as high as 50%, but are often much smaller in practice [164]. Previous work has shown that this overhead can be reduced through region-based memory management [80, 177] or decoupling the control and data path of objects [164, 165]. Another approach taken by some runtimes is to support value types [33, 184].

Finally, runtime systems often incur startup overheads from warming up the code cache and JIT-compiling functions when they are executed for the first time (this can account for up to 33% of runtime [143]). This problem has been addressed by caching code between executions [143, 162, 243] and ahead-of-time compilation [115, 116, 119].

Garbage Collection: A key problem of managed runtime systems is overhead introduced by the garbage collector. Some applications spend 38% of their overall runtime in the garbage collector alone [43], and while it has been shown that garbage collection can *sometimes* outperform explicit memory management due to increased locality [109], it can be 17–70% slower in memory-constrained environments [101].

For latency-sensitive workloads, pauses introduced by the garbage collector are a major challenge. As we will discuss below, these pauses can range from milliseconds, for concurrent collectors, to minutes, for stop-the-world collectors on large heaps. As these pauses typically cause the application to stall, they can lead to requests missing their deadline. This makes GC a main contributor to long tail-latencies. Many projects have tried to work around this problem, through better concurrent collectors [73, 75, 211], request redistribution [212], reducing memory pressure [164, 165] or avoiding GC in the first place [80].

Performance Unpredictability: The garbage collector is a major source of unpredictability in a language runtime system, but not the only one. For example, JIT-compiled code can lead to performance variations between different runs [27]. Other problems stem from the JIT compiler tuning itself to a specific input and delivering unpredictable performance when confronted with a different input (e.g., a large number of requests after a period of idleness).

Communication Overheads: One problem in managed runtimes is that they typically require serialization and deserialization of data to communicate with other applications. While native-compiled applications can simply share pages between their address spaces, managed-language runtimes manage their own heap and therefore have high communication latencies [172]. Recent projects such as Apache Arrow [11] have tried to improve sharing across managed runtimes, and Sun's Multitasking Virtual Machine [124] and the Microsoft CLR [14] had mechanisms for sharing between applications. More recently, speed-ups of 30× have been shown by being able to optimize across multiple managed frameworks [175].

2.2.2 Summary

While managed runtimes are widely used, they introduce challenges and a large amount of research has been done to address them. The stricter latency requirements, larger working set sizes and architectural changes in future data centers only exacerbate these problems. In this thesis, we are making the case that existing techniques are not sufficient and that some problems can be addressed better by working across the language runtime system barrier.

2.3 Garbage Collection (GC)

Throughout this thesis, we will focus on one specific challenge associated with managed runtime systems, *garbage collection*. We will therefore introduce an overview of how garbage collectors work and briefly survey existing research.

Managed languages typically have mechanisms to allocate memory (e.g., in the form of objects) but not to explicitly deallocate it. The garbage collector's responsibility is to monitor memory and recycle those objects that are not needed anymore[1]. Fundamentally, there are two different types of strategies:

- **Tracing**: Tracing garbage collectors start from a set of roots (e.g., pointers held in stack frames, static variables, or VM data structures) and determine the reachability of objects by traversing the object graph, which consists of the *objects* and *references* (i.e., pointers) between them. Objects are *marked* upon visiting. When the traversal completes, the set of marked objects is the set of reachable objects – all other objects can be recycled (Figure 2.4).
- **Reference counting**: In this case, a count is stored with each object, which indicates the number of incoming edges in the object graph (i.e., how many references point to it). Whenever a reference to the object is stored, the count is increased, and whenever a reference is removed, the count decreases. As soon as the count reaches zero, the object can be deallocated. Reference counting cannot detect cycles and therefore requires a tracing *backup collector* to periodically perform a tracing pass and detect cycles.

As reference-counting collectors still require a tracing backup collector, we see reference counting as an optimization and focus on tracing collectors throughout this thesis. Tracing collectors can be further classified according to how they handle objects once their liveness has been determined:

- **Non-relocating collectors**, such as the classic Mark & Sweep collector, do not move objects in memory and instead add recycled memory to a free list. This is oftentimes used in conjunction with a *segregated free list* allocator, where different free lists are maintained for different size classes of objects. Non-relocating collectors are simple to reason about but have two shortcomings: (1) they introduce fragmentation, which makes them unsuitable for long-running server workloads [74], and (2) they have a slow allocation path, as the allocator needs to traverse the free list.
- **Relocating collectors**, in contrast, move objects in memory, a process known as *compaction*. This reduces fragmentation and enables the memory allocator to allocate from a contiguous region in memory, known as *bump-pointer allocation*. There are many schemes of relocating collectors, from classic semi-space collectors that fold the tracing into the copying phase, to garbage-first collectors [66], where memory is divided into regions and regions with the largest amount of garbage are collected first.

[1]Throughout this thesis, we will use the term *object* to describe the granularity of reclamation. There are non-object-oriented language with garbage collection, such as functional languages where the entity that is collected are stack frames. We use the term to describe these entities as well.

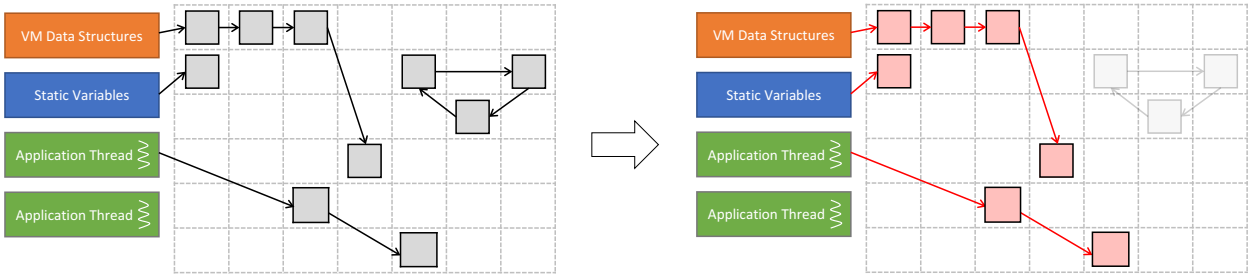


Figure 2.4: *Operation of a Tracing Garbage Collector.* Squares indicate objects in the object graph, and arrows indicate references between them. The collector performs a graph traversal starting from a set of roots, and marks all objects that are reachable from them.

2.3.1 Garbage Collector Trade-offs

Garbage Collection has a large design space. Numerous techniques, algorithms and optimizations exist, and searching for "java garbage collection" on Google Scholar yields almost 50,000 results. Meanwhile, it is rare that one garbage collector is strictly better than another. Instead, the relative performance of collectors depends largely on the specific application, as well as configuration parameters such as the maximum heap size [38]. In fact, a collector that outperforms another by a large margin for one configuration point or application may be substantially slower at a different configuration point (e.g., with twice the heap size). Despite this sensitivity to the configuration, garbage collectors need to make some fundamental trade-offs between several conflicting goals [216]:

- **Application Throughput:** GCs should maximize the overall application throughput. This can be measured as overall execution time for a batch jobs, or as requests per second for latency-sensitive jobs.
- **GC Pause Times:** GCs should minimize pause times due to garbage collection. There are different ways to measure this, and it is important to not only take the mean and median pause times into account, but to consider the tail of the distribution as well (e.g., the maximum pause time and the 99.9 percentile).
- **Memory Utilization:** GCs should make maximum use of the available memory. For example, an application may be operating on a 16 GB heap but can only use 8 GB of the memory due to using a semi-space garbage collection scheme.

The conventional wisdom is that garbage collectors can perform well among any two of these metrics but that there are no garbage collectors that perform well among all of them (Table 2.2). Different types of collectors maximize a different set of goals:

	Stop-the-World	Concurrent	No GC	Ideal
Application Throughput	✓	✗	✓	✓
GC Pause Times	✗	✓	✓	✓
Memory Utilization	✓	✓	✗	✓

Table 2.2: *Garbage Collector Trade-Offs.*

Stop-the-World GC: Stop-the-world collectors perform all of their garbage collection at once. When they run out of memory, they stop the application threads (which are also called *mutators*), perform the collection as quickly as possible (typically using several of the machine's cores, which is known as *parallel garbage collection*), and then resume the application. This maximizes application throughput (as the GC does not interfere with the running application and performs GC as quickly as possible), but leads to long GC pause times, ranging from seconds to several minutes for large heaps [16]. Memory utilization can be high in this scenario, as the GC runs whenever the application is out of memory.

Concurrent GC: Concurrent collectors run at the same time as the application (typically on a subset of the machine's cores). This minimizes pause times (and can avoid them almost completely [53]), but comes at a cost: Since the collector and the mutators are running at the same time, they need to be kept in sync. This is typically done by adding a small instruction sequence known as a *barrier* to every read or write of a reference (Section 2.3.3). However, as this is a very frequent operation, it slows down the mutators and therefore reduces application throughput. While each individual barrier is negligible, the overall application slow-down can be larger than for a stop-the-world GC [31]. Running the GC concurrently also leads to unpredictability from interference, and to lower memory utilization from lower GC throughput (when the collector cannot keep up with the application's allocation rate).

No GC: An extreme design point would be not to collect garbage at all. In this case, the application runs at full throughput and there are no pauses, but the application cannot use most of its memory as it is fills up over time and cannot be reclaimed. This is a largely hypothetical design point to show that collectors can achieve any two out of the three properties, but anecdotally, there have been deployments that preventatively restart the runtime system on a regular basis, to prevent the heap from filling up [88].

There exists an intermediate point between stop-the-world and concurrent collectors, known as *incremental GC*. In this case, the application stops for performing GC but the GC pause time is bounded by a certain time limit – typically below 100ms – according to a configuration parameter. This can achieve better application throughput than a concurrent GC (as well as more predictability), as the barriers required for this type of collector can be implemented with less overhead. This kind of collector has been popular in real-time systems, and IBM's Metronome [23] collector is a prominent example.

Many language runtime systems use some form of stop-the-world garbage collection by default (at least for the old generation). For example, the OpenJDK HotSpot JVM's default collector is a parallel stop-the-world collector and another popular collector is CMS, which is concurrent for the young generation but stop-the-world for major collections. However, concurrent garbage collection has seen renewed focus in recent years. While Azul Systems has sold specialized JVMs with fully concurrent collectors since 2005 [219], there now exist incremental [75] and fully concurrent [73] garbage collectors for OpenJDK. The Go programming language [111] now has a concurrent garbage collector by default as well.

The reason for this development is arguably a combination of shorter response-time requirements (e.g., Go is used for a large number of latency-sensitive systems workloads [212]) and ever-increasing heap sizes. For example, a full GC of a 100 GB heap in HotSpot can reportedly take over a minute [128]. As garbage collection times typically increase linearly with the size of the heap and there are already data center servers with 2 TB of DRAM available [166], we are reaching a point where the only way to achieve acceptable response times is to perform GC concurrently with the application.

2.3.2 Generational Garbage Collectors

Most production-grade collectors combine multiple collection schemes for different portions of memory. This is based on the generational hypothesis, which states that most objects only survive for a short amount of time but that those who survive longer will persist for a very long time. This idea led to *generational collectors* where memory is divided into a young generation that contains freshly allocated objects and an old generation that contains objects that have survived a certain number of collections in the young generation and have been *tenured*. The young and old generation typically employ different collection strategies. In this scenario, a collection of the young generation is known as a *minor* GC, while a full-heap collection including the old generation is called a *major* or *full* GC.

The performance benefits of generational collectors stem from the fact that the young generation is typically much smaller than the old generation and can therefore be collected more quickly. To run a young-generation GC without performing an old-generation collection as well, the collector needs to remember references that point from the old to the new generation in what is known as a *remembered set*. This set is typically maintained through a *write barrier* (Section 2.3.3), which contains code that is executed on every reference write to the old generation and keeps track of all references to the new generation.

This barrier can be efficiently implemented using a scheme called *card marking*: the collector maintains a bitmap that covers different regions of the old generation, and whenever a reference to a new-generation object is written into one of these regions, the corresponding bit is set to true. At the beginning of a minor collection, these regions represent the remembered set and need to be scanned for references.

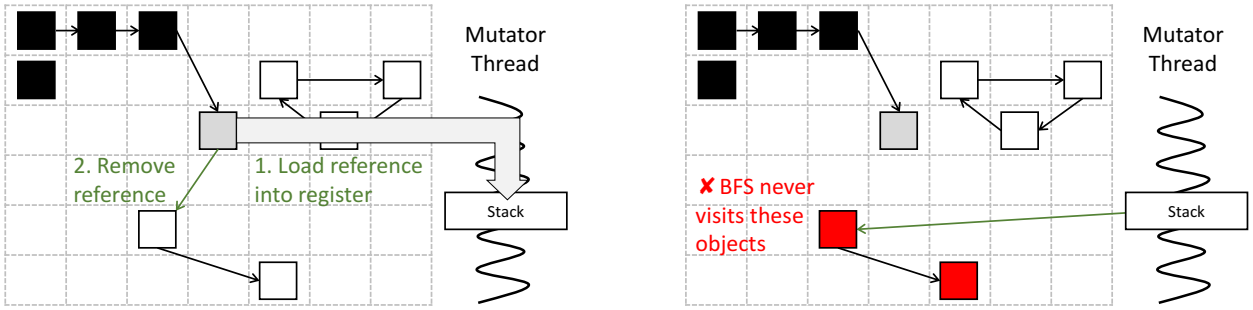


Figure 2.5: Concurrent collectors may violate the property that all reachable objects are marked at the end of the breadth-first-search (BFS). A mutator (green) may read and overwrite a reference during an ongoing BFS, before it has been visited by the collector. In this case, the object is reachable but not marked during the BFS.

2.3.3 Concurrent Garbage Collectors

As concurrent garbage collectors will play a larger role throughout this thesis, we will now briefly describe how they work. While there are many techniques, they all have to solve the same fundamental problem: In a concurrent collector, the GC traverses the object graph at the same time as the application is modifying it, marking objects as it goes along. Meanwhile, the GC may be moving objects in memory while the application is operating on them (this is only the case for relocating garbage collectors). Throughout the execution, the GC needs to ensure that (1) no reachable objects are ever unmarked at the end of the traversal, and (2) the application never accesses an object's old location after it has been moved. Ensuring these two properties is necessary and sufficient for a correctly operating concurrent GC.

We now analyze why these properties might be violated by an incorrect garbage collector, and how concurrent collectors can avoid these problems. For ease of exposition, we assume that the traversal is implemented as a breadth-first search (BFS), which is the most common traversal order in modern garbage collectors.

All reachable objects are marked at the end of the BFS (Figure 2.5)

In a breadth-first search over a graph, there are three types of nodes at any given time during the execution: nodes that have already been visited (*black nodes*), nodes that have not been encountered yet (*white nodes*) and nodes that have been encountered but not visited yet, i.e., they are on the BFS's frontier (*grey nodes*).

In a concurrent collector, the mutator may be overwriting the only reference to a white object with a reference to a black object, while also retaining a reference to the white object, either on its stack or by writing it into a black object. Since the reference was not on the stack at the beginning of the BFS, it is not in the root set, but since the reference was overwritten before it could be traced by the BFS, the object (and any other white object

reachable only via this object) will never be visited and hence not marked. The collector will therefore reclaim the memory of objects that are still reachable, which is incorrect.

Concurrent GCs typically address this problem through a *write barrier*, a small amount of code that is executed on every reference-write by a mutator. There are two options:

1. The write barrier can ensure that each reference that is overwritten gets processed by the garbage collector (i.e., turned into a grey object). This is called *snapshot-at-the-beginning*, since it processes all references that existed at the start of the GC. The barrier code is shown in red below (the `write_field(o, f, r)` operation refers to the operation of writing reference r into field f of object o, `read_field` is the corresponding read operation and `bfs_queue` refers to the frontier of a concurrently running trace/breadth-first search):

```
bfs_queue.add(read_field(obj, REF_FIELD));
write_field(obj, REF_FIELD, newRef);
```

2. The write barrier can ensure that when a reference is written back to a black object, this object is revisited again (i.e., turned into a grey object). This approach may require multiple passes through the heap and also requires scanning the stack for any references to unvisited objects at the end. However, it reduces the amount of floating garbage (i.e., objects that died while the concurrent collection is ongoing). This approach is called *incremental update*. The write barrier looks as follows (`is_marked` returns whether an object is either black or grey, and the barrier protects against the case that a white object is written to such an object, by re-adding it to the queue):

```
if (is_marked(obj))
  bfs_queue.add(obj);
write_field(obj, REF_FIELD, newRef);
```

There are many variations on this general paradigm, and many different ways to implement these barriers. However, the basic idea remains the same.

The application never accesses an object's old location (Figure 2.6)

The other main problem occurs in relocating collectors, when the collector is moving an object while the mutator is trying to access it. This is typically solved through a form of *read barrier*: whenever a reference is read from memory into a register, the barrier checks whether the object has moved and looks up the new location if necessary.

```
ref = read_field(obj, REF_FIELD);
if (has_moved(ref))
  ref = new_location(ref);
```

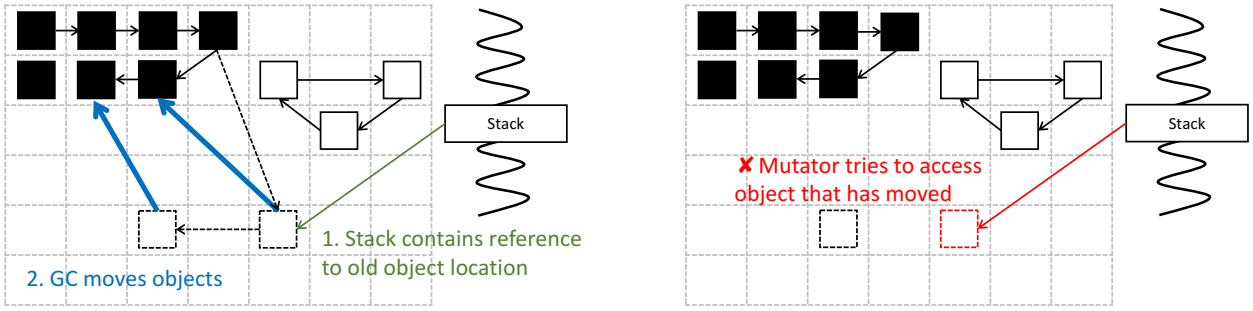


Figure 2.6: Concurrent collectors may violate the property that an application never accesses an object's old location. The collector (blue) may move an object after a mutator (green) has already read the object's reference into one of its registers and is using it to access the object, reading from the old (wrong) location.

One key challenge for read barriers is that they are on the critical path and may involve expensive checks. Further, they do not protect against stale references that are already in the register file. There are two strategies to address this problem: (1) implement the read barrier in a way that can detect when a stale reference is used, e.g., by triggering an exception in the virtual memory system, or (2) only relocate objects at fixed points in the execution (e.g., safepoints) and purge all references from the register file at this point.

From a performance standpoint, barriers need to trade off fast-path and slow-path performance. For example, it is possible to implement read barriers that have very little impact in the common case that the object has not moved, but are expensive if it has (we discuss an example of this approach in Section 7.2.1). Meanwhile, other barrier designs introduce less overheads for the slow path, but are expensive in the common case (an extreme point would be a table of forwarding pointers). Prompted by these overheads, there have been several proposals of moving these barriers into hardware. In this thesis, we discuss several of them (Section 7.2) and propose a new barrier design (Section 7.6.1).

2.3.4 Summary

This section only represents a small subset of work on garbage collectors. A wide range of collector designs exist and have been implemented. A comprehensive summary is provided by Jones & Lin [122], as well as Jones, Hosking and Moss [123]. The mechanisms described here have different trade-offs and may be suited to a varying degree to different execution environments (e.g., mobile, server, embedded).

Chapter 3

Holistic Language Runtime Systems

This chapter introduces and motivates the concept of a Holistic Language Runtime System. We start by discussing why managed languages are widely used in data centers, and demonstrate challenges in real-world managed language workloads that exist today. We then argue that these challenges can be addressed by treating the runtime systems underpinning a distributed application as a distributed system itself.

3.1 Managed Languages in the Cloud

High-level managed languages and frameworks are already a popular way to program the public cloud. PaaS and serverless platforms almost exclusively use managed languages [84, 19, 20], and many frameworks running in IaaS deployments are written in managed languages as well. Arguably, the current source of this popularity are the good productivity and safety properties of managed languages. However, we also believe that there are fundamental reasons that make managed languages an ideal fit for the cloud setting, and we therefore think that the current trends will continue and application-level workloads will almost exclusively run on managed-language runtime systems in the future:

- *Managed languages raise the level of abstraction*: Managed languages hide the complexity of the underlying architecture. This is often considered a disadvantage for system-level software and application scenarios such as HPC (since it prevents some low-level tuning), but in the cloud setting – and specifically the PaaS/serverless setting – a high level of abstraction is necessary since hand-tuning will not be possible.
- *Managed languages provide automatic memory management*: Explicit memory management is prone to errors, and bugs such as memory leaks or buffer overruns can be particularly damaging in a cloud scenario of long-running, security-sensitive workloads that are sharing the same machine. At the same time, there is little benefit of explicit management in an opaque system – automatic memory management and garbage collection (GC) are therefore a significant advantage.

- *Managed languages operate on bytecode*: This allows transparent recompilation and auto-tuning to a particular architecture based on runtime performance counters. There are also high-level frameworks such as SEJITS [45] or Dandelion [191] that can help to program accelerators in a high-level language by using introspection into compiled programs. In particular, this can be used to target the new FPGA architectures presented in Section 2.1.3, by compiling high-level languages to FPGAs [108, 50]. Finally, the high level of abstraction makes it easier to target different instruction sets, which reduces the vendor lock-in for cloud providers. In fact, Google and Microsoft have both announced that they are considering a shift to ARM [86, 222]. Google is also a member of the RISC-V Foundation supporting the free and open RISC-V ISA [225].

- *Managed languages operate on references instead of pointers*: This allows transparent migration of data and execution. This is particularly important with the emergence of disaggregated systems (Section 2.1.3). With resources disaggregated at the rack or data-center level, application-level knowledge is required to decide how to move data between different pools of memory, including high-bandwidth memory on chip and remote memory elsewhere in the rack. While these decisions could be directly exposed to the application, the programmer may not have the information to make the best decision, and the resulting code may not be portable. Instead, previous work has explored page-based migration mechanisms [77], but those work at a coarse granularity and cannot take application-level knowledge into account. Improving over this approach is challenging in native languages, as moving data at a finer granularity than a page requires rewriting pointers. In contrast, managed runtime systems already have a mechanism to relocate individual objects and redirect pointers as necessary. This could enable them to transparently migrate different parts of the heap. Managed runtimes also have mechanisms to measure performance profiles (such as access frequencies), which makes it possible for them to dynamically decide where to place data. As such, they may be able to do a better job than the programmer, and to do so transparently.

3.2 Managed Data Center Workload Challenges

As discussed in Section 2.2.1, managed languages are known to cause performance overheads in some use cases [43, 112], and much work has been done to address these. However, most of this work focuses on individual nodes. When running distributed managed applications and frameworks such as Hadoop or Spark at a rack or data-center scale, the application spans many processes across multiple nodes, each with its own runtime system (Figure 3.1). This causes a largely orthogonal set of problems beyond those encountered within an individual runtime system. Specifically, the language runtime systems on different nodes make completely independent decisions, such as when to perform GC or how to JIT their code. This causes performance problems for many workloads, including both throughput-oriented and latency-sensitive jobs. We identify the following five problems:

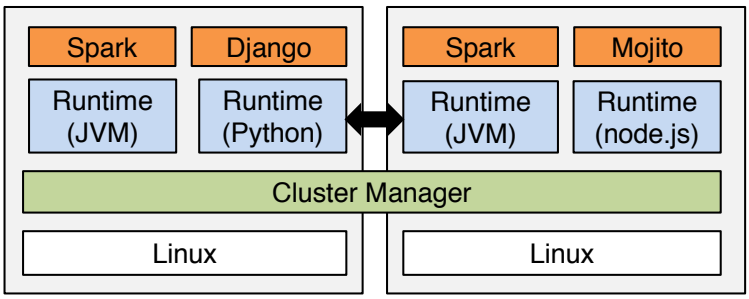


Figure 3.1: Currently, applications run across a number of different machines and different runtime systems. These runtime systems act completely independently from one another and are unaware of the fact that they underpin a distributed applications.

The Coordination Problem: Runtime systems on different nodes are unaware of each other. This means that they make all decisions independently, e.g., when to perform GC. As we show in the next section, this can have a significant impact on distributed workloads, both batch workloads and interactive jobs. Similar problems have been reported in several real-world deployments [98, 181, 80].

The Interference Problem: Runtime systems on the same node do not coordinate. While co-scheduling of parallel [176, 94] and data center workloads [56, 65, 94, 136] is well-investigated and has been addressed by many projects, most of these projects do not look at managed-language-specific issues such as GC interference or instruction cache pollution from multiple copies of the same code. Some distributed Java applications have 100s of instances on the same node – this can make these overheads substantial, and motivated the Multi-tasking Virtual Machine project [124], JSR-121 [126] and Application Domains [14]. These approaches execute multiple applications within the same runtime system, but failed to achieve widespread adoption, presumably due to concerns about failure propagation and difficulty of deployment (two issues we are addressing in this work).

Unfortunately, this problem cannot be solved at either the language runtime or the OS level alone, as the required high-level information about the threads is not available to the OS scheduler, and the runtime system cannot control scheduling across applications.

The Composition Problem: For productivity and maintainability, applications are often broken into a large number of services – e.g. page requests to Amazon.com typically require over 150 service calls [63]. However, when services are shared between different runtime instances, a service call requires crossing the boundary between two processes. This prevents optimizations such as code inlining or service fusion, and adds overheads from context switches and scheduling. At the same time, simply putting all services into the same process would cause problems in current systems, such as losing failure isolation and not being able to dynamically migrate service instances between nodes.

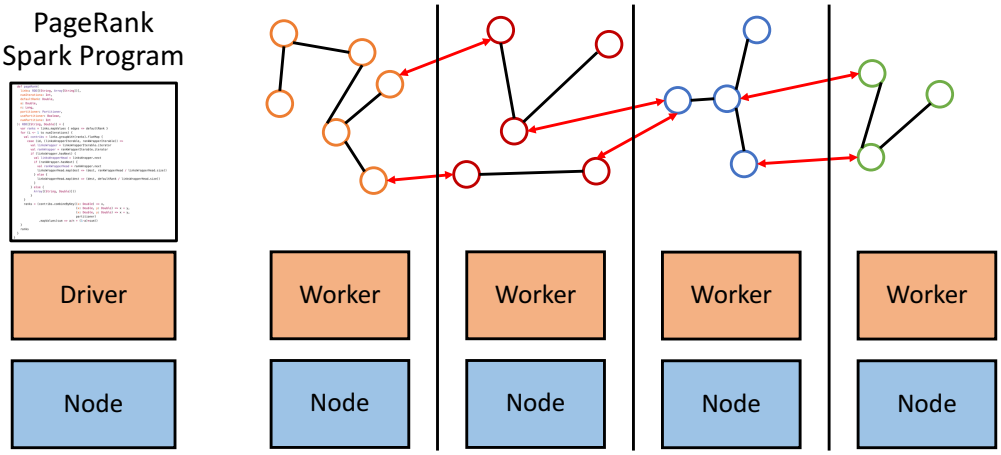


Figure 3.2: *Basic Overview of a Spark Graph Computation.* Vertices of the graph are partitioned across nodes and intermediate data is exchanged between the partitions at the end of every PageRank superstep.

The Redundancy Problem: Every runtime system has its own JIT. As distributed workloads often run the same executable on every node (or even multiple instances of the same executable), this causes wasteful re-JITing of shared code (for example, OpenJDK 7 loads 379 classes for a minimal "Hello World" program). Since the JITed code is tied into the logical data structures of the runtime, page sharing between processes cannot avoid this problem. Companies such as Microsoft are therefore starting to forego the JIT in favor of statically pre-compiled binaries [115, 116]. However, this loses benefits from profile-directed dynamic optimization (e.g., trace-driven compilation and dynamic inlining). Meanwhile, in VM-based deployments, the redundancy problem applies to the OS layer as well. Managed runtimes have a large number of dependencies and VM images often contain an entire OS with a large amount of unused libraries and code, even if only a small subset is used (which can lead to three orders of magnitude overhead in binary size [152]).

The Elasticity Problem: Managed language runtimes take a long time to boot and reach their full performance, partly due to warming up the code cache [143]. This introduces overhead when scaling applications horizontally, as well as time skew.

3.3 Motivating Examples

We will now focus on one of these problems specifically, the *coordination problem*. We show how it materializes in the two main categories of workloads encountered in data centers: throughput-oriented batch workloads and latency-sensitive workloads (Section 2.1.2).

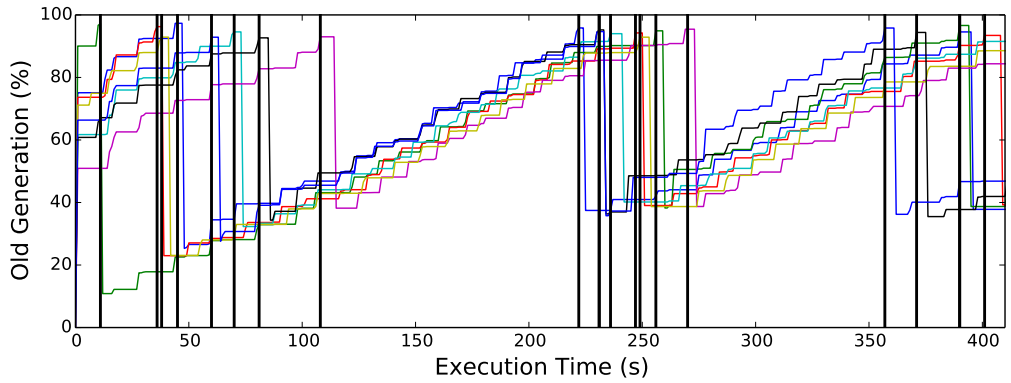


Figure 3.3: *Memory occupancy in the old generation over time on different nodes in a Spark cluster running PageRank.* Colors indicate different nodes. Nodes are filling up at different rates and trigger a GC when they run out of memory, indicated by vertical lines.

3.3.1 Apache Spark (Batch Workloads)

To illustrate the problems caused by a lack of coordination between runtime systems for batch workloads, we use iterative computations in Apache Spark [239] as an example. Spark is a popular Scala-based framework for distributed computation. It can run a wide range of workloads, including graph computations [83], machine learning, database workloads [235] and stream computations [238].

Figure 3.2 shows an example of a Spark workload. Spark is built around the concept of *resilient distributed datasets* (RDDs), which are potentially large data sets that are partitioned across nodes in the cluster and can be kept in memory. Spark programs are written in a high-level language such as Scala or Python and apply operators and transformations to these data sets. Each worker applies these operators to its own partition, while some operators result in communication between the different workers (e.g., shuffle operations).

We use a PageRank graph computation as an example. The graph is first loaded and partitioned across the different nodes. PageRank is an *iterative* computation, and executes in multiple *iterations* (or *supersteps*). In each iteration, the workers operate in parallel on their own data and then need to exchange the intermediate results with one another before being able to continue with the next iteration. Figure 3.4 shows a Spark implementation of the PageRank algorithm, taken from Spark's library of examples (the version we run for experiments differs slightly, using the *Bagel* library).

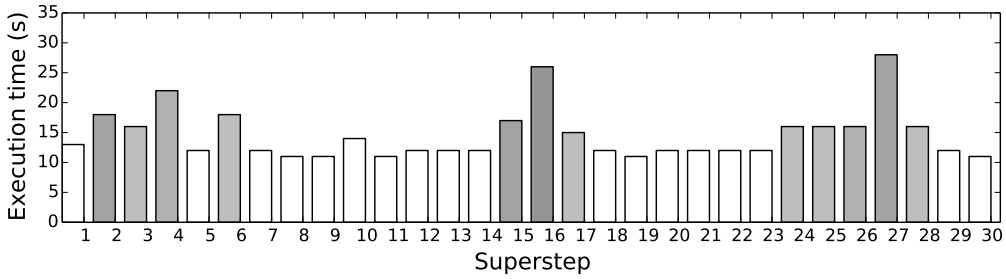
Spark workloads are often memory-intensive. Our example runs across 32 GB heaps, and deployments with heap sizes of 100 GB are known [128]. Spark, by default, uses the parallel scavenge/serial mark-sweep-compact stop-the-world GC in the Hotspot JVM, which has very high collection performance but frequently incurs short pauses to perform young-generation GC (on the order of 100s of ms) and every once in a while incurs very long pauses (up to multiple seconds, and in some cases even minutes [128]) for full GC.

```
var ranks = links.mapValues { edges => defaultRank }
for (i <- 1 to numIterations) {
  val contribs = links.groupWith(ranks).flatMap {
    case (id, (linksWrapperIterable, rankWrapperIterable)) =>
      val linksWrapper = linksWrapperIterable.iterator
      val rankWrapper = rankWrapperIterable.iterator
      if (linksWrapper.hasNext) {
        val linksWrapperHead = linksWrapper.next
        if (rankWrapper.hasNext) {
          val rankWrapperHead = rankWrapper.next
          linksWrapperHead.map(dest =>
              (dest, rankWrapperHead / linksWrapperHead.size))
        } else {
          linksWrapperHead.map(dest =>
              (dest, defaultRank / linksWrapperHead.size))
        }
      } else {
        Array[(String, Double)]()
      }
  }
  ranks = (contribs.combineByKey((x: Double) => x,
                                 (x: Double, y: Double) => x + y,
                                 (x: Double, y: Double) => x + y,
                                 partitioner)
           .mapValues(sum => a/n + (1-a)*sum))
}
```

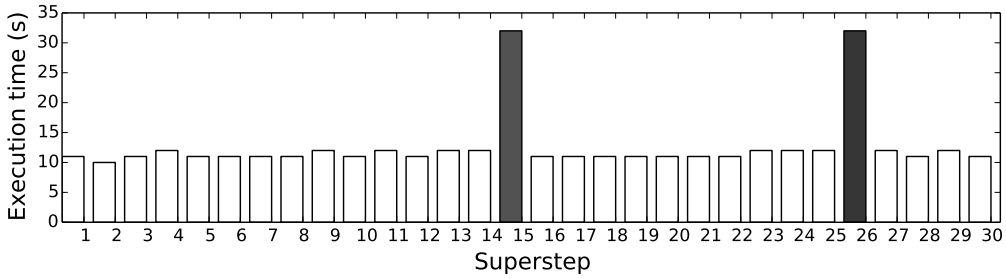
Figure 3.4: *Code of PageRank implemented in Apache Spark.* This listing was adapted from `WikipediaPageRankStandalone.scala` in the Spark 1.1.0 distribution package.

These long pauses cause difficulties during iterative computations. Throughout execution, the memory of each node fills up over time, and once it reaches a threshold, a collection is triggered. Figure 3.3 shows this for a small 8-node cluster running PageRank on the 54 GB *Wikipedia* graph from the original Spark paper [239][1]. During each superstep, Spark launches a number of tasks on each worker node. Once all tasks have completed, the nodes exchange results and can only continue after this exchange has completed. As a result, this step effectively acts as a barrier between supersteps.

[1]Chapter 5 uses a slightly different cluster of 16 nodes, using Spark 1.1.1, OpenJDK 1.7.0_75 with a heap size of 32 GB and the default GC settings. Section 5.2 describes the setup in detail.



(a) Baseline System (no coordination)



(b) Coordinating GC (Stop-the-Universe)

Figure 3.5: *Relation between GC and the superstep durations of Spark PageRank* (shade represents the number of nodes performing GC during a superstep; white = no GC). As soon as one node performs GC, the superstep takes much longer.

In the absence of GC, every superstep takes a similar time (Figure 3.5a). However, when one node performs a full GC, it pauses for multiple seconds and all other nodes have to wait at the barrier for that node to become available again. While waiting, they cannot do any work themselves. Worse, when they continue, they will at some point incur a GC themselves, and become the reason for other nodes to wait (Figure 3.6).

The root cause of this problem is that the runtime systems make independent decisions about when to perform GC. The memory on the different nodes fills up at a different rate, and therefore their GCs will occur at different times. But what if the system was globally coordinated? In that case, the best decision would be to let all nodes do GC at the same time: Since nodes have to wait for a single node in GC, they can use that time efficiently by performing their own GC. We call this policy *Stop-the-Universe*. Figure 3.5b shows its effect: even on this small cluster, coordination leads to a speedup of 15% (excluding the initial loading of the graph from disk).

We note that this problem is common to all distributed iterative computations with global barriers. This includes many machine learning algorithms (e.g., logistic regression) and graph workloads (e.g., shortest path). The problem is somewhat reminiscent of inter-thread

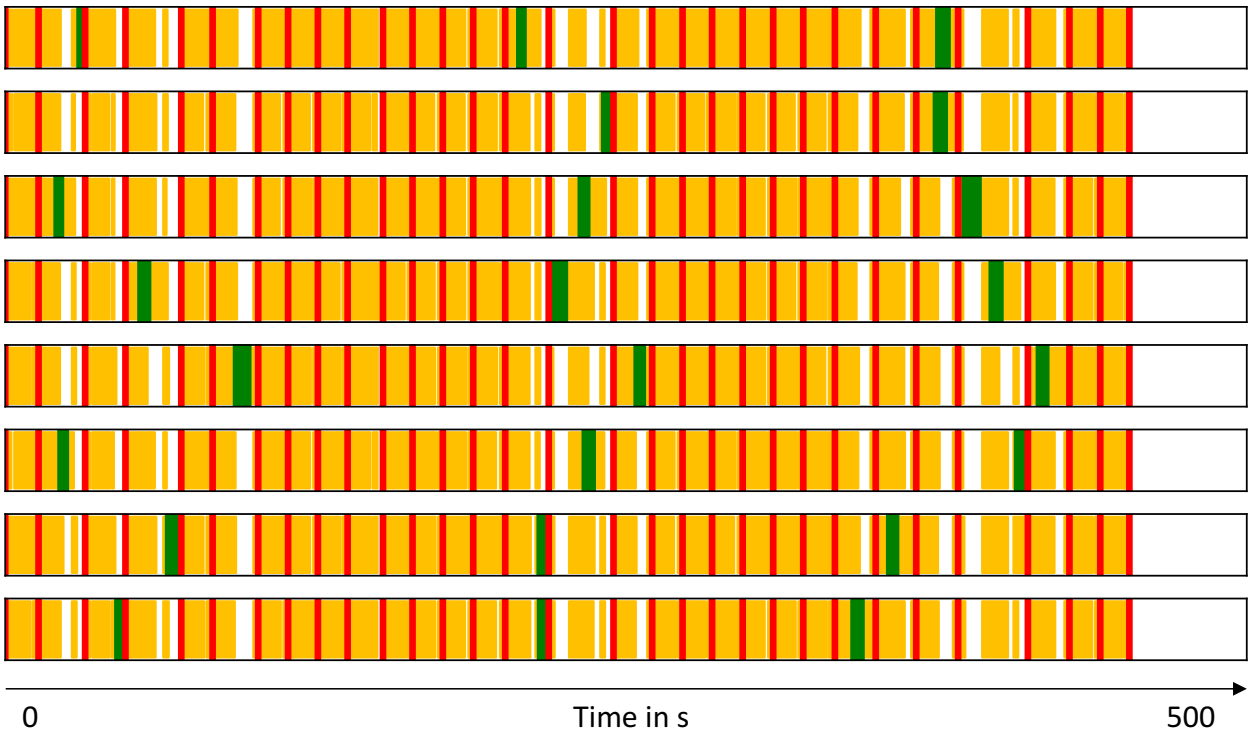


Figure 3.6: *Garbage collection pauses throughout the execution of Spark PageRank on an 8-node cluster* (each row represents a node). White indicates that the node was idle, yellow indicates that it performed work, and red lines indicate the end of a PageRank iteration. Green indicates major GC pauses. GC on one node cause all other nodes to get delayed.

synchronization in parallel workloads and the OS Noise problem in HPC [215]. Both problems are often solved using gang-scheduling, and the *Stop-the-Universe* policy can be seen as its language-runtime equivalent.

While the specific potential speed-up is highly dependent on the application, data set and setup, we think it can be approximated as follows. If, in an iterative computation, iterations take $t_{\text{Iteration}}$ on average (excluding any GC pauses), nodes perform a GC every s iterations on average and spend t_{GC} per GC pause, the potential speed-up is:

$$1 + \frac{(s-1) \cdot t_{\text{GC}}}{s \cdot t_{\text{Iteration}} + t_{\text{GC}}}$$

The rationale is that for a very large number of nodes and long-running computations, at least one node will perform GC within every iteration, while in a system with coordinated GC, only one collection would be necessary every s iterations.

Independently and concurrently to our work, similar problems have been confirmed for Naiad workloads [80], which are based on C#. This indicates that this class of problems applies to a wider range of workloads and managed runtime systems.

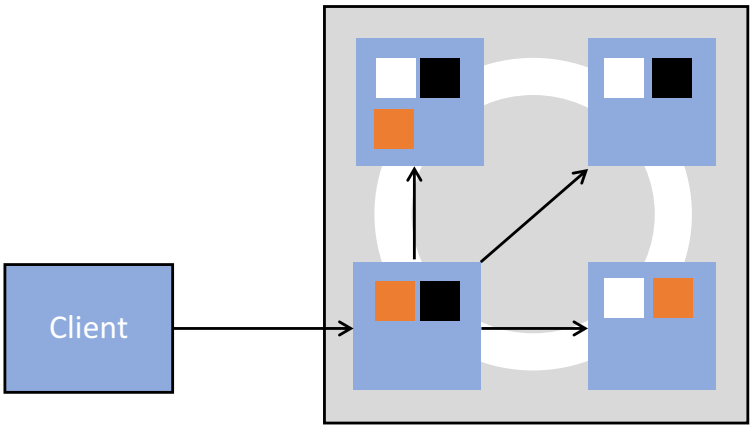
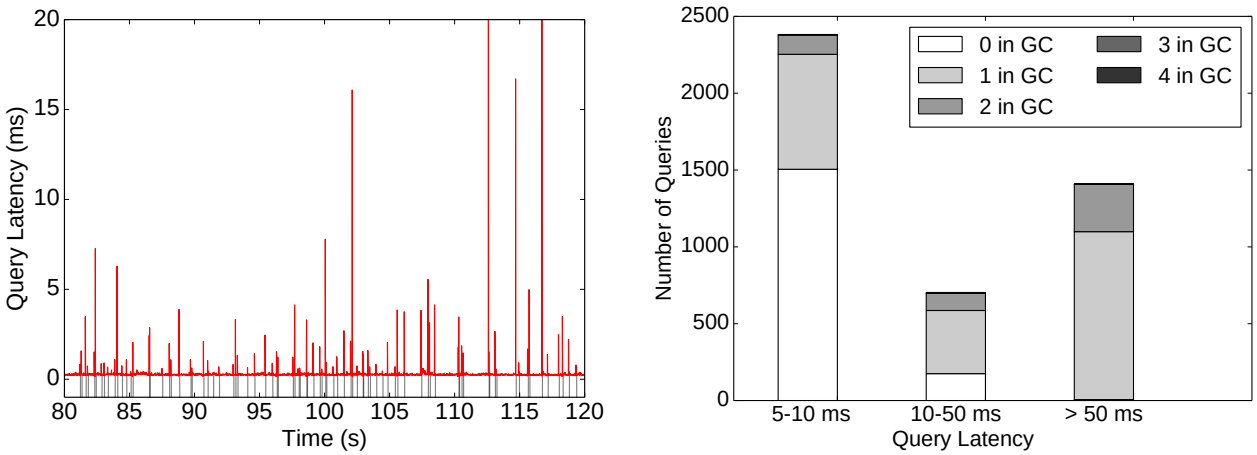


Figure 3.7: *A 4-Node Cassandra Cluster.* The colored boxes represent replicas of key-value pairs. The nodes are logically arranged in a ring and replicas are stored on subsequent nodes in the ring (in this example, there are three replicas per entry). These nodes are determined through consistent hashing. Read and update queries can be sent from a client to any node in the cluster, which then contacts the replicas and assembles a quorum.

3.3.2 Apache Cassandra (Interactive Workloads)

We now demonstrate a different set of problems, which is encountered by latency-sensitive workloads. This includes data stores such as Cassandra [134], client applications such as the Apache SOLR search engine [202] or systems-level software such as ZooKeeper [113]. Data center applications are interacting at ever smaller time-scales (such as in algorithmic trading, real-time bidding for ads, or low-latency storage [171]). Further, applications are often composed of hundreds of services [63] and the expected latencies for individual services have decreased to micro-second granularities. In such a scenario, stragglers are a significant problem since a single straggler can cause an entire request to miss its deadline. GC can be a significant contributor to this problem – even minor-GC pauses on the order of milliseconds are problematic when services operate at micro-second granularity.

We use the Cassandra key-value store as an example. Cassandra uses consistent hashing to replicate data across a subset of nodes. Requests can be sent to any node, which then assembles a quorum by contacting the replicas (Figure 3.7). While Cassandra uses a concurrent collector, it still experiences multi-millisecond pauses for young-generation GC (note that most concurrent collectors still introduce some form of jitter like this). To show the impact of these pauses, we ran a 4-node Cassandra cluster with the YCSB benchmark [59] for 10M queries (details can be found in Section 5.3). While the average latency for reads was 277μs, we occasionally incurred latencies that were over 100× larger. Figure 3.8a shows that these spikes mostly coincide with GC pauses (grey lines). Figure 3.8b shows that, in fact, most requests longer than 10 ms coincided with a GC pause, either in the node that received the request or in a node required to assemble a read or write quorum (others have presented



(a) Request latencies over time

(b) Stragglers coinciding with GC

Figure 3.8: *Correlation between stragglers in Cassandra read queries and GC.* Grey lines in (a) are minor GCs and latencies of (potentially concurrent) request are averaged over 10ms intervals. The average request latency was 277μs.

related results [71]). Note that these numbers might even understate the impact of GC, due to correlated omission in YCSB [223].

The fundamental problem is, once again, ignoring the language runtime system as part of the distributed system. Runtime systems make their GC decisions independently and collect as soon as their young generation fills up, stalling the application without warning. For example, two replicas holding the same entry can become unavailable due to GC at the same time, making it impossible to assemble a quorum. However, services often have a choice where to send a request. For example, Cassandra requests can be handled by any node, but sending a request to a node that starts a GC pause before being able to send a response introduces a straggler.

Many of these problems could be avoided in a globally coordinated system. One approach is to expose the state of all runtime systems to the logic that directs requests to different servers (e.g., the load balancer). This makes it possible to avoid nodes that are close to GC – we call this *Request Steering*. Another strategy is to globally schedule GC such that there is always a sufficient number of replicas of any service available – we call this approach *Staggered GC*. Others have confirmed the efficiency of similar strategies [212].

3.4 What Does Software Do Today?

Problems such as the ones above have been reported for a wide range of applications and frameworks, including Hadoop [98], SOLR [202] and financial applications [181, 61]. They are often solved by rewriting parts of the application in a native language such as C++ and

managing large data structures off-heap [183, 133]. Others use non-idiomatic Java (e.g., large byte arrays), split applications into smaller VMs or control allocation carefully to reduce memory pressure and avoid GC pauses. In fact, the latest versions of Spark and Cassandra both use such techniques (note that our experiments use versions from before these changes). The problem with these approaches is that they lose many advantages of using a managed language in the first place, including safety and productivity.

There is also anecdotal evidence that some distributed applications treat GC as a failure mode like others, and restart the process when a full GC is necessary. The problem with this approach is that it is only viable if GC is rare. As we saw in this chapter, major GCs may occur every few minutes. In the past, some applications also tried to control GC explicitly, using functions such as `System.gc()`. This turned out to be insufficient, since it was difficult for application developers to make good decisions based on the knowledge available. Some runtime systems hence ignore such calls today.

We have also seen a commercial application that steers requests away from nodes paused for GC, similar to our proposed strategy above [181]. Implementation of such strategies is facilitated through language extensions such as C#'s GC Notification API [78] that allow applications to respond to upcoming GC pauses [212]. However, while explicitly designed for this use, we have not seen these APIs being widely used. Our hypothesis is that the mechanism is too low-level; programmers still need to solve distributed systems problems such as multi-node coordination or failures. Making the implementation of such strategies much easier is one of the key goals of the work presented in this thesis.

Finally, a solution to the problem are concurrent garbage collectors such as C4 [211] or G1 [75] (Section 2.3.3). These collectors avoid GC pauses, but instead incur a performance cost from constantly having to trace and compact the heap, and the overhead from read and write barriers. This means that more memory bandwidth and CPU resources are used for GC to achieve the same GC throughput as a parallel stop-the-world collector. Furthermore, most popular concurrent GCs still introduce short pauses or jitter (e.g., the previous Cassandra example used the concurrent CMS collector from OpenJDK). With the very short request latencies required by many services today, this can still be problematic. In fact, we believe that it may be preferable to implement strategies to tolerate rare, predictable stop-the-world pauses than incurring unpredictable jitter from concurrent GC.

While all these solutions work, they often result in error-prone ad-hoc approaches, reduce productivity, redo a large amount of work, are not portable or yield poor performance. We believe that the fundamental problem is that these solutions either treat the runtime system or the underlying systems layer as a black box. In the remainder of this chapter, we argue that we need to work across these layers to solve these problems at a more fundamental level. Specifically, most managed language runtime systems were not originally designed for the data center scenario and have not evolved to meet the new latency requirements in the data center, target new data center architectures and scale to the sizes required by modern data center workloads.

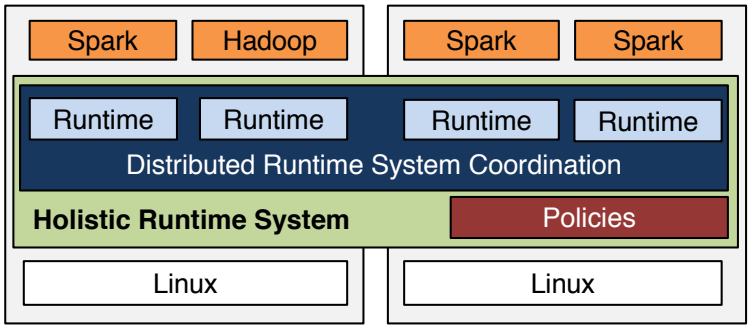


Figure 3.9: A Holistic Runtime System coordinates individual language runtime systems across machines in a cluster, based on a policy.

