High Performance Garbage Collection: A Microarchitectural Perspective

Claire Huang

Supervised by Steve Blackburn and Zixian Cai

A thesis submitted in partial fulfilment for the degree of Bachelor of Philosophy (Honours) at The Australian National University

May 2025

© Claire Huang 2025

I declare that this work:

- upholds the principles of academic integrity, as defined in the Academic Integrity Rule;
- is original, except where collaboration (for example group work) has been authorised in writing by the course convener in the class summary and/or LMS course site;
- is produced for the purposes of this assessment task and has not been submitted for assessment in any other context, except where authorised in writing by the course convener;
- gives appropriate acknowledgement of the ideas, scholarship and intellectual property of others insofar as these have been used;
- in no part involves copying, cheating, collusion, fabrication, plagiarism or recycling.

I acknowledge that I am expected to have undertaken Academic Integrity training through the Epigeum Academic Integrity modules prior to submitting an assessment, and so acknowledge that ignorance of the rules around academic integrity cannot be an excuse for any breach.

Claire Huang 23 May 2025

To my animal companions, Pepper, Apricot and Maple, who always make me smile

Acknowledgments

First and foremost, I would like to thank Steve Blackburn, for being the best imaginable supervisor and mentor. You first took me on as a fresh first year who knew absolutely nothing about garbage collection and introduced me to the crazy but exciting world of GC research. You have always been generous with your time, support and encouragement and I learn so much everyday under your guidance. You inspire me to be a better researcher and person.

To Zixian Cai, my co-supervisor. Thank you for your constant guidance, mentorship, friendship and supervision through the past four years. I have appreciated everything from your candid advice on research careers, to patiently teaching me the ropes of hairy complex systems like linux perf, to random cat photos.

To the MMTk team: Steve Blackburn, Zixian Cai, Wenyu Zhao, Kunal Sareen, Tianle Qiu, Kunshan Wang, Yi Lin, Yasaswini Gownivaripalli, Eduardo Souza and Hayley Patton. Your willingness to teach me the internals of MMTk, answer my *many* questions and help with explaining strange results has been invaluable to me. Thank you for always making the lab fun with your cola-flavoured oreos and oreo-flavoured colas, among other silly things. I have loved working with you all.

To Angus Atkinson, thank you for spending an afternoon walking me through your thesis respository and giving me advice on honours. I also thank Trevor Carlson (from NUS) for your expertise in understanding a strange experiment result.

I thank all the amazing researchers at ANU who have inspired me. To Michael Norrish, thank you for supervising me on a undergraduate research project. To Peter Hoefner, Fabian Muehlboeck and Tony Hosking, thank you for introducing me to the wonderful world of programming languages and compilers.

To Zack, thank you for listening to my all stressed garbles, proofreading terrible early iterations of my thesis and always being my happy place.

To Ella, thank you for being my built-in best friend, eating chocolate and strawberries with me at 10pm for no good reason, telling me random stories from your day and always making me laugh.

Finally, to my mum and dad. Thank you for adhering to my two-week "you can't say honours" ban, your constant unwavering belief in me, and above all, your unconditional love and support.

Abstract

Garbage collection (GC) is a performance-critical component in systems from the datacenter to the phone, and improving GC performance remains a problem. However, GC developers often rely on analysis methodologies with coarse-grained attribution, received wisdom and intuition to make performance-critical design decisions. This is likely because of limitations in existing tooling and the complexity of GC in a landscape where hardware and software is rapidly changing. The dynamically changing environment amplifies the weakness of this approach, potentially invalidating prevailing folklore and understandings. Improved microarchitectural support and tooling now exists, making it possible and more accessible for GC developers to understand the cache performance problems of GC deeply.

My thesis is that, by leveraging new hardware tools and methodologies, a deeper microarchitectural examination can localise GC performance problems and provide new insights.

In this thesis, I perform novel microarchitectural analysis for GC. To do this, I develop a methodology which allows for fine-grained attribution of load latency to different software functions and cache levels. I apply this alongside the existing arsenal of analysis techniques to gain a deeper understanding of the microarchitectural performance of the Immix collector on an Intel Coffee Lake machine. Armed with these insights, I also examine the performance of existing hardware and software prefetching schemes for GC.

I find that L1 misses are uncommon but expensive, accounting for only 2.0% of loads but 14.9% of load latency, and demonstrate that mitigating L1 misses is crucial for reducing load latency. I also discover that during collection, heap accesses are surprisingly only responsible for 3.2% of loads and 10.9% of load latency, indicating limited headroom for heap-targeted optimisations.

I identify that the tracing loop accounts for 7.8% of latency cycles from L1 misses, and find that the addition of software prefetching covers almost all of the available headroom, leading to a 9% speedup of GC on Coffee Lake and 18% speedup on an AMD Zen 4 machine. Unexpectedly, I also find that software prefetching leads to improvement in the load latency of L1 hits.

Finally, I demonstrate that hardware prefetching is effective for each of the canonical collection algorithms I study, but sensitive to the locality properties of the collector and workload.

Through a novel approach to GC microarchitectural analysis, I am able to uncover various new insights on GC load performance, and efficacy of software and hardware prefetching for GC. I also expose surprising contradictions to existing GC understandings, emphasising traditional understandings of GC hardware behaviour may

no longer be true. This new analysis opens up many new questions and interesting problems for future work, and reaffirms the continued need for comprehensive analysis of GC performance in a world where software, hardware and workload changes and complexities outpace the understandings of the GC community.

Contents

Acknowledgments vii						
A	Abstract ix					
1	Intr	oductio	on 1			
	1.1	Proble	em Statement			
	1.2	Contr	ibutions			
	1.3	Thesis	S Structure			
2	Bac	kgroun	d 5			
	2.1	Taxon	omy of Collectors 5			
		2.1.1	Heap Layouts			
		2.1.2	Heap Operations			
	2.2	Memo	pry Management Toolkit			
		2.2.1	Work Packets in MMTk 7			
	2.3	Tracir	g Collectors			
		2.3.1	Heap Objects			
		2.3.2	Tracing Loop			
		2.3.3	Copying Collectors			
		2.3.4	Generational Collection			
		2.3.5	Relevant Collection Algorithms			
	2.4	Memo	ory Caches			
		2.4.1	Locality			
		2.4.2	Placement and Replacement			
		2.4.3	Coffee Lake Memory Subsystem			
	2.5	Proce	ssor Based Event Sampling			
		2.5.1	Performance Monitoring Unit (PMU)			
		2.5.2	Counting vs Sampling			
		2.5.3	Interrupt Based Sampling vs PEBS			
		2.5.4	Memory Latency Measurements with PEBS			
	2.6	Prefet	ching			
		2.6.1	Hardware Prefetching			
		2.6.2	Software Prefetching			
	2.7	Sumn	\tilde{r}			

3 Related Work				
	3.1	Microarchitectural Performance Analysis for GC	17	
3.2 Software Prefetching		Software Prefetching	19	
	3.3	Hardware Prefetching	19	
	3.4	Summary	20	
4	Exe	cution Methodology	23	
	4.1	Software Platform	23	
	4.2	Hardware Platform	25	
5	Mic	roarchitectural Analysis of GC	27	
	5.1	Preliminary Work	27	
		5.1.1 TLB Efficacy	27	
		5.1.2 Examining L1 and L3 misses	28	
	5.2	Performing Comprehensive Microarchitectural GC Analysis with PEBS	31	
		5.2.1 Methodology	31	
		5.2.2 Methodological Contribution	33	
	5.3	Microarchitectural GC Analysis of Immix with Coarse-Grained Attribu-		
		tion	34	
		5.3.1 GC-level Analysis	35	
		5.3.2 An Initial Upper Bound for Optimisation Headroom	35	
		5.3.3 Focusing on Heap Accesses	36	
		5.3.4 Frequency of Heap Accesses	36	
		5.3.5 Impact of Workload	38	
		5.3.6 Examining Heap Accesses of Different Workloads	40	
	5.4	Fine-Grained Microarchitectural GC Analysis of Immix	41	
		5.4.1 Analysing Function Level Results	43	
		5.4.2 Deep Dive: ProcessEdgesWork::do work and the Tracing Loop.	44	
	5.5	Other Microarchitectural Investigations	46	
		5.5.1 Misses by Cache Line	46	
		5.5.2 Last Branch Record (LBR)	47	
		5.5.3 LFB Full Counters	49	
	5.6	Summary	50	
6	Soft	ware Prefetching for GC	51	
	6.1	Implementing Edge and Object Prefetching in MMTk	51	
		6.1.1 Minor Modifications for Prefetching Support	52	
	6.2	Preliminary Checks	52	
		6.2.1 Instruction Footprint	52	
		6.2.2 Overhead of Issuing Prefetches	52	
	6.3	Performance of Software Prefetching in MMTk	53	
		6.3.1 Results for Zen 4	54	
		6.3.2 Results for Coffee Lake	55	
	6.4	Unpacking the Performance of Software Prefetching	56	

		6.4.1	Microarchitectural Analysis	56	
		6.4.2 6.4.3	Reevaluating Why the Performance Exceeds the Apparent Head-	60	
			room	60	
	6.5	Explor	ring Dynamic Prefetching	62	
		6.5.1	Performing Headroom Analysis	62	
		6.5.2	Headroom Analysis for Edge Prefetching	63	
		6.5.3	Headroom Analysis for Object Reference Prefetching	65	
	6.6	Summ	ary	66	
7	Har	dware I	Prefetching for GC	69	
	7.1	Experi	mental Setup	69	
	7.2	Efficac	y of Hardware Prefetchers	69	
		7.2.1	Impact of Collector Algorithm	71	
		7.2.2	Impact of Workload	73	
	7.3	Microa	architectural Analysis of Hardware Prefetchers	74	
		7.3.1	Top-Level Microarchitectural Analysis for Immix	74	
		7.3.2	Examining Outlier Workloads	75	
	7.4	Summ	ary	78	
8	Futu	ire Wor	k	79	
	8.1	Microa	architectural GC Analysis Extensions	79	
		8.1.1	Store Latency	79	
		8.1.2	Deeper LFB Analysis	79	
		8.1.3	Other Collection Algorithms	80	
		8.1.4	Concurrent GC	80	
		8.1.5	Analysis on AMD	80	
		8.1.6	Understanding Variance in L1 Load Penalty	80	
		8.1.7	Reevaluation of Immix line sizes	81	
		8.1.8	Object Arrays	81	
		8.1.9	Workload Analysis	81	
		8.1.10	Metadata Accesses	81	
	8.2	Lightv	veight Benchmark-Level Dynamic Prefetching	82	
		8.2.1	Dynamic Adjustment Mechanism	82	
		8.2.2	Heuristics	82	
	8.3	Furthe	er Research on Hardware Prefetching	83	
		8.3.1	Performance Across GC Phases	83	
		8.3.2	Work Packet Size	83	
		8.3.3	Focusing on Specific Hardware Prefetchers	83	
		8.3.4	Data Dependent Prefetching for GC	84	
		8.3.5	Understanding Hardware Prefetcher Efficacy on Java Workloads	84	
9	Con	clusion	L	85	
Bi	Bibliography 87				

xiii

Contents

List of Figures

2.1	This depicts a program with both temporal and spatial locality	11
5.1	This figure depicts the overall methodological workflow for microar- chitectural analysis of GC load performance. Different stages of the methodology are shown in blue, intermediate data is shown in white, and the final analysis output is given in green	31
5.2	L1 misses are responsible for around 15 % of the latency overhead on average, however this number is sensitive to the choice of workload. This plot gives the breakdown of the total latency overhead attributable to different cache levels, omitting L1. For cleaner visualisation, I use percentages given to 1dp here.	38
5.3	Despite only occurring 3.23% of the time, accesses to heap objects are responsible for 10.9% of total load latency. Moreover, this varies for different workloads. For each benchmark, this figure shows the percentage of total load latency cycles occupied by accesses to heap objects for each cache level. Results are rounded to 1dp	40
5.4	Loads called from seven key functions dominate accesses to all cache levels. This figure shows the breakdown of accesses at each cache level attributable to each function. Results are given to 1dp	42
5.5	Loads called from seven key functions dominate the latency overhead of all cache levels. This figure shows the breakdown of the percent- age of latency cycles at each cache level attributable to each function. Results are given to 1dp	42
5.6	The attribution of the load latency of ProcessEdgesWork::do_work to different cache levels for each benchmark is an indicator of which workloads exhibit good locality during tracing. All numbers are given as percentages rounded to 1dp	45
6.1	The memory bandwidth utilisation of h2 remains low throughout its execution. This figure shows the results of VTune memory bandwidth measurements for an execution of h2.	53
6.2	The addition of software prefetching improves GC performance by 9% on Coffee Lake. This figure shows the GC time of Immix with prefetching on the Coffee Lake machine, normalised to the baseline. The geomean is included in orange.	54
	0	

6.3	The addition of software prefetching improves GC performance by 18 % on Zen 4. This figure shows the GC time of Immix with prefetching on the Zen 4 machine, normalised to the baseline. The geomean across benchmarks is in included in orange. I omit tradesoap due to issues	
6.4	encountered during benchmark execution	54
	the results of a linear regression on this relationship.	56
6.5	Prefetching effectively reduces latency cycles attributable to L1 misses for majority of benchmarks compared to the baseline (Section 5.3.5). This table presents the percentage of total latency cycles attributable each level of the cache for Immix with prefetching. Numbers are given	
	to 1dp	58
6.6	For all benchmarks, L1 hits now dominate the load latency cycles of ProcessEdgesWork::do_work. This figure shows the percentages of the load latency cycles of ProcessEdgesWork::do work which occur at	
	each cache level (1dp).	58
6.7	This figure shows the proportion of ineffective prefetches which occur due to hits in L1 or LFB.	60
6.8	This figure shows, for each benchmark, the headroom available for	63
6.9	This figure shows, for each benchmark, the headroom available for benchmark-level and GC-level dynamicism for object reference prefetch-	03
	ing	65
7.1	The efficacy of hardware prefetching for GC is correlated with how col- lectors move objects, and also sensitive to the choice of workload. This heatmap shows the GC time without hardware prefetching normalised to baseline MMTk for six different collectors across the DaCapo bench- marks. Two configuration pairings failed to record results and are	
	omitted.	70
7.2	Surprisingly, the performance of mutators appears to be mostly unaf- fected by the absence of hardware prefetchers. This heatmap shows the mutator time without hardware prefetching normalised to baseline MMTk for six different collectors across the DaCapo benchmarks. Two	
	configuration pairings failed to record results and are omitted	71
7.3	This figure shows the proportions of objects in contiguous runs of out- going references of different lengths. For example, biojava has 97.85 %	
	or objects in large arrays between 2 ¹⁰ and 2 ¹¹ . The array sizes are sorted from most to least common	72
7.4	Disabling hardware prefetching degrades performance by increasing the load latency of L1 misses for all benchmarks. This figure shows the percentage of total load latency which is occupied by L1 misses for	73
	Immix without hardware prefetching (1dp)	75

List of Tables

2.1	Characterisations of the four GC algorithms used throughout the thesis.	9
2.2	This table gives the cache specifications for the Intel Core i9-9900K Coffee Lake machine. Note that the minimal load-to-use latencies are nominal values given by Intel for Skylake microarchitectures (this is	
	the same as Coffee Lake).	12
4.1	This table describes the DaCapo Chopin benchmark suite, alongside the minheap values for Immix [Blackburn et al., 2025]	24
4.2	The L1 hit rates and L3 miss rates of single and multi-threaded Immix are similar.	24
4.3	Processors used in my evaluations	25
5.1	During collection time, page walks are uncommon and unlikely to dominate load performance. This table shows the TLB hit rates and associated latency overheads during GC time for the Immix collector on Coffee Lake. Results are reported as the geomean over the DaCapo benchmark suite (2dp).	28
5.2	Key L1 and L3 cache metrics can give high-level insight into GC per- formance. This table gives the L1 hit rate, L3 miss rate and RAM access rate of six collectors, reported as the arithmetic mean over benchmarks. Results are rounded to 3sf	29
5.3	This table shows the number of GCs (rounded to the nearest integer) performed by each collector.	29
5.4	This table shows the GC time of six key collection algorithms, nor- malised to MarkCompact (2sf).	30
5.5	Despite a high L1 hit rate, a disproportionate number of cycles are spent resolving L1 misses, half of which are to heap objects. This table shows the average cache performance across the four metrics for the DaCapo benchmarks. Load proportion and latency overhead for heap accesses are relative to the overall GC performance. All figures are rounded to 3sf.	34
5.6	Seven key functions make up 93.0% of L1 hits which are not accesses to heap objects. All results are given as 3sf	36
5.7	This table summarises the functions which, together, are responsible for majority of load latency.	37

5.8	The efficacy of biojava is likely explained by a high L1 hit rate of 99.40 %. This table presents the cache hit rates, load proportions, latency over-	20
5.9	h2 has an unexpectedly high number of LFB hits and RAM accesses, both with high average penalties. This table presents the cache hit rates, load proportions, latency overheads, and average penalties for	
5.10	h2, given to 3st	39
5.11	For each benchmark, this table gives the percentage of total load accesses and the associated percentage of load latency overheads which are attributable to ProcessEdgesWork::do_work. Numbers are given	41
5.12	Load attribution of a toy example. I write PE for ProcessEdgesWork:	44
5.13	do_work to make the table fit	46 47
6.1	Software prefetching effectively improves cache performance on al- most all key metrics. This table presents the results of microarchitec- tural analysis of Immix with software prefetching (swpf) and without	
6.2	prefetching (baseline), with all numbers rounded to 3sf There is very limited remaining headroom for further prefetching op- timisations for ProcessEdgesWork::do_work. This table computes the remaining optimisation of ProcessEdgesWork::do work headroom for	57
6.3	the DaCapo benchmarks (3sf)	59
6.4	to the current prefetch distance of 32	64 66
7.1	This table compares L1 cache hit rates, L3 miss rates, and RAM access rates with and without hardware prefetching.	72
7.2	Cache performance suffers under almost all metrics with the hardware prefetchers disabled. This table shows top-level microarchitectural analysis for Immix with and without hardware prefetching, with values	
	rounded to 3st.	-74

7.3	The hardware prefetchers likely play a key role in the cache perfor-	
	mance of biojava (a key outlier in performance in Section 5.3.5). This	
	table presents the top-level microarchitectural analysis for biojava with	
	and without hardware prefetching	76
7.4	The hardware prefetchers also play a key role in the cache performance	
	of graphchi, improving the hitrate of all three levels of cache. This table	
	presents the top-level microarchitectural analysis for graphchi with and	
	without hardware prefetching	76
7.5	The hardware prefetcher is less effective for avrora, where the LFB	
	penalty remains similar with and without the hardware prefetcher.	
	This table presents the top-level microarchitectural analysis for avrora	
	with and without hardware prefetching	77
7.6	The hardware prefetcher is less effective for zxing, where the propor-	
	tion of load latency attributable to RAM does not decrease by much	
	with hardware prefetching. This table presents the top-level microar-	
	chitectural analysis for zxing with and without hardware prefetching.	78

LIST OF TABLES

Introduction

Garbage collection (GC) automatically manages memory used by a program, identifying objects in-use and reclaiming unreachable objects. Without GC, programmers must manually manage objects, which is tedious, error-prone, and a major source of critical security problems. GC underpins most widely used managed programming languages, such as Java, C#, Ruby, Haskell and JavaScript. With increasing societal emphasis on security, managed languages are continuing to gain popularity, predominantly due to safety and productivity benefits of GC [The White House, 2024]. Popular web services like Shopify, GitHub and Airbnb are built by GC-backed frameworks, while Android OS and Twitter are examples of Java-based products. Thus, GC is ubiquitous, and performant GC permits improved user experiences such as responsive websites and applications, and energy efficient devices.

Despite the ubiquity of GC, and the maturity of GC research, the cost of GC remains surprisingly high [Cai et al., 2022; Blackburn et al., 2025]. Consequently, the development of high performance GC algorithms and optimised, robust implementations of performant GC are both increasingly important and relevant orthogonal concerns.

Prerequisite to this is comprehensive evaluation and understanding of performance of existing GCs, and more importantly, insight into why GC might perform this way. However, constant seismic changes in the hardware, software and workloads greatly affects the behaviour and performance of GC. Workloads are increasingly server-based, and differ in design and behaviour from traditional C-style programs, making use of object-oriented paradigms and languages, like Java, with parallel programming abstractions and concurrency primitives [Blackburn et al., 2025]. Modern speculative, out-of-order, superscalar CPUs also have sophisticated cache hierarchies, and compilers are complex and optimised through decades of development.

As a result, GC developers often rely on received wisdom, analysis techniques with coarse-grained attribution, such as performance counters, and intuition to make performance-critical design decisions. This approach has numerous limitations. The aforementioned technological shifts in both hardware and software can render folklore as less relevant, and thus, at a minimum, reevaluation of prior work is necessary. Additionally, while useful in gaining high-level insights on how GC performs, analysis with only coarse-grained attribution limits the scope of understanding as it lacks the ability to thoroughly diagnose why GC performs the way it does. These limitations can lead to evaluation criteria used by GC researchers which are insufficient, unrepresentative or even incorrect, leading to misguided design choices. Ultimately, it can hinder the development of performant, low-cost GCs. Therefore, both methodological development for GC analysis and reevaluation of GC performance are needed.

Recent software-focused methodological advancements, such as tracing and distillation, have partially addressed this issue [Huang et al., 2023; Cai et al., 2022]. However, there remains a methodological gap for comprehensive microarchitectural exploration of GC. This is significant because microarchitectural optimisations, such as prefetching, can lead to significant improvements in GC performance [Garner et al., 2007; Atkinson, 2023]. Therefore, understanding the microarchitectural behaviour of GC is key to unlocking greater GC performance. The limitations of existing methodologies are exposed in existing work, such as Blackburn et al. [2004a], Carpen-Amarie et al. [2023] and Papadakis et al. [2023], which provide insight into the hardware perspective, but offer limited fine-grained observability. This is because they predominantly rely on methodologies which examine performance using topdown approaches or performance counters which have coarse-grained attribution of latencies and can only measure metrics across the entire GC execution.

The addition of data linear address (DLA) facilities for Processor Event Based Sampling (PEBS) to newer Intel machines provides hardware support for detailed measurement of load-specific information, such as the linear address of the target of the load operation, the load latency and the cache level the load hits at, which I will exploit [Intel Corporation, 2016].

1.1 Problem Statement

Due to the fast-changing nature of hardware, software and workloads, and methodologies which focus only on coarse-grained attribution, the GC community has a limited understanding of the behaviour of GC at a microarchitectural level. This can have deep consequences for GC development, leading to misguided design choices and missed optimisation opportunities. Improved microarchitectural measurement tools now exists, providing an avenue for revalidation of existing wisdom and discovery of new results. This thesis aims to leverage improved microarchitectural support to understand GC microarchitectural performance in a new light.

1.2 Contributions

The contributions of the thesis are as follows:

1. I develop a novel methodology for comprehensive microarchitectural GC analysis with fine-grained attribution of load latency to different cache levels and software functions, and enables isolated measurements of performance on accesses to heap objects.

- 2. I leverage this to perform a thorough and principled microarchitectural analysis of GC load performance for the Immix collector, reevaluating the conventional wisdom on GC performance, uncovering new insights into GC performance, and exposing new opportunities for optimisations. This opens up many new questions and rich, interesting problems for future work.
- 3. Using these insights, I analyse the performance of a software prefetching scheme for the Immix collector in MMTk. Software prefetching is used to improve cache performance by loading objects into the cache ahead of time, and is helpful for tracing GC, which suffers locality issues from irregular pointer chasing. I also demonstrate that adaptive software prefetching schemes which dynamically adjust to variations in workloads and hardware characteristics are not profitable.
- 4. I perform microarchitectural analysis of the efficacy of hardware prefetchers for GC, revealing the effects of workloads and collector algorithms on hardware prefetching performance.

1.3 Thesis Structure

In Chapter 1, Chapter 2 and Chapter 3, I analyse and dissect the existing literature, with a focus on microarchitectural GC analyses and prefetching. I explain the current gaps and how my work fits in.

Chapter 4 details the hardware and software execution methodologies used throughout the thesis.

