Software Cache Prefetching for Tracing Garbage Collectors

Angus Atkinson

A thesis submitted for the degree of
Bachelor of Advanced Computing (Honours)
The Australian National University

October 2023

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

Angus Atkinson
31 October 2023

To all the educators who have inspired me to learn.

Acknowledgements

Firstly, I would like to thank Professor Steve Blackburn, my supervisor and mentor. Thank you for taking a chance on an inexperienced first year back in 2020, for teaching me about garbage collection and for spawning my interest in computer systems, which has guided my degree and future career path. Thank you for accepting me back into the research group for my Honours year, for giving me your time, guidance, and ideas, and for your never-ending patience. Thank you for teaching me the principles of being a good researcher, a good computer scientist and good communicator - the lessons I have learned will serve me well throughout the rest of my career.

Thank you to the rest of the MMTk research group - Zixian Cai, Claire Huang, Yi Lin, Tianle Qui, Kunal Sareen, Eduardo Souza, Kunshan Wang, and Wenyu Zhao - for showing me the ropes, helping me understand the MMTk codebase, and for letting me reserve all of our most important machines for weeks on end for my final performance tests.

I am thankful for the support of those around me over the years. Thank you to my family - Sandra, Bruce and Yolande Atkinson - for listening to all the late-night rants and my fruitless attempts to explain my research to you. Thank you to Peter Oslington for helping me understand Intel's poorly documented performance counters, and the endless thought provoking discussions on each of our research projects. Thank you to my other close friends - Taskil Dastoor, Kieran Hammond, Gemma O'Kane, and Katerina Simeonoff - for all of the dinner parties, D&D sessions and MarioKart races. They have really helped me stay sane throughout my honours year.

Finally, I would like to thank the Tuckwell Scholarship program for their generous financial support and mentorship throughout my entire degree.

Abstract

Garbage collection (GC) is a core component of almost all modern memory safe programming languages. The tracing loop is one of the largest costs in tracing garbage collectors, constituting a significant portion of total collection time. Furthermore, many reference counting collectors rely on tracing to break cycles. Therefore improving tracing performance is paramount to reducing the cost of garbage collection.

Despite the high cost of the tracing loop, there has been little research on how the structure of the loop itself affects tracing performance. Whilst previous work has identified performance differences between Node- and Edge-ordered enqueueing, other design choices have not been explored in great detail, including the queue item type and dual-queueing variants.

One of the primary contributors to the cost of tracing is the poor cache locality of traversing the heap. Software cache prefetching has been identified as one technique for improving the speed of tracing garbage collectors by reducing cache misses. However, in order for it to be an effective optimisation, software cache prefetching needs to be constantly re-evaluated, as rapidly-evolving modern hardware presents a moving target. With the last research on the topic being over a decade old, this technique is long overdue for a revisit. Furthermore, exploring new tracing loop structures could reveal previously unexplored software prefetching opportunities, which could lead to further performance gains.

This thesis explores how the performance of production tracing garbage collectors can be improved by choosing the tracing loop structure that enables the most effective software prefetching. It provides three key contributions: (i) a novel framework for evaluating variations on the tracing loop of garbage collection algorithms, which I call *auxiliary tracing*; (ii) a taxonomy for classifying tracing loop designs, alongside performance evaluations of seven possible structural variations; (iii) a comprehensive performance evaluation of software cache prefetching techniques when applied to the tracing loop of garbage collectors. I conduct my studies in the context of real-world benchmarks on modern hardware (including recent CPU microarchitectures).

My results show that the *Edge-ObjRef* loop (which performs edge-ordered enqueueing of object references) is the highest-performing tracing loop design, yielding an 8.3% performance improvement on AMD Zen 4, and a 15.6% improvement on Intel Coffee Lake, with respect to the default MMTk tracing loop implementation. Furthermore, the *Edge-ObjRef* algorithm demonstrates an 8.1 to 11.1% performance advantage over the canonical tracing loop design (*Node-ObjRef*), despite requiring 2.6 times as many queueing operations on average.

Furthermore, I demonstrate that software cache prefetching consistently increases tracing speed across a wide variety of benchmarks. On Zen 4, the addition of prefetching boosts the performance of the Edge-ObjRef and Node-ObjRef tracing loops by

10.7% and 15.1% respectively, reducing the performance gap but still leaving Edge-ObjRef as a clear winner. On Coffee Lake, the Edge-ObjRef design does not benefit as much from prefetching, allowing the optimised Node-ObjRef algorithm to perform 2.2% better on average.

This work fills a major gap in our understanding of the performance characteristics of tracing loops in garbage collectors. It enables language designers and implementers to make informed choices about the structure of the core tracing loop, regardless of the constraints of their collector (whether it be non-moving or copying). My in-depth analysis of numerous prefetching strategies shows high potential performance improvements, but also highlights the microarchitectural sensitivity of prefetching, demonstrating a need for hardware-specific tuning. Furthermore, my research demonstrates the power and flexibility of the auxiliary tracing framework as a tool for evaluating changes to the core tracing loop.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In a world where computer security is paramount, software developers are increasingly choosing to use memory-safe programming languages for their projects. One of the ways that managed languages provide memory safety is through *automatic memory management* of dynamically allocated memory. Apart from the notable example of Rust, almost all modern programming languages use *garbage collection* for memory management.

One of the most common paradigms of garbage collection is *tracing* garbage collection, which involves performing a transitive closure of the heap to identify live objects. Tracing is a core component of many production garbage collectors, including collectors for V8 [Marshall, 2019], Ruby [Ruby Contributors, 2023], and OpenJDK (G1 [Detlefs et al., 2004], Shenandoah [Flood et al., 2016], ZGC [Liden and Karlsson, 2018a,b] and C4 [Tene et al., 2011]). Even modern high-performance reference counting collectors such as RCImmix [Shahriyar et al., 2013] and LXR [Zhao et al., 2022] rely on tracing to collect cycles in the object graph, in order to avoid memory leaks.

The tracing loop is responsible for traversing the object graph, marking and scanning all objects. This makes it one of the most performance-critical aspects of a tracing collector. Previous research has estimated that the tracing loop constitutes as much as 57% [Boehm, 2000], 70% [Cher et al., 2004], or 96% [Garner, 2011] of total collection time - primarily due to the low cache locality of tracing the heap. Despite the high cost of the tracing loop, there has been relatively little research of alternative designs and structural variations which could improve performance. Although some variations on the canonical tracing loop design have been explored, including edge-ordered enqueueing [Garner et al., 2007], other aspects have not been examined in as much detail. This includes enqueuing slots instead of (or in addition to) object references, and dual-queue designs. This thesis explores several new designs and evaluates their effectiveness.

Since the cache locality of the tracing loop is so poor, one other area of previous research has been to introduce software cache prefetching instructions into the loop, to preemptively bring data into cache before it is needed [Boehm, 2000; Cher et al., 2004; Garner et al., 2007]. This technique has been shown to be relatively successful, reducing garbage collection time by up to 20-30%. However, despite the ever-changing nature of modern hardware, these results have not been re-evaluated in a modern

context. Furthermore, I believe that exploration of new tracing loop designs could uncover additional opportunities for software cache prefetching that were not possible before, potentially improving tracing performance. Thus, the potential impact of software prefetching in contemporary setting is an open question.

In this thesis, I primarily examine the application of prefetching to the core tracing loop. Other prefetching opportunities in garbage collectors (e.g. for allocation or reference counting) are possible interesting avenues for future research, but are beyond the scope of this thesis.

1.1 Thesis Statement

Optimisation of tracing loop performance is critical to reducing the cost of garbage collection. I believe that the performance of production tracing garbage collectors can be improved by choosing the tracing loop structure that enables the most effective software prefetching.

1.2 Contributions

In this thesis, I provide four major contributions:

1. I introduce *auxiliary tracing*, a novel evaluation framework which can be used to:

   (a) measure and compare the performance of structural variations on the core tracing loop of garbage collectors;

   (b) assess the performance implications of the heap layouts generated by a wide spectrum of garbage collection algorithms;

2. I propose a new taxonomy for classifying tracing loop designs, based on two primary axes: the queueing strategy, and the queue item type;

3. I evaluate seven different tracing loop designs, comparing performance characteristics and identifying potential compatibility issues;

4. I perform a comprehensive performance evaluation of a wide variety of software cache prefetching techniques which can be applied to the tracing garbage collectors, in the context of five of the previously identified tracing loop designs.

The results presented by this thesis provide a much-needed update to an area of research that has not been revisited in over a decade. All of my studies are performed in a modern environment, with recent CPU microarchitectures and contemporary benchmark suites which are representative of real-world applications.

1.3 Thesis Structure

Chapter 2 provides background information on several of the key topics that form the basis of this thesis, including garbage collection, the Memory Management Toolkit (MMTk), and hardware/software cache prefetching.

Chapter 3 explores existing research on the applicability of software cache prefetching to tracing garbage collectors. Furthermore, I evaluate the software cache prefetching techniques implemented into OpenJDK's stock collectors, to determine their effectiveness.

Chapter 4 introduces auxiliary tracing, a novel methodology for evaluating variations on the GC tracing loop. I discuss the design and implementation decisions made, validate auxiliary tracing correctness and performance against existing production collectors, and explore two case studies to demonstrate the power and flexibility of this new tool.

Chapter 5 details the possible design choices that can be made when implementing the tracing loop of a garbage collector, and proposes a new taxonomy which classifies new and existing designs. I evaluate the performance of seven different tracing loop algorithms and compare key statistics.

Chapter 6 discusses several opportunities for software cache prefetching in the context of GC tracing loops. A wide range of prefetching configurations are evaluated, followed by a critical evaluation of the the results.

Chapter 7 summarises the key findings of this thesis and expands on several potential avenues for further research.

Chapter 2

Background

This chapter introduces the background information necessary to understand several of the core topics in this thesis. Section 2.1 begins with an introduction to *garbage collection*, a common technique used for automatic memory management. Section 2.2 discusses MMTk, the broader framework used for implementation and evaluation in this thesis. Finally, Section 2.3 gives an overview of hardware and software cache prefetching techniques.

2.1 Garbage Collection Terminology

Garbage collection (GC) is one of the most common forms of automatic memory management used in modern programming languages. The purpose of a garbage collector is to run in tandem with an application (known as the *mutator*) and manage the *heap*, which is the region of memory that stores dynamically allocated objects. Objects in the heap are said to be *live* if they are transitively reachable through the mutator *roots* (that is, the stack, local and global variables, and other objects directly accessible by the application), or *dead* if they are not. Garbage collectors are responsible for the *allocation* of new objects, and the *identification* and *reclamation* of dead objects.

Allocation can be implemented via either *bump-pointer* or *free-list* allocation. Bump-pointer allocation involves allocating objects contiguously in a region of free memory at the current *cursor*, which is subsequently advanced by the size of the object to point to the next free space. If the cursor reaches the the end of the allocation buffer (known as the *limit* or *watermark*), or if there is not enough space left in the buffer, a new chunk of memory is allocated by the memory management system for future allocations. Alternatively, a free-list allocator can be used, which stores chunks of free memory in one or more linked lists (known as *free lists*). During allocation, the mutator traverses the free list to find a chunk that is large enough to store the desired object, pops it from the free list, and overwrites the free list metadata with the object contents.

Dead objects can be identified indirectly via *tracing garbage collection*, or directly via *reference counting*. Tracing garbage collection, which was first invented by McCarthy [1960] for the Lisp programming language, involves performing a transitive closure of the heap, marking all live objects. This is discussed in detail in Section 2.1.1. Reference

counting, which was later introduced by Collins [1960], involves keeping track of the number of references pointing to each object, through a *reference count*. Whenever a reference is created to an object, its reference count is incremented, and when the reference is overwritten or removed, the reference count is decremented. When an object's reference count reaches zero, it is deallocated, which in turn involves decrementing the reference count of child objects and potentially triggering their deallocation. Note that if a cyclic reference is present, the reference count will never reach zero and thus the object will never be deallocated. To avoid leaking memory, it is necessary to occasionally run a tracing garbage collector, or force the programmer to manually break cycles using *weak references*.

Once an object is identified as being dead, it needs to be reclaimed so that its memory can be reused for other objects. *Non-moving* garbage collectors perform a *sweep* over the heap, identifying objects which were not marked during an earlier tracing phase (i.e. objects which are dead), and joining them onto one of the allocator free lists. *Moving* garbage collectors (also known as *copying* collectors) may choose to copy the object to a new location, either within the same heap space (*compaction*) or to a new space (*evacuation*). Although object copying can be expensive (due to the cost of the moving the object and updating references to it), it can also improve mutator performance by reducing fragmentation and improving the locality of the heap.

These allocation, identification and reclamation strategies can be combined to form a variety of garbage collection algorithms. Common algorithms include *Mark-Sweep* (which uses free-list allocation, tracing for identification, and sweeping for reclamation), *Mark-Compact* (free-list, tracing, compaction), *SemiSpace* (bump-pointer, tracing, evacuation), and *Reference Counting* (free-list, reference counting, sweeping). Most production garbage collectors combine several *spaces* managed by one or more algorithms. For instance, *generational garbage collection* involves allocating new objects into a small *nursery space*, which is frequently collected. Any objects which survive a collection are evacuated to the *mature space*, which is only collected when a full-heap GC occurs. According to the *weak generational hypothesis* [Lieberman and Hewitt, 1983; Ungar, 1984], most objects will die quickly (often before their first collection), and thus frequent collections of the nursery will result in shorter pause times than collecting the whole heap.

For further discussion of garbage collection terminology, see Jones et al. [2023].

2.1.1 Tracing Garbage Collection

At the heart of all tracing garbage collectors is the *tracing loop*, which is responsible for identifying all live objects by performing a transitive closure of the heap (or some part of the heap), starting at the roots and marking/scanning all objects.

Before discussing the mechanical details of the tracing loop, it is first necessary to understand terminology relating to object structure. An object consists of several *fields* (also known as *slots*), each of which may contain an *object reference* (a pointer to another object on the heap), or a *scalar* (an immediate value stored within the field, like an integer or boolean). This is shown in Figure 2.1. In this thesis, we typically use

the term 'slot' to refer to the memory location that contains a reference to the current object, and 'fields' to denote the words inside the current object which may contain pointers to other objects. However, these two terms are often used interchangeably, because the slot which contains a reference to this object is always a field of another object.

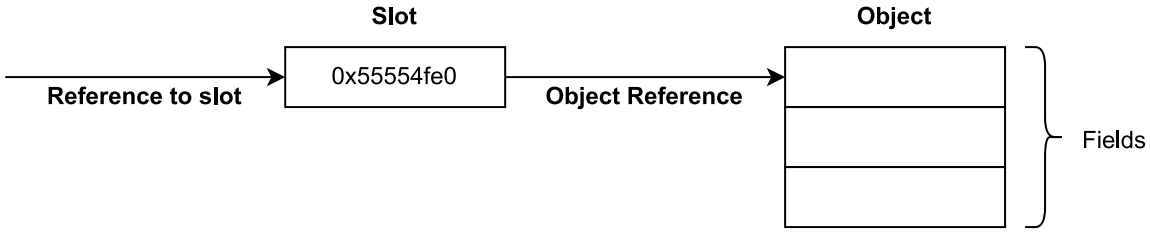


Figure 2.1: Key terminology in tracing garbage collection: the slot vs the object reference

To trace an object, it is first necessary to load the slot which contains a reference to the object. Then, the referent object should be marked; either by setting a mark bit in the header of the object, or by modifying some side metadata (e.g. a mark bitmap) at a location determined by the object reference. If the object was already marked, then we can stop; this ensures that the tracing loop terminates. Otherwise, it is necessary to scan the object. This is achieved by identifying the fields of the object which contain pointers (as opposed to scalars), and tracing each of the child objects referenced from those fields.

In order to keep track of the objects waiting to be scanned, tracing loops usually employ some form of queue or stack. In single-threaded collectors, the use of a stack data structure will force a depth-first traversal of the heap, whereas a first-in-first-out queue will cause the traversal order to be breadth-first. A depth-first traversal of the heap is known to be preferable [Garner, 2011], as it exhibits better cache locality due to the fact that parent and child objects are often allocated near each other. However, many modern collectors parallelise the tracing loop, which makes the traversal order less clear. Ossia et al. [2002] note that the *work packet* mechanism employed by many parallel collectors (including MMTk) searches the heap breadth-first locally, but depth-first globally.

When tracing an object, it is necessary to determine which fields contain references to child objects[1]. There are a wide variety of techniques which can be used, many of which are discussed by Garner et al. [2011]. For instance, in statically typed object oriented programming languages like Java, a common pattern is to include a pointer to the class information in the object header, which can be used to determine the pointer fields. As another example, some dynamically typed languages like Ruby tag the lower-order bits of each field with type information which can be used to determine whether the field contains a pointer or some scalar value (integer, boolean, floating point number, etc) [Shaughnessy, 2013].

[1] *Conservative* garbage collectors are able to collect garbage without precise knowledge of the pointer fields of objects; however they are beyond the scope of this thesis.

2.2 MMTk

The Memory Management Toolkit (MMTk) [2023] is a language agnostic library which enables easy integration of a large selection of high-performance garbage collection algorithms into production programming language runtimes. MMTk is designed to be as flexible as possible and is built around the core principles of modularity, composability, and abstraction without cost, which makes it an excellent platform for researchers to test new garbage collection techniques in a real-world environment.

The original version of MMTk [Blackburn et al., 2004a,b] was part of JikesRVM [Alpern et al., 2000], a meta-circular JVM written in Java. After a successful pilot study by Lin et al. [2016], MMTk was rewritten in Rust to improve safety and performance, and to enable integration with wider variety of runtimes. It currently has full or partial support for OpenJDK (Java), JikesRVM (Java), V8 (JavaScript), MRI (Ruby), GHC (Haskell) and Julia.

MMTk implements a variety of stop-the-world tracing and reference counting[2] garbage collectors. Almost all collection work is parallelised using *work packets*, which are finite units of work that can be distributed and processed by a pool of stateless GC worker threads. Work packets are organised into *work buckets*, which are used to order and synchronise the phases of the collection. Workers will only process work packets from a work bucket if all previous work bucket stages are empty. For more information on MMTk's work distribution system, see Xu et al. [2022] and Huang et al. [2023].

2.3 Cache Prefetching

Over time, the gap between CPU performance and memory bandwidth/latency has widened significantly [Wulf and McKee, 1995]. In order to mitigate the effects of high memory latency, almost all modern CPUs feature a complex hierarchy of caches designed to exploit two common memory access patterns: *spatial locality* (after accessing a memory location, an application is likely to access nearby memory locations), and *temporal locality* (applications are likely to access the same memory addresses multiple times in short succession). This is typically achieved by loading chunks of memory into the cache in units of cache lines (typically 64 B on modern systems) and using sophisticated cache replacement policies (typically some approximation of the least-recently-used policy) to avoid evicting recently-accessed data.

However, these techniques alone are not enough to mitigate all cache misses. For instance, even a simple linear traversal of an array will trigger a compulsory cache miss every time it reaches the end of a cache line and loads the first word of the next line. As another example, given an array of pointers which need to be loaded, it is possible to easily predict in advance the sequence of memory locations which will be accessed, despite the memory access pattern exhibiting poor locality.

[2]Note that LXR [Zhao et al., 2022], the only reference counting collector implemented in MMTk so far, is not part of the mainline release. However, it is currently in the process of being upstreamed.

In such cases, the hardware and/or software may have additional insights which could enable more effective cache usage. *Cache prefetching* is one technique which can be used to reduce cache misses by bringing data into the cache before it is needed. This exploits the memory parallelism of the system to hide the latency of memory accesses behind other instructions, reducing back-end stalls in memory-bound applications.

There are two forms of prefetching: *hardware prefetching* (Section 2.3.1), where the CPU speculatively issues prefetches based on the past pattern of accesses; and *software prefetching* (2.3.2), where the programmer or compiler may manually introduce prefetch instructions based on application-specific insights.

Regardless of whether prefetching is implemented on a hardware or software level, there are several fundamental tradeoffs which need to be considered. The *timing* of the prefetch matters - if the prefetch is issued too early, other useful data may be prematurely displaced from the cache (leading to unnecessary cache misses), or even worse, the prefetched data may be evicted before it ever gets used, wasting memory bandwidth. If a prefetch is issued too late, the memory subsystem will not have enough time to load the data into cache, yielding little to no benefit. In the extreme case of prefetching immediately prior to a load, the performance may decrease, as there is a non-zero cost associated with processing the prefetch instruction itself. Therefore the most effective prefetching distance is inherently related to the latency of main memory accesses. Futhermore, *coverage* and *accuracy* need to be balanced; correctly prefetching all upcoming loads can reduce the number of cache misses, but incorrectly prefetching memory locations that will never be loaded can lead to cache pollution and memory bandwidth saturation.

2.3.1 Hardware Prefetching

Hardware prefetchers watch the pattern of memory locations accessed by the program and attempt to predict future accesses. Since they lack application-level insights, hardware prefetchers are generally limited to predicting relatively simple patterns, such as array accesses with a regular stride distance. Hardware prefetchers are usually less effective on short access stream sequences, as the prefetching unit must observe a sequence of cache misses in order to 'warm up' [Lee et al., 2012]. This is particularly the case in prefetcher units for the lower-level caches, which can only observe the subset of memory accesses which miss the higher level caches. Furthermore, large numbers of concurrent access streams or negative stride lengths can limit prefetcher effectiveness due to hardware limitations [Falsafi and Wenisch, 2014]. Therefore for complex access patterns, software prefetching is more likely to yield performance benefits.

As an example, modern Intel x86 processors feature a streaming prefetcher (which detects sequential accesses within the same cache line, triggering a prefetch of the next line), a stride prefetcher (which separately tracks the stride length of individual load instructions), and a spatial prefetcher (which for every 64 B cache line loaded into the L2 cache, prefetches the adjacent cache line to form a 128 B-aligned pair)

[Intel, 2023].

Some recent microarchitectures (including those from Intel [2022] and Apple [Vicarte et al., 2022]) are capable of performing *data-dependent prefetching* (also known as *data memory-dependent prefetching*), which optimises for the 'array of pointers' access pattern. This type of prefetcher can monitor prefetched cache lines for words which look like a 64-bit pointer, and prefetch the data referenced by those pointers. In the context of garbage collection, data-dependent prefetching is more likely to improve tracing performance than simple stride-based prefetchers.

Note that hardware prefetching mechanisms can vary from one microarchitecture to another (even between microarchitectures from the same vendor), so software developers should not rely on the exact operation of hardware prefetchers. Furthermore, hardware prefetching is inherently speculative, and is not guaranteed to improve application performance.

2.3.2 Software Prefetching

In addition to hardware prefetching, it is possible for software to issue *prefetch instructions* which tell the CPU to prefetch the relevant section of memory. This is useful in cases where the programmer or compiler has higher-level insights into the future memory access patterns of the application (e.g. based on the data structures being used). Software prefetch instructions may be inserted manually by the programmer using compiler intrinsics (e.g. `_mm_prefetch`), or automatically by the compiler (as described in Luk and Mowry [1996]). Software prefetch instructions are usually non-blocking and do not affect the behaviour of the program.

x86-64 offers a variety of instructions for software prefetching. The `PREFETCHT0`, `PREFETCHT1`, and `PREFETCHT2` instructions fetch the relevant cache line into the L1, L2, or L3 cache respectively. The `PREFETCHNTA` (non-temporal prefetch) instruction fetches data as close as possible to the core whilst minimising cache pollution, which is useful for prefetching data that is unlikely to be used more than once. Finally, when a write to a memory location is expected, the `PREFETCHW` instruction can be used to both prefetch the relevant data, and transition the cache line to a 'modified' state. This invalidates any instances of the cache line in other cores' caches, and avoids having to wait for the cache coherency protocols to give exclusive ownership of the cache line.

Note that software prefetch instructions are merely hints and their effect is often dependent on the exact microarchitectural implementation. For instance, the memory management unit may ignore the prefetch locality hint (`T0`, `T1`, etc) or skip prefetch requests where loading the memory location might trigger a page fault. Therefore the performance impact of software prefetching can be microarchitecturally sensitive.

2.4 Summary

This chapter gave a general overview of garbage collection, including the tracing loop, which is at the core of all tracing garbage collectors. Furthermore, MMTk

was introduced as a platform for experimental GC research, and cache prefetching techniques were discussed. The following chapters build on this discussion, exploring ways in which software cache prefetching can be used to speed up the tracing loop.

Chapter 3

Related Work

The previous chapter discussed garbage collection, a form of automatic memory management, and software cache prefetching, a technique for reducing cache misses. The combination of these two ideas, using software prefetching to improve the speed of the tracing loop, has been one area of interest in the research community. This chapter gives an overview of prior work in this area (Section 3.1), and presents an evaluation of the software prefetching implemented into OpenJDK's included garbage collectors.

3.1 Software Cache Prefetching for Garbage Collection

3.1.1 Applications to Tracing Garbage Collection

Boehm [2000] was one of the first to apply software cache prefetching in the context of tracing garbage collection. Building on the tricolour abstraction introduced by Dijkstra et al. [1978], Boehm proposes the strategy of *prefetch-on-grey*. Whenever an object is first visited, it should be 'greyed' (marked) as normal, but also prefetched. This increases the probability that the object will be in the cache by the time it is popped from the mark stack, reducing the number of cache misses. In addition to this, Boehm suggests prefetching a few cache lines ahead when scanning the fields of each object, with the goal of reducing the number of cache misses incurred when scanning objects larger than a cache line. This has the supplementary effect of prefetching adjacent objects, which may be traced in the near future.

Boehm implements these changes into the BDW collector [Boehm and Weiser, 1988], a library-based conservative garbage collector for C and C++. Together, these techniques eliminate over a third of cache miss overheads, reducing benchmark time by 1-17%. However, it should be noted that the most impressive of these results were only observed on a small set of synthetic GC-intensive microbenchmarks, which inflate the overall impact of these improvements. Real-world applications (ptc and ghostscript) only saw up to a 5% speedup.