3.5 The Case for Holistic Runtimes

To address these problems, we propose what we call a *Holistic Language Runtime System* (Figure 3.9). We define Holistic Runtime Systems as follows:

> *Definition:* A **Holistic Language Runtime System** is a distributed language runtime that treats the runtimes underlying a distributed application as a distributed system itself and enables them to make federate decisions globally rather than individually.

As such, Holistic Runtime Systems adopt some of the ideas from distributed operating systems [49, 137, 160, 170, 197] and apply them to language runtimes systems. This approach is motivated by today's low-latency data center networks and a trend towards rack-scale machines that enable a tighter integration of nodes in distributed systems. At the same time, data center workloads interact at ever smaller time-scales. These developments make it both feasible and necessary to integrate the runtime systems in a cluster more closely.

One design point of a Holistic Runtime System would be a monolithic distributed runtime system with a single-system view [149]. However, this approach raises challenges regarding (1) compatibility with existing applications, (2) predictability by hiding the distinction between local and remote memory, (3) failure isolation, and (4) scalability bottlenecks such as those from distributed GC across a shared heap. We therefore propose an intermediate approach that retains the boundaries between individual runtime systems but enables coordination between them. Applications do not need to be aware that the runtime systems they are running across are part of the same logical entity. This enables running unmodified workloads, where the Holistic Runtime transparently applies policies for runtime system components such as the garbage collector or JIT compiler. At the same time, workloads that know they are running on such a system can communicate with it to take advantage of its global knowledge and coordination capabilities.

The Holistic Runtime System has similarities to previous work on Distributed JVMs [10, 15, 147, 241, 242]. However, these JVMs are targeted at monolithic applications, not distributed workloads. There are also projects that share some of our goals: Forseti [42] investigates holistic heap sizing. The MVM [124] looked at running multiple applications in the same JVM. A^2-VM [200] cooperatively schedules Java applications across machines, making the JVM and its services resource-aware to enable cluster-wide thread scheduling based on policies (which bear some resemblance to our policies). Finally, Terracotta [39] deploys Java applications across multiple JVMs through clustering. The last two differ from Holistic Runtimes in that they provide a platform for writing distributed workloads rather than a transparent support layer.

Holistic Runtimes are also related to work on cluster schedulers [102, 198], and share some of their responsibilities. While Holistic Runtimes schedule and coordinate workloads as well, they do so at a much coarser granularity. However, it is possible that Holistic Runtime functionality could eventually be integrated into the cluster scheduler, to reuse its available information and failover mechanisms.

3.6 Application Scenarios

We believe that the idea of a Holistic Runtime System can be applied to a wide range of application scenarios:

- **Coordinating Maintenance Events**: We previously showed an example where uncoordinated garbage collection can cause problems for distributed workloads. The same applies to other maintenance tasks such as log compaction or writing buffers to disk. A Holistic Runtime System can coordinate these tasks across different nodes.
- **Sharing JIT Code Caches**: One instance of the redundancy problem in managed runtimes is that each node JITs the same code and therefore performs redundant work. A Holistic Runtime System could avoid this work by sharing code caches across different runtime instances. It could also reduce instruction cache pressure by enabling runtimes running on the same node to share pages, avoiding the need for multiple copies of the same code in the instruction cache. This also helps address the elasticity problems as it can help nodes to forego warming up code caches.
- **Establishing Communication Fast Paths**: One instance of the composition problem in managed runtimes are the overheads introduced by serialization and deserialization of data when it is transferred between different runtimes. A Holistic Runtime could detect the availability of faster communication mechanisms (e.g., RDMA within a rack, shared memory within a node) and allow applications to bypass this step.
- **Co-Scheduling Runtimes**: Holistic Runtimes could help reduce the interference problem by co-scheduling runtime systems on the same node, similar to approaches that have successfully demonstrated such strategies for parallel workloads [94].

3.7 Summary

In this chapter, we introduced and motivated the concept of a Holistic Runtime System. While there is a large design space for these Holistic Runtime Systems, we will now present a specific implementation of such a system that is particularly targeted at coordinating garbage collection across multiple nodes in a cluster.

Chapter 4

The Taurus Holistic Runtime System

This chapter describes the implementation of Taurus, a Holistic Runtime System based on the OpenJDK HotSpot JVM. We first present an overview of the system, following by a detailed description of its design. We then evaluate the system's performance and scalability using microbenchmarks and show that its overheads are negligible.

4.1 System Overview

We now give a high-level overview of our prototype Holistic Runtime System implementation, Taurus. As described in the previous chapter, a Holistic Runtime System is a distributed language runtime system that coordinates a set of language runtimes across a cluster. It aims to be a general solution that enables developers to deploy and experiment with strategies to work around the problems from Section 3.2, while abstracting away potential sources of errors and increasing productivity.

From the perspective of the application that is running across the runtime systems, nothing changes: the application processes are still isolated, and there is no shared heap.

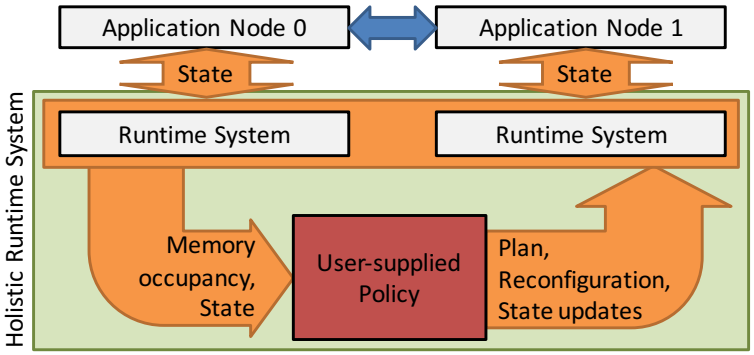


Figure 4.1: The Holistic Runtime System approach.

Traditionally, each runtime system would now make decisions independently, based on its configuration (for example, the Hotspot JVM allows users to configure the GC, generation sizes, tenure rates, and many other settings). In a Holistic Runtime, these decisions are made globally for the entire distributed workload, through a configurable *policy* provided by the application or administrator. We show how policies are defined in Section 4.2.

Coordination is performed by dividing time into epochs of varying length. At the end of each epoch, a *leader* executes the policy. The policy considers the state of all the nodes and produces a *plan* that contains runtime events to be executed during the next epoch, as well as information to be shared between the nodes. This plan is then distributed to the other nodes and executed in a decentralized manner (reminiscent of the approach taken in operating systems such as Tessellation [56] for decentralized scheduling). At the end of the epoch, all nodes report back to the leader with any state updates, so that the policy can execute again and the next epoch can begin (Figure 4.1).

This approach enables a wide range of coordination patterns, while giving a clear abstraction and hiding the challenges of maintaining the distributed system, failure tolerance, time synchronization and interfacing to the runtime system. In Taurus, our prototype, we use these mechanisms to implement and investigate GC coordination. However, the same mechanisms could be used to coordinate JITs, share profiling data, reduce startup times or co-tune applications running on the same node.

4.1.1 Design Requirements for Taurus

One of our primary goals for Taurus is compatibility. While it would be possible to design an entirely new system (and potentially language) from the ground up, it would be challenging to bring up workloads for it: many existing workloads require a fully standard-compliant runtime system, and even close approximations (such as Apache Harmony [12]) do not work reliably. We therefore decided to base our work on the OpenJDK Hotspot JVM, which is the reference implementation of Java and runs the vast majority of software.

Another important goal is usability. For Taurus to be useful, its deployment must be substantially less effort than reimplementing coordination at the application level. We therefore designed Taurus to be a drop-in replacement for the JVM; the goal is that the user only has to install a different `java` executable and everything else behaves the exact same way as before. Behind the scenes, this executable calls into Hotspot and brings up Taurus, which then connects and transparently coordinates the runtime systems.

A third goal is failure isolation. For the system to be adopted in real-world data center settings, it must not substantially increase the probability of failures in any part of the system. For this reason, Taurus has fault tolerance built in and can tolerate node-failures by electing leaders and migrating state using a distributed consensus protocol (Section 4.3.3). This is an important argument for foregoing a model that supports a single-system view (which might reduce failure isolation between nodes).

We are also concerned about failure propagation into the JVM. We therefore avoid direct changes to the Hotspot code base but instead interact with the JVM through its management

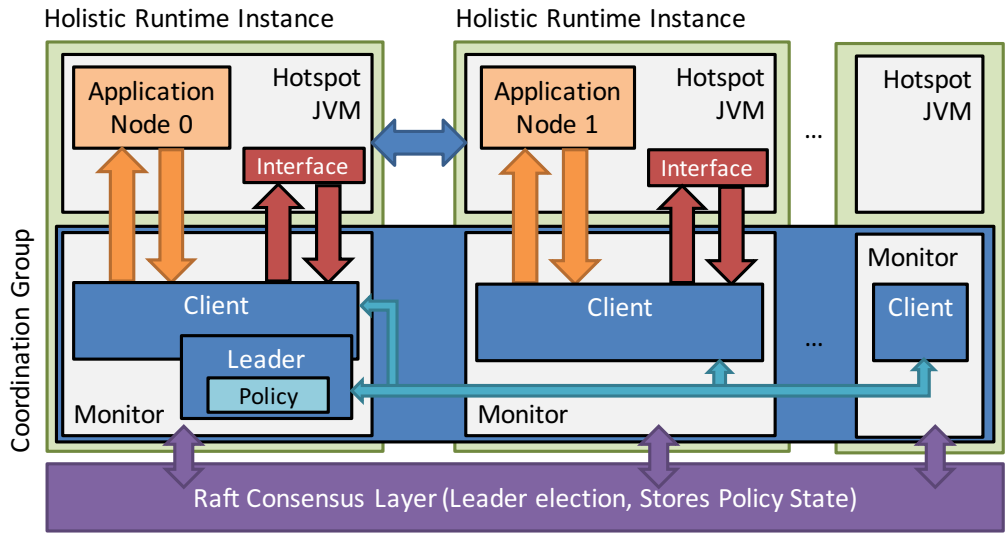


Figure 4.2: The high-level components of Taurus.

interface. Hotspot provides a rich interface to install libraries and performance monitors within the JVM, which can communicate to the outside world. Taurus itself is implemented as a co-process for Hotspot – this ensures that most errors in Taurus stay outside the JVM process barrier, and failures do not bring down the JVM.

4.1.2 Components of the System

Figure 4.2 shows the high-level components of Taurus. With Taurus installed, every JVM instance is augmented with a *monitor* process at startup, to form a *Holistic Runtime Instance*. The monitor connects to the JVM's management interface, which allows it to measure memory occupancy and other internals, and trigger JVM operations such as GC (Section 4.3.1). The monitor also opens a communication channel to the application space of the JVM. This allows the application to exchange information with Taurus (Section 4.1.4). Using this feature is optional and requires modifications to the application.

On startup, the monitor instantiates a *client* thread, which exposes an RPC interface that other nodes can connect to (Section 4.3.2). Monitors also connect to a consensus layer that provides us with a small amount of replicated, consistent storage. This layer is used for features such as leader election or node discovery (Section 4.3.3). Taurus uses Stanford's LogCabin [145] implementation of the Raft consensus protocol [169] – we assume that this layer is available on startup, but it could also be launched automatically.

When launching a Holistic Runtime Instance, the application can select a policy, usually through a new set of special command line flags (we use the `-XX:HVM:flag=value` namespace, which is not used by Hotspot; adding new `-XX` arguments does not break J2SE-compliance).

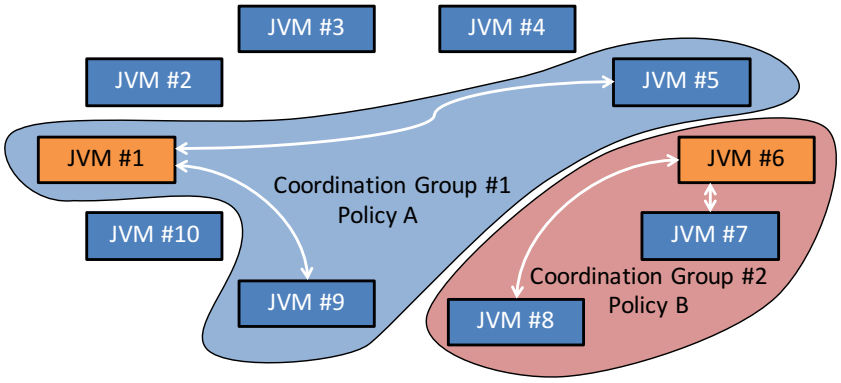


Figure 4.3: *A cluster divided into multiple coordination groups.* Orange nodes are the leaders, which execute the policy and distribute plans to the other nodes in the group.

4.1.3 Policy Execution

Policies are at the core of Taurus: they describe the strategies for coordinating the runtime systems. Taurus's policies are written in a high-level DSL we describe in Section 4.2.

Policy execution in Taurus follows a two-level approach. Using the consensus layer for all coordination in the entire cluster would result in scalability issues, and prevent us from fully exploiting fast rack-level interconnects for applications that only span a subset of nodes within the same rack. We therefore allow multiple independent policies to be active within the same cluster and operate on what we call a *coordination group*.

A coordination group is a subset of runtime systems that is subject to a policy at a given time (Figure 4.3). Every runtime system can be a member of at most one coordination group, and policies can choose to add or remove unclaimed runtimes from their group. Coordination groups operate independently from each other and the consensus layer is only used to handle group membership, leader election and recovery (we assume such changes to be infrequent). This means that a Holistic Runtime System can manage a large set of machines while fine-grained coordination occurs for subsets of nodes, such as those within a rack or those belonging to the same distributed application.

A policy is a pure function that takes as input the state of the coordination group (e.g., memory occupancy of all runtime systems, or user-defined state as described in Section 4.1.4) and produces a plan that contains runtime events for the next epoch and state updates. It also contains coordination group changes (e.g., runtime systems to add or remove).

Each coordination group has an elected leader; all other nodes are followers. The leader establishes a synchronized time base for the group and is responsible for executing the policy once every epoch. When a runtime system instance is launched with a policy selected, it will try to become the leader for this policy by contacting the consensus layer. If there is no leader yet, the node will become the leader, spawn off a *leader* thread, and start executing the policy. Otherwise, it will become a *follower* and connect to the existing leader.

By using this approach, we address the scalability issues since no global consensus is necessary within the coordination group; the only scalability bottleneck is the leader, which needs to receive one sample from each member of the coordination group per epoch and distribute the plan (in Section 4.4.3, we show that this scales well to at least 180 nodes). Furthermore, failures in the coordination group are "softer" than failures in the global consensus layer. While they may lead to performance degradation, missing a small number of epochs will not cause a failure of the application running on Taurus, as the state can be corrected throughout later epochs (work in cluster scheduling shows that such an *optimistic* approach can work well in a distributed setting [198]). The downside is that since we are not relying on the consensus layer within coordination groups, we need to be able to recover from both leader and follower failures (Section 4.3.4).

4.1.4 Communication with the Application

In addition to coordination at the runtime system level, Taurus can optionally communicate with the application itself. This is useful to implement policies such as the *Steering* policy from Section 3.3.2, where the runtime system needs to communicate to the application which nodes to avoid. The communication abstraction we chose is a shared set of key-value pairs visible to both the runtime system and the application space.

For this purpose, each monitor installs a globally visible class in the application space that exposes two thread-safe static methods to the application, `HVM.getKeyValuePair(k)` and `HVM.setKeyValuePair(k,v)`. To use this interface within an existing application, it suffices to add a `.jar` file to the classpath and recompile the application.

When running, the policy receives as part of its input the full set of key-value pairs on all nodes of the coordination group, and can produce and distribute (as part of its plan) updates to key-value pairs on any node. This allows policies to implement a wide range of communication patterns with the application.

4.1.5 Reconfiguration

When launching a new policy, its coordination group only contains one node: the leader of the group. The policy can query the list of available *runtimes* in the cluster, and whether they are a member of any other group. They can then select unassigned nodes to add to their coordination group as part of the plan. These nodes join during the next epoch, and are considered during the next execution of the policy (conflicts are avoided by making policy executions atomic with respect to one another). Node terminations or failures are visible in a similar way: the policy can see that a node has timed out and can remove it from its coordination group (Section 4.3.4).

4.2 Policy Description Language

To facilitate the development of policies (and therefore adoption), we designed a Domain-Specific Language (DSL) for policy descriptions (Figure 4.5). Policies are high-level imperative programs that are atomically executed by the leader at the beginning of each epoch. Accessing different data structures, policies gain insights into the state of all runtime systems belonging to the coordination group, and assemble a *plan* that is distributed to all nodes in the group and contains actions to perform, as well as the duration of the epoch.

Figure 4.4 describes the grammar of the policy language. Each policy starts with a `policy` declaration, which gives the policy a unique name. It contains a `run` block that describes the policy function and defines policy parameters. This block contains a sequential program that can access any state visible to the system and builds up a plan. Instead of a fully Turing-complete language, we do not provide general for or while loops, but only `foreach` loops over finite sets. This ensures that policies *always terminate*, which prevents the leader getting stuck. This is a key advantage of using a DSL.

In addition to the primitive types `double`, `int` and `string` and two parameterized collection types `Set` and `Map`, the language provides two composite types to describe runtime systems (two types are required to distinguish between runtime systems that are under the control of the policy and those registered but not under the policy's control):

- `Runtime` (Table 4.1): This type describes a runtime system that is part of the Holistic Runtime but not necessarily the current coordination group. This type is used for managing group membership (e.g., adding/removing runtimes). The type contains fields such as the runtime's server address, command line, a collection of tags set at startup, and the current coordination group membership.

- `Member` (Table 4.2): This type describes a member of the policy's coordination group. This type is used to directly interact with the runtime system. It contains fields for memory occupancy of the different memory spaces, GC-related statistics, whether the node is unresponsive (`busy`) and the set of key-value pairs.

Note that the `Members` are a subset of the `Runtimes`: Given a `Member m`, the corresponding `Runtime` can be accessed with `m.runtime`. The sets of all runtimes and all members can be accessed using global keywords `runtimes` and `members`. Note that the `runtimes` set can become large, which is why monitors cache it instead of updating it from the consensus layer at every epoch (comparing only a version number to check for updates).

The language allows filtering sets with a predicate. An example can be found in Figure 4.5b. In this case, a filter predicate is used to determine the set of members with a certain memory occupancy, and check whether it is empty.

We provide a special construct of the form `plan <- Action(...)` to add commands to the policy's plan constructed during the current epoch. A plan is effectively the policy's return value and contains a set of commands to execute on each runtime system. Some

⟨*program*⟩ ::= '`policy`' ⟨*ident*⟩ '`{`' ⟨*policy*⟩ '`}`'

⟨*policy*⟩ ::= ⟨*decl*⟩ '`run(`' ⟨*args*⟩ '`) {`' ⟨*stmt*⟩ '`}`'

⟨*decl*⟩ ::= ⟨*decl*⟩ '`;`' ⟨*decl*⟩
| ⟨*empty*⟩
| '`import`' ⟨*ident*⟩ '`(`' ⟨*args*⟩ '`)`'
| '`state`' ⟨*type*⟩ ⟨*ident*⟩ = ⟨*expr*⟩

⟨*stmt*⟩ ::= ⟨*stmt*⟩ '`;`' ⟨*stmt*⟩
| '`if (`' ⟨*pred*⟩ '`) {`' ⟨*stmt*⟩ '`}`'
| '`foreach (`' ⟨*type*⟩ ⟨*ident*⟩ : ⟨*expr*⟩ '`) {`' ⟨*stmt*⟩ '`}`'
| '`plan <-`' ⟨*plan-action*⟩
| ⟨*ident*⟩ '`(`' ⟨*args*⟩ '`)`'
| ⟨*cstmt*⟩

⟨*plan-action*⟩ ::= '`ReconfigureAddMember(`' ⟨*expr*⟩ '`)`'
| '`MajorGC(`' ⟨*expr*⟩ '`)`'
| '`MinorGC(`' ⟨*expr*⟩ '`)`'
| '`EpochLength(`' ⟨*expr*⟩ '`)`'
| '`SetKV(`' ⟨*ident*⟩ '`,`' ⟨*string*⟩ '`,`' ⟨*expr*⟩ '`)`'

⟨*expr*⟩ ::= '`name.filter([`' ⟨*type*⟩ ⟨*ident*⟩ '`:`' ⟨*pred*⟩ '`])`'
| ⟨*cexpr*⟩

⟨*type*⟩ ::= '`int`' | '`double`' | '`string`'
| '`Member`'
| '`Runtime`'
| '`Set<`' ⟨*type*⟩ '`>`'
| '`Map<`' ⟨*type*⟩ '`,`' ⟨*type*⟩ '`>`'

⟨*args*⟩ ::= ⟨*arglist*⟩
| ⟨*empty*⟩

⟨*arglist*⟩ ::= ⟨*arglist*⟩ '`,`' ⟨*arglist*⟩
| ⟨*ident*⟩ '`=`' ⟨*expr*⟩

Figure 4.4: Grammar describing the policy description language. ⟨*cexpr*⟩ and ⟨*cstmt*⟩ describe statements in the C language with any variables and arguments declared in the policy added to their scope. ⟨*pred*⟩ is an ⟨*expr*⟩ that evaluates to true or false. ⟨*ident*⟩ describes identifiers such as variable names.

Field	Description
`id`	A unique integer ID that is assigned to the runtime system by Taurus.
`tag`	A string that is initialized to a value that can be passed to the Java executable using a non-standard `-XX:HVM:tag=val` argument. This can be used to identify the runtime systems belonging to a particular application.
`cmd`	A string that contains the command line arguments the JVM was called with.
`status`	An integer ID belonging to the coordination group that this runtime is part of, or `UNASSIGNED` if it is not part of any coordination group.

Table 4.1: Fields of the `Runtime` composite type.

Field	Description
`uptime`	Time elapsed since the runtime system was launched (in s), as a floating point number.
`busy`	A boolean that is true iff the leader did not receive a sample from this runtime system during the last epoch. This enables the policy to detect and handle failures.
`kv`	A field to query the set of key-value pairs associated with this member. To access a key-value pair, the policy can call `m.kv(name)`. The resulting key-value pair's value can be converted to other types using methods `toString()`, `toInt(default)` and `toDouble(default)`, where the default is used if the conversion fails.
`runtime`	The `Runtime` instance belonging to this member.
`memory`	Memory occupancy of the different memory spaces in the JVM. Fields include `eden`, `old`, `perm`, `s0`, `s1`. Values are floating point numbers ranging from 0.0 to 100.0.
`gc`	Statistics about the number of full and young-generation, and the time spent in them. Fields are floating point and include `young.count`, `young.time`, `full.count`, `full.time`.

Table 4.2: Fields of the `Member` composite type.

commands take parameters, such as a subset of members or runtimes they apply to (e.g., adding a set of runtimes, performing GC on a set of members). By repeatedly using this construct, the policy builds the plan (an instance of the *Builder* pattern).

A member's key-value pairs are accessed through a `kv` field in `Member`, and updates to them are added to the plan by adding a `SetKV` command (Figure 4.6). Key-value pairs are stored as strings and it is often necessary to convert values to or from primitive types such as integers. To prevent error conditions in the case of malformed strings, all such conversions need to be provided a default value that is used in case the conversion fails.

Finally, policies support state that is kept around between epochs using the `state` keyword (Figure 4.6). State is atomically updated during policy execution and in the absence of failures, status is maintained between epochs. Note that all state variables also need a default value that is used in case of errors (e.g., if a key is not found in a Map, or on failure).

```
policy AutoAdd {
  run(Set<Runtime> s = runtimes) {
    foreach(Runtime r : s) {
      if (r.status == UNASSIGNED && r.tag == "gc") {
        plan <- ReconfigureAddMember(r);
      }
    }
  }
}
```

(a) Policy that automatically adds all unassigned runtimes to the coordination group that were launched with the `-XX:HVM:tag=gc` command line argument. This operation is cheap and does not introduce scalability challenges (Section 4.4.3).

```
policy STU {
  extern double cutoff = 10.0;

  run(Set<Member> stu = members,
      Set<Member> collect = members) {
    if (!(stu.filter([Member m : (! m.busy) && m.memory.old > cutoff]).empty())) {
      plan <- MajorGC(collect);
    }
  }
}
```

(b) *Stop-the-Universe* Policy that performs a full GC for all members in `collect` if at least one of `stu` has reached a memory occupancy of `cutoff`.

```
policy Example {
  import STU(cutoff=90.0);
  import AutoAdd();

  run() {
    STU(members.filter([Member m: m.runtime.tag == "gc"]), members);
    AutoAdd(runtimes);
    plan <- EpochLength(200.0); //ms
  }
}
```

(c) Composing policies: `Example` imports and calls into the `STU` policy from above (with two dynamic parameters), followed by a call into `AutoAdd`. The STU policy is only applied to runtimes that were launched with the `-XX:HVM:tag=gc` command line argument.

Figure 4.5: Examples of policies written in Taurus's DSL.

```
policy PingPong {
  state Map<Member,int> prev = 0;

  run() {
    foreach (Member m : members) {
      int pp = m.kv("pingpong").toInt(0);
      if (prev.get(m) != pp) {
        plan <- SetKV(m, "pingpong", pp+1);
        prev.set(m, pp+1);
      }
    }
    plan <- EpochLength(200.0);
  }}
```

Figure 4.6: Example policy using key-value pairs and policy state. The policy monitors a key-value pair and increases it when it sees a change; the application does the same.

4.2.1 Configurability and Composability

We hypothesize that most workloads will require variations of a small set of basic policies, potentially with some application-specific extensions (Section 5.4). Composability is therefore an important feature in our Policy DSL: it allows us to build a repository of basic policies over time, and combine them into application-specific solutions. To achieve this flexibility, policies need to be configurable and composable.

We allow policies to be included into other policies through an `import` statement. This will include the policy and allows it to be called within the `run` block. Figure 4.5c shows an example of this. Policies are parametrizable with two types of parameters: dynamic and static parameters. Static parameters are defined at the time of import and do not change at runtime (these are the parameters defined as `extern` outside the run block). Dynamic parameters are defined with the `run` function and passed to the policy whenever it is called from within another policy. Figure 4.5b shows examples for both types of parameters: `cutoff` is static, while `stu` is dynamic. All parameters can have default values.

4.2.2 Policy Compilation

Our DSL is embedded into C++11, and we reuse many C++ features including numerical and logical operations, if statements and stream operators. A recursive-descent parser written in Python transforms our policy code into C++ code, which is then compiled into a dynamic library. The parser does not split the code down to individual tokens but only into pieces that can be directly transformed into C++ code (e.g., the predicate within a filter). We then find and replace any DSL-specific keywords and idioms with their C++ equivalents (while taking into account scopes, string delimiters, etc.). Policies are transformed into classes, foreach

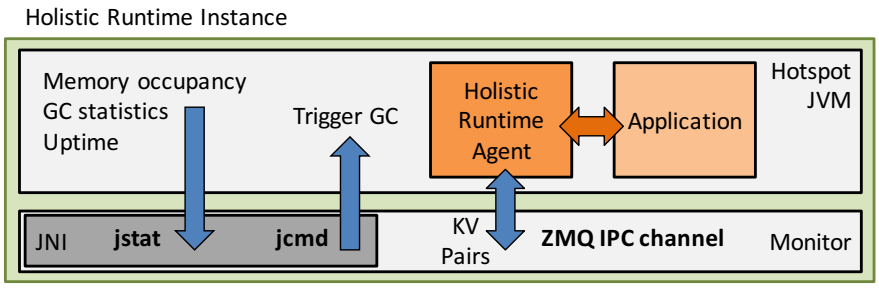


Figure 4.7: *Interface between Hotspot and the monitor.* ZMQ (also called ∅MQ or *ZeroMQ* [103]) is a popular communication framework.

loops into iterators and filter predicates into C++11 lambda expressions. We use C++11 move semantics to chain filters without copying data.

4.3 Implementation

After the high-level design, we will now present implementation details of Taurus. Taurus is entirely written in C++11, to avoid GC-induced pauses in the Holistic Runtime itself.

4.3.1 JVM Interface

The monitor connects to Hotspot through three different interfaces (Figure 4.7). It queries memory occupancy information through the Hotspot JVM's `jstat` interface, which exposes the JVM's performance counters by writing them to a shared page that can be mapped by a different process (by default, Hotspot updates these counters every 50 ms; as we require a finer granularity, we set the interval to 1 ms instead). Using this mechanism, the monitor can access this data directly without blocking on the JVM.

Commands in the JVM (primarily triggering of Major GC) are performed through the `jcmd` interface, which allows calling into the JVM to trigger activities (it may stall if the JVM is unresponsive, such as during a GC pause).

Finally, key-value pairs are exposed to the Java application through a *Java agent* that is installed into the JVM's application space at start-up. This agent connects to the monitor through an IPC mechanism provided by the ZMQ library [103], which is the same library we use for inter-node communication. The agent then updates key-value pairs in the `HVM` class (Section 4.1.4), which is accessible from the application. The agent is also responsible for triggering minor GCs: since Hotspot does not support this, the agent can force a minor GC by allocating unreachable objects until the young generation is full (using the `MemoryPoolMXBean` interface to determine how much memory it needs to fill).

4.3.2 Inter-Node Communication

Inter-node communication is implemented using a simple RPC protocol. We use ZMQ [103] for communication, since it gives us a higher level of abstraction than regular sockets (e.g., managing concurrency and high-level communication patterns), and supports low-latency communication over Infiniband. Our RPC protocol is using Protocol Buffers [218].

To add a node to the coordination group, the leader (prompted by its policy) sends a reconfigure request to the node's client, which will then send a request to join the leader's coordination group. The leader then confirms the join request and sends the plan of the currently active epoch to the client, after which the node is part of the coordination group. Leader and client also repeatedly exchange timestamps through a separate connection, to determine the drift between them (no special hardware is needed for this). From then on, all timestamps are expressed relative to the leaders's clock.

4.3.3 Consensus Layer

We use LogCabin [145, 169], Stanford's Raft implementation, as our consensus layer. We run three LogCabin instances by default, and monitors can connect to any of them. LogCabin provides a small amount of consistent, highly replicated storage. We use this storage to track the set of instances currently registered with Taurus, as well as active policies. The set of instances has a version number; for performance reasons, nodes cache the instances locally and only compare against the version number once per epoch.

When launching a new JVM, its monitor will connect to LogCabin and create an entry for its runtime system instance. This includes information such as its command line options and address/port (the monitor selects a free port automatically on startup). Next, it will try to launch a policy (if requested through a command line option), and become the leader for this policy. When the policy runs, it can detect other instances in the cluster and add them to its coordination group, using a policy such as that in Figure 4.5a. To help policies distinguish between multiple distributed applications, the `-XX:HVM:tag` command line flag is used to assign a unique application name to all JVMs belonging to one application; the first node with this name will become the leader and runs the policy, which adds the other nodes.

4.3.4 Execution & Failure Handling

Once a leader has been selected and has started running the policy, it enters a loop that consists of three stages, visualized in Figure 4.8:

1. Distribute the current plan to all nodes in the coordination group (at the beginning, the plan is empty). This plan includes the length of the next epoch.

2. Followers execute the instructions in the plan, wait until the end of the epoch, atomically take a sample of their state, and send it to the leader. If they cannot take a sample in

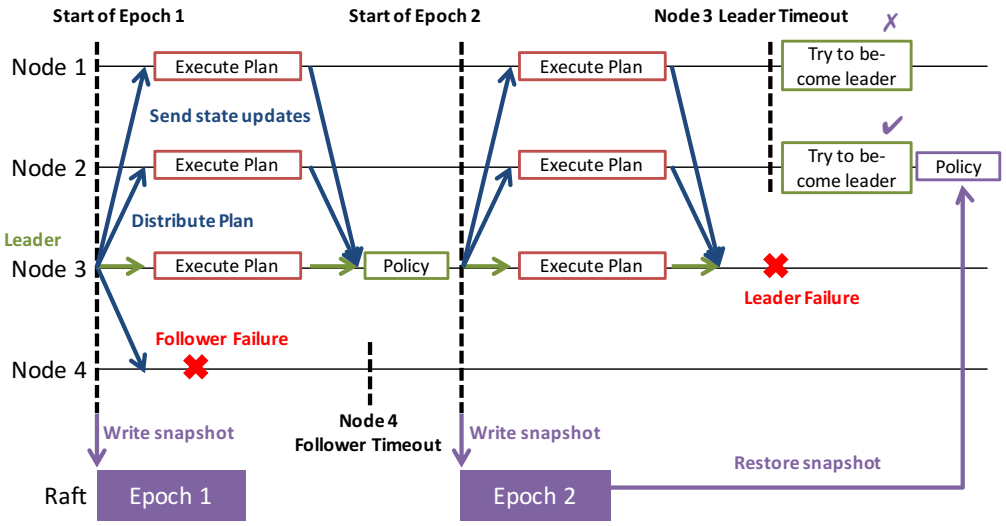


Figure 4.8: *A sample policy execution in Taurus.* When a follower fails, the policy sees it as unavailable. When the leader fails, another node becomes leader, restores the policy from the last executed epoch, and continues (marking all other nodes as unavailable for the next epoch, before stabilizing).

time, they will send a response that they are "busy" (which can happen if the JVM is in a GC pause or if there is contention in the system).

3. The leader collects the state updates from all followers. Once the epoch has ended, it marks all nodes from which it has not received an update as "unavailable", executes the policy, produces a new plan, and sends it to all nodes (including those that are unavailable; failures are discussed below).

The length of the epoch is set by the plan itself and can adapt to the circumstances (e.g., if the workload is exhibiting irregular allocation rates, the policy can decide to decrease the epoch to react more quickly to changes). However, when dynamically adjusting the epoch length, policy authors have to be careful about control loops.

Leader failures are tolerated by storing the policy meta-data (including the most recent plan) to the consensus layer every epoch (i.e., the epoch is made persistent before sending out a plan). All followers have a timeout by which they expect the next plan to arrive. If no plan arrives, they assume that the leader has failed and will attempt to become leader themselves by trying to write the entry of the current epoch to the consensus layer. If it has been written before, it means that the original leader is either still alive and the message was delayed, or that some other node has become leader in its place. In either case, the node will continue as a follower. If the node succeeds in writing the policy entry, it becomes the next leader, pulls the policy data from the consensus layer, and starts executing.

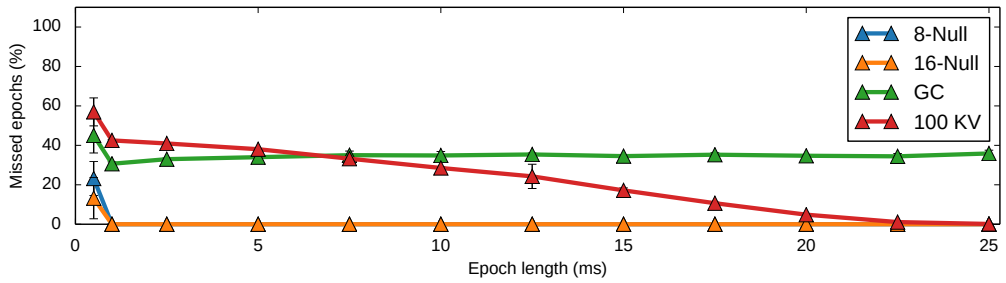


Figure 4.9: *Missed epochs depending on the epoch length.* Varying the epoch length demonstrates the coordination granularity supported by Taurus. "8-Null" and "16-Null" are policies that perform no operation; this shows that an epoch length of 1–2 ms can be supported on a 16-node cluster. "GC" is a policy that regularly triggers GC, confirming that a constant fraction of epochs is missed in this case. "100 KV" is a policy that sets 100 key-value pairs on each node during each epoch, showing that a large number of key-value pairs can increase the minimum coordination granularity to 20 ms.

This approach moves all failure handling into the consensus layer, making it simpler to reason about fault tolerance. At the same time, it puts little load on the consensus layer in the absence of failures (requiring only the leader to access it, and only once per epoch). This maintains the advantages of the two-level approach while tolerating failures.

4.4 Evaluation

We now characterize Taurus's coordination performance, scalability and overheads through a series of microbenchmarks. Most of our evaluation was performed on a 16-node cluster connected with 40 GbE using Mellanox dual port MCX314A-BCBT cards. Each node has an Intel IvyBridge E5-1680V2 3.0GHz CPU with 8 cores (16 hardware threads) and 64 GB RAM. In addition to our workloads, the cluster ran YARN, as well as HDFS and Tachyon file systems. To demonstrate the generality and scalability of Taurus, we also run a scalability microbenchmark on a set of 200 `g1-small` instances on Google Compute Engine.

All nodes in our cluster are running Linux 3.13.0. We run a snapshot of LogCabin from 4/10/15 with the `Segmented` storage module on a RAM disk (as we do not require persistence across machine failures). We use OpenJDK 1.7.0_75, ZeroMQ 4.0.5 and Google Protocol Buffers 2.6.1. Unless noted otherwise, we used the default settings for all applications.

4.4.1 Coordination Granularity

We are first interested in the granularity of coordination that Taurus enables, specifically the minimum sustainable epoch length. We ran Taurus on a Java program that sleeps for one second at a time in an infinite loop, and ran it for different policies and epoch lengths

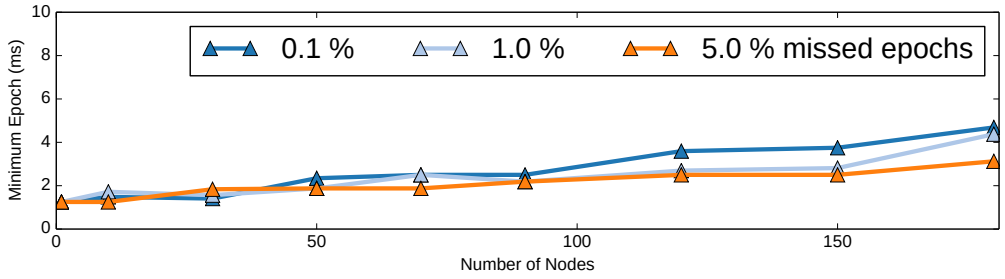


Figure 4.10: *Scaling to 180 Google Compute Engine nodes.* This experiments fixes the fraction of missed epochs and adjusts the epoch length to achieve this target (in large deployments, there are always some missed epochs). This shows that the minimum coordination granularity increases for larger numbers of nodes, and that a coordination granularity of 10 ms is achievable, even for large numbers of nodes and few missed epochs.

(Figure 4.9). We report how often clients fail to report back by the end of the epoch, indicating that the epoch was too short. In all cases, we let the system reach a stable state before performing our measurements. Error bars here and later are the σ of 5 runs.

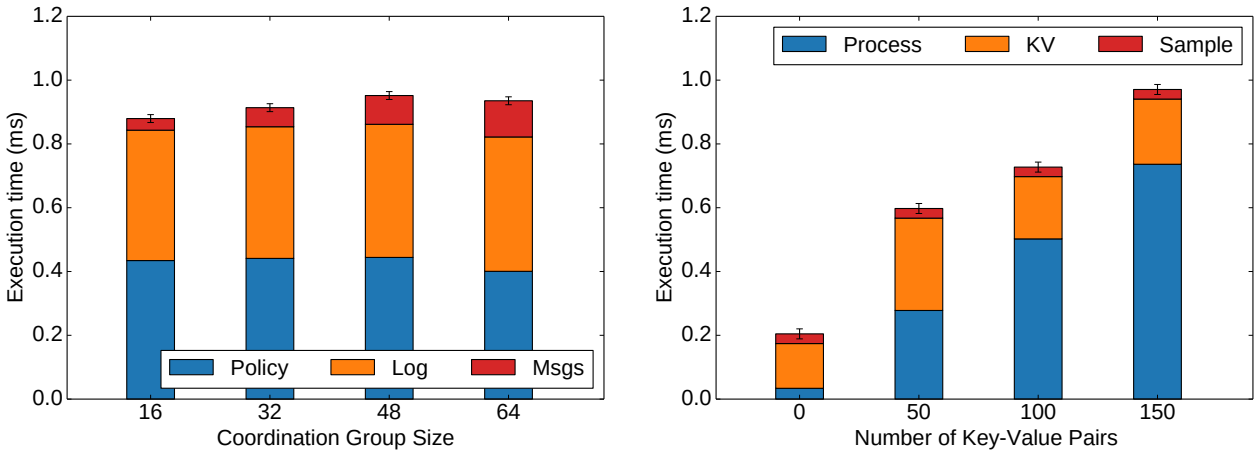
8-Null and 16-Null are policies that do nothing and only check for new runtimes (on 8 and 16 nodes). In both cases, no epochs were missed until reducing the epoch to below 2 ms, indicating the minimum sustainable epoch length is 1–2 ms (for the scale we looked at). Since the minimum `jstat` sampling rate of the Hotspot JVM is 1 ms, this is sufficient.

We also experimented with two other policies: `KV` sets 100 key-value pairs every epoch on each node – this stresses communication and increases the minimum epoch length to 20 ms. We also ran a policy that triggers a full GC every 10 s (`GC`), on a workload that constantly allocates data. As expected, this misses a constant fraction of epochs due to GC pauses.

4.4.2 Scalability

To demonstrate that Taurus scales to a large number of nodes in a more realistic data center deployment, we ran a similar microbenchmark on a set of up to 180 Google Compute Engine `g1-small` instances. We used a policy that constantly measures the number of missed epochs and adjusts the epoch length – similar to a binary search – until a target fraction of missed epochs is reached (note that in a large deployment, there are always some missed epochs).

Figure 4.10 shows that even in such a deployment without Infiniband, Taurus performs well and can achieve epoch times below 10 ms. With ping latencies of around 300 μs (and LogCabin running on a separate set of nodes), we believe these results to be reasonable. As we will see in Section 5.3, a 10 ms epoch is sufficient even for fine-grained coordination in latency-sensitive systems. In addition to helping us determine the minimum epoch length, this experiment also shows an example of a policy that automatically adjusts the epoch length (a good strategy to make policies more portable).



(a) Leader Execution **(b)** Follower Execution

Figure 4.11: *Impact and scalability of the different components of policy execution.* The leader execution time is dominated by the time to access the consensus layer and executing the policy itself, while communication cost is low. Followers spend most time processing commands and key-value pairs, while the time to sample the JVM is negligible.

4.4.3 Performance Breakdown

We are now interested in how much the different components of the execution contribute to the epoch length. For the leader, we ran a policy that reads out all nodes' GC statistics while varying the coordination group size (with four JVMs per node). For the followers, we chose the KV policy from Section 4.4.1, varying the number of key-value pairs. igure 4.11 shows the different contributors: For the leader, this consists of the policy execution (*Policy*), writing to the consensus layer (*Log*) and sending/receiving messages (*Msgs*). For followers, this consists of serializing and deserializing messages (*Process*), updating the key-value pairs (*KV*) and collecting the sample from the Hotspot JVM (*Sample*).

Understanding the contribution of these different components is important, since the leader's execution time limits the coordination granularity as much as the epoch length. The bottleneck appears to be the access to LogCabin: snapshotting the epoch takes about 400us on average (with a LogCabin instance on the leader's node). The policy takes a similar amount, as it needs to perform one access to the LogCabin cluster as well, in order to check version numbers. Sending out messages is a small fraction of the execution time, and scales linearly with the coordination group size.

For followers, the execution time is currently dominated by receiving, decoding and applying the instructions of the plan (which grows with the number of key-value pairs). It is likely that a more efficient format could reduce these overheads. Neither sampling the JVM nor communicating key-value pairs to the JVM appear to be a substantial bottleneck. Note that this example uses an unusually large number of key-value pairs and execution times in practice are likely going to be lower.

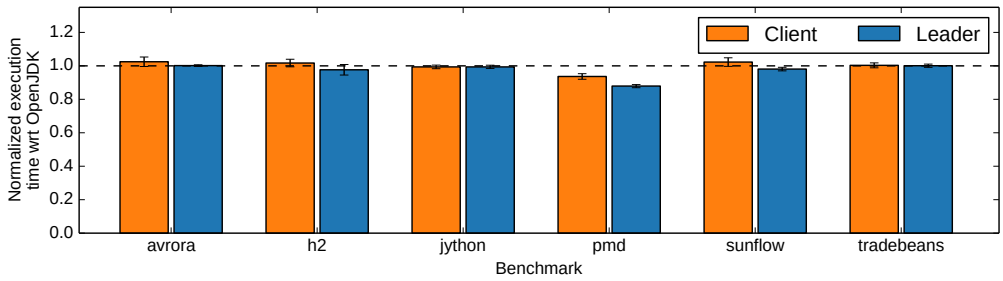


Figure 4.12: *Overhead of running the DaCapo benchmarks with Taurus*, compared to OpenJDK's Hotspot JVM without Taurus. Taurus introduces negligible overheads. Speed-ups relative to the baseline are likely due to Hotspot's sensitivity to the environment.

4.4.4 Overhead

To determine the overhead of Taurus, we used the DaCapo benchmarks [37] (we chose the subset of benchmarks that worked correctly with OpenJDK 7: `avrora`, `h2`, `jython`, `pmd`, `sunflow`, and `tradebeans`). We compare the performance degradation both for followers and leaders. Taurus does not introduce substantial overheads, at most 3.0% in our measurements (Figure 4.12). Given that JVMs are known to be sensitive to changes in the environment and that we introduce many such changes (e.g., adding an agent, using management interfaces, changing the sampling interval), we believe most of the differences in performance to be noise (including the speed-up for pmd).

4.5 Summary

We presented the design of Taurus, a prototype Holistic Runtime System to coordinate distributed applications in data centers. Taurus is a JVM drop-in replacement, runs unmodified real-world applications, requires no modifications to the underlying runtime system and provides a simple DSL to implement policies. Our goal is to enable developers to implement their own policies and use Taurus as a research vehicle for exploring coordination strategies.

So far, we have demonstrated how Taurus can be used to coordinate garbage collection between different nodes in a cluster. However, we believe that Taurus can be used beyond GC. Specifically, it could be used for distributed monitoring and profiling, coordinating code generation in JIT compilers, and reducing interference between JVMs. It would also be possible to integrate Taurus with other runtime systems than the JVM, to coordinate workloads across different programming languages (such as PHP or Python).

Chapter 5

Using Taurus for GC Coordination

This chapter describes how Taurus can be used, by applying it to the problem of coordinating garbage collection pauses in data center applications. We first demonstrate these techniques for the two examples from Chapter 3. We then present a general classification of GC coordination techniques and how they apply to a range of different data center workloads.

5.1 Running Applications with Taurus

As described previously, running workloads under Taurus requires relatively few changes. Depending on the workloads, this includes the development of a workload-specific Taurus policy, as well as modifications to the application itself. In many cases, the policy can be built more easily by combining existing policies (Figure 4.5).

To run Taurus, we first need to extend the `PATH` environment variable to add Taurus's directory. This allows the system to pick up Taurus's utility programs and replaces the default `java` executable with Taurus's version. We then need to run a `taurus` command to launch and initialize the consensus layer. This command accepts a list of addresses of nodes to run instances of the consensus layer on (we use three nodes for all of our experiments).

Finally, policies are written into a file and compiled by calling the `compile-policy` program. This produces a dynamic library, which needs to be placed into the same directory

```
export PATH=/home/user/bin/taurus:$PATH
taurus start --nodes=ip0,ip1,ip2
compile-policy MyPolicy.policy
java -XX:HVM:policy=MyPolicy MainProgram
HVM_POLICY=MyPolicy ./run_main_program.sh
```

Figure 5.1: Commands to run an application using Taurus.

on all nodes (typically Taurus's setup directory). This directory can be adjusted through the `-XX:HVM:policy_searchpath` command line option.

Once these steps have been completed, the actual application can be launched. Existing run scripts can be used as before, but we now need to tell the executable to use our new policy. This can either be done through the `-XX:HVM:policy` command line option when calling into `java`, or by setting an `HVM_POLICY` environment variable. The latter is advantageous if an application requires complex run scripts, since it does not require any changes to those scripts. Throughout the rest of this chapter, we rely on the latter approach, using run scripts from the BITS benchmark suite[1].

From a user perspective, no further changes are needed. Taurus will launch transparently in the background, connect all runtime systems, elect a leader and start running the policy. We found that for debugging purposes, it can be useful to have the policy output additional information, which will be stored in a file. Figure 5.3 will show an example of this.

We will now demonstrate Taurus by applying this approach to the two real-world workloads from Section 3.3. These workloads are representative for both batch workloads and interactive workloads, and demonstrate instances of a range of general coordination strategies.

5.2 Apache Spark (Batch Workload)

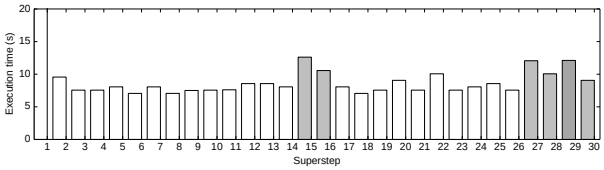
As an example for a batch workload, we used the same Spark PageRank computation as in Section 3.3.1. This workload is representative of iterative computations, such as other graph algorithms or repeated gradient descent. We use the implementation of the PageRank algorithm that ships with Spark 1.1.1 (`org.apache.spark.examples.bagel.WikipediaPageRank`) and run it on the 54 GB *Wikipedia* graph from the original Spark paper [239].

We recall that the computation is divided into multiple iterations (i.e., PageRank supersteps). At the end of each iteration, the nodes need to exchange intermediate results, which effectively acts as a global barrier. When one node stops for a garbage collection pause, its iteration takes longer and it will therefore delay the other nodes who will wait for it at the barrier. Figure 5.2a shows this effect running on our 16-node cluster: PageRank supersteps take a similar amount of time in the absence of GC, but as soon as one node stops for a GC pause, the superstep as a whole takes longer.

