Work Packets: A New Abstraction for GC Software Engineering, Optimization, and Innovation

WENYU ZHAO, Australian National University, Australia
STEPHEN M. BLACKBURN, Google and Australian National University, Australia
KATHRYN S. MCKINLEY, Google, United States

Garbage collection (GC) implementations must meet efficiency and maintainability requirements, which are often perceived to be at odds. Moreover, the desire for efficiency typically sacrifices agility, undermining rapid development and innovation, with unintended consequences on longer-term performance aspirations. Prior GC implementations struggle to: i) maximize efficiency, parallelism, and hardware utilization, while ii) correctly and elegantly implementing optimizations and scheduling constraints. This struggle is reflected in today's implementations, which tend to be monolithic and depend on coarse phase-based synchronization.

This paper presents a new design for GC implementations that emphasizes both agility and efficiency. The design simplifies and unifies all GC tasks into *work packets* which define: i) work items, ii) kernels that process them, and iii) scheduling constraints. Our simple insights are that execution is dominated by a few very small, heavily executed kernels, and that GC implementations are high-level algorithms that orchestrate vast numbers of performance-critical work items. Work packets comprise groups of like work items, such as the scanning of a thread's stack or the tracing of a single object in a multi-million object heap. The kernel attached to a packet specifies how to process items within the packet, such as how to scan a stack, or how to trace an object. The scheduling constraints express dependencies, e.g. all mutators must stop before copying any objects. Fully parallel activities, such as scanning roots and performing a transitive closure, proceed with little synchronization. The implementation of a GC algorithm reduces to declaring required work packets, their kernels, and dependencies. The execution model operates transparently of GC algorithms and work packet type. We broaden the scope of work-stealing, applying it to any type of GC work and introduce a novel two-tier work-stealing algorithm to further optimize parallelism at fine granularity.

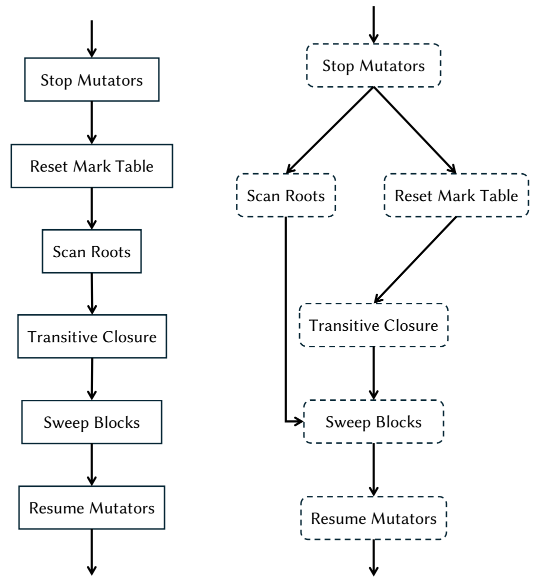
We show the software engineering benefits of this design via eight collectors that use 23 common work packet types in the MMTk GC framework. We use the LXR collector to show that the work packet abstraction supports innovation and high performance: i) comparing versions of LXR, work packets deliver performance benefits over a phase-based approach, and ii) LXR with work packets outperforms the highly-tuned latest (OpenJDK 24), state-of-the-art G1 garbage collector. We thus demonstrate that work packets achieve high performance, while simplifying GC implementation, making them inherently easier to optimize and verify.

CCS Concepts: • **Software and its engineering → Garbage collection**; **Software performance**; **Abstraction, modeling and modularity**.

Additional Key Words and Phrases: Garbage Collection, Software Engineering, Performance, Parallelism

ACM Reference Format:
Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2025. Work Packets: A New Abstraction for GC Software Engineering, Optimization, and Innovation. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 361 (October 2025), 29 pages. https://doi.org/10.1145/3763139

ACM 2475-1421/2025/10-ART361
https://doi.org/10.1145/3763139



```
# GC workers main execution loop
def thread_main():
    loop:
        # Fetch a packet
        loop:
            p = get_or_steal_packet()
            if p != null:
                break
            sleep()
        # Execute the packet
        p.execute()
        # Update the bucket
        b = get_bucket(p)
        b.remaining_packets -= 1
        if b.remaining_packets == 0:
            bucket_is_empty(b)

def bucket_is_empty(bucket: Bucket):
    # Try to activate child buckets
    for s in bucket.successors:
        if s.all_predecessors_empty():
            # Activate the bucket
            succ.activate()
```

(a) Immix with phases (b) Immix with work-buckets (c) Work packet scheduling pseudo code

Fig. 1. **Simple phase-based scheduling limits parallelism.** Dependency graphs with (a) traditional phases (in rectangles), (b) work packets (in rounded corners) for Immix, and (c) pseudo code for work packet scheduling. The edges indicate the dependencies. Scan Roots and Reset Mark Table can execute concurrently.

1 Introduction

Garbage collection (GC) algorithms are performance-critical components of managed runtimes. A highly optimized GC implementation is of course essential for achieving high application performance. On the other hand, GC implementations are part of large and complex runtime systems, so agile, maintainable software design and correct implementation are equally critical. Modern GC implementations often fail to meet both requirements. Consider the mature high performance G1 and Parallel GC algorithms in OpenJDK [14]. First, **they address efficiency through distinct, highly-specialized, monolithic implementations**, with carefully optimized hot code paths. Both GCs have their own highly optimized implementation of the transitive closure [33, 36], despite their very similar implementations. Their monolithic designs make it difficult to reuse code and optimizations across different GC algorithms. Second, **they address high-level correctness by using synchronous phases** that ensure temporal ordering across each step of the GC algorithm. Figure 1(a) shows an example of sequential phase ordering using Immix [6] which is indicative of both OpenJDK and MMTk collectors. In a phased approach, all GC worker threads must, for example, finish Transitive Closure before Sweep Blocks to ensure correctness. Synchronization barriers between phases are unavoidable for correctly ordering phases, introducing unnecessary synchronization overhead and suboptimal load balancing. Third, **they make it difficult to innovate and to integrate systematic performance analysis or verification tools** due to their monolithic design and the algorithm-specific phases. Such tools are challenging to share across different GC algorithms and require extra engineering effort to integrate into each algorithm.

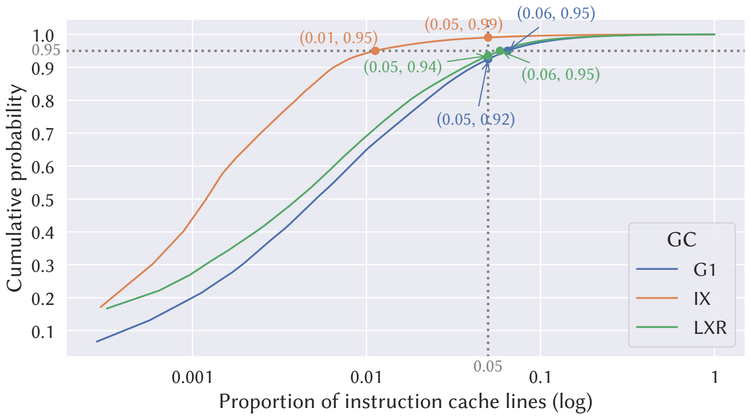


Fig. 2. **92–99% of GC execution is concentrated in 5% of GC code.** Cumulative distribution of execution time for different instructions that GC worker thread execute in G1, Immix, and LXR, measured at instruction cache line (64 B) granularity. The point (0.01, 0.95) on the Immix curve indicates that 95% of samples are in the top 1% of cache lines, i.e., 1% of code accounts for 95% of execution time.

In this paper, we introduce a new work packet design based on two insights.

i) Most performance-critical GC work is concentrated in a few lines of code.
ii) Phases introduce unnecessary synchronization.

Figure 2 shows the evidence for the first insight. We sample instructions executed by GC worker threads for three GC algorithms: G1, Immix, and LXR in OpenJDK [6, 14, 62, 65]. We plot the distribution of instruction hotness at cache line (64 B) granularity. These GCs spend 92–99% of their execution time in 5% of the code. This 5% encompasses core performance-critical GC components such as transitive closure and GC metadata access. GC execution is thus highly concentrated. In addition, key kernels are generic to most GC algorithms, e.g., they inspect GC metadata and perform tracing, suggesting potential for sharing implementations across multiple algorithms.

Our second insight is that the traditional phase ordering does not fully exploit available parallelism. For example, scanning roots does not depend on clearing the mark table and not all roots need to be scanned in order to start the transitive closure, as depicted in Figure 1(b). However, some work does have dependencies. For example, all mutator threads must stop before the GC may move any object (unless the collector supports concurrent copying).

Motivated by these insights, we rethink GC design by distilling performance-critical code into a kernel-centric GC execution model. This design seeks to maximize efficiency and maintainability. We break down algorithm-specific GC tasks and phases into algorithm-independent *work packets*. Each work packet contains a set of fine-grained *work items*, a small *kernel* that processes them, and *dependencies* that determine when to schedule them. For example, a work packet may contain a single mutator stack work item, a stack scanning kernel, and a dependence on mutators being stopped. A scan stack work item may generate many trace object work items. We build an execution model that respects dependencies while maximizing parallelism, as shown in Figure 1(b). For example, by removing the synchronization barrier between Scan Roots and Transitive Closure, a root scanning packet will generate transitive closure work items while it identifies root pointers, and any idle workers will immediately pick the packets up and perform tracing without waiting for all root scanning to complete. GC worker threads are entirely work-agnostic. Whenever idle, they simply pick up any ready work packet and invoke its kernel.

By introducing a unified work packet runtime, we automatically broaden the scope of work-stealing to apply to any type of GC work, not only the transitive closure. In contrast, existing OpenJDK collectors perform work stealing only during the transitive closure. We introduce a two-tier work-stealing algorithm in which worker threads steal coarse- and fine-grained work items from other workers, enhancing parallelism beyond coarse-grained work packet stealing. In the common case, GC workers consume work packets from their local queues. Idle workers may steal either one packet or a single work item from other workers' local queues. The rationale is that neither packets nor work items are always the right granularity for work stealing. When work packets infrequently generate new work packets because the heap shape is such that they consume most of the work that they produce, coarse-grained packet stealing is inefficient. Stealing just one item from the packet in this situation can significantly improve load balancing, but only operating at a fine grain is expensive, hence our two-tier approach.

We implemented our design in two stages. First, we implemented an initial version of the work packet runtime that uses coarse synchronization between phases, and contributed it to the Rust MMTk GC framework [50]. Then, we implemented a series of optimizations including work packet dependencies and two-tier work stealing that expose and exploit more parallelism, which we evaluate here and plan to contribute. We evaluate the software engineering and performance benefits of work packets in this context.

For the software engineering benefit, we show that the work packet design achieves high kernel reuse between algorithms. MMTk implements eight collectors with work packets. All together the collectors instantiate a total of 49 unique work packet types. Seven collectors require zero to three unique work packet types. Only one collector (LXR) uses more, at 15 unique types. LXR is the only reference counting collector, so this result is expected. The eight collectors share 23 or more work packet types, demonstrating many kernels are common and reusable (Table 2).

For the performance evaluation, we have two parts. i) An apples-to-apples comparison of Immix and LXR with three work packet versions: phased based, fine-grain dependencies, and two-tier work stealing in OpenJDK 11, and ii) a comparison of LXR in OpenJDK 11 to G1 in OpenJDK 11, 21, and 24. For part one, we show that both fine-grain dependencies and two-tier work stealing improve both Immix and LXR garbage performance and total performance on the DaCapo Benchmarks. For example, total performance for Immix improves by 6% and stop-the-world time improves by 10% on a tight heap (Table 6). For part one and two, we ideally would also compare apples-to-apples: a single collector implemented both with the work packet runtime and with a traditional monolithic implementation. Unfortunately, re-engineering a highly-optimized monolithic production collector such as G1 to use a modular work packet framework is likely a multi-year effort and thus impractical and out of scope. Likewise, because our initial design was integrated into MMTk in OpenJDK 11 early in its development, such a comparison within MMTk is also out of scope. We therefore compare the best MMTk collector, LXR, to G1, the state-of-the-art monolithic system. LXR is implemented in OpenJDK 11 and we compare to G1 in OpenJDK 11, 21, and 24. At the time of writing, OpenJDK 21 is the latest long term support release and we have not finished porting MMTk to OpenJDK 21. LXR outperforms all versions of G1 by at least 6% in total execution time on a moderate heap size and by at least 40% in GC time. These results demonstrate that a collector built with work packets can outperform the state-of-the-art collector.