Cher et al. [2004] conduct a deeper analysis of Boehm's prefetching strategy and propose an alternate design. They confirm that prefetch-on-grey improves the performance of the BDW collector by an average of 16% in GC-intensive benchmarks.

However, when simulating this technique using the SimpleScalar [Austin et al., 2002] architecture simulator, the authors observe that a large proportion of prefetch instructions are mistimed, with up to 27% arriving too early and 15% arriving too late. This is because the BDW collector traverses objects in a *depth-first* order (using a first-in-last-out mark stack), and hence the time between an object being prefetched and its fields being examined can vary greatly, depending on the shape of the object graph. Thus, prefetches are often rendered ineffective.

To solve this problem, Cher et al. propose altering the core tracing loop to add a small queue known as a *prefetch buffer*. As each object is popped off the mark stack, it is prefetched and moved onto the prefetch buffer queue. Then, the object at the head of queue (which was prefetched some time ago) can be popped, loaded and scanned, allowing its child objects to be marked as usual. This *buffered prefetching* design ensures that objects are prefetched and scanned in a predictable, first-in-first-out order. Furthermore, for optimal timing, the size of the prefetch buffer can be tuned according to the memory latency of the system. Overall, buffered prefetching substantially reduces the number of ineffective prefetches, yielding an average speedup of 27% in GC-intensive benchmarks.

Since Cher et al.'s paper was published, multi-core CPUs have become commonplace, and parallel collection is now the norm. Modern parallel GC algorithms do not necessarily traverse the heap in the same depth-first manner as the single-threaded BDW collector used in this paper. Despite this, the core findings of the paper still stand today: software prefetching is most effective when applied in a consistent, timing-controlled manner, independent of the heap traversal order.

Van Groningen [2004] implements buffered prefetching into the runtime of Clean [Brus et al., 1987], a functional programming language. Similar to the approach taken by Cher et al., they add a FIFO mark buffer in front of the existing mark-sweep tracing loop. Van Groningen observes speedups of 11-34% in a compiler compilation benchmark, 19-81% in a linker benchmark, and 20-61% in a merge sort benchmark.

Garner et al. [2007] explore how an alternative tracing loop design can influence the performance of the mark-sweep collector in JikesRVM [Alpern et al., 2000]. The authors develop a sophisticated analysis methodology which they call *replay tracing*, and use it to examine the performance of each individual component of the tracing loop. They observe that the primary costs associated with scanning an object are the two separate memory accesses which need to occur: the load/store of the metadata to mark the object, and the load of the object itself to scan it for pointers. Furthermore, they find that the cost of the mark stack queueing operations (push/pop) are relatively low, accounting for a maximum of 11% of the total running time of the tracing loop.

Based on this insight, Garner et al. propose a new tracing algorithm which delays the mark operation until after the object is popped from the mark stack, improving the temporal locality of the mark and scan operations for a given object. This has the side effect of unconditionally enqueueing all non-null objects, even ones that may already be marked. Although this inflates the number of queueing operations by an average

of 40%, the queueing operations themselves are not the bottleneck; any additional costs are outweighed by the improved locality of marking/scanning. The authors describe this variation of the tracing loop as *edge-ordered enqueueing*, since the number of queue operations will be proportional to the number of edges (references) in the heap graph. *Node-ordered enqueueing* is the term they use to describe the standard mark-sweep tracing loop design, since every node (object) will only be enqueued exactly once.

One major advantage of this alternate tracing design is the increased opportunities for software prefetching. In configurations where the mark bits are stored in the object header, Garner et al. find that the combination of edge enqueueing and buffered software cache prefetching can reduce collector time by 20-30% across a variety of real-world benchmarks.

Wu et al. [2013] build upon the optimisations described by Cher et al. and Garner et al., and propose a new technique to reduce the number of unnecessary prefetches. Whenever a CPU issues a prefetch, the entire cache line is brought into cache. The authors observe that if multiple objects in the same cache line are queued for scanning, then multiple prefetch instructions will be issued for the same cache line. These unnecessary prefetches could potentially reduce performance.

In order to eliminate duplicate prefetches, Wu et al. modify the JikesRVM tracing loop to keep track of the last prefetched address in a local variable. Then, before issuing a prefetch to the next object address, they check to make sure that the distance between the two addresses is less than a particular threshold. The value of this threshold can be varied according to the cache line size. In their evaluations of this technique, the authors demonstrate a 5% average speedup in tracing time of both the mark-sweep and semispace collectors.

Tracing garbage collection can be viewed as a graph traversal problem. Graph traversal involves a lot of pointer chasing, which cannot easily be accelerated through prefetching. This is due to the fundamental limitation that in a pointer chain, the address of the n^{th} node can only be determined after loading the contents of the $(n-1)^{th}$ node. There have been several attempts [Luk and Mowry, 1996; Roth and Sohi, 1999] by compiler researchers to improve graph traversal speed by adding *jump pointers* to each node. Jump pointers are fields which contain the address of another node that will be accessed in the near future, based on the ordering recorded during a previous traversal of the graph. This allows the traversal algorithm to determine which nodes will be accessed in advance, without having to load the intermediary pointer chain. Jump pointers are useful when combined with software prefetching, in a technique known as *jump pointer prefetching* or *history-based prefetching*.

Cahoon [2002] implements compile-time generation of jump pointers into Vortex [Dean et al., 1996], an optimising compiler for object oriented languages like Java. During mutator execution, jump pointer prefetching is used to optimise the traversal speed of linked data structures. To maintain the jump pointers, Cahoon uses the GC traversal order to update the jump pointer values on each collection. Furthermore,

they modify the garbage collector to respect the weak reference semantics of jump pointers. However, Cahoon does not use jump pointer prefetching to optimise the actual garbage collection trace itself.

To my knowledge, there has been no other research on the effectiveness of using jump-pointer prefetching for tracing garbage collection. However, one major disadvantage of any such attempted optimisation would be the considerable space overhead (one extra word for every object). Garbage collection is fundamentally a space-time tradeoff, and benefits of jump-pointer prefetching are unlikely to outweigh the higher costs of a larger heap. Furthermore, in modern production environments, heap sizes are deliberately kept small to enable compressed pointers to be used, which are known to have significant performance[1] and memory footprint advantages.

3.1.2 Other Applications to Garbage Collection

Cache prefetching has also been applied to aspects of garbage collection beyond the main tracing loop. Although such applications are beyond the scope of this thesis, they are discussed here for context.

Paz and Petrank [2007] explore the applications of software cache prefetching to *reference counting* collectors. They identify five major opportunities for prefetching: two in increment processing, two in decrement processing, and one when scanning the mark bitmap to determine which objects to move to the free list. Based on these observations, they implement software cache prefetching into JikesRVM's reference counting collector. Overall, these changes result in an overall reduction in GC time of 8.7%, and an overall benchmark time improvement of 2.2%.

Yang et al. [2011] measure the direct and indirect costs of bulk zeroing, hotpath allocator zeroing, and concurrent zeroing in OpenJDK and JikesRVM. The authors evaluate a wide range of zeroing methodologies, both with hardware prefetching enabled and disabled. They find that an effective hardware prefetcher is essential for high performance hotpath zeroing, in order to avoid cache misses on each allocation.

Newer versions of OpenJDK [2023] expand on this idea by introducing *software* prefetching into the allocator hotpath. This is discussed in further detail in Section 3.2.

3.1.3 Reflection on Prior Work

The field of garbage collection has evolved considerably since many of these works were published. Heap sizes have grown from megabytes to gigabytes and even terabytes. Parallel garbage collection is necessary to make use of the hardware available; commodity desktop-class CPUs now come with up to 16 cores, and recent

[1] A quick A/B comparison of enabling and disabling compressed pointers in OpenJDK indicates that they improve mutator time by 2-3% and GC time by up to 35%.

server hardware sports up to 128 cores (256 threads) per socket with over 1 GB of last-level cache [Alcorn, 2023]. The architectural simulators used to obtain measurements in several of the prior papers are not necessarily representative of the immense complexity of today's speculative, out of order, superscalar CPUs with asymmetric core designs and complex cache hierarchies. Recent work uses richer workloads, including the DaCapo [Blackburn et al., 2006] and Renaissance [Prokopec et al., 2019] benchmark suites which are more representative of today's production applications that usually have many threads and large heap sizes. It is more common to evaluate new designs in the context of production language runtimes such as OpenJDK, as opposed to research virtual machines like JikesRVM. Operating systems now provide powerful, low-overhead mechanisms such as eBPF which can be used to obtain high-fidelity measurements of collector internals [Huang et al., 2023]. Given the changing hardware and software landscape, it is clear that new research on the topic is necessary.

3.2 Software Cache Prefetching in OpenJDK

In the previous section, prior work on software cache prefetching for tracing garbage collectors was discussed. However, findings from academic research are not always directly integrated into production software. In this section, the software prefetching implementations in OpenJDK are evaluated, in order to gain an understanding of how state-of-the-art real-world collectors are taking advantage of these techniques.

3.2.1 Overview of OpenJDK Prefetching

All OpenJDK GCs perform bump-pointer allocation into a thread-local allocation buffer (TLAB) [Shipilev, 2019]. OpenJDK uses software prefetch instructions to fetch memory locations ahead of the allocation cursor, so that when new objects are allocated and initialised the relevant memory will already be in cache. In workloads with high volumes of allocation, this can avoid some back-end stalls resulting from full store buffers [Jaffer and Warburton, 2020]. OpenJDK has several run-time flags which can be used to configure the behaviour of the allocator prefetcher, including the prefetch distance ahead of the cursor, the number of cache lines to be prefetched, the type of prefetch instruction to use, and whether to check the TLAB limit before prefetching.

Software cache prefetching is used relatively minimally within the collectors themselves. In the Parallel GC used for our evaluation below, prefetching is employed to optimise the compaction phase of full-heap collections. When compacting the heap, the collector maintains a source cursor (which performs a linear scan through the space, looking for live objects), and a destination cursor (which indicates the free memory location that live objects will be copied to). Prefetch instructions are issued ahead of both of these pointers, to avoid cache misses on subsequent compaction operations. Notably, prefetching does not appear to be used as part of the tracing loop.

3.2.2 Evaluation

In order to evaluate the effectiveness of software cache prefetching in OpenJDK, three configurations were tested:

- All prefetching disabled (this was achieved by commenting out the source code responsible for generating prefetch instructions);
- GC prefetching enabled, allocator prefetching disabled (controlled by passing the `-XX:AllocatePrefetchStyle=0` option)
- Both GC and allocator prefetching enabled (this was the default configuration of OpenJDK)

All experiments were run on OpenJDK 11 (`jdk-11.0.19+1-mmtk` branch[2]), using the Parallel GC (`-XX:+UseParallelGC`). This collector was chosen due to it being a stop-the-world GC (rather than a concurrent GC, like G1, ZGC, etc), which makes it more comparable to the collector implementations used in prior research. The DaCapo suite [Blackburn et al., 2006] was used for benchmarking, configured to run 5 iterations and 40 invocations for each benchmark. Other details of the test environment, including the hardware and operating system information, are discussed in Section 4.3.1.

Figure 3.1 plots the total stop-the-world GC time on AMD Zen 4 across the three configurations, with times normalised to the scenario where prefetching is completely disabled. The results demonstrate that the combination of both GC and allocation prefetching has little to no impact on GC time, averaging only a 0.4% performance improvement. The mutator observes a small slowdown of 0.3%. Enabling GC prefetching (but leaving allocation prefetching disabled) actually negatively impacts the performance of some benchmarks, with `biojava` and `spring` observing an 18.2% and 10.7% increase in GC time respectively. This is despite a 1% reduction in L1 cache misses and a 4% reduction in back-end stalls. Notably, there is not a single benchmark that exhibits a statistically significant performance improvement as a result of enabling some form of prefetching.

Figure 3.2 shows the same experiments run on Intel Coffee Lake. Only a single benchmark (`h2`) experiences a statistically significant reduction in GC time (1.2%) as a result of enabling prefetching. On average, the performance impact of GC and allocation prefetching is slightly negative, with the GC time increasing by 0.4% (for GC prefetching only) and 0.8% (GC and allocation prefetching). Furthermore, these prefetching techniques had no measurable impact on mutator time.

In summary, this evaluation demonstrates that OpenJDK's prefetching strategy is in the best cases ineffective, and sometimes results in a performance regression. This is in stark contrast to the existing body of research on software prefetching for tracing garbage collection, which has been able to achieve consistent, significant performance

[2]Commit 28e56ee32525c32c5a88391d0b01f24e5cd16c0f

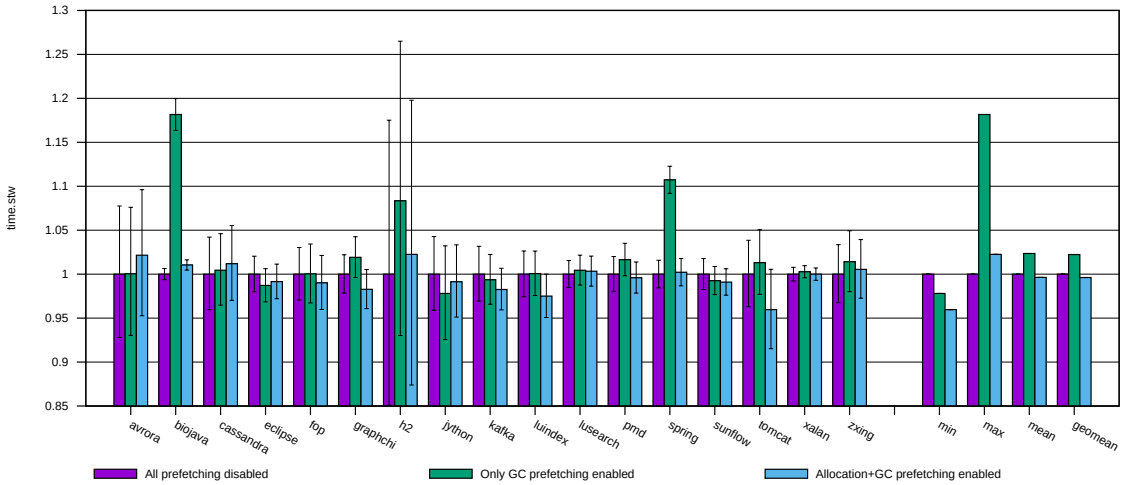


Figure 3.1: Comparison of software prefetching options in OpenJDK (Zen 4)

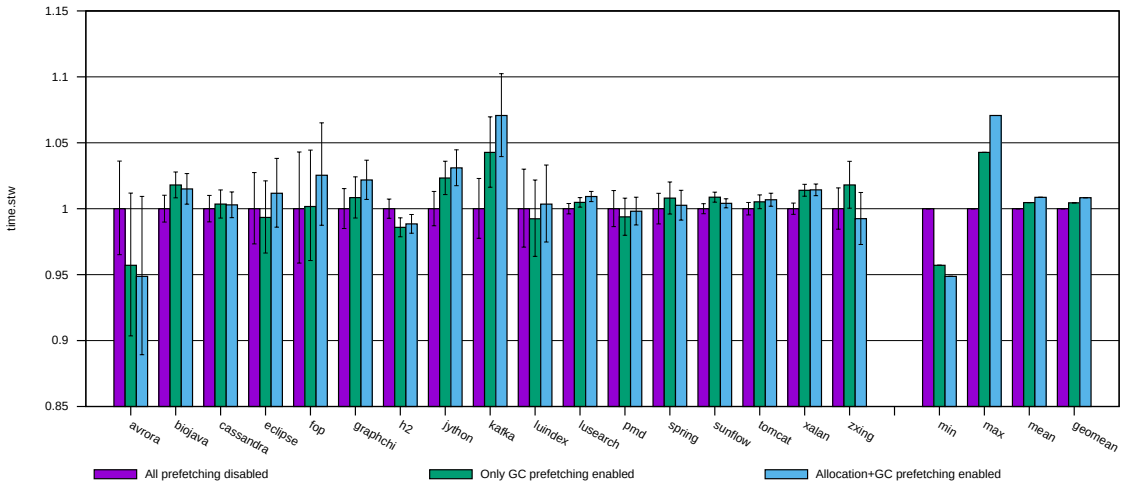


Figure 3.2: Comparison of software prefetching options in OpenJDK (Coffee Lake)

improvements. These results could be explained by a number of factors, including the possibility that software prefetching may not be as effective when applied to moving garbage collectors, or that OpenJDK's prefetching implementation may be misguided.

3.3 Summary

This chapter discussed a variety of prior work on software cache prefetching for tracing garbage collection. GC prefetching has been explored in a number of papers over the years, however the full design space of tracing loops, and its relation to software cache prefetching, has not been explored in great depth. Furthermore, it is clear that the prefetching implemented into OpenJDK is not very effective. These findings indicate there is a significant mismatch between the known state-of-the-art in academic research, and the real-world implementations of mature, production-grade language runtimes.

In the following chapter, I introduce *auxiliary tracing*, a framework for evaluating new tracing loop designs and prefetching implementations in a controlled environment. This enables significantly greater scope for analysis of these ideas, in a way that would be difficult or impossible in the context of OpenJDK's native GCs.

Chapter 4

Auxiliary Tracing

The previous chapter discussed a variety of previous work on reducing the cost of garbage collection by improving the performance of the main tracing loop. In this chapter, I propose a new technique - called *auxiliary tracing* - that enables efficient implementation, evaluation and comparison of both new and existing tracing loop designs.

I begin in Section 4.1 by introducing the auxiliary tracing framework and discussing the motivating factors that led to it's creation. Section 4.2 discusses how I designed and implemented the auxiliary tracing framework into MMTk, a versatile platform for GC research. Next, in Section 4.3.2, I validate the correctness and performance of this implementation. Finally, I demonstrate versatility of auxiliary tracing through two case studies: one which determines the effect of heap 'layout' on tracing speed (Section 4.4), and second, I perform a performance evaluation of an object scanning technique.

4.1 Introduction

The tracing loop is known to be one of the most performance-critical components of any tracing garbage collector. Understandably, it has been the focus of significant research efforts, with many design alternatives being explored including structural variations, new object scanning techniques, and different heap traversal orders.

Research of new tracing techniques requires (i) implementation of changes to the tracing loop, and (ii) rigorous evaluation, with respect to a pre-existing baseline.

However, from a software engineering perspective, depending on the specific nature of the required changes, this can be can be very difficult. In the same way that assumptions about garbage collection behaviour often leak into programming language implementations and specifications [Jibaja et al., 2011], assumptions about tracing loop behaviour are also often baked into the garbage collector itself. This makes it hard to change the core tracing loop without introducing unintended side effects that may affect collector performance or correctness.

To enable easier research and experimentation, I developed *auxiliary tracing*, a novel framework for implementing and evaluating new and existing tracing loop designs. The auxiliary trace is a standalone trace of the heap which occurs directly

before the main closure phase of the collection cycle (Figure 4.1). It solely marks and scans objects; no actual collection work is done as part of the auxiliary trace. Instead, the main garbage collection algorithm is responsible for tracing the heap a second time, and sweeping, evacuating or compacting objects as usual.

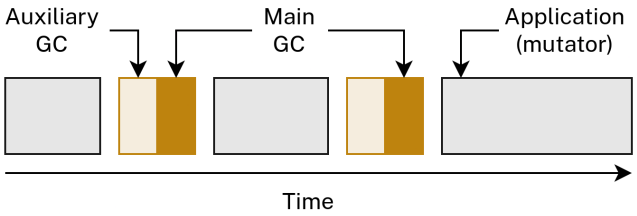


Figure 4.1: Timing of the auxiliary trace with respect to the main closure phase

Crucially, the auxiliary trace is entirely *side-effect free*: it maintains it's own metadata, and never makes any modifications to the heap which could interfere with the main collection. This makes it an extremely useful framework for experimentation and research.

The auxiliary trace is deliberately run before the main trace, to ensure that the cache reacts similarly to the auxiliary trace as it would to the main trace; if the auxiliary trace were run after the main trace, then that would displace data in the cache and make it harder to form reliable, realistic measurements of tracing speed.

Auxiliary tracing offers two main possibilities which may have previously been difficult to implement:

- The ability to test tracing irrespective of heap layout or collection algorithm (Section 4.4).
- The ability to easily implement and evaluate a wide variety of tracing algorithm designs.

Whilst the auxiliary trace may seem similar to the Sanity GC implemented in MMTk (and similar debugging GCs implemented in other collection frameworks), it has several key differences.

- Sanity GC is designed for debugging, and has numerous assertions enabled that are unnecessary during regular garbage collector operation. The auxiliary trace is designed for performance evaluation, and thus only performs the checks necessary for collector correctness.
- The sanity GC is designed to be as simple as possible, so that its correctness can be verified by visual inspection (even if this simplicity comes at the expense of performance[1]). On the other hand, the auxiliary trace must have maximum performance, and may sacrifice simplicity to achieve this goal. (However, the

[1]For instance, the sanity GC in the current version of MMTk core 'marks' objects by adding their object reference to a hash set. This is extremely time and space inefficient.

axillary tracer is still significantly simpler than the main trace, due to the lack of object reclamation)

- The auxiliary trace runs before the main trace, whereas the sanity GC runs after it. This has implications with regards to the heap 'shape', as well as the observed cache behaviour (since the main trace pollutes the cache turing the heap traversal).

4.2 Design and Implementation

I implement auxiliary tracing into the new Rust version of MMTk [MMTk Contributors, 2023]. On a high level, the auxiliary tracing framework presents a trait called `AuxiliaryTraceWork`, which describes a work packet for an abstract tracing loop algorithm that is based around a queue containing some form of `Item`. Every concrete tracing loop algorithm that implements the `AuxiliaryTraceWork` trait needs to choose a specific item type for the queue (e.g. `Edge`, `ObjectReference`, etc), and implement four methods:

- **`new`** - this is simply boilerplate code to create a new `AuxiliaryTraceWork` packet of the correct type.
- **`process_queue_item`** - this defines how a single queue `Item` should be processed.
  - For instance, in a Node-ObjRef tracing algorithm (Section 5.2, Listing 5.3), this method will be passed a single `ObjectReference` as input. It needs to scan the object, load the pointer fields, mark each of the children, and call `add_child_element()` on every child reference that needs to be enqueued.
- **`scan_unmarked_slots`** and **`scan_unmarked_objrefs`** - these define how roots should be processed.
  - Depending on the requirements of the runtime being used, roots may be provided as slots or object references or both.
  - However, depending on the tracing algorithm, it may expect items on the queue to be already marked or waiting to be marked, or might expect the queue items to be of a different type (e.g. if the roots are provided as slots but the algorithm expects object references).
  - Therefore some pre-processing of roots may be necessary before they can be added to the main queue. For instance, if provided with a list of root slots, the Node-ObjRef algorithm may load each slot to obtain an object reference, then mark the child objects and enqueue them for scanning using regular `AuxiliaryTraceWork` packets.
  - Where possible, I parallelise the process of processing the roots using a dedicated work packet. Otherwise, the performance of the auxiliary trace is harmed due to the serial execution of the root processing function.

Furthermore, the `AuxiliaryTraceWork` trait has several other methods that provide the machinery necessary for processing the queue, marking objects, adding new elements to the queue, and flushing the output queue to a new `AuxiliaryTraceWork` packet when it gets full. These methods have default implementations that are adequate for most designs. However, if a particular tracing loop algorithm wishes to override a specific aspect of the auxiliary tracing framework (e.g. to implement dual-queues, or implement prefetching), it can simply re-implement one of the provided methods.

Finally, the auxiliary tracing framework has two additional work packet types (called `AuxiliaryPrepare` and `AuxiliaryRelease`) which are responsible for starting and stopping the trace. This involves passing the roots to the `scan_unmarked_slots` and `scan_unmarked_objrefs` functions to generate the first `AuxiliaryTraceWork` packets, resetting metadata from the previous collection, and starting/stopping statistics loggers.

To enable the auxiliary trace to run before the main trace, I added several new work buckets called `AuxPrepare`, `AuxClosure`, and `AuxRelease` (Listing 4.1). The `AuxiliaryPrepare`, `AuxiliaryTraceWork` and `AuxiliaryRelease` packets are added to each of these buckets. This ensures that the auxiliary trace only begins after root scanning is finished, and prevents the main closure phase from starting until after the auxiliary trace is complete. Furthermore, the guarantees of MMTk's work bucket system ensure that no other work will be interleaved with the auxiliary trace, allowing fair measurement of the tracing time.

Listing 4.1: Work bucket stages for the auxiliary trace

```
pub enum WorkBucketStage {
  Unconstrained,  // Always open
  Prepare,        // Preparation work & root scanning
  AuxPrepare,     // Generate AuxiliaryTraceWork packets
  AuxClosure,     // Perform trace of the heap
  AuxRelease,     // Auxiliary tracing cleanup
  Closure,        // Transitive closure via strong references
  SoftRefClosure, // Handle Java-style soft references
  ...
}
```

Together, these design elements allow a fully-functional, high-performance, parallelised tracing loop to be created by implementing just four simple methods. A visualisation of an auxiliary trace is shown in Figure 4.2.

4.3 Validation

In order for the auxiliary trace to be an effective model of the main tracing loop, it needs to be both *correct* and *performant*. This section discusses how the auxiliary trace was validated against existing baseline implementations.

Figure 4.2: A visualisation of an auxiliary trace, generated using the GC visualisation techniques developed by Huang et al. [2023]. The lilac work packets (left of center) represent the auxiliary trace, and the orange work packets (center of the diagram) represent the main trace. This trace performs the Edge-Slot-Dual tracing algorithm, using the `AuxTraceEdgeSlotPackets` which implement the `AuxiliaryTraceWork` trait.

