Advancing Performance via a Systematic Application of Research and Industrial Best Practice

WENYU ZHAO, Australian National University, Australia
STEPHEN M. BLACKBURN, Google and Australian National University, Australia
KATHRYN S. MCKINLEY, Google, United States
MAN CAO, Google, United States
SARA S. HAMOUDA[*], Canva, Australia

An elusive facet of high-impact research is translation to production. Production deployments are *intrinsically complex and specialized*, whereas research exploration requires stripping away incidental complexity and extraneous requirements to create *clarity and generality*. Conventional wisdom suggests that promising research rarely holds up once simplifying assumptions and missing features are addressed. This paper describes a productization methodology that led to a striking result: outperforming the mature and highly optimized state of the art by more than 10%.

Concretely, this experience paper captures lessons from translating a high-performance research garbage collector published at PLDI'22, called LXR, to a hyperscale revenue-critical application. Key to our success was a new process that dovetails best practice research methodology with industrial processes, at each step assuring that neither research metrics nor production measures regressed. This paper makes three contributions. i) We advance the state of the art, nearly halving the cost of garbage collection. ii) We advance research translation by sharing our approach and five actionable lessons. iii) We pose questions about how the community should evaluate innovative ideas in mature and heavily productized fields.

We deliver an optimized version of LXR. This collector, as far as we are aware, is the fastest general-purpose garbage collector for Java to date. On standard workloads, it substantially outperforms OpenJDK's default G1 collector and the prior version of LXR, while also meeting Google internal correctness and uptime requirements. We address all of the limitations identified in the PLDI'22 paper. We use this experience to illustrate the following key lessons that are concrete, actionable, and transferable. L1) A systematic combination of research and production methodologies can meet production objectives while simultaneously advancing the research state-of-the-art. L2) A benchmark suite cannot and need not contain facsimiles of production workloads. It is sufficient to replicate individual pathologies that manifest in production (e.g., allocation rates, trace rates, heap constraints, etc.) or configure the JVM to force a workload to manifest the pathology. L3) Productization experiences can strengthen research workloads; we upstreamed one such workload. L4) Production environment requirements are myriad and sometimes prosaic, extending well beyond those that can reasonably be addressed in a research paper. L5) This collector is not yet in production, reminding us that replacing core technology in industry is a challenging sociotechnical problem.

The artifact we deliver gives practitioners and researchers a new benchmark for high performance garbage collection. The lessons we enumerate should help academic and industrial researchers with actionable steps to close the gap between research paper results and industrial impact. The 10% 'productization dividend' the process delivered should spark discussion about how our field evaluates innovative ideas in mature areas.

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

Additional Key Words and Phrases: Garbage Collection, Productization, Performance Optimization

[*]Work done while employed by Google Australia.

ACM 2475-1421/2025/10-ART416
https://doi.org/10.1145/3764118

ACM Reference Format:
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 (October 2025), 27 pages. https://doi.org/10.1145/3764118

1 Introduction

Translating research ideas to production can be slow, even for the most promising work. It took five years from when LLVM was published [12] until its C front-end, Clang, was first considered *'production quality'* per the LLVM version 2.6 release notes [22]. It took five years for the G1 garbage collector to go from publication to shipping [8, 17], and another six years before it was considered *'a fully-featured garbage collector, ready to be the default'* [10, 11], and further two years before it was released as OpenJDK's default garbage collector — a thirteen year process. Slow translation reduces research impact and holds back industrial innovation.

Despite its importance, the literature rarely discusses research translation, in terms of the process, or the likely impact. For example, neither LLVM nor G1 have publications capturing their productization journey despite their industrial importance and the substantial innovations and performance improvements that took place along the way [11, 22]. Nor are we aware of any work that quantifies the impact of productization on performance. This experience paper addresses both. It records the productization journey, that is still ongoing, of a recently published research garbage collector in a challenging industrial setting.

The difficulty of research translation is rooted in differing priorities and objectives. Good research will clearly demonstrate the value of an idea while minimizing unnecessary engineering effort in doing so. On the other hand in industry, failure to a deliver secure, robust, and performant system means failure of the project. The former requires pruning out extraneous complexity to focus on clarity and generality while the latter must rigorously consider every corner case in all its complexity, sacrificing extraneous generality in often highly specialized, opaque, proprietary, and inherently non-reproducible settings.

In this paper, we take LXR through a productization process. LXR is a research garbage collector published at PLDI'22 [29]. Our target application is a large performance and revenue-critical Google service, which we will call LSJ (a Latency-Sensitive Java application). LSJ's requirements include supporting a very high allocation rate, very heavy use of class loading and unloading, and high occupancy of 32 GBs of available virtual memory in a compressed pointer Java heap. The default production OpenJDK collectors often struggle with such demanding workloads. For example, Uber noted high allocation rates induced large pauses on their data query engine using G1 [25]. They configured G1 with a large to-space reservation and forced concurrent marking more often to reduce, but not eliminate completely, substantial mature-space GC pause times and frequent full GCs. Similarly, LSJ configures a specialized version of G1. Because of the high cost of concurrent full (major) mature space collections, they are disabled for LSJ. Instead, user requests are regularly diverted away from LSJ instances and LSJ explicitly triggers a G1 major collection only when it is not receiving traffic. This workaround avoids impacting user latency at the cost of replicating services. Although LXR appeared to be a promising alternative to this ad hoc workaround, LXR was a research artifact, untested in a production setting.

Advancing the State of the Art. The result of our work is an optimized LXR collector in OpenJDK that improves total execution time by 11% compared to the default state of the art G1 production collector, at a moderately sized heap, across a wide range of workloads. We are not aware of any prior work that has made an improvement of this magnitude to the widely-used, highly-optimized

OpenJDK runtime. Notably, this improvement *did not* come from a particular major innovation. Instead it came from many minor innovations and a systematic productization methodology.

Productization Methodology. We develop a two-part methodology that combines standard research methodology with the practices of a mature production software engineering team. The first element is incremental refinement, initially applied to correctness and missing features and later to performance. At each step, we implement a fix then evaluate it, iterating until the desired change is achieved *without regressing research or production metrics*. The second element is to systematically map pathologies observed in production to standard workloads where they can be far more easily debugged. We first identify a symptomatic metric, such as high allocation rate, slow trace rate, limited heap headroom, etc., and then use it to find a workload that manifests the pathology, if necessary reconfiguring the JVM to expose the problem.

Productization Dividend. The shift from the 1% advantage reported by Zhao et al. to the 11% advantage we show here amounts to a *productization dividend* of 10%. We found this both surprising and disconcerting. Had the original LXR been just 2% slower and lagged G1 by 1% (instead of leading it), the work likely would not have been published at PLDI nor attracted the resources required for productization. In this hypothetical, an idea with the potential to improve over the state-of-the-art by 9% never sees the light of day. Work that incrementally advances an established production system will tend to profit from a built-in productization dividend, while entirely new work will not. This suggests that the peer review process and its natural desire to see improvements over the state-of-the-art will implicitly favor incremental work over completely fresh work.

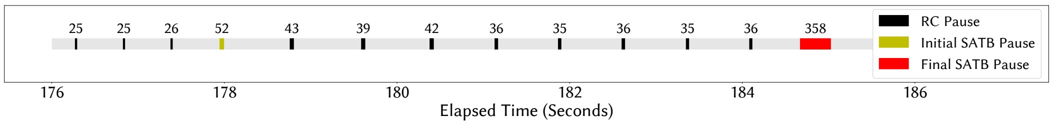
The current status of our project is that LXR passes all internal JDK tests, and can run in the data center receiving bifurcated live traffic while meeting uptime and throughput objectives. The process is ongoing and LXR is not yet serving live traffic. We are working through an unrelated rebase to JDK 21 and have yet to meet all our latency objectives. Nonetheless, the process of translating LXR to production highlights the many challenges due to differing priorities and objectives in research and industry. We identify and discuss five lessons that we have drawn from our experience in Section 7. This productization effort has already led to a significant advance in the state of the art for garbage collection performance, new methodologies for systematically translating research to production, and new lessons for research translation.