In summary, this paper contributes work packets, a new abstraction for GC implementation which achieves state-of-the-art performance while simplifying GC implementation. Work packets achieve high performance via kernels that are easy to optimize and schedule. The abstraction simplifies GC implementation through modularity, creating high code reuse, minimizing algorithm-specificity, facilitating verification and encouraging the exploration of innovative GC designs.

2 Background and Related Work

2.1 Phase-Based GC Design in OpenJDK

OpenJDK is the most widely used Java virtual machine implementation and includes a number of high-performance garbage collectors. The Garbage-First (G1) collector [14] was introduced in JDK 1.6 and subsequently became the default GC in JDK 9. G1 is a region-based generational collector. It divides the heap into fixed-sized regions and, in the common case, performs young GC pauses to collect the newly allocated regions, copying nursery survivors. G1 also performs mixed GC with concurrent marking or full GCs to collect mature regions.

G1 and other OpenJDK GCs organize high-level tasks as a list of sequentially-ordered function calls. Each function call can be considered a standalone phase, launching GC workers to perform a specific high-level task in parallel, and synchronizing before moving to the next step. For example, G1's young GC pause has the following phases: i) root scanning, ii) remembered set scanning, iii) transitive closure-based evacuation, iv) weak reference processing, v) weak processor processing, and vi) optional string deduplication (disabled by default).

Within phases i) to iv), multiple worker threads execute in parallel. Only one thread executes phases v) and vi). Except for the transition from phase ii) to iii), synchronization barriers between each phase ensure sequential ordering [34, 35]. Although this phase-based design simplifies implementing GC algorithms and enforces temporal correctness constraints, it introduces unnecessary barriers and single-threaded phases, hindering parallelism (Section 6). Although OpenJDK native collectors express their phases as clearly separated function calls, this separation is unrelated to code reuse — each phase is specialized and built monolithically, without sharing of common components among collectors. In Section 4, we show that our work packet runtime significantly improves code reuse and maintainability by building most GC components in an algorithm-neutral way and sharing them across multiple GCs.

2.2 MMTk$_J$: Phase-Based Design

MMTk is a memory management toolkit [5] that was originally implemented in JikesRVM [1] and systematically used a phase-based design for all of its garbage collectors. In contrast to OpenJDK, its implementation was highly modular with a major focus on code reuse and its phases were reused across GCs. Listing 1 illustrates a simplified list of phases for JikesRVM's implementation of the Immix GC [6]. Each GC collection is divided into multiple phases. Some phases, such as PREPARE and RELEASE, are global phases that are executed once by a single worker thread (i.e., the primary thread). Other phases are run in parallel by all available worker threads. Similar to OpenJDK, JikesRVM phase execution is sequentially ordered with synchronization barriers between each phase.

Listing 1. **A simplified phase-based execution model for Immix in JikesRVM's MMTk.** Immix pre-defines a list of phases, calling collectionPhase once per phase in order with all GC workers synchronizing before moving to the next phase.

```
void collectionPhase(short phaseId,
                     boolean primary) {
  if (phaseId == PREPARE && primary) {
    immix.prepare();
    return;
  }
  if (phaseId == STACK_ROOTS) {
    VM.scanning.computeThreadRoots();
    return;
  }
  if (phaseId == ROOTS) {
    VM.scanning.computeGlobalRoots();
    VM.scanning.computeStaticRoots();
    VM.scanning.computeBootImageRoots();
    return;
  }
  if (phaseId == CLOSURE) {
    currentTrace.completeTrace();
    return;
  }
  if (phaseId == RELEASE && primary) {
    immix.release();
    return;
  }
}
```

2.3 MMTk$_R$: OpenJDK and Other Runtimes

In 2020, MMTk was rewritten in Rust as a standalone GC library that is now connected to multiple virtual machines, including JikesRVM, OpenJDK, V8, Ruby, and Julia [13, 48, 51, 52, 57]. We refer to this new version as MMTk$_R$, and the JikesRVM version as MMTk$_J$. Our work is undertaken in MMTk$_R$.

We contributed our initial phase-oriented work packet runtime to MMTk$_R$ and all of the MMTk$_R$ collector implementations use it today [60]. Our initial implementation uses phases rather than generalized work packet scheduling and it lacks work stealing. More recently we added: i) fine-grained dependencies, and ii) two-tier work-stealing, which we will contribute to MMTk$_R$. Throughout the remainder of the paper, we consider three variants of MMTk.

i) **MMTk$_P$** implements our **P**hase-based work packets and work-packet stealing. (MMTk$_P$ is identical to MMTk$_R$ with the addition of instrumentation features.)
ii) **MMTk$_D$** implements fine-grained **D**ependencies on top of MMTk$_P$.
iii) **MMTk$_S$** implements both fine-grained dependencies and two-tier work-**S**tealing.

The contributions described in this paper comprise the entire design and implementation of the work packet runtime, including the implementation already upstreamed to MMTk.

2.4 OpenJDK 11 and Later Versions

Although a port of MMTk to OpenJDK 21 is underway, MMTk's OpenJDK binding currently only supports OpenJDK 11. Therefore, we implement our systems using OpenJDK 11 and use both MMTk$_P$ and G1 in OpenJDK 11 as baselines for our evaluation. At the time of writing, OpenJDK 21 is the latest long-term support (LTS) release, while OpenJDK 24 is the most recent version. In Section 6.2, we show that when using G1, OpenJDK 21 improves over OpenJDK 11 in terms of throughput by 5.1% and 3.2% with heap sizes of 1.5× and 2× the minimum required, respectively.

2.5 Immix and LXR

We use Immix and LXR, two high performance GCs, to demonstrate the work packet runtime [6, 62]. Immix is a stop-the-world full-heap tracing collector that uses a two-tier mark-region hierarchical heap structure, consisting of blocks, divided into lines, to manage memory [6]. The Immix implementation in MMTk$_P$ retains a phase sequence similar to the original implementation in MMTk$_J$, as shown in Figure 1(a), but it implements work packets within each phase.

LXR is a state-of-the-art garbage collector implemented in MMTk$_R$ [62, 65]. It leverages reference counting with backup concurrent marking to achieve high throughput and low request latency. Like other collectors in MMTk$_R$, LXR is built on the work packet runtime, organizing tasks as work packets and distributing them as phases. LXR shares 26 work packet types with other collectors in MMTk$_R$, such as the root scanning and weak reference processing packets (see Section 4). Work packets implement mechanisms but are algorithm-oblivious, with the 26 common packets shared by almost all collectors in MMTk$_R$. Any GC that requires these work packets types can directly reuse and instantiate them without modification. In this work, we port Immix and LXR to our new work packet implementations, MMTk$_D$ and MMTk$_S$, and evaluate their performance.

2.6 Work Stealing

Work stealing [7] is a technique for scheduling tasks efficiently across multiple worker threads by allowing idle workers to 'steal' work from busier workers, maximizing CPU utilization and parallelism. This technique is widely implemented in several language runtimes, including Cilk [19] and X10 [9], and is adopted in application-level parallelism libraries such as Intel oneTBB [45]. Our algorithm is implemented as part of the MMTk framework.

Garbage collectors often use work stealing to enhance load balancing during tracing. It allows idle worker threads to steal objects from the mark stacks of other workers [38–40], improving overall CPU utilization, especially when tracing irregular heap shapes. OpenJDK uses a modified ABP-style deque (double-ended queue) [2]. Each GC thread maintains a local deque as the local mark queue for the transitive closure. The deque has a both public end for stealing and a private end for the owner, effectively reducing contention between the stealers and the victim. In the common case, workers push and pop items from the deque's private end during heap traversal. When the deque is empty, workers steal items from other workers' public ends. The ABP-style deque uses fixed-sized arrays, limiting the number of stored items. The Chase-Lev algorithm [10] solves this problem by dynamically resizing the deque, but it introduces contention overhead due to an atomic buffer resizing operation. Instead, we incorporate a resizable private stack for overflow items. The private stack is not visible to other workers, thus operations on it do not require synchronization. This design avoids the contention cost associated with frequent stack operations but limits available items for stealing. This design is also used by OpenJDK's native collectors [37].

Work Stealing in ZGC. ZGC performs work stealing differently. Instead of using a work-stealing deque and stealing items one at a time, ZGC maintains each local mark stack as a set of mark stack segments. By default, one segment can hold up to two thousand object references. Starved workers steal an entire segment at a time from other workers' local mark stacks instead of just one item [40]. This design reduces contention overhead associated with frequent stealing operations, though its coarse stealing granularity may lead to suboptimal load balancing.

Work Stealing in MMTk$_R$. Similar to ZGC, the work packet design in MMTk$_R$ and MMTk$_P$ implements a stack of work items in each transitive closure work packet. Idle workers steal one packet of any type from other workers, not limited to transitive closure. We thus broaden the scope of work stealing by exposing parallelism across all parts of the GC. We further enhance this design by introducing a two-tier work-stealing algorithm. Workers may steal either a coarse-grained work packet or a single work *item* to maximize load balancing. Section 3.3 presents our approach.

2.7 Other Related Work

Our GC task scheduling system manages a set of fine-grained parallel tasks across multiple workers. The system maintains dependencies between tasks to ensure correctness, while also bulk processing tasks to improve throughput. Ossia et al. [42] proposed a GC algorithm that uses a similar scheduling approach, grouping heap pointers into packets to improve load balancing during marking. In contrast, our design goes beyond partitioning of work to provide a general abstraction for GC implementation. As such, it is algorithm-agnostic and applies to the entire GC process, not just marking, and it associates a kernel and dependencies with each packet. Our abstraction can support any GC algorithm and enhances software engineering by improving code reuse and maintainability.

Our design adheres to data-oriented design principles. Other systems use a similar design pattern. For example, the Vector Packet Processor [54] batch processes network packets of the same type to improve throughput and cache locality. Similarly, the soft update algorithm in file systems [20] reduces synchronous updates by tracking fine-grained job dependencies. Different from these systems, our work-packet scheduler must handle the generation of new work items and packets.

Also related to work packets is Miao et al.'s execution model for stream processing [29]. They group stream records as *cascading containers*, and schedule them in parallel under the constraints of local container dependencies. Although we have similar multi-level task granularity, our design has a completely different execution model which necessitates a different dependence tracking mechanism (called work buckets) to ensure correct task ordering. We also introduced a novel multi-granularity work stealing algorithm to further improve load balancing.

3 Design

Our implementation consists of the following components. **Work packets** encapsulate work items, the kernels that process them, and scheduling dependencies. **Work buckets** manage packet dependencies and execution order without explicit synchronization. **A simple runtime** manages scheduling and parallel execution. **Tools** analyze work packet performance and perform verification.

We use work packets to execute all GC tasks, including mutator yielding, root scanning, remembered set processing, sweeping, and more. One or more work-agnostic worker threads executes all GC work. They loop indefinitely acquiring and processing available work packets. With this structure, we instantiate the entire GC algorithm as a collection of work packets and their execution order constraints. Concretely, MMTk uses the work packet runtime to implement eight collectors that have a mixture of stop-the-world and concurrent execution elements (Table 2).

We next discuss the complete design that covers all four components, and explain how they are integrated to create a GC implementation that is both efficient and general.

3.1 Work Packets

A typical GC collection epoch may process many millions of work items. Coarse grained work items such as stack scanning may take microseconds, while fine grained work items like marking an object take just nanoseconds. Efficiently scheduling a sea of heterogenous work items is challenging. To address this challenge, we introduce work packets as the default unit of scheduling. Each work packet contains one or more work items, with packets sized large enough to garner locality and amortize fixed costs, but not so large as to inhibit load balancing. We design work packets to include the processing kernel for their work items, such that our GC workers can be oblivious to work-packet type, allowing them to process any ready work packet.

3.1.1 Work Items. We break down all work required by a GC into units called *work items*. Each item is the smallest unit of work of a given type that can be processed by a single thread. An example of a large work item is scanning one mutator stack. An example of a small work item is marking the reference to one heap object. By abstracting the concept of work items and generalizing it to all types of GC work, we more easily transfer optimizations between GC components. For example, work stealing generalizes to any work type and is not limited to objects in the transitive closure (see Section 3.3).

