Improving Garbage Collection Observability with Performance Tracing

Claire Huang
Claire.Huang@anu.edu.au
Australian National University
Australia

Stephen M. Blackburn
steveblackburn@google.com
Google
Australia
Australian National University
Australia

Zixian Cai
zixian.cai@anu.edu.au
Australian National University
Australia

Abstract

Debugging garbage collectors for performance and correctness is notoriously difficult. Among the arsenal of tools available to systems engineers, support for one of the most powerful, *tracing*, is lacking in most garbage collectors. Instead, engineers must rely on counting, sampling, and logging. Counting and sampling are limited to statistical analyses while logging is limited to hard-wired metrics. This results in cognitive friction, curtailing innovation and optimization.

We demonstrate that tracing is well suited to GC performance debugging. We leverage the modular design of MMTk to deliver a powerful VM and collector-neutral tool. We find that tracing allows: i) cheap insertion of tracepoints—just 14 lines of code and no measurable run-time overhead, ii) decoupling of the declaration of tracepoints from tracing logic, iii) high fidelity measurement able to detect subtle performance regressions, while also allowing iv) interrogation of a running binary. Our tools crisply highlight several classes of performance bug, such as poor scalability in multi-threaded GCs, and lock contention in the allocation sequence. These observations uncover optimization opportunities in collectors, and even reveal bugs in application programs.

We showcase tracing as a powerful tool for GC designers and practitioners. Tracing can uncover missed opportunities and lead to novel algorithms and new engineering practices.

CCS Concepts: • **Software and its engineering → Garbage collection**; **Software performance**.

Keywords: Garbage collection, Performance analysis, eBPF

ACM Reference Format:
Claire Huang, Stephen M. Blackburn, and Zixian Cai. 2023. Improving Garbage Collection Observability with Performance Tracing. In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '23), October 22, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3617651.3622986

MPLR '23, October 22, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0380-5/23/10.
https://doi.org/10.1145/3617651.3622986

1 Introduction

Dan Luu [27] and Dick Sites [37–39] eloquently explain why low-overhead tracing has become one of the most important performance debugging tools for systems engineering. Tracing complements and improves over widely-used sampling and logging. Unlike sampling, which does not preserve order and thus is inherently *correlative*, tracing creates an execution trace by recording information every time certain program points are executed, preserving order and exposing *causation*. Unlike logging, tracing offers *observability* in the sense that one can "ask arbitrary questions about [the GC] without—and this is the key part—having to know ahead of time what [one] wanted to ask" [22]. Yet, although high-performance garbage collection (GC) is critical to managed languages, and is heavily tuned in production virtual machines, modern production collectors do not support tracing.

This lack of causality and observability among standard tools hinders our ability to inquire about the inner workings of these intricate collectors and thus our ability to generate new insights which might advance new designs and engineering practices. We show how modern tracing frameworks make GC more observable, especially when combined with modular collector design, doing so *cheaply*, *flexibly*, and with *high fidelity*.

An extensive body of literature compares the performance of different GCs [6, 8, 11, 50], analyzing the cost of various GC components [9, 21, 46, 48, 49], and, more recently, estimating the overall cost of GCs [12]. However, these methodologies and techniques do not address GC observability. Current GC implementations suffer from three related problems: i) metrics that operate at the temporal granularity of GC phases lack fidelity, ii) implementors must guess at the needs of unseen problems in order to predict a set of useful but low-overhead metrics to build in, and iii) altering the set of built-in metrics requires non-trivial effort. Widely-used sampling profilers such as VTune [13] side-step these problems, but run into others: i) their temporal resolution is intrinsically coarse due to their reliance on interrupts [27, 47], and ii) they

report correlation not causation. We now delve deeper into each of these problems, and outline our contributions.

Low-fidelity metrics. In the production GCs we survey, most metrics logged are timing data organized around the phases of GC (such as root scanning). However, GC-phase metrics are insufficient for performance debugging. For example, a slow transitive closure could be an artifact of the shape of the application's object graph [5], or due to frequent synchronization among threads for work stealing, poor load balancing, etc. Moreover, coarse-grained metrics mask subtle performance regressions, inviting an accumulation of performance debt. Likewise, mutator behaviors like allocation and barriers are critical to understanding overall performance, yet are often not reported by GCs.

Crystal ball required. One cause of poor fidelity in these metrics is the cost of logging. Logging everything is impractical, so GC developers have to predict the set of metrics to include for each logging scope and verbosity level. Some logging statements are even guarded by macros, as run-time checks in performance hot-paths are too costly.[1] Lack of fidelity is especially problematic when troubleshooting transient performance bugs on a live system, where reported metrics are insufficient and adding more metrics requires modifying and rebuilding the virtual machine.

Non-trivial engineering and recompilation. To add new instrumentation, programmers need intimate knowledge of the codebase: where to add instrumentation; and the correct data structure that is thread-safe and low-overhead. Worse, when metrics are emitted as unstructured text, each additional metric requires changes to analysis tools' parsers. As a result, several parsers are built for the same VM [3, 30, 43].

Sampling is Not Enough. Dan Luu notes: *"Sampling profilers are great. They tend to be easy-to-use and low-overhead compared to most alternatives. However, there are large classes of performance problems sampling profilers can't debug effectively, and those problems are becoming more important."* [27]. A sampler only reports how many times it witnessed particular functions being executed. It cannot observe how long a particular function invocation took, what occurred immediately before it, or the distribution of invocation times for that function over the course of an execution. Sample rates are uniform, so such tools must sample the entire application with sufficient frequency to expose the rarest events they wish to reveal. Worse, sampling's temporal resolution is limited by its dependence on interrupt handling [47].

To address these problems, we combine two simple ideas: low-overhead tracepoints and modular collector design. Low overhead tracepoints can be compiled into a production binary and attached to on demand, including on a running binary. A highly modular collector design means that just a few tracepoints (totalling just 14 lines of code) provide a very rich perspective on collector behavior. We illustrate this using MMTk, which is runtime-agnostic, and uses a modular collector-neutral work packet system [45].

This paper makes the following key contributions:

- We demonstrate that inserting tracepoints in the codebase is *inexpensive* both in software engineering terms (14 lines of code, no parser required), and performance terms (no statistically significant performance overhead).
- We demonstrate the *flexibility* of writing different tracing tools against the same set of events in a running binary. In our case study of the transitive closure phase of the GC, we are able to interrogate a live system and measure the scalability of multithreaded GCs, monitor the properties of work packets [45] in-flight in a scheduling system, and examine how different workloads affect these run-time behaviors. The entire investigation is performed on the same binary.
- We show that these events are *high fidelity*, amplifying subtle performance regressions.

Because we combine tracepoints with a modular code base, these advantages readily translate to each language implementation and collector that MMTk supports. Still, we expect that any collector implementation will benefit from tracing.

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

2 Background and Related Work

Systems engineers rely on an arsenal of performance analysis tools. We can group these tools into three approaches [38]: i) *counting*, ii) *sampling*, iii) and *tracing*. These are complementary—all are important to the performance analysis toolkit. We describe each in more detail and show how they apply to garbage collectors. We then survey popular tracing frameworks, and describe their use in managed language runtimes. We show that flexible, low-overhead tracing is absent in most garbage collectors, a gap which we will address in this paper. Finally, we give a brief overview of the memory management framework we use, MMTk, and its work packet system to which we apply performance tracing.

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

2.1 A Taxonomy of Performance Analysis Tools

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

2.1.2 Sampling. Sampling tools quasi-periodically [2] collect measurements to create a statistical profile of the program. Timer interrupts are used to power widely sampling profilers, such as Intel VTune [13], and OProfile [33]. When an interrupt fires, these profilers observe program state such as call stacks. When the program ends, the stack samples are used to identify "hot" methods and their calling context, which can provide clues to programmers as to where to focus optimization effort. Though a very useful performance debugging tool, sampling is often inadequate when performance debugging GCs due to: i) lack of context, ii) sampling bias, iii) and not able to establish causal relationships.

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

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

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

Sampling Bias. It is well understood [4, 32, 47] that sampling profilers are prone to sampling bias. In particular, they are not well suited to analyze rare events—these events can be *unpredictable* but greatly affect the tail performance of workloads. Examples of rare events in GC are large object allocations and long GC pauses due to class unloading.