In Chapter 5, I introduce novel methodology for microarchitectural GC analysis with fine-grained attribution. I perform microarchitectural analysis for the Immix collector, uncovering interesting new insights and finding surprising contradictions to existing work. In particular, I find that the tracing loop is a key optimisation target. This leads me to analyse the performance software prefetching for MMTk's tracing loop in Chapter 6. The insights from the microarchitectural analysis also motivates a case study in hardware prefetching, presented in Chapter 7.

Finally, in Chapter 8, I discuss numerous interesting questions and problems for future research, and in Chapter 9, I summarise the findings and takeaways of the thesis.

Introduction

Background

In this thesis, I follow the convention in the literature, and refer to the garbage collector as the *collector* and the user application as the *mutator*. Mutators execute application code, allocating and modifying objects and mutating the heap, while collectors perform automatic memory management for the mutator. Collectors provide the mutator with guarantees of *safety* (the collector never reclaims live objects) and *liveness* (collection terminates), allowing the mutator to avoid the tedious and error-prone task of manual memory management. In this chapter, I introduce all the relevant background information for my thesis.

In Section 2.1, I will introduce the taxonomy of collectors, and discuss different choices for *heap layout* and the fundamental *heap operations* of allocation, identification and reclamation. In Section 2.2, I introduce the Memory Management Toolkit (MMTk), which is a platform for building efficient, flexible and robust collectors, and explain the work packet system which lies at the heart of MMTk. Next, Section 2.3, I give details on *tracing*, which is the most widely used strategy for reclamation. I survey the key tracing collectors used in this thesis, and discuss their behaviour and performance.

The heart of this thesis focuses on understanding the microarchitectural behaviour of GC using hardware-supported measurement techniques. Therefore, in Section 2.4, I explain the memory cache subsystem which underpins modern processors and allows for reduced load latencies. Then, in Section 2.5, I discuss Processor Event Based Sampling (PEBS) and how it can be leveraged for measuring latency and other characteristics of loads. Finally, I discuss prefetching, which is an important technique in used to improve load latencies by reducing cache misses.

2.1 Taxonomy of Collectors

In this section, I discuss the taxonomy of collectors. Managed languages perform dynamic allocation in the heap, which collectors are responsible for managing. They do this by performing three key heap operations: they allocate objects, identify objects in-use and reclaim the space occupied by unreachable objects. I identify collector algorithms by their heap organisation and how they perform the heap operations [Blackburn and McKinley, 2008].

2.1.1 Heap Layouts

As in standard GC terminology, I refer to heap organisations as *spaces*. Spaces may be further split into partitions. Spaces are classified into three types: monotone, freelist or regions.

Monotone spaces are contiguous ranges of memory with a single boundary that separates live and free memory. They are typically reclaimed en masse.

Freelist spaces stores blocks of free heap memory into a freelist. Allocators acquire memory from the freelist, and reclamation returns memory to the freelist.

A *region-based* organisation splits the heap into multiple monotone spaces, called *regions*. This allows for partial reclamation of a space. A number of production collectors, including G1, ZGC and Shenandoah, use a single fixed-size region [Liden and Karlsson, 2018; Flood et al., 2016].

Note that any collector implementation may have multiple heap organisations by segregating objects using characteristics such as object lifetime, size, type, and write activity. These spaces may be collected separately for improved performance. An example is that many collectors handle large objects separately. This is because of the costs associated with copying large objects and potential differences in hardware and operating system support for allocation of smaller objects.

The choice of space often informs which heap operations are most suitable. I discuss these operations in the next section.

2.1.2 Heap Operations

Garbage collectors rely on three key operations to manage the heap.

Allocation refers to the provision of memory to the mutator. Two approaches are *bump pointer allocation,* where heap space is allocated contiguously by bumping a pointer along the address space, or *freelist allocation,* where the collector maintains a list of free blocks of memory which can be allocated.

An optimisation on freelist allocation is *segregated size free lists*, where the collector manages a list for each size class separately [Jones et al., 2011]. The size choice here is dependent on typical workload behaviour. For example, Java objects are typically small, so many collectors use freelists in multiples of 4. Freelists may suffer from both internal and external fragmentation, and increasing the number of size classes can improve internal fragmentation at the cost of worsening external fragmentation [Jones et al., 2011].

In the *identification* phase, the collector is responsible for finding all live objects. Objects are *live* if they are transitively reachable from the mutator *roots* (i.e. stack and local/global variables), and *dead* otherwise. One common approach for identification is performing a transitive closure from a root set marking all live objects. This is known as *tracing*.

Reference counting can also be used to indirectly identify dead objects. To do this, reference counting collectors maintain a reference count for each object, incrementing when a reference to it is created, and decrementing when one is deleted. Any object with a reference count of 0 is reclaimed.

Finally, *reclamation*, which is tied closely to identification, is responsible for recycling the memory of unreachable objects. *Non-moving* collectors simply reclaim all the dead objects by checking mark bits. *Copying* collectors may also attempt to perform defragmentation. When copying, objects can be reorganised in the same space (*compaction*) or *evacutated* to a different space.

2.2 Memory Management Toolkit

In my work, I leverage MMTk, which is a robust, efficient, portable, and flexible platform for building collectors. MMTk defines different heap spaces and policies on how to perform the heap operations; these reuseable components are brought together to define collector algorithms. Algorithm implementations are separate from bindings, which implement the required language specific functionalities, making MMTk language and VM agnostic. This allows easy integration of high performance collectors into various production language runtimes. This makes MMtk an ideal platform for researchers to test GC techniques in a production-level environment. Moreover, optimisations in MMTk can be high impact, improving multiple algorithms across runtimes.

MMTk was first implemented in JikesRVM and later rewritten in Rust for improved performance, safety, and better integration with a wider range of languages [Blackburn et al., 2004b; Lin et al., 2016]. MMTk currently officially supports Open-JDK, Julia, CRuby, and JikesRVM runtimes and has unofficial ports to other runtimes including Android and GHC.

2.2.1 Work Packets in MMTk

In MMTk, almost all collection work is performed by stateless workers executing self-contained work packets [Xu et al., 2022]. Work packets contain items of the same type and a function pointer instructing the worker on how to process the packet. Each packet has a set of preconditions which dictate when it may be executed.

Work packets are organised into work buckets by preconditions, and each bucket opens when the preconditions are met. Once a bucket is open, its work packets are now available to the workers. For example, the packets for performing the transitive closure are in a bucket that cannot open until all root scanning packets have completed.

Worker threads first acquire work packets from a global pool and then consume and produce further packets into thread-local pools. If both the global and thread local pool is exhausted, a worker steals work from other thread local pools. The scalability and performance of the system lies in the distribution of work among packets and packets among workers [Huang et al., 2023].

2.3 Tracing Collectors

Tracing GC identifies live objects in the *tracing loop*, performing a transitive closure of the heap, starting at the roots and marking and scanning objects.

2.3.1 Heap Objects

To understand tracing, I must first understand how heap objects are encoded [Atkinson, 2023]. An *object* consists of a *header* and a number of *fields*. Each field contains either a *scalar*, which is an immediate value (e.g. an integer) stored directly in the field, or an *object reference*, which is a pointer to another heap object.

A *slot* is any memory location which can hold a pointer to an object. Examples of slots include the reference field of an object and the word holding a pointer from the stack.

2.3.2 Tracing Loop

To trace an object, the collector is responsible for three main actions: it must load the address of the object from the referring slot, mark the object, and if unmarked, scan the object to determine which fields contain further references.

To do this, collectors keep track of untraced objects using a queue or stack. The choice of data structure dictates the traversal order; stacks result in depth first traversal (DFS), while queues are traversed with breadth first search (BFS). For this reason, stacks are typically preferred since parent/child pairs are processed close together. This affords better locality as these pairs are also typically allocated close together. More advanced tracing collectors may use work packet systems which may combine multiple data structures. MMTk's work packet system, which was discussed in Section 2.2, can be regarded as "approximate" DFS. This is because work packets perform local BFS, however the execution of packets has no ordering.

Tracing loops are classified based on two main factors: the timing of the mark and the type of items in the queue/stack [Atkinson, 2023]. *Node-ordered* tracing performs marking before placing into the queue/stack. In particular, it does not queue objects which are already marked. *Edge-ordered* tracing unconditionally queues objects, and marks on dequeue.

The type of items also determines the behaviour of the loop. Commonly, collectors will place object references (load slot, then queue the object reference), slots (queue slot), or tuples which hold (slot, object reference) pairs into the stack/queue.

The MMTk work packet system (see Section 2.2.1) uses a dual-queue and dualloop algorithm where packets contain lists of slots (this is the "edge queue") [Atkinson, 2023]. The first loop is responsible for dereferencing the slot to obtain the object reference and for marking the object. Unmarked objects are then added to a secondary "node queue". This second loop is responsible for scanning each object and enqueueing newly discovered objects into a new work packet, which, when full, it added to the appropriate work bucket.

2.3.3 Copying Collectors

Non-moving tracing collectors suffer space inefficiencies caused by fragmentation. This can be a performance problem when the mutator has a large liveset as many expensive but ineffective collection cycles may be triggered. Copying collectors attempt to address this issue by reorganising the heap during the collection cycle [Jones et al., 2011]. This helps to minimise fragmentation and improves locality during tracing, since objects which are allocated close in time are typically placed next to each other in space.

Copying can occur in the same heap space (compaction) or into a different space (evacuation). Compaction is generally more space-efficient as it does not need to split the heap into multiple spaces.

Note that collectors may combine multiple copying algorithms or implement them to varying degrees. For example, Immix performs opportunistic evacuation [Blackburn and McKinley, 2008]. Copying GCs may also have mark-compact fallbacks which occasionally perform an expensive GC to save space [McGachey and Hosking, 2006].

2.3.4 Generational Collection

Full heap collection is expensive, especially when the heap is large. Generational collection attempts to exploit the weak generational hypothesis, which is the empirical observation that most objects die young, by splitting the heap into separate spaces based on age [Ungar, 1984]. Objects are allocated in the nursery and promoted to the mature space when they survive a certain number of GCs. Minor collections occur in nursery frequently. Occasional full-heap major GCs occur periodically, or when nursery collection is insufficient. Write barriers are used to track references crossing the space boundary to ensure correctness.

2.3.5 Relevant Collection Algorithms

In my thesis, I use four canonical collectors, characterised in Table 2.1.

	MarkSweep	Immix	SemiSpace	MarkCompact
Heap layout	Segregated size freelists	Two-level region	Two monotone heaps	Single monotone heap
Allocation	Size fit	Bump pointer	Bump pointer	Bump pointer
Identification	Tracing	Tracing	Tracing	Tracing
Reclamation	Free to list	Free to region, opportunistic evacuation	Evacuation	Compaction

Table 2.1: Characterisations of the four GC algorithms used throughout the thesis.

I also use two modified versions of Immix, which are GenImmix, the generational counterpart of Immix, and NMImmix, which is a non-moving version of Immix which does not perform defragmentation.

For most key experiments, I predominantly use the Immix collector [Blackburn and McKinley, 2008], which I now explain in greater detail. The Immix algorithm uses a two-level region hierarchy of coarse *blocks*, which are made up of fine-grained

lines. Bump pointer allocation occurs for lines first into fullest blocks. Immix uses freeto-region reclamation with opportunistic copying for defragmentation. This moves blocks with the most empty lines into blocks with the most full lines.

The Immix algorithm provides a balance between space efficiency, low collection overhead, and improved mutator locality and performance.

This is in contrast to the other collectors, which all sacrifice one of these objectives.

MarkCompact allocates contiguously, and upon collection, performs three passes of heap. These are responsible for object marking, pointer forwarding, and en masse compaction of live objects to one end of the heap respectively [Cheney, 1970; Jones et al., 2011]. It addresses defragmentation, which provides good mutator locality, and is the most space efficient, since it compresses all objects to one end of the heap. However, it has overwhelming collection times due to the high overheads of compaction, which requires multiple heap passes.

Meanwhile, the SemiSpace algorithm, relies on evacuation of objects to a new space for defragmentation [Jones et al., 2011]. To do this, it uses two identical spaces, known as the "to" and "from" spaces. Allocation occurs contiguously into the "from" space only, and upon collection, when an object is traced, it is copied into the "to" space and all references to the object are updated (this is known as *forwarding*). Then, the two semi-spaces are swapped and allocation recommences in the new "from" space. SemiSpace addresses defragmentation, and therefore has good mutator locality and performance. It also only requires a single heap pass, so it typically has lower collection times than MarkCompact. However, this comes at the cost of a $2 \times$ space overhead.

Finally, MarkSweep simply marks all live objects and frees any unmarked objects to the appropriate free list [Jones et al., 2011]. This means that it does not address defragmentation, which can harm mutator performance, but it has the smallest collection overhead. Consequently, MarkSweep is most performant for small heaps, where collection time dominates, but less performant on large heaps, due to worse mutator cache locality.

2.4 Memory Caches

GC performance is dominated by the tracing, which makes intensive use of memory, and minimising cache misses and their associated latency penalties are key to minimising GC overhead. At its core, this thesis explores the interactions between GC and the cache. Therefore, it is important to understand the foundational principles of memory caches, which I discuss in this section.

A memory cache system consists of a storage component, known as the cache, which resides in faster memory (typically SRAM) and a collection of algorithms which determine how data is organised, stored and replaced/evicted in the cache. The purpose of a cache is to improve memory performance by storing frequently accessed data closer to the CPU, thus reducing the memory access time. When performing a memory access, the memory system first checks the cache and only

performs a load to main memory if the target is not found in the cache. This is known as a *cache miss* (as opposed to a *cache hit* if the memory is found in cache). Caches usually contain block-sized sections of memory, referred to as *cache lines*.

2.4.1 Locality

Well-written programs tend to exhibit a form of locality, where future access patterns are closely correlated to access history. There are two main types of locality: *spatial locality*, where accesses in nearby locations are often accessed close in time, and *temporal locality*, which observes that locations are often accessed multiple times in close succession. This is illustrated in Fig. 2.1. If programs accessed memory completely randomly, caches would be unable to improve memory performance. However, due to locality of programs, past accesses have good prediction value. Thus, locality underpins cache efficacy.



Figure 2.1: This depicts a program with both temporal and spatial locality.

2.4.2 Placement and Replacement

Placement of blocks into a cache depends on constraints of the cache policy. Caches often adopt a set-associative structure. This means the cache is divided into n sets and each set contains m cache lines. Any memory block that is brought into the cache is first mapped onto a set, and can be placed in any available line. When n = 1, the cache is "fully associative", and when m = 1, the cache is "direct mapped". Note that fully associative caches tend to have improved hit rates at the expense of increased retrieval time, and direct mapped caches have the complete opposite tradeoff.

When a cache miss occurs, the requested contents must be brought into cache. Since caches reside in faster, more expensive memory, they tend to have limited capacity. Therefore, if there is no space for cache placement, the cache must choose an object to evict. This is known as the eviction policy. The choice of eviction policy dictates the behaviour of the cache and is closely tied with cache performance. Typical cache systems rely on heuristics which aim to minimise the average data access time.

2.4.3 Coffee Lake Memory Subsystem

In this thesis, I perform microarchitectural analysis using a pair of identical Intel Core i9-9900K machines (Coffee Lake). I discuss the Coffee Lake memory subsystem in this section.

The Coffee Lake machine has three levels of caches: L1, L2 and L3. Caches closer to the CPU are smaller but faster. These will be referred to as *higher* levels of the memory hierarchy.

The closest cache, L1, is split into L1D and L1I for data and instructions. In this thesis, I focus on data accesses, so I do not discuss L1I further, and use L1 to refer to the L1 data cache, unless otherwise specified. The next cache, L2, is shared between data and instructions, but still *private* (i.e. separate cores have separate L2 caches). The furthest cache, L3, is unified between all cores and shared between data and instructions.

The specifications of the caches are given in Table 2.2.

Table 2.2: This table gives the cache specifications for the Intel Core i9-9900K Coffee Lake machine. Note that the minimal load-to-use latencies are nominal values given by Intel for Skylake microarchitectures (this is the same as Coffee Lake).

Cache level	Capacity	Associativity	Minimal load-to-use latency (cycles)
L1D	32KB per core (8)	8-way	4
L2	256KB per core (8)	4-way	12
L3	16MB unified	16-way	44

All Coffee Lake caches are *writeback* caches, which means that on cache store hits, lower cache levels are not updated. For example, on an L1 store hit, L2, L3 and main memory are not updated. Instead, a dirty bit is used to mark that the entry is inconsistent with the memory subsystem, and the new value is written to the lower cache levels only on an eviction.

Upon dispatch of a load requests, the caches are searched from L1 to L3. If the target data is not present in any cache level, a RAM access is requested. On the Coffee Lake, an additional structure, the Line Fill Buffer (LFB), sits between the L1D and L2 and tracks outstanding L1D misses. It also performs multiple optimisations, such as merging in-flight stores to the same cache line [van Schaik et al., 2019]. Any L1D miss cannot request data from the remaining memory subsystem until a slot in the LFB becomes available.

2.5 Processor Based Event Sampling

The heart of my work lies in microarchitectural analysis. To do this, I rely on *Processor Based Event Sampling (PEBS)* provided by the Intel hardware. In this section, I discuss what event based sampling is and explain how PEBS can be used to take measurements of memory latency.

2.5.1 Performance Monitoring Unit (PMU)

Intel processors contain a piece of hardware called the Performance Monitoring Unit (PMU) which is used to measure performance events [Intel Corporation]. This mechanism is provided as a series of "counters" which are held in Model Specific Registers (MSRs). Events can be captured via counting or sampling modes.

2.5.2 Counting vs Sampling

In counting, the PMU simply records the number of occurrences of an event during the selected time interval. When sampling, the PMU repeatedly sets the counter to an initial value and decrements the counter, periodically recording selected information, such as the instruction pointer (IP) and register states, on underflow. The value of the initial value is selected based on the desired sampling rate.

Counting ensures that all events are captured, however it lacks the ability to capture additional information. On the other hand, sampling is able to associate microarchitectural events with concrete execution contexts, allowing for greater observability of event correlations. However, it may suffer from sampling bias and can have a higher overhead, depending on the sampling rate.

2.5.3 Interrupt Based Sampling vs PEBS

Sampling may differ in the mechanism in which counter underflows are handled. In interrupt based sampling, the processor triggers the performance interrupt handler. This approach can introduce an attribution issue called *skidding*, where the IP stored in the sample refers to where the interrupt occurred rather than where the counter overflowed [Bakhvalov, 2018]. The reason for this is that it is difficult to precisely attribute hardware events to instructions in a complex out of order superscalar processor.

This issue can be mitigated through hardware support where the processor is responsible for storing the IP (and other requested information) in a designated memory buffer. This ensures the instruction pointer is never off by more than one instruction. On Intel machines, this capability is known as PEBS [Intel Corporation, 2016].

2.5.4 Memory Latency Measurements with PEBS

In Haswell, a pre-cursor to Coffee Lake, Intel introduced the data linear address (DLA) facility which allows for detailed profiling of loads [Intel Corporation, 2016]. This leverages PEBS and allows for load-specific information, such as the linear address of the target of the load operation, the load latency and the cache level the load hits at, to be recorded.

2.6 Prefetching

Tracing GC suffers from poor locality due to irregular pointer chasing when processing the tracing loop. In this section, I discuss prefetching, which is an important technique used to minimise cache misses by loading objects into the cache ahead of time. This helps to overlap the memory latency with other productive work. There are two forms of prefetching: hardware prefetching, which is done by the machine, and software prefetching, which the collector can trigger using prefetch hint instructions.

2.6.1 Hardware Prefetching

Hardware prefetchers use the memory access history of programs to predict future accesses and drive prefetching. Since they lack application-level insights, hardware prefetchers generally focus on pattern matching against common access patterns, such as array access with fixed stride.

Intel Coffee Lake Prefetchers

The behaviour of hardware prefetchers is dependent on the specific microarchitecture. In this thesis, I focus on an Intel Coffee Lake machine. The machine specifications are given in Section 4.2.

There are four prefetchers on the Coffee Lake microarchitecture, described in Intel Corporation [2023]. I will summarise important details.

The L1 Data Cache Unit (DCU) Hardware Prefetcher is a *stream* prefetcher triggered by an ascending request to a recent access. Upon occurrence, the prefetcher assumes that the next access is part of a stream and fetches the next line.

The L1 DCU IP Prefetcher is a *stride* prefetcher which uses sequential load history to determine whether to prefetch additional lines. For any loads where a regular stride is detected, a prefetch is issued for the address which is the sum of the current address and stride length. The prefetcher is able to detect stride lengths up to 2KB both forwards and backwards.

The L2 Hardware Prefetcher is a stream prefetcher which monitors all L1 read requests, including reads/writes/prefetches from L1D and code fetches from L1I, to identify both ascending and descending streams. Upon identification of a stream, it prefetches cache lines in the predicted direction. It may run up to 20 cache lines ahead of the load request, and can maintain up to 32 different streams at once. This prefetcher is unable to prefetch across 4KB page boundaries.

Though the exact implementation of this prefetcher is not stated, bidirectional stream prefetchers typically use a warm up window to determine the direction and confirm the stream [Bertschi, 2022]. Consequently, these prefetchers are typically more effective on longer sequences [Lee et al., 2012].

Finally, the L2 Adjacent Cache Line Prefetcher is a *spatial* prefetcher which attempts to prefetch the *pair* line of each access. Here, pairs are defined as two lines which complete to a 128-byte aligned chunk.

Note that both L2 prefetchers prefetch data to both the L2 and L3 caches unless the L2 cache is heavily loaded with missing demand requests [Intel Corporation, 2023]. In this case, the data is only prefetched to L3. Moreover, to avoid evicting useful cache lines, if the L2 stream prefetcher is far ahead¹ of the access stream, it will also only prefetch to L3. This means that the L2 prefetchers should improve cache hit rates at both L2 and L3.

2.6.2 Software Prefetching

The software may also issue prefetch instructions which provide hints to the Memory Management Unit (MMU) on where to prefetch. This is known as *software prefetching*.

This allows programmer or compilers to exploit application-level insights into memory access patterns and offers finer-grained control.

The behaviour of a prefetch depends on the instruction used. On x64-64, there are four instructions for prefetching loads, and they can be inserted using the _mm_prefetch compiler intrinsic [mmp]. The PREFETCHT0, PREFETCHT1 and PREFETCHT2 instructions target L1, L2 and L3 respectively. These instructions also prefetch data into lower cache levels. For example, PREFETCHT1 prefetches into L2 and L3 but not L1. The fourth instruction, PREFETCHTA, performs a *non-temporal prefetch*. This is used to indicate that the data will only be used once. This knowledge helps the processor make adjustments to the cache replacement policy, thus minimising cache pollution. The MMU may also place these prefetches outside the cache hierarchy but close to the processor. Note that these are all hints, and the MMU can choose not to issue a prefetch.

Factors Affecting Prefetch Efficacy

When devising a software prefetching scheme, there are multiple factors to consider.

First, the *time gap* (also called *prefetch distance*) between the prefetch and use of data is crucial. If the prefetch distance is too small, the load latencies will not be hidden entirely. Conversely, if the distance is too big, data can be prematurely evicted before it is used, thereby failing to resolve the cache miss.

The tradeoff between *coverage* and *accuracy* also needs to be managed. This is because correct prefetches can help improve cache misses and load latency, but incorrect prefetches can result in cache pollution, eviction of other useful data and saturation of the memory bus [Atkinson, 2023].

¹The optimisation manual does not state the specific distance

Adaptive Prefetching Schemes

Software prefetching schemes may hold the prefetch distance constant (*fixed-distance* scheme) or they may adaptively adjust the prefetch distance (*dynamic prefetching*). Dynamic prefetchers can be difficult to implement, as they must be able to accurately predict optimal prefetch distances from historical data points. However, they can provide greater accuracy as they can accommodate different hardware and workloads, and even account for variations across the execution of an application. Examples in the context of GC include:

- Long-running workloads where the live set and object graph vary greatly across GCs at different application stages, thus changing the optimal prefetch distance may be different across GCs.
- Varied efficacy of software prefetching schemes for traversing pointer arrays on machines with data dependent hardware prefetchers.
- Workloads with bursts of high memory bandwidth utilisation, where software prefetching may even harm performance by saturating the bandwidth and polluting the cache.