2 Background and Related Work

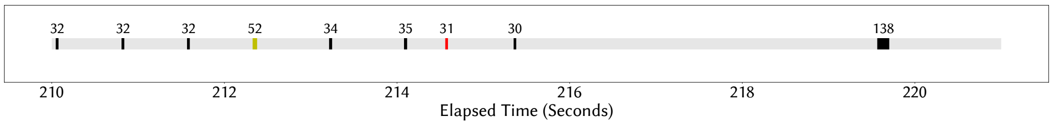
2.1 OpenJDK's Production Collectors

OpenJDK is the most widely used Java virtual machine today. It is actively maintained with contributions from major vendors and the open source community. The Garbage-First (G1) collector [8] has been the default in OpenJDK since 2017. It manages the heap in fixed-sized regions, is generational and uses concurrent marking to minimize GC pauses. OpenJDK ships with four other standard GCs. Our focus is on the new research collector, LXR, and OpenJDK's default collector, G1, a modified form of which is used by LSJ today. Our production environment notably configures G1 with a lower overhead write barrier and a system for dynamically shifting load away from LSJ servers before they trigger major garbage collections. LSJ's requests are thus not exposed to G1's costly major collections.

Garbage collectors, like many system research artifacts, require massive effort to become production ready. For example, G1 underwent thirteen years of optimization before becoming the default in OpenJDK. G1 and ZGC did not introduce full class unloading support until after their initial release [10, 18]. Although class unloading is a fundamental JVM feature, a lack of support for it



(a) LXR GC pauses **without** our optimizations.



(b) LXR$_O$ GC pauses **with** new optimizations.

Fig. 1. **Our optimizations reduce LXR's pauses.** Traces of LXR's GC pauses on a 10-second execution window for h2-large on a 2× minimum heap (20.3 GB). We label each GC pause interval with its duration in milliseconds. Timeline (a) lacks optimizations (Section 3 and 4) and timeline (b) includes them.

does not compromise garbage collection *correctness*, just efficiency. For complex workloads such as LSJ, efficient class unloading is vital. Consequently, adding support for class unloading to LXR was part of our productization process. Sections 3 and 4 discuss this and other improvements.

2.2 LXR and MMTk

LXR differs substantially from the other OpenJDK production collectors. i) Rather than only rely on tracing, LXR uses high performance reference counting (RC) [21, 29] to reclaim both young and old garbage at every RC pause. ii) It uses an inexpensive write barrier [29] that combines reference counting [14], remembered set maintenance, and concurrent tracing with Snapshot-At-The-Beginning (SATB) [26]. iii) It avoids most copying by using the Immix heap structure (a hierarchy of fixed-sized *blocks* divided up into fixed-sized *lines*) and opportunistic copying versus more costly region evacuation. iv) It is built on top of the MMTk garbage collection framework [15] in OpenJDK.

Figure 1(a) illustrates LXR's behavior using a trace of the h2-large benchmark running in 20.3 GB heap. In the common case, it performs regular very short RC pauses to efficiently reclaim both young and acyclic mature garbage, as shown by the black intervals. It relies on occasional concurrent marking to reclaim cyclic garbage. The gold and red intervals represent the pauses for the start and the end of the SATB marking cycles, respectively.

LXR uses final SATB pauses as an opportunity to perform defragmentation. (1) It evacuates (i.e. copies) live objects out of fragmented memory blocks (from-space blocks), compacting them in other memory blocks, and then (2) releases the now defragmented memory. LXR constructs a remembered set during concurrent marking which remembers all references into blocks targeted for defragmentation. The remembered set allows defragmentation to be tightly focussed, avoiding costly traversal of the whole heap.

Zhao et al. demonstrated that on DaCapo benchmarks, with 2× the minimum heap, LXR achieved 4% higher throughput and 8% lower 99.99% tail latency than the default G1 GC.[1] These results motivated our effort to productize LXR. We refer to the resulting optimized LXR as LXR$_O$. Figure 1(b) illustrates how LXR$_O$ exhibits significantly shorter marking cycles and reduces final SATB pause times. Section 6.2 shows that LXR$_O$ substantially improves overall performance over LXR.

[1]We performed the same comparison in a slightly different environment and saw 1% and 12% respectively (Section 6.2).

2.3 Research and Industrial Development Methodologies

Development methodologies vary greatly, and we will not attempt to categorize them fully. However, the approach that we found so successful in translating a research artefact to production borrows on elements of these, so we'll briefly outline them here.

Performance-oriented research tends to be driven by the goal of making a clear demonstration of the efficacy of a novel idea. Evaluation is often framed with respect to the prior state-of-the-art over a set of established workloads. A common pattern is to develop a prototype that is (only) sufficiently robust to run the workloads without failure. The next step is to engage in a process we call *performance optimization whack-a-mole*: i) Evaluate the prototype with respect to the prior work. ii) If overall performance delivers a publishable result, write it up and you are finished! iii) Otherwise, identify the simplest workload(s) on which the prototype most egregiously under performs, investigate, and fix. iv) Return to step i). Sometimes the performance goal may never be met, leading to abandonment and perhaps a negative results paper. Note that the performance delivered by this approach is entirely contingent on the comparison with prior work on standard workloads. The approach therefore may not expose the true potential of an idea, simply because the researchers seek to be efficient in this time-consuming phase, so will tend to stop as soon as the process reaches a clear result or exhaustion, which could be well short of the ideas's full potential.

Industrial development will tend to differ in two key respects. The system will first be subjected to *extensive* and rigorous correctness tests, perhaps tens of thousands of unit tests — the project cannot proceed until all correctness tests pass. The system will then be performance optimized, but unlike the research setting, optimizations typically will not use a set of standard benchmarks, but instead will target the intended workload, which may be large and inherently non-deterministic. The system is subjected to load testing, which is similar to the intended deployment workload, but more contained and manageable. In our system, production evaluations use a deployed mirror instance which is given duplicated traffic and its responses are discarded. If the proposed changes meet the performance goals, the new system will roll out to some percentage of live traffic. If successful, it will then be fully deployed. The criteria are often in absolute performance terms (a certain query-per-second (QPS), for example) and various business-derived expectations, rather than relative to those in a previous research paper.

Section 3 describes our approach and how it brings together the above methodologies.

2.4 LSJ and Our Production Environment

We evaluate the LXR GC on a large-scale, externally-facing latency-sensitive proprietory production Java workload, LSJ. During testing, LSJ instances run in the datacenter receiving duplicated live traffic. This setup is a common industrial practice, but limits our ability to measure and debug.

Shared Hardware. Instead of a dedicated physical machine, each LSJ instance shares CPU and memory with other production services. This setting highlights the importance of opportunity cost — memory and CPU that we save can be utilized by other services or other LSJ instances. Huge page availability is one example of how this setting impacts performance engineering. OS huge page availability tends to diminish over time and we do not have the luxury of rebooting the machine to make huge page availability more deterministic, as we might have in a research setting.

Data Confidentiality. LSJ requests contain user-sensitive data, and thus we may not inspect the process state or examine how different request types impact LXR. We only observe overall performance and prescribed telemetry.

JVM Configuration. LSJ configures the JVM and G1 GC in a highly specialized manner. It uses compressed pointers [13] and a heap size almost equal to the 32 GB compressed pointer address space, introducing the challenge of *virtual* memory exhaustion. LSJ carefully configures G1 with a large young generation. In production, it disables concurrent marking because full-heap collections significantly degraded tail latency. Instead, LSJ is specially configured to periodically stop sending requests to each LSJ instance and then force a full heap collection. G1 then performs an expensive stop-the-world full heap collection without impacting tail latency of any requests. Our goal is for LXR to outperform this G1 configuration *without* performing traffic redirection, eliminating these additional LSJ instances.

Class Loading. LSJ frequently loads new classes into the JVM, putting high pressure on the meta-space (the space for storing class metadata) and thus the garbage collector. Class unloading was not supported by the original LXR.

Nondeterminism. The nature of LSJ means that it does not need to process requests in a reproducible way, resulting in potentially different responses for identical requests. Moreover, the number of user requests directed to LSJ is a fixed proportion of the total received during any given time window. Thus its QPS (query-per-second) is influenced not only by GC performance but by time of day and noticeably varies between peak and off-peak periods. QPS is a critical performance metric for LSJ. These uncertainties exacerbated the noise in our evaluations, making GC behavior harder to reason about.

Many of the above environmental factors are likely shared by other very large scale industrial services. We therefore believe that our experiences and lessons generalize beyond this concrete experience with LXR and LSJ.