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

If sampling tools were to observe relatively rare events, they would need to sample the whole program at the level of fidelity they need for the rare event, which incurs very high overheads. The sampling frequency is also upper-bounded by the frequency of kernel non-maskable interrupts: any faster sampling frequency can prevent the entire OS from making progress. As a result, most sampling-driven profilers (e.g., Intel VTune) operate between 1–1000 Hz for user-space sampling, and up to 100 000 Hz for hardware event-based sampling, which is inadequate for analyzing rare events [14].

Sampling tools are oblivious to non-CPU waiting time (i.e., a thread is sleeping instead of spinning), and certain kernel time (such as the timer interrupt routine itself, or non-interruptible kernel code), distorting user-space time [38].

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

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

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

Sampling-based profilers sometimes adopt elements of tracing. For example, the Concurrency Visualizer in Visual Studio [15] normally samples the program stack, but *traces*

every context switch. This is so that the control flow leading to blocking IO or synchronization can be comprehensively captured, rather than be subject to vagaries of sampling.

2.2 Tracing Technologies

Though powerful, tracing tools are still subject to the trade-off between precision[3] and the impact on the system under test. Our goal is to develop low-overhead, high-fidelity performance tracing for GC. We survey potential tracing technologies in this section. In this paper, we use eBPF.

eBPF. eBPF (extended Berkeley Packet Filter) provides a safe and flexible way for developers to extend the capability of the operating system kernel to facilitate low overhead tracing. The key component is a sandboxed in-kernel BPF virtual machine (VM) with a verifier. Users can write BPF programs targeting the BPF VM to respond to kernel and userspace events [18, 20]. An example of a kernel event is a context switch, while a user event might be the execution of a user-defined tracepoint. eBPF has a wide range of applications, including troubleshooting slow disk IO requests, etc., as well as its namesake application, filtering network packets.

eBPF has two main tracing frontends: BCC [23] and bpftrace [24]. BCC provides a more customizable, C-like programming interface, while bpftrace provides a convenient scripting language inspired by DTrace [42] and SystemTap [35]. Both compile tracing programs (probes) into BPF VM bytecode. In this paper, we focus on using bpftrace to trace userspace events in a memory management framework.

We consider `usdt` probes and `uprobe`/`uretprobe` pairs. `usdt` probes attach to User Statically-Defined Tracepoints (USDTs), while `uprobes` and `uretprobes` are tracing programs attached to userspace function entries and exits respectively. While `uprobes` can in principle be used with any application, their utility is limited to non-inlined functions with exposed symbols and their pre-existing function arguments.

Developers can use USDTs to mark interesting points in the code base (Figure 1). Each USDT has several parts: the provider (namespace), name, and zero or more arguments[4] to provide the context of execution. USDTs are implemented by inserting `nop` instructions into the application binary at compile time. The locations (instruction pointers) of each USDT are added to the notes section of the ELF file. A probe attaches to a USDT by patching the `nop` into a breakpoint, and the kernel executes the corresponding tracing programs when handling the breakpoint.

The main overheads of tracing a userspace program come from a privilege mode switch (user space into kernel space),

[3]A precise measurement distorted by observer effects is not accurate.
[4]Each USDT can optionally have a semaphore. The semaphore provides a tradeoff between an extra check before a USDT can fire and potentially expensive computation of USDT arguments.

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

Figure 1. Inserting a USDT tracepoint into Rust code, which expands into a `nop`. The first argument to `probe!` is the provider/namespace, and the second is the name of the tracepoint, followed by zero or more arguments. In this case, the tracepoint has two arguments, one for the number of edges, and another for whether the edges are root edges.

collecting the userspace thread context (registers), and executing the specific tracing logic defined by the user in the in-kernel VM.

Our target, MMTk, is written in Rust. We use the Rust `probe` crate [41] to add USDT tracepoints to our source code, such as to trace the number of object graph edges a function iterates over and whether the edges are root edges (Figure 1). Then, we can attach a probe (Figure 2) to this tracepoint to understand the distribution of the number of edges in all `ProcessEdges` packets, and root `ProcessEdges` packets. (The significance of `ProcessEdges` is expanded upon later.)

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

Figure 2. A probe written in the bpftrace language. It attaches to the `process_edges` tracepoint in Figure 1. It records the number of edges (`arg0`) into histogram data structures to estimate the distribution of the number of edges in each function invocation, separately counting root edges (when `arg1` is true).

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

LTTng and Perfetto. Unlike eBPF, which is implemented completely inside the operating system kernel, LTTng [17] and Perfetto [2] can trace userspace programs from user space. They achieve this through shared memory ring buffers between the program under trace and the tracing daemon process.[5] The buffer is periodically flushed to disk through serialization. Since no kernel/userspace transition is required when tracing userspace programs, LTTng and Perfetto can have lower tracing overhead, but can incur substantial disk usage and post-processing cost, which may be prohibitive when tracing high frequency events. Both LTTng and Perfetto can trace kernel events, through a kernel module, and the Linux ftrace interface respectively.

Perfetto comes with a web UI that can visualize several trace formats. We use its UI frontend to visualize garbage collection work. We do not use Perfetto to perform tracing.

2.3 Tracing in Managed Language Virtual Machines

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

Python, Ruby, and Julia. Python [28], Ruby [36], and Julia [25] all provide GC tracepoints. Any tracer that understands the USDT format, including bpftrace and DTrace, can attach to these tracepoints. For these runtimes, the tracepoints implemented within their native collectors only tell us when a GC phase begins or ends, and when allocations occur. These tracepoints do not provide sufficient detail to properly understand collector behavior. We claim that with a sufficiently modular collector design, just a few USDT tracepoints can provide rich insights into collector behavior.

V8 and Android Runtime. V8 [44] uses the Chromium tracing framework. It reports the time spent in most GC phases and the sizes of spaces. It also collects the memory footprint of different types of objects. Android Runtime (ART) [1] uses the Perfetto tracing framework. ART records different phases of GC, and how heap sizes change with each GC epoch. Both V8 and ART use Perfetto as the UI frontend for visualizing traces. Neither provide the level of detail and generality that we demonstrate here.

.NET and OpenJDK. Compared with the other language runtimes listed above, .NET [29] and OpenJDK allow more events to be observed. They report more statistics after each GC, such as the size of each generation, and allow profiling of allocations and heap composition. Their tracing interfaces can be enabled during run time, supporting tools like GCRealTimeMon [40] and JDK Mission Control [34] respectively. A key distinction with the work we describe here is due to MMTk's modular design and use of work packets, which we describe below. These features allow us to reveal rich insights into the collectors' behavior in ways that are general, collector-neutral, and very low overhead. Without modularity and generality, specificity is necessary and the developer can only guess at which elements of the collector are going to be most useful to the end user.

[5] Perfetto has an in-process tracing mode which does not require a daemon process but it can only trace the residing program.

2.4 MMTk and its Work Packets

MMTk is a garbage collection framework that provides a robust, efficient, portable, and flexible platform for developers to build collectors [7, 31]. Almost all collection work is done by stateless GC workers executing *work packets* [45]. Work items of the same type (such as objects to be scanned) are placed into work packets (such as object scanning work packets), whose *sizes* are the numbers of items within. Each work packet contains a function pointer instructing the GC worker on how to process these work items, and preconditions dictating when it may be executed.

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

The distribution of work packets among worker threads is key to scalability. Worker threads acquire work packets from an initial global pool and then consume and produce work packets into thread-local pools. Once the global pool is empty and a thread's local pool is exhausted, it steals work from local work packet pools of other threads. Both work stealing and acquisition of global work packets requires synchronization of threads to avoid race conditions. Thus, the choice of packet size involves a trade-off between increased parallelization and increased synchronization.

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

3 Implementing Tracing in MMTk

We added to MMTk just 14 lines of substantive Rust code, comprising USDT tracepoints that provide observability to five major behaviors: i) the start and end of each collection (2 LOC), ii) the start and end of processing each work packet (5 LOC), iii) the start and end of each execution of the allocator slow path (2 LOC), iv) the opening of each work bucket

(2 LOC), and v) the `process_edges` method (3 LOC). There are an additional ten lines of code to make the above compile conditionally, and nine more lines of import statements and trailing braces. We will argue (Section 5) that these tracepoints impose an overhead so low that there is no need to conditionally compile them.