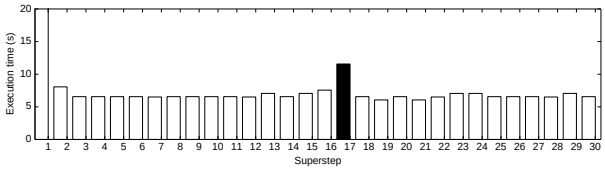
As discussed in Section 3.3.1, this problem can be addressed using a Stop-the-Universe (STU) policy. Instead of performing GC whenever a node runs out of memory, we perform collections on all nodes at the same time. Implementing such a strategy in Taurus is simple. It requires no modification of Spark and a simple policy shown in Figure 5.3.

The effect of using this STU policy for the Spark PageRank workload is shown in Figure 5.2b. We used Spark 1.1.1 on 16 nodes with 32 GB heaps. No modification to Spark was required, and even this simple policy reduced execution time by 21%.

[1]BITS stands for *Berkeley Interactive and Throughput Suite* and was developed to support several projects at UC Berkeley, including Taurus. The source code is available at `https://github.com/ucb-bar/bits`.



(a) Baseline System (no coordination)



(b) Coordinating GC (Stop-the-Universe)

Figure 5.2: *Relation between GC and the superstep durations of Spark PageRank* (shade represents the number of nodes performing GC during a superstep; white = no GC). As soon as one node performs GC, the superstep takes much longer. With the STU policy, all nodes perform garbage collection at the same time and therefore only incur this delay once.

```
policy SparkPageRankSTU() {
  extern double threshold = 90.0;
  import AutoAdd();

  run() {
    // The main Stop-The-Universe (STU) logic: detect whether there are
    // nodes whose old generation is almost full. Ignore busy nodes, as
    // "busy" means no sample is available for the last epoch.
    foreach (!members.filter([Member m : m.memory.old > threshold &&
        (! m.busy)]).empty()) {
      plan <- MajorGC(members);
          out << "Trigger GC" << endl; // Debug output
    }

    // We only need to coordinate the SparkSubmit application (which runs
    // the main algorithm) and the executors, which perform the main work
    AutoAdd(runtimes.filter([Runtime r :
      contains(r.cmd, "spark.executor") ||
      contains(r.cmd, "deploy.SparkSubmit")]));

    // Epochs have a constant length of 500ms
    plan <- EpochLength(500.0);
  }
}
```

Figure 5.3: *Stop-the-Universe Policy for Spark.* The policy coordinates the main processes that belong to Apache Spark, detects whether any of them has an old generation that is more than 90% full, and triggers a major collection on all of these nodes if this is the case. The `AutoAdd` policy was introduced in Figure 4.5a and automatically adds runtimes to the coordination group when they are discovered.

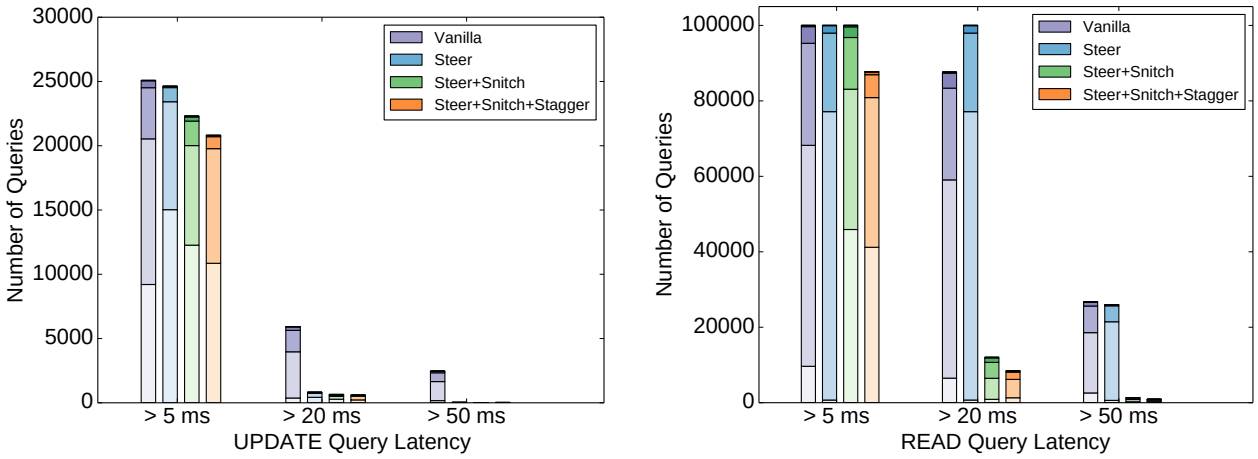


Figure 5.4: *Cumulative query latency distributions of Cassandra running the YCSB benchmark.* We report the slowest 100K queries (99.9 percentile) of a representative run. Stacked bars represent the number of GCs during a request. Each experiment represents a different coordination strategy that reduces the number of stragglers: *Vanilla* uses no coordination. *Steer* uses a strategy that instructs the load balancer to send requests to nodes that are unlikely to enter GC. *Snitch* applies the same strategy for selecting replicas within Cassandra. *Stagger* schedules garbage collection pauses across Cassandra nodes.

5.3 Apache Cassandra (Interactive Workload)

As an example for an interactive workload, we used Taurus to improve Cassandra tail latencies (Section 3.3.2). We run a YCSB snapshot from 3/11/15 against Cassandra 1.0.6 with a replication factor of 3 and a cluster size of 8. We run workload A on a keyspace with 10M entries for 100M queries (50s warmup). We chose a 32GB heap with a 4GB young generation.

Running this experiment, we observed that average query latencies were very short, 173.6 us for updates and 522.8 us for reads. However, some requests took over 100× longer than this. Figure 5.4 shows the cumulative latency distribution of the slowest 100K queries (> 5 ms latency), which is the tail from the 99.9 percentile. We show that Taurus can eliminate most of these stragglers (in particular, queries with latencies of over 20 ms).

Cassandra requires more sophisticated coordination strategies than the Spark example from the previous section. Specifically, we had to implement several different GC coordination policies, and introduce minor modifications to the application itself. We will now present the strategies in turn, and discuss their effectiveness for improving query latencies.

We assume a setup similar to that introduced in Section 3.3.2: A cluster of 8 Cassandra nodes is deployed to separate machines, and a load balancer on a different machine evenly distributes requests to these nodes. We use the YCSB workload generator in place of the load balancer, simulating a set of clients connecting to the Cassandra cluster.

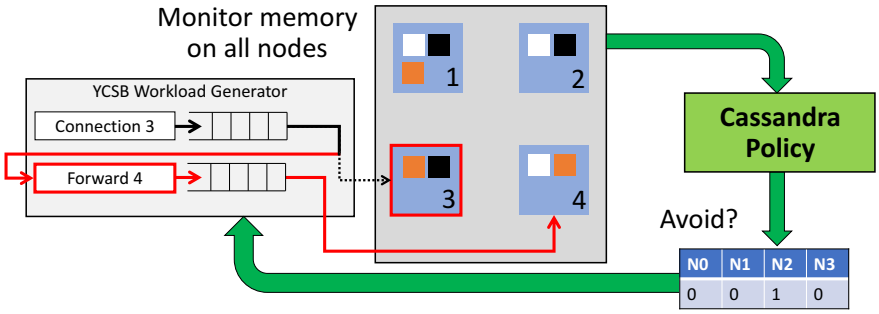


Figure 5.5: *Cassandra Request Steering.* The policy monitors the young generation on all Cassandra nodes and communicates one key-value pair per node to the load balancer (i.e., the workload generator in our case). The key-value pair has a value of 1 if the node is close to a GC pause, and 0 otherwise. The load balancer then checks these key-value pairs before dispatching a request. If the key-value pair is 1, the request is forwarded to a random node with a non-zero key-value pair. Forwarding works by creating a second *forwarding* connection thread per Cassandra node – whenever a connection is about to send a request to a node that is close to GC, it will instead enqueue this request for a random forwarding thread to process.

```
policy CassandraSteer() {
  import AutoAdd();

  run {
    AutoAdd(runtimes);

    // Identify the node running the YCSB workload generator/load balancer
    Set<Member> ycsb = members.filter([Member m : m.runtime.tag == "ycsb"]);

    if (!ycsb.empty()) {
      foreach (Member m : members) {
        if (m.memory.eden > 80.0 || m.busy || m.memory.eden < 5.0) {
          plan <- SetKV(ycsb, m.runtime.tag, 1);
        } else {
          plan <- SetKV(ycsb, m.runtime.tag, 0);
        }
      }
    }

    plan <- EpochLength(10.0);
  }
}
```

Figure 5.6: Implementation of the Cassandra Steering Policy.

Recall that nodes in Cassandra are logically arranged in a ring, and requests can be sent to any node (Section 3.3.2). This node will then act as the coordinator for the request, assemble a quorum by contacting the nodes that hold replicas of the requested key-value pair, and return the result. Replicas are assigned to nodes through consistent hashing which maps replicas to successive nodes in the ring. Through inspection and instrumentation of Cassandra executions, we identified three fundamental reasons for stragglers:

1. The node that acts as the coordinator for a request stalls for garbage collection while handling the request, delaying the response to the client.
2. The coordinator does not stall, but one of the replicas it chooses to contact is delayed due to a garbage collection pause.
3. Several features in Cassandra that are not related to garbage collection may cause delays as well (e.g., log compaction or the anti-entropy mechanism).

We now show how to address the first two problems. The third problem is out of scope for this thesis, but may be addressed by Taurus as well, using its key-value pair mechanism to schedule non-GC operations at suitable times.

5.3.1 Request Steering Policy

This policy addresses the problem of nodes stalling for GC while processing a request. The key insight is that instead of distributing requests equally between all Cassandra nodes, we modify the load balancer to avoid sending requests to nodes that are likely to stall for GC in the near future. To achieve this, we identify all nodes that are close to GC and expose this information to the load balancer (in our case, the YCSB client). We implemented this approach using a policy that monitors the young-generation occupancy of each node during every 10ms epoch and communicates one key-value pair per node to the YCSB client, which indicates whether or not the node is close to GC.

Figure 5.5 visualizes this strategy. The policy first identifies the node that runs YCSB, based on its tag. It then builds a set of key-value pairs for this node which are 1 if a node is close to performing GC, or 0 if it is not. These key-value pairs are then made visible to YCSB. YCSB then checks these key-value pairs before dispatching each request. If the key-value pair is 1, YCSB sends the request to a different node.

This required minor modifications to YCSB. By default, YCSB maintains one connection to each node, with an associated thread that generates a continuous stream of requests to this node. We introduced a second connection per node that does not generate requests itself but instead drains a queue of requests that are forwarded from other nodes. Before sending a request, the original connection threads first check the corresponding key-value pair, and if the value is 1, enqueue the request to one of the secondary connections, skipping those whose key-value pair is set to 1 themselves.

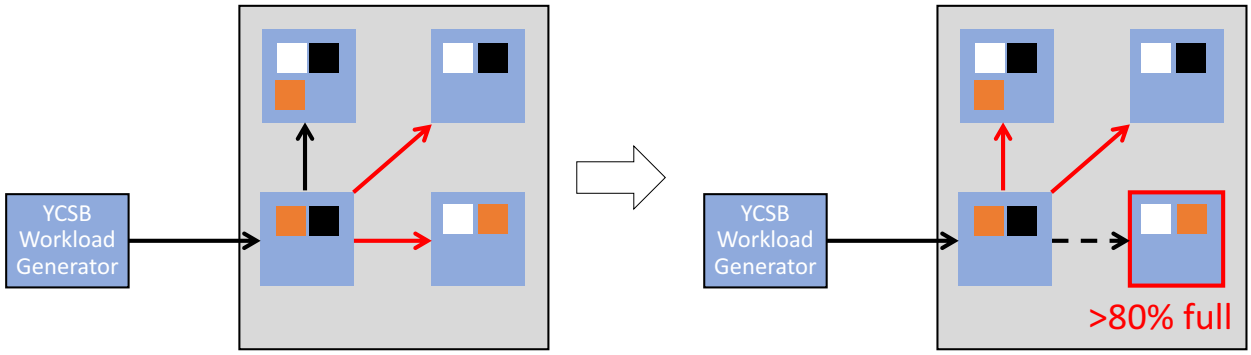


Figure 5.7: *Cassandra Snitch Steering Strategy.* When assembling a quorum, Cassandra picks a subset of replicas based on a mechanism known as "Snitch". This mechanism ranks Cassandra nodes according to preference, typically based on proximity. We modified the Snitch to instead give precedence to nodes that are not close to GC; in this example, the node that is close to GC (i.e., has a young generation that is > 80 % full) is given least precedence.

This approach eliminates most update stragglers over 20ms, as shown by the "Steer" results in Figure 5.4). However, read queries do not improve. This is because updates can be delayed by the coordinator if a quorum of replicas cannot be assembled. In contrast, reads need to receive responses from a quorum of replicas before the result can be returned. The next strategy will address this problem.

5.3.2 Snitch Steering Policy

To improve read queries as well, we implemented a second coordination strategy. This policy can be seen as another instance of the Steering policy from Section 5.3.1. To serve a read query, the node executing the query has to assemble a quorum of replicas (Figure 5.7). This quorum is chosen based on a *Snitch*, a feature in Cassandra that is normally used to describe the data center topology and discover nodes that are close in the data center (e.g., in the same rack). The snitch ranks existing nodes in the system (typically by proximity), and Cassandra then picks and contacts the highest-ranking nodes to form a quorum.

We modified Cassandra's dynamic snitch to read Taurus's key-value pairs and give precedence to replicas that are not close to GC. This approach uses the key-value API presented in Section 4.1.4 and introduces fewer than 50 lines of code. As Figure 5.8 shows, we had to modify the `compareEndpoints` function of the snitch to give precedence to nodes that are not close to GC. This is determined by a function `isBusy`, which reads Taurus's key-value pairs using the `HVM` API (Figure 5.6 shows the policy code that sets them). We maintain a `wasBusy` table that represents the most recent knowledge whether or not a node is close to GC. Note that similar to accessing key-value pairs in policies, accessing key-value pairs through the `HVM` API can fail – by maintaining the `wasBusy` table, we are robust to this case (which can happen during start-up and failure recovery).

```
public class HVMSnitch extends AbstractEndpointSnitch {
    public List<InetAddress> getSortedListByProximity(InetAddress address,
            Collection<InetAddress> unsortedAddress) {
      List<InetAddress> preferred = new ArrayList<InetAddress>(unsortedAddress);
      sortByProximity(address, preferred);
      return preferred;
    }

    public void sortByProximity(final InetAddress address,
            List<InetAddress> addresses) {
      Collections.sort(addresses, new Comparator<InetAddress>() {
        public int compare(InetAddress a1, InetAddress a2) {
          return compareEndpoints(address, a1, a2);
        }
      });
    }

    public int compareEndpoints(InetAddress target,
            InetAddress a1, InetAddress a2) {
      boolean unavailable1 = isBusy(a1);
      boolean unavailable2 = isBusy(a2);
      if (unavailable1 && !unavailable2)
        return 1;
      else if (!unavailable1 && unavailable2)
        return -1;
      return 0;
    }

    public static boolean[] wasBusy; // Keeps track of nodes close to GC

    public static boolean isBusy(InetAddress a1) {
      int tag = nodeMapping.get(a1);
      String str = HVM.getKeyValuePair(Integer.toString(tag));
      if (str == null) { /* No KV pair (e.g., due to start-up or failures) */ }
      else if (str.equals("1")) { wasBusy[tag] = true; }
      else { wasBusy[tag] = false; }
      return wasBusy[tag];
    }
}
```

Figure 5.8: Abridged version of the modified `HVMSnitch` class implementing Snitch Steering in Apache Cassandra. The snitch orderes replicas by preference – we modify the snitch such that it gives precedence to replicas that are unlikely to become unavailable due to GC.

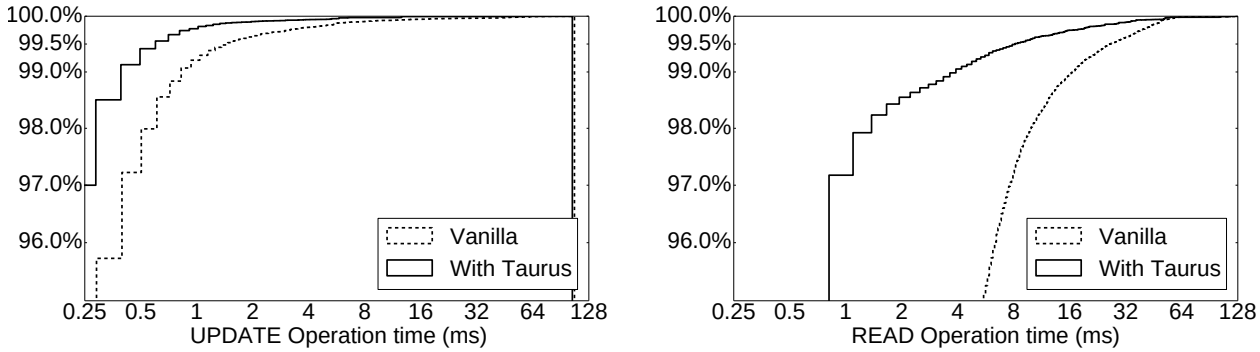


Figure 5.9: Fraction of total execution time (sum of all query latencies) spent in requests of at least a certain length. This demonstrates that Taurus strictly outperforms the baseline.

The "Snitch" results from Figure 5.4 shows that running with this policy, in addition to request steering, improves read query latencies substantially.

5.3.3 Staggering Policy

While the two steering policies substantially improve query latencies, additional read stragglers remain. Further analysis revealed that some of these stragglers stem from multiple GCs occurring at the same time. Serving a request requires 2 out of 3 replicas to be available simultaneously. Otherwise, the coordinator cannot form a quorum, even with steering.

To address this problem, we modified the policy to ensure that only one node is performing GC at any time. Each epoch, we trigger a minor GC on the node with the highest memory occupancy over 80%, if any, and trigger no other GC before it has finished. For our 8-node cluster, this is sufficient. Larger clusters could stagger GC by triggering simultaneous GCs on nodes 3 steps apart in the Cassandra ring. This would avoid stalling multiple replicas belonging to the same key, as replicas are placed on subsequent nodes in the ring.

5.3.4 Impact of Coordination Policies

The overall impact of all coordination techniques is shown in Figure 5.9. The y axis shows the fraction of total time spent in requests of at least the latency on the x axis, averaged over 10ms intervals. As our coordinated version is well to the left of the original (note the log scale), we eliminate a large fraction of stragglers. On a per-request basis, the 99.99%ile latency improves from 65.7 ms to 33.8 ms for reads (40.7 ms to 10.1 ms for updates) and the 99.999%ile from 128.6 ms to 54.6 ms for reads (67.7 ms to 21.0 ms for updates).

Figure 5.10 demonstrates the same effect visually: When we switch from running without coordination (red) to applying the coordination techniques (green), most of the long stragglers disappear. However, some stragglers remain – this is mostly due to non-GC related reasons.

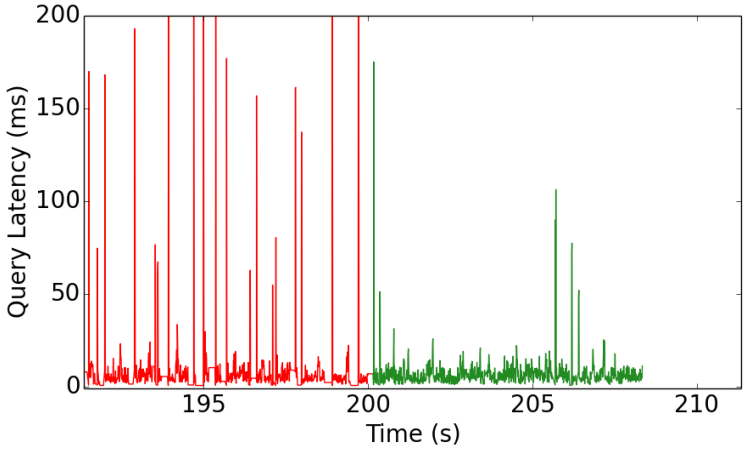


Figure 5.10: *Effect of Taurus on Cassandra READ latencies.* The measurements are averaged over 10 ms intervals. At 200 seconds, we switch from no coordination to a policy implementing all coordination strategies discussed in Section 5.3.

While it may be possible to use Taurus to take these causes of delays into account when generating the key-value pairs, we have not further explored this direction yet.

Note that we use an untuned configuration of Cassandra, with a large heap and young generation. In practice, Cassandra deployments are heavily hand-tuned. We argue that with a system such as Taurus, tuning becomes less important.

5.4 Generalizing GC Coordination Strategies

While the previous sections show specific instances of Taurus coordination policies, we now attempt to generalize these strategies to a wider range of workloads. Looking at a range of data center workloads, we discovered that most garbage collection policies we found could be expressed as a combination of three fundamental coordination strategies:

- **Schedule**: Both the *Stop-the-Universe* policy we used in Spark and the *Staggering* used for Cassandra trigger garbage collection pauses deliberately instead of incurring them when the runtime system runs out of memory. This generalizes to a class of strategies where Taurus triggers GC at suitable or convenient times (what constitutes such a time is workload-dependent).

- **Redirect**: Both the *Request steering* and *Snitch steering* policies for Cassandra are instances of strategies that redirect requests to nodes that are unlikely to stall for garbage collection. This generalizes to any case where a task or request can be handled by multiple nodes. In this case, a policy can ensure that requests are always redirected to one of the nodes that is not unavailable due to GC.

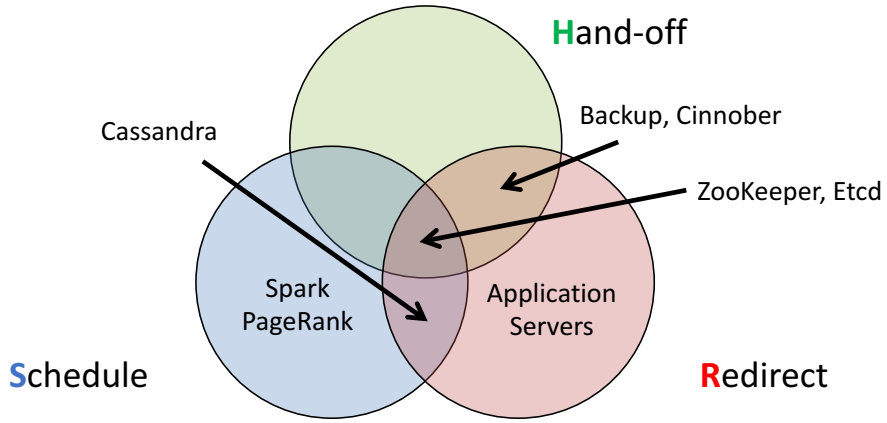


Figure 5.11: *Mapping of GC policies to base strategies.* Almost all GC coordination strategies that we are aware of can be decomposed into three fundamental approaches. *Schedule* strategies trigger GC deliberately at times that are convenient for the application. *Redirect* applies to systems where multiple nodes can handle the same request, and steers requests to those nodes that are unlikely to stall for GC. *Hand-off* is used for systems that have a centralized point of failure, and causes this node to pass their responsibilities to another node as it is getting closer to GC.

- **Hand-off**: A third class of strategies does not appear in our examples but has been investigated in the literature [212]. These strategies concern cases where a single node has the sole responsibility for a task or a piece of data (e.g., a master node in a centralized distributed system). To handle GC pauses in these cases, the policy can hand the node's responsibility to another node whenever the node is close to GC.

We introduce the shorthands *S*, *R* and *H* for these fundamental strategies. Looking at the published literature and our own work, we were able to describe nearly all policies that we are aware of in terms of these fundamental strategies, providing a strong indication that they are sufficiently general to span the space of possible coordination strategies:

1. **Stop-the-Universe (S).** Synchronize GC such that collections are performed on all nodes in the distributed system at the same time. This strategy is useful for iterative batch workloads, such as the Spark example we presented in Section 5.2. This includes a wide range of algorithms, including graph algorithms, machine learning algorithms and scientific computations.

2. **GC-when-Idle (S).** Trigger GC during a period of idleness. For example, many computations involve multiple phases such as reading data from a data store (e.g., HDFS), processing this data and then writing back the results. GC could be scheduled such that the component in question is idle during the GC (e.g., we might schedule the GC for the data store while the data is being processed).

3. **Steering (R).** In a system where multiple servers can handle the same request (e.g., key-value stores such as Cassandra, sharded search engines such as SOLR [202], or replicated services such as fleets of web servers), steer requests away from nodes that are close to GC. This ensures that nodes do not incur GC pauses while handling requests. Steering can happen within applications (e.g., within the application's consensus algorithm) or at a load-balancer. We saw two instances of this strategy in Section 5.3.

4. **Staggering (S).** In a system that requires quorums of replicas for consistency, ensure that only a sufficiently small subset of the replicas is performing GC at any given time. This can be achieved by staggering garbage collection across the different nodes in the system, as we saw in Section 5.3.3.

5. **Backup (SR).** Have two instances of a component, but use only one of them at a time, by steering all requests to this node. As soon as this node gets close to garbage collection, we switch over to the other node instead and force a collection. This approach has been successfully used in latency-sensitive finance applications [181].

6. **Leadership Transfer (H).** When the leader of a centralized distributed system is about to stall for GC, hand off leadership to another node in the system. This can often be achieved using the recovery mechanism that is already part of the system, since this behavior resembles leadership re-election after a failure [212].

7. **Timeout Extension.** This is the only strategy we found that could not be expressed in terms of the basic strategies. It has been reported that garbage collection in some distributed workloads cause leases or connections to time out, which can trigger the failure recovery mechanism [113]. An application running on Taurus could instead extend leases whenever a collection is being performed.

Applications can combine these policies to fit their needs. Figure 5.11 shows examples: Spark (Section 5.2) benefits from *Stop-the-Universe* (*S*), Cassandra (Section 5.3) uses *Steering* & *Staggering* (*SR*), a Zookeeper-like system [212] has been shown to benefit from a *Steering*, *Staggering* & *Leadership Transfer* (SRH), and replicated application servers [180] have been shown to benefit from *Steering* (*R*) alone.

We note that there is a connection between the fundamental policies and their implementation complexity: *S*-type strategies oftentimes require no changes to the application, *R* strategies do require changes but those are minimal if it is possible to modify the replica selection algorithm already available in applications, while *H* strategies can be more work, unless the application already has a hand-off mechanism as part of its failover handling.

Taurus enables and simplifies the implementation of these policies, and we hypothesize that they provide a basic set that can be combined to cover most applications (and could be incorporated into a standard library of policies).

5.5 Summary

We believe that the presented strategies generalizes to a wide range of both batch and interactive workloads. We also think that the same mechanisms may be used for other system-level and maintenance events, such as log compaction.

An important conclusion from this work is that these strategies were possible by working on neither the managed-runtime layer or the systems layer in isolation, but by working across the boundary between them. However, while Taurus can help applications tolerate GC pauses better, it does not solve the fundamental underlying problem. Specifically:

1. Applications still spend up to 38% of their CPU time in garbage collection, taking up a large number of CPU cycles and energy [43].

2. While garbage collection pauses can often be tolerated, they do not disappear, and some irregular applications may not be able to use coordination effectively to avoid them. Further, there is a large amount of legacy code that is difficult to change and adapt to a system such as Taurus.

It would therefore be preferable to reduce the impact of garbage collection altogether, both in terms of cycles and energy spent on it, as well as GC pauses. We believe that this is possible by looking at the problem from a different perspective: Instead of only investigating the problem in the context of software, we believe that we can address garbage collection more fundamentally by considering the hardware as well. Specifically, we hypothesize that the impact of garbage collection could be reduced by offloading it from the CPU onto specialized hardware. This idea is what the remainder of this thesis will be exploring.

Chapter 6

Offloading Garbage Collection

This chapter presents a different approach to the garbage collection problem. Instead of enabling applications to tolerate GC-related pauses better, we remove garbage collection from the CPU and offload it onto a data-parallel accelerator. We demonstrate one instance of this approach by offloading GC to an integrated GPU in a combined GPU-CPU part, and show how this motivates the design of a custom data-parallel accelerator for GC.

6.1 Offloading Garbage Collection to the GPU

While Taurus enabled tolerating GC pauses by working across the language runtime and systems layers, we now investigate ways to address the garbage collection problem at a more fundamental level, by working across the language runtime and hardware layers to design custom accelerators that perform garbage collection more efficiently than a CPU.

As a first step, we investigated offloading garbage collection to accelerators that already exist in commodity systems and data center servers. Specifically, we offload the compute-intensive mark phase of a garbage collector to the integrated on-chip Graphics Processing Units (GPUs) that are widely available in desktop and server-class machines.

GPUs have been part of commodity systems for over 15 years. While they were originally designed to accelerate 3D graphics (e.g., for video games), frameworks such as CUDA and OpenCL have enabled GPUs to run general-purpose workloads, including special-purpose computations such as deep learning. In the absence of such workloads, the GPU is often underutilized. This is true despite the recent renaissance of GPUs in machine learning: While this led to a large amount of work on dedicated GPUs, this work is typically limited to high-end devices such as NVIDIA's Tesla GPU systems [167].

While GPUs have long been discrete devices, CPUs and GPUs are now often integrated in a single system on chip (SoC). This setup opens up a whole new set of application scenarios, since it eliminates the copying overhead that is traditionally associated with moving data between the CPU and a dedicated GPU. While early implementations of this paradigm were

simplistic and only shared physical memory, modern hardware now provides a shared address space and cache coherence between CPU and GPU [40].

We believe that this integration provides an opportunity to move traditional systems workloads to the GPU. Garbage collection appears to be a particularly good candidate for this, since garbage-collected languages such as C# and Java account for a significant portion of code running both in data centers and on consumer devices. Offloading their GC workloads to the GPU allows us to harvest the GPU's unused compute power, leaving the CPU free to perform other tasks such as JIT compilation, garbage collection for other memory spaces, or running mutator threads (if the GPU is used for concurrent garbage collection).

Our intuition was that garbage collection is a workload that is well-suited for running on the GPU, especially once the copy overhead between CPU and GPU disappears. Graph traversals (a key component of many garbage collectors) have already been efficiently demonstrated on GPUs [105], and previous work by Veldema and Philippsen [220] has shown that garbage collection for GPU programs can be efficiently performed on the GPU itself.

In this chapter, we take the next step and investigate whether it is feasible to offload GC from conventional programs running on the CPU, and what it takes to achieve this goal. We show that GPUs can, on average, perform GC with overheads ranging from 40–100% compared to a CPU, despite the GPU's SIMD-style programming model and the need to design algorithms that make explicit use of available parallelism and memory bandwidth, while avoiding serialization of execution. The contributions of this chapter are as follows:

- We present an analysis of the heap graphs of several Java benchmarks, to evaluate the theoretic feasibility of using GPUs and data-parallel accelerators for garbage collection (Section 6.3). These insights inform both our GPU-based GC design as well as our custom garbage collection accelerator (Chapter 7).

- We prototype a GPU-based garbage collector within the *Jikes Research Virtual Machine* [4] (Section 6.4), a Java VM maintained by the research community. We later adopt this infrastructure as part of a broader evaluation methodology (Chapter 8).

- We show a new algorithm, and variations thereof, for performing the mark phase of a Mark & Sweep garbage collector on a GPU. Our algorithm differs from previous work by using a frontier queue approach instead of a data-parallel algorithm. We also discuss trade-offs and optimizations to make it efficient on a GPU, which form the foundation for the custom GC accelerator presented in the next chapter.

The objective of this work is not to present a single tuned implementation, and the implementation presented in Section 6.4 is mainly for the purpose of illustrating a particular point in the design space. Our main goal is to assess whether GPUs – and data-parallel accelerators in general – are feasible targets for offloading garbage collection, and to identify obstacles that need to be overcome. These insights led us to design our own custom hardware in the next chapter, which takes the lessons we learned from implementing garbage collection on GPUs, and distills them into a custom data-parallel accelerator.

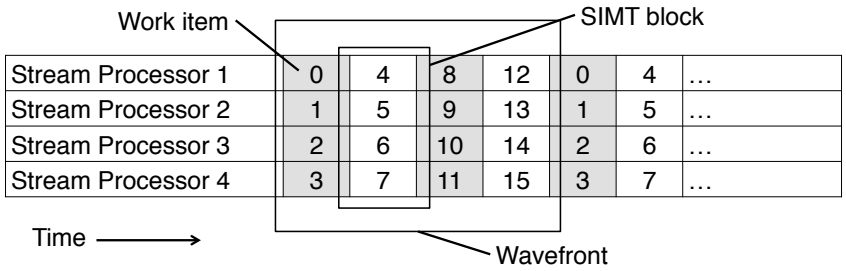


Figure 6.1: Illustration of the spatial and temporal scheduling of work-items for a fictional compute unit with 4 stream processors and a wavefront size of 16.

6.2 GPU Programming Model

This section provides a general introduction to the hardware and programming model of a GPU. Note that throughout this chapter, we use the terminology from *OpenCL* – the terminology used by *CUDA* (the other major framework) is synonymous[1]. We also want to note that we describe GPU architectures and terminology as of 2011-2012, when we originally did this work and when the GPU that we are using was released. We discuss differences to the current state-of-the-art in Section 6.7.5.

GPUs provide a SIMT (*Single Instruction Multiple Thread*) programming model. SIMT is an extension of SIMD (Single Instruction Multiple Data) with support for hardware execution masking to handle divergent control paths within an instruction block. Computation is described in terms of a *kernel* which is executed by a set of *work-items* (i.e., threads).

The basic building block of a GPU is a *streaming multiprocessor* (SM), or *compute unit*, which contains a single instruction decoder and a number – typically between 8 and 64 – of *stream processors* (SP). The stream processors execute the same instruction in lockstep, but with different register contexts (each of them stores the registers and a small amount of memory for each of its work-items). Within the compute unit, stream processors share access to a *Local Data Store*, a small fast block of dedicated memory.

Work-items are grouped into *wavefronts* (Figure 6.1). Each wavefront typically contains four times the number of stream processors. The work-items of the wavefront are interwoven such that each stream processor executes the same instruction four times – once for each quarter of the wavefront. To handle divergence of control flow within a wavefront (e.g., one work-item takes a branch while another work-item does not), the hardware will perform masked execution. Both sides of the branch will be executed, but only some of the work-items will be enabled. For good performance, it is critically important to minimize the amount of divergence in the control flow.

Wavefronts are in turn grouped into *workgroups* (256 to 1,024 work-items per workgroup are common). When a given wavefront stalls because of a memory access, another ready

[1]For further reading, the AMD OpenCL Programming Guide [8] and the OpenCL v1.2 Specification [130] provide all the detail one might require.

wavefront begins executing. Context switches between wavefronts are extremely fast (usually a single cycle), since each work-item in the entire workgroup retains its dedicated registers at all times. To maximize memory bandwidth, a kernel should maximize the number of wavefronts that are able to perform independent memory accesses.

GPUs have a number of compute units which share access to a memory region known as *global memory*. On the devices we investigated in 2012, the number of compute units varied from 2-4 on a low-end device to as many as 12-40 in high-end devices (today, NVIDIA sells parts with up to 512 compute units). For discrete GPUs such as graphics cards, global memory is dedicated hardware on the device; for integrated GPUs, where CPU and GPU share the same package, it will often be a reserved area of the main system memory.

The discrete approach has the advantage of much faster access times, but requires slow – on the order of 8 GB/s in 2012 – explicit copies between CPU and GPU, using DMA over the PCIe bus (modern NVLink-based systems can achieve 300 GB/s). In 2012, neither approach participated in the cache-coherence protocol of the CPU; this meant that communication between CPU and GPU had to be done explicitly through the OpenCL interface. Today, there exist cache-coherent CPU-GPU combinations.

6.3 Preliminary Analysis

Garbage collection in general – and the mark phase of a Mark & Sweep garbage collector specifically – is a memory-bound problem. As such, the main challenge of any implementation is to process and issue memory requests at a sufficiently high rate to fully utilize the available memory bandwidth. As we will discuss more in Section 6.5, being able to process a large number (i.e., hundreds) of objects in parallel is essential for meeting this goal on a GPU (and, as we will see in the next chapter, on a data-parallel accelerator).

The core of our GPU-based mark algorithm is a highly parallel queue-based breadth-first search (recall Section 2.3). Objects to be processed are added to a *frontier queue*. For each item in the queue, we remove it, mark the object, and then add each outbound reference to the queue. This processes objects in order of increasing distance from the root set (i.e., increasing *depth*). At each depth, there is a fixed *width* (or *beam*) of nodes available for processing. If this available width is greater than the number of work-items, we can keep the entire device busy and make efficient progress through the traversal.

Any practical collector can do no better than an ideal collector which examines every object at a given depth in a single iteration. To understand this theoretical best case garbage collector on real programs, we examined the heap structure of benchmarks from the DaCapo 9.12 benchmark suite [37]. We examine two attributes of heap graphs: their general shape (i.e., depth, width per iteration, etc.) and the distribution of outbound references across objects. The latter has a significant performance impact on GPUs due to the divergence problem described previously (Section 6.2).

The data collection for this section was performed using an instrumented Jikes garbage collection plan. All measurements were performed using the optimizing compiler; optimization

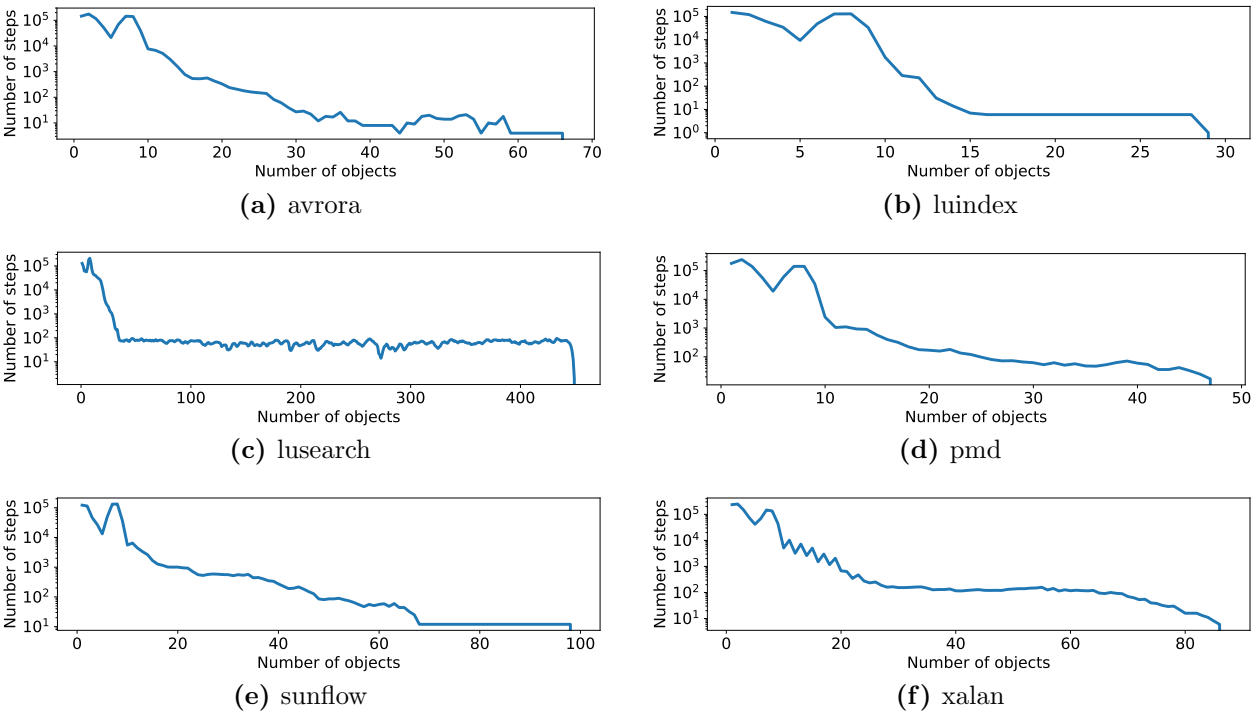


Figure 6.2: *Number of objects at each depth during an idealized breadth-first traversal starting from the root set.* This exemplifies the degree of parallelism available.

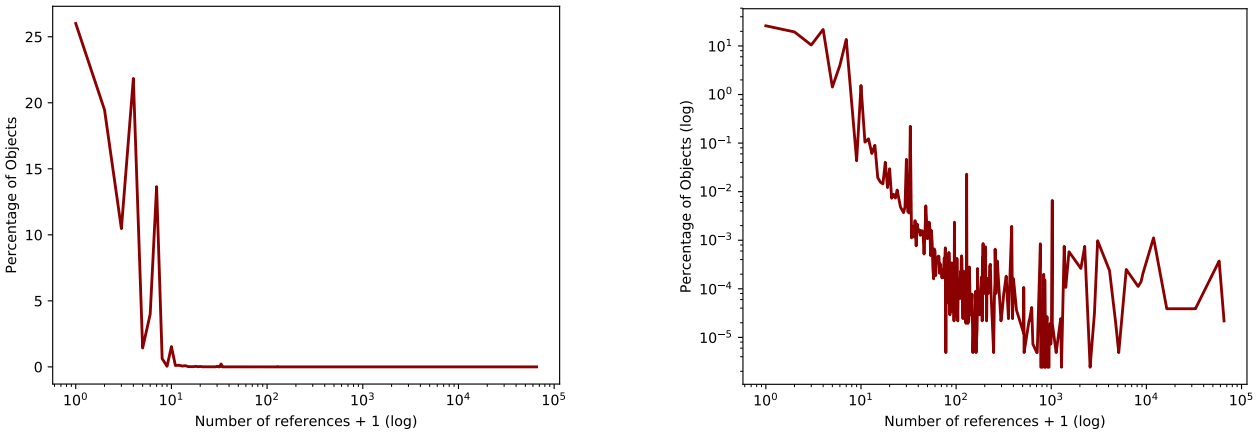
affects the frequency of collection and thus the heap graphs' shapes. We ran the small and default configurations of a subset of the benchmarks[2]. For further details about the structure of common Java heaps, we recommend Barabash and Petrank's paper [26]. They analyze heap depth and approximate shape for a previous release of the DaCapo benchmarks, as well as several Java SPEC benchmarks.

6.3.1 Structural Limits on Parallelism

We first examined the general shape of the heap graphs as traversed by the ideal collector. We were interested in determining whether there were structural limits that would prevent the degree of parallel processing that data-parallel architectures require for efficiency.

A selection of the graphs generated from the DaCapo benchmarks is shown in Figure 6.2. The figures show the number of objects reachable – marked or unmarked – from a given step of the ideal breadth-first traversal starting at the root set. All of the benchmarks begin with a short section of extreme parallelism. The first step is limited to the size of the root set (typically 600-1,000 objects), but the next few steps expand rapidly.

[2]We only report a subset of the benchmark suite since several benchmarks did not work on a vanilla Jikes RVM running on our evaluation system. This is a known problem and unrelated to our garbage collector.



(a) Linear Scale **(b)** Log Scale

Figure 6.3: *Distribution of the number of references within objects on the heap.* Most objects have few outbound references, but some objects have hundreds or thousands.

Once this startup section is completed, our benchmarks fell into three categories. Some of the benchmarks – such as `luindex` – then complete within a small number of additional steps. A few – such as `avrora` – had moderate length sections of structurally limited parallelism. Unfortunately, there were also a few benchmarks – such as `lusearch` – which had long narrow sections ("tails") following the parallel beginning. A width of 30 to 80 represents at most 1/3 of the available parallelism on the GPU. As the number of available work-items per workgroup increases with time, this fraction may drop precipitously.

Since no hardware can execute an infinite number of threads, we repeated the analysis above while limiting the maximum step widths to 128, 256, 1,024, and 32,768. As expected, decreasing the number of items processed in each iteration increased the effective depth of the graph, but did not change the overall shape of any of the benchmarks.

Despite the limited parallelism towards the end of some collections, we conclude that heap graphs are sufficiently parallel for the purposes of garbage collection on GPUs and custom accelerators. However, if one wants to minimize collection latency, having a mechanism to deal efficiently with long narrow tails in the heap graph is critical; we discuss our solution in Section 6.5.4. An alternative approach would be to insert artificial shortcut edges into the heap graph. Barabash and Petrank describe this strategy in detail [26].

6.3.2 Distribution of Outbound References

Prompted by Veldema and Philippsen's [220] findings on divergence when performing garbage collection on GPUs, the second issue we examined was the distribution of the number of outbound references within each object. When processing one object per thread in a SIMT environment where each thread loops over the outbound references within its object, this

distribution is critical to understanding and controlling divergence. Our results show that the vast majority of objects have a small out-degree: 26% of objects have no outbound references (other than their class pointer), 76% have four or fewer, and 98% have 12 or fewer. However, this distribution also has a very long and noisy tail. A small fraction of objects (less than 0.01%) have hundreds to thousands of references. It is worth noting that our analysis does not distinguish between objects and arrays of references, but we have manually confirmed that some of the high double-digit out-degree nodes are, in fact, objects.

The distribution of the number of references within objects can be seen in Figure 6.3. Given that the results across benchmarks are fairly uniform, we chose to present the distribution across all the collections of all the benchmarks for which we collected results.

Even leaving aside the extreme tail of the distribution, the distribution of references between objects means that blindly looping over the number of references will result in unacceptable divergence of threads. We discuss one solution for distributing references between work-items in Section 6.5.2.

6.4 System Integration

In this section, we present challenges for offloading garbage collection to the GPU and discuss different performance trade-offs. To substantiate our claims, we implemented a proof-of-concept GPU-based garbage collector for the *Jikes Research Virtual Machine* [4]. This allows us to investigate performance trade-offs for full executions of real Java programs, by performing a series of macro and micro benchmarks.

Our test platform was an AMD E-350 APU, which was one of the first chips that integrated a CPU and GPGPU into a single device (Intel's Sandy Bridge architecture had a similar integrated GPU, but it was not programmable). APU (*Accelerated Processing Unit*) is a term coined by AMD to describe their integrated CPU/GPU solution marketed as AMD Fusion. The E-350 targeted low-end laptops and tablets and has since been subsumed by several successor generations (the latest being the *Bristol Ridge* series).

6.4.1 High-level Overview

We modified Jikes's Mark & Sweep garbage collector to offload its mark phase to the GPU. The operation of the collector can be divided into three phases:

1. **Root scanning**: JikesRVM stops all application threads, scans their stacks for references and copies all outgoing references from the set of static variables.

2. **Mark phase**: The collector performs a breadth-first search through the heap graph and sets the mark bits of all visited objects (as described in Section 2.3).

3. **Sweep phase**: The collector traverses all blocks in memory and recycles those cells that have not been marked in the previous phase.

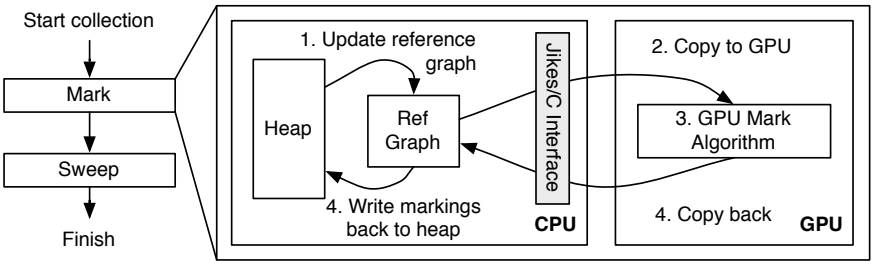


Figure 6.4: Overview of the collector integration.

Execution time is dominated by the mark and sweep phases. Both expose a high degree of parallelism – while we have shown the mark phase's parallelism in the previous section, the sweep phase is embarrassingly parallel, as different blocks can be swept simultaneously.

We integrated our GPU-based GC into Jikes's stop-the-world `MarkSweep` collector. We modified the collector to offload the mark phase to a GPU and investigate the challenges of targeting it to a data-parallel architecture. While a real-world collector would have to offload the sweep phase as well, we think that Veldema et al. have sufficiently covered this aspect in their work, and therefore focus on the mark phase for brevity. A complete collector would perform the sweep phase on the GPU, immediately after the mark algorithm, and only copy the resulting free lists back to JikesRVM running on the CPU.

The steps performed by the collector are shown in Figure 6.4. For experimentation purposes, our mark phase is performed on a *reference graph* data structure, a self-contained version of the heap that only contains references but no other fields. We discuss this structure in the next sections. The reference graph is kept up-to-date during program execution or filled in on each collection. We then invoke our collector in a native call which sets up our mark kernel and runs it on the GPU. The CPU is idle until the mark completes (a production-grade implementation would perform other tasks during this time). Upon completion, execution returns to Jikes and the markings are transferred back into the heap. The sweep phase is then performed on the CPU. Note that the intermediary copying steps are merely an implementation detail of our prototype, and not inherent to our approach.

6.4.2 Bidirectional Object Layout

A major challenge of offloading the mark phase to a GPU is the JVM's object layout. Every object has reference and non-reference fields. During the mark phase, the collector needs to follow non-zero pointers in all reference fields (i.e., the outgoing references).

In a traditional object layout (Figure 6.5a), the header is at the beginning of the object, followed by the fields declared by the parent class and then the fields of the class itself. This enables inheritance, as it allows an object to be treated as an instance of any of its parent classes. However, this layout means that reference and non-reference fields are interspersed.

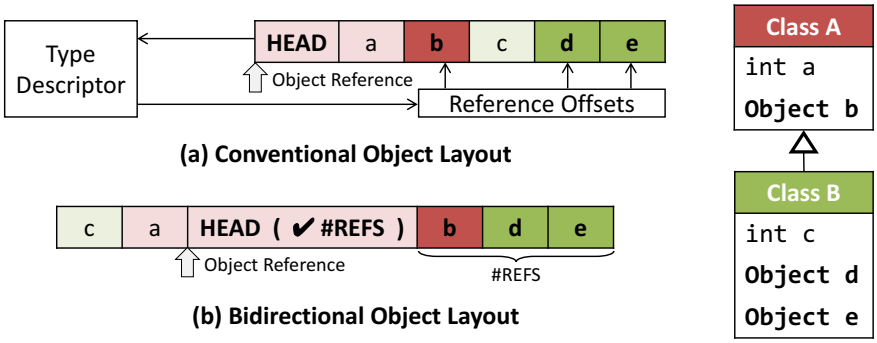


Figure 6.5: *Bidirectional Object Layout.* In a traditional object layout, the header is at the beginning of the object and reference and non-reference fields are interspersed throughout the object (with a side-table to indicate which fields contain references). In a bidirectional layout layout, the header is in the middle, references are to one side and non-reference fields to the other. This saves lookups in the type descriptor, provides a more advantageous memory access pattern, but still supports inheritance.