4.3.1 Experimental Methodology

The following is the standard empirical evaluation methodology used throughout this thesis. For completeness, all hardware and software information is discussed, even aspects which are not directly relevant to the following section.

Three primary hardware platforms are used to perform all experiments: two AMD machines (based on the Zen 2 and Zen 4 microarchitectures), and an Intel machine (based on the Coffee Lake microarchitecture). Their specifications are listed in Table 4.1. All three machines run Ubuntu 22.04.3 LTS with a `6.2.0-33-generic` kernel. To reduce noise in measurements, frequency scaling is disabled on all machines, and almost all background daemons are disabled.

All evaluations use a development version of `mmtk-core`[2], based on the v0.18 release. I use the OpenJDK runtime [OpenJDK contributors, 2023] for my evaluations, as it is the most mature and well-optimised VM integrated with MMTk. Compatible versions of the `mmtk-openjdk` binding[3] and OpenJDK fork[4] are used. MMTk is built with version 1.66.1 of the Rust compiler in `release` mode using profile-guided optimisation[5].

Experiments are run across seventeen up-to-date, diverse benchmarks from RC1[6] of the Chopin branch of the DaCapo Benchmark Suite [Blackburn et al., 2006]. The `batik` and `jme` benchmarks are excluded because they do not perform a GC with the configured heap size. The `tradesoap` and `tradebeans` benchmarks are also excluded

[2]Commit f1a0bb7fbec97dd84e35a40e8be01cf5018f2f9e

[3]Commit 54a249e877e1cbea147a71aafaafb8583f33843d

[4]Commit 28e56ee32525c32c5a88391d0b01f24e5cd16c0f (branch `jdk-11.0.19+1-mmtk`)

[5]PGO is based on profiling data from 5 iterations of the `fop` benchmark with a 4 MB stress factor to increase the frequency of garbage collection.

[6]DaCapo 23.8-Chopin Release Candidate 1. At the time of writing, the official release of Chopin, the new major DaCapo version, is imminent.

Table 4.1: Hardware specifications

Architecture	Coffee Lake	Zen 2	Zen 4
Manufacturer	Intel	AMD	AMD
Model	Core i9-9900K	Ryzen 9 3900X	Ryzen 9 7950X
Year	2018	2019	2022
Technology node	14 nm	6 nm/7 nm (Core/IO)	5 nm/6 nm (Core/IO)
Clock	3.6 GHz	3.8 GHz	4.5 GHz
SMT $\times$ Cores	2×8	2×12	2×16
L1 Data Cache	32 KB $\times$ 8	32 KB $\times$ 12	32 KB $\times$ 16
L2 Cache	256 KB $\times$ 8	512 KB $\times$ 12	1 MB $\times$ 16
L3 Cache	16 MB $\times$ 1	16 MB $\times$ 4	32 MB $\times$ 2
Memory Size	128 GB	64 GB	64 GB
Memory Type	DDR4 3200 MHz	DDR4 2133 MHz	DDR5 4800 MHz

due to known compatibility issues[7]. Unless otherwise specified, the heap size for each benchmark is set to be twice the minimum heap size needed to complete the benchmark within the timeout period of 20 mins (i.e. a 2x minheap).

When executing benchmarks, the following JVM arguments are applied: `-server` to use the C2 JIT compiler; `-XX:-TieredCompilation -Xcomp` to speed up the warmup of the JVM; `-XX:+DisableExplicitGC` to disable application-initiated collections; `-XX:MetaspaceSize=500M` to make the metaspace large enough to avoid any need for metaspace garbage collection (since MMTk does not handle this); and `-XX:+UseThirdPartyHeap` to force OpenJDK to use MMTk as its memory manager.

Each benchmark is invoked 20 times, unless otherwise specified. In a deviation from standard empirical evaluation guidelines in the garbage collection literature, only a single iteration (`-n 1`) is used for each benchmark invocation, rather than performing multiple iterations and measuring the last. This approach is taken for three main reasons: (i) mutator performance is not being measured, so there is no need to wait for the JIT compiler to warm up; (ii) in our testing, increasing the number of invocations and reducing the number of iterations gives the most stable results with the smallest spread; and (iii) performing additional iterations dramatically increases the total running time of the benchmark suite[8], without any noticeable benefit.

Finally, unless otherwise specified, the Immix [Blackburn and McKinley, 2008] algorithm is used in the main trace to perform the actual garbage collection. This is configured at run-time using the environment variable `MMTK_PLAN=Immix`.

[7] https://github.com/dacapobench/dacapobench/issues/198

[8] This was a major concern, as longer benchmark running times would have reduced the number of experiments that could be run, particularly in Chapter 6, where there was a very large state space to explore.

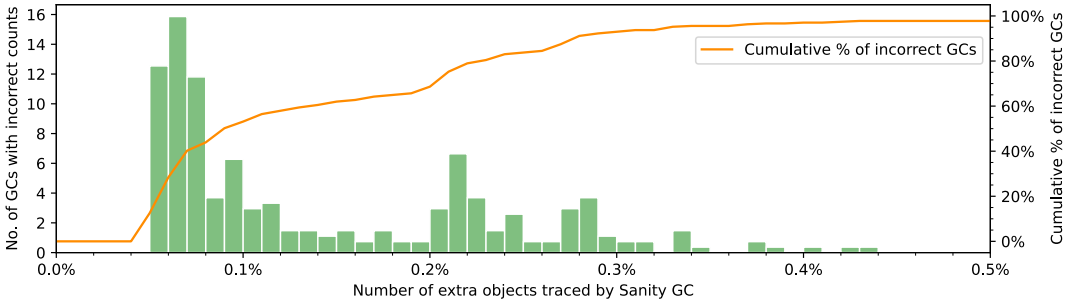


Figure 4.3: Distribution of errors in GCs with an incorrect number of marked objects (Zen 2)

4.3.2 Validation of Correctness

To validate the correctness of the auxiliary tracer, I configure MMTk to run an auxiliary trace and a sanity trace on every GC, and log the number of *unique* objects marked by each. If the two traces report the same object counts, then we can have a high degree of confidence that the auxiliary trace is correctly implemented.

For the majority of benchmarks, the sanity and auxiliary trace report exactly the same number of marked objects for every collection cycle. However, for three of the benchmarks (`lusearch`, `sunflow`, and `xalan`), in a small number of collection cycles (0.7%) the sanity GC marks *more* objects than the auxiliary trace. For the remaining 99.3% of collections, the number of objects marked by each trace is identical. Of the collection cycles which have incorrect object counts, the error is usually less than 0.1%. Figure 4.3 plots the relative magnitude of the error for all collection cycles which exhibited a different number of marked objects.

Although we were not able to definitively diagnose the source of this issue, my hypothesis is that it is related to Java's finalisation semantics. In Java, an object which is undergoing destruction can resurrect itself using a finaliser, by adding a reference to itself from another object which is still live. In MMTk, finalisers are processed in the `FinalRefClosure` work bucket stage, which occurs after the auxiliary trace is finished. However, the sanity GC is not scheduled until the `Final` work bucket stage, which can only be processed after finalisation is complete. Therefore it is possible that the size of the heap may increase between the auxiliary GC and the sanity GC. This hypothesis is supported by the observation that the sanity GC never marks *less* objects than the auxiliary trace, only more. Furthermore, only specific benchmarks are affected by this issue, which makes sense as not all benchmarks would implement finalisers.

Regardless of the root cause of this problem, my measurements show that the issue is extremely rare. Even when the auxiliary trace marks a different number of objects, the relative difference is small enough that any variation in the time taken to execute the auxiliary trace would be smaller than our measurement precision. Therefore, I argue that these minute differences do not cast doubt on the main findings of this thesis.

4.3.3 Validation of Performance

In order for auxiliary tracing to be an effective model of the main tracing loop, it must have competitive performance. One way to validate the tracing speed of the auxiliary tracing implementation is to compare its execution time to that of the closure phase of the main garbage collector.

Since the auxiliary trace does not compact or evacuate objects, it is important to compare it to a non-generational non-moving garbage collection algorithm. This ensures that both traces perform a similar amount of work. MMTk features two garbage collectors which meet this criteria: non-moving immix (a variation of the immix [Blackburn and McKinley, 2008] collection algorithm that does not perform defragmentation), and mark-sweep. However, MMTk's mark-sweep implementation is known to have some serious performance issues, so the non-moving immix collector is chosen as a baseline.

Note that when measuring the time taken by the main trace, it is important not to run the auxiliary trace beforehand, as it may displace data in the cache and affect the performance results. Therefore I create two builds of MMTk, one which has the auxiliary trace enabled, and another which has it disabled. Both builds use non-moving immix as the underlying garbage collection algorithm, to ensure that the underlying heap structure is roughly the same.

I run twenty invocations of the benchmark suite on Zen 2, with the heap size set to four times the minimum heap[9]. For each configuration, I record the total time spent executing the relevant tracing loop across all garbage collection cycles. This is divided by the number of GCs triggered during the benchmark, to obtain the average duration of each trace of the heap. The results are presented in Figure 4.4. Overall, the auxiliary trace is 4% faster, beating the non-moving immix trace in 14 of the 16 benchmarks. However, in two of the benchmarks, `eclipse` and `zxing`, it performed substantially worse, taking 2.2x and 2.9x as long to trace the heap, respectively. When these two benchmarks are excluded, the auxiliary trace outperforms the main trace by 16.5% on average.

The cause of these two outlier results is unclear. Performance counter data indicates that on `eclipse`, the auxiliary tracer experiences 1.4x as many L1D cache misses and 1.7x as many back-end stalls as compared to the main trace; however on `zxing` the auxiliary tracer exhibits opposite statistics, with 6.8% and 27% less misses and stalls respectively. Many of the other benchmarks also display wide-ranging variations in these statistics, indicating that cache misses and back-end stalls may not have a large impact on some benchmarks. Furthermore, a visual inspection of the execution the auxiliary trace (using the GC visualisation techniques introduced by Huang et al. [2023]) on the `eclipse` benchmark did not uncover any major work packet distribution problems that might have an outsized impact on tracing time.

Overall, these results show that the auxiliary tracer is competitive with the existing tracing loop implementation. This makes it a credible platform for evaluating new

[9]Smaller heap sizes were unable to run most benchmarks due to the inability of the non-moving immix GC to defragment the heap, which resulted in out-of-memory errors.

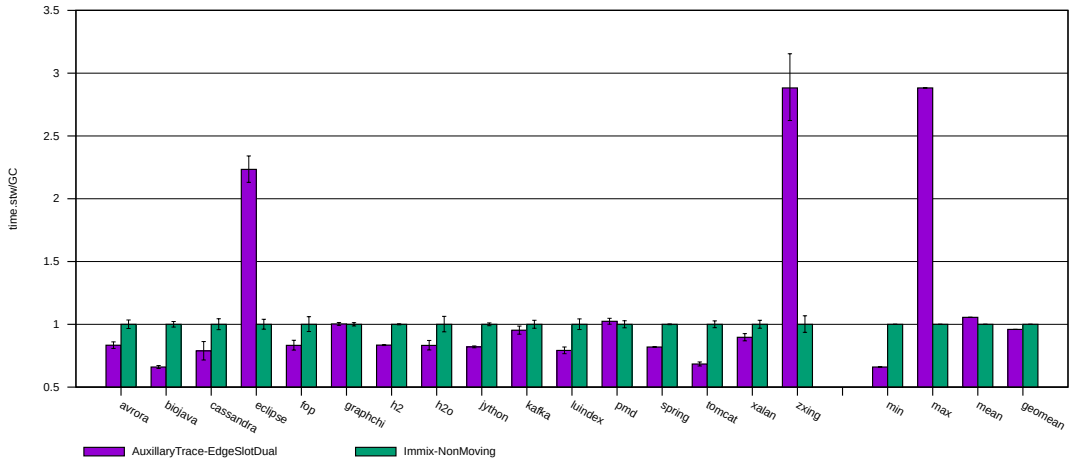


Figure 4.4: Average execution time of auxiliary trace vs non-moving immix trace (Zen 2)

performance optimisations in the context of the tracing loop.

4.4 Case Study: Comparison of Heap Layouts

The logical structure of the heap is defined by the shape of the object graph. However, the physical organisation of the heap in memory can change depending on where each object is located. These varying *heap layouts* can have an impact on performance, particularly if the physical placement of objects is reflective of the logical relationships between them (with respect to the object graph).

Compacting and evacuating garbage collectors can alter the layout of the heap by moving objects, thereby changing application performance. For instance, it is well known that the semispace collector improves mutator performance by copying objects to the *tospace* in heap traversal order [Blackburn et al., 2004a]. Researchers have even explored *online object reordering* schemes which reorder objects based on observed mutator access patterns to improve locality [Huang et al., 2004; Yang et al., 2020]. Even non-moving collectors can influence the heap layout: design choices in the allocator (e.g. free list vs slab vs bump-pointer, global vs thread-local buffers, range of size classes, etc) can impact where objects are placed in memory. Although the potential impact of heap layout on mutator performance is clear, one question remains unanswered: how does the heap layout affect *garbage collector* performance? In particular, how does the heap layout produced by a garbage collector affect the speed of the subsequent tracing collection?

In the past, this has been difficult to benchmark because object movement happens during the tracing phase, meaning that the tracing speed will be directly impacted by any changes to the object movement code. However, auxiliary tracing enables us to decouple these two aspects of the collector, allowing us to measure tracing speed independently of the algorithm responsible for generating a particular heap shape.

To answer the question posed by this case study, I select five different garbage

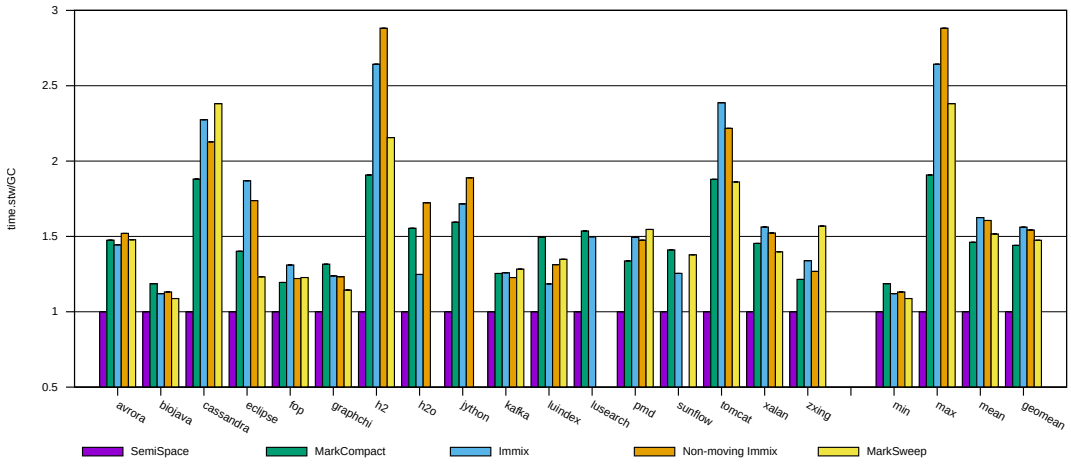


Figure 4.5: Average auxiliary trace time, by garbage collection algorithm (Zen 2)

collection algorithms which generate a spectrum of heap layouts: semispace, mark-compact, immix, non-moving immix, and mark-sweep. These algorithms vary significantly based on allocation policies and the degree to which they move objects. I configure the auxiliary tracer to use the canonical mark-sweep tracing loop design (also known as *Node-ObjRef*[10]), and run it in front of each of the five GC algorithms. The average execution time of the auxiliary tracing loop for each GC algorithm is shown in Figure 4.5 below, normalised to the semispace results.

Unsurprisingly, the heap layout produced by the semispace algorithm results in the shortest auxiliary tracing time. This is because semispace copies objects to the tospace in direct traversal order; hence, any subsequent traversals of the heap will effectively be a linear scan through memory, which is a very cache-friendly operation. On the other hand, the mark-compact algorithm, despite producing a heap layout with a small memory footprint, does not improve auxiliary trace times as much as the semispace collector. This could possibly be because the Lisp 2 compaction algorithm implemented in MMTk simply compacts all objects down to the start of the space, without changing their ordering. Therefore the compaction operation does not improve the locality of objects with respect to the heap traversal order. Surprisingly, the Immix algorithm, which can perform defragmentation, leads to lower tracing performance than the mark-sweep collector, which is non-moving. This could be because the defragmentation algorithm used in Immix does not take traversal order locality into account when moving objects. Finally, the non-moving immix collector exhibits similar results to the main immix algorithm, presumably because they use the same heap structure.

This case study demonstrates that the auxiliary tracing framework opens up new evaluation possibilities which were not previously possible.

[10]See Section 5.2 for more details on this naming scheme

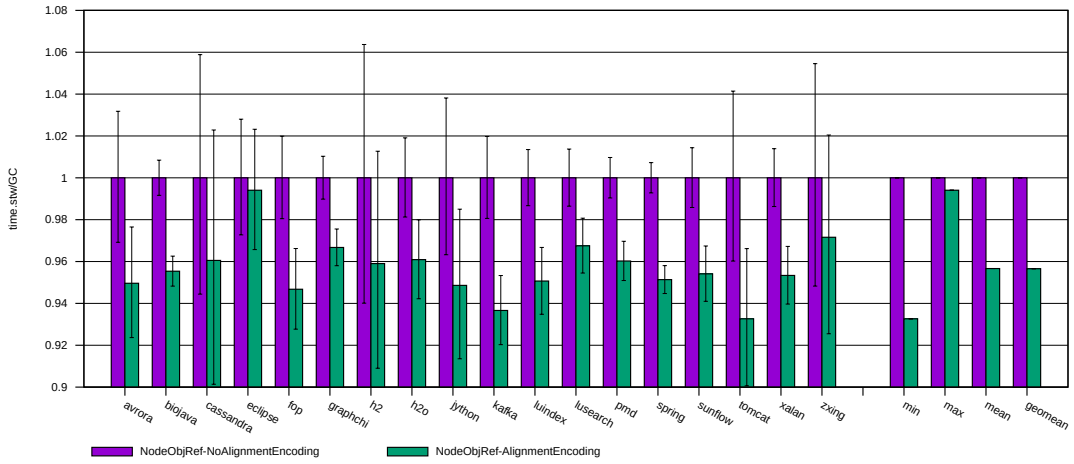


Figure 4.6: Execution time impact of alignment encoding (Zen 2)

4.5 Case Study: Alignment Encoding

Recently, Cai [2023] implemented an object scanning optimisation known as *alignment encoding* into the MMTk OpenJDK binding. This technique, which was originally proposed by Garner et al. [2011], involves encoding within the `klass` pointer (that resides in the header of each object) the locations of the object's pointer fields. This optimisation is motivated by the observation that (i) all instances of a class must contain a klass pointer to the same type information block (TIB), and (ii) all instances of a class have the same layout of pointer fields. By controlling the location at which the TIB is allocated, it is possible to encode some information about the object 'shape' in the address of the TIB. Therefore, for most objects, the locations of the pointer fields can be determined simply by reading the klass field, without needing to actually load the pointer. This reduces the cost of object scanning.

As a case study of the flexibility of the auxiliary tracing algorithm, I added support for alignment encoding to the auxiliary tracing loop, and benchmarked it against the standard object scanning strategy. The results are shown in Figure 4.6.

Overall, the alignment encoding optimisation reduces heap tracing times by approximately 4.3%. This is in line with the results observed by Cai, who saw a 4.0% speedup. The fact that these two results are extremely similar helps validate the effectiveness of the auxiliary tracing framework as a model of the main tracing loop.

4.6 Limitations

Whilst it is clear that auxiliary tracing is a flexible and powerful tool for evaluating new tracing loop designs, it is not a perfect model. Namely, there are a few limitations of what can be tested using the auxiliary tracing loop:

- The auxiliary trace cannot be used to evaluate evacuation or compaction techniques, since it is designed to be side-effect free, and hence must not move

objects.

- In order to avoid interfering with the main tracing loop, all marking metadata for the auxiliary trace is stored on the side in a dedicated bytemap. Although it would be possible to implement in-header mark bits for the auxiliary trace, this would either require the addition of an extra word in front of every allocation, runtime-specific support for the auxiliary mark bits, or reusing the existing mark bits in the object header and clearing them before the main GC. None of these solutions are optimal. Therefore, the main tracing loop may be a better environment for evaluating the effect of the location of the mark bit.
- As currently implemented, the auxiliary tracer can only model a full-heap trace, not nursery collection. Therefore the insights gained from experimentation with the auxiliary trace may not be directly applicable to generational collectors. However, in principle, it shouldn't be impossible to implement auxiliary tracing of only part of the heap.
- Auxiliary tracing can only measure changes in execution time for the tracing loop, not the overall GC time

Despite these limitations, I believe that auxiliary tracing is a useful tool, as demonstrated by the following chapters of this thesis.

4.7 Summary

In this chapter we introduced a new framework for evaluating variations on the tracing loop. We also presented a study of how this technique can be used to evaluate object scanning techniques. In the next chapter, we use our new auxiliary tracing framework to evaluate structural variations on the core tracing loop.

Chapter 5

Study of Tracing Loop Algorithms

Chapter 4 presented *auxiliary tracing*, a novel framework for performing performance analysis on the canonical garbage collection tracing loop. In this chapter, I leverage this framework to evaluate variations on the structure of the core tracing loop present in all tracing garbage collectors.

In Section 5.1, I discuss several possible design decisions that can be made when implementing a tracing loop. Then, in Section 5.2, I present a novel taxonomy which can be used to classify tracing loop algorithms. Finally, in Section 5.3 I implement seven different tracing loop algorithms in the auxiliary tracing framework and evaluate both their performance, and the number of queueing operations performed by each algorithm.

5.1 Introduction

Although it is well known that the tracing loop is one of the most significant cost contributors to tracing garbage collection, there has been surprisingly little research on how various tracing loop design decisions can impact performance.

For instance, consider the canonical tracing loop algorithm for mark-sweep garbage collection (Listing 5.1), as described by Jones et al. [2023]. For a detailed discussion of each of the steps in this algorithm, see Section 2.1.1.

Listing 5.1: The canonical mark-sweep tracing loop [Jones et al., 2023]

```
1  for objRef in root_set:
2    objRef.testAndMark()
3    queue.add(objRef)
4
5  while not queue.isEmpty():
6    objRef = queue.remove()
7    gcmap = objRef.getGcMap()
8    for field in gcmap.pointers():
9      childRef = field.load()
10     if childRef != null:
11       if childRef.testAndMark():
12         queue.add(childRef)
```

This tracing algorithm maintains a queue of references to objects which have been marked, but are still waiting to be scanned. When an object is popped from the queue, it gets scanned, and each of the child objects are marked and possibly enqueued for further processing. According to the conventional wisdom, this is the best tracing loop design, because objects are only added to the queue if they have not already been scanned. This limits the number of queueing operations to be proportional to the number of objects in the heap, reducing queue overheads.

However, Garner et al. [2007] conducted an analysis of this tracing loop design using replay tracing, and found that queueing operations were *not* a bottleneck. In fact, queue operations accounted for no more than 11% of the total trace execution time. They proposed an alternative design, called 'edge-ordered enqueueing' (Listing 5.2), which enqueues objects *before* they are marked. This change increases the number of queueing operations (since there is nothing stopping multiple references to the same object being enqueued), however that is counteracted by other positive factors. Namely, the temporal locality of the tracing loop is improved, as both the mark and scan operations occur contemporaneously during the same iteration of the loop.

Listing 5.2: Edge enqueueing tracing loop [Garner et al., 2007]

```
1  for objRef in root_set:
2    objRef.testAndMark()
3    queue.add(objRef)
4
5  while not queue.isEmpty():
6    objRef = queue.pop()
7    if objRef.testAndMark():
8      gcmap = objRef.getGcMap()
9      for field in gcmap.pointers():
10       childRef = field.load()
11       if childRef != null:
12         queue.add(childRef)
```

Since Garner et al.'s study was conducted in the context of the mark-sweep garbage collector, there is one limitation of the edge-ordered design that was not directly addressed. Namely, it is not compatible with copying garbage collection algorithms. Typically, in the tracing loop of a copying collector, the `objRef.testAndMark()` operation is replaced by a `field.forward()` operation. If the object referenced by the field has not previously been marked, the forwarding function is responsible for moving the object and updating the field to point to its new location[1]. However, in the edge-ordered tracing algorithm, if the object needs to be moved (on line 7), it is impossible to store the new object reference back to the slot, as the location of the field was discarded in a previous iteration of the loop.

[1]This is a gross simplification of what is involved in the forwarding operation. Typically, an object can be in one of three states: unforwarded, being forwarded, or forwarded. The state of the object is tested and set atomically to ensure that multiple threads do not attempt to forward the same object. Any thread which observes an object in the 'being forwarded' state waits for it to be moved before proceeding. `True` or `False` is returned by the forwarding function to indicate whether the calling thread was the one which moved the object, and thus is responsible for scanning it.

This limitation of the edge-ordered tracing algorithm naturally leads to two possible design alternatives: enqueueing slots rather than object references, or possibly enqueueing a tuple of both the slot and the object reference. Both of these tracing loop designs would enable support for copying garbage collection, at the expense of either lower temporal locality of slot accesses, or higher queueing overheads. The possible performance tradeoffs of these two designs have not been evaluated.

This short thought exercise exposes a previously-underexplored design aspect of tracing loop algorithms, and suggests the need for a more thorough exploration of the design space.

5.2 Taxonomy of Tracing Loop Designs

In the previous section, I demonstrated that there are two key design decisions which can influence the structure of the tracing loop: when objects are marked, and what type of items are added to the queue. Based on this observation, I propose a novel taxonomy which classifies existing tracing loop designs and uncovers several new tracing loop structures which have not previously been evaluated.