While logging dumps textual summaries of what a collector has done (requiring foreknowledge of what the consumer may find useful), the tracepoints we add provide fine-grained semantically rich building blocks from which the user can perform powerful analysis (Section 6). The power of this approach is greatly assisted by a highly modular design of MMTk. It is not our goal to be exhaustive in the tracepoints we add. Rather, we demonstrate that a few tracepoints add tremendous observability to the garbage collector, and yet, they come with negligible overhead.[6] We now describe each of the five behaviors observed by the tracepoints we added.

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

Work Packet Start and End. This pair of tracepoints allows the time spent processing each work packet to be observed. The tracepoints also capture the type of the packet (passed as a string reflecting the Rust type). Together, this information provides users with insight into the details of how work is being performed within a garbage collection, how well it is being parallelized, and which packets and packet types are responsible for performance anomalies. These work packet events can be readily organized into the format of various tracing visualization tools, such as the JSON format of Perfetto [19]. Combined with the bucket opening tracepoint (below), we can further visualize these events by the stages of GC they appear in. These events give us an intuitive yet rich understanding of the dynamics of collection work, which we later illustrate (Figure 5). When attached, they fire roughly once every microsecond during GC.

Allocator Slow Path. Most allocators are designed around a very efficient common-case fast path, which is usually thread-local and unsynchronized, and an infrequently-taken slow path, which is used only when the fast path exhausts its cache of available memory. This pair of tracepoints allows the time spent in allocation slow path code to be observed. When attached, they fire approximately once for every 64 KiB of allocation, which can be up to once every 10 µs.

Bucket Opening. This tracepoint allows us to observe the opening of each work bucket. Since buckets are opened at different points during the GC only when certain preconditions are satisfied (Section 2.4), the timing of bucket openings provides insight into how long various stages of the garbage collection take. When attached, this tracepoint fires in the order of once a millisecond during GC.

The `process_edges` Function. Each of the above tracepoints is very generic. In contrast, this tracepoint provides observability into `ProcessEdges` work packets. These packets collectively perform the transitive closure over the object graph, a task which dominates most tracing-based collection work, so we add this tracepoint as an important special case.

We add the tracepoint to the body of `ProcessEdges` work packets—the `process_edges()` function—to report the number of edges the function visits and record whether these edges are roots of the transitive closure.

Measuring the `ProcessEdges` work packets' sizes allows us to gain more insight into problems regarding the scalability and efficiency of tracing. As discussed in Section 2.4, this is because larger packets better amortize the fixed cost of fetching a packet, while smaller packets are more evenly distributed among worker threads. When attached, they fire approximately as frequently as the work packet start and end tracepoint; up to once every microsecond during GC.

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

4 Methodology

We now present the baseline methodology used throughout the remainder of this paper. In the overhead analysis and case studies that follow, we describe the relevant methodology in terms of variations on this baseline methodology.

Hardware and Software Platform. Our work is available as part of the new Rust-based implementation of MMTk [6, 7, 31]. We use the 0.19.0 release of MMTk and its OpenJDK binding (based on OpenJDK `11.0.19+1`) in our evaluation. MMTk is built using version 1.71.1 of the Rust compiler, utilizing profile guided optimization. We use version 0.5 of the Rust `probe` crate to add USDT tracepoints to MMTk. Both machines we use have Ubuntu 22.04.3 installed with the `5.15.0-79-generic` kernel. We use the official binary release of version 0.18 of bpftrace.

We use two hardware platforms described in Table 1, which we refer to as Zen3 and CoffeeLake respectively.

JVM parameters. We set the heap size relative to a modest three times the minimum heap size required by each benchmark using the Immix collector. We also use `-XX:-TieredCompilation -Xcomp` flags to speed up the warmup of the JVM, except for tradebeans and tradesoap because they cannot

[6] i.e. with no statistically significant overhead.

Table 1. Specifications for the hardware used in the study.

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

run when forcing C2 compilation.[7] We disable compressed pointers, weak references, and class unloading because the MMTk/OpenJDK binding does not support these features as of writing. The tracepoints we add are entirely collector-agnostic. The methodology, including the specific scripts we use, is applicable to other collectors in MMTk. We focus on the Immix collector [10] due to space constraints. This full heap, stop-the-world collector is performant and easier to understand than concurrent or generational collectors, which aids exposition.

Benchmarks. We use 21 diverse up-to-date benchmarks from the Chopin development branch of the DaCapo benchmark suite [8, 16].[8] We exclude batik because it does not perform GC with the heap size we use. We fix a bug in lusearch during our case studies (Section 6.2), and we include its original buggy version (lusearch_bug)[9] in our evaluation.

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

5 Tracing Overheads

We evaluate the overheads associated with tracing considering three scenarios: i) the overhead of adding tracepoints (nops) to MMTk (but without attaching probes to any of them at run time), ii) the overhead of attaching a probe to the bucket open tracepoint to observe GC phases, iii) the overhead of attaching a probe to the `process_edges` tracepoint to observe packet sizes, and iv) the overhead of attaching probes to the allocator slow path start and end tracepoints to observe the slow path durations. For each of the three observation applications, we attach probes to the GC start/end tracepoints to count the number of GC epochs. We use both the Zen3 and CoffeeLake platforms to assess the architectural sensitivity of the results, presented in Table 2 and Table 3 respectively.

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

In analyzing Table 2, we first consider the overhead of adding tracepoints (first column, tp). The average impact on total execution time is within the noise of our measurement: a nominal 0.4 % slowdown on the Zen3, and a 0.3 % speedup on CoffeeLake. Looking at the per-benchmark results on Zen3, we see that the overhead is within measurement noise for every benchmark. With respect to collector time, we see slowdowns of 0.8 % and 0.3 % on the Zen3 and CoffeeLake respectively. These results show that the impact of adding the tracepoints to the codebase is negligible. This low overhead paves the way for compiling in and deploying tracepoints in production settings. This will allow high fidelity measurements of the system at any time without recompiling or redeploying.

Now we consider the cost of attaching a probe to the bucket open tracepoint (second column, ob). We see no overhead in total time (0.0 %) on the Zen3, while on the CoffeeLake we see a nominal speedup of 0.2 %. With respect to collector time, we see slowdowns of 1.0 % on the Zen3 and 0.7 % on the CoffeeLake. These overheads are low enough that the bucket open tracepoints could be left attached in most cases.

Next, we look at the case where `process_edges` is observed, allowing us to measure the size of each `ProcessEdges` work packet (third column, pe). This is executed many orders of magnitude more frequently than the bucket open tracepoint. Total time overheads are now measurable, 1.7 % and 1.3 % respectively on the Zen3 and CoffeeLake. The collector sees slowdowns of 7.3 % and 12.6 % on the Zen3 and CoffeeLake respectively. While measurable slowdowns, they remain small enough that the tracepoints could also be continuously attached in some contexts.

The above two studies concern the collector. Now we measure the overhead of measuring the allocation slow path (fourth column, as). Although a copying collector also allocates during GC, allocation is predominately performed by mutators. The total time overheads are similar to the above study, 1.0 % and 2.2 % respectively on the Zen3 and CoffeeLake. The mutator sees slowdowns of 1.1 % and 2.4 % on the Zen3 and CoffeeLake respectively. Again, the overhead is

[7] https://github.com/dacapobench/dacapobench/issues/198
[8] Commit 6ea164a5, 2023/09/10
[9] Commit 0d047f55, 2023/02/02.

Table 2. Tracing overheads on Zen3 for total time, stop-the-world collector time, and mutator time, for each benchmark normalized to the base case of an unmodified MMTk. In each group of four columns, the first measures tracepoints compiled in but not attached at run time (tp). The second uses the open bucket tracepoint (ob), the third uses the `process_edges` tracepoint (pe), and the last one uses the allocation slow path tracepoints (as). In each case, we show the mean and the 95 % confidence interval (gray) of the time normalized to the base case, aggregating across 30 invocations.