Existing JVMs get around this problem by having a side-table which contains the offsets of all reference fields. When tracing an object during the mark phase, the collector can look up the offsets of the reference fields in this side table. As these tables are typically in cache, this does not add a substantial overhead on a CPU. However, on architectures without caches – such as GPUs – this adds up to three levels of indirection (type information block, type info, offset array), and incurs a performance penalty.

Another strategy that is often used on a CPU is to synthesize specialized tracing functions for the most common patterns of references. However, this is not feasible on a GPU either, as it would cause divergence to run a different tracing function for every object.

We therefore require a different object layout, which lays out the references of an object consecutively and contains the object's number of references in the object header (Figure 6.5b). A *bidirectional* layout [231] such as that used by the Sable VM [76] achieves this requirement with minimal overhead [79], by laying out reference fields to the right of the object header, and non-reference fields to the left. While this layout has only moderate performance benefits on CPUs [90], and has therefore seen little adoption, we found that it is crucial for making GC work on data-parallel architectures, as we will show in Chapter 7.

6.4.3 Reference Graph

A real-world GPU-based garbage collector would likely be implemented on an integrated GPU that is cache-coherent with the CPU and can operate on the same virtual address space as the application running on the CPU. However, the test platform that was available to us in 2012 (see Section 6.6.1) did not support either feature. Instead, it could only map a

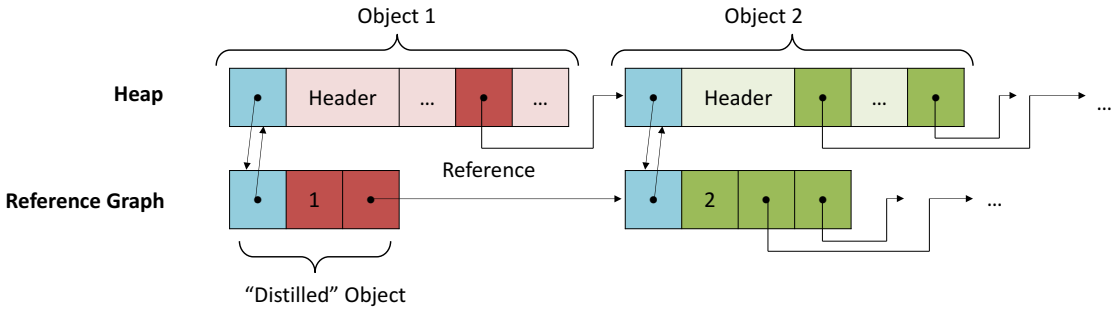


Figure 6.6: *The reference graph structure.* Each object on the heap has a corresponding object in the reference graph which only contains the reference fields.

128 MB subset of the application's memory at a time[3], by unmapping it on the CPU and mapping it on the GPU. This became problematic, as this size was too small to hold the heaps of several DaCapo benchmarks (when including Jikes's own memory spaces).

For the purposes of evaluation, we solved this problem by building a condensed version of the heap which we call a *reference graph*. The reference graph is stored in a separate space and contains an entry for each object on the main heap, consisting of a pointer to the original object, the number of references, and a consecutive list of all outbound references as pointers into the reference graph (Figure 6.6). Arrays are represented in the same way. This emulates the object layout presented in Section 6.4.2, but reduces the size of the heap such that it fits on the GPU. Due to the lack of caching on the GPU, this approach does not give a performance advantage, while it allows us to evaluate our collector on real-world heaps that otherwise would have been too large to fit on the GPU.

We found that the reference graph gave us a sufficient reduction in size to evaluate the DaCapo benchmarks on our collector. The following table shows the cumulative sum of heap sizes across all collections within a run, as well as the equivalent numbers for the reference graph (we used a maximum heap size of 100 MB for all runs). This allows us to estimate that the reference graph approach reduces the size of our graph by about 75% on average:

	# GCs	Cumulative Heap	Cumulative Reference Graph	Ratio
avrora	9	256 MB	80 MB	31.2%
jython	114	10499 MB	3301 MB	31.4%
luindex	7	178 MB	35 MB	19.8%
lusearch	77	7078 MB	515 MB	7.3%
pmd	14	809 MB	233 MB	28.9%
sunflow	39	2935 MB	658 MB	22.4%
xalan	23	1686 MB	456 MB	27.1%

We experimented with two different approaches for building and maintaining the reference graph. Both of these approaches allocate a node in the reference graph whenever a new

[3]This value was determined experimentally.

object is allocated, but differ in how they maintain references between them.

- The most basic approach fills in the reference graph immediately before performing a collection. It performs a linear scan through the "distilled" objects in the reference graph, follows the references of each corresponding original object, and copies the pointers for the corresponding distilled objects to the reference graph (Figure 6.6).
- The reference graph can also be built while running the mutator threads: this turns every reference write into a double-write to two locations, which can be implemented as either a write barrier or issuing a second write instruction in the compiler. We prototyped the simpler write barrier approach.

While these approaches differ in performance, we refrained from performing a deeper analysis, as this problem is somewhat orthogonal to our approach and lost its relevance shortly after this work was completed in 2012.

6.4.4 Launch Overhead

Equipped with the reference graph, our collector calls into a C function (using Jikes's SysCalls mechanism) which initializes OpenCL, copies (or maps) the reference graph to the GPU and launches the mark kernel. Launching a kernel execution incurs both a fixed startup cost, and a variable cost related to the kernel itself and the size of memory being mapped to the GPU. We incur these costs once per garbage collection.

While our original platform incurred launch overheads on the order of hundreds of milliseconds for remapping memory, modern architectures do not incur these overheads. At the same time, launch times have been trending downward at a steep rate, and are now in the low microsecond range [228]. For this reason, we exclude launch overheads from the execution times of our kernels – as long as the number of kernel launches is small, they can be discounted for the purpose of assessing the feasibility of offloading GC to the GPU[4].

6.4.5 Copy-back Overhead

After performing the mark phase on the GPU, our collector incurs an additional overhead from copying the marked reference graph back into main memory and transcribing the mark bits into the original heap structure. This is necessary for the integration with Jikes, but would not occur in a real-world collector that integrates both mark and sweep phase: after finishing the sweep phase on the GPU, the collector would simply move the resulting free-list to the CPU, ideally in a low-overhead, zero-copy operation. For this reason, we ignore this overhead for the purpose of our evaluation.

[4]Please see the original paper for detailed measurements of launch overheads on our platform [148].

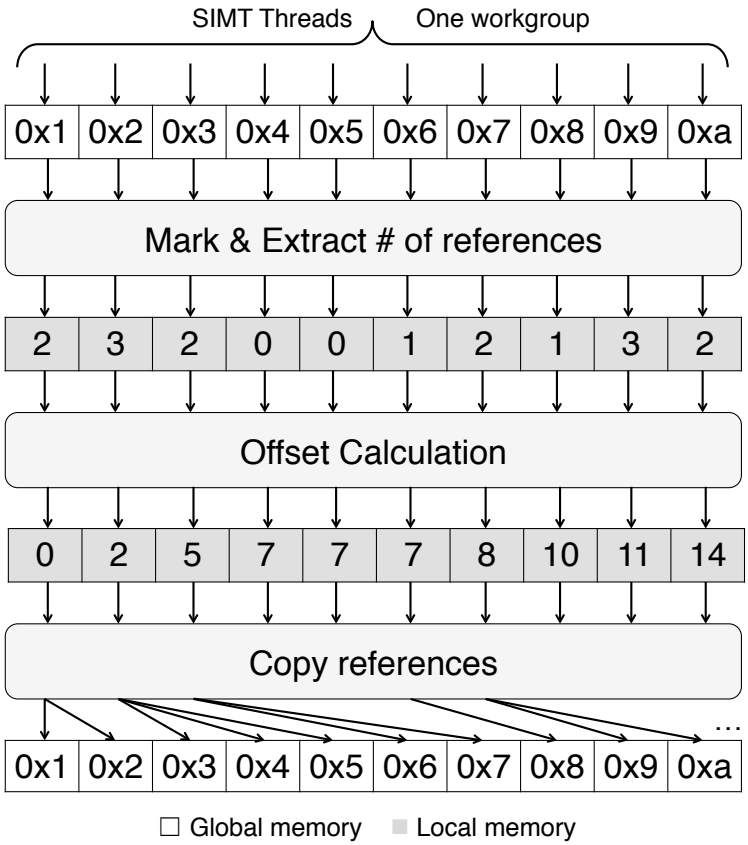


Figure 6.7: *Structure of the basic mark algorithm on the GPU.* Each SIMT thread is responsible for one object at the beginning of the current frontier (i.e., mark queue). It first reads the object's address from the frontier, followed by marking its header and extracting the number of outbound references (in one atomic operation). The threads then use this information to calculate the locations (i.e., offsets) where the outbound references need to be copied at the end of the frontier queue, in order to prevent them from overwriting each others' data. Finally, each thread copies its object's outbound references into the designated portion of the frontier queue. Once all have finished, the cycle repeats.

6.5 Algorithm and Optimizations

The core part of our collector consists of an algorithm that performs a parallel mark phase on the GPU, using n work-items (on our platform, $n = 256$). Our approach is based on maintaining a *frontier queue* that contains pointers to objects to be processed; we do not differentiate between arrays and objects. The kernel processes these elements in a loop: at each iteration, it removes up to n pointers from the queue, marks them, and adds the address of any referenced objects to the end of the queue.

Algorithm 1 Pseudo-code describing one step of the GPU mark phase. *id* is the SIMT thread ID, *in_queue* points towards the beginning of the frontier and *out_queue* points towards the end (we assume these regions are contiguous and do not overlap). x is a temporary array in local memory, and $MARK\_BIT$ represents the new value of the mark bit.

function MARK_PHASE (*id*, *in_queue*, *out_queue*)
1: $x \leftarrow (0, \ldots, 0)$
2: $l \leftarrow \min(length(in\_queue), WORK\_GROUP\_SIZE)$
3: **if** $id \geq l$ **or** $in\_queue\,[id] == NULL$ **then** return
4:
5: $old\_header \leftarrow fetch\_and\_mark(in\_queue\,[id]\,, MARK\_BIT)$
6: **if** $\neg marked(old\_header)$ **then**
7: $\quad refcount \leftarrow ref\_count(old\_header)$
8: **else**
9: $\quad refcount \leftarrow 0$
10: **end if**
11: $x\,[id] \leftarrow refcount$
12:
13: $offset \leftarrow compute\_offset(id, x, l)$
14:
15: **for** $i = 0$ to $refcount - 1$ **do**
16: $\quad refptr \leftarrow in\_queue\,[id] + i + HEADER\_SIZE$
17: $\quad out\_queue\,[offset + i] \leftarrow *refptr$
18: **end for**

Veldema and Philippsen [220] identified synchronization as a core problem of such an approach: in an implementation where each work-item accesses the queue in an atomic operation, execution would be serialized and therefore very inefficient. Based on this observation, they discard the queue-based approach and instead show a data-parallel implementation that switches to the CPU after every iteration, to spawn a new set of work-items.

In contrast, we execute the entire mark phase on the GPU. We avoid the problem of serialization by calculating in on-chip memory the total number of elements to remove and add to the queue, as well as their offsets. This avoids the need for per work-item atomic operations on the critical regions of the queue. At the same time, running completely on the GPU avoids switching between CPU and GPU, since we found the associated launch overhead to be too significant for this approach (Section 6.4.4).

Our algorithm is implemented as an OpenCL kernel which executes the code in Algorithm 1 in a loop until the frontier queue is empty (using $WORK\_GROUP\_SIZE$ work items). We discuss the details of this algorithm below. For the purposes of this explanation, assume that *in_queue* and *out_queue* are pointers to the parts of the queue where we are extracting elements from and where we store new elements, respectively. On each iteration, we remove up to $WORK\_GROUP\_SIZE$ pointers from *in_queue* and examine the corresponding

objects in parallel. We then mark all objects that have not been marked before and copy their references to the end of *out_queue*. This is done in three steps (Figure 6.7):

1. For all objects, read the number of outgoing references and whether the object has been marked. Objects that have already been marked are treated like an object with zero references. This is facilitated by the bidirectional object layout (Section 6.4.2): By storing the mark bit and the number of references in the header, this step can be performed in a single memory operation.

2. Compute the offsets that the references of each object will have in *out_queue*, using either a prefix sum or histogram approach (discussed in Section 6.5.1). For the ease of exposition, assume the prefix sum approach for now, which lays out the references of an object consecutively, one object after another.

3. Copy all outgoing references to their new location in the frontier, using the previously calculated offsets to determine where to store the references of each object.

Only between iterations do we update the queue's start- and end-offsets. This can be done by a single work-item per workgroup, since all work-items know the number of elements that are removed from the queue (l) and the number of elements that are added to the queue (which is given by the offset calculation – e.g. the right-most entry of the prefix-sum).

The following gives a more detailed description of the algorithm that is executed by each work-item. Note that *id* is the offset of each individual work-item within the workgroup.

- Lines 1-3 set up the necessary data structures and drop out of the function if there is no work to do for the work-item. Notably, x is allocated in the local scratchpad memory and used to efficiently calculate the offsets into the output queue.

- Lines 5-11 implement the first part of the algorithm. Each work item retrieves its object's header, in order to extract the marking and the reference count. It then stores the reference count in x.

- Line 13 calculates the offsets for writing into the output queue. We implemented several options, which are discussed in Section 6.5.1. The presented algorithm uses a simple prefix-sum operation to determine the offset for each object in the output queue, and stores the object's references in consecutive slots after this offset. The next section discusses this aspect in detail.

- Lines 15-18 describe the final part of the algorithm. The references are copied one-by-one into their dedicated locations in the output queue. The output calculation in the previous step ensures that no two references are written into the same slot, avoiding the requirement for synchronization or locking. In the given code, the fixed constant *HEADER_SIZE* represents the offset of the first reference from the beginning of the object header (which is constant for all objects).

Algorithm 2 Histogram approach for the offset calculation.

```
1: hist ← (0, ..., 0)
2: global_offset ← 0
3: atomic_max(&max_refcount, refcount)
4: for i = 0 to max_refcount − 1 do
5:    if i < refcount then
6:       local_offset ← atomic_increment(&hist[i], 1)
7:       refptr ← in_queue[id] + i + HEADER_SIZE
8:       out_queue[global_offset + local_offset] ← refptr
9:    end if
10:   (memory barrier)
11:   global_offset+=hist[i]
12: end for
```

It is important to note that the *mark* operation does not need to use an atomic operation. Setting the mark bit is an idempotent operation and there is no correctness concern if a single object is processed multiple times. The slight performance loss due to redundant work – if an unmarked object gets added to the frontier multiple times and processed within the same iteration – is vastly outweighed by the cost of atomic operations[5].

The described version of the algorithm performs no coordination between workgroups and can thus only exploit one compute unit per device. In Section 6.7.4 we expand on it and discuss load balancing and synchronization concerns in detail. We present a naïve proof-of-concept solution in Section 6.5.5.

6.5.1 Offset Calculation

Our first strategy for calculating the offsets for the output queue used Blelloch's prefix-sum algorithm from [93]. With this approach, all references of an object are stored in consecutive slots in the queue, and the offset of an object's first reference is the total number of references from work-items with a lower *id* than the work item processing that object (Figure 6.8). When performed in local memory, the complexity of this approach is $O(\log n)$ parallel addition and local memory operations.

However, when implementing this strategy, we discovered that the approach often takes up 40-50% of the kernel's total execution time, arguably due to the large number of accesses to local memory. We therefore implemented a different strategy, based on a histogram. In this approach, the first references from all work-items with at least one reference are copied first, followed by the second references, third references, etc.

[5]On past AMD hardware, atomic operations were up to 5× slower than normal accesses, because they use the *complete path* vs the *fast path* for memory access [8] In the compiler we had available in our version of the SDK (AMD APP SDK v2.6), using any atomic operation on global memory causes *all* global memory accesses to use the complete path. In practice, this has led us to avoid atomic operations wherever possible.

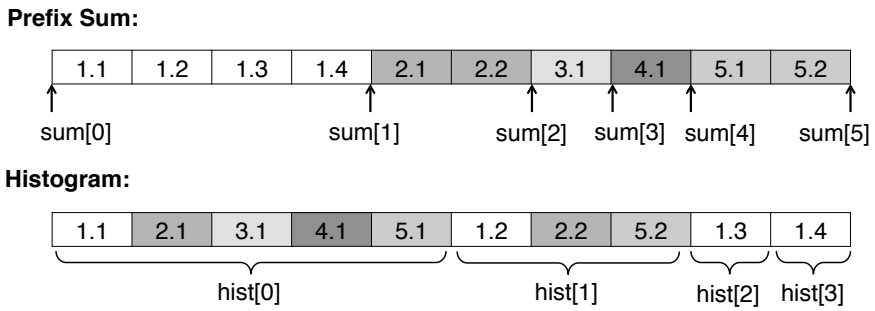


Figure 6.8: Comparing our two different approaches for calculating offsets. $m.n$ describes the n'th reference of the m'th object (i.e. work-item).

Offsets for this layout are calculated by generating a histogram that counts the number of work-items that have at least one reference, at least two references, etc. The histogram is generated using atomic operations in local memory, by atomically counting the number of work-items with an object having $i = 1, 2, \ldots$ references or more. The sum of the first $i - 1$ entries of the histogram gives the global offset of the part of the output array where the i'th references begin (Figure 6.8). The atomic counting operation also gives each work-item a unique local offset into the i'th part, where it will write its reference. The following code replaces the last part of Algorithm 1, starting from line 13.

This uses $O(max\_refcount)$ parallel atomic operations, where *max_refcount* is the maximum number of references among the objects currently processed by any of the work-items. Since the atomics are executed in local memory, they do not slow down global memory accesses and are comparatively fast. An additional advantage of this approach is that we are writing to consecutive items in the queue, which is efficient on our hardware.

6.5.2 Reducing Divergence

As discussed in Section 6.3, the majority of objects have a small out-degree (i.e., a small number of references), while a few objects have a large numbers of references. In the algorithm above, each work-item loops over all references within the object it handles. This behavior is problematic for SIMT execution: when one work-item encounters a high-degree node, the remainder of the workgroup will stall until that work-item has completed its task. This results in low utilization of available parallelism and wastes available memory bandwidth.

To avoid this case, we extended our algorithm to let each work-item process at most a fixed number of references for each object (currently 16, based on our study in Section 6.3). This minimizes the worst-case divergence in the loop. Objects that are longer than this are then stored on a non-blocking stack and (in the same iteration of the algorithm) processed in parallel. This is done by letting all work-items process one reference each in parallel, a very efficient way to perform a large copy operation.

Like Veldema and Philippsen, we consider this an essential optimization. Our approach bears resemblance to theirs, but processes large arrays (and objects) immediately and does not require a new kernel launch.

6.5.3 Vectorized Memory Accesses

We explored the possibility of using vector reads to decrease the number of individual memory requests. OpenCL supports 4-wide vector types, which allow reading 128 bits at a time. We rewrote our algorithm to use vector loads to access the header and the first three references at the same time (and then read references in groups of four). This made it necessary to lay out the objects in such a way that headers are aligned to 128 bit boundaries, which we achieved by introducing additional padding to our reference graph.

Vector reads can lead to performing extra work, as the algorithm may read more references into local memory than necessary. Overall, however, we expected a speed-up due to the reduced number of memory requests.

6.5.4 Cut-off for Long, Narrow Heaps

As we show in our heap analysis (Section 6.3), some workloads exhibit very long narrow tails (e.g., due to linked lists). From a performance perspective, it is beneficial to detect such cases and return execution to the CPU. We believe that without the ability to saturate memory with requests, the GPU loses out to the CPU due to the CPU's much lower average memory latency as a result of caching. The CPU benefits from any spatial locality of the memory graph that may exist, whereas the GPU does not. The CPU also benefits from the fact that the (very small) active section of the queue ends up in L1 cache.

We therefore implemented a mechanism that returns execution to the CPU once the size of the queue drops below a certain threshold. As a safeguard, we require a minimum number of iterations on the GPU to complete before returning.

Veldama and Philippsen identified a similar optimization, but in a different context: their discussion focuses on avoiding context switches to and from the GPU. To handle linked lists, their algorithm runs multiple iterations on the GPU without switching to the CPU. This optimization does not apply to our approach.

6.5.5 Multiple Compute Units

To achieve high throughput, it is desirable to leverage all of the GPU's compute units. For the purposes of our evaluation, we chose a naïve proof-of-concept approach to run the algorithm on the two compute units that our platform provides. We first divide the root set into two halves and hand one of them to each compute unit. Each compute unit then runs the algorithm independently, without any load balancing or synchronization. This approach has two drawbacks:

- If the initial partitioning results in an uneven distribution of work, one compute unit may be idle for most of the execution.
- We may perform redundant work in cases where the two compute units race to mark an object and retrace the same part of the heap.

While this results in a negative performance impact, our approach is nonetheless correct: marking a node is an idempotent operation and can be performed multiple times without harm. Better results can be achieved by using dynamic load balancing between compute units – we discuss this aspect in Section 6.7.4.

6.6 Evaluation

In this section, we present the results of experiments we ran to evaluate the performance of our mark algorithm. We first describe our evaluation platform and then use microbenchmarks to highlight strengths and weaknesses of our algorithm and collector implementation. We then examine the performance of our implementation on real-world application benchmarks from the DaCapo 9.12 benchmark suite. We conclude with a brief discussion of additional overheads that were excluded from the previous subsections.

6.6.1 Test Platform

Our test platform was an AMD E-350 APU, which was one of the first chips that integrate a CPU and GPU into a single device (Intel's Sandy Bridge architecture has a similar integrated GPU, but it is not programmable). The system was configured with 3.5 GB of DDR3 1066 RAM. The APU's "Bobcat" CPU is a dual core running at 1.6 GHz with a 512 KB L2 cache [9]. Its "Brazos" GPU is running at 492 MHz with 2 compute units, 16 stream processors, an 8 KB L1 cache per compute unit, and a 64 KB L2 cache per GPU [8]. Measurements show that the caches are disabled for accesses to local and shared memory. The CPU and GPU share memory and a single memory controller on which they compete for bandwidth; we experimentally determined that the GPU can only map 128 MB in any given kernel invocation. All experiments were conducted on Fedora Linux (kernel version 2.6.35.14-103).

6.6.2 Microbenchmark Results

To explain the performance of the baseline algorithm and explore potential optimizations, we used a set of simple microbenchmarks. These benchmarks were handwritten and do not run through the Jikes environment. This approach was chosen to get pure forms of the heap graphs, since even a small Java program creates enough internal objects to obscure the microbenchmark results. Table 6.1 presents the execution times of the microbenchmarks for a set of different configurations of the garbage collector.

	Size	CPU	GPU	GPU+P	GPU+V	GPU+F	GPU+D	GPU+VD	GPU+2CU
Single Item	28 bytes	0.001	0.027	0.027	0.028	0.028	0.027	0.028	0.028
Long Linked List	156 KB	0.151	94.604	74.400	83.451	0.360	96.279	85.412	84.173
256 Linked Lists	39 MB	129.723	140.465	192.823	118.982	140.783	142.556	120.984	102.092
2560 Linked Lists	117 MB	1074.920	415.553	572.153	350.523	416.389	421.589	356.986	194.317
Very Wide Object	3.92 KB	0.018	1.862	1.077	1.696	1.861	0.218	0.221	0.220
VP Linked List	4 MB	0.150	77.462	60.950	68.123	0.317	78.757	69.802	68.843
VP Arrays	20 MB	1.347	5.361	6.308	3.516	5.378	5.843	3.095	1.693

Table 6.1: *Average mark times for microbenchmarks.* All times are in ms and do not include overheads. CPU is a baseline CPU implementation. GPU is our baseline algorithm using the histogram approach. GPU+P is the variant using prefix-sum. GPU+V is the vectorization of that algorithm. GPU+F falls back to the CPU if a narrow tail is encountered. GPU+D has special support for large objects to prevent divergence. GPU+VD combines vector and divergence. GPU+2CU enables both compute units for the GPU+VD configuration.

	Jikes MS	Serial CPU	Baseline GPU	Optimized GPU	Opt GPU + 2CU	GPU Slowdown	Opt Speedup
lusearch	1566.10	1084.18	11739.50	2404.18	1490.96	1.38	7.69
pmd	211.98	356.09	1651.66	634.24	357.49	1.69	4.55
sunflow	1422.54	401.36	3446.52	724.92	554.25	1.38	6.25
xalan	809.79	423.31	2836.33	1088.78	750.12	1.77	3.85

Table 6.2: *Cumulative mark times for DaCapo benchmarks (default sizes).* All times are in ms and do not include overheads. "Optimized GPU" includes all optimizations except using multiple compute units.

Methodology We ran our microbenchmarks on the following configurations: *CPU* is our implementation of a serial single CPU mark phase. *GPU* is the baseline algorithm described previously. *GPU+V* is the vectorization of that algorithm (Section 6.5.3). *GPU+D* is the variant with special support for large objects to prevent divergence (Section 6.5.2). *GPU+F* is a variant which falls back to the CPU once the queue length drops below a threshold and a minimum number of iterations have run (Section 6.5.4); we use 20 as the threshold and 5 as the minimum number of iterations. We report the sum of the GPU and CPU runtime. *GPU+P* uses the prefix-sum approach instead of the histogram (Section 6.5.1), for comparison. *GPU+2CU* contains the first two optimizations but also uses both compute units. Each configuration was run for 20 iterations and the average runtime is reported; variation between runs was extremely low.

Benchmark Descriptions For each benchmark, we also provide the overall size of the reference graph that is associated with it:

- **Single Item** (28 bytes) - This benchmark consists of a single item in the heap, with a corresponding pointer in the root set. The purpose of this benchmark is to measure the overhead (excluding copy overhead) of the algorithm. As would be expected, the startup cost for the GPU variants are similar. The CPU is an order of magnitude faster since the data is already in cache.
- **Long Linked List** (156 KB) - This benchmark consists of a single long linked list with 10,000 elements. This case is the worst for the GPU since it cannot exploit any parallelism in the graph. All of the GPU implementations perform badly, but the one with the option to fall back to the CPU fares best. It runs the minimum number of iterations on the GPU, then returns to the CPU for the majority of the execution. Unfortunately, the few iterations it does run on the GPU prove quite expensive.
- **256 Parallel Linked Lists** (39 MB) - This benchmark consists of 256 parallel linked lists with 10,000 elements each. The root set contains a pointer to each linked list. The effect of this is that each work-item within the workgroup can operate independently, which allows the GPU implementations to perform relatively well, some even beating the CPU by a small amount.
- **2560 Parallel Linked Lists** (117 MB) - This benchmark extends the previous experiment by adding more linked lists. Due to our hardware's limited amount of mappable memory, we shortened each list to 3,000 elements each. This case can arguably be seen as the best for the GPU since there is abundant parallelism and little locality between objects in the queue. This microbenchmark is the only one where the GPU solidly outperforms the CPU.
- **Very Wide Object** (3.92 KB) - This benchmark consists of a single array containing 1,000 individual objects. This is an extreme case designed to illustrate the effects when SIMT divergence is not addressed.

Discussion These benchmarks allow us to evaluate the impact of the optimizations discussed in the previous section.

- **Histogram** (Section 6.5.1) - Comparing the GPU and GPU+P results shows the difference in performance characteristics of the histogram and prefix styles of offset calculations. The prefix sum implementation performs well for cases in which a small subset of the work-items perform useful work, while the histogram fares better when many work-items are active. On real-world benchmarks (not presented), the histogram is clearly better, but it may be worth exploring a combination of both approaches (e.g. by switching dynamically between them).
- **Divergence Handling** (Section 6.5.2) - This causes a slight slowdown for those benchmarks that do not contain objects with large numbers of references. For benchmarks that do (such as *Very Wide Object* above), the performance improvement is substantial (a 89% improvement). For real workloads, we believe divergence handling to be a critical and necessary optimization.
- **Vectorization of Loads** (Section 6.5.3) - This optimization shows an improvement on most of the microbenchmarks we report. The improvements range from 12% to 40% for all benchmarks except the *Single Item* case. This case is hinting at a more general problem which is that vectorization can (and does) hurt performance in some cases: if the vectorization causes memory words to be read that are not used, and if memory bandwidth is already running at the hardware limit, vectorization can slow down the algorithm. However, from our experiences, this seems to be a rare case.
- **Falling back to the CPU for narrow tails** (Section 6.5.4) - Our implementation of fallback has a barely perceptible negative impact on performance for most benchmarks. However, for cases where the GPU would perform extremely poorly (such as the *Long Linked List* microbenchmark), it recovers some, but not all, of the performance lost. There is still a significant amount of time spent on the GPU to handle narrow sections before the cut-off is invoked; we believe this to be a necessary evil to prevent temporary drops in parallelism from triggering overly eager fall-back.
- **Multiple Compute Units** (Section 6.5.5) - Despite our naïve approach, we still obtain perceptible improvements by using both compute units. It is important to note that this improvement is not guaranteed: using a second compute unit can hurt performance if the first unit would otherwise get additional bandwidth and the second unit is performing only redundant work.

Comparison with related work The last two benchmarks are modeled closely after those presented by Veldema and Philippsen for evaluating their GPU mark algorithm. Unfortunately, the results are not directly comparable due to different experimental setups. We would like to note that their results were collected on a significantly more powerful GPU. Nonetheless, our mark algorithm appears to fare well in comparison.

- **VP Linked List** (4MB) - This benchmark consists of 16 linked lists of 8,192 element each, of which all but one is immediately garbage. Only one of the linked lists is traced by the mark phase. As a result, this is structurally very similar to the *Long Linked List* benchmark above.

- **VP Arrays of Objects** (20MB) - This benchmark consists of 1,024 arrays, each containing exactly 1,024 objects. Only the first 64 arrays are retained. All others immediately become garbage and are not traced.

It should be noted that we do not report launch overheads, while Veldema and Philippsen report complete execution times. Furthermore, they perform 8 collector runs while we only measure one execution. As we do not compare our numbers directly to their results, these differences do not affect our conclusions.

6.6.3 DaCapo Benchmarks

We measured the performance of our GPU-based collector for real-world application benchmarks from the DaCapo 9.12 benchmark suite. The results are shown in Table 6.2.

Methodology In these results, the *Optimized GPU* implementation includes the vectorization and divergence handling optimizations discussed previously. Both the optimized and unoptimized results use the histogram method for offset calculation. Neither version includes the long tail cutoff, to avoid the issue of confusing what is actually running on the GPU. The *Jikes MS* column is an unmodified instance of Jikes' `MarkSweep` collector. The second column is a trivial CPU implementation which operates on the reference graph and runs outside Jikes's Java environment. We present these numbers to offset any minor locality advantage the reference graph structure may give us. The final two columns present the slowdown of the GPU over the best of the two CPU implementations and the improvement resulting from optimization of the GPU algorithm respectively.

The Jikes RVM was configured with a maximum heap size of 192 MB - the largest we could map on the GPU even with the reference graph. We do not report collection times for `avrora` or `luindex` since neither consistently triggers a collection at the heap size we are using. All results were generated running the benchmarks with their default configurations and using the "converge" (`-C`) option provided by the suite. We report the cumulative time of all garbage collections conducted during the final iteration.

Discussion As can be seen from the results in Table 6.2, our GPU mark implementation is within a factor of two for all of the benchmarks we report. As a reminder to the reader, we are conservatively comparing against the better of Jikes' `MarkSweep` and our own CPU implementation working off the reference graph. When comparing only against the `MarkSweep` collector, our implementation fares significantly better; the GPU outperforms Jikes on 3 of 4 benchmarks. We consider this to be a highly encouraging result.

We would like to note that these performance results are extremely sensitive to the heap size. As the heap size increased, the relative performance of our GPU implementation to Jikes increased sharply. We present the largest heap sizes supported by our evaluation platform, but even those are small for real program heaps. We suspect that relative performance would continue to improve as the heap size increases.

6.6.4 Overheads of Our Implementation

In the preceding discussion, we excluded the copy overhead and kernel launch overhead for any of the GPU configurations; we report kernel execution only (for Jikes, we only report the mark phase). Our reference graph implementation adds some additional overhead outside the mark phase. Allocating each object requires that a corresponding reference graph node be allocated as well; this introduced mutator overhead of approximately 40% in an allocation stress test microbenchmark. This overhead is less pronounced in the DaCapo results, but is still significant, varying between 7% and 25%. It should be mentioned that this overhead could presumably be reduced by adding this functionality through the compiler, rather than adding an extra function call.

Using the basic approach of filling in the entire reference graph before every collection adds a major overhead to each collection, taking several times as long as the mark phase on the CPU, arguably due to a highly untuned implementation. The double-write approach eliminates this at the cost of an additional 11% runtime overhead in the microbenchmark, for a cumulative total of 57%.

Some collector overhead is also added in copying markings from the reference graph back to the heap in preparation for running an unmodified Jikes sweep phase.

Let us emphasize that all overheads discussed in this subsection are artifacts of either the copying of data to the GPU (Section 6.4.4) or our need to reduce the size of the space being collected (Section 6.4.3). Neither is intrinsic to the problem and both are likely to be less pronounced on modern hardware.

6.7 Discussion

While our numbers imply that our GPU-based garbage collector is a factor of 1.4–2× slower than our CPU-based collector and therefore not directly competitive in terms of performance, our experimental results nonetheless answer the questions we set out to investigate. We identified the key points for offloading garbage collection to the GPU, some of which are surprising in hindsight. We were also able to assess the suitability of today's GPUs for garbage collection. But most importantly, we confirmed the feasibility of offloading GC to data-parallel accelerators in general, and identified the bottlenecks that could be improved either by future GPU generations or through custom hardware. It is these insights that lead to the custom hardware accelerator for garbage collection presented in the next chapter.

6.7.1 Lessons from the Mark Algorithm

Somewhat counter-intuitively, the primary goal for garbage collection on the GPU is not to parallelize the computational steps of the algorithm but to maximize the hardware's ability to schedule memory requests. As we saw in Section 6.3, the degree of theoretical parallelism that is available throughout most of the mark phase ranges from 100 to 100,000-fold. This degree is too high for a single CPU to take effective advantage of this parallelism, and is therefore much better suited for data-parallel architectures.

On the GPU, the key to achieving high memory bandwidth is to ensure that each work-item can effectively generate and handle memory requests every cycle. It is therefore crucially important to reduce divergence between threads (Section 6.5.2), but also to avoid serialization of execution (as ensured by our queue approach). The numbers presented in Section 6.3 confirm that heaps typically contain rare but long objects and arrays which cause this divergence. This led to the insight that an effective architecture for GC needs to be able to decouple processing of long objects, since such an object can otherwise block other memory requests from executing. We show our solution to this problem in Section 7.5.1.

In our algorithm, the number of outstanding requests is limited by the maximum size of a workgroup (which is limited by the hardware). Notably, this is different from the number of streaming processors in the GPU: while the number of streaming processors limits the throughput in terms of instructions per cycle, the workgroup size limits the number of work-items that can be in-flight at a given time, and therefore the number of outstanding memory requests that can be issued. As we saw in this work, keeping as many outstanding memory requests in-flight as possible is crucial for the performance of a garbage collector. This led to the insight that the best architecture for garbage collection would be one that can keep an arbitrary number of memory requests in flight at a low cost. While a GPU is suitable for this type of workload, the relatively large amount of state per SIMT thread still puts a substantial limit on the number of simultaneously active work items.

As with many GPU algorithms, it is only feasible to run the mark algorithm on the GPU if the number of objects to mark is sufficiently large. For small collections, the launch overhead dominates the entire collection time, in which case it is beneficial to run the collection on the CPU in the first place. Predicting the size of a collection is non-trivial, but it may be possible to apply heuristics, similar to Garbage-First collectors [66].

6.7.2 Lessons from the Reference Graph

Our reference-graph approach is somewhat orthogonal to the problem we are trying to solve: we hypothesize that modern GPUs could instead map the entire heap's address space, perhaps even cache-coherently, with the CPU. Surprisingly, we noticed that the reference graph gave significantly better performance for a CPU collector: our untuned CPU collector beat the optimized Jikes collector on several occasions, arguably due to increased cache locality. We believe that this approach could be exploited by specialized CPUs: While we did not explore

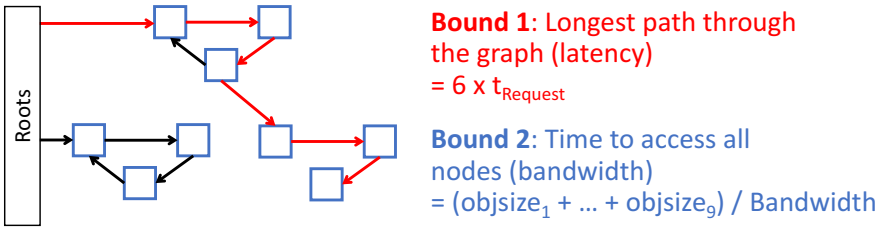


Figure 6.9: *Two independent ways of constructing a lower bound on GC latency.* The minimum execution time is bound by the time it takes to access every reachable element of the graph, and the latency of traversing the maximum shortest path to any object.

this direction further, we briefly want to discuss the performance trade-offs one would face when using this approach. These trade-off are not specific to GPUs.

Our numbers from Section 6.6.4 indicate that keeping the reference graph up-to-date when running the mutator seems to be the most promising approach. We believe that modifying the compiler to issue duplicate writes whenever a reference is written will lead to a significantly lower performance impact than we are incurring with our naïve, write-barrier based approach (which incurs a function call for every write). An alternative approach consists of splitting each object into two parts, one only containing the references, the other containing the non-references. This avoids the need for duplicating data and substantially improves collector locality, at the cost of access locality.

The latter approach might be particularly promising for specialized CPUs: It would allow us to decouple reference accesses and non-reference computation in a similar way to a decoupled access/execute architecture [203], potentially eliminating the performance overhead of read and write barriers, and improving locality and caching. We did not investigate this approach further but found it worth mentioning as we found the trade-offs intriguing.

6.7.3 Estimation of Performance Limits

To estimate the potential of our approach, we need to quantify how the performance of our implementation compares to the theoretical best case on the given hardware. To do this, we present two weak, but independent, constructions of a lower bound on execution time for the *2,560 Parallel Linked List* benchmark from Section 6.6. Following this, we discuss performance measurements that lead us to believe that the actual bound is even tighter.

Figure 6.9 shows these bounds. The first bound can be constructed by examining the minimum time required to touch every memory location in the reference graph exactly once. As constructed, the reference graph contains only the edges in the heap graph and some minor padding. While there may be a more compact representation, we believe that this is a reasonable first-order approximation for a minimum-size representation of the heap graph. Using only the size of the benchmark (117 MB) and the peak memory bandwidth for our device (9 GB/s), we can establish a lower bound for GPU execution of ∼12.7 ms.

For the second bound, we can consider the minimum number of dependent loads from main memory and the stall latency implied by each. Without the presence of caches, each step of the list traversal requires at least one round trip to main memory. As a result, a lower bound on the run-time of the algorithm is given by $depth \times stall\_penalty\_in\_cycles \times 1/gpu\_frequency$. We benchmarked a stall latency of 256 cycles under load and the benchmark requires an absolute minimum of 3,000 dependent loads (one per linked-list element). Taken together, this gives us a lower bound of $\sim$ 1.5 ms. For this benchmark, the bound is not particularly tight, but we present it nonetheless since it reflects structural features of the heap graph that cannot be avoided (Section 6.3). Together, these two approaches gives us a bound that is about 15$\times$ better than our best measured performance on the GPU.

We also examined the sustained memory bandwidth achieved by our implementation over an entire execution of the mark phase and compare it against the peak memory bandwidth available on the device. For our benchmark, the optimized dual compute unit configuration achieves a sustained bandwidth of 3.016 GB/s, or roughly one third of peak. As expected, the single compute unit version of the same code achieves roughly 1/2 of the bandwidth at 1.72 GB/s. It is worth noting that these are measurements of our actual implementation and thus may not reflect an actual bound due to errors in the implementation or missed optimizations. As an illustrative example, disabling the vectorization and divergence handling for the dual compute unit code gives a higher sustained bandwidth (4.317 GB/s), but lower overall performance. (We believe this to be due to the fact that the native memory request size is 2 words. Reading the two words separately can result in separate requests being issued in some cases and artificially inflate bandwidth.) An additional caution is that the profiler is known by the vendor to provide unreliable results under some circumstances.[6]

Taking these points together, we believe our algorithm to be within around 3$\times$ the optimal performance on our hardware. However, these results show that in order to achieve maximum GC performance, it is crucial to make maximum use of the available memory bandwidth. Our GPU's 9 GB/s is small compared to the 50 GB/s that are available on modern server processors, even without more than 2 memory channels. High performance GCs therefore need to be able to saturate a very large amount of memory bandwidth. In Chapter 7, we show how custom hardware can help achieve this goal.

6.7.4 Load Balancing on Multiple Compute Units

As explained in Section 6.5.5, our current implementation statically distributes the load between the two compute units on the device. We believe that static load balancing will not suffice for a real implementation (or even our own implementation on a device with more compute units). Given modern GPU parts have a 1-2 orders of magnitude larger number of compute units, this is an urgent concern. With this in mind, we experimented with a number of options for synchronization between compute units.

[6] As noted in the Developer Release Notes for AMD APP SDK v2.6.

There are two fundamental approaches: pre-partitioning the graph between compute units before the start of the computation, or synchronizing between compute units. Graph traversal is a highly irregular computation. The analogy of pre-partitioning and re-partitioning for graph traversal is to stop the GPU kernel after regular intervals, have the CPU inspects all queues, load balance if necessary, and then relaunch the kernels. This is related to the option chosen by Veldema and Philippsen [220]. As they showed, it can be used effectively, but incurs significant overhead since kernel launch and termination are expensive synchronization actions (see Section 6.4.4 for discussion of launch overheads). Additionally, this solution would interfere with our goal of leaving the CPU available for other processing. Potential alternatives include:

- Using global atomics to synchronize through shared memory. As discussed previously and documented by Elteir et al. [69], global operations used to be prohibitively expensive on AMD hardware, but this has likely improved since 2012.
- Using on device hardware counters to construct a fast software lock. This seemed like a possible workaround on our test platform, but after a trial implementation, we were forced to conclude that the counters were not appropriate for our goals.
- Having each compute unit copy content from the other compute unit's queue into its own if its queue length dropped below the number of work items per unit (i.e., work stealing). This scheme does not use any form of synchronization and thus can not update the source queue safely. As a result, redundant work can and will be performed. From preliminary results, it appears that the overhead caused by the inspection outweighs any benefit provided by the load balancing. We did not explore this idea further.

Based on our investigation, the only dynamic load balancing scheme that seemed viable at the time was to use the CPU for coordination as suggested by Veldema and Philippsen [220]. However, this has changed on modern hardware: Abhinav and Nasre [3] showed four years later that on a modern Intel i7-3610QM, the work-stealing approach enables scaling to 16 compute units (and potentially beyond).

A corollary from being able to scale to multiple compute units is that this enables proportionality of garbage collection and memory allocation rates, by scaling GC performance continuously based on memory pressure. This makes it possible to reduce the use of GPU resources (and therefore energy consumption) when GC is a lower priority, and increase GPU resources as memory pressure grows. The same idea can be applied to custom GC hardware, by implementing multiple GC acceleration units, or enable throttling for existing ones.

6.7.5 Assessment of Garbage Collection on GPUs

Our results show that it is possible, with a large amount of overhead, to build a GPU-based garbage collector on current hardware. The numbers from the microbenchmarks show that an optimized GPU mark algorithm can, for the best case, significantly outperform a mark

algorithm running on the CPU. However, our results for the DaCapo benchmarks show that the mark phase for real-world benchmarks is a factor of 1.4-2× slower than on the CPU (but sometimes outperforms Jikes's `MarkSweep` collector).

Many of the performance limitations were a result of the hardware that we were using, and integrated GPUs have come a long way since 2012. Specifically, our approach was limited by (1) only being able to map small amounts of memory, and only with large start-up overheads, (2) lack of synchronization between compute units and (3) a limited amount of memory bandwidth and compute resources available on the GPU.

In 2016, Abhinav and Nasre [3] repeated and extended our study with OpenJDK 8 running on an Intel i7-3610QM SoC with an integrated Mobile HD 4000 GPU. This part has 16 compute units, low zero-copy launch overheads and enables synchronization between the different compute units. Using this part, they were able to achieve a 4-5× speed-up of the GPU-based garbage collector relative to HotSpot's parallel collector running on the CPU. While this approach still relied on the reference graph, it demonstrates that a real-world GPU-based collector is much closer to reality at this point.

6.8 Related Work

The idea of performing garbage collection on the GPU existed prior to our work. Jiva and Frost describe the basic approach in a patent application in 2010 [120], while Sun and Ricci [206] describe the idea as part of a larger vision of using GPUs to speed-up a variety of traditional operating system tasks. However, to our knowledge, none of them has published an appropriate algorithm or publicly disclosed a working GPU-based collector.

While the work by Veldema and Philippsen [220] explored the implementation of a mark & sweep garbage collector on the GPU, their work differs from ours in a number of important points. First and foremost, their goal was not to use the GPU to accelerate garbage collection for programs running on the CPU, but to provide garbage collection facilities for CUDA-like programs written in a Java dialect and running on the GPU. Additionally, our mark algorithm bears little resemblance to theirs.

Prior to our work, there had been a few recent proposing potential non-numeric applications for GPUs. Naghmouchi et al. [161] investigated using GPUs for regular expression matching. Smith et al. [204] evaluated GPUs as a platform for network packet inspection.

By 2012, several groups had already investigated algorithms for performing breadth-first-search on a GPU. One of the first publications in this space was the work by Harish and Narayanan [92], who presented the first algorithm to perform an efficient breadth-first-search on the GPU. However, this approach was based on visiting every node at every iteration, and was less efficient than the most efficient CPU implementation at that time. Their approach was improved by Luo et al. [146] who used an approach based on hierarchical queues to achieve better performance. Work by Hong et al. [105] improved the performance even further. Veldema and Philippsen's [220] approach resembles the work by Harish and Narayanan [92], whereas ours takes the approach of Luo et al. [146] and Hong et al. [105]

Another body of work that is related to garbage collection on the GPU describes the use of other parallel architectures or heterogeneous platforms to perform garbage collection. An example for this is the work by Cher and Gschwind which demonstrates how to use the Cell processor to accelerate garbage collection [48]. Barabash and Petrank cover the problem of garbage collection on highly parallel platforms from a more general perspective and perform a heap analysis similar to ours [26]. An early paper by Appel and Bendiksen [13] covers GC on vector processors and our approach has been influenced by some of their ideas.

6.9 Summary

By implementing garbage collection on the GPU, we showed that there is potential for offloading garbage collection to data-parallel accelerators. While we believe that this work stands on its own, it demonstrates important lessons for taking the next step and implementing a fully custom hardware accelerator for garbage collection.

Specifically, our study of heap graphs from the DaCapo benchmark suite shows that there were no structural features that would prevent effective parallelization, and that heap graphs are fairly regular and therefore lend themselves to specialization. By implementing a GPU-based garbage collector, we showed that even existing data-parallel accelerators can perform GC reliably within 2× the performance of a CPU. However, most importantly, we learned important lesson about making garbage collection efficient on data-parallel architectures and integrating it into a full system design:

- While garbage collection is compute-intensive on traditional platforms, its efficiency is dominated by being able to keep as many memory-requests in-flight as possible. On traditional CPUs (and GPUs), this requires being able to issue a large number of memory operations – data-parallel architectures excel at this type of task.
- While a bidirectional memory layout has limited impact on a CPU, it is crucial for making garbage collectors performance-efficient on architectures without caches. This is especially important in accelerators, where area and power are crucial metrics.
- Divergence is a problem for the mark phase of a garbage collector, and much of it is the result of imbalance in the copying of outgoing references. This led to the insight that decoupling the copying of outgoing references from marking headers can lead to higher throughput (similar to our divergence optimization from Section 6.5.2).

While these insights are important on GPUs, this work led to the realization that all of them could be exploited much more efficiently by building custom hardware for garbage collection, which is integrated into the SoC and performs GC for the CPU. In the next chapter, we will further explore this direction and show that this approach can lead to a design that is much more efficient than either a CPU or a GPU.

Chapter 7

Hardware Support for Pause-Free GC

In this chapter, we take some of the insights we gained from offloading garbage collection to GPUs and distill them into a design for a custom accelerator that performs garbage collection in hardware. We argue that such an accelerator will be able to perform garbage collection more efficiently than a CPU or GPU, at a much lower die-area cost.

7.1 Why Hardware Support for Garbage Collection?

Garbage Collection research over the past 50 years has led to several fundamental improvements in collector designs (such as generational collectors) and GC design points that enable new use cases (such as concurrent real-time collectors). However, as we describe in Section 2.3.1, all garbage collectors have to make fundamental trade-offs between application throughput, pause times and memory utilization. Arguably, we still have not found what has been called the "Holy Grail" of garbage collection [159]: a pause-free collector that achieves high memory utilization and high GC throughput (i.e., sustaining high allocation rates), without a large resource cost for the application.

Many recent GC improvements have focused on pause times and GC throughput. As a result, modern collectors can be made effectively pause-free at the cost of slowing down application threads and using a substantial amount of resources. Moreover, many approaches ignore another factor that is very important in data centers: energy consumption. Previous work [43] has shown that GC can account for up to 25% of energy and 38% of execution time in common workloads (10% on average). Worse, as big data systems are processing ever larger heaps, these numbers will likely increase.

What all widely used approaches today have in common is that they only investigate the GC problem at the software level. This fundamentally limits the ability to reconcile pause times and application performance. In particular, the fundamental work that the garbage collector has to perform does not become cheaper, but is instead shifted from garbage collector threads into application threads.