3 Our Methodology for Performance Debugging in Production

This paper reports a productization process that led to a striking performance improvement. At the heart of the methodology are two simple ideas:

(1) iterative refinement combining both research and industrial development practices, and
(2) systematic mapping of production pathologies to metrics matched in manageable workloads that we then use for debugging.

Iterative Refinement. The classic iterative refinement process is one of repeatedly and systematically following each fix with testing that is designed to ensure that fixing one metric does not regress another. Our twist on this approach is to overlay the metrics applicable to our industrial target with the research metrics associated with the original research. We thus integrate conventional iterative development processes with performance optimization whack-a-mole (Section 2.3). We first applied this approach to the exhausting and relatively unexciting process of addressing LXR's shortcomings using Google's large internal JVM test suite, fixing bugs, and adding missing features until we had LXR$_F$, a fully featured version of LXR that passed all tests. The feature-complete LXR$_F$ performed 3.1% better than G1 (Table 4) on a 2× heap, a modest improvement over the 1% advantage we saw in our initial evaluation of LXR (Table 3). We next applied the refinement process to the task of reaching our performance objectives on the LSJ workload, iteratively addressing performance challenges exposed by LSJ and reevaluating on the standard workloads to ensure performance did not regress. Ultimately this process delivered a complete, compliant, optimized LXR, LXR$_O$, that stretched LXR's performance advantage further, to 10.9% (Table 4).

Mapping Pathologies to Metrics Exhibited in Other Workloads. The practicalities of a hyperscale industrial workload such as LSJ make performance debugging challenging (Section 2.4). To address this challenge, we employ a pattern of systematically finding pathologies in LSJ, determining a symptomatic metric that reveals the problem, and then use that metric to identify a workload with similar symptoms. We use the DaCapo benchmarks [3, 4] for these workloads as well as some internal workloads that are amenable to desktop debugging and analysis. The DaCapo benchmark suite is diverse and comes with nearly fifty metrics characterizing each benchmark's behavior, which made the process of mapping pathologies to benchmarks easier. It is important to stress that our methodology *does not* depend on having a benchmark suite containing a workload similar to the target industrial workload. Instead, we map *individual pathologies* to benchmarks, so the requirement on the benchmark suite we use is just that there exist one or more benchmarks that exhibit each of the production pathologies we identify. Note that the problem may not necessarily be on the critical path of the benchmark in its default configuration. Sometimes, we found it was necessary to adjust JVM options in order to push a benchmark to a corner case that captured the pathology clearly. Improving the problematic metric may or may not improve the target benchmark in its default configuration, but it almost always improved LSJ. Table 1 lists examples of identifying problems, metrics that reveal the underlying pathology, and mapping them to workloads.

3.1 Systematically Mapping Pathologies from LSJ to Manageable Workloads

Table 1 lists six concrete examples of the pathology mapping process, listing each problem, its symptoms, the metric/s we used, the workloads that exhibited the same problem, and the section where we discuss its resolution. The first problem (Slow SATB) is slow snapshot-at-the-beginning pauses (Section 2.2). The symptom was unacceptable latency in LSJ from expensive final SATB pauses due to concurrent tracing not keeping up with LSJ's allocation rate. The metric is SATB tracing rate. We quickly observed that the h2 DaCapo benchmark exhibited the same behavior in tight heaps. We then debugged this complex issue using this well-understood benchmark. Note that Zhao et al. [29]'s performance analysis does not expose this problem, yet it is a major problem for LSJ. Figure 1 illustrates the problem and fix and Section 3.2 explains our optimization.

We handled all problems in Table 1 similarly. We identify a problem with LSJ, narrow down its symptoms, identify key metrics, and map it to a standard workload that exhibits the same problem. For virtual memory fragmentation, we did not have any workloads whose minimum heap size was close to the 32 GB of virtual memory available with compressed pointers. However, we easily modified the h2 benchmark to give it a much larger heap footprint for debugging. We think that workloads with such large footprints are valuable and successfully upstreamed our modified h2.

In addition to capturing the major problems we identified with LXR, Table 1 highlights that: i) we do not need a silver bullet — we do not require a facsimile of LSJ among the standard benchmarks, and ii) Zhao et al. left performance on the table, as evidenced by diversity of benchmarks affected by these performance problems and their subsequent improvements.

3.2 Case Study: SATB Performance

We use the slow SATB problem from Table 1 as a case study to illustrate our process concretely. GC logs revealed that LXR's Snapshot-At-The-Beginning (SATB) concurrent marker could not keep up with the application's allocation rate. This problem is a classic one for concurrent collectors. Because a concurrent marker cannot free any space until it completes its trace of the entire heap, it must trace at a rate that can keep up with the application's allocation demands, otherwise it will force the application to stall. LXR, like other collectors, tries to avoid this situation but if it does arise, it falls back to an expensive stop-the-world full heap collection, harming both throughput and latency. Further analysis revealed that the slowdown was not attributable to poor load balancing or GC task scheduling, pointing to the performance of the main tracing loop as the likely issue.

Mapping to a Benchmark. We found similarities between LSJ and the `large` variant of the DaCapo Chopin h2 benchmark, including their memory use, allocation rate, and object size demographics. We modified h2's configuration to match LSJ's very large heap footprint. With this change, we

Table 1. **Problems we observed in LSJ using LXR**, how they manifest (symptoms), a metric that reveals the problem, the workloads with similar metrics that we use for debugging, and cross references to solutions.

LSJ Problem	Symptom	Metric	Workload	Soln.
Slow SATB	Unacceptable latency due to long final SATB pauses when LXR can not finish concurrent tracing before heap exhaustion. LXR thus pushes lots of marking into the final mark pause. Surprisingly, this pause grows larger than an emergency GC pause, which scans all live heap objects. This pathology indicates LXR concurrent marking is slower than stop the world (STW) tracing.	SATB tracing rate: number of objects traced per unit time	fop, h2, lusearch, tomcat	§3.2
Virtual memory fragmentation	LSJ runs out of memory and crashes on a large memory allocation in the large object space (LOS) when the Java heap is not full, revealing inter-chunk fragmentation as the LOS requires contiguous pages.	Large minimum heap size (minheap): configure very large minheap workload	h2	§4.2
Block sweeping is slow	We observe LXR's sweeping phase on large heaps is slow and does not scale well because LXR sequentially sweeps all the young blocks during a pause. The very large heap size for LSJ exposes this problem. Instead of parallelizing the sweep, LXR_O uses lazy sweeping, where each mutator sweeps its own blocks.	Stop the world (STW) time	lusearch, h2	§4.3
High resource unloading time	MMTk's phase timers show that apart from slow SATB tracing, LXR's final mark pause on LSJ is dominated by resource unloading, especially the weak reference processor and class unloading.	STW and class unloading time	jython, kafka, tradebeans	§4.4 §4.5
GC trigger tuning	In GC logs, we find LXR triggers very frequent GCs that led to high CPU overheads for LSJ.	Allocation rate, survival ratio, GC frequency (SATBs and emergency GCs)	h2 and xalan	§4.5
High resident set size (RSS)	LXR crashes quickly due to high RSS usage when running on a different internal workload.	RSS usage: use bpftrace to dump RSS usage over time	fop	§4.5

reproduced the concurrent tracing problem in LSJ in h2. Figure 1(a) illustrates the original problem. The gap between the initial SATB pause (gold) and the final pause (red) is nearly seven seconds, spanning 8 minor RC collections, and the final pause is 358 ms.

Precisely Identifying the Issue/s. Precisely measuring concurrent collector performance is complicated by the fact that (by definition) the application is running concurrently, making cost attribution difficult. We addressed this problem by (temporarily) modifying LXR to pause all application threads while the SATB trace runs. Measurements of this setup precisely identified five distinct problems: i) a duplicate load when marking, ii) a dynamic dispatch in the hot loop, iii) an expensive mmap check, iv) a suboptimal reference count check, and v) suboptimal loop ordering.

The first three are relatively straightforward performance bugs, which were likely missed because tracing performance was not previously as stressed, but they were exposed by LSJ and our modified h2. The last two issues are more interesting, so we discuss them next.

For the fourth problem, it is necessary to explain details of the implementation of important optimization introduced by the original LXR: the exclusion of young objects from the SATB trace [29]. The implementation exploits the fact that LXR maintains the invariant that all young objects have zero reference counts until they are promoted to mature objects [20]. Thus by checking their reference counts, LXR can identify and skip young objects when performing the SATB trace.

