

Garbage Collection Across the Stack: From Software to Near-Memory Acceleration

Zixian Cai
蔡子弦

A thesis submitted for the degree of
Doctor of Philosophy at
The Australian National University

June 2026

© Zixian Cai 2026

Typeset in Iowan Old Style, Concourse, and Triplicate by Lua^ATeX.

Except where otherwise indicated, this thesis is my own original work.

Zixian Cai
7 June 2026

To my late grandparents.

Acknowledgments

First and foremost, I thank my *shifu*, Steve Blackburn. In addition to your mentorship during my undergraduate studies, you have helped me gradually establish myself as an independent researcher over the past four years of my PhD. Rough occasions were more tolerable thanks to your constant supply of wisdom, optimism, and support. Not only have you passed on invaluable technical knowledge, but you also taught me the art of being a teacher, a mentor, and a leader. I could not have asked for a better advisor.

I thank my coadvisors: Mike Bond from The Ohio State University, and Martin Maas from Google DeepMind. I am incredibly grateful that you agreed to be on my panel, and set aside time from your busy schedules to meet with me every week. In addition to coauthoring manuscripts, you brought vast technical expertise, and helped me keep complex projects on course. This thesis would not have happened without your collaboration.

I thank my fellow PhD students in the lab: Ysaswini Gownivaripalli, Hayley Patton, Tianle Qiu, Kunal Sareen, and Wenyu Zhao. You lent an ear when programs inevitably crashed with SIGSEGV, or when C++ black magic caused third-degree burns. Your friendly banter made the lab a fun place to work. I also thank Yi Lin, Eduardo Souza, and Kunshan Wang for maintaining the MMTk plumbing, which is the foundation of much of the work presented in this thesis.

I had the privilege of working with several bright undergraduate students. Thank you for having faith in the projects I proposed, and for being patient as I learned how to be an effective advisor. I am proud of your achievements, and I am sure you will all go on to do great things. In particular, I thank Claire Huang for collaborating on the performance tracing project, which forms an important part of this thesis.

I thank my colleagues in the School of Computing for dragging me away from the world of garbage collection when I was working too hard. You generously gave me advice, and helped me navigate outside my immediate research area. I always looked forward to our weekly board games. Fabian Mühlböck was a constant source of chocolate and sweets, which were essential for long debugging or writing sessions. The School Admin Team provided invaluable institutional knowledge that helped me wrangle the bureaucracy.

I thank the broader SIGPLAN community for fostering a welcoming and inclusive environment. Chatting with you during hallway tracks was often the highlight of a conference. In particular, I had the privilege of being part of the [SIGPLAN Audio/Visual](#) team. Although organizing a conference is stressful, it was fun working with you all, and it let me contribute back to the community.

Doing a PhD is rewarding, but at times also difficult. I thank all of my friends who

made my life brighter. There are too many of you to list here, but special thanks go to Zoey (Zhuo) Chen, Jiawen He, Adrian Herrera, Claire Huang, Ashleigh Johannes, Anh Phuong Le, Tiange Liu, Alex Mirrlees-Black, Paige Reeves, Jack Stodart, Brenda Wang, Yichen Wang, and Bingyi Ye. I thank the Factorio crew, in particular Alex, Fabian, Jack, Kunal, and Paige, for reminding me that there is always more hard “work” to do after work. I thank Kunal for being a wonderful roommate, and for keeping everything organized and running smoothly. I thank Brenda for always being just a phone call away, sharing stories, giving advice, and reminding me that there is a world outside academia.

Finally, I thank my family. “It takes a village to raise a child.” My achievements today would not have been possible without your support.

This research was supported by an Australian Government Research Training Program (RTP) Scholarship [doi.org:10.82133/C42F-K220](https://doi.org/10.82133/C42F-K220).

Use of Previously Published or Coauthored Material

This thesis is based on the papers listed below. These papers were coauthored with my advisors, as well as several additional collaborators.

1. *MAGC-DIMM: A Near-Memory Processing Garbage Collection Accelerator on DIMMs* by Cai et al. [2025]: I was the lead author on the paper.
2. *Rethinking Java Performance Analysis* by Blackburn et al. [2025]: This was joint work with the DaCapo benchmark developers. I helped debug several benchmarks, wrote the performance analysis tools, and conducted data analysis for the final paper.
3. *Improving Garbage Collection Observability with Performance Tracing* by Huang et al. [2023]: This was based on Huang's undergraduate research project, which I proposed the topic for and supervised. I wrote the majority of the final paper, and closely engaged with the experimental design and data analysis.
4. *Distilling the Real Cost of Production Garbage Collectors* by Cai et al. [2022]: I was the lead author on the paper.

Abstract

Garbage collection (GC) provides memory safety, while improving programmer productivity. These properties have led to the increasing deployment of garbage-collected languages, from low-power mobile devices to latency-critical servers. This ubiquity has driven significant investment in GC design and implementation, hoping to improve GC's efficiency and responsiveness.

Despite the considerable resources dedicated to GC research, a surprising lack of clarity remains regarding the true costs of GC. As I will show in this thesis, when implemented in software, GC imposes substantial overheads. Without this quantification, there is little incentive to invest in techniques requiring significant research and engineering effort, such as near-memory GC accelerators suitable for integration into commodity machines.

My thesis is that software GCs are surprisingly costly, and that new techniques across the stack, such as hardware support, can deliver significant improvements.

I develop three new methodologies and techniques across the computational stack, from software to hardware. First, I present the *lower bound overhead (LBO)* methodology, and quantify the substantial costs imposed by production software GCs. This revelation strongly motivates the development of kernel-based GC performance tracing tools, and a near-memory GC accelerator. Second, I show how kernel-based *performance tracing* improves GC observability, providing agility in performance debugging and optimization. Finally, I design and evaluate MAGC-DIMM—a first step towards offloading GC to commodity DIMMs by integrating acceleration logic directly into the buffer devices.

The LBO methodology has helped the research community evolve our collective methodological foundations, and it has been used to evaluate GCs on platforms from phones to servers. The GC observability tooling is now a part of MMTk—a popular memory management framework—and has benefited the development of new research collectors, and the productization of existing ones. MAGC-DIMM demonstrates that near-memory GC acceleration can be feasible and efficient. Compared with a 16-hardware-thread CPU, MAGC-DIMM uses $72 \times$ less cache area and significantly less logic area, while still achieving 68 % throughput.

My thesis reaffirms that GC is by no means a solved problem, and shows that we can reap substantial benefits by designing and optimizing GC across the entire stack. This thesis encourages the research community to reexamine how we build, evaluate, and optimize low-level systems, such as GC. It also reinvigorates research into GC accelerators and provides a solid foundation for realizing them on a wider range of NMP architectures, encouraging their adoption in commodity hardware.

Contents

Acknowledgments	vii
Use of Previously Published or Coauthored Material	ix
Abstract	xi
1 Introduction	1
1.1 Motivation	1
1.2 Contributions and Results	3
1.3 Thesis Outline	4
2 Background	7
2.1 Organization of Garbage Collectors and Challenges	7
2.1.1 Allocation	7
2.1.2 Identification	8
2.1.3 Reclamation	8
2.2 Tracing GC	8
2.2.1 Mechanism and Canonical Implementation	8
2.2.2 Parallel Tracing and Scalability	10
2.2.3 Barriers	10
2.2.4 Concurrent Tracing	10
2.2.5 Tracing on CPUs	11
2.3 Summary	12
3 Distilling the Real Costs of Garbage Collectors	13
3.1 Two Key Problems in Understanding Garbage Collection Costs	14
3.1.1 Quantifying Real GC Costs through Distillation	14
3.1.2 Misinterpretation of Evaluation Results	15
3.2 Garbage Collection Cost is Hard to Measure	16
3.2.1 Related Work	16
3.2.2 Tightly-Coupled Costs	17
3.2.3 Indirect Costs and Benefits	18
3.2.4 Aspects of GC Costs in OpenJDK	18
3.3 The Lower Bound Overhead Methodology	20
3.3.1 Definition with Examples	20
3.3.2 Discussion	23
3.4 Evaluation Methodology	24

3.4.1	Benchmarks and Latency Measures	24
3.4.2	Cost Metrics	25
3.4.3	JVM Parameters	25
3.4.4	Execution Methodology	27
3.4.5	Hardware	27
3.4.6	Software	27
3.5	Case Study: Collectors in OpenJDK	28
3.5.1	Results: Costs of Garbage Collection	29
3.5.2	Analysis of Results	29
3.5.3	Common Misinterpretation of Evaluation Results	34
3.6	Recommendations	39
3.7	Discussion and Summary	40
4	Improving GC Observability with Performance Tracing	41
4.1	A Lack of Observability in GC	41
4.2	A Taxonomy of Performance Analysis Tools	44
4.2.1	Counting	44
4.2.2	Sampling	44
4.2.3	Tracing	46
4.3	Tracing Technologies and Existing Use Cases	46
4.3.1	Tracing Technologies	47
4.3.2	Tracing in Managed Language Virtual Machines	49
4.3.3	Choosing a Tracing Technology	50
4.4	Adding Tracing to GC	51
4.4.1	MMTk and Its Work Packets	51
4.4.2	Tracepoints Added	52
4.5	Methodology	54
4.6	Tracing Overheads	55
4.7	Case Studies	58
4.7.1	Heap Traversal Scalability: The ProcessEdges Work Packet	58
4.7.2	Allocator Scalability	64
4.8	Discussion and Summary	65
5	MAGC-DIMM: A Near-Memory Processing Garbage Collection Accelerator on DIMMs	67
5.1	On-DIMM GC: Three Questions	68
5.1.1	Why On-DIMM GC	68
5.1.2	Challenges and Responses	68
5.2	DRAM Organization and Contiguity	70
5.3	Taxonomy of Hardware Garbage Collection and Related Work	71
5.3.1	Target Application	72
5.3.2	Generality	72
5.3.3	Placement	74
5.3.4	Fine-Grained GC Offloading to HMC	74

5.3.5	Graph Processing Hardware	74
5.4	MAGC-DIMM Architecture	75
5.4.1	A Focus on Heap Traversal	75
5.4.2	Making DIMMs Suitable for GC	75
5.4.3	Distributing GC Across DIMMs	76
5.4.4	MAGC-DIMM Components	78
5.4.5	Passing GC Messages Efficiently	82
5.5	Workload Analysis and Design Optimizations	82
5.5.1	Quantifying Communication	83
5.5.2	Improving Object Scanning	84
5.5.3	Improving Utilization via Better Distributed Termination Detec- tion	86
5.6	Evaluation Methodology	88
5.6.1	Benchmarks and Runtime	88
5.6.2	Baseline	88
5.6.3	Hardware Configuration	88
5.7	Evaluation	90
5.7.1	Performance and Area Efficiency	90
5.7.2	Behavioral Correctness	92
5.7.3	MAGC-DIMM Scalability	93
5.7.4	Impact of Load Imbalance	93
5.8	Future Extensions	94
5.9	Summary	95
6	Discussion	97
6.1	Rethinking Specialized Systems	97
6.2	Collective Investment in Research Infrastructure	98
7	Conclusion	99
7.1	Future Work	100
7.1.1	Microarchitectural GC Analysis with Better Delineation	100
7.1.2	User-Space Tracing	100
7.1.3	End-to-End Integration and Evaluation of MAGC-DIMM	101
	Bibliography	103

List of Figures

2.1	A GC tracing algorithm that enqueues object references before marking, and then scans the referenced objects.	9
3.1	Estimating the real cost of GC by using the minimum distilled application cost.	15
3.2	Lower bounds on the overheads of five OpenJDK 21 production garbage collectors.	28
3.3	GC pause time vs. metered latency for lusearch in a $3.0 \times \text{heap}$	38
3.4	GC pause time continues to be (mis)used as a proxy for responsiveness.	38
4.1	Inserting a USDT tracepoint into Rust code.	48
4.2	A probe written in the bpftrace language.	48
4.3	Distribution of packet sizes for each benchmark.	59
4.4	The scalability of MMTk's heap traversal phase for each benchmark.	60
4.5	Per-work packet tracepoints visualize scalability problems in Perfetto UI.	61
4.6	A regression in allocator slow path performance on lusearch visualized as a cumulative density function.	65
5.1	Hierarchy of DRAM components.	70
5.2	MAGC-DIMM is implemented inside buffer devices of interconnected DIMMs.	76
5.3	Examples of a normal object and an object array splitting across DIMMs.	77
5.4	A message-passing distributed GC algorithm for scanning non-array objects.	78
5.5	Components inside the buffer device.	79
5.6	Memory contiguity and the number of GC units affect the communication overhead.	83
5.7	The fractions of remote and local messages for each type across all benchmarks.	84
5.8	The distribution of lengths of contiguous runs of outgoing reference fields encountered during tracing.	84
5.9	Enhancing Fig. 5.4 with run-length encoding and eager loading.	85
5.10	Our optimizations reduce messages compared to the baseline.	86
5.11	GC execution traces from the fop visualizing the benefits of a better termination detection mechanism.	87
5.12	Heap traversal speedup of MAGC-DIMM against a software baseline across benchmarks.	90

5.13	RTL prototype scalability across benchmarks.	93
5.14	Load imbalance for the MAGC-DIMM prototype.	94

List of Tables

3.1	Garbage collectors used in the lower bound overhead studies.	19
3.2	The total CPU cycles consumed when running the DaCapo benchmark h2.	21
3.3	Distilling out cycles used during stop-the-world (STW) pauses from the total cycles.	22
3.4	The empirical lower bound (LBO) on the cycle cost for each collector. .	22
3.5	Minimum heap size required to run each benchmark.	26
3.6	Time LBOs averaged over 16 benchmarks.	30
3.7	Cycle LBOs averaged over 16 benchmarks.	30
3.8	Time LBOs at $3 \times \text{heap}$	31
3.9	Cycle LBOs at $3 \times \text{heap}$	32
3.10	Percent of wall-clock time spent in STW pauses averaged over 16 benchmarks.	36
3.11	Percent of total cycles spent in STW pauses averaged over 16 benchmarks.	36
4.1	Specifications for the hardware used in the study.	54
4.2	Tracing overheads on Zen 3.	56
4.3	Summary of overheads for Coffee Lake.	57
5.1	A taxonomy of hardware GC.	73
5.2	System configuration.	89
5.3	Performance and area comparison of MAGC-DIMM against the software baseline.	91

Introduction

This thesis presents a novel architecture for offloading garbage collection (GC) onto DIMMs. Before embarking on a completely new hardware design, I first needed to establish the potential gain. Surprisingly, the literature did not hold an answer to this most fundamental question. Thus, the first contribution of this thesis is a new methodology that measures the real costs of software GC. The second contribution emerged from the need for agile performance analysis and debugging, leading to the development of novel tools that allow us to understand GC behavior in new ways. The third and final contribution is the design and evaluation of the on-DIMM GC accelerator itself.

In this chapter, I first discuss the broader context of GC research and engineering in which I carry out this work, and I point out deficiencies that hold this field back. I then outline the methodologies and techniques I develop across the stack, from software to near-memory acceleration, to address these deficiencies.

1.1 Motivation

Complex and pervasive software systems underpin society. GC helps us build *correct* software by offering *memory safety*. This thesis aims to build a near-memory GC accelerator, which could help us build software that is both *correct* and *efficient*.

Lack of memory safety has long been the leading cause of software vulnerabilities, accounting for more than half of the discovered cases in recent surveys [Stoep and Zhang, 2019; Microsoft Corporation, 2019]. Error-prone manual memory management in languages such as C and C++ causes these memory bugs. With the increasing deployment of software systems in mission-critical roles, memory safety is paramount.

In recent years, there has been a renewed call for the adoption of memory-safe languages, which eliminate memory vulnerabilities by design [Lord, 2023; United States Cybersecurity and Infrastructure Security Agency et al., 2023]. The predominant approach, used by languages like Java, Go, and Swift, is automatic memory management, also known as GC. Other examples include Rust, which statically ensures memory safety via its ownership model.

Adopting garbage-collected languages over unsafe ones is a sociotechnical challenge, where one of the most common concerns is the high performance overheads of

GC [Stroustrup, 2021]. However, this perceived performance cost apparently contradicts the promises of low costs from newer collector designs [Johansson, 2021; Kennke, 2021; Lidén, 2021].

Given this apparent contradiction, the natural question is: “Should we even consider running GC on anything other than conventional CPUs?”

Research Question 1. What are the real costs of GC?

There is a surprising lack of clarity regarding the *real* costs of GC, despite a substantial literature on how GCs compare, how they are designed, and their key sources of cost [Blackburn et al., 2008, 2004a; Zorn, 1990; Hosking et al., 1992; Blackburn and Hosking, 2004; Yang et al., 2012; Huang et al., 2004; Zorn, 1993; Blackburn et al., 2006; Zhao and Blackburn, 2020; Sareen and Blackburn, 2022]. This is a *fundamental* problem. Without a clear understanding of GC costs, we risk: i) discouraging the adoption of GC where the benefits outweigh the costs, and ii) failing to provide sufficient incentives to invest in the development of GC techniques that could substantially reduce the costs but require significant research and engineering effort, such as a hardware GC accelerator.

The *first contribution* of this thesis is the *lower bound overhead (LBO)* methodology, which allows us to establish a clear lower bound on GC overheads for the first time, and shows how substantial software GC costs are. With this clear motivation, in the rest of the thesis, I focus on an on-DIMM GC accelerator design.

However, deficiencies in existing GC debugging and optimization methods hinder the development of any high-performance collector, whether in software or hardware. The reliance on logging and sampling lacks *agility* and *fidelity*, preventing a fast feedback loop for identifying potential optimizations, and for evaluating the effectiveness of new changes. These deficiencies demand a new approach to GC debugging and optimization.

Research Question 2. How can we better debug and optimize GCs?

Debugging and optimizing GCs has been part art, part science. Researchers and practitioners rely on received wisdom [Oracle, 2021a], and primitive telemetry, such as *sampling*, *logging*, and *counting*, to reason about collector behavior and drive optimization efforts. These tools critically lack *observability*, which is the ability to “ask arbitrary questions about [the GC] without—and this is the key part—having to know ahead of time what [one] wanted to ask” [Hound Technology, Inc., 2018].

This lack of observability hinders our ability to investigate the inner workings of collectors, and thus our ability to generate new insights that might advance new designs and engineering practices. The *second contribution* of this thesis is the improvement of GC observability via kernel-based performance tracing.

Despite mainstream GC research focusing on the design and implementation of software GCs, there has been a long history of attempts to implement GC in hardware since the 1980s. Some proposals implement specific languages in hardware [Steele Jr. and Sussman, 1980; Ungar et al., 1984; Williams and Wolczko, 1990; ARM Limited, 2008; Puffitsch, 2011; Ramsay and Stewart, 2024], while others involve invasive changes

to the host [Click et al., 2005; Joao et al., 2009; Tang et al., 2015; García et al., 2021]. The constraints of these approaches make them unattractive for adoption in modern computational systems, where general-purpose software and commodity hardware are dominant.

Research Question 3. Can we build a near-memory GC accelerator suitable for integration into general-purpose commodity machines?

Near-memory GC occupies an attractive point in the design space. Such a design limits most of the changes to the memory subsystem, and provides ample internal memory bandwidth that might help memory-intensive GC tasks. While previous proposals [Jang et al., 2019; Thomas et al., 2022] explored such an approach, they relied on obsolete technology, leaving the feasibility of GC on modern, commodity memory systems an open question.

Thus, *the third and final* contribution of this thesis is the design and evaluation of MAGC-DIMM—a first step towards offloading GC to DIMMs that integrates acceleration logic directly into the buffer devices. This approach could be more easily integrated into the DIMMs of commodity machines, encouraging adoption.

1.2 Contributions and Results

The core contributions of this thesis are the responses to the above three problems. My thesis is that, *software GCs are surprisingly costly, and that new techniques across the stack, such as hardware support, can deliver significant improvements.*

Lower bound overhead methodology The first contribution of the thesis is the *lower bound overhead (LBO)* methodology (Chapter 3), which transforms how we quantify the hard-to-measure overheads of critical functionality, such as GC.

Applied to GC, LBO reveals, for the first time, the substantial costs of software GC—OpenJDK’s heavily-tuned default production G1 collector [Detlefs et al., 2004] incurs 16 % overheads even in the most favorable settings. The newer low-pause collectors [Flood et al., 2016; Lidén and Karlsson, 2018] in OpenJDK incur even higher overheads—63 % even in the most favorable situations. The surprisingly expensive overhead of software GCs provides a clear motivation for the rest of this thesis. Furthermore, this worrying regression of GC overheads in newer collectors highlights the danger of neglecting our field’s methodological foundations.

Kernel-based performance tracing The second contribution of this thesis is improving the *observability* of GC via kernel-based performance tracing (Chapter 4). Tracepoints are nops that the kernel can dynamically patch when they are attached. They provide flexibility, yet incur negligible overheads when not in use. When combined with the modular design of the MMTk memory management framework, just a few tracepoints (totaling just 14 lines of code) provide a rich perspective on collector behavior.

I demonstrate that tracing is well-suited to GC performance debugging via several case studies. While using a fixed underlying set of tracepoints, we build tracing tools that crisply highlight several classes of performance bugs, including poor scalability in multi-threaded GCs, and lock contention in the allocation sequence. These observations uncover optimization opportunities in collectors, and even reveal bugs in application programs. These results show that tracing can be an invaluable part of our toolkit when working with complex, low-level systems such as GC.

On-DIMM GC accelerator The third and final contribution of this thesis is the design and evaluation of an on-DIMM GC accelerator MAGC-DIMM (Chapter 5). Such an approach introduces a fundamental challenge: memory interleaving across DIMMs frequently splits objects and impedes pointer chasing.

We address these problems by leveraging inter-DIMM communication and design a new distributed GC protocol based on message passing. We further optimize our design by exploiting GC-specific properties to reduce communication and idle times.

Our evaluation, utilizing both event-driven simulation and an RTL FPGA prototype, establishes the feasibility and area efficiency of our design. MAGC-DIMM uses $72 \times$ less cache area than a CPU. While our primary goal is proving feasibility, results show our design achieves 68 % throughput of a 16-thread high-end CPU, demonstrating that automatic memory management can achieve competitive performance at a fraction of the hardware footprint.

Our results show that near-memory GC can be practical. By offloading GC onto DIMMs, our design could be more easily integrated into the DIMMs of commodity machines, from low-power laptops to general-purpose servers. Our design also provides a foundation for future research on offloading complex system tasks to the memory hierarchy, and encourages future research on hardware support for managed languages.

1.3 Thesis Outline

The body of this thesis is structured around the three core contributions I listed in the previous section: i) the lower bound overhead (LBO) methodology that reveals the surprisingly substantial overheads incurred by software GCs, ii) kernel-based performance tracing that improves GC observability, and iii) the design and evaluation of MAGC-DIMM—an on-DIMM GC accelerator.

In Chapter 2, I start with an overview of the different components used in canonical GC algorithms. Although conceptually simple, the implementation of these components is complex in heavily-tuned production software GCs. This complexity leads to a lack of clarity about their costs, and about how we can better analyze and optimize them.

Chapter 3 describes the lower bound overhead (LBO) methodology, revealing the real costs incurred by software GCs. Chapter 4 shows how kernel-based performance tracing can help us analyze and debug GC performance *cheaply*, *flexibly*, and with *high*

fidelity.

These two contributions set the stage for the next chapter by clearly motivating a GC accelerator design, and by providing new ways to generate insights and uncover optimization opportunities when building an accelerator. [Chapter 5](#) then presents the design and evaluation of MAGC-DIMM, an on-DIMM GC accelerator, to concretely explore a novel paradigm of offloading GC onto DIMMs.

Finally, I discuss some of the lessons learned during this journey in [Chapter 6](#), and [Chapter 7](#) concludes with a summary of this thesis and future work.

Background

In this chapter, I give an overview of three key components of garbage collectors (GCs): allocation, identification, and reclamation ([Section 2.1](#)). In particular, I discuss a key identification mechanism used in production GCs—*tracing* ([Section 2.2](#)). In high-performance collectors, tracing is done in parallel (multithreaded), and sometimes concurrently with respect to the application. These characteristics of GC tracing have major implications for GC performance analysis and debugging ([Chapters 3 and 4](#)), as well as for how GC can be offloaded onto DIMMs ([Chapter 5](#)).

2.1 Organization of Garbage Collectors and Challenges

Managed languages, such as Java, provide an important run-time service: automatic memory management, also known as garbage collection. The field of GC research has made great strides since the development of the first GC algorithms in the 1960s [[McCarthy, 1960](#); [Fenichel and Yochelson, 1969](#); [Cheney, 1970](#)]. To better understand these different algorithms, we consider three key aspects: allocation, identification, and reclamation [[Blackburn and McKinley, 2008](#); [Jones et al., 2011](#)].

2.1.1 Allocation

The allocation mechanism concerns the placement of freshly constructed objects in unused heap space. Two common types of allocation schemes are bump-pointer allocators, and free-list allocators.

Bump-pointer allocators are conceptually simple: starting with a contiguous chunk of free memory, they place newly constructed objects consecutively until the available space is exhausted. Free-list allocators organize unused heap space into linked lists (*free lists*), and accommodate new objects by finding good fits from these free lists.

Regardless of the allocation mechanism used, high-performance allocators are implemented using a “fast-and-slow-path” paradigm [[Blackburn et al., 2004b](#)]. The allocator comprises an efficient common-case fast path, which is usually thread-local and unsynchronized, and an infrequently taken slow path, which is used only when the fast path exhausts its cache of available memory.

Although the allocation fast path is logically part of GC, it is often compiled and optimized with the application code for performance reasons. This tight coupling makes it difficult to measure the allocation costs of GC [Blackburn et al., 2004a]. I discuss this challenge further in Chapter 3.

The allocation slow path is less performance-critical than the fast path, but in applications with high parallelism and fast allocation rates, the scalability of the slow path can become a major concern. As a case study in Chapter 4, I show how kernel-based performance tracing tools can identify lock contention in the allocation slow path.

2.1.2 Identification

An important part of GC is the identification of objects to reclaim. The two main strategies are reference counting (RC) and tracing.

In reference counting, each object has a counter tracking the number of incoming references [Collins, 1960]. The compiler usually maintains these counters by inserting increments and decrements as the reference fields of objects are changed. Once an object reaches a reference count of zero, it can be reclaimed. Even though naïve RC is very expensive, RC-based collectors can be performant [Shahriyar et al., 2012], as demonstrated by recent high-performance research collectors [Shahriyar et al., 2013, 2014; Zhao et al., 2022a]. One problem with RC, however, is its inability to collect cycles. Therefore, even these state-of-the-art RC collectors employ backup tracing.

Tracing is the predominant identification mechanism used in production GCs. Reachable objects are identified by a transitive closure over the heap graph via pointer chasing; the remaining objects are implicitly unreachable and thus can be reclaimed. Due to the irregular and memory-intensive pointer chasing, tracing collectors have unique performance challenges, which I tackle in this thesis across the computational stack, spanning software, kernel, and hardware. I discuss tracing further in Section 2.2.

2.1.3 Reclamation

Though not a focus of this thesis, I now give a brief overview of reclamation. Reclamation concerns reusing the space occupied by unreachable objects. The reclamation mechanism is often dictated by the allocation mechanism. For example, for heaps governed by a bump-allocator scheme, space occupied by unreachable objects can only be reclaimed en masse by resetting the bump pointer. A free-list allocator, however, supports sweeping individual objects by returning them to the free lists.

2.2 Tracing GC

2.2.1 Mechanism and Canonical Implementation

As briefly mentioned in Section 2.1.2, before GC can free unreachable objects, it must first identify reachable objects. The dominant mechanism for identification is tracing.

```

1  let work_list: Stack = roots;
2  while !work_list.is_empty() {
3      let obj = work_list.pop();
4      if obj.marked { continue; }
5      obj.mark();
6      let type_info = *obj.type_info_ptr;
7      for child_offset in type_info.offsets() {
8          let child = *(obj + child_offset);
9          if (child != NULL) work_list.push(child);
10 } }
```

Figure 2.1: A GC tracing algorithm that enqueues object references before marking, and then scans the referenced objects.

For tracing-based GCs, tracing can account for over 90 % of GC time [Zhao et al., 2025a].

Tracing traverses the heap via pointer chasing and identifies reachable objects iteratively (Fig. 2.1). A canonical implementation maintains the traversal wavefront in a *work stack/queue* of object references. The runtime initially populates this queue with a set of *program roots*. When invoking GC, the programming language runtime (such as OpenJDK) provides the roots, which include all runtime references into the heap, such as global variables and local variables held on the program stack.

The trace proceeds by taking a reference off the work queue. If the object has not yet been visited, the GC marks it (line 4) by setting a bit either in its header or in a side bitmap. After marking an unvisited object, the GC looks up the object’s type information (line 6), uses it to scan for references to child objects (line 8), and inserts these children into the work queue (line 9). The trace continues iteratively, taking references from the work queue and skipping already marked ones, until the queue is empty.

The need for a full heap trace can sometimes be reduced at an algorithmic level via generational GCs [Ungar, 1984] or region-based GCs [Detlefs et al., 2004; Blackburn and McKinley, 2008]. These algorithmic improvements are not a focus of this thesis.

Though tracing is conceptually simple, analyzing and optimizing its performance is difficult. There are three challenges: i) measuring the costs of tracing, including concurrent tracing, ii) identifying and fixing scalability bottlenecks of parallel tracing on modern multicore CPUs, and iii) quantifying the potential benefits of hardware acceleration, and devising an approach to speed up tracing. Addressing these challenges forms the core of my thesis, and I explain each of these challenges in more detail below.

2.2.2 Parallel Tracing and Scalability

With the wide availability of highly parallel modern processors,¹ parallelizing tracing to increase GC throughput is increasingly relevant. The challenges of parallelizing GC are twofold: the intrinsic amenability of different workloads to parallelization, and the engineering of the GC.

Barabash and Petrank [2010] studied the theoretical scalability of tracing, and showed that the shape of the object graph is the ultimate limiting factor when parallelizing tracing. For example, when traversing a linked list, each list node must be visited sequentially, preventing parallelism. This pathology can manifest in real-world workloads, as I show in a case study (Section 4.7).

Apart from theoretical studies, there is a vast body of engineering effort on efficient work distribution among GC workers to maximize utilization and minimize synchronization overheads. These techniques include work-stealing queues in HotSpot GCs [OpenJDK Developers, 2023] and, more recently, a work packet system [Zhao et al., 2025a].

Analyzing and debugging the performance of these systems, however, is difficult. In Chapter 4, I show that traditional approaches—relying on *sampling* and *logging*—are insufficient. In contrast, the kernel-based performance tracing tools I build enable agile interrogation of running binaries, and inspection of important aspects of parallel GC, such as load balancing. These tools not only uncover missed optimization opportunities in the collector, but also reveal bugs in application programs.

2.2.3 Barriers

GC barriers are mechanisms that mediate the mutators’ access to the heap, such as reference writes or reads, to facilitate GC. Just like allocation fast paths (Section 2.1.1), barriers are often compiled and optimized alongside the application code for performance reasons. Measuring barrier costs requires careful methodology, and has been the subject of extensive studies in the past [Zorn, 1990; Hosking et al., 1992; Blackburn and Hosking, 2004; Yang et al., 2012]. However, modern low-pause collectors that use read barriers extensively have been less studied, as I show in Chapter 3.

Barriers have many use cases. For tracing GC, barriers can be used for purposes such as generational GC and concurrent tracing. For generational GC, write barriers can be used to track references from one part of the heap to another, thus enabling traversal of a subset of the heap in isolation [Ungar, 1984]. I now discuss concurrent tracing in more detail.

2.2.4 Concurrent Tracing

GCs are increasingly used in workloads that demand responsiveness, such as transaction processing, or gaming. On the surface, one can minimize GC’s impact on application responsiveness by minimizing GC *pauses* and shifting GC activities to run

¹At the time of writing, the largest are the latest AMD EPYC 9965 server chips, with 192 cores and 384 hardware contexts per socket.

concurrently with mutators. In [Chapter 3](#), I examine the performance pitfalls of concurrent GCs.

For now, I discuss the race conditions introduced by this GC concurrency with mutators. An example of such a race is when mutators attach objects not yet discovered by collectors to objects already processed by collectors. This could result in a reachable object being reclaimed at the end of a GC epoch. To address this race, concurrent tracing GCs employ *write barriers* [[Steele Jr., 1975](#); [Dijkstra et al., 1978](#); [Yuasa, 1990](#)].

In this thesis, I am not concerned with the different designs of concurrent GCs and their respective barrier costs. Instead, in [Chapter 3](#), I show how to measure the real cost of GCs, which includes the barrier costs, without explicitly separating them. I also discuss the opportunity cost of concurrent GCs, and how GC pause time is used as a proxy metric for application responsiveness at one’s peril.

2.2.5 Tracing on CPUs

Tracing is an identification mechanism based on heap traversal via pointer chasing. As discussed in [Section 2.1.2](#), tracing remains an important backup mechanism even in state-of-the-art reference-counting GCs. Therefore, this mechanism has been heavily tuned in software GCs for decades, including parallelization ([Section 2.2.2](#)) and concurrency with mutators ([Section 2.2.4](#)). There are also attempts at microarchitectural optimizations for tracing, such as improving its memory-level parallelism using software prefetching [[Boehm, 2000](#); [Garner et al., 2007](#); [Wu et al., 2013](#)].

Despite these efforts, I show in [Chapter 3](#) and [Chapter 4](#) that the costs of tracing remain substantial. This is because tracing is inherently a poor fit for modern CPUs:

1. Pointer chasing exhibits poor locality [[Cao et al., 2012](#)].²
2. Tracing performance is often limited by the number of outstanding loads, which is, in turn, limited by the reorder buffer size [[Beamer et al., 2015](#)]. As a result, tracing benefits little from having multiple hardware contexts per core (simultaneous multithreading) [[Cao et al., 2012](#); [Beamer et al., 2015](#)].
3. Tracing leaves the area- and power-hungry functional units of modern CPU cores underutilized since tracing is dominated by simple arithmetic and memory operations [[Maas et al., 2018](#)].

In [Chapter 5](#), I discuss the strengths and weaknesses of different hardware acceleration schemes for tracing. By doing so, I motivate the development of on-DIMM GC accelerators, and devise a baseline on-DIMM GC design. I then improve upon this baseline design using the insights gathered via the cross-stack techniques I present throughout this thesis.

²Recent work by [Huang \[2025\]](#) suggests that GC locality is more nuanced.

2.3 Summary

In this chapter, I gave an overview of GC, which comprises three key mechanisms: allocation, identification, and reclamation. In particular, I focused on a key identification mechanism—tracing. Despite significant engineering efforts in the past decades, there are still significant gaps in how we build high-performance GCs in each of these aspects. These gaps include:

1. Increasingly parallel and concurrent GCs induce costs that are hard to measure.
2. Scalable and responsive GCs are increasingly demanded, yet our methodology and engineering practices fail to keep up.
3. Software GC performance, especially regarding transitive closures, is bounded by the microarchitectural bottlenecks of CPUs.

In the remainder of this thesis, I examine each of these gaps in more detail, and propose cross-stack improvements, spanning software, kernel, and hardware. In the next chapter, I start with the surprising lack of clarity regarding the true cost of GCs, and present a novel methodology for empirically estimating the cost of GC.

Distilling the Real Costs of Garbage Collectors

Before embarking on building a hardware garbage collection (GC) accelerator, one must ask what the potential gain would be. This simple question turns out to be difficult to answer. Despite the long history of GC and its prevalence in modern programming languages, there is surprisingly little clarity about its true cost.

In this chapter, I present the *lower bound overhead (LBO)*¹ methodology, which transforms how we quantify the hard-to-measure overheads of critical functionality, such as GC. Attribution of overheads can be very complex due to factors such as optimizing compilers, and microarchitectural side effects. The LBO methodology estimates overheads by approximating an ideal baseline. Applied to GC, LBO reveals, for the first time, the substantial costs of software GC—OpenJDK’s heavily-tuned default production G1 collector [Detlefs et al., 2004] incurs 16 % overhead even in the most favorable settings. The results reaffirm that there is a huge potential for specialized architectural support for GC, and they also highlight the danger of allowing our field’s methodological foundations to lag innovation.

I start this chapter by discussing how even modern, well-established methodologies fail to quantify the real cost of GCs (Section 3.2). I then use a series of examples to illustrate the LBO methodology’s core principle, which is to estimate an ideal baseline by subtracting explicitly identifiable costs (Section 3.3). When applying the LBO methodology to state-of-the-art language runtimes, I find that existing software GCs incur substantial overheads (Section 3.5), and I highlight common pitfalls that lead to the misinterpretation of evaluation results. Drawing on these observations, I make recommendations on GC evaluation (Section 3.6). The LBO methodology has improved GC research since its introduction, and it has potential applicability to other systems, such as malloc/free-based explicit memory management.

This chapter incorporates material from two papers I have coauthored [Cai et al., 2022; Blackburn et al., 2025]. Their respective artifacts for reproducing the results can be found on Zenodo [Cai, 2022; Blackburn et al., 2024].

¹Pronounced *elbow*.

3.1 Two Key Problems in Understanding Garbage Collection Costs

Garbage collection (GC) is ubiquitous in software systems, and the research community has extensively studied GC performance. Prior work i) characterizes specific elements of GC behavior, ii) performs comparative evaluations of different GC algorithms, and iii) deconstructs the performance of specific GCs [Blackburn et al., 2008, 2004a; Zorn, 1990; Hosking et al., 1992; Blackburn and Hosking, 2004; Yang et al., 2012; Huang et al., 2004; Zorn, 1993; Blackburn et al., 2006; Zhao and Blackburn, 2020]. These are non-goals of this chapter.

While this rich literature helps us understand how GCs compare, how they are designed, and what the key sources of cost are, there is a surprising lack of clarity regarding the *real* costs that GC brings to a programming language. In this chapter, I focus on two key problems: i) lack of clarity about the cost of GC, and ii) misinterpretations of GC evaluations due to limited cost metrics. I now further motivate each of these two problems and outline how this chapter addresses them.