3.1.2 Kernels. Each work packet has a *kernel* for processing its work items. A kernel is a fully general Rust function, with an inner loop that, for example, might be just a few lines of code that mark an object, or a call to a runtime-specific C++ implementation of stack-scanning. Each kernel processes work items and may generate new work items of the same or different type. For instance, a kernel for stack scanning consumes a mutator stack, scans it, and produces work items for the transitive closure. This kernel's responsibility is limited to root scanning. It generates work items of another type, which will ensure that all of the pointers it discovers are processed. By limiting the scope of the kernel strictly to processing its homogenous work items, we keep the kernel as small as possible. For example, the transitive closure's hot path (L17-24 in Figure 3(a)) contains seven lines of highly optimized code. This design encourages developers to write short, concentrated code leading to implementations that are easy to write, maintain, and optimize.

Because kernels are just functions that operate over work items, they are very flexible. For example, one can simply replace the `Stack` data structure (L2 in Figure 3) with a FIFO queue to process work items in breadth-first order instead of depth-first.

3.1.3 Dependencies. We implement work packet dependencies by associating work packets with work buckets (Section 3.2). A work bucket captures all packets that share the same dependencies.

```
 1  class TransitiveClosurePacket:
 2    stack: Stack[Address]
 3
 4    # Create a new packet on overflow
 5    def flush_half_stack():
 6      half_stack = split_stack(stack)
 7      spawn TransitiveClosurePacket(
 8        half_stack
 9      )
10
11    # Push a newly discovered item to the
12    # private stack
13    def push_item(slot: Address):
14      stack.push(slot)
15
16    # Mark the object and scan fields
17    def process_item(slot: Address):
18      o = slot.load_object()
19      if o.is_null(): return
20
21      if o.attempt_mark():
22        for f in scan_fields(o):
23          push_item(f)
24          if overflow:
25            flush_half_stack()
26
27    def execute():
28      # Drain the private stack
29      while !stack.empty():
30        slot = stack.pop()
31        process_item(slot)
```

(a) *Without* two-tier work stealing

```
 1  class TransitiveClosurePacket:
 2    stack: Stack[Address]
 3    ...
 4
 5    # Push to work-stealing deque first to
 6    # make them visible to other threads
 7    def push_item(slot: Address):
 8      if !TLS.item_deque.push(slot):
 9        stack.push(slot)
10
11    def execute():
12      while !stack.empty() and !TLS.
             item_deque.empty():
13        # Drain both the stack
14        # and TLS deque
15        while !stack.empty() and \
              !TLS.item_deque.empty():
16
17          s = pop_from_stack_or_tls()
18          process_item(s)
19        if there are pending packets:
20          # Finish this packet
21          # so the worker
22          # will work on other packets
23          return
24        # No available packets.
25        # Try to steal items.
26        try_steal_items()
27
28  class ThreadLocalStorage:
29    # Work stealing deque with fixed size
30    item_deque: CappedDeque[Address]
```

(b) *With* two-tier work stealing

Fig. 3. **Transitive closure work packet implementation with and without two-tier work stealing.** The packet contains a stack of heap or root addresses to trace, adding any discovered fields during tracing. Figure 3(b) shows two-tier work stealing implementation for fine-grained load balancing by placing a small fixed-sized deque in front of the packet-private stack.

Only once a bucket's dependencies are satisfied and the runtime marks it as *active* can GC worker threads commence executing the work packets it contains.

This work packet design has a number of performance benefits. i) The kernel is highly iterative, *enabling the inlining and specialization of hot code*, reducing overheads such as dynamic dispatch. ii) Processing multiple items of the same type *enhances cache locality*. iii) Each worker thread polls one packet at a time for execution, effectively *reducing the overhead of thread synchronization*.

Figure 3(a) illustrates the pseudo code for the transitive closure work packet. The work packet comprises a private stack that stores its work items — addresses of objects to trace (L 2). The kernel operates in a tight loop, draining the stack and tracing each address (L 28–30). During tracing, the kernel may identify new work in the form of additional fields to trace. When the new work is the same work packet type, the kernel pushes them onto its own stack (L 22). Allowing the stack to grow indefinitely, however, hinders load balancing. We thus flush the stack periodically when either: i) the stack size exceeds its capacity, or ii) the number of push operations surpasses a threshold (implemented within the Stack, not shown). When these conditions are satisfied (L 23), we flush half of the stack to a new packet, making the work immediately available to other threads (L 5–9). We chose a stack capacity of 1024 items, and a threshold of 512 push operations as these values provided the best performance across a range of workloads. Future work could explore dynamically adjusting these values based on run-time statistics.

The work packet design simplifies developing and debugging GC algorithms as follows.

Simplified implementation. All GC work is encapsulated within work packets, making the development of new GC algorithms easier. The upstreamed MMTk_R successfully demonstrates that work packets simplified and accelerated the prototyping and development of advanced algorithms such as Iso [44] and LXR [62], which both outperform the state-of-the-art.

Small and focused kernels simplify optimization. Each work packet frequently executes its corresponding kernel inlined within a hot loop, enabling developers to focus on optimizing the critical path rather than tracking performance issues across the entire codebase. As an example, Table 1 shows the execution time breakdown of the dominant work packet types for the Immix GC. Over 90% of execution time is spent in transitive closure work packets, naturally capturing the most focused and performance-critical GC code. This characteristic easily identifies critical code for optimization.

Highly reusable code. Breaking down algorithms into small kernels enhances GC code reuse. For example, all MMTk_R GC algorithms share the same root scanning work packets with only minor modifications, rather than each implementing its own version in specialized phases as in most of the OpenJDK collectors [30–32, 41].[1] Consequently, exploring a new GC algorithm becomes simpler and less error-prone.

3.2 Work Buckets and Scheduling

We use work buckets to schedule work packets onto work-agnostic GC worker threads. This design gives the scheduler the liberty to flexibly schedule any kind of work to any worker. We implement dependencies at the granularity of work packet type rather than through more rigid phases, giving the scheduler further flexibly. By contrast, traditional implementations' use of sequential phases over-constrains scheduling and limits parallelism.

Table 1. **Breakdown of Immix work packet types by relative execution time**. Immix is a full-heap algorithm, tracing the entire heap at every collection. The dominance of transitive closure is therefore unsurprising.

Work Packet Type	Time (%)
Sweep Blocks	1.1
Reset Mark Table	1.2
Process Weak Refs	1.8
Resurrect Finalizables	2.7
Transitive Closure	90.7
All Other Packet Types	*2.6*

Parallel Execution Without Fences. The scheduler dispatches work packets in an implementation-agnostic approach, interleaving the execution of different types of work packets without needing fences or thread synchronization. We maintain a global shared queue for all ready to execute work packets, mixing all packet types. For example, work packets that clear the mark table and scan stacks are ready to execute at the same time (Figure 1(b)) and may thus interleave. The scheduler manages a pool of parallel workers that: i) avoid explicit synchronization, ii) pop work packets from the global queue and execute them, and iii) enter a sleep state when the queue is empty. To maximize CPU utilization, worker threads may steal work packets and more fine-grained work items from others (see Section 3.3).

[1]Earlier OpenJDK GCs, such as Serial, Parallel, and CMS, share the same root scanning implementation. However, later GCs such as G1, Shenandoah, and ZGC, duplicate this code even though much of the root scanning process remains unaltered.

Work Buckets and Dependencies. The correctness of GC algorithms is contingent on maintaining strict temporal orderings between various collector tasks. For example, no transitive closure work should start before all mark table zeroing is complete, to avoid missing objects due to incorrect mark states. On the other hand, scanning root work packets generates transitive closure packets (an implicit dependence), but these two packet types may correctly execute independently and thus require no explicit dependency relationship.

We use *work buckets* to manage work packets with common dependencies. For example, each space in the heap has its own implementation of mark table zeroing, but they share the same dependencies so the scheduler puts them in the same work bucket. We statically analyze the work packet dependencies for each GC, and attach dependencies to work buckets, forming a bucket dependency graph for each collector. The scheduler activates buckets in an order that satisfies their dependencies, ensuring that no packet executes before all of its dependencies are satisfied. This process trivially satisfies transitive dependencies. We maintain a global queue of active packets and each work bucket has an *inactive* queue with a count of packets. The inactive queues are unavailable to GC workers. We track the number of predecessor buckets that are incomplete for each *inactive* bucket. When all predecessor buckets are empty and thus complete, we activate the dependent bucket, transferring all packets from the *inactive* queue to the global shared *active* queue, making them available for execution.

We also support prioritized work packet execution, maintaining separate queues within each work bucket for packets of different priorities. The scheduler is designed to preferentially select high-priority packets when they are available. For instance, the LXR garbage collector [62] utilizes this feature to prioritize lazy RC decrement work packets over concurrent marking packets, facilitating faster RC completion and the timely release of memory.

Dependency-based Scheduling. Our scheduler is shared across all GC algorithms, executing work packets agnostic to the packet type and GC algorithm. The work packet design generalizes over classic phase-based ordering, which is trivially expressed in work packets by specifying simple linear dependencies, as in Figure 1(a). The baseline system in MMTk$_R$ groups packets into sequential phases, with each phase dependent on the completion of the previous one. (MMTk$_R$ does not contain the work bucket implementation described above.) Each phase corresponds to a work packet queue and a state indicating whether the phase is active. Worker threads poll packets by inspecting queues within the active phases. When the queue of a phase is emptied, the scheduler activates the next phase, ensuring that the GC tasks are executed in the correct sequence. For example, an arrow from the Reset Mark Table phase to the Transitive Closure phase in Figure 1(a) shows that no tracing packets in the Transitive Closure phase can execute until the entire Reset Mark Table phase completes.

Figure 1(b) illustrates work buckets and their dependencies and Figure 1(c) provides the pseudo code for dependency-based scheduling. We relax unnecessary ordering constraints between work buckets that may execute in parallel. During each GC epoch, the scheduler first activates the Stop Mutators bucket. This bucket contains a single packet instance that stops the mutators and adds initial work packets to other buckets to kickstart the GC process. A worker thread will then execute the packet, and once complete, will mark the bucket empty, activating its successors, the Scan Roots and Reset Mark Table buckets. Workers then consume packets from either bucket, performing root scanning and mark table zeroing with maximum parallelism. Note that although some tracing work is emitted by the Scan Roots packets, the Transitive Closure bucket remains unavailable until the Reset Mark Table bucket completes.

In our initial implementation, we observed that the phase-based scheduler requires GC workers to inspect multiple active work packet queues to find available packets, leading to increased contention and reduced CPU utilization. To reduce this overhead, we created a global queue to hold all available packets. When a bucket is activated, we transfer all packets to this global queue. Workers then retrieve packets from a single queue, effectively reducing the cost associated with dequeuing packets. The next section discusses how we prevent dequeuing contention.

Work buckets and the dependency graph manage the ordering and dependencies of work packets, ensuring correctness. Packet execution is highly parallelized, avoiding thread synchronization barriers except where dependencies specifically demand it.

3.3 Two-tier Work Stealing

To maximize parallelism and reduce thread starvation, we introduce a novel two-tier work stealing mechanism to our runtime. Idle threads may steal work packets, and in some circumstances, work items for fine-grained load balancing.

Work Packet Stealing. In addition to the global work packet queue, we add a thread-local work packet queue to each GC worker thread. During execution, when a new work packet is generated for an active bucket, the work packet runtime puts the packet in the worker's local queue, minimizing contention on the global shared queue. (Packets for closed buckets are not schedulable, so always go straight to the respective bucket.) To load balance, when a worker exhausts its local queue and the global queue is empty, it can steal work packets from the local queues of others. We use the Chase-Lev style work-stealing deque [10] to manage the local work packet queues. Under this model, worker threads push and pop packets from their deque's private end, while other threads use the public end of the deque to steal work.

Work Item Stealing. Load balancing of numerous performance-critical transitive closure work packets is challenging. Work packet stealing depends on workers producing new packets. However, during a transitive closure a worker may run for long periods pushing and popping work in approximately balanced quantities and therefore not generating any net new work. Such workers may nonetheless have substantial work in their local stack, albeit less than a full packet. To address this problem, we allow worker threads to steal individual work items from busy workers, enhancing load balancing when no packets are available locally, globally, or for stealing.