LXR employs a densely-packed table (metadata stored off heap) to track per-object reference counts. Every two bits in this table map to a 16 B 'memory cell' in the heap, which is also the minimum size of a Java object. Thus, each Java object maps to a unique 2-bit RC counter in the table, irrespective of its actual size and alignment. This mapping ensures that one byte in the RC table maps to a 64 B memory slice in the heap, which may contain up to four Java objects. Hence, checking the RC for each object involves: i) loading the relevant byte from the RC table, ii) extracting the appropriate bits, and iii) comparing them with zero. These operations when executed within the tracing hot loop can be expensive. However, we observed that the nature of the Immix allocator prevents the interleaving of young and mature objects within the same 256 B Immix line. Each Immix line maps to four bytes in the RC table. This implies that, in step (i) above, the four RC counters in the loaded byte would either be all zeros for young objects or all non-zeros for mature objects. This insight allows us to simplify the RC check to just a load and a compare with zero, eliminating the bit extraction step.

To fix the last problem, we adopted the observation from Garner et al. [9] that inverting the order of the traditional tracing loop can counter-intuitively yield a performance benefit. Instead of enqueuing objects for scanning after they are successfully marked ('grey'), Garner et al. enqueue all children objects ('white') for marking and scanning, improving locality at the expense of more queuing operations. They refer to this approach as *edge enqueuing*. We adopt this approach and optimize LXR by enqueuing 'white' objects instead of 'grey' objects, further improving the marking performance.

Evaluation. Together, the five tracing optimizations described above improve LXR's SATB tracing rate by 63.0%. Figure 2 shows the impact of each of the five optimizations on the SATB tracing

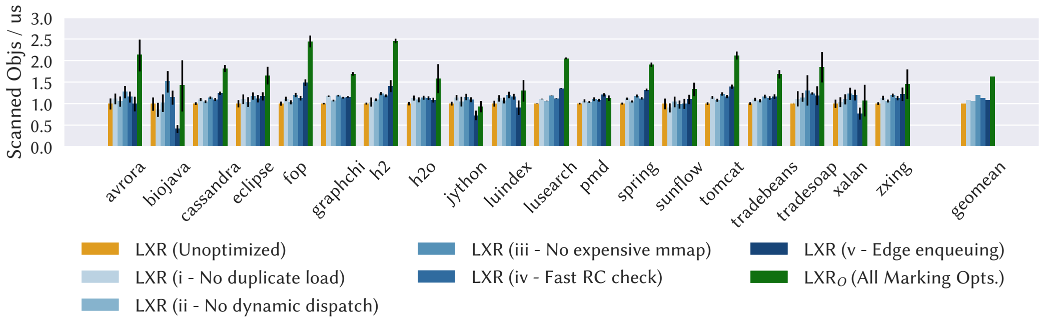


Fig. 2. **Tracing rate for concurrent marking optimizations.** We show the number of scanned objects per microsecond, normalized to the baseline without any optimizations (higher is better). The yellow bar is LXR without any optimizations. The middle five bars show LXR with a different single optimization for incremental comparison. The last bar shows the cumulative improvement from all optimizations. We exclude three benchmarks from the trace rate measurements as they do not perform an SATB trace.

rate on the DaCapo Benchmarks, following the methodology discussed in Section 5.[2] The five optimizations contribute to tracing rate improvements of 8.2% (no duplicate load), 6.4% (no dynamic dispatch), 19.6% (no mmap), 13.2% (fast RC check), and 8.2% (edge enqueuing), averaged across the benchmarks. No benchmark saw an overall regression larger than the 95-percent confidence intervals. The only optimization to regress any benchmark was edge enqueuing, which slowed down biojava, jython, luindex, and xalan by 58.5%, 27.3%, 9.3%, and 22.9% respectively. However, all together they improve the average tracing rate by 63.0% and together never degrade the tracing rate. h2 showed a tracing rate improvement of 145.5%, which is perhaps unsurprising given that h2 was our target benchmark for the tracing optimizations.

Figure 1(b) illustrates the pause reduction from these optimizations. The final pause is an order of magnitude shorter (31 ms v 358 ms), the total duration of the SATB is reduced from about 7 s to about 2 s, and the number of intervening RC pauses is reduced from 8 to 2.

After verifying each optimization on the DaCapo benchmarks, we applied the fix to the production setting. The optimizations are similarly successful for LSJ— the SATB trace and final pause now keep pace with LSJ's allocation rate, eliminating long pauses.

The SATB performance case study highlights the importance of mapping pathologies from the production setting onto standard workloads which we then performance debug using standard techniques. We emphasize that at each step, we evaluated each potential change against the entire DaCapo benchmark suite and did not press forward until we were able to deliver an overwhelmingly consistent improvement. We constantly monitored the performance of LSJ, inspecting GC logs and monitoring server QPS, CPU usage and memory usage for each change.

4 Major Issues Addressed During Productization

Having outlined our approach and used SATB performance as a case study (Section 3.2), we now briefly discuss the other major issues we addressed during the productization process: i) missing JVM features, ii) virtual address space fragmentation, iii) block allocation and sweeping, iv) VM resource unloading, and v) noteworthy secondary issues.

4.1 Missing JVM Features

The original LXR did not support three key JVM features: compressed pointers, weak reference processing, and class unloading. While these features do not affect correctness, the performance requirements of our production environment makes them essential. For instance, LSJ performs frequent class loading and unloading to apply dynamic software updates, which imposes significant memory pressure on the meta-space. Given strict memory constraints, a lack of class unloading rapidly exhausts the meta-space. We implement all these three missing features in MMTk and LXR.

Compressed Pointers. OpenJDK's pointer compression improves heap utilization by using 32 bits to represent 64-bit pointers [13, 19]. In its default implementation, OpenJDK's compressed pointers exploit eight-byte object alignment to offer a 35-bit (32 GB) effective address space. When OpenJDK's compressed pointers are enabled, all intra-heap pointers are compressed. However, the runtime may hold a mixture of compressed and uncompressed pointers. As a consequence, when we enumerate roots from OpenJDK and enqueue them for LXR, we need to introduce tagging so that LXR will know whether the root holds a compressed or uncompressed value. We were able to implement this tagging and the associated checks with no observable performance impact.

Class Unloading. LXR's class unloading implementation follows G1's, which marks live OpenJDK class metadata during the transitive closure and unloads dead metadata after each full heap trace.

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

Similar to G1, class unloading occurs under two conditions: at the end of each full heap trace, or when a meta-space GC is triggered.

Weak Reference Processing. Java has three weak reference semantics: WeakReference, SoftReference, and PhantomReference. LXR piggybacks all weak reference processing on full heap traces, treating them as strong during RC pauses. Like other OpenJDK GCs, LXR discovers weak references during the transitive closure, and unloads them when necessary at the end of each full heap trace.

We implemented each of these missing features without any great difficulty, but all of them were time-consuming engineering exercises. By following our methodology, we were able to implement each of them without regressing performance, as we show in Section 6.2.

4.2 Virtual Memory Fragmentation

We found that on LSJ, using LXR with a 31 GB heap, the server would run out of memory due to a large object space (LOS) allocation failure within 10–30 minutes after boot. Notably, the out of memory failure was *not* due to the heap being full, but due to MMTk being unable to find sufficient contiguous virtual memory to satisfy the large object allocation request. Counterintuitively, the behavior could be mitigated by *reducing* the heap size. Recall that in OpenJDK, with compressed pointers, the available address space is 32 GB by default.

Mapping to a Benchmark. Initially, none of the DaCapo benchmarks exhibited this pathology, however, as mentioned in Section 3, we soon found that by increasing the size of the h2 benchmark, which was just a matter of changing some configuration parameters, we could readily reproduce the problem by running h2-large with heap sizes near the 32 GB virtual memory limit.

Precisely Identifying the Issue/s. We soon realized that compressed pointers and our management of virtual memory were the key concerns. Specifically, it was the combination of LSJ's heap using most of the available 35-bit address space and our use of MMTk's discontiguous spaces. MMTk supports two strategies for mapping virtual memory to distinct policy domains, or *spaces*, inherited from its earlier incarnation in JikesRVM [1]: *discontiguous* and *contiguous*. Both mechanisms provide a way to allocate address space and an efficient way to map from object addresses to the space/policy to which the object belongs. For 32-bit address spaces, MMTk uses the discontiguous strategy: it breaks the usable address space into 4 MB 'chunks', and uses a table to map chunks to spaces. For 64-bit address spaces, it uses the contiguous strategy, assigning each space 4 TB of contiguous virtual memory, allowing it to cheaply identify each space by masking pointer bits.