We believe that we can reconcile low pause times and application performance by revisiting the old idea of moving GC into hardware, and co-designing the hardware and software layers. Applying the ideas we learned from offloading GC to GPUs, we propose the design of a custom accelerator that can be integrated into server SoCs and performs the GC operations at a much lower area and energy cost than either a CPU or a GPU.

Our goal is to build a collector that simultaneously achieves high GC throughput, good memory utilization, pause times indistinguishable from LLC misses and energy efficiency. It is possible to design a GC algorithm that performs well on the first three criteria but is resource-intensive (e.g., Azul's Pauseless GC [53]). Our key insight is that such an algorithm can be made energy efficient as well by moving its work-intensive phases into hardware, combined with several software-level and algorithmic changes.

We are not the first to propose hardware support for GC [24, 53, 121, 157, 232, 234]. However, none of these schemes has been widely adopted. We believe there are three reasons:

1. Garbage-collected languages are widely used (due to their productivity and safety properties), but they are rarely the only workload on a system. Systems designed for specific languages mostly lost out to general-purpose cores, partly due to Moore's law and economies of scale allowing these cores to quickly outperform the specialized ones. This is changing, as the end of Moore's law makes it more attractive to use chip area for accelerators that improve common workloads – such as garbage-collected applications.

2. Most garbage-collected workloads run on servers (note that there are exceptions, such as Android applications). Servers traditionally use commodity CPUs and the bar for adding hardware-support into such a chip is very high. However, this is changing: cloud hardware and rack-scale machines in general are expected to switch to custom SoCs and FPGAs, which could easily incorporate IP to improve GC performance and efficiency.

3. Many Hardware GC proposals were very invasive and would require re-architecting of the memory system or other components [121, 230, 232, 234]. We believe an approach has to be relatively non-invasive to be adopted. The current trend to accelerators and processing near memory may make it easier to adopt similar techniques for GC without substantial modifications to the architecture. Furthermore, Mobile SoCs have set a precedent for assembling systems from non-invasive IP components and accelerators.

We therefore think that the time is ripe to revisit hardware-assisted GC. In contrast to many previous schemes, we focus on making our design sufficiently non-invasive to incorporate into a server or mobile SoC. This requires isolating the GC logic into a small number of IP blocks and limiting changes outside these blocks to a minimum.

Our design exploits two insights: First, overheads of concurrent GCs stem from a large number of small but frequent slow-downs due to read and write barriers spread throughout the execution of the program. We propose moving the culprits into hardware to alleviate

their impact and allow out-of-order cores to speculate over them. Second, the most resource-intensive phases of a GC (marking and reclamation) are a poor fit for general-purpose CPUs. We move them into accelerators close to DRAM, to save power and area.

In this chapter, we first describe our proposal in the most general form, and present the design space associated with it. To evaluate trade-offs, we then focus on the accelerator portion and present an end-to-end RTL prototype of a specific incarnation of this design, integrated into a RocketChip RISC-V SoC running full Java benchmarks with JikesRVM on top of Linux on FPGAs (Chapter 9). To enable this work, we had to build a new research platform and evaluation methodology, which we present in Chapter 8.

7.2 The Hardware GC Design Space

We start out by describing the general design space of hardware-assisted garbage collection. The trade-offs span both software and hardware components, and we **highlight** these different design decisions throughout the text.

Recalling Section 2.3, tracing garbage collectors have to periodically perform two operations. First, they have to perform a *mark* pass through the heap to identify all objects that are reachable from a given set of roots. This phase typically performs a breadth-first search (BFS) across the object graph and sets the mark bits within all objects that it encounters. The second operation is a *reclamation* step where all objects that are not reachable are recycled, either through compaction (i.e., packing live objects into a new space) or sweeping (i.e., keeping live objects in place and creating a free list to link the newly freed memory cells together). The first option is *relocating*, as it moves objects in memory, the second is not.

Both phases can be executed while stopping the application threads, or run concurrently with the application and with each other[1]. If the phases run concurrently with the application threads (mutators), barrier code needs to be injected into the instruction stream to keep the collector and mutators in sync (Figure 7.1). This leads to the first design trade-off we need to consider: whether or not to design a **relocating or a non-relocating collector**.

Relocating collectors achieve shorter allocation latencies, more locality and incur less fragmentation, which is important in server workloads. However, they either require forwarding pointers or additional barriers. For a non-relocating collector, only a write-barrier is required to inform the collector when a mutator hides an object from it. A relocating collector also needs to either follow a level of indirection or inject read barriers, which check on every reference access that the object has not moved and look up the new location if it has. Since read barriers are on the critical path of a reference load from memory, they are more expensive than write barriers.

Among read and write barriers, there is a wide design space that trades off **fast-path latency** (e.g., if a read barrier's object has not moved), **slow-path latency** if the barrier gets triggered (e.g., if the object has moved), the **instruction footprint** and how it maps

[1]To run marking and reclamation in parallel, the collector needs to keep multiple sets of mark bits, which allows the reclamation phase to operate on a set that was generated by the previous mark phase.

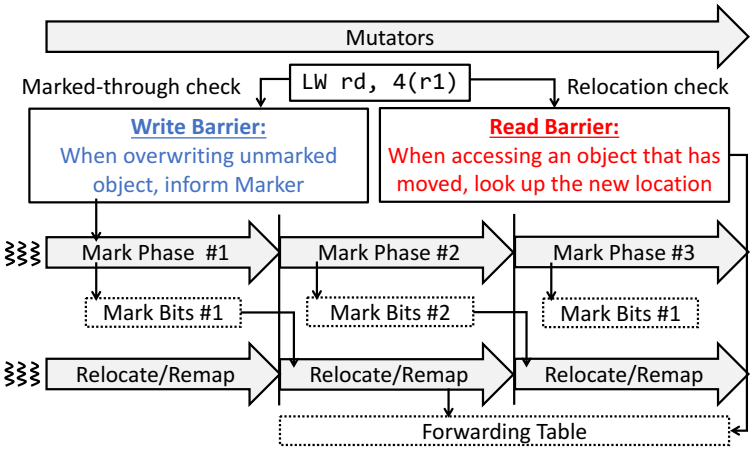


Figure 7.1: *Read and Write Barriers in their general form.* There are many different ways to implement read and write barriers. In our analysis, we consider the above general functionality of these barriers (parallel horizontal arrows describe concurrent execution). Write barriers are required for all concurrent collectors while read barriers are only required for relocating concurrent collectors. Stop-the-world collectors require no barriers.

to the underlying microarchitecture (i.e., how well it can be interleaved with existing code). As barriers are very common operations – they typically have to be added to every reference read and/or write operation – these design decisions have a substantial impact on application and collector performance. Fundamentally, there are three approaches:

1. Compile the code for checking the barrier condition into the instruction stream and branch to a slow-path handler whenever the barrier triggers.

2. Reuse the virtual memory system to fold the barrier check into existing memory accesses and incur a trap if the barrier triggers. For example, the OS can unmap pages before compacting the objects contained within them into a new space, which means that any accesses using old references will raise a page fault. The trap handler can then look up the new location and fix the old reference.

3. Introduce a barrier instruction in hardware. The semantics of these instructions vary, but they typically have similarities to the second approach and raise a fast user-level trap when the barrier triggers.

This leads to a trade-off between **invasiveness**, **programmability** and **performance**. Most existing designs choose option (1), minimizing invasiveness by operating solely in software (e.g., the G1 collector [75] and Go's concurrent GC [111]). Options (2) and (3) have been used in systems such as Sun Lab's Project Maxwell [234], IBM's z14 [118] or Azul's Vega [219]. For example, Azul's Pauseless algorithm [53] relied on a barrier instruction that can raise

fast user-level traps in the slow-path case, while IBM's *Guarded Storage Facility* [67] can protect a set of memory regions and raise a trap when a reference to these regions is loaded. These designs are invasive as they change the CPU, but still prioritize programmability via the software trap handlers.

We take a complementary approach. In addition to considering hardware support for barriers, our insight is that GC is a bad fit for CPUs, and regular enough to be executed in hardware (a similar argument as for hardware page-table walkers). By offloading GC to a small accelerator that takes up a fraction of the area and power of a CPU core, we can reduce the impact of the GC work (at the cost of programmability).

Such an accelerator has a design-space on its own, trading off **area**, **GC performance**, **power** and overall **energy per collection**. As an accelerator is more invasive than a software-only solution, it has to outperform a CPU along at least some of these metrics to be a feasible design point. In particular, the area consumption has to be small, since the accelerator will be idle a large fraction of the time.

The idea of a GC accelerator was previously proposed by Sun [230, 234] and at least one publication in the context of embedded systems [157]. While similar to our proposal, we are not aware of any work that has fully built out such a system and explored the idea in the context of a full-stack high-performance SoC with a focus on area and power consumption. Our accelerator could be used in either a stop-the-world setting (freeing up CPU cores to run other applications) or can be combined with barriers to be used in a pause-free collector.

To describe our system in a more tangible way, we will now present one specific concurrent GC algorithm, Pauseless GC [53], which will act as a straw-man throughout this chapter and serve as an example of how our hardware changes could be integrated with a real collector design. We choose this example because it exercises the most general design point, as it is both relocating and concurrent. Pauseless is also representative of a modern, state-of-the-art collector and has been implemented in different settings (including a specialized hardware appliance), covering different points in the design space.

7.2.1 Example: The Pauseless GC Algorithm

Pauseless GC has two key components: a mark and a relocation (i.e., reclamation) phase. The mark phase (Figure 7.2) regularly performs breadth-first search (BFS) passes over the heap to produce a fresh set of *mark bits* that indicate whether each object is reachable or not (multiple mark bits are maintained for each object). The relocation phase uses the most recent set of mark bits to pick pages in memory that are mostly garbage (i.e., unreachable objects) and compacts them into a fresh page (Figure 7.3). For this to be efficient, page sizes are typically chosen to be large. Figure 7.4 shows this operation in detail.

One challenge is that the mark phase can mistake a reachable object as unreachable if a concurrently running mutator moves an unvisited reference from memory into a register and therefore "hides" it from the mark phase (Section 2.3.3). Like other schemes, Pauseless GC solves this problem through a barrier: whenever a reference is loaded into a register, it is also passed to the mark queue to be *marked through* (added to the BFS). Figure 7.5 shows the

```
while (!q.empty()):
   node = q.dequeue()
   if (!marked(node)):
      mark(node)
      for r in outgoing_refs(node):
          q.enqueue(r);
```

Figure 7.2: *Abstract presentation of the basic mark phase.* Tracing is a breadth-first search (BFS) where the current frontier is kept in a *mark queue* and per-object *mark bits* indicate whether an object has been visited. Every step, we take an object reference off the mark queue, identify all outgoing references stored in object fields and add them to the queue. Once the BFS has finished, the set of mark bits indicates the reachable objects.

```
for cell in from_space:
    new_location = to_space.alloc(size(cell))
    copy_from_to(cell, new_location)
    forwarding_table[cell] = new_location

for object in heap:
    for r in outgoing_refs(object)
        remap(r, forwarding_table[r])
```

Figure 7.3: *Abstract presentation of the basic relocation phase.* A page (`from_space`) is selected for compaction, based on which pages contain the most garbage. A new page is designated as the target (`to_space`) and all live objects are copied into this space, keeping a table of forwarding pointers outside the page. Once the old page has been evacuated, all references to objects in the old page need to be rewritten to point to the new page. This operation can be folded into the next mark phase.

operation of this barrier (one distinguishing feature of Pauseless is that it folds both read and write barriers into a single barrier primitive).

To avoid passing the same reference many times, the barrier is "self-healing": it tags the reference in its original memory location such that next time it is encountered, we know that it was already communicated to the marker. This is implemented by using the most significant bit of each reference to store a NMT (not-marked-through) bit. The bit indicates whether we have already encountered this reference during the current mark. The barrier is only triggered if the bit does not match the current mark phase, and once it has been triggered, the barrier code also flips the NMT of the reference that triggered it. This way,

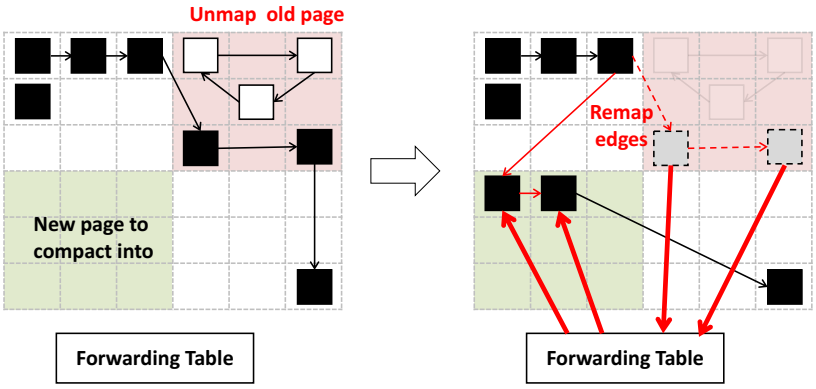


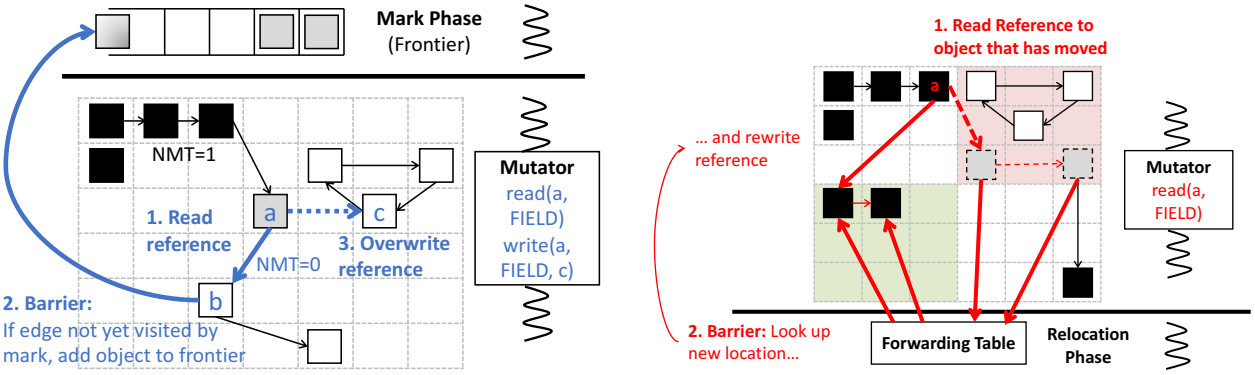
Figure 7.4: *Relocation in Pauseless GC.* A page is selected for evacuation and unmapped to protect it from future memory accesses. All live objects in this page are then compacted (copied) into a different page. A forwarding table is maintained, which can be used to remap any stale references when they are encountered.

the read barrier is only triggered once for each reference. Note that the NMT-bit mechanism means that this bit needs to be masked out whenever a reference is loaded into a register.

The relocation phase also needs to use a barrier mechanism: when an object is moved to another page, other objects may still contain stale references to the old location. To *remap* these references to the new location, another self-healing read barrier is used. When evacuating a page, it is first marked as protected, which will trigger the barrier when it is accessed (depending on the barrier implementation, this may involve updating the page table to cause a trap when the page is accessed). The relocation phase maintains a forwarding table outside the original page, which maps the old location of each object to its location in the new page. If a mutator tries to access the old page (due to a stale reference), it will trigger the read barrier, use the forwarding table to determine the new location of the object (potentially copying the object if it has not been evacuated yet), and update the location where the reference that caused the barrier to trigger came from.

It would be possible to perform all updates in the barriers, but the mark phase also remaps references when it encounters them, to guarantee that all references to an object have been remapped once the mark phase has completed. At this point, the old page (and forwarding table) can be freed.

Azul proposed three ways of implementing the barrier: in software by interleaving it with the instruction stream, in hardware on the Vega platform (which delivers a fast user-level trap if the NMT bit is wrong), or reusing Virtual Memory (mapping all pages into the half of the virtual address space with the right NMT bit and trapping if the barrier is triggered). This leads to a trade-off: The trap-based approaches have a cheap fast path and do not increase the code size substantially, but introduce large overheads when the barrier is triggered. In contrast, interleaving the barrier with the application code increases the code size, and therefore mutator and energy overheads (the effect on code size and instruction



(a) Mark-Through Check **(b)** Relocation Check

Figure 7.5: *Barrier Operation in Pauseless GC.* All barrier checks are folded into a single read barrier. The barrier ensures that references that are loaded into registers are marked through (replacing write-barrier checks). The most significant bit of each reference indicates whether this reference was seen before in the current mark phase, and is updated by the barrier (this is known as "self-healing"). The barrier also ensures that mutators only see the new location of an object. When trying to access a stale location, they look up the new location in a forwarding table and update the faulty reference.

cache pressure can be significant, as the barrier adds instructions each time a reference is loaded from memory). The latter effect is particularly pronounced in data center servers, which have been shown to be sensitive to instruction cache pressure [129].

7.3 Motivation & Challenges of Hardware GC

The Pauseless GC algorithm almost fully eliminates garbage collection pauses, albeit at the cost of slowing down the application. While we were unable to find public performance numbers to quantify this slow-down, it is important to note that the garbage collector still has to spend at least as much time in the collector as a stop-the-world collector would, but spread out across different cores and application threads. It is therefore a safe assumption that these overheads can feasibly reach 38% of execution time and 25% of energy consumption, as they do with stop-the-world collectors.

We believe that these overheads primarily stem from two sources. These are not limited to Pauseless GC specifically, but apply to any concurrent garbage collector:

1. **CPU cycles spent on garbage collection**. As we confirmed in Chapter 6, GC is a highly parallel workload that is dominated by being able to keep as many memory requests in-flight as possible. CPUs are bad at this task, since they are limited by the size of their load-store queue and the instruction window [30]. For this reason, parallel collectors use a large number of the system's CPU cores only to issue memory requests.

These are CPU cycles that are taken away from the application, assuming that the application could otherwise fully utilize all cores.

Furthermore, CPUs consume a large amount of area and power due to features such as caches, specialized functional units or instruction fetch and decode, which are not useful for garbage collection. Specifically, GC does not benefit from large caches, as it only visits every object once during the GC. As we discovered in Chapter 6, it is possible to make GC efficient on architectures without caches. A dedicated unit without the caches and general-purpose hardware available in CPUs should therefore be able to perform the same GC operation as a CPU, but at a much smaller area and energy footprint.

2. **Mutator overheads from barrier executions**. Read barriers in software introduce large overheads due to executing additional instructions, as well as increasing the code size (and therefore instruction cache pressure). To address this problem, Azul, Sun and (more recently) IBM introduced hardware implementations of read barriers that perform the barrier check efficiently in hardware and raise a user-level trap if the barrier triggers. While this improves the fast path of the barrier, the slow path (when the barrier is triggered) is still expensive, as it requires an instruction stream redirect that involves flushing the pipeline. Worse, out-of-order processors cannot speculate over these instruction stream redirects. As such, even hardware read barriers can cause substantial overheads when many of them trigger the slow path, which is an effect that Azul described as "trap storm" [53]. This effect is particularly common on heaps with a large amount of churn and therefore GC activity.

To address these two sources of overhead, we propose an approach that moves both of them into hardware, through a GC accelerator close to memory combined with optional barrier support integrated into the CPU. We are presenting our design in the context of a general concurrent, relocating mark-compact collector, and use Pauseless as one specific example of such a collector (while our design generalizes to a wider range of schemes).

We believe a concurrent, relocating collector is the right design point for most modern server applications, as many of them are affected by GC pauses (even when short) and fragmentation is a problem for long-running data center workloads. If such a GC can be made efficient, it may replace OpenJDK's CMS and G1 (which still have pauses) as the default.

7.4 Hardware Overview

As we observed in Section 7.3, both the mark and relocation phase in the Pauseless GC algorithm are a poor fit for general-purpose CPUs – as such, they inefficiently use cores that could otherwise be used for the application. Furthermore, taking a trap for each triggered read barrier is inefficient, similar to the argument for refilling TLBs with a hardware page-table walker. Our proposal adds two modifications to an otherwise unmodified SoC, allowing our design to be integrated in a non-invasive way (Figure 7.6):

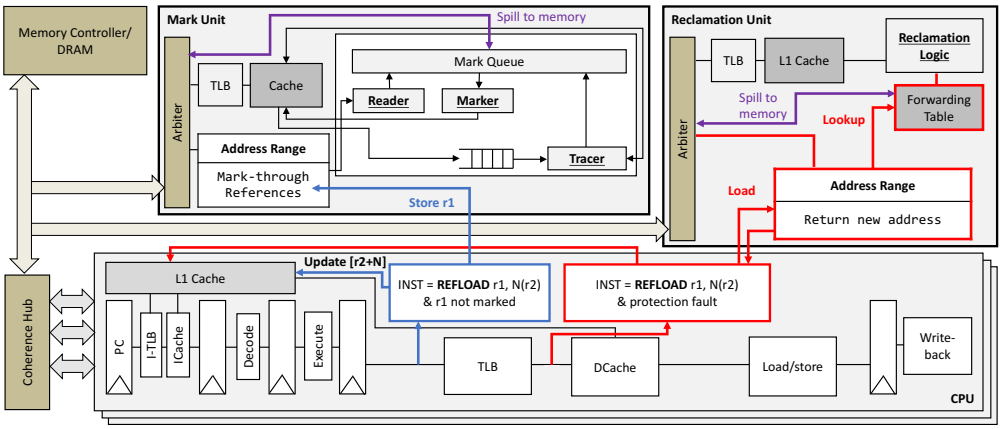


Figure 7.6: *Overview of our GC Design.* We introduce two specialized units, a *Mark Unit* that performs the GC's breath-first search, and a *Reclamation Unit* that identifies and frees dead memory. These units are connected to the interconnect, similar to other DMA-capable devices. Blue shows optional changes that would be required for any concurrent GC. Red are changes that would be required for concurrent, relocating GC. The proposed `REFLOAD` instruction acts semantically as loading a reference into a register, but automatically performs read and write barrier checks in hardware. We use the strategy from Pauseless GC (Section 7.2.1) as an example, using an NMT-bit to check that an edge has not been marked yet, and virtual memory faults to detect relocation. For simplicity, we show the changes for an in-order pipeline, but they could be implemented in an out-of-order core as well.

7.4.1 Hardware Accelerator for Tracing GC

We introduce two new hardware units that perform the mark and reclamation operations. For the mark's BFS traversal to be efficient, we need to keep as many memory requests in flight as possible to maximize memory bandwidth. While an out-of-order core is very good at this, it adds overheads in terms of power and area, since most of its logic, including instruction fetch, decode, issue window, reorder buffer, etc. are not required for a BFS. Further, the caching behavior of the mark phase is unfavorable for a general-purpose core: no data is ever reused except the mark bits (as we mark through every object once). Since caches cannot hold individual bits, this leaves a choice of not caching the mark bits by using non-allocating loads (wasting locality and performance) or caching the entire cache line (wasting space). The latter also pollutes the cache, potentially slowing down mutators. Analogously, the reclamation phase is an embarrassingly parallel copy or sweep operation, which does not benefit from caching or out-of-order execution (and can be parallelized well in hardware).

Finally, both phases benefit from being executed close to memory, reducing energy consumption from data movement. The mark and relocation unit are therefore located beyond the LLC, share the virtual address space with the process they are operating on (i.e., have their own TLBs) and are cache-coherent with the cores. Both units carve out a small range of the physical address space for communication with the CPUs.

7.4.2 Hardware Support for Barriers

While our evaluation focuses entirely on the hardware accelerator, we propose two ways it could be integrated with a concurrent garbage collector in a way that avoids the traps and instruction stream redirects that are currently associated with read barriers, whether implemented in software or through hardware support.

Specifically, we propose a scheme to move barrier functionality to the hardware accelerator without modifying the CPU (Section 7.6.1). While non-invasive, this scheme increases TLB, instruction and data cache pressure. We therefore also propose a more efficient scheme that modifies a CPU and adds a new *reference load* instruction to the CPU's instruction set (`REFLOAD`), similar to *Project Maxwell* [234]. Figure 7.6 shows an overview of this design. In contrast to Maxwell and existing hardware read barriers, our proposed instruction does not raise a trap but handles both the fast and slow path in hardware. In practice, this makes it semantically equivalent to a memory load that is handled by the load-store queue but may take longer to resolve (similar to a last-level cache miss). As a result, the processor does not have to flush the pipeline and can speculate over the barrier, even if it is triggered.

7.5 GC Accelerator Design

We will now describe the two parts of our accelerator, a *Mark Unit* that performs the BFS, and a *Reclamation Unit* that frees up memory and places the resulting allocation lists into main memory for the application on the CPU to use.

7.5.1 The Mark Unit

The mark unit implements a fully pipelined breadth first search (BFS) in hardware. The beginning and end of the BFS's mark queue are held on-chip ❶, and two units – the *Marker* ❷ and the *Tracer* ❸ perform the main operation of taking objects off the mark queue, marking them and copying their outgoing references back into the mark queue.

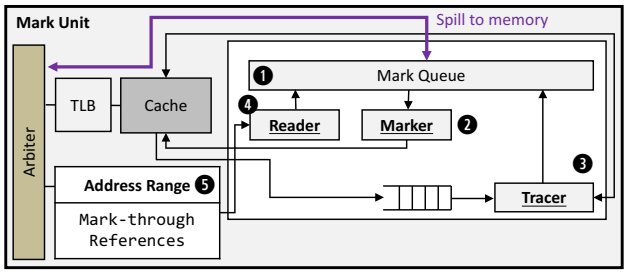


Figure 7.7: *Overview of the Mark Unit.* The unit performs a breadth-first search (BFS) in hardware and communicates with the CPU through an address range in physical memory.

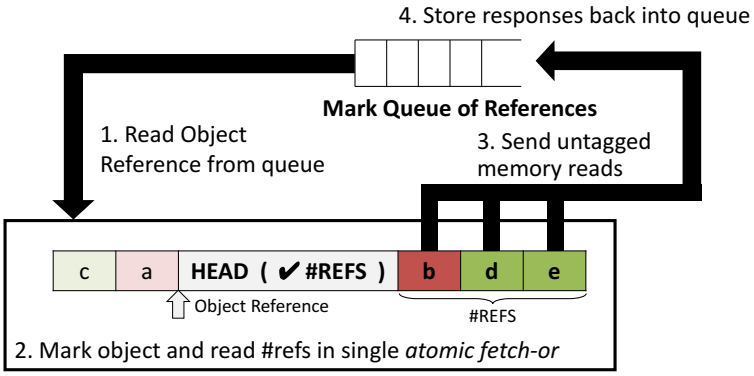


Figure 7.8: *The Basic Mark Unit Operation.* Our key insight is that memory requests copying outgoing references into the mark queue do not need to be tagged.

After launching a mark phase, the *Reader* ❹ first loads all roots into the mark queue. It communicates with the CPUs through a range of the physical address space. This range is cacheable and each CPU can write to it, to send addresses of reachable objects to the mark unit. The reader polls the range until it has received all roots (CPUs terminate root lists with a special word). Roots are collected in software, an infrequent operation that can be implemented without stalling [185].

The mark queue is implemented as an on-chip SRAM, and is expected to be small. There is a design space associated with the size of the queue: the smaller the queue, the more often its middle part has to be spilled to memory. We found that a queue size of 2KB is sufficient, and will explore this design space in Section 9.7.2.

Once the roots have been loaded, the *Marker* ❷ and the *Tracer* ❸ perform the actual mark phase. The basic mark operation is shown in Figure 7.8. As on the GPU, there are two fundamental types of operation. The first is the *marking*, which takes a reference off the mark queue and checks and sets its mark bit. We fold this in with reading the number of outgoing references, by storing both this number and the mark bits in the same word. This means that this operation becomes a single atomic `fetch-or` memory request. The second type of operation is the copying of all outgoing references back into the mark queue, if the object had not been marked before. We call this a *tracing* operation.

There are three insights that, taken together, enable our mark unit design to outperform a CPU both in terms of performance and area: First, we use the same kind of bidirectional object layout that already enabled our GPU-based garbage collector (Chapter 6.4.2). Second, we build on our insights from Chapter 6 that the number of outgoing references in objects is bimodal and therefore, similar to our *Reducing Divergence* optimization on the GPU, decouple marking and copying of references, in order to prevent long objects from stalling the mark operation. Finally, we realized that the order in which references are added to the mark queue does not affect correctness. We therefore can perform these operations without tracking the associated memory requests. We now describe each of these aspects in turn.

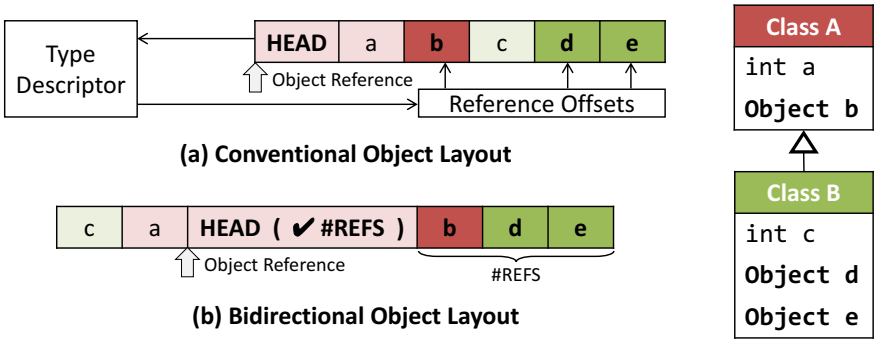


Figure 7.9: *Bidirectional Object Layout.* Recalling Section 6.4.2, to support inheritance, objects are traditionally laid out with the fields of all parents first, followed by the class's own fields. This results in reference fields interspersed throughout the object. In a bidirectional layout, the header is placed in the middle, all reference fields are stored on one side of the header, and all non-reference fields on the other. This still supports inheritance, but identifies reference fields without extra accesses.

1. **Bidirectional Object Layout** GC does not benefit much from caches, and the mark unit can therefore save most of this area and power. However, as we discovered in Chapter 6, if one were to use a cacheless accelerator design with an unmodified language runtime system, the performance would be poor. As on the GPU, the reason is that when copying the outbound references of an object back into the mark queue, the collector has to identify which fields contain these references. While this can be done using specialized functions or tag bits on architectures that support them [233], most runtime systems use a layout where the object's header points to a type information block (TIB), which contains the offsets of the fields that contain references (Figure 7.9a).

   This approach works well on systems with caches, since most TIBs are in cache. However, it adds two additional memory accesses per object in a cacheless system. To address this, we use a *bidirectional layout* (Figure 7.9b). The benefits of the bidirectional layout on CPUs are limited [90], but as we confirmed on the GPU, such a layout helps greatly on a system without caches, as it eliminates the extra accesses. Its access pattern (a unit-stride copy) is beneficial as well. While this approach requires adapting the runtime system to target our accelerator, the changes are invisible to the application and contained at the language-runtime level.

2. **Decoupling Marking and Tracing** With the bidirectional layout in place, we can store the mark bit and the number of references in a single header word, which allows us to mark an object and receive the outbound number of references in a single *fetch-or* atomic memory operation (AMO). On a CPU, a limited number of these requests can be in flight, due to the finite load-store queue. Since the outcome of the mark operation determines whether or not references are being copied, this limits how far a CPU can speculate ahead in the control flow and causes expensive branch mispredicts.

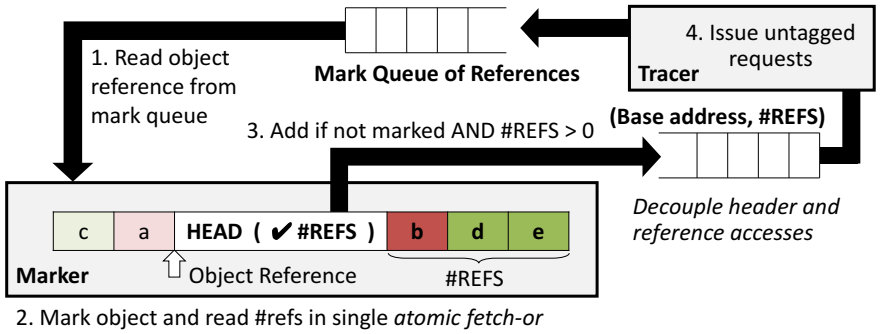


Figure 7.10: *Mark Unit Operation.* The Marker removes objects from the frontier queue, uses a single AMO to mark the header word and receive the number of outbound references, and (if the object has not been marked yet), enqueues it in the tracer queue. The Tracer removes elements from the tracer queue and copies the object's references into the mark queue using untagged memory requests.

In our mark unit, the number of outstanding requests can be larger but is still limited by the number of available tags, similar to MSHRs. If the unit encounters a large number of objects without outbound references or that have already been marked, our effective bandwidth is limited by the mark operation. Similarly, if there are very long objects, we are limited by tracing (i.e., copying) this one long object.

We therefore decouple the marking and tracing from each other. Our mark unit consists of a pipeline with a Marker and a Tracer connected via a tracer queue (Figure 7.10). If a long object is being examined by the tracer, the Marker continues to operate and the queue fills up. Likewise, if there are few objects to trace, the queue is drained. This design allows us to make better use of the available memory bandwidth than a control-flow-limited CPU could (Section 9.7.1).

3. **Untagged Reference Tracing** While the Marker needs to track memory requests (to match returning mark bits to their objects), the order in which references are added to the mark queue does not affect the correctness of the BFS. The Tracer therefore does not need to store request state, but can instead send as many requests into the memory system as possible, adding responses to the mark queue in the order in which they return. This increases bandwidth utilization, as we will show in Chapter 9.

Taken together, the previous three strategies enable the mark unit to achieve a higher memory bandwidth than the CPU, at a fraction of on-chip area and power. Note that there are multiple components that communicate with the memory system (marker, tracer, mark queue and root loader). We experimented with several strategies to perform and prioritize these different types of memory accesses (Section 9.7.2).

7.5.2 The Reclamation Unit

The task of the reclamation unit is to take the mark bits produced by the mark unit and free the memory associated with unmarked objects. While this operation is highly dependent on the underlying GC algorithm, it typically involves iterating through a list of blocks and either (1) evacuating all live objects in a block into a new location (for relocating GC) or (2) arranging all dead objects into a free list (for non-relocating GC).

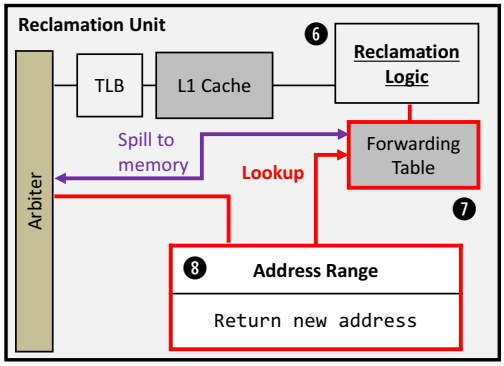


Figure 7.11: *Overview of the Reclamation Unit.* The specific reclamation logic depends on the underlying GC algorithm. For a relocating collector, a forwarding table is required, as well as a mechanism for the CPU to query it through a memory-mapped region.

Each of these operations can be performed with a small state machine ❻, and can be parallelized across blocks. It is this property that leads to a potential for improving this task through a hardware implementation, as it is an embarrassingly parallel operation with a high degree of parallelism. We now give two examples of how this unit would work in a non-relocating *Mark & Sweep* collector as well as within the Pauseless GC algorithm.

Example: Mark & Sweep GC

For a non-relocating Mark & Sweep collector, the relocation logic iterates through all blocks, parallelizes them across a set of *block sweeper* units, which each reclaim memory in a block independently and then return the block either to a list of free or live blocks (Figure 7.12). As each block sweeper is negligibly small, the design is primarily dominated by the size of the cross-bar connecting them.

Example: Pauseless GC

For a concurrent, relocating collector such as Pauseless GC, the reclamation logic would be more complex. As described in Section 7.2.1, the reclamation unit in this case has to regularly (1) find pages that are mostly garbage and should therefore be evacuated, (2) build a side-table ❼ of forwarding pointers, (3) protect the original page in the page table, and (4)

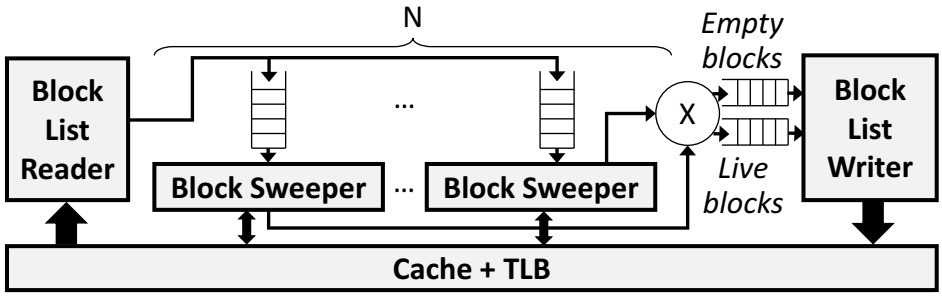


Figure 7.12: *Reclamation Logic for a Non-Relocating Mark & Sweep Collector.* Blocks are read from a global free list, distributed to parallel blocks weepers that reclaim them in parallel, and then returned to the respective empty and live free lists.

move objects to the new page. At the same time, the relocation unit receives requests from CPU cores when they need to find the new location of an object ❽. This can be implemented by providing a region in the physical address space – similar to the mark unit – from which a CPU can read a specific location to read the new location of an object.

If the data has already been relocated, the relocation unit will respond immediately with the new address, otherwise it will relocate the object and then respond. CPUs hence always get a response with the new location, but it may be delayed, which is seen by the CPU as a long memory load. One advantage over the software approach is that this makes concurrency much simpler. Like the mark queue, the forwarding table may have to be spilled to memory, but a part of it can be cached by the relocation unit. This gives rise to a large design space, such as whether to use a CAM or hash table.

Once the relocation unit has finished relocating a page and remapping has completed, it frees the physical memory and writes the addresses of available blocks into a free-list in memory, which can be accessed by conventional bump-pointer allocators on the CPUs.

7.6 Software Integration

Using the unit in a stop-the-world setting requires minimum integration beyond the new object layout. The runtime system first needs to identify the set of roots (which can be done in software without stalling the application [185]) and write them into a memory region visible to the accelerator. It also has to inform the unit where in memory the allocation regions are located, as well as configuration parameters (e.g., available size classes). Beyond this, the unit acts autonomously and the runtime system polls a control register to wait for it to be ready. Note that no modifications to the CPU or memory system are required. Instead, the unit acts as a memory-mapped device, similar to a NIC. For a concurrent garbage collector, further modifications are required.

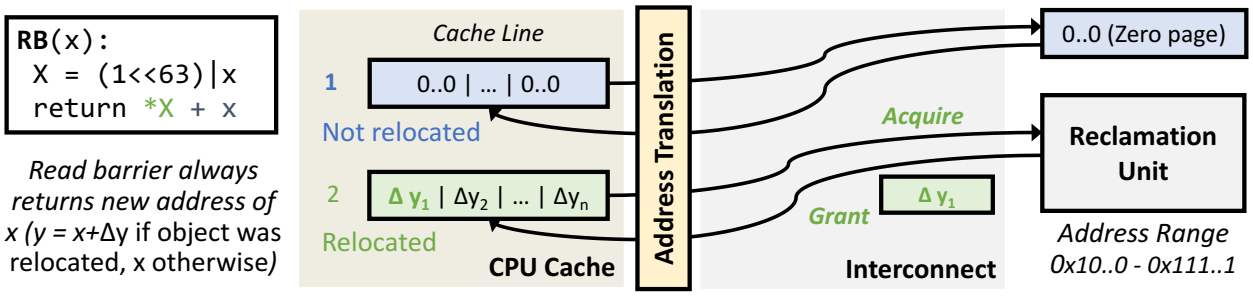


Figure 7.13: *Read Barrier Without CPU Modifications.* The barrier checks object references and ensures they point to their new location, whether or not they have been moved. By reinterpreting coherence messages, we can implement the functionality in hardware without changing the CPU (at the cost of increased TLB and cache pressure).

7.6.1 Software Integration for Concurrent GC

Our design can be integrated into a concurrent GC without modifying the CPU. While our prototype is evaluated in a stop-the-world setting, we propose a novel barrier design for efficient integration with concurrent GC. Our insight is that by "hijacking" the cache coherence protocol, we can implement the barriers without causing traps or branch mispredicts.

Write Barrier When overwriting a reference, write the reference of the object it belongs to into the same region in memory that is used to communicate the roots. The mark unit copies all references that are written into this region to the mark queue. This has similarities to the approach taken by Maxwell [234].

Read Barrier For a relocating collector, the read barrier needs to check whether an object has moved and get the object's new location if it has. This is very collector-dependent, but for the purpose of this description, we will assume Pauseless GC, which invalidates all objects within a page at the same time, compacts these objects into new locations, and keeps a forwarding table to map old to new addresses.

Azul's barrier implementation requires either a hardware barrier that raises a trap when accessing an object from a page that has moved, or a check in software that may lead to an instruction-stream redirect. We propose avoiding both of these sources of overhead by performing the barrier operation in the relocation unit and using the existing cache coherence protocol to synchronize this operation with the CPU.

We propose adding a new range to the physical address space that nominally belongs to the Reclamation Unit but is not backed by actual DRAM (Figure 7.13). We then steal one bit of each virtual address (say, the MSB), mapping the heap to the bottom half of the virtual address space. Whenever we read a reference into a register, we add instructions that take the address, flip this special bit, read from that location in virtual memory and add the result to the original address.

By default, we map the top half of this virtual address space to a page that is all zeros. All these loads hence return 0 (i.e., have no impact on the address, as the object has not moved). However, when we are relocating objects on a page, we map the corresponding VM page to the reclamation unit's physical address range instead. The unit then sends out *probe* messages across the interconnect to take exclusive ownership of all cache lines in this page. This means that whenever a thread tries to access an object within this page, the CPU needs to acquire this cache line from the reclamation unit. When responding to the request, the unit simply writes the deltas by which the new addresses differ to the original addresses of the objects in the cache line. It then releases the cache line, the CPU reads from it and adds the value to the original address, updating it to the new one. This communication only has to happen once, since the cache line is in the cache the next time the object is accessed.

This operation could be further optimized by storing addresses directly with the MSB bit set to 1, so that the extra operation of flipping the bit is not necessary. While this operation does not require modifying the CPU – and allows the CPU to speculate over the read barrier whether it triggers or not, rather than introducing an instruction stream redirect if it does – this approach does introduce additional overheads:

- The effective TLB size is divided by half, as every object now occupies two entries instead of one: one for the cache line including the deltas.
- The effective data cache size is divided by half, as additional cache lines need to be resident in the cache. However, this is less than a factor of two, since most additional cache lines are mapped to the same page that contains all zeros.
- Instruction cache pressure is increased from the additional instructions for the read barrier. An indirect effect is that the distance of some branches may increase, potentially introducing indirect branches where a direct branch could be used before.

If we allow ourselves to modify the CPU as well, we can avoid these sources of overhead. We will now describe a scheme that achieves this goal. Note that we have not implemented these changes (or, in fact, any concurrent collector), but are considering them for future work.

7.7 Optional CPU Extensions

To avoid the overheads introduced by the scheme from the previous section, we propose the introduction of a `REFLOAD` instruction which semantically behaves like a load and is loading a reference from memory into a register. If the object that the reference is pointing to is in a page that is being relocated, the new location is looked up and returned (the response has to be delayed until the object has actually been copied). The instruction also takes care of any other read and write barrier functionality. As a result, the cache and TLB overheads of the software approach are eliminated, and the operation can be completed in a single instruction instead of increasing instruction cache pressure.

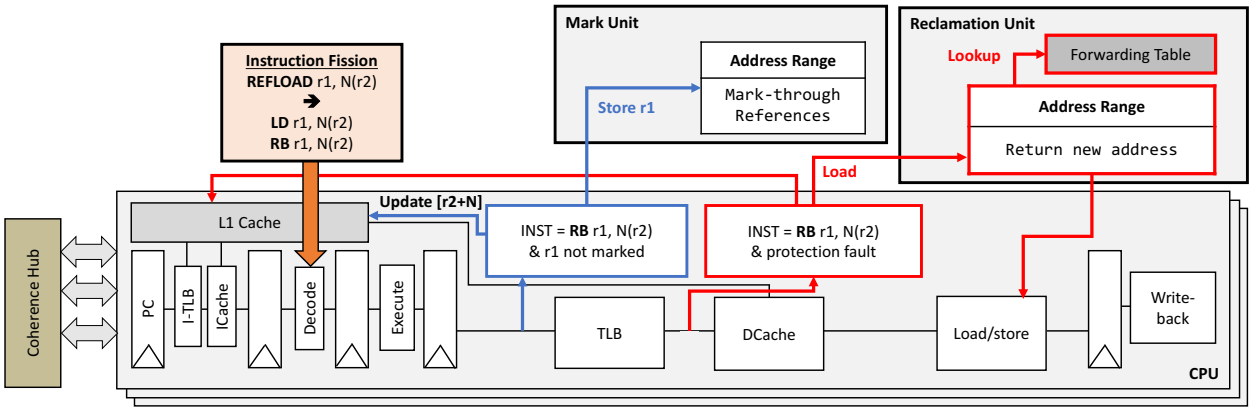


Figure 7.14: *Overview of the Optional CPU Extensions.* A `REFLOAD` instruction is added to load a reference into a register. The write barrier path informs the mark unit of the reference load. The read barrier path uses virtual memory to check whether the page the loaded reference points to is being relocated, and looks up the new location if it is.

While the precise barrier functionality is, once again, collector dependent, a Pauseless GC style barrier is representative of a wide range of collector designs. In this case, the `REFLOAD` instructions performs two checks. First, it checks whether the NMT bit is set. If not, it writes the reference into the mark unit's memory, updates the NMT bit in the cache and continues. Second, it checks whether the object the reference is pointing to is being relocated. This can be folded into the virtual memory system, similar to Azul's approach. Specifically, accessing a page that is being relocated triggers a protection fault in the TLB.

We can therefore implement this check cheaply in hardware by splitting each `REFLOAD` instruction into two instructions during the decode stage (i.e., *instruction fission*), one regular load and a custom `RB` micro-op, which is treated by the TLB like a load from the address that was just loaded, but does not always proceed into the memory stage. If the object is not being relocated, `RB` completes quickly and simply writes its first operand into the target register, but otherwise, it triggers a TLB fault, which is intercepted and translated into a load from the reclamation unit's physical address space (similar to our approach in Section 7.6.1). This load then proceeds to the memory stage, is added to the load-store queue and can be speculated over like any other load (the GC unit will not release the cache line until it has looked up the new location). When the load completes, it results in the new reference being stored in the `REFLOAD` instruction's target register.

This approach means that the read barrier can now be speculated over like any other load, without introducing any additional traps, instruction stream redirects or pauses longer than a last-level cache miss. Other than a reduction of trap storms and improvement of overall performance, this has the advantage that it may make the system more predictable.

7.8 Related Work

Our work has similarities to Azul System's Vega [219], a commercial processor specialized for Java applications. Vega also adds hardware-support for concurrent GC, but still executes most of the algorithm in software. Its main hardware feature is a read barrier instruction that delivers a fast user-level trap to respond to relocation. Azul has since stopped producing hardware, implementing the read barrier in software on commodity CPUs [244]. We believe that by moving much more of the algorithm into hardware, we may substantially improve over Vega in terms of energy efficiency, and potentially mutator and GC throughput. This is a different design point than Vega's, which appears to prioritize generality and flexibility.

Recent IBM mainframe processors of the z14 series have introduced a new hardware mechanism called "Guarded Storage Facility" [67]. This mechanism is similar to Azul's read barrier instructions, allowing the software to configure up to 64 *guarded* regions that trigger a trap when performing a guarded load of a reference to that region. As in Azul's Vega hardware, the main garbage collection is still performed on the CPU cores, and the application incurs regular traps due to barrier activity.

In 2008, Sun worked on a very similar idea to ours [230, 234], with specialized GC units close to memory and hardware support for barriers. The design provides a fully concurrent GC but relies on an object-based memory system which requires changes to the caches and memory hierarchy (e.g., to translate between object IDs and physical addresses, and to keep track of forwarded objects). This work also proposed the use of a bidirectional layout in a GC accelerator. However, to our knowledge, the system was never released.

There exists work on GC coprocessors in the embedded and real-time space, including an extensive body of work on *Java processors* [154, 157, 195]. Some of these processors have dedicated features for garbage collection, such as support for non-blocking object copying in real-time systems [196]. Other work proposes support for read or write barriers [96, 157] and reference counting in hardware [121] (the latter is non-relocating and requires a backup tracing collector). Additionally, there exist a range of general mechanisms to support Java workloads at the micro-architectural level [163], and some of these mechanisms can be applied to garbage collection [99]. Our work has similarities to many of these projects, but its focus on energy efficiency and data center workloads is somewhat different.

Finally, a hardware tracing unit was presented by Bacon et al. [24]. However, this work was in the context of garbage collection for Block RAMs (BRAMs) on FPGAs, which is a special case and very different from conventional GC on DRAM. Specifically, BRAMs are multi-ported and have perfectly predictable timing, which allows a garbage collector to give much stronger timing guarantees.

7.9 Summary

In this chapter, we showed a general version of our Hardware GC design, and described the associated design space. In the next chapters, we will show a specific incarnation of this

design, implemented in the context of a real System-on-Chip (SoC). As hardware support for garbage collection interacts with many layers of both the hardware and the software stack, we developed research infrastructure and a new hardware-software research methodology to enable this research. Chapter 8 will present this research methodology. Chapter 9 will then describe a prototype of our hardware-assisted garbage collector, integrated into a full SoC and co-designed with the JikesRVM research virtual machine. Through this design, we show the potential of hardware support for garbage collection and demonstrate that our design can be integrated into a production-grade system without invasive changes.

Chapter 8

A RISC-V based Managed-Language Research Methodology

Existing research methodologies for hardware-software research are inadequate to evaluate many systems with fine-grained interactions between the different layers of the stack, such as our proposed hardware-assisted GC design. We therefore present a new approach to managed-language research, which uses FPGA-based simulation of production-grade RISC-V SoCs combined with a port of the Jikes Research VM to execute Java workloads. This chapter describes the details of this port, briefly evaluates its performance and presents a case study that demonstrates research that is facilitated by this methodology.