	Total				STW Collector			Mu.
Benchmark	tp	ob	pe	as	tp	ob	pe	as
avrora	0.995 −1.3 % +1.4 %	1.000 −1.3 % +1.3 %	0.999 −1.5 % +1.5 %	1.003 −1.4 % +1.5 %	1.004 −2.5 % +2.6 %	1.022 −2.2 % +2.3 %	1.048 −2.2 % +2.2 %	1.002 −1.4 % +1.5 %
biojava	1.003 −0.5 % +0.5 %	1.003 −0.5 % +0.5 %	1.000 −0.5 % +0.5 %	1.021 −0.4 % +0.4 %	1.015 −1.1 % +1.1 %	1.014 −1.1 % +1.1 %	1.018 −1.0 % +1.0 %	1.022 −0.4 % +0.4 %
cassandra	1.000 −0.5 % +0.5 %	0.999 −0.5 % +0.5 %	0.998 −0.5 % +0.5 %	1.000 −0.5 % +0.5 %	0.995 −2.2 % +2.2 %	1.001 −2.6 % +2.6 %	1.016 −2.3 % +2.4 %	1.000 −0.5 % +0.5 %
eclipse	1.009 −11.6 % +12.4 %	0.956 −5.8 % +6.5 %	1.007 −11.4 % +12.1 %	0.984 −6.0 % +6.7 %	1.002 −0.9 % +0.9 %	1.008 −0.9 % +0.9 %	1.020 −0.9 % +0.9 %	0.983 −6.0 % +6.8 %
fop	0.995 −0.5 % +0.5 %	0.996 −0.6 % +0.6 %	0.996 −0.5 % +0.5 %	1.033 −0.5 % +0.5 %	0.989 −4.5 % +4.7 %	0.972 −4.1 % +4.3 %	0.997 −4.2 % +4.4 %	1.034 −0.5 % +0.5 %
graphchi	1.000 −0.1 % +0.1 %	0.999 −0.5 % +0.5 %	1.002 −0.1 % +0.1 %	1.009 −0.3 % +0.3 %	1.015 −1.0 % +1.0 %	1.021 −0.9 % +0.9 %	1.025 −0.9 % +0.9 %	1.009 −0.3 % +0.3 %
h2	1.008 −1.0 % +1.0 %	1.005 −1.0 % +1.0 %	1.011 −1.0 % +1.0 %	1.018 −1.0 % +1.0 %	1.022 −1.3 % +1.3 %	1.010 −1.2 % +1.2 %	1.018 −1.1 % +1.1 %	1.016 −1.1 % +1.1 %
h2o	1.002 −0.9 % +0.9 %	1.005 −0.9 % +0.9 %	1.002 −0.9 % +0.9 %	1.058 −0.9 % +0.9 %	1.003 −1.6 % +1.7 %	1.014 −1.6 % +1.6 %	1.025 −1.8 % +1.8 %	1.062 −0.9 % +0.9 %
jme	1.000 −0.0 % +0.0 %	1.000 −0.0 % +0.0 %	1.000 −0.0 % +0.0 %	1.001 −0.0 % +0.0 %	1.023 −3.4 % +3.5 %	0.999 −4.1 % +4.2 %	1.312 −3.6 % +3.7 %	1.001 −0.0 % +0.0 %
jython	1.005 −0.3 % +0.3 %	1.002 −0.3 % +0.3 %	1.004 −0.3 % +0.3 %	1.036 −0.2 % +0.2 %	1.015 −1.6 % +1.6 %	1.005 −1.2 % +1.2 %	1.017 −1.2 % +1.3 %	1.038 −0.3 % +0.3 %
kafka	0.992 −4.8 % +5.1 %	0.997 −5.1 % +5.4 %	0.980 −3.8 % +4.1 %	0.979 −3.8 % +4.1 %	1.047 −6.5 % +7.5 %	1.045 −6.5 % +7.5 %	1.053 −6.5 % +7.5 %	0.978 −3.9 % +4.2 %
luindex	1.006 −0.5 % +0.5 %	1.006 −0.5 % +0.5 %	1.011 −0.5 % +0.5 %	1.021 −0.6 % +0.6 %	0.998 −2.4 % +2.4 %	1.013 −2.1 % +2.1 %	1.041 −2.2 % +2.3 %	1.022 −0.6 % +0.6 %
lusearch	1.008 −0.8 % +0.8 %	1.036 −4.8 % +4.8 %	1.015 −0.7 % +0.7 %	1.021 −0.7 % +0.8 %	1.027 −2.4 % +2.4 %	1.030 −2.5 % +2.5 %	1.046 −2.2 % +2.3 %	1.019 −0.3 % +0.3 %
lusearch_bug	1.070 −8.8 % +8.8 %	1.026 −4.7 % +4.7 %	1.333 −0.6 % +0.6 %	1.004 −0.6 % +0.6 %	1.005 −1.9 % +1.9 %	1.006 −2.0 % +2.0 %	1.829 −1.5 % +1.6 %	1.001 −0.6 % +0.6 %
pmd	0.997 −0.7 % +0.7 %	0.998 −0.7 % +0.7 %	0.998 −0.7 % +0.7 %	1.007 −0.7 % +0.7 %	0.982 −3.2 % +3.3 %	0.986 −3.4 % +3.5 %	1.003 −3.3 % +3.4 %	1.012 −0.3 % +0.3 %
spring	0.998 −0.7 % +0.7 %	0.995 −0.7 % +0.7 %	1.034 −0.8 % +0.8 %	1.008 −0.7 % +0.8 %	1.016 −1.5 % +1.5 %	1.009 −1.5 % +1.5 %	1.237 −1.3 % +1.3 %	1.006 −0.8 % +0.8 %
sunflow	1.017 −1.9 % +1.9 %	1.002 −1.9 % +2.0 %	1.028 −2.5 % +2.5 %	1.007 −1.9 % +1.9 %	1.017 −1.5 % +1.5 %	1.003 −1.3 % +1.3 %	1.020 −1.3 % +1.3 %	1.006 −2.2 % +2.3 %
tomcat	0.998 −0.3 % +0.3 %	0.999 −0.3 % +0.3 %	1.003 −0.3 % +0.3 %	1.007 −0.3 % +0.3 %	1.003 −1.8 % +1.9 %	1.009 −1.9 % +2.0 %	1.013 −1.7 % +1.7 %	1.007 −0.3 % +0.3 %
tradebeans	0.999 −0.9 % +0.9 %	0.999 −1.0 % +1.0 %	1.003 −0.9 % +0.9 %	1.017 −0.9 % +0.9 %	1.002 −1.6 % +1.6 %	1.005 −1.7 % +1.8 %	1.025 −1.7 % +1.7 %	1.019 −0.9 % +0.9 %
tradesoap	0.997 −0.6 % +0.6 %	1.001 −0.6 % +0.6 %	1.002 −0.6 % +0.6 %	1.012 −0.6 % +0.7 %	1.005 −1.6 % +1.6 %	1.003 −1.5 % +1.5 %	1.024 −1.5 % +1.5 %	1.013 −0.6 % +0.6 %
xalan	0.986 −0.8 % +0.8 %	0.997 −0.9 % +0.9 %	1.001 −0.9 % +0.9 %	0.990 −0.8 % +0.8 %	0.971 −1.8 % +1.8 %	0.990 −2.0 % +2.0 %	1.005 −2.0 % +2.0 %	0.993 −1.0 % +1.0 %
zxing	0.993 −1.3 % +1.3 %	0.992 −1.2 % +1.2 %	0.993 −1.2 % +1.2 %	0.997 −1.2 % +1.2 %	1.026 −4.6 % +4.8 %	1.056 −4.8 % +5.0 %	1.052 −4.6 % +4.8 %	0.996 −1.2 % +1.2 %
min	0.986	0.956	0.980	0.979	0.971	0.972	0.997	0.978
max	1.070	1.036	1.333	1.058	1.047	1.056	1.829	1.062
geomean	1.004	1.000	1.017	1.010	1.008	1.010	1.073	1.011

low enough that the allocation tracepoint could be routinely left attached, allowing us to correlate the memory manager behaviors with the benchmark events.

Together these results illustrate that the overheads of tracing via eBPF are very modest. Compiling the tracepoints into the binary incurs negligible overhead, enabling their use even in production settings. Furthermore, we show that interesting

Table 3. Summary of overheads for CoffeeLake. This table omits per-benchmark results for concision. The data here corresponds to the last three rows of Table 2, which reports overheads for Zen3.

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

tracing applications have such low overhead that in many cases they could be performed in production too.

6 Case Studies

In the following sections, we use two case studies to explore examples of how observability with performance tracing offers insights into collector and mutator performance. Each of the studies requires negligible storage, since the data is aggregated on the fly, except for the detailed per-work packet trace (Figure 5), which uses about 20 kB per GC epoch using a naïve JSON format after gzip compression.