One straightforward implementation would be to replace the private stack in each packet with a work-stealing deque to allow work item stealing. However, frequent deque operations require expensive memory barriers [25]. From our experiments, using just a work-stealing deque significantly increased contention and degraded GC performance. Adapting the design from OpenJDK collectors, we split the stack into a small fixed-sized ABP-style work stealing deque [2] and a private atomic-free stack. When pushing new items, we first attempt to push into the work-stealing deque, making items visible to other threads quickly. If the deque is full, items are pushed onto the packet's private stack. Similarly, workers prioritize popping items from their private stack. These operations on the private stack are synchronization-free and do not require heavy memory barriers, thereby reducing contention costs significantly.

Figure 3(b) presents the pseudo code for the transitive closure work packet with two-tier work stealing. Each worker's Thread Local Storage (TLS) includes a small fixed-sized work-stealing deque (L 24), while each packet retains a private stack (L 2). The kernel function processes the stack and TLS deque (L 11, L 13) followed by attempts to steal items from other workers' deques (L 20). When at least one item is stolen and added to the local stack, the kernel restarts the entire process to process the local work. Work packet stealing (i.e., stealing an entire packet) is notably more efficient than stealing individual items. Consequently, a crucial optimization is early work packet termination if new packets are available in the global queue or other workers' local work-packet queues, allowing quick acquisition of new packets for processing.

While achieving performance improvements in the common case, frequent work-item stealing may lead to increased contention. We will discuss some examples in Section 6.1, where we saw degradation due to such synchronization overhead. We currently apply work-item stealing only to transitive closure packets, as these are the most common packets throughout the entire GC cycle (Table 1). We plan to add item stealing to other packet types.

4 Software Engineering Evaluation

This section analyzes and discusses the software engineering benefits of the work packet design, including its generality. We quantify code reuse and show how the concentration of critical code in kernels simplifies optimization, verification, and performance analysis.

Code Reusability. Table 2 shows substantial code reuse across GC algorithms in the work packet-based MMTk. The table presents the number of unique work packet types for each GC and the number of shared (reused) work packets. MMTk defines 28 reusable and algorithm-oblivious work packet types. Each GC uses a subset of them and may define its own packets. For example, all GCs reuse 14 work packet types that together implement root scanning for MMTk's OpenJDK binding. Immix (IX) uses 25 types of packets, 23 are shared with other GCs, and 2 are unique to Immix. LXR has relatively more unique work packet types at 15, as the only reference counting GC in MMTk$_R$, but LXR still reuses 26 types. If MMTk$_R$ adds more reference counting collectors in the future, they will likely reuse some if LXR's work packet types. Note that GenCopy, GenIX, StickyIX, and SemiSpace require zero unique types, and MarkSweep and MarkCompact only need one and three unique packets respectively. Please refer to the appendix for the complete list of work packet types that our system currently defines.

This high level of code reuse significantly reduces code duplication across different GC algorithms, simplifies the maintenance of the GC framework, and enables rapid development and experimentation. Reusing work packets streamlines the optimization process, since optimizing a shared packet benefits all GCs that use it. In contrast, most existing GC frameworks implement GC algorithms as a monolithic entity, leading to large amounts of code duplication and a complex code base with significant maintenance burden.

Table 3 shows the lines of code (LOC) breakdown for the GC modules in OpenJDK and MMTk with work packets. We use LOC as a metric due to the difficulty of attaining reliable cross-language complexity metrics on these code bases, and past evidence that LOC is a reasonable measure of code complexity [22]. LOC excludes comments, blank lines, and platform-specific code (e.g., object scanning implementations, which are closely tied to the runtime rather than the GC module). Only 19.4% of OpenJDK GC code is shared across GCs, while MMTk$_R$ achieves 85.6% code reuse. *First*, because of this high code reuse, MMTk's LOC is significantly lower than OpenJDK's LOC, while still implementing 10 GCs including the high-performance LXR collector. *Second*, the LOC required to implement a new GC algorithm is reduced. For example, comparing to the default G1 GC, LXR

Table 2. **Reuse of work packet types for eight GC algorithms in MMTk$_R$.** We exclude two debugging GCs (NoGC and PageProtect). MMTk$_R$ and its OpenJDK binding define 49 work packets types. 28 types are reused across different GCs. The first row shows the number of types that are unique to each GC. The second row shows the number of types common to two or more GC algorithms.

	IX	LXR	GenCopy	GenIX	StickyIX	SemiSpace	MarkSweep	MarkCompact
Unique	2	15	0	0	0	0	1	3
Shared	23	26	25	25	23	23	23	23

Table 3. **Unique Lines of Code (LOC) breakdown for the GC algorithms in OpenJDK and MMTk$_R$,** excluding comments, blank lines, and platform-specific code in absolute numbers and percentages.

(a) OpenJDK GC Module

GC	LOC	%
CMS	14 531	12.5
Epsilon	698	0.6
G1	27 625	23.8
Parallel	15 025	13.0
Serial	1 975	1.7
Shenandoah	20 393	17.6
ZGC	13 247	11.4
Shared	**22 437**	**19.4**
Total	*115 931*	*100.0*

(b) MMTk$_R$ with Work Packets

GC	LOC	%
GenCopy	301	0.7
GenImmix	301	0.7
Immix	508	1.1
LXR	3 886	8.6
MarkCompact	305	0.7
MarkSweep	228	0.5
NoGC	164	0.4
PageProtect	157	0.3
SemiSpace	236	0.5
StickyImmix	388	0.9
Shared	**38 613**	**85.6**
Total	*45 087*	*100.0*

requires 85% less algorithm-dependent code, which subsequently simplifies its implementation, performance tuning, and maintenance.

Our two-tier work stealing code is a patch against the transitive closure work packet code, and applies directly to any heap traversal component in MMTk$_R$, including IX's full heap tracing, LXR's RC increments, concurrent marking, and mature evacuation. This design structure thus delivers scheduling optimizations to all the work packet types, efficiently boosting the performance in both concurrent and stop the world collectors easily *without porting this optimization* (see Section 6.3).

Verification and Performance Analysis Tools. The use of well-encapsulated kernels simplifies verification and performance analysis, as prior work [21, 58] demonstrates. This research on performance tracing and verification used an early prototype of work packets in MMTk$_R$. Integrating these tools into kernels is straightforward. The high reuse of work packets ensures that most trace points and verification processes require one implementation per kernel for all GC algorithms that use the kernel. Huang et al. [21] demonstrate that the modular work packet design facilitated high-fidelity performance measurements with very little code, exposing subtle performance regressions. Xu et al. [58] apply model checking to work packet kernels. They show that the small kernels make model checking tractable, isolating critical components in a way that is very hard to do in a monolithic garbage collector.

Transferability and Flexibility. The work packet system is designed to be language-agnostic. In addition to OpenJDK, which we evaluate in Section 6, the upstream version of the work packet runtime in MMTk$_R$ already supports four other language runtimes: JikesRVM, V8, CRuby, and Julia [13, 48, 49, 53, 57], demonstrating the design's flexibility in handling diverse and complex platforms.

Generality. The work packet framework is algorithm-agnostic and general by construction. Although it is out of scope to demonstrate generality by reimplementing extant production collectors in the work packet framework, we demonstrate generality two ways.

First, we point to the diversity of collectors already implemented with work packets (Table 2). Among them are the canonical full-heap algorithms: semi-space [11, 17], mark-sweep [28], mark-compact [47], and Immix [6], as well as their generational variants [56]. LXR [62] additionally builds upon reference counting [12], coalescing reference counting [26], region-based collection [24], remembered sets [56], and snapshot at the beginning concurrent collection [59]. These canonical

Table 4. **The algorithmic building blocks of four of today's state of the art GCs**, replicated from [61]. MMTk implements all of these building blocks aside from the loaded value barrier (LVB).

	L&D [24]	**G1** [14]	**Shenandoah** [18]	**C4** [55]	**ZGC** [27]
Heap structure	Single-Level, Fixed-Size, Region-Based				
Primary Liveness	Whole-Heap Trace				
Reclamation	Mixed	Evacuation Only			
Trace	S.T.W.	Concurrent SATB [59]		Concurrent LVB [55]	
Evacuation	Stop-the-world		Concurrent LVB [55]		
Generational GC	Not supported	Supported	Not supported	Supported	Not supported

collectors are powerful algorithmic foundations that form the basis for other garbage collectors. For example, G1 [14] is built upon region-based collection [24], remembered sets [56], and snapshot at the beginning [59]. More recently, two new collectors use the work packet framework. Iso [44] combines DLG-style thread-local garbage collection [15, 16] with Immix to deliver state-of-the-art performance on request-based workloads. An implementation of the Compressor [23] was recently contributed to MMTk, marking another significant algorithm supported by the framework. While the coverage of implementations will never be complete, we claim that these building blocks are a strong demonstration of generality of our system, covering a significant part of the algorithmic design space of modern garbage collectors.

Second, we consider existing state of the art production collectors, in particular G1 [14], Shenandoah [18], and ZGC [27]. To understand the algorithmic building blocks of these collectors, we refer to [61]. The authors document the algorithmic relationships between these collectors and the high degree of commonality between them. In Table 4, we replicate a table from that paper, decomposing the algorithmic structure of each collector. The current implementation of MMTk already supports all building blocks needed by G1. MMTk does not currently have a loaded value barrier (LVB), required to implement C4 [55], ZGC [27], and Shenandoah [18]. Read barriers are injected by the compiler into the application code and their implementations are dominated by runtime and compiler challenges. They are thus largely incidental to the work packet framework we present here. This analysis gives us confidence that there are no in principle challenges associated with implementing G1, Shenandoah, ZGC or C4 on the work packet framework, although all of them would be significant engineering undertakings. ZGC lead Erik Österlund at his ISMM'25 keynote noted that: *"building a production garbage collector [...] and sorting out all the performance issues [...] takes about 10 years."* [43]. His implementation experiences further motivate our work.

5 Performance Evaluation Methodology

We evaluate our design in MMTk$_R$ on OpenJDK 11.0.19+1 [53]. We report total execution time, GC stop-the-world (STW) time, and CPU utilization for the following two high-performance collectors (also see Section 2.5): i) *LXR*, a state-of-the-art GC algorithm [62], and ii) *Immix* [46], a fully stop-the-world full heap tracing collector, abbreviated to 'IX' for the remainder of this paper. We compare the following three configurations.

P: This configuration implements a traditional **P**hase-based GC task scheduling.
D: This configuration replaces phases with **D**ependence scheduling of fine-grain work packets.
S: This configuration adds the two-tier work-**S**tealing algorithm to ***D***.

We also measure and report total execution time and GC STW time for G1 in OpenJDK versions 11, 21 (the latest long-term support (LTS) release), and 24 (the most recent available release). All experiments use G1's default configuration. As a phase-based collector, G1 serves as an alternative baseline for evaluating our work packet design. Blackburn et al. [4] found that G1 in OpenJDK 21 achieves the best performance and the lowest GC overhead on most DaCapo benchmarks when compared to other OpenJDK collectors. As such, G1 represents a production-ready, state-of-the-art OpenJDK-native collector in our evaluation. Three out of the 22 DaCapo benchmarks — Cassandra, Tradebeans, and Tradesoap— use deprecated JVM APIs and cannot run with OpenJDK 24. Therefore, in all experiments involving OpenJDK 24, we exclude these benchmarks from all configurations, including G1, Immix, and LXR.

P is the original implementation we contributed to MMTk_R with minor changes for instrumentation. Both ***D*** and ***S*** implementations are patches against ***P***. All three variants implement work-packet stealing, but only ***S*** implements both work-packet and work-item stealing. All three configurations differ only in their schedulers and otherwise they share the same GC implementation and runtime, precisely revealing the impact of the scheduler design changes.

Ideally we would make apples-to-apples comparisons between two or more collectors that differ only in whether they use work packets. Unfortunately, MMTk_J does not use work packets and is part of an entirely different runtime system, JikesRVM. OpenJDK's native GCs also do not use work packets. They implement different GC algorithms than MMTk_R and use a different implementation lanuguage (C++). Thus neither MMTk_J nor OpenJDK's native collectors are suitable for an apples-to-apples comparison. For this reason, we use MMTk_P as the baseline for our analysis. Even though it does not allow us to compare with a non-work packet-based system, MMTk_P does serve as a good baseline to precisely measure the impact of removing phases and synchronizations, which is the difference between the ***P*** and ***D*** variants. In addition, Section 6.4 analyzes the scalability of MMTk_P and G1, demonstrating the scalability of our design.