The first axis of my taxonomy describes the *timing* of when objects are marked. To mirror the existing literature, I use the following terminology:

- **Node**-ordered enqueueing describes algorithms which test and mark objects before they are enqueued. Objects which were previously marked are not added to queue to avoid scanning the same object multiple times.

- **Edge**-ordered enqueueing describes algorithms which unconditionally add objects to the queue. Instead, objects are only tested and marked after they are popped from the queue. Although multiple references to the same object may be enqueued, each object is still scanned exactly once.

The second axis of my taxonomy corresponds to the *type* of the items pushed to the queue. There are three possibilities:

- **ObjRef** enqueuing, which involves first loading the slot to obtain a reference to the object, and then enqueueing the object reference.

- **Slot** enqueueing, which simply involves enqueueing the location of the slot itself. After the slot is popped from the queue, it is necessary to load it to determine the object reference.

- **Tuple** enqueueing involves pushing a tuple of both the slot and the object reference to the queue.

This taxonomy describes six possible tracing algorithms: Node-ObjRef, Node-Slot, Node-Tuple, Edge-ObjRef, Edge-Slot, and Edge-Tuple. For completeness, the pseudocode of each tracing algorithm is listed in Figure 5.1.

Listing 5.3: Node-ObjRef

```
while not queue.isEmpty():
  objRef = queue.remove()
  gcmap = objRef.getGcMap()
  for field in gcmap.pointers():
    childRef = field.load()
    if childRef != null:
      if childRef.testAndMark():
        queue.add(childRef)
```

Listing 5.4: Edge-ObjRef

```
while not queue.isEmpty():
  objRef = queue.remove()
  if objRef.testAndMark():
    gcmap = objRef.getGcMap()
    for field in gcmap.pointers():
      childRef = field.load()
      if childRef != null:
        queue.add(childRef)
```

Listing 5.5: Node-Slot

```
while not queue.isEmpty():
  slot = queue.remove()
  objRef = slot.load()
  gcmap = objRef.getGcMap()
  for field in gcmap.pointers():
    childRef = field.load()
    if childRef != null:
      if childRef.testAndMark():
        queue.add(field)
```

Listing 5.6: Edge-Slot

```
while not queue.isEmpty():
  slot = queue.remove()
  objRef = slot.load()
  if objRef != null:
    if objRef.testAndMark():
      gcmap = objRef.getGcMap()
      for field in gcmap.pointers():
        queue.add(field)
```

Listing 5.7: Node-Tuple

```
while !queue.isEmpty():
  slot, objRef = queue.remove()
  gcmap = objRef.getGcMap()
  for field in gcmap.pointers():
    childRef = field.load()
    if childRef != null:
      if childRef.testAndMark():
        queue.add((field, childRef))
```

Listing 5.8: Edge-Tuple

```
while not queue.isEmpty():
  slot, objRef = queue.remove()
  if objRef.testAndMark():
    gcmap = objRef.getGcMap()
    for field in gcmap.pointers():
      childRef = field.load()
      if childRef != null:
        queue.add((field, childRef))
```

Figure 5.1: Pseudocode of the six tracing algorithms described by my taxonomy. Each tracing algorithm has two key design factors: the queueing strategy (edge- or node-ordered enqueueing), and the queue item type (slot, object reference, or a tuple of both). As discussed in Section 5.2.2, the Node-Slot and Node-Tuple algorithms are not necessarily sensible designs. Although they are included here for completeness, they are shown in grey.

5.2.1 Dual-Queue Designs

MMTk's main tracing loop implements an interesting variation on the Edge-Slot design. Instead of having a single loop which performs a load, mark, and scan operation on every item in a queue, MMTk uses a dual-queue and dual-loop algorithm[2], which I call *Edge-Slot-Dual*. Each `ProcessEdgesWork` packet accepts a list of slots as input, known as the *edge queue*. The first loop processes this input list, dereferencing each slot to obtain the contained object reference, and marking each object. Objects which were not previously marked (i.e. objects which require scanning) are added to a secondary *node queue*. The second loop then processes the node queue, scanning each object and enqueueing it's pointer fields into a new `ProcessEdgesWork` packet. Once the new work packet is full, it is added to the relevant work bucket and processed by the next available GC worker. This tracing loop design can be expressed by the pseudocode in Listing 5.9.

Listing 5.9: Edge-Slot-Dual tracing loop

```
1  while not edgeQueue.isEmpty():  # Handled by the work bucket system
2    while not edgeQueue.isEmpty():
3      slot = edgeQueue.pop()
4      objRef = slot.load()
5      if objRef != null:
6        if objRef.testAndMark():
7          nodeQueue.add(obj);
8
9    while not nodeQueue.isEmpty():
10     objRef = nodeQueue.pop()
11     gcmap = objRef.getGcMap()
12     for field in gcmap.pointers():
13       edgeQueue.add(field)
```

This dual-queue style of design is adaptable to several of the other tracing algorithms in my taxonomy. For instance, a Node-ObjRef-Dual algorithm could be created by exchanging the loops on lines 2-7 and 9-13. This makes it a candidate for a third axis of the taxonomy. However, in this thesis, I only focus on single queue designs, because I believe that the memory access patterns generated by a single loop are generally more likely to be cache-friendly. This hypothesis is supported by the performance measurements shown in Section 5.3.1 and Chapter 6. However, I still include the Edge-Slot-Dual algorithm in my performance evaluations as a baseline for comparison. Further exploration of dual-queue designs is left as an area for future research.

[2]This design is originated in historical necessity. Early in the porting phase of each runtime to MMTk, it is necessary to call out to the runtime to scan each object. The most practical way to collect the results of this operation is to have the runtime add the work a separate queue. Once optimized, the scanning code can be moved to the binding, opening the possibility of easily implementing other designs. This dual queue design was established during the original ports of MMTk and has remained in place, apparently on the basis of historical inertia.

Table 5.1: Properties of tracing loop designs

Name	Supports copying GCs	Before enqueueing...	
		Checks if null	Checks if marked
Edge-Slot-Dual	✓	✗	✗
Edge-Slot	✓	✗	✗
Edge-ObjRef	✗	✓	✗
Edge-Tuple	✓	✓	✗
Node-ObjRef	✓	✓	✓
Node-Slot	✓	✓	✓
Node-Tuple	✓	✓	✓

5.2.2 Discussion

The seven algorithms discussed above present fundamentally different tradeoffs with respect to the number of items enqueued, the size of the items enqueued, the spatial and temporal locality of the slot loading, object scanning and marking operations, and more. Several of the differences between the tracing algorithms are summarised in Table 5.1 and discussed below.

The total number of queueing operations performed by each tracing design is dependent on the timing of the mark operation. In node-ordered designs, each object is only enqueued for scanning when it is first marked. Therefore the number of queue operations will be proportional to the number of nodes (objects) in the heap. On the other hand, the Edge-ObjRef and Edge-Tuple designs enqueue all non-null references, regardless of whether they point to an object that has already been marked. Hence, the amount of queue operations is proportional to the number of edges (references) in the heap. However, the Edge-Slot and Edge-Slot-Dual algorithms are even worse, enqueueing null references in addition to the non-null references. This is because it is impossible to determine whether a slot contains a null reference without first loading the slot. The Edge-Slot algorithms delay the slot loading operation until after the slot is popped from the queue, which means that even null-containing slots get enqueued.

As discussed in Section 5.1, the Edge-ObjRef design is not directly adaptable to moving garbage collection. This is because the `field.forward()` operation (which normally replaces the `testAndMark` operation) only occurs after the location of the slot has been discarded. Therefore it is not possible to write the new location of the object back into the slot that originally referenced it. All of the other edge-ordered designs are compatible with copying garbage collection, because the slot is passed through the queue so that it is accessible at the time of the forwarding operation. On the other hand, the node-ordered designs are all implicitly compatible with copying garbage collectors, because the forwarding operation occurs immediately after loading the slot.

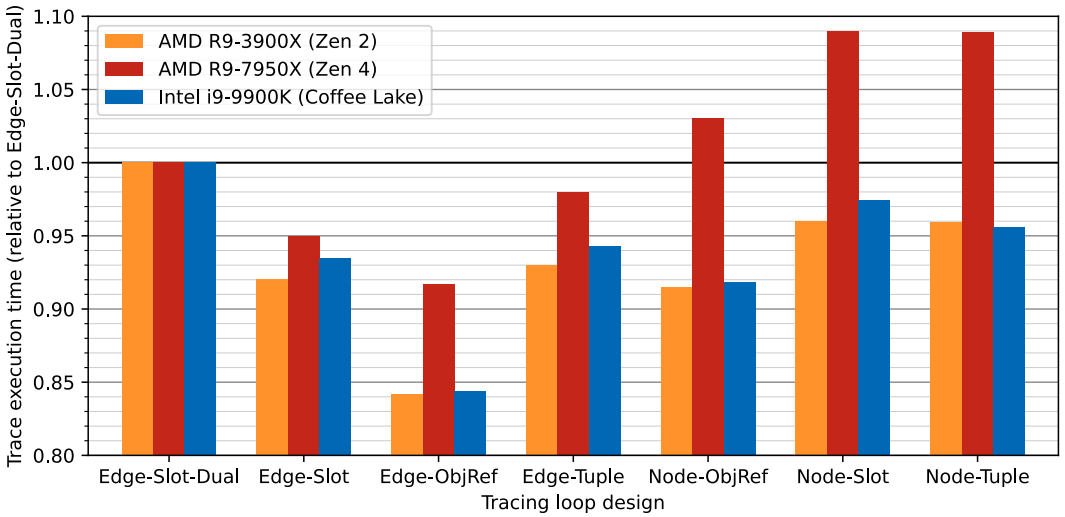


Figure 5.2: Average auxiliary execution time, by tracing algorithm (Zen 2, Zen 4, Coffee Lake)

Although the taxonomy describes Node-Slot and Node-Tuple designs, they are not likely to be sensible algorithms to implement. The Node-Slot design (Listing 5.5) is similar to Node-ObjRef, except the slot is enqueued, instead of the object reference. This means that the slot needs to be loaded twice: once on line 6, and a second time on line 3. The Node-Tuple design (Listing 5.7) is slightly better in that it enqueues both the object reference and the slot, to avoid any need to load the slot a second time. However, after the slot is popped from the queue on line 2, it is never used again. This introduces extra queueing overheads no reason. Furthermore, neither of these designs offer alternative advantages (e.g. better support for copying GCs) that would make them a good choice over Node-ObjRef. Since neither the Node-Slot nor the Node-Tuple design are sensible, in Figure 5.1 they are both greyed out. However, for the sake of completeness, I still implement both designs and evaluate their performance characteristics.

5.3 Evaluation

I implemented all seven of the tracing loop algorithms using the auxiliary tracing framework. All implementations were run through the same validation procedures described in Section 4.3.2 to ensure that they are both correct and performant.

5.3.1 Performance Analysis

Figure 5.2 shows the relative performance of each of the seven tracing loop designs. The graph is normalised to the Edge-Slot-Dual result for each microarchitecture, to enable easy comparison to MMTk's default tracing algorithm.

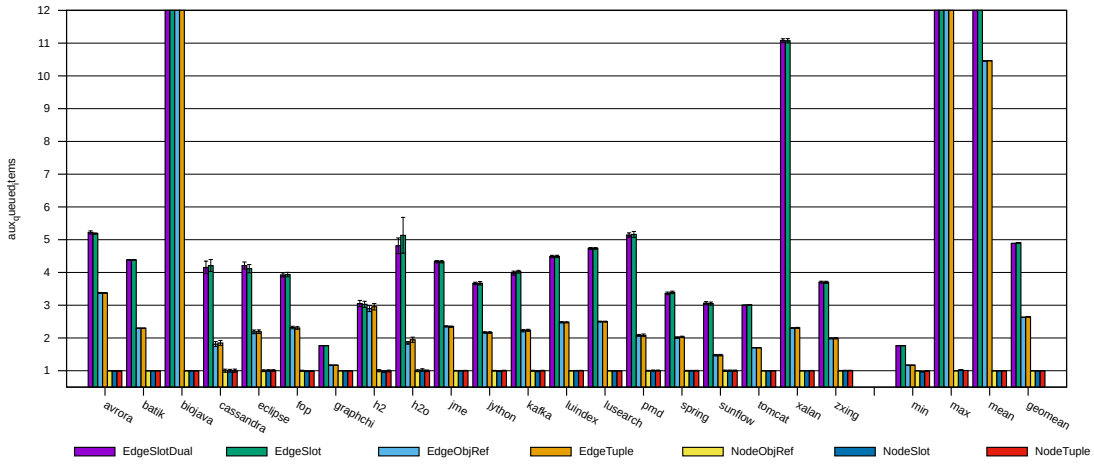


Figure 5.3: Number of objects enqueued by each tracing algorithm (Zen 2)

These results demonstrate that Edge-ObjRef is the fastest tracing algorithm. On Zen 4, it is 8.3% faster than the default Edge-Slot-Dual tracing algorithm; on Zen 2 and Coffee lake, it is almost 16% faster. Generally speaking, the Edge-Slot algorithm performs better than the Edge-Tuple algorithm, which may make it a better candidate for copying garbage collectors. Unsurprisingly, the Node-Tuple and Node-Slot designs perform worse than Node-ObjRef, which makes sense as they are not sensible algorithms. Therefore we exclude them from further analysis.

Node-ObjRef performs very differently depending on the microarchitecture: on Zen 2 and Coffee Lake, it is the second-fastest tracing algorithm; on Zen 4, it is the worst out of the 5 sensible tracing loop designs. This was a surprise, as I expected Zen 4 to behave comparably to the Zen 2 microarchitecture, which is very similar. However, even with fresh builds of each of the tracing loops and a rerun of the evaluation process, the results were reproducible.

5.3.2 Enqueued Objects

I instrumented the `add_child_element()` function of the auxiliary tracing framework to count every item which gets added to the queue. Figure 5.3 shows the number of objects enqueued by each tracing loop.

The three node-enqueueing designs perform a similar number of enqueueing operations, as all non-null objects get enqueued exactly once (since they are only queued when they are first marked). The Edge-ObjRef and Edge-Tuple designs incur a greater number of queueing operations (2.6 times as many as the node-ordered designs) because every object reference is enqueued, regardless of whether the object it points to is already marked. The Edge-Slot and Edge-Slot-Dual designs see an even even larger number of queueing operations (4.9x as many as the node-ordered designs), as null references are additionally enqueued. However, these factors are all very workload dependent, as they can vary significantly based on the connectedness

of the object graph, the frequency of null references, etc. For instance, h2 has little to no null references, so there is little difference between the Edge-Slot and Edge-ObjRef designs; whereas pmd has a significant number of null references. Similarly, in avora, each object is referenced on average 3.4 times, whereas on tomcat it is only referenced 1.7x times, therefore increasing the number of operations that the edge-enqueueing designs have to perform. The biojava benchmark is a massive outlier, with almost 168 times as many references as objects.

One major observation based on these results is that *the queue operations do not define tracing loop performance*. Even in the biojava benchmark, which forces the edge-enqueueing designs to perform a very large number of queueing operations, there is relatively little difference in measured performance between edge- and node-ordered designs. See Appendix B for benchmark-specific comparisons of each of the tracing loops, both with and without prefetching.

5.4 Summary

The object tracing loop is at the heart of every tracing garbage collector and is critical to garbage collection performance. In this chapter, I identified various design decisions that can be made with respect to the structure of the tracing loop, and proposed a taxonomy that can be used to categorise the design space. Using the auxiliary tracing framework, I evaluated several variations on the standard tracing loop structure. I discovered that the Edge-ObjRef tracing loop performs the best by default (without any prefetching) on all tested hardware platforms. However, the comparative speed of each tracing loop can vary significantly depending on the microarchitecture.

Chapter 6

Prefetching

In Chapter 5, I identified five viable tracing loop designs, each with a unique set of tradeoffs. One factor that was left unexplored was the effect of prefetching on each of the designs: what opportunities are there for software cache prefetching, and how effective are they?

In this chapter, I explore the relationship between software cache prefetching and tracing loop algorithms. In Section 6.1, I identify the three main prefetching opportunities (slots, objects and metadata), and identify the structural limitations that each tracing algorithm places on prefetching. Section 6.2 discusses how I physically implemented the prefetching into MMTk without introducing unintended overheads. In Section 6.3, I measure the effects of each type of prefetching (and various prefetching distances) on each of the tracing loops. Following this, in Sections 6.4 and 6.5 I evaluate each of the different types of prefetching instruction.

6.1 Opportunities for Prefetching

In order to scan an object, there are three[1] primary memory accesses which need to occur:

i) Load the slot (to obtain the object reference)

ii) Load the object (to scan the fields)

iii) Test and set the mark bits for that object (to mark it)

In theory, each of these accesses are potential candidates for prefetching. However, in practice, the effectiveness of prefetching in each of these scenarios may be limited by the structure of the tracing loop.

For instance, in node-ordered designs, during the process of object scanning it is necessary mark each of the object's children. This involves loading each of the fields containing child object references, and modifying the metadata for those objects (in

[1] In object-oriented programming languages like Java, a fourth load is also necessary, in order to fetch the type information block so that the pointer fields of the object can be found. Implementing prefetching for this load is left as an area for future work.

order to mark them). However, it is not feasible to prefetch the fields (slots) in this case, because the location of each field is only identified immediately before it is loaded; there is no delay during which the field contents could be prefetched into memory. Similarly, it is not possible to prefetch the metadata without knowing the value of the object reference stored in the field, and since the marking operation occurs immediately after the field load, there is no opportunity for prefetching. Therefore, in node-ordered tracing loop designs, only prefetching of child object references is effective, since the prefetch operation can occur whilst the object reference is on the queue waiting to be scanned.

The Edge-ObjRef and Edge-Tuple tracing algorithms are similar to the node-ordered designs, in that they do not support slot prefetching. During the scanning process, each field is immediately loaded so that the object reference can be added to the queue, so there is no benefit to prefetching the slots. However, the mark operation is delayed until after the object reference (or the tuple) is popped from the queue; therefore it is possible to prefetch the metadata in advance, whilst the object is waiting in the queue.

Although the Edge-Slot and Edge-Slot-Dual algorithms support all three types of prefetching, there are still some restrictions. Namely, the object and metadata prefetch operations both require the object reference. However, the value of the object reference is not known until the slot is loaded. Hence, if the object and metadata need to be prefetched at distance X, it is necessary to first prefetch the slot at some distance $X + Y$, so that the slot is in the cache when it is loaded at time X.

Prefetching is also possible in the root processing packets. For instance, the Node-Slot tracing algorithm expects all roots to be marked and enqueued as object references. Since OpenJDK provides the roots as a list of unmarked slots, a `AuxTraceNodeObjRefRoots` packet is created to load each slot, mark the object reference contained within it, and enqueue the object reference for regular processing. This is a simple queue processing problem, which is easily adapted to use prefetching (by prefetching ahead of the head of the queue). However, the effect of adding prefetching to these packets would be relatively small, since root processing packets only constitute a relatively small proportion of the overall tracing work.

The prefetching opportunities available in each tracing loop design are summarised in Table 6.1.

6.2 Implementation

The modular design of MMTk enables easy implementation of prefetching operations on slots, objects and metadata.

First, it is necessary to implement prefetch operations on the `Address` data type, which is the fundamental representation of a location in memory. I add two methods, `prefetch_load` and `prefetch_store`, which conditionally call the relevant compiler intrinsics, depending on architecture support[2]. Initially, I use the non-temporal

[2]Stable versions of Rust currently only support prefetching on x86/x86_64 architectures which have

Table 6.1: Prefetching opportunities by tracing loop design

Name	Slot	Object	Metadata	Root Slots	Root Metadata
Edge-Slot-Dual	✓	✓	✓†	-	-
Edge-Slot	✓	✓†	✓†	-	-
Edge-ObjRef	-	✓	✓	✓	-
Edge-Tuple	-	✓	✓	✓	-
Node-Slot	-	✓	-	✓	✓†

†Dependent on a prior prefetch and load of the slot

prefetch instruction (`PREFETCHNTA`) for load operations, and the write prefetch instruction (`PREFETCHW`) for store operations. Alternatives to these instructions are evaluated in Section 6.4.

Listing 6.1: Implementation of prefetching on the Address type

```
pub struct Address(usize);
impl Address {
  // ...
  pub fn prefetch_load(self) {
    #[cfg(all(target_arch = "x86_64", target_feature = "sse"))]
    unsafe {
      arch::_mm_prefetch(self.to_ptr(), arch::_MM_HINT_NTA);
    }
  }

  pub fn prefetch_store(self) {
    #[cfg(all(target_arch = "x86_64", target_feature = "sse"))]
    unsafe {
      arch::_mm_prefetch(self.to_ptr(), arch::_MM_HINT_ET0);
    }
  }
  // ...
}
```

Object references and metadata are represented by the `ObjectReference` and `SideMetadataSpec` types respectively. For each of these types, I implement two methods (`prefetch_store` and `prefetch_load`), which convert the object reference to a object/metadata address and call the relevant prefetch operations on the underlying `Address` type. See Listing A.1 and A.2 for the relevant code snippets.

Finally, slots are represented by the `Edge` trait[3]. As before, I add two additional methods to the trait, `prefetch_load()` and `prefetch_store` (Listing A.3). Although

the SSE extension.

[3]Throughout this thesis, I have used the term 'slot' to refer to the MMTk `Edge` datatype, for consistency with the terminology used in my taxonomy. In future, we plan to change the name of the `Edge` datatype to `Slot`, to avoid any confusion with edge-ordered enqueueing. See https://github.com/mmtk/mmtk-core/issues/687 for more details.

the exact implementation of the trait is VM-specific, I provide a default empty implementation, so that VMs which do not support slot prefetching can simply fall back to a no-op. In the `mmtk-openjdk` binding, the `Address` type is used to implement the `Edge` trait, and thus the relevant prefetch methods are already implemented.

Within the auxiliary tracing code itself, prefetching is implemented by overriding the `process_queue` function to look a certain distance ahead in the queue and prefetch the relevant items. For instance, Listing 6.2 shows how object prefetching is implemented into the Node-Objref tracing loop.

Listing 6.2: Implementation of object prefetching in Node-ObjRef

```
impl<VM: VMBinding> AuxiliaryTraceWork for AuxTraceNodeObjRef<VM> {
  // ...
  fn process_queue(&mut self) {
    for i in 0..self.queue.len() {
      #[cfg(feature = "prefetch_objects")]
      {
        let obj_pf_index = i + Self::PF_OBJECT_DIST;
        if obj_pf_index < self.queue.len() {
          self.queue[obj_pf_index].prefetch_load::<VM>();
        }
      }

      self.process_queue_item(self.queue[i])
    }
  }
  // ...
}
```

Conditional compilation is used so that each of the types of prefetching can be individually enabled and disabled via Cargo feature flags. Furthermore, for each type, the prefetch distance is configurable via a build-time environment variable, which sets the relevant constants (`PF_OBJECT_DIST`, `PF_METADATA_DIST`, etc). I decided to use build-time configuration to eliminate any possibility of introducing additional run-time overheads[4]. For instance, this ensures that the prefetch distance is directly encoded as a constant offset in the relevant instruction rather than triggering an additional load (as shown in Listing 6.3).

Listing 6.3: Encoding of prefetch distance in assembly instruction

```
# rax: head of the queue
# rdi: index of item currently being processed
# 0x80: 16 queue items, the distance to prefetch ahead
226729:  48 8b 84 f8 80 00 00  mov    rax,QWORD PTR [rax+rdi*8+0x80]
```

[4]In retrospect, it would have been worthwhile measuring the overhead of implementing run-time configuration of prefetching types and distance. I expect that the costs of checking an environment variable and doing a comparison would be minimal, since those operations both are cache and branch predictor friendly. Furthermore, build-time configuration is very inconvenient: hundreds of OpenJDK builds take up hundreds of gigabytes of disk space, and having to generate a profile-guided optimised build for every configuration takes a considerable amount of time. I would also expect that in a production build, in order for the prefetch distances to be tuned depending on the microarchitecture of the target machine, the prefetch distance would have to be run-time configurable anyway.

```
226730:  00
226731:  0f 18 00                    prefetchnta BYTE PTR [rax]
```

6.3 Prefetching Types & Distance

As discussed in Section 2.3.2, one of the major factors which can influence prefetching effectiveness is the prefetch *timing* or *distance*. Too early, and the prefetch may prematurely evict other data from the cache and cause unnecessary pollution; too late, and the memory hierarchy will not have enough time to bring the relevant data into cache before the load occurs. Therefore, it is necessary to test a wide variety of prefetching distances and choose the configuration that performs best on a given hardware platform.

However, with 5 tracing algorithms, 3 prefetching types that can be turned on or off (each of which can be tested with a wide variety of prefetch distances), and 2 microarchitectures, the state space available for exploration is extremely large. In the interests of time, it was necessary to make some informed decisions in order to reduce the number of configurations that needed to be tested[5]. I ran some initial experiments with a small number of invocations to test my assumptions of prefetching behaviour. Based on these initial experiments, I made the following decisions with regards to my experimental methodology:

- I only test the distances 0, 4, 8, 16, and 32. I found that no tracing algorithm (on either microarchitecture) gained additional meaningful improvements from prefetch distances beyond 32. Although a zero prefetch distance (i.e. prefetching immediately prior to each load) is known to negatively impact performance, I include it in the benchmarks to measure the cost of the prefetch instruction itself.
- When object or metadata prefetching is dependent on loading the relevant slot (in order to obtain the object reference), I only test object/metadata prefetching configurations where slot prefetching is also enabled. After all, there is little sense in enabling object or metadata prefetching if loading the slot will cause a stall anyway.
  - In these cases, the slot prefetch distance may be greater than 32, as the slot prefetch needs to be issued in advance of the object/metadata prefetch.
- When choosing the configurations to test, I begin by measuring the impact of enabling only a single type of prefetching. This type of prefetching is chosen based on my estimates of which would be the most impactful for the given tracing loop design - usually slot or object prefetching. Based on the results from these experiments, I pick the best-performing configuration, and add another type of prefetching, and so on.