3.1.1 Quantifying Real GC Costs through Distillation

The first problem is that knowing the *total* GC cost is a prerequisite for making hard-to-reverse design decisions and resource allocation, yet it has eluded the research community to date. Hertz and Berger [2005] attempted to quantify the cost of garbage collection compared to an explicit memory management regime, but their approach has several limitations (more on this in Section 3.2.1).

For language users, knowing the cost of GC can help them decide whether to use a managed language or alternatives such as C/C++, and Rust. This is a decision that often cannot be easily reversed.

For programming language implementers and hardware architects, understanding the cost of GC helps them put that cost in the context of the rest of the language runtimes, or computer systems. This in turn helps them decide where to spend research and engineering resources, such as whether to build a near-memory GC accelerator (Chapter 5).

In this chapter, I present a language- and runtime-agnostic methodology to empirically place a lower bound on the cost of low-level systems that are otherwise hard to measure. Although I use GC throughout this chapter, the methodology is applicable to other systems as well (Section 3.7).

The intuition behind LBO is simple, as I illustrate in Fig. 3.1. If we knew the ideal cost of running an application without any of the costs of a garbage collector, then we could use that as a baseline to quantify the cost imposed by any garbage collector. However, in practice, it is difficult to obtain this baseline, because some of the GC costs are hard to tease out.

The key insight is that we can estimate the ideal application cost by subtracting explicitly identifiable GC costs from the total execution cost. Using the analogy of distilling a liquid mixture, we call this estimated application cost *the distilled cost*. Since

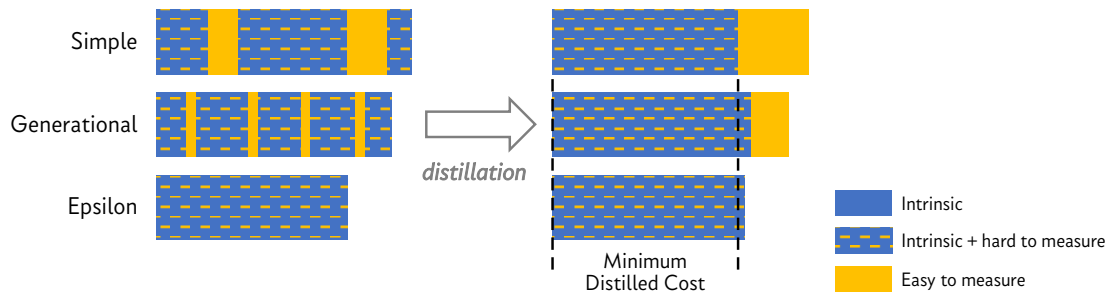


Figure 3.1: Application execution comprises the ideal application cost shown in blue, and GC costs shown in yellow. Some of the GC costs, shown with yellow dashed lines, are tightly coupled with the execution and are hard to measure. The distillation process subtracts the easy-to-measure GC costs from the total cost, and what remains approximates the ideal application cost. The minimum distilled cost (black dashed lines) can be used to estimate the real cost of each GC.

the distilled cost is a conservative overestimation, it allows us to put a *lower bound* on the cost of GC—hence the name Lower Bound Overhead (LBO).

The LBO methodology is simple yet effective: even a conservative underestimation of the GC cost turns out to be substantial across a diverse set of workloads. When applied to OpenJDK 17 and 21—industrial-strength, high-performance Java runtimes, LBO estimates that GC imposes *at least* 30–243 % wall-clock time and 23–123 % CPU cycle overheads relative to the ideal application cost across benchmarks using a modest heap size (twice the minimum required heap size). In addition to these two metrics, LBO is applicable across a wide range of cost metrics, including cache misses, and energy consumption.

3.1.2 Misinterpretation of Evaluation Results

The second problem is that insufficient cost metrics used to measure GC lead to misinterpreting GC evaluations even when modern, well-established methodologies (e.g., varying the heap size) are used. Blackburn et al. [2016] warn that “such misinterpretation risks misdirecting our field, abandoning promising ideas while pursuing ideas that we should have abandoned.”

This risk is acutely clear when we examine the new low-pause collectors. These algorithms are popularly understood to make applications more responsive with little cost to throughput [Flood et al., 2016; Lidén and Karlsson, 2018]. However, the case studies in this chapter show this belief to be unfounded; although these collectors deliver low *pause times*, in some settings, their application overheads are so great that *responsiveness* is often no better than that of simple stop-the-world GCs (Section 3.5.3).

In this chapter, I analyze three important types of misinterpretation of GC evaluations caused by using limited metrics. I show how key performance trade-offs made by different GCs can go unnoticed. I recommend mitigating this problem by using more cost and performance metrics, including wall-clock time, CPU cycles, and application latency.

3.2 Garbage Collection Cost is Hard to Measure

This section discusses a few attempts to measure GC costs, and points out their limitations. I then highlight two categories of GC costs that are hard to measure: costs tightly coupled with application execution, and indirect costs. I describe the established methodologies to measure such costs, but they are insufficient to adequately characterize the total GC cost on their own. Finally, I give an overview of the production garbage collectors studied in this chapter, and how they present such hard-to-measure sources of GC cost.

3.2.1 Related Work

Hertz and Berger [2005] attempted to quantify the cost of garbage collection compared to an explicit memory management regime. Their work shares the same basic intuition as the LBO methodology (comparing against a hypothetical “zero-cost” baseline), but is an entirely different approach.

By tracing the program execution, they construct a database of object liveness information. This database allows them to create an “oracular GC” that inserts `malloc` and `free` calls into the program execution trace, which is processed by an architectural simulator—Dynamic SimpleScalar [Burger and Austin, 1997; Huang et al., 2003].

Although it is an interesting study, it is limited by the fidelity of the simulation infrastructure (a problem whenever using architectural simulation) and by the requirement of being able to make such changes to the runtime and the appropriate modeling within the simulator. The particular analysis has other limitations, including the fact that it is now dated by changes in architecture, workloads, and algorithms.

In contrast, the LBO methodology is very simple. It measures any production collector running on native hardware, making minimal assumptions about the GC implementation, and requiring no access to source code, entirely avoiding the issues in prior work [Hertz and Berger, 2005].

Some work (e.g., [Zorn, 1993]) measures the cost of conservative GC in the context of explicit memory management. In this chapter, I focus on precise GC in managed languages.

Numerous studies (e.g., [Blackburn et al., 2004a]) measure GC pauses to quantify GC costs. As I discuss further in Section 3.5.3, this naïve approach is problematic for multiple reasons.

1. Some GC activities (such as barriers) are carried out by mutators on behalf of the GC, and are hard to separate.
2. The majority of the work in concurrent GCs is shifted to run concurrently with the mutator threads outside GC pauses.
3. With respect to user-experienced latency, several short pauses may have a similar or worse effect than a long pause [Cheng and Blelloch, 2001].

3.2.2 Tightly-Coupled Costs

As discussed in [Section 2.1](#), a GC can be considered in terms of three principal activities: *allocation*, *identification*, and *reclamation* [[Blackburn and McKinley, 2008](#)]. Allocation is performed by the mutator using space provided by the GC. Identification establishes which part of the heap is live and which may be reclaimed. Reclamation makes space occupied by unreachable objects available for reuse.

While some GC activities are performed directly by the collector, others are performed by the mutator. Below are some examples.

1. Mutators can allocate on their respective local allocation buffers, and only require periodic synchronization with the GC ([Section 2.1.1](#)).
2. Mutators can assist identification and reclamation via *read and write barriers*—code snippets that mediate mutators’ heap operations on behalf of the GC ([Section 2.2.3](#)).
3. Mutators can reclaim unreachable objects. For example, in the case of naïve reference counting (RC), the code that decrements the reference count of an object to zero can immediately reclaim the memory ([Section 2.1.3](#)).

For performance reasons, the above mechanisms, such as allocation sequences and barriers, are tightly coupled with mutator execution, often implemented using a “fast-and-slow-path” paradigm ([Section 2.1.1](#)). Fast paths are short but frequently executed code snippets, often inlined into the mutator code by the just-in-time (JIT) compiler. Because they are so tightly integrated into the mutator, measuring their costs requires carefully crafted methodologies that minimize the observer effect due to instrumentation overheads.

1. [Blackburn et al. \[2004a\]](#) placed an upper bound on the cost of a bump-pointer allocator by compiling the allocation fast path out-of-line. Then, that upper bound was used to derive the cost of a free-list allocator using their relative performance.
2. The cost of barriers has also been extensively studied [[Zorn, 1990](#); [Hosking et al., 1992](#); [Blackburn and Hosking, 2004](#); [Yang et al., 2012](#)]. These methodologies remove the requirement of barriers for correctness, e.g., through a full-heap trace for a generational GC, and then measure the execution with and without barriers.
3. [Blackburn et al. \[2004a\]](#) measured the cost of a deferred reference counting collector in terms of the mutator time.

These specialized methodologies require a deep understanding of the GC implementation and modifications to the collector and/or runtime, and consequently are not trivially applied to arbitrary language runtimes. Furthermore, they are insufficient to adequately characterize the total GC cost on their own. In this chapter, I focus on devising a methodology that encompasses all of these costs instead of measuring individual components.

3.2.3 Indirect Costs and Benefits

Apart from GC's direct costs, it can also bring indirect costs and even benefits. A common source of indirect cost (or benefit) is reduced (increased) mutator locality.

GC impacts locality by sharing caches with the mutator. In the case of concurrent GC, GC threads contend with mutator threads when they share caches. In the case of stop-the-world pauses, memory touched by GC code pollutes the cache, leaving mutator threads resuming with a cold cache.

However, GC can improve mutator locality by rearranging objects. A compacting or evacuating GC can improve spatial locality by moving objects that are frequently accessed together to be spatially closer to each other [Huang et al., 2004].

Some of the locality impact might be observable through hardware performance counters, such as cache miss events (e.g., LLC and TLB misses). However, in general, it is hard to tease out the locality impact of GC running on real hardware, because that would require achieving the same GC effects (like compacting the heap) without affecting cache state in the process. It may be possible to achieve such an effect under simulation, but it would likely require significant compute resources to evaluate and would suffer from the same limitations as prior work [Hertz and Berger, 2005].

In this chapter, I am not concerned with teasing out the indirect impact of GC. The LBO methodology measures the total cost of GC, including its indirect cost.

3.2.4 Aspects of GC Costs in OpenJDK

Section 3.5 measures the cost of production GCs in OpenJDK as a case study, and reveal common misinterpretations in their evaluations. It uses the most recent two long-term stable (LTS) releases of OpenJDK, 17 and 21. I highlight the key properties of each of the collectors studied in Table 3.1. OpenJDK 17 and 21 share the same set of collectors, except Generational ZGC, which is exclusive to OpenJDK 21. However, Generational ZGC is excluded in this study, since it cannot run to completion on one or more benchmarks for any of the heap sizes less than $6 \times$.

Note that analyzing the performance of a particular collector and proposing improvements to its design are explicitly not objectives of this chapter. There exists a substantial literature addressing those, including a recent paper by Zhao and Blackburn [2020] that gives a detailed performance analysis of a number of collectors, which include ones studied in this chapter.

The first of these collectors, Epsilon, does not actually collect garbage. Just like any other collector, its real cost—which turns out to be non-zero—can be measured by the LBO methodology. The other five collectors can be divided into three groups:

1. Stop-the-world collectors (Serial and Parallel): the collector requires all mutators to be stopped while it is running; i.e., it does not exhibit any concurrency with respect to the mutator.

²Targeted for release in OpenJDK 25, see <https://openjdk.org/jeps/521>.

³Brooks prior to OpenJDK 13, Baker after OpenJDK 14.

⁴The generational version is not studied.

Table 3.1: Five garbage collectors plus Epsilon (no GC) in OpenJDK that this chapter studies: Serial, Parallel, G1 [Detlefs et al., 2004], Shenandoah [Flood et al., 2016], and ZGC [Lidén and Karlsson, 2018]. SATB = snapshot at the beginning [Yuasa, 1990]. LVB = loaded value barrier [Tene et al., 2011].

Collector	Year	Generational	Parallel GC	Concurrent GC	Barriers
Epsilon	2018	No (no GC)	No (no GC)	No (no GC)	No (no GC)
Serial	1998	Yes	No	No	Write (card-marking)
Parallel	2005	Yes	Yes	No	Write (card-marking)
G1	2009	Yes	Yes	Yes (tracing)	Write (card-marking and SATB)
Shenandoah	2019	No ²	Yes	Yes (tracing and copying)	Write (SATB) and read (LVB ³)
ZGC	2018	Optional ⁴	Yes	Yes (tracing and copying)	Read (LVB)

2. Concurrent tracing collector (G1): the collector performs garbage identification concurrently via a trace. This trace does not modify the heap, but marks reachable objects as live. The correctness of the concurrent trace is typically ensured by write barriers.
3. Concurrent copying collectors (Shenandoah and ZGC): in addition to concurrent tracing, the collector performs reclamation concurrently by copying objects. This involves modifying the heap and ensuring that the concurrently executing mutator maintains a coherent view of the heap even when objects it references are moved. The correctness of concurrent copying is typically protected by read barriers.

All of the OpenJDK collectors (except Epsilon) use barriers. Apart from allowing for concurrency with respect to the mutators, barriers can also allow a part of the heap (such as a generation) to be independently collected. The stated design goal of the two concurrent copying collectors is to improve the responsiveness for latency-sensitive applications by reducing the GC pause times (the extent to which they achieve this goal is discussed in [Section 3.5.3](#)). Therefore, I also refer to them as the *low-pause collectors* in the rest of the chapter.

Note that all of the collectors except ZGC use compressed pointers by default. In general, one should take care when comparing ZGC with other collectors. In [Section 3.5](#), we evaluate all collectors using their default configurations so as not to overlook collectors' space overheads due to their algorithmic limitations.

These collectors all include hard-to-measure sources of GC cost discussed previously in this section, hindering the quantification of their real costs. For example, the allocation sequence and the barriers are tightly integrated with the application by inline compilation. Moreover, when collector threads run concurrently with mutators, it is hard to attribute overheads from resource contention to the originating threads.

These collectors (including Epsilon) therefore motivate the LBO methodology. We will now see through some concrete examples how LBO measures and places a lower bound on implicit and explicit GC costs.

3.3 The Lower Bound Overhead Methodology

Due to the prevalence of GC, its cost is a hot—and sometimes contentious—topic. The extensive existing literature focuses on *comparative evaluation* (e.g., [Table 3.2](#)), which helps the community understand how GCs compare. In this section, I address a distinctly different goal: measuring the total overhead that GCs impose on a runtime. First, I describe the lower bound overhead (LBO) methodology using a concrete example. Then, I discuss its applicability, advantages, and limitations.

3.3.1 Definition with Examples

[Section 3.1](#) sketched the intuitions underpinning the LBO methodology using the distillation analogy ([Fig. 3.1](#)). To deepen the understanding, I now use concrete evaluation results from three production GCs in OpenJDK 17—Serial, Parallel, and Shenandoah—running a real-world application, the h2 database. This also serves as a running example throughout this section.

Note that in the examples, I use one particular cost metric—total CPU cycles used by all JVM threads—and one particular way of excluding explicitly identifiable GC cost—costs incurred during stop-the-world (STW) pauses. I emphasize that these choices are orthogonal to the mechanism of the LBO methodology. The LBO methodology is broadly applicable to complex systems where costs are hard to attribute, regardless of the methods used to exclude explicitly identifiable costs, and the concrete cost metrics.

It is easy to measure the total costs of program executions (inclusive of GC costs), and see how they compare. [Table 3.2](#) shows the total CPU cycle costs running the h2 benchmark at a fixed $3 \times$ heap size, varying the collectors used. The Serial collector has the smallest total cycle usage, and Parallel and Shenandoah are 0.2 % and 102.3 % more expensive, respectively. This *comparative analysis* is a well-understood methodology in the literature, and shows which GC is the best to use to minimize total CPU cycle consumption for a given heap size. However, it does not measure the real CPU cycle cost of each *collector alone*.

Definition 3.3.1 (Ideal application cost). The ideal application cost is the *theoretical* cost of running an application. This ideal cost includes the best GC benefit (such as improved locality) any GC configuration could bring but none of the GC costs (such as barriers).

For example, if one knew the ideal cost of the application h2 (inclusive of the best GC benefit but exclusive of the GC costs) in terms of CPU cycles, one could subtract that from the total cycles used by each collector, yielding the real cost of the respective collector.

Table 3.2: The total CPU cycles consumed when running the DaCapo benchmark h2 with a $3 \times$ heap (see Section 3.4) using three different collectors. Lower is better. The cycles are also normalized to the best collector (Serial) shown in green.

Collector	Total	Normalized
	billion cycles	
Parallel	108.33	1.002
Serial	108.12	1.000
Shenandoah	218.72	2.023

Of course, the ideal application cost of a given workload is unknown. The key insight is that the ideal application cost is approximately the total costs minus the costs easily attributed to GC.

This insight can perhaps be better understood using the distillation analogy. The total cost is a mixture of the ideal application cost (after considering the best GC benefit) and the GC cost. Some of the GC costs are easy to attribute (such as the cost incurred during STW pauses), while others are hard (such as barriers). It is difficult to remove all hard-to-attribute GC costs from the mixture, but it is easy to distill out explicitly attributable GC costs.

Definition 3.3.2 (Distilled application cost).

$$\text{Distilled cost}_g \equiv \text{Total cost}_g - \text{Attributable GC cost}_g \geq \text{Ideal cost}$$

That is, a distilled cost (measured using a particular GC configuration g) overestimates the ideal application cost.

Definition 3.3.3 (Minimum distilled application cost).

$$\left(\text{MDC} \equiv \min_{g \in \text{GCs}} \text{Distilled cost}_g \right) \geq \text{Ideal cost}$$

The *minimum distilled cost* (MDC) is the minimum of the distilled application costs from running the application with different GC configurations (different g). These configurations can be different collector algorithms, or the same algorithm with *different* tuning parameters (e.g., heap size, the number of collector threads). The MDC is the best overestimate of the ideal cost.

As shown in Table 3.3, the distilled application cost for each GC is the total cost minus the easily attributable cost (such as costs incurred during STW pauses). The set of three collectors used generates three distilled costs, and the minimum of which (MDC) is 103.87 billion cycles (Parallel, shown in green).

The MDC in turn allows one to place an empirical lower bound on the cost of the collectors, which I call the lower bound overhead (LBO). Recall that for each GC configuration g , $\text{GC cost} \equiv \text{Total cost} - \text{Ideal cost}$, and that $\text{MDC} \geq \text{Ideal cost}$.

Table 3.3: Distilling out cycles used during stop-the-world (STW) pauses from the total cycles in Table 3.2. The minimum distilled application cost (MDC) in cycles is shown in green. The MDC value is used to calculate the empirical lower bound on the cost of each of the collectors in Table 3.4.

Collector	Total	STW	Distilled
billion cycles			
Parallel	108.33	4.46	108.33 - 4.46 = 103.87
Serial	108.12	2.75	108.12 - 2.75 = 105.37
Shenandoah	218.72	0.03	218.72 - 0.03 = 218.69

Definition 3.3.4 (Lower Bound Overhead of Garbage Collectors).

$$(\text{LBO}_g \equiv \text{Total cost}_g - \text{MDC}) \leq \text{GC cost}_g$$

It is easy to see that the quality of LBO estimation hinges on how close the MDC is to the ideal application cost. I later discuss how one might obtain better MDCs (Section 3.3.2).

To compare the (estimated) *real* cost of each collector configuration, one can normalize the LBO to the MDC. I use the normalized LBO in the rest of this chapter.

Definition 3.3.5 (Normalized LBO).

$$\text{Normalized LBO (NLBO)} \equiv (\text{LBO} + \text{MDC}) / \text{MDC}$$

Table 3.4: The empirical lower bound (LBO) on the cycle cost for each collector can be obtained by subtracting the MDC from the respective total cycles in Table 3.3. The lowest LBO in green.

Collector	LBO	Normalized LBO
	Total - MDC	(LBO + MDC) / MDC
Parallel	108.33 - 103.87 = 4.46	1.043
Serial	108.12 - 103.87 = 4.25	1.041
Shenandoah	218.72 - 103.87 = 114.85	2.106

As shown in Table 3.4, a lower bound on the cycle cost of each GC is the total cycles of each of the GCs minus the MDC. Compared with the ideal cost of running the h2 database on OpenJDK, Serial, Parallel, and Shenandoah incur *at least* 4.1 %, 4.3 %, and 110.6 % more CPU cycles, respectively. Note that even though Shenandoah is not useful when calculating the MDC, the cost of Shenandoah is still accurately captured.

First, I emphasize the LBO methodology does not rely on a no-GC scheme. In the above example, all three collectors perform GC activities, and one can calculate their respective LBOs without using a no-GC scheme.

Second, I emphasize that the distilled cost is not the same as the cost of a no-GC scheme (such as Epsilon in OpenJDK). In fact, as I will show later in [Section 3.5](#), the LBO methodology can measure the non-zero cost of Epsilon.

3.3.2 Discussion

The LBO methodology allows us to place a lower bound on the cost of low-level systems, such as GC. Implementing LBO is easy; the only requirement of the methodology is to exclude the easily attributable costs of the system being measured. In the case of GC, language runtimes often provide callbacks that allow us to delineate GC pauses, allowing the attribution of GC costs during STW pauses [[Oracle, 2021b](#); [Stephens, 2021](#)].

The LBO methodology is metric- and language-agnostic. In addition to wall-clock time, and CPU cycles, one can use a wide set of cost metrics, such as hardware performance counters. These metrics can include cache misses, and energy metrics (such as Intel’s Running Average Power Limit (RAPL) energy readings). The methodology can be used on different programming language runtimes, and GC algorithms.

The quality of the LBO result depends on how close the minimum distilled cost is to the ideal cost. To illustrate this, I will use GC as an example.

Semispace is a canonical copying GC algorithm that is not practical to use in high-performance production systems, because it uses half the heap space as the copy reserve, and evacuates all live objects during each GC invocation [[Fenichel and Yochelson, 1969](#); [Cheney, 1970](#)]. Despite being impractical, semispace is useful for establishing a good baseline for LBO analysis.

Since it defragments the heap during each GC through evacuation, the mutators can have better spatial locality (more GC benefits). Since it does not require barriers (being a non-generational, stop-the-world collector), and has a very simple allocation sequence, it also incurs little hard-to-attribute costs. Therefore, adding semispace (especially when running at a larger heap size, accounting for the copy reserve) to the mix of configurations to measure can improve the MDC.

Conversely, if only concurrent copying collectors are measured, the LBO methodology can still be used, but there is little cost easily attributable to GC, since these collectors have little stop-the-world components. In this scenario, the minimum distilled cost would be much higher than the ideal cost, and would lead to a very poor empirical lower bound on GC cost. One would need to use more complex attribution techniques, such as measuring the costs on a per-thread basis, to improve the estimation. But even that would still not consider the substantial cost of read barriers.

The above example illustrates that to obtain good LBO results, one often needs to measure a diverse set of configurations exhibiting different tradeoffs, and use as many approaches as possible to attribute costs.

3.4 Evaluation Methodology

In the previous section, I presented the lower bound overhead (LBO) methodology that empirically measures the real costs of garbage collection (GC). In this section, I detail the evaluation and execution methodology used in the case studies that apply the LBO methodology to measure the real costs of OpenJDK GCs (Section 3.5).

The case studies include results for OpenJDK 17 conducted by Cai et al. [2022], and results for OpenJDK 21 conducted by Blackburn et al. [2025]. In this chapter, I focus on case studies using OpenJDK 17, and use OpenJDK 21 to validate the general performance trends in the latest long-term stable (LTS) OpenJDK release. For each set of experiments, we followed the contemporary best practices, and I detail the setup differences between experiments in the following sections where applicable.

3.4.1 Benchmarks and Latency Measures

When evaluating OpenJDK 17, we use a pre-release version of the Chopin release of the DaCapo benchmark suite [DaCapo Developers, 2021]. We exclude the benchmarks cassandra, h2o, and kafka, because they use deprecated Java features that are not compatible with JDK 17. When evaluating OpenJDK 21, we use the stable 23.11-chopin-MR2 release of the DaCapo benchmark suite [Blackburn et al., 2024, 2025], and exercise all 22 benchmarks.

DaCapo’s Chopin release includes a number of latency-sensitive benchmarks. For these latency-sensitive benchmarks, DaCapo reports two measures of latency, *simple* and *metered*, which I now discuss in more detail.

Most real-world request-based services implement a queuing system. Requests enter the queuing system at some externally determined rate, dictated by factors such as when customers make purchases or when people launch queries. A key DaCapo design principle is that each benchmark runs in a single JVM on a single machine. Consequently, attempting to implement a realistic distributed load-balancing system was out of scope for DaCapo Chopin. Sacrificing some realism for determinism, the DaCapo request-based workloads are driven by a pre-determined set of requests, with each worker consuming consecutive requests until all have been completed. Within each thread, the start time of each request is thus dictated by the completion of the request before.

In a real system, request/event start times are *externally defined*, so a delay will affect not only all running events, but also all subsequent events that are forced to wait in the queue due to the backlog of work. Without a queue, DaCapo’s workloads cannot directly model the cascading effect of delays.

Instead, DaCapo models a similar effect with what it calls *metered latency*. At the completion of the workload, each event is assigned an assumed start time based on all events having been hypothetically received at *uniform* intervals throughout the execution of the benchmark. The metered latency for each event is then defined as the time between its end time and the earlier of its actual and assumed start times. Thus, when the application is paused, the effect of the pause is not felt just by those

events that were running when the pause occurred, but also by those whose end time was delayed.

The uniform synthetic start times are implemented by applying a smoothing function to the actual start times using a sliding average. A window size of one affords no smoothing, so it is identical to simple latency, reflecting no queuing effect. On the other hand, an arbitrarily large window gives all events uniformly distributed synthetic start times. DaCapo reports metered latency using window sizes from 1 ms up to the length of the benchmark execution, in powers of ten.

In this chapter, when discussing metered latency without a specific window size, I refer to the window size that is the length of the benchmark execution (full smoothing). We also evaluate with a smoothing window of 100 ms, which is a reasonable middle ground, allowing for variation in request completion rate over the whole execution (e.g., due to compiler or file cache warm-up), while exposing effects of disruptions due to garbage collection, etc.

We also investigate the SPECjbb2015 benchmark, which is often used in the literature when evaluating low-pause GCs. However, unlike the DaCapo suite, SPECjbb2015 fixes the workload to a constant amount of time rather than to constant work, preventing one from easily measuring the costs of different GCs executing the same amount of work. Therefore, SPECjbb2015 is not included in our analysis.

3.4.2 Cost Metrics

We implement the LBO methodology as a Java Virtual Machine Tool Interface (JVMTI) agent [Oracle, 2021b]. The agent implements the distilled cost in the methodology by excluding costs incurred in stop-the-world (STW) pauses. The STW points are provided by the JVM through callbacks for the starts and ends of GC pauses.

In addition to wall-clock time, the JVMTI agent can capture a wide range of hardware performance counters, including CPU cycles, instruction counts, cache misses, and Intel RAPL energy measurements. To read these counters, we use the `perf_events` subsystem in the Linux kernel. In this chapter, I focus on two important metrics: wall-clock time and CPU cycles. On machines with frequency scaling disabled, the CPU cycles can also be measured using the software `TASK_CLOCK` event; cycles can be derived by multiplying the nominal CPU frequency by the lapsed `TASK_CLOCK`.

3.4.3 JVM Parameters

In this chapter, I focus on the out-of-the-box performance characteristics of GCs, and *deliberately do not* set any GC-related parameters except for the heap size. GC tuning is often specific to particular (classes of) workloads, whereas my concern is the real cost of each GC when handling a diverse set of workloads. Also, in general, GC tuning is an open-ended problem, which is outside the scope of this chapter.

We report the performance of each GC for different heap sizes, because the performance of GC is sensitive to the heap size [Blackburn et al., 2006]. Except for Epsilon, which does not perform GC, we set the heap size relative to the minimum

Table 3.5: Minimum heap size required to run each benchmark.

Benchmark	Heap size (MB)
avro	7
batik	189
biojava	95
eclipse	411
fop	15
graphchi	255
h2	773
jme	29
jython	25
luindex	42
lusearch	21
pmd	156
sunflow	29
tomcat	21
tradebeans	131
tradesoap	103
xalan	8
zxing	97

heap size (Table 3.5) required to run each benchmark (e.g., $2.0 \times$ means that the heap size is set to be twice as big as the minimum heap required for a particular benchmark). The minimum heap size for each benchmark is measured using G1 because it is the most space-efficient GC among the ones we study.

For the older experiments conducted on OpenJDK 17, we followed contemporary best practices, and additional set `-XX:-TieredCompilation -Xcomp` to eagerly compile methods on first use. These parameters speed up the warmup of the JVM and reduce experimental noise due to JIT compilation. We omit these parameters for tradebeans and tradesoap, because these parameters cause these two benchmarks to crash when running on the version of OpenJDK we use.

For the newer experiments on OpenJDK 21, we do not use the above compilation flags, because we find that across benchmarks, eager compilation leads to worse code generation quality due to a lack of profiling feedback.

3.4.4 Execution Methodology

For each configuration, we invoke each benchmark 10 and 20 times on OpenJDK 21 and 17, respectively. We interleave invocations of different configurations to minimize bias due to systemic interference. In each *invocation*, the benchmark performs five *iterations*, and we report results from the last iteration, with the first four iterations serving to warm up the runtime. For each configuration, we report the mean and the 95 % confidence interval (CI).

3.4.5 Hardware

While the microarchitectural sensitivity of a GC algorithm is important, it is orthogonal to this work. We choose contemporary high-performance desktop CPUs for each set of experiments.

For OpenJDK 17 experiments, we use two identical machines with Intel Core i9-9900K (Coffee Lake) CPUs (8 cores, 16 threads), with 4×32 GB DDR4-3200 memory.

For OpenJDK 21 experiments, we use two identical machines with AMD Ryzen 9 7950X (Zen 4) CPUs (16 cores, 32 threads), with 2×32 GB DDR5-4800 memory.

We disable dynamic frequency scaling to reduce experimental noise.

3.4.6 Software

Each pair of experimental machines runs identical software. All benchmarks are executed on an otherwise idle machine, with as many background daemons and periodic tasks disabled as possible.

For OpenJDK 17 experiments, all machines use Ubuntu 18.04.6 LTS images with 5.4.0-89-generic kernels. The JVM is the Temurin-17.0.1+12 release of OpenJDK.

For OpenJDK 21 experiments, all machines use Ubuntu 22.04.4 LTS images with 6.8.0-40-generic kernels. The JVM is the Temurin 21.0.3+9 release of OpenJDK.

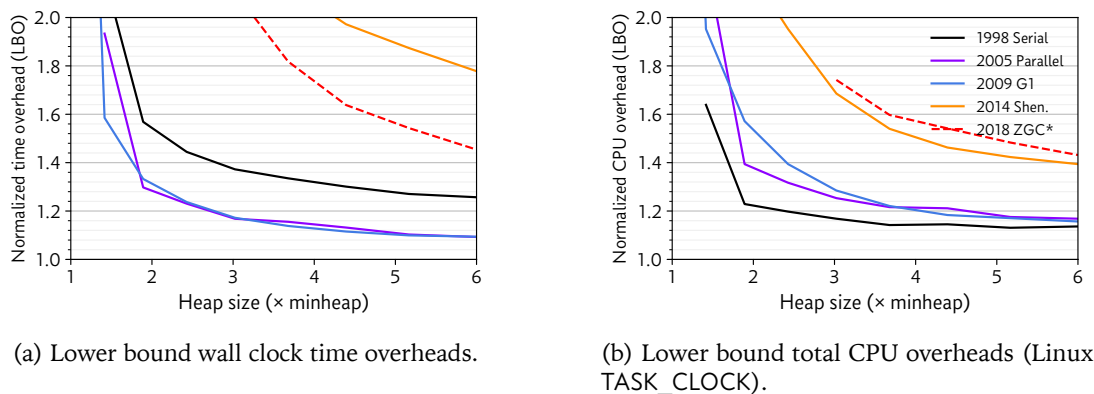


Figure 3.2: Lower bounds on the overheads of five OpenJDK 21 production garbage collectors with their default settings, as a function of heap size, showing the geometric mean of overhead over all 22 DaCapo Chopin benchmarks. I only plot data points where the respective collector can run all 22 benchmarks to completion. In the best case, wall clock overheads are 9 % (G1 and Parallel) and total CPU overheads are 15 % (Serial). At smaller heaps, overheads exceed 2×.

We also measured OpenJDK 11 using the AdoptOpenJDK-11.0.11+9 release, and the overall results were quite similar to those of OpenJDK 17 or 21.

3.5 Case Study: Collectors in OpenJDK

In Section 3.3, I proposed the lower bound overhead (LBO) methodology to estimate the cost of garbage collection (GC) for any given set of metrics. In this section, I apply the LBO methodology and reveal, for the first time, the substantial costs imposed by production GCs (Section 3.2.4).

Figure 3.2 shows the overhead of five OpenJDK 21 GCs as a function of heap size, taking the geometric mean over all 22 DaCapo Chopin benchmarks. The graph shows the time-space tradeoff inherent to GC: resource consumption of GC rises as the available memory shrinks. Figure 3.2(a) shows the wall clock time overhead, while Fig. 3.2(b) shows the total computational overhead by summing the total running time/cycles across all threads in the process.

First, note that the CPU overhead of GC is 15 % in the best case in OpenJDK 21, the latest long-term stable (LTS) release of OpenJDK. Using a modest heap size (100 % heap headroom), applications spend at least 30–243 % more wall-clock time and 23–123 % more CPU cycles relative to their ideal costs. OpenJDK’s heavily tuned default production G1 collector [Detlefs et al., 2004] incurs at least 54 % CPU cycle overhead when given 100 % headroom and 16 % with 500 % headroom. As I will show later in this section using OpenJDK 17, these results are robust across JDK versions and benchmark versions.

Second, note that there is a regression when considering when each collector was introduced into the JVM (Section 3.2.4). As I will show later in this section, the newer low-pause collectors fail to deliver better application latency for the evaluated

workloads despite being significantly slower and cycle-intensive—as high as $17\times$ in small heaps, and 63 % even in the most favorable situations.

If one accepts for a moment that our analysis is sound and that these figures accurately reflect the state of the art, this should give researchers pause. How did this happen? In the remainder of this section, I attempt to answer this question through a series of case studies, and reveal two likely causes: i) lack of clarity about the costs imposed by GC, and ii) misinterpretation of GC evaluations due to limited cost metrics. I highlight three important types of misinterpretation: i) not considering opportunity cost, ii) not considering overheads due to concurrency, and iii) measuring pause time instead of application latency.

In [Section 3.6](#), I will make concrete recommendations on how to mitigate the misinterpretation by using a richer set of cost and performance metrics.

The remaining parts of this section use OpenJDK 17.

3.5.1 Results: Costs of Garbage Collection

First, we estimate the costs of each *configuration* (a collector at a heap size) using the LBO methodology. The distilled cost for each configuration is the total cost excluding the STW cost.

[Table 3.6](#) shows the time LBOs of the collectors we study, while [Table 3.7](#) shows the cycle LBOs. Each table covers eight different heap sizes, ranging from a small $1.4\times$ heap to a generous $6.0\times$ heap (see [Section 3.4](#) for the multiplier notation). The LBO value for each configuration is the geometric mean over 16 DaCapo benchmarks.⁵

The production GCs incur substantial costs. For a modest $2.4\times$ heap, production GCs on average spend *at least* 7–82 % more wall-clock time and 6–92 % more cycles relative to the ideal costs. Even for a generous $6.0\times$ heap, the costs are as much as 23 % in terms of time and 34 % in terms of cycles. For all heap sizes shown, Serial achieves the lowest costs in terms of cycles, while Parallel achieves the lowest costs in terms of time for all but the smallest heap size.

Note that even though Serial is single-threaded, the rest of the VM is still multithreaded, hence the difference between the time and cycle LBOs for Serial. Unsurprisingly, this difference is more pronounced in benchmarks with more mutator parallelism.

3.5.2 Analysis of Results

[Tables 3.8](#) and [3.9](#) take a closer look at the $3.0\times$ heap size. This allows me to observe how different collectors behave when challenged with a diverse, modern workload. I compare the performance trends among and within three groups of GCs (see [Section 3.2.4](#)): i) stop-the-world (STW) vs. concurrent GCs, ii) single-threaded vs. multi-threaded STW GC, and iii) concurrent tracing vs. concurrent copying GC. But first, I

⁵We use 18 DaCapo benchmarks in total ([Section 3.4.1](#)). However, eclipse and xalan are excluded in the geometric mean calculation because too many collectors were not able to run these two benchmarks for small heap sizes.

Table 3.6: Time LBOs averaged over 16 benchmarks. The best value for each heap size is shown in green. Where a collector cannot run all benchmarks at a particular heap size, the entry is left blank. Parallel outperforms other collectors, except at 1.4 ×, where G1 has the lowest cost.

GC	1.4 ×	1.9 ×	2.4 ×	3.0 ×	3.7 ×	4.4 ×	5.2 ×	6.0 ×
Ser.	1.42	1.17	1.14	1.13	1.11	1.10	1.09	1.09
Par.	1.41	1.09	1.07	1.06	1.05	1.04	1.04	1.03
G1	1.24	1.16	1.11	1.09	1.08	1.07	1.07	1.06
Shen.	*	1.94	1.64	1.43	1.37	1.30	1.25	1.23
ZGC	*	*	1.82	1.54	1.39	1.32	1.27	1.23

Table 3.7: Cycle LBOs averaged over 16 benchmarks. The best value for each heap size is shown in green. Where a collector cannot run all benchmarks at a particular heap size, the entry is left blank. Serial consistently outperforms other collectors for all heap sizes.

GC	1.4 ×	1.9 ×	2.4 ×	3.0 ×	3.7 ×	4.4 ×	5.2 ×	6.0 ×
Ser.	1.22	1.08	1.06	1.06	1.04	1.04	1.04	1.04
Par.	1.70	1.15	1.12	1.10	1.07	1.06	1.06	1.05
G1	1.54	1.34	1.17	1.14	1.10	1.09	1.09	1.09
Shen.	*	1.75	1.54	1.47	1.42	1.39	1.36	1.33
ZGC	*	*	1.92	1.68	1.55	1.46	1.40	1.34

discuss Epsilon, which is a no-GC scheme that does no collection.