MMTk is designed to rely entirely on Rust's profile-guided optimization (PGO) framework for build-time optimization. It eschews manually-inserted optimization directives. We use GCBench [8], which is *not* part of our benchmark evaluation set, to generate the profiles used at build time. OpenJDK's collectors are written in C++ and use established approaches for performance optimization including templates, inline functions, macros and pragmas, not relying on PGO.

5.1 Benchmarks

We use the 23.11-chopin version of the DaCapo benchmark suite [4] for our evaluations, in the default configurations. We report results for all 22 benchmarks from the suite. Unless otherwise stated, we run each benchmark 20 times and report the average for each benchmark. For each run, we repeat the workload five times, only reporting the last iteration's statistics. This methodology warms up the JVM to reach steady state before we collect the statistics, reducing experimental noise. We set the heap size individually for each benchmark by determining its minimum runnable heap size using a bisection search and set the heap size to multiples of this minimum heap size.

5.2 Machine Configuration

We use Zen 3 and Zen 4 machines. We use Zen 3 machines for most experiments. They have 16/32 cores with 3.4 GHz CPUs, and 64 GB DDR4-3200 memory. We use Zen 4 machines to evaluate scalability. They have 64/128 cores with 2.45 GHz CPUs, and 256 GB DDR5-4800 memory. All machines use the same system image: Ubuntu 22.04.4 with Linux kernel 6.8.0-40-generic.

5.3 Total and Stop-the-world Time Measurements

We measure and report the impact of MMTk_P, MMTk_D, and MMTk_S variants on total execution time and GC stop-the-world time for IX and LXR. Although G1 *is not directly comparable*, we normalize to G1 because: i) it is the default production collector in OpenJDK, and ii) it is a state-of-the-art GC algorithm that fully employs phase-based GC task scheduling with synchronization barriers. It therefore grounds our results in the state-of-the-art. For each configuration, we measure the per-benchmark total execution time and stop-the-world time, and report the geometric mean across all 22 benchmarks. We perform the analysis on both a moderate 2× heap and a tight 1.5× heap. We use default JVM parameters for all experiments.

5.4 Utilization Measurements

We show that a major advantage of our design is better load balancing. We report the CPU utilization for each DaCapo benchmark and the geometric mean. We calculate utilization as follows. i) For a given GC pause, we calculate $T_{\Sigma x}$, the sum of time spent executing work packets by all workers, and $T_{\Sigma e}$, the sum of elapsed time (elapsed time × number of workers). ii) We calculate the utilization for each pause as the ratio $T_{\Sigma x}/T_{\Sigma e}$. iii) For each benchmark, we report the average utilization across all GC pauses. We also report the geometric mean across all benchmarks. We use this approach with respect to three different types of utilization.

- Overall utilization for the entire GC pause.
- Utilization when the GC is performing the main transitive closure, including IX's marking and evacuation trace and LXR's mature evacuation trace. This period is the most time-consuming part of their pauses, where our two-tier work-stealing algorithm aims to improve, and is thus reported separately. The transitive closure for weak reference processing is excluded.
- Utilization only during the period when LXR is performing reference counting increments. RC increments dominate LXR's pause time and benefit from the two-tier work-stealing algorithm, so we study it separately.

5.5 Scalability Measurements

We assess the scalability of our design by running the benchmarks across different numbers of hardware threads, ranging from 1 to 128. We use both the `taskset` command and JVM options to control the number of hardware threads available to the JVM and the number of GC threads. We had to exclude 11 benchmarks from these results because they initialize more mutator threads when more hardware threads are available, leading to increased memory allocation and out-of-memory errors on a moderate 2× min-heap. Increasing heap size to prevent OOM error was not feasible as it leads to fewer GCs with fewer and more noisy data points. For each benchmark and core count, we report the mean over 10 runs. We also report the geometric mean across all 11 benchmarks.

For the scalability test on utilization, we also report the geometric mean ideal utilization across the 11 benchmarks, and compare it against the real utilization. We report the transitive closure utilization on three Immix variants: IX_P, IX_D, and IX_S, as the real utilization.

Ideal Utilization. We use the methodology proposed by Barabash and Petrank [3] to measure ideal CPU utilization during a garbage collector's transitive closure. Ideal utilization is calculated by simulating a transitive closure on a heap graph. The simulator is parameterized with the number of available CPU threads denoted as C, simulating per-tick tracing jobs. For each tick, C objects are popped from the mark stack, marked, and their fields scanned, pushing child objects back onto the stack. The simulator assumes processing each object by a thread takes exactly one tick. If fewer than C objects are processed in a tick, it indicates idle threads during that tick. The simulator reports overall ideal utilization for a heap graph by summarizing per-thread idle and busy ticks.

We perform our ideal utilization measurement offline using heap snapshots, containing all reachable live objects and their fields. We use a customized garbage collector to generate snapshots by recording accessed objects and fields during each GC pause, dumping data for later analysis. For each benchmark, we collect multiple heap snapshots — one for each GC pause. We calculate each snapshot's utilization as the ideal utilization for this GC pause's transitive closure period. We then report the geometric mean across all the benchmarks.

Table 5. **The impact of fine-grained dependencies and two-tier work stealing.** We evaluate three variants of IX and LXR: the phase-based baseline (***P***), adding fine-grained work packet dependencies (***D***), and adding fine-grained dependencies and two-tier work stealing (***S***). The best values across the three variants are in green, and the worst values in orange. We show total stop-the-world (STW) time, total utilization, trace utilization, and reference counting increment utilization, measured at a heap size 2× G1's min-heap. We report the geometric mean across all benchmarks. Times are normalized to G1's STW time.

	STW Time / G1			**Utilization**			**Trace Util.**			**RC Inc Util.**		
	Lower is better			*Higher is better*			*Higher is better*			*Higher is better*		
	P	***D***	***S***	***P***	***D***	***S***	***P***	***D***	***S***	***P***	***D***	***S***
G1	1.00	–	–	–	–	–	–	–	–	–	–	–
IX	1.94	1.86	1.80	0.66	0.70	0.81	0.85	0.85	0.92	–	–	–
LXR	0.50	0.50	0.47	0.32	0.31	0.40	0.53	0.56	0.75	0.42	0.41	0.52

We use the ideal utilization to qualitatively evaluate how closely the work packet runtime approximates the ideal. We use IX for this experiment because it is a stop-the-world full-heap tracing GC and its transitive closure utilization is directly comparable to the ideal utilization, which is also measured based on the heap snapshot collected by IX. In this methodology, ideal utilization is strictly related only to the heap shape. In real-world GC pauses, factors such as thread contention, operating system scheduling, and the varying scanning and copying costs for different object types impact pause times. Therefore, even when a GC traces the heap in an optimal order, it is unlikely to achieve ideal utilization.

6 Performance Evaluation

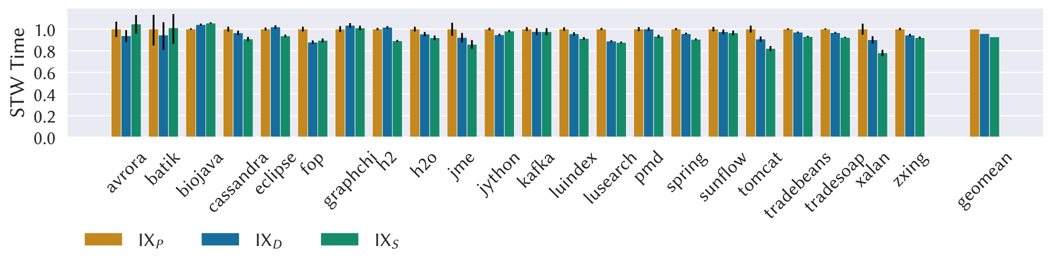
Recall that we cannot perform a direct apples-to-apples comparison of a garbage collector using the work-packet runtime against a highly tuned monolithic implementation of the same collector using ad hoc work stealing. Instead we point to the performance of LXR in Table 5, Table 6, and Figure 5. While the overall performance of LXR is not directly attributable to the work packet runtime, these results highlight that the framework is sufficiently flexible and performant as to support the development of innovative collectors that deliver state of the art performance [44, 65]. **We focus most of our performance study on the advantages of fine-grained dependency (D) and two-tier work stealing (S), relative to the baseline work packet runtime (P).**

6.1 Stop-the-world Time

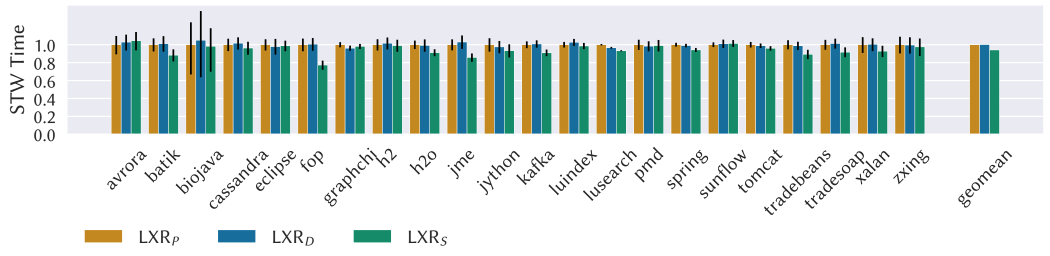
The first column of Table 5 shows that work packet variants ***D*** and ***S*** reduce stop-the-world (STW) time for both IX and LXR. Per-benchmark total STW time is reported in Figure 4, normalized to ***P***.[2]

Comparing with OpenJDK GCs. Table 6(b) presents the stop-the-world (STW) times for IX and LXR on 1.5× and 2× heap sizes, normalized against the STW time of OpenJDK 11 G1. For comparison, we also evaluate and present G1 in OpenJDK 21 in the same table. G1 in OpenJDK 21 achieves significantly lower STW times than G1 in OpenJDK 11, with reductions of 19.8% and 22.1% on 1.5× and 2× heaps, respectively. Nonetheless, our highest performing collector, LXR$_S$, surpasses G1 in OpenJDK 21 by 48.4% and 40.1% on these two heap sizes in terms of STW time. LXR and G1 are fundamentally different algorithms, so this striking result cannot be directly attributed to the work packet framework, however the result clearly demonstrates that the work packet framework is suitable for constructing innovative, high performance collectors.

[2]Please refer to the Appendix for detailed tabulated results.



(a) IX: normalized stop-the-world time for IX_P, IX_D, and IX_S



(b) LXR: normalized stop-the-world time for LXR_P, LXR_D, and LXR_S

Fig. 4. **The impact of P, D, and S variants on stop-the-world (STW) time for IX and LXR**, measured on a 2× heap. Both figures are normalized to the corresponding *P* variants for each GC.

Fine-grained Dependency (D). On the moderate 2× heap, using fine-grain dependencies reduces IX_D's STW time by 4.2% over IX_P on average, and nine out of 22 benchmarks exhibit a STW time reduction of more than 5%. Fop shows the largest reduction, 12.1%. This reduction is primarily due to the removal of synchronization barriers between Scan Roots, Reset Mark Table, and Transitive Closure buckets. In IX_D, Scan Roots and Reset Mark Table start simultaneously. If GC threads complete Reset Mark Table early, they can promptly begin Transitive Closure jobs without waiting for Scan Roots to finish. However, this improvement is not universally evident. Pmd and H2 show no observable STW time change, with a difference of less than 2%. Two benchmarks, Biojava and Graphchi, exhibits a STW time increase (4.1% and 3.4% respectively) compared to IX_P. We observed that these benchmarks tend to complete their Reset Mark Table bucket after the Scan Roots bucket is finished. Consequently, the execution of Transitive Closure packets is less likely to overlap with Scan Roots packets, resulting in negligible performance gain.