When we implemented compressed pointers (Section 4.1), we adopted MMTk's discontiguous strategy. In practice, LXR uses just two MMTk spaces: the Immix space, which manages small objects and acquires only one 4 MB chunk at a time from the global virtual memory resource, and the large object space (LOS), which may allocate multiple contiguous chunks in a single allocation. This mixing of multi-chunk and single-chunk allocations will eventually fragment the address space. Unless there is abundant address space headroom, eventually the heap may not have a contiguous hole of sufficient size to accommodate a large object space allocation request.

One way to address this would be to more aggressively defragment the Immix space, creating larger holes for use by the LOS. However, this solution is costly because it requires extensive copying, which LXR is designed to carefully minimize. Copying for defragmentation is so expensive in the case of G1's full heap compactions that LSJ must redirect traffic while it is happening.

Instead, we address the problem with two strategies. i) We make Immix and the LOS use the address space from opposite ends, avoiding interleaving them. ii) We increase virtual memory headroom by reducing the heap size given to LSJ. We determined via bisection search that LSJ has a minimum heap size of 8 GB. It is normally configured to run at nearly 4× this size (31 GB), but we reduced it to 2× (16 GB). This change reduces heap headroom from 23 GB to 8 GB but increases virtual memory headroom from 1 GB to 16 GB. Although this change very effectively addressed the problem with LSJ, we are exploring additional strategies to deal with applications with minimum heap sizes closer to the 32 GB limit.

Evaluation. Together, these approaches solved the problem of out of memory failures for LSJ. In Section 6.3 we conduct a lower bound overhead (LBO) analysis [6], demonstrating that LXR is more space-efficient than G1, explaining why we could reduce the heap size for LSJ so successfully.

4.3 Block Sweeping and Allocation

While inspecting GC logs from LSJ, we found that LXR spent considerable time sweeping blocks at the end of each stop-the-world pause. This sweep identifies blocks that have become free during the collection.

Mapping to a Benchmark. We noted that the cost and importance of block sweeping grows as allocation rate increases and young object survival decreases. With this observation in mind, we quickly found that lusearch and h2, the two highest allocation rate, low survival DaCapo benchmarks (Table 2) manifest the same problem, which we had not noticed previously.

Precisely Identifying the Issue/s. Like many GCs, LXR minimizes synchronization during allocation by using thread-local allocation buffers. When a buffer is exhausted, LXR sources a new block from a global pool. Our investigation revealed two related issues. (1) We noticed that block sweeping time is proportional to the young allocation volume, significantly increasing GC pause time on large heaps. (2) We also noticed that LXR's block allocation strategy necessitates global synchronization, leading to poor scalability on parallel workloads with high allocation rates.

We tackled these two issues at once by introducing a new block allocator that manages blocks hierarchically: global blocks and thread-local blocks. Mutator threads perform thread-local block allocations for the common case, and only jump to the global allocator if thread-local block allocation is unsuccessful. Each mutator thread holds a local block list containing blocks which it currently owns. After each RC pause, blocks in the list may either be full, partially free, or completely free. When allocating a block, the mutator traverses this list, and sweeps each block to find available lines. This approach allows us to remove GC-time block sweeping to reduce pause time. We also observe that most young blocks are strictly evacuated and become completely free after a RC pause. We optimize for this case, skipping mutator-time sweeping.

On local allocation failure, the local allocator grabs N blocks at a time from the global allocator. By default N is set to 32. The global allocator holds all blocks in the heap in a global list. On the fast path, it traverses the list to identify blocks without owners. If it fails to find a sufficient number of blocks, it traverses the list again and steals blocks from other mutators. By employing stealing, the allocator efficiently improves block utilization, preventing inactive mutators from holding blocks without using them. Finally, when the global allocator fails to find or steal any blocks, it transitions to a synchronized slow-path that expands the heap and gets more blocks from the operating system.

This approach optimizes the GC and block allocator in three ways. i) Thread-local and global block allocation are lock-free for the common case, significantly improving mutator performance. ii) Mutators tend to reuse thread-local blocks when possible, enhancing locality and performance. iii) Mutators sweep the blocks in their local list to find blocks for allocation, thereby eliminating the need for block sweeping during GC pauses, significantly reducing GC pause time.

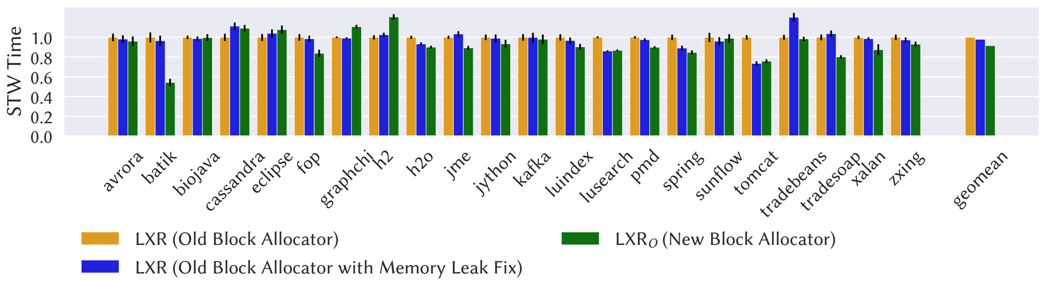


Fig. 3. **Different block allocators and their effect on total stop-the-world (STW) time**. We compare LXR with the old block allocator, the old block allocator with the memory leak fix (§4.5), and LXR with the new block allocator, normalized to the baseline with the old block allocator.

Evaluation. Figure 3 shows a significant reduction on stop-the-world time with LXR's new block allocator.[3] It shows three different block allocators: the old block allocator, the old allocator with the memory leak fix (§4.5), and the new block allocator. It normalizes results to the minimum value. Note that the new allocator contains the memory leak fix. The memory leak fix leads to a reduction in the number of stop-the-world (STW) pauses by 5.1%, which resulted in a 2.0% STW time reduction. The new allocator also significantly improved mutator utilization. LXR had a 10.0% decrease in number of STW pauses, and 8.4% reduction in STW time. The extreme case was batik, which had a substantial STW time reduction of 45.7%. However, on four benchmarks: cassandra, eclipse, graphchi, and h2, we observe a STW time increase of 9.2%, 7.8%, 10.8%, and 20.6% respectively. Upon detailed inspection of the worst case, h2, we discovered that the new allocator's block allocation order changed heap fragmentation for h2, leading to 42.9% more concurrent marking epochs. The extra cost of performing mature evacuation at the end of each concurrent marking leads to higher STW time overhead. This result indicates that the new block allocator may degrade GC scheduling behavior for LXR's GC trigger, which is less-tuned in production settings. Although GC time is reduced for most benchmarks, we did not observe any performance change on mutator time.

4.4 Weak Processor Performance

LSJ's GC logs revealed that LXR's final mark pause was dominated by OpenJDK's *WeakProcessor* and class unloading. We discuss *WeakProcessor* below and briefly mention class unloading in Section 4.5.

Mapping to a Benchmark. We examined work distribution in final mark pauses for DaCapo benchmarks and found three DaCapo benchmarks that also suffered significant *WeakProcessor* overheads in the final mark pause: jython, kafka, tradesoap.