Table 3.8: Time LBOs at $3 \times$ heap (except Epsilon) using distillation. Lower is better. xalan is excluded from the summary statistics due to ZGC failing to run, and the corresponding row is grayed out. The best results for each benchmark are shown in green (light green for xalan). Each LBO is the mean of 20 invocations, with its 95 % CI shown below in gray. Parallel outperforms other collectors for most benchmarks.

Bench.	Serial	Para.	G1	Shen.	ZGC	Eps.
avroa	1.01 $\pm 0.7\%$	1.03 $\pm 0.5\%$	1.03 $\pm 0.6\%$	1.10 $\pm 0.8\%$	1.07 $\pm 0.7\%$	1.01 $\pm 0.7\%$
batik	1.15 $\pm 0.2\%$	1.09 $\pm 0.4\%$	1.07 $\pm 1.0\%$	1.09 $\pm 0.4\%$	1.08 $\pm 0.3\%$	1.09 $\pm 0.2\%$
biojava	1.01 $\pm 0.1\%$	1.01 $\pm 0.1\%$	1.03 $\pm 0.2\%$	1.22 $\pm 0.2\%$	1.88 $\pm 1.0\%$	1.16 $\pm 0.1\%$
eclipse	1.05 $\pm 0.4\%$	1.01 $\pm 0.1\%$	1.06 $\pm 0.2\%$	1.13 $\pm 0.2\%$	1.14 $\pm 0.1\%$	1.18 $\pm 0.3\%$
fop	1.16 $\pm 0.2\%$	1.13 $\pm 1.6\%$	1.20 $\pm 1.2\%$	1.41 $\pm 0.8\%$	2.11 $\pm 0.6\%$	1.17 $\pm 0.3\%$
graphchi	1.02 $\pm 0.3\%$	1.01 $\pm 0.3\%$	1.02 $\pm 0.4\%$	1.19 $\pm 1.5\%$	1.13 $\pm 0.4\%$	1.05 $\pm 0.7\%$
h2	1.12 $\pm 0.3\%$	1.04 $\pm 0.3\%$	1.09 $\pm 0.6\%$	1.57 $\pm 0.3\%$	1.96 $\pm 0.6\%$	1.48 $\pm 0.2\%$
jme	1.00 $\pm 0.0\%$	1.00 $\pm 0.0\%$	1.00 $\pm 0.0\%$	1.01 $\pm 0.0\%$	1.01 $\pm 0.1\%$	1.01 $\pm 0.0\%$
jython	1.04 $\pm 1.7\%$	1.02 $\pm 0.2\%$	1.06 $\pm 0.2\%$	1.61 $\pm 0.9\%$	2.20 $\pm 0.5\%$	1.23 $\pm 0.3\%$
luindex	1.01 $\pm 0.2\%$	1.01 $\pm 0.3\%$	1.04 $\pm 0.4\%$	1.15 $\pm 0.7\%$	1.10 $\pm 0.3\%$	1.06 $\pm 0.8\%$
lusearch	1.23 $\pm 0.2\%$	1.11 $\pm 0.3\%$	1.13 $\pm 0.4\%$	3.77 $\pm 1.6\%$	2.70 $\pm 0.4\%$	
pmd	1.61 $\pm 0.2\%$	1.12 $\pm 0.2\%$	1.19 $\pm 0.3\%$	1.56 $\pm 0.6\%$	1.78 $\pm 8.7\%$	1.17 $\pm 0.4\%$
sunflow	1.41 $\pm 2.0\%$	1.16 $\pm 2.0\%$	1.16 $\pm 1.2\%$	2.18 $\pm 2.2\%$	2.11 $\pm 0.6\%$	1.00 $\pm 0.4\%$
tomcat	1.16 $\pm 0.2\%$	1.14 $\pm 0.2\%$	1.14 $\pm 0.1\%$	1.31 $\pm 0.2\%$	1.91 $\pm 0.6\%$	1.15 $\pm 0.1\%$
tradebeans	1.12 $\pm 0.6\%$	1.02 $\pm 0.6\%$	1.16 $\pm 0.7\%$	1.42 $\pm 0.8\%$	1.33 $\pm 0.5\%$	1.41 $\pm 0.5\%$
tradesoap	1.09 $\pm 0.4\%$	1.01 $\pm 0.2\%$	1.11 $\pm 0.3\%$	1.34 $\pm 0.2\%$	1.28 $\pm 0.2\%$	1.38 $\pm 0.2\%$
xalan	3.38 $\pm 0.3\%$	3.07 $\pm 0.4\%$	3.50 $\pm 0.3\%$	30.18 $\pm 1.7\%$		1.11 $\pm 0.6\%$
zxing	1.03 $\pm 1.1\%$	1.06 $\pm 1.8\%$	1.03 $\pm 1.3\%$	1.27 $\pm 1.1\%$	1.24 $\pm 1.1\%$	1.06 $\pm 1.3\%$
min	1.00	1.00	1.00	1.01	1.01	
max	1.61	1.16	1.20	3.77	2.70	
mean	1.13	1.06	1.09	1.49	1.59	
geomean	1.12	1.06	1.09	1.41	1.51	

No GC

Epsilon does not actually collect garbage, and is not a practical GC. Its distilled cost—same as its total cost—could potentially be smaller than the distilled costs of other collectors. This, in turn, could lead to a smaller minimum distilled cost (MDC), allowing me to better estimate the ideal cost of a workload.

Unfortunately, Epsilon’s distilled cost does not contribute to the MDC in practice. This is evident in [Tables 3.8](#) and [3.9](#), showing the LBOs of Epsilon alongside those of standard collectors in cases where Epsilon is able to run a benchmark without exhausting the machine’s available memory. The only benchmark where Epsilon contributes to MDC is sunflow, indicated by the 1.000 time LBO and the 1.000 cycle LBO.

Table 3.9: Cycle LBOs at $3 \times$ heap (except Epsilon) using distillation. Lower is better. xalan is excluded from the summary statistics due to ZGC failing to run, and the corresponding row is grayed out. The best results for each benchmark are shown in green (light green for xalan). Each LBO is the mean of 20 invocations, with its 95 % CI shown below in gray. Serial outperforms other collectors for most benchmarks.

Bench.	Serial	Para.	G1	Shen.	ZGC	Eps.
avroa	1.01 $\pm 0.6\%$	1.01 $\pm 0.7\%$	1.05 $\pm 0.7\%$	1.20 $\pm 0.6\%$	1.21 $\pm 0.7\%$	1.02 $\pm 0.4\%$
batik	1.15 $\pm 0.2\%$	1.99 $\pm 2.1\%$	1.57 $\pm 0.8\%$	1.45 $\pm 0.5\%$	1.91 $\pm 3.3\%$	1.09 $\pm 0.2\%$
biojava	1.01 $\pm 0.1\%$	1.01 $\pm 0.1\%$	1.05 $\pm 0.3\%$	1.52 $\pm 0.7\%$	3.96 $\pm 1.0\%$	1.16 $\pm 0.1\%$
eclipse	1.05 $\pm 0.4\%$	1.11 $\pm 0.1\%$	1.32 $\pm 1.2\%$	1.39 $\pm 0.3\%$	1.47 $\pm 0.5\%$	1.20 $\pm 0.1\%$
fop	1.16 $\pm 0.3\%$	1.20 $\pm 1.5\%$	1.46 $\pm 1.9\%$	1.89 $\pm 1.1\%$	2.23 $\pm 1.1\%$	1.18 $\pm 0.3\%$
graphchi	1.01 $\pm 0.3\%$	1.03 $\pm 0.3\%$	1.05 $\pm 0.3\%$	1.23 $\pm 2.2\%$	1.23 $\pm 0.7\%$	1.10 $\pm 0.6\%$
h2	1.05 $\pm 0.1\%$	1.06 $\pm 0.2\%$	1.12 $\pm 0.4\%$	2.13 $\pm 0.3\%$	2.65 $\pm 0.7\%$	1.37 $\pm 0.2\%$
jme	1.07 $\pm 0.3\%$	1.13 $\pm 0.4\%$	1.09 $\pm 0.3\%$	1.52 $\pm 0.5\%$	1.47 $\pm 1.0\%$	1.13 $\pm 0.3\%$
jython	1.04 $\pm 1.7\%$	1.03 $\pm 0.2\%$	1.07 $\pm 0.2\%$	2.04 $\pm 0.7\%$	2.44 $\pm 0.4\%$	1.22 $\pm 0.3\%$
luindex	1.01 $\pm 0.2\%$	1.02 $\pm 0.2\%$	1.07 $\pm 1.3\%$	1.21 $\pm 0.8\%$	1.20 $\pm 0.5\%$	1.06 $\pm 0.9\%$
lusearch	1.05 $\pm 0.3\%$	1.03 $\pm 0.2\%$	1.08 $\pm 0.2\%$	1.21 $\pm 0.3\%$	1.27 $\pm 0.3\%$	
pmd	1.10 $\pm 0.5\%$	1.18 $\pm 0.4\%$	1.29 $\pm 0.6\%$	1.56 $\pm 0.5\%$	1.65 $\pm 10.1\%$	1.27 $\pm 0.2\%$
sunflow	1.12 $\pm 2.4\%$	1.09 $\pm 2.1\%$	1.09 $\pm 1.2\%$	1.24 $\pm 0.8\%$	1.08 $\pm 0.8\%$	1.00 $\pm 0.4\%$
tomcat	1.01 $\pm 0.1\%$	1.01 $\pm 0.1\%$	1.02 $\pm 0.1\%$	1.12 $\pm 0.1\%$	1.12 $\pm 0.1\%$	1.03 $\pm 0.1\%$
tradebeans	1.06 $\pm 0.4\%$	1.05 $\pm 0.3\%$	1.21 $\pm 0.7\%$	1.64 $\pm 0.3\%$	2.17 $\pm 0.5\%$	1.29 $\pm 0.6\%$
tradesoap	1.05 $\pm 0.4\%$	1.03 $\pm 0.4\%$	1.12 $\pm 0.8\%$	1.71 $\pm 0.4\%$	2.15 $\pm 0.4\%$	1.25 $\pm 0.2\%$
xalan	1.21 $\pm 0.4\%$	1.63 $\pm 0.3\%$	1.78 $\pm 0.3\%$	1.74 $\pm 0.4\%$		1.14 $\pm 0.3\%$
zxing	1.02 $\pm 1.1\%$	1.03 $\pm 1.8\%$	1.03 $\pm 1.4\%$	1.28 $\pm 1.2\%$	1.24 $\pm 1.1\%$	1.06 $\pm 1.4\%$
min	1.01	1.01	1.02	1.12	1.08	
max	1.16	1.99	1.57	2.13	3.96	
mean	1.06	1.12	1.16	1.49	1.79	
geomean	1.06	1.10	1.15	1.46	1.67	

This observation might be counterintuitive since Epsilon does not incur any explicit costs. However, as I explained in [Section 3.2.2](#), every GC, including a no-GC scheme, incurs implicit costs tightly coupled with the application execution. In the case of Epsilon, it does not move objects to reduce fragmentation, and therefore the spatial locality of objects is poor. In addition, Epsilon does not reclaim unreachable objects, and new allocations will always need to request more physical memory from the operating system.

For the rest of the section, I focus on the costs of collectors other than Epsilon.

Stop-the-World (STW) GCs vs. Concurrent GCs

Overall, STW collectors (Serial and Parallel) are cheaper than concurrent collectors (G1, Shenandoah, and ZGC), both in terms of time and cycles. The exception is that Serial is uncompetitive in terms of time due to its lack of parallelism. At a $3.0 \times$ heap, in terms of cycles, STW collectors are never more expensive than concurrent collectors.⁶ In terms of time, STW collectors are only more expensive than concurrent collectors for 2 out of 17 benchmarks: batik and zxing.⁷

Single-Threaded vs. Multi-Threaded STW GC

Between two stop-the-world collectors, Parallel is more expensive than Serial in terms of cycles (as much as 48 % for a small $1.4 \times$ heap), and the reverse is true in terms of time. This effect occurs presumably because a multi-threaded collector exploits the available parallelism to shorten pauses but introduces synchronization overhead among collector threads. At $3.0 \times$ heap, in terms of cycles, Parallel is only cheaper than Serial for 3 out of 17 benchmarks: lusearch, tradebeans, and tradesoap.⁸ In terms of time, Serial is only cheaper than Parallel for avrora.⁹

Concurrent Tracing vs. Concurrent Copying GC

Among concurrent collectors, the newer, concurrent copying collectors (Shenandoah and ZGC) are significantly more costly than the concurrent tracing collector (G1). The difference is up to 75 % in terms of cycles (G1 vs. ZGC at $2.4 \times$ heap) and 78 % in terms of time (G1 vs. Shenandoah at $1.9 \times$ heap). This is presumably due to the use of costly read barriers and the (un)timeliness of reclamation, which I discuss below. At $3.0 \times$ heap, G1 consistently outperforms Shenandoah and ZGC for all benchmarks in terms of time. In terms of cycles, Shenandoah and ZGC are only cheaper than G1 for batik and xalan (not statistically significant for sunflow).

⁶For sunflow, Parallel is 1 % more expensive than ZGC, but the confidence intervals overlap.

⁷The differences for jme and sunflow are negligible (and also not statistically significant for sunflow).

⁸The differences for jython and sunflow are not statistically significant.

⁹The differences for biojava, luindex, and sunflow are not statistically significant.

Pathological Modes of Concurrent Copying Collectors

We observe two pathological modes of concurrent copying collectors when challenged with workloads that have high allocation rates but low survival rates. In such scenarios, the concurrent copying collectors often fall back to STW collections and/or stall the mutators to keep up with the allocation, resulting in poor performance.

We observe a stark result for xalan, where Shenandoah has an enormous time LBO of 30.2, about ten times that of Serial/Parallel/G1. ZGC simply failed to run xalan with OOM errors. Despite the high time LBO, Shenandoah has a modest cycle LBO of 1.74, which is close to the 1.63 LBO of Parallel and even slightly better than the 1.78 LBO of G1. To understand this behavior, recall that Shenandoah and ZGC rely on tracing to establish liveness, and strictly rely on evacuation for reclamation (see [Section 3.2.4](#)). The consequence is a substantial delay between when an object becomes unreachable and when the unreachable object is reclaimed. The delay’s impact is amplified by the high allocation rates and low survival rates of benchmarks such as xalan and lusearch;¹⁰ because allocation would fail if reclamation did not keep up with the allocation rate, the collector resorts to STW collections. In this case, it would be more beneficial to use a STW collector in the first place and thus avoid the concurrency overhead (as I discuss in [Section 3.5.3](#)).

By examining the logs from Shenandoah, we observe two pathological modes. First, the untimeliness of reclamation causes allocation failures, and Shenandoah requires STW collection to finish an in-flight concurrent collection (known as degenerated GCs in Shenandoah). Second, in order to avoid STW collections, Shenandoah throttles allocations by stalling the mutator at allocation sites (known as pacing in Shenandoah, or “allocation stall” in ZGC). Since sleeping threads do not contribute to the cycles consumed, but increase the wall-clock time needed to run a workload, this explains the much higher time cost but modest cycle cost.

3.5.3 Common Misinterpretation of Evaluation Results

In the previous section, I compared the costs of different types of GCs. In this section, I use these results to highlight three important types of misinterpretation of GC performance measurements: i) ignoring opportunity cost, ii) ignoring concurrency overhead, and iii) measuring GC pause time instead of user-experienced latency. Such misinterpretation could explain why the substantial costs incurred by these collectors have gone unnoticed. I want to emphasize that the misinterpretation I present in this section is also applicable in classic comparative analysis (e.g., [Table 3.2](#)).

Prior work points out common pitfalls and respective mitigations when evaluating GCs [Blackburn et al., 2004a, 2008]. We apply these suggestions where possible (see [Section 3.4](#)). For example, we control the heap size relative to the minimum heap size required to run a given workload. We reaffirm that the fundamental time–space tradeoff is still applicable in a modern, diverse benchmark suite ([Tables 3.6](#) and [3.7](#)).

¹⁰Both benchmarks allocate multiple GBs per second, and their minimum heap sizes are merely 8 MB and 21 MB, respectively.

However, the results in this chapter show that these suggestions are not sufficient to avoid misinterpretation of evaluation results, especially in light of modern hardware, concurrent GCs, and emerging latency-sensitive workloads. In the next section, I recommend techniques to mitigate such misinterpretation.

Opportunity Cost

Examining [Table 3.6](#), it appears that GC has low wall-clock cost. This apparent low wall-clock cost of GC does not come for free. In fact, the speedup provided by multithreaded GC comes with a significant *opportunity cost*, which is the CPU cycles that could have been used to run other workloads.

Comparing [Table 3.6](#) and [Table 3.7](#), a key trend is that all collectors studied have higher cycle LBOs than time LBOs. The only exceptions are Serial, which is a single-threaded GC, and Shenandoah when using small heap sizes due to pacing ([Section 3.5.2](#)).

Hardware parallelism exploited by multithreaded GC causes this difference between time LBOs and cycle LBOs. Examples of hardware parallelism include multicore CPUs or simultaneous multithreading (SMT).

This opportunity cost matters when multiple applications with GC coexist, for example on personal computers and mobile devices. Such multi-tenant deployment is now increasingly common to increase utilization in datacenters. On these hosts, GC of one application can slow down other applications on the same host by consuming too many CPU cores. Conversely, the GC itself now runs slower because fewer cores are available to run GC. Even when the host has only one application, using fewer CPU cycles to run GC can reduce energy consumption.

This opportunity cost has gone unnoticed, because historically GC performance has been mostly measured using wall-clock time only. But the discrepancy I show in this section crisply highlights the danger of only measuring the wall-clock time costs of GC. This prevalent methodology incentivizes reducing the time overheads, while ignoring other costs. In other words, this methodology implicitly assumes that on machines with free cores available, the free cores may be used by GC with no repercussions. This assumption, however, ignores the incurred opportunity cost.

On modern, massively parallel hardware, this assumption can mean that a significant portion of the hardware is dedicated to the GC. For example, the case studies in this section show that the cycle LBOs of Parallel and G1 are about $0.3\times$ larger than the respective time LBO at a small $1.4\times$ heap. In the extreme case, such as for batik at $3.0\times$ heap, Parallel has a mere 1.09 wall-clock time LBO but a significant 1.99 cycle LBO, the highest among all collectors studied. This difference reveals that substantial opportunity costs can go unnoticed when only the wall-clock time is reported.

Concurrency Overhead

Examining [Tables 3.10](#) and [3.11](#), it *appears* that the newer low-pause collectors have both low time and cycle overheads—close to 0 % in fact. These collectors shift most of the collection work from stop-the-world (STW) pauses to run concurrently with

Table 3.10: Percent of wall-clock time spent in STW pauses averaged over 16 benchmarks. Lower is better. The best value for each heap size is shown in green. Where a collector cannot run all benchmarks at a particular heap size, the corresponding entry is left blank. ZGC consistently outperforms other collectors where it runs.

GC	1.4 ×	1.9 ×	2.4 ×	3.0 ×	3.7 ×	4.4 ×	5.2 ×	6.0 ×
Ser.	9.0	4.7	3.7	3.1	2.7	2.5	2.4	2.3
Par.	9.0	2.9	2.1	1.7	1.3	1.1	1.0	0.9
G1	5.2	2.8	1.6	1.3	0.9	0.9	0.7	0.7
Shen.		0.3	0.2	0.2	0.1	0.1	0.1	0.1
ZGC			0.1	0.0	0.0	0.0	0.0	0.0

Table 3.11: Percent of total cycles spent in STW pauses averaged over 16 benchmarks. Lower is better. The best value for each heap size is shown in green. Where a collector cannot run all benchmarks at a particular heap size, the corresponding entry is left blank. ZGC consistently outperforms other collectors where it runs.

GC	1.4 ×	1.9 ×	2.4 ×	3.0 ×	3.7 ×	4.4 ×	5.2 ×	6.0 ×
Ser.	4.3	2.1	1.6	1.4	1.2	1.1	1.0	1.0
Par.	13.1	5.3	4.0	3.4	2.8	2.6	2.4	2.2
G1	8.1	5.0	3.3	2.8	2.1	2.1	1.9	1.8
Shen.		0.1	0.1	0.1	0.1	0.1	0.1	0.0
ZGC			0.0	0.0	0.0	0.0	0.0	0.0

the application. But focusing on GC pauses ignores the significant *concurrency overhead* imposed by the concurrent collectors.

With fixed hardware resources, concurrent GC threads compete with mutator threads, e.g., for cache capacity and memory bandwidth. In other words, concurrent GC costs are high not only from the expensive mechanisms they use, such as read, and write barriers, but also from resource contention. For workloads with higher mutator parallelism, this interference is more pronounced.

Unfortunately, optimizing for low GC pause times is a common practice among GC practitioners and researchers. For example, the GC optimization guide from a JVM vendor suggests tuning GC for throughput by adjusting the fraction of time spent in GC pauses (`-XX:GCTimeRatio`) [Oracle, 2021a]. Hoping to achieve better application responsiveness, the newer collectors also have low pause times as their stated design goal [Flood et al., 2016; Lidén and Karlsson, 2018]. But as I will show in the next section, this design does not always deliver better application responsiveness.

Here, I focus on the high GC costs that are easily overlooked when focusing only on GC pauses. Recall that Tables 3.6 and 3.7 show the lower bounds (underestimates) of the overheads of each GC as the geometric means over 16 benchmarks, grouped by heap sizes. The tables show the overheads in terms of wall-clock time, and CPU cycles, respectively. In contrast, Tables 3.10 and 3.11 show the fraction of time and cycles spent in STW pauses for different GCs. It is clear that the classic methodology of estimating the GC costs from the time/cycles spent in GC pauses undercounts the significant overheads of concurrent collectors. In particular, the two concurrent copying collectors spend a negligible fraction of time/cycles in GC pauses, but have enormous LBOs.

Low Pause $\neq$ Low Latency

In the previous section, I discussed how the substantial overheads of concurrent GCs are hidden when focusing only on the GC overheads incurred in pauses. In this section, I discuss a related problem—using pause times as a surrogate when optimizing GCs for latency-sensitive workloads. As a result, language implementers can unintentionally work against the motivating goal (the responsiveness of the application), despite achieving the stated design goals.

Figure 3.3 crisply illustrates the problem. The figures show the distribution of pause times and query latencies (using metered latency; see Section 3.4) of different GCs running the lusearch benchmark at a $3.0 \times$ heap. The low-pause collectors (Shenandoah and ZGC) indeed achieve better pause times (Fig. 3.3(a)) in general, with ZGC consistently having the lowest pause time for all percentiles, while Shenandoah has lower pause times than the other three GCs under the 90th percentile. However, low pause times do not automatically confer low application latencies. Indeed, both Shenandoah and ZGC have worse (by factors of 10–100 $\times$) query latencies (Fig. 3.3(b)) than the other three collectors.

Unlike other misinterpretations I discussed in this chapter, this misinterpretation is not new. Two decades ago, Cheng and Blleloch [2001] clearly explained why pause

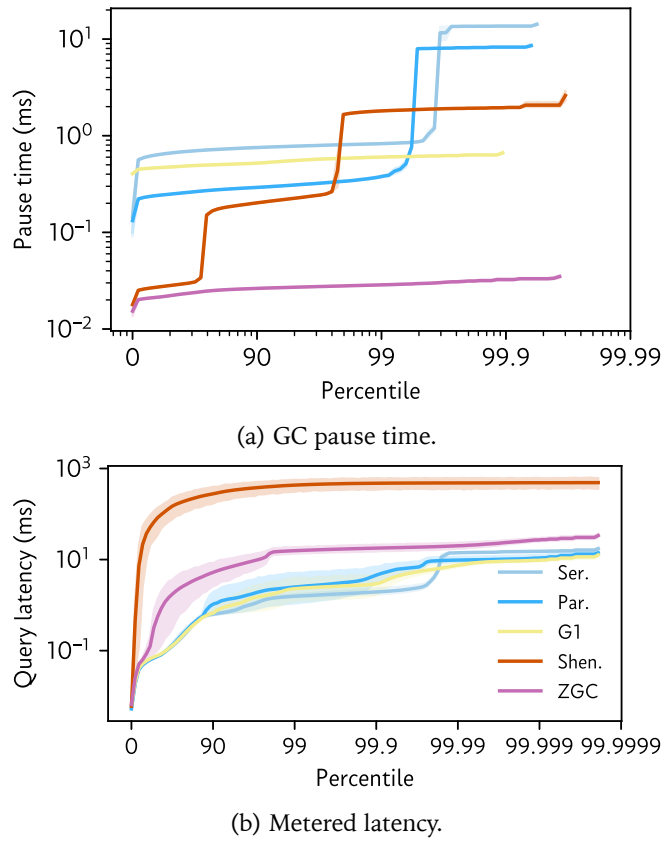


Figure 3.3: GC pause time vs. metered latency for lusearch in a $3.0 \times$ heap. Each line and its shade show the mean and 95% CI over 20 invocations.

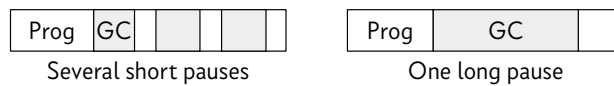


Figure 3.4: Cheng and Blelloch [2001] used a figure like this to illustrate the problem with using GC pauses as a measure of responsiveness and proposed the minimum mutator utilization (MMU) metric as a response. GC pause time continues to be widely (mis)used as a proxy for responsiveness more than twenty years later.

times should never be used as a proxy for user-experienced latency, but GC pauses continue to be (mis)used this way. Application latencies are affected by GC pauses—both by their durations and frequency. As [Cheng and Blleloch](#) pointed out, several short pauses may have a similar or worse effect than a long pause ([Fig. 3.4](#)). They proposed the minimum mutator utilization (MMU) metric to reflect how much CPU was available to the mutator over a sliding window of time, for various window sizes.

Even so, MMU is not ideal since it is a single-threaded measure, and cannot capture the concurrency overhead I discussed previously. In addition to GC interruptions, user-experienced latencies are also a function of mutator performance. As discussed in previous sections, concurrent copying GC can affect mutator performance through expensive mechanisms, such as read barriers, and through resource contention.

In [Section 3.5.2](#), I discussed the pathological modes of the concurrent copying GCs, where they throttle mutator threads when the GC cannot keep up with allocation. Such throttling indeed avoids triggering a STW collection, keeping each GC pause short. However, it comes at a great cost: if the mutators are sufficiently throttled, both the application latency and throughput will be worse than those of a simple STW GC, as evident in [Fig. 3.3](#) and [Table 3.8](#).

One might counter that some of these collectors were designed (only) to be used with generous heaps. Recall that the LBO analysis in [Tables 3.6](#) and [3.7](#) extends to 6× the minimum required memory. Given the cost of memory across mobile, desktop, and datacenter, a 500 % overhead is generous.

I want to emphasize that these collectors have their use cases; for example, when there is generous memory and compute headroom, concurrent copying can indeed improve user-experienced latency by avoiding GC pauses. However, any evaluation methodology must transparently reveal the tradeoffs made by these collectors, including their substantial memory and compute overheads.

3.6 Recommendations

The results presented in [Section 3.5](#) highlight the danger of allowing methodologies to lag behind innovation. First, the significant overhead of GC is overlooked due to common misinterpretation. Given the scale at which Java is deployed, the likely impact is substantial. Second, GC costs increase over time when we consider the years in which each collector design is introduced into the JVM. Although each of these collectors can have valid use cases, we risk misdirecting design and implementation innovations if we cannot crisply identify the tradeoffs made by different designs.

Drawing on these observations, I make two recommendations for improving GC evaluation in future research.

Recommendation 3.6.1. Use the lower bound overhead (LBO) methodology to estimate the real costs of GCs.

This helps us better understand the scale of GC’s impact on program execution.

Recommendation 3.6.2. Use a richer set of performance and cost metrics to evaluate GCs holistically. At a minimum, both wall-clock time and CPU cycles used should be reported.

Any additional metric can help us better understand the performance characteristics of different GCs. This includes measuring application latency for applications with latency requirements, instead of using pause times as a surrogate. Note that outside LBO, using more metrics can still improve evaluations, including comparative analysis.

3.7 Discussion and Summary

In this chapter, I presented the LBO methodology for quantifying the real cost of garbage collectors. The methodology quantifies the substantial costs incurred by highly tuned production software GCs: (as an underestimate) at least 30–243 % more time and 23–123 % more cycle overhead relative to the ideal application cost for a modest $2 \times$ heap. The results additionally highlight a worrying regression in modern collector designs caused by methodological deficiencies.

The results also show how a lack of diverse cost/performance metrics can lead to the misinterpretation of GC evaluations, hindering GC development. I identified three important types of misinterpretation: neglecting opportunity cost, neglecting concurrency overhead, and using GC pause times as a proxy metric for application latency. To mitigate these types of misinterpretation, I recommend that more metrics be used when evaluating GCs, as this helps reveal tradeoffs and limitations that often go unnoticed.

It is heartening to see that the methodology and the results presented in this section have already motivated the development of the LXR collector [Zhao et al., 2022b], and have been used in the evaluation of GCs, including collectors on mobile devices [Zhao et al., 2022b; Sareen et al., 2024].

The LBO methodology is language- and runtime-agnostic. It can benefit GC research on a wide range of managed languages and platforms, such as Go, .NET (for C# and other CLI languages) and V8 (for JavaScript). Additionally, we can use this general framework to evaluate other hard-to-measure overheads in low-level software components, such as the free-list allocators used by malloc/free.

The novel LBO methodology advances our collective methodological foundation of programming language and systems research. It reveals the substantial overheads of software GC, and therefore, provides a resounding motivation for the rest of this thesis. In the next chapter, I move from software to kernel, and investigate how kernel-based tracing tools can provide GC with *observability*.

Improving GC Observability with Performance Tracing

In the previous chapter, I presented the lower bound overhead (LBO) methodology. It allows measuring hard-to-attribute costs in complex systems, such as garbage collectors (GCs), for the first time. The LBO results showed that production software GCs incur substantial overheads despite significant engineering efforts over the past decades. This finding reaffirms that further GC research has great potential.

In this chapter, I address how to better debug and optimize GC through kernel-based performance tracing.¹ The key benefit of tracing is *observability*, allowing one to “ask arbitrary questions about [the system] without—and this is the key part—having to know ahead of time what [one] wanted to ask” [Hound Technology, Inc., 2018]. Conventional techniques, such as *sampling* and *logging*, critically lack observability, causing cognitive friction, and slowing down innovation. In this chapter, I show how tracing can be integrated into GC with minimal code changes and overheads, while still yielding rich insights into the intricacies of the collector behavior, and even revealing bugs in application programs. The results show that tracing can be an invaluable part of our toolkit when working with complex, low-level systems such as GC.

This chapter is based on a published paper I coauthored [Huang et al., 2023]. This publication resulted from an undergraduate research project conducted by Claire Huang, which I cosupervised (Page ix). The implementation is now available as part of MMTk.

4.1 A Lack of Observability in GC

Luu [2016] and Sites [2015, 2021, 2023] eloquently explain why low-overhead tracing has become one of the most important performance debugging tools for systems engineering. Tracing complements and improves upon widely used sampling and logging. Unlike sampling, which does not preserve order and thus is inherently *correlative*, tracing creates an execution trace by recording information every time certain program

¹Note that *performance tracing*—the observability technique described in this chapter—is distinct from *tracing in GC* (Section 2.2), which refers to establishing the object reachability via heap traversal.

points are executed, preserving order and exposing *causation*. Unlike logging, which requires inserting specific log statements in advance, tracing offers *observability*, allowing us to ask arbitrary questions without knowing ahead of time what to ask [Hound Technology, Inc., 2018].

High-performance GC is critical to managed languages, and is heavily tuned in production virtual machines. At the same time, debugging and optimizing GC performance is notoriously difficult. Yet, modern production GCs do not support tracing, despite these advantages of tracing, and its wide use in other fields.

Sampling and logging lack causality and observability, and hinder our ability to examine and inquire about the intricate inner workings of GCs. This in turn hinders our ability to generate new insights that might advance new designs and engineering practices. In this chapter, I show how modern tracing frameworks make GC more observable, especially when combined with modular collector design, doing so *cheaply*, *flexibly*, and with *high fidelity*.

An extensive body of literature i) compares the performance of different GCs [Blackburn et al., 2008, 2004a, 2006; Zorn, 1993], ii) analyzes the cost of various GC components [Zorn, 1990; Hosking et al., 1992; Blackburn and Hosking, 2004; Yang et al., 2012; Zhao and Blackburn, 2020], and iii) more recently, estimates the overall cost of GCs (Chapter 3). However, these methodologies and techniques do not address GC observability.

Current GC implementations suffer from three related problems: i) metrics that operate at the temporal granularity of GC phases lack fidelity, ii) implementors must anticipate the needs of unseen problems in order to predict a set of useful but low-overhead metrics to bake in, and iii) altering the set of built-in metrics requires non-trivial effort. Widely used sampling profilers such as VTune [Intel Corporation, 2023a] sidestep these problems, but run into others: i) their temporal resolution is intrinsically coarse due to their reliance on interrupts [Yang et al., 2015; Luu, 2016], and ii) they report correlation not causation. I now delve deeper into these problems, and outline how this chapter addresses each one of them.

Low-fidelity metrics. Production GCs, such as OpenJDK GCs, organize timing data around the phases of GC (such as root scanning), and then emit it as logs. However, GC-phase metrics are insufficient for performance debugging. For example, a slow heap traversal could be an artifact of the shape of the application’s object graph [Barabash and Petrunk, 2010], or due to frequent synchronization among threads for work stealing, poor load balancing, etc. Moreover, coarse-grained metrics mask subtle performance regressions, inviting an accumulation of performance debt. Likewise, mutator behaviors like allocation and barriers (Section 2.1) are critical to understanding overall performance, yet are often not reported by GCs.

Crystal ball required. One cause of poor fidelity in these metrics is the cost of logging. Logging everything is impractical, so GC developers have to predict the set of metrics to include for each logging scope and verbosity level. Some logging

statements have to be removed by the compiler for production builds, because even run-time checks of whether to log something are too costly in performance hot-paths.² Lack of fidelity is especially problematic when troubleshooting transient performance bugs on a live system, where existing metrics are insufficient but adding more metrics requires modifying and rebuilding the virtual machine.

Non-trivial engineering and recompilation. To add new instrumentation, programmers need intimate knowledge of the codebase: choosing where to add instrumentation, and choosing the appropriate data structure that is thread-safe and low-overhead. Worse, when metrics are emitted as unstructured text in logging statements, each additional metric requires changes to log parsers. As a result, several parsers are built for the same VM [Microsoft Corporaion, 2021; Apple Inc., 2022; Tier1App, 2022].

Sampling is not enough. Luu [2016] notes: “Sampling profilers are great. They tend to be easy-to-use and low-overhead compared to most alternatives. However, there are large classes of performance problems sampling profilers can’t debug effectively, and those problems are becoming more important.” A sampler only reports how many times it witnessed particular functions being executed. It cannot observe how long a particular function invocation took, what occurred immediately before it, or the distribution of invocation times for that function over the course of an execution. Sample rates are uniform, so such tools must sample the entire application with sufficient frequency to expose the rarest events they wish to reveal. Worse, sampling’s temporal resolution is limited by its dependence on interrupt handling [Yang et al., 2015].

To address these problems, we combine two simple ideas: low-overhead tracepoints, and modular collector design. Tracepoints can be shipped with production binaries, and incur negligible overheads when not used. One can attach arbitrary tracing logic to these tracepoints on demand, including on a running binary.

The power of tracepoints is amplified when combined with a modular GC design. One such example is MMTk, a language-agnostic memory management framework that uses a modular algorithm-neutral work packet system [Zhao et al., 2025a]. I illustrate this in Section 4.4, where modifications to MMTk total just 14 lines of code, yet provide a very rich perspective on collector behavior (Section 4.7).

Tracing improves upon sampling and logging in the following ways.

- Inserting tracepoints in the codebase is *inexpensive* both in software engineering terms (14 lines of code, no parser required), and performance terms (no statistically significant performance overhead).
- Tracepoints offer the *flexibility* of writing different tracing logic against the same set of events in a running binary. In our case study of the heap traversal phase

²As an example, OpenJDK provides `log_develop_*` variants of logging that are not compiled for production builds, and OpenJDK GCs use these logging variants in performance hot paths like work-stealing.

of the GC, we are able to interrogate a live system and i) measure the scalability of multithreaded GCs, ii) monitor the properties of work packets [Zhao et al., 2025a] in-flight in a scheduling system, and iii) examine how different workloads affect these run-time behaviors. The entire investigation is performed on the same binary.

- These events are of *high fidelity*, revealing subtle performance regressions.

Since MMTk is modular, and language-agnostic, these advantages readily translate to each language implementation and collector that MMTk supports. Still, I argue that any collector implementation will benefit from tracing.

The case studies yield insights that will guide future research and GC performance tuning in production settings. The techniques I demonstrate are VM- and collector-agnostic, creating a solid foundation on which a set of interoperable tools can be built. Our experiences should motivate the community to build new GCs and improve existing GCs with observability in mind, benefiting GC designers and practitioners.

4.2 A Taxonomy of Performance Analysis Tools

Systems engineers rely on an arsenal of performance analysis tools. They comprise three categories: i) *counting*, ii) *sampling*, iii) and *tracing* [Sites, 2021]. These are complementary—all are important to the performance analysis toolkit. I describe each category of tools in more detail and show how these tools apply to garbage collectors (GCs).

4.2.1 Counting

Counting tools count the occurrence of discrete events (such as the number of GCs), and the sum of quantities (such as the total pages collected). Counters generally have low overheads, and run continuously. They can be read on demand, or printed out by the logging system. Counters are useful for understanding the gross behavior of GC, such as whether an excessive number of GCs are triggered, or what fraction of the heap is alive after collection. However, counters are aggregate measures and lack the detail needed for deeper performance analysis.