6.1 Tracing Scalability: The `ProcessEdges` Work Packet

The performance of tracing collectors is typically dominated by how efficiently they perform a transitive closure over all live objects within the scope of the collection. The scope may be one or more regions, the nursery, or the whole heap. Most modern collectors utilize hardware parallelism to trace the heap. The performance of such a trace is influenced by three factors: i) the efficiency with which each object in the graph is processed, ii) the efficiency with which work is distributed among available workers, and iii) the amenability of the shape of the object graph to parallelization [5].

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

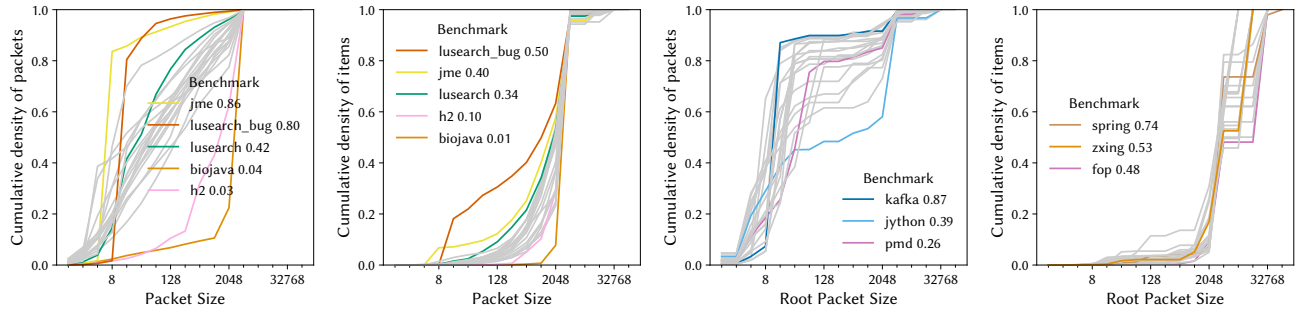
Methodology. We first attach a probe to the bucket open tracepoint to measure the length of the closure phase of the collection. We then attach a probe to the `process_edges` tracepoint to measure `ProcessEdges` packet sizes. Finally, we attach probes to the work packet start/end tracepoints to save raw timestamped data for Perfetto visualization.

Results and Discussion. We measure four different attributes of the `ProcessEdges` work: i) the overall scalability of the closure phase, in which it dominates, ii) the distribution of packet sizes, iii) the distribution of root packet sizes, and iv) the distribution of packets among threads over time.

Figure 4 shows the scalability of the closure phase of collections on each of our hardware platforms. On the Zen3 it ranges from pmd, which scales with 81.5 % efficiency at 8 threads (6.52/8), to fop, which is just 50.7 % efficient (3.96/8). xalan exhibits super-linear speedup on the Zen3. We ascribe this to an as-yet unconfirmed architecture-specific performance problem when running with a single thread. On the CoffeeLake scalability ranges from spring, which scales with 79.5 % efficiency (6.36/8) at 8 threads, down to fop, which is 52.6 % efficient (4.21/8). We now explore possible reasons for the lack of scalability seen by some of these workloads.

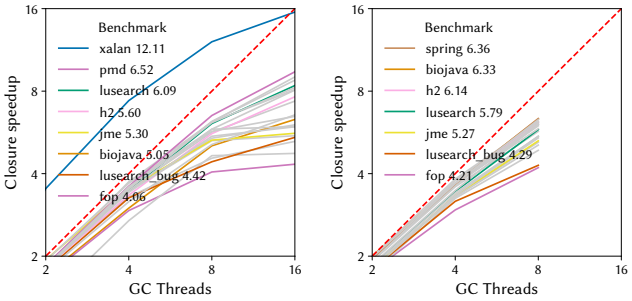
The most likely sources of a scalability problem in the closure phase are poor load balancing and some form of contention. We start by observing that the scalability is very workload-sensitive, and that for some workloads, the scalability is quite good. On this basis we investigate the distribution of work packet sizes. Small work packets indicate a performance problem since there is a fixed cost associated with acquiring a new packet. On the other hand, very large packets may inhibit load balancing, since MMTk workers can steal packets, but cannot steal work from within another packet. We also evaluate root packet sizes, since root packets prime the transitive closure and if they were large and unevenly distributed among workers, they could lead to load imbalance and starvation.

Figure 3(a) shows the distribution of packet sizes for all benchmarks on Zen3 as a cumulative distribution function. The distribution is similar on CoffeeLake. The outliers at the small end are jme and the buggy version of lusearch, for which about 86 % and 80 % of all packets are less than size 32. The bug fix for lusearch (Figure 5(c) and Figure 5(e)) significantly reduces the number of small work packets (down to



(a) All packets, by packet (16). **(b)** All packets, by item (1024). **(c)** Root packets, by packet (16). **(d)** Root packets, by item (8192).

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



(a) Zen3 (16 cores). **(b)** CoffeeLake (8 cores).

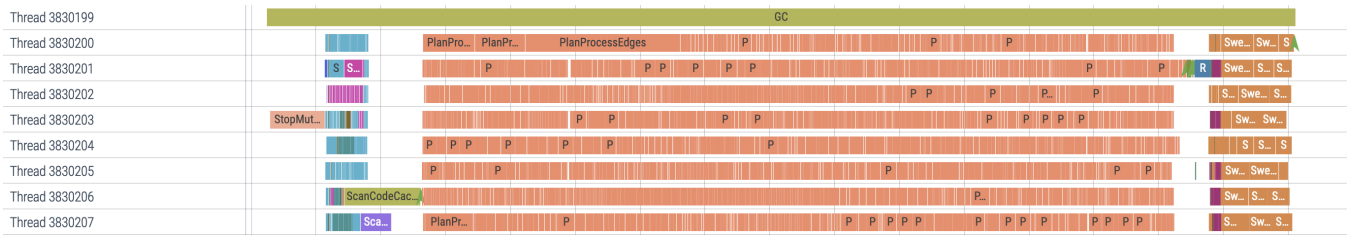
Figure 4. The scalability of MMTk's closure phase for each of the DaCapo benchmarks, showing speedup relative to using one worker thread (y-axis, log) as a function of the number of worker threads (x-axis, log). The dashed red line indicates perfect scaling. Selected benchmarks are labelled, the remainder shown in grey to indicate the distribution of results. Labels indicate speedup at 8 cores.

43 %). At the large end, the outliers are biojava, where about 90 % of packets are of size 1024 or more, and h2 where about 85 % of packets are at least 512 in size. Interestingly, these observations are *not* well correlated with scalability. In fact the four outliers from Figure 3(a) are clustered together in the middle of the Zen3 scalability curve (Figure 4(a))—despite their extremes in packet size distribution they all have unremarkable scalability. In Figure 3(b), we weight the curves by the size of the packets so that rather than reflecting the fraction of all packets, they reflect the fraction of all items. This different weighting is interesting. However, it comes no closer to explaining the lack of scalability.

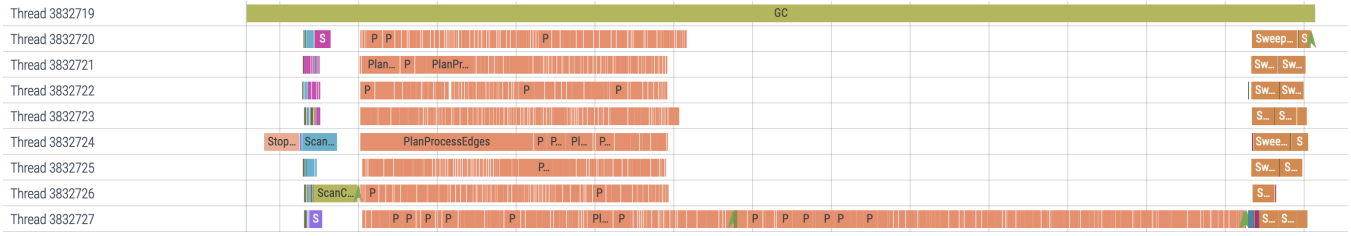
Another factor that may affect load balancing is the size of root packets—the work packets that prime the transitive closure. If the priming work is not well distributed among the threads, it might lead to poor load balancing and poor scalability. Fortunately, this is easy to investigate with the existing tracepoints. Figure 3(c) illustrates the distribution of root packet sizes and Figure 3(d) shows the same, weighted by packet size. Unfortunately, neither of these show any correlation with the scalability results in Figure 4. However, they reveal that some of the root work packets are very large, and Figure 3(d) indicates that most root edges come from packets of size 4096 or larger. This suggests an opportunity to improve root processing in MMTk's OpenJDK binding.