8.1 Introduction

One of the challenges in evaluating proposals such as our hardware-assisted GC design is that they are poorly supported by existing evaluation methodologies for computer architecture research. Existing research methodologies either employ off-the-shelf hardware, high-level full-system simulators such as SIMICS [153], or software-based cycle-accurate simulators such as GEM5 [35]. While these approaches are suitable for simulating non-managed workloads such as SPEC CPU, they fall short for managed workloads such as Java applications.

This disconnect has existed for a long time, and was pointed out in a prominent *Communications of the ACM* article in 2008 [38]. However, not much has changed since then. A part of the problem is arguably that the properties of managed languages make them a poor fit for the most widely used computer architecture research methodologies:

- High-level full-system simulators do not provide the fidelity to fully capture managed-language workloads. These workloads often interact at very small time-scales. For example, GCs may introduce small delays from barriers (≈ 10 cycles each), scattered through the application [53]. Cumulatively, these delays may add up to substantial overheads but individually, they can only be captured with a high-fidelity model.

- Software-based cycle-accurate simulators are too slow for managed workloads. These simulators typically achieve on the order of 400 KIPS [179], or 1s of simulated time per 1.5h of simulation per core. Managed-language workloads are typically long-running (i.e., a minute and more) and run across a large number of cores, which means that simulating an 8-core workload for 1 minute takes around a month.

- Native workloads often take advantage of sampling-based approaches, or use solutions such as Simpoints [199] to determine regions of interest in workloads and then only simulate those regions. This does not work for managed workloads, as they consist of several components running in parallel and affecting each other, including the garbage collector, JIT compiler and features with dynamically changing state such as biased locks, inline caching for dynamic dispatch, etc. In addition, managed application performance is often not dominated by specific kernels or regions of interests, which makes approaches that change between high-level and detailed simulation modes (e.g., MARSSx86 [179], Sniper [44]) unsuitable for many of these workloads.

For these reasons, a large fraction of managed-language research relies on stock hardware for experimentation. While this has enabled programming-languages research on improving garbage collectors, JIT compilers and runtime system abstractions, there has been relatively little research on hardware-software co-design for managed languages. Furthermore, the research that does exist in this area typically explores a single design point, often in the context of a released chip or product, such as Azul's Vega appliance [53] or IBM's z14. Architectural design-space exploration is rare, especially in academia.

We believe that credible hardware-software co-design research requires an experimentation platform where all layers of the stack can be readily modified, including the hardware, the operating system and the language runtime system. At the same time, such a platform needs to capture the essential properties of commercial systems, to be representative of industry-grade implementations and enable industry adoption.

One approach is to integrate a high-speed software simulator with a language-runtime system, to enable modifications across the full stack. This is the approach taken by MaxSim [190], which combines the Maxine Research JVM [229] with the Zsim simulation framework [193], and McPAT [141] for power analysis. This infrastructure can simulate the full set of DaCapo benchmarks in a day and enables the exploration of hardware-software co-designed features such as object layout transformations and tagged pointers. While such a platform enables a wide range of studies at the hardware-software interface, it does not model micro-architectural details beyond those found in a conventional CPU, and is therefore less suited for studying an accelerator with a new micro-architectural design, such as ours.

An alternative approach is to build a research platform around RTL of real hardware designs and run these designs on FPGAs, together with a full software stack. Such a platform was not previously possible, as the vast majority of hardware has traditionally been proprietary, and available open-source hardware was either not representative of high-performance implementations (e.g., OpenRISC [210]) or difficult to modify (e.g., OpenSPARC [178]).

However, recent years have seen an increase in activity around hardware that is both open and can be easily modified. One example is OpenPiton [25], an open-source SoC from Princeton, which can be targeted to FPGAs and has been used in ASIC implementations. Another strand of activity is based on the RISC-V ISA, which is a free and open instruction set that originated at UC Berkeley. RISC-V enables both open and proprietary implementations, and has seen a growing ecosystem of open-source hardware evolve around it.

This open-source hardware enables a new kind of hardware-software co-design research approach that builds ideas into the same open-source RTL used by industry and contributes the results back to the community, similar to how Linux is used in OS research. A promising example for this approach is the open-source Rocket Chip SoC generator [18], which provides a framework to generate full SoCs that are realistic (i.e., used in products) and can target both ASIC and FPGA flows. Combining this infrastructure with an FPGA-based simulation framework such as MIDAS [131] enables *simulating* the performance of real Rocket Chip SoCs at full cycle-accuracy, while running at FPGA frequencies of up to 190 MHz.

While FPGA-based simulation infrastructure was traditionally constrained by the size of available FPGAs, this has changed in recent years, and there are now large FPGA boards available, including for rent in the public cloud [6]. These boards can address several gigabytes of DRAM and are capable of running managed workloads on simulated multi-core SoCs. This means that this infrastructure can achieve the realism, fidelity and simulation speed required for credibly evaluating complete managed-language applications.

We believe that combining this infrastructure with an easy-to-modify managed-language runtime system provides an opportunity to perform hardware-software research on managed runtimes that was infeasible before. We identified the Jikes Research Virtual Machine (JikesRVM), which we already used in Chapter 6, as the most promising candidate for this managed-language runtime system. Jikes is Java VM geared towards experimentation and therefore particularly well-suited for research. Meanwhile, JikesRVM is easy to modify, thanks to being written in Java and using a modular software design that decouples components such as the object layout, GC or JIT passes from each other.

We believe that bringing RISC-V and JikesRVM together will enable novel hardware-software research, while facilitating replicability and industry adoption of research. In this chapter, we present an important step towards this vision, by porting JikesRVM to RISC-V. We first discuss why such a port is necessary. We then describe the porting effort in detail. Finally, we demonstrate the running system, and show the research that it enables.

8.2 The RISC-V Ecosystem

RISC-V is a free and open ISA that was originally developed at UC Berkeley and is now managed by a standard body, the *RISC-V Foundation*. Since RISC-V is an open standard, anyone can implement it, either within the context of a proprietary product or as open-source hardware. The main benefit of RISC-V stems from its ecosystem: while other ISAs are largely controlled by individual companies, RISC-V allows different parties to leverage the

investment that has been made within the software and hardware ecosystems, and contribute back to these ecosystems themselves. This is similar to the model behind other open-source software infrastructure such as Linux, Kubernetes [41] or TensorFlow [2].

From a technical perspective, the RISC-V ISA distinguishes itself through simplicity and modularity. The base ISA contains only 47 instructions and is sufficient for a fully functional processor. Advanced features such as floating point or atomics are provided as extensions, which are optional and enabled through compiler flags. RISC-V supports three address sizes – 32, 64 and 128 bit – and separates the supervisor and machine-level portions of the ISA from the user-level ISA. This modularity means that the same ISA can be used for systems ranging from microcontrollers to full server SoCs with high-performance out-of-order processors. This enables a substantial amount of software reuse, as the same compiler and software ecosystem can target a wide range of microarchitectures and application scenarios.

Since its inception in 2010, there has been a growing ecosystem of both commercial implementations of RISC-V, as well as RISC-V-based open-source hardware. A major project in this space is Rocket Chip [18], which is a System-on-Chip (SoC) generator written in Chisel, a hardware description language developed at UC Berkeley. Rocket Chip contains a selection of processor designs (including in-order and out-of-order cores), a configurable on-chip interconnect called TileLink, caches, as well as various peripherals and devices. All parts of Rocket Chip are implemented as *generators*, which means that rather than implementing a single instance, they can instantiate a wide range of designs based on a central set of configuration parameters. This design – combined with Chisel as the source language – makes Rocket Chip highly extensible and easy to modify.

While Rocket Chip has been used in over a dozen tapeouts and in several shipped products, the same RTL can be used in FPGA-based simulation. This is enabled by additional infrastructure that originated at UC Berkeley, specifically the MIDAS [131] simulation framework. MIDAS uses a Decoupled FAME approach [209] to simulate ASIC designs on FPGAs: The target RTL runs cycle-accurately on the FPGA and a token-based mechanism ensures that timing of any off-chip requests (such as DRAM accesses) is adjusted to ensure that the relative speed of peripherals is consistent with what it would be if they were attached to an ASIC. An addition to this methodology, Strober [132], extends this approach to detailed energy simulation numbers by sampling MIDAS state at random intervals and simulating these snapshots in a gate-level power modelling tool.

In addition to this hardware infrastructure, RISC-V supports a growing software ecosystem, including ports of GCC, Linux, glibc, binutils, LLVM, QEMU, Go, FreeBSD and coreboot (among others). This enables a wide range of real software to run on Rocket Chip, letting researchers execute a variety of software on hardware that can be easily modified.

8.3 The Jikes RVM

To apply this approach to managed runtimes, we require a runtime system that can be easily modified as well. We picked the Jikes Research VM [4], which is the de facto standard in

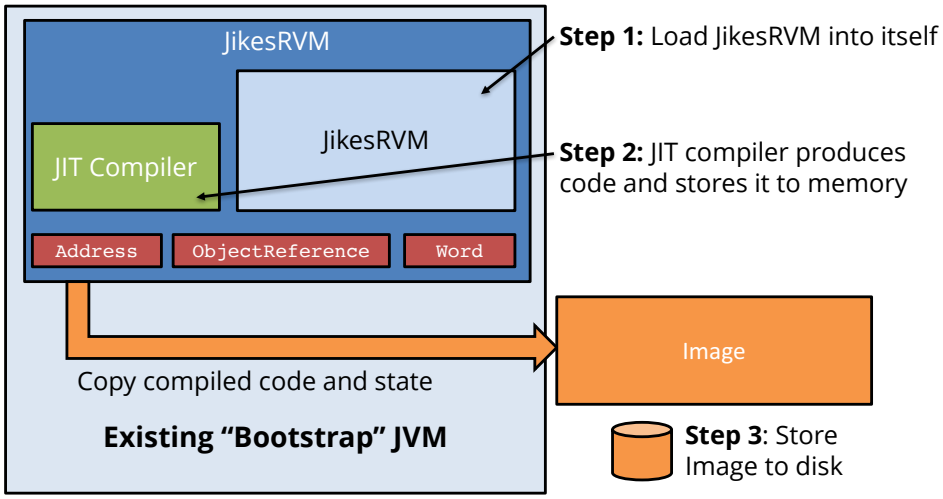


Figure 8.1: *Building the JikesRVM.* Jikes uses a meta-circular design and is written in Java. To bootstrap Jikes, it is first loaded into an existing JVM, and then loads itself. This results in Jikes's JIT compiler compiling Jikes's own code, which can then be stored to disk.

managed-language research. Jikes is a VM for Java, and is representative of other managed-runtime systems. We ported JikesRVM and its non-optimizing *Baseline* JIT compiler to RISC-V[1]. To our knowledge, this results in the first RTL-based full-system platform for hardware-software research on Java applications that allows modification of the entire hardware and software stack. In the following section, we describe our port.

8.3.1 Jikes's Software Design

In order to make the runtime system easy to modify, JikesRVM embraces object-oriented design principles and is written in Java. This design is often called a *meta-circular* runtime system, a runtime system written in the same language it executes.

This approach introduces new challenges, as Java is not intended for the low-level system programming required to implement a runtime system such as a JVM. Jikes solves this problem by providing a library called *VM Magic*, with classes representing low-level primitives such as pointers (`Address`) or references (`ObjectReference`). From a Java perspective, these primitives are normal objects with methods such as `Address.loadInt(addr)`. However, Jikes's own JIT compiler detects them and handles them specially.

8.3.2 Bootstrap Process

JikesRVM requires an existing "bootstrap" JVM, such as OpenJDK's Hotspot JVM (Figure 8.1). To compile Jikes, it is first loaded into this existing JVM, as a normal Java program

[1]The lack of an optimizing compiler means that we cannot compare against highly tuned systems directly, but need to account for this difference when conducting experiments. We show an example in Section 9.7.

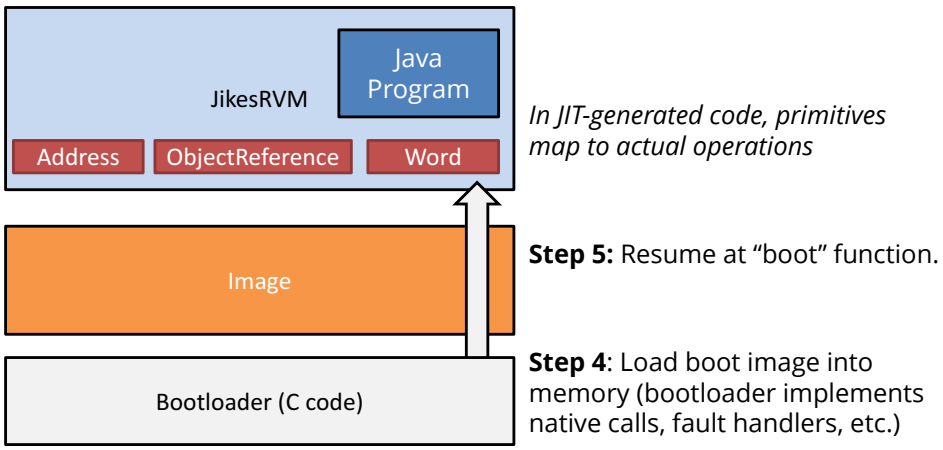


Figure 8.2: *Running the JikesRVM.* To run the JVM, a small "bootloader" program maps the image that was generated during the bootstrap process into the virtual address space. This image contains JITed code of the essential classes of the JVM, as well as a basic heap to operate on. The bootloader then sets up a Java stack and jumps to the JVM's entrypoint.

where the VM Magic primitives are regular objects with an implementation that emulates their intended behavior. Once JikesRVM runs in the bootstrap VM, it loads an instance of itself, which results in Jikes's own classes being loaded and compiled by its JIT compiler (this initial set of classes is called *primordials*). However, as this is now Jikes's JIT and not the bootstrap VM's, it will detect calls to VM Magic classes and replace them with the actual machine code executing low-level operations, such as memory stores.

In a final step, the instantiated objects belonging to the nested JikesRVM instance – including their JIT-compiled code – are taken and copied into an image, which is then stored to disk. This image now contains compiled code for all of Jikes's core classes, which can be executed without the bootstrap JVM in place.

8.3.3 Running Jikes RVM

Once Jikes has been compiled, it can be run by executing a small bootloader program written in C, which takes the image generated during the boostrapping process and maps it into its address space (Figure 8.2). This part of the address space represents the initial heap that the JVM is executing on. The bootloader then sets up the Java stack and jumps into a boot function that initializes the different components of the JVM. This process involves many steps and requires loading and executing initializers for 93 classes.

Once the JVM has booted up, it parses the command line arguments, uses them to determine a `.jar` or `.class` file to load, and then jumps into the main function of the program. Jikes will almost exclusively operate without intervention from the bootloader at this point, except for handling traps such as `null` pointer exceptions, spawning new threads and executing low-level functions such as writing to the console.

8.4 Porting the Jikes RVM to RISC-V

Porting JikesRVM to a new ISA is complicated by Jikes's meta-circular nature. Luckily, the JVM already supports two ISAs (x86 and PowerPC), and therefore has infrastructure in place to factor out ISA-specific portions of code, such as the assembler, compiler, native-function interface or stack walker. Porting JikesRVM therefore primarily required creating RISC-V implementations of these different components. Overall, our port involved modifications to 86 files and added around 15,000 lines of code.

8.4.1 Bringing up the Environment

The first step in porting JikesRVM was to bring up an environment that contains all of its dependencies. Specifically, this included a Linux distribution with a basic set of tools and libraries, such as `glibc` and `bash`. JikesRVM also requires compiling the GNU Classpath class library for a RISC-V target, which further depends on various different libraries.

To facilitate building these different dependencies, we ported the Yocto Linux distribution generator to RISC-V [189]. Yocto provides an environment that can cross-compile the Linux kernel and a range of packages on a host system, and generates an image that can then be booted in a RISC-V emulator or on actual RISC-V hardware. We used Yocto to generate an image which we then used as the environment to run JikesRVM in `riscv-qemu`.

In addition to generating the image, targeting JikesRVM to RISC-V also required us to have the cross-compiler and libraries available during the build process, to compile components such as the bootloader or the C libraries backing GNU Classpath. Yocto facilitates this by creating an *SDK*, which is a package that includes the entire cross-compile toolchain and development packages such as common libraries or `autoconf`. This SDK can be installed on any machine and contains a script that adds the cross-compilers to the current environment. Using a Yocto SDK provides us with all the tools and libraries we need to build Jikes, without setting up a full RISC-V development environment.

8.4.2 Debugging Infrastructure

To achieve a fast compile loop, we used a Python script that cross-compiles JikesRVM on an x86-64 host system, copies the output into the Yocto-generated image and runs this image in QEMU. We also modified the image with a custom `/etc/inittab` script that launches JikesRVM, redirects the output into a file and then shuts down the QEMU instance. This approach provided us with a fast turnaround for debugging.

After setting up these scripts, the next step consisted of porting JikesRVM's bootloader code. The code only includes a small number of architecture-dependent portions, specifically the assembly code that sets up the Java stack and jumps into a Java function.

Once this step was completed, the next task was to port the JIT compiler. To do this incrementally, we added test code at the beginning of the JVM's boot function (`VM.boot()`), which is the first function the bootloader jumps into after setting up the stack. This allowed

```
def make_jikesrvm():
  for name in namelist:
    args = arguments[name]

    funargs = []
    substargs = []

    register_order = ['rd', 'rs1', 'rs2', 'rs3']

    # loads/stores have different order of registers
    if name in ['lb', 'lh', 'lw', 'ld', 'flw', 'fld', 'sb', 'sh', 'sw', 'sd', 'fsw', 'fsd']:
      register_order = ['rd', 'rs2', 'rs1']

    fcvt_float2int = []
    fcvt_int2float = []
    fcmp = []

    for f in ['d', 's']:
      for i in ['l', 'w', 'lu', 'wu']:
          fcvt_float2int.append('fcvt.%s.%s' % (i,f))
          fcvt_int2float.append('fcvt.%s.%s' % (f,i))

      for instr in ['fle', 'flt', 'feq']:
        fcmp.append('%s.%s' % (instr, f))

    for r in register_order:
      if r in args:
        if name.startswith('f'):
                if name in ['flw', 'fld', 'fsw', 'fsd'] and r == 'rs1':
                        funargs.append('GPR ' + r)
                elif name in fcvt_float2int and r == 'rd':
                        funargs.append('GPR ' + r)
                elif name in fcvt_int2float and r == 'rs1':
                        funargs.append('GPR ' + r)
                elif name in fcmp and r == 'rd':
                        funargs.append('GPR ' + r)
                elif name in ['frflags', 'fsflags', 'fsflagsi']:
                        funargs.append('GPR ' + r)
                else:
                        funargs.append('FPR ' + r)
        else:
                funargs.append('GPR ' + r)

        substargs.append('%s.value() << %d' % (r, arglut[r][1]))
        args.remove(r)
```

```
  public final void emitLH(GPR rd, GPR rs1, int imm12) {
    int mi = 0x1003 | rd.value() << 7 | rs1.value() << 15 | make_imm12(imm12);
    mIP++;
    mc.addInstruction(mi);
  }

  public final void emitLW(GPR rd, GPR rs1, int imm12) {
    int mi = 0x2003 | rd.value() << 7 | rs1.value() << 15 | make_imm12(imm12);
    mIP++;
    mc.addInstruction(mi);
  }

  public final void emitLD(GPR rd, GPR rs1, int imm12) {
    int mi = 0x3003 | rd.value() << 7 | rs1.value() << 15 | make_imm12(imm12);
    mIP++;
    mc.addInstruction(mi);
  }

  public final void emitLBU(GPR rd, GPR rs1, int imm12) {
    int mi = 0x4003 | rd.value() << 7 | rs1.value() << 15 | make_imm12(imm12);
    mIP++;
    mc.addInstruction(mi);
  }

  public final void emitLHU(GPR rd, GPR rs1, int imm12) {
    int mi = 0x5003 | rd.value() << 7 | rs1.value() << 15 | make_imm12(imm12);
    mIP++;
    mc.addInstruction(mi);
  }

  public final void emitLWU(GPR rd, GPR rs1, int imm12) {
    int mi = 0x6003 | rd.value() << 7 | rs1.value() << 15 | make_imm12(imm12);
    mIP++;
    mc.addInstruction(mi);
  }

  public final void emitSB(GPR rs2, GPR rs1, int imm12) {
    int mi = 0x23 | rs2.value() << 20 | rs1.value() << 15 | make_imm12hi(imm12);
    mIP++;
    mc.addInstruction(mi);
  }

  public final void emitSH(GPR rs2, GPR rs1, int imm12) {
    int mi = 0x1023 | rs2.value() << 20 | rs1.value() << 15 | make_imm12hi(imm12);
    mIP++;
    mc.addInstruction(mi);
  }
```

Figure 8.3: *Part of the Python script that auto-generates the assembler, and the code that it emits.* Note that registers are typed and common flags are replaced by enums.

us to first implement simple Java opcodes such as integer operations, static function calls or conditionals, and then incrementally extend our implementation.

8.4.3 Porting the Assembler

Before we could start porting the JIT compiler, we had to implement an assembler that can generate RISC-V instructions. While Jikes's assemblers for PPC and x86 are hand-written, we were able to automate this process for RISC-V, by using the open-source `riscv-opcodes` repository [188]. This repository provides a machine-readable version of all RISC-V instructions. Building on a Python script that is available as part of `riscv-opcodes`, we generated most of Jikes's assembler automatically, creating an `emitX()` function for every instruction `X` in the instruction set (Figure 8.3).

One case that needed special attention were branches. The JIT compiler often generates branches with placeholders for the target offset, which are rewritten at a later point. In RISC-V, we had to be careful to distinguish between short branches (that fit into the branch instruction's 12-bit offset) and general branches, for which we need to emit a branch followed by a `jal` instruction. The assembler provides functions to emit both types of branches. If the target is unknown in advance, a general branch is emitted.

8.4.4 Porting the JIT Compiler

The non-optimizing JIT compiler is template-based. It contains a set of functions corresponding to Java bytecode instructions. Each of these functions calls into the assembler to emit a RISC-V instruction sequence that implements the specific Java bytecode. The JIT compiler also provides instruction sequences for the VM Magic functions described in Section 8.3.1. Finally, the JIT compiler provides functions that emit code for special circumstances, such as function prologues, epilogues and yield points. Yield points are emitted at certain points throughout the program and check whether a thread is supposed to block – e.g., because of garbage collection or revoking a biased lock.

We started by implementing prologues, epilogues and several basic integer instructions. This allowed us to run small test programs by injecting them into Jikes's boot function. However, for programs that were more complex, we required additional information to debug the execution. Due to the lack of debug information, it is difficult to debug this code with traditional debuggers such as GDB. We therefore chose a different approach.

We instrumented the JIT compiler to emit a trace of its execution. For each executed bytecode, we print the name of the opcode, the corresponding instruction sequence, and the current state (i.e., the top elements of the stack). We achieve this by prefixing the instruction sequence for each bytecode with an invalid load that will trigger a `SEGFAULT`. Additionally, we also include auxiliary information:

```
0x...000: LD X0, 1024(X0) # SEGFAULT
0x...004: (Number of instructions)
0x...008: (Opcode)
0x...00c: (Stack Offset)
```

When the load is reached, it will trigger an exception that can be caught in the bootloader program. The bootloader then reads the auxiliary information and outputs the desired debug information, including a disassembled version of the instructions associated with this bytecode (Figure 8.4). Note that we did not have to write our own disassembler to achieve this. Instead, we printed `DASM(INST)` to the standard output, and redirected the output to the `spike-dasm` program that ships with the Spike ISA simulator.

As the test programs grew, we found that the debug output became too cumbersome to work with. We therefore added a modification to JikesRVM which allows us to only selectively inject this instrumentation. Specifically, we added a `@SoftwareBreakpoints` annotation that can be attached to a function in JikesRVM. If this annotation is present, the instrumentation code will be injected by the JIT compiler (and we will receive a trace of its execution), otherwise the function will be compiled normally.

8.4.5 Foreign-Function Calls

One of the most challenging aspects of porting Jikes was to support foreign-function calls. Jikes provides two mechanisms to call into C code: JNI calls (which is Java's mechanism to

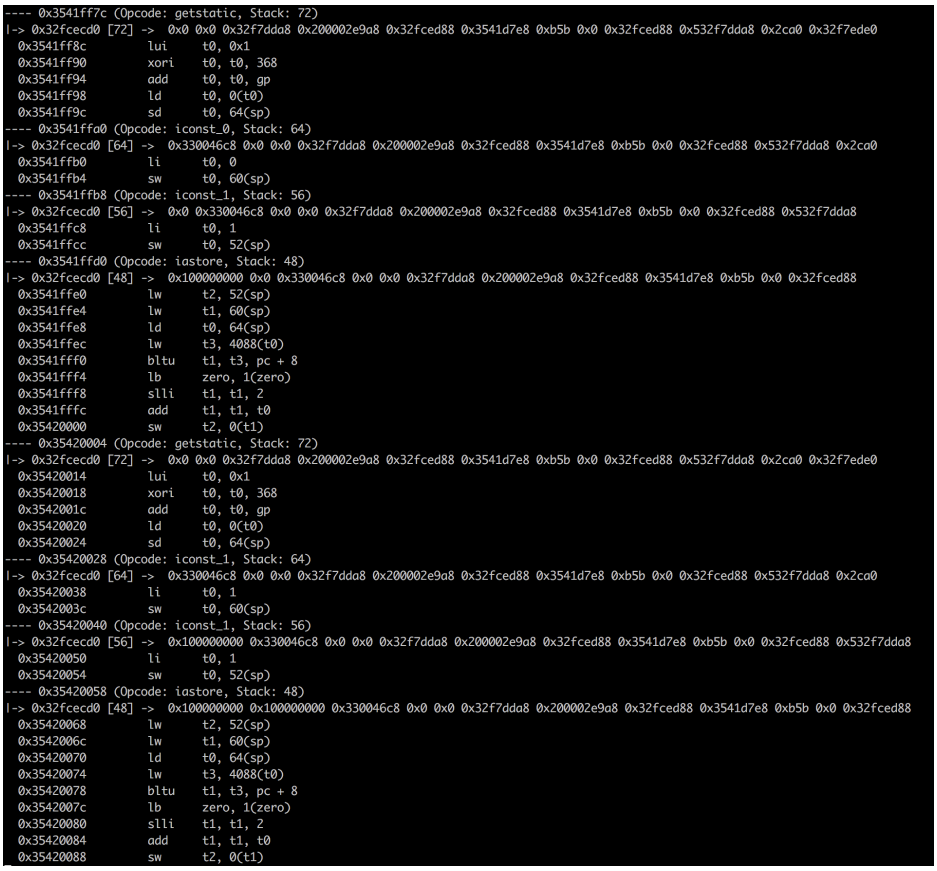


Figure 8.4: Debug output for the JIT compiler.

call into C functions) and a simpler mechanism named *syscalls*. JNI is a complex framework that enables calls in both directions (C to Java and Java to C). This makes it possible that a mix of both Java and C stack-frames can co-exist on the same stack. Jikes therefore needs to be able to unwind both types of frames for delivering exceptions, and scan them for spilled pointers at the beginning of GC passes. This means that JNI calls require maintaining a side table of pointers for stack scanning, check for yield points when crossing a language barrier, and support the full C calling convention.

Avoiding this complexity, Jikes's syscalls mechanism is intended to implement simple functions such as writing bytes to a stream or executing math functions like `sqrt`. Instead of supporting the full calling convention, it only supports simple calls, does not check for yield points and cannot call back into Java. For debugging purposes, we found it important to implement syscalls early. Meanwhile, as JNI functions require a large amount of work, we decided to leave them to the end. Note that syscalls are emitted by the JIT, while JNI calls are generated by a special `JNICompiler`. Implementing syscalls is sufficient to run test programs with simple command line output, provided they are added to `VM.boot()`.

8.4.6 Exceptions & Run-time Checks

Java checks for a number of corner cases and triggers exceptions if necessary, such as array bounds checks or divide-by-zero checks. We found that the best approach in RISC-V was to trigger exceptions through loads to invalid addresses. This causes execution to drop back into the bootloader, where we can determine which exception was triggered (based on the failing instruction) and then jump into a Java function that delivers the exception and unwinds the stack. The exception delivery itself requires architecture-specific code for unwinding both Java and JNI (i.e., native) stack-frames.

8.4.7 Additional Features

While the features discussed so far enable simple test programs and executing a large part of the `VM.boot` function, completing the full boot sequence requires a large number of architecture-specific features, including locks, lazy compilation trampolines, dynamic bridges (which are necessary for JIT-compiling a function by running the JIT on the same stack, and then transparently transferring execution to the JITed code) and interface method tables (which require synthesizing architecture-specific code to traverse a search tree). Completing the boot sequence therefore requires a mostly complete port.

8.4.8 Summary

Our port of the baseline compiler is complete enough to pass all of the unit tests that are part of JikesRVM and runs the subset of DaCapo [37] benchmarks that are supported by our version of JikesRVM (`avrora`, `luindex`, `lusearch`, `pmd`, `sunflow`, `xalan`). We successfully ran these benchmarks both in simulation and on Rocket Chip mapped to an FPGA.

Figure 8.5 shows one of these benchmarks running on a RISC-V Rocket Chip instance captured from an FPGA setup. Being able to run DaCapo benchmarks gives us a high degree of confidence in the functional correctness of our port, as the DaCapo suite consists of large and complex benchmarks. For example, the benchmarks presented here include a raytracer, the Lucene search engine, and a code analyzer.

8.5 Running Java on RISC-V Hardware

With a complete port of JikesRVM, we now have the ability to run Java workloads on RISC-V systems and modify both the JVM and the underlying hardware. To demonstrate this experimental setup, we ran JikesRVM on Rocket Chip in FPGA-based simulation.

We use Xilinx ZC706 development boards, which are comprised of an Zynq XC7Z045 FPGA with 8 GB of fabric-attached DRAM. We use an FPGA-based simulation framework based on an early version of MIDAS [131], with timing models to simulate DRAM accesses. This setup simulates a single Rocket 5-stage in-order CPU, with 16 KB L1 instruction and data caches and a simulated 1 MB L2 cache.

Using this setup, we achieved effective simulation speeds of 10 MIPS and more. We simulate a design with an operating frequency of 1 GHz, a L2 latency of 23 cycles and a DRAM latency of 80 cycles. Running these experiments for the DaCapo benchmarks (default input size) allowed us to collect performance data, as well as instruction counts and other metrics on our platform. Executing the full set of benchmarks takes over 1.2 trillion instructions, which would take 35 days if simulated at 400 KIPS.

The following table presents the number of dynamic instructions for each of the benchmarks, as well as their simulated runtime in this setup:

Benchmarks	**Instructions (B)**	**Runtime (s)**
avrora	118.0	311.8
luindex	47.7	103.5
lusearch	263.5	597.2
pmd	158.5	346.8
sunflow	504.8	1,352.9
xalan	190.8	466.4

JikesRVM is configured to use the *Mark & Sweep* garbage collector. With a 100 MB maximum heap size, the JVM spends the following fraction of time in GC for each benchmark:

Benchmarks	**GC Pauses**	**Time in GC**
avrora	10	6%
luindex	8	13%
lusearch	90	35%
pmd	26	30%
sunflow	52	9%
xalan	39	27%

While this implies that the JVM's performance can be improved substantially, note that the baseline compiler's primary responsibility is to run code that executes rarely. In order to generate performance-competitive code, we need to port the optimizing JIT compiler as well (for which the baseline compiler is a prerequisite).

8.6 Research Case Study

We believe that FPGA-based full-system simulation of JikesRVM workloads on RISC-V hardware enables studies that are difficult to perform in a traditional setup. Specifically, we can modify the software stack as well as the underlying hardware, while collecting cycle-accurate numbers that can capture fine-grained interactions for full workload executions with short simulation times. This enables design-space explorations that modify both hardware and software layers, and detailed instrumentation of the entire system.

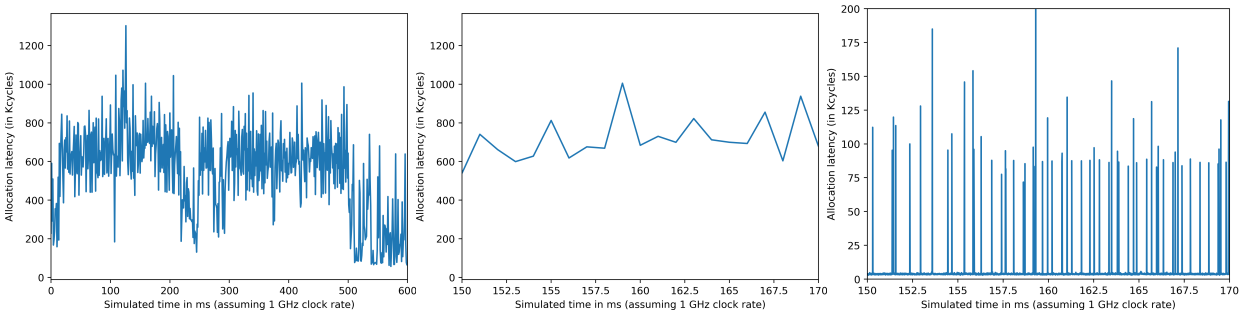
```
Setting default thread count for MMTk to minimum of default thread count 1 and
 maximal thread count 2147483647 supported by current GC plan.
New default thread count value is 1
Setting actual thread count for MMTk to minimum of desired thread count 1 and
maximal thread count 2147483647 supported by current GC plan.
New actual thread count is 1
[GC 1 Start 33.97 s   20480KB -> 12424KB   11825.91 ms]
[GC 2 Start 73.41 s   24576KB -> 17136KB   13466.31 ms]
[GC 3 Start 137.59 s   31740KB -> 20040KB   15324.46 ms]
[GC 4 Start 176.83 s   27688KB -> 20180KB   15999.70 ms]
===== DaCapo head-runknown avrora starting =====
[GC 5 Start 241.26 s   37888KB -> 30732KB   19705.49 ms]
[GC 6 Start 319.30 s   50176KB -> 36896KB   22377.38 ms]
[GC 7 Start 422.90 s   63488KB -> 45776KB   26419.36 ms]
[GC 8 Start 854.55 s   78848KB -> 47232KB   30183.02 ms]
[GC 9 Start 1941.52 s   79872KB -> 43472KB   30285.10 ms]
[GC 10 Start 3053.29 s   79872KB -> 43648KB   31433.10 ms]
===== DaCapo head-runknown avrora PASSED in 2849150 msec =====
[End 3084.72 s]
```

Figure 8.5: *Output of JikesRVM running one of the DaCapo benchmarks, with verbose GC output.* This is a processed log file – all MIDAS debug output was removed.

To demonstrate these capabilities, we conduct a study inspired by a 2005 paper by Hertz and Berger [101]. In order to investigate trade-offs between manual and automatic memory management, the authors instrumented JikesRVM to extract a trace of allocated memory addresses, and – in a second pass – inject another trace of addresses produced by an oracle. The authors found that this was difficult to achieve in software, as the software instrumentation led to a 2-33% perturbation in execution time, which was larger than the effect they were measuring. They therefore used a software architectural simulator.

This problem is common: Many interactions in managed-runtime systems are fine-grained and therefore difficult to measure. One example of these interactions are memory allocations, which occur frequently but complete quickly most of the time. We are often interested in the causes of long allocations, and want to measure time spent in allocation routines.

To record these allocations in a traditional system, we would have two options: we could either use an instrumentation-based approach or a sampling-based approach. However, the former introduces observer effects and perturbs the execution time, while the latter traditionally achieves low sampling frequencies and can hide important details. Figure 8.6 shows an example of this: While sampling at 1 KHz helps us understand the macro-behavior of the application (Figure 8.6a), it does not tell us about individual allocations (Figure 8.6b). To gain additional insight into the behavior of the memory allocator, we need to be able to record every single allocation latency – without perturbing the execution time.



(a) 600ms slice sampled at 1 KHz **(b)** 20ms slice sampled at 1 KHz **(c)** The same 20ms in MIDAS

Figure 8.6: *Time spent in allocation routines per sampling interval.* We contrast an approach that samples time spent in allocation at a 1 KHz-granularity with using the MIDAS simulation framework to record every allocation in hardware without introducing observer effects to the application (numbers are from the `pmd` DaCapo benchmark). Results were collected in a single run and aggregated to demonstrate the effect of different sampling rates.

One approach to this problem would be to use a system like SHIM [237], which enables high-resolution sampling while minimizing the observer effect. It achieves this by running an observer thread in a second hardware context on an SMT-enabled multiprocessor, sampling at a resolution of 1,200 cycles at only 2% overhead. SHIM can also perform measurements at an extremely fine-grained solution of ≈ 15 cycles, but then the perturbation becomes large, at an overhead of 61%. While this is sufficient to understand program behavior, it is limited to counters exposed by the hardware. Specifically, SHIM is designed for existing SMT processors and cannot instrument arbitrary signals in a modifiable hardware design.

Infrastructure such as JikesRVM running on Rocket Chip enables us to instead instrument arbitrary signals in hardware: by adding on-chip buffers to the RTL design, we can record every allocation in the execution of the program, and produce a detailed trace without perturbing the execution time. In this case, we record the start and end time, as well as the size class and memory address associated with every allocation.

Figure 8.6c shows the result: by looking at the duration of every allocation, we see that most allocations complete in $\approx 4,000$ cycles, while some allocations take 10–100$\times$ longer. This gives us insight into the behavior of the memory allocator (in this case, a segregated free-list allocator). In the common case, the allocator consumes a set of per-size-class free lists, and completes quickly if a cell is available on the list. If not, the allocator has to remove a new block from the global free list, zero the block's memory, and create a new free list.

There are other insights that can be gained from this trace as well. For example, looking at allocations for the same size class and counting how many of them use the fast path, we can deduce the amount of memory fragmentation. We can also analyze locality: looking at the addresses that are returned by the allocator (Figure 8.7), we see that subsequent allocations to the same size class are typically contiguous, but overall locality is low. This confirms that segregated free-list allocators produce poor locality.

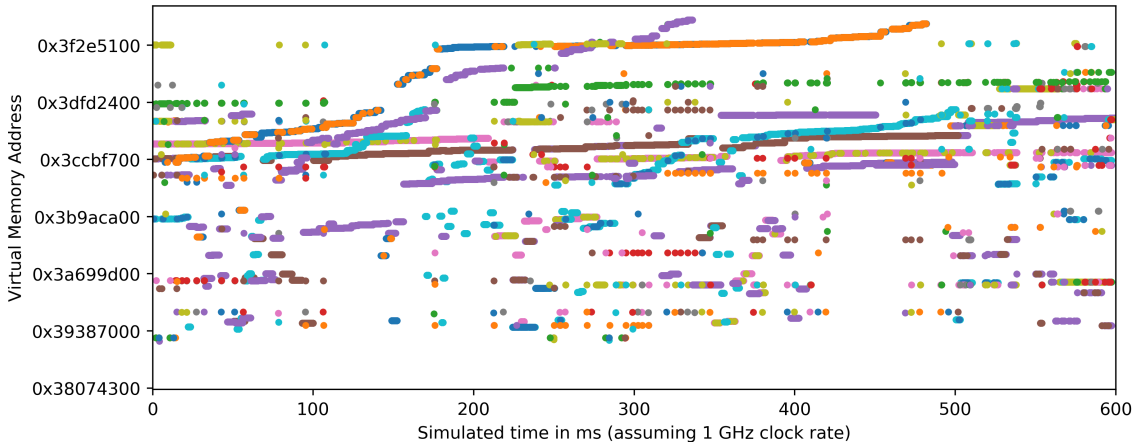


Figure 8.7: *Virtual addresses returned by the JikesRVM free-list allocator over time.* Colors indicate the allocation size class. Segregated free list allocators provide poor locality.

Memory allocators are only one example of experiments that are possible with this infrastructure, and we believe that it will open up new research directions in a wide range of areas. We are particularly interested in using this infrastructure to evaluate our hardware-assisted garbage collector design from Chapter 7.

8.7 Summary

In this chapter, we presented our port of JikesRVM to RISC-V, and demonstrated how it runs on FPGA-based RISC-V hardware. We believe that the combination of a managed-runtime system and hardware that can be easily modified will enable new kinds of hardware-software research, as demonstrated by our case study.

Equipped with this infrastructure, we can now implement our hardware-assisted GC design by modifying JikesRVM to use our bidirectional object layout, add our hardware extensions to Rocket Chip and add a driver to Linux to connect them to one another. The next chapter will describe the prototype that resulted from using this infrastructure, and shows a performance evaluation and design space exploration for this prototype.

Chapter 9

The GC Accelerator Prototype

In this chapter, we present a prototype of a specific incarnation of our general Hardware GC design, built using the research infrastructure from the previous chapter. We demonstrate an end-to-end RTL prototype, integrated into a Rocket Chip RISC-V System-on-Chip (SoC) executing full Java benchmarks within JikesRVM running on top of Linux on FPGAs. Our prototype performs the mark phase of a tracing GC at 4.2× *the performance of an in-order CPU, while using only 18.5% the area (an amount equivalent to 64KB of SRAM). By prototyping our design in a real system, we show that our accelerator can be adopted without invasive changes to the SoC, and estimate its performance, area and energy.*

9.1 Our Prototype Implementation

Following a description of our general Hardware GC scheme in Chapter 7, we now describe our specific implementation of this general design. We focus on the accelerator portion and implemented an RTL prototype of the GC unit within a Rocket Chip [18] SoC. As in our GPU garbage collector, the unit is integrated with JikesRVM, using our JikesRVM RISC-V port described in the previous chapter. We evaluate our prototype in a stop-the-world setting, but as shown in Section 7.6.1, it could be used in a concurrent collector as well. Figure 9.1 shows an overview of how our design is integrated into the system.

We start by presenting our prototype in detail, describe how it is integrated into Rocket Chip and the modifications that we had to make to the hardware and software layers. We then present a detailed evaluation of this design, to show that it outperforms a CPU by 4.2× on the mark phase, while only consuming 18.5% the area and reducing the energy consumption per GC cycle by 14.5% . By building out the full design and integrating it into a real SoC, we also show that it is non-invasive enough to be integrated into a realistic system. This allows us to evaluate the potential and the feasibility of integrating such an accelerator into real-world production SoCs.

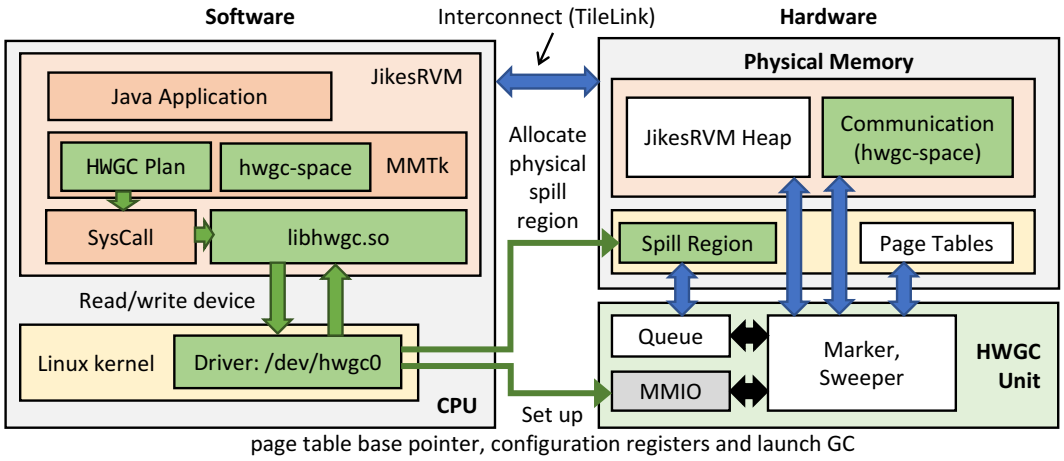


Figure 9.1: *System Integration.* Green boxes refer to components we added to the system. Identically colored boxes on the left and the right refer to the same memory spaces.

9.2 JikesRVM Modifications

We build on our RISC-V port of JikesRVM from Chapter 8. In order to integrate this port with our hardware unit, we had to make several modifications to JikesRVM. These modifications are representative of changes any language runtime system would have to make to target our hardware unit, and are limited to the language-runtime level (i.e., nothing changes from the perspective of an application running on top of the JVM).

9.2.1 Memory Management Toolkit (MMTk) Integration

MMTk [36] is the framework within JikesRVM that implements most memory-management functionality. By factoring out all memory-management related functionality into a single frameworks, MMTk makes it easier to change GC logic within the runtime system.

The integration of our GC unit is reminiscent of the way we integrated our GPU garbage collector into Jikes (Section 6.4). Specifically, we implemented our collector as a new `HWGC` *plan* in MMTk. A plan describes the spaces that make up the heap, as well as allocation and GC strategies for each of them. We base our work on the `MarkSweep` plan, which consists of 9 spaces, including large object space, code space and immortal space.

Our collector traces all of these spaces, but only reclaims the main MarkSweep space (which contains most of the newly allocated objects). The other spaces, such as the code space, are still managed by Jikes (however, there is no fundamental reason they could not use the GC unit). This is particularly relevant in the case of the large object space, which by default uses a treadmill-based collector (i.e., objects are moved into the new space as they are discovered, not during a sweep phase). We therefore had to modify this collector to use the mark bits generated by our unit instead.

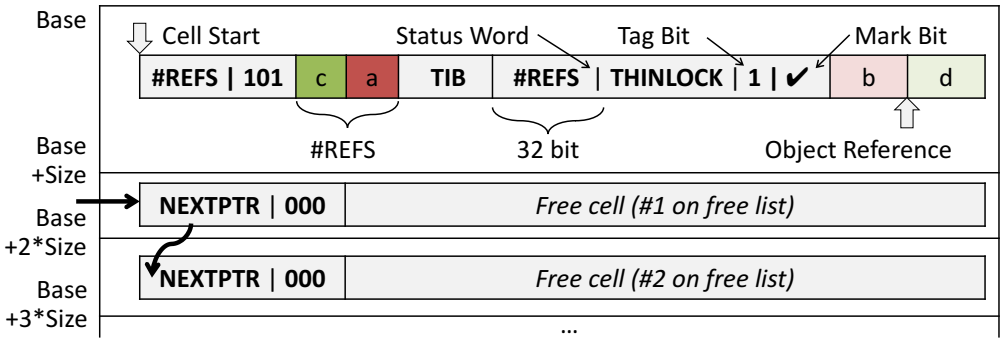


Figure 9.2: *JikesRVM Integration.* Memory is divided into blocks, which are split into equisized cells (each field represents a 64-bit word). Cells either include objects or free list entries. References point to the second object field to facilitate array offset calculation.

At a high level, the `HWGC` plan uses a similar strategy to our GPU collector, redirecting calls to perform the mark and sweep portions of the GC to C functions which are running within the JVM's address space and interact with our GC unit.

We also modified the root scanning mechanism in JikesRVM to not write the references into the software GC's mark queue but instead write them into a region in memory that is visible to the GC unit (`heap-space`). We implement this region as a new space within our garbage collector, and ensure that this memory is fully mapped and contiguous (as it will be accessed by the GC unit, which cannot currently handle page faults).

9.2.2 Bidirectional Object Layout

In contrast to our reference-graph-based approach on the GPU (Chapter 6), we modified JikesRVM to fully implement the bidirectional object layout described in Chapter 7. JikesRVM factors the object layout into a small number of classes. In particular, a `FieldLayout` class is responsible for calculating the offsets of all fields within a class, and a `JavaHeader` class is responsible for interpreting and modifying header fields. Beyond these classes, several other changes were required for the JVM to work correctly with the new layout, including fixing a bug in the thin lock implementation that was not triggered without our changes. We also had to ensure that weak references are treated as regular references in our layout.

One important question when implementing the bidirectional layout is where to store the number of references. We found 34 unused bits in the header's status word (Figure 9.2). We use 32 of these bits to store the number of references in an object (for arrays, we set the MSB of these 32 bits to 1 to distinguish them). The remaining two bits are used for the actual mark bit and for a tag bit that we set to 1 for all objects (this will be useful for the reclamation unit). Furthermore, we also replicate the reference count at the beginning of the array, which is necessary to enable linear scans through the heap.

We had to be particularly careful how this layout integrates with Jikes's memory allocator. Jikes's Mark & Sweep plan uses a segregated free list allocator: Memory is divided into blocks, and each block is assigned a size class, which determines the size of the cells that the block is divided into. Each cell either contains an object or a free list entry, which link all empty cells together. During the sweep phase, the collector has to be able to distinguish between objects and empty cells containing free list entries (this is the purpose of the previously mentioned tag bit). The collector then has to generate the same kinds of free lists and make them available for the memory allocator to use (by writing them into a list of head pointers).

9.3 Integration into Rocket Chip

As mentioned previously, we prototyped our design in the context of Rocket Chip (Section 8.2). By implementing our accelerator in an SoC that is used in commercial projects, we show that it is realistic, non-invasive and can be adopted in an existing design. Building on Rocket Chip also facilitated our development, as it provided us with a library of parameterizable components such as cores, devices, caches and other functional units. For example, we were able to reuse Rocket's TLB and page-table walker (PTW) implementations.

Rocket Chip is written in Chisel [22], a hardware description language embedded in Scala. Chisel is not a high-level synthesis tool but operates directly at the RTL-level. By being embedded in Scala, it makes it easier to write parametrizable generators. Rocket Chip takes advantage of this flexibility to allow developers to describe SoC instances at a high level. These SoC designs connect devices and cores together using a shared-memory interconnect called TileLink, which automatically negotiates communication contracts between endpoints using a framework called *Diplomacy* [58].

To integrate our GC accelerator into this system, we created a new Chisel module which we register as a TileLink client and connect it to the *system bus* (which is the Rocket Chip equivalent of the Northbridge). We also connect a set of memory mapped (MMIO) registers to the *periphery bus* (Southbridge), for configuration and communication with the CPU. Diplomacy automatically infers the necessary protocol logic, connects the module into the SoC and produces a device-tree structure that can be used to configure Linux drivers.