2.7 Summary

In this chapter, I gave an overview of GC, introduced MMTk as a platform for high performance GC research, and discussed relevant tracing collection algorithms and their performance characteristics. I also introduced the Coffee Lake memory subsystem, discussing how cache performance directly relates to load latency, and explained how PEBS can be used to measure and understand the performance of memory accesses. Finally, I discussed prefetching as a strategy for reducing cache misses and improving load latency.
Related Work

In this chapter, I discuss the current literature on microarchitectural analysis, software prefetching and hardware prefetching for GC.

I explain how constantly changing hardware, workloads and software necessitates deeper microarchitectural performance analysis for GC, and how improved microarchitectural support provides a yet to be exploited avenue to achieve this. This helps to motivate my work on microarchitectural GC analysis (Chapter 5). I also expose the gaps in the current literature on software and hardware prefetching for GC which I attempt to resolve using microarchitectural analysis in Chapter 6 and Chapter 7.

3.1 Microarchitectural Performance Analysis for GC

Comprehensive evaluation and understanding of performance of GC performance underpins the development of performant GC. On the software side, methodological advancements such as tracing and distillation have helped to address this issue [Huang et al., 2023; Cai et al., 2022]. However, methodologies for microarchitectural exploration and cache focused evaluation of GC remain limited.

The effects of GC on cache performance have been studied as early as in Zorn [1991], which simulated the data cache miss rate of both generational and nongenerational MarkSweep and SemiSpace algorithms, and in Reinhold [1994], which estimated the data cache overhead of a Cheney-style SemiSpace collector. Both these works focus only on simple data cache performance metrics and their findings likely no longer apply because collector algorithms, workloads and hardware have all changed substantially since.

Prevailing microarchitectural GC research relies on measurements such as performance counters which only have coarse attribution. Blackburn et al. [2004a] measure L1 and L2 cache misses, among other metrics, as part of work which quantifies GC behaviour for semi-space, mark-sweep and reference counting collectors and their generational counterparts using MMTk in JikesRVM [Blackburn et al., 2004b; Alpern et al., 2000]. This is a precursor to my work, but takes measurements across the full GC execution and therefore lacks fine-grained attribution. Moreover, this work uses the SPEC JVM benchmarks, which are built off C based workload characteristics. With the development of representative Java benchmark suites, such as DaCapo, which better incorporate rich object behaviours and have higher memory demands [Blackburn et al., 2025]. This work also predates development of the Immix collection algorithm, which MMTk now uses for its superior performance across goals of space efficiency, fast reclamation, and mutator performance when compared to the canonical tracing algorithms analysed here [Blackburn and McKinley, 2008].

More recently, Carpen-Amarie et al. [2023] introduced methodology to quantify the effect of concurrent GCs, with a focus on L3 misses. This provides an important step towards a deeper microarchitectural understanding of GC, but does not examine the whole memory subsystem. Furthermore, this work is limited in scope since it only examines metrics across the entire execution of the workloads. In similar vein, Papadakis et al. [2023] proposed a multifaceted profiling approach of Java benchmarks, which includes examination of the frequency of L1 reads and writes, and measures misses per kilo instructions and miss rates at all levels of the memory subsystem. However, this work measures the entire application execution, and therefore does not directly help with understanding GC.

The aforementioned methodologies can provide high-level understandings of GC cache behaviours, however they are limited for GC observability and performance analysis, as they cannot measure the cache performance at finer granularities, such as for GC heap accesses only or focusing on the cache performance GC tracing loop. Moreover, these studies focus predominantly on miss rates, which are correlated to but do not directly translate to load latency. This is because accesses at each cache level can vary in latency. Therefore, there remains a methodological gap for comprehensive microarchitectural analysis of GC.

Intel machines provide support for processor event based sampling (PEBS) for a key subset of hardware events (see Section 2.5). This triggers samples based on events and provides the ability to samples to a precise program context. Since the introduction of data linear addressing (DLA) in Haswell, there is additional capability for measuring load-specific information of instructions, including the load latency and the cache level where the cache hit occurred [Intel Corporation, 2016]. This allows for fine-grained attribution of latencies to load events at different cache levels and calling functions, thus providing an avenue for GC microarchitectural analysis.

There are many examples of comparable work outside GC. In Mohror and Rountree [2012], the authors propose a methodology for determining the proportion of memory accesses which hit at each level of the memory subsystem. IBS, AMD's equivalent to PEBS, was used to build a data profiler, DProf, which attributes cache misses to data types and was used to locate and fix cache performance bottlenecks in Linux [Pesterev et al., 2010]. Helm and Taura [2019] build PerfMemPlus, which helps to identify the source of the most expensive memory accesses in a workload. My thesis draws inspiration from the core ideas of these works and applies them to the new domain of GC.

3.2 Software Prefetching

Software prefetching in the context GC was initially researched by Boehm [2000], who proposed a technique "prefetch on grey" which prefetches objects when they are first visited and marked. This aims to reduce cache misses by increasing the probability that objects reside in cache when they are popped from the mark stack. Boehm [2000] also linearly prefetches a few cache lines ahead of object scanning to reduce cache misses for scanning large objects and bring adjacent objects into the cache.

For collectors with FIFO tracing loops, a further optimisation called "buffered prefetching" was introduced by Cher et al. [2004]. This prefetches objects upon retrieval from the mark stack and moves them into a prefetch buffer. Scanning then occurs via the prefetch buffer, ensuring that prefetching adheres to the FIFO traversal of objects and enforcing consistent and timing-controlled prefetching.

Paz and Petrank [2007] also explored software prefetching in the context of a reference counting collector. Reference counting collectors are responsible for maintaining reference counts for objects. This allows objects to be reclaimed when their reference counts are decremented to zero. By exploiting five key opportunities for prefetching in the reference counting collector in JikesRVM, the authors were able to speed up GC performance by 8.7%.

In Garner et al. [2007], the authors explore the combination of edge enqueueing and buffered software cache prefetching for MMTk in JikesRVM, focusing on marksweep collection [Blackburn et al., 2004b; Alpern et al., 2000]. Based on this, Atkinson [2023] performs a comprehensive reevaluation of software prefetching for tracing collectors using modern hardware and the Immix collector on the DaCapo workloads [Blackburn and McKinley, 2008; Blackburn et al., 2025]. To do this, Atkinson [2023] introduces a novel framework, the auxiliary tracing loop, for evaluating the efficacy of prefetching on variations of tracing loop designs. Atkinson [2023] also determines an optimal prefetching scheme for the dual-queue, dual-loop tracing algorithm used in MMTk. This work provides a strong basis for understanding software prefetching for tracing GC, but lacks evaluation within the context of fully-functional collector. Moreover, the focus of this work is evaluation of prefetching schemes rather than understanding their performance. I attempt to leverage microarchitectural analysis to fill this gap.

3.3 Hardware Prefetching

To the best of my knowledge, evaluation of how hardware prefetchers perform for GC is limited. Therefore, the gap I attempt to fill is evaluation of hardware prefetching for GC, and use of microarchitectural analysis to understand the performance more deeply. I discuss adjacent prior work from other applications below and how they relate to hardware prefetching on GC.

In Lee et al. [2012], the authors simulate hardware prefetchers and construct methodology for classification of ineffective hardware prefetches as incorrect, early, late or redundant. This work is significant in gaining a deeper understanding of how hardware prefetchers work and their benefits and limitations. However, the analysis is performed solely on the SPEC CPU 2006 benchmarks, which tend to have different characteristics to complex, real-world object-oriented workloads like ones provided by the DaCapo and Renaissance suites [Blackburn et al., 2025; Prokopec et al., 2019]. Moreover, the addition of GC may affect the efficacy of the hardware prefetchers, since GC behaves differently to standard applications, which this work cannot account for.

Other relevant prior work includes Yang et al. [2011], where the authors evaluate the direct and indirect costs of two existing designs of zero initialisation: *hotpath zeroing*, which zeroes upon allocation, immediately prior to the first use, and *bulk zeroing*, which initializes blocks of free memory to zero prior to returning to the allocator. They find that the hardware prefetcher is integral to the performance of hotpath zeroing. This is because by interspersing zeroing with allocation, hotpath zeroing places less pressure on the memory bus, allowing the hardware prefetcher to more aggressively reduce cache misses on allocations. The methodologies from this work are applicable to GC.

Finally, there exists prior work focusing on hardware prefetching for graph traversals, which relates to GC since the transitive closure, which comprises of graph traversal, makes up the majority of GC. Beamer et al. [2015] measure the performance of graph processing and find that the Ivy Bridge hardware prefetcher is able to prefetch aggressively and effectively when the memory bandwidth utilization would otherwise be low, but ceasing prefetching when the existing memory bandwidth is already high. Kaushik et al. [2021] implement and measure a data dependent prefetcher for graph traversals, and find it improves over conventional stride prefetching by 25 %.

3.4 Summary

GC has significantly evolved, with increased heap sizes, considerable hardware parallelisation capabilities, pointer compression and newer, more complex collection algorithms. Modern superscalar CPUs have complex cache hierarchies and speculative, out of order processing which makes performance more difficult to reason about. Increased focus on safety and security has also led to widespread popularity of garbage collected languages, and production workloads are increasingly rich, complex and parallel. This constantly changing software and hardware landscape necessitates both reevaluation of existing folklore and fresh research of new optimisation opportunities. Many of the related works discussed in this chapter help to bridge this gap.

However, these works either focus on fields outside of GC (in the case of hardware prefetching), or on evaluation of overall GC performance (in the cases of microarchitectural analysis and software prefetching). Therefore, there remains a need for methodologies which allow for comprehensive GC analysis through the microarchitectural lens, with a focus on observability at finer granularities rather than performance evaluation. In other words, I wish to understand *why* rather than *if* different

components of GC are effective or ineffective. This is made possible by improved microarchitectural support, such as PEBS with DLA, and I seek to leverage this.

Related Work

Execution Methodology

The work of this thesis is grounded in empirical analysis and evaluations. In this chapter, I will detail the standard execution methodology used. Since specific experimental methodology differs greatly depending on the experiment, I will explain those where they appear.

4.1 Software Platform

I use a development version¹ of the mmtk-corev0.30 release with the OpenJDK runtime² using the associated mmtk-openjdk binding³. I build MMTk in release mode using v1.83.0 of the Rust compiler.

I use 20 diverse, real world, memory intensive benchmarks from 23.11.MR2 maintenance release of the DaCapo Chopin Benchmark Suite [Blackburn et al., 2025]. Batik, jme and tradebeans are excluded because they do not perform GCs at a 3x minheap size.

For most experiments, I use MMTk's Immix collector. I use a moderate heap of 3x the minimum heap size (minheap). Descriptions of the benchmarks and their minheaps on Immix are given in Table 4.1.

For simplicity, I use single-threaded GC by default. This is considered an acceptable tradeoff for many reasons. Firstly, the work of this thesis focuses on predominantly on understanding GC behaviour and using single-threaded GC allows for evaluation of a high performance collector while reducing complexities for ease of understanding. Additionally, GC performance in MMTk scales well [Huang et al., 2023] and sanity checks on L1 hit rate and L3 miss rate, which are both commonly used GC performance metrics [Papadakis et al., 2023; Carpen-Amarie et al., 2023], show that single-threaded GC performs similarly at a hardware level (Table 4.2). The number of GCs is also identical on all but 3 benchmarks, h2o, spring and tomcat, and on these, the difference is under 2%.

The Linux distribution used is Ubuntu 22.04.5 LTS with the 6.8.0-50-generic kernel. When benchmarking, as many of the background daemons are disabled as

¹commit 1abbc89

²commit 28e56ee

³commit 84545cc

Benchmark	Description	Minheap
avrora	A simulation and analysis framework for AVR microcontrollers	9
biojava	A generator for physio-chemical properties of protein sequences	192
cassandra	A highly-scalable partitioned row store	152
eclipse	An integrated development environment	250
fop	An output-independent print formatter	29
graphchi	An ALS matrix factoriser	184
h2	An SQL relational database engine written in Java	1959
h2o	An Open source fast scalable machine learning platform	101
jython	A python interpreter written in Java	400
kafka	A distributed streaming platform	217
luindex	A text indexing tool	27
lusearch	A text search tool	25
pmd	A source code analyser for Java	360
spring	A PetClinic application running as a Spring microservice	139
sunflow	A photo-realistic rendering system	44
tomcat	A Tomcat servlet container	74
tradesoap	A Tradesoap SOAP Daytrader benchmark	193
xalan	An XSLT processor for transforming XML documents	23
zxing	A multi-format 1D/2D barcode image processing library	468

Table 4.1: This table describes the DaCapo Chopin benchmark suite, alongside the minheapvalues for Immix [Blackburn et al., 2025]

Table 4.2: The L1 hit rates and L3 miss rates of single and multi-threaded Immix are similar.

Metric	Single-threaded	Multi-threaded	
L1 hit rate	97.5	97.7	
L3 miss rate	0.23	0.20	

possible, and the system is essentially idle outside of the experimental processes. This helps to reduce variance caused by background noise.

I use the running-ng v0.4.7 platform for benchmarking [Cai, 2024]. I apply the following JVM arguments when executing benchmarks:

- -XX:+DisableExplicitGC: disables collections triggered by applications
- -XX:MetaspaceSize=500M: sets the metaspace size to avoid metaspace GC
- -XX:+UseThirdPartyHeap: ensures that OpenJDK uses MMTk for memory management

For each benchmark, 5 iterations are run to allow the just-in-time (JIT) compiler to warm up, with results recorded on the final iteration. Unless otherwise specified, 10 invocations are run. This was chosen because it still produces stable results without dramatically increasing the total running time of experiments or the size of result files. This was particularly relevant for experiments in Chapter 5, where experiments produced large sampling data files. For benchmarks with large error, more invocations are repeatedly performed where effective for error reduction and time permitting.

For all experiments, I take the arithmetic mean over invocations. For direct measurements, I compute the overall result as an arithmetic mean over benchmarks. For normalised results, a geometric mean is taken as it is the mathematically equivalent mean in logarithmic space [Fleming and Wallace, 1986].

4.2 Hardware Platform

The machines used in the evaluation are listed in Table 4.3. For the microarchitectural analysis in Chapter 5, I predominantly use the Coffee Lake machine since my analysis uses Processor Event Based Sampling (PEBS), which is only available on Intel. For other key performance evaluations, I use both Coffee Lake ane Zen 4, as these are the newest AMD and Intel microarchitectures. This allows for a better understanding of the impact of the choice of hardware on results. For experiments that are not timing sensitive, machines are used based on availability.

Architecture	Coffee Lake	Zen 4	Zen 3	Zen 2
Manufacturer	Intel	AMD	AMD	AMD
Model	Core i9-9900K	Ryzen 9 7950X	Ryzen 9 5950X	Ryzen 9 3900X
Technology	14nm	5nm	7nm	7nm
Clock	3.6GHz	4.5GHz	3.4GHz	3.8GHz
$\mathbf{Cores}\times\mathbf{SMT}$	8 × 2	16 imes 2	16×2	12×2
L1D Cache	32 KB imes 8	$32 \mathrm{KB} imes 16$	$64 \mathrm{KB} imes 16$	$32 \mathrm{KB} imes 12$
L2 Cache	$256 \mathrm{KB} imes 8$	1 MB imes 16	$512 \mathrm{KB} imes 16$	$512 \mathrm{KB} imes 12$
L3 Cache	16 MB imes 1	32 MB imes 2	64MB	$16 \mathrm{MB} imes 4$
RAM	128GB	64GB	64GB	64GB
Memory type	DDR4-2666	DDR5-5200	DDR4-3200	DDR4-3200

Table 4.3: Processors used in my evaluations.

Execution Methodology

Microarchitectural Analysis of GC

In Chapter 1 and Chapter 3, I discussed prior work on microarchitectural GC analysis and prevailing folklore on GC behaviour. I explained how constant, substantial change in hardware, software and workloads necessitates revalidation of the folklore. Furthermore, these changes, in conjunction with improved hardware analysis support, present an opportunity to uncover new GC understandings and discover potential optimisations. Both require in-depth, fresh evaluation. To that end, in this chapter, I introduce a new methodology which leverages PEBS (Section 2.5) to attribute latency to different functions and cache levels, and isolate accesses to heap objects from other loads. I combine this with existing performance analysis techniques like performance counters to perform microarchitectural cache analysis. My work provides in-depth analysis of the memory behaviours of high performance GC on modern hardware and workloads. I debunk the common belief that GC load latency is dominated by heap accesses, reveal potential missed opportunities in metadata-targeted optimisation, and identify that the tracing loop is a key optimisation target.

Note that throughout this chapter, I focus on load performance since this is the bulk of the memory latency and blind stores are uncommon. However, analogous work is possible for stores (Section 8.1.1).

5.1 Preliminary Work

In this section, I use existing performance analysis methodologies to examine GC behaviour on modern hardware and workloads. This work has two main goals. Firstly, I examine TLB misses and verify that this is not a major limiting factor for GC performance (for the workloads I evaluate). Next, I examine L1 and L3 misses, which are traditionally used hardware metrics to analyse GC performance, and explain why they do not suffice for microarchitectural analysis.

5.1.1 TLB Efficacy

First, I examine the TLB efficacy for loads. To do this, I use record samples of loads performed by the collector and their associated TLB statuses using the perf mem tool. I performed this on a Coffee Lake machine (Section 4.2) using the DaCapo

benchmarks and the Immix collector (Section 4.1). I use a 3x heap size. This machine has two levels of TLB (L1 and L2) and a miss at L2 triggers a full page table walk [Intel Corporation, 2023]. The results are reported in Table 5.1.

Table 5.1: During collection time, page walks are uncommon and unlikely to dominate load performance. This table shows the TLB hit rates and associated latency overheads during GC time for the Immix collector on Coffee Lake. Results are reported as the geomean over the DaCapo benchmark suite (2dp).

TLB level	proportion of requests (%)	attributable overhead (%)
TLB L1 or L2 hit	99.89	98.17
TLB (L2) miss	0.11	1.83

I find a miss rate of 0.11% in the TLB, which accounts for 1.83% of latency cycles (table 5.1. These results indicate that address translation is unlikely to be a dominating factor in load performance.

5.1.2 Examining L1 and L3 misses

I now focus on understanding GC performance across the memory subsystem. To do this, I use performance counters to measure standard cache metrics:

- L1 hit rate: This metric gives the proportion of load instructions which hit the L1 data cache. It is computed as <u>L1 load hits</u>.
- L3 miss rate: This metric gives the proportion of load instructions to the L3 which miss. It is computed as L3 load miss L3 load misses.
- RAM access rate: This metric counts the proportion of total load instructions which fall the through entire cache subsystem, requiring a RAM access. It is computed as <u>L3 load misses</u>.

This follows prior work, such as Papadakis et al. [2023] and Carpen-Amarie et al. [2023], which predominantly focus on L1 and L3 misses as indicators of GC backend performance.

To perform this experiment, I use the Coffee Lake machine and take the arithmetic mean over the results of the DaCapo benchmarks (Section 4.2, Section 4.1). I include the full suite of collectors, all at 3x minheap. The results are shown in table 5.2.

For each benchmark, I also record the number of GCs and GC time. This gives a basic characterisation of the GC behaviour of the benchmark. These are shown in Table 5.3 and Table 5.4.

These metrics give some insight into the cache performance of GC. I find that all collectors have high L1 hit rates. Moreover, moving collectors, which I expect to exhibit better locality during tracing, and therefore experience better cache performance overall, indeed have higher hit rates, with MarkCompact hitting the L1 cache in 98.8% of loads. Interestingly, this effect is not seen when comparing Immix and

Metric	MarkSweep	Immix	Genlmmix	SemiSpace	MarkCompact	NMImmix
L1 hit rate	97.1	97.5	96.8	97.2	98.9	97.7
L3 miss rate	36.9	34.9	37.7	32.3	32.0	33.7
RAM access rate	0.241	0.204	0.136	0.102	0.0430	0.158

Table 5.2: Key L1 and L3 cache metrics can give high-level insight into GC performance. This table gives the L1 hit rate, L3 miss rate and RAM access rate of six collectors, reported as the arithmetic mean over benchmarks. Results are rounded to 3sf.

Table 5.3: This table shows the number of GCs (rounded to the nearest integer) performed
by each collector.	_

Benchmark	MarkSweep	MarkCompact	Immix	Genlmmix	SemiSpace	NMImmix
avrora	4	3	14	7	4	3
biojava	19	20	39	48	25	20
eclipse	11	11	21	18	10	10
fop	5	5	11	10	5	5
graphchi	31	28	68	61	31	28
h2	3	5	7	7	4	4
h2o	111	84	126	154	72	62
jython	3	3	6	6	3	3
kafka	6	4	11	8	5	4
luindex	58	38	51	100	43	34
lusearch	775	422	1380	882	472	423
pmd	5	5	14	11	6	5
spring	44	43	56	51	27	28
sunflow	177	180	416	348	195	183
tomcat	55	43	64	77	36	36
tradesoap	2	1	1	3	3	2
xalan	164	87	196	214	104	86
zxing	1	1	2	2	1	1

NMImmix. However on all metrics, these two collectors perform similarly, indicating that opportunistic copying may not occur frequently enough to noticeably improve the cache performance. For loads which reach L3, all collectors require a further RAM access for about one third of L3 loads, with less than 0.25% of total loads reaching RAM.

On the surface, these numbers are promising, and suggest good collector locality. However, I find a number of issues with this approach which limit the ability to easily

Table 5.4: This table shows the GC time of six key collection algorithms, normalised to MarkCompact (2sf).

Benchmark	MarkSweep	Immix	Genlmmix	SemiSpace	MarkCompact	NMImmix
avrora	0.11	0.088	0.25	0.18	1.0	0.088
biojava	0.12	0.13	0.0072	0.27	1.0	0.13
eclipse	0.089	0.11	0.15	0.14	1.0	0.089
fop	0.11	0.11	0.11	0.19	1.0	0.12
graphchi	0.07	0.069	0.036	0.13	1.0	0.069
h2	0.09	0.21	0.018	0.14	1.0	0.14
h2o	0.17	0.14	0.056	0.21	1.0	0.094
jython	0.033	0.038	0.019	0.051	1.0	0.038
kafka	0.076	0.059	0.026	0.088	1.0	0.056
luindex	0.11	0.075	0.019	0.18	1.0	0.062
lusearch	0.2	0.11	0.23	0.18	1.0	0.11
pmd	0.077	0.087	0.1	0.14	1.0	0.081
spring	0.25	0.29	0.015	0.21	1.0	0.15
sunflow	0.053	0.06	0.038	0.1	1.0	0.059
tomcat	0.22	0.19	0.013	0.25	1.0	0.15
tradesoap	0.3	0.14	0.04	0.36	1.0	0.34
xalan	0.14	0.076	0.24	0.17	1.0	0.074
zxing	0.026	0.03	0.013	0.13	1.0	0.028

and comprehensively understand GC. First, this naive usage lacks granularity beyond cache hit rates. This means that it is difficult to understand the exact cost incurred at each level of the cache hierarchy. While this could be remedied with additional counters, this approach can still only measure overall GC performance, which makes it difficult to see understand performance at finer granularities and narrow down culprits much further. Therefore, there is need for more comprehensive methodology, which I will develop and use throughout the rest of the chapter.

5.2 Performing Comprehensive Microarchitectural GC Analysis with PEBS

The preliminary work used existing analysis methodologies to analyse cache behaviour for GC, and while I was able to gain good high-level insight into the memory system, these approaches cannot perform more comprehensive cache analysis for GC. In this section, I introduce a PEBS-based methodology for identifying and quantifying the latency overhead of functions in a program. I leverage this to calculate the cache hit rates and load latency overheads for a program and its constituent functions. This allows us to perform novel GC performance analysis in Section 5.3.

PEBS is sampling based, and therefore suffers from the limitations of sampling, including limited fidelity and sampling bias [Huang et al., 2023]. However, in outof-order (OoO) CPUs, in order to associate microarchitectural events with concrete execution contexts, hardware support is necessary. Therefore use of PEBS is crucial here, and in the following sections, I show that PEBS is able to yield important new insights despite its limitations.