[5]The benchmarks for this section alone already required over a thousand hours of machine time.

- In general, I assume that object and metadata prefetch distances should be the same. This is simply to reduce the size of the state space requiring exploration
  - This assumption is not necessarily valid, as the metadata for an object is more likely to be in cache than the object itself (since the metadata for many objects is densely packed into a small space, which increases the 'hotness' of the relevant cache lines). Therefore it may be more appropriate to use shorter metadata prefetching distance than object prefetching distance. This is left as an area for future research.
  - In order to test a wider variety of configurations, I only perform 10 invocations for each benchmark. Although this means that the confidence intervals on my results are wider than with 20 invocations (as shown in Appendix B), it is enough to display overall trends. When comparing the best prefetching configurations in Section 6.3.4, I re-run the benchmarks with 20 invocations to improve the accuracy of the results.
  - Since garbage collection is a single-pass algorithm, I use the `PREFETCHNTA` instruction for prefetching, as recommended by Intel [2023]. Alternative prefetch instructions are evaluated in Sections 6.4 and 6.5.
  - I do not measure the effect of prefetching on the Node-Slot and Node-Tuple tracing algorithms, as they are not sensible designs. The reason for this is discussed in Section 5.2.2.

In the following sections, prefetch configurations are described as follows. The suffix `PFExx` denotes a slot[6] prefetching distance of `xx`. Similarly, the `PFOyy` and `PFMzz` suffixes represent the object and metadata prefetching distances respectively. If a suffix is absent, then that indicates that the corresponding type of prefetching is completely disabled for that configuration.

In each graph, the Zen 4 results are displayed on the left and the Coffee Lake results are displayed on the right. The y-axis is truncated to the range [0.80, 1.05] to make the trends across prefetching distances easier to see. A consistent colouring scheme is used across all graphs to indicate which combinations of prefetching are enabled (`PFE`, `PFE+PFO`, etc).

6.3.1 Edge-Slot-Dual & Edge-Slot

The evaluation results demonstrate that the Edge-Slot-Dual (Figure 6.1) and Edge-Slot (Figure 6.2) tracing loops show similar performance characteristics when prefetching is implemented. Both loops show a high degree of sensitivity to the prefetching distance, particularly on Zen 4, where there is a clear trend between higher prefetch distances and improved tracing performance. Benchmark-specific results are available in Figure B.1 & B.2 (Edge-Slot-Dual), and in Figure B.3 & B.4 (Edge-Slot).

[6]The 'E' in 'PFE' stands for `Edge`, which is the MMTk data type that represents slots. Internally, within my code I use the term edge; in this thesis, I use the term slot, to avoid confusion with edge-ordered enqueueing.

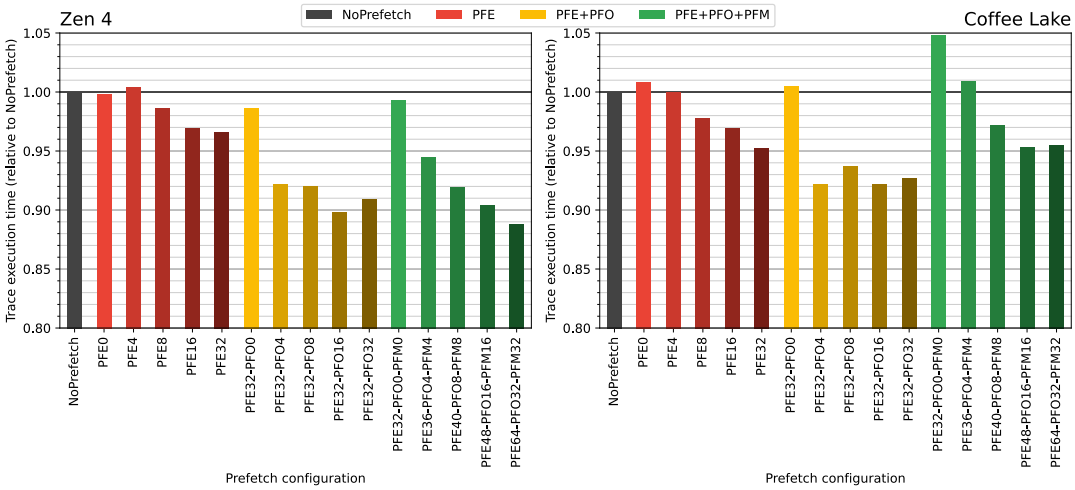


Figure 6.1: Edge-Slot-Dual prefetch distances (Zen 4, Coffee Lake)

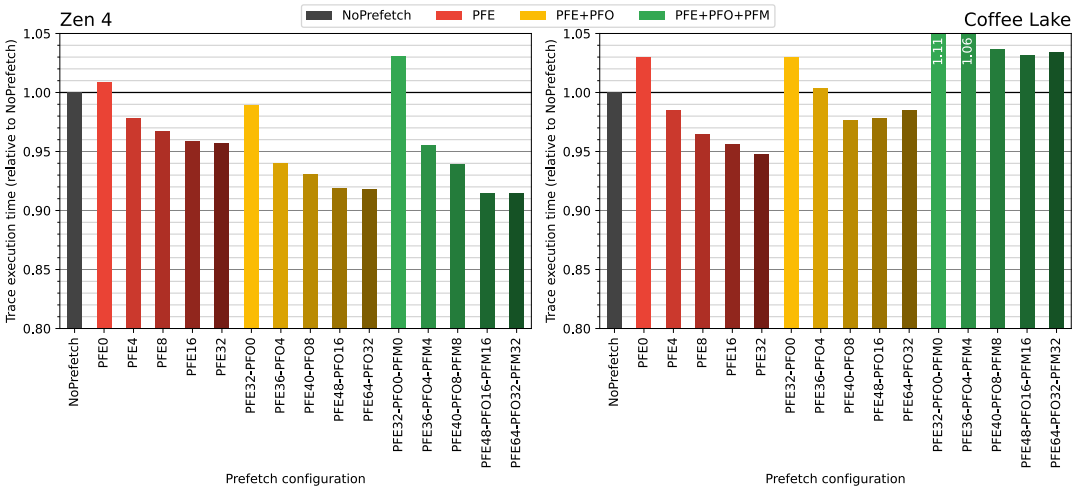


Figure 6.2: Edge-Slot prefetch distances (Zen 4, Coffee Lake)

Across the board, slot prefetching yields a consistent performance improvement. Interestingly, for a given distance configuration, the Edge-Slot algorithm benefits more from slot prefetching than the the Edge-Slot-Dual algorithm. For instance, in PFE32 (the highest performing slot-prefetch configuration), the Edge-Slot-Dual tracing times were reduced by 3.4% on Zen 4 and 4.8% on Coffee Lake, whereas the Edge-Slot tracing times were reduced by 4.3% and 5.2% respectively. This can be explained by the differences between the two algorithms. Each iteration of the Edge-Slot loop performs more work than a single iteration of the relevant loop in the Edge-Slot-Dual algorithm (lines 2-7 of Listing 5.9), as it needs to scan each object in addition to loading the slot and marking the object. If a prefetch is issued at some distance X, the Edge-Slot algorithm will take a greater number of clock cycles to reach that point the queue (as compared to Edge-Slot-Dual), since it takes longer to process each item. Therefore the *absolute* time between the prefetch and the load will be larger for the Edge-Slot algorithm, allowing more time for the relevant data to be brought into cache. Since both Edge-Slot-style designs benefit from earlier slot prefetching, this yields better performance at a given 'distance'.

On Zen 4, the addition of object prefetching further improves tracing performance. In a similar trend to the slot prefetching results, longer prefetch distances are preferable. However, on Coffee Lake, the story is not as clear. Object prefetching yields a performance improvement for Edge-Slot-Dual, whereas on Edge-Slot, it results in a performance degradation with respect to slot prefetching alone. This could possibly be because in the Edge-Slot algorithm, in order to prefetch the object it is first necessary to prefetch the slot. This dependency could cause a stall if the contents of the slot are not in the cache at the time it needs to be loaded in order to obtain the object reference for object prefetching. On the other hand, the Edge-Slot-Dual algorithm loads all of the slots as part of the first half of the tracing loop (lines 2-7 of Listing 5.9), so when the objects are accessed in the second loop (lines 9-13), there is no dependency which could reduce the effectiveness of prefetching. However, this hypothesis does not explain the variances between microarchitectures.

Finally, implementing metadata prefetching in addition to slot and object prefetching has a small positive performance impact on Zen 4, but reduces tracing speed on Coffee Lake. Generally speaking, longer object and metadata prefetching distances yield higher performance.

Performance counter data indicates that these performance improvements are primarily driven by a reduction in back-end stalls where the CPU is blocked on the memory subsystem. Slot prefetching eliminates 24% of L3 cache misses and reduces the number of stalls by 9-10%, whereas the addition of object and/or metadata prefetching eliminates up to 80% of L3 cache misses, resulting in a total stall reduction of 18-20%. Interestingly, as the prefetch distance increases the number of L3 cache misses also increases, but the number of stalls decreases, in line with the total tracing time.

6.3.2 Edge-ObjRef & Edge-Tuple

The Edge-ObjRef and Edge-Tuple tracing algorithms are very similar, with the only difference being that the Edge-Tuple algorithm additionally adds the slot to the queue in order to enable support for copying garbage collectors. Thus, unsurprisingly, they have very similar performance characteristics, as shown in Figure 6.3 and Figure 6.4. On Zen 4, the best performance is achieved with a short slot prefetching distance (4) and a medium object/metadata prefetch distance (16), whereas on Coffee Lake the slot and metadata prefetching should be disabled for improved tracing performance.

Both tracing loop algorithms benefit from object prefetching, with the Edge-Tuple algorithm experiencing speedups of up to 6.1% and 3.8% and the Edge-ObjRef algorithm displaying speedups of up to 7.3% and 3.3%, on Zen 4 and Coffee Lake respectively. However, the effect of metadata prefetching is relatively mixed, with neither tracing loop or microarchitecture demonstrating major gains as a result of enabling it. Slot prefetching is relatively ineffective for these two tracing loop designs, as slots are only be prefetched in the root processing packets, which represent a small portion of the overall tracing time. In the main tracing loop, it is impossible to prefetch the slots because they are immediately loaded during object scanning.

One interesting phenomenon in the benchmark-specific data (Figure B.5 & B.6 for Edge-ObjRef, and Figure B.7 & B.8 for Edge-Tuple) is that biojava exhibits almost no change execution time as a result of adding any form of prefetching. Looking at the performance counter data, it appears that this occurs because the biojava benchmark only experiences modest decrease in L3 cache misses (in the order of 20-30%) when enabling prefetching, as opposed to the substantial reductions observed in other benchmarks (usually up to 60-70% with all three types of prefetching enabled). Therefore the number of memory-bound back-end stalls is higher.

6.3.3 Node-ObjRef

Figure 6.5 displays the effects of adding object, slot and metadata prefetching to the Node-ObjRef tracing algorithm. On both microarchitectures, the best results were obtained when using a short object prefetch distance and a long slot prefetch distance.

Although the Node-ObjRef algorithm is not the best performing tracing loop design by default (Figure 5.2), it does experience some of the most substantial gains from prefetching. Object prefetching alone can reduce tracing time by up to 14.7% on Zen 4 and 11.8% on Coffee Lake. Improvements are consistently seen across all benchmarks on both Zen 4 (Figure B.9) and Coffee Lake (Figure B.10). Performance counter data from Coffee Lake indicates that object prefetching reduces the number of last level cache misses by approximately 50%, therefore decreasing the number of cycles stalled on the memory subsystem by 18%. However, on Zen 4, the root cause of these performance improvements is less clear: the frequency of back-end stalls varied significantly between different benchmarks and prefetch configurations.

Slot and metadata prefetching have relatively little impact on the performance of the Node-ObjRef tracing loop. This is because slot and metadata prefetching are only

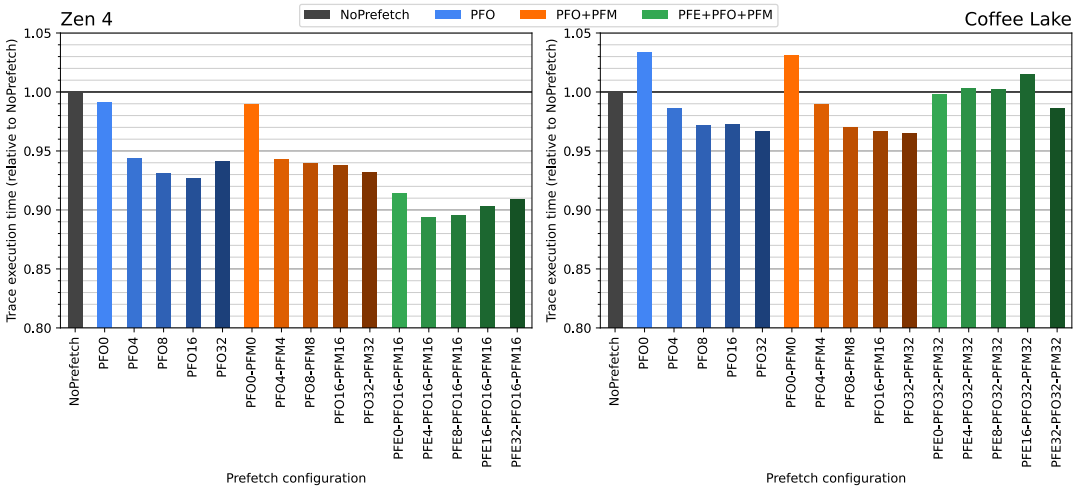


Figure 6.3: Edge-ObjRef prefetch distances (Zen 4, Coffee Lake)

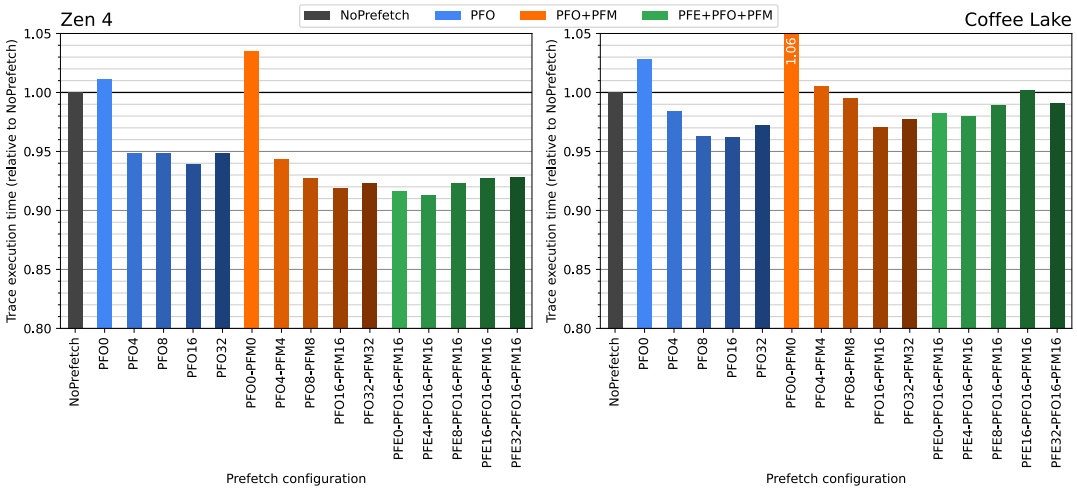


Figure 6.4: Edge-Tuple prefetch distances (Zen 4, Coffee Lake)

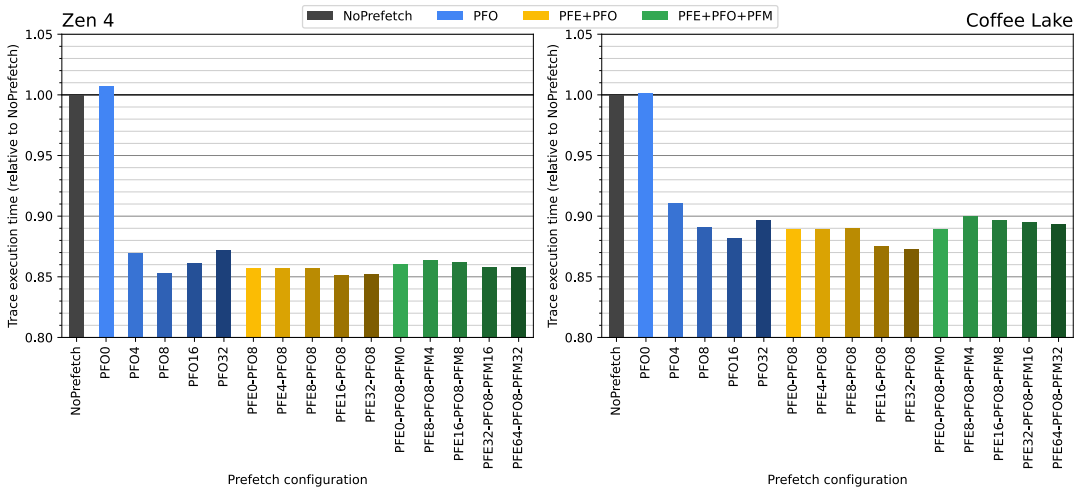


Figure 6.5: Node-ObjRef prefetch distances (Zen 4, Coffee Lake)

applied to the root processing packets, which constitute a relatively small portion of the overall tracing workload.

6.3.4 Comparison

Now that I had a broad understanding of prefetching behaviour across microarchitectures and tracing loop algorithms, I selected the best prefetching configuration for each type of prefetching and tracing loop. Then, I re-ran these configurations through the benchmark suite (this time with 20 invocations), so that all of the results can be compared against each other.

Figure 6.6 presents the results on Zen 4, normalised to Edge-Slot-Dual with no prefetching. The Edge-ObjRef tracing loop had the lowest tracing time overall, achieving a total speedup of 18.1% over Edge-Slot-Dual-NoPrefetch (a model of MMTk's primary tracing algorithm). Although the Node-ObjRef tracing algorithm achieved the most significant speedups as a result of prefetching, it was only the third fastest tracing design overall. If compatibility with copying garbage collectors is desired, then the Edge-Slot tracing algorithm would be a better choice.

The Coffee Lake results, shown in Figure 6.7, paint a remarkably different picture. Firstly, without any prefetching enabled, the Node-ObjRef algorithm was positioned second overall, as opposed to on Zen 4, where it was the slowest tracing loop design. Then, when prefetching was applied, the Node-ObjRef algorithm saw the greatest performance improvements, which allowed it to overtake the Edge-ObjRef algorithm. Therefore, for both copying and non-copying garbage collectors, the Node-ObjRef tracing loop is the best choice on Coffee Lake.

These microarchitectural differences highlight a need for hardware-specific tuning of both the tracing loop design *and* the software prefetching configuration. Neither element can be

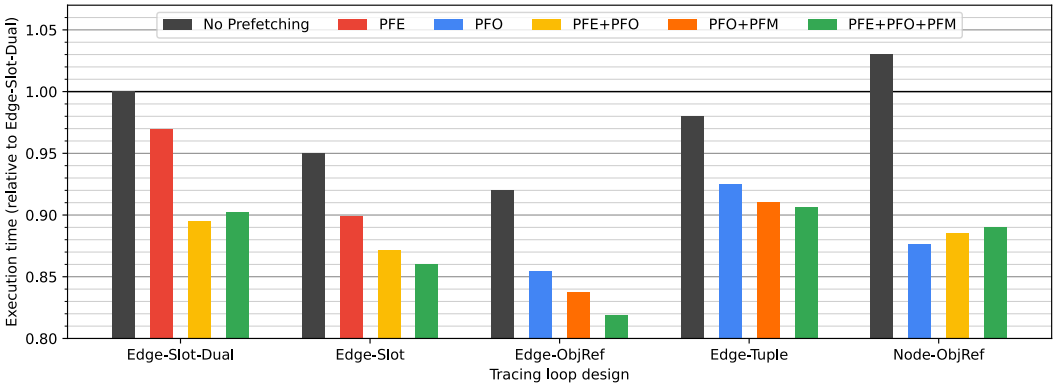


Figure 6.6: Overall comparison of prefetching performance (Zen 4)

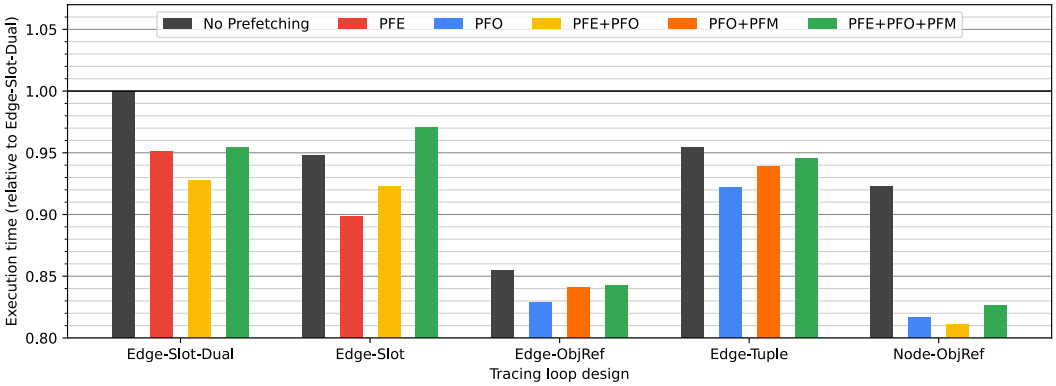


Figure 6.7: Overall comparison of prefetching performance (Coffee Lake)

6.3.5 Discussion

To measure the cost of the prefetch instruction itself, I included tests of a zero prefetch distance in all of my evaluations. A zero prefetch distance effectively means that the prefetch is issued immediately prior to the load. This

Across all of the tracing loop algorithms, metadata prefetching had relatively little impact on tracing performance, particularly when compared to the gains observed by enabling slot and/or object prefetching. One possible explanation is the fact that metadata is more likely to already be in cache than objects or slots. This is because the side metadata is contiguously allocated and tightly packed (each cache line contains the metadata for 64 objects), and is accessed very frequently, which prevents it from being evicted from the cache. Therefore software prefetch instructions are unlikely to have any major effect, and the overhead of the instruction itself may lead to a performance decrease.

In general, Coffee Lake seems to perform better when less types of prefetching are enabled, whereas Zen 4 benefits from having all types of prefetching enabled. One reason for this could be because the Coffee Lake CPU has a significantly smaller cache capacity (8x less L2 cache and 4x less L3 cache) than the Zen 4 machine. Therefore, if too many prefetch operations are issued, useful data may have to be evicted from the cache, inducing unnecessary cache misses and wasting memory bus traffic.

One overarching factor which may be limiting the effectiveness of prefetching is the size of each `AuxiliaryTraceWork` packet. If a work packet is smaller than the prefetch distance, then none of the items in the work packet will be prefetched. Based on my visual observation of the auxiliary tracer (using Perfetto traces like the one shown in Figure 4.2), many work packets appear to be small. However, I have not quantitatively evaluated the distribution of work packet sizes to determine whether this is a major issue. This is one area for potential future research.

6.4 Prefetch Locality

One interesting phenomenon which was observed in the performance counter data for each of the benchmarks in Section 6.3 was that the introduction of prefetching can noticeably increase the number of level 1 data cache misses. For instance, the Edge-Tuple, Edge-Slot and Edge-ObjRef tracing algorithms exhibit an 8%, 10% and 12% increase in L1 cache misses respectively. On the other hand, the Edge-Slot-Dual and Node-ObjRef algorithms exhibit neither an increase nor decrease in cache misses. This phenomenon is relatively stable across all benchmarks (with the exception of `biojava`, which rarely displays any difference, regardless of the tracing algorithm), and does not appear to be dependent on the specific prefetching configuration (distance, types of prefetching enabled, etc).

This data is in stark contrast to the other recorded statistics. Generally speaking, prefetching causes a large reduction in last-level cache misses, memory-bound stalls, and reduces the overall time taken to execute the auxiliary trace. One possible explanation for this is that the prefetching instructions pollute the L1 cache, but do

Table 6.2: Microarchitectural implementations of prefetch instructions

Microarchitecture	Instruction	L1	L2	L3
Zen 4	PREFETCHT0 PREFETCHT1 PREFETCHT2	✓	✓	✓
	PREFETCHNTA	✓	✓*	✗
Coffee Lake	PREFETCHT0	✓	✓	✓
	PREFETCHT1 PREFETCHT2	✗	✓	✓
	PREFETCHNTA	✓	✗	✓†

*Marked for faster replacement; will not fill L3 cache upon eviction [AMD, 2023a]

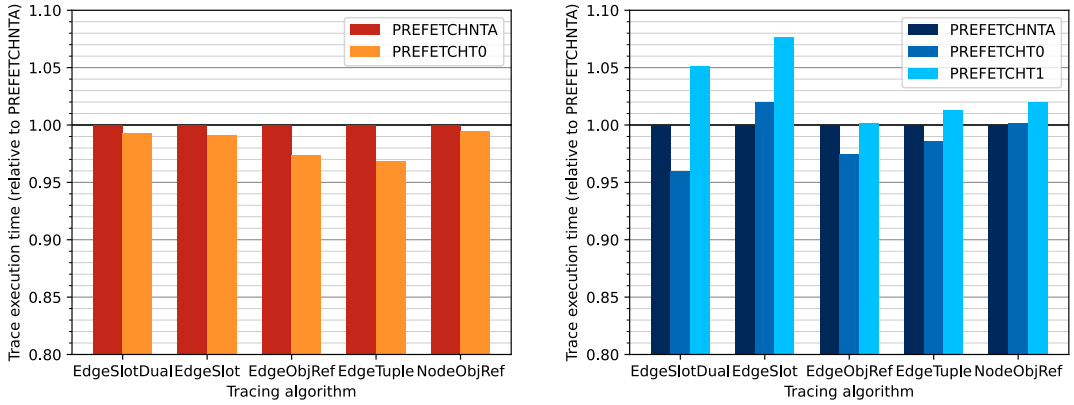
†Marked for faster replacement than a regular cache fill [Intel, 2023]

an excellent job at bringing the relevant data into the second and third level caches, therefore reducing stalls on L2 and L3 cache misses, which are far more costly.