Precisely Identifying the Issue/s. OpenJDK maintains references into the Java heap for a variety of reasons. Many of these must be weakly held to avoid the runtime keeping application heap objects alive as a side effect of referencing them. The are implemented as handles which are maintained by the *WeakProcessor*. (An example is in the implementation of JNI's Weak Global References, which allow weak references into the Java heap from JNI.) When the garbage collector moves or reclaims an object referenced by the *WeakProcessor*, the *WeakProcessor* must be scanned to forward references to moved objects and remove handles for reclaimed objects. OpenJDK collectors perform this work at the end of every collection. We found that the reason for LXR's costly *WeakProcessor*

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

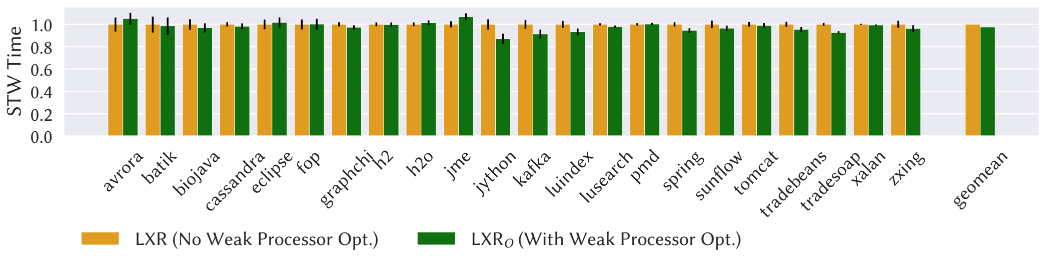


Fig. 4. **The effect of the *WeakProcessor* optimization on total stop-the-world time.** We show stop-the-world time normalized to the baseline without the optimization.

unloading is that it frequently calls into MMTk to determine heap object liveness, incurring large numbers of expensive C++ to Rust transitions.

Ultimately, the solution to this problem is to lift the liveness test into C++ code, following the pattern for other hot cross-component calls such as for write barriers and allocation sequences. Instead we decided to address the issue with a global remembered-set that records all newly created weak handles, keeping them alive and updated across RC pauses. The remembered-set is managed on the MMTk side, thus frequent cross-component calls are avoided, significantly reducing the scanning cost. We perform occasional unloading only at the end of each full heap trace, which still requires a full traversal of the storage and some costly cross-component calls. While investigating this issue, we also observed that LXR may over-mark objects and leak dead *CodeCache* objects by incorrectly sending references in the *WeakProcessor* to the concurrent marking queue. This issue was not exposed by most DaCapo benchmarks. However, for workloads like LSJ with large code cache sizes, addressing this leakage substantially reduced the heap live size and concurrent marking time. We fixed this issue.

Evaluation. We measure the stop-the-world time change in Figure 4, comparing LXR with and without the *WeakProcessor* optimization and bug fix.[4] We report results for individual benchmarks and the geometric mean, normalized to the minimum value. LXR achieves an overall STW time reduction of 2.4% on average with this change. Specifically, jython shows the greatest improvement, with a pause time reduction of 13.0%. However, jme has a slowdown of 6.7%. We found that the optimization does not change the number of pauses jme does. Instead, it defers significant *WeakProcessor* unloading work to the concurrent marking pauses, rather than processing them earlier in every GC pause as G1 does. A potential future optimization for such cases might be to slightly increase the frequency of concurrent marking for more timely and efficient clearing out of obsolete *WeakProcessor* entries.

4.5 Other Minor Issues and Enhancements

We also resolved some other minor correctness and performance issues. These issues were surprisingly impactful in terms of DaCapo performance and robustness and performance of LSJ. We list the most notable ones below.

i) We resolved a memory leak that prevented partially-free blocks from being reused, significantly reducing the total number of GCs.
ii) We reverted an LXR optimization that reduces evacuation cost by selectively evacuating young objects. Evacuating all young objects reduces fragmentation, leading to fewer GCs.

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

iii) We reduced block allocation contention by removing unnecessary spin locks.
iv) We tuned GC trigger parameters using both the DaCapo benchmarks and LSJ.
v) We resolved a C2 compiler issue where barrier elision fails to function correctly without special processing of all slow-path allocated young objects. This issue impacts all generational collectors; for instance, G1 must apply deferred and conservative barriers to these objects [16].
vi) We identified and fixed a high resident set size (RSS) issue we observed in MMTk that resulted in LSJ being terminated by the OS. MMTk, by design, eagerly initializes all off-heap metadata. We replaced it with lazy initialization, which completely eliminated the crashes due to high RSS, and also sped up the VM boot process. We upstreamed this fix to MMTk.
vii) We ported an optimization from G1 that skips some of the expensive class metadata cleaning steps when class unloading does not occur [24]. The performance impact was not revealed by DaCapo benchmarks, but was exposed by LSJ. Although the fix was trivial, it took a complete problem resolution cycle to identify and fix the issue.

Having described our systemic approach to productization and outlined key changes we made to LXR along the way, we now evaluate the performance impact of these improvements.

Table 2. **Benchmark characteristics**: minimum G1 heap size; total allocation volume; ratio of allocation to minimum heap; allocation rate relative to G1's mutator time; mean object size in bytes; percentage of large object bytes to total bytes; and percentage of survivor bytes to total bytes for a 32 MB nursery. The first eight benchmarks are the request-based latency-sensitive subset.

	Heap	Allocation			Obj	%	
Benchmark	MB	GB	/heap	MB/s	Size	Large	Survival
cassandra	124	4.1	34	681	39	0	5
h2	679	21.2	32	8 192	43	0	2
kafka	205	2.7	13	413	53	18	1
lusearch	19	23.1	1247	7 782	78	0	2
spring	51	9.0	180	4 448	60	6	2
tomcat	21	5.4	265	1 525	90	0	2
tradebeans	135	7.5	57	1 175	44	0	4
tradesoap	91	3.6	40	852	41	0	3
avrora	7	0.1	19	22	31	0	5
batik	194	0.3	2	227	55	12	45
biojava	97	9.1	96	1 323	29	0	1
eclipse	137	6.4	48	566	83	28	9
fop	15	0.3	23	639	42	1	17
graphchi	183	10.6	59	3 416	162	3	4
h2o	71	11.3	163	2 928	165	1	17
jme	29	0.0	1	5	161	41	5
jython	29	3.6	129	674	43	0	1
luindex	47	2.1	45	484	275	69	6
pmd	196	5.4	28	3 613	32	4	12
sunflow	35	17.6	515	4 851	45	0	3
xalan	15	4.6	311	4 783	90	12	43
zxing	141	1.5	11	1 765	194	52	24

5 Evaluation Methodology

5.1 LXR and OpenJDK Configuration

LXR is implemented in MMTk on OpenJDK 11.0.19+1 [23]. Unless otherwise stated, we use three versions of LXR:. i) **LXR**, the original version of LXR [28], using source code published by Zhao et al. in April 2022, based on JDK-11.0.11+6 [27]. ii) **LXR**$_F$, in which we fix all major correctness issues found in production and add all JVM features required by LSJ. iii) **LXR**$_O$, which includes all performance optimizations as well as all fixes and features in LXR$_F$. Both LXR$_O$ and LXR$_F$ are patches against Zhao et al.'s publicly available LXR codebase. We compare LXR's performance against other modern OpenJDK collectors, including G1 and Shenandoah from OpenJDK version 11.0.19+1, unless otherwise stated. ZGC does not support compressed pointers, which are required by LSJ, so we do not include it in our evaluation. Although LXR and MMTk are not available in OpenJDK 21 at the time of writing, we include comparisons with G1$_{21}$ — the G1 collector with OpenJDK 21 — in Table 4 and Figure 6. OpenJDK 21 is the latest OpenJDK LTS at the time of writing, allowing us to anchor our results with the latest state of the art OpenJDK collector.

5.2 Machine Configuration

We perform our evaluation on Zen 3 machines, with 16/32 cores 3.4 GHz CPU, and 64 GB DDR4 3200MHz memory. All machines run Ubuntu 22.04.

5.3 Benchmarks and Their Characteristics

We use all 22 benchmarks from the DaCapo Chopin (v. chopin-23.11) benchmark suite unless otherwise stated [3]. Table 2 presents their characteristics, including the minimum heap in which G1 will run, allocation statistics, object sizes, and survival ratio. Twelve benchmarks exhibit high allocation rates of more than 1 GB/s, with the highest being h2, which allocates over 8 GB per second. The large object ratio quantifies the percentage of allocated bytes for objects larger than 32 KB. Notably, two benchmarks, luindex and zxing, have a large object allocation ratio exceeding 50%. The survival ratio is measured with a fixed nursery size of 32 MB, with batik showing the highest value at 45%. Eight benchmarks are request-based latency-sensitive workloads simulating popular, real-world production servers for interactive web applications, the first eight listed in Table 2. We use them for latency analysis.

We conducted a similar analysis on LSJ. We found that LSJ has a minimum G1 heap size of 8 GB and an allocation rate of 6-8 GB/s. It has a high survival ratio of more than 20% with a 32 MB nursery. Aside from the survival ratio, we found that h2 and h2-large are often our best choices for replicating production issues. h2-large, with a 10 GB minimum heap and an 8 GB/s allocation rate, often mirrors LSJ's behavior.