5.2.1 Methodology

In this section, I introduce a novel methodology for microarchitectural analysis of load performance of GC. The high-level workflow is shown in Fig. 5.1.



Figure 5.1: This figure depicts the overall methodological workflow for microarchitectural analysis of GC load performance. Different stages of the methodology are shown in blue, intermediate data is shown in white, and the final analysis output is given in green.

Recording Load Data

To generate a profile of memory load instructions, I utilise PEBS with data linear addressing (DLA). As discussed in 2, DLA provides information about the memory addresses and latency of accesses. This allows us to record data including the function where the load was requested from, the requested address, the cache level where the load was processed and the total load latency.

On the Coffee Lake machine, this is encoded via the *load latency* field, represented by "weight" in the perf mem output. Intel defines the field as measuring the core cycles (accounting for re-dispatches) between dispatch and the final data writeback from the memory subsystem [Intel Corporation, 2016].

To do this, I use the perf mem tool with arguments -t load to restrict to load data, -U to hide unresolved symbols and --ldlat 0 to set the minimum sample latency to 0. It can useful to raise the sample latency for better fidelity on slow loads, but for the purposes of understanding overall load performance, a representative sample must be captured, so this sample latency should be at 0. The full command is: perf mem -t load -U record -U --ldlat 0 -- cprog>.

Additionally, to ensure I can differentiate loads performed by collector and application threads, I leverage the fact that perf mem attributes loads to threads, and I modify MMTk to label the collector threads as MMTk Collector {tid}.

Extracting Load Data

In this section, I detail the specific data extraction process (in hopes that any interested readers can replicate it themselves).

The raw data is output into a .data file. I extract it into csv format using the python integration of perf provided by linux6.5. This provides a file perf-script. py which reads the raw data into a csv.

Since the raw data is very large, I apply additional data processing by modifying the function process_events:

- Using the command field ("comm"), I filter for samples where the calling threads are MMTk collector thread. This limits the analysis to work performed by the garbage collector.
- I extract the data for load latency ("weight"), associated symbol ("symbol"), load address ("addr"). These will be used to for fine-grained attribution of load latencies.
- I extract the cache access ("cache_level") field from "datasrc_decode". This is useful for understanding performance at each level of the cache.

Each entry in the final result is therefore of the form: {"is_collector": False, "weight": 9, "addr": 18446741874686296288, "symbol": "mmtk::policy::sft_map ::create sft map", "cache level":"LVL L1 hit"}

Multi-Tiered Microarchitectural Analysis

I introduce a novel approach to microarchitectural analysis. To do this, I first define four derived metrics which form the basis for my analysis:

- 1. "Hit rate" is the percentage of accesses to a cache level which record a cache hit. Specifically, "{cache_level} hit rate = number of {cache_level} hits number of {cache_level} accesses"
- "Load proportion" is the percentage of total loads attributable to a cache level. It is computed as "{cache_level} load proportion = <u>number of {cache_level} hits</u>". This metric was used for microarchitectural analysis in Mohror and Rountree [2012].
- 3. "Latency overhead" is the percentage of total load latency attributable to a cache level. The precise computation is "{cache_level} latency overhead = <u>sum of load latency of {cache_level} hits</u>".
- 4. "Penalty" is the average latency (in core clock cycles) incurred at a cache level. It is computed as "{cache_level} penalty = $\frac{\text{sum of load latency of {cache_level} hits}}{\text{number of {cache_level} hits}}$ ".

Using the above metrics, my analysis occurs in three tiers of latency attribution:

- Overall GC performance: I compute hit rate, access proportion and the associated latency overheads for each cache level across all GC time. This is useful because, as seen in Section 2.4.3, the load latency achieved is directly correlated with the cache hit rates at each level of the cache.
- Heap access performance: Next, I examine performance separately for accesses to heap objects (as opposed to GC metadata or C++/Rust heap/stack). I expect this to be particularly relevant for workloads with large livesets and complex object graphs.
- Function level performance: Finally, I examine attribute load latencies to functions, allowing for microarchitectural analysis of individual functions performance. This allows identification of performance potential optimisation targets. The ideas of this analysis tier are similar to Helm and Taura [2019].

5.2.2 Methodological Contribution

Though my analysis methodology is targeted at GC, it is easily adaptable for microarchitectural analysis of other applications. There are three main advantages of my approach, which also apply in the context of more general microarchitectural analysis:

1. Targeted metrics: I provide a concrete methodology which allows GC developers to quickly gain new insights into cache performance through four important derived metrics. These provide deeper insights than existing analysis methodologies which focus on cache hit rates.

- 2. Localising performance problems: My methodology provides rich microarchitectural analysis, allowing for attribution of load latency at both coarse and fine granualities. This is done using location information such as the thread name, function name, and memory addresses. This allows for analysis of GC performance as a whole, as well as identification of potential targets for optimisation and computation of optimisation headrooms. It also makes testing performance optimisations easier, as it can separately analyse the performance of different functions.
- 3. Improved accessibility: Precisely understanding hardware events is tricky on modern CPUs where there is a multitude of PEBS support and limited practical documentation. Events can also behave in non-obvious ways, which can lead to incorrect measurements when used naively. For example, Helm and Taura [2020] found that some Intel Skylake offcore events for total memory accesses unexpectedly failed to include prefetches. By hiding the boilerplate, my methodology helps to address an accessibility issue, helping to make microarchitectural analysis less tedious and error prone.

In the following two sections, I will apply the methodology to perform microarchitectural GC analysis for the Immix collector.

5.3 Microarchitectural GC Analysis of Immix with Coarse-Grained Attribution

In this section, I focus on the top two tiers of analysis, namely GC-level performance and performance of heap accesses. I use the Coffee Lake machine, with reported results taken as an average over the DaCapo Benchmark Suite.

Table 5.5 gives the results of both tiers of analysis. For ease of comparison, all the data for GC-level performance is presented as raw results, while the load proportion and latency overhead for heap accesses is given relative to the overall GC performance.

Table 5.5: Despite a high L1 hit rate, a disproportionate number of cycles are spent resolving L1 misses, half of which are to heap objects. This table shows the average cache performance across the four metrics for the DaCapo benchmarks. Load proportion and latency overhead for heap accesses are relative to the overall GC performance. All figures are rounded to 3sf.

Cache Level	Hi	it Rate	Load Proportion		Latency Overhead		Average Penalty	
	All Loads	Heap Objects	All Loads	Heap Objects	All Loads	Heap Objects	All Loads	Heap Objects
L1	0.980	0.0223	0.980	2.28 %	0.851	2.02 %	8.89	7.85
LFB	0.352	0.343	0.00717	50.2 %	0.0601	58.7 %	90.2	103.5
L2	0.536	0.481	0.00637	45.8 %	0.0100	43.1 %	15.9	14.9
L3	0.623	0.554	0.0037	50.8 %	0.0207	51.1 %	59.3	58.4
RAM	n/a	n/a	0.00236	67.7 %	0.0578	67.1 %	268	260

5.3.1 GC-level Analysis

First, I note that the hit rates (about 0.5% higher) are similar to what I observed through end-to-end performance counters Table 5.2. This provides a good sanity check for the analysis, as no clear sampling bias is clear.

I observe that despite only missing the L1 cache 2.00% of the time, a disproportionate 14.9% of cache latency cycles are spent resolving these misses (Table 5.5: 100 - 98.00 = 2.00 and 100 - 85.1 = 14.9). This indicates that minimising L1 misses is important for improved load performance, and suggests that cache optimisation techniques such as prefetching could be effective. This is particularly crucial for GC, which is memory intensive and performs many heap accesses across a short period of time.

Next, I note that while rare, RAM accesses are the most comparatively damaging, occurring only 0.236% of the time, but resulting in 5.78% of latency overhead. The average latency penalty is $4.52\times$ that of an L3 hit, and $30.1\times$ the penalty of an L1 hit. This suggests that if an optimisation technique is able to reduce L3 misses, it would likely improve GC performance.

Examining the latency penalties, I note that nominally, the fastest load-to-use latencies on this machine are 4 (L1), 12 (L2) and 44 (L3) (Table 2.2). There may also be delays if the memory queue is full. Moreover, L3 accesses may have high variation depending on which core the hit occurs at. Keeping these numbers in mind, I expect slight inflation over the nominal statistics, and therefore find that the numbers appear to be reasonably consistent.

Another key observation is that LFB hits have a latency penalty which is higher than L3 hits. Recall from Section 2.4.3 that an LFB hit simply means there exists existing outstanding misses to the same cache line, so the LFB access latency can be resolved via any of the lower cache levels (L2, L3 and RAM). While naively, since L2 and L3 hits are more common, I might expect that the penalty of LFB hits would be lower, since RAM accesses take longer to resolve, there is a larger window for further accesses to the existing pending cache lines. This means the proportion of LFB hits which wait for RAM accesses is likely higher than expected, thus this result is plausible. Consequently, I expect that any cache optimisation which improves the L3 hit rate would therefore also reduce the load latency from LFB hits.

5.3.2 An Initial Upper Bound for Optimisation Headroom

I can concretely compute an upper bound estimate for the optimisation headroom which the top-level results suggest. To do this, I note that if I were able to convert all L1 misses to L1 hits, then the expected average load latency would be 8.89 cycles, while currently, the average load latency is $8.89 \times 0.980 + 90.2 \times 0.00717 + 15.9 \times 0.00637 + 59.3 \times 0.0037 + 268 \times 0.00236 = 10.31$ cycles. This gives us about 13.8 % of load latency optimisation headroom (8.89/10.31).

5.3.3 Focusing on Heap Accesses

I now focus on the load performance of the collector threads when accessing the heap, ignoring for now other access such as access to stack and metadata.

I find that around half of L1 misses and L1 miss latency overhead occurs on heap accesses, which account for 48.09 % and 58.45 % of these respectively. This indicates that a large proportion of the optimisation headroom may be attainable through optimisations within MMTk targeting heap accesses. As above, RAM accesses and LFB hits appear to be key bottlenecks. Interestingly, the latency for all other cache levels are slightly improved, though this may simply be variance in access penalty.

5.3.4 Frequency of Heap Accesses

A surprising result is that only 2.23% of L1 hits were heap accesses. In particular, I compute that only $2.28 \times 0.980 + 50.2 \times 0.00717 + 45.8 \times 0.00637 + 50.8 \times 0.0037 + 0.00236 \times 67.7 = 3.23\%$ of total loads are accesses to heap objects. This number apparently contradicts the intuition that GC threads spending most of the time accessing the heap. I break down the plausibility and implications of this result in this section.

To understand how this result may be possible, I first acknowledge that there may be sampling biases and other observer effects which I was unable to isolate. With this in mind, I now focus on L1 hits which are not to the heap and examine the functions which are the most commonly load sources (> 2% of L1 hits). These are given in Table 5.6.

Function	% L1 accesses
ProcessEdgesWork::do_work	31.7
<pre>side_metadata_access</pre>	27.3
trace_object	14.1
oop_iterate	7.2
<pre>get_descriptor_for_address</pre>	6.9
<pre>mark_lines_for_object</pre>	3.6
SweepChunk::do_work	2.2

Table 5.6: Seven key functions make up 93.0% of L1 hits which are not accesses to heap objects. All results are given as 3sf.

In order to explore the results further, I must first explain each of the functions featured in this analysis. This is given in Table 5.7.

I now attempt to understand what each of the functions may load outside of the heap. This gives confidence in the plausibility of the result.

For side_metadata_access, these are clearly side metadata accesses. However, I note that the number of side_metadata_access L1 hits may be slightly inflated because the spin loop inside the access can may generate extra accesses if some threads spin in the CAS loop, which is responsible for atomic marking, too long.

Function	Purpose
ProcessEdgesWork:: do_work	This function is responsible for defining the work per- formed during the tracing loop for tracing work pack- ets. Each packet maintains a mark stack which collector thread processes.
<pre>side_metadata_access</pre>	This function is a wrapper for all accesses to the side metadata.
trace_object	This function traces an object, scanning it for additional edges that need to be processed.
oop_iterate	This function implements object scanning for the Open- JDK binding.
<pre>get_descriptor_for_address</pre>	This function retrieves the space descriptor for a given address.
<pre>mark_lines_for_object</pre>	This function marks all the Immix lines which an object spans.
SweepChunk::do_work	This function is responsible for sweeping chunks of memory to the global pool during reclamation.

Table 5.7: This table summarises the functions which, together, are responsible for majority of load latency.

Additionally, in MMTk, this loop currently has an extra load in it which can be removed (it loads the metadata once at L762, and loads the same value again in the function call at L772). An unpublished evaluation by Wenyu Zhao found that removing it leads to a performance improvement. This side metadata is likely also be the source of the L1 accesses in trace_object, since it runs the same CAS for marking. This indicates those numbers may also be inflated.

For ProcessEdgesWork::do_work, when tracing, the collector manages the mark stack which reside in the ProcessEdges packets. These reside on the rust heap. Since MMTk uses dual enqueuing, this may also double the stack size. Additionally, although objects must be loaded to perform object scanning, once an object is marked, revisits to this object can simply check the mark bit, which resides in side metadata, and so do not require heap accesses.

For oop_iterate and mark_lines_for_object, I expect that the accesses are to OpenJDK class metadata, since these functions must inspect that to determine the object fields of objects. This metadata lives outside the heap.

For get_descriptor_for_address, it accesses the space descriptor map, which records which space an object resides in. This is stored on the Rust heap. Finally for SweepChunk::do_work, the collector touches metadata, including live and forwarding bits.

This result has multiple implications for how GC developers understand and approach GC optimisations. An important design parameter of Immix is the line size, which was chosen with cache line locality in mind, and thus, it might be worth revisiting this design decision [Blackburn and McKinley, 2008]. Also, any optimisations that focus on improving heap accesses, such as prefetching heap objects, are unlikely to be particularly effective. This is significant as much optimisation effort is targeted at heap accesses, in part due to prevailing wisdom that GC frequently accesses the heap. Given the low frequency of heap accesses, these results indicate possible missed opportunities in improving the latency of various metadata accesses or even moving away from load latency targeted optimisations.

The analysis also amplifies potential latency issues in implementations. For example, the high number of L1 hits in side_metadata_access and trace_object point to frequent metadata accesses in the shared CAS loop; this makes identifying the extra load much easier.

5.3.5 Impact of Workload

Having analysed the overall GC performance, I now examine the variance among different benchmarks and attempt to draw connections between the cache performance and the workload characterisation. I focus on L1 misses. The GC latency overhead attributable to each cache level is shown in Section 5.3.5.



Figure 5.2: L1 misses are responsible for around 15% of the latency overhead on average, however this number is sensitive to the choice of workload. This plot gives the breakdown of the total latency overhead attributable to different cache levels, omitting L1. For cleaner visualisation, I use percentages given to 1dp here.

I immediately notice three outliers in size: biojava, where L1 cache misses only contribute to about 2.2 % of the latency overhead, and h2, where the latency overhead contribution is around 26.8 %. I examine these benchmarks in further detail.

In Table 5.8, I present the results of the microarchitectural analysis for biojava.

cache level	hit rate	load proportion	latency overhead	average penalty
L1	0.994	0.994	0.978	8.51
LFB	0.164	0.000932	0.00671	62.3
L2	0.888	0.00421	0.00706	14.5
L3	0.663	0.000351	0.00238	58.7
RAM	n/a	0.000178	0.00603	292

Table 5.8: The efficacy of biojava is likely explained by a high L1 hit rate of 99.40%. This table presents the cache hit rates, load proportions, latency overheads and average penalties for biojava. All figures are given to 3sf.

I find that the small latency overhead is due to high hit rates at all cache levels. In particular, biojava has a very high L1 hit rate of 99.40%. This result makes sense due to the characterisation and behaviour of the benchmark. Biojava simulates physio-chemical properties of protein sequences. The accesses to the reference array (biological sequences) exhibits regularity (streaming access), and there are only few unique referent (nucleotides), which are likely to reside in the L1 cache. Consequently, hardware prefetching is likely to be effective, further improving the cache performance. Moreover, since the optimisation headroom for biojava is so small, software prefetching schemes for the tracing loop are unlikely to be effective. I discuss these in greater detail in Section 7.2.2 and Section 6.3.2.

Next, I examine h2 in greater depth. The results are shown in Table 5.9.

Table 5.9: h2 has an unexpectedly high number of LFB hits and RAM accesses, both with high average penalties. This table presents the cache hit rates, load proportions, latency overheads, and average penalties for h2, given to 3sf.

cache level	hit rate	load proportion	latency overhead	average penalty
L1	0.977	0.977	0.732	10.6
LFB	0.409	0.00937	0.105	158
L2	0.221	0.003	0.00336	15.9
L3	0.33	0.00348	0.0169	68.3
RAM	n/a	0.00706	0.143	286

For both h2, RAM accesses are the likely culprit. I note that h2 is an outliers with 0.706 % of accesses reaching RAM, resulting in a latency overhead of 14.3 % from these accesses alone. It also has a high number of LFB accesses (0.937 % of loads) which occupies a further 10.5 % of the latency overhead. The LFB penalty of 158 cycles is also much higher than the average of 90.2 cycles across the entire benchmark suite (Table 5.5), indicating that a large number of these hits likely need to wait for RAM accesses. Therefore, in reality, the overhead due to RAM accesses (from both true RAM accesses and LFB accesses which wait for RAM) is likely closer to 20 %. I also note that the hit rates for LFB, L2 and L3 are lower than the average across benchmarks.

The benchmark characteristics may explain this behaviour. h2 performs queries over an in-memory database with a very large object graph. This means that the liveset is quite large and less of it will fit in cache. Additionally, this means there are many work items, resulting in more accesses when handling work queues.

5.3.6 Examining Heap Accesses of Different Workloads

I now perform the per benchmark analysis focusing on heap accesses. Fig. 5.3 presents the percentage of the total latency overhead attributable to heap accesses.



Figure 5.3: Despite only occurring 3.23% of the time, accesses to heap objects are responsible for 10.9% of total load latency. Moreover, this varies for different workloads. For each benchmark, this figure shows the percentage of total load latency cycles occupied by accesses to heap objects for each cache level. Results are rounded to 1dp.

First, I note that for all benchmarks, L1 heap hits contribute to a very small proportion of the latency overhead. This reinforces the previous result (Section 5.3.3 that while rare, heap accesses tend to exhibit much worse locality.

Next, examining outliers, I find that once again, biojava is an outlier in performance, with around 2.5% of load latency attributable to accesses to heap objects. Moreover, the majority of this already small overhead is in L1 hits. This confirms that any heap-based cache optimisations are unlikely to benefit biojava.

On the other end of the spectrum, eclipse, jython, pmd and tradesoap all have a relatively high percentage of the total latency overhead attributable to L1 misses on heap accesses. Here, the key reason is that heap accesses are responsible for a larger proportion of the latency overhead of LFB hits and RAM accesses, as seen in Table 5.10. This is compounded by the fact that for all of these benchmarks, LFB hits and RAM accesses already account for a large proportion of the total latency overhead (Section 5.3.5).

cache level	eclipse	h2	jython	pmd	tradesoap	mean
L1	0.0320	0.00857	0.0139	0.0215	0.0219	0.0202
L2	0.382	0.170	0.318	0.421	0.387	0.431
L3	0.363	0.354	0.593	0.581	0.502	0.511
LFB	0.750	0.457	0.692	0.696	0.671	0.587
RAM	0.748	0.453	0.807	0.775	0.748	0.671

Table 5.10: Focusing on outlier benchmarks, this table shows the proportion of the latency overhead at each cache level which is attributable to heap accesses (3sf). For example, heap accesses account for only 3.20% of L1 latency experienced by GC when running the eclipse benchmark, but 74.8% of RAM latency.

For example, for eclipse, RAM accesses to heap objects are responsible for 5.6% of the total latency cycles. This is because RAM accesses already account for 7.4% of total latency cycles, and of these accesses, 74.8% of them are to objects in the heap.

Interestingly, h2, which was an outlier in the overall GC latency overhead, was closer to the mean when focusing on heap accesses. This is also explained by examining the proportion of the latency overheads of each cache level which was attributable to heap accesses; h2 is well below average on every cache level here. This may be due to relatively good cache locality on heap accesses, or relatively poor locality on stack and metadata accesses. The latter seems plausible due to the frequency of large work packets, which can result in a large number of non-heap memory accesses due to queue operations. This may represent an opportunity for optimisation.

5.4 Fine-Grained Microarchitectural GC Analysis of Immix

I now perform fine-grained analysis and examine function level results. I hypothesise that majority of the collector latency overhead occurs in the tracing loop, where the irregular pointer chasing results in frequent accesses to metadata (for marking), heap objects (for scanning) and other side structures (for processing tracing work queues).

I present the results for the functions which initialise the most load accesses in Fig. 5.4, which presents the percentage of accesses at each cache level which each function is responsible for, and Fig. 5.5, which presents the percentage of latency overhead at each cache level which each function is responsible for. The percentage of total load accesses and load latency respectively are given in the right-most column of both figures. I focus on Fig. 5.5 since this maps directly to load performance, and thus, collector performance, but Fig. 5.4 is included for completeness/reference.

Note that almost all of the functions are the same as in Section 5.3.4, so I refer to the descriptions given there. The only difference is that the function is the aggregation of all instances of the do_work function from other work packet types. I include it mainly to illustrate that the bulk of loads when processing work packets occur during tracing; this is unsurprising since tracing dominates GC time [Huang et al., 2023]. I



Figure 5.4: Loads called from seven key functions dominate accesses to all cache levels. This figure shows the breakdown of accesses at each cache level attributable to each function. Results are given to 1dp.



Figure 5.5: Loads called from seven key functions dominate the latency overhead of all cache levels. This figure shows the breakdown of the percentage of latency cycles at each cache level attributable to each function. Results are given to 1dp.

also include, in grey, the aggregation of all other load accesses.

5.4.1 Analysing Function Level Results

I now analyse the results of Fig. 5.4 and Fig. 5.5. I first observe that over half of the total load latency is attributable to two functions, ProcessEdgesWork::do_work and side_metadata_access. Naively, this seems to indicate that these should both be the focus of optimisation efforts, however, the optimisation opportunities are actually very different.

Focusing first on ProcessEdgesWork::do_work, I observe that it accounts for 33.6% of load latency. Moreover, L1 hits from this function are responsible for approximately $31.7\% \times 85.1\% = 25.8\%$ of total load latency. Therefore, from this function alone, there is about 33.6 - 25.8 = 7.8% of load optimisation headroom. I will compute this number more precisely when I examine ProcessEdgesWork::do_work in depth in Section 5.4.2.

Next, I examine side_metadata_access, which accounts for 30.5% of load latency. However, L1 hits from this function are responsible for $33.8\% \times 85.1\% = 28.8\%$ of total load latency. This means the headroom is actually fairly small (30.5 - 28.8 = 1.7%), and therefore, this is probably not a worthwhile optimisation target. This is unsurprising since side metadata is very dense, so good locality is expected. The hardware prefetcher is also likely to be effective here. Another contributing factor could be that the removable load in the CAS loop, which I discussed in Section 5.3.4, is inflating the number of accesses called from side_metadata_access.

trace_object, oop_iterate and get_descriptor_for_address are the next three functions. They are all predominantly dominated by L1 hits, indicating that these have good cache locality as well. Thus, they are also unlikely to be worthwhile optimisation targets.

mark_lines_for_object is interesting. It contributes to 4.4% of total load latency, including $30.1\% \times 6.01\% = 1.8\%$ of total load latency from LFB hits. Examining Fig. 5.4 tells us that only 25.1% of LFB hits are from accesses called by mark_lines_for_object, indicating that most likely, more than usual of these wait for RAM accesses.

Indeed, checking the average penalty for LFB hits from mark_lines_for_object, I find that it is 108 cycles, which is 19.7 % higher than the average LFB penalty (90.2 cycles, from Table 5.5) and 82.1 % higher than the average L3 penalty (59.3 cycles, also from Table 5.5). This is slightly surprising since RAM accesses from this function are very infrequent. One explanation could be that adjacent load requests occur in high density. This would mean the first load would trigger a RAM access, but the subsequent ones would be slow LFB hits. If the density is high enough, some subsequent accesses may even need to wait for LFB slots, which would further worsen the load latency.