9.4 Mark Unit Implementation

The mark unit closely follows the design in Figure 7.6. We explored several versions of the marker and tracer: one version connects to a shared 16KB data cache, one partitions this cache among the units and one directly talks to the TileLink interconnect. As the GC unit operates on virtual addresses, we added a page-table walker and TLBs to the design (the PTW is backed by an 8KB cache, to hold the top levels of the page table).

At the beginning of a GC, a *reader* copies all references from the *hwgc-space* into the mark queue. Then, the marker and tracer begin dequeuing references from their respective

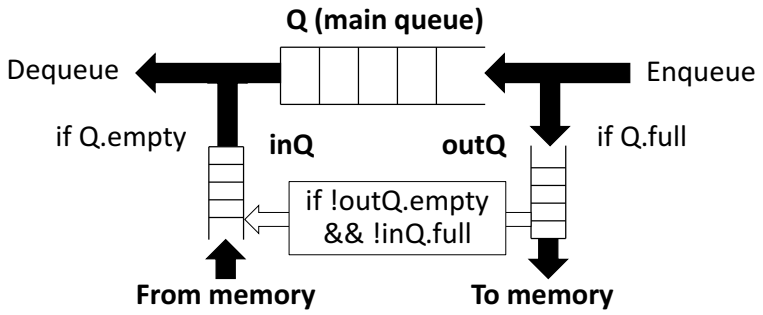


Figure 9.3: *Mark Queue Spilling.* When the main mark queue (`Q`) fills up, requests are redirected to `outQ`, which is written to memory. When `Q` empties, these requests are read back from memory and buffered within `inQ`.

input queues and pass their results to their output (the queues exert back-pressure to avoid overflowing, and marker and tracer can only issue requests if there is space). This repeats until all queues and the hwgc-space are empty.

9.4.1 Mark Queue Spilling

As the mark queue can theoretically grow arbitrarily, we need to spill it to memory when it fills up. Figure 9.3 shows our approach. We add two additional queues, *inQ* and *outQ*. A small state machine writes entries from *outQ* into a physical memory range not shared with JikesRVM, and reads entries from this memory into *inQ* if there is space (and *outQ* is empty). We always give priority to the main queue, but if it is full, we enqueue to *outQ* (when it is empty, we dequeue from *inQ*). When *outQ* reaches a certain fill level, we assert a signal that tells the tracer to stop issuing memory requests, to avoid *outQ* from filling up. If there are elements in *outQ* and free slots in *inQ*, we copy them directly into *inQ*, reducing the number of memory requests. By prioritizing memory requests from *outQ*, we avoid deadlock.

9.4.2 Marker

We started out with a design that sends atomic memory operations (AMOs) to a non-blocking L1 cache. However, this limits the number of requests in flights (a typical design has 32 MSHRs). MSHRs operate on 64B cache lines, and need to store an entire request. In contrast, all requests in the marker are the same, operate on less than a cache line, and do not need to be ordered. We therefore built a custom marker that talks to the interconnect directly (Figure 9.4). We only hold a tag and a 64-bit address for each request, translate them using a dedicated TLB, send the resulting reads into the memory system and then handle responses in the order in which they return. For each response, we then also issue the corresponding write-back request to store the updated mark bit and free the request slot (note that we can elide write-backs if the object was already marked).

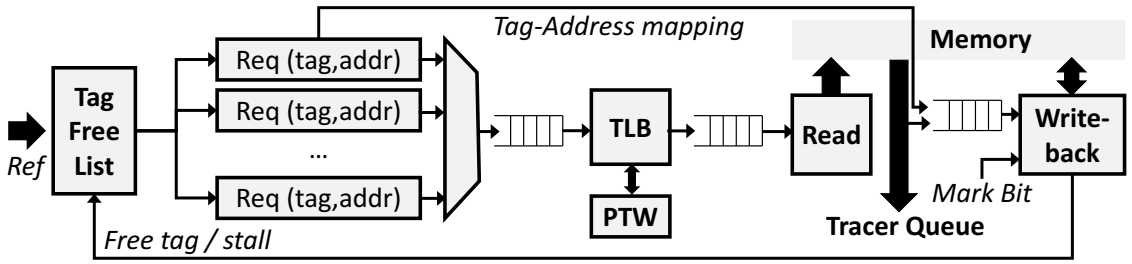


Figure 9.4: *Marker Microarchitecture.* Instead of using a cache with MSHRs, we manage our own requests, as they are identical and unordered.

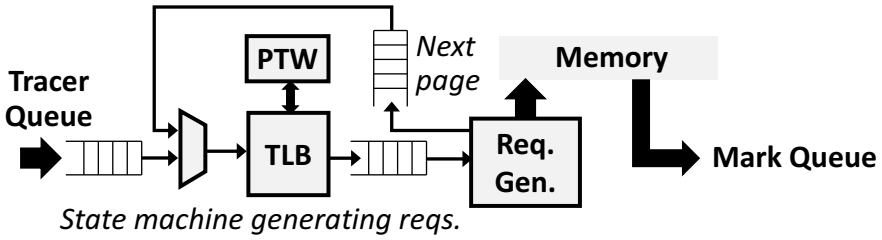


Figure 9.5: *Tracer Microarchitecture.* The *request generator* walks the reference section and issues the largest requests it can for the current alignment.

9.4.3 Tracer

We built a custom tracer that can keep an arbitrary number of requests in flight. After translating the virtual address of the object, it enters a *request generator*, which sends `Get` coherence messages into the memory system. Our interconnect supports transfer sizes from 8 to 64B, as long as they are aligned. If we need to copy 15 references (15×8 bytes) at `0x1a18`, we therefore issue requests of transfer sizes 8, 32, 64, 16 (in this order). Note that we need to detect when we reach a page boundary; in this case, the request is interrupted and re-enqueued to pass through the TLB again.

9.4.4 Address Compression

We operate on 64-bit pointers, but runtime systems do not typically use the whole address space. For example, our JikesRVM heap uses the upper 36 bit of each address to denote the space, and the lowest 3 bit are 0 since pointers are 64-bit aligned. Many runtime systems also use bits in the address to encode meta-data, which may be safely ignored by the GC unit. Our design provides a general mechanism to exploit this property: before enqueuing a reference to the mark queue, it can be mapped to a smaller number of bits using a custom function. The reverse function is applied when the object is dequeued. We demonstrate this strategy by compressing addresses into 32 bits, which doubles the effective size of the mark

queue and halves the amount of traffic for spilling. Runtime systems with larger heaps may use a larger number of bits instead (e.g., 48).

9.4.5 Mark Bit Cache

Most objects are only accessed once and therefore do not benefit from caching (Figure 9.11a). However, we found that there are a small number of objects that are an exception to the rule: about 10% of mark operations access the same 56 objects in our benchmarks. We therefore conclude that a small mark bit cache that stores a set of recently accessed objects can be efficient at reducing traffic. This has similarities to dynamic filtering, which has been shown to be effective in similar scenarios [96].

9.5 Reclamation Unit Implementation

In our prototype, we implement the simplest version of the reclamation unit, which executes a non-relocating sweep. As such, it does not require the forwarding table or read-barrier backend from Section 7.6.1. Each block sweeper receives as input the base address of a block, as well as its cell and header sizes. It then steps through the cells in a linear fashion.

The unit first needs to identify whether a cell contains an object or free list entry (recall Figure 9.2). It first reads the word at the beginning of the cell – if the LSB is 1, it is an object with the bidirectional layout. Otherwise, it is a next-pointer in a free cell, or a TIB for an object without references or an array. Based on each of these cases, we can calculate the location of the word containing the mark bit, which then allows us to tell whether the cell is free (i.e., the tag bit is zero), it is live but not reachable (the tag bit is one, the mark bit is not set) or contains a reachable object (both bits are set). In the first two cases, we rewrite the first word of the cell to add it to the free list, otherwise we skip to the next cell.

9.6 System-Level Integration

With the JikesRVM modifications and the GC unit in place, the missing part is to allow the two components to communicate. This communication is established through a Linux driver that we integrated into the kernel. This driver installs a character device that a process can write to in order to initialize the settings of the GC unit, initiate GC and poll its status. When a process accesses the device, the driver reads its process state, including the page-table base register and status bits, which are written to memory-mapped registers in the GC unit and used to configure its page-table walker. This allows the GC unit to operate in the same address space as the process on the CPU.

When the driver is initialized at boot time, it allocates a *spill region* in physical memory, whose bounds are then passed to the GC unit. This region has to be contiguous in physical memory and we currently allocate 4MB by default. To communicate with the driver, we

Processor Design (Rocket In-Order CPU @ 1 GHz)	
Physical Registers	32 (int), 32 (fp)
ITLB/DTLB Reach	128 KiB (32 entries each)
L1 Caches	16 KiB ICache, 16 KiB DCache
L2 Cache	256 KiB (8-way set-associative)
Memory Model (2 GiB Single Rank, DDR3-2000)	
Memory Access Scheduler	FR-FCFS MAS (16/8 req. in flight)
Page Policy	Open-Page
DRAM Latencies (ns)	14-14-14-47

Table 9.1: Rocket Chip Configuration

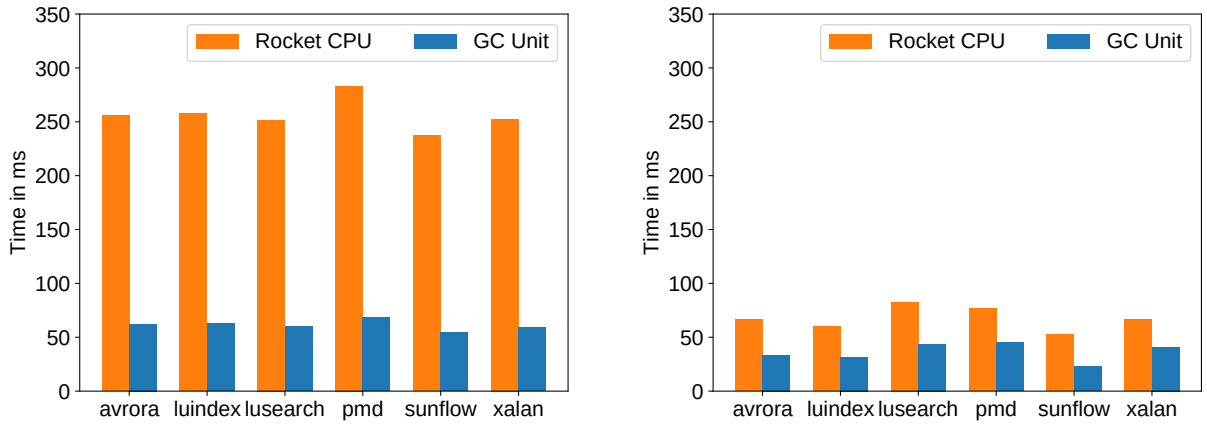
also extend JikesRVM with a C library (`libhwgc.so`). Our MMTk plan uses Jikes's SysCall foreign function interface to call into this C library, which in turn communicates with the device to configure the hardware collector (e.g., setting the pointer to the `hwgc-space`) and to initiate a new collection. By replacing `libhwgc` with different implementations, we can swap in a software implementation of our GC, as well as a version that performs software checks of the hardware unit (or produces a snapshot of the heap). This approach helped for debugging, as it allowed us to work on hardware and software modifications in parallel.

9.7 Evaluation

To evaluate our design, we ran it in FPGA-based simulation, using the simulation infrastructure described in Chapter 8. We run our design within Rocket Chip on MIDAS, using Amazon EC2 F1 instances [6]. This allowed us to run cycle-accurate simulation at effective simulation rates of up to 125 MHz (on Xilinx UltraScale+ XCVU9P FPGAs). While our target RTL executes cycle-accurately on the FPGA, we use MIDAS's timing models to simulate the memory system (in particular, DRAM and memory controller timing).

Table 9.1 shows the configuration of our Rocket Chip SoC and the parameters of the memory model that we are using. Note that we currently compare against an in-order *Rocket* core, rather than Rocket Chip's BOOM out-of-order core. Through preliminary analysis of running heap snapshots on an older version of BOOM with DRAMSim, we found that it outperformed Rocket on these workloads by only around 12% on average. This may seem surprising, but limited benefits of out-of-order cores over in-order cores for GC have been confirmed on Intel systems [43]. We therefore consider Rocket a reasonable baseline.

The goal of our prototype and evaluation is (1) to demonstrate the potential efficiency of our unit in terms of GC performance and area use, (2) characterize the design space of parameters, and (3) show that these benefits can be gained without invasive changes to the SoC. While we integrated our design into Rocket Chip, our goal is not to improve this specific system, but instead understand high-level estimates and trade-offs for our GC unit design.



(a) Mark Phase **(b)** Sweep Phase

Figure 9.6: *GC Performance.* On average, the GC Unit outperforms the CPU by a factor of 4.2× for mark and 1.9× for sweep.

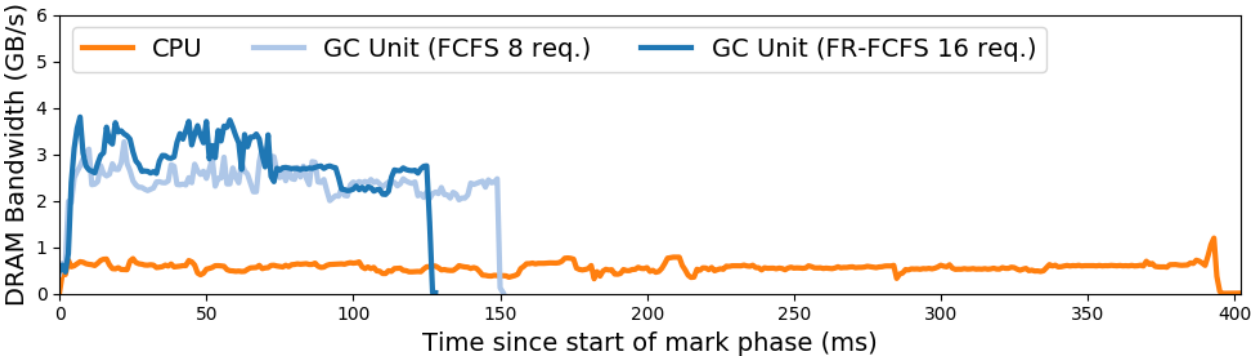
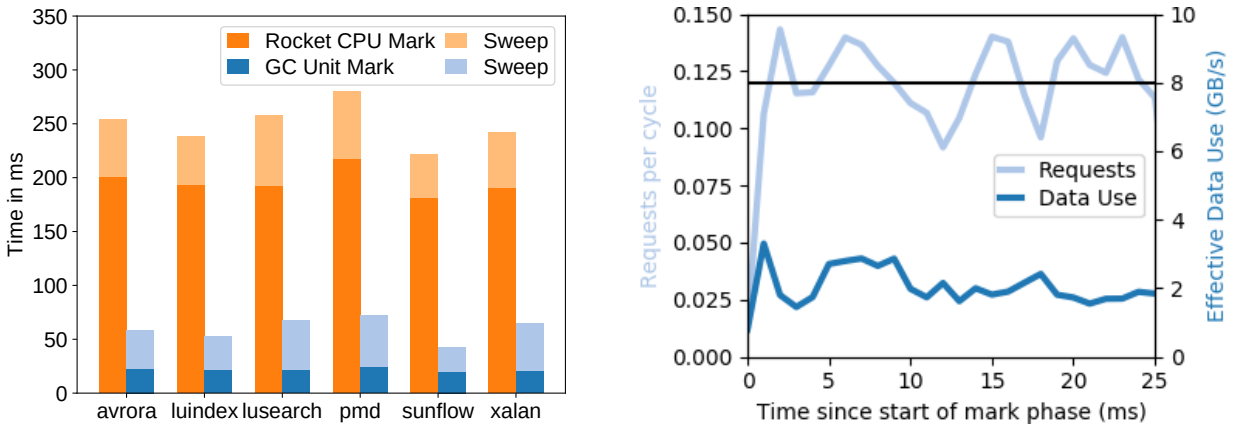


Figure 9.7: *Memory Bandwidth.* Measured for the last GC pause of the *avrora* benchmark, based on 64B cache line accesses.

9.7.1 Garbage Collection Performance

As in Chapter 6, we evaluate performance using the subset of DaCapo benchmarks [37] that runs on our version of JikesRVM. We use the small benchmark size on a 200 MB maximum heap and average across all GC pauses that occur during the benchmark execution.

Our JikesRVM port does not include the optimizing JIT compiler. As Jikes JIT-compiles itself, this would have resulted in a slow baseline of the CPU-version of the GC. We therefore rewrote Jikes's GC in C, compiling it with `-O3` and linking it into the JVM using the same `libhwgc.so` library that we use to communicate with our hardware unit.



(a) Mark/Sweep Performance

(b) Memory Bandwidth

Figure 9.8: *GC Performance with 1 cycle DRAM and 8 GB/s bandwidth.* As some request sizes are smaller than a full cache line, the amount of usable data is smaller than the theoretical peak bandwidth.

Overall Performance

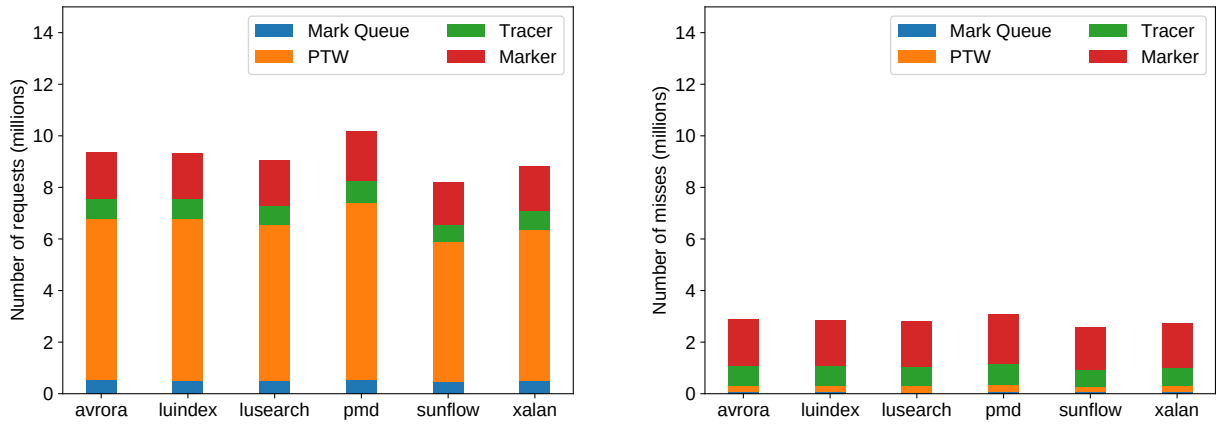
Our baseline GC unit design contains 2 sweepers, a 1,024 entry mark-queue, 16 request slots for the marker, 32-entry TLBs and a 128-entry shared L2 TLB. This configuration outperforms Rocket by 4.2× on mark and 1.9× on sweep (Figure 9.6). Figure 9.7 shows the source of this gain: our unit is more effective at exploiting memory bandwidth, particularly during the mark phase. This was confirmed by experimenting with different memory scheduling strategies: While Rocket was insensitive to the configuration, we found that our performance was significantly improved changing from FIFO MAS to FR-FCFS and increasing the maximum number of outstanding reads from 8 to 16.

Potential Performance

While the previous experiment showed a specific design point with a realistic memory model, we want to fundamentally understand how much memory bandwidth our unit can exploit if it was given a faster memory system. We therefore replaced our model with a latency-bandwidth pipe of latency 1 cycle and bandwidth 8 GB/s.

In this regime, we outperform the CPU by an average of 9.0× on the mark phase. We believe that this is similar to the speed-ups we could see in a high-end SoC, based on preliminary evaluations. Note that the limited speedup for the sweep phase is based on using only two sweepers, and can be increased (Section 9.7.2).

Instrumenting our unit, we found that our TileLink port is busy 88% of all mark cycles. Figure 9.8b shows that this translates to a request being sent into the memory system every 8.66 cycles. With 64B cache lines, one request every 8 cycles would be the full bandwidth of



(a) L1 Cache Requests

(b) Memory Requests

Figure 9.9: *Mark Unit Memory Requests.* In order to reduce contention, we partition the L1 cache into smaller caches.

the 8GB/s system, but as our requests are less than 64B in size, we sometimes exceed this limit. Using small requests also means that, depending on the memory system, we may not be able to make full use of all 8 GB/s (we consume a maximum 3.3GB/s of data).

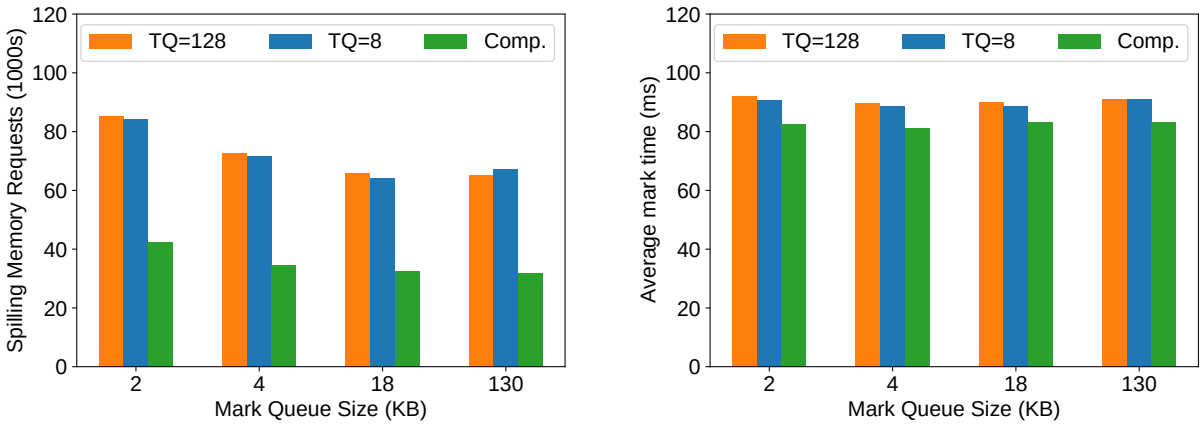
Performance Limits and Impact of TLBs

To understand what prevents our baseline from reaching this 9.0× speedup, we instrumented the unit to explore sources of stalls. We also compared to preliminary simulations on DRAMSim [224] with 8 banks and no virtual memory, which showed 8.5× speedup over Rocket. While not directly comparable, one of the major differences between the two experiments was the presence of virtual memory.

One bottleneck are currently TLB accesses in Marker and Tracer. Our current TLB is blocking (i.e., cannot respond to TLB requests while waiting for a page-table walk to complete), and TLB misses can therefore serialize execution. Future work should therefore introduce a non-blocking TLB that can perform multiple page-table walks concurrently while still serving requests that hit in the TLB. As the unit is pipelined, there is also an opportunity to use bigger multi-cycle TLBs, which might reduce TLB pressure and improve area, as they can use sequential SRAM memories.

9.7.2 Impact of Design Parameters

Cache Partitioning As described in Section 9.4, we started with a design that had a small, shared cache. We found that this performed barely better than the CPU. Figure 9.9a shows why: 2/3 of requests to the cache are from the page-table walker, as the mark phase has little locality and therefore introduces a large number of TLB misses.



(a) Spilling Memory Requests **(b)** Mark Performance

Figure 9.10: *Mark Queue Size Trade-Offs.* Sizes include *inQ*/*outQ* and we show numbers for two different tracer queue (TQ) sizes, as well as with compressed references.

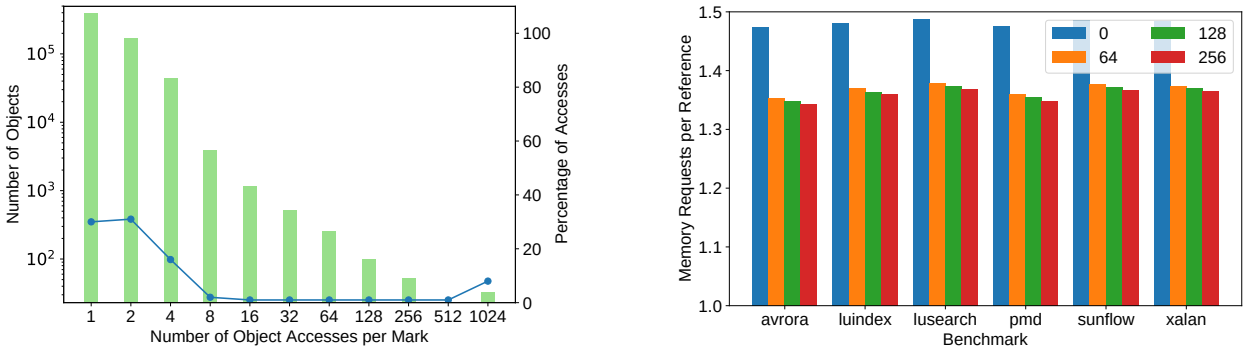
This creates a large amount of contention on the cache's crossbar, effectively drowning out requests by other units. This led us to apply cache partitioning: The PTW benefits from a small 8 KB cache to hold the top levels of the page table, while the mark queue and sweeper access memory sequentially and therefore only need 2 cache lines. Meanwhile, the marker and tracer can connect to the interconnect directly. Another advantage of this setup is that we can remove features from caches that are not needed. For example, the mark queue only operates on physical memory and therefore does not need a TLB.

The result is shown in Figure 9.9b: In terms of memory requests that are sent into the actual memory system, marker and tracer now dominate. This is the intention, as these are the units that perform the main work.

Impact of Mark Queue Size The mark queue is the largest data structure of our unit and we assumed that its size has a major impact on performance. As expected, Figure 9.10 shows that the size has an impact on the amount of spilled data. However, spilling accounts for only $\approx 2\%$ of memory requests.

We were surprised to find that the mark queue's impact on overall performance is small. The reason is that most of the parallelism in the heap traversal exists at the beginning: The queue fills up, almost all of this data is spilled into memory and then enqueuing and dequeueing happen at a similar rate, which means that the queue stays mostly full.

We can therefore choose a very small queue size (e.g., 2 KB) without sacrificing performance. An interesting trade-off is that we can throttle the tracer to match the dequeueing rate of the mark queue. As every reference in the tracer queue expands to multiple references in the mark queue, this may help manage the amount of spilling.



(a) Access Frequencies **(b)** Effect of Caching

Figure 9.11: *Impact of the Mark Bit Cache.* 56 objects account for 10% of accesses (8th GC of *luindex*), and we can filter these duplicate requests with a cache.

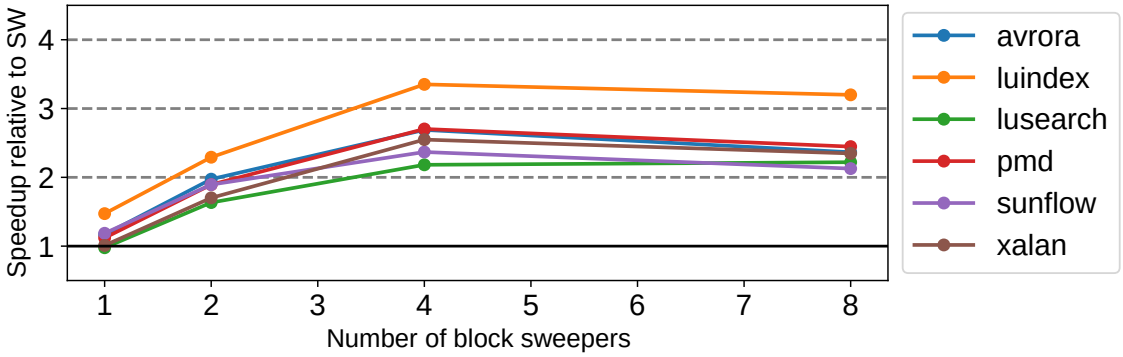
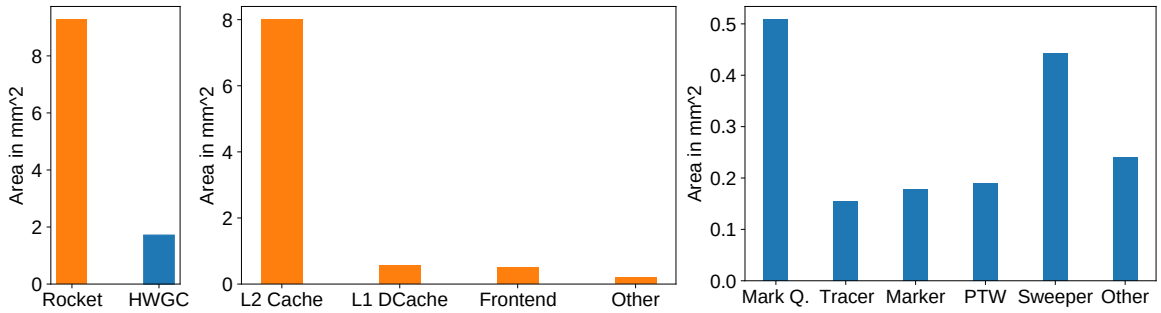


Figure 9.12: *Scaling the number of block sweepers.* Performance is reported as speed-up relative to the software implementation.

Mark Bit Caching A small number of objects account for 10% of all memory accesses (Figure 9.11). Storing the most recently seen references therefore helps reduce the number of marks requests. The largest gain per area can be achieved with a small cache (< 64 elements). At the same time, we found this to not have a substantial impact on the mark performance. We believe that this may change closer to peak bandwidth.

Mark Queue Compression Figure 9.10 shows that our compression scheme from Section 9.4 reduces spilling by a factor of 2. Note that this scheme compresses to 32 bit – real implementations would likely need to preserve at least 48 bit.

Sweeper Parallelism Figure 9.12 shows how additional block sweepers improve sweep performance. We found that we scale linearly to 2 sweepers but that beyond this point, speed-ups start to reduce. At 8 sweepers, the contention on the memory system starts to outweigh the benefits from parallelism. 4 sweepers outperform the CPU by 2–3×.



(a) Total **(b)** Rocket CPU **(c)** GC Unit

Figure 9.13: *Area Synthesis Results.* Estimated using Synopsys DC with the SAED EDK 32/28 standard cell library. Note that Rocket is a small CPU.

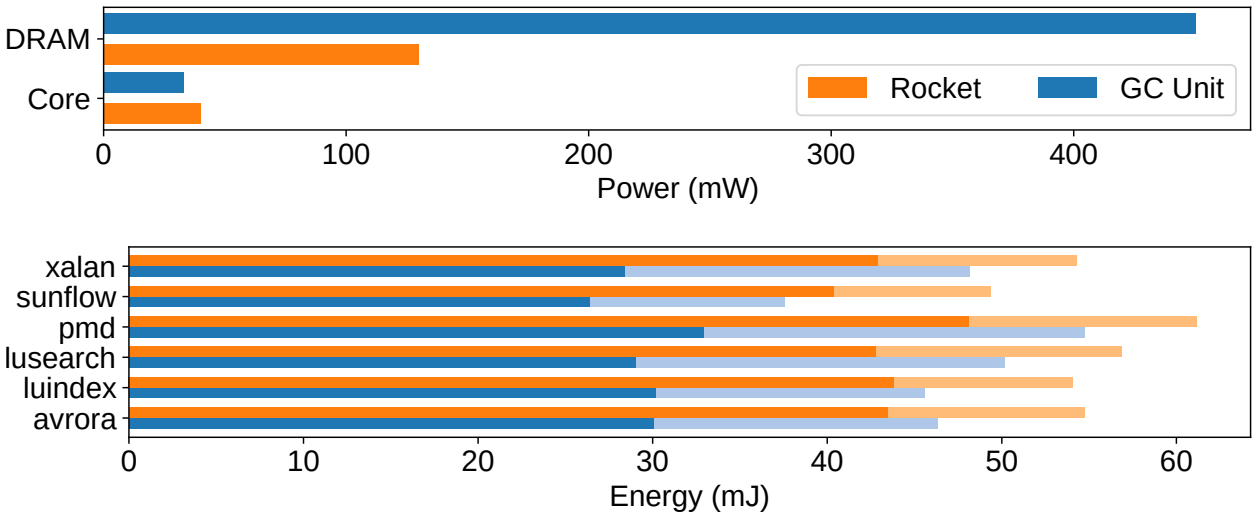


Figure 9.14: *Power and Energy Synthesis Results.* Due to higher bandwidth utilization, DRAM power is much higher, but the overall energy is still lower.

9.7.3 Area & Power Synthesis Results

We ran our design through Synopsys Design Compiler in topographical mode with the SAED EDK 32/28 standard cell library [208]. This provides us with ballpark estimates of area and power numbers. Figure 9.13 shows that our GC unit is 18.5% the size of the CPU, most of which is taken up by the mark queue. This is comparable to the area of 64KB of SRAM. Note that we are comparing to a small CPU – the trade-off would be much more pronounced in a server or mobile SoC, where a block of this size is negligible.

To estimate energy, we collected DRAM-level counters for the GC pauses in Figure 9.7 and ran them through MICRONs DDR3 Power Calculator spread sheet [158]. Power numbers for the GC unit and processor were taken from Design Compiler. Using these power numbers and execution times, we calculate the total energy for mark and sweep (Figure 9.14). Without

activity counters, these results are not exact, but we conclude that the overall energy for the GC unit will likely improve over the CPU (by 14.5% in our results).

9.8 Summary

In this chapter, we presented a prototype of our hardware accelerator for GC that can be integrated into a server or mobile SoC and performs GC for the application on the CPU. The unit can be implemented at a very low hardware cost (equivalent to 64KB of SRAM), generalizes to stop-the-world and concurrent collectors and does not require modifications to the SoC beyond those required by any DMA-capable device.

By implementing a prototype of this design in the context of a real SoC, we demonstrate that the integration effort is manageable, and that the collector design takes up 18.5% the area of a small CPU core, while performing GC at 3.3× the speed. At this low cost, we believe that a case can be made to integrate such accelerators into production SoCs.

Chapter 10

Discussion & Future Work

In this chapter, we discuss implications of the work presented in this thesis, and future research directions that result from it. We then summarize why we believe that future data center research should not consider the managed runtime system in isolation, but instead co-design it with the hardware and the software systems layers.

10.1 Hardware Support for Garbage Collection

Chapter 9 presented a prototype of a basic GC accelerator design. We showed that such a unit is small enough to be integrated into future SoCs at an area cost comparable to 64 KB of SRAM. As this size is negligible within the context of a high-performance SoC, we believe that there is little reason not to add such an accelerator.

The performance benefits may be significant. Moving garbage collection to the accelerator frees up the CPU cycles that are currently spent on GC, accounting for up to 38% of cycles (10% on average [43]). In the context of a data center, these savings are substantial, especially if they are available across a wide range of workloads. Due to its higher GC performance, the accelerator may also improve memory utilization if used in a concurrent collector.

At the same time, we are seeing a trend towards systems software being written in languages such as Go, which have concurrent GC that runs continuously. If the operating system is implemented in such a garbage-collected language, it is important to ensure that the kernel never blocks due to memory pressure, as this would stall other threads and may require the kernel to deschedule applications to run GC instead. This problem could be solved by our hardware accelerator, since it would never require CPU cores to run GC.

In addition to the performance benefits, the accelerator not only decreases on-chip area and power, but also overall energy for the GC operation – this is in contrast to accelerator designs which increase performance but not the energy of the overall operation.

Finally, our design is simple and consists of only 3,122 lines of Chisel code (and less than 50 lines to integrate it into the RocketChip SoC). This is important, since it indicates that verification effort is limited and that the accelerator is unlikely to add significant design effort.

We also confirmed this through our own development, where we found that the design was very amenable to testing using snapshots from previous GC runs.

In order to adopt our accelerator design in a real system, several other problems need to be solved as well. On one hand, the unit needs to be extended to support concurrent GC, which is becoming increasingly important in data centers. The techniques for one possible integration were presented in Section 7.6.1. On the other hand, there are several additional extensions that future work could explore:

- **Supporting Different Object Layouts**: While the bidirectional layout is key to performance, forcing runtimes to adapt it to support our unit is limiting. A more general accelerator could support arbitrary layouts by replacing the marker with a small RISC-V microcontroller (only implementing the base ISA). We could then load a small program into this core which parses the object layout, schedules the appropriate requests and enqueues outgoing references for the tracer.

  One interesting question is how to handle dynamic languages where the same field may either contain a reference or a non-reference value. This could be handled through tag bits (like in SOAR [217]), and allowing the microcontroller to install a predicate that checks these tag bits before adding an entry to the mark queue. This is similar to how the unit currently filters out null pointers when adding references to the mark queue.

- **Bandwidth Throttling**: Our GC unit aims to maximize bandwidth, potentially interfering with applications on the CPU. This interference could be reduced by communicating with the memory controller to only use residual bandwidth. This could allow the accelerator to reduce interference with the application, as it would only use memory bandwidth when it is not needed by the application.

- **Proportionality and Parallelism**: The accelerator bandwidth could potentially be increased by replicating units. Switching these units on and off would allow a concurrent GC to throttle or boost tracing, depending on memory pressure in the application. This approach can improve energy consumption [114].

- **Supporting Multiple Applications**: Our current design only supports one process at a time, but the same unit could perform GC for multiple processes simultaneously. In this case, the unit would handle references from different processeses at the same time, sharing the same mark queue, marker and tracer (similar to simultaneous multithreading). To enable this strategy, the unit would have to support multiple page tables and tag references by process as they travel through the design (to ensure that it always uses the correct page table to translate references).

- **Page Faults**: The JVM currently has to map the entire address space (in our prototype, we force all pages to be mapped before the JVM begins execution). A more general system may want to handle page faults as well (likely by forwarding them to the CPU).

- **Generational GC**: Most production-grade collectors are generational, as the generational approach helps to significantly reduce GC pressure. Our collector design is compatible with this strategy and could be extended to a generational collector, similar to C4's approach of extending Pauseless GC to a generational design [211].

None of these aspects represents a limitation. Instead, they open up a new design space that could explore further trade-offs along each of these directions. Taking these directions together would lead to a light-weight GC accelerator that performs garbage collection for all applications on the CPU and communicates with the memory controller to make use of any residual memory bandwidth that is not used by the application. This accelerator would support different language runtime systems and replace their concurrent GC schemes.

The overarching vision is a system that makes garbage collection invisible and free from the application's perspective. Instead of incurring traps and slow-downs, all GC functionality is handled by the accelerator, without slowing down the application. Using the barrier scheme from Section 7.7, the only visible effect of GC would be `REFLOAD` instructions that may occasionally take longer to complete. Similar to LLC misses, speculation may help to tolerate these latencies. As such, the application does not need to be aware of garbage collection and while constantly receiving new buffers to allocate from.

10.2 Holistic Runtime Systems

Our work around Holistic Runtime Systems opens up several new research directions as well. While our prototype was applied to coordination of garbage collection pauses, the same Holistic Runtime approach could also be applied to other areas, such as sharing the content of code caches, or establishing communication fast paths (Section 3.6).

We also believe that there is potential to improve how policies are developed. Currently, the application developer or administrator has to manually specify a policy and tune it to the specific application scenario. An alternative approach could instead use reinforcement learning to learn and tune policies automatically. Machine learning techniques have been successfully used in this type of scenarios [236].

As the cloud is further moving towards a Platform-as-a-Service (PaaS) model, we believe that the scope of what a Holistic Runtime System could do will expand. With cloud data centers moving towards a serverless model, rack-scale disaggregation and FPGA accelerators, the language runtime system will need to adapt to support these future scenarios. Specifically, we believe that the language runtime system could improve composition of serverless function, transparently access and manage disaggregated storage (reusing existing managed-runtime machinery), and map high-level managed code to FPGA-based accelerators.

We believe that these trends only exacerbate the need to rethink the way that managed language runtime systems are designed for these future data center scenarios. Our Holistic Runtime approach provides a foundation for this work, and could be expanded into a general runtime system design for future cloud data centers.

10.3 Summary

Through two different projects, we showed that several problems related to managed runtime systems in data centers can be solved by working across multiple layers of the data center stack. We therefore argue that research should not consider the language runtime system in isolation, but instead co-design it with the hardware and software systems layers.

Both projects presented in this thesis only represent a first step in this direction and there are several future research directions that may result from this initial work. We also believe that our cross-layer approach generalizes and has the potential to be applied to other areas as well, from using the managed runtime system to transparently manage disaggregated memory in rack-scale machines, to programming FPGAs using high-level programming models embedded into managed languages.

Chapter 11

Conclusion

In this thesis, we showed that it is possible to improve the performance, efficiency and responsiveness of managed data center applications by co-designing the managed runtime system with both the hardware and the software systems layer.

On the software side, we demonstrated that a Holistic Runtime System can be used to improve distributed managed-language workloads by treating the managed runtime systems underpinning the application as a distributed system themselves. Using this approach, we built a prototype Holistic Runtime system that can improve tail-latencies of latency-sensitive workloads by 2-4×, while speeding up a large-scale data-parallel computation by 21% without modifications to the application.

On the hardware side, we showed that we can design a hardware accelerator for garbage collection that performs GC at 3.3× the performance of a CPU, at only 18.5% the area. This design is small and non-invasive enough to be integrated into data center SoCs. We also showed the potential of this approach for eliminating GC-related pauses, by presenting a theoretical design for integrating it into a concurrent collector.

Taken together, these two results demonstrate that the data center operator's full control of the entire stack provides a potential to significantly improve managed workloads running in this scenario. We therefore argue that future cloud data centers should co-design the language runtime system with the underlying hardware and system software.

Bibliography

[1] *2017 BDO Technology Outlook Survey*. Tech. rep. URL: https://www.bdo.com/getattachment/022227f4-aa2e-4a8b-9739-b0ad6b855415/attachment.aspx?2017-Technology-Outlook-Report_2-17.pdf.

[2] Martín Abadi et al. "TensorFlow: A System for Large-Scale Machine Learning". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 265–283. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

[3] Abhinav and Rupesh Nasre. "FastCollect: Offloading Generational Garbage Collection to Integrated GPUs". In: *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. CASES '16. New York, NY, USA: ACM, 2016, 21:1–21:10. ISBN: 978-1-4503-4482-1. DOI: 10.1145/2968455.2968520.

[4] Bowen Alpern et al. "The Jikes Research Virtual Machine Project: Building an Open-Source Research Community". In: *IBM Systems Journal* 44.2 (2005), pp. 399–417. ISSN: 0018-8670. DOI: 10.1147/sj.442.0399.

[5] *Amazon Aurora - Amazon Web Services*. URL: http://aws.amazon.com/rds/aurora/ (visited on 09/25/2017).

[6] *Amazon EC2 F1 Instances*. URL: http://aws.amazon.com/ec2/instance-types/f1/ (visited on 01/25/2017).

[7] *Amazon Polly – Lifelike Text-to-Speech*. URL: http://aws.amazon.com/polly/ (visited on 09/25/2017).

[8] AMD. *AMD Accelerated Parallel Processing (APP) SDK OpenCL Programming Guide*. URL: http://developer.amd.com/sdks/AMDAPPSDK/assets/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide.pdf.

[9] AMD. *AMD Embedded G-Series Platform: The World's First Combination of Low-Power CPU and Advanced GPU Integrated into a Single Embedded Device*. URL: http://www.amd.com/us/Documents/49282_G-Series_platform_brief.pdf.

[10] Johan Andersson et al. "Kaffemik - a Distributed JVM Featuring a Single Address Space Architecture". In: *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*. JVM'01. Berkeley, CA, USA: USENIX Association, 2001, pp. 9–9. URL: http://dl.acm.org/citation.cfm?id=1267847.1267856.

[11] *Apache Arrow*. URL: https://arrow.apache.org/ (visited on 09/29/2017).

[12] *Apache Harmony*. URL: http://harmony.apache.org/ (visited on 12/11/2017).

[13] Andrew W. Appel and Aage Bendiksen. "Vectorized Garbage Collection". In: *The Journal of Supercomputing* 3.3 (1989), pp. 151–160.

[14] *Application Domains*. URL: https://msdn.microsoft.com/en-us/library/2bh4z9hs(v=vs.110).aspx (visited on 04/27/2017).

[15] Y. Aridor, M. Factor, and A. Teperman. "cJVM: A Single System Image of a JVM on a Cluster". In: *Proceedings of the 1999 International Conference on Parallel Processing*. ICPP '99. 1999, pp. 4–11. DOI: 10.1109/ICPP.1999.797382.

[16] Nikita Artyushov. *Revealing the Length of Garbage Collection Pauses*. URL: https://plumbr.eu/blog/garbage-collection/revealing-the-length-of-garbage-collection-pauses (visited on 09/29/2017).

[17] Krste Asanovic and David Patterson. "Firebox: A Hardware Building Block for 2020 Warehouse-Scale Computers". In: *USENIX FAST*. Vol. 13. 2014.

[18] Krste Asanovic et al. *The Rocket Chip Generator*. Tech. rep. UCB/EECS-2016-17. EECS Department, University of California, Berkeley, Apr. 2016. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html.

[19] *AWS Lambda - Serverless Compute*. URL: //aws.amazon.com/lambda/ (visited on 01/25/2017).

[20] *Azure Functions—Serverless Architecture*. URL: https://azure.microsoft.com/en-us/services/functions/ (visited on 09/25/2017).

[21] M. Bächle and P. Kirchberg. "Ruby on Rails". In: *IEEE Software* 24.6 (Nov. 2007), pp. 105–108. ISSN: 0740-7459. DOI: 10.1109/MS.2007.176.

[22] Jonathan Bachrach et al. "Chisel: Constructing Hardware in a Scala Embedded Language". In: *Proceedings of the 49th Annual Design Automation Conference*. DAC '12. New York, NY, USA: ACM, 2012, pp. 1216–1225. ISBN: 978-1-4503-1199-1. DOI: 10.1145/2228360.2228584. (Visited on 04/21/2014).

[23] David F. Bacon, Perry Cheng, and V. T. Rajan. "The Metronome: A Simpler Approach to Garbage Collection in Real-Time Systems". In: *In Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), OTM Workshops*. 2003, pp. 466–478.

[24] David F. Bacon, Perry Cheng, and Sunil Shukla. "And Then There Were None: A Stall-Free Real-Time Garbage Collector for Reconfigurable Hardware". In: *Commun. ACM* 56.12 (Dec. 2013), pp. 101–109. ISSN: 0001-0782. DOI: 10.1145/2534706.2534726. (Visited on 11/26/2013).

[25] Jonathan Balkind et al. "OpenPiton: An Open Source Manycore Research Framework". In: *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '16. New York, NY, USA: ACM, 2016, pp. 217–232. ISBN: 978-1-4503-4091-5. DOI: 10.1145/2872362.2872414.

[26] Katherine Barabash and Erez Petrank. "Tracing Garbage Collection on Highly Parallel Platforms". In: *SIGPLAN Not.* 45.8 (June 2010), pp. 1–10.

[27] Edd Barrett et al. "Virtual Machine Warmup Blows Hot and Cold". In: *arXiv:1602.00602 [cs]* (Feb. 2016). arXiv: 1602.00602 [cs]. URL: http://arxiv.org/abs/1602.00602.

[28] Luiz Barroso et al. "Attack of the Killer Microseconds". In: *Commun. ACM* 60.4 (Mar. 2017), pp. 48–54. ISSN: 0001-0782. DOI: 10.1145/3015146.

[29] Andrew Baumann 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 '09. New York, NY, USA: ACM, 2009, pp. 29–44. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629579.

[30] S. Beamer, K. Asanovic, and D. Patterson. "Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server". In: *2015 IEEE International Symposium on Workload Characterization*. Oct. 2015, pp. 56–65. DOI: 10.1109/IISWC.2015.12.

[31] *Benchmarking G1 and Other Java 7 Garbage Collectors - Tuning Guidelines for Java Garbage Collection, Part 2*. URL: http://blog.mgm-tp.com/2013/12/benchmarking-g1-and-other-java-7-garbage-collectors/,%20http://blog.mgm-tp.com/2013/12/benchmarking-g1-and-other-java-7-garbage-collectors/ (visited on 10/12/2017).

[32] *BigQuery - Analytics Data Warehouse*. URL: https://cloud.google.com/bigquery/ (visited on 09/25/2017).

[33] BillWagner. *Value Types (C# Reference)*. URL: https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/value-types (visited on 09/28/2017).

[34] *Bing Speech API—Speech Recognition*. URL: https://azure.microsoft.com/en-us/services/cognitive-services/speech/ (visited on 09/25/2017).

[35] Nathan Binkert et al. "The Gem5 Simulator". In: *SIGARCH Comput. Archit. News* 39.2 (Aug. 2011), pp. 1–7. ISSN: 0163-5964. DOI: 10.1145/2024716.2024718. (Visited on 11/26/2017).

[36] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. "Oil and Water? High Performance Garbage Collection in Java with MMTk". In: *Proceedings of the 26th International Conference on Software Engineering*. ICSE '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 137–146. ISBN: 0-7695-2163-0. URL: http://dl.acm.org/citation.cfm?id=998675.999420.

[37] Stephen M. Blackburn et al. "The DaCapo Benchmarks: Java Benchmarking Development and Analysis". In: *OOPSLA '06: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programing, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190. DOI: http://doi.acm.org/10.1145/1167473.1167488.

[38] Stephen M. Blackburn et al. "Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century". In: *Commun. ACM* 51.8 (Aug. 2008), pp. 83–89. ISSN: 0001-0782. DOI: 10.1145/1378704.1378723.

[39] Jonas Bonér and Eugene Kuleshov. "Clustering the Java Virtual Machine Using Aspect-Oriented Programming". In: *AOSD'07: Proceedings of the 6th International Conference on Aspect-Oriented Software Development*. 2007.

[40] Dan Bouvier and Ben Sander. "Applying AMD's Kaveri APU for Heterogeneous Computing". In: *Hot Chips 26 Symposium (HCS)*. 2014, pp. 1–42.

[41] Brendan Burns et al. "Borg, Omega, and Kubernetes". In: *Commun. ACM* 59.5 (Apr. 2016), pp. 50–57. ISSN: 0001-0782. DOI: 10.1145/2890784. (Visited on 01/25/2017).