4.2.2 Sampling

Sampling tools quasi-periodically³ collect measurements to create a statistical profile of the program. Timer interrupts are used to power widely used sampling profilers, such as Intel VTune [Intel Corporation, 2023a], and OProfile [OProfile Developers, 2023]. When an interrupt fires, these profilers observe program state such as call stacks. When the program ends, the stack samples are used to identify “hot” methods and

³Sampling frequencies often vary slightly over time so that they are not in lockstep with the signals to be observed.

their calling context, which can provide clues to programmers as to where to focus optimization effort. Though a very useful performance debugging tool, sampling is often inadequate when performance debugging GCs due to: i) lack of context, ii) sampling bias, iii) and the inability to establish causal relationships.

Lack of Context

Stack sampling is the most often used sampling method. While such samples provide calling context, they do not reveal the execution context, or program semantics. For example, the Linux perf tool applied to OpenJDK's Parallel collector in OpenJDK reveals object marking, copying, and work queue management as the top functions. This will not surprise any GC engineers, and does not offer actionable information for performance engineers.

Calls to the same function can exhibit very different behaviors depending on the input. For example, when analyzing a function that scans an array of references, the length of the array is a key piece of the puzzle. If one knows the inputs to this function and their distribution, one can then determine the throughput of this function, and subsequently identify outliers for investigation. Sampling loses the context of function invocations, and therefore cannot do any such analysis.

Finally, function symbols may not be the level of abstraction one wants to analyze: a logical event might start in one function call and finish in another function call, or multiple logical events might finish in the same function call.

Sampling Bias

It is well understood that sampling profilers are prone to sampling bias [Arnold and Grove, 2005; Mytkowicz et al., 2010; Yang et al., 2015]. In particular, they are not well suited to analyze rare events—these events can be *unpredictable* but greatly affect the tail performance of workloads. Examples of rare events in GC are large object allocations and long GC pauses due to class unloading.

If sampling tools were to observe relatively rare events, they would need to sample the whole program at the level of fidelity they need for the rare event, which incurs very high overheads. The sampling frequency is also upper-bounded by the frequency of kernel non-maskable interrupts; any faster sampling frequency can prevent the entire OS from making progress. As a result, most sampling-driven profilers (e.g., Intel VTune) operate between 1 Hz to 1000 Hz for user-space sampling, and up to 100 000 Hz for hardware event-based sampling, which is inadequate for analyzing rare events [Intel Corporation, 2023b].

Sampling tools are oblivious to non-CPU waiting time (i.e., a thread is sleeping instead of spinning), and certain kernel time (such as the timer interrupt routine itself, or non-interruptible kernel code), distorting user-space time [Sites, 2021].

Correlation but not causation

Sampling generates a statistical profile of the program execution. Though the profile can provide hints when investigating performance problems, it is impossible to establish causal relationships between events. This is particularly a problem in GC, where GC work is often dynamically generated while recursively traversing the object graph from the root set, and the work is freely distributed among worker threads to achieve better load balancing. GC work can also be triggered by application activities (allocators, barriers). Tracing a pathological event back to its root cause is something sampling cannot do.

4.2.3 Tracing

Tracing creates a dynamic trace of a program's execution by recording a small trace entry each time an active *tracepoint* is encountered. A tracepoint may be: i) any non-inlined function call/return specified at run time by the user, or ii) a user statically-defined tracepoint (USDT) added to the source code. When disabled, tracepoints have very little overhead, a result of very careful engineering (Section 4.3.1). When enabled, each tracepoint may be used to capture information such as the name of a function being entered / exited and a timestamp. Trace entries are written to a ring buffer and are comprehensively processed in temporal order. Tracepoints can be enabled at run time, allowing a developer to attach to a running process exhibiting anomalous behavior and immediately start performing high-fidelity performance analysis.

Tracing addresses shortcomings of counting and sampling, making it a vital part of the performance analysis toolkit. First, tracing instrumentation allows programmers to debug events at the desired level of abstraction, and attach context to the events. Second, tracing allows capturing rare events with very low cost, since the tracing overhead is proportional to the frequency of the events traced. Finally, tracing allows capturing of all events in a time series, and associating them with unique IDs, allowing programmers to establish causal relationships between events.

Sampling-based profilers sometimes adopt elements of tracing. For example, the Concurrency Visualizer in Visual Studio normally samples the program stack, but *traces* every context switch [Microsoft Corporation, 2022]. This is so that the control flow leading to blocking IO or synchronization can be comprehensively captured, rather than be subject to the vagaries of sampling.

4.3 Tracing Technologies and Existing Use Cases

I survey popular tracing frameworks, and overview their limited uses in existing language runtimes. I show that flexible, low-overhead tracing is absent in most garbage collectors, a gap which I will address in this chapter.

4.3.1 Tracing Technologies

The goal of this chapter is to develop low-overhead, high-fidelity performance tracing for GC. Thus, I first survey potential tracing technologies.

eBPF eBPF (extended Berkeley Packet Filter) provides a safe and flexible way for developers to extend the capability of the Linux kernel to facilitate low-overhead tracing. The key component is a sandboxed in-kernel BPF virtual machine (VM) with a verifier. Users can write BPF programs targeting the BPF VM to respond to kernel and user-space events [Gregg, 2019; eBPF Foundation, 2021]. For example, a kernel event can be a context switch, while a user-space event can be the execution of a user-defined tracepoint (more below). eBPF has a wide range of applications, including troubleshooting slow disk IO requests, as well as its namesake application, filtering network packets.

Since eBPF is implemented in the Linux kernel, we primarily interact with it through user-space frontends, such as BCC [IO Visor Project, 2015] and bpfttrace [IO Visor Project, 2018]. BCC provides a more customizable, C-like programming interface, while bpfttrace provides a convenient scripting language inspired by DTrace [Sun Microsystems, 2006] and SystemTap [Prasad et al., 2005]. Both compile tracing programs (probes) into BPF VM bytecode.

There are many types of probes. Since I focus on using bpfttrace to trace user-space events in MMTk—a memory management framework, `usdt` probes and `uprobe`/`uretprobe` pairs are relevant.

`usdt` probes attach to User Statically-Defined Tracepoints (USDTs), which can be inserted anywhere in a program. `uprobes` and `uretprobes` attach to user-space function entries and exits respectively. However, `uprobe` definitions depend on the locations of these functions, and therefore, their utility is limited to non-inlined functions with exported symbols.

As I mentioned above, developers can use USDTs to mark interesting points in the code base (Fig. 4.1). Each USDT has several parts: the provider (namespace), name, and zero or more arguments⁴ to provide the context of execution. USDTs are implemented by inserting `nop` instructions into the application binary at compile time. The locations (instruction pointers) of each USDT are added to the notes section of the output binary file (an ELF file). A probe attaches to a USDT by patching the corresponding `nop` into a breakpoint, and the kernel executes the corresponding tracing programs when handling the breakpoint.

Since our target, MMTk, is written in Rust, we use a third-party Rust library `probe` [Stone, 2014] to add USDT tracepoints to our source code. For example, Fig. 4.1 shows a tracepoint for the number of object graph edges (references) that a function iterates over, and whether the edges are root edges. Then, we can attach a probe (Fig. 4.2) to this tracepoint to understand the distribution of the number of edges in all invocations of `process_edges`, and specially when the edges are root edges.

⁴Each USDT can optionally have a semaphore. The semaphore controls whether a USDT can fire, and if not, skip the potentially expensive computation of USDT arguments.

```
1 fn process_edges(&mut self) {  
2   + probe!(mmtk, process_edges, self.edges.len(),  
3   +       self.is_roots());  
4   for i in 0..self.edges.len() {  
5       self.process_edge(self.edges[i])  
6   }  
7 }
```

Figure 4.1: Inserting a USDT tracepoint into Rust code, which expands into a nop. The original code is in black, and the additional code for the tracepoint is in green. The first argument to `probe!` is the provider/namespace, and the second is the name of the tracepoint, followed by zero or more arguments. In this case, the tracepoint has two arguments, one for the number of edges, and another for whether the edges are root edges.

```
1 usdt:libmmtk_openjdk.so:mmtk:process_edges {  
2     @dist_edges = hist(arg0);  
3     if (arg1) {  
4         @dist_root_edges = hist(arg0);  
5     }  
6 }
```

Figure 4.2: A probe written in the bpftime language. It attaches to the `process_edges` tracepoint in [Fig. 4.1](#). It records the number of edges (`arg0`) into histogram data structures to estimate the distribution of the number of edges in each function invocation, separately counting root edges (when `arg1` is true).

The scripting language of bpftrace provides standard data structures, such as associative arrays. The language also provides several ergonomic data structures like histograms and statistical aggregates (count, minimum, maximum, average) so that tracing observations can be aggregated on the fly. Operations on these data structures are done in the kernel space by BPF programs compiled from bpftrace scripts. Some bpftrace functions (printing, symbolizing addresses, etc.) are asynchronous: they are queued into a kernel–user space buffer, and processed in user space. Invoking these functions too frequently can overrun the buffer and drop events.

The main overheads of tracing a user-space program come from a privilege mode switch (user space into kernel space), collecting the user-space thread context (registers), and executing the specific tracing logic defined by the user in the in-kernel VM.

LTTng and Perfetto Unlike eBPF, which is implemented completely inside the Linux kernel, LTTng [Desnoyers and Dagenais, 2008] and Perfetto [Android Open Source Project, 2017b] can trace user-space programs from user space. Since no kernel/user-space transition is required when tracing user-space programs, LTTng and Perfetto can have lower tracing overhead. Both LTTng and Perfetto can additionally trace kernel events, through a kernel module, and the Linux ftrace interface respectively.

The programs under trace emit tracing events to the tracing daemon process through a shared memory ring buffer.⁵ The ring buffer is periodically flushed to disk through serialization.

Perfetto comes with a web UI that can visualize several trace formats. We do not use Perfetto to perform tracing, but we use Perfetto UI to visualize garbage collection traces collected through bpftrace.

4.3.2 Tracing in Managed Language Virtual Machines

Understanding the performance of the GC is crucial for GC developers. It is also critical for engineers responsible for deploying virtual machines (VMs) in performance-critical settings. While tracing technologies have been used in VMs, as I show in this section, they do not provide insights into the inner workings of garbage collectors, and thus are complementary to this chapter.

Python, Ruby, and Julia Python [Malcolm and Langa, 2016], Ruby [Ruby Developers, 2013], and Julia [Julia Developers, 2011] all provide GC tracepoints. Any tracing tool that understands the USDT format, including bpftrace and DTrace, can attach to these tracepoints. For these runtimes, the tracepoints implemented within their native collectors only tell us when a GC phase begins or ends, and when allocations occur. These tracepoints do not provide sufficient detail to properly understand collector behavior. In Section 4.4, I show that with a sufficiently modular collector design, just a few USDT tracepoints can provide rich insights into collector behavior.

⁵Perfetto has an in-process tracing mode which does not require a daemon process but it can only trace the program it co-locates with.

V8 and Android Runtime V8 [V8 Developers, 2018] uses the Chromium tracing framework. It reports the time spent in most GC phases and the sizes of spaces. It also collects the memory footprint of different types of objects. Android Runtime (ART) [Android Open Source Project, 2017a] uses the Perfetto tracing framework. ART records different phases of GC, and how heap sizes change with each GC epoch. Both V8 and ART use Perfetto as the UI frontend for visualizing traces. Neither provides the level of detail and generality that I show in Section 4.7.

.NET and OpenJDK Compared with the other language runtimes listed above, .NET [Microsoft Corporation, 2016] and OpenJDK allow more events to be observed. They report more statistics after each GC, such as the size of each generation, and allow profiling of allocation sites and heap composition. Their tracing interfaces can be enabled during run time, supporting tools like GCRealTimeMon [Stephens, 2021] and JDK Mission Control [Oracle Corporation and OpenJDK Developers, 2013] respectively.

Although these runtimes have a richer surface for observation, the detailed behaviors of each collector are primarily analyzed through specific and baked-in logging statements. A key reason is that these collectors are built as monoliths, and the developers of each collector necessarily have to guess at which elements of the collector are going to be most useful to the end user. Additionally, they lack facilities to measure the performance of key GC hot-paths, such as allocation sequences (Section 2.1.1).

In contrast, MMTk uses a modular design, and uses self-contained work packets for scheduling, which I discuss more in Section 4.4. These features allow us to reveal rich insights into the collectors' behavior, including the allocation sequences, in ways that are general, collector-neutral, and have very low overhead.

4.3.3 Choosing a Tracing Technology

Though powerful, tracing tools are still subject to the tradeoff between precision⁶ and the impact on the system under test. Although I focus on eBPF (in particular, its bpftrace frontend) in this chapter, there are more aspects to consider when choosing a specific tracing technology.

One important practical consideration is the ecosystem into which the tracing will be applied. For example, Android and Chromium are extensively instrumented using their respective tracing frameworks (Section 4.3.2). Therefore, using the same framework when adding tracing to their GCs would lead to easier integration with the existing workflow and easier correlation of events from different system components.

Another important consideration is the tracing overheads, and the need to trace kernel events. When using eBPF to trace user-space programs, the main overheads come from a privilege mode switch (user space into kernel space), collecting the user-space thread context (registers), and executing the specific tracing logic defined by the user in the in-kernel VM.

⁶A precise measurement distorted by observer effects is not accurate.

Despite its overheads, we use eBPF in this work, because it is widely available as part of modern Linux kernels, and has a rich ecosystem, including the frontends I enumerated above. Since eBPF executes tracing logic in kernel, it does not incur extra overheads when querying the kernel states when handling a user-space event. In [Section 7.1](#), I will discuss emerging user-space eBPF tracing tools.

However, if one wants to purely trace user-space events, and preserve the raw events for offline analysis, Perfetto or LTTng might be a better choice. This is due to the shared memory buffer between user-space processes not requiring a privilege mode switch, and their more efficient on-disk representation of the data than, say, the plain text used by bpftrace.

These choices of tracing technology are of practical importance but are largely orthogonal to the key message of this chapter, which is that tracing combined with modular software design provides low-cost, highly general observability.

4.4 Adding Tracing to GC

I now give an overview of how one can integrate tracing into GC. Using MMTk—a memory management framework—as an example, I show how one can integrate tracing into GC with minimal changes to an existing code base. In later sections, I show that these modifications incur minimal performance overheads ([Section 4.6](#)), and allow us to observe intricate system behaviors ([Section 4.7](#)).

4.4.1 MMTk and Its Work Packets

MMTk is a memory management framework that provides a robust, efficient, portable, and flexible platform for developers to build garbage collectors [[MMTk Developers, 2020](#); [Blackburn et al., 2004b](#)]. Almost all collection work is done by stateless GC workers executing *work packets* [[Zhao et al., 2025a](#)]. Note that MMTk’s work packet system evolves over time. Here, I describe the behavior in the 0.19 release.

Work packets (such as object scanning work packets) contain work items of the same type (such as objects to be scanned). The *size* of a packet is the number of items within it. Each work packet contains a function pointer instructing the GC worker on how to process these work items, and preconditions dictating when it may be executed.

Work packets that share the same preconditions are put into the same *work buckets*. A work bucket only *opens* when all of its preconditions are met; worker threads can then fetch packets from that bucket. For example, the packets for the heap traversal might be put in a bucket that only opens once all root scanning packets have completed.

The distribution of work packets among worker threads is key to scalability. Worker threads acquire work packets from an initial global pool and then consume and produce work packets into thread-local pools. Once the global pool is empty and a thread’s local pool is exhausted, it steals work from the local work packet pools of other threads. Both work stealing and the acquisition of global work packets require synchronization

of threads to avoid race conditions. Thus, the choice of packet size involves a trade-off between increased parallelization and increased synchronization.

Smaller packets allow for greater parallelization as the work can be spread more evenly among the threads. However, if packets are too small, then fixed per-packet overheads can dominate. Analyzing the distribution of work packet sizes is important; it can provide insight into why particular benchmarks scale better and help us identify possible areas where scalability can be improved.

4.4.2 Tracepoints Added

We added just 14 lines of substantive Rust code to MMTk, comprising USDT tracepoints that provide observability into five major behaviors: i) the start and end of each collection (2 LOC), ii) the start and end of processing each work packet (5 LOC), iii) the start and end of each execution of the allocator slow path (2 LOC), iv) the opening of each work bucket (2 LOC), and v) the `process_edges` method (3 LOC). There are an additional ten lines of code to make the above compile conditionally, and nine more lines of import statements and trailing braces. In [Section 4.6](#), I will show that these tracepoints impose an overhead so low that there is no need to conditionally compile them in practice.

While logging dumps textual summaries of what a collector has done (requiring foreknowledge of what the consumer may find useful), the tracepoints we add provide fine-grained, semantically rich building blocks from which the user can perform powerful analysis ([Section 4.7](#)). The power of this approach is greatly assisted by the highly modular design of MMTk. Our goal is not to be exhaustive regarding the tracepoints one can add. Rather, I demonstrate that just a few tracepoints add tremendous observability to the garbage collector while incurring negligible overhead.⁷ I now describe each of the five behaviors observed by the tracepoints we added.

Collection Start and End

When attached, these tracepoints fire once at the start and at the end of each collection, enabling observation of the time spent in the stop-the-world portion of each collection. Like most collectors, MMTk already has explicit instrumentation to measure the time spent in stop-the-world collection, but for completeness, we added them as tracepoints.

Work Packet Start and End

This pair of tracepoints allows the time spent processing each work packet to be observed. The tracepoints also capture the type of the packet (passed as a string reflecting the Rust type). Together, this information provides users with insight into the details of how work is being performed within a garbage collection, how well it is being parallelized, and which packets and packet types are responsible for performance

⁷i.e. with no statistically significant overhead.

anomalies. These work packet events can be readily organized into the format of various tracing visualization tools, such as the JSON format of Perfetto [Google, 2016]. Combined with the bucket opening tracepoint (below), we can further visualize these events by the GC stages in which they appear. These events give us an intuitive yet rich understanding of the dynamics of collection work, which I later illustrate (Fig. 4.5). When attached, they fire roughly once every microsecond during GC.

Allocator Slow Path

Most allocators are designed around an efficient common-case fast path, which is usually thread-local and unsynchronized, and an infrequently taken slow path, which is used only when the fast path exhausts its cache of available memory. This pair of tracepoints enables observation of the time spent in the allocation slow path. When attached, they fire approximately once for every 64 KiB of allocation, which can be up to once every $10\mu\text{s}$.

Bucket Opening

This tracepoint allows us to observe the opening of each work bucket. Since buckets are opened at different points during the GC only when certain preconditions are satisfied (Section 4.4.1), the timing of bucket openings provides insight into how long various stages of the garbage collection take. When attached, this tracepoint fires on the order of once a millisecond during GC.

The `process_edges` Function

Each of the above tracepoints is very generic. In contrast, the tracepoint I am about to describe is specific, and provides observability into `ProcessEdges` work packets. These packets collectively perform the heap traversal over the object graph, a task which dominates most tracing-based collection work, so we add this tracepoint as an important special case.

We add the tracepoint to the substantive inner function of `ProcessEdges` work packets (`process_edges`) to report the number of edges the function visits and record whether these edges are roots of the heap traversal.

Measuring the size of `ProcessEdges` work packets allows us to gain more insight into problems regarding the scalability and efficiency of tracing. As discussed in Section 4.4.1, this is because larger packets better amortize the fixed cost of fetching a packet, while smaller packets are more evenly distributed among worker threads. When attached, they fire approximately as frequently as the work packet start and end tracepoints; up to once every microsecond during GC.

I will outline our methodology and evaluate the overheads associated with these tracepoints before stepping through case studies that illustrate the utility of the tracepoints.

4.5 Methodology

I now present the baseline methodology used throughout the remainder of this chapter. In the overhead analysis and case studies that follow this section, I describe any deviations from this baseline methodology where applicable.

Hardware and Software Platform

Our work is available as part of the new Rust-based implementation of MMTk [Blackburn et al., 2004b,a; MMTk Developers, 2020]. We use the 0.19.0 release of MMTk and its OpenJDK binding (based on OpenJDK 11.0.19+1) in our evaluation. MMTk is built using version 1.71.1 of the Rust compiler, utilizing profile-guided optimization. We use version 0.5 of the Rust `probe` crate to add USDT tracepoints to MMTk. Both machines we use run Ubuntu 22.04.3 installed with the 5.15.0-79-generic kernel. We use the official binary release of version 0.18 of bpftool.

We use two hardware platforms described in Table 4.1, which we refer to as Zen 3 and Coffee Lake respectively.

Table 4.1: Specifications for the hardware used in the study.

	Coffee Lake	Zen 3
Model	Core i9-9900K	Ryzen 9 5950X
Year	2018	2020
Technology	14 nm	7 nm
Clock	3.6 GHz	3.4 GHz
SMT × Cores	2 × 8	2 × 16
L1 Data Cache	32 KB × 8	32 KB × 16
L2 Cache	256 KB × 8	512 KB × 16
LLC	16 MB	64 MB
Memory Size	128 GB	64 GB
Memory Type	DDR4-3200	DDR4-3200

JVM Parameters

We set the heap size to a modest three times the minimum heap size required by each benchmark using the Immix collector. We also use `-XX:-TieredCompilation -Xcomp` flags to speed up the warmup of the JVM, except for `tradebeans` and `tradesoap` because they cannot run when forcing compilation.⁸ We disable compressed pointers, weak references, and class unloading because the MMTk/OpenJDK binding does not support these features in the version we use. The tracepoints we add are entirely

⁸<https://github.com/dacapobench/dacapobench/issues/198>

collector-agnostic. The methodology, including the specific scripts we use, is applicable to other collectors in MMTk. We focus on the Immix collector [Blackburn and McKinley, 2008]; this full-heap, stop-the-world collector is performant and easier to understand than concurrent or generational collectors, which aids exposition.

Benchmarks

We use 21 diverse, up-to-date benchmarks from the Chopin development branch of the DaCapo benchmark suite [Blackburn et al., 2025; DaCapo Developers, 2022].⁹ We exclude batik because it does not perform GC with the heap size we use. We fix a bug in lusearch during our case studies (Section 4.7.2), and include its original buggy version (lusearch_bug)¹⁰ in our evaluation.

Execution Methodology

For each case study, we invoke each benchmark 30 times. We interleave invocations of different case studies to minimize bias due to systematic interference. In each invocation, the benchmark warms up using four iterations, and we report the results from the fifth iteration. For each case study, we report the aggregated result from the 30 invocations for each configuration using the mean and 95 % confidence interval where appropriate.

4.6 Tracing Overheads

This section quantifies the overheads associated with tracing considering three scenarios: i) the overhead of adding tracepoints (nops) to MMTk (but without attaching probes to any of them at run time), ii) the overhead of attaching a probe to the bucket open tracepoint to observe GC phases, iii) the overhead of attaching a probe to observe the size of `ProcessEdges` work packets, and iv) the overhead of attaching probes to the allocator slow path start and end tracepoints to observe the slow path durations. For each of the three observation applications, we attach probes to the GC start/end tracepoints to count the number of GC epochs. We use both the Zen 3 and Coffee Lake platforms to assess the architectural sensitivity of the results, presented in Table 4.2 and Table 4.3 respectively.

We do not measure the overhead of observing the start/end timestamps of each packet (used by Perfetto visualization) because we cannot directly compare across benchmarks since the sample rate for each is hand-tuned. This is necessary because the work packet execution frequency can be as high as once a microsecond. Observations can sometimes be lost because printing in eBPF is done asynchronously through a fixed-size kernel–user space buffer (Section 4.3.1).

⁹Commit [6ea164a5](#), 2023/09/10

¹⁰Commit [odo47f55](#), 2023/02/02.

Table 4.2: Tracing overheads on Zen 3 for total time, stop-the-world collector time, and mutator time, for each benchmark normalized to the base case of an unmodified MMTk. In each group of four columns, the first measures tracepoints compiled in but not attached at run time (tp). The second uses the open bucket tracepoint (ob), the third uses the `process_edges` tracepoint (pe), and the last one uses the allocation slow path tracepoints (as). In each case, we show the mean and the 95 % confidence interval (gray) of the time normalized to the base case, aggregating across 30 invocations.

Benchmark	Total				STW Collector			Mu.
	tp	ob	pe	as	tp	ob	pe	as
avroa	0.995 ±1.3 %	1.000 ±1.3 %	0.999 ±1.5 %	1.003 ±1.4 %	1.004 ±2.5 %	1.022 ±2.2 %	1.048 ±2.2 %	1.002 ±1.4 %
biojava	1.003 ±0.5 %	1.003 ±0.5 %	1.000 ±0.5 %	1.021 ±0.4 %	1.015 ±1.1 %	1.014 ±1.1 %	1.018 ±1.0 %	1.022 ±0.4 %
cassandra	1.000 ±0.5 %	0.999 ±0.5 %	0.998 ±0.5 %	1.000 ±0.5 %	0.995 ±2.2 %	1.001 ±2.6 %	1.016 ±2.3 %	1.000 ±0.5 %
eclipse	1.009 ±11.6 %	0.956 ±5.8 %	1.007 ±11.4 %	0.984 ±6.0 %	1.002 ±0.9 %	1.008 ±0.9 %	1.020 ±0.9 %	0.983 ±6.0 %
fop	0.995 ±0.5 %	0.996 ±0.6 %	0.996 ±0.5 %	1.033 ±0.5 %	0.989 ±4.5 %	0.972 ±4.1 %	0.997 ±4.2 %	1.034 ±0.5 %
graphchi	1.000 ±0.1 %	0.999 ±0.5 %	1.002 ±0.1 %	1.009 ±0.3 %	1.015 ±1.0 %	1.021 ±0.9 %	1.025 ±0.9 %	1.009 ±0.3 %
h2	1.008 ±1.0 %	1.005 ±1.0 %	1.011 ±1.0 %	1.018 ±1.0 %	1.022 ±1.3 %	1.010 ±1.2 %	1.018 ±1.1 %	1.016 ±1.1 %
h2o	1.002 ±0.9 %	1.005 ±0.9 %	1.002 ±0.9 %	1.058 ±0.9 %	1.003 ±1.6 %	1.014 ±1.6 %	1.025 ±1.8 %	1.062 ±0.9 %
jme	1.000 ±0.0 %	1.000 ±0.0 %	1.000 ±0.0 %	1.001 ±0.0 %	1.023 ±3.4 %	0.999 ±4.1 %	1.312 ±3.6 %	1.001 ±0.0 %
jython	1.005 ±0.3 %	1.002 ±0.3 %	1.004 ±0.3 %	1.036 ±0.2 %	1.015 ±1.6 %	1.005 ±1.2 %	1.017 ±1.2 %	1.038 ±0.3 %
kafka	0.992 ±4.8 %	0.997 ±5.1 %	0.980 ±3.8 %	0.979 ±3.8 %	1.047 ±6.5 %	1.045 ±6.5 %	1.053 ±6.5 %	0.978 ±3.9 %
luindex	1.006 ±0.5 %	1.006 ±0.5 %	1.011 ±0.5 %	1.021 ±0.6 %	0.998 ±2.4 %	1.013 ±2.1 %	1.041 ±2.2 %	1.022 ±0.6 %
lusearch	1.008 ±0.8 %	1.036 ±4.8 %	1.015 ±0.7 %	1.021 ±0.7 %	1.027 ±2.4 %	1.030 ±2.5 %	1.046 ±2.2 %	1.019 ±0.3 %
lusearch_bug	1.070 ±8.8 %	1.026 ±4.7 %	1.333 ±0.6 %	1.004 ±0.6 %	1.005 ±1.9 %	1.006 ±2.0 %	1.829 ±1.5 %	1.001 ±0.6 %
pmd	0.997 ±0.7 %	0.998 ±0.7 %	0.998 ±0.7 %	1.007 ±0.7 %	0.982 ±3.2 %	0.986 ±3.4 %	1.003 ±3.3 %	1.012 ±0.3 %
spring	0.998 ±0.7 %	0.995 ±0.7 %	1.034 ±0.8 %	1.008 ±0.7 %	1.016 ±1.5 %	1.009 ±1.5 %	1.237 ±1.3 %	1.006 ±0.8 %
sunflow	1.017 ±1.9 %	1.002 ±1.9 %	1.028 ±2.5 %	1.007 ±1.9 %	1.017 ±1.5 %	1.003 ±1.3 %	1.020 ±1.3 %	1.006 ±2.2 %
tomcat	0.998 ±0.3 %	0.999 ±0.3 %	1.003 ±0.3 %	1.007 ±0.3 %	1.003 ±1.8 %	1.009 ±1.9 %	1.013 ±1.7 %	1.007 ±0.3 %
tradebeans	0.999 ±0.9 %	0.999 ±1.0 %	1.003 ±0.9 %	1.017 ±0.9 %	1.002 ±1.6 %	1.005 ±1.7 %	1.025 ±1.7 %	1.019 ±0.9 %
tradesoap	0.997 ±0.6 %	1.001 ±0.6 %	1.002 ±0.6 %	1.012 ±0.6 %	1.005 ±1.6 %	1.003 ±1.5 %	1.024 ±1.5 %	1.013 ±0.6 %
xalan	0.986 ±0.8 %	0.997 ±0.9 %	1.001 ±0.9 %	0.990 ±0.8 %	0.971 ±1.8 %	0.990 ±2.0 %	1.005 ±2.0 %	0.993 ±1.0 %
zxing	0.993 ±1.3 %	0.992 ±1.2 %	0.993 ±1.2 %	0.997 ±1.2 %	1.026 ±4.6 %	1.056 ±4.8 %	1.052 ±4.6 %	0.996 ±1.2 %
min	0.986	0.956	0.980	0.979	0.971	0.972	0.997	0.978
max	1.070	1.036	1.333	1.058	1.047	1.056	1.829	1.062
geomean	1.004	1.000	1.017	1.010	1.008	1.010	1.073	1.011

Table 4.3: Summary of overheads for Coffee Lake. This table omits per-benchmark results for concision. The data here corresponds to the last three rows of Table 4.2, which reports overheads for Zen 3.

	Total				Collector			Mu.
	tp	ob	pe	as	tp	ob	pe	as
min	0.972	0.976	0.991	0.985	0.973	0.971	0.984	0.985
max	1.005	1.004	1.239	1.115	1.034	1.067	2.164	1.121
geomean	0.997	0.998	1.013	1.022	1.003	1.007	1.126	1.024

Using Table 4.2, we first evaluate the overhead of adding tracepoints (first column, tp). The average impact on total execution time is within the noise of our measurement: a nominal 0.4 % slowdown on the Zen 3, and a 0.3 % speedup on Coffee Lake. Looking at the per-benchmark results on Zen 3, we see that the overhead is within measurement noise for every benchmark. With respect to collector time, we see slowdowns of 0.8 % and 0.3 % on the Zen 3 and Coffee Lake respectively. These results show that the impact of adding the tracepoints to the codebase is negligible. This low overhead paves the way for compiling in and deploying tracepoints in production settings. This will allow high-fidelity measurements of the system at any time without recompiling or redeploying.

Next, we evaluate the cost of attaching a probe to the bucket open tracepoint (second column, ob). We see no overhead in total time (0.0 %) on the Zen 3, while on the Coffee Lake we see a nominal speedup of 0.2 %. With respect to collector time, we see slowdowns of 1.0 % on the Zen 3 and 0.7 % on the Coffee Lake. These overheads are low enough that the bucket open tracepoints could be left attached in most cases.

Next, I discuss the case where `process_edges` is observed, allowing us to measure the size of each `ProcessEdges` work packet (third column, pe). This is executed many orders of magnitude more frequently than the bucket open tracepoint. Total time overheads are now measurable, 1.7 % and 1.3 % respectively on the Zen 3 and Coffee Lake. The collector sees slowdowns of 7.3 % and 12.6 % on the Zen 3 and Coffee Lake respectively. While measurable, these slowdowns remain small enough that the tracepoints could also be continuously attached in some contexts.

The above quantifications concern the collector. Now we evaluate the overhead of measuring the allocation slow path (fourth column, as). Although a copying collector also allocates during GC, allocation is predominantly performed by mutators. The total time overheads are similar to the above study, 1.0 % and 2.2 % respectively on the Zen 3 and Coffee Lake. The mutator sees slowdowns of 1.1 % and 2.4 % on the Zen 3 and Coffee Lake respectively. Again, the overhead is low enough that the allocation tracepoint could be routinely left attached, allowing us to correlate memory manager behaviors with benchmark events.

Together, these results illustrate that the overheads of tracing via eBPF are very mod-

est. Compiling the tracepoints into the binary incurs negligible overhead, enabling their use even in production settings. Furthermore, I show that interesting tracing applications have such low overhead that in many cases they could be performed in production too.

4.7 Case Studies

In this section, I use two case studies to show how the observability offered by tracing leads to insights into collector and mutator performance. Each of the studies incurs negligible storage overhead, since the data is aggregated on the fly, except for the detailed per-work packet trace (Fig. 4.5), which uses about 20 kB per garbage collection (GC) invocation using a naïve JSON format after gzip compression.

4.7.1 Heap Traversal Scalability: The `ProcessEdges` Work Packet

The performance of tracing collectors (Section 2.2) is typically dominated by how efficiently they perform a heap traversal over all live objects within specific parts of the heap. The traversal may be within one or more regions, the nursery, or the whole heap. Most modern collectors are multithreaded, distributing the work among worker threads to exploit hardware parallelism (Section 2.2.2). The performance of such a traversal is influenced by three factors: i) the efficiency with which each object in the graph is processed, ii) the efficiency with which work is distributed among available workers, and iii) the amenability of the shape of the object graph to parallelization [Barabash and Petrank, 2010].

In this case study, I discuss the scalability of heap traversal performed by MMTk's Immix collector. MMTk uses the `ProcessEdges` work packet to perform the traversal. The traversal is bootstrapped by `ProcessEdges` work packets consisting of roots of the object graph (Section 2.2). Examples of roots are stack references and global references. Each item in a `ProcessEdges` work packet is a single *edge*, the address of a reference to be traced. The work packet processes each edge by dereferencing the edge, and if the referent object has not already been marked, it is marked. In the case of a copying collector, an unmarked referent is also moved. These referents (objects) are then scanned in batches to discover the outgoing edges of each node. The discovered edges are put in a new `ProcessEdges` work packet. This lifecycle of `ProcessEdges` packets repeats until the object graph is fully traversed. For a copying collector, if the referent object was already moved earlier in the collection or is moved by this packet, then during the processing of the edge, the edge is updated to point to the forwarded object.

Methodology

We first attach a probe to the bucket open tracepoint to measure the length of the heap traversal phase of the collection. We then attach a probe to the `process_edges` trace-

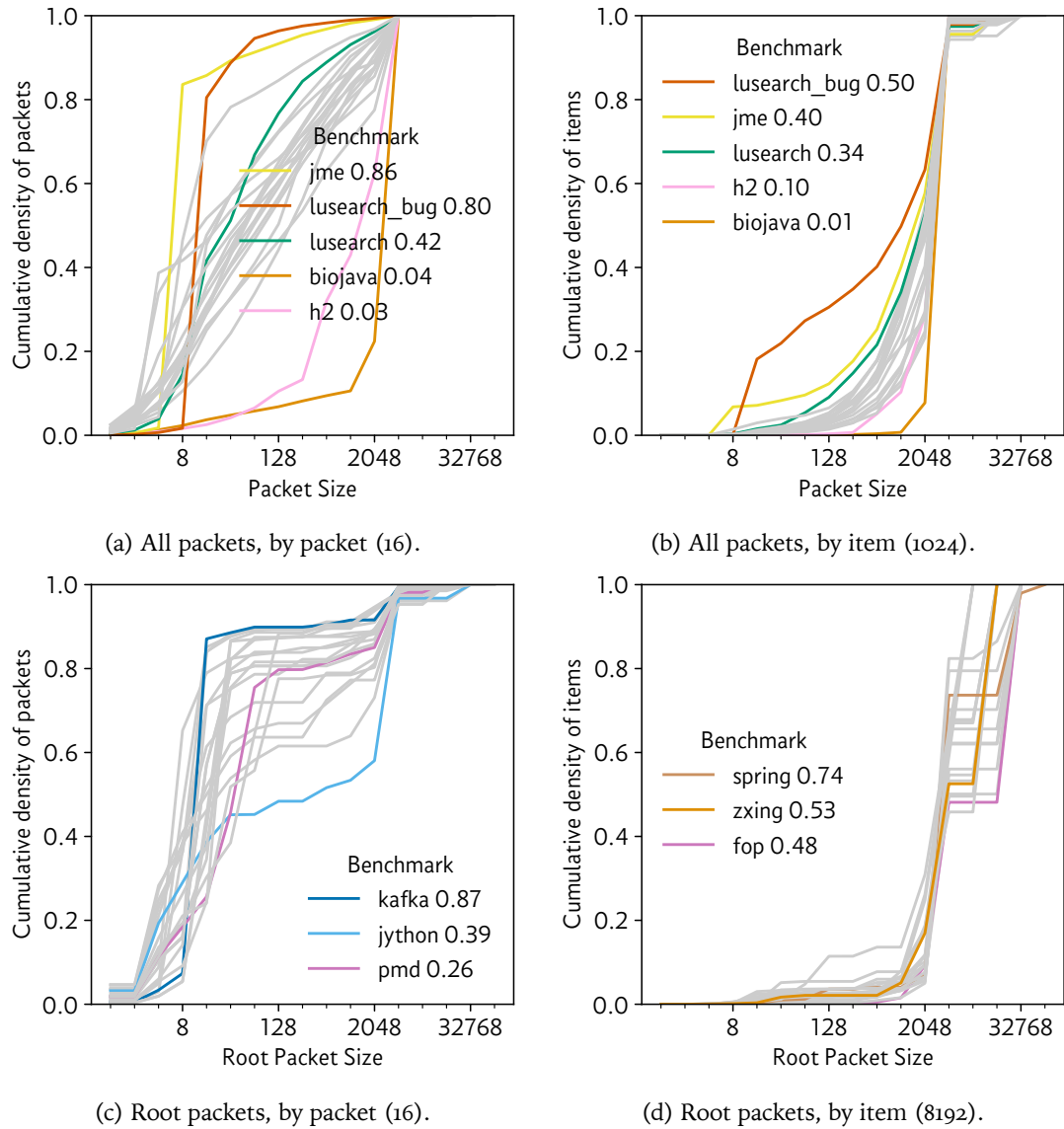


Figure 4.3: Distribution of packet sizes for each benchmark. Key results are shown in color, with the remainder in grey. Packet sizes are rounded down to powers of two and are presented on a logarithmic scale (x-axis). Graphs (a) and (b) measure *all* `ProcessEdges` work packets, while (c) and (d) measure only *root* packets. Graphs (a) and (c) show the cumulative density function (CDF) of packet sizes as a proportion of all packets, while (b) and (d) show packet sizes as a proportion of work items, so larger packets have proportionately heavier weight. All benchmarks are shown (gray) and I highlight three or four notable results in color in each case. The labels indicate the proportion (y-axis) at the packet size (x-axis) indicated in parentheses in the caption. For example, in (a), for `biojava` and `h2`, packets of size less than 16 account for just 3 % of all packets.

point to measure `ProcessEdges` packet sizes. Finally, we attach probes to the work packet start/end tracepoints to save raw timestamped data for Perfetto visualization.