From the analysis, I conclude that the best candidate for further optimisation efforts is ProcessEdgesWork::do_work and the tracing loop. Indeed, this is the focus of prior work, such as scalability studies and software prefetching for tracing collec-

tors [Huang et al., 2023; Atkinson, 2023] and in Chapter 6, I will examine software prefetching for the tracing loop.

5.4.2 Deep Dive: ProcessEdgesWork::do_work and the Tracing Loop

In this section, I explore the cache performance of ProcessEdgesWork::do_work further. To do this, I examine the cache analysis results of ProcessEdgesWork::do_work across different workloads and collector algorithms.

Impact of Workload

First, I examine the percentage of total loads and load latency which is caused by ProcessEdgesWork::do work. The results are shown in Table 5.11.

Table 5.11: For each benchmark, this table gives the percentage of total load accesses and the associated percentage of load latency overheads which are attributable to ProcessEdgesWork ::do work. Numbers are given to 3sf.

Benchmark	L	1 hit	LF	B hit	L	2 hit	Ľ	3 hit	RAM	access	г	otal
	Access	Overhead										
avrora	32.6	29.5	0.123	0.454	0.385	0.639	0.210	1.13	0.0875	2.40	33.4	34.1
biojava	41.3	44.6	0.0398	0.342	0.367	0.601	0.0211	0.138	0.0155	0.531	41.8	46.3
eclipse	33.7	25.4	0.309	2.40	0.459	0.611	0.145	0.751	0.219	5.67	34.8	34.8
fop	30.7	24.3	0.254	2.21	0.517	0.777	0.263	1.47	0.233	6.42	32.0	35.1
graphchi	28.8	25.2	0.154	0.718	0.806	1.27	0.0750	0.420	0.125	2.81	29.9	30.4
h2	32.0	18.1	0.320	3.48	0.135	0.147	0.251	1.22	0.525	10.8	33.3	33.7
h2o	32.1	28.6	0.440	2.53	0.564	0.860	0.150	0.874	0.103	2.63	33.3	35.5
jython	28.4	20.0	0.379	2.59	0.452	0.613	0.377	2.12	0.304	6.41	29.9	31.7
kafka	28.5	21.2	0.313	2.56	0.417	0.590	0.331	1.89	0.207	5.13	29.7	31.4
luindex	30.4	26.0	0.285	1.09	0.613	0.985	0.144	0.856	0.0640	1.89	31.5	30.9
lusearch	30.4	25.1	0.241	1.38	0.403	0.638	0.220	1.39	0.115	3.07	31.3	31.5
pmd	33.3	25.0	0.533	4.45	0.413	0.605	0.348	1.92	0.215	5.71	34.8	37.7
spring	30.8	21.8	0.196	2.08	0.233	0.334	0.237	1.55	0.279	7.06	31.7	32.8
sunflow	28.9	25.2	0.427	1.88	0.700	1.09	0.0709	0.394	0.0792	2.35	30.2	30.9
tomcat	29.6	22.9	0.172	1.56	0.284	0.435	0.315	1.72	0.183	4.53	30.5	31.2
tradesoap	30.5	21.9	0.381	3.15	0.405	0.537	0.266	1.52	0.274	6.75	31.9	33.9
xalan	35.6	32.1	0.815	3.66	0.554	0.857	0.226	1.20	0.107	2.60	37.3	40.5
zxing	22.2	16.7	0.120	0.563	0.419	0.585	0.319	2.00	0.120	2.99	23.2	22.8
mean	31.1	25.2	0.306	2.06	0.451	0.677	0.221	1.25	0.181	4.43	32.3	33.6

I also examine the percentages of the load latency of ProcessEdgesWork::do_work which is attributable to each load level, shown in Fig. 5.6. This helps give an indication on which benchmarks exhibit locality for tracing.

I first examine the RAM accesses and identify a number of outliers. On the low end, only 0.531 % of the RAM load latency of biojava is from ProcessEdgesWork ::do_work. Examining biojava further, I find that it is also a big outlier in that ProcessEdgesWork::do_work is responsible for 46.3 % of the total load latency of the benchmark. I notice that biojava has the highest proportion of L1 hits by far, though this is not unexpected given the benchmark's high hit rate (Section 5.3.5).

On the high end, h2 has a RAM latency overhead of 10.8 % from ProcessEdgesWork ::do_work alone. In Section 5.3.5, this benchmark was an outlier in RAM latency overheads. Comparing the overheads, I conclude that ProcessEdgesWork::do_work is the main culprit for the RAM overheads. However, this story is not the same for



Figure 5.6: The attribution of the load latency of ProcessEdgesWork::do_work to different cache levels for each benchmark is an indicator of which workloads exhibit good locality during tracing. All numbers are given as percentages rounded to 1dp.

LFB latency overheads, which were 10% overall, but are only 3.48% when isolating to considering ProcessEdgesWork::do_work. This suggests that other functions may be bigger culprits for the LFB overhead of h2.

Other interesting benchmarks are graphchi and sunflow, which have high latency overheads of 1.09 % and 1.27 % on L2 hits from ProcessEdgesWork::do_work. This is due to a larger proportion of L2 hits (0.806 % and 0.700 %). However, both also have a low numbers of L3 hits (0.0750 % and 0.0709 % of accesses on the respective benchmarks) and RAM accesses (0.12 % and 0.08 %), and consequently, low L3 overheads of 0.420 % and 0.394 % and low RAM overheads of 2.81 % and 2.35 %. This seems to indicate that on these benchmarks, ProcessEdgesWork::do_work has good locality. Examining the relative percentages of ProcessEdgesWork::do_work accesses in each cache level (Fig. 5.6) for these functions further supports this. I also notice another benchmark luindex, which possibly also exhibits good locality.

Optimisation Headroom

Having concluded that ProcessEdgesWork::do_work is a possible optimisation target, I compute the optimisation headroom for ProcessEdgesWork::do_work. This is given by first computing the current average penalty for each benchmark. This is given by taking the weighted sum over cache levels, using the proportion of accesses as the weight. Then, an optimal penalty is computed by replacing all latency contributions from ProcessEdgesWork::do_work by the average L1 hit penalty for ProcessEdgesWork::do_work for that benchmark. In other words, this computes the new average penalty if all L1 misses could be turned into L1 hits, assuming that the L1 hit penalty remains constant¹. Comparing the speedup $(1 - \frac{optimal}{current})$ gives the

¹I find that this is not the case for software prefetching, which I discuss in Section 6.4.3

headroom.

To illustrate the idea, I give a toy example here. Assume that I have a benchmark which has load attribution given in Table 5.12. For simplicity, I assume all accesses are L1 or L2 hits, or RAM accesses.

Table 5.12: Load attribution of a toy example. I write PE for ProcessEdgesWork::do_work to make the table fit.

Level	PE on L1	PE on L2	PE on RAM	not PE on L1	not PE on L2	not PE on RAM
% of Accesses	20	10	5	35	20	10
Average Penalty	12	20	200	10	30	150

Then, the current average penalty is $0.2 \times 12 + 0.1 \times 20 + 0.05 \times 200 + 0.35 \times 10 + 0.2 \times 30 + 0.1 \times 150 = 38.9$ cycles. However, the optimal headroom achievable through optimisation of ProcessEdgesWork::do_work is $0.2 \times 12 + 0.1 \times 12 + 0.05 \times 12 + 0.35 \times 10 + 0.2 \times 30 + 0.1 \times 150 = 28.7$ cycles. Therefore, the optimisation headroom is $1 - \frac{28.7}{38.9} = 0.26$, or 26%.

The results are given in Table 5.13.

I note that the headroom I compute here is as a percentage of total load latency, rather than total GC speedup. The headroom in wall clock time is likely smaller, though the exact conversion is difficult to determine because of the complexity of modern processors. This is because reduced load stalls can change pipelining behaviours and in turn speed up other non-load instructions. A rough guideline I use is that nominally, the DaCapo benchmarks are roughly 45% memory bound when run with the G1 garbage collector. So I estimate the GC time headroom to be close to half.

5.5 Other Microarchitectural Investigations

In this section, I discuss other non-central work towards understanding the microarchitectural performance and behaviour of GC. I include this work both for completeness and in hopes that they may be insightful in understanding other use cases of these tools.

5.5.1 Misses by Cache Line

I used PEBS to examine the distribution of misses by cache line. This is possible with DLA because each cache miss is recorded alongside its precise address, so the associated cache line can be calculated.

I observed that cache misses were often concentrated on a few lines, which could indicate a high number of conflict misses. One possibility for this could be that the cache design and placement policy is too restrictive, leading to possible cache inefficiences for GC.

Benchmark	Per	alty	Headroom (%)	
	Current	Optimal	-	
avrora	8.87	8.52	3.90	
biojava	8.65	8.55	1.13	
eclipse	11.1	10.2	8.58	
fop	9.90	8.92	9.87	
graphchi	9.10	8.71	4.20	
h2	14.1	12.0	15.0	
h2o	9.63	9.07	5.77	
jython	11.2	10.0	10.6	
kafka	10.8	9.80	9.22	
luindex	9.18	8.82	3.87	
lusearch	9.75	9.19	5.67	
pmd	10.8	9.59	11.6	
spring	11.2	10.1	10.4	
sunflow	9.36	8.92	4.61	
tomcat	10.1	9.34	7.50	
tradesoap	11.4	10.1	11.0	
xalan	10.1	9.39	6.78	
zxing	10.4	9.82	5.41	
mean	9.47	8.67	8.50	

Table 5.13: For each benchmark, this table computes the current optimisation headroom of ProcessEdgesWork::do_work as a percentage of total latency cycles (3sf). For example, perfectly converting all L1 misses to hits for avrora would improve total load latency by 3.90 %.

However, this hypothesis is simply speculation and could only be easily tested in simulators, which are not representative of the full tradeoff space. Moreover, the distribution of cache misses are many degrees removed from GC performance, and what is being observed is a high order effect. Thus, it was difficult to conclude anything from this with regards to GC behaviour. However, I believe these capabilities may be useful for testing the correctness and performance of cache implementations and optimisations.

5.5.2 Last Branch Record (LBR)

Prior to exploring PEBS, I attempted to leverage LBR for precise latency measurements of key code sections. In this section, I explain what LBR is, how it works and its integration into MMTk, and its limitations for my work.

Background

Some Intel CPUs have several Model Specific Registers (MSRs) called Last Branch Records (LBR) which hold a fixed-size ring buffer of the most recent branch decisions [Bakhvalov, 2018]. The CPU can log branches in parallel to executing instructions with zero performance penalty. These registers can be read, albeit at a small penalty, so by collecting a sample, they provide a useful tool for reconstructing and understanding the program's behaviour and control flow. For example, LBR can be used to generate a histogram of hot branches, or understand the typical penalty of taking a slow path. On Coffee Lake, LBR holds 32 branches [Intel Corporation, 2019].

Integration into Heapdumps

I use an LBR analysis toolchain built into an internal testing and development environment called heapdumps [Cai et al., 2025]. The heapdump environment consists of a collector which completes graph traversal on a heap. No collection is performed. There is also no copying and weak references are not supported. The heaps are generated from snapshots of different collection cycles from runs of the DaCapo benchmarks.

While the simplicity of heapdumps does make it a "best case" scenario, so not all performance gains may be transferrable to MMTk, the advantages make it an excellent prototyping environment. The heapdump environment has no mutator, no collection work, and no other aspects of a fully-fledged JVM which I need to integrate with, making it easier to modify and quicker to run. Targeted testing and debugging is also simple since more pathological heaps can be generated and used.

The tool maintains two structures: Branches, which are potential exit points encoded by the branching point and a set of targets, and Blocks, which consist of an entry point and a series of possible Branches. Then, to perform analysis, the tool considers all start and end instruction points, and performs DFS to determine all possible execution paths and their associated statistics.

Issues and Takeaways

I attempted to use LBR to time subsections of the tracing loop. However, I found this difficult because the ring buffer size limitations prevented us unwinding to a sufficient depth to analyse the entire tracing loop. Therefore, I did not pursue LBR approaches further in my work.

Though my experience was eventually unfruitful, I found that the LBR tool provides an easy interface for previously difficult analyses such as precise machine timing of (shallow enough) code sections, estimating branching probabilities and understanding execution counts and distributions of basic blocks.

5.5.3 LFB Full Counters

The performance counter L1D_PEND_MISS:FB_FULL measures the number of stalls caused by waiting for a line fill buffer slot. One idea I had was to examine the ratio $\frac{FB_FULL}{L1D_MISS}$. This indicates how many L1 load misses may be held up by full LFB slots. Additionally, it would be interesting to understand the concentration of when full LFB slots occur to identify points of high demand.

However, this requires first filtering the FB_FULL counts for load misses only (since stores also go through the LFB), and this counter is not supported by PEBS, making it difficult to determine the associated operation. Therefore I do not pursue this idea further or perform important systematic experimentation, simply noting a few interesting observations below:

Observation 1: The ratio $\frac{FB_FULL}{GC}$ is sensitive to heap size

Examining the ratio for 9 different multipliers between 1x and 6x, using the default spread factor in running-ng², I find that for Immix, there is a slight dip from $1 \times$ minheap, and then the ratio increases close to linearly.

I see two possible explanations for the increase in ratio from moderate heaps onward. First, when the heap is larger, fragmentation may be more severe. In particular, this means that more cache lines might be touched, requiring more use of the LFB. Another possibility is that there may be misses deeper in the cache hierarchy. Since these take longer to resolve, which results in more blocking in the LFB. I do not explore these possibilities further in interest of focusing on other investigations, but I note in the event of unexpectedly poor load or prefetch performance, this could be something to reconsider.

The drop off from $1 \times$ heap is trickier to explain. I hypothesise that the collection algorithm, Immix, may be the cause. Immix has opportunistic defragmentation which fails to account for traversal order locality when moving objects. This might be altering the cache behaviours, causing a higher density of misses. Another possibility is that due to the large number of GCs, there might be higher mutator cache pollution, resulting in more pressure on the LFB each GC.

Observation 2: The ratio $\frac{FB_FULL}{GC}$ is sensitive to collection algorithm

Examining the same ratio on, SemiSpace and MarkCompact, I find that the behaviour differs when the collection algorithm is different. On SemiSpace, the ratio sees a large drop off between $1 \times$ and $1.5 \times$ heap, but is near constant after. Meanwhile, the ratio for MarkCompact stays fairly flat throughout, with a few benchmarks experiencing a slight linear ratio increase.

²This means the gaps are smaller at the start and coarser at the end

5.6 Summary

In this chapter, I introduced a novel methodology for microarchitectural analysis of GC performance which provides fine-grained attribution of load latencies to functions, cache levels, and memory segments (heap objects). Leveraging this methodology, I performed in-depth microarchitectural analysis of the cache behaviours and performance characteristics of GC.

I found that L1 misses are uncommon but expensive, accounting for only 2.00% of all loads, but 14.9% of load latency cycles. Moreover, GC cache performance is sensitive to the workload, with load latency overheads as low as 2.2% on biojava, and as high as 26.8% on h2. I note hardware prefetcher efficacy may be a relevant factor here, and investigate this further in Chapter 7.

Surprisingly, I discovered that heap accesses are responsible for only 3.23% of loads and 10.9% of load latency. This has implications for GC optimisations, including possible re-evaluation of Immix line sizes (Section 8.1.7) and potential optimisation opportunities targeting metadata (Section 8.1.10).

By attributing load latencies to individual functions, I also identified that the tracing loop is a key optimisation target, with around 9% of load latency optimisation headroom. This work indicates that the tracing loop may benefit from software prefetching, which I investigate further in Chapter 6.

Software Prefetching for GC

In Chapter 5, I introduced a new microarchitectural analysis methodology for GC, and used it to thoroughly evaluate GC performance. Of particular interest, I found potential headroom for performance optimisations in the tracing loop. Tracing GC algorithms have bad locality due to irregular pointer chasing, and one possible optimisation which targets this is software prefetching. In Atkinson [2023], an empirically determined prefetching scheme for tracing GC was determined. However, this work was completed in a tracing framework, called *auxiliary tracing*, rather than in the context of a fully-functional collector. In this chapter, I implement Atkinson [2023]'s GC prefetching scheme in MMTk. I leverage microarchitectural analysis to investigate its performance and explore opportunity for further optimisations, focusing on the Immix collector.

6.1 Implementing Edge and Object Prefetching in MMTk

I implement software prefetching for edges and objects in MMTk based on schemes described in Atkinson [2023]. These schemes uses the PREFETCHNTA instruction, which is described in Section 2.6.2. This instruction is use to indicate a non temporal prefetch for data that is only used once. I focus on load prefetching only. For concision, I will simply write prefetching to refer to this for the remainder of this chapter.

The prefetching scheme targets two primary accesses during tracing. These are:

- A memory access when **loading the slot** to obtain the object reference. This can be prefetched in the slot processing queue¹. This type of prefetching is called *edge prefetching*.
- A memory access when **loading the object** to scan the field. This can be prefetched in the slot scanning queue². This type of prefetching is called *object reference prefetching*.

¹https://github.com/mmtk/mmtk-core/blob/a592f872f46172b2834228104cfcda3672c3d6ab/src/scheduler/gc_work.rs#L637

²https://github.com/mmtk/mmtk-core/blob/a592f872f46172b2834228104cfcda3672c3d6ab/src/scheduler/gc_work.rs#L838

I use an object reference prefetch distance of 16 and an edge prefetch distance of 32, which were empirically optimal values in Atkinson [2023]. For both implementations, I check the assembly to confirm instructions are inserted.

6.1.1 Minor Modifications for Prefetching Support

In MMTk, edges and object references are VM-specific constructs, which are decoded into memory addresses during GC time, and hence I implement prefetching on the Address type for generality. This also allows for easy extensibility to future prefetch targets, such as metadata. Consequently, implementing prefetching for MMTk requires only a few minor changes:

- 1. Implementing a function to prefetch a given address
- 2. Using this function to implement prefetching for edges and object references
- 3. Adding prefetching to the tracing loop

6.2 **Preliminary Checks**

I perform a few preliminary checks to ensure there are no clear issues with the prefetching implementation.

6.2.1 Instruction Footprint

I first verify that adding prefetching does not greatly increase the number of instructions which must be consumed. This is important because due to the complexity of compiler optimisations, innocent seeming refactors can change behaviour. Examples include dramatically increased instruction count or triggering slow path entries more frequently. This can result in prefetching at the right place and distance without seeing the expected performance gains [Garner et al., 2007].

To do this, I measure the difference in the number of instructions which are retired when adding software prefetching and find that 0.8% of retired instructions are prefetches, and in total, 3.2% more instructions are retired.

6.2.2 Overhead of Issuing Prefetches

In Atkinson [2023], the overhead of issuing only zero prefetches, which is prefetching immediately before use, was around 0.1%. This indicates that the underling mechanism, when performing no useful work, has low overhead. Low cost of prefetch issuance opens the door for use of software prefetching to help resolve other key cache bottlenecks, such as work packet processing and forwarding. Therefore, it is important to verify this result in MMTk. To do this, I keep the same prefetching setup, but set the prefetch distance to zero for both edge and object prefetching. This means that the prefetch instruction is issued immediately before the load, so the difference
in GC time is attributable to the cost of issuing the prefetch. I perform the experiment on the Zen 4.

I find that the prefetch issuance overhead remains low at 0.09 %. This result is promising, and indicates that software prefetching may be feasible for improving load performance of other crucial functions. Outlier benchmarks include h2 and avrora, where the overheads are 15.8 % and 5.3 % respectively. While these are significant overheads, the characteristics of the benchmarks can provide additional insight into why they might have occurred.

The main culprit for h2 is likely variance in the number of GCs (between 4 and 5). In particular, when holding the number of GCs fixed, the build with zero prefetching is actually marginally faster. This is partially explained by an inflated minheap in MMTk for h2, which causes each extra GC to be expensive. Another possible explanation is that h2 is memory intensive, and so, by issuing an extra useless prefetch for each load, the increased memory pressure could be a reason for the slowdown. I debunk this using VTune by examining the memory bandwidth utilisation. I find that h2 does not fully utilise the maximum bandwidth which the memory system can sustain (Fig. 6.1). Other explanations could be that additional prefetches impact



Figure 6.1: The memory bandwidth utilisation of h2 remains low throughout its execution. This figure shows the results of VTune memory bandwidth measurements for an execution of h2.

the behaviour of load history driven caching mechanisms, such as the hardware prefetching or cache eviction policies.

On the other hand, avrora is slightly unstable as it relies on fine grained concurrency, of which the scheduling dramatically affects the heap makeup and performance.

I consider both results as acceptable variance.

6.3 Performance of Software Prefetching in MMTk

In this section, I evaluate the performance of the prefetching scheme in MMTk. I first examine the GC time speedup. To do this, I normalise the GC time with software prefetching to the baseline. I perform the experiment on Coffee Lake and Zen 4. This gives more representative data since specifics of the cache hierarchy differs slightly between Intel and AMD architectures. The results are given in Fig. 6.2 and Fig. 6.3.

I find that the software prefetching scheme performs well, leading to 9% speedup



Figure 6.2: The addition of software prefetching improves GC performance by 9% on Coffee Lake. This figure shows the GC time of Immix with prefetching on the Coffee Lake machine, normalised to the baseline. The geomean is included in orange.



Figure 6.3: The addition of software prefetching improves GC performance by 18% on Zen 4. This figure shows the GC time of Immix with prefetching on the Zen 4 machine, normalised to the baseline. The geomean across benchmarks is in included in orange. I omit tradesoap due to issues encountered during benchmark execution.

of GC on Coffee Lake and 18% speedup on Zen 4. Both are promising results, but larger than I expected. I will discuss these in greater depth in the following sections.

6.3.1 Results for Zen 4

Next, I examine the results for Zen 4, where prefetching resulted in an 18% improvement. This result is surprising due to both the size of the speedup, and the difference with the Coffee Lake results. However, comparing to work by Atkinson [2023], I find that while the speedup differs (the author found a 10% speedup), prefetching efficacy and behaviour tends to be microarchitecturally sensitive, so this result is not completely unfounded.

One possibility is that since Zen 4 has larger caches, prefetches may be more effective since they are less likely to be evicted or cause cache pollution. Inspecting the individual benchmarks, I observe that the trends between benchmarks remain similar on both machines, the size of the speedup is just generally larger on Zen 4. This supports the above theory, since all workloads appear to benefit similar amounts.

A key takeaway from this is that GC performance may differ greatly on different hardware. One future avenue for research is microarchitectural analysis for AMD in a similar manner to Section 5.2. This could help provide a greater understanding of the microarchitectural behaviours and differences between different hardware; gaining insight into the microarchitectural sensitivity in prefetching performance provides just one such example which could benefit from this. I discuss this further in Section 8.1.5.

6.3.2 Results for Coffee Lake

Examining Coffee Lake first, recall that I estimated roughly 9% of total load latency optimisation headroom for the tracing loop (Section 5.4.1). However, GC consists of non-load time, therefore his result appears to indicate prefetching exceeds the headroom. I hypothesise that this is caused by a combination of improved cache hit rates and other microarchitectural complexities which I explore in greater depth in Section 6.4.3.

I note that my result is also consistent with Atkinson [2023] where the author found an 8% speedup in tracing time with this prefetching scheme. This was on an older different version of the DaCapo benchmarks and MMTk, and the experiment was conducted in a simulated trace with a simplified implementation of the dual mark stack. However, the results are within the same ballpark which provides a good sanity check.

Examining the results for individual benchmarks, I note a number of outliers and utilise the insights from the microarchitectural GC analysis in Chapter 5 to help explain these. This helps us better understand prefetching performance, demonstrating the additional observability which microarchitectural analysis provides.

The only benchmark to get slower was biojava, and this is explainable by my analysis in Section 5.3.5, where I found that biojava has minimal optimisation headroom. As a result, the slowdown may be due to the slight overhead of issuing prefetches.