As discussed in Section 2.3.2, the x86 architecture offers several variations on the prefetch instruction which may allow us to reduce the number of L1 cache misses, without sacrificing the benefits of prefetching data into the lower-level cache. All of the benchmarks in Section 6.3 were conducted with the `PREFETCHNTA` instruction, which aims to bring the data into the L1 cache without polluting the lower levels. There are also `PREFETCHT0`, `PREFETCHT1` and `PREFETCHT2` instructions which bring data into the L1, L2 and L3 cache respectively. However, the exact microarchitectural implementation of each of these instructions can vary. For instance, on Zen microarchitectures, AMD ignores the locality hint, so the `PREFETCHT0/T1/T2` instructions behave identically [AMD, 2023a]. On modern Intel Core processors, the `PREFETCHT1/T2` instructions are implemented identically, however the `PREFETCHT0` instruction is distinct. The differences in behaviour between these instructions on different microarchitectures is summarised in Table 6.2.

For each tracing loop, I chose the best performing prefetch configuration from Section 6.3 and created a new build for each of the different types of prefetching instructions. On each microarchitecture, we only test instructions with distinct underlying behaviour. These were evaluated across 20 invocations of the benchmark suite. The results are shown in Figure 6.8, and are normalised to the non-temporal access (NTA) instruction, which was used in prior sections.

On Zen 4, the `PREFETCHT0` instruction yields a small performance improvement across all tracing algorithms, ranging from 0.5% on Node-ObjRef to 3.1% on Edge-Tuple. Although the exact magnitude of these improvements varies from benchmark to benchmark (Figure B.11), most benchmarks exhibit some form of improvement from the `PREFETCHT0` configuration. On Intel, the same instruction yields a performance improvement of up to 4.0% on Edge-Slot-Dual, but decreases performance



(a) `NTA` vs `T0` (Zen 4)

(b) `NTA` vs `T0` vs `T1` (Coffee Lake)

Figure 6.8: Relative performance of the auxiliary trace when the `PREFETCHNTA`, `PREFETCHT0` and `PREFETCHT1` instructions are used. On Zen 4, the `PREFETCHT0` instruction yields a consistent performance improvement, whereas on Coffee Lake it has mixed effects. The `PREFETCHT1` instruction generally harms performance.

by 2.0% on Edge-Slot. The `PREFETCHT1` decreases performance across all tracing algorithms. This differed from my expectations, as I thought that the `PREFETCHT1` instruction would reduce the number of Level 1 cache misses caused by pollution whilst maintaining the benefits of having the data prefetched into the lower-level caches. Although the performance counter data indicates that the L1 cache misses marginally decreased with this configuration, it appears to have had no positive impact on tracing performance.

6.5 Write Prefetching for Metadata

When marking an object, it is necessary to perform an atomic compare and exchange operation on the side metadata. Internally within the CPU, this requires two things to occur: 1. the relevant cache line needs to be brought into the L1 cache, and 2. with respect to the cache coherency protocol, the core needs to transition the relevant cache line to the 'modified' state, and if other cores have the same line in cache, they need to be invalidated. Although the read prefetch instructions (`PREFETCHNTA/T0/T1/T2`) can speed up the first of these operations, the second still needs to be performed at the time of the store operation. The `PREFETCHW` instruction is designed to speed up this use case, by performing both actions in advance of the line being written to.

I generated new builds of MMTk with the metadata store prefetch operations replaced with the `PREFETCHW` instruction. I used the `objdump` command to view the assembly listing for the generated MMTk library to confirm that a `PREFETCHW` instruction was actually generated[7]. Then, I benchmarked the best prefetch configuration

[7]One would think that this is an unnecessary step. However, after a cursory glance at the assembly listing, I noticed that the Rust compiler was emitting a `PREFETCHT0` instruction instead of a `PREFETCHW`

for each tracing loop (according to the results in Section 6.3) using both the NTA and write prefetch instructions, measuring across 20 invocations of the benchmark suite. I only evaluated write metadata prefetching on the Zen 4 microarchitecture, as none of optimal prefetching configurations on Coffee lake use metadata prefetching.

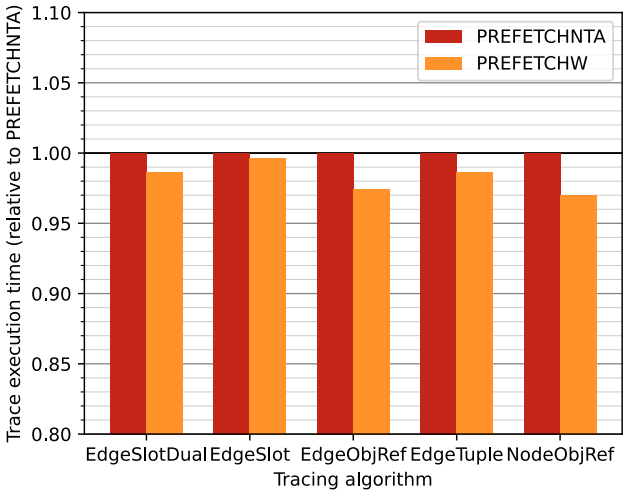


Figure 6.9: Relative performance of the auxiliary trace when using `PREFETCHNTA` and `PREFETCHW` for metadata writes (Zen 4). The write prefetch instruction causes a small but consistent speedup in most benchmarks.

A summary of the results are shown in Figure 6.9. Benchmark-specific results are shown in Figure B.13. In most of the benchmarks tested, write prefetching offers a small but consistent performance improvement, in the range of 0.4% to 3%.

One major outlier is the `biojava` benchmark, which takes 12-14 times as long to complete when write prefetching is enabled on the edge-ordered designs. This unusual behaviour is explainable by the fact that in `biojava`, on average each object is referenced a disproportionately large number of times compared to other benchmarks (as shown by our queueing operation counts in Figure 5.3). In edge-ordered designs, every reference to an object triggers a write prefetch of the metadata in preparation for the upcoming `testAndMark()` operation. This invalidates the metadata cache line in all other cores, causing them to incur a full cache miss when they need to write to the cache line. Furthermore, a single cache line shares the metadata for 64 objects, therefore amplifying the number of invalidations that occur. Obviously, this cache contention puts enormous pressure on the memory subsystem, leading to poor performance.

However, Node-ObjRef does not suffer the same performance penalties exhibited by edge-ordered designs, despite executing the same number of `testAndMark()` operations. This is because metadata prefetching is only implemented in the root tracing

instruction. After some investigation, I discovered that this was occurring because `PREFETCHW` is part of the x86 3DNow! ISA extension, which is not enabled as a feature on the default target CPU used for Rust compilation. As Thompson [1984] said, *"You can't trust code that you did not totally create yourself"*!

packets for node-ordered loops. The main `AuxiliaryTraceWork` packets, which perform the majority of tracing work, do not use metadata prefetching. Therefore when we enable write prefetching, Node-ObjRef does not suffer from the same pathological performance penalties, since it uses significantly less metadata prefetching.

In summary, despite the switch from read to write prefetching being a very small change, we see that the performance impacts can vary wildly depending on the tracing loop design and workload-specific behaviour. Therefore this technique should only be applied on a case-by-case basis.

6.6 Summary

In this chapter, I explored three main prefetching opportunities in the core tracing loop: slots, objects, and metadata. I evaluated a range of prefetch types and distances, and concluded that the Edge-ObjRef tracing algorithm performs the best on Zen 4 and the Node-ObjRef algorithm performs the best on Coffee Lake when prefetching is enabled. Finally, I explored the effects of each of the different prefetching instructions available within MMTk, and found that the `PREFETCHT0` and `PREFETCHW` instructions can provide small additional improvements in tracing speed if applied correctly.

Chapter 7

Conclusion & Future Work

7.1 Concluding Remarks

The tracing loop is one of the greatest contributors to the cost of tracing garbage collection. This thesis demonstrates that choosing the tracing loop structure which enables the most effective software prefetching can dramatically improve the performance of the tracing loop.

In this thesis, I introduced auxiliary tracing, a novel framework for evaluating both new and existing tracing loop algorithms. I also demonstrated how it can be used to fairly measure the impact of different 'heap layouts' on tracing performance.

Furthermore, I proposed a new taxonomy of tracing loop algorithms, which classifies designs based on two factors: the queueing strategy (edge- or node-ordered enqueueing), and the queue item type (slot, object reference, or a tuple of both). Using the auxiliary tracing framework, I implemented seven different tracing loop algorithms, and evaluated the performance of each of the designs. I found that the Edge-ObjRef loop was the highest performing tracing algorithm, yielding an 8.3% and 15.6% performance improvement on the Zen 4 and Coffee Lake architectures respectively.

Finally, I conducted a detailed study of the effects of software cache prefetching on five tracing loop algorithms. I was able to demonstrate consistent improvements in tracing speed across a wide variety of benchmarks. For instance, on Zen 4, the addition of prefetching boosted the performance of the Edge-ObjRef and Node-ObjRef tracing loops by 10.7% and 15.1% respectively. Overall, the Edge-ObjRef tracing algorithm was the best performer. On Coffee Lake, the Node-ObjRef tracing algorithm saw more significant speedups from prefetching, thus allowing it to perform 2.2% better than the Edge-ObjRef loop on average. These microarchitectural differences highlight a need for hardware-specific tuning to take full advantage of software cache prefetching.

In summary, my thesis deepens our understanding of the performance characteristics of tracing loops in garbage collectors, including the relationships between the tracing loop structure and software cache prefetching techniques. It delivers net improvements over MMTk's current default Edge-Slot-Dual design of 18.1% (Edge-ObjRef) and 18.9% (Node-ObjRef) for the Zen 4 and Coffee Lake microarchitectures respectively.

This thesis provides a much-needed re-examination of the performance of a core element of tracing garbage collectors, using modern workloads, a modern runtime, modern hardware, and modern methodology. It reaffirms that prefetching is an important, if overlooked, aspect of garbage collection performance.

7.2 Future Work

Although, as an undergraduate research thesis, the scope of this work was limited by the time frame available, it poses several new avenues for possible research. In this section, I discuss some areas that could be the basis of future work.

7.2.1 Implementation into MMTk's Tracing Loop

This thesis explores many prefetching techniques in the context of the auxiliary tracing framework. A natural next step is to implement the best prefetching configurations for the Edge-Slot-Dual algorithm into the main tracing loop of MMTk. Not only would this improve the performance of MMTk, but it would also enable us to evaluate the effectiveness of my prefetching techniques on a copying collector like Immix.

7.2.2 Copying and Concurrent Garbage Collectors

By design, the current implementation of the auxiliary tracing framework does not support moving garbage collectors, in order to stay side effect free. One possible way that auxiliary tracing could be modified to emulate copying collection algorithms would be to duplicate the entire heap prior to performing the auxiliary trace. This would allow the auxiliary trace to freely move and copy objects without affecting the main tracing algorithm. However, this approach has two major downsides. First, duplicating the heap is extremely expensive (especially when the heap size is large), and hence this would slow down benchmarks significantly. Second, this technique would cause significant cache pollution, which would alter the memory access timings observed by the auxiliary trace. However, an auxiliary tracing framework that supports copying operations may be able to yield previously undiscovered insights, which may make these downsides worthwhile.

Auxiliary tracing could also be applied in the context of concurrent garbage collectors to evaluate prefetching. It is well known that concurrent collectors cause significant cache pollution [Carpen-Amarie et al., 2023]; hence adding prefetching to the tracing loop may be more of a risk than it is in a non-moving, stop-the-world context. It would be very interesting to evaluate the tradeoffs involved in software cache prefetching for concurrent garbage collectors. However, one significant methodological challenge would be determining how to exactly measure the side effects of the prefetches in a concurrent environment.

7.2.3 Further Evaluation on Different (Micro)architectures

The results of Chapter 6 demonstrate that the same prefetching configuration can have very different performance implications on distinct microarchitectures. There are many possible causes for this, including differences in cache capacity, cache replacement policies and existing hardware prefetching mechanisms. One potential area for future research would be to evaluate how a wide cross-section of x86 microarchitectures (both old and new) respond to software prefetching techniques, to determine whether any trends are emerging over time.

One other possible avenue for future work would be to explore the effects of software prefetching on completely different architectures such as ARM. Research on this topic has been difficult in the past, as the ARM ecosystem is still maturing. For instance, OpenJDK did not have adequate support for ARM until relatively recently. Furthermore, high-performance desktop-class ARM hardware has not been readily available for purchase. However, with the emergence of Apple's M1/M2 chips, and the broadening deployment of server-class ARM hardware in the cloud (e.g. from vendors like Ampere), GC performance tuning on ARM has never been more important. Therefore this is an area worth exploring.

It would also be interesting to investigate the applications of software prefetching to tracing garbage collection in RISC-V, an architecture which is the subject of significant excitement in the research community.

7.2.4 Relationships Between Scanning Techniques and Prefetching

Section 6.1 describes three main memory loads during the tracing loop: the slot, the object, and the metadata. However, in many object oriented languages like Java, it is also necessary to load the class information (based on a `klass` pointer in the head of the object) in order to determine the locations of the pointer fields of the object. With the application of prefetching elsewhere in the tracing loop, this operation may be a new bottleneck. This suggests three possible strategies which should be investigated further:

1. Prefetch the class information. This would require a minimal amount of modification to the runtime.

2. Encode the locations of the pointer fields as tag bits in the *klass* pointer. This is known as alignment encoding, and is discussed in Section 4.5.

3. Encode the locations of the pointer fields as tag bits on the *object reference*. This would enable the addresses of the slots within a given object to be determined without having to touch the object itself.

7.2.5 Hardware Prefetchers

Currently, the impact of hardware prefetching on tracing garbage collection algorithms is poorly understood. However, on some Zen 4 server CPUs, it is possible to

individually turn each of the five hardware prefetching units on or off [AMD, 2023b]. Another area for further research would be to measure the impact of enabling and disabling each of these prefetching units to determine whether they make a positive or negative contribution to tracing garbage collection.

Furthermore, modern CPU microarchitectures have been integrating smarter and smarter hardware prefetching units. One hardware prefetching mechanism of particular interest is data-dependent prefetching (discussed in Section 2.3.1). Many tracing algorithms exhibit some form of array-of-pointers access pattern. For instance, the Node-ObjRef tracing loop maintains a queue of object references, which it iterates through and loads each object reference in turn (in order to scan the object). Data-dependent prefetchers are designed to optimise for this access pattern, and thus they could render some forms of software prefetching ineffective or counterproductive. It would be worthwhile investigating how the findings of this thesis change in the face of data-dependent prefetchers.

7.2.6 Prefetching for Other Types of Garbage Collection

Prefetching could also be applied to reference counting garbage collectors like LXR [Zhao et al., 2022]. In particular, reference increment and decrement processing could benefit from software prefetching. Additionally, the prefetch techniques that are applicable to tracing algorithms could be applied to the SATB trace, which is used by LXR to collect cycles and objects with stuck reference counts. The work done by Paz and Petrank [2007] could be a good starting point for this.

7.2.7 More Tracing Loop Designs

In this work, we a single dual-queue design (Edge-Slot-Dual) as a baseline comparison for MMTk's core tracing loop. However, no other dual-queue designs were tested because I made a prediction that dual-queue designs may have lower performance than single-queue designs. Although my evaulation demonstrated this to be true for the Edge-Slot-Dual tracing loop, this may be not be the case for other dual-queue designs. It may be desirable to implement and evaluate several other dual-queue designs.

Furthermore, I imagine there are also other variations on the tracing loop which I have not envisaged, that could be implemented in the auxiliary tracing framework and evaluated.

Appendix A

Prefetching Implementation

Listing A.1: Implementation of prefetching on the ObjectReference type

```
pub struct ObjectReference(usize);
impl ObjectReference {
  // ...
  pub fn prefetch_load<VM: VMBinding>(self) {
    self.to_address::<VM>().prefetch_load();
  }

  pub fn prefetch_store<VM: VMBinding>(self) {
    self.to_address::<VM>().prefetch_store();
  }
}
```

Listing A.2: Implementation of prefetching for metadata

```
pub trait AuxiliaryTraceWork {
  // ...
  fn prefetch_load_metadata(o: ObjectReference) {
    MARK_BITS.prefetch_load(o.to_raw_address());
  }

  fn prefetch_store_metadata(o: ObjectReference) {
    MARK_BITS.prefetch_store(o.to_raw_address());
  }
  // ...
}

pub struct SideMetadataSpec {...}
impl SideMetadataSpec {
  // ...
  pub fn prefetch_load(&self, data_addr: Address) {
    let meta_addr = address_to_meta_address(self, data_addr);
    meta_addr.prefetch_load();
  }

  pub fn prefetch_store(&self, data_addr: Address) {
    let meta_addr = address_to_meta_address(self, data_addr);
    meta_addr.prefetch_store();
  }
  // ...
}
```

Listing A.3: Implementation of prefetching on the Edge trait

```
pub trait Edge: Copy + Send + Debug + PartialEq + Eq + Hash {
  fn load(&self) -> ObjectReference;
  fn store(&self, object: ObjectReference);

  fn prefetch_load(&self) {
    // no-op
  }

  fn prefetch_store(&self) {
    // no-op
  }
}
```

Appendix B

Benchmark-Specific Results

In Chapter 4, I presented a variety of graphs which summarise the effect of various prefetching algorithms on tracing loop performance. However, in order to present the data in a way that aids visual analysis, I had to discard benchmark-specific data. This appendix includes expanded versions of each the graphs in Chapter 6, to illustrate the results of each of the 18 benchmarks used for evaluation in this thesis. These can be used to determine whether the overall summary statistics are representative of individual benchmark results, or simply a product of noise.

For readability, each graph is set on a standalone page.