In contrast to IX's noticeable improvement, LXR_D shows little improvement in STW time compared to LXR_P, with a difference of less than 0.1%. Graphchi exhibits the most considerable STW time reduction of 3.9%, primarily due to the lower synchronization overhead on maintaining bucket dependencies than phases. No benchmarks exhibit observable slowdowns or speedups. The largest slowdown is observed on Biojava, with a STW time increase of 4.9%, but this is within the margin of error. The major reason for the minimal stop-the-world time improvement is that LXR_P's STW time is dominated by root scanning and RC increments. LXR already executes these packet types in the same bucket to minimize synchronization overhead. Thus, adding fine-grained dependencies has minimal impact on LXR's STW time. These results motivate the search for other optimization opportunities, such as the two-tier work stealing algorithm, which we discuss next.

Two-tier Work Stealing (S). The *S* variants of our system add two-tier work stealing, consistently improving parallelism and reducing STW time for both IX_S and LXR_S on most benchmarks compared to the *P* and *D* variants. IX_S decreases the geometric mean STW time by 3.2% relative to IX_D. Eight out of 22 benchmarks exhibit a STW time reduction of more than 5%. H2 and Xalan show the most significant reductions, 12.3% and 13.4%, respectively. The improvements are primarily due to the two-tier work stealing algorithm successfully balancing the fine-grained work items across all worker threads, which cannot be achieved by stealing only coarse-grained work packets.

Table 6. **Total execution time and stop-the-world time for IX, LXR, G1 on JDK 11, and G1 on JDK 21**, measured on 1.5× of G1's min-heap. The best values for each GC are highlighted in green, and the worst values in orange. We report geometric mean across all benchmarks. All values are normalized to G1. All collectors are built based on JDK 11, except for $G1_{21}$ in the gray columns, which is from JDK 21, the latest LTS version.

(a) **Total execution time**

Heap	$G1_{21}$	G1	IX_P	IX_D	IX_S	LXR_P	LXR_D	LXR_S
1.5×	0.95	1.00	1.29	1.26	1.21	0.86	0.86	0.88
2×	0.97	1.00	1.08	1.06	1.05	0.90	0.90	0.89

(b) **Total stop-the-world time**

Heap	$G1_{21}$	G1	IX_P	IX_D	IX_S	LXR_P	LXR_D	LXR_S
1.5×	0.80	1.00	2.45	2.29	2.21	0.45	0.45	0.41
2×	0.78	1.00	1.94	1.86	1.80	0.50	0.50	0.47

Conversely, Avrora, Batik, and Jython show the largest STW time increase of 11.6%, 7.0%, and 3.7%, respectively. Avrora and Jython show an increase in the number of GC pauses by 14.9% and 6.3%, respectively compared to IX_D. We observe that the two-tier stealing algorithm changes the order in which the collector visits objects, thus changing the order in which the collector allocates memory as it evacuates objects. Consequently, different heap fragmentation patterns lead to different numbers of pauses. For Batik, the number of pauses is only increased by 2.2%, not matching the significant STW time increase. However, we find that it has a substantial increase in the retired instructions of 17.1% within GC pauses. This result suggests that fine-grained work item stealing may compel worker threads to spend more time spinning and stealing items. A possible solution is to reduce the frequency of stealing and the maximum retry limit for stealing in the spin loop dynamically at run time to avoid excessive CPU consumption.

LXR_S reduces its geometric mean STW time by 6.0% compared to LXR_D, with Fop seeing the most substantial reduction of 23.1%. Ten out of 22 benchmarks show STW time reductions exceeding 5%. The improvement seen in LXR_S shows that the fine-grained load balancing provided by the two-tier work stealing algorithm is necessary, especially on pauses like LXR's RC pauses, where fine-grained dependency-based scheduling does not offer any improvements. No benchmarks exhibit an observable slowdown in STW time for LXR_S.

6.2 Total Execution Time

On a smaller 1.5× heap, total time improvements for IX_S and LXR_S are about the same. No benchmark exhibits an observable slowdown for either IX or LXR. Unlike IX, which always performs full-heap tracing, LXR mostly performs short RC pauses in a single highly-optimized bucket, thus our optimizations of fine-grained dependency and work-item stealing do not show an observable total time reduction. On the other hand, on the 1.5× heap, IX GC time improvements impact total execution time, as shown in Table 6(a) and demonstrate the benefits of the work packet design. IX_S shows a reduction in total execution time of 6.4% compared to IX_P. In a moderate 2× heap, the GC times of IX or LXR do not dominate total performance, so the STW reductions do not translate to substantial total time improvements. We saw less than 3% change in overall execution time.

Our primary evaluation has been in JDK 11 because that is the version supported by MMTk at the time of writing. However, since G1 has been improved since JDK 11, we also include comparisons with G1 in JDK 21 and JDK 24. JDK 21 is the latest LTS version available at the time of writing. JDK 24 is the most recent available OpenJDK release. $G1_{21}$ achieves speedups of 5.1% and 3.2% over G1 on JDK 11 for the 1.5× and 2× heap sizes, respectively. However, it remains slower on total performance than LXR_S by 8.3% and 8.4% on these two heap sizes, respectively. JDK 24 cannot run three benchmarks from DaCapo due to compatibility issues, so we cannot directly compare it with the results in Table 6(a). Excluding the incompatible benchmarks, $G1_{24}$ outperforms G1 on JDK 11 by 7.9% and 5.8% on the 1.5× and 2× heaps, respectively. Nevertheless, its total execution time is still 6.3% and 7.5% slower than LXR_S on these configurations.

Although these results cannot be directly attributed to the work packet runtime alone, they demonstrate that it is capable of supporting the construction of high-performance garbage collectors, such as LXR, which outperform the latest G1 implementations in both JDK 21 and JDK 24. We note that many of the performance gains seen in $G1_{21}$ and $G1_{24}$ are attributed to optimizations in other components of the virtual machine, such as the compiler and runtime. We believe most of these non-GC optimizations are orthogonal to specific garbage collector implementations, so expect them to also improve performance for systems built on the work packet runtime.

6.3 Utilization

Fine-grained Dependency (D). IX_D improves overall utilization within GC pauses by 4.9% with fine-grained work packet execution when compared to IX_P by increasing effective parallelism. Utilization during tracing increases by 0.3%. Most benchmarks exhibit consistent improvement, except for Biojava, which shows a slight decrease of 1.6% in total utilization, and 1.5% in trace utilization. Biojava also reports an increased STW time with IX_D. This result suggests that tracing work packets are less likely to run in parallel with root scanning packets.

LXR is dominated by RC pauses. Unlike IX, LXR's RC pauses already put all root scanning and transitive closure packets into a single bucket. Thus, the fine-grained dependency scheduler should and does have minimal effect on LXR's STW time and utilization, with no significant change observed across all benchmarks. However, LXR_D shows a slight 1.1% slowdown in overall utilization compared to LXR_P. Utilization during RC increments decreases by 1.2%, while trace utilization improves by 2.9%. The most notable slowdown occurs in Batik, with a decrease in overall utilization of 7.7%. We ran Batik with only one CPU core to limit the task scheduling and parallelism noise, and found that it has a 11.0% reduction in CPU utilization compared to LXR_P, but the STW time has no observable change, with a minor reduction of 0.8%. This result suggests that although dependency-based scheduling does not affect Batik's RC performance, it leads to a higher synchronization overhead. Moreover, LXR on Batik only performs RC pauses, and the RC utilization on one CPU core also shows no observable change on LXR_D, with a difference of less than 1% compared to LXR_P. The synchronization overhead likely comes from the other components of RC pauses, primarily the block sweeping tasks after RC increments. The slowdown on Batik shows that dependency-based scheduling can sometimes be more costly than phase-based scheduling. However, due to the improvement in stop-the-world time on IX and LXR, especially the significant performance gain with a large number of hardware threads (Section 6.4), it is still beneficial to replace phases with fine-grained buckets.

Two-tier Work Stealing (S). The ***S*** variants of IX_S and LXR_S add two-tier work stealing. They improve load balancing over their ***D*** variants and consistently deliver the highest utilization across all benchmarks. IX_S enhances overall utilization by 11.0% and trace utilization by 7.0% over IX_D. Graphchi and Luindex experience the greatest improvement in total utilization, at 23.9% and 25.5%,

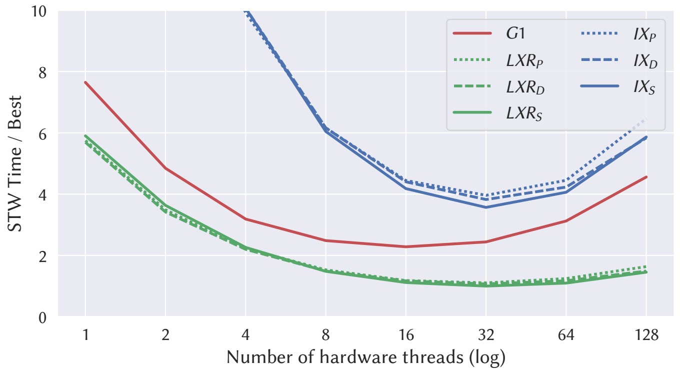


Fig. 5. **Stop-the-world time for G1, IX, and LXR** at 2× G1's min-heap, with the three variants of IX and LXR: *P*, *D*, and *S*. We report STW time across on various hardware counts, from 1 up to 128. We normalize each benchmark's result to G1's single core STW time, and show the geometric mean across all benchmarks. We further normalize the results to the global best STW time across all configurations.

respectively. Similarly, LXR_S increases overall, trace, and RC increment utilization by 8.4%, 19.9%, and 11.1%, respectively. The most notable improvements are in Fop and Jme, with overall utilization increases of 20.3% and 26.9%, respectively. No benchmark exhibits a decrease in utilization.

High utilization on average yields improved performance. Luindex, with a 25.5% utilization improvement, demonstrates a STW time reduction of 4.6% on IX_S, compared to IX_D. Similarly, Graphchi on IX_S improves utilization by 23.9% over IX_D, with a pause time reduction of 2.3%. However, higher utilization does not always translate into STW time reduction. For example, Jython on IX_S has a 5.8% increase in utilization, but a 3.7% increase in STW time, compared to IX_D. As discussed in Section 6.1, we found that Jython on IX_S increased the number of pauses by 6.3%. This result reveals that heap fragmentation changes due to reordering from parallelism in IX_S may cause either regressions or improvements. As discussed previously, high overheads due to ineffective item stealing may waste CPU cycles. An even more effective item stealing policy remains to be explored.

6.4 Scalability

We evaluate scalability of STW time and utilization on large numbers of hardware threads to show the parallelism benefits of our design. We compare the STW time scalability with G1, which uses a traditional phase-based design. We compare the utilization with the ideal utilization, to demonstrate the load balancing changes and how it approaches the ideal as we add ***D*** and ***S*** optimizations.

Stop-the-world Time. Figure 5 illustrates STW time for G1, IX, and LXR over different hardware thread counts, measured with a heap 2× G1's min-heap.[3] We evaluate a total of seven GC configurations. For each GC and benchmark, we normalize the results to G1's single core STW time, and present the geometric mean across 11 benchmarks. We then normalize the results to the best observed value across all configurations, 69.4 ms, which occurs on LXR_S at 32 cores. (Section 5.5 explains why we only use 11 benchmarks in this evaluation.) A key trend revealed in the figure is the diminishing returns in STW time reduction as the thread count increases to a certain threshold. Beyond this threshold, scaling is negative. This threshold is 16 hardware threads for G1 and 32 for all IX and LXR variants. Both IX and LXR exhibit more scalability than G1.

For G1, its best STW time occurs at 16 threads, at 2.3× the best value. G1's STW time escalates when the thread count exceeds 16. At 128 threads, G1's STW time degrades to 4.6× the best value. Because Immix (IX) is a full-heap stop-the-world GC, it exhibits the highest STW time among all evaluated GCs. IX_P starts with STW time 25.3× the baseline value at 1 hardware thread (not shown), and its minimum STW time is at 32 hardware threads with 4.0× the baseline. IX_D and IX_S reflect

[3]Please refer to the Appendix for detailed tabulated results.

Table 7. **128 thread total stop-the-world (STW) time for G1, IX, and LXR** at 2× of G1's min-heap. The first row normalizes time to each collector's ***P*** variant. The second row normalizes STW time to the best STW time for each GC variant. The best values are highlighted in green and the worst in orange.