Next, xalan did not improve with software prefetching, but the issue here is different. Examining the microarchitectural analysis in Section 5.3.5, I note that there was ample optimisation headroom (about 40% of load latency was attributable to ProcessEdgesWork::do_work, and 8% to L1 misses in ProcessEdgesWork::do_work). I initially theorise that the prefetch distance may just be suboptimal for this particular benchmark, and hypothesise that a prefetching scheme which dynamically adjusts distance may lead to better performance on xalan. However I find this is not the

case in Section 6.5. Other benchmarks with poor prefetching performance include avrora, where the culprit may be the high overhead of issuing prefetches as seen in Section 6.2.2, and luindex, which I noted in Section 5.4.2 as already having a low L1 miss latency overhead from ProcessEdgesWork::do_work.

Next, I discuss benchmarks which performed well. The largest outlier is h2, which sees 30 % improvement, though with error of almost 10 %.

In general, I notice that high optimisation headroom for ProcessEdgesWork:: do_work (in Section 5.4.2) correlates with a high degree of prefetching efficacy. I verify this by performing a linear regression on this relationship, shown in Fig. 6.4.



Figure 6.4: The optimisation headroom for ProcessEdgesWork::do_work is well-correlated with the efficacy of software prefetching. This graph shows the results of a linear regression on this relationship.

6.4 Unpacking the Performance of Software Prefetching

In this section, I leverage microarchitectural analysis and ineffective prefetch counters to further unpack the performance of software prefetching. This gives a better understanding of when prefetching is particularly effective (or ineffective) and may even uncover opportunities to improve the prefetching scheme.

6.4.1 Microarchitectural Analysis

I perform microarchitectural analysis on MMTk with software prefetching and compare the results to the baseline, which I analysed in Section 5.2. I use the Coffee Lake machine with the Immix collector.

Top-Level Analysis

The results of top-level microarchitectural analysis are given in Table 6.1.

Table 6.1: Software prefetching effectively improves cache performance on almost all key metrics. This table presents the results of microarchitectural analysis of Immix with software prefetching (**swpf**) and without prefetching (**baseline**), with all numbers rounded to 3sf.

cache level	hi	t rate	load pro	portion	latency o	overhead	average penalty		
	swpf	baseline	swpf	baseline	swpf	baseline	swpf	baseline	
L1	0.990	0.980	0.990	0.980	0.927	0.851	8.23	8.89	
LFB	0.336	0.352	0.00332	0.00717	0.0297	0.0601	80.0	90.2	
L2	0.590	0.536	0.00361	0.00637	0.00708	0.0100	17.2	15.9	
L3	0.714	0.623	0.00195	0.00370	0.0124	0.0207	56.9	59.3	
RAM	n/a	n/a	0.000769	0.00236	0.0234	0.0578	272	268	

I first note that the hit rates at L1, L2 and L3 are improved, while the LFB hit rate decreases slightly. Of particular interest, the L1 miss rate is halved. Additionally, the proportion of loads which are LFB hits and RAM accesses, the two most expensive types of loads, have halved. Consequently, the load latency overhead attributable to L1 misses is now only 7.3%, compared to 14.9% from before.

I find that the average penalty across all loads is now $8.23 \times 0.990 + 80.0 \times 0.00332 + 17.2 \times 0.00361 + 56.9 \times 0.00195 + 272 \times 0.000769 = 8.80$ cycles, compared to 10.31 cycles from before Section 5.4.2. This is a 14.6% improvement in load latency time. Interestingly, this exceeds my estimate for headroom. This is because in that computation, I assumed the best case scenario was all loads becoming L1 hits (which had 8.89 cycle penalty on average), but the L1 hits got cheaper with prefetching. I discuss this further in Section 6.4.3. With the new L1 penalty, the predicted remaining headroom for load latency improvement is 7.5%.

Performance by Benchmark

Next, I examine the results separately for each benchmark, focusing on the load latency overhead of L1 misses. This is shown in Fig. 6.5.

Comparing the results to the same figure for the baseline (Section 5.3.5) I find that for prefetching is effective at reducing L1 misses across the entire benchmark suite. In particular, for h2, which represents the highest latency overhead from L1 misses, this value is now 10.6%, compared to 26.8% in the baseline (Section 5.3.5). LFB and RAM overheads, which were dominant in the baseline, are also greatly reduced. These results suggest that software prefetching has successfully covered much of the optimisation headroom for ProcessEdgesWork::do work.



Figure 6.5: Prefetching effectively reduces latency cycles attributable to L1 misses for majority of benchmarks compared to the baseline (Section 5.3.5). This table presents the percentage of total latency cycles attributable each level of the cache for Immix with prefetching. Numbers are given to 1dp.

Microarchitectural Analysis of ProcessEdgesWork::do_work

To verify this, I examine the results of microarchitectural analysis for ProcessEdgesWork ::do_work. First, I reexamine the percentages of the load latency of ProcessEdgesWork ::do_work which are attributable to each load level, shown in Fig. 6.6.



Figure 6.6: For all benchmarks, L1 hits now dominate the load latency cycles of ProcessEdgesWork::do_work. This figure shows the percentages of the load latency cycles of ProcessEdgesWork::do_work which occur at each cache level (1dp).

I note that for all benchmarks, L1 misses from ProcessEdgesWork::do_work now contribute less than 10% of load latency. This is a drastic improvement to the baseline (Fig. 5.6) where L1 misses from ProcessEdgesWork::do_work were responsible for an average of 25.1% of load latency, with single benchmark figures as high as 46.5% (h2). This suggests that prefetching was able to effectively reduce L1 misses and the associated load latency.

This also suggests limited headroom for further optimisation. I verify this by explicitly computing the new optimisation headroom for ProcessEdgesWork::do_work (the method of computation was explained in Section 5.4.2), shown in Table 6.2.

Benchmark	Average	Penalty	Headroom (%)
	Current	Optimal	
avrora	8.19	8.11	0.906
biojava	8.17	8.11	0.756
eclipse	9.14	9.01	1.48
fop	8.74	8.55	2.18
graphchi	8.36	8.27	1.05
h2	9.72	9.60	1.31
h2o	8.56	8.40	1.86
jython	8.98	8.82	1.69
kafka	8.95	8.79	1.81
luindex	8.35	8.19	1.96
lusearch	8.58	8.44	1.65
pmd	8.85	8.68	1.86
spring	9.27	9.04	2.52
sunflow	8.50	8.41	1.10
tomcat	8.42	8.28	1.60
tradesoap	9.20	9.01	2.05
xalan	9.09	8.86	2.53
zxing	9.35	9.14	2.24
mean	8.53	8.37	1.85

Table 6.2: There is very limited remaining headroom for further prefetching optimisations for ProcessEdgesWork::do_work. This table computes the remaining optimisation of ProcessEdgesWork::do_work headroom for the DaCapo benchmarks (3sf).

Indeed, I find that converting all L1 misses to L1 hits would only improve load latency by 1.85%. This indicates very minimal further headroom, so I expect further optimisations, such as dynamic prefetching, to have limited efficacy. I verify this for dynamic prefetching in Section 6.5.

6.4.2 Ineffective Prefetch Counters

AMD machines provide two performance counters which count ineffective prefetches due to hits in the L1 cache or LFB:

- ls_inef_sw_pref.data_pipe_sw_pf_dc_hit: counts prefetches which hit in L1, indicating the prefetch was unnecessary
- ls_inef_sw_pref.mab_mch_cnt: counts hits on LFB, indicating there is already a load request in flight

This provides another avenue for understanding prefetch efficacy. I use these counters to determine what percentage of prefetches are useless. I perform the experiment on the Zen 4 machine with the Immix collector. The results are given in Fig. 6.7.



Figure 6.7: This figure shows the proportion of ineffective prefetches which occur due to hits in L1 or LFB.

I note that, as expected, the proportion of ineffective prefetches due to L1 hits is highly correlated with the frequency of L1 hits of the workload in the baseline (Fig. 5.6). Next, I find that only 14% of prefetches are ineffective due to LFB hits. This is encouraging, as it indicates that the prefetching scheme does not frequently complete unnecessary duplicate work. Overall, 23% of total prefetches are to new cache lines, which is comparable to the available prefetch headroom for ProcessEdgesWork ::do_work (Fig. 5.6), thus indicating good prefetch coverage.

6.4.3 Reevaluating Why the Performance Exceeds the Apparent Headroom

The performance of the software prefetching scheme appears to exceed the headroom computed in Section 5.4.1. In this section I explore possible explanations for this. The core of my hypothesis is that, due to the complexities of modern CPUs, the

improved load latency and reduced stalls from software prefetching in turn benefits other components of GC in non-obvious ways.

I propose three main explanations for explaining why the apparent discrepancy may not be contradictory:

- 1. Improving load latency reduces memory stalls in the instruction pipeline, which in turn can help superscalar optimisations. This is because by reducing the constraints, the CPU has more freedom to rearrange instructions.
- 2. In Section 6.4.1, I observed that on average, L1 hits got faster. This helps to explain the apparent discrepancy since the headroom analysis assumes that the L1 hit penalty remains unchanged. This also indicates a possible limitation in this methodology for computing headroom. I hypothesise that the cause of this improvement in the L1 hit penalty is due to nonobvious microarchitectural complexities and side effects.
- 3. There may be some "observer effect" whereby the use of measurement tools like perf mem indirectly affects the GC behaviour.

To test the first hypothesis (improved instruction pipeline), I measure the instruction level parallelism (ILP) with software prefetching and compare this to baseline MMTk. I find that the ILP improves by 2%. Moreover, on many of the benchmarks where the headroom was exceeded by the most, such as jython, pmd, h2, eclipse and fop, the improvement is higher, ranging between 3% on fop and 7.8% on h2. This supports the first hypothesis, and helps to partially explain the apparent discrepancy.

Understanding the second hypothesis is more nuanced, since it is unclear why the L1 hits are cheaper with prefetching. I note that despite being private, the L1 cache can have some variance in access simply due to the complexity of modern CPU design. Therefore, it appears plausible that the L1 access latency could be unintentionally affected by software prefetching. I suggest two possibilities below which were raised through private communication with Trevor E. Carlson:

- 1. In Intel, L1 caches are banked for efficiency. Reducing the cache misses via prefetching can lead to changes in the timing of cache accesses. This in turn can have implications for the cache latency in a banked cache design.
- 2. Interference with other buffers, such as the memory disambiguation unit in the load-store queue, can affect the L1 hit latency.

Other factors include the complex and nuanced behaviour of the specific load latency counter, which Intel states may observe high latencies in certain conditions despite a close source [Intel Corporation, 2016]. There may also be other microarchitectural side effects that cannot be isolated which I have not considered.

I defer further research to future work (Section 8.1.6), but note that the complexity in understanding and explaining this result further emphasises that hardware is changing, and both difficult to study and understudied.

6.5 Exploring Dynamic Prefetching

Fixed-distance software prefetching is limited in its ability to account for and exploit variance in hardware, workloads and GC performance through time. Dynamic adjustment of the prefetch distance offers a solution to this. However, the results of Section 6.4.1 indicate limited headroom for further optimisation. I hypothesise that in the case of GC, dynamic prefetching is unlikely to be worthwhile. In this section, I concretely explore the headroom for further gains through dynamic prefetching and attempt to verify this hypothesis.

6.5.1 Performing Headroom Analysis

To determine whether dynamic prefetching is viable, I consider two classes of possible dynamic prefetchers:

- 1. Benchmark-level dynamicism: this involves determining the optimal distance for a given workload, and then using this distance throughout the execution of the workload. It would most likely use an initial data gathering phase. This type of dynamic prefetcher can capture variation among different workloads.
- 2. GC-level/epoch-level dynamicism: this involves reevaluating the optimal distance at each GC epoch, and then using this distance throughout that GC. It would most likely rely on history-driven heuristics which use data collected during previous GCs. This type of dynamic prefetcher can capture variation in the execution of a workload. For example, long running workloads like web applications requiring user input may exhibit different and non-deterministic behaviour at different points, and would benefit from this type of dynamic prefetching.

Note that both classes of dynamic prefetchers are able to account for variation caused by the choice of hardware and collector.

I consider edge and object reference prefetching separately and for each, I examine the headroom for both classes of dynamic prefetching. For all experiments I use the Coffee Lake machine with the Immix collector. I will discuss my methodology for evaluating the headroom of both classes of dynamicism in the following sections.

Headroom of Benchmark-Level Dynamicism

To compute the headroom of benchmark-level dynamic prefetchers, I select a set of prefetching configurations (no prefetching, 0, 1, 2, 4, 8, 16, 32 and 64) and measure the GC time of each benchmark. I determine how much speedup I can get by picking the locally optimal values for each benchmark instead of the current empirically determined prefetch distance.

GC-Level Dynamicism

To compute the head room of GC-level dynamic prefetchers, I modify MMTk to print out the GC time of each GC epoch. Using the same set of prefetching configurations (no prefetching, 0, 1, 2, 4, 8, 16, 32 and 64), I compute the optimal prefetch distance for each GC epoch. The headroom is determined by comparing the total GC time by picking the local optima each epoch compared to the current empirically determined prefetch distance. The results are taken as the mean across 10 invocations. Since the variance in GC time can vary greatly in an epoch, I drop any outliers outside 2 standard deviations.

6.5.2 Headroom Analysis for Edge Prefetching

I present the results for the two dynamic edge prefetching schemes in Fig. 6.8.



Figure 6.8: This figure shows, for each benchmark, the headroom available for benchmark-level and GC-level dynamicism for edge prefetching.

I also present the locally optimal distances for benchmark-level dynamicism in Table 6.3.

I find that when focusing on benchmark-level dynamicism, the optimisation headroom is 2.42%. Furthermore, narrowing the scope to GC-level dynamicism only provides minor additional gains, with a total optimisation headroom of 3.51%.

A key outlier is avrora which sees an 8% just from using distance 2 prefetching alone. Moreover, introducing GC-level dynamicism increases the headroom to just over 14%. I hypothesise that the mutator is to blame here, as avrora has fine grained concurrency and locking behaviours which are sensitive to scheduling. This means that the heap and performance can vary greatly between runs, making it generally a more unstable benchmark.

Benchmark	Normalised GC Time at Optimal Distance	Optimal Distance
avrora	0.920	2
biojava	0.945	1
eclipse	0.987	8
fop	0.939	8
graphchi	0.986	8
h2o	0.976	2
jython	0.992	8
luindex	0.982	8
lusearch	0.970	4
pmd	0.983	8
spring	0.924	8
sunflow	0.964	8
tomcat	0.982	64
tradesoap	1.000	32
xalan	0.971	1
zxing	0.988	32

Table 6.3: This figure shows the optimal distance for edge prefetching for each benchmark and the GC time (3sf) at that prefetch distance normalised to the current prefetch distance of 32.

Another interesting observation is that for many benchmarks, the optimal distance is 8, and moreover, 32 is the optimal distance for only two benchmarks. This is likely because the globally optimal prefetching configuration was chosen by considering results across different pairings of edge and object reference prefetching across different hardware.

For three benchmarks (biojava, spring and tradesoap), the optimal prefetch distance for that benchmark is also the optimal distance for all GC epochs. In particular, for tradesoap, a distance 32 prefetch is optimal through all GCs. Moreover, more most benchmarks, the improvement through GC-level dynamicism is very marginal, and more all but five benchmarks, the headroom is below the geomean of 3.51 %.

My results indicate that fully-fledged GC-level adaptive prefetching may not be particularly fruitful, since it requires us to be able to accurately determine the optimal distance for a GC epoch ahead of time, and even if done perfectly, will only yield limited performance gains. Instead, if attempting to build a dynamic prefetcher, the simpler adaptive approach of choosing the optimum for a benchmark is more profitable. I believe the headroom here is still too little to be worthwhile, but discuss the pathway to building such a prefetcher in 8.2.

6.5.3 Headroom Analysis for Object Reference Prefetching

I present the headroom for the two dynamic object reference prefetching schemes in Fig. 6.9.



Figure 6.9: This figure shows, for each benchmark, the headroom available for benchmark-level and GC-level dynamicism for object reference prefetching.

I also present the locally optimal distances for benchmark-level dynamicism in Table 6.4.

I first note that both dynamic schemes are less effective than for edge prefetching, with benchmark-level dynamicism providing only 0.75% speedup and GC-level dynamicism providing only 2.28% speedup. One reason for this is that the globally optimal distance of 16 is also the benchmark-level optima for 8 benchmarks. This suggests the globally optimal prefetching configuration is nicely compatible with my particular hardware choice.

One interesting outlier here is biojava, which has a benchmark-level optimal distance of 16 but 5 % headroom when performing GC-level dynamicism. I recall that the hit rates for biojava are already high without prefetching (Section 5.3.5), and moreover, prefetching was ineffective, likely due to generating unnecessary traffic (Fig. 6.5). As a result, it is reasonable that a dynamic approach which can maximise the minimal existing headroom could be effective.

Overall, as before, I believe the headroom is too little to be realistically exploitable so I do not pursue dynamic prefetching further for object references.

Benchmark	Normalised GC Time at Optimal Distance	Optimal Distance
avrora	0.952	2
biojava	0.948	16
eclipse	0.984	64
fop	0.996	16
graphchi	0.973	32
h2o	0.959	16
jython	0.986	8
kafka	0.965	8
luindex	0.992	8
lusearch	0.974	16
pmd	0.996	16
spring	0.997	4
sunflow	0.969	16
tomcat	0.970	32
tradesoap	0.985	16
xalan	0.973	16
zxing	0.996	64

Table 6.4: This figure shows the optimal distance for object reference prefetching for each benchmark and the GC time (3sf) at that prefetch distance normalised to the current prefetch distance of 16.

6.6 Summary

In this chapter, I used microarchitectural analysis to understand the performance of a software prefetching scheme for the Immix collector in MMTk.

I found that adding software prefetching leads to a 9% speedup of GC on Coffee Lake and 18% speedup on Zen 4. Moreover, I found that the efficacy of prefetching on individual benchmarks has a high degree of correlation ($R^2 = 0.708$) to the optimisation headroom of ProcessEdgesWork::do_work computed in Section 5.4.2, highlighting the usefulness of microarchitectural analysis.

Through microarchitectural evaluation, I discovered that the efficacy of prefetching was due to a roughly 50% improvement in key cache performance metrics, such as the frequency and latency of L1 misses. Furthermore, on all benchmarks, the load performance of the tracing loop improves drastically, with 94.4% of load latency cycles of ProcessEdgesWork::do_work attributable to L1 hits, an increase of 26.0% from the baseline (74.9%, see Fig. 5.6).

Surprisingly, I also found that software prefetching improves the load latency of L1 hits. This emphasises how increasingly complex hardware and cache systems are both difficult to study and understudied, and highlights an important area for future research (Section 8.1.6).

Finally, I evaluated the headroom for two types of dynamic prefetching schemes (benchmark-level dynamicism and GC-level dynamicism) for both edge prefetching and object reference prefetching. I found that the best of these schemes has only 3.51% headroom, and conclude that dynamic prefetching is not currently necessary or profitable.

Software Prefetching for GC

Hardware Prefetching for GC

In Chapter 6, I applied microarchitectural analysis methodologies to evaluate and understand the performance of software prefetching for tracing collectors in MMTk. In this chapter, I leverage microarchitectural analysis alongside other analysis techniques to reevaluate and understand the efficacy of hardware prefetchers for GC.

7.1 Experimental Setup

I use an identical pair of Coffee Lake machines. On one machine, I turn off all the hardware prefetchers. These are all controlled by the MSR_MISC_FEATURE_CONTROL register, which has register address 420 [Intel Corporation, 2019]. The prefetchers are:

- L2 stream prefetcher, which fetches additional lines of code/data (controlled by 420:0)
- L2 adjacent cache line prefetcher, which retrieves the pair line which completes the 128-byte aligned chunk (controlled by 420:1)
- L1 stream prefetcher, which fetches the next cache line into L1 data cache (controlled by 420:2)
- L1 stride prefetcher, which uses sequential load history to decide whether additional lines should be fetched (controlled by 420:3)

Further details of these prefetchers were explained in Section 2.6.1.

7.2 Efficacy of Hardware Prefetchers

First, I attempt to quantify the efficacy of the hardware prefetcher. To do this, I measure the GC time without hardware prefetching and compare it to a standard baseline run. I examine the full spectrum of collector algorithms because different algorithms have different locality properties, and so I expect the algorithm choice to directly affect the efficacy of the prefetchers.

The results are normalised to the baseline machine, which has all hardware prefetchers on by default. I take the geometric mean over all benchmarks (discussed in Chapter 4).

The results are shown in Fig. 7.1.

GC time without hardware prefetching as a factor of baseline MMTk												
avrora -	1.23 1.11-1.37	1.27 1.18-1.37	1.20 1.13-1.28	1.17 1.03-1.33	1.35 1.23-1.49	1.37 1.35-1.39		- 2.4				
biojava -	1.84 1.81-1.86	1.73 1.71-1.75	1.83 1.81-1.85	2.44 1.83-3.13	1.93 1.91-1.95	1.80 1.78-1.81						
eclipse -	1.22 1.17-1.26	1.23 1.22-1.25	1.29 1.28-1.30	1.45 1.37-1.53	1.47 1.44-1.49	1.78 1.76-1.80						
fop -	1.20 1.19-1.21	1.22 1.19-1.26	1.24 1.19-1.29	1.35 1.29-1.40	1.51 1.44-1.57	1.67 1.65-1.68		- 2.2				
graphchi -	1.43 1.42-1.43	1.64 1.64-1.65	1.63 1.62-1.64	1.77 1.77-1.78	1.70 1.67-1.73	1.59 1.58-1.61						
h2 -	1.37 1.35-1.38	1.30 1.29-1.31	1.33 1.10-1.62	1.29 1.28-1.30	1.81 1.79-1.83	1.91 1.91-1.92		- 2 0				
h2o -	1.25 1.22-1.27	1.32 1.29-1.34	1.37 1.32-1.41	1.46 1.31-1.62	1.60 1.57-1.62	1.31 1.28-1.35		- 2.0				
jython –	1.14 1.13-1.15	1.22 1.21-1.23	1.23 1.22-1.24	1.33 1.31-1.34	1.44 1.42-1.47	1.82 1.81-1.83		a				
놑 kafka -	1.17 1.12-1.21	1.26 1.13-1.40	1.27 1.17-1.36	1.16 1.10-1.24	1.55 1.47-1.62	1.79 1.78-1.80		≓ 1.8 පු				
토 luindex -	1.17 1.16-1.18	1.21 1.20-1.22	1.30 1.28-1.31	1.22 1.21-1.23	1.42 1.41-1.44	1.22 1.21-1.22		ised 0				
🗟 lusearch -	1.13 1.12-1.14		1.25 1.23-1.27	1.13 1.10-1.15	1.48 1.48-1.48	1.54 1.54-1.54		rmal				
pmd -	1.23 1.22-1.25	1.26 1.24-1.29	1.28 1.26-1.30	1.29 1.18-1.40	1.48 1.44-1.51	1.75 1.65-1.86		- 1.6 ²				
spring -	1.19 1.17-1.20	1.19 1.16-1.22	1.42 1.22-1.67	1.28 1.26-1.30	1.57 1.57-1.57	1.75 1.75-1.76						
sunflow -	1.36 1.36-1.37		1.47 1.46-1.49	1.36 1.35-1.36	1.59 1.58-1.59	2.02 2.02-2.02						
tomcat -	1.22 1.21-1.23	1.24 1.23-1.25	1.27 1.17-1.39	1.23 1.21-1.25	1.56 1.56-1.57	1.55 1.55-1.55		- 1.4				
tradesoap -	1.21 1.19-1.22	1.23 1.22-1.25	1.35 1.29-1.41	1.15 1.14-1.16	1.50 1.47-1.53	1.75 1.74-1.76						
xalan -	1.25 1.23-1.27	1.27 1.27-1.28	1.29 1.26-1.33	1.92 1.82-2.01	1.38 1.37-1.38	1.42 1.42-1.42		1.2				
zxing -	1.20 1.16-1.24	1.15 1.04-1.30	1.18 1.06-1.35	1.06 1.02-1.11	1.30 1.23-1.37	1.62 1.60-1.64		- 1.2				
geomean -	1.26 1.23-1.28	1.29 1.25-1.33	1.34 1.27-1.41	1.36 1.29-1.43	1.53 1.49-1.56	1.63 1.62-1.65						
	weep	mit	mnit	minit	CO3CE	N ^{3Č-}						
	MartSu	NWII.	N .	Gentin	Semisr	arcon						
			0.11			412						

Figure 7.1: The efficacy of hardware prefetching for GC is correlated with how collectors move objects, and also sensitive to the choice of workload. This heatmap shows the GC time without hardware prefetching normalised to baseline MMTk for six different collectors across the DaCapo benchmarks. Two configuration pairings failed to record results and are omitted.