5.4 Throughput and Latency Methodology

We run each DaCapo benchmark 20 times, repeating the workload for 5 iterations within each run, and only report the performance results from the last iteration. This methodology minimizes noise by thoroughly warming up the compilers and the cache before measuring peak performance. We report the geometric mean, with 95% confidence intervals across the 20 runs.

We measure request latency distribution on the first eight benchmarks in Table 2. We measure the user-experienced latency, which captures three major factors. i) the uninterrupted time to process each request, ii) interruptions due to garbage collection or OS task scheduling, and iii) the time consumed by request queuing. The DaCapo suite reports two types of latency: simple and metered. The simple latency captures the first two factors, whereas the metered latency captures all three by modelling the effect of a queue. We present metered latency with latency distribution curves.

We use two methodologies, one for consistency, when comparing with Zhao et al. [29], the other unconstrained by the limitations of the original LXR.

5.4.1 Comparison with the Original LXR. When we compare LXR_O to the original LXR, we adjust our methodology to match the prior work. Since the original LXR lacked support for JVM features discussed in Section 4.1, we disable these features on all GCs. We used the same DaCapo development version [2] and heap sizes as those used by Zhao et al. We also adhere to their methodology, which disables C1, and forces C2 compilation by using the `-Xcomp` JVM flag. We report both throughput and latency results across multiple factors of the min-heap, from 1.3× to 6×, as Zhao et al. did. We limit this comparison to 17 of the 22 DaCapo benchmarks, due to limitations in the original LXR. We deviate from Zhao et al. by using more performant *unexploded* OpenJDK builds rather than the exploded builds they used.

5.4.2 Evaluation of the Optimizations. We evaluate each optimization discussed in Sections 3 and 4, as well as the cumulative performance improvement of all the optimizations, tweaks, and bug fixes. We use the latest DaCapo benchmark stable release, chopin-23.11, and report results for all the 22 benchmarks in the suite. We use 2× G1's min-heap unless otherwise stated. For the cumulative performance analysis, we compare LXR_O with the LXR_F and report both throughput and latency results. LXR_F contains only correctness fixes without any performance optimizations, and completes all 22 DaCapo benchmarks. It is thus a suitable baseline for evaluating the cumulative performance contributions from the optimizations. To align with the original methodology employed by Zhao et al., the throughput results are measured on 2× G1's min-heap, and the latency results are measured on a 1.3× min-heap. Since the MMTk codebase is implemented with no manually-tagged inlining hints, we use profile-guided optimization (PGO) on both LXR_F and LXR_O to improve the inlining decisions. To follow best practices, we train PGO on completely distinct applications. We train PGO with GCBench [5], which is not in DaCapo. We use the resulting PGO profile to produce the final binary for evaluation.

5.5 LBO Analysis

We perform a Lower Bound Overhead (LBO) analysis to expose the GC costs of both LXR_F and LXR_O, following the methodology described by Cai et al. Our analysis includes all available OpenJDK collectors except ZGC, which does not support compressed pointers. The LBO methodology measures overheads as the difference between total cost for a system and an *approximation to the ideal.* The approximation to the ideal is found for each benchmark by running with all available GCs and for each result subtracting all overheads easily attributable to GC, such as STW time, then finding the fastest among these. We then derive the LBO value for each benchmark as the ratio of the total wall clock time of each collector to this idealized baseline. We compute LBO for each collector on each of the 22 DaCapo benchmarks and report the geometric mean across all benchmarks. We conduct LBO analysis on multiple heap sizes, ranging from 1.3× to 6× G1's minimum heap size, to expose the total costs of each GC and the time-space trade-off it makes.

6 Evaluation

6.1 Comparison with the Original LXR

Table 3 replicates Table 5 from Zhao et al. [29] and adds LXR_O, using the *modified methodology* described in Section 5.4.1. Consistent with prior results, the original LXR outperforms the other two production collectors in OpenJDK, from a tight 1.3× min-heap to a generous 6× min-heap.

Table 3. **Reproduction of Table 5 from Zhao et al. plus optimized LXR (LXR$_O$).** Using the methodology described in §5.4.1, we report latency and throughput across heap sizes. LXR$_O$ outperforms the state-of-the-art on latency and throughput. On the left is the geometric mean of 99.99th percentile latency for the *eight latency-sensitive benchmarks* and on the right throughput for *all benchmarks* normalized to G1.

	99.99th Percentile Latency (ms)				**Time/G1**			
Heap	G1	LXR *Zhao et al.*	LXR$_O$ *Ours*	Shen	G1	LXR *Zhao et al.*	LXR$_O$ *Ours*	Shen
1.3×	143.5	98.7	101.2	177.7	1.00	0.98	0.91	1.96
2×	100.1	88.0	80.1	203.8	1.00	0.99	0.94	1.42
6×	103.3	87.0	79.7	111.6	1.00	1.02	0.97	1.07

Our results for G1 and LXR are close to, but not identical to, Zhao et al.'s. The small differences could be accounted for by different operating systems versions and our use of unexploded builds. LXR$_O$ outperforms the other collectors on five of the six measures reported in the table. The only exception is 99.99% latency on a 1.3× min-heap, where LXR$_O$ is slightly worse than the original LXR by an average of just 1 ms, primarily due to noise. LXR$_O$'s 99.99% latency is significantly better than LXR, G1, and Shenandoah on moderate and large heaps.

6.2 Overall Correctness, Throughput, and Latency

We evaluate correctness, throughput, and latency with two variants of LXR:

i) **LXR$_F$**, which primarily includes bug fixes and support for missing JVM features, and
ii) **LXR$_O$**, which includes fixes for both correctness and performance issues,

and two OpenJDK collectors: G1 and Shenandoah. This comparison and the remainder of the paper use the methodology from Section 5.4.2.

Correctness. Benefitting from all the fixes discussed in Section 4, both LXR$_F$ and LXR$_O$ correctly complete all 22 DaCapo benchmarks, compared to the original LXR which was limited to 17.

Throughput. Table 4 reports throughput using a moderate 2× min-heap, relative to the minimum heap required for G1. The second column reports G1's total running time in milliseconds and the remaining four collectors in subsequent columns normalize running times to G1. All collectors complete all 22 benchmarks. The benchmarks were invoked with compressed pointers enabled, leading to a reduced memory footprint and thus smaller minimum heap sizes for G1.

On average, LXR$_F$ achieves a 3.1% throughput improvement over G1. This improvement is substantial compared to the 1% for the original LXR in Table 3. While the two results by necessity use different methodologies (Section 5.4.1 v Section 5.4.2), this result nonetheless offers a nice repudiation of the conventional wisdom that once the functional limitations of published research are addressed, performance results will be less impressive.

The highest gain was observed on xalan, which improved by 43.5%. In contrast, six benchmarks are slower than G1. The largest is tomcat at a slowdown of 14.8%. We observed that a slowdown in total STW time dominates the total throughput reduction for these benchmarks. Correctness fixes to LXR$_F$ result in notable changes in GC behavior in these benchmarks, leading to 46.8% increases in number of pauses when compared to G1. This STW time overhead suggests potential problems in GC scheduling. Additionally, the spring benchmark exhibited a 6.2% increase in mutator time overhead relative to G1. The slowdown in mutator time suggests that the concurrent operations of LXR, such as concurrent marking and lazy decrements, are less effective in the spring benchmark.

Table 4. **Benchmark throughput with a 2× min-heap and fully featured collectors.** We report G1 total execution time in milliseconds and normalize the other four collectors to G1. We show the best results for each collector in green and the worst in orange. The confidence intervals (unshown) are less than 1% for most systems. For three benchmarks – eclipse, kafka, and sunflow – the intervals exceed 5%, with the highest value on kafka at 15%. The comparison includes G1$_{21}$, which uses OpenJDK 21, the LTS at the time of writing.