Finally, we use the tracepoints at the start and end of each work packet to visualize the execution of work packets as a function of time and thread affinity. Figures 5(a) – 5(c) show three collections, two from poorly scaling workloads (fop and lusearch_bug), and one from a well scaling workload (tomcat). In each case, time runs from left to right, and threads are presented as rows. There are eight worker threads and one coordinator thread (which is almost always idle). It is immediately obvious that the `ProcessEdges` work packets (orange) are poorly load balanced in the fop and lusearch_bug collections. Closer inspection reveals that the causes are different.

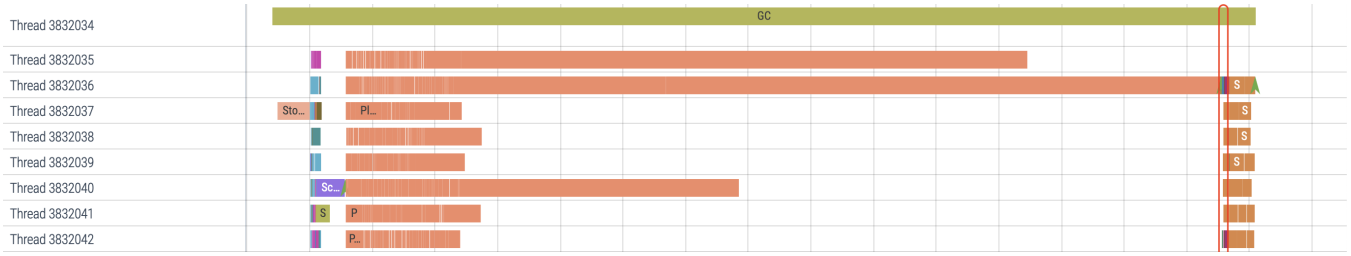
In the case of fop, the collector starts performing finalization work at about halfway through. The change in collection phase is denoted by a small green arrow on the bottom-most thread, which indicates the opening of the work bucket responsible for finalization. Although it is not immediately obvious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



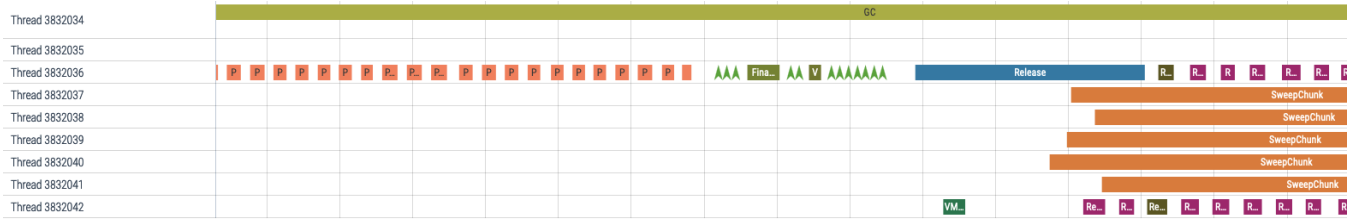
(a) A collection from tomcat, showing good load balancing.



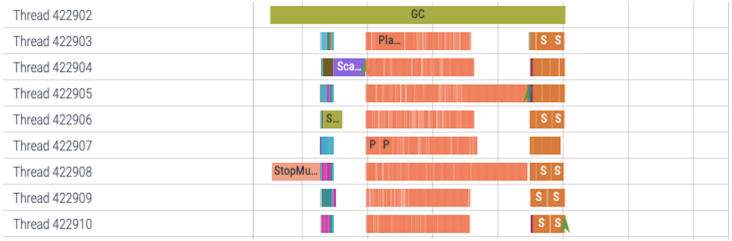
(b) A poorly load-balanced collection from fop.



(c) A poorly load-balanced collection from lusearch_bug. The period within the thin red box on the right is shown in detail in Figure 5(d).



(d) Zooming in on a very thin slice near the end of Figure 5(c). Thread 3832036 is executing a series of short `ProcessEdges` packets.



(e) A collection from lusearch after fixing the bug in lusearch that led to poor load-balancing. Compare to Figure 5(c).

Figure 5. Per-work packet tracepoints allow scalability problems to be quickly diagnosed using a visualization tool such as Perfetto, depicted in the screenshots above. The problems with fop (5(b)) and lusearch_bug (5(c)) are immediately clear: poor load balancing. Perfetto renders different work packet types in different colors and labels them with their type. In each case, the collection is dominated by the `ProcessEdges` work packet, which is rendered in orange. Bucket opening events are marked with green arrows. The first three and the last screenshots were taken at a fixed resolution to show the entire collections. Perfetto also allows the user to easily zoom in and out (5(d)), inspecting the trace at different levels of detail.

In the case of lusearch_bug, the problem is very different. Zooming in with Perfetto to reveal more detail (Figure 5(d)), we find that the three threads that continue working for a long time (thread IDs 3832035, 3832036, and 3832040) are each busy processing a long series of very short packets. Furthermore, by using the `process_edges` tracepoint, we can annotate each of these short packets with the number edges, which is 25 in this case. Finally, we see that this same pattern occurs at nearly every garbage collection. We hypothesized that this is due to the collector traversing long linked list data structures, which are inherently non-parallelizable [5].

Having used the GC tracing framework to narrow the problem down to this point, we modified the MMTk source code to dump the types of the objects in each packet of size 25. This immediately revealed the problem. The Java class `jdk.internal.ref.PhantomCleanable` uses a doubly linked list and extends `PhantomReference`. This linked list is traversed at every collection, and the traversing is inherently unparallelizable [5]. Each node in the linked list carries the same data structure, which happens to contain exactly 25 edges, which is why each packet was 25 items in size. The remaining question was why this problem showed up in lusearch_bug only. The print out of the objects in each packet answered this question too. Each `PhantomCleanable` corresponded to one of lusearch_bug's 2048 input query files. We discovered that the benchmark was not closing the query files after use, which meant they led to a very long list. We verified this by fixing the benchmark. After this fix, we discovered another less dramatic instance of the same problem. Benchmark checksum files used by the suite before starting any benchmark were not being closed. We fixed this too and confirmed that the collector scalability problem was solved. The impact of the fix is clear in the Perfetto screenshot in Figure 5(e) and in Figure 4.

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

Together these various perspectives on collector behavior provided by the tracepoints give the developer a wealth of information, allowing them to conduct performance debugging at a very low cost. Traditional performance debugging tools and expert knowledge remain important, but the process is dramatically faster. We will next consider a case study that uses tracepoints to understand mutator behavior.

6.2 Allocator Scalability

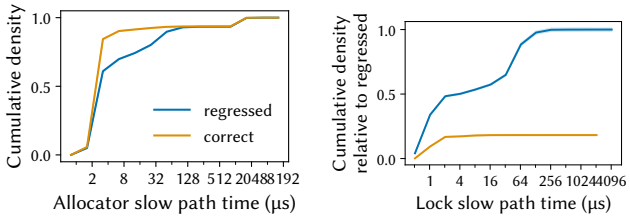
The context for this case study is a scalability regression in MMTk that was not immediately detected. It was only evident with the lusearch workload when running on eight or more cores [26]. The problem was debugged and fixed before this work, so this case study is a retrospective analysis.

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

Methodology. We use the `usdt` tracepoint that we added to MMTk's allocator slow path (Section 3), allowing us to observe the time spent in every execution of the allocator slow path. We also use eBPF's `uprobe` capability, which allows us to monitor the entries and exits of any function without any source code modification (Section 2.2). By attaching `uprobes` to the `Mutex::lock_contended` function symbol exported by the Rust standard library, we are able to precisely measure the time spent in waiting for locks, amplifying any allocator slow path regression caused by lock contention.