I note that for all collector algorithms, disabling hardware prefetching leads to significant slowdown. Moreover, the size of the slowdown appears highly correlated with the workload and collector algorithm. I will discuss these factors in Section 7.2.1 and Section 7.2.2.

I further note that the extent of slowdown during GC is much larger than the slowdown experienced during mutator activities, which is shown in Fig. 7.2.

This may be because collection is dominated by the tracing loop, which, with good load balancing, generally uses large work packets which the hardware prefetcher may be able to effectively prefetch. Meanwhile, the mutator workloads, which are complex real-world object-oriented (OO) programs, may not provide as much structure for the hardware prefetchers. I focus on GC time in this thesis, however this finding indicates that further investigation into hardware prefetching efficacy on OO workloads may be interesting and necessary (see Section 8.3.5).

Mutator time without hardware prefetching as a factor of baseline MMTk



Figure 7.2: Surprisingly, the performance of mutators appears to be mostly unaffected by the absence of hardware prefetchers. This heatmap shows the mutator time without hardware prefetching normalised to baseline MMTk for six different collectors across the DaCapo benchmarks. Two configuration pairings failed to record results and are omitted.

7.2.1 Impact of Collector Algorithm

I notice that the slowdown when disabling the hardware prefetcher is correlated to how much collectors move objects. This can be seen by comparing the results for moving collectors (SemiSpace and MarkCompact, which have overheads of 1.53x and 1.63x) to nonmoving collectors (MarkSweep and NMImmix, which have overheads of 1.26x and 1.29x). Another useful comparison is NMImmix and Immix, which experience overheads of 1.29x and 1.24x respectively without prefetching, since the only difference is that NMImmix does not perform opportunistic copying defragmentation. This further supports the observation.

I theorise that this is because moving collectors structure the heap in ways which are better exploitable by the hardware prefetchers, which allows the prefetchers to be more effective at improving cache hits. This is because moving collectors tend to place objects that are allocated close in time close in memory, allowing the hardware prefetchers to better exploit the locality. For example, given an array of pointers to newly created objects, MarkCompact would place these close in memory, thus assisting the stride prefetchers. Additionally, I expect hardware prefetching to benefit when memory is less fragmented, since the objects I are tracing are more likely to be prefetched by stream prefetchers.

Examining L1 Hits and L3 Misses

To help verify this explanation, I use performance counters to examine general metrics for cache behaviour. The metrics are gathered and computed analogous to those in Section 5.1.2 and the results are shown in Table 7.1.

Table 7.1: This table compares L1 cache hit rates, L3 miss rates, and RAM access rates with and without hardware prefetching.

Collector	L1 hit	rate (%)	L3 miss	s rate (%)	RAM access rate (%)		
Collector	with	without	with	without	with	without	
Immix	97.55	96.22	34.90	45.34	0.20	0.54	
NMImmix	97.67	96.40	33.69	46.60	0.16	0.50	
GenImmix	96.84	95.04	37.66	40.81	0.14	0.42	
MarkCompact	98.94	97.35	32.00	72.48	0.04	0.53	
MarkSweep	97.07	96.25	36.89	47.81	0.24	0.62	
SemiSpace	97.16	95.10	32.25	45.38	0.10	0.57	

I observe that for all collectors, the L1 hit rate with hardware prefetching is higher. If I omit GenImmix, the size of the gap is also highly correlated with how much the algorithm moves objects: the drop in hit rate is largest for SemiSpace and MarkCompact, which have differences of 2.06 % and 1.59 %, and smallest for MarkSweep, where the hit rate is just 0.82 % worse. This is consistent with the efficacy measurements, and strengthens the explanation that moving collectors exhibit good locality, which in turn helps the L1 hardware prefetchers.

Another interesting observation is that Immix and NMImmix have very similar cache behaviours. This is likely because opportunistic copying is rare in Immix on moderately large heaps (I use $3\times$).

The L3 miss rates are higher across the board without hardware prefetching. Consequently, the frequency of RAM accesses also increases. This is expected, since the L2 prefetchers prefetch data to both L2 and L3 Section 2.6.1. Key outliers are GenImmix, where the miss rate improves only by 3.15 % with prefetching, and MarkCompact, where the L3 misses are nearly doubled from 32 % to 72.48 %.

For GenImmix, this is likely due to nursery collections which trace a smaller part of the heap. This means tracing occupies a smaller proportion of GC for GenImmix than other collectors. One possibility is that tracing provides more exploitable access patterns, such as retrieval off the mark stack and linearly scanning an object for references, than other GC components.

For MarkCompact, this result is likely due to an increase in the number of L3 accesses (since hardware prefetching appears to be very effective for MarkCompact), and additionally, a greater number of L3 misses. This also helps explain why MarkCompact improves drastically with hardware prefetching.

7.2.2 Impact of Workload

In this section, I examine the results at a per benchmark-level to try understand if certain workload characterisations are more suited to hardware prefetching.

One immediate observation is that for all collectors, biojava performance suffers the most without hardware prefetching. Other interesting benchmarks include graphchi, where the hardware prefetcher is highly effective, and kafka, luindex and zxing, where the slowdown without hardware prefetching is less pronounced.

Number of References

To understand these outlier results better, I consider the distribution of contiguous runs of references per object. For an array of pointers, I expect that hardware prefetching should be highly effective. This is as opposed to arrays of slots, where I need either software prefetching or more sophisticated data memory dependent prefetchers to handle the extra indirections. Therefore, I hypothesise that there is correlation between the distibution of runs of references per object and hardware prefetch efficacy.

To test this hypothesis, I consider the length of contiguous runs of outgoing references encountered in tracing for each benchmark, which was measured for Cai et al. [2025]. This tells us about the sizes of object arrays. These results are provided in Fig. 7.3 with permission from the author.

	ŝ	re				dra			mi					+	ð			4	028	
rank	chunk	mear	1 aure	ra bio	Part case	santecti	PSE top	253	phe h20	jme	W	on val	ea win	de' Inse	are priv	s spi	ne sun	ilo trat	ese +ali	in tring
1	$[2^1, 2^2)$	22.57	17.24	0.45	15.40	11.63	18.50	56.23	12.04	18.23	34.76	26.10	18.05	18.57	25.79	22.39	58.01	23.54	9.15	20.25
2	$(2^2, 2^3)$	14.63	13.96	0.46	14.99	12.66	24.53	6.06	12.10	19.59	11.53	23.00	23.78	31.10	10.25	16.23	5.06	12.52	6.23	19.36
3	$(2^4, 2^5)$	13.58	12.70	0.63	5.93	13.89	20.43	6.78	10.29	24.70	10.42	17.01	26.35	25.67	3.63	17.84	6.15	16.99	6.21	18.89
4	$(2^3, 2^4)$	10.10	5.49	0.19	11.98	8.88	8.26	3.83	13.87	15.67	14.88	9.57	10.34	11.70	13.78	17.87	2.09	11.37	3.81	18.29
5	$[2^{12}, 2^{13})$	9.79		97.85	10.13	0.71	0.24		0.27		0.76	0.58		0.70	0.17	0.23	17.86	11.14	35.57	
6	$[2^0, 2^1)$	6.00	7.53	0.20	9.60	5.29	9.02	2.07	7.06	7.11	7.36	7.43	6.58	5.85	4.54	9,94	1.93	6.64	2.19	7.62
7	$[2^{13}, 2^{14})$	5.29	26.60		0.98	6.86	0.94		33.30			1.03			0.31	0.43		1.72	23.10	
8	$[2^6, 2^7)$	3.77	1.56	0.03	0.76	7.23	3.26	0.59	0.79	2.50	1.90	1.28	2.43	0.88	37.48	1.20	0.60	2.12	1.31	1.86
9	$[2^{10}, 2^{11})$	2.93	1.91	0.05	8.27	1.52	2.82	0.84	1.19	3.31	3.37	6.66	2.88	1.09	0.34	1.52	4.92	7.87	0.77	3.32
10	$[2^8, 2^9)$	2.46	3.22	0.03	16.82	2.59	4.40	0.76	1.37	1.83	1.63	2.11	2.07	0.76	0.22	1.02	0.87	0.97	1.26	2.33
11	$(2^5, 2^6)$	2.00	1.25	0.03	0.72	7.18	1.81	0.58	0.60	2.24	6.15	1.56	2.04	1.09	2.48	1.69	0.60	1.70	2.33	1.87
12	$2^7, 2^8$	1.87	1.90	0.03	0.49	4.59	1.89	0.66	4.58	2.08	2.28	1.09	1.80	1.08	0.31	0.88	1.18	0.97	5.94	1.84
13	$[2^{11}, 2^{12})$	1.37	6.01	0.03	0.27	2.81	1.70	0.72	0.67	1.27	0.35	0.78	2.42	1.04	0.13	0.94	0.47	1.30	0.38	3.45
14	$[2^9, 2^{10})$	1.11	0.63	0.01	3.01	1.77	2.20	0.26	1.14	1.48	0.89	1.80	1.26	0.48	0.08	0.82	0.27	1.16	1.76	0.93
15	$\begin{bmatrix} 2^{17}, 2^{18} \end{bmatrix}$	0.95				0.37		16.76												017.0
16	214 215	0.62			0.63	2.22		3.84	0.73						0.27	3.56				
17	[2 ¹⁵ 2 ¹⁶]	0.33			0.00	3.96		0.01	0.70						0.21	1 72				
18	216 217	0.30				5.70					3 7 2				0.21	1 72				
19	[219 220]	0.16				292					5.72					1.72				
20	[2 ¹⁸ 2 ¹⁹]	0.16				2.92														
20	12,2)	0.10				2.92														

Figure 7.3: This figure shows the proportions of objects in contiguous runs of outgoing references of different lengths. For example, biojava has 97.85% of objects in large arrays between 2^{13} and 2^{14} . The array sizes are sorted from most to least common.

Explaining Outlier Benchmarks

I now refer back to the outlier benchmarks.

For biojava, I find it has many objects are in large arrays between 2¹³ and 2¹⁴ (97.85%). This is likely due to high levels of predictability in access patterns, which I discussed in Section 5.3.5. This result also supports my hypothesis that the strong L1 cache hit rates of biojava were strongly correlated to effective hardware prefetching. I will examine this further for Immix in Section 7.3.2.

Similarly, graphchi is a benchmark which has good hardware prefetcher performance for all collector algorithms and I note that while 56% of objects in small reference arrays, it also has 16.76% of objects in massive arrays (2^{17} to 2^{18}), which tend to be more suited to stride prefetching.

A similar relationship holds up for kafka, luindex and zxing, which have 83.11%, 85.10%, and 84.41% of objects in arrays smaller than 2^5 . Therefore, these are also explainable by the reference arrays.

7.3 Microarchitectural Analysis of Hardware Prefetchers

I use some of the analysis methodologies in Chapter 5 to perform microarchitectural analysis and compare the results to the baseline (which I discussed in depth in Section 5.2). I hope this will give further insight into the performance of hardware prefetching for GC. In this section, I focus on Immix.

7.3.1 Top-Level Microarchitectural Analysis for Immix

I perform top-level microarchitectural analysis for Immix and compare the results to the baseline (which I discussed in depth in Section 5.3).

Table 7.2: Cache performance suffers under almost all metrics with the hardware prefetchers disabled. This table shows top-level microarchitectural analysis for Immix with and without hardware prefetching, with values rounded to 3sf.

cache level	hit rate		load pro	oportion	latency o	verhead	average penalty		
	without	with	without	with	without	with	without	with	
L1	0.971	0.980	0.971	0.980	0.713	0.851	9.68	8.89	
LFB	0.504	0.352	0.0150	0.00717	0.157	0.0601	141	90.2	
L2	0.226	0.536	0.00314	0.00637	0.00426	0.0100	17.6	15.9	
L3	0.479	0.623	0.00525	0.00370	0.0215	0.0207	54.1	59.3	
RAM	n/a	n/a	0.00555	0.00236	0.105	0.0578	256	268	

First, I note that without hardware prefetching, hit rates at L1, L2 and L3 all suffer. This indicates that the hardware prefetchers are effective at reducing misses at these levels. Surprisingly, the LFB hit rate is improved. One possible explanation for this is that without the hardware prefetcher, some lines which would usually reside in cache may instead need to be loaded from memory, and accesses to these lines while



Figure 7.4: Disabling hardware prefetching degrades performance by increasing the load latency of L1 misses for all benchmarks. This figure shows the percentage of total load latency which is occupied by L1 misses for Immix without hardware prefetching (1dp).

the RAM access occurs can result in LFB hits. This explanation is supported by the high average LFB penalty and the increased frequency of RAM accesses.

The distribution of load accesses among cache levels is also different. Of particular interest, the percentage of accesses which are L2 hits is halved (0.637% to 0.314%) and the percentage of L3 hits and RAM accesses are increased. This further supports that the prefetchers are effective at reducing accesses to lower levels (L3 and RAM) of the memory subsystem.

7.3.2 Examining Outlier Workloads

I also present results for the percentage of total load latency which is occupied by L1 misses in Fig. 7.4. This is comparable to Section 5.3.5, which gives results for the same experiment with hardware prefetchers.

I use this, alongside the full top-level microarchitectural data for selected benchmarks, to examine the benchmarks where disabling hardware prefetching was most (avrora and zxing) and least detrimental (biojava and graphchi) to attempt to gain further insight into hardware prefetcher efficacy.

Outliers: biojava and graphchi

In Section 5.3.5, I hypothesised that hardware prefetcher efficacy could be a factor in the high hit rate and low L1 miss latency overheads which biojava exhibits. This explanation is further supported by the strong performance degradation when the hardware prefetchers are disabled (Section 7.2.2). I verify this by examining the toplevel microarchitectural performance of biojava without the hardware prefetcher. The results are given in Table 7.3.

Table 7.3: The hardware prefetchers likely play a key role in the cache performance of biojava (a key outlier in performance in Section 5.3.5). This table presents the top-level microarchitectural analysis for biojava with and without hardware prefetching.

cache level	hit rate		load pro	oportion	latency	overhead	average penalty		
	with	without	with	without	with	without	with	without	
L1	0.994	0.99	0.994	0.99	0.978	0.84	8.51	10.8	
LFB	0.164	0.448	0.000932	0.0046	0.00671	0.0703	62.3	194	
L2	0.888	0.112	0.00421	0.000636	0.00706	0.000845	14.5	16.9	
L3	0.663	0.145	0.000351	0.000733	0.00238	0.00298	58.7	51.5	
RAM	n/a	n/a	0.000178	0.00431	0.00603	0.0857	292	252	

Indeed, I find that while the L1 hit rate only drops by 0.4%, the latency overhead of L1 misses is now 16%. This is mostly attributable to RAM accesses and LFB hits, which have a high average penalty of 194 cycles, implying that many wait for RAM. Another key observation is that the L2 hit rate is only 11.2%, indicating that the L2 hardware prefetchers are quite effective for biojava. Moreover, inspecting Fig. 7.4 and comparing to Section 5.3.5 highlights the hardware prefetcher efficacy, as the percentage of load latency attributable to L1 misses climbs from around 3% to just over 15%.

This helps to verify my hypothesis that the load performance of biojava is likely attributable to good locality, and that the hardware prefetcher plays a key role in exploiting this.

Next, I examine graphchi. The microarchitectural analysis results are given in Table 7.4.

Table 7.4: The hardware prefetchers also play a key role in the cache performance of graphchi,
improving the hitrate of all three levels of cache. This table presents the top-level microarchi-
tectural analysis for graphchi with and without hardware prefetching.

cache level	hit rate		load pro	oportion	latency o	overhead	average penalty		
	with	without	with	without	with	without	with	without	
L1	0.982	0.973	0.982	0.973	0.913	0.711	8.46	10	
LFB	0.347	0.543	0.00637	0.0148	0.0317	0.161	45.2	150	
L2	0.774	0.265	0.00928	0.00329	0.0153	0.00375	15	15.6	
L3	0.471	0.286	0.00128	0.00261	0.00738	0.00907	52.5	47.7	
RAM	n/a	n/a	0.00144	0.00653	0.0329	0.115	208	242	

I observe that without hardware prefetchers, the L1, L2 and L3 hit rates all suffer. Moreover, less accesses hit at L1 and L2, and the amount of accesses which require RAM accesses is about $4.5 \times$, resulting is 11.5% of latency overhead occuring on RAM accesses, compared to 3.29% in the baseline. The LFB hit rate actually increases, however the penalty for an LFB hit is around 3 times worse, indicating that many of these hits are likely waiting for RAM accesses. Therefore, it is clear that removing hardware prefetching impacts performance at all levels of the cache.

This is even clearer when comparing Section 5.3.5, where graphchi has the second lowest percentage of load latency from L1 misses with hardware prefetchers, to Fig. 7.4, where the percentage of L1 miss load latency of graphchi is near the mean.

Outliers: avrora and zxing

Next, I examine avrora, which was only 20% slower without hardware prefetching. The full toplevel microarchitectural analysis results are given in Table 7.5.

Table 7.5: The hardware prefetcher is less effective for avrora, where the LFB penalty remains similar with and without the hardware prefetcher. This table presents the top-level microarchitectural analysis for avrora with and without hardware prefetching.

cache level	hit rate		load pro	oportion	latency o	overhead	average penalty		
	with	without	with	without	with	without	with	without	
L1	0.987	0.978	0.987	0.978	0.917	0.792	8.24	8.95	
LFB	0.236	0.429	0.00298	0.00963	0.0278	0.095	82.8	109	
L2	0.582	0.344	0.0056	0.00441	0.00993	0.00661	15.7	16.5	
L3	0.739	0.571	0.00298	0.00482	0.0162	0.0218	48.4	50.1	
RAM	n/a	n/a	0.00105	0.00361	0.0292	0.0848	246	259	

While cache performance still improves here with hardware prefetching and the latency overheads attributable to LFB, L2 and L3 hits improve, I notice a key difference between avrora and benchmarks where the hardware prefetcher performs better: the LFB penalty remains similar with and without the hardware prefetcher. This differs from graphchi and biojava, where the penalty decreased by about $3\times$. This may explain part of the result.

Another observation is that even without hardware prefetching, avrora has less headroom for improvement, as the load latency overhead attributable to L1 cache misses is lower than all benchmarks except biojava. The characterisation of the benchmark can also help explain the result, as avrora has fine-grained concurrency and locking behaviours which are sensitive to scheduling. This means that the heap and performance can vary greatly between runs, so it is often an outlier in analyses.

Therefore this result is not unreasonable either.

The final benchmark I analyse in depth is zxing, and the toplevel microarchitectural analysis for this is given in Table 7.6.

Examining Fig. 7.4 and comparing to Section 5.3.5, it initially appears that the hardware prefetchers are successful in reducing the overhead of L1 misses for zxing. However, upon closer inspection, I notice that the load latency attributable to RAM

cache level	hit rate		load proportion		latency overhead		average penalty	
	with	without	with	without	with	without	with	without
L1	0.979	0.963	0.979	0.963	0.845	0.649	8.97	9.41
LFB	0.302	0.65	0.00639	0.0239	0.0591	0.259	96	151
L2	0.514	0.286	0.00759	0.00368	0.0128	0.00495	17.5	18.8
L3	0.694	0.545	0.00499	0.00501	0.0291	0.0209	60.5	58.2
RAM	n/a	n/a	0.0022	0.00418	0.0538	0.0669	254	224

Table 7.6: The hardware prefetcher is less effective for zxing, where the proportion of load latency attributable to RAM does not decrease by much with hardware prefetching. This table presents the top-level microarchitectural analysis for zxing with and without hardware prefetching.

does not decrease by much with hardware prefetching. The results in Table 7.6 help to explain this. I notice that the proportion of loads which require RAM access is almost halved with hardware prefetching, however the penalty also increases slightly from 224 cycles to 254 cycles. This means the latency overhead of RAM accesses only decreases by about 20 %.

7.4 Summary

In this chapter, I evaluated the efficacy of hardware prefetching. I found that for all collection algorithms, disabling the hardware prefetcher is detrimental for GC. Moreover the difference is correlated with the extent of copying performed by the collector, with degradations of $1.26 \times$ for MarkSweep compared to $1.63 \times$ for MarkCompact on the two ends of the spectrum. The results are also sensitive to the workload, with all outliers explainable by the size distribution of arrays which objects are stored in. Namely, benchmarks where the prefetcher is more effective, such as biojava, have a large proportion of objects in large arrays, while benchmarks with less prefetcher reliance, like luindex, have majority of objects in small arrays.

Focusing on Immix, I also performed microarchitectural analysis to help explain the results, and found that without the hardware prefetcher, hit rates at L1, L2 and L3 all suffer. Moreover, I observed that the variance in prefetcher efficacy between benchmarks is correlated to the change in the distribution of load latency among accesses at different cache levels. This highlights how the microarchitectural analysis is able to provide deeper insights into reasons for the performance of the prefetcher.

Future Work

In Chapter 5, I introduced a novel approach for microarchitectural cache analysis for GC, and in Chapter 6 and Chapter 7, I applied these insights to understand the current state of prefetching techniques and explore potential optimisations. These new analyses have opened up many new questions and interesting problems for future work. In this chapter, I will discuss some of these possibilities.

8.1 Microarchitectural GC Analysis Extensions

I showed the versatility of the methodology for microarchitectural analysis by using it to gain deeper insight into GC performance, understand the performance of software prefetching in the tracing loop, and rationalise about the efficacy of hardware prefetching for GC. In this section, I explore possible future research avenues which leverage the microarchitectural analysis methodology.

8.1.1 Store Latency

My analysis focused on load latency. This was chosen since stores without preceding loads are uncommon in GC, so I expected the headroom for store optimisations to be lower. However, it is possible to do similar microarchitectural analysis for stores by instead isolating data collection to store operations. This would help gain a more comprehensive microarchitectural picture of GC performance.

8.1.2 Deeper LFB Analysis

One way to improve the granularity of analysis would be distinguishing different types of LFB hits based on what cache levels they wait for. This would allow more precise attribution of load latencies to different cache levels. Doing this would require a different data recording mechanism to PEBS since sampling does not provide temporal guarantees. This means that there is no way to searching for the closest hit on the same cache line as previous accesses may be recorded out of order (buffered writes) or even just have not have been recorded.

8.1.3 Other Collection Algorithms

I focus on the Immix collector throughout the thesis. However, the microarchitectural GC analysis can be directly applied to other tracing collection algorithms. Another possible avenue for research here is examining reference counting collectors. Reference counting relies on counting references to objects for identification, freeing any objects whose references are zero. This is as opposed to the tracing collectors I discuss in this thesis. Both of these ideas for future research could help provide deeper understandings of how different collectors perform, and reveal new insights into the effect of algorithm choice on GC performance.

8.1.4 Concurrent GC

My analysis focused on stop-the-world collection. However, the same analysis techniques could easily be applied to concurrent collector algorithms, since attribution of loads to the collector were determined by examining the caller thread (as opposed to looking at the phase). This could be interesting since concurrent collectors have significantly more cache pollution, so the load latency is likely more significant for overall GC performance. Moreover, many highly performant, production-level collectors, such as G1, Shenandoah, ZGC and LXR, are concurrent [Detlefs et al., 2004; Flood et al., 2016; Liden and Karlsson, 2018; Zhao et al., 2022]. The analysis could be useful in identifying optimisation opportunities there.

8.1.5 Analysis on AMD

My analysis was focused on Intel machines since it relied on PEBS. However, as seen in Section 6.3, the efficacy of software prefetching is very sensitive to the choice of microarchitecture. This indicates that hardware behaviours of GC may differ greatly between different hardware. It would be interesting to verify this by performing microarchitectural analysis on AMD.

AMD machines provide a similar mechanism PEBS called Instruction Based Sampling (IBS). Future work could involve understanding the capabilities of IBS and modifying the methodology in Section 5.2 accordingly.