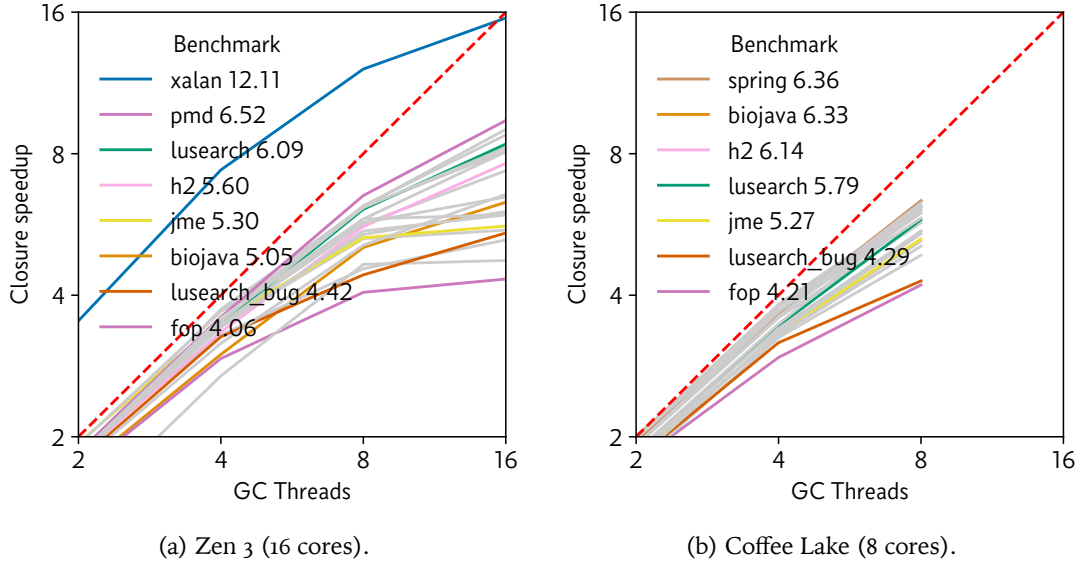


Figure 4.4: The scalability of MMTk’s heap traversal phase (closure) for each of the DaCapo benchmarks, showing speedup relative to using one worker thread (y-axis, log) as a function of the number of worker threads (x-axis, log). The dashed red line indicates perfect scaling. Selected benchmarks are labelled, the remainder shown in grey to indicate the distribution of results. Labels indicate speedup at 8 cores.

Results and Discussion

I now discuss four aspects of the `ProcessEdges` work: i) the overall scalability of the heap traversal phase, in which it dominates, ii) the distribution of packet sizes, iii) the distribution of root packet sizes, and iv) the distribution of packets among threads over time.

Figure 4.4 shows the scalability of the heap traversal phase of collections on each of our hardware platforms. On the Zen 3, it ranges from pmd, which scales with 81.5 % efficiency at 8 threads ($6.52 / 8$), to fop, which is just 50.7 % efficient ($3.96 / 8$). xalan exhibits super-linear speedup on the Zen 3. We ascribe this to an as-yet unconfirmed architecture-specific performance problem when running with a single thread. On the Coffee Lake, scalability ranges from spring, which scales with 79.5 % efficiency ($6.36 / 8$) at 8 threads, down to fop, which is 52.6 % efficient ($4.21 / 8$). I now explore possible reasons for the lack of scalability seen by some of these workloads.

The most likely sources of a scalability problem in the heap traversal phase are poor load balancing and contention. We start by observing that the scalability is very workload-sensitive, and that for some workloads, the scalability is quite good. On this basis, we investigate the distribution of work packet sizes. Small work packets indicate a performance problem since there is a fixed cost associated with acquiring a new packet. On the other hand, very large packets may inhibit load balancing, since

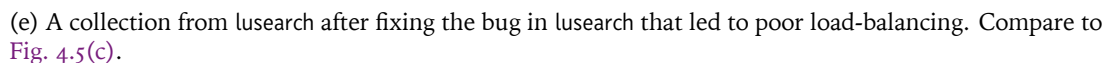
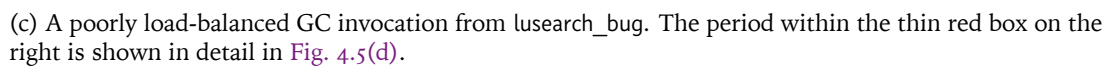
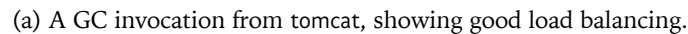


Figure 4.5: Per-work packet tracepoints allow scalability problems to be quickly diagnosed using a visualization tool such as Perfetto, depicted in the screenshots above. The problems with `fop` (4.5(b)) and `lusearch_bug` (4.5(c)) are immediately clear: poor load balancing. Perfetto renders different work packet types in different colors and labels them with their type. Each GC invocation is dominated by the `ProcessEdges` work packet, which is rendered in orange. Bucket opening events are marked with green arrows. The first three and the last screenshots were taken at a fixed resolution to show the entire collections. Perfetto also allows the user to easily zoom in and out (4.5(d)), inspecting the trace at different levels of detail.

MMTk workers can steal packets, but cannot steal work from within another packet.¹¹ We also evaluate root packet sizes, since root packets bootstrap the heap traversal and if they were large and unevenly distributed among workers, they could lead to load imbalance and starvation.

Figure 4.3(a) shows the distribution of packet sizes for all benchmarks on Zen 3 as a cumulative distribution function. The distribution is similar on Coffee Lake. The outliers at the small end are *jme* and the buggy version of *lusearch*, for which about 86 % and 80 % of all packets are less than size 32. The bug fix for *lusearch* (Fig. 4.5(c) and Fig. 4.5(e)) significantly reduces the number of small work packets (down to 43 %). At the large end, the outliers are *biojava*, where about 90 % of packets are of size 1024 or more, and *h2* where about 85 % of packets are at least 512 in size. Interestingly, these observations do *not* correlate well with scalability. In fact, the four outliers from Fig. 4.3(a) are clustered together in the middle of the Zen 3 scalability curve (Fig. 4.4(a)); despite their extremes in packet size distribution, they all have unremarkable scalability. In Fig. 4.3(b), we weight the curves by the size of the packets so that rather than reflecting the fraction of all packets, they reflect the fraction of all items. This different weighting is interesting. However, it comes no closer to explaining the lack of scalability.

Another factor that may affect load balancing is the size of root packets—the work packets that bootstrap the heap traversal. If the bootstrapping work is not well distributed among the threads, it might lead to poor load balancing and poor scalability. Fortunately, this is easy to investigate with the existing tracepoints. Figure 4.3(c) illustrates the distribution of root packet sizes and Fig. 4.3(d) shows the same, weighted by packet size. Unfortunately, neither of these show any correlation with the scalability results in Fig. 4.4. However, they reveal that some of the root work packets are very large, and Fig. 4.3(d) indicates that most root edges come from packets of size 4096 or larger. This suggests an opportunity to improve root processing in MMTk’s OpenJDK binding.

Finally, I use the tracepoints at the start and end of each work packet to visualize the execution of work packets as a function of time and thread affinity. Figures 4.5(a) to 4.5(c) show three GC invocations, two from poorly scaling workloads (*fop* and *lusearch_bug*), and one from a well-scaling workload (*tomcat*). In each case, time runs from left to right, and threads are presented as rows. There are eight worker threads and one coordinator thread (which is almost always idle). It is immediately obvious that the *ProcessEdges* work packets (orange) are poorly load-balanced in the *fop* and *lusearch_bug* collections. Closer inspection reveals that the causes are different.

In the case of *fop*, the collector starts performing finalization work about halfway through. The change in collection phase is denoted by a small green arrow on the bottom-most thread, which indicates the opening of the work bucket responsible for finalization. Although it is not immediately obvious why the finalization work is not shared across the other threads, the source of *fop*’s poor scalability is very clear: it is due to the serial execution of finalization.

¹¹This limitation is fixed in later versions of MMTk [Zhao et al., 2025a].

In the case of `lusearch_bug`, the problem is very different. Zooming in with Perfetto to reveal more detail (Fig. 4.5(d)), we find that the three threads that continue working for a long time (thread IDs 3832035, 3832036, and 3832040) are each busy processing a long series of very short packets. Furthermore, by using the `process_edges` tracepoint, we can annotate each of these short packets with the number of edges, which is 25 in this case. Finally, we see that this same pattern occurs at nearly every garbage collection. We hypothesize that this is due to the collector traversing long linked list data structures, which are inherently non-parallelizable [Barabash and Petrank, 2010].

Having used the GC tracing framework to narrow the problem down to this point, we modified the MMTk source code to dump the types of the objects in each packet of size 25. This immediately revealed the problem. The JDK internal class `PhantomCleanable` uses a doubly linked list to record objects that require cleaning, such as files to be closed when they are no longer used. Each entry in the linked list carries the same data structure, which happens to contain exactly 25 edges, explaining why each packet was 25 items in size. An entry is removed from the linked list only after the referent is cleaned.

This linked list is traversed at every GC invocation, and the traversal is inherently unparallelizable [Barabash and Petrank, 2010]. The remaining question was why this problem showed up in `lusearch_bug` only. Printing out of the objects in each packet answered this question too. Each `PhantomCleanable` corresponded to one of `lusearch_bug`'s 2048 input query files. We discovered that the benchmark was not closing the query files after use, resulting in a very long list. We verified this by fixing the benchmark. After this fix, we discovered another less dramatic instance of the same problem. Benchmark checksum files used by the suite before starting any benchmark were not being closed. We fixed this too and confirmed that the collector scalability problem was solved. The impact of the fix is clear in the Perfetto screenshot in Fig. 4.5(e) and in Fig. 4.4.

Programming errors like the above are simple yet subtle. They would have gone unnoticed without exhaustive per-packet traces (c.f. sampling) and their visualization, which crisply revealed the problem. The timelines also reveal a significant amount of other information, such as the time waiting for mutators to yield (the leftmost packet, labeled `StopMutators`). It also reveals that root processing (blue and magenta) generally scales well, with the exception of the `ScanCodeCache` work packet (olive).

Together, these various perspectives on collector behavior provided by the tracepoints give the developer a wealth of information, allowing them to conduct performance debugging at a very low cost. Traditional performance debugging tools and expert knowledge remain important (such as for identifying the `PhantomCleanable` above), but the process is dramatically faster than one without tracing. Next, I present a case study that uses tracepoints to understand mutator behavior.

4.7.2 Allocator Scalability

The context for this case study is a scalability regression in MMTk that was not immediately detected. It was only evident with the lusearch workload when running on eight or more cores [Lin, 2022]. The problem was debugged and fixed before this work, so this case study is a retrospective analysis.

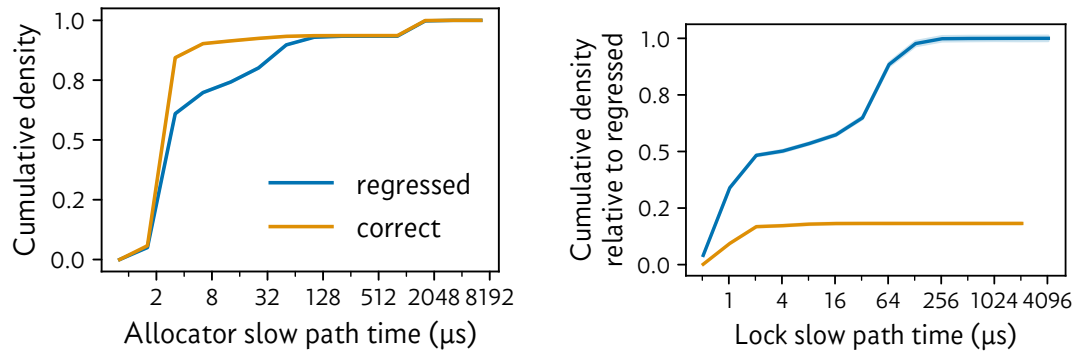
lusearch is interesting because it allocates at a very high rate (about 16 GiB/s) and the workload is embarrassingly parallel—each thread performs a large number of independent search queries against a shared index. It is therefore a good test of allocator scalability. The progression of the bug diagnosis was that the performance of lusearch regressed after a change to MMTk. When the MMTk team looked more closely, they identified that the regression was not due to an increase in collector time but a slowdown in the mutator. Moreover, the slowdown was only evident on machines with high core counts. The problem was evident even in a stop-the-world, whole-heap collector, where the only element of the garbage collector directly impacting the mutator is the allocator (no write barriers, no concurrent collection). Since the allocator fast path (Section 2.1.1) is strictly thread-local, the fast path was unlikely to be the source of a scalability regression. Thus, the focus of this study became the allocator slow path, which is responsible for periodically acquiring new memory from a global pool to replenish the thread-local allocator.

Methodology

We use the `usdt` tracepoint that we added to MMTk’s allocator slow path (Section 4.4), allowing us to observe the time spent in every execution of the allocator slow path. We also use eBPF’s `uprobe` capability, which allows us to monitor the entries and exits of any function without any source code modification (Section 4.3.1). By attaching uprobes to the `Mutex::lock_contended` function symbol exported by the Rust standard library, we are able to precisely measure the time spent waiting for locks, amplifying any allocator slow path regression caused by lock contention.

Results and Discussion

Figure 4.6(a) shows a cumulative distribution function plotting the distribution of allocator slow path execution times as a function of their duration for the correct code (orange) and the regressed code (blue). The regressed code is much less likely to have a short duration ($\leq 2\mu\text{s}$) slow path. This result quickly confirmed the initial hypothesis that the allocator slow path was the source of poor scalability. It was then a matter of identifying changes that affected the allocator slow path within the window of when the regression was observed. In this case, the problem was an unintended coarsening of the scope of a lock used to acquire pages from a global resource. We attached a `uprobe/uretprobe` pair to the `Mutex::lock_contended` function. Figure 4.6(b) shows that the bug fix made a dramatic difference to the time spent on the lock slow path, with almost all cases taking less than $4\mu\text{s}$. Furthermore, the graph shows that



(a) The allocation slow path tracepoint reveals the problem.

(b) Tracing lock contention amplifies the problem further.

Figure 4.6: A regression in allocator slow path performance on lusearch visualized as a cumulative density function. The top graph compares the allocator slow path time (x-axis, log) of the correct code (orange) and that of the regressed code (blue). We would expect two cases for the allocator slow path time, the common case ($2\mu\text{s}$) and the uncommon case (upper right) due to garbage collection pauses, etc. In the regressed case (blue), about 30 % of slow paths are taking $3\mu\text{s}$ to $100\mu\text{s}$, which are suspicious. The bottom graph amplifies the problem by tracing the time spent in the contended state.

the slow path was taken one-fifth as frequently, indicating the lock was far less likely to be contended.

The ability of tracepoints to allow rapid confirmation of such a hypothesis is valuable. Without precompiled tracepoints, each such a hypothesis might have required ad hoc modifications to the source code and a fresh build before any measurement and analysis could be done. The workflow available via eBPF allows this kind of analysis to be conducted systematically and cheaply, even on a running binary.

4.8 Discussion and Summary

In this chapter, I demonstrated that kernel-based performance tracing tools, such as eBPF, provide very rich observability into garbage collectors. This observability is implemented with negligible overhead—lower than our measurement noise, and with just 14 lines of code when building on top of MMTk—a modular collector framework. I used two case studies to show the power of this combination, applying it first to a collector performance bug, and then to an allocator performance bug. I made the case that this approach to performance debugging of collectors is more powerful and more flexible than the dominant state of the art: logging and tracing. Although I use eBPF and MMTk as concrete examples, the same approach can be applied to other tracing frameworks (Section 4.3.1), and garbage collectors.

The tracing tools I presented in this chapter have been made available in the upstream MMTk codebase. They have benefited the development of the MMTk Ruby binding [Wang, 2023], and assisted the development of the next generation of packet-

based work scheduling system [Zhao et al., 2025a].

These results evidence the benefits of rapid feedback, which helps us generate and iterate on new designs and optimizations when working on performance-critical, low-level systems. In fact, as I will show in the next chapter, tracing not only helps with software development, but also when developing a new hardware accelerator (Section 5.5.3). Thus, the principle I presented in this chapter forms an important piece of the cross-stack approach I take in this thesis. In the next chapter, I move from kernel to hardware, presenting a new approach to near-memory GC acceleration.

MAGC-DIMM: A Near-Memory Processing Garbage Collection Accelerator on DIMMs

In [Chapter 3](#), I show that software garbage collectors (GCs) consume substantial CPU resources even in the most favorable circumstances. This substantial overhead raises a fundamental question: should the CPU be performing GC at all? Just as direct memory access (DMA) freed CPUs from data transfers, GC should be delegated to more appropriate hardware, freeing expensive general-purpose cores for application logic. When considering the wide adoption of garbage-collected languages, the benefits to reap here can be substantial.

However, many existing proposals for hardware GC, including near-memory GC, are not practical for commodity machines. They require invasive changes to the host, or rely on obsolete memory technology ([Section 5.3](#)).

In this chapter, I introduce MAGC-DIMM—a new approach to near-memory GC that extends buffered DIMMs, integrating GC acceleration with each DIMM. However, placing GC on a buffer device impedes pointer chasing, because dereferencing pointers can discover objects on other DIMMs, and a single object can be split across DIMMs ([Section 5.4.2](#)). I show how to address this by utilizing inter-DIMM communication architectures, breaking down the heap traversal into low-level operations that can be executed locally on a DIMM, or delegated to remote DIMMs through message passing ([Section 5.4.3](#)). I discuss further optimizations ([Section 5.5](#)) to our scheme that exploit properties of GC. These optimizations allow more compact messages, reduce the number of messages sent, and reduce the idle waiting times of GC units. Some of these insights are derived using a tracing framework similar to what I present in [Chapter 4](#). The evaluation results ([Section 5.7](#)) show that our approach is highly area-efficient—requiring $72\times$ less cache area and significantly less logic area than the CPU, while achieving 68 % of the CPU’s throughput in heap traversal. Finally, I show the behavioral correctness of our approach via an FPGA prototype ([Section 5.7.2](#)).

Our experience shows that near-memory GC can be practical. By offloading GC onto DIMMs, our design could be more easily integrated into the DIMMs of commodity machines, from low-power laptops to general-purpose servers. Our design also

provides a foundation for future research on implementing GC on a wider range of near-memory processing architectures.

This chapter is based on a manuscript that I coauthored [Cai et al., 2025].

5.1 On-DIMM GC: Three Questions

5.1.1 Why On-DIMM GC

Garbage collection (GC) provides memory safety to modern programming languages. However, when implemented in software, GC is expensive. The main activity of GC is identifying live objects through pointer chasing, an iterative process that starts with an initial set of objects (*roots*) and follows outgoing pointers in each object to discover all reachable objects (Section 2.2). As I discuss in Section 2.2.5, this memory-intensive heap traversal is poorly suited to CPUs [Beamer et al., 2015; Maas et al., 2018]. For example, OpenJDK’s heavily tuned default production G1 collector incurs a 16 % overhead even in the most favorable settings (Chapter 3). This substantial overhead, combined with the wide adoption of garbage-collected languages, makes hardware support for GC in commodity machines increasingly attractive, freeing up precious CPU resources for other tasks.

However, many existing proposals for hardware GC, including near-memory GC, are not practical for commodity machines. They require invasive changes to the host, or rely on unconventional memory technology (Section 5.3). On the other hand, near-memory processing (NMP) DIMMs seem to be a promising technology, because they can be adopted on a wide range of commodity machines that use DIMMs.

NMP DIMMs have been used to address the memory bottleneck in a wide class of workloads, such as linear algebra kernels [Gómez-Luna et al., 2022], machine learning [Kwon et al., 2021], and recommendation systems [Ke et al., 2022]. However, on-DIMM GC has not been attempted. In this chapter, I explore this possibility and present MAGC-DIMM¹—a new approach to near-memory GC that offloads heap traversal onto DIMMs.

5.1.2 Challenges and Responses

This new approach to near-memory GC introduces unique challenges. Memory controllers interleave cache lines across DIMMs to promote parallelism for CPU memory operations (Section 5.2). Consequently, an on-DIMM GC unit sees memory only as discontinuous fragments as small as a cache line (64 B). Since cache lines interleave across DIMMs, even a single object may be split across DIMMs.

On the surface, such interleaving seems incompatible with GC—good pointer chasing performance relies on fast access to neighboring objects, which might live on different DIMMs. Consequently, this chapter addresses three questions.

Q1: Is on-DIMM GC possible? (Section 5.4)

¹MAGC is an acronym for Memory Augmented with Garbage Collection.

Q2: Can on-DIMM GC be efficient? (Section 5.5)

Q3: Can on-DIMM GC compete with software GC? (Section 5.7)

Q1 Existing workloads that leverage NMP DIMMs work around the cache-line interleaving problem. They adopt bespoke in-memory data layouts, rearranging the input data when copying from the host to the accelerator.

Because application accesses constitute the vast majority of total heap accesses, adopting a data layout optimized solely for the accelerator would penalize the application’s critical path (e.g., the host is required to transposing memory on every read of a heap cache line). Such a strategy would degrade overall performance and is unworkable.

Therefore, the GC must work with the program heap *as is* in memory. Because cache lines are interleaved across DIMMs, pointer chasing during GC frequently dereferences pointers to data stored on remote DIMMs. Worse, a single object may span multiple DIMMs.

This creates a fundamental architectural challenge: can a distributed system perform global, coherent heap traversal without a centralized view of memory? Additionally, can it do so efficiently, as the overhead of frequent remote accesses could threaten the feasibility of on-DIMM GC?

Contribution: We make on-DIMM GC possible by treating the memory system as a distributed system and leveraging three techniques, applying them to the distinctive properties of garbage collection: i) inter-DIMM communication via external data links [Sun et al., 2021], ii) offloading work to remote memory via message-passing [Ahn et al., 2015], and iii) time-sharing memory between the host and the GC by replicating timing states [Cho et al., 2020].

We design a distributed GC protocol, and decompose GC into several low-level mechanisms, each requiring only local memory access. During heap traversal, as we follow pointers and discover neighboring objects, we push work to other DIMMs when accessing remote memory contents. Note that piggybacking inter-DIMM communication on memory buses is incompatible with the increased bus speed of DDR4/DDR5 due to signal integrity issues (Section 5.3). Thus, we send work to remote memory using external data links, which also avoids congesting the memory bus. Together, these address the challenge of distributed on-DIMM GC efficiently with minimal logic area.

Q2 Our baseline protocol is functionally correct; however, a naïve implementation of it leads to frequent communication and, thus, high overhead. In addition, since heap traversal is recursive and dynamically generates work, determining when the traversal should terminate in a distributed environment is subtle and difficult. Detecting termination via frequent global synchronization is detrimental to performance.

Contribution: We analyze the typical demographics of heap objects across diverse garbage-collected applications, and design a protocol where individual messages

are smaller, and fewer messages are sent overall. This significantly reduces the communication overhead inherent to a distributed memory setting. We also improve termination detection via optimistic spinning, increasing overall utilization.

Q3 The advantage of our approach over performing GC on the CPU is twofold. First, our design uses $72 \times$ smaller cache and significantly less logic area than the CPU, easily fitting inside a DIMM buffer device. Second, on large server workloads where the heap is typically not cache-resident, on-DIMM GC avoids cache thrashing, and the substantial energy cost of fetching the traced heap through the CPU memory hierarchy.

Nonetheless, can a highly area-efficient, near-memory design match the throughput of a power-intensive, general-purpose core? Our evaluation simulates a dual-channel (8-rank) DDR4 system, and compares our on-DIMM GC with a multithreaded software GC parallelized using state-of-the-art techniques [Zhao et al., 2025a].

Despite its small footprint, our design achieves 68 % of the throughput for heap traversal compared with a contemporary 16-hardware-thread, high-end CPU. We demonstrate the behavioral correctness of our approach by prototyping our design in RTL inside a RISC-V SoC running on an FPGA. Our results show that the area-efficiency gains of near-memory GC can justify the modest performance difference, liberating CPU cores to focus on primary application logic.

5.2 DRAM Organization and Contiguity

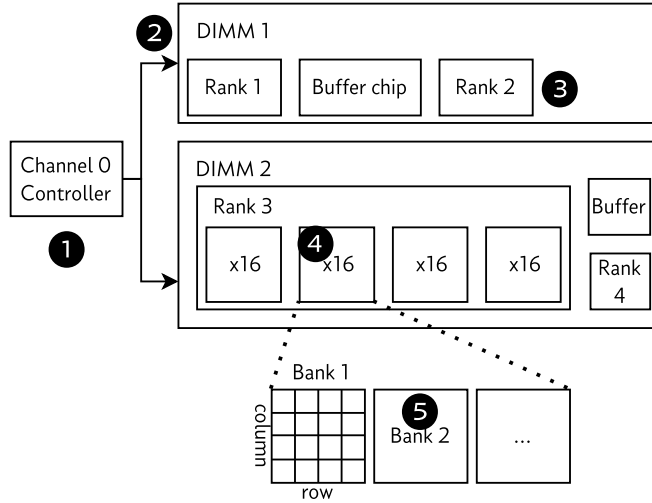


Figure 5.1: Hierarchy of DRAM components.

To understand why on-DIMM garbage collection (GC) is difficult (**Q1**), we examine the hierarchy of components that make up a DIMM-based main memory (Fig. 5.1). These components are channels (1), DIMMs (2), ranks (3), chips (4), and banks (5).

Memory controllers typically reside on the CPU uncore; they use *address mapping schemes* to map cache lines from *linear* address spaces to DRAM components. These schemes use address bits to index specific physical locations, aiming to provide good performance for software running on the CPU. In addition to the CPU’s memory performance, these schemes also affect near-memory acceleration, such as on-DIMM GC, since memory performance is best when the accelerator logic accesses local (co-located) memory.

`row:column:rank:bank:channel:blkoffset` is a commonly used mapping scheme (`blkoffset` denotes locations within a cache line). In this scheme, two consecutive cache lines differ by their channel number, and therefore are placed on different channels. Since channels connect to distinct DIMMs, each comprising one or more ranks, these two cache lines necessarily reside on different DIMMs and ranks.

This scheme promotes channel-level parallelism by utilizing all memory buses and maximizing bandwidth for streaming data. However, because cache lines interleave, GC logic on a rank can access at most 64 contiguous bytes under this scheme. I use the term *contiguity* to refer to the size of the largest contiguous address range locally visible to a particular DRAM component.

Address mapping scheme design is a distinct research field orthogonal to the work presented in this chapter. However, a scheme that optimizes for software workloads (e.g., better at overlapping CPU’s DRAM accesses) can reduce contiguity, affecting on-DIMM GC. I further explore contiguity as a design parameter in [Section 5.5](#).

For on-DIMM GC, ranks (❸) are the lowest-level components with which GC can effectively operate. This is because GC predominantly operates on word-sized pointers, but machine words are split across chips (❹)—all chips within a rank (e.g., $\times 16$, denoting the chip’s width) collectively contribute to the transfer of a single machine word. This striping optimizes for overlapping DRAM operations. Note that memory hierarchies designed for object-oriented languages exist [[Tsai et al., 2018](#); [Tsai and Sánchez, 2019](#)], but they are not general-purpose.

Therefore, in this chapter, I focus on ranks, as they provide the most parallelism while maintaining cache-line contiguity. However, as we will see, cache-line contiguity is hardly a panacea: an object may cross cache-line boundaries, and an object may point to an object on a different rank.

5.3 Taxonomy of Hardware Garbage Collection and Related Work

Specialized hardware for garbage collection (GC) is not a new concept. I present a taxonomy of existing hardware GC proposals, and compare our proposal with them ([Table 5.1](#)).

These designs vary across three main dimensions: i) target applications, ii) GC algorithm generality, and iii) physical placement of the GC logic. These choices significantly influence performance and suitability for general-purpose systems (Implication

columns).

5.3.1 Target Application

Some architectures directly implement specific languages in hardware, such as Scheme-78 [Steele Jr. and Sussman, 1980], SOAR [Ungar et al., 1984], MUSHROOM [Williams and Wolczko, 1990], Jazelle [ARM Limited, 2008], JOP [Puffitsch, 2011], and Cloaca [Ramsey and Stewart, 2024]. Similar designs exist for microcontrollers [García et al., 2021]. These specialized architectures are difficult to adopt in commodity machines.

This chapter focuses on supporting applications written in garbage-collected programming languages running on general-purpose CPUs, such as Java and Go, which are prevalent in environments ranging from mobile devices to datacenters. Some prior work targets these applications, such as Java runtimes [Tang et al., 2015; Maas et al., 2018; Jang et al., 2019; Thomas et al., 2022]. Although we use Java benchmarks [Blackburn et al., 2025] to derive optimization insights, our approach tackles the general challenge of heap traversal (Section 5.4.1), and generalizes across a wide range of garbage-collected languages and applications.

We implement a new form of near-memory processing (NMP) GC that performance heap traversal on DIMMs (Section 5.4). We use FPGAs for prototyping (Section 5.7.2). Other proposals, while seemingly similar to ours, target different contexts. Synthesized hardware GCs designed for FPGA workloads (application logic in RTL) exist [Bacon et al., 2012, 2014; Barker et al., 2022b,a], but they do not target general-purpose software running on CPUs. Similarly, GCs designed for specific near-memory processing workloads also exist [Morimoto et al., 2025].

5.3.2 Generality

Hardware GC schemes typically fall into three levels of generality: i) fixed-function hardware (fixed algorithm or primitives), ii) flexible hardware with limited programmability (e.g., a GC-specific ISA), or iii) fully programmable hardware (e.g., general-purpose embedded CPU cores).

Most prior work implements fixed-function hardware, limiting flexibility. These implementations target fixed algorithms, such as mark-sweep [Maas et al., 2018], or fixed GC primitives, such as object scanning [Jang et al., 2019]. A preliminary attempt to implement GC near memory used a general-purpose CPU for GC, but it does not describe how this approach can scale to multiple cores [Thomas et al., 2022].

Because the in-memory representation of objects differs across language runtimes, and no single GC design fits all workloads, some programmability is desirable. Our design can be implemented in DIMM buffer devices, either in RTL as fixed-function hardware for area and energy efficiency (Section 5.4) or using general-purpose cores for flexibility (Section 5.7.2).

Table 5.1: A taxonomy of hardware GC.

	Suitable for General-Purpose Software	Generality	Placement	Implication		
				Scale w/ capacity	Global memory view	Invasiveness to host
IHGC [García et al., 2021]	✗ Microcontroller	Fixed algorithm	Tightly coupled	✗ No	✓ Yes	✗ High
Scheme-78 [Steele Jr. and Sussman, 1980], JOP [Puffitsch, 2011], Cloaca [Ramsay and Stewart, 2024], and others	✗ Specialized language processor	Fixed algorithm	Tightly coupled	✗ No	✓ Yes	✗ High
In-BRAM GC [Barker et al., 2022b,a], and others	✗ FPGA	Fixed algorithm	Tightly coupled	✗ No	✓ Yes	N/A
Grey-in-Young [Morimoto et al., 2025]	✗ DPU	Fixed algorithm	Tightly coupled	✓ Yes	✗ No	N/A
Tang et al. [2015]	✓	Fixed primitives	Tightly coupled (on core)	✗ No	✓ Yes (through cache hierarchy)	✗ High
Maas et al. [2018]	✓	Fixed algorithm	Direct-memory-access device	✗ No	✓ Yes (through memory bus)	? Medium
Thomas et al. [2022]	✓	Programmable	Direct-memory-access device	✗ No	✓ Yes (through memory bus)	? Medium
Charon [Jang et al., 2019]	✓	Fixed primitives	3D-stacked memory	✓ Yes	✓ Yes (through SerDes)	? Low but requires HMC system
MAGC-DIMM (this paper)	✓	Programmable/fixed primitives	DIMM buffer device	✓ Yes	✗ No	✓ Low

5.3.3 Placement

Some proposals are tightly coupled with specialized hardware or require invasive CPU modification; these proposals are unsuitable for general-purpose systems. GC accelerators that reside on the host side of the memory bus [Maas et al., 2018] have a simpler design, but they reduce aggregate memory bandwidth, and do not automatically scale with the memory capacity. Their advantage often lies in reduced area or energy consumption compared to software GC.

On-memory hardware GCs can achieve better scalability than host-side accelerators by utilizing higher internal memory bandwidth. The only attempt to date targeted Hybrid Memory Cube (HMC) [Jang et al., 2019] (Section 5.3.4). In addition to HMC being an obsolete architecture, the prioritization of bandwidth over capacity means that 3D-stacked memory designs inherently suffer from thermal management and packaging limitations [Hadidi et al., 2017]. Consequently, HBM cannot offer the large memory capacities typically required by datacenter servers. For example, contemporary server CPUs may cap out at 128 GB of integrated HBM per socket while being able to support 6 TB of DDR5 memory per socket. Therefore, we target DIMMs, which are suitable for general-purpose machines.

5.3.4 Fine-Grained GC Offloading to HMC

Charon [Jang et al., 2019] is an on-HMC GC accelerator. It uses the existing software tracing loop (cf. Fig. 2.1) to drive the GC process. For some GC primitives, such as object copying and scanning, the software invokes the accelerator by sending a special memory transaction to offload the work. These fine-grained primitives cause frequent offloading. Despite this, Charon achieves speedup over a software baseline on two big-data workloads, Spark and Graphchi.

However, it is unclear whether this speedup would translate beyond those workloads, which have simple, data-heavy heap shapes, where large numerical data structures dominate. For example, Graphchi’s wide, shallow heap—global object arrays and leaf objects—is easily traversed in parallel, and copying large integer arrays (input graph edges) efficiently leverages HMC’s internal memory bandwidth.

Furthermore, because Charon targeted HMC, which has memory contiguity on the order of several GBs, its design does not address the core challenge faced by on-DIMM GC: cache-line-grained memory sharding. Additionally, HMC’s packetized SerDes interface permits shared DRAM access between the GC and the host. In contrast, DIMMs operate subordinately to the host’s memory controllers, necessitating careful control over GC memory access to meet timing requirements.

5.3.5 Graph Processing Hardware

Although not a hardware GC, and thus not shown in our taxonomy, Tesseract [Ahn et al., 2015] is an HMC-based graph-processing accelerator relevant to our work, because GC heap traversal is a specialized form of graph traversal. It implements a

message-passing protocol for visiting neighboring graph nodes. We also use message-passing to handle dynamically generated work during GC that requires remote memory access.

However, Tesseract’s approach and speedup do not directly translate to GC. It targets graph algorithms like PageRank, where all nodes are statically known and easily parallelized. Its evaluation uses real-world graphs exhibiting power-law degree distributions and small diameters, enabling high memory parallelism and exploitation of NMP bandwidth. In contrast, GC heap traversal is analogous to BFS or DFS, computing reachability from roots, which is not trivially parallelized. In addition, heap graphs usually have larger diameters, and low in-/out-degrees [Barabash and Petrank, 2010; Shahriyar et al., 2012].

5.4 MAGC-DIMM Architecture

Existing near-memory garbage collection (GC) proposals are promising but impractical (Section 5.3). We design MAGC-DIMM—an on-DIMM GC architecture—as a potential alternative to conventional DIMMs in general-purpose machines.

5.4.1 A Focus on Heap Traversal

In Section 2.2, I explain that tracing is the dominant mechanism for identifying reachable objects, and that it can account for over 90 % of GC time in tracing-based GCs. Thus, in this chapter, I examine the three on-DIMM GC research questions posed earlier (Section 5.1, Q1–3) through the lens of tracing.

As discussed in Section 2.2.2, production GCs optimize tracing heavily, using parallel workers and work-stealing. I compare our design with a software baseline that uses such techniques (Section 5.6.3).

Implementing tracing on DIMMs is particularly challenging because accessing data on other DIMMs requires frequent, explicit inter-DIMM communication (Section 5.2). In contrast, a software GC has a coherent, global view of memory through the cache hierarchy and the memory bus.

In the remainder of this section, I first explain how we make DIMMs suitable for GC, focusing on heap traversal. I then present a distributed GC protocol that decomposes heap traversal into low-level, local operations, enabling collaborative execution across DIMMs. Finally, I detail the hardware components that implement this protocol.

5.4.2 Making DIMMs Suitable for GC

Emerging near-memory processing DIMMs (NMP-DIMMs) show promise for memory-bound workloads but do not directly support GC. Applying NMP-DIMMs to GC presents three primary issues: i) the inability to serve as main memory, ii) the inability to read whole machine words when compute is placed near memory banks, and iii) a lack of inter-DIMM communication. While existing proposals address individual issues, they do not demonstrate how to build GC on top of them.

We address these issues by integrating MAGC-DIMM on the buffer device of interconnected DIMMs (Fig. 5.2). I explain the components inside the buffer device in Section 5.4.4.