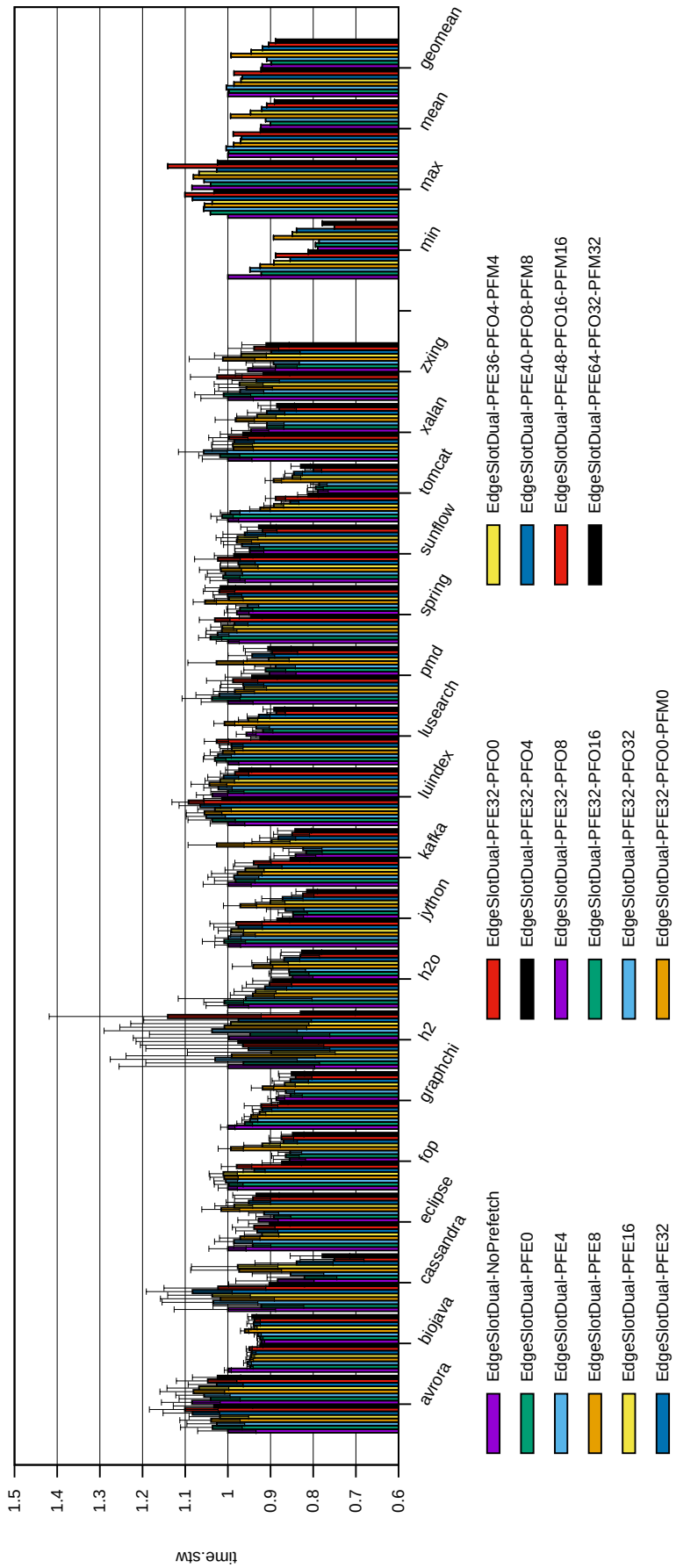


Figure B.1: Edge-Slot-Dual prefetch distances (Zen 4) - all benchmarks

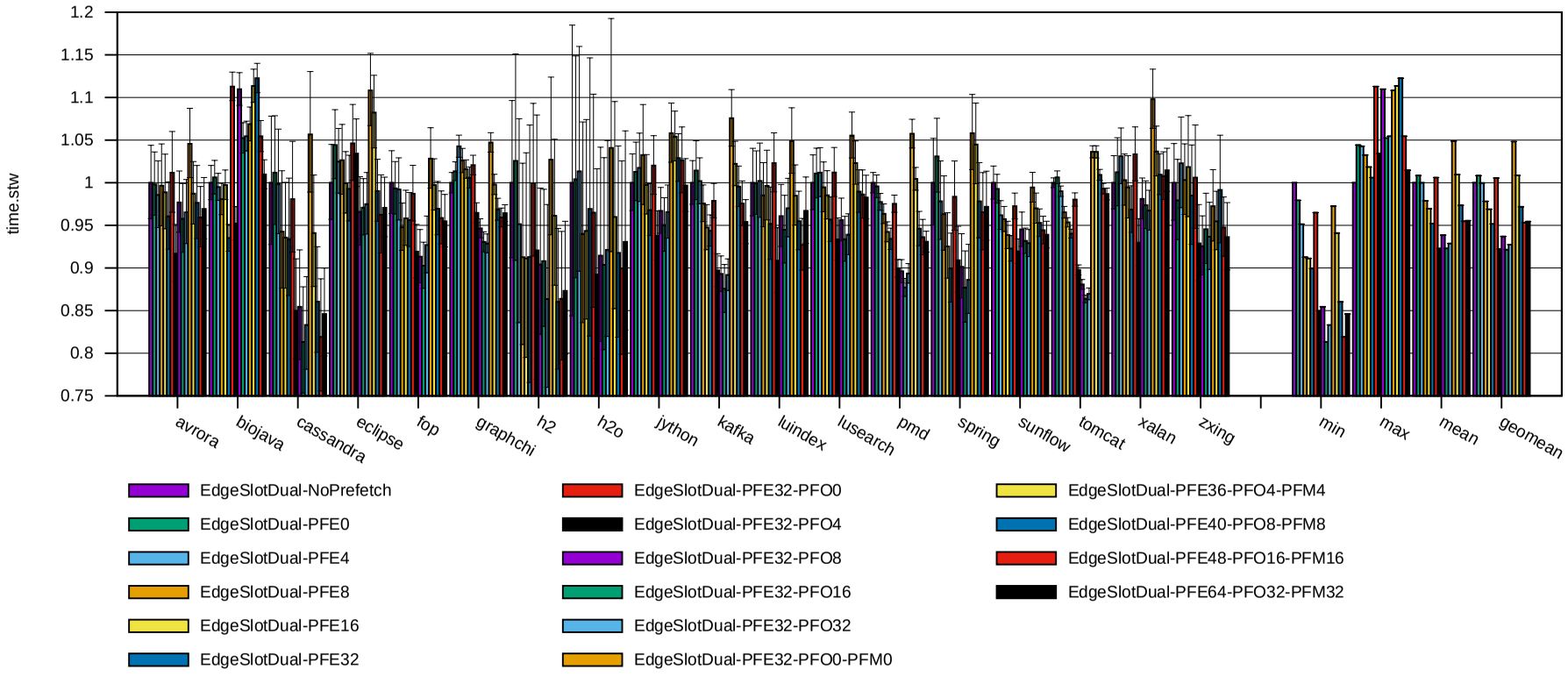


Figure B.2: Edge-Slot-Dual prefetch distances (Coffee Lake) - all benchmarks

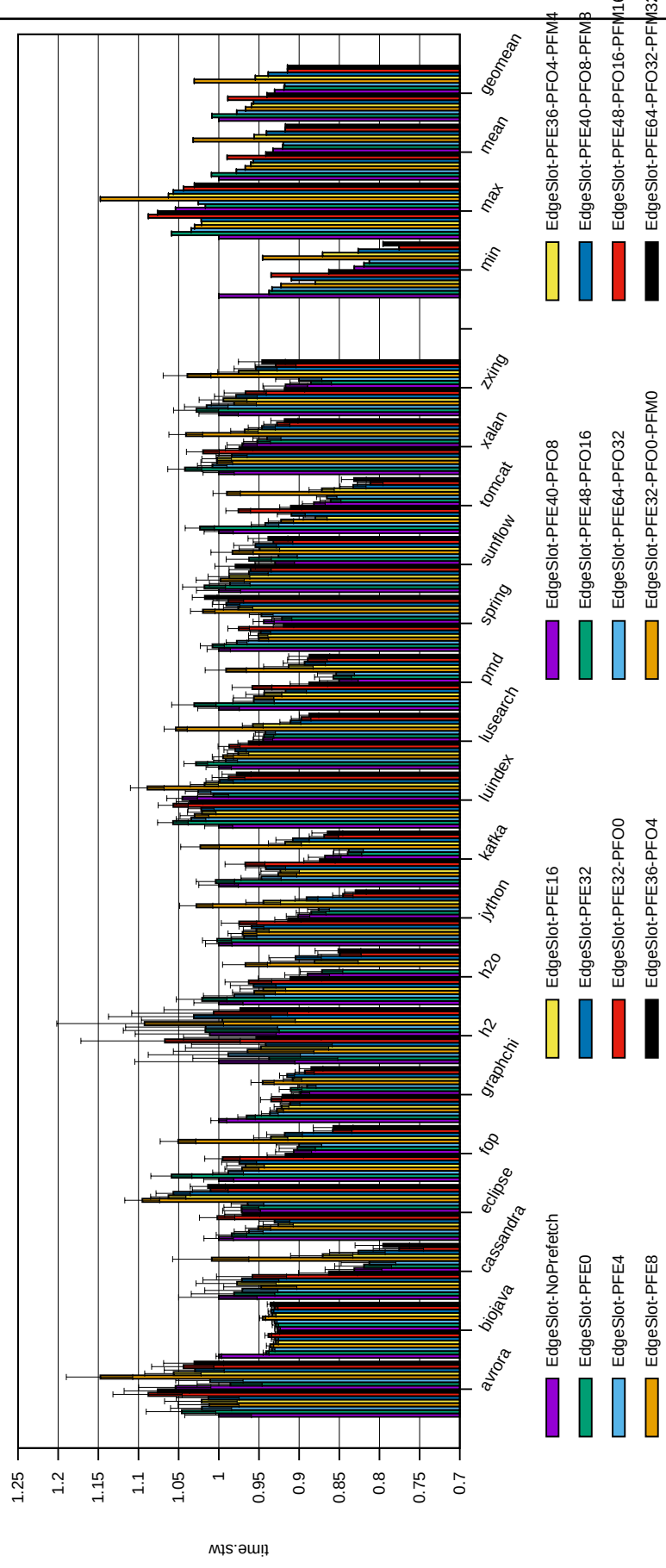


Figure B.3: Edge-Slot prefetch distances (Zen 4) - all benchmarks

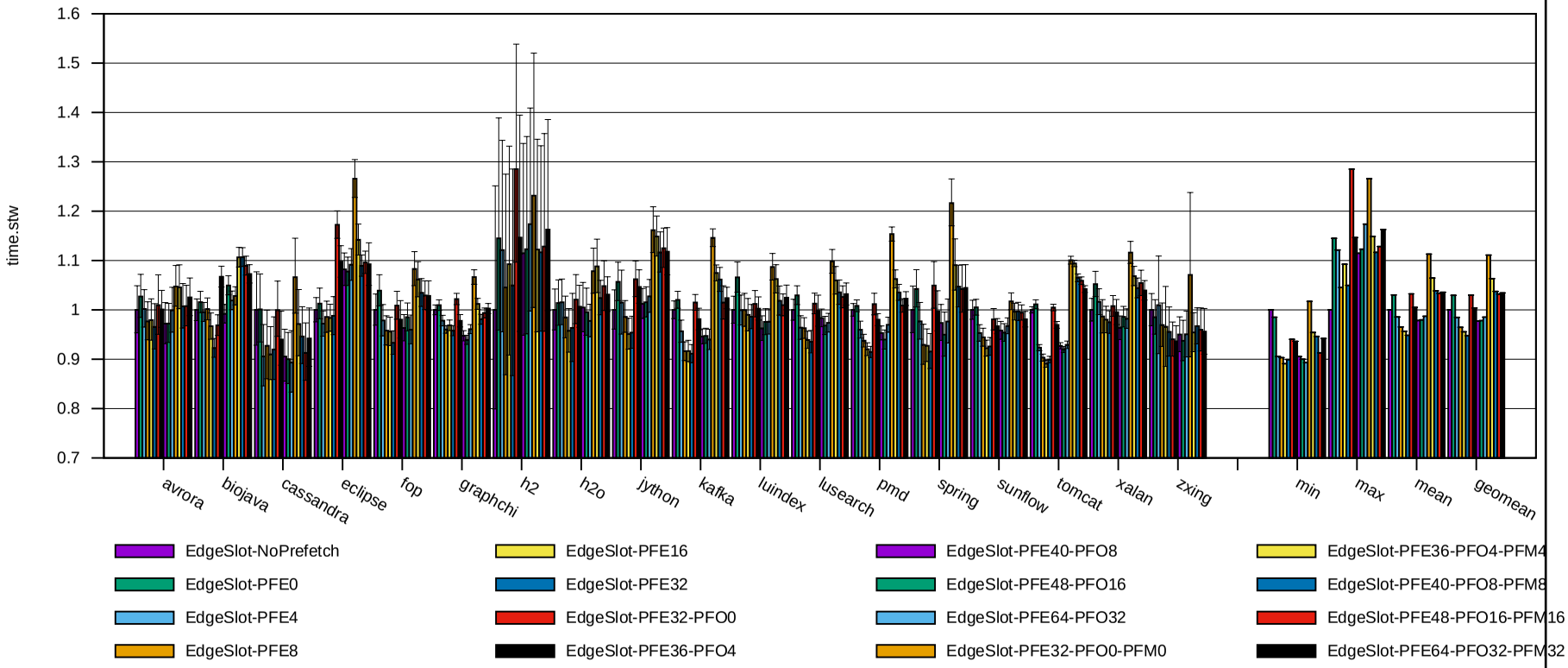


Figure B.4: Edge-Slot prefetch distances (Coffee Lake) - all benchmarks

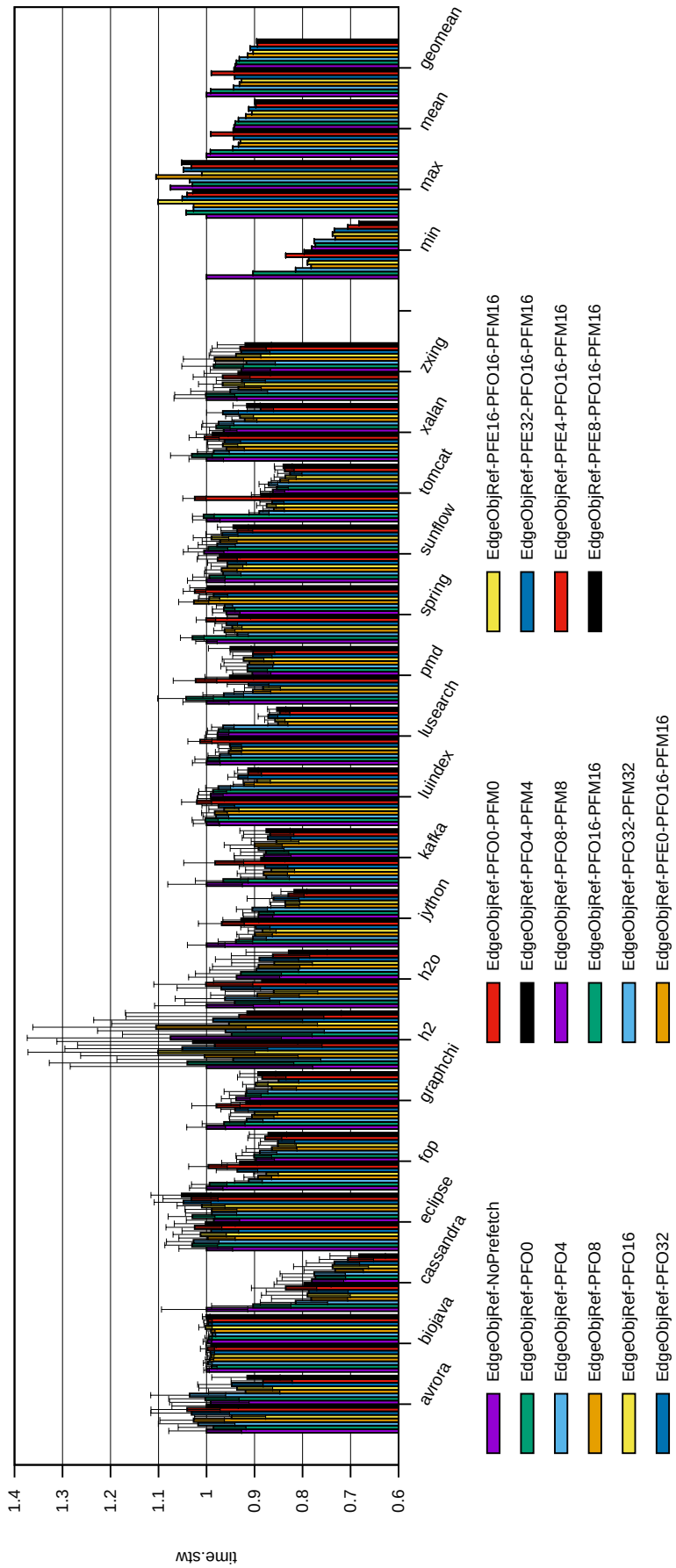


Figure B.5: Edge-ObjRef prefetch distances (Zen 4) - all benchmarks

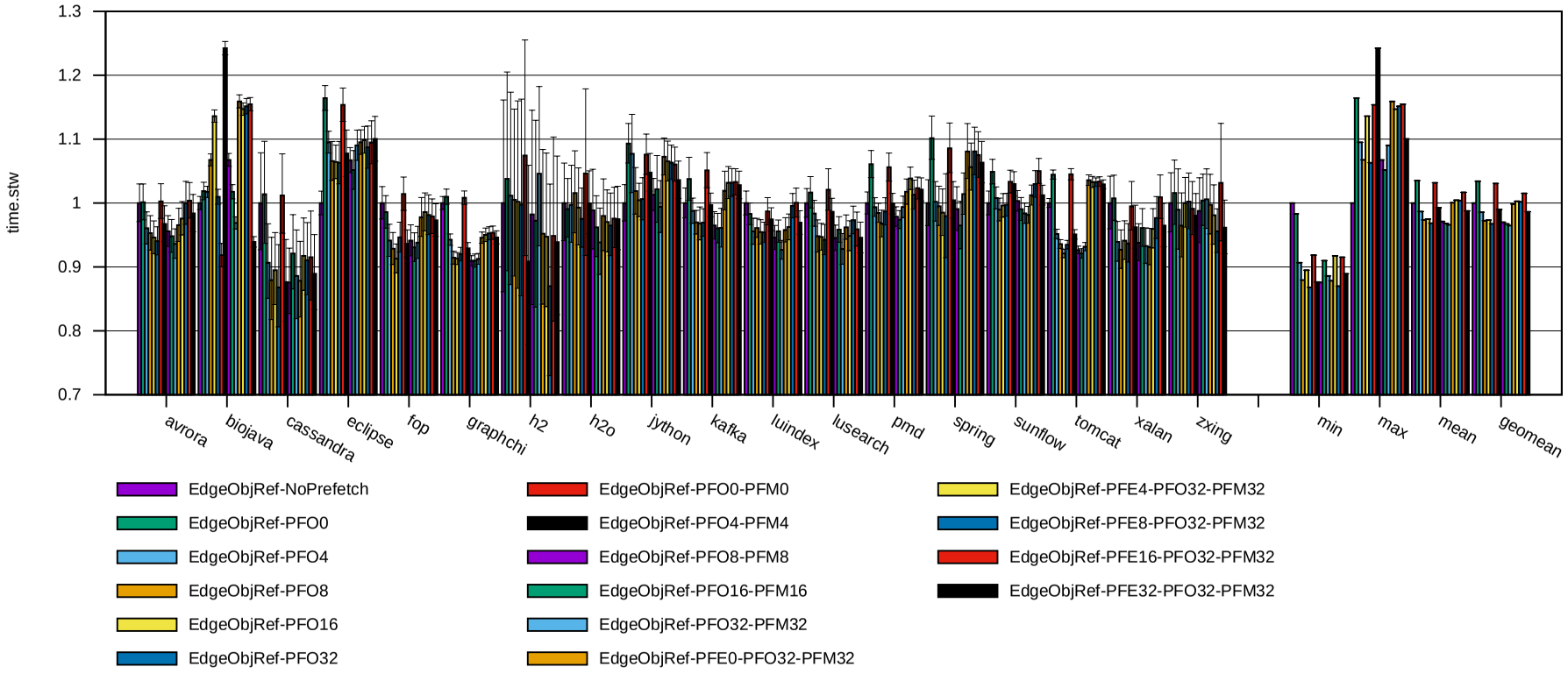


Figure B.6: Edge-ObjRef prefetch distances (Coffee Lake) - all benchmarks

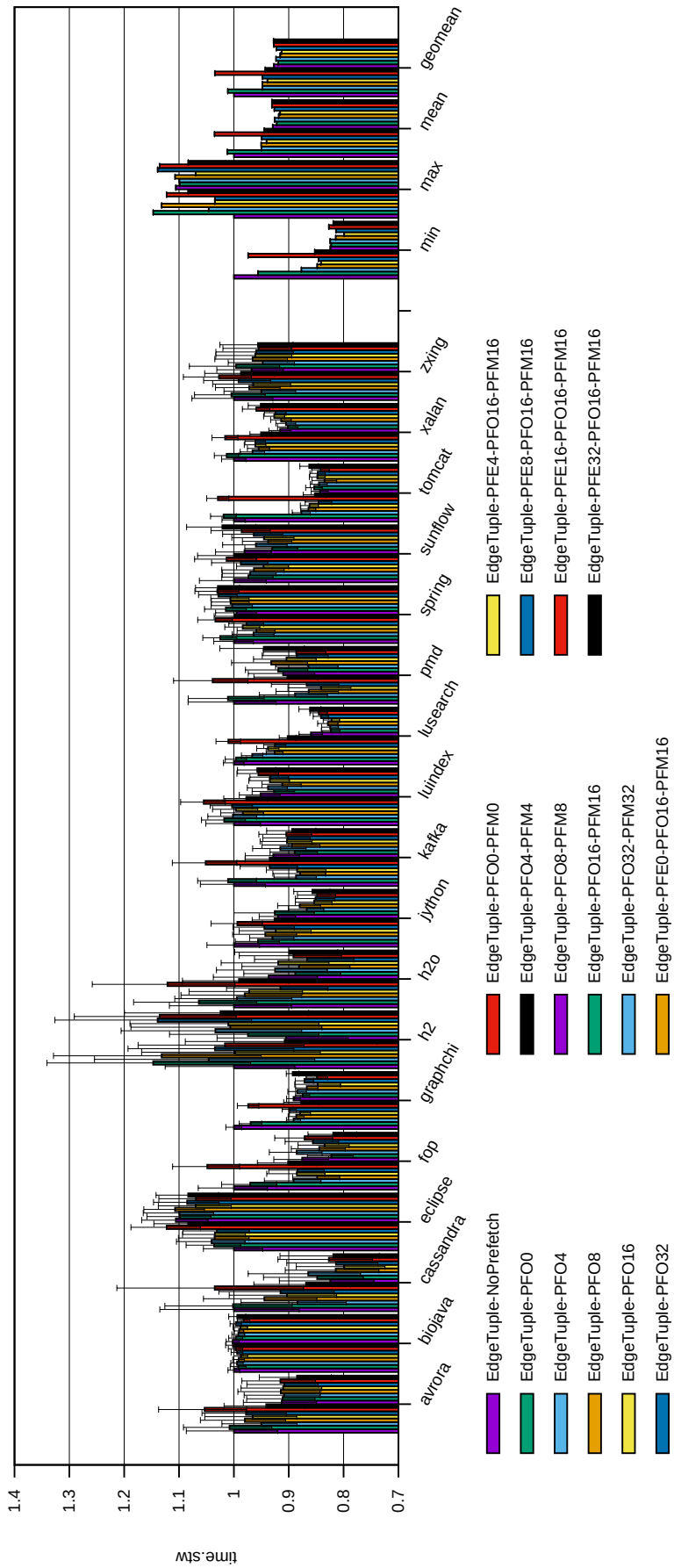


Figure B.7: Edge-Tuple prefetch distances (Zen 4) - all benchmarks

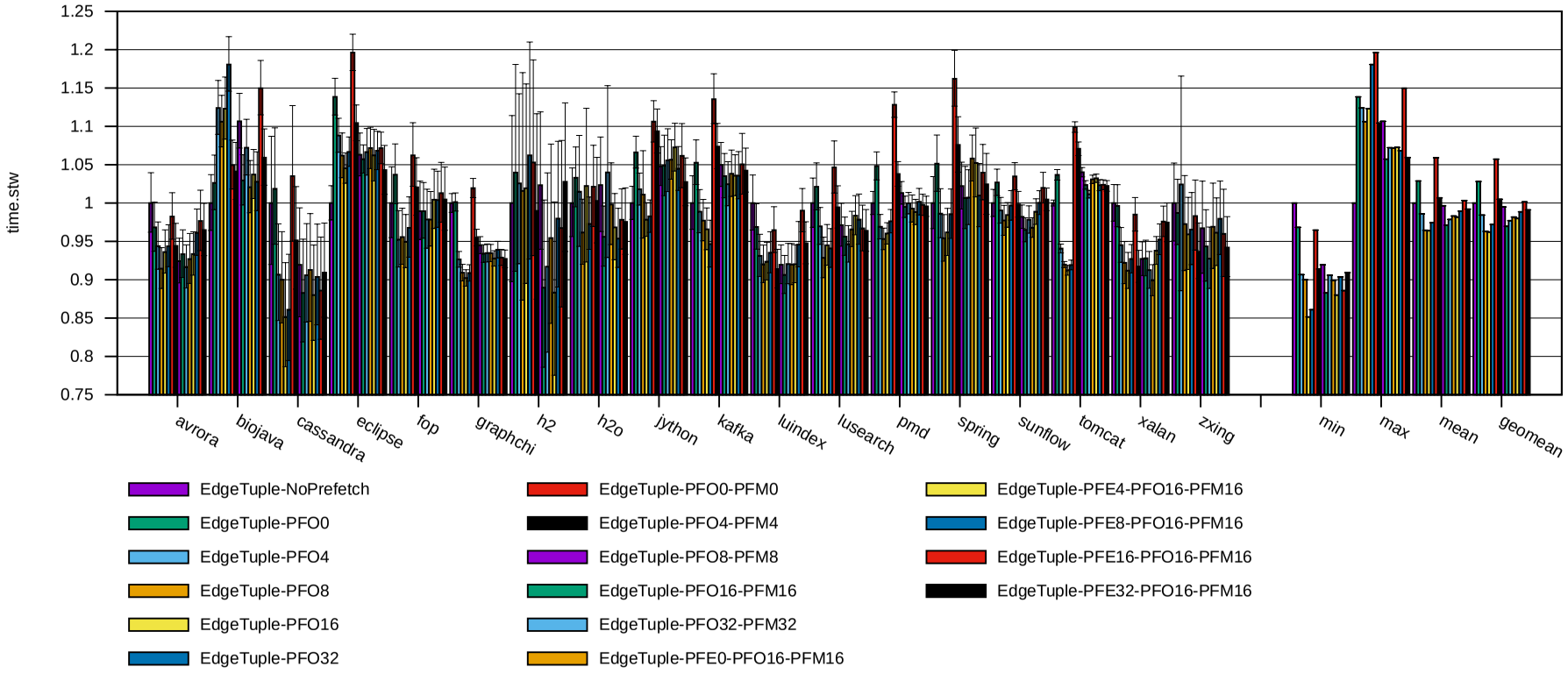


Figure B.8: Edge-Tuple prefetch distances (Coffee Lake) - all benchmarks

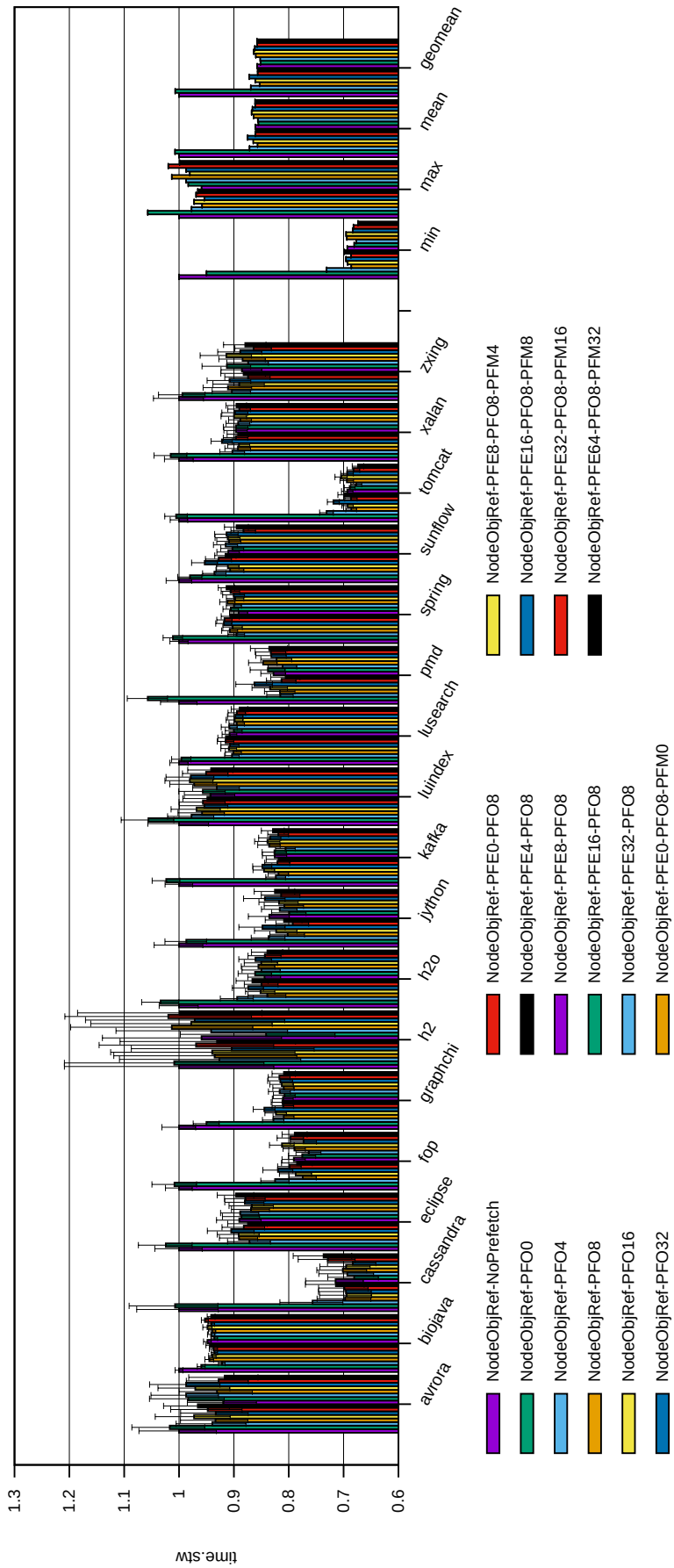


Figure B.9: Node-ObjRef prefetch distances (Zen 4) - all benchmarks

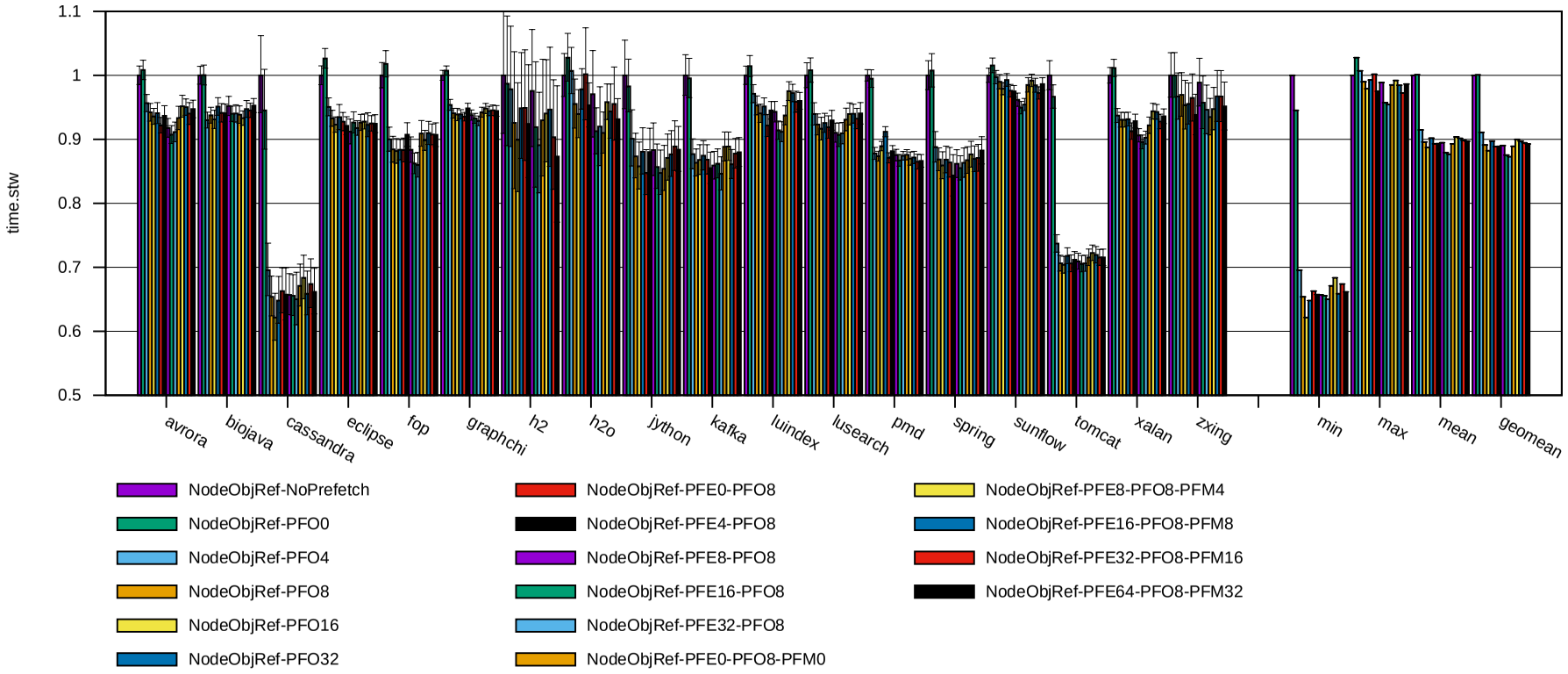


Figure B.10: Node-ObjRef prefetch distances (Coffee Lake) - all benchmarks

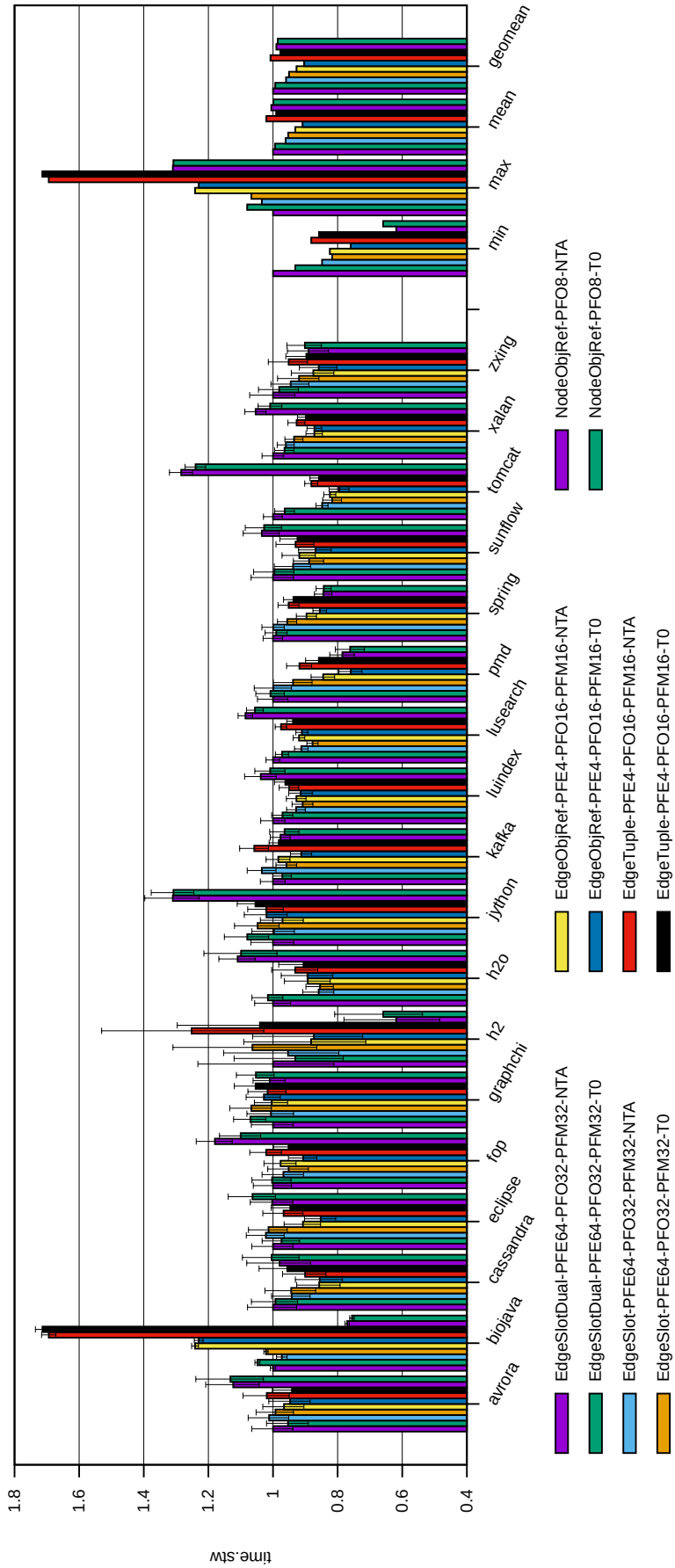


Figure B.11: NTA vs T0 prefetch instructions (Zen 4) - all benchmarks

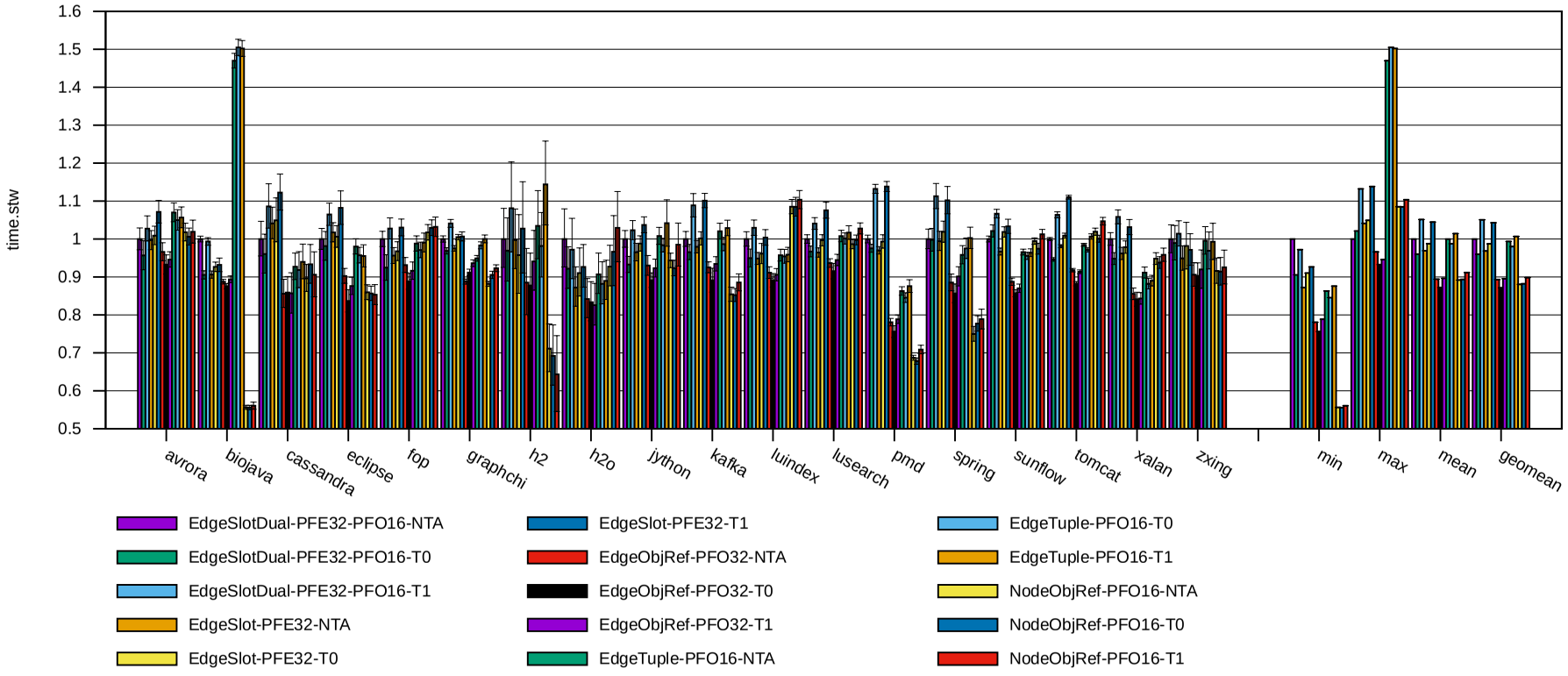


Figure B.12: NTA vs T0 vs T1 prefetch instructions (Coffee Lake) - all benchmarks

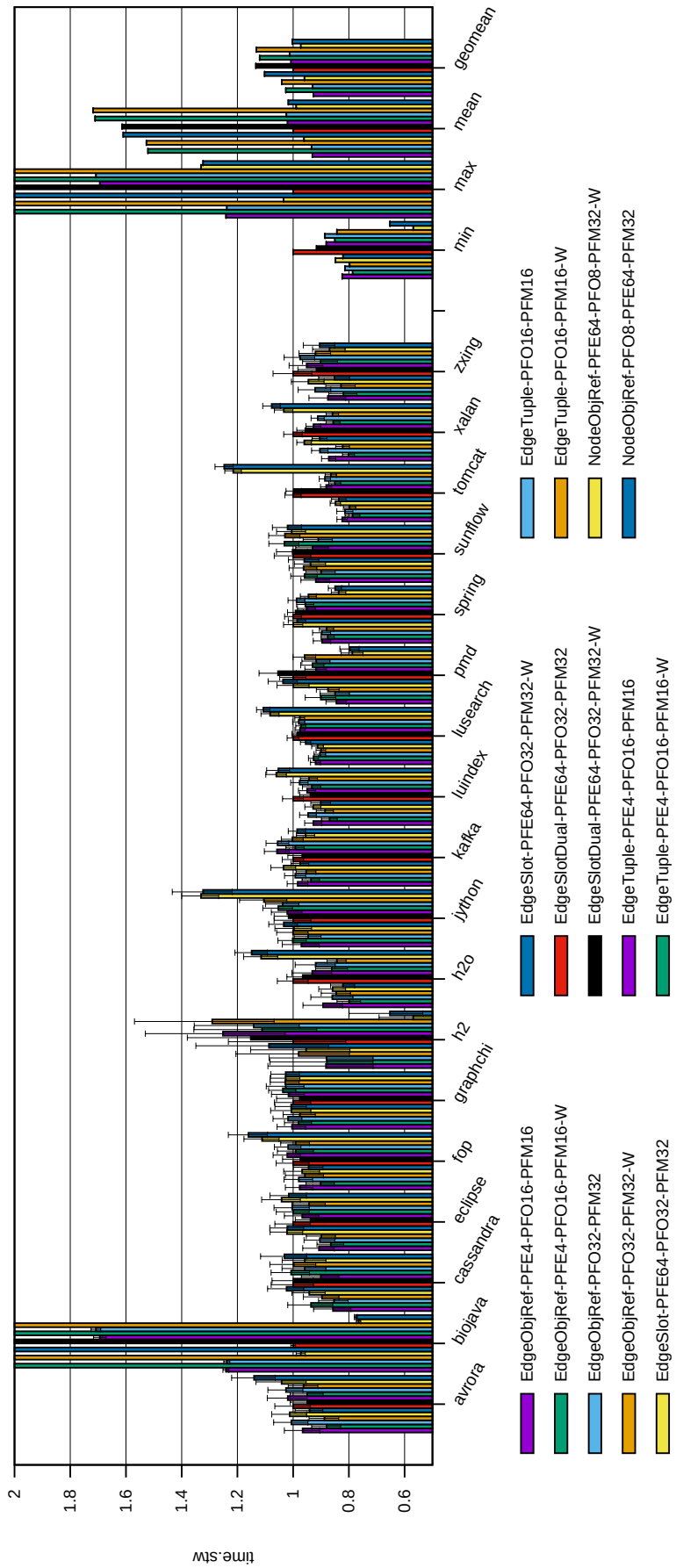


Figure B.13: Metadata write prefetching (Zen 4) - all benchmarks

Bibliography

Alcorn, P., 2023. AMD Details EPYC Bergamo CPUs With 128 Zen 4C Cores, Available Now. https://www.tomshardware.com/news/amd-details-epyc-bergamo-cpus-with-128-zen-4c-cores. (cited on page 17)

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 Systems Journal*, 39, 1 (Jan. 2000), 211–238. doi:10.1147/sj.391.0211. https://doi.org/10.1147/sj.391.0211. (cited on pages 8 and 14)

AMD, 2023a. *AMD64 Architecture Programmer's Manual Volumes 1-5*. Advanced Micro Devices, Inc. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/40332.pdf. (cited on page 56)

AMD, 2023b. *Processor Programming Reference for AMD Family 19h, Model 61h, Revision B1 Processors*. Advanced Micro Devices, Inc. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/56713-B1_3_05.zip. (cited on page 64)

Austin, T.; Larson, E.; and Ernst, D., 2002. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35, 2 (Feb. 2002), 59–67. doi:10.1109/2.982917. https://doi.org/10.1109/2.982917. (cited on page 14)

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*, SIGMETRICS '04/Performance '04, 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 8 and 29)

Blackburn, S. M.; Cheng, P.; and McKinley, K. S., 2004b. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, 137–146. IEEE Computer Society, USA. (cited on page 8)

Blackburn, S. M.; Garner, R.; Hoffmann, C.; Khang, A. M.; McKinley, K. S.; Bentzur, R.; Diwan, A.; Feinberg, D.; Frampton, D.; Guyer, S. Z.; Hirzel, M.; Hosking, A.; Jump, M.; Lee, H.; Moss, J. E. B.; Phansalkar, A.; Stefanović, D.; VanDrunen, T.; von Dincklage, D.; and Wiedermann, B., 2006.

The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, OOPSLA '06, 169–190. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1167473.1167488. https://dl.acm.org/doi/10.1145/1167473.1167488. (cited on pages 17, 18, and 25)

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, 22–32. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1375581.1375586. https://dl.acm.org/doi/10.1145/1375581.1375586. (cited on pages 26 and 28)

Boehm, H.-J., 2000. Reducing garbage collector cache misses. In *Proceedings of the 2nd International Symposium on Memory Management*, ISMM '00, 59–64. Association for Computing Machinery, New York, NY, USA. doi:10.1145/362422.362438. https://dl.acm.org/doi/10.1145/362422.362438. (cited on pages 1 and 13)

Boehm, H.-J. and Weiser, M., 1988. Garbage collection in an uncooperative environment. *Software—Practice & Experience*, 18, 9 (Sep. 1988), 807–820. doi:10.1002/spe.4380180902. https://doi.org/10.1002/spe.4380180902. (cited on page 13)

Brus, T. H.; van Eekelen, C. J. D.; van Leer, M. O.; and Plasmeijer, M. J., 1987. CLEAN: A language for functional graph rewriting. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, FPCA '87, 364–384. Springer-Verlag, Berlin, Heidelberg. (cited on page 14)

Cahoon, B. D., 2002. *Effective Compile-Time Analysis for Data Prefetching in Java*. Ph.D. thesis, University of Massachusetts Amherst. (cited on pages 15 and 16)

Cai, Z., 2023. Shape branch of caizixian/mmtk-core. https://github.com/caizixian/mmtk-core/tree/shape. (cited on page 31)

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, 71–84. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3591195.3595269. https://dl.acm.org/doi/10.1145/3591195.3595269. (cited on page 62)

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, 199–210. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1024393.1024417. https://dl.acm.org/doi/10.1145/1024393.1024417. (cited on pages 1, 13, 14, and 15)

Collins, G. E., 1960. A method for overlapping and erasure of lists. *Communications of the ACM*, 3, 12 (Dec. 1960), 655–657. doi:10.1145/367487.367501. https://dl.acm.org/doi/10.1145/367487.367501. (cited on page 6)

Dean, J.; DeFouw, G.; Grove, D.; Litvinov, V.; and Chambers, C., 1996. Vortex: An optimizing compiler for object-oriented languages. In *Proceedings of the 11th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '96, 83–100. Association for Computing Machinery, New York, NY, USA. doi:10.1145/236337.236344. https://dl.acm.org/doi/10.1145/236337.236344. (cited on page 15)

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, 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 1)

Dijkstra, E. W.; Lamport, L.; Martin, A. J.; Scholten, C. S.; and Steffens, E. F. M., 1978. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21, 11 (Nov. 1978), 966–975. doi:10.1145/359642.359655. https://dl.acm.org/doi/10.1145/359642.359655. (cited on page 13)

Falsafi, B. and Wenisch, T. F., 2014. *A Primer on Hardware Prefetching*. Morgan & Claypool Publishers. ISBN 978-1-60845-952-0. (cited on page 9)

Flood, C. H.; Kennke, R.; Dinn, A.; Haley, A.; and Westrelin, R., 2016. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *Proceedings of the 13th International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '16, 1–9. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2972206.2972210. https://doi.org/10.1145/2972206.2972210. (cited on page 1)

Garner, R., 2011. *The Design and Construction of High Performance Garbage Collectors*. Ph.D. thesis, Research School of Computer Science, ANU College of Engineering & Computer Science. (cited on pages 1 and 7)

Garner, R.; Blackburn, S. M.; and Frampton, D., 2007. Effective prefetch for mark-sweep garbage collection. In *Proceedings of the 6th International Symposium on Memory Management*, ISMM '07, 43–54. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1296907.1296915. https://dl.acm.org/doi/10.1145/1296907.1296915. (cited on pages 1, 14, 15, and 34)

Garner, R. J.; Blackburn, S. M.; and Frampton, D., 2011. A comprehensive evaluation of object scanning techniques. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, 33–42. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1993478.1993484. https://dl.acm.org/doi/10.1145/1993478.1993484. (cited on pages 7 and 31)

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, 85–99. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3617651.3622986. https://dl.acm.org/doi/10.1145/3617651.3622986. (cited on pages 8, 17, 25, and 28)

Huang, X.; Blackburn, S. M.; McKinley, K. S.; Moss, J. E. B.; Wang, Z.; and Cheng, P., 2004. The garbage collection advantage: Improving program locality. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '04, 69–80. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1028976.1028983. https://dl.acm.org/doi/10.1145/1028976.1028983. (cited on page 29)