Results and Discussion. Figure 6(a) shows a cumulative distribution function plotting the distribution of allocator slow path execution times as a function of their duration for the correct code (orange) and the regressed code (blue). The regressed code is much less likely to have a short duration ($\leq 2\,\mu s$) slow path. This result quickly confirms the initial hypothesis that the allocator slow path may have been the source of poor scalability. It is then a matter of identifying changes that affected the allocator slow path within the window of when the regression was observed. In this case, the problem was an unintended coarsening of the scope of a lock used to acquire pages from a global resource. We attached a `uprobe/uretprobe` pair to the `Mutex::lock_contended` function. Figure 6(b) shows that the bug fix made a dramatic difference to the time spent on the lock slow path, with almost all cases taking less than $4\,\mu s$. Furthermore, the graph shows that the slow path was taken one fifth as frequently, indicating the lock was far less likely to be contended.

Tracepoints' ability to allow rapid confirmation of such a hypothesis is valuable. Without precompiled tracepoints,



(a) The allocation slowpath tracepoint reveals the problem.

(b) Tracing lock contention amplifies the problem further.

Figure 6. A regression in allocator slow path performance on lusearch visualized as a cumulative density function. The top graph compares the allocator slow path time (x-axis, log) of the correct code (orange) and that of the regressed code (blue). We would expect two cases for the allocator slow path time, the common case (2 µs) and the uncommon case (upper right) due to garbage collection pauses, etc. In the regressed case (blue), about 30 % of slow paths are taking 3–100 µs, which are suspicious. The bottom graph amplifies the problem by tracing the time spent in the contended state.

each such hypothesis might have required ad hoc modifications to the source code and a fresh build before any measurement and analysis could be done. The workflow available via eBPF allows this kind of analysis to be conducted systematically and cheaply, even on a running binary.

7 Discussion

In this paper, we show the power of tracing and how modular software engineering amplifies this power to deliver rich observability at low overhead. We use eBPF tools as an example, but one can choose other tracing frameworks where more appropriate. We also use MMTk, but the approach can be applied to other garbage collectors or collector frameworks. The principal advantage of MMTk here is that the software engineering of the toolkit plays very well into our goal of generality and observability. This allows a few well-placed tracepoints to provide rich, collector-neutral observability.

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

Another important consideration is tracing across the operating system stack. If one wants to correlate kernel space events, such as physical memory allocation or context switches, eBPF is likely a good choice since there's no privilege mode switch to handle kernel events since the tracing logic is already running inside the kernel. However, if one wants to purely trace userspace events, and preserve the raw events for offline analysis, Perfetto or LTTng might be a better choice. This is due to the shared memory buffer between userspace processes not requiring a privilege mode switch, and their more efficient on-disk representation of the data than, say, the plain text used by bpftrace. These choices of tracing technology are of practical importance but are largely orthogonal to the key message of this paper, which is that tracing combined with modular software design provides low cost, highly general observability.

8 Conclusion

We have demonstrated that performance tracing using tools like eBPF adds very low overhead observability to garbage collectors, and moreover, when combined with a modular collector implementation, can provide very rich observability with the addition of just a few lines of code. We used two case studies to show the power of this combination when applied first to a collector performance bug and then to an allocator performance bug. We demonstrated that a high degree of observability can be provided with just 14 lines of code added to the collector, and that the run time overhead is negligible—lower than our measurement noise.

We make the case that this approach to performance debugging of collectors is more powerful and more flexible than the dominant state of the art: GC logging and tracing implementations that are less general than what we present here.

Acknowledgments

We thank Xi Yang and the anonymous reviewers for their detailed feedback and insightful suggestions for improving the paper. This material is based upon work supported by the Australian Research Council under Grant No. DP190103367. Zixian Cai is supported by an Australian Government Research Training Program Scholarship.

References

[1] Android Open Source Project. 2017. Overview of system tracing | Android Developers. https://developer.android.com/topic/performance/tracing. http://web.archive.org/web/20221221134831/https://developer.android.com/topic/performance/tracing Accessed: 2022-12-21.

[2] Android Open Source Project. 2017. Perfetto. https://perfetto.dev/. http://web.archive.org/web/20230216013847/https://perfetto.dev/ Accessed: 2023-02-16.

[3] Apple Inc. 2022. GCGC : Garbage Collection Graph Collector. https://github.com/apple/GCGC. http://web.archive.org/web/20230102224957/https://github.com/apple/GCGC Accessed: 2023-01-02.

[4] Matthew Arnold and David Grove. 2005. Collecting and Exploiting High-Accuracy Call Graph Profiles in Virtual Machines. In *3nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO 2005), 20-23 March 2005, San Jose, CA, USA*. IEEE Computer Society, 51–62. https://doi.org/10.1109/CGO.2005.9

[5] Katherine Barabash and Erez Petrank. 2010. Tracing garbage collection on highly parallel platforms. In *Proceedings of the 9th International Symposium on Memory Management, ISMM 2010, Toronto, Ontario,*

Canada, June 5-6, 2010, Jan Vitek and Doug Lea (Eds.). ACM, 1–10. https://doi.org/10.1145/1806651.1806653

[6] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Myths and realities: the performance impact of garbage collection. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems, SIGMETRICS 2004, June 10-14, 2004, New York, NY, USA*, Edward G. Coffman Jr., Zhen Liu, and Arif Merchant (Eds.). ACM, 25–36. https://doi.org/10.1145/1005686.1005693

[7] Stephen M. Blackburn, Perry Cheng, and Kathryn S. McKinley. 2004. Oil and Water? High Performance Garbage Collection in Java with MMTk. In *26th International Conference on Software Engineering (ICSE 2004), 23-28 May 2004, Edinburgh, United Kingdom*, Anthony Finkelstein, Jacky Estublier, and David S. Rosenblum (Eds.). IEEE Computer Society, 137–146. https://doi.org/10.1109/ICSE.2004.1317436

[8] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2006, October 22-26, 2006, Portland, Oregon, USA*, Peri L. Tarr and William R. Cook (Eds.). ACM, 169–190. https://doi.org/10.1145/1167473.1167488

[9] Stephen M. Blackburn and Antony L. Hosking. 2004. Barriers: friend or foe?. In *Proceedings of the 4th International Symposium on Memory Management, ISMM 2004, Vancouver, BC, Canada, October 24-25, 2004*, David F. Bacon and Amer Diwan (Eds.). ACM, 143–151. https://doi.org/10.1145/1029873.1029891

[10] Stephen M. Blackburn and Kathryn S. McKinley. 2008. Immix: a mark-region garbage collector with space efficiency, fast collection, and mutator performance. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, Rajiv Gupta and Saman P. Amarasinghe (Eds.). ACM, 22–32. https://doi.org/10.1145/1375581.1375586

[11] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2008. Wake up and smell the coffee: evaluation methodology for the 21st century. *Commun. ACM* 51, 8 (2008), 83–89. https://doi.org/10.1145/1378704.1378723

[12] Zixian Cai, Stephen M. Blackburn, Michael D. Bond, and Martin Maas. 2022. Distilling the Real Cost of Production Garbage Collectors. In *International IEEE Symposium on Performance Analysis of Systems and Software, ISPASS 2022, Singapore, May 22-24, 2022*. IEEE, 46–57. https://doi.org/10.1109/ISPASS55109.2022.00005

[13] Intel Corporation. 2023. Fix Performance Bottlenecks with Intel® VTune™ Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html. http://web.archive.org/web/20230623012318/https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html Accessed: 2023-06-23.

[14] Intel Corporation. 2023. Intel® VTune™ Profiler User Guide: Sampling Interval. https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-1/sampling-interval.html. http://web.archive.org/web/20230627040259/https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-1/sampling-interval.html Accessed: 2023-06-27.

[15] Microsoft Corporation. 2022. Threads view in the Concurrency Visualizer. https://learn.microsoft.com/en-us/visualstudio/profiling/threads-view-parallel-performance?view=vs-2022. http://web.archive.org/web/20230601064429/https://learn.microsoft.com/en-us/visualstudio/profiling/threads-view-parallel-performance?view=vs-2022 Accessed: 2023-06-01.

[16] DaCapo Developers. 2022. DaCapo benchmarks evaluation snapshot 6e411f33. https://github.com/dacapobench/dacapobench/commit/6e411f33cd8d19f9928eb36c10ceeb868d9b6b23. https://github.com/dacapobench/dacapobench/commit/6e411f33cd8d19f9928eb36c10ceeb868d9b6b23 Accessed: 2022-11-25.