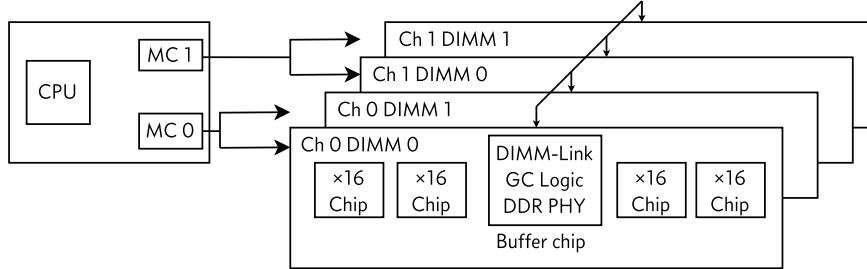


Figure 5.2: MAGC-DIMM is implemented inside buffer devices of interconnected DIMMs. MC = memory controller.

First, most NMP-DIMMs are designed as separate accelerators distinct from main memory. They require copying data from host memory in bulk and do not allow ordinary CPU memory operations. This separation avoids interleaving memory requests from the host and the accelerator, alleviating timing and arbitration complexity. However, a GC must operate directly on the program heap shared with the CPU.

Second, many NMP-DIMMs place compute logic close to memory chips, aiming to provide high internal memory bandwidth. However, since machine words are striped across chips (Section 5.2), any logic on chips has severe contiguity limitations—less than a machine word—unless data is reshuffled when copied to NMP-DIMMs. However, as explained above, copying from the host to NMP-DIMMs is infeasible for GC. Prior work shows that integrating logic inside a DIMM buffer device is feasible [Alian et al., 2018; Ke et al., 2022]. Our design is an instance of this approach; placing GC on a DIMM buffer device enables cache-line-sized contiguity, allowing the GC to access entire word-sized pointers.

Third, many NMP-DIMMs lack direct inter-DIMM communication. Workloads like linear algebra allow static partitioning [Gómez-Luna et al., 2022; Kwon et al., 2021]. In these cases, communication between partitions is typically infrequent, and the host CPU can mediate it. Heap traversal frequently and dynamically demands inter-DIMM communication because pointers may refer to remote memory or objects may be split across DIMMs. As prior work shows, relying on the narrow memory bus and the host CPU to forward inter-DIMM communication severely hinders performance [Zhou et al., 2023]. Thus, we use interconnected DIMMs [Cong et al., 2017; Alian et al., 2018], and add external data links (Fig. 5.2) to facilitate direct, high-speed inter-DIMM communication, thereby bypassing the narrow memory bus and addressing signal integrity issues in modern DDR4/DDR5 memory [Sun et al., 2021; Zhou et al., 2023].

5.4.3 Distributing GC Across DIMMs

Substantial literature exists on distributed GC [Jones et al., 2011], such as for rack-scale disaggregated memory architectures. However, interleaving memory at the cache-line

level presents unique challenges (Fig. 5.3): i) an object can be split across multiple ranks; ii) an object can refer to other objects stored on other ranks; and iii) an object can only be decoded by referring to its type information, which can be stored on other ranks.

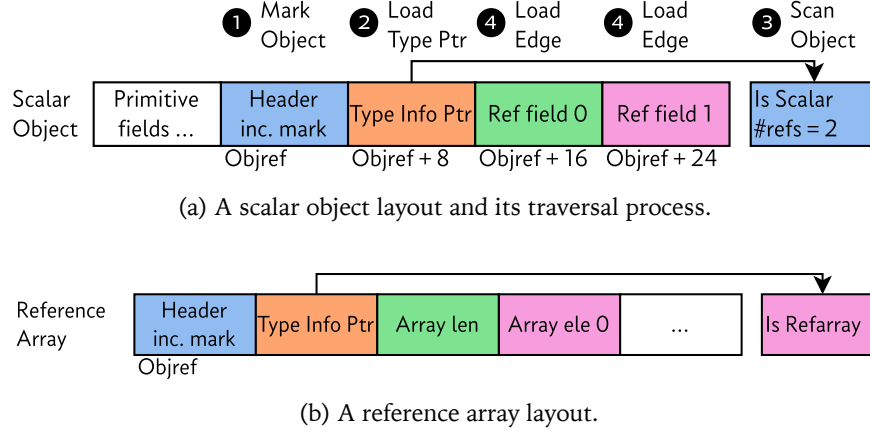


Figure 5.3: Examples of a normal object and an object array splitting across DIMMs (with exaggerated word-size contiguity). Differently colored words represent different DIMM placement. Heap traversal needs to be decomposed into multiple steps through message passing, since an object can be split across DIMMs, and pointers can refer to a remote DIMM.

We overcome these challenges using a distributed GC protocol based on message passing. We decompose GC into low-level operations, each requiring only word-sized local memory access. When an operation requires remote memory access, the GC unit sends a new operation to the relevant remote DIMM,² efficiently encoded as an 8-byte message (Section 5.4.5).

Figure 5.3(a) illustrates the operations required to process a scalar object. This is exaggerated for exposition—each machine word resides on a different DIMM, and thus, each operation is sent as a message to a different DIMM. The semantics of each operation is provided as pseudocode in Fig. 5.4 (cf. Fig. 2.1).

On-DIMM GC units run in parallel, each maintaining its own work queue. When the queue is nonempty, the unit dequeues an object and checks whether it has been marked. If not, we mark it by writing to a side bitmap (1 MarkObject). To decode this object, we dereference a pointer (2 LoadTypePtr) to the type information object, which lies on a different DIMM. Next, the DIMM storing the type information object loads it (3 ScanObject), discovering that the object is a scalar object with two reference fields, and delegating other DIMMs to process those reference fields (generating two 4 LoadEdge messages). Finally, respective DIMMs load those two fields, generating more MarkObject operations for the child objects.

The language runtime bootstraps a GC by sending program roots (Section 2.2) as LoadEdge messages to the appropriate GC units. The GC terminates when all units

²A DIMM can contain one or more ranks, so the destination is a specific rank in the case of a multi-rank DIMM. Since the buffer device can access all ranks on the same DIMM, we refer to DIMMs for simplicity.

```

1  message MarkObject(obj) {
2      if !obj.marked {
3          obj.mark();
4          send LoadTypePtr(obj) to get_dimm(obj + 8);
5      } }
6  message LoadTypePtr(obj) {
7      let type_ptr = *(obj+8);
8      send ScanObject(obj, type_ptr) to get_dimm(type_ptr);
9  }
10 message ScanObject(obj, type_ptr) {
11     let type_info = *type_info_ptr;
12     for i in 0..type_info.num_refs() {
13         off = 16 + i * 8
14         send LoadEdge(obj + off) to get_dimm(obj + off);
15     } }
16 message LoadEdge(edge) {
17     let child = *edge;
18     send MarkObject(child) to get_dimm(child);
19 }

```

Figure 5.4: A message-passing distributed GC algorithm for scanning scalar (non-array) objects. The `send` keyword sends messages to the indicated DIMM. The message recipients are chosen such that the actions in red access the recipient’s local memory. This algorithm is optimized in Fig. 5.9.

are idle and no messages are in flight.

Intuitively, a naïve implementation of this protocol incurs high communication overhead because the total number of messages is proportional to the number of live objects and live pointers. However, as I show in Section 5.5, typical program heap demographics allow for significant optimization.

5.4.4 MAGC-DIMM Components

I now detail how our design implements the distributed GC protocol and fits into the taxonomy in Section 5.3.

Target Application MAGC-DIMM offloads GC work from CPUs, supporting general-purpose, garbage-collected languages like Java and Go. Due to its message-passing architecture, our protocol distributes heap traversal across DIMMs without host coordination (cf. Section 5.3.4). The runtime can invoke a GC, and poll to determine whether a GC has finished by writing to and reading from MMIO registers (Sections 2.2 and 5.4.3). The runtime transmits program roots (potentially in multiple batches) by writing them to memory, and then writes the starting address and the number of roots to MMIO registers.

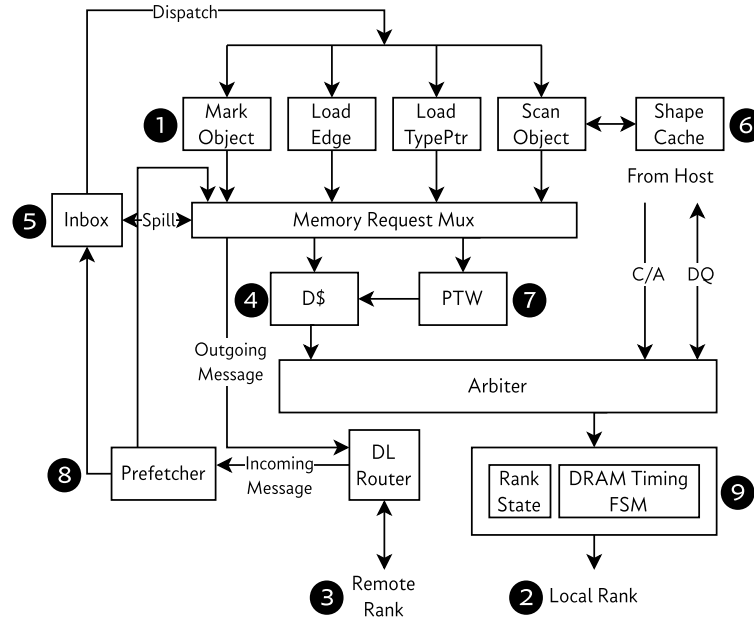


Figure 5.5: Components inside the buffer device.

Generality For optimal area and energy efficiency, each GC unit can implement operations, such as marking objects or looking up type information, directly in RTL as fixed-function execution units (❶). Prior work has shown the feasibility of integrating general-purpose cores inside buffer devices [Alian et al., 2018]. Thus, for flexibility, these GC units can also be general-purpose cores with GC mechanisms written in software, as demonstrated in our prototype (Section 5.7.2).

Placement We instantiate GC units for each DRAM rank within each DIMM’s buffer device, co-located with the inter-DIMM communication interface. This approach scales compute with memory capacity. Each GC unit accesses only its local memory rank (❷), which necessitates collaborative heap traversal through message passing (❸), as described in Fig. 5.4.

Area Our design targets area efficiency, and fits easily inside DIMM buffer devices. For an 8-rank system, the total area consumed by all GC units is estimated to be less than a single CPU core. Each MAGC-DIMM unit includes a 32 kB data cache (❹) and a configurable message queue (❺). For reference, an in-order RISC-V Rocket core with a 32 kB cache takes 0.64 mm² (TSMC 40GPLUS). Although no published figures exist, a DIMM’s buffer chip is roughly the same size as the memory array (60–75 mm²), easily accommodating the GC and communication logic.

Our design offers significant area savings over multithreaded software GC—the total GC area utilized in a dual-channel (8-rank) system is smaller than a single CPU core. For reference, a Skylake core uses roughly 9 mm², and a 2-core group (inclusive of LLC) uses roughly 25 mm² [WikiChip, 2025]. Commercial cores contain complex

execution units, out-of-order mechanisms, and substantial caches (e.g., a Skylake core has 32 kB L1 data cache, 256 kB unified L2 cache, and shared LLC slices per core). Note that Skylake is manufactured using a much more advanced processing node than the TSMC 40GPLUS used in MAGC-DIMM’s area estimation; therefore, the area savings of our design in practice will be much more significant.

Efficient Caching of Type Information Object sizes and shapes (i.e., the location of pointer fields relative to the object header) vary by type, requiring the GC to look up type information for each object to locate pointers (Fig. 5.4 LoadTypePtr). Because type information may reside on remote DIMMs, this lookup can become a bottleneck, causing the local memory to idle. This lookup can also overload DIMMs that store type information for popular types. To avoid expensive remote lookups, we introduce a shape cache that stores frequently accessed type information (6).

In fact, such a cache can be very compact and have high hit rates. While objects are numerous, object types are few, and object shapes are even fewer. Since objects of a few popular types often dominate the heap, 4 bits are enough for a shape cache entry to cover 98 % of objects [Garner et al., 2011].

Virtual Memory We require the host to ensure all heap pages are in memory before GC (e.g., by using `mlock`). For address translation, we integrate a TLB within each GC unit, similar to prior work (Fig. 5 [Maas et al., 2018], Fig. 5(b) [Jang et al., 2019]). Similar to the program heap, page tables can also be distributed across DIMMs; therefore, page table walkers (7) also use a distributed protocol similar to GC to access page table entries on other DIMMs, with the results cached in the TLB.

Coherency and Concurrency Similar to prior work [Maas et al., 2018; Jang et al., 2019], we focus on stop-the-world (STW) GC in this chapter, and do not consider concurrent host modifications to the heap. Prior work argues that flushing the cache at the beginning of a GC incurs relatively small overhead [Jang et al., 2019], and that mutators resuming with a cold cache only require a very short period of time to warm up the cache again [Sareen and Blackburn, 2022]. Nevertheless, our design further relaxes the constraint, and only requires the host to write back dirty cache lines without invalidating them. This cost is small³ compared to the cost of software GC bringing the heap memory through the cache hierarchy via the narrow memory bus. This ensures that the on-DIMM GC units have a consistent view of the heap, while not degrading program performance when resuming after a GC.

Instead of marking objects using header bits (Section 5.4.3), we use a side bitmap on the same rank as the object header for marking. The bitmap is indexed by the object’s physical address after translation. Since object headers are word-aligned, this only incurs a space overhead of 1.6 %.

GC units reserve area for bitmaps by asking the host to block out some physical memory available for the operating system. Since the bitmap is managed by the GC,

³Writing back the entire 16 MiB LLC on a dual-channel DDR4-3200 system takes $\sim 300 \mu\text{s}$.

and is invisible to the program, it avoids invalidating the host’s cache as on-DIMM GC units traverse the heap and mark visited objects. The coherency between GC units is not an issue as GC units operate on spatially non-overlapping sets of objects, and the bitmap is strictly written to by the GC unit on the same rank.

Although we focus on STW GC, we do not lock out the host (which also runs background processes) during GC. We do so by leveraging [Cho et al. \[2020\]](#)’s technique to arbitrate different memory requests, and replicate memory timing states on DIMMs (9). This allows the host’s memory controllers to still directly manage memory refreshes, etc., when an on-DIMM GC is underway. This enables MAGC-DIMM to function as the system’s main memory without a packetized interface.

Termination Detection Detecting termination in a distributed, task-generating algorithm is hard. The heap traversal ends only when all GC units are idle and no messages are in flight. To ensure correctness, each time a GC unit drains its inbox, it enters a global barrier and waits for all other units to be locally done as well. Once all units rendezvous, each unit checks its inbox again. If any inbox is non-empty, the heap traversal resumes. Otherwise, the algorithm has globally terminated. In [Section 5.5.3](#), I discuss further optimization to this algorithm.

Scope and Non-Goals The primary objective of this chapter is to establish the architectural feasibility of near-memory heap traversal within the constraints of modern, interleaved DIMM architectures. To isolate the fundamental challenges of distributed pointer chasing and inter-DIMM coordination, we scope the research as follows:

1. **Focus on Feasibility, Not Productization:** This chapter explores whether a distributed protocol can successfully traverse a heap sharded across DIMMs without CPU intervention. Rather than engineering a “drop-in” replacement for production runtimes, this chapter serves as a proof-of-concept for a new paradigm of hardware GC designs.
2. **Stop-the-World (STW) GC:** Our proof-of-concept focuses on STW GC to establish a clean baseline for traversal performance and area efficiency. Concurrent collection is critical for many production systems, but it is an orthogonal challenge; I discuss potential hardware write-barrier extensions to support it in [Section 5.8](#).

Still, our approach can benefit throughput-oriented applications that choose to use STW GC. For those applications, if running in a single-tenant environment, the power-hungry CPU cores can be put into deep sleep states for overall power saving, which can be significant in the aggregate in datacenters with an overall power cap. When running in a multi-tenant environment, other workloads on the same machine can elastically increase the number of worker threads, and/or schedule back in other workloads to utilize the available CPU compute.

3. **Managed Heap Residency:** We require the host to ensure heap residency prior to GC. While a future production implementation of MAGC-DIMM could dele-

gate page faults to the host CPU and postpone the faulting work item, modern performance-critical GC services are already designed to be heap-resident in all but the most specialized cases [Wu et al., 2024]. Thus, enforcing heap residency provides a practical and realistic baseline for evaluating this research prototype.

4. **Traversal as a Foundation:** We prioritize the tracing/marking phase because it is the most memory-intensive and dominant portion of GC time (over 90 % [Zhao et al., 2025a]). Subsequent phases, such as reclamation, are embarrassingly parallel and well-suited to leverage NMP’s parallelism and internal memory bandwidth; we defer these implementations to future work.

By narrowing our scope to the challenging problem of distributed pointer chasing, we provide a foundation for future research into on-DIMM distributed garbage collection.

5.4.5 Passing GC Messages Efficiently

While Tesseract demonstrates message passing for graph algorithms on 3D-stacked memory [Ahn et al., 2015], its message encoding is not compact, requiring 1.5 kB of SRAM for only 32 entries. These large messages support a wide range of algorithms, carrying function pointers and payloads up to 32 B.

GC exhibits properties allowing more compact messages, requiring only 8 bytes. This offers significant savings over Tesseract’s approach. We decompose GC into simple operations (Section 5.4.3), each encodable using 3 bits, rather than requiring a word-sized function pointer. A 32-bit address field suffices to encode an operand within a rank, addressing up to 32 GB of word-aligned memory, and future DRAM densities can be accommodated with few additional bits.

Furthermore, GC mostly uses non-blocking message passing (a sender does not wait until the receiver acknowledges). Unlike graph algorithms such as PageRank, which require lock-step synchronization, heap traversal can proceed largely without explicit synchronization; it only needs a final check of whether all GC units are done and there is no work in flight. Each incoming message buffers into a small SRAM inbox (256 B instead of 1.5 KiB for Tesseract), and any overflow spills to DRAM (⑤, which can be swapped to disk). This allows GC units to work independently, maximizing utilization and reducing synchronization.

Finally, we incorporate message-based prefetching (⑧), similar to Tesseract. Unlike speculative hardware prefetchers, each received GC message definitively indicates data required in the near future, making it an ideal candidate for prefetching to hide memory latency.

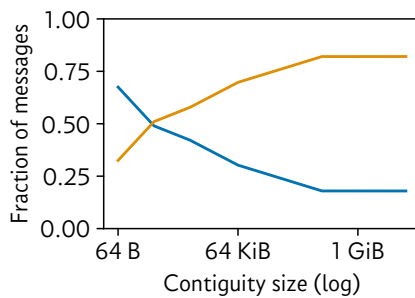
5.5 Workload Analysis and Design Optimizations

The baseline protocol described in Section 5.4 sends messages to remote GC units when objects or pointers cross rank boundaries. These remote messages congest

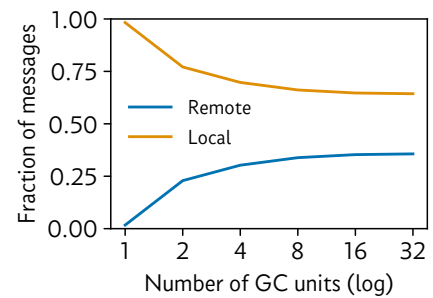
the inter-DIMM links, delay generating subsequent memory requests, and reduce memory-level parallelism. I now demonstrate how to optimize this baseline design.

5.5.1 Quantifying Communication

Figure 5.6 quantifies the communication overhead of our baseline protocol, and shows how this overhead varies with the contiguity induced by specific address mapping schemes (Section 5.2). It shows how the fraction of remote messages changes with memory contiguity and the number of GC units, using the *avroa* benchmark as an example (Section 5.6.1).



(a) Remote messages decrease as memory contiguity increases for each GC unit. The number of GC units is fixed at 4.



(b) Remote messages increase as more GC units participate in the heap traversal. The contiguity window is fixed at 64 KiB.

Figure 5.6: Using the *avroa* benchmark as an example, I show how design parameters affect the fraction of remote messages, and thus, the communication overhead.

In Fig. 5.6(a), I fix the number of GC units at four and vary contiguity. As contiguity increases, fewer messages are remote, because it is less likely: i) for an object to cross contiguity boundaries, and ii) for a pointer to point to remote memory—parent objects and child objects are often spatially close [Huang et al., 2004]. However, large contiguity is unattractive because it reduces memory parallelism.

Interestingly, in practice, even with very large contiguity (1 GiB), 20 % of messages remain remote for a benchmark with a small memory footprint like *avroa*. Even when the total size of live objects in the heap appears to fit within a contiguous range of physical memory on one DIMM, the operating system’s page allocation can still split the heap across multiple DIMMs. Similar trends appear across other benchmarks we study.

In Fig. 5.6(b), I fix contiguity at 64 KiB and vary the number of GC units. More units increase communication, posing a scalability challenge. With just four units, nearly 70 % of messages are remote.

We first apply standard optimizations, such as encoding object shapes in the object header [Gagnon and Hendren, 2001; Garner et al., 2011], avoiding nearly all LoadTypePtr and ScanObject messages for scalar objects (Fig. 5.7). The remaining messages are still substantial, and most are of type LoadEdge, which dereference object fields, accounting for approximately 70 % of all messages. This makes them a key candidate

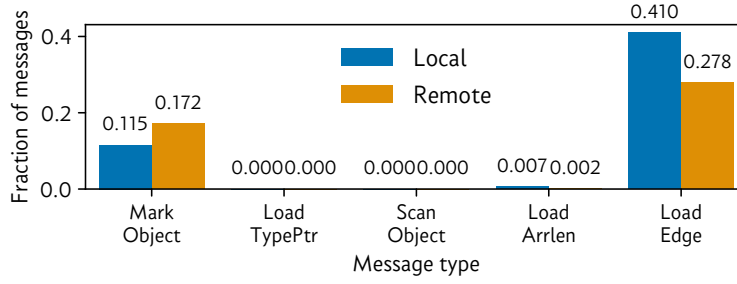


Figure 5.7: The fractions of remote and local messages for each type across all benchmarks.

for optimization.

5.5.2 Improving Object Scanning

I show above that LoadEdge messages are a key candidate for optimization. Remote LoadEdge messages occur because some of an object’s fields do not reside on the same DIMM that scans the object (Fig. 5.4 line 14). They are common because, when placed randomly, even a small 32 B object has a 50 % probability of straddling a cache line. Any object larger than 64 B is guaranteed to straddle at least one cache line.

Our key insight is that although objects often straddle cache lines, the reference fields within an object (the remaining are primitive fields) tend to stay together and form runs—groups of consecutive references. This clustering occurs because: i) programmers tend to put reference fields together when defining types, and ii) object arrays store all references together.

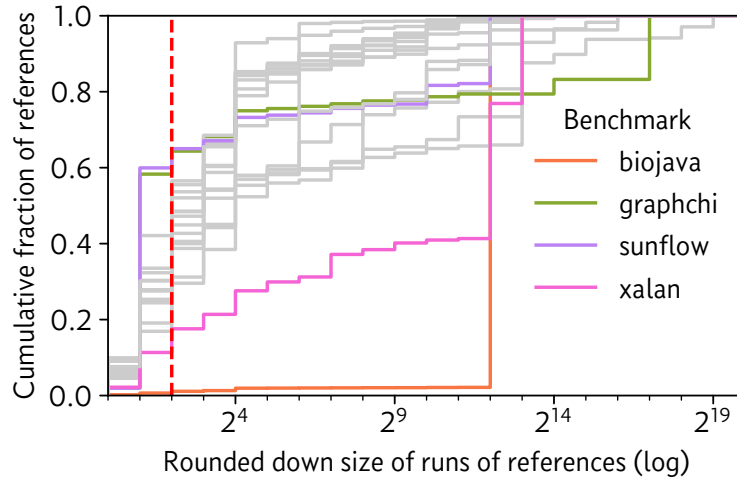


Figure 5.8: The distribution of lengths of contiguous runs of outgoing reference fields encountered during tracing.

These runs can be efficiently broadcast in a single message instead of requiring individual messages. Figure 5.8 shows an analysis of contiguous runs of references

within objects during tracing. For most benchmarks (Section 5.6.1), 60–80 % of outgoing references occur in runs of four or more, as the dashed line indicates. I now discuss the outliers.

biojava and xalan exhibit the highest fraction of references in long runs. biojava, a bioinformatics workload, encodes genetic sequences as arrays of nucleotide base objects. xalan stores XML elements in numerous `ObjectStack` objects, which are 4096-element reference arrays..

sunflow and graphchi show the opposite behavior. sunflow, a raytracing workload, primarily works with numerical values, and has few references. graphchi, a graph processing workload, assigns IDs to vertices and identifies neighbors using numerical IDs, thus also lacking long reference runs.

```

1  message ScanObject(obj, type_info_ptr) {
2    let type_info = *type_info_ptr;
3    - for i in 0..type_info.num_refs() {
4      -   off = 16 + i * 8
5      -   send LoadEdge(obj + off) to get_rank(obj + off);
6    + let num_refs = type_info.num_refs();
7    + for rank in get_ranks(obj + 16, num_refs) {
8      +   if rank == here // Eager loading
9      +     LoadEdges(obj + 16, num_refs);
10   +   else
11   +     send LoadEdges(obj + 16, num_refs) to rank;
12   } }
13 message LoadEdges(start, count) {
14   for edge in start..start + count {
15     // The ownership check can be fused for
16     // adjacent edges on the same rank
17     if get_rank(edge) == here {
18       let child = *edge;
19       send MarkObject(child) to get_rank(child);
20   } } }
```

Figure 5.9: Enhancing Fig. 5.4 with run-length encoding and eager loading.

This locality of reference fields suggests optimizing `LoadEdge` messages using run-length encoding (RLE). Figure 5.9 enhances our baseline protocol with a new message type, `LoadEdges`. Instead of sending one message per reference, a single `LoadEdges` message encodes an entire contiguous run. The message contains the starting address and the run length. Each recipient uses the system’s address mapping scheme (fixed for a particular machine) to compute which references are local. While some software GCs use RLE to store object arrays in the work queue (Section 5.4.1), we are the first to analyze and apply it to scalar (i.e., non-array) objects.

Based on this insight, I now introduce a second optimization: eager loading. Recall that the shape cache (Section 5.4.4) allows a GC unit to scan an object without sending a remote message on a cache hit. During a shape cache hit, if a reference field resides on the same rank as the object being scanned, we can process it immediately instead

of generating a local message to process it later. Because the field likely shares a cache line with the object header, prompt processing also improves the data cache hit rate. Although eager loading can change the order in which objects are visited, this does not change the correctness (only reachability matters), nor does it lead to any redundant work since any visited object can be skipped by checking its mark bit.

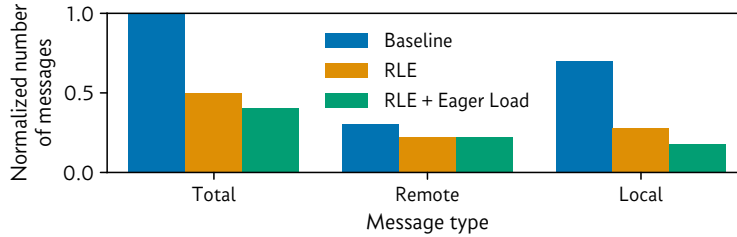


Figure 5.10: How our optimizations reduce messages compared to the baseline. RLE means run-length encoding, and eager loading means a GC unit tries to avoid sending a message to itself when scanning an object.

Figure 5.10 shows that these two optimizations substantially reduce the number of messages.

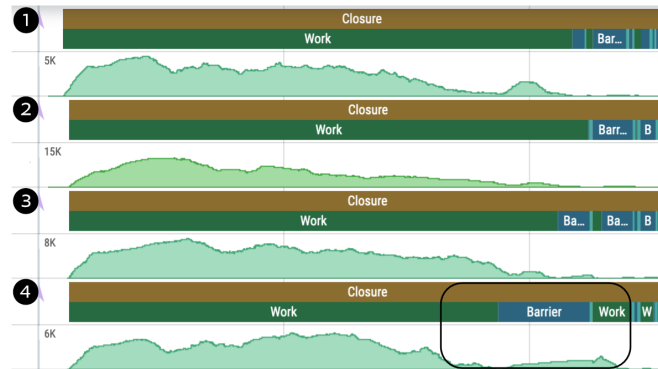
5.5.3 Improving Utilization via Better Distributed Termination Detection

We instrument our simulation using a tracing framework similar to the workflow I present in Chapter 4. Our baseline termination detection mechanism leads to poor utilization, as Fig. 5.11 illustrates using the fop benchmark. The figure shows an execution timeline for each GC unit, with two traces. Each upper trace depicts activity in a tree-like view, showing the overarching heap traversal (Closure in brown), which consists of productive work (Work in dark green), waiting for other GC units to finish their local work (Barrier in dark blue), and brief global termination checks (light blue). Each lower trace plots the number of items in its inbox (light green).

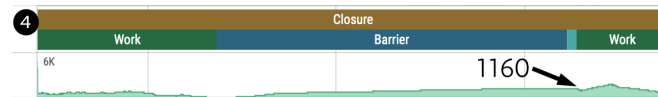
A unit that runs out of work immediately waits at a global barrier for all other units, limited by the slowest unit (Fig. 5.11(a)). Consequently, units idle while new work accumulates, wasting cycles; for example, unit 4 idles while over 1,000 items accumulate in its inbox (Fig. 5.11(b)).

We improve this behavior with optimistic spinning (Spin in magenta). When a unit finds no work, it spins for a short period before rechecking. We use exponential backoff to increase the spin time after each unsuccessful check. If it still finds no work after reaching a threshold, it then joins the global barrier.

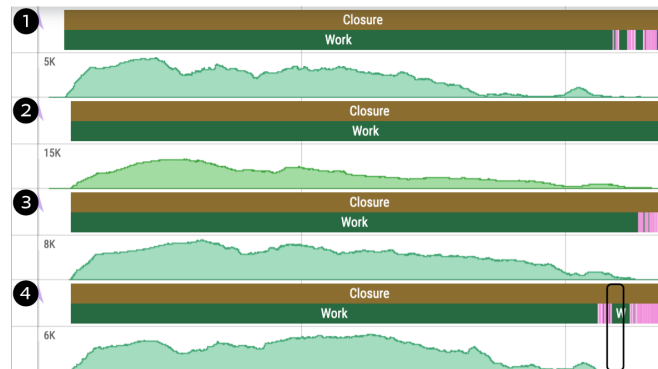
This change significantly improves utilization, as units spend more time in the Work state (Fig. 5.11(a) versus Fig. 5.11(c)). For example, the same unit 4 only enters the barrier when the entire system has little work left (Fig. 5.11(d)). Across all benchmarks, optimistic spinning improves utilization by 28 %. Although the figures show timelines of equal width for clarity, spinning improves utilization, thereby reducing the heap traversal’s wall-clock time.



(a) A GC invocation showing poor utilization, which is the fraction of Closure (brown) time spent in Work (dark green).



(b) Zooming in on a slice near the end of Fig. 5.11(a), messages accumulate while the processor idles, waiting for the barrier.



(c) A GC invocation showing good utilization.



(d) Zooming in on a very thin slice near the end of Fig. 5.11(c). Compared to Fig. 5.11(b), significantly fewer items remain in the queues when GC unit 4 resumes working.

Figure 5.11: GC execution traces from the fop benchmark showing that when the termination condition for distributed GCs is implemented naïvely, utilization can be poor. Each timeline shows two traces per unit: the upper indicating the GC's state, the lower a graph indicating total pending work at that unit.

5.6 Evaluation Methodology

5.6.1 Benchmarks and Runtime

We evaluate our design using heap snapshots from the DaCapo Chopin Java benchmarks [Blackburn et al., 2025]. We defer a full, end-to-end language runtime integration to future work (Section 5.8).

We capture the entire heap by running OpenJDK with the *full-heap* Immix collector from MMTk [MMTk Developers, 2020] instead of OpenJDK’s production collectors. This is because the production collectors are all *generational* or *regional*. We instrument Immix to serialize the heap to a file at each GC invocation. We run each program with a heap $3 \times$ its minimum required size. We exclude programs that did not trigger a GC with this heap size: batik, h2, tomcat, and tradebeans.

For hardware simulation feasibility (the simulation runs much slower than the target design), we sample 20 evenly spaced heap snapshots for each program. We measure our software baseline (Section 5.6.2) on the same set of snapshots.

5.6.2 Baseline

We compare our design with a multithreaded software tracing loop running on a contemporary CPU (configuration in Section 5.6.3). The baseline uses the work-packet technique, which is the state of the art in parallelizing heap traversal in high-performance software GCs [Zhao et al., 2025a]. Though our software baseline runs on a CPU with 16 hardware threads when contemporary CPUs have more cores, prior work suggests that additional threads yield diminishing returns [Barabash and Petrank, 2010; Zhao et al., 2025a], which we also reconfirm in Fig. 5.12.

We run the software baseline on real hardware instead of software simulators for fidelity. We run it for five iterations for each heap snapshot and only report the last iteration to make sure the microarchitectural states are warmed up.

We are unable to compare with the most closely related prior work [Maas et al., 2018; Jang et al., 2019] due to a lack of publicly available implementation, and reimplementation/replication studies are outside the scope of this thesis.

5.6.3 Hardware Configuration

We evaluate MAGC-DIMM primarily using a custom-built, cycle-level, event-driven simulator, which models a fixed memory configuration matching our baseline (Table 5.2). We built this simulator because no public tools model our target architecture: interconnected DIMMs with custom logic inside buffer devices. Prior work relied on proprietary hardware and simulators [Alian et al., 2018; Ke et al., 2022].

Table 5.2 shows configurations of our simulation and the physical hardware, where we run the software baseline.

Our simulator uses DRAMsim3 [Li et al., 2020] as the DRAM backend, and uses the 8 Gbit $\times$ 8 chip timing parameters supplied by DRAMsim3, which we then cross-check with the vendor datasheet [Micron Technology, Inc, 2023]. We model a 32-entry

Table 5.2: System configuration.

Host Processor	
Intel i9-9900K	8 cores, 2-way SMT
Core	Coffee Lake @ 3.6 GHz, 224 ROB
L1D\$	32 KiB, 8 ways, 64 sets
L2	256 KiB, unified, 4-way set associative
L1 DTLB	64 entries (4 KiB pages), 4-way set associative
L2 TLB	Unified, 1536 entries, 12-way set associative
LLC	Unified, 16 MiB, 16-way set associative
DDR Memory System	
DRAM	128 GiB total
Organization	2 channels, 2 DIMMs per channel, 2 ranks per DIMM 8 × 8 chips per rank, 16 Gibit in 16 banks per chip
Timing	DDR4-3200 22-22-22-22 (CL- t_{RCD} - t_{RP} - t_{RAS})
Addressing	row:rank:bank:channel:column (64 B contiguity)
MAGC-DIMM	1.6 GHz
Per rank D\$	32 KiB, blocking, 8 ways, 64 sets, LRU
Per rank TLB (2 MiB)	32 entries, 4-way set-associative, LRU
Per rank TLB (4 KiB)	64 entries, 4-way set-associative, LRU
Page table walker	average $24t_{\text{CK}}$ cost
DIMM-Link topology	Line (max. 3 hops between any two DIMMs)
Inter-DIMM latency	$4t_{\text{CK}}$ per hop + $2t_{\text{CK}}$ router ingress/egress
Inter-DIMM bandwidth	25 GiB/s per link over 8 pins

TLB, following prior work [Maas et al., 2018; Jang et al., 2019], and use a 2 MiB page size to reduce TLB misses. To explore the impact of different page sizes (Section 5.7), we also model a 64-entry 4 KiB-page TLB similar to our CPU baseline.

For the inter-DIMM interconnect, we model dedicated per-DIMM routing hardware (similar to DIMM-Link [Zhou et al., 2023]), allowing a GC unit to hand off messages to the router and resume local work without blocking. Messages are sent as pipelined chunks, with a per-hop latency of 4 cycles and an additional 2-cycle latency for a GC unit to communicate with the router. In Section 5.7, I further discuss the utilization of these inter-DIMM links.

Our simulator does not implement all optimizations from Section 5.5, such as run-length encoding and message-based prefetching. The performance we report is therefore conservative.

5.7 Evaluation

Previous sections explored a new paradigm for offloading garbage collection (GC) to DIMMs to improve its efficiency (Q₁ and Q₂). This section demonstrates that MAGC-DIMM achieves significant area efficiency while still achieving 68 % of the throughput of a highly tuned software baseline (Q₃).

5.7.1 Performance and Area Efficiency

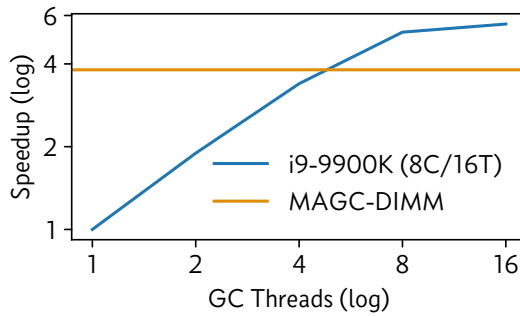


Figure 5.12: Heap traversal speedup compared with a software baseline running on CPU, normalized to single-threaded software. We report geometric means across benchmarks.

Our design requires $72 \times$ less cache, and each GC unit consumes negligible logic area because GC requires only simple arithmetic and memory operations. In fact, our design uses even less area than on-DIMM accelerators for other workloads (e.g., MCN uses 1 MB cache per DIMM [Alian et al., 2018]).

This key advantage in area efficiency is clear in Table 5.3, which compares MAGC-DIMM’s performance and area against a multithreaded software baseline. We conservatively estimate that the entire 8-rank MAGC-DIMM system uses approximately $0.22 \times$ the area of a single-core baseline (one core plus the shared LLC), while the 8-core software GC occupies approximately $2.52 \times$ —over $11 \times$ the area of our approach

Table 5-3: Performance and area comparison of MAGC-DIMM against the software baseline. All speedups and area estimates are normalized to single-threaded software (which uses a physical core and benefits from the entire shared LLC). Our baseline uses all 8 cores on an i9-9900K. MAGC-DIMM results are from our event-driven cycle-accurate simulator with 8 ranks. Cache and TLB hit rates are from across all ranks. Inter-DIMM link bandwidths are measured in GB/s. The rightmost column shows MAGC-DIMM with 4 KiB pages instead of 2 MiB.