8.1.6 Understanding Variance in L1 Load Penalty

In Section 6.3.2, I discovered an apparent contradiction where the software prefetching scheme outperformed the headroom I computed in Section 5.3.2. One reason for this was a reduced average L1 load penalty.

Understanding why the L1 loads might get cheaper is interesting as it can help provide better insights into the inner workings of the complex caching system in modern CPUs. It may also reveal opportunities for optimisations which target improving latency even on L1 hits. In the context of my work on microarchitectural analysis, it may assist with the construction of a more robust and accurate method to compute the optimisation headroom. In Section 6.4.3, I discussed possible causes, but did not explore these further. Future work might involve exploring the possibilities raised in Section 6.4.3. Another idea is concretely examining different instructions and their associated latencies to try to narrow down the culprit. Use of newer Intel machines, such as Alder Lake, which provide both instruction and cache latency could also be helpful here.

8.1.7 Reevaluation of Immix line sizes

In Section 5.3.4, I found that heap accesses during GC are surprisingly infrequent, reshaping prevailing understandings of GC cache locality. This indicates a possible need for reevaluation of the line size of Immix, which is a crucial design parameter which was chosen with cache line locality in mind [Blackburn and McKinley, 2008].

8.1.8 Object Arrays

As seen in Section 7.2.1, many benchmarks are dominated by reference array slices. This could be exploitable through optimisations like run-length encoding, which encodes a reference array as its starting address and the array length. This reduces the work queue operations, since slots do not need to be individually enqueued. In particular, this can have implications for hardware prefetching efficacy.

8.1.9 Workload Analysis

The microarchitectural analysis could be made more powerful through if there were more concrete characterisations of the object graphs of benchmarks. This would allow GC researchers to draw stronger correlations between workload behaviour and GC performance. Examples of avenues for research include better understanding characteristics like:

- Mark bit access distribution.
- Max and average object graph depth.
- Number of live objects through time.
- Number of references per object.

8.1.10 Metadata Accesses

In Section 5.3.4, I found that most loads initiated during GC were to metadata. However, most GC optimisation effort focuses on heap accesses, which indicates a possibly overlooked optimisation opportunity in metadata accesses. This requires better understanding the distribution of accesses among different metadata. In MMTk, there are many types of metadata, including global metadata, local side metadata, VMdefined global metadata and VM-defined local side metadata. To better characterise metadata accesses, the bounds for each of these would need to be determined and then incorporated into the analysis. A similar approach to Section 5.3.3 could then be applied.

8.2 Lightweight Benchmark-Level Dynamic Prefetching

I found in Section 6.5 that dynamic prefetching is overall unprofitable for tracing GC. However, for some workloads with large amounts of headroom, dynamic prefetching for edge prefetching at a benchmark-level could be plausible. This would need to be lightweight and would likely involve choosing between a few fixed-distance options based on a heuristic. To create such a prefetcher, there remains a need for research into two components:

- 1. A mechanism for selecting different prefetch distances at VM boot time
- 2. A heuristic to determine when to guide adjustments

Both are possible avenues for research.

8.2.1 Dynamic Adjustment Mechanism

To prefetch adaptively, there must be runtime capabilities to modify the distance. Moreover, it must be little to no overhead.

In Rust, this can be done with specialisation, which essentially creates different versions of functions for each possible prefetch value. This allows the compiler to make advantage of inlining. In theory, this would lead to essentially no overhead from switching the prefetch distance. I prototyped this in heapdumps but did not implement it in MMTk. Future work involves porting the prototype to MMTk and running quick measurements to understand if it is indeed inexpensive.

8.2.2 Heuristics

Another research avenue is finding lightweight heuristics for simple benchmark-level prefetching. Some candidates for research are:

- Understanding reasons for ineffective prefetches: Ineffective prefetch counters could be used to develop techniques for determining whether a prefetch is early, late or incorrect. This could help drive a dynamic prefetcher.
- Memory bandwidth: If the bandwidth utilisation is high, prefetchers may be less effective or even harmful. This idea could even be combined with turning off hardware prefetching in instances where the hardware prefetcher may be utilising too much bandwidth, and can be replaced by software prefetching, where the collector has more control and precision. An example of this type of research is Jain et al. [2024]; its principles may be applicable to GC.

- Adaptation of existing dynamic prefetchers: There exists multiple adaptive prefetchers, such as APT-GET [Jamilan et al., 2022], which leverages LBR to predict future prefetches, and a performance-driven prefetching scheme by Beyler and Clauss [2007], which monitors all loads to adjust the prefetching scheme. These may be adaptable to lightweight versions for the GC tracing loop.
- Microarchitectural analysis on the fly: Since the L1 miss overheads are wellcorrelated to prefetch efficacy Fig. 6.4, a lightweight version of my microarchitectural analysis could be used to guide dynamic prefetching.
- Microbenchmarking: Running the microarchitectural analysis on microbenchmarks with and without prefetching could be a useful starting point to help find other heuristic candidates. This is because the DaCapo benchmarks are rich and complicated real-world workloads where it is harder to draw direct correlations.

8.3 Further Research on Hardware Prefetching

There are also many possible future avenues for understanding the hardware prefetcher more deeply.

8.3.1 Performance Across GC Phases

My analysis focused on the overall GC performance of hardware prefetching. One possibility is that efficacy varies between GC phases. This is because some GC phases, such as tracing, may have larger work packets which provide the hardware prefetcher with more structure to exploit, while others may traverse singly-linked list structures which require extra indirections that the L1 and L2 prefetchers cannot handle.

The efficacy could also vary depending on the collector algorithm. I hypothesise the hardware prefetcher may have greater efficacy on the tracing loop for moving collectors, since they can exploit better access locality. Conversely, other GC components, such as root scanning, may be less likely to be affected.

8.3.2 Work Packet Size

I found initial correlations between the distribution and hardware prefetch efficacy. This could be explored in greater depth by artificially adjusting the work packet distribution. This may also help inform work packet system design choices which better exploit hardware prefetching.

8.3.3 Focusing on Specific Hardware Prefetchers

My analysis focused on the full set of hardware prefetchers. It would also be interesting to understand how each specific prefetcher behaves on GC.

8.3.4 Data Dependent Prefetching for GC

Newer Intel processors have a data dependent (DD) prefetcher which is able to examine data values in memory. This allows contents of pointers to be prefetched, which may render software prefetching somewhat ineffective. Future work involves analysing the performance of the DD prefetcher for GC and understanding the interactions between software prefetching and the DD prefetcher.

8.3.5 Understanding Hardware Prefetcher Efficacy on Java Workloads

In Section 7.2, I found that hardware prefetchers have limited efficacy on mutator performance for some workloads. As this thesis focuses on GC performance, I did not explore this further. However, since the DaCapo benchmarks are modern object-oriented server-based Java workloads, many have more complex object graphs and exhibit different behaviours to the standard C workloads which may be more exploitable by hardware prefetchers [Blackburn et al., 2025]. Interesting avenues for future research include deeply understanding the performance of hardware prefetchers which can better exploit modern Java workloads.

Conclusion

From the datacenter to the phone, Garbage Collection (GC) underpins most modern programming languages. Understanding GC performance from both software and hardware perspectives is critical for improved user experiences, particularly in a landscape where hardware and software is rapidly changing. However, existing methodologies for microarchitectural GC analysis lack fine-grained attribution.

In this thesis, I introduced a novel methodology for microarchitectural analysis of GC performance which provides fine-grained attribution of load latencies to functions, cache levels, and memory segments (heap vs non-heap accesses). Leveraging this methodology, I performed in-depth microarchitectural analysis of the cache behaviours and performance characteristics of GC, and analysed the efficacy of Coffee Lake hardware prefetchers and a software prefetching scheme for tracing GC.

I discovered that L1 misses are rare but expensive, accounting for only 2.0% of loads, but 14.9% of load latency cycles. However, the addition of software prefetching reduces both these gaps by half, leading to a 9% speedup of GC on Coffee Lake and 18% speedup on Zen 4. Moreover, I found software prefetching efficacy on individual benchmarks is highly correlated ($R^2 = 0.708$) with the optimisation headroom of the tracing loop, highlighting the usefulness of microarchitectural analysis.

I found that disabling the hardware prefetcher is detrimental for GC, and the level of damage correlates with how much copying is performed by the collector. I also observed that hardware prefetching efficacy is sensitive to the workload, with all outliers explainable by the size distribution of arrays which objects are stored in.

I discovered that heap accesses are surprisingly only responsible for 3.2% of loads and 10.9% of load latency, indicating that on modern hardware, software and workloads, traditional understandings of GC hardware behaviour may no longer be true. Moreover, metadata accesses are surprisingly common, uncovering a potential missed opportunity for metadata-focused optimisation.

I also found that software prefetching unexpectedly improves the load latency of L1 hits. This emphasises how modern hardware and cache systems are complex and understudied, and highlights an important area for future research.

In summary, my thesis proposes a novel approach to GC microarchitectural analysis, and demonstrates its rich observability and ability to localise and expose microarchitectural performance problems. In the process, I encounter multiple apparent contradictions to existing GC folklore and discover multiple new insights on GC behaviour. It opens up many new questions and interesting problems for future work, and reaffirms the continued need for comprehensive microarchitectural analysis of GC performance.

Bibliography

- Function _mm_prefetch. https://doc.rust-lang.org/beta/core/arch/x86_64/fn._mm_prefetch. html. (cited on page 15)
- ALPERN, B.; ATTANASIO, C. R.; BARTON, J. J.; BURKE, M. G.; CHENG, P.; CHOI, J.-D.; COCCHI, A.; FINK, S. J.; GROVE, D.; HIND, M.; HUMMEL, S. F.; LIEBER, D.; LITVINOV, V.; MERGEN, M. F.; NGO, T.; RUSSELL, J. R.; SARKAR, V.; SERRANO, M. J.; SHEPHERD, J. C.; SMITH, S. E.; SREEDHAR, V. C.; SRINIVASAN, H.; AND WHALEY, J., 2000. The Jalapeño virtual machine. *IBM Syst. J.*, 39, 1 (Jan. 2000), 211–238. doi:10.1147/sj.391.0211. https://doi.org/10.1147/sj.391.0211. (cited on pages 17 and 19)
- ATKINSON, A., 2023. Software Cache Prefetching for Tracing Garbage Collectors. Available at https://www.steveblackburn.org/pubs/theses/atkinson-2023.pdf. (cited on pages 2, 8, 15, 19, 44, 51, 52, 54, and 55)
- BAKHVALOV, D., 2018. Advanced profiling topics. PEBS and LBR. https://easyperf.net/ blog/2018/06/08/Advanced-profiling-topics-PEBS-and-LBR. (cited on pages 13 and 48)
- BEAMER, S.; ASANOVIC, K.; AND PATTERSON, D., 2015. Locality exists in graph processing: Workload characterization on an Ivy Bridge server. In 2015 IEEE International Symposium on Workload Characterization, 56–65. doi:10.1109/IISWC.2015.12. (cited on page 20)
- BERTSCHI, A., 2022. Battling the prefetcher: Exploring coffee lake (part 1). https://abertschi.ch/blog/2022/prefetching/. (cited on page 15)
- BEYLER, J. C. AND CLAUSS, P., 2007. Performance driven data cache prefetching in a dynamic software optimization system. In *Proceedings of the 21st Annual International Conference on Supercomputing*, ICS '07 (Seattle, Washington, 2007), 202–209. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1274971.1275000. https://doi.org/10.1145/1274971.1275000. (cited on page 83)
- BLACKBURN, S. M.; CAI, Z.; CHEN, R.; YANG, X.; ZHANG, J.; AND ZIGMAN, J., 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, Netherlands, 30 March 2025 - 3 April 2025.* ACM. doi:10.1145/3669940.3707217. https://doi.org/10.1145/3669940.3707217. (cited on pages xvii, 1, 18, 19, 20, 23, 24, and 84)
- BLACKBURN, S. M.; CHENG, P.; AND MCKINLEY, K. S., 2004a. Myths and realities: the performance impact of garbage collection. In *Proceedings of the Joint*

International Conference on Measurement and Modeling of Computer Systems, SIG-METRICS '04/Performance '04 (New York, NY, USA, 2004), 25–36. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1005686.1005693. https://doi.org/10.1145/1005686.1005693. (cited on pages 2 and 17)

- BLACKBURN, S. M.; CHENG, P.; AND MCKINLEY, K. S., 2004b. Oil and water? High performance garbage collection in Java with MMTk. In *Proceedings*. 26th International Conference on Software Engineering, 137–146. doi:10.1109/ICSE.2004.1317436. (cited on pages 7, 17, and 19)
- 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* 29th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '08 (Tucson, AZ, USA, 2008), 22–32. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1375581.1375586. https://doi.org/10.1145/1375581. 1375586. (cited on pages 5, 9, 18, 19, 38, and 81)
- BOEHM, H.-J., 2000. Reducing garbage collector cache misses. In *Proceedings of the 2nd International Symposium on Memory Management*, ISMM '00 (Minneapolis, Minnesota, USA, 2000), 59–64. Association for Computing Machinery, New York, NY, USA. doi:10.1145/362422.362438. https://doi.org/10.1145/362422.362438. (cited on page 19)
- CAI, Z., 2024. running-ng. https://github.com/anupli/running-ng. (cited on page 24)
- CAI, Z.; BLACKBURN, S. M.; BOND, M. D.; AND MAAS, M., 2022. Distilling the real cost of production garbage collectors. In 2022 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 46–57. doi:10.1109/ISPASS55109.2022.00005. (cited on pages 1, 2, and 17)
- 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 48 and 73)
- CARPEN-AMARIE, M.; VAVOULIOTIS, G.; TOVLETOGLOU, K.; GROT, B.; AND MUELLER, R., 2023. Concurrent GCs and Modern Java Workloads: A Cache Perspective. In Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management, ISMM 2023 (Orlando, FL, USA, 2023), 71–84. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3591195.3595269. https: //doi.org/10.1145/3591195.3595269. (cited on pages 2, 18, 23, and 28)
- CHENEY, C. J., 1970. A nonrecursive list compacting algorithm. *Commun. ACM*, 13, 11 (Nov. 1970), 677–678. doi:10.1145/362790.362798. https://doi.org/10.1145/362790. 362798. (cited on page 10)
- CHER, C.-Y.; HOSKING, A. L.; AND VIJAYKUMAR, T. N., 2004. Software prefetching for mark-sweep garbage collection: hardware analysis and software redesign. In
Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XI (Boston, MA, USA, 2004), 199–210. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1024393. 1024417. https://doi.org/10.1145/1024393.1024417. (cited on page 19)

- DETLEFS, D.; FLOOD, C.; HELLER, S.; AND PRINTEZIS, T., 2004. Garbage-first garbage collection. In *Proceedings of the 4th International Symposium on Memory Management*, ISMM '04 (Vancouver, BC, Canada, 2004), 37–48. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1029873.1029879. https: //doi.org/10.1145/1029873.1029879. (cited on page 80)
- FLEMING, P. J. AND WALLACE, J. J., 1986. How not to lie with statistics: the correct way to summarize benchmark results. *Commun. ACM*, 29, 3 (Mar. 1986), 218–221. doi:10.1145/5666.5673. https://doi.org/10.1145/5666.5673. (cited on page 25)
- FLOOD, C. H.; KENNKE, R.; DINN, A.; HALEY, A.; AND WESTRELIN, R., 2016. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. PPPJ '16 (Lugano, Switzerland, 2016). Association for Computing Machinery, New York, NY, USA. doi:10.1145/2972206.2972210. https://doi.org/10.1145/2972206.2972210. (cited on pages 6 and 80)
- GARNER, R.; BLACKBURN, S. M.; AND FRAMPTON, D., 2007. Effective prefetch for marksweep garbage collection. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07 (Montreal, Quebec, Canada, 2007), 43–54. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1296907.1296915. https: //doi.org/10.1145/1296907.1296915. (cited on pages 2, 19, and 52)
- HELM, C. AND TAURA, K., 2019. PerfMemPlus: A tool for automatic discovery of memory performance problems. In *High Performance Computing*, 209–226. Springer International Publishing, Cham. (cited on pages 18 and 33)
- HELM, C. AND TAURA, K., 2020. On the correct measurement of application memory bandwidth and memory access latency. In *Proceedings of the International Conference* on High Performance Computing in Asia-Pacific Region, HPCAsia '20 (Fukuoka, Japan, 2020), 131–141. Association for Computing Machinery, New York, NY, USA. doi: 10.1145/3368474.3368476. https://doi.org/10.1145/3368474.3368476. (cited on page 34)
- HUANG, C.; BLACKBURN, S.; 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, 2023), 85–99. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3617651.3622986. https://doi.org/10.1145/3617651.3622986. (cited on pages 2, 7, 17, 23, 31, 41, and 44)
- INTEL CORPORATION. PerfMon Events. https://perfmon-events.intel.com/. (cited on page 13)

- CORPORATION, 2016. Intel® IA-32 Architectures Soft-INTEL 64 and Developer's Manual Volume 3B: System Programming Part ware Guide, 2. https://www.intel.com/content/dam/www/public/us/en/documents/manuals/ 64-ia-32-architectures-software-developer-vol-3b-part-2-manual.pdf. (cited on pages 2, 13, 14, 18, 32, and 61)
- INTEL CORPORATION, 2019. Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 4: Model-Specific Registers. https://www.intel.com/content/dam/develop/external/us/en/documents/335592-sdm-vol-4.pdf. (cited on pages 48 and 69)
- INTEL CORPORATION, 2023. Intel® 64 and IA-32 Architectures Optimization Reference Manual Volume 1. https://www.intel.com/content/www/us/en/content-details/671488/ intel-64-and-ia-32-architectures-optimization-reference-manual-volume-1.html. (cited on pages 14, 15, and 28)
- JAIN, A.; LIN, H.; VILLAVIEJA, C.; KASIKCI, B.; KENNELLY, C.; HASHEMI, M.; AND RANGANATHAN, P., 2024. Limoncello: Prefetchers for scale. In Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS '24 (La Jolla, CA, USA, 2024), 577–590. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3620666. 3651373. https://doi.org/10.1145/3620666.3651373. (cited on page 82)
- JAMILAN, S.; KHAN, T. A.; AYERS, G.; KASIKCI, B.; AND LITZ, H., 2022. APT-GET: profile-guided timely software prefetching. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22 (Rennes, France, 2022), 747–764. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3492321.3519583. https://doi.org/10.1145/3492321.3519583. (cited on page 83)
- JONES, R.; HOSKING, A.; AND MOSS, E., 2011. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edn. ISBN 1420082795. (cited on pages 6, 9, and 10)
- KAUSHIK, A. M.; PEKHIMENKO, G.; AND PATEL, H., 2021. Gretch: A hardware prefetcher for graph analytics. *ACM Trans. Archit. Code Optim.*, 18, 2 (Feb. 2021). doi:10.1145/ 3439803. https://doi.org/10.1145/3439803. (cited on page 20)
- LEE, J.; KIM, H.; AND VUDUC, R., 2012. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9, 1 (Mar. 2012). doi:10.1145/2133382.2133384. https://doi.org/10.1145/2133382.2133384. (cited on pages 15 and 19)
- LIDEN, P. AND KARLSSON, S., 2018. Jep 333: ZGC: A scalable low-latency garbage collector (experimental). https://openjdk.org/jeps/333. (cited on pages 6 and 80)
- LIN, Y.; BLACKBURN, S. M.; HOSKING, A. L.; AND NORRISH, M., 2016. Rust as a language for high performance GC implementation. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016 (Santa Barbara, CA, USA, 2016), 89–98. Association for Computing Machinery, New York, NY, USA.

doi:10.1145/2926697.2926707. https://doi.org/10.1145/2926697.2926707. (cited on page 7)

- McGACHEY, P. AND HOSKING, A. L., 2006. Reducing generational copy reserve overhead with fallback compaction. In *Proceedings of the 5th International Symposium on Memory Management*, ISMM '06 (Ottawa, Ontario, Canada, 2006), 17–28. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1133956.1133960. https://doi.org/10.1145/1133956.1133960. (cited on page 9)
- MOHROR, K. AND ROUNTREE, B., 2012. Cache performance analysis and optimization. Technical report, Lawrence Livermore National Laboratory. (cited on pages 18 and 33)
- PAPADAKIS, O.; ANDRONIKAKIS, A.; FOUTRIS, N.; PAPADIMITRIOU, M.; STRATIKOPOULOS, A.; ZAKKAK, F. S.; XEKALAKIS, P.; AND KOTSELIDIS, C., 2023. A Multifaceted Memory Analysis of Java Benchmarks. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*, MPLR 2023 (Cascais, Portugal, 2023), 70–84. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3617651.3622978. https://doi.org/10.1145/3617651.3622978. (cited on pages 2, 18, 23, and 28)
- PAZ, H. AND PETRANK, E., 2007. Using prefetching to improve reference-counting garbage collectors. In *Compiler Construction*, 48–63. Springer Berlin Heidelberg, Berlin, Heidelberg. (cited on page 19)
- PESTEREV, A.; ZELDOVICH, N.; AND MORRIS, R. T., 2010. Locating cache performance bottlenecks using data profiling. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10 (Paris, France, 2010), 335–348. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1755913.1755947. https://doi.org/10.1145/1755913.1755947. (cited on page 18)
- PROKOPEC, A.; ROSÀ, A.; LEOPOLDSEDER, D.; DUBOSCQ, G.; TŮMA, P.; STUDENER, M.;
 BULEJ, L.; ZHENG, Y.; VILLAZÓN, A.; SIMON, D.; WÜRTHINGER, T.; AND BINDER, W.,
 2019. Renaissance: benchmarking suite for parallel applications on the JVM. In
 Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design
 and Implementation, PLDI 2019 (Phoenix, AZ, USA, 2019), 31–47. Association for
 Computing Machinery, New York, NY, USA. doi:10.1145/3314221.3314637. https:
 //doi.org/10.1145/3314221.3314637. (cited on page 20)
- REINHOLD, M. B., 1994. Cache performance of garbage-collected programs. *SIGPLAN Not.*, 29, 6 (Jun. 1994), 206–217. doi:10.1145/773473.178261. https://doi.org/10.1145/ 773473.178261. (cited on page 17)
- THE WHITE HOUSE, 2024. Press Release: Future Software Should Be Memory Safe. https://bidenwhitehouse.archives.gov/oncd/briefing-room/2024/02/26/ press-release-technical-report. Accessed May 23, 2025. (cited on page 1)

- UNGAR, D., 1984. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. SDE 1, 157–167. Association for Computing Machinery, New York, NY, USA. doi:10.1145/800020.808261. https://doi.org/10.1145/800020.808261. (cited on page 9)
- VAN SCHAIK, S.; MILBURN, A.; ÖSTERLUND, S.; FRIGO, P.; MAISURADZE, G.; RAZAVI, K.; BOS, H.; AND GIUFFRIDA, C., 2019. Ridl: Rogue in-flight data load. In 2019 IEEE Symposium on Security and Privacy (SP), 88–105. doi:10.1109/SP.2019.00087. (cited on page 12)
- XU, B.; MOSS, E.; AND BLACKBURN, S. M., 2022. Towards a model checking framework for a new collector framework. MPLR '22 (Brussels, Belgium, 2022), 128–139. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3546918.3546923. https://doi.org/10.1145/3546918.3546923. (cited on page 7)
- YANG, X.; BLACKBURN, S. M.; FRAMPTON, D.; SARTOR, J. B.; AND MCKINLEY, K. S., 2011. Why nothing matters: the impact of zeroing. In *Proceedings of the 2011* ACM International Conference on Object Oriented Programming Systems Languages and Applications, OOPSLA '11 (Portland, Oregon, USA, 2011), 307–324. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2048066.2048092. https: //doi.org/10.1145/2048066.2048092. (cited on page 20)
- ZHAO, W.; BLACKBURN, S. M.; AND MCKINLEY, K. S., 2022. Low-latency, highthroughput garbage collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022 (San Diego, CA, USA, 2022), 76–91. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3519939.3523440. https://doi.org/10.1145/3519939.3523440. (cited on page 80)
- ZORN, B., 1991. The effect of garbage collection on cache performance. Technical report, University of Colorado Boulder. (cited on page 17)