[17] Mathieu Desnoyers and Michel Dagenais. 2008. LTTng: Tracing across execution layers, from the hypervisor to user-space. In *Proceedings of the Linux Symposium—Volume One, Ottawa, Ontario, Canada, July 23-26, 2008*. 101–106. http://web.archive.org/web/20221227045309/https://www.kernel.org/doc/ols/2008/ols2008v1-pages-101-106.pdf

[18] eBPF Foundation. 2021. Extended Berkeley Packet Filter. https://ebpf.io/. https://ebpf.io/ Accessed: 2022-11-25.

[19] Google. 2016. Trace Event Format. https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview. http://web.archive.org/web/20230127150401/https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview Accessed: 2023-01-27.

[20] Brendan Gregg. 2019. *BPF Performance Tools*. Pearson Education, Philadelphia, PA.

[21] Antony L. Hosking, J. Eliot B. Moss, and Darko Stefanovic. 1992. A Comparative Performance Evaluation of Write Barrier Implementations. In *Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'92), Seventh Annual Conference, Vancouver, British Columbia, Canada, October 18-22, 1992, Proceedings*, John R. Pugh (Ed.). ACM, 92–109. https://doi.org/10.1145/141936.141946

[22] Hound Technology, Inc. 2018. honeycomb.io Guide: Achieving Observability. https://www.honeycomb.io/wp-content/uploads/2018/07/Honeycomb-Guide-Achieving-Observability-v1.pdf. http://web.archive.org/web/20220903112906/https://www.honeycomb.io/wp-content/uploads/2018/07/Honeycomb-Guide-Achieving-Observability-v1.pdf Accessed: 2022-09-03.

[23] IO Visor Project. 2015. BCC. https://github.com/iovisor/bcc. https://github.com/iovisor/bcc Accessed: 2022-11-25.

[24] IO Visor Project. 2018. bpftrace. https://github.com/iovisor/bpftrace. https://github.com/iovisor/bpftrace Accessed: 2022-11-25.

[25] Julia Developers. 2011. Instrumenting Julia with DTrace, and bpftrace. https://docs.julialang.org/en/v1/devdocs/probes/. http://web.archive.org/web/20230120124737/https://docs.julialang.org/en/v1/devdocs/probes/ Accessed: 2023-01-20.

[26] Yi Lin. 2022. `mmtk-core` Pull Request #614: Minimize the scope of `acquire_lock`. https://github.com/mmtk/mmtk-core/pull/614. https://github.com/mmtk/mmtk-core/pull/614 Accessed: 2022-11-25.

[27] Dan Luu. 2016. Sampling v. Tracing. https://danluu.com/perf-tracing/ Accessed: 2023-05-23.

[28] David Malcolm and Łukasz Langa. 2016. Instrumenting CPython with DTrace and SystemTap. https://docs.python.org/3/howto/instrumentation.html. http://web.archive.org/web/20230227184647/https://docs.python.org/3/howto/instrumentation.html Accessed: 2023-02-27.

[29] Microsoft Corporation. 2016. PerfView and TraceEvent. https://github.com/microsoft/perfview/blob/main/src/TraceEvent/Computers/TraceManagedProcess.cs. http://web.archive.org/web/20221205191156/https://github.com/microsoft/perfview/blob/main/src/TraceEvent/Computers/TraceManagedProcess.cs Accessed: 2022-12-05.

[30] Microsoft Corportaion. 2021. Microsoft GCToolKit. https://devblogs.microsoft.com/java/introducing-microsoft-gctoolkit/. http://web.archive.org/web/20221003152322/https://devblogs.microsoft.com/java/introducing-microsoft-gctoolkit/ Accessed:

2022-10-03.

[31] MMTk Developers. 2020. MMTk: A high performance language-independent memory management framework. https://www.mmtk.io/. https://www.mmtk.io/ Accessed: 2022-11-25.

[32] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. 2010. Evaluating the accuracy of Java profilers. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, Benjamin G. Zorn and Alexander Aiken (Eds.). ACM, 187–197. https://doi.org/10.1145/1806596.1806618

[33] OProfile Developers. 2023. About OProfile. https://oprofile.sourceforge.io/about/. https://web.archive.org/web/20230629065218/https://oprofile.sourceforge.io/about/ Accessed: 2023-06-29.

[34] Oracle Corporation and OpenJDK Developers. 2013. JDK Mission Control. www.oracle.com/java/technologies/jdk-mission-control.html. http://web.archive.org/web/20230219111918/https://www.oracle.com/java/technologies/jdk-mission-control.html Accessed: 2023-02-19.

[35] Vara Prasad, William Cohen, Frank Ch. Eigler, Martin Hunt, Jim Keniston, and Chen Brad. 2005. Locating System Problems Using Dynamic Instrumentation. In *Proceedings of the Linux Symposium—Volume Two, Ottawa, Ontario, Canada, July 20-23, 2005*. 49–64. http://web.archive.org/web/20220308130912/https://www.kernel.org/doc/ols/2005/ols2005v2-pages-57-72.pdf

[36] Ruby Developers. 2013. DTrace Probes - RDoc Documentation. https://ruby-doc.org/3.2.1/dtrace_probes_rdoc.html. http://web.archive.org/web/20230303115753/https://ruby-doc.org/3.2.1/dtrace_probes_rdoc.html Accessed: 2023-03-03.

[37] Richard L. Sites. 2015. Data Center Computers: Modern Challenges in CPU Design. https://www.youtube.com/watch?v=QBu2Ae8-8LM Accessed: 2023-05-23.

[38] Richard L. Sites. 2021. *Understanding Software Dynamics*. Addison-Wesley.

[39] Richard L. Sites. 2023. KUtrace. https://github.com/dicksites/kutrace Accessed: 2023-05-23.

[40] Maoni Stephens. 2021. GCRealTimeMon. https://github.com/Maoni0/realmon. http://web.archive.org/web/20221205193717/https://github.com/maoni0/realmon Accessed: 2022-12-05.

[41] Josh Stone. 2014. libprobe: Static probes for Rust. https://crates.io/crates/probe. http://web.archive.org/web/20230303073513/https://crates.io/crates/probe Accessed: 2023-03-03.

[42] Sun Microsystems. 2006. DTrace User Guide. https://docs.oracle.com/cd/E19253-01/819-5488/819-5488.pdf. http://web.archive.org/web/20230303065912/https://docs.oracle.com/cd/E19253-01/819-5488/819-5488.pdf Accessed: 2023-03-03.

[43] Tier1App. 2022. GCeasy. https://gceasy.io/. http://web.archive.org/web/20230227155647/https://gceasy.io/ Accessed: 2023-02-27.

[44] V8 Developers. 2018. Tracing V8. https://v8.dev/docs/trace. http://web.archive.org/web/20230216012529/https://v8.dev/docs/trace Accessed: 2023-02-16.

[45] Bochen Xu, Eliot Moss, and Stephen M. Blackburn. 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 2022, Brussels, Belgium, September 14-15, 2022*, Elisa Gonzalez Boix and Tobias Wrigstad (Eds.). ACM, 128–139. https://doi.org/10.1145/3546918.3546923

[46] Xi Yang, Stephen M. Blackburn, Daniel Frampton, and Antony L. Hosking. 2012. Barriers reconsidered, friendlier still!. In *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, Martin T. Vechev and Kathryn S. McKinley (Eds.). ACM, 37–48. https://doi.org/10.1145/2258996.2259004

[47] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. 2015. Computer performance microscopy with Shim. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, Deborah T. Marr and David H. Albonesi (Eds.). ACM, 170–184. https://doi.org/10.1145/2749469.2750401

[48] Wenyu Zhao and Stephen M. Blackburn. 2020. Deconstructing the garbage-first collector. In *VEE '20: 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, virtual event [Lausanne, Switzerland], March 17, 2020*, Santosh Nagarakatte, Andrew Baumann, and Baris Kasikci (Eds.). ACM, 15–29. https://doi.org/10.1145/3381052.3381320

[49] Benjamin G. Zorn. 1990. *Barrier Methods for Garbage Collection*. Technical Report. University of Colorado Boulder. https://scholar.colorado.edu/concern/reports/47429970d

[50] Benjamin G. Zorn. 1993. The Measured Cost of Conservative Garbage Collection. *Softw. Pract. Exp.* 23, 7 (1993), 733–756. https://doi.org/10.1002/spe.4380230704

Received 2023-06-29; accepted 2023-07-31