	G1	**IX**$_P$	**IX**$_D$	**IX**$_S$	**LXR**$_P$	**LXR**$_D$	**LXR**$_S$
X_{128} / $X_{128,P}$	–	1.00	0.91	0.91	1.00	0.91	0.89
X_{128} / X_{Best}	2.00 *_/ 16 Threads_*	1.63 */ 32 Threads*	1.53 */ 32 Threads*	1.64 */ 32 Threads*	1.48 */ 32 Threads*	1.39 */ 32 Threads*	1.45 */ 32 Threads*

similar patterns, reaching minimum STW times at 32 threads with 3.8× and 3.6×, respectively. At 128 threads, the STW time of IX$_P$ increases to 6.5× the baseline value. In contrast, IX$_D$ and IX$_S$ display a reduced growth at 5.8× and 5.9×, respectively. Improved load balancing through the two-tier work-stealing algorithm more effectively curtails STW time growth as the thread count increase.

Across varying thread counts, LXR consistently maintains lower STW time than both Immix and G1. With one hardware thread, LXR$_P$, LXR$_D$, and LXR$_S$ have STW times of 5.7×, 5.7×, and 5.9× the baseline, respectively. They all reach their minimum STW time at 32 threads, with STW times of 1.1×, 1.1×, and 1.0× the baseline, respectively. Like G1 and Immix, LXR's STW time increases as the thread count rises beyond its best value (32). On 128 threads, LXR$_P$ has STW time of 1.6× the best value. Both LXR$_D$ and LXR$_S$ improve STW time significantly and record the lowest STW time among all GC variants on 128 threads, at 1.5× the optimal. LXR$_S$ has better scalability than both G1, LXR$_P$ and LXR$_D$ on 128 threads, primarily due to the improved load balancing.

Stop-the-world Time on 128 Threads. We present 128-thread STW time results separately as they show the most significant STW time reduction with our system where the traditional phase-based design does not deliver. Table 7 presents STW time for G1, IX, and LXR on 128 threads.

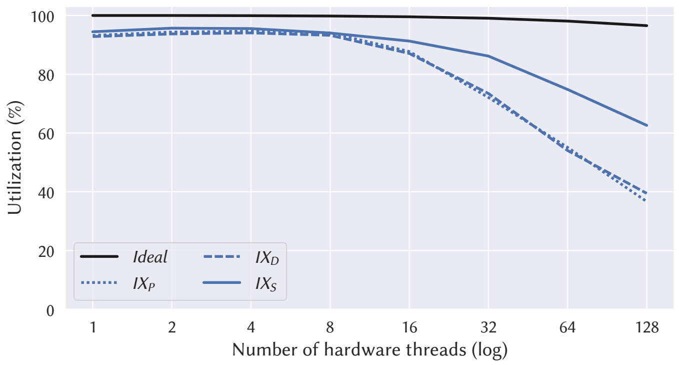
The first row shows STW time for each collector normalized to their ***P*** variants. Both fine-grained dependencies and two-tier work stealing notably reduce STW time on 128 hardware threads. As expected, the ***S*** variants exhibit the largest reduction, with IX$_S$ and LXR$_S$ displaying reductions of 8.6% and 10.8%, respectively, compared to their ***P*** variants. This results indicates that both designs effectively exploit parallelism and reduce STW time on a large number of hardware threads.

The second row presents STW time normalized to each GC variant's best STW time. For G1, optimal STW time is attained at 8 threads. At 128 threads, G1's STW time is 99.8% more than its optimal value at 16 threads. Both IX$_P$ and LXR$_P$ show improved results compared to G1. They show a STW time increase of only 63.2% and 48.0%, respectively, when comparing to their optimal STW time across different thread counts. While STW time reaches a minima at a certain level of parallelism and degrades after that, IX and LXR both take longer to reach this point compared to G1. Additionally, IX$_S$ and LXR$_S$ demonstrate similar or better relative improvements compared to IX$_P$ and LXR$_P$. Specifically, at 128 threads they experience degradations of 64.4% and 45.4%, respectively, when compared to their optimal thread count, which are both at 32 threads. This result confirms that the two-tier work-stealing algorithm and dependency-based scheduler can mitigate the negative scaling of STW time on large thread counts.

Utilization. Figure 6 illustrates the *ideal* utilization and the real transitive closure utilization for IX$_P$, IX$_D$, and IX$_S$ across different thread counts, ranging from 1 to 128.[4] The ideal utilization is measured using the methodology proposed by Barabash and Petrank [3], representing the theoretical maximum utilization for the benchmark suite. (See Section 5.5 for details.) For a single

[4]Please refer to the Appendix for detailed tabulated results.

Fig. 6. **Ideal utilization and the real transitive closure utilization** for IX_P, IX_D, and IX_S across 1 to 128 hardware threads. Each utilization is the geometric mean across the 11 evaluated benchmarks.



hardware thread, the ideal utilization is 100% , as the single GC thread remains busy throughout each simulated cycle. However, as thread count increases, workload imbalance due to heap shape causes a slight decline in ideal utilization. By 128 threads, it decreases to 96.5%, still significantly higher than the real utilization results. We observe that the heap graph shape for most benchmarks is sufficiently wide as to keep all threads busy in most simulated cycles.

On a single hardware thread, the real utilization is 93.2% for IX_P, 92.8% for IX_D, and 94.5% for IX_S, all lower than 100%. Real utilization is measured as the sum of time spent executing transitive closure work packets divided by the elapsed time spent in the transitive closure. Utilization values under 100% indicate non-negligible costs associated with activities outside of work packet execution, such as maintaining work bucket dependencies and stealing work packets. IX_S effectively improved the utilization by 1.7% compared to IX_D, but does not completely eliminate the overhead.

As the thread count rises, real utilization for all three variants quickly diminishes, dropping below 90% at 16 threads. Notably, this closely precedes the turning point at 32 threads where the STW time for the three variants begins to regress (Figure 5). Starting at 16 threads, IX_S utilization degrades significantly more slowly than both IX_D and IX_P, achieving 4.2% better on 16 threads, 12.7% on 32 threads, and 23.1% on 128 threads, compared to IX_D. This result demonstrates the effectiveness of the two-tier work-stealing algorithm in improving load balancing and maintaining utilization, particularly with a large number of hardware threads.

Once again, high utilization does not necessarily translate to low STW time. As discussed in Section 6.1, there are instances where the synchronization overhead of work-item stealing outweighs the benefits of improved load balancing. Our load balancing heuristics, derived from static empirical results, do not universally apply across all benchmarks. We believe that dynamic policies using run-time statistics will likely further improve work stealing.

7 Threats to Validity

Baseline. An ideal comparison would have used otherwise-identical implementations of one or more state-of-the-art garbage collectors based on the work packet abstraction and the traditional monolithic phased-based approach. Unfortunately the size and complexity of state-of-the-art collectors (Table 3(a)) makes such a comparison untenable for a research paper. We contributed an early, incomplete version of our work packet implementation to $\mathrm{MMTk_R}$ during its initial development phase. This version includes work packets and packet-level work stealing but retains the use of phases (later addressed by $\mathrm{MMTk_D}$) and lacks the two-level work stealing optimization (implemented in $\mathrm{MMTk_S}$). Reverting $\mathrm{MMTk_R}$ to a traditional phase-based implementation such as $\mathrm{MMTk_J}$ is non-trivial and would require substantial effort. Thus, we use our instrumented version of $\mathrm{MMTk_R}$, $\mathrm{MMTk_P}$, as the baseline for our evaluation, emulating a phase-based design.

OpenJDK 11. Because the MMTk's OpenJDK binding is only available for OpenJDK 11, most of our experiments are in OpenJDK 11. However, G1 has improved in subsequent releases of OpenJDK. To mitigate this, our comparison in Section 6.2 and Table 6(a) includes G1 in OpenJDK 21 (the latest LTS), and 24 (the newest version at the time of writing), demonstrating the performance of the latest OpenJDK GCs. While this comparison is not apples-to-apples, due to different LXR and G1 GC algorithms, our results demonstrate that the work packet design supports the development of interesting new collectors that offer state of the art performance. Porting MMTk to newer OpenJDK versions is beyond the scope of this paper, but we believe our performance improvements will carry over when MMTk is ported to future OpenJDK releases. Moreover, we note that many of the JIT compiler and runtime optimizations introduced in OpenJDK 21 and 24 are orthogonal to the GC algorithm. These enhancements will benefit applications using collectors built on the work packet framework once MMTk is available in later OpenJDK versions.

8 Future Work

The work packet framework is young and growing. We are excited to see algorithms from the literature such as the Compressor [23] and new algorithms such as Iso [44] and LXR [62] built with it. We believe the emergence of these new algorithms is evidence that the reuse and simplicity of work packets has made it a catalyst for rapid innovation. The growing move toward managed languages and the substantial overheads of today's production collectors [4] point to a need for innovation in a field that has historically demonstrated surprisingly little [61]. We hope that the community will see this challenge as an opportunity to explore fresh ideas, rapidly realize and evaluate them in a production-quality runtime. We are also excited by ongoing efforts to leverage work packets to verify correctness of GC algorithms [58], an ongoing source of security vulnerabilities.

9 Conclusion

We design and build work packets, a novel framework for garbage collector implementations that exploits two observations: most performance-critical GC code is concentrated in small kernels and the classic phase-based approach to temporal correctness limits parallelism. A work packet combines work item(s), a kernel, and scheduling constraints, distilling critical GC operations. Work packets are used within MMTk to implement eight distinct GC algorithms, with more under development. They facilitate a large amount of sharing, delivering significant software maintenance, reuse, verification, and optimization benefits. We introduce an efficient two-tier work stealing runtime system to orchestrate work packets in parallel, with minimal synchronization overhead. We show that work packets achieve software engineering benefits and performance. They improve performance and parallelism for Immix and LXR GCs across a large number of hardware threads and benchmarks compared to phase based implementations. We believe that work packets lay an excellent foundation for GC innovation and for developing more efficient and maintainable GC implementations.

10 Data-Availability Statement

We made our source code, the DaCapo benchmark suite, and detailed instructions to reproduce the results publicly available [63, 64].

References

[1] Bowen Alpern, Steve Augart, Stephen M. Blackburn, Maria A. Butrico, Anthony Cocchi, Perry Cheng, Julian Dolby, Stephen J. Fink, David Grove, Michael Hind, Kathryn S. McKinley, Mark F. Mergen, J. Eliot B. Moss, Ton Anh Ngo, Vivek Sarkar, and Martin Trapp. 2005. The Jikes Research Virtual Machine project: Building an open-source research community. *IBM Syst. J.* 44, 2 (2005), 399–418. doi:10.1147/SJ.442.0399

[2] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. 1998. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures* (Puerto Vallarta, Mexico) *(SPAA '98)*. Association for Computing Machinery, New York, NY, USA, 119–129. doi:10.1145/277651.277678

[3] Katherine Barabash and Erez Petrank. 2010. Tracing garbage collection on highly parallel platforms. In *Proceedings of the 2010 International Symposium on Memory Management* (Toronto, Ontario, Canada) *(ISMM '10)*. Association for Computing Machinery, New York, NY, USA, 1–10. doi:10.1145/1806651.1806653

[4] Stephen M. Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. 2025. Rethinking Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 940–954. doi:10.1145/3669940.3707217

[5] 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. doi:10.1109/ICSE.2004.1317436

[6] 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 29th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Tucson, AZ, USA) *(PLDI '08)*. Association for Computing Machinery, New York, NY, USA, 22–32. doi:10.1145/1375581.1375586

[7] Robert D. Blumofe and Charles E. Leiserson. 1999. Scheduling multithreaded computations by work stealing. *J. ACM* 46, 5 (Sept. 1999), 720–748. doi:10.1145/324133.324234

[8] Hans Boehm. 2024. GCBench. https://www.hboehm.info/gc/gc_bench/GCBench.java

[9] Philippe Charles, Christian Grothoff, Vijay A. Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. 2005. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, October 16-20, 2005, San Diego, CA, USA*, Ralph E. Johnson and Richard P. Gabriel (Eds.). ACM, 519–538. doi:10.1145/1094811.1094852

[10] David Chase and Yossi Lev. 2005. Dynamic circular work-stealing deque. In *Proceedings of the Seventeenth Annual ACM Symposium on Parallelism in Algorithms and Architectures* (Las Vegas, Nevada, USA) *(SPAA '05)*. Association for Computing Machinery, New York, NY, USA, 21–28. doi:10.1145/1073970.1073974