[42] Callum Cameron, Jeremy Singer, and David Vengerov. "The Judgment of Forseti: Economic Utility for Dynamic Heap Sizing of Multiple Runtimes". In: *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*. ISMM 2015. New York, NY, USA: ACM, 2015, pp. 143–156. ISBN: 978-1-4503-3589-8. DOI: 10.1145/2754169.2754180. (Visited on 08/07/2015).

[43] Ting Cao et al. "The Yin and Yang of Power and Performance for Asymmetric Hardware and Managed Software". In: *Proceedings of the 39th Annual International Symposium on Computer Architecture*. ISCA '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 225–236. ISBN: 978-1-4503-1642-2. URL: http://dl.acm.org/citation.cfm?id=2337159.2337185 (visited on 03/16/2014).

[44] Trevor E. Carlson, Wim Heirman, and Lieven Eeckhout. "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulation". In: *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '11. Seattle, Washington: ACM, 2011, 52:1–52:12. ISBN: 978-1-4503-0771-0. DOI: 10.1145/2063384.2063454.

[45] Bryan Catanzaro et al. "SEJITS: Getting Productivity and Performance with Selective Embedded JIT Specialization". In: *Programming Models for Emerging Architectures* (2009).

[46] Adrian M. Caulfield et al. "A Cloud-Scale Acceleration Architecture". In: *49th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2016, Taipei, Taiwan, October 15-19, 2016*. 2016, pp. 1–13. DOI: 10.1109/MICRO.2016.7783710.

[47] Yunji Chen et al. "DianNao Family: Energy-Efficient Hardware Accelerators for Machine Learning". In: *Commun. ACM* 59.11 (Oct. 2016), pp. 105–112. ISSN: 0001-0782. DOI: 10.1145/2996864. (Visited on 01/25/2017).

[48] Chen-yong Cher and Michael Gschwind. "Cell GC: Using the Cell Synergistic Processor as a Garbage Collection Coprocessor". In: *VEE '08: Proceedings of the 4th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. ACM, 2008, pp. 141–150.

[49] David Cheriton. "The V Distributed System". In: *Commun. ACM* 31.3 (Mar. 1988), pp. 314–333. ISSN: 0001-0782. DOI: 10.1145/42392.42400. (Visited on 03/22/2015).

[50] Eric S. Chung, John D. Davis, and Jaewon Lee. "LINQits: Big Data on Little Clients". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. New York, NY, USA: ACM, 2013, pp. 261–272. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485945.

[51] Eric Chung and Jeremy Fowers. "Accelerating Persistent Neural Networks at Datacenter Scale". In: *Hot Chips 29 Symposium (HCS)*. 2017.

[52] *Cisco Global Cloud Index: Forecast and Methodology, 2015–2020*. Tech. rep. 2016.

[53] Cliff Click, Gil Tene, and Michael Wolf. "The Pauseless GC Algorithm". In: *Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*. VEE '05. New York, NY, USA: ACM, 2005, pp. 46–56. ISBN: 1-59593-047-7. DOI: 10.1145/1064979.1064988.

[54] Daniel Clifford et al. "Memento Mori: Dynamic Allocation-Site-Based Optimizations". In: *Proceedings of the 2015 International Symposium on Memory Management*. ISMM '15. New York, NY, USA: ACM, 2015, pp. 105–117. ISBN: 978-1-4503-3589-8. DOI: 10.1145/2754169.2754181.

[55] *Cloud Dataflow - Batch & Stream Data Processing*. URL: https://cloud.google.com/dataflow/ (visited on 01/25/2017).

[56] Juan A. Colmenares et al. "Tessellation: Refactoring the OS Around Explicit Resource Containers with Continuous Adaptation". In: *Proceedings of the 50th Annual Design Automation Conference*. DAC '13. New York, NY, USA: ACM, 2013, 76:1–76:10. ISBN: 978-1-4503-2071-9. DOI: 10.1145/2463209.2488827.

[57] Louis Columbus. *Roundup Of Cloud Computing Forecasts, 2017*. 2017. URL: https://www.forbes.com/sites/louiscolumbus/2017/04/29/roundup-of-cloud-computing-forecasts-2017/ (visited on 09/01/2017).

[58] Henry Cook, Wesley Terpstra, and Yunsup Lee. "Diplomatic Design Patterns: A TileLink Case Study". In: *First Workshop on Computer Architecture Research with RISC-V*. CARRV 2017. 2017.

[59] Brian F. Cooper et al. "Benchmarking Cloud Serving Systems with YCSB". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. New York, NY, USA: ACM, 2010, pp. 143–154. ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152. (Visited on 12/30/2014).

[60] Jesus Corbal et al. "Knights Mill: Intel Xeon Phi Processor for Machine Learning". In: *Hot Chips 29 Symposium (HCS)*. 2017.

[61] *Credit Suisse Case Study*. URL: http://www.azulsystems.com/customers/creditsuisse (visited on 03/22/2015).

[62] Jeffrey Dean and Luiz André Barroso. "The Tail at Scale". In: *Commun. ACM* 56.2 (Feb. 2013), pp. 74–80. ISSN: 0001-0782. DOI: 10.1145/2408776.2408794. (Visited on 11/24/2013).

[63] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-Value Store". In: *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*. SOSP '07. New York, NY, USA: ACM, 2007, pp. 205–220. ISBN: 978-1-59593-591-5. DOI: 10.1145/1294261.1294281.

[64] Christina Delimitrou and Christos Kozyrakis. "Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 77–88. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451125.

[65] Christina Delimitrou and Christos Kozyrakis. "Quasar: Resource-Efficient and QoS-Aware Cluster Management". In: *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '14. New York, NY, USA: ACM, 2014, pp. 127–144. ISBN: 978-1-4503-2305-5. DOI: 10.1145/2541940.2541941. (Visited on 06/30/2014).

[66] David Detlefs et al. "Garbage-First Garbage Collection". In: *Proceedings of the 4th International Symposium on Memory Management*. ISMM '04. New York, NY, USA: ACM, 2004, pp. 37–48. ISBN: 978-1-58113-945-7. DOI: 10.1145/1029873.1029879.

[67] Irwin D'Souza. *How Concurrent Scavenge Using the Guarded Storage Facility Works*. Sept. 2017. URL: https://developer.ibm.com/javasdk/2017/09/25/concurrent-scavenge-using-guarded-storage-facility-works/ (visited on 10/01/2017).

[68] Thomas E. Anderson, David Culler, and David A. Patterson. "The Berkeley Networks of Workstations (NOW) Project". In: *Proceedings of the 40th IEEE Computer Society International Conference*. COMPCON '95 (1995), pp. 322–326.

[69] M. Elteir, Heshan Lin, and Wu-Chun Feng. "Performance Characterization and Optimization of Atomic Operations on AMD GPUs". In: *2011 IEEE International Conference on Cluster Computing (CLUSTER)*. Sept. 2011, pp. 234–243.

[70] *Face API - Facial Recognition*. URL: `https://azure.microsoft.com/en-us/services/cognitive-services/face/` (visited on 09/25/2017).

[71] Hua Fan et al. "Understanding the Causes of Consistency Anomalies in Apache Cassandra". In: *Proceedings of the VLDB Endowment* 8.7 (2015).

[72] Paolo Faraboschi et al. "Beyond Processor-Centric Operating Systems". In: *15th Workshop on Hot Topics in Operating Systems (HotOS XV)*. Kartause Ittingen, Switzerland: USENIX Association, 2015. URL: `http://blogs.usenix.org/conference/hotos15/workshop-program/presentation/faraboschi`.

[73] Christine H. Flood et al. "Shenandoah: An Open-Source Concurrent Compacting Garbage Collector for OpenJDK". In: *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. PPPJ '16. New York, NY, USA: ACM, 2016, 13:1–13:9. ISBN: 978-1-4503-4135-6. DOI: `10.1145/2972206.2972210`.

[74] *Fragger Tool*. URL: `https://www.azul.com/resources/fragger-tool/` (visited on 09/29/2017).

[75] *G1: One Garbage Collector To Rule Them All*. URL: `http://www.infoq.com/articles/G1-One-Garbage-Collector-To-Rule-Them-All` (visited on 01/04/2015).

[76] Etienne M. Gagnon and Laurie J. Hendren. "SableVM: A Research Framework for the Efficient Execution of Java Bytecode". In: *In Proceedings of the Java Virtual Machine Research and Technology Symposium*. 2000, pp. 27–40.

[77] Peter X. Gao et al. "Network Requirements for Resource Disaggregation". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 249–264. ISBN: 978-1-931971-33-1. URL: `https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao`.

[78] *Garbage Collection Notifications*. URL: `https://msdn.microsoft.com/en-us/library/cc713687(v=vs.110).aspx`.

[79] Robin J. Garner, Stephen M. Blackburn, and Daniel Frampton. "A Comprehensive Evaluation of Object Scanning Techniques". In: *Proceedings of the International Symposium on Memory Management*. ISMM '11. San Jose, California, USA, 2011, pp. 33–42.

[80] Ionel Gog et al. "Broom: Sweeping out Garbage Collection from Big Data Systems". In: *Proceedings of the 15th USENIX/ACM Workshop on Hot Topics in Operating Systems (HotOS 2015)*. 2015.

[81] Ionel Gog et al. "Firmament: Fast, Centralized Cluster Scheduling at Scale". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'16. Berkeley, CA, USA: USENIX Association, 2016, pp. 99–115. ISBN: 978-1-931971-33-1. URL: http://dl.acm.org/citation.cfm?id=3026877.3026886.

[82] Ionel Gog et al. "Musketeer: All for One, One for All in Data Processing Systems". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. New York, NY, USA: ACM, 2015, 2:1–2:16. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741968. (Visited on 01/25/2017).

[83] Joseph E. Gonzalez et al. "GraphX: Graph Processing in a Distributed Dataflow Framework". In: *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 599–613. ISBN: 978-1-931971-16-4. URL: http://dl.acm.org/citation.cfm?id=2685048.2685096 (visited on 03/22/2015).

[84] *Google App Engine: Platform as a Service*. (Visited on 02/11/2014).

[85] *Google Data Center FAQ*. Mar. 2017. URL: http://www.datacenterknowledge.com/archives/2017/03/16/google-data-center-faq (visited on 09/28/2017).

[86] *Google Is Said to Endorse ARM Server Chips, but Don't Get Excited yet — PCWorld*. URL: https://www.pcworld.com/article/3029740/components-processors/google-is-said-to-endorse-arm-server-chips-but-dont-get-excited-yet.html (visited on 09/29/2017).

[87] *Google Uncloaks Once-Secret Server*. URL: https://www.cnet.com/news/google-uncloaks-once-secret-server-10209580/ (visited on 09/28/2017).

[88] Wolfgang Gottesheim. *The DevOps Way to Solving JVM Memory Issues*. Aug. 2013. URL: https://www.dynatrace.com/blog/the-devops-way-to-solving-jvm-memory-issues/ (visited on 10/11/2017).

[89] John John Gough and K. John Gough. *Compiling for the .Net Common Language Runtime*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. ISBN: 978-0-13-062296-9.

[90] Dayong Gu, Clark Verbrugge, and Etienne M. Gagnon. "Relative Factors in Performance Analysis of Java Virtual Machines". In: *Proceedings of the 2nd International Conference on Virtual Execution Environments*. VEE '06. New York, NY, USA: ACM, 2006, pp. 111–121. ISBN: 978-1-59593-332-4. DOI: 10.1145/1134760.1134776.

[91] James Hamilton. *Cost of Power in Large-Scale Data Centers – Perspectives*. URL: http://perspectives.mvdirona.com/2008/11/cost-of-power-in-large-scale-data-centers/ (visited on 09/02/2017).

[92] Pawan Harish and P J Narayanan. "Accelerating Large Graph Algorithms on the GPU Using CUDA". In: *Technology* 4873 (2007). Ed. by Srinivas Aluru et al., pp. 197–208.

[93] Mark Harris. "Parallel Prefix Sum (Scan) with CUDA". In: *GPU Gems* 3.April (2007). Ed. by HubertEditor Nguyen, pp. 851–876.

[94] Tim Harris, Martin Maas, and Virendra J. Marathe. "Callisto: Co-Scheduling Parallel Runtime Systems". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. New York, NY, USA: ACM, 2014, 24:1–24:14. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592807. (Visited on 08/19/2014).

[95] Timothy L. Harris. "Dynamic Adaptive Pre-Tenuring". In: *Proceedings of the 2Nd International Symposium on Memory Management*. ISMM '00. New York, NY, USA: ACM, 2000, pp. 127–136. ISBN: 978-1-58113-263-2. DOI: 10.1145/362422.362476.

[96] Tim Harris et al. "Dynamic Filtering: Multi-Purpose Architecture Support for Language Runtime Systems". In: *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*. ASPLOS XV. New York, NY, USA: ACM, 2010, pp. 39–52. ISBN: 978-1-60558-839-1. DOI: 10.1145/1736020.1736027. (Visited on 03/17/2014).

[97] Chris Hawblitzel et al. "IronFleet: Proving Practical Distributed Systems Correct". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. New York, NY, USA: ACM, 2015, pp. 1–17. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815428.

[98] *HDFS Issue 7244: "Reduce Namenode Memory Using Flyweight Pattern"*. URL: https://issues.apache.org/jira/browse/HDFS-7244 (visited on 03/22/2015).

[99] Timothy H. Heil and James E. Smith. "Concurrent Garbage Collection Using Hardware-Assisted Profiling". In: *Proceedings of the 2Nd International Symposium on Memory Management*. ISMM '00. New York, NY, USA: ACM, 2000, pp. 80–93. ISBN: 978-1-58113-263-2. DOI: 10.1145/362422.362466. (Visited on 12/15/2017).

[100] *Here's How Much Energy All US Data Centers Consume — Data Center Knowledge*. URL: http://www.datacenterknowledge.com/archives/2016/06/27/heres-how-much-energy-all-us-data-centers-consume/ (visited on 09/02/2017).

[101] Matthew Hertz and Emery D. Berger. "Quantifying the Performance of Garbage Collection vs. Explicit Memory Management". In: *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '05. New York, NY, USA: ACM, 2005, pp. 313–326. ISBN: 1-59593-031-0. DOI: 10.1145/1094811.1094836.

[102] Benjamin Hindman 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'11. Berkeley, CA, USA: USENIX Association, 2011, pp. 22–22. URL: http://dl.acm.org/citation.cfm?id=1972457.1972488 (visited on 09/24/2013).

[103] Pieter Hintjens. *ZeroMQ: The Guide*. Tech. rep. 2010. URL: http://zguide.zeromq.org/page:all.

[104] Urs Hölzle, Craig Chambers, and David Ungar. "Debugging Optimized Code with Dynamic Deoptimization". In: *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*. PLDI '92. New York, NY, USA: ACM, 1992, pp. 32–43. ISBN: 978-0-89791-475-8. DOI: 10.1145/143095.143114.

[105] Sungpack Hong et al. "Accelerating CUDA Graph Algorithms at Maximum Warp". In: *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*. PPoPP '11. New York, NY, USA: ACM, 2011, pp. 267–276. ISBN: 978-1-4503-0119-0. DOI: 10.1145/1941553.1941590.

[106] Jon Howell and Mark Montague. *Hey, You Got Your Language In My Operating System!* Tech. rep. Hanover, NH, USA: Dartmouth College, 1998.

[107] *HP Moonshot System*. URL: www.hp.com/go/moonshot (visited on 03/25/2014).

[108] Shan Shan Huang et al. "Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary". In: *Proceedings of the 22Nd European Conference on Object-Oriented Programming*. ECOOP '08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 76–103. ISBN: 978-3-540-70591-8. DOI: 10.1007/978-3-540-70592-5_5. (Visited on 01/25/2017).

[109] Xianglong Huang et al. "The Garbage Collection Advantage: Improving Program Locality". In: *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '04. New York, NY, USA: ACM, 2004, pp. 69–80. ISBN: 978-1-58113-831-3. DOI: 10.1145/1028976.1028983.

[110] *Huawei Proposed DC 3.0 Architecture of Future Data Center to Meet the Requirement of Real-Time Data Processing in Big Data Era - Huawei Press Center*. URL: http://pr.huawei.com/en/news/hw-423134-3.0.htm%5C#.WIjlerYrLiE (visited on 01/25/2017).

[111] Richard Hudson. *Go GC: Prioritizing Low Latency and Simplicity - The Go Blog*. URL: https://blog.golang.org/go15gc (visited on 09/29/2017).

[112] Galen C. Hunt and James R. Larus. "Singularity: Rethinking the Software Stack". In: *SIGOPS Oper. Syst. Rev.* 41.2 (Apr. 2007), pp. 37–49. ISSN: 0163-5980. DOI: 10.1145/1243418.1243424. (Visited on 09/24/2013).

[113] Patrick Hunt et al. "ZooKeeper: Wait-Free Coordination for Internet-Scale Systems". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*. USENIXATC'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 11–11. URL: http://dl.acm.org/citation.cfm?id=1855840.1855851 (visited on 03/22/2015).

[114] Ahmed Hussein et al. "Don't Race the Memory Bus: Taming the GC Leadfoot". In: *Proceedings of the 2015 International Symposium on Memory Management*. ISMM '15. New York, NY, USA: ACM, 2015, pp. 15–27. ISBN: 978-1-4503-3589-8. DOI: 10.1145/2754169.2754182.

[115] *Implementing ART Just-In-Time (JIT) Compiler — Android Open Source Project*. URL: https://source.android.com/devices/tech/dalvik/jit-compiler.html (visited on 01/25/2017).

[116] *Inside .NET Native (Channel 9)*. URL: http://channel9.msdn.com/Shows/Going+Deep/Inside-NET-Native (visited on 03/22/2015).

[117] *Intel® Rack Scale Design*. URL: http://www.intel.com/content/www/us/en/architecture-and-technology/rack-scale-design-overview.html (visited on 01/25/2017).

[118] Christian Jacobi and Anthony Saporito. "The Next Generation IBM Z Systems Processor". In: *2017 IEEE Hot Chips 29 Symposium (HCS)* (2017).

[119] *JEP 295: Ahead-of-Time Compilation*. URL: http://openjdk.java.net/jeps/295 (visited on 01/25/2017).

[120] Azeem S. Jiva and Gary R. Frost. "GPU Assisted Garbage Collection". U.S. Patent 8,301,672 (US). 1, 42010. URL: https://www.google.com/patents/US8301672.

[121] José A. Joao, Onur Mutlu, and Yale N. Patt. "Flexible Reference-Counting-Based Hardware Acceleration for Garbage Collection". In: *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ISCA '09. New York, NY, USA: ACM, 2009, pp. 418–428. ISBN: 978-1-60558-526-0. DOI: 10.1145/1555754.1555806.

[122] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Sept. 1996. ISBN: 0-471-94148-4. (Visited on 05/09/2014).

[123] Richard Jones et al. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Japanese translation of *JHM2011. Shoeisha, Mar. 2016. ISBN: 978-47981342098.

[124] Mick Jordan et al. *Scaling J2EE Application Servers with the Multi-Tasking Virtual Machine*. Tech. rep. Mountain View, CA, USA: Sun Microsystems, Inc., 2004.

[125] Norman P. Jouppi et al. "In-Datacenter Performance Analysis of a Tensor Processing Unit". In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ISCA '17. New York, NY, USA: ACM, 2017, pp. 1–12. ISBN: 978-1-4503-4892-8. DOI: 10.1145/3079856.3080246.

[126] *JSR-000121 Application Isolation API Specification*. URL: https://jcp.org/aboutJava/communityprocess/final/jsr121/ (visited on 02/11/2014).

[127] Myoungsoo Jung et al. "HIOS: A Host Interface I/O Scheduler for Solid State Disks". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 289–300. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665715.

[128] Eric Kaczmarek and Liqi Yi. *Taming GC Pauses for Humongous Java Heaps in Spark Graph Computing*. Data & Analytics. 2015. URL: https://www.slideshare.net/SparkSummit/kaczmarek-yi.

[129] Svilen Kanev et al. "Profiling a Warehouse-Scale Computer". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. New York, NY, USA: ACM, 2015, pp. 158–169. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750392.

[130] Khronos Group. *OpenCL 1.2 Specification*. Tech. rep. URL: http://www.khronos.org/registry/cl/specs/opencl-1.2.pdf.

[131] Donggyu Kim et al. "Evaluation of RISC-V RTL Designs with FPGA Simulation". In: *First Workshop on Computer Architecture Research with RISC-V*. CARRV 2017. 2017.

[132] Donggyu Kim et al. "Strober: Fast and Accurate Sample-Based Energy Simulation for Arbitrary RTL". In: *Proceedings of the 43rd International Symposium on Computer Architecture*. ISCA '16. Piscataway, NJ, USA: IEEE Press, 2016, pp. 128–139. ISBN: 978-1-4673-8947-1. DOI: 10.1109/ISCA.2016.21.

[133] Marcel Kornacker et al. "Impala: A Modern, Open-Source SQL Engine for Hadoop". In: *CIDR 2015, Seventh Biennial Conference on Innovative Data Systems Research*. 2015. URL: http://www.cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf.

[134] Avinash Lakshman and Prashant Malik. "Cassandra: A Decentralized Structured Storage System". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 35–40. ISSN: 0163-5980. DOI: 10.1145/1773912.1773922. (Visited on 01/03/2015).

[135] Frederic Lardinois. *Mesosphere Open Sources Its Data Center OS*. URL: http://social.techcrunch.com/2016/04/19/mesosphere-open-sources-its-data-center-os/ (visited on 09/28/2017).

[136] Michael A. Laurenzano et al. "Protean Code: Achieving Near-Free Online Code Transformations for Warehouse Scale Computers". In: *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO-47. Washington, DC, USA: IEEE Computer Society, 2014, pp. 558–570. ISBN: 978-1-4799-6998-2. DOI: 10.1109/MICRO.2014.21. (Visited on 03/22/2015).

[137] Edward D. Lazowska et al. "The Architecture of the Eden System". In: *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*. SOSP '81. New York, NY, USA: ACM, 1981, pp. 148–159. ISBN: 0-89791-062-1. DOI: 10.1145/800216.806603. (Visited on 03/22/2015).

[138] Jacob Leverich and Christos Kozyrakis. "Reconciling High Server Utilization and Sub-Millisecond Quality-of-Service". In: *Proceedings of the Ninth European Conference on Computer Systems*. EuroSys '14. New York, NY, USA: ACM, 2014, 4:1–4:14. ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592821.

[139] Ari Levy. *Microsoft Azure Growing Faster than AWS, Google Cloud Behind*. 2017-04-27T18:00:51-0400. URL: https://www.cnbc.com/2017/04/27/microsoft-azure-growing-faster-than-aws-google-cloud-behind.html (visited on 09/01/2017).

[140] Jialin Li et al. "Tales of the Tail: Hardware, OS, and Application-Level Sources of Tail Latency". In: *Proceedings of the ACM Symposium on Cloud Computing*. SOCC '14. New York, NY, USA: ACM, 2014, 9:1–9:14. ISBN: 978-1-4503-3252-1. DOI: 10.1145/2670979.2670988.

[141] S. Li et al. "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures". In: *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Dec. 2009, pp. 469–480.

[142] *Libffi*. URL: https://sourceware.org/libffi/ (visited on 09/28/2017).

[143] David Lion et al. "Don't Get Caught in the Cold, Warm-up Your JVM: Understand and Eliminate JVM Warm-up Overhead in Data-Parallel Systems". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 383–400. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/lion.

[144] David Lo et al. "Heracles: Improving Resource Efficiency at Scale". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. New York, NY, USA: ACM, 2015, pp. 450–462. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2749475.

[145] *LogCabin (GitHub)*. URL: http://github.com/logcabin/logcabin.

[146] Lijuan Luo, Martin Wong, and Wen-mei Hwu. "An Effective GPU Implementation of Breadth-First Search". In: *Proceedings of the 47th Design Automation Conference*. DAC '10. Anaheim, California, 2010, pp. 52–55.

[147] Martin Maas and Ross McIlroy. "A JVM for the Barrelfish Operating System". In: Bern, Switzerland, 2012.

[148] Martin Maas et al. "GPUs As an Opportunity for Offloading Garbage Collection". In: *Proceedings of the 2012 International Symposium on Memory Management*. ISMM '12. New York, NY, USA: ACM, 2012, pp. 25–36. ISBN: 978-1-4503-1350-6. DOI: 10.1145/2258996.2259002. (Visited on 05/09/2014).

[149] Martin Maas et al. "The Case for the Holistic Language Runtime System". In: *First International Workshop on Rack-Scale Computing (WRSC '14)*. 2014.

[150] Martin Maas et al. "Trash Day: Coordinating Garbage Collection in Distributed Systems". In: *Proceedings of the 15th USENIX/ACM Workshop on Hot Topics in Operating Systems (HotOS 2015)*. 2015.

[151] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. "Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems". In: *Proceedings of the 25th Symposium on Operating Systems Principles*. SOSP '15. New York, NY, USA: ACM, 2015, pp. 378–393. ISBN: 978-1-4503-3834-9. DOI: 10.1145/2815400.2815415.

[152] Anil Madhavapeddy et al. "Unikernels: Library Operating Systems for the Cloud". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 461–472. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451167. (Visited on 01/25/2017).

[153] P. S. Magnusson et al. "Simics: A Full System Simulation Platform". In: *Computer* 35.2 (Feb. 2002), pp. 50–58. ISSN: 0018-9162. DOI: 10.1109/2.982916.

[154] Ahmed El-Mahdy, Ian Watson, and Greg Wright. "Java Machine and Integrated Circuit Architecture (Jamaica)". In: *Java Microarchitectures*. Springer, 2002, pp. 187–206.

[155] Jason Mars et al. "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-44. New York, NY, USA: ACM, 2011, pp. 248–259. ISBN: 978-1-4503-1053-6. DOI: 10.1145/2155620.2155650.

[156] *Meet the Future of Data Center Rack Technologies*. Feb. 2013. URL: http://www.datacenterknowledge.com/archives/2013/02/20/meet-the-future-of-data-center-rack-technologies/ (visited on 01/25/2017).

[157] Matthias Meyer. "A True Hardware Read Barrier". In: *Proceedings of the 5th International Symposium on Memory Management*. ISMM '06. New York, NY, USA: ACM, 2006, pp. 3–16. ISBN: 978-1-59593-221-1. DOI: 10.1145/1133956.1133959.

[158] *Micron Technology, Inc. - System Power Calculator Information*. URL: https://www.micron.com/support/tools-and-utilities/power-calc (visited on 11/21/2017).

[159] Eliot Moss. "The Cleanest Garbage Collection: Technical Perspective". In: *Commun. ACM* 56.12 (Dec. 2013), pp. 100–100. ISSN: 0001-0782. DOI: 10.1145/2534706.2534725.

[160] S.J. Mullender et al. "Amoeba: A Distributed Operating System for the 1990s". In: *Computer* 23.5 (May 1990), pp. 44–53. ISSN: 0018-9162. DOI: 10.1109/2.53354.

[161] Jamin Naghmouchi, Daniele Paolo Scarpazza, and Mladen Berekovic. "Small-Ruleset Regular Expression Matching on GPGPUs: Quantitative Performance Analysis and Optimization". In: *Proceedings of the 24th ACM International Conference on Supercomputing*. ICS '10. Tsukuba, Ibaraki, Japan, 2010, pp. 337–348.

[162] *Nailgun Background*. URL: http://www.martiansoftware.com/nailgun/background.html (visited on 09/28/2017).

[163] Vijaykrishnan Narayanan and Mario L. Wolczko. *Java Microarchitectures*. Springer Publishing Company, Incorporated, 2012. ISBN: 978-1-4613-5341-6.

[164] Khanh Nguyen et al. "FACADE: A Compiler and Runtime for (Almost) Object-Bounded Big Data Applications". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 675–690. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694345. (Visited on 01/25/2017).

[165] Khanh Nguyen et al. "Yak: A High-Performance Big-Data-Friendly Garbage Collector". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. GA: USENIX Association, 2016, pp. 349–365. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/nguyen.

[166] *Now Available: X1 Instances, the Largest Amazon EC2 Memory-Optimized Instance with 2 TB of Memory*. URL: //aws.amazon.com/about-aws/whats-new/2016/05/now-available-x1-instances-the-largest-amazon-ec2-memory-optimized-instance-with-2-tb-of-memory/ (visited on 09/29/2017).

[167] *NVDIA Developer Resources for Deep Learning and AI*. URL: https://www.nvidia.com/en-us/deep-learning-ai/developer/ (visited on 09/29/2017).

[168] Takeshi Ogasawara, Hideaki Komatsu, and Toshio Nakatani. "A Study of Exception Handling and Its Dynamic Optimization in Java". In: *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA '01. New York, NY, USA: ACM, 2001, pp. 83–95. ISBN: 978-1-58113-335-6. DOI: 10.1145/504282.504289.

[169] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*. USENIX ATC'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 305–320. ISBN: 978-1-931971-10-2. URL: http://dl.acm.org/citation.cfm?id=2643634.2643666 (visited on 01/03/2015).

[170] John K. Ousterhout et al. "The Sprite Network Operating System". In: *Computer* 21.2 (Feb. 1988), pp. 23–36. ISSN: 0018-9162. DOI: 10.1109/2.16. (Visited on 03/22/2015).

[171] John Ousterhout et al. "The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM". In: *SIGOPS Oper. Syst. Rev.* 43.4 (Jan. 2010), pp. 92–105. ISSN: 0163-5980. DOI: 10.1145/1713254.1713276. (Visited on 03/23/2015).

[172] Kay Ousterhout et al. "Making Sense of Performance in Data Analytics Frameworks". In: *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. Oakland, CA: USENIX Association, 2015. URL: https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout.

[173] Kay Ousterhout et al. "Performance Clarity As a First-Class Design Principle". In: *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*. HotOS '17. New York, NY, USA: ACM, 2017, pp. 1–6. ISBN: 978-1-4503-5068-6. DOI: 10.1145/3102980.3102981.

[174] Jian Ouyang et al. "SDA: Software-Defined Accelerator for Large-Scale DNN Systems". In: *2014 IEEE Hot Chips 26 Symposium (HCS)* (2014), pp. 1–23. DOI: doi.ieeecomputersociety.org/10.1109/HOTCHIPS.2014.7478821.

[175] Shoumik Palkar et al. "Weld: A Common Runtime for High Performance Data Analytics". In: *8th Biennial Conference on Innovative Data Systems Research (CIDR 2017)*. 2017.

[176] Heidi Pan, Benjamin Hindman, and Krste Asanović. "Composing Parallel Software Efficiently with Lithe". In: *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '10. New York, NY, USA: ACM, 2010, pp. 376–387. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806639.

[177] Matthew Parkinson et al. *Project Snowflake: Non-Blocking Safe Manual Memory Management in .NET*. Tech. rep. July 2017. URL: https://www.microsoft.com/en-us/research/publication/project-snowflake-non-blocking-safe-manual-memory-management-net/.

[178] Ishwar Parulkar et al. "OpenSPARC: An Open Platform for Hardware Reliability Experimentation". In: *In Proc.Ofthe Workshop on Silicon Errorsin Logic - System Effects*. 2008.

[179] Avadh Patel et al. "MARSSx86: A Full System Simulator for X86 CPUs". In: *Design Automation Conference 2011 (DAC'11)*. 2011.

[180] Andrés Omar Portillo-Domínguez et al. "Load Balancing of Java Applications by Forecasting Garbage Collections". In: *2014 IEEE 13th International Symposium on Parallel and Distributed Computing* (2014). URL: http://ulir.ul.ie/handle/10344/4369 (visited on 08/07/2015).

[181] *Predictable Low Latency: "Cinnober on GC Pause-Free Java Applications through Orchestrated Memory Management"*. Tech. rep. URL: http://www.cinnober.com/sites/cinnober.com/files/news/Cinnober%20on%20GC%20pause%20free%20Java%20applications.pdf.

[182] *Predictive Analytics - Cloud Machine Learning Engine*. URL: https://cloud.google.com/ml-engine/ (visited on 09/25/2017).

[183] *Project Tungsten: Bringing Spark Closer to Bare Metal*. URL: https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html (visited on 07/28/2015).

[184] *Project Valhalla – Value Types – Jesper de Jong*. URL: http://www.jesperdj.com/2015/10/04/project-valhalla-value-types/ (visited on 01/25/2017).

[185] Wolfgang Puffitsch and Martin Schoeberl. "Non-Blocking Root Scanning for Real-Time Garbage Collection". In: JTRES '08.

[186] Andrew Putnam. "Large-Scale Reconfigurable Computing in a Microsoft Datacenter". In: *Hot Chips 26 Symposium (HCS)*. 2014, pp. 1–38.

[187] Andrew Putnam et al. "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 13–24. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665678.

[188] *Riscv-Opcodes: RISC-V Opcodes*. Aug. 2017. URL: https://github.com/riscv/riscv-opcodes.

[189] *Riscv-Poky: Port of the Yocto Project to the RISC-V ISA*. Aug. 2017. URL: https://github.com/riscv/riscv-poky.

[190] A. Rodchenko et al. "MaxSim: A Simulation Platform for Managed Applications". In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Apr. 2017, pp. 141–152. DOI: 10.1109/ISPASS.2017.7975286.

[191] Christopher J. Rossbach et al. "Dandelion: A Compiler and Runtime for Heterogeneous Systems". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. New York, NY, USA: ACM, 2013, pp. 49–68. ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522715. (Visited on 11/24/2013).

[192] Arjun Roy et al. "Inside the Social Network's (Datacenter) Network". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 123–137. ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2787472.

[193] Daniel Sanchez and Christos Kozyrakis. "ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems". In: *Proceedings of the 40th Annual International Symposium on Computer Architecture*. ISCA '13. New York, NY, USA: ACM, 2013, pp. 475–486. ISBN: 978-1-4503-2079-5. DOI: 10.1145/2485922.2485963. (Visited on 05/10/2014).

[194] Tony Savor et al. "Continuous Deployment at Facebook and OANDA". In: *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. May 2016, pp. 21–30.

[195] Martin Schoeberl. "JOP: A Java Optimized Processor". en. In: *On The Move to Meaningful Internet Systems 2003: OTM 2003 Workshops*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, Nov. 2003, pp. 346–359. ISBN: 978-3-540-20494-7 978-3-540-39962-9. DOI: 10.1007/978-3-540-39962-9_43. (Visited on 10/01/2017).

[196] Martin Schoeberl and Wolfgang Puffitsch. "Non-Blocking Object Copy for Real-Time Garbage Collection". In: *Proceedings of the 6th International Workshop on Java Technologies for Real-Time and Embedded Systems*. JTRES '08. New York, NY, USA: ACM, 2008, pp. 77–84. ISBN: 978-1-60558-337-2. DOI: 10.1145/1434790.1434802.

[197] Malte Schwarzkopf. "Operating System Support for Warehouse-Scale Computing". PhD Thesis. University of Cambridge Computer Laboratory, 2015.

[198] Malte Schwarzkopf et al. "Omega: Flexible, Scalable Schedulers for Large Compute Clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 351–364. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465386. (Visited on 09/24/2013).

[199] Timothy Sherwood et al. "Automatically Characterizing Large Scale Program Behavior". In: *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS X. New York, NY, USA: ACM, 2002, pp. 45–57. ISBN: 978-1-58113-574-9. DOI: 10.1145/605397.605403.

[200] José Simão, João Lemos, and Luís Veiga. "A 2 -VM : A Cooperative Java VM with Support for Resource-Awareness and Cluster-Wide Thread Scheduling". en. In: *On the Move to Meaningful Internet Systems: OTM 2011*. Ed. by Robert Meersman et al. Lecture Notes in Computer Science 7044. Springer Berlin Heidelberg, 2011, pp. 302–320. ISBN: 978-3-642-25108-5 978-3-642-25109-2. URL: http://link.springer.com/chapter/10.1007/978-3-642-25109-2_20 (visited on 01/26/2015).

[201] Arjun Singh et al. "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network". In: *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*. SIGCOMM '15. New York, NY, USA: ACM, 2015, pp. 183–197. ISBN: 978-1-4503-3542-3. DOI: 10.1145/2785956.2787508.

[202] David Smiley and David Eric Pugh. *Apache Solr 3 Enterprise Search Server*. Packt Publishing Ltd, 2011.

[203] James E. Smith. "Decoupled Access/Execute Computer Architectures". In: *ACM Trans. Comput. Syst.* 2.4 (Nov. 1984), pp. 289–308. ISSN: 0734-2071. DOI: 10.1145/357401.357403.

[204] R. Smith et al. "Evaluating GPUs for Network Packet Signature Matching". In: *International Symposium on Performance Analysis of Systems and Software, 2009. ISPASS 2009*. Apr. 2009, pp. 175–184.

[205] *Speech API - Speech Recognition*. URL: https://cloud.google.com/speech/ (visited on 01/25/2017).

[206] Weibin Sun and Robert Ricci. *Augmenting Operating Systems With the GPU*. Tech. rep. University of Utah, 2010.

[207] Zehra Sura et al. "Compiler Techniques for High Performance Sequentially Consistent Java Programs". In: *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 2–13. ISBN: 978-1-59593-080-4. DOI: 10.1145/1065944.1065947.

[208] *Synopsys SAED_EDK32/28_CORE Databook*. Tech. rep. Version 1.0.0. 2012.

[209] Z. Tan et al. "RAMP Gold: An FPGA-Based Architecture Simulator for Multiprocessors". In: *Design Automation Conference*. June 2010, pp. 463–468.

[210] James Tandon. "The OpenRISC Processor: Open Hardware and Linux". In: *Linux J.* 2011.212 (Dec. 2011). ISSN: 1075-3583. URL: http://dl.acm.org/citation.cfm?id=2123870.2123876.

[211] Gil Tene, Balaji Iyengar, and Michael Wolf. "C4: The Continuously Concurrent Compacting Collector". In: *Proceedings of the International Symposium on Memory Management*. ISMM '11. New York, NY, USA: ACM, 2011, pp. 79–88. ISBN: 978-1-4503-0263-0. DOI: 10.1145/1993478.1993491. (Visited on 01/03/2015).

[212] David Terei and Amit Levy. "Blade: A Data Center Garbage Collector". In: *arXiv:1504.02578 [cs]* (Apr. 2015). arXiv: 1504.. URL: http://arxiv.org/abs/1504.02578 (visited on 07/28/2015).

[213] *The OpenCompute Project*. URL: http://www.opencompute.org/home/ (visited on 09/28/2017).

[214] *TIOBE Index — TIOBE - The Software Quality Company*. URL: https://www.tiobe.com/tiobe-index/ (visited on 09/28/2017).

[215] Dan Tsafrir et al. "System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications". In: *Proceedings of the 19th Annual International Conference on Supercomputing*. ICS '05. New York, NY, USA: ACM, 2005, pp. 303–312. ISBN: 1-59593-167-8. DOI: 10.1145/1088149.1088190. (Visited on 07/28/2015).

[216] *Understanding the Tuning Trade-Offs*. URL: https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/tuning_tradeoffs.html (visited on 09/29/2017).

[217] David Ungar et al. "Architecture of SOAR: Smalltalk on a RISC". In: *Proceedings of the 11th Annual International Symposium on Computer Architecture*. ISCA '84. New York, NY, USA: ACM, 1984, pp. 188–197. ISBN: 0-8186-0538-3. DOI: 10.1145/800015.808182.

[218] Kenton Varda. *Protocol Buffers: Google's Data Interchange Format*. 2008. URL: https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html.

[219] *Vega 3 Processor*. URL: http://www.azulsystems.com/products/vega/processor (visited on 01/03/2015).

[220] Ronald Veldema and Michael Philippsen. "Iterative Data-Parallel Mark & Sweep on a GPU". In: *Proceedings of the International Symposium on Memory Management*. ISMM '11. San Jose, California, USA, 2011, pp. 1–10.

[221] Abhishek Verma et al. "Large-Scale Cluster Management at Google with Borg". In: *Proceedings of the Tenth European Conference on Computer Systems*. EuroSys '15. New York, NY, USA: ACM, 2015, 18:1–18:17. ISBN: 978-1-4503-3238-5. DOI: 10.1145/2741948.2741964.

[222] James Vincent. *Microsoft Unveils New ARM Server Designs, Threatening Intel's Dominance*. Mar. 2017. URL: https://www.theverge.com/2017/3/9/14867310/arm-servers-microsoft-intel-compute-conference (visited on 09/29/2017).

[223] Nitsan Wakart. *Correcting YCSB's Coordinated Omission Problem*. Nov. 2015. URL: http://psy-lob-saw.blogspot.com/2015/03/fixing-ycsb-coordinated-omission.html (visited on 01/18/2016).

[224] David Wang et al. "DRAMsim: A Memory System Simulator". In: *SIGARCH Comput. Archit. News* 33.4 (Nov. 2005), pp. 100–107. ISSN: 0163-5964. DOI: 10.1145/1105734.1105748.

[225] Andrew Waterman et al. *The RISC-V Instruction Set Manual. Volume 1: User-Level ISA, Version 2.0*. Tech. rep. EECS Department, University of California, Berkeley, 2014.

[226] Tom White. *Hadoop: The Definitive Guide: The Definitive Guide*. O'Reilly Media, 2009.

[227] James R. Wilcox et al. "Verdi: A Framework for Implementing and Formally Verifying Distributed Systems". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. New York, NY, USA: ACM, 2015, pp. 357–368. ISBN: 978-1-4503-3468-6. DOI: 10.1145/2737924.2737958.

[228] Nicholas Wilt. *The Cuda Handbook: A Comprehensive Guide to Gpu Programming*. Pearson Education, 2013.

[229] Christian Wimmer et al. "Maxine: An Approachable Virtual Machine for, and in, Java". In: *ACM Trans. Archit. Code Optim.* 9.4 (Jan. 2013), 30:1–30:24. ISSN: 1544-3566. DOI: 10.1145/2400682.2400689. (Visited on 11/27/2017).

[230] M. Wolczko, G. Wright, and M. Seidl. "Methods and Apparatus for Marking Objects for Garbage Collection in an Object-Based Memory System". U.S. Patent 8,825,718. URL: http://www.google.com/patents/US8825718.

[231] Mario I. Wolczko and David M. Ungar. "United States Patent: 5900001 - Method and Apparatus for Optimizing Exact Garbage Collection Using a Bifurcated Data Structure". 5900001. May 1999.

[232] Mario Wolczko and Ifor Williams. "Multi-Level Garbage Collection in a High-Performance Persistent Object System". In: *POS*. 1992, pp. 396–418.

[233] Jonathan Woodruff et al. "The CHERI Capability Model: Revisiting RISC in an Age of Risk". In: *Proceeding of the 41st Annual International Symposium on Computer Architecuture*. ISCA '14. Piscataway, NJ, USA: IEEE Press, 2014, pp. 457–468. ISBN: 978-1-4799-4394-4. URL: http://dl.acm.org/citation.cfm?id=2665671.2665740.

[234] Greg Wright. "A Hardware-Assisted Concurrent & Parallel Garbage Collection Algorithm (Unpublished Manuscript)". 2008.

[235] Reynold S. Xin et al. "Shark: SQL and Rich Analytics at Scale". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, NY, USA: ACM, 2013, pp. 13–24. ISBN: 978-1-4503-2037-5. DOI: 10.1145/2463676.2465288. (Visited on 03/22/2015).

[236] Neeraja J. Yadwadkar et al. "Multi-Task Learning for Straggler Avoiding Predictive Job Scheduling". In: *J. Mach. Learn. Res.* 17.1 (Jan. 2016), pp. 3692–3728. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=2946645.3007059 (visited on 11/27/2017).

[237] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. "Computer Performance Microscopy with SHIM". In: *Proceedings of the 42nd Annual International Symposium on Computer Architecture*. ISCA '15. Portland, Oregon: ACM, 2015, pp. 170–184. ISBN: 978-1-4503-3402-0. DOI: 10.1145/2749469.2750401.

[238] Matei Zaharia et al. "Discretized Streams: An Efficient and Fault-Tolerant Model for Stream Processing on Large Clusters". In: *Proceedings of the 4th USENIX Conference on Hot Topics in Cloud Ccomputing*. HotCloud'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 10–10. URL: http://dl.acm.org/citation.cfm?id=2342763.2342773 (visited on 03/22/2015).

[239] Matei Zaharia 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'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 2–2. URL: http://dl.acm.org/citation.cfm?id=2228298.2228301 (visited on 12/29/2014).

[240] Xiao Zhang et al. "CPI2: CPU Performance Isolation for Shared Compute Clusters". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. EuroSys '13. New York, NY, USA: ACM, 2013, pp. 379–391. ISBN: 978-1-4503-1994-2. DOI: 10.1145/2465351.2465388. (Visited on 09/24/2013).

[241] Wenzhang Zhu, Cho-Li Wang, and F.C.M. Lau. "JESSICA2: A Distributed Java Virtual Machine with Transparent Thread Migration Support". In: *2002 IEEE International Conference on Cluster Computing, 2002. Proceedings*. 2002, pp. 381–388. DOI: 10.1109/CLUSTR.2002.1137770.

[242] J. N Zigman and R. Sankaranarayana. *dJVM-A Distributed JVM on a Cluster*. Tech. rep. Australian National University, 2002.

[243] *Zing ReadyNow Technology Solves Java Warmup Problems*. URL: https://www.azul.com/products/zing/readynow-technology-for-zing/ (visited on 09/28/2017).

[244] *Zing Runtime for Java – Zing Solves Java Garbage Collection Problems and Java Performance Issues*. URL: https://www.azul.com/products/zing/ (visited on 10/01/2017).

Use of Previously Published or Co-Authored Material

This dissertation is based on the papers listed below. These papers were co-authored with my advisors, as well as several additional collaborators.

- **Full-System Simulation of Java Workloads with RISC-V and the Jikes Research Virtual Machine**. Martin Maas, Krste Asanović, John Kubiatowicz. 1st Workshop on Computer Architecture Research with RISC-V (CARRV '17), Boston, MA, October 2017
- **Return of the Runtimes: Rethinking the Language Runtime System for the Cloud 3.0 Era**. Martin Maas, Krste Asanović, John Kubiatowicz. 16th Workshop on Hot Topics in Operating Systems (HotOS '17), Whistler, Canada, May 2017
- **Grail Quest: A New Proposal for Hardware-assisted Garbage Collection**. Martin Maas, Krste Asanović, John Kubiatowicz. Workshop on Architectures and Systems for Big Data (ASBD '16), Seoul, Korea, June 2016
- **Taurus: A Holistic Language Runtime System for Coordinating Distributed Managed-Language Applications**. Martin Maas, Krste Asanović, Tim Harris, John Kubiatowicz. International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16), Atlanta, Georgia, April 2016
- **Trash Day: Coordinating Garbage Collection in Distributed Systems**. Martin Maas, Tim Harris, Krste Asanović, John Kubiatowicz. 15th Workshop on Hot Topics in Operating Systems (HotOS '15), Kartause Ittingen, Switzerland, May 2015
- **The Case for the Holistic Language Runtime System**. Martin Maas, Krste Asanović, Tim Harris, John Kubiatowicz. International Workshop on Rack-scale Computing (WRSC '14), Amsterdam, Netherlands, April 2014
- **GPUs as an Opportunity for Offloading Garbage Collection**. Martin Maas, Philip Reames, Jeffrey Morlan, Krste Asanović, Anthony D. Joseph, John Kubiatowicz. 2012 International Symposium on Memory Management (ISMM '12), Beijing, China, June 2012

In addition to these papers, the dissertation incorporates a paper on hardware support for garbage collection that is under double-blind submission at the time of writing. The paper under submission is co-authored by myself, Krste Asanović and John Kubiatowicz.

The majority of material in this thesis is directly taken from the above papers, including (but not limited to) text, figures and experimental results. Most of the text has been edited to include additional details and improve the wording or flow. Furthermore, in accordance with university regulations, the following steps have been taken:

- Permission for using any of the material from the above papers within this thesis has been requested and was granted by the Dean of the Graduate Division at UC Berkeley. Permission to integrate the material under submission was granted as well.
- Permission to use any material from the above papers has been given by all co-authors of these papers, and was received in writing (e-mail).
- The material from these papers has been "incorporated into a larger argument that binds together the whole thesis"[1].

While the final editing pass is my own work, the work presented in the original papers was oftentimes a close collaboration between all of the authors. As such, I want to explicitly clarify that I do not claim sole credit for the work in this thesis.

Specifically, I want to highlight my co-authors' contributions to Chapter 6. Philip Reames and I worked very closely on the design and implementation of the entire project, and it is impossible to distinguish most of these individual contributions. The paper itself was a close collaboration as well, with both of us working on all sections of the text. Notwithstanding, I want to specifically credit Philip for collecting the data that underpins the preliminary heap analysis in Section 6.3 (the actual analysis of this data is joint work). Further, I want to credit Jeffrey Morlan with implementing the reference graph integration in JikesRVM.

Taurus was implemented by myself at UC Berkeley, but precursor work was performed at Oracle Labs in close collaboration with Tim Harris.

[1] `https://grad.berkeley.edu/policies/guides/thesis-filing/`

Funding Information

The following funding information covers all authors from papers that have been incorporated into this dissertation.

GPU-based Garbage Collection (ISMM'12): Research supported by Microsoft (Award #024263) and Intel (Award #024894) funding and by matching funding by U.C. Discovery (Award #DIG07-10227). Additional support comes from Par Lab affiliates National Instruments, Nokia, NVIDIA, Oracle, and Samsung. Philip Reames was also supported by the National Science Foundation (Award #CCF-1017810).

Holistic Runtime Systems (WRSC'14, HotOS'15, ASPLOS'16, Hot'17): Precursor work was done at Oracle Labs, Cambridge. Research at UC Berkeley was partially funded by DARPA Award Number HR0011-12-2-0016, DOE grant #DE-AC02-05CH11231, the Center for Future Architecture Research, a member of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA, and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Huawei, LG, NVIDIA, Oracle, and Samsung.

Hardware Support for Garbage Collection (ASBD'16, CARRV'17): Research was partially funded by DARPA Award Number HR0011-12-2-0016, DOE grant #DE-AC02-05CH11231, the STARnet Center for Future Architecture Research (C-FAR), and ASPIRE Lab sponsors and affiliates Intel, Google, HPE, Huawei, LGE, NVIDIA, Oracle, and Samsung.