Architecture (area vs. 1C) Benchmark	Software (8C), ~2.52x		MAGC-DIMM, ~0.22x				MAGC-DIMM (4 KiB), ~0.22x	
	Speedup	Speedup	Cache Hit	TLB Hit	Peak BW	Avg. BW	Speedup	TLB Hit
avroa	5.34	3.17	77.6%	100.0%	22.4	0.6	3.06	97.7%
biojava	7.24	1.02	93.4%	100.0%	32.0	0.7	1.02	99.8%
cassandra	5.92	5.66	75.6%	98.0%	25.6	0.3	5.33	94.5%
eclipse	6.10	4.25	71.9%	96.9%	25.6	0.4	3.87	90.5%
fop	5.30	3.69	70.7%	100.0%	25.6	0.3	3.39	94.4%
graphchi	4.70	3.76	69.0%	100.0%	22.4	0.3	3.66	98.1%
h2o	6.29	4.77	77.5%	99.7%	25.6	0.5	4.53	96.8%
jme	3.90	2.98	71.1%	100.0%	19.2	0.2	2.75	93.9%
jython	6.31	5.19	70.9%	99.9%	25.6	0.3	4.68	92.7%
kafka	5.92	5.24	71.8%	98.0%	22.4	0.2	4.81	92.1%
luindex	4.53	3.40	74.4%	99.9%	22.4	0.3	3.20	96.9%
lusearch	4.89	3.48	77.1%	100.0%	22.4	0.3	3.20	95.4%
pmd	7.40	5.61	80.0%	98.5%	25.6	0.3	5.32	96.3%
spring	6.75	5.16	71.6%	97.2%	25.6	0.3	4.70	90.8%
sunflow	5.45	3.07	72.0%	99.9%	25.6	0.4	2.99	98.1%
tradesoap	6.61	5.21	74.5%	97.3%	25.6	0.4	4.84	92.8%
xalan	7.49	3.68	81.5%	100.0%	25.6	0.8	3.56	98.2%
zxing	2.94	3.35	70.6%	99.0%	19.2	0.2	3.11	94.3%
min max mean geomean geomean/area	2.94	1.02	69.0%	96.9%	19.2	0.2	1.02	90.5%
	7.49	5.66	93.4%	100.0%	32.0	0.8	5.33	99.8%
	5.73	4.04	75.1%	99.1%	24.4	0.4	3.78	95.2%
	5.59	3.81	74.9%	99.1%	24.2	0.3	3.58	95.1%
	2.22	17.55					16.49	

(Section 5.4.4). MAGC-DIMM delivers roughly $8 \times$ more speedup per unit area than the software baseline across benchmarks.

In addition to this area advantage, MAGC-DIMM still achieves competitive overall throughput. Across benchmarks, the 16-thread software baseline achieves $5.59 \times$ speedup over a single thread, while MAGC-DIMM achieves $3.81 \times$ —68 % of the throughput of a high-end CPU running GC *exclusively* with all 8 cores. The software baseline achieves sublinear speedup with 16 threads as its performance gains diminish as parallelism approaches the physical core count (Fig. 5.12).

Most benchmarks show MAGC-DIMM within 50 % of the 16-thread baseline; the outlier is biojava (see also Section 5.5.2), whose heap contains a large reference array pointing to very few objects, causing severe load imbalance across ranks and yielding a speedup of only $1.02 \times$.

Next, we examine the cache and TLB behavior. The per-rank read cache hit rate ranges from 69 % to 93 %, with a geometric mean of 75 %, indicating that most GC accesses are served locally. The TLB achieves near-perfect hit rates (99.1 % geometric mean), confirming that 2 MiB pages effectively eliminate address translation overhead. With 4 KiB pages, the TLB hit rate drops to 95.1 % (geometric mean) as the smaller page size increases TLB demand; nonetheless, the hit rate remains above 90 % for all benchmarks. We also simulate the performance where the average PTW latency is 88 instead of 24 (Table 5.2). The impact of this additional latency on throughput is 2.6 %, not a significant step-change in the overall performance.

We also consider the inter-DIMM link bandwidth. Apart from biojava, peak bandwidth stays under 25 GB/s across benchmarks, which is the bandwidth proposed in the DIMM-Link design [Zhou et al., 2023]. The average utilization is low (mean 0.4 GB/s)—roughly 2 % of the link’s peak capacity—reflecting the bursty, short-lived nature of inter-rank communication. This low average utilization indicates that congestion is unlikely, and when it occurs, it is short-lived.

Finally, the rightmost columns show MAGC-DIMM with 4 KiB pages instead of 2 MiB pages. The geometric mean speedup decreases from $3.81 \times$ to $3.58 \times$ —a modest 6 % reduction—confirming that the design remains effective even without requiring the runtime to use 2 MiB pages.

5.7.2 Behavioral Correctness

To show that our design is implementable, we build a parameterizable RTL prototype on an FPGA. We modify a RISC-V Rocket Chip SoC [Asanović et al., 2016], adapting its core interconnect to model communication among DIMMs. Due to limitations of the SoC framework, the prototype uses single-channel DDR3-2133 memory instead of DDR4. We follow standard SoC parameters and best practices from prior work that used the same SoC [Maas et al., 2018].

The prototype establishes behavioral correctness and implementation feasibility. It uses in-order cores to implement the GC units. Its GC operations (Fig. 5.4) are implemented in software, allowing easy modification of the GC logic without altering the RTL design.

This RTL implementation is not timing-accurate (since these SoC components were not designed for our purpose), and as a result, we do not use the traditional synthesis workflow to obtain area/energy estimates. Importantly, however, the prototype gives us the agility to explore design parameters, not just because it is programmable, but also because of its execution speed. The prototype simulates a 1 GHz target design at 60 MHz using FPGA-accelerated simulation [Karandikar et al., 2019; Biancolin et al., 2019]. Compared with our event-driven simulator, this simulation speed enables detailed instrumentation, which informed the optimizations in Section 5.5. I now discuss two performance trends gleaned from this prototype: scalability and load imbalance.

5.7.3 MAGC-DIMM Scalability

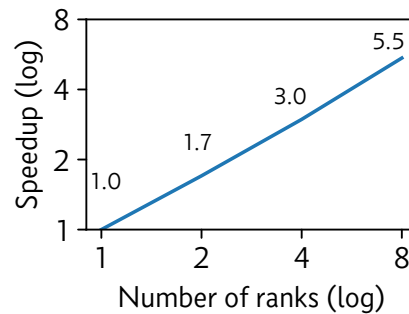


Figure 5.13: RTL prototype scalability across benchmarks.

We use our parameterizable RTL prototype to study scalability. Figure 5.13 shows that our design achieves near-linear scalability as we increase the number of memory ranks from one to eight. This result demonstrates the effectiveness of our optimizations in reducing communication and improving utilization (Section 5.5).

5.7.4 Impact of Load Imbalance

On-DIMM GC is subject to intrinsic load imbalance caused by the physical distribution of live objects across memory ranks. This effect is most prominent in applications with small heaps. As the heap size grows, a larger number of objects tend to be distributed more evenly across ranks, reducing the imbalance.

Figure 5.14 quantifies this work imbalance as the ratio between the busiest and most idle ranks. Benchmarks with the smallest heaps, such as *avro* at just 5 MB, exhibit the highest imbalance. This result is consistent with the intuition that larger heaps tend to be better balanced. This cause of imbalance—object placement—differs from that of multithreaded software GCs, where imbalance stems from heap shape, such as the inherently serial traversal of a long linked list [Barabash and Petrank, 2010].

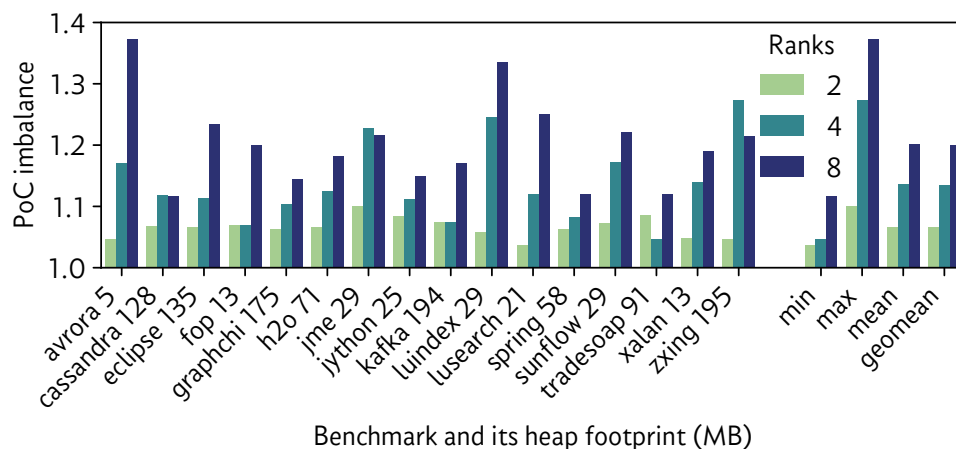


Figure 5.14: Load imbalance for the MAGC-DIMM prototype. Each benchmark’s minimum heap size is indicated.

5.8 Future Extensions

I now sketch how one can extend our design for integration into production systems.

Other GC Phases In this chapter, I focus on identifying live objects through pointer chasing; MAGC-DIMM performs this entirely on DIMMs via a distributed GC protocol. Other GC activities, such as reclaiming dead objects, can also benefit from the internal memory bandwidth of near-memory processing. Once live objects are identified, a linear scan of the heap can then identify dead objects (unmarked ones), which is embarrassingly parallel. However, language runtimes track free memory in software on the host side for performance. Therefore, one can extend the host–GC interface to allow the GC units to return memory to the host as tuples of starting addresses and sizes. Another extension is to allow GC units to reserve free memory from the host. This allows the GC to move objects to reduce fragmentation. The amount of memory to reserve can be determined ahead of time based on the current maximum heap size. Alternatively, the GC can reserve memory in small batches, and request more from the host when it exhausts its current supply.

Concurrent Tracing GC Race conditions arise if the program modifies the heap while the GC traverses it. As I discuss in [Sections 2.2.3](#) and [2.2.4](#), this problem is well understood in software GC literature, and is commonly solved using write barriers. Recall that write barriers are short code snippets inserted by the compiler to monitor a program’s pointer writes. Canonical concurrent tracing algorithms, such as Snapshot at the Beginning [Yuasa, 1990], use them.

To support concurrent traversal in MAGC-DIMM, we can monitor the program’s pointer writes in two ways: i) extending the host–GC interface to allow software barriers to send pointer writes to GC units, or ii) implementing the barrier in hardware,

since the buffer device’s position allows monitoring of the host’s cache line write-backs.

Operating System Integration To differentiate among processes, the host needs to pass the current process’s page table base address along with its roots when triggering a GC. When multiple processes share memory pages (e.g., after forking), the operating system must clone copy-on-write (CoW) pages before invoking a GC if writable CoW pages exist. Handling virtualization and nested page tables requires enhancing the on-DIMM page table walker.

Generality Language runtimes can and do evolve over time. A recent example is OpenJDK Lilliput [OpenJDK, 2024], which tightens the in-memory layout of object headers that encode object metadata. To prevent a hardwired design from becoming obsolete, a production version of our design needs to support some programmability. We demonstrate this possibility by using small general-purpose cores in our RTL prototype (Section 5.7.2).

When implemented using general-purpose cores, MAGC-DIMM can also be used to offload other workloads that are arithmetically simple but memory bottlenecked. For managed languages, one example is object or page zeroing [?]. Outside of language runtimes, possible uses include other popular targets of near-memory acceleration, such as database operations, and embedding lookups.

5.9 Summary

In this chapter, I presented MAGC-DIMM, a distributed on-DIMM garbage collection (GC) accelerator that performs GC inside DIMM buffer devices. Our design handles extreme cases of memory interleaving, where each DIMM can access as little as one cache line of *contiguous* memory addresses. I show that despite this challenge, on-DIMM GC can be efficient through optimizations exploiting GC’s special properties to reduce communication and synchronization, and to improve utilization. Our design uses significantly less cache and logic area than software GC. Through simulation, I show that our design still achieves 68 % of the throughput of a 16-hardware-thread, high-end CPU running GC *exclusively*. I demonstrate the scalability and flexibility of our approach through an RTL prototype. MAGC-DIMM is a first step toward future systems where GC, and the memory safety it provides, are delivered by the memory system—without burdening the CPU.

Discussion

This thesis makes three major contributions to building and evaluating garbage collectors (GCs) across the computational stack—software, kernel, and hardware. In this chapter, I reflect on this journey, and discuss ongoing challenges and research efforts in our community that are relevant to this thesis.

6.1 Rethinking Specialized Systems

Early computers often relied on specialized hardware—only mainframes could provide the stability and performance required for enterprise workloads. An example is the IBM Z, which remains in use today.

The industry then shifted toward commodity hardware. Commodity hardware offered flexibility, allowing companies to choose from many vendors. For a long time, Dennard scaling provided performance improvement with each feature size shrink, disincentivizing the construction of specialized hardware. Google famously pioneered the use of commodity components to build large-scale datacenters [Google, 1999], which used clever software algorithms to compensate for the reduced reliability of individual machines [Barroso et al., 2018].

Specialized systems were still developed afterwards, such as the Azul Systems Vega appliance optimized for Java, but they were the exception [Azul, 2008]. Most of the world’s computing power came from systems built with commodity parts, although server-grade components offered better features and performance.

However, the end of Dennard scaling led to a reinvigoration of specialized hardware. More recently, two major trends have driven this change: the ubiquity of mobile devices, and the continued rise of hyperscale datacenters. Energy efficiency is a primary concern for both, and the slowdown of Moore’s law and Dennard scaling compounds this problem. Simultaneously, a few key workload classes now dominate computing. These include machine learning, Android runtime/Swift for mobile, client-side JavaScript, and server-side runtimes, such as JVM, .NET Core, and Go.

Workload consolidation and the demand for efficiency create a significant opportunity for specialized systems. In fact, with workload consolidation, specialized hardware can now have broader utility. For example, machine learning accelerators (such as Apple’s Neural Engine or Google’s TPU) are now deployed on devices ranging from

low-power mobile phones to beefy datacenter servers. Similarly, hardware support for managed runtimes might also have wide benefits [Maas, 2020].

However, unlike machine learning workloads, managed runtimes have largely remained bound to general-purpose CPUs for now. As I show in Chapter 3, GC incurs substantial overheads for applications running on such runtimes. This thesis presents a cross-stack approach to GC optimization, which includes a specialized hardware accelerator for GC—MAGC-DIMM. The results presented in this thesis demonstrate the opportunity to reap substantial benefits by designing and optimizing systems across the entire stack, despite the development costs and integration efforts.

Crucially, the shifting economics of GC also allow for the reconsideration of opportunities and ideas that have been discarded as “impractical.” The consequences of such innovations might extend far beyond improving the speed of existing applications; they could lead to a much wider uptake of managed languages, and potentially enable new classes of algorithms. Recent work reaffirms this position [Maas et al., 2024; Yang et al., 2025].

6.2 Collective Investment in Research Infrastructure

Shared research infrastructure provides many benefits. It allows researchers to build confidence in each other’s results. It enables us to reuse prior work instead of reinventing it. It also helps us prototype new ideas quickly.

A good example of such infrastructure is the RISC-V ecosystem for CPU and system-on-a-chip (SoC) designs. Open-source frameworks, such as Chippyard [Amid et al., 2020], Rocket Chip [Asanović et al., 2016], and FireSim [Karandikar et al., 2019], enable rapid prototyping that accelerates innovation. In fact, the MAGC-DIMM prototype presented in Chapter 5 is built on top of these frameworks.

On the other hand, a failure to invest in such infrastructure creates significant risk. A promising idea can become bogged down by the sheer engineering effort required just to build a working prototype. This discourages ambitious, trailblazing work, and biases the research community toward incremental improvements to the status quo, as those approaches often yield the best short-term progress [Zhao et al., 2025b].

A key part of this thesis is building an on-DIMM GC accelerator—MAGC-DIMM. Section 5.7.2 leverages the Rocket Chip RISC-V SoC framework to prototype MAGC-DIMM for functional correctness, and implementability. However, this prototype is no substitute for cycle-accurate end-to-end simulators, and reference RTL designs that faithfully model the underlying interconnected near-memory-processing DIMMs. As a result, modeling and analyzing MAGC-DIMM performance required significantly more effort than it would have taken if the requisite infrastructure were in place. Therefore, this is a call for our community to collectively invest in shared research infrastructure to accelerate progress in emerging architectures, such as near-memory processing.

Conclusion

This thesis takes a cross-stack approach to optimize garbage collection (GC). As the dominant approach to memory safety, GC needs to continue to evolve to lower the barrier for adoption, and to make existing garbage-collected software more efficient. At the start of the thesis, I identify three problems: i) an absence of necessary methodology to measure the real cost of GC, and worse still, ignorance of this absence, ii) difficulties in performance debugging and optimization, and iii) impractical near-memory acceleration for GC. I then address each of these problems.

First, I present the Lower Bound Overhead (LBO) methodology. LBO helps us measure hard-to-attribute overheads of low-level components. When applied to GC, LBO reveals significant overheads of software GCs. What is interesting and concerning is not just these overheads, but that they have largely gone unnoticed until now. This lack of awareness is likely due to our field's collective methodological inattention. LBO is a response to that inattention, and adds another tool to our field's methodological toolkit. Though alarming, the GC overheads revealed by LBO also highlight new opportunities for improvement, such as architectural support for GC.

Second, I show how kernel-based performance tracing can add another invaluable tool to our field's methodological toolkit. Using GC as an example, I demonstrate that tracing can enhance the observability of low-level systems software, especially when combined with a modular design. Tracing aids in debugging and optimizing existing systems. It also provides the rapid feedback necessary to generate insights for designing future systems.

Finally, I present a new approach to near-memory acceleration for managed languages. MAGC-DIMM—an on-DIMM accelerator—represents a first step toward near-memory GC that is suitable for integration into general-purpose systems. Despite the architectural challenges, such as the interleaving of cache lines across DIMMs, on-DIMM GC can be efficient—using $72\times$ less cache, and significantly less logic area than the software baseline, while achieving 68% throughput.

These outcomes are evidence that a cross-stack approach can bring significant benefits even to a field as extensively studied as GC. This thesis encourages the community to rethink system design, evaluation, and optimization as we face emerging systems challenges that may benefit from a similar cross-stack approach.

7.1 Future Work

7.1.1 Microarchitectural GC Analysis with Better Delineation

In [Chapter 3](#), I use the wall-clock time overhead, and the CPU cycle overhead as examples of applying the lower bound overhead (LBO) methodology. Since LBO is metric-agnostic, it can be applied to other metrics, including microarchitectural metrics like cache misses.

Indeed, [Huang \[2025\]](#)’s recent work on analyzing GC’s cache behavior demonstrates that there are more insights to glean from such analysis. For example, her work challenges the conventional wisdom that GC exhibits poor locality, showing instead that the reality is more nuanced.

In fact, the tracing infrastructure presented in [Chapter 4](#) can complement LBO analysis by providing better delineation of GC activities, and thus better estimation of the intrinsic application costs. The case studies in [Chapter 3](#) only attribute stop-the-world collection activities to GC. While this attribution is sound, tracepoints might enable more fine-grained attribution and analysis. This might in turn allow us to better analyze concurrent GCs (especially when all available GCs provided by a VM are concurrent), and also allow us to examine the cost incurred by different phases within GC.

7.1.2 User-Space Tracing

[Chapter 4](#) uses eBPF—a kernel-based tracing framework, and its command line tooling, such as BCC and bpftool. These tools are mature and ergonomic, and allow us to quickly evaluate the feasibility of integrating tracing with GC. Furthermore, these tools run tracing logic in kernel mode, which is convenient if one wants to correlate GC activities with kernel activities (such as virtual memory management).

However, running tracing logic in kernel mode incurs the cost of privilege level switches from user space to kernel, which can be significant for frequently invoked tracepoints. Therefore, it is worth investigating executing tracing logic in user space when kernel information is not needed.

One possible extension is to tightly integrate user-space tracing into the system being studied. For example, a memory management framework, such as MMTk, can provide infrastructure for GC implementers and users to write tracing logic in a domain-specific language, which can then be just-in-time (JIT) compiled. The run-time patching of the tracepoints in this case is done by MMTk itself in user space instead of by the kernel. However, JIT and run-time code patching are notoriously difficult to get right, and one needs to weigh the benefit of providing a more tailored integration against the engineering effort.

Another possibility is to leverage existing user-space tracing frameworks. In [Section 4.3.1](#), I discussed tools such as LTTng and Perfetto, which have their respective limitations. More recently, bpftime provides a user-space tracing API similar to kernel-based eBPF tracing [[Zheng et al., 2025](#)]. This might be a more promising target to investigate.

7.1.3 End-to-End Integration and Evaluation of MAGC-DIMM

Chapter 5 presents the design and evaluation of MAGC-DIMM, representing a first step toward on-DIMM hardware acceleration for managed languages. Evolving the design for use in real-world systems requires more end-to-end integration and evaluation.

In Section 5.8, I enumerate possible extensions to MAGC-DIMM to make it suitable for integration with production-grade language runtimes. Note that this integration challenge is not unique to a hardware accelerator; a research software GC design also needs to undergo the same process to be feature-complete to handle real-world workloads [Zhao et al., 2025b].

To better explore the design space of MAGC-DIMM, one needs to extend the event-driven simulator to model additional design parameters, including DIMM topology, address mapping schemes, and virtual address translation. In particular, since MAGC-DIMM distributes GC tasks across interconnected DIMMs, a fixed topology (such as the line topology) will not scale to datacenter servers with many DIMMs.

To better gauge the performance of MAGC-DIMM, one needs to evaluate it together with the software stack it supports. This likely involves extending existing full-system simulators or system-on-a-chip frameworks to allow implementing hardware designs both in the CPU and on DIMMs, and simulating them simultaneously with high fidelity. Note that the common way of simulating a near-memory design using instruction/memory traces often overlooks offloading overheads and micro-architectural side effects.

If the end-to-end simulation results prove fruitful, the ultimate goal is to tape out silicon, and build hardware harnesses to test the design, which can then encourage productization and adoption in real-world hardware.

Bibliography

- AHN, J.; HONG, S.; YOO, S.; MUTLU, O.; AND CHOI, K., 2015. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, 105–117. ACM. doi:10.1145/2749469.2750386. <https://doi.org/10.1145/2749469.2750386>. (cited on pages 69, 74, and 82)
- ALIAN, M.; MIN, S.; ASGHARIMOGHADDAM, H.; DHAR, A.; WANG, D. K.; ROEWER, T.; MCPADDEN, A. J.; O'HALLORAN, O.; CHEN, D.; XIONG, J.; KIM, D.; HWU, W. W.; AND KIM, N. S., 2018. Application-transparent near-memory processing architecture with memory channel network. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, 802–814. IEEE Computer Society. doi:10.1109/MICRO.2018.00070. <https://doi.org/10.1109/MICRO.2018.00070>. (cited on pages 76, 79, 88, and 90)
- AMID, A.; BIANCOLIN, D.; GONZALEZ, A.; GRUBB, D.; KARANDIKAR, S.; LIEW, H.; MAGYAR, A.; MAO, H.; OU, A. J.; PEMBERTON, N.; RIGGE, P.; SCHMIDT, C.; WRIGHT, J. C.; ZHAO, J.; SHAO, Y. S.; ASANOVIC, K.; AND NIKOLIC, B., 2020. Chipyard: Integrated design, simulation, and implementation framework for custom SoCs. *IEEE Micro*, 40, 4 (2020), 10–21. doi:10.1109/MM.2020.2996616. <https://doi.org/10.1109/MM.2020.2996616>. (cited on page 98)
- ANDROID OPEN SOURCE PROJECT, 2017a. Overview of system tracing | Android developers. <https://developer.android.com/topic/performance/tracing>. <http://web.archive.org/web/20221221134831/https://developer.android.com/topic/performance/tracing>. Accessed: 2022-12-21. (cited on page 50)
- ANDROID OPEN SOURCE PROJECT, 2017b. Perfetto. <https://perfetto.dev/>. <http://web.archive.org/web/20230216013847/https://perfetto.dev/>. Accessed: 2023-02-16. (cited on page 49)
- APPLE INC., 2022. GCGC : Garbage collection graph collector. <https://github.com/apple/GCGC>. <http://web.archive.org/web/20230102224957/https://github.com/apple/GCGC>. Accessed: 2023-01-02. (cited on page 43)
- ARM LIMITED, 2008. Jazelle direct bytecode execution support. <http://web.archive.org/web/20230102202151/https://developer.arm.com/documentation/ddi0406/b/Application-Level-Architecture/Application-Level-Programmers--Model/Execution-environment-support/Jazelle-direct-bytecode-execution-support?lang=en>. <https://developer.arm.com/documentation/ddi0406/b/Application-Level-Architecture/Application-Level-Programmers--Model/Execution-environment-support/Jazelle-direct-bytecode-execution-support?lang=en>

Architecture/Application-Level-Programmers--Model/Execution-environment-support/Jazelle-direct-bytecode-execution-support?lang=en. Accessed: 2023-01-02. (cited on pages 2 and 72)

ARNOLD, M. AND GROVE, D., 2005. Collecting and exploiting high-accuracy call graph profiles in virtual machines. In *3rd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005)*, 20-23 March 2005, San Jose, CA, USA, 51–62. IEEE Computer Society. doi:10.1109/CGO.2005.9. <https://doi.org/10.1109/CGO.2005.9>. (cited on page 45)

ASANOVIĆ, K.; AVIZIENIS, R.; BACHRACH, J.; BEAMER, S.; BIANCOLIN, D.; CELIO, C.; COOK, H.; DABBELT, D.; HAUSER, J.; IZRAELEVITZ, A.; KARANDIKAR, S.; KELLER, B.; KIM, D.; KOENIG, J.; LEE, Y.; LOVE, E.; MAAS, M.; MAGYAR, A.; MAO, H.; MORETO, M.; OU, A.; PATTERSON, D. A.; RICHARDS, B.; SCHMIDT, C.; TWIGG, S.; VO, H.; AND WATERMAN, A., 2016. The Rocket Chip generator. Technical Report UCB/EECS-2016-17, EECS Department, University of California, Berkeley. <http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.html>. (cited on pages 92 and 98)

AZUL, 2008. Azul Systems® extends leadership in business-critical Java™ applications performance with the new Vega series. <https://www.azul.com/newsroom/azul-systems-extends-leadership-in-business-critical-java-applications-performance-with-the-new-vega-series/>. <https://web.archive.org/web/20251007061839/https://www.azul.com/newsroom/azul-systems-extends-leadership-in-business-critical-java-applications-performance-with-the-new-vega-series/>. Accessed: 2025-10-07. (cited on page 97)

BACON, D. F.; CHENG, P.; AND SHUKLA, S., 2012. And then there were none: a stall-free real-time garbage collector for reconfigurable hardware. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, 23–34. ACM. doi:10.1145/2254064.2254068. <https://doi.org/10.1145/2254064.2254068>. (cited on page 72)

BACON, D. F.; CHENG, P.; AND SHUKLA, S., 2014. Parallel real-time garbage collection of multiple heaps in reconfigurable hardware. In *International Symposium on Memory Management, ISMM '14, Edinburgh, United Kingdom, June 12, 2014*, 117–127. ACM. doi:10.1145/2602988.2602996. <https://doi.org/10.1145/2602988.2602996>. (cited on page 72)

BARABASH, K. AND PETRANK, E., 2010. Tracing garbage collection on highly parallel platforms. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario, Canada, June 5-6, 2010*, 1–10. ACM. doi:10.1145/1806651.1806653. <https://doi.org/10.1145/1806651.1806653>. (cited on pages 10, 42, 58, 63, 75, 88, and 93)

BARKER, M.; EDWARDS, S. A.; AND KIM, M. A., 2022a. Synthesized garbage collection for FPGA accelerators. In *FPGA '22: The 2022 ACM/SIGDA International Symposium on*

-
- Field-Programmable Gate Arrays, Virtual Event, USA, 27 February 2022 - 1 March 2022*, 53. ACM. doi:10.1145/3490422.3502341. <https://doi.org/10.1145/3490422.3502341>. (cited on pages 72 and 73)
- BARKER, M.; EDWARDS, S. A.; AND KIM, M. A., 2022b. Synthesized in-BRAM garbage collection for accelerators with immutable memory. In *32nd International Conference on Field-Programmable Logic and Applications, FPL 2022, Belfast, United Kingdom, August 29 - Sept. 2, 2022*, 47–53. IEEE. doi:10.1109/FPL57034.2022.00019. <https://doi.org/10.1109/FPL57034.2022.00019>. (cited on pages 72 and 73)
- BARROSO, L. A.; HÖLZLE, U.; AND RANGANATHAN, P., 2018. *The Datacenter as a Computer: Designing Warehouse-Scale Machines, Third Edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers. ISBN 978-3-031-00633-3. doi:10.2200/S00874ED3V01Y201809CAC046. <https://doi.org/10.2200/S00874ED3V01Y201809CAC046>. (cited on page 97)
- BEAMER, S.; ASANOVIC, K.; AND PATTERSON, D. A., 2015. Locality exists in graph processing: Workload characterization on an Ivy Bridge server. In *2015 IEEE International Symposium on Workload Characterization, IISWC 2015, Atlanta, GA, USA, October 4-6, 2015*, 56–65. IEEE Computer Society. doi:10.1109/IISWC.2015.12. <https://doi.org/10.1109/IISWC.2015.12>. (cited on pages 11 and 68)
- BIANCOLIN, D.; KARANDIKAR, S.; KIM, D.; KOENIG, J.; WATERMAN, A.; BACHRACH, J.; AND ASANOVIC, K., 2019. FASED: FPGA-accelerated simulation and evaluation of DRAM. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, FPGA 2019, Seaside, CA, USA, February 24-26, 2019*, 330–339. ACM. doi:10.1145/3289602.3293894. <https://doi.org/10.1145/3289602.3293894>. (cited on page 93)
- BLACKBURN, S. M.; CAI, Z.; CHEN, R.; YANG, X.; ZHANG, J.; AND ZIGMAN, J., 2024. ASPLOS 2025 artifact: Rethinking Java performance analysis. doi:10.5281/zenodo.12682890. <https://doi.org/10.5281/zenodo.12682890>. (cited on pages 13 and 24)
- BLACKBURN, S. M.; CAI, Z.; CHEN, R.; YANG, X.; ZHANG, J.; AND ZIGMAN, J. N., 2025. Rethinking Java performance analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2025, Rotterdam, The Netherlands, 30 March 2025 - 3 April 2025*, 940–954. ACM. doi:10.1145/3669940.3707217. <https://doi.org/10.1145/3669940.3707217>. (cited on pages ix, 13, 24, 55, 72, and 88)
- BLACKBURN, S. M.; CHENG, P.; AND MCKINLEY, K. S., 2004a. Myths and realities: the performance impact of garbage collection. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, 25–36. ACM. doi:10.1145/1005686.1005693. <https://doi.org/10.1145/1005686.1005693>. (cited on pages 2, 8, 14, 16, 17, 34, 42, and 54)

-
- BLACKBURN, S. M.; CHENG, P.; AND MCKINLEY, K. S., 2004b. Oil and water? high performance garbage collection in Java with MMTk. In *26th International Conference on Software Engineering (ICSE 2004)*, 23-28 May 2004, Edinburgh, United Kingdom, 137–146. IEEE Computer Society. doi:10.1109/ICSE.2004.1317436. <https://doi.org/10.1109/ICSE.2004.1317436>. (cited on pages 7, 51, and 54)
- BLACKBURN, S. M.; DIWAN, A.; HAUSWIRTH, M.; SWEENEY, P. F.; AMARAL, J. N.; BRECHT, T.; BULEJ, L.; CLICK, C.; EECKHOUT, L.; FISCHMEISTER, S.; FRAMPTON, D.; HENDREN, L. J.; HIND, M.; HOSKING, A. L.; JONES, R. E.; KALIBERA, T.; KEYNES, N.; NYSTROM, N.; AND ZELLER, A., 2016. The truth, the whole truth, and nothing but the truth: A pragmatic guide to assessing empirical evaluations. *ACM Trans. Program. Lang. Syst.*, 38, 4 (2016), 15:1–15:20. <http://dl.acm.org/citation.cfm?id=2983574>. (cited on page 15)
- BLACKBURN, S. M.; GARNER, R.; HOFFMANN, C.; KHAN, A. M.; MCKINLEY, K. S.; BENTZUR, R.; DIWAN, A.; FEINBERG, D.; FRAMPTON, D.; GUYER, S. Z.; HIRZEL, M.; HOSKING, A. L.; JUMP, M.; LEE, H. B.; MOSS, J. E. B.; PHANSALKAR, A.; STEFANOVIC, D.; VANDRUNEN, T.; VON DINCKLAGE, D.; AND WIEDERMANN, B., 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, 169–190. ACM. doi:10.1145/1167473.1167488. <https://doi.org/10.1145/1167473.1167488>. (cited on pages 2, 14, 25, and 42)
- BLACKBURN, S. M. AND HOSKING, A. L., 2004. Barriers: friend or foe? In *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, 143–151. ACM. doi:10.1145/1029873.1029891. <https://doi.org/10.1145/1029873.1029891>. (cited on pages 2, 10, 14, 17, and 42)
- BLACKBURN, S. M. AND MCKINLEY, K. S., 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, 22–32. ACM. doi:10.1145/1375581.1375586. <https://doi.org/10.1145/1375581.1375586>. (cited on pages 7, 9, 17, and 55)
- BLACKBURN, S. M.; MCKINLEY, K. S.; GARNER, R.; HOFFMANN, C.; KHAN, A. M.; BENTZUR, R.; DIWAN, A.; FEINBERG, D.; FRAMPTON, D.; GUYER, S. Z.; HIRZEL, M.; HOSKING, A. L.; JUMP, M.; LEE, H.; MOSS, J. E. B.; PHANSALKAR, A.; STEFANOVIC, D.; VANDRUNEN, T.; VON DINCKLAGE, D.; AND WIEDERMANN, B., 2008. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM*, 51, 8 (2008), 83–89. doi:10.1145/1378704.1378723. <https://doi.org/10.1145/1378704.1378723>. (cited on pages 2, 14, 34, and 42)
- BOEHM, H., 2000. Reducing garbage collector cache misses. In *ISMM 2000, International Symposium on Memory Management, Minneapolis, Minnesota, USA, October 15-16, 2000 (in conjunction with OOPSLA 2000)*, Conference Proceedings, 59–64. ACM.