[11] Chris J. Cheney. 1970. A Nonrecursive List Compacting Algorithm. *Commun. ACM* 13, 11 (1970), 677–678. doi:10.1145/362790.362798

[12] George E. Collins. 1960. A method for overlapping and erasure of lists. *Commun. ACM* 3, 12 (1960), 655–657. doi:10.1145/367487.367501

[13] Luís Eduardo de Souza Amorim, Yi Lin, Stephen M. Blackburn, Diogo Netto, Gabriel Baraldi, Nathan Daly, Antony L. Hosking, Kiran Pamnany, and Oscar Smith. 2025. Reconsidering Garbage Collection in Julia: A Practitioner Report. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management, ISMM 2025, Seoul, Republic of Korea, 17 June 2025*, Martin Maas, Tim Harris, and Onur Mutlu (Eds.). ACM, 72–83. doi:10.1145/3735950.3735957

[14] David Detlefs, Christine Flood, Steve Heller, and Tony Printezis. 2004. Garbage-first Garbage Collection. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management (ISMM), Vancouver, BC, Canada.* ACM, New York, NY, USA, 37–48. doi:10.1145/1029873.1029879

[15] Damien Doligez and Georges Gonthier. 1994. Portable, Unobtrusive Garbage Collection for Multiprocessor Systems. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin (Eds.). ACM Press, 70–83. doi:10.1145/174675.174673

[16] Damien Doligez and Xavier Leroy. 1993. A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, Mary S. Van Deusen and Bernard Lang (Eds.). ACM Press, 113–123. doi:10.1145/158511.158611

[17] Robert Fenichel and Jerome C. Yochelson. 1969. A LISP garbage-collector for virtual-memory computer systems. *Commun. ACM* 12, 11 (1969), 611–612. doi:10.1145/363269.363280

[18] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. 2016. Shenandoah: An open-source concurrent compacting garbage collector for OpenJDK. In *ACM Proceedings of the International Conference*

on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools (PPPJ), Lugano, Switzerland, Walter Binder and Petr Tuma (Eds.). ACM, New York, NY, USA, 13:1–13:9. doi:10.1145/2972206.2972210

[19] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. 1998. The implementation of the Cilk-5 multithreaded language. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 212–223. doi:10.1145/277650.277725

[20] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. 2000. Soft updates: a solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.* 18, 2 (May 2000), 127–153. doi:10.1145/350853.350863

[21] Claire Huang, Stephen 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* (Cascais, Portugal) *(MPLR 2023)*. Association for Computing Machinery, New York, NY, USA, 85–99. doi:10.1145/3617651.3622986

[22] Graylin Jay, Joanne E. Hale, Randy K. Smith, David P. Hale, Nicholas A. Kraft, and Charles Ward. 2009. Cyclomatic Complexity and Lines of Code: Empirical Evidence of a Stable Linear Relationship. *J. Softw. Eng. Appl.* 2, 3 (2009), 137–143. doi:10.4236/JSEA.2009.23020

[23] Haim Kermany and Erez Petrank. 2006. The Compressor: concurrent, incremental, and parallel compaction. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation, Ottawa, Ontario, Canada, June 11-14, 2006*, Michael I. Schwartzbach and Thomas Ball (Eds.). ACM, 354–363. doi:10.1145/1133981.1134023

[24] B. Lang and F. Dupont. 1987. Incremental incrementally compacting garbage collection. In *Papers of the Symposium on Interpreters and Interpretive Techniques* (St. Paul, Minnesota, USA) *(SIGPLAN '87)*. Association for Computing Machinery, New York, NY, USA, 253–263. doi:10.1145/29650.29677

[25] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. 2013. Correct and efficient work-stealing for weak memory models. *SIGPLAN Not.* 48, 8 (Feb. 2013), 69–80. doi:10.1145/2517327.2442524

[26] Yossi Levanoni and Erez Petrank. 2001. An On-the-Fly Reference Counting Garbage Collector for Java. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2001, Tampa, Florida, USA, October 14-18, 2001*, Linda M. Northrop and John M. Vlissides (Eds.). ACM, 367–380. doi:10.1145/504282.504309

[27] Per Lidén and et al. 2018. ZGC: The Z Garbage Collector. https://wiki.openjdk.java.net/display/zgc/Main

[28] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (1960), 184–195. doi:10.1145/367177.367199

[29] Hongyu Miao, Heejin Park, Myeongjae Jeon, Gennady Pekhimenko, Kathryn S. McKinley, and Felix Xiaozhu Lin. 2017. StreamBox: modern stream processing on a multicore machine. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA, USA) *(USENIX ATC '17)*. USENIX Association, USA, 617–629.

[30] OpenJDK. 2023. G1 GC root scanning implementation. https://github.com/openjdk/jdk/blob/jdk-21%2B35/src/hotspot/share/gc/g1/g1RootProcessor.cpp

[31] OpenJDK. 2023. Root scannning implementation for Serial, Parallel and CMS GC. https://github.com/openjdk/jdk/blob/jdk-21%2B35/src/hotspot/share/gc/shared/genCollectedHeap.cpp#L704-L738

[32] OpenJDK. 2023. Shenandoah GC root scannning implementation. https://github.com/openjdk/jdk/blob/jdk-21%2B35/src/hotspot/share/gc/shenandoah/shenandoahRootProcessor.hpp

[33] OpenJDK. 2023. Transitive closure implementation in G1 GC. https://github.com/openjdk/jdk/blob/master/src/hotspot/share/gc/g1/g1YoungCollector.cpp#L537-L589

[34] OpenJDK. 2023. Transitive closure implementation in G1 GC. https://github.com/openjdk/jdk11u/blob/jdk-11%2B28/src/hotspot/share/gc/g1/g1CollectedHeap.cpp#L3161-L3186

[35] OpenJDK. 2023. Transitive closure implementation in G1 GC. https://github.com/openjdk/jdk11u/blob/jdk-11%2B28/src/hotspot/share/gc/g1/g1CollectedHeap.cpp#L4039-L4065

[36] OpenJDK. 2023. Transitive closure implementation in Parallel GC. https://github.com/openjdk/jdk/blob/jdk-21%2B35/src/hotspot/share/gc/parallel/psScavenge.cpp#L152-L200

[37] OpenJDK. 2023. Work stealing deque implementation in OpenJDK. https://github.com/openjdk/jdk/blob/jdk-21%2B35/src/hotspot/share/gc/shared/taskqueue.hpp#L296-L325

[38] OpenJDK. 2023. Work stealing in G1 GC. https://github.com/openjdk/jdk/blob/jdk-21%2B35/src/hotspot/share/gc/g1/g1ParScanThreadState.cpp#L319

[39] OpenJDK. 2023. Work stealing in Shenandoah GC. https://github.com/openjdk/jdk/blob/jdk-21%2B35/src/hotspot/share/gc/shenandoah/shenandoahMark.cpp#L177

[40] OpenJDK. 2023. Work stealing in ZGC. https://github.com/openjdk/jdk/blob/jdk-21%2B35/src/hotspot/share/gc/z/zMark.cpp#L520-L529

[41] OpenJDK. 2023. ZGC root scannning implementation. https://github.com/openjdk/jdk/blob/jdk-21%2B35/src/hotspot/share/gc/z/zRootsIterator.hpp

[42] Yoav Ossia, Ori Ben-Yitzhak, Irit Goft, Elliot K. Kolodner, Victor Leikehman, and Avi Owshanko. 2002. A Parallel, Incremental and Concurrent GC for Servers. In *Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 17-19, 2002*, Jens Knoop and Laurie J. Hendren (Eds.). ACM, 129–140. doi:10.1145/512529.512546

[43] Erik Österlund. 2025. Keynote: Industry GC Insights from OpenJDK. https://conf.researchr.org/details/ismm-2025/ismm-2025-papers/12/Keynote-Industry-GC-Insights-from-OpenJDK

[44] Tianle Qiu and Stephen M. Blackburn. 2025. Iso: Request-Private Garbage Collection. *Proc. ACM Program. Lang.* 9, PLDI, 874–896. doi:10.1145/3729285

[45] James Reinders. 2007. *Intel threading building blocks* (first ed.). O'Reilly & Associates, Inc., USA.

[46] Rifat Shahriyar, Stephen M. Blackburn, Xi Yang, and Kathryn S. McKinley. 2013. Taking off the gloves with reference counting Immix. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA), Indianapolis, IN, USA*. ACM, New York, NY, USA, 93–110. doi:10.1145/2509136.2509527

[47] Peter Styger. 1967. *LISP 2 garbage collector specifications.* Technical Report TM-3417/500/00. System Development Cooperation, Santa Monica. https://www.softwarepreservation.org/projects/LISP/lisp2/TM-3417_500_00_LISP2_GC_Spec.pdf

[48] The MMTk Team. 2019. V8 with MMTk support. https://chromium-review.googlesource.com/c/v8/v8/+/1928860

[49] The MMTk Team. 2024. JikesRVM with MMTk support. https://github.com/mmtk/jikesrvm/tree/f2d3178c2c74e8c8daeb105d98e48feac51dd44d

[50] The MMTk Team. 2024. Memory Management Toolkit. https://www.mmtk.io/

[51] The MMTk Team. 2024. MMTk's JikesRVM binding. https://github.com/mmtk/mmtk-jikesrvm

[52] The MMTk Team. 2024. MMTk's OpenJDK binding. https://github.com/mmtk/mmtk-openjdk

[53] The MMTk Team. 2024. OpenJDK 11.0.19+1 with MMTk support. https://github.com/mmtk/openjdk/tree/bc9669aaedc07924d08b939adedcad33f3e76065

[54] VPP Team. 2024. Vector Packet Processing Platform. https://s3-docs.fd.io/vpp/25.02

[55] Gil Tene, Balaji Iyengar, and Michael Wolf. 2011. C4: the continuously concurrent compacting collector. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, Hans-Juergen Boehm and David F. Bacon (Eds.). ACM, 79–88. doi:10.1145/1993478.1993491

[56] David M. Ungar. 1984. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, Pennsylvania, USA, April 23-25, 1984*, William E. Riddle and Peter B. Henderson (Eds.). ACM, 157–167. doi:10.1145/800020.808261

[57] Kunshan Wang, Stephen M. Blackburn, Peter Zhu, and Matthew Valentine-House. 2025. Reworking Memory Management in CRuby: A Practitioner Report. In *Proceedings of the 2025 ACM SIGPLAN International Symposium on Memory Management, ISMM 2025, Seoul, Republic of Korea, 17 June 2025*, Martin Maas, Tim Harris, and Onur Mutlu (Eds.). ACM, 109–121. doi:10.1145/3735950.3735960

[58] Bochen Xu, J. Eliot B. 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* (Brussels, Belgium) *(MPLR '22)*. Association for Computing Machinery, New York, NY, USA, 128–139. doi:10.1145/3546918.3546923

[59] Taiichi Yuasa. 1990. Real-time garbage collection on general-purpose machines. *J. Syst. Softw.* 11, 3 (1990), 181–198. doi:10.1016/0164-1212(90)90084-Y

[60] Wenyu Zhao. 2020. Work Packets #136. https://github.com/mmtk/mmtk-core/commit/b9dc15173c881861644f8f48d7abf2e1fc6b4fc7

[61] 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. doi:10.1145/3381052.3381320

[62] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2022. Low-latency, high-throughput garbage collection. In *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*, Ranjit Jhala and Isil Dillig (Eds.). ACM, New York, NY, USA, 76–91. doi:10.1145/3519939.3523440

[63] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2025. Latest Work Packets Artifact. https://doi.org/10.5281/zenodo.15760953

[64] Wenyu Zhao, Stephen M. Blackburn, and Kathryn S. McKinley. 2025. Work Packets Artifact. https://doi.org/10.5281/zenodo.15760954

[65] Wenyu Zhao, Stephen M. Blackburn, Kathryn S. McKinley, Man Cao, and Sara S. Hamouda. 2025. Advancing Performance via a Systematic Application of Research and Industrial Best Practice. *Proc. ACM Program. Lang.* 9,

OOPSLA2, Article 416 (Oct. 2025), 27 pages. doi:10.1145/3764118

Received 2025-03-25; accepted 2025-08-12