Intel, 2022. Data Dependent Prefetcher. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/data-dependent-prefetcher.html. (cited on page 10)

Intel, 2023. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel Corporation. https://www.intel.com/content/www/us/en/content-details/671488/intel-64-and-ia-32-architectures-optimization-reference-manual.html. (cited on pages 10, 48, and 56)

Jaffer, S. and Warburton, R., 2020. The JVM's mysterious AllocatePrefetch options: What do they actually do? https://www.opsian.com/blog/jvms-allocateprefetch-options/. (cited on page 17)

Jibaja, I.; Blackburn, S. M.; Haghighat, M. R.; and McKinley, K. S., 2011. Deferred gratification: Engineering for high performance garbage collection from the get go. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, 58–65. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1988915.1988930. https://dl.acm.org/doi/10.1145/1988915.1988930. (cited on page 21)

Jones, R.; Hosking, A.; and Moss, E., 2023. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman and Hall/CRC, New York, 2 edn. ISBN 978-1-03-221803-8. doi:10.1201/9781003276142. https://gchandbook.org/. (cited on pages 6 and 33)

Lee, J.; Kim, H.; and Vuduc, R., 2012. When Prefetching Works, When It Doesn't, and Why. *ACM Transactions on Architecture and Code Optimization*, 9, 1 (Mar. 2012), 2:1–2:29. doi:10.1145/2133382.2133384. https://dl.acm.org/doi/10.1145/2133382.2133384. (cited on page 9)

Liden, P. and Karlsson, S., 2018a. JEP 333: ZGC: A Scalable Low-Latency Garbage Collector (Experimental). https://openjdk.org/jeps/333. (cited on page 1)

Liden, P. and Karlsson, S., 2018b. The Z Garbage Collector - Low Latency GC for OpenJDK. (cited on page 1)

Lieberman, H. and Hewitt, C., 1983. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26, 6 (Jun. 1983), 419–429. doi:10.1145/358141.358147. https://dl.acm.org/doi/10.1145/358141.358147. (cited on page 6)

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, 89–98. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2926697.2926707. https://dl.acm.org/doi/10.1145/2926697.2926707. (cited on page 8)

Luk, C.-K. and Mowry, T. C., 1996. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, 222–233. Association for Computing Machinery, New York, NY, USA. doi:10.1145/237090.237190. https://dl.acm.org/doi/10.1145/237090.237190. (cited on pages 10 and 15)

Marshall, P., 2019. Trash talk: The Orinoco garbage collector · V8. https://v8.dev/blog/trash-talk. (cited on page 1)

McCarthy, J., 1960. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3, 4 (Apr. 1960), 184–195. doi:10.1145/367177.367199. https://dl.acm.org/doi/10.1145/367177.367199. (cited on page 5)

MMTk Contributors, 2023. Memory Management Toolkit. https://www.mmtk.io/. (cited on pages 8 and 23)

OpenJDK contributors, 2023. The HotSpot Group. https://openjdk.org/groups/hotspot/. (cited on pages 16 and 25)

Ossia, Y.; Ben-Yitzhak, O.; Goft, I.; Kolodner, E. K.; Leikehman, V.; and Owshanko, A., 2002. A parallel, incremental and concurrent GC for servers. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, 129–140. Association for Computing Machinery, New York, NY, USA. doi:10.1145/512529.512546. https://doi.org/10.1145/512529.512546. (cited on page 7)

Paz, H. and Petrank, E., 2007. Using Prefetching to Improve Reference-Counting Garbage Collectors. In *Proceedings of the 16th International Conference on Compiler Construction*, CC'07, 48–63. Springer-Verlag, Berlin, Heidelberg. (cited on pages 16 and 64)

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, 31–47. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3314221.3314637. https://dl.acm.org/doi/10.1145/3314221.3314637. (cited on page 17)

Roth, A. and Sohi, G. S., 1999. Effective jump-pointer prefetching for linked data structures. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, 111–121. IEEE Computer Society, USA. doi:10.1145/300979.300989. https://dl.acm.org/doi/10.1145/300979.300989. (cited on page 15)

Ruby Contributors, 2023. Gc.c: Ruby garbage collection code. https://github.com/ruby/ruby/blob/master/gc.c. (cited on page 1)

Shahriyar, R.; Blackburn, S. M.; Yang, X.; and McKinley, K. S., 2013. Taking off the gloves with reference counting Immix. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, 93–110. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2509136.2509527. https://doi.org/10.1145/2509136.2509527. (cited on page 1)

Shaughnessy, P., 2013. *Ruby Under a Microscope: An Illustrated Guide to Ruby Internals*. No Starch Press. ISBN 978-1-59327-527-3. https://nostarch.com/rum. (cited on page 7)

Shipilev, A., 2019. JVM Anatomy Quark #4: TLAB allocation. https://shipilev.net/jvm/anatomy-quarks/4-tlab-allocation/. (cited on page 17)

Tene, G.; Iyengar, B.; and Wolf, M., 2011. C4: The continuously concurrent compacting collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, 79–88. Association for Computing Machinery, New York, NY, USA. doi:10.1145/1993478.1993491. https://doi.org/10.1145/1993478.1993491. (cited on page 1)

Thompson, K., 1984. Reflections on trusting trust. *Communications of the ACM*, 27, 8 (Aug. 1984), 761–763. doi:10.1145/358198.358210. https://dl.acm.org/doi/10.1145/358198.358210. (cited on page 58)

Ungar, D., 1984. Generation Scavenging: A non-disruptive high performance storage reclamation algorithm. *ACM SIGPLAN Notices*, 19, 5 (Apr. 1984), 157–167. doi:10.1145/390011.808261. https://dl.acm.org/doi/10.1145/390011.808261. (cited on page 6)

Van Groningen, J., 2004. Faster Garbage Collection Using Prefetching. In *Proceedings of the Sixteenth International Workshop on Implementation and Application of Functional Languages*, IFL '04. Lubeck, Germany. https://core.ac.uk/reader/16146278. (cited on page 14)

Vicarte, J. R. S.; Flanders, M.; Paccagnella, R.; Garrett-Grossman, G.; Morrison, A.; Fletcher, C. W.; and Kohlbrenner, D., 2022. Augury: Using Data Memory-Dependent Prefetchers to Leak Data at Rest. In *2022 IEEE Symposium on Security and Privacy (SP)*, 1491–1505. doi:10.1109/SP46214.2022.9833570. https://ieeexplore.ieee.org/document/9833570. (cited on page 10)

Wu, H.; Ji, Z.; Zhu, S.; and Chen, Z., 2013. A Performance Study of Software Prefetching for Tracing Garbage Collectors. In *Advanced Parallel Processing Technologies*, Lecture Notes in Computer Science, 160–169. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-45293-2_12. (cited on page 15)

Wulf, Wm. A. and McKee, S. A., 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23, 1 (Mar. 1995), 20–24. doi:10.1145/216585.216588. https://dl.acm.org/doi/10.1145/216585.216588. (cited on page 8)

Xu, B.; Moss, E.; and Blackburn, S. M., 2022. Towards a Model Checking Framework for a New Collector Framework. In *Proceedings of the 19th International Conference on Managed Programming Languages and Runtimes*, MPLR '22, 128–139. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3546918.3546923. https://dl.acm.org/doi/10.1145/3546918.3546923. (cited on page 8)

Yang, A. M.; Österlund, E.; and Wrigstad, T., 2020. Improving program locality in the GC using hotness. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, 301–313. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3385412.3385977. https://dl.acm.org/doi/10.1145/3385412.3385977. (cited on page 29)

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, 307–324. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2048066.2048092. https://dl.acm.org/doi/10.1145/2048066.2048092. (cited on page 16)

Zhao, W.; Blackburn, S. M.; and McKinley, K. S., 2022. Low-latency, high-throughput garbage collection. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, 76–91. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3519939.3523440. https://dl.acm.org/doi/10.1145/3519939.3523440. (cited on pages 1, 8, and 64)