- [doi:10.1145/362422.362438](https://doi.org/10.1145/362422.362438). <https://doi.org/10.1145/362422.362438>. (cited on page 11)
- BURGER, D. AND AUSTIN, T. M., 1997. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25, 3 (1997), 13–25. [doi:10.1145/268806.268810](https://doi.org/10.1145/268806.268810). <https://doi.org/10.1145/268806.268810>. (cited on page 16)
- CAI, Z., 2022. ISPASS 2022 artifact: Distilling the real cost of production garbage collectors. [doi:10.5281/zenodo.6476821](https://doi.org/10.5281/zenodo.6476821). <https://doi.org/10.5281/zenodo.6476821>. (cited on page 13)
- CAI, Z.; BLACKBURN, S. M.; BOND, M. D.; AND MAAS, M., 2022. Distilling the real cost of production garbage collectors. In *International IEEE Symposium on Performance Analysis of Systems and Software, ISPASS 2022, Singapore, May 22-24, 2022*, 46–57. IEEE. [doi:10.1109/ISPASS55109.2022.00005](https://doi.org/10.1109/ISPASS55109.2022.00005). <https://doi.org/10.1109/ISPASS55109.2022.00005>. (cited on pages ix, 13, and 24)
- CAI, Z.; BLACKBURN, S. M.; BOND, M. D.; AND MAAS, M., 2025. MAGC-DIMM: A near-memory processing garbage collection accelerator on DIMMs. (2025). Under submission. (cited on pages ix and 68)
- CAO, T.; BLACKBURN, S. M.; GAO, T.; AND MCKINLEY, K. S., 2012. The Yin and Yang of power and performance for asymmetric hardware and managed software. In *39th International Symposium on Computer Architecture (ISCA 2012), June 9-13, 2012, Portland, OR, USA*, 225–236. IEEE Computer Society. [doi:10.1109/ISCA.2012.6237020](https://doi.org/10.1109/ISCA.2012.6237020). <https://doi.org/10.1109/ISCA.2012.6237020>. (cited on page 11)
- CHENEY, C. J., 1970. A nonrecursive list compacting algorithm. *Commun. ACM*, 13, 11 (1970), 677–678. [doi:10.1145/362790.362798](https://doi.org/10.1145/362790.362798). <https://doi.org/10.1145/362790.362798>. (cited on pages 7 and 23)
- CHENG, P. AND BLELLOCH, G. E., 2001. A parallel, real-time garbage collector. In *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, 125–136. ACM. [doi:10.1145/378795.378823](https://doi.org/10.1145/378795.378823). <https://doi.org/10.1145/378795.378823>. (cited on pages 16, 37, 38, and 39)
- CHO, B. Y.; KWON, Y.; LYM, S.; AND EREZ, M., 2020. Near data acceleration with concurrent host access. In *47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Virtual Event / Valencia, Spain, May 30 - June 3, 2020*, 818–831. IEEE. [doi:10.1109/ISCA45697.2020.00072](https://doi.org/10.1109/ISCA45697.2020.00072). <https://doi.org/10.1109/ISCA45697.2020.00072>. (cited on pages 69 and 81)
- CLICK, C.; TENE, G.; AND WOLF, M., 2005. The Pauseless GC algorithm. In *Proceedings of the 1st International Conference on Virtual Execution Environments, VEE 2005, Chicago, IL, USA, June 11-12, 2005*, 46–56. ACM. [doi:10.1145/1064979.1064988](https://doi.org/10.1145/1064979.1064988). <https://doi.org/10.1145/1064979.1064988>. (cited on page 3)

-
- COLLINS, G. E., 1960. A method for overlapping and erasure of lists. *Commun. ACM*, 3, 12 (1960), 655–657. doi:10.1145/367487.367501. <https://doi.org/10.1145/367487.367501>. (cited on page 8)
- CONG, J.; FANG, Z.; GILL, M.; JAVADI, F.; AND REINMAN, G., 2017. AIM: accelerating computational genomics through scalable and noninvasive accelerator-interposed memory. In *Proceedings of the International Symposium on Memory Systems, MEMSYS 2017, Alexandria, VA, USA, October 02 - 05, 2017*, 3–14. ACM. doi:10.1145/3132402.3132406. <https://doi.org/10.1145/3132402.3132406>. (cited on page 76)
- DACAPO DEVELOPERS, 2021. DaCapo benchmarks evaluation snapshot 29a657f. doi:10.5281/zenodo.6475255. <https://doi.org/10.5281/zenodo.6475255>. (cited on page 24)
- DACAPO DEVELOPERS, 2022. DaCapo benchmarks evaluation snapshot 6e411f33. <https://github.com/dacapobench/dacapobench/commit/6e411f33cd8d19f9928eb36c10ceeb868d9b6b23>. <https://github.com/dacapobench/dacapobench/commit/6e411f33cd8d19f9928eb36c10ceeb868d9b6b23>. Accessed: 2022-11-25. (cited on page 55)
- DESNOYERS, M. AND DAGENAIS, M., 2008. LTTng: Tracing across execution layers, from the hypervisor to user-space. In *Proceedings of the Linux Symposium—Volume One, Ottawa, Ontario, Canada, July 23-26, 2008*, 101–106. <http://web.archive.org/web/20221227045309/https://www.kernel.org/doc/ols/2008/ols2008v1-pages-101-106.pdf>. (cited on page 49)
- DETLEFS, D.; FLOOD, C. H.; HELLER, S.; AND PRINTEZIS, T., 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, 37–48. ACM. doi:10.1145/1029873.1029879. <https://doi.org/10.1145/1029873.1029879>. (cited on pages 3, 9, 13, 19, and 28)
- DIJKSTRA, E. W.; LAMPORT, L.; MARTIN, A. J.; SCHOLTEN, C. S.; AND STEFFENS, E. F. M., 1978. On-the-fly garbage collection: An exercise in cooperation. *Commun. ACM*, 21, 11 (1978), 966–975. doi:10.1145/359642.359655. <https://doi.org/10.1145/359642.359655>. (cited on page 11)
- EBPF FOUNDATION, 2021. Extended berkeley packet filter. <https://ebpf.io/>. <https://ebpf.io/>. Accessed: 2022-11-25. (cited on page 47)
- FENICHEL, R. AND YOCHELSON, J. C., 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12, 11 (1969), 611–612. doi:10.1145/363269.363280. <https://doi.org/10.1145/363269.363280>. (cited on pages 7 and 23)
- FLOOD, C. H.; KENNKE, R.; DINN, A.; HALEY, A.; AND WESTRELIN, R., 2016. 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, Lugano, Switzerland, August 29 - September 2, 2016, 13:1–13:9. ACM. doi:10.1145/2972206.2972210. <https://doi.org/10.1145/2972206.2972210>. (cited on pages 3, 15, 19, and 37)
- GAGNON, E. M. AND HENDREN, L. J., 2001. SableVM: A research framework for the efficient execution of Java bytecode. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium, April 23-24, 2001, Monterey, CA, USA*, 27–40. USENIX. <http://www.usenix.org/publications/library/proceedings/jvm01/gagnon.html>. (cited on page 83)
- GARCÍA, A. A.; MAY, D.; AND NUTTING, E., 2021. Integrated hardware garbage collection. *ACM Trans. Embed. Comput. Syst.*, 20, 5 (2021), 40:1–40:25. doi:10.1145/3450147. <https://doi.org/10.1145/3450147>. (cited on pages 3, 72, and 73)
- GARNER, R.; BLACKBURN, S. M.; AND FRAMPTON, D., 2007. Effective prefetch for mark-sweep garbage collection. In *Proceedings of the 6th International Symposium on Memory Management, ISMM 2007, Montreal, Quebec, Canada, October 21-22, 2007*, 43–54. ACM. doi:10.1145/1296907.1296915. <https://doi.org/10.1145/1296907.1296915>. (cited on page 11)
- GARNER, R.; BLACKBURN, S. M.; AND FRAMPTON, D., 2011. A comprehensive evaluation of object scanning techniques. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, 33–42. ACM. doi:10.1145/1993478.1993484. <https://doi.org/10.1145/1993478.1993484>. (cited on pages 80 and 83)
- GÓMEZ-LUNA, J.; HAJJ, I. E.; FERNANDEZ, I.; GIANNOULA, C.; OLIVEIRA, G. F.; AND MUTLU, O., 2022. Benchmarking a new paradigm: Experimental analysis and characterization of a real processing-in-memory system. *IEEE Access*, 10 (2022), 52565–52608. doi:10.1109/ACCESS.2022.3174101. <https://doi.org/10.1109/ACCESS.2022.3174101>. (cited on pages 68 and 76)
- GOOGLE, 1999. Google Stanford hardware. <https://web.archive.org/web/19990209043945/google.stanford.edu/googlehardware.html>. <http://google.stanford.edu/googlehardware.html>. (cited on page 97)
- GOOGLE, 2016. Trace event format. <https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6lOnSsKchNAySU/preview>. <http://web.archive.org/web/20230127150401/https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6lOnSsKchNAySU/preview>. Accessed: 2023-01-27. (cited on page 53)
- GREGG, B., 2019. *BPF Performance Tools*. Pearson Education, Philadelphia, PA. (cited on page 47)
- HADIDI, R.; ASGARI, B.; MUDASSAR, B. A.; MUKHOPADHYAY, S.; YALAMANCHILI, S.; AND KIM, H., 2017. Demystifying the characteristics of 3D-stacked memories: A

-
- case study for Hybrid Memory Cube. In *2017 IEEE International Symposium on Workload Characterization, IISWC 2017, Seattle, WA, USA, October 1-3, 2017*, 66–75. IEEE Computer Society. doi:10.1109/IISWC.2017.8167757. <https://doi.org/10.1109/IISWC.2017.8167757>. (cited on page 74)
- HERTZ, M. AND BERGER, E. D., 2005. 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 2005, October 16-20, 2005, San Diego, CA, USA*, 313–326. ACM. doi:10.1145/1094811.1094836. <https://doi.org/10.1145/1094811.1094836>. (cited on pages 14, 16, and 18)
- HOSKING, A. L.; MOSS, J. E. B.; AND STEFANOVIC, D., 1992. A comparative performance evaluation of write barrier implementations. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92), Seventh Annual Conference, Vancouver, British Columbia, Canada, October 18-22, 1992, Proceedings*, 92–109. ACM. doi:10.1145/141936.141946. <https://doi.org/10.1145/141936.141946>. (cited on pages 2, 10, 14, 17, and 42)
- HOUND TECHNOLOGY, INC., 2018. honeycomb.io guide: Achieving observability. <https://www.honeycomb.io/wp-content/uploads/2018/07/Honeycomb-Guide-Achieving-Observability-v1.pdf>. <http://web.archive.org/web/20220903112906/https://www.honeycomb.io/wp-content/uploads/2018/07/Honeycomb-Guide-Achieving-Observability-v1.pdf>. Accessed: 2022-09-03. (cited on pages 2, 41, and 42)
- HUANG, C., 2025. *High Performance Garbage Collection: A Microarchitectural Perspective*. Bachelor's thesis, Australian National University. <https://web.archive.org/web/20250710083913/https://www.steveblackburn.org/pubs/theses/huang-2025.pdf>. (cited on pages 11 and 100)
- HUANG, C.; BLACKBURN, S. M.; AND CAI, Z., 2023. Improving garbage collection observability with performance tracing. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes, MPLR 2023, Cascais, Portugal, 22 October 2023*, 85–99. ACM. doi:10.1145/3617651.3622986. <https://doi.org/10.1145/3617651.3622986>. (cited on pages ix and 41)
- HUANG, X.; BLACKBURN, S. M.; MCKINLEY, K. S.; MOSS, J. E. B.; WANG, Z.; AND CHENG, P., 2004. 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 2004, October 24-28, 2004, Vancouver, BC, Canada*, 69–80. ACM. doi:10.1145/1028976.1028983. <https://doi.org/10.1145/1028976.1028983>. (cited on pages 2, 14, 18, and 83)
- HUANG, X.; MOSS, J. E. B.; MCKINLEY, K. S.; BLACKBURN, S. M.; AND BURGER, D., 2003. Dynamic SimpleScalar: Simulating Java Virtual Machines. Technical Report TR03-03, Department of Computer Sciences, The University of Texas

-
- at Austin. https://web.archive.org/web/20240201000000*/https://www.cs.utexas.edu/ftp/techreports/tr03-03.pdf. (cited on page 16)
- INTEL CORPORATION, 2023a. Fix performance bottlenecks with Intel® VTune™ profiler. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>. <http://web.archive.org/web/20230623012318/https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>. Accessed: 2023-06-23. (cited on pages 42 and 44)
- INTEL CORPORATION, 2023b. Intel® VTune™ profiler user guide: Sampling interval. <https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-1/sampling-interval.html>. <http://web.archive.org/web/20230627040259/https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-1/sampling-interval.html>. Accessed: 2023-06-27. (cited on page 45)
- IO VISOR PROJECT, 2015. BCC. <https://github.com/iovisor/bcc>. <https://github.com/iovisor/bcc>. Accessed: 2022-11-25. (cited on page 47)
- IO VISOR PROJECT, 2018. bpftrace. <https://github.com/iovisor/bpftrace>. <https://github.com/iovisor/bpftrace>. Accessed: 2022-11-25. (cited on page 47)
- JANG, J.; HEO, J.; LEE, Y.; WON, J.; KIM, S.; JUNG, S.; JANG, H.; HAM, T. J.; AND LEE, J. W., 2019. Charon: Specialized near-memory processing architecture for clearing dead objects in memory. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*, 726–739. ACM. doi:10.1145/3352460.3358297. <https://doi.org/10.1145/3352460.3358297>. (cited on pages 3, 72, 73, 74, 80, 88, and 90)
- JOAO, J. A.; MUTLU, O.; AND PATT, Y. N., 2009. Flexible reference-counting-based hardware acceleration for garbage collection. In *36th International Symposium on Computer Architecture (ISCA 2009), June 20-24, 2009, Austin, TX, USA*, 418–428. ACM. doi:10.1145/1555754.1555806. <https://doi.org/10.1145/1555754.1555806>. (cited on page 3)
- JOHANSSON, S., 2021. GC progress from JDK 8 to JDK 17. <https://kstefanj.github.io/2021/11/24/gc-progress-8-17.html>. <https://web.archive.org/web/20250403092812/https://kstefanj.github.io/2021/11/24/gc-progress-8-17.html>. Accessed: 2025-04-23. (cited on page 2)
- JONES, R. E.; HOSKING, A. L.; AND MOSS, J. E. B., 2011. *The Garbage Collection Handbook: The art of automatic memory management*. Chapman and Hall / CRC Applied Algorithms and Data Structures Series. CRC Press. ISBN 978-1-4200-8279-1. <http://gchandbook.org/>. (cited on pages 7 and 76)
- JULIA DEVELOPERS, 2011. Instrumenting Julia with DTrace, and bpftrace. <https://docs.julialang.org/en/v1/devdocs/probes/>. <http://web.archive.org/web/20230120124737/https://docs.julialang.org/en/v1/devdocs/probes/>. Accessed: 2023-01-20. (cited on page 49)

-
- KARANDIKAR, S.; MAO, H.; KIM, D.; BIANCOLIN, D.; AMID, A.; LEE, D.; PEMBERTON, N.; AMARO, E.; SCHMIDT, C.; CHOPRA, A.; HUANG, Q.; KOVACS, K.; NIKOLIC, B.; KATZ, R. H.; BACHRACH, J.; AND ASANOVIC, K., 2019. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. *IEEE Micro*, 39, 3 (2019), 56–65. doi:10.1109/MM.2019.2910175. <https://doi.org/10.1109/MM.2019.2910175>. (cited on pages 93 and 98)
- KE, L.; ZHANG, X.; SO, J.; LEE, J.; KANG, S.; LEE, S.; HAN, S.; CHO, Y.; KIM, J. H.; KWON, Y.; KIM, K.; JUNG, J.; YUN, I.; PARK, S. J.; PARK, H.; SONG, J.; CHO, J.; SOHN, K.; KIM, N. S.; AND LEE, H. S., 2022. Near-memory processing in action: Accelerating personalized recommendation with AxDIMM. *IEEE Micro*, 42, 1 (2022), 116–127. doi:10.1109/MM.2021.3097700. <https://doi.org/10.1109/MM.2021.3097700>. (cited on pages 68, 76, and 88)
- KENNKE, R., 2021. Shenandoah in OpenJDK 17: Sub-millisecond GC pauses. <https://developers.redhat.com/articles/2021/09/16/shenandoah-openjdk-17-sub-millisecond-gc-pauses>. <http://web.archive.org/web/20211207114248/https://developers.redhat.com/articles/2021/09/16/shenandoah-openjdk-17-sub-millisecond-gc-pauses>. Accessed: 2021-12-07. (cited on page 2)
- KWON, Y.; LEE, S. H.; LEE, J.; KWON, S.; RYU, J.; SON, J.; O, S.; YU, H.; LEE, H.; KIM, S. Y.; CHO, Y.; KIM, J. G.; CHOI, J.; SHIN, H.; KIM, J.; PHUAH, B.; KIM, H.; SONG, M. J.; CHOI, A.; KIM, D.; KIM, S.; KIM, E.; WANG, D.; KANG, S.; RO, Y.; SEO, S.; SONG, J.; YOUN, J.; SOHN, K.; AND KIM, N. S., 2021. A 2onm 6GB Function-In-Memory DRAM, based on HBM2 with a 1.2TFLOPS programmable computing unit using bank-level parallelism, for machine learning applications. In *IEEE International Solid-State Circuits Conference, ISSCC 2021, San Francisco, CA, USA, February 13-22, 2021*, 350–352. IEEE. doi:10.1109/ISSCC42613.2021.9365862. <https://doi.org/10.1109/ISSCC42613.2021.9365862>. (cited on pages 68 and 76)
- LI, S.; YANG, Z.; REDDY, D.; SRIVASTAVA, A.; AND JACOB, B. L., 2020. Dramsim3: A cycle-accurate, thermal-capable DRAM simulator. *IEEE Comput. Archit. Lett.*, 19, 2 (2020), 110–113. doi:10.1109/LCA.2020.2973991. <https://doi.org/10.1109/LCA.2020.2973991>. (cited on page 88)
- LIDÉN, P., 2021. ZGC | what’s new in JDK 17. <https://malloc.se/blog/zgc-jdk17>. <http://web.archive.org/web/20211115113228/https://malloc.se/blog/zgc-jdk17>. Accessed: 2021-12-07. (cited on page 2)
- LIDÉN, P. AND KARLSSON, S., 2018. The Z garbage collector. <http://cr.openjdk.java.net/~pliden/slides/ZGC-FOSDEM-2018.pdf>. <http://web.archive.org/web/20211207112406/http://cr.openjdk.java.net/~pliden/slides/ZGC-FOSDEM-2018.pdf>. Accessed: 2021-12-07. (cited on pages 3, 15, 19, and 37)
- LIN, Y., 2022. mmtk-core Pull Request #614: Minimize the scope of acquire_lock. <https://github.com/mmtk/mmtk-core/pull/614>. <https://github.com/mmtk/mmtk-core/pull/614>. Accessed: 2022-11-25. (cited on page 64)

-
- LORD, B., 2023. The urgent need for memory safety in software products. <https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products>. <https://web.archive.org/web/20250423005750/https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products>. Accessed: 2025-04-23. (cited on page 1)
- LUU, D., 2016. Sampling v. tracing. <https://danluu.com/perf-tracing/>. Accessed: 2023-05-23. (cited on pages 41, 42, and 43)
- MAAS, M., 2020. Hardware support for managed languages: an old idea whose time has finally come? (keynote). In *MPLR '20: 17th International Conference on Managed Programming Languages and Runtimes, Virtual Event, UK, November 4-6, 2020*, 2. ACM. doi:10.1145/3426182.3431580. <https://doi.org/10.1145/3426182.3431580>. (cited on page 98)
- MAAS, M.; ANDERSEN, D. G.; ISARD, M.; JAVANMARD, M. M.; MCKINLEY, K. S.; AND RAFFEL, C., 2024. Combining machine learning and lifetime-based resource management for memory allocation and beyond. *Commun. ACM*, 67, 4 (2024), 87–96. doi:10.1145/3611018. <https://doi.org/10.1145/3611018>. (cited on page 98)
- MAAS, M.; ASANOVIC, K.; AND KUBIATOWICZ, J., 2018. A hardware accelerator for tracing garbage collection. In *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, 138–151. IEEE Computer Society. doi:10.1109/ISCA.2018.00022. <https://doi.org/10.1109/ISCA.2018.00022>. (cited on pages 11, 68, 72, 73, 74, 80, 88, 90, and 92)
- MALCOLM, D. AND LANGA, L., 2016. Instrumenting CPython with DTrace and System-Tap. <https://docs.python.org/3/howto/instrumentation.html>. <http://web.archive.org/web/20230227184647/https://docs.python.org/3/howto/instrumentation.html>. Accessed: 2023-02-27. (cited on page 49)
- MCCARTHY, J., 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Commun. ACM*, 3, 4 (1960), 184–195. doi:10.1145/367177.367199. <https://doi.org/10.1145/367177.367199>. (cited on page 7)
- MICRON TECHNOLOGY, INC., 2023. DDR4 SDRAM, Rev. U edn. https://web.archive.org/web/20251029061949/https://www.mouser.com/datasheet/2/671/8gb_ddr4_sdram-3178010.pdf. (cited on page 88)
- MICROSOFT CORPORATION, 2016. PerfView and TraceEvent. <https://github.com/microsoft/perfview/blob/main/src/TraceEvent/Computers/TraceManagedProcess.cs>. <http://web.archive.org/web/20221205191156/https://github.com/microsoft/perfview/blob/main/src/TraceEvent/Computers/TraceManagedProcess.cs>. Accessed: 2022-12-05. (cited on page 50)
- MICROSOFT CORPORATION, 2019. We need a safer systems programming language. <https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming->

-
- language/. <https://web.archive.org/web/20250413235336/https://msrc.microsoft.com/blog/2019/07/we-need-a-safer-systems-programming-language/>. Accessed: 2025-04-13. (cited on page 1)
- MICROSOFT CORPORATION, 2022. Threads view in the concurrency visualizer. <https://learn.microsoft.com/en-us/visualstudio/profiling/threads-view-parallel-performance?view=vs-2022>. <http://web.archive.org/web/20230601064429/https://learn.microsoft.com/en-us/visualstudio/profiling/threads-view-parallel-performance?view=vs-2022>. Accessed: 2023-06-01. (cited on page 46)
- MICROSOFT CORPORAION, 2021. Microsoft GCToolKit. <https://devblogs.microsoft.com/java/introducing-microsoft-gctoolkit/>. <http://web.archive.org/web/20221003152322/https://devblogs.microsoft.com/java/introducing-microsoft-gctoolkit/>. Accessed: 2022-10-03. (cited on page 43)
- MMTK DEVELOPERS, 2020. MMTk: A high performance language-independent memory management framework. <https://www.mmtk.io/>. <https://www.mmtk.io/>. Accessed: 2022-11-25. (cited on pages 51, 54, and 88)
- MORIMOTO, R.; ICHINOSE, K.; AND UGAWA, T., 2025. Gray-in-young: A generational garbage collection for processing-in-memory. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management, ISMM 2025, Seoul, Republic of Korea, 17 June 2025*, 122–133. ACM. doi:10.1145/3735950.3735961. <https://doi.org/10.1145/3735950.3735961>. (cited on pages 72 and 73)
- MYTKOWICZ, T.; DIWAN, A.; HAUSWIRTH, M.; AND SWEENEY, P. F., 2010. Evaluating the accuracy of Java profilers. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, 187–197. ACM. doi:10.1145/1806596.1806618. <https://doi.org/10.1145/1806596.1806618>. (cited on page 45)
- OPENJDK, 2024. Project Lilliput. <https://wiki.openjdk.org/display/lilliput>. (cited on page 95)
- OPENJDK DEVELOPERS, 2023. Work stealing deque implementation in OpenJDK. <https://github.com/openjdk/jdk/blob/jdk-21%2B35/src/hotspot/share/gc/shared/taskqueue.hpp#L296-L325>. (cited on page 10)
- OPROFILE DEVELOPERS, 2023. About OProfile. <https://oprofile.sourceforge.io/about/>. <https://web.archive.org/web/20230629065218/https://oprofile.sourceforge.io/about/>. Accessed: 2023-06-29. (cited on page 44)
- ORACLE, 2021a. Java platform, standard edition HotSpot virtual machine garbage collection tuning guide, release 17. <https://docs.oracle.com/en/java/javase/17/gctuning/hotspot-virtual-machine-garbage-collection-tuning-guide.pdf>. <http://web.archive.org/web/20211213041034/https://docs.oracle.com/en/java/javase/17/gctuning/hotspot-virtual-machine-garbage-collection-tuning-guide.pdf>. Accessed: 2021-12-13. (cited on pages 2 and 37)

-
- ORACLE, 2021b. JVM™ tool interface. <https://docs.oracle.com/en/java/javase/17/docs/specs/jvmti.html>. <https://web.archive.org/web/20211209121820/https://docs.oracle.com/en/java/javase/17/docs/specs/jvmti.html>. Accessed: 2021-12-09. (cited on pages 23 and 25)
- ORACLE CORPORATION AND OPENJDK DEVELOPERS, 2013. JDK mission control. www.oracle.com/java/technologies/jdk-mission-control.html. <http://web.archive.org/web/20230219111918/https://www.oracle.com/java/technologies/jdk-mission-control.html>. Accessed: 2023-02-19. (cited on page 50)
- PRASAD, V.; COHEN, W.; EIGLER, F. C.; HUNT, M.; KENISTON, J.; AND CHEN, B., 2005. Locating system problems using dynamic instrumentation. In *Proceedings of the Linux Symposium—Volume Two, Ottawa, Ontario, Canada, July 20-23, 2005*, 49–64. <http://web.archive.org/web/20220308130912/https://www.kernel.org/doc/ols/2005/ols2005v2-pages-57-72.pdf>. (cited on page 47)
- PUFFITSCH, W., 2011. Hard real-time garbage collection for a java chip multi-processor. In *The 9th International Workshop on Java Technologies for Real-time and Embedded Systems, JTRES '11, York, United Kingdom, September 26-28, 2011*, 64–73. ACM. doi:10.1145/2043910.2043921. <https://doi.org/10.1145/2043910.2043921>. (cited on pages 2, 72, and 73)
- RAMSAY, C. AND STEWART, R. J., 2024. Cloaca: A concurrent hardware garbage collector for non-strict functional languages. In *Proceedings of the 17th ACM SIGPLAN International Haskell Symposium, Haskell 2024, Milan, Italy, September 6-7, 2024*, 41–54. ACM. doi:10.1145/3677999.3678277. <https://doi.org/10.1145/3677999.3678277>. (cited on pages 2, 72, and 73)
- RUBY DEVELOPERS, 2013. DTrace probes - RDoc documentation. https://ruby-doc.org/3.2.1/dtrace_probes_rdoc.html. http://web.archive.org/web/20230303115753/https://ruby-doc.org/3.2.1/dtrace_probes_rdoc.html. Accessed: 2023-03-03. (cited on page 49)
- SAREEN, K. AND BLACKBURN, S. M., 2022. Better understanding the costs and benefits of automatic memory management. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes, MPLR 2022, Brussels, Belgium, September 14-15, 2022*, 29–44. ACM. doi:10.1145/3546918.3546926. <https://doi.org/10.1145/3546918.3546926>. (cited on pages 2 and 80)
- SAREEN, K.; BLACKBURN, S. M.; HAMOUDA, S. S.; AND GIDRA, L., 2024. Memory management on mobile devices. In *Proceedings of the 2024 ACM SIGPLAN International Symposium on Memory Management, ISMM 2024, Copenhagen, Denmark, 25 June 2024*, 15–29. ACM. doi:10.1145/3652024.3665510. <https://doi.org/10.1145/3652024.3665510>. (cited on page 40)

-
- SHAHRIYAR, R.; BLACKBURN, S. M.; AND FRAMPTON, D., 2012. Down for the count? getting reference counting back in the ring. In *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, 73–84. ACM. doi:10.1145/2258996.2259008. <https://doi.org/10.1145/2258996.2259008>. (cited on pages 8 and 75)
- SHAHRIYAR, R.; BLACKBURN, S. M.; AND MCKINLEY, K. S., 2014. Fast conservative garbage collection. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, 121–139. ACM. doi:10.1145/2660193.2660198. <https://doi.org/10.1145/2660193.2660198>. (cited on page 8)
- SHAHRIYAR, R.; BLACKBURN, S. M.; YANG, X.; AND MCKINLEY, K. S., 2013. Taking off the gloves with reference counting Immix. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, 93–110. ACM. doi:10.1145/2509136.2509527. <https://doi.org/10.1145/2509136.2509527>. (cited on page 8)
- SITES, R. L., 2015. Data center computers: Modern challenges in CPU design. <https://www.youtube.com/watch?v=QBu2Ae8-8LM>. Accessed: 2023-05-23. (cited on page 41)
- SITES, R. L., 2021. *Understanding Software Dynamics*. Addison-Wesley. (cited on pages 41, 44, and 45)
- SITES, R. L., 2023. KUtrace. <https://github.com/dicksites/kutrace>. Accessed: 2023-05-23. (cited on page 41)
- STEELE JR., G. L., 1975. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18, 9 (1975), 495–508. doi:10.1145/361002.361005. <https://doi.org/10.1145/361002.361005>. (cited on page 11)
- STEELE JR., G. L. AND SUSSMAN, G. J., 1980. Design of a LISP-based microprocessor. *Commun. ACM*, 23, 11 (1980), 628–645. doi:10.1145/359024.359031. <https://doi.org/10.1145/359024.359031>. (cited on pages 2, 72, and 73)
- STEPHENS, M., 2021. GCRealTimeMon. <https://github.com/MaoniO/realmon>. <http://web.archive.org/web/20221205193717/https://github.com/maoniO/realmon>. Accessed: 2022-12-05. (cited on pages 23 and 50)
- STOEP, J. V. AND ZHANG, C., 2019. Queue the hardening enhancements. <https://security.googleblog.com/2019/05/queue-hardening-enhancements.html>. <https://web.archive.org/web/20250427174635/https://security.googleblog.com/2019/05/queue-hardening-enhancements.html>. Accessed: 2025-04-27. (cited on page 1)

-
- STONE, J., 2014. libprobe: Static probes for Rust. <https://crates.io/crates/probe>. <http://web.archive.org/web/20230303073513/https://crates.io/crates/probe>. Accessed: 2023-03-03. (cited on page 47)
- STROUSTRUP, B., 2021. C++—an invisible foundation of everything. *ACCU Overload*, 161 (2021). <https://web.archive.org/web/20210215063236/https://www.stroustrup.com/invisible.pdf>. Accessed: 2021-02-15. (cited on page 2)
- SUN, W.; LI, Z.; YIN, S.; WEI, S.; AND LIU, L., 2021. ABC-DIMM: alleviating the bottleneck of communication in DIMM-based near-memory processing with inter-DIMM broadcast. In *48th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2021, Virtual Event / Valencia, Spain, June 14-18, 2021*, 237–250. IEEE. doi:10.1109/ISCA52012.2021.00027. <https://doi.org/10.1109/ISCA52012.2021.00027>. (cited on pages 69 and 76)
- SUN MICROSYSTEMS, 2006. DTrace user guide. <https://docs.oracle.com/cd/E19253-01/819-5488/819-5488.pdf>. <http://web.archive.org/web/20230303065912/https://docs.oracle.com/cd/E19253-01/819-5488/819-5488.pdf>. Accessed: 2023-03-03. (cited on page 47)
- TANG, J.; LIU, C.; AND GAUDIOT, J., 2015. How can garbage collection be energy efficient by dynamic offloading? In *26th IEEE International Conference on Application-specific Systems, Architectures and Processors, ASAP 2015, Toronto, ON, Canada, July 27-29, 2015*, 156–157. IEEE Computer Society. doi:10.1109/ASAP.2015.7245725. <https://doi.org/10.1109/ASAP.2015.7245725>. (cited on pages 3, 72, and 73)
- TENE, G.; IYENGAR, B.; AND WOLF, M., 2011. C4: the continuously concurrent compacting collector. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, 79–88. ACM. doi:10.1145/1993478.1993491. <https://doi.org/10.1145/1993478.1993491>. (cited on page 19)
- THOMAS, S.; CHOE, J.; GORDON, O.; PETRANK, E.; MORESHET, T.; HERLIHY, M.; AND BAHAR, R. I., 2022. Towards hardware accelerated garbage collection with near-memory processing. In *IEEE High Performance Extreme Computing Conference, HPEC 2022, Waltham, MA, USA, September 19-23, 2022*, 1–6. IEEE. doi:10.1109/HPEC55821.2022.9926323. <https://doi.org/10.1109/HPEC55821.2022.9926323>. (cited on pages 3, 72, and 73)
- TIER1APP, 2022. GCeasy. <https://gceasy.io/>. <http://web.archive.org/web/20230227155647/https://gceasy.io/>. Accessed: 2023-02-27. (cited on page 43)
- TSAI, P.; GAN, Y. L.; AND SÁNCHEZ, D., 2018. Rethinking the memory hierarchy for modern languages. In *51st Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2018, Fukuoka, Japan, October 20-24, 2018*, 203–216. IEEE Computer Society. doi:10.1109/MICRO.2018.00025. <https://doi.org/10.1109/MICRO.2018.00025>. (cited on page 71)

-
- TSAI, P. AND SÁNCHEZ, D., 2019. Compress objects, not cache lines: An object-based compressed memory hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, 229–242. ACM. doi:10.1145/3297858.3304006. <https://doi.org/10.1145/3297858.3304006>. (cited on page 71)
- UNGAR, D. M., 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*, 157–167. ACM. doi:10.1145/800020.808261. <https://doi.org/10.1145/800020.808261>. (cited on pages 9 and 10)
- UNGAR, D. M.; BLAU, R.; FOLEY, P.; SAMPLES, A. D.; AND PATTERSON, D. A., 1984. Architecture of SOAR: Smalltalk on a RISC. In *Proceedings of the 11th Annual Symposium on Computer Architecture, Ann Arbor, USA, June 1984*, 188–197. ACM. doi:10.1145/800015.808182. <https://doi.org/10.1145/800015.808182>. (cited on pages 2 and 72)
- UNITED STATES CYBERSECURITY AND INFRASTRUCTURE SECURITY AGENCY; UNITED STATES NATIONAL SECURITY AGENCY; UNITED STATES FEDERAL BUREAU OF INVESTIGATION; AUSTRALIAN SIGNALS DIRECTORATE’S AUSTRALIAN CYBER SECURITY CENTRE; CANADIAN CENTRE FOR CYBER SECURITY; UNITED KINGDOM NATIONAL CYBER SECURITY CENTRE; NEW ZEALAND NATIONAL CYBER SECURITY CENTRE; AND COMPUTER EMERGENCY RESPONSE TEAM NEW ZEALAND, 2023. The case for memory safe roadmaps. <https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products>. <https://web.archive.org/web/20250403140304/https://www.cisa.gov/sites/default/files/2023-12/The-Case-for-Memory-Safe-Roadmaps-508c.pdf>. Accessed: 2025-04-23. (cited on page 1)
- V8 DEVELOPERS, 2018. Tracing V8. <https://v8.dev/docs/trace>. <http://web.archive.org/web/20230216012529/https://v8.dev/docs/trace>. Accessed: 2023-02-16. (cited on page 50)
- WANG, K., 2023. eBPF and Perfetto UI proved me wrong. <https://wks.github.io/blog/2023/10/04/proved-me-wrong.html>. <https://web.archive.org/web/20250204022849/https://wks.github.io/blog/2023/10/04/proved-me-wrong.html>. Accessed: 2025-02-04. (cited on page 65)
- WIKICHIP, 2025. Skylake (client) - microarchitectures - Intel. [https://web.archive.org/web/20250213052954/https://en.wikichip.org/wiki/intel/microarchitectures/skylake_\(client\)](https://web.archive.org/web/20250213052954/https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)). [https://en.wikichip.org/wiki/intel/microarchitectures/skylake_\(client\)](https://en.wikichip.org/wiki/intel/microarchitectures/skylake_(client)). Accessed: 2025-02-13. (cited on page 79)
- WILLIAMS, I. AND WOLCZKO, M., 1990. An object-based memory architecture. In *Implementing Persistent Object Bases, Principles and Practice, Proceedings of the Fourth In-*

-
- ternational Workshop on Persistent Objects*, 23-27 September 1990, Martha's Vineyard, MA, USA, 114-130. Morgan Kaufmann. (cited on pages 2 and 72)
- WU, H.; JI, Z.; ZHU, S.; AND CHEN, Z., 2013. A performance study of software prefetching for tracing garbage collectors. In *Advanced Parallel Processing Technologies - 10th International Symposium, APPT 2013, Stockholm, Sweden, August 27-28, 2013, Revised Selected Papers*, vol. 8299 of *Lecture Notes in Computer Science*, 160-169. Springer. doi:10.1007/978-3-642-45293-2_12. https://doi.org/10.1007/978-3-642-45293-2_12. (cited on page 11)
- WU, M.; LI, Z.; CHEN, H.; ZANG, B.; WANG, S.; YU, L.; LI, S.; AND SONG, H., 2024. Toward an SGX-friendly Java runtime. *IEEE Trans. Computers*, 73, 1 (2024), 44-57. doi:10.1109/TC.2023.3318400. <https://doi.org/10.1109/TC.2023.3318400>. (cited on page 82)
- YANG, C.; LI, Y.; MAAS, M.; UYSAL, M.; HAFEEZ, U. U.; MERCHANT, A.; AND MCDUGALL, R., 2025. A practical cross-layer approach for ML-driven storage placement in warehouse-scale computers. *CoRR*, abs/2501.05651 (2025). doi:10.48550/ARXIV.2501.05651. <https://doi.org/10.48550/arXiv.2501.05651>. (cited on page 98)
- YANG, X.; BLACKBURN, S. M.; FRAMPTON, D.; AND HOSKING, A. L., 2012. Barriers reconsidered, friendlier still! In *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, 37-48. ACM. doi:10.1145/2258996.2259004. <https://doi.org/10.1145/2258996.2259004>. (cited on pages 2, 10, 14, 17, and 42)
- YANG, X.; BLACKBURN, S. M.; AND MCKINLEY, K. S., 2015. Computer performance microscopy with Shim. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, 170-184. ACM. doi:10.1145/2749469.2750401. <https://doi.org/10.1145/2749469.2750401>. (cited on pages 42, 43, and 45)
- YUASA, T., 1990. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.*, 11, 3 (1990), 181-198. doi:10.1016/0164-1212(90)90084-Y. [https://doi.org/10.1016/0164-1212\(90\)90084-Y](https://doi.org/10.1016/0164-1212(90)90084-Y). (cited on pages 11, 19, and 94)
- ZHAO, W. AND BLACKBURN, S. M., 2020. Deconstructing the garbage-first collector. In *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020*, 15-29. ACM. doi:10.1145/3381052.3381320. <https://doi.org/10.1145/3381052.3381320>. (cited on pages 2, 14, 18, and 42)
- ZHAO, W.; BLACKBURN, S. M.; AND MCKINLEY, K. S., 2022a. Low-latency, high-throughput garbage collection. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, 76-91. ACM. doi:10.1145/3519939.3523440. <https://doi.org/10.1145/3519939.3523440>. (cited on page 8)

-
- ZHAO, W.; BLACKBURN, S. M.; AND MCKINLEY, K. S., 2022b. Low-latency, high-throughput garbage collection (extended version). *CoRR*, abs/2210.17175 (2022). doi:10.48550/ARXIV.2210.17175. <https://doi.org/10.48550/arXiv.2210.17175>. (cited on page 40)
- ZHAO, W.; BLACKBURN, S. M.; AND MCKINLEY, K. S., 2025a. Work packets: A new abstraction for GC software engineering, optimization, and innovation. *Proc. ACM Program. Lang.*, 9, OOPSLA2 (Oct. 2025). doi:10.1145/3763139. <https://doi.org/10.1145/3763139>. (cited on pages 9, 10, 43, 44, 51, 62, 66, 70, 82, and 88)
- ZHAO, W.; BLACKBURN, S. M.; MCKINLEY, K. S.; CAO, M.; AND HAMOUDA, S. S., 2025b. Advancing performance via a systematic application of research and industrial best practice. *Proc. ACM Program. Lang.*, 9, OOPSLA2 (Oct. 2025). doi:10.1145/3764118. <https://doi.org/10.1145/3764118>. (cited on pages 98 and 101)
- ZHENG, Y.; YU, T.; YANG, Y.; HU, Y.; LAI, X.; WILLIAMS, D.; AND QUINN, A., 2025. Extending applications safely and efficiently. In *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, 557–574. USENIX Association. <https://www.usenix.org/conference/osdi25/presentation/zheng-yusheng>. (cited on page 100)
- ZHOU, Z.; LI, C.; YANG, F.; AND SUN, G., 2023. DIMM-Link: Enabling efficient inter-DIMM communication for near-memory processing. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2023, Montreal, QC, Canada, February 25 - March 1, 2023*, 302–316. IEEE. doi:10.1109/HPCA56546.2023.10071005. <https://doi.org/10.1109/HPCA56546.2023.10071005>. (cited on pages 76, 90, and 92)
- ZORN, B. G., 1990. Barrier methods for garbage collection. Technical report, University of Colorado Boulder. <https://scholar.colorado.edu/concern/reports/47429970d>. (cited on pages 2, 10, 14, 17, and 42)
- ZORN, B. G., 1993. The measured cost of conservative garbage collection. *Softw. Pract. Exp.*, 23, 7 (1993), 733–756. doi:10.1002/spe.4380230704. <https://doi.org/10.1002/spe.4380230704>. (cited on pages 2, 14, 16, and 42)