Benchmark	**G1**	**G1$_{21}$**	**LXR$_F$** *w/o opt.*	**LXR$_O$** *w/ opt.*	**Shen.**
cassandra	6 459	0.988	0.983	0.973	1.104
h2	2 875	0.975	1.050	0.956	2.987
kafka	7 023	0.913	1.101	0.928	0.979
lusearch	3 153	1.618	1.062	0.869	14.171
spring	2 551	0.888	1.086	0.914	3.531
tomcat	3 893	1.059	1.148	1.006	2.528
tradebeans	6 921	0.977	1.017	1.070	1.091
tradesoap	4 579	1.018	1.022	0.965	1.162
avrora	4 524	1.459	0.902	0.905	1.018
batik	1 660	0.962	1.054	1.004	0.990
biojava	7 087	0.992	1.002	0.965	1.660
eclipse	12 003	0.885	1.016	1.008	1.015
fop	690	1.050	0.981	0.908	1.491
graphchi	3 413	1.101	0.911	0.910	2.167
h2o	5 343	0.812	0.999	0.897	1.863
jme	6 915	1.003	0.998	0.998	1.005
jython	6 190	0.976	0.971	0.963	1.779
luindex	4 588	0.970	0.946	0.938	1.066
pmd	1 997	0.870	0.970	0.872	1.693
sunflow	4 525	0.954	0.896	0.651	6.955
xalan	3 930	0.812	0.565	0.406	7.536
zxing	1 007	0.843	0.825	0.817	0.950
geomean		*0.992*	*0.969*	*0.891*	*1.860*

LXR$_O$ exhibited improvements across nearly all benchmarks, achieving an average total time speedup of 10.9%. xalan saw the most throughput improvement, with a 59.4% increase over G1. The worst-performing case, tradebeans, exhibited a slowdown of 7.0% compared to G1.

LXR$_O$ shows the greatest improvements over LXR$_F$ on xalan, sunflow, and lusearch, with total time reductions of 28.1%, 27.4%, and 18.2% respectively. The GC trigger improvements reduced the number of GC pauses by 41.3%, 29.6%, and 29.9% for these three benchmarks, respectively. Fewer pauses translate to reductions in total STW time by 38.8%, 30.6%, and 54.6% for these benchmarks. The sunflow improvement also stems from the new block allocator, resulting in a total time reduction of 7.8%. xalan and lusearch also benefited from the minor optimizations discussed in Section 4.5. Optimizations ii) and iii), when combined, lead to total time reductions of 10.8% and 16.0%, respectively. These two optimizations also enhance the throughput for sunflow by 5.6%. The geometric mean of the throughput improvements of LXR$_O$ over G1 is a surprising 10.9%.

Request Latency. Figure 5 reports the request latency distribution for the eight latency-sensitive benchmarks, measured on 1.3× G1's min-heap. LXR$_F$ exhibits significantly worse latency on six benchmarks compared to G1. On the other two benchmarks, tradesoap and kafka, it shows

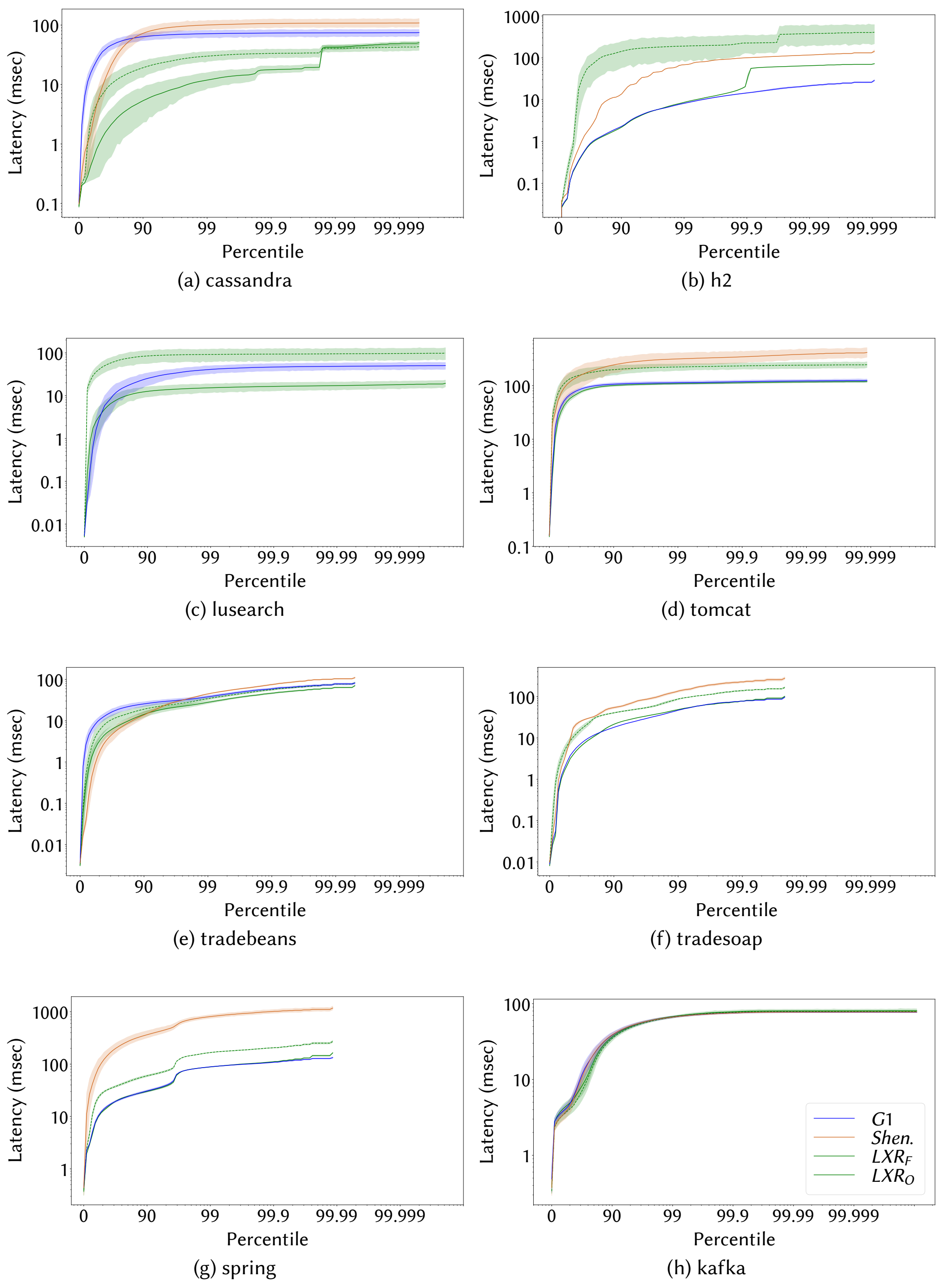


Fig. 5. **Request latency curves on 1.3× min-heap.** The y-axis plots latency in milliseconds, while the x-axis plots the percentile of requests of that latency. For instance, '90' on the x-axis corresponds to the latency observed by the 90th percentile of requests. Each solid line denotes the average latency from 20 invocations, accompanied by a shaded area that illustrates the 95th percentile variance.

comparable latency to G1. This trend suggests that addressing correctness issues and adding missing platform features led to latency regressions.

With all the performance fixes, LXR_O demonstrated improved latency over LXR_F across all eight benchmarks. h2 exhibits the most significant improvement, achieving reductions of 82.9% respectively in 99.99% tail latency compared to LXR_F. The substantial improvement is mainly due to the 53.9% fewer pauses, and a 49.9% reduction in total pause time. We believe this result is primarily attributable to the minor fixes discussed in Section 4.5, which significantly reduce both latency and pause time. Similarly, on lusearch, LXR_O exhibits a 71.3% reduction in total pause time compared to LXR_F, leading to a 81.9% decrease in 99.99% tail latency.

Nevertheless, the tail latency beyond the 99% mark for both LXR_F and LXR_O on h2 is still worse than G1. As illustrated in Figure 1, we believe that the large RC pause following each SATB cycle is a major contributor to the high tail latency. We observed that these relatively longer RC pauses are due to both a substantial allocation volume and a high survival rate. The longer pauses are likely a consequence of the aggressive class and *WeakProcessor* unloading policy we added to LXR, which may force certain objects to be recreated frequently, thereby leading to increased GC pressure. We intend to address this issue in our future work.

6.3 Lower Bound Overhead

Figure 6 presents the geometric mean of the lower bound overhead (LBO) [6] on each collector for each of the 22 benchmarks, relative to the approximation to the ideal collector. We evaluate seven collectors, including LXR_F, LXR_O, and four OpenJDK collectors. The LBO results explore heap sizes range from 1.3× to 6× the minimum heap size of G1. Each line in the figure represents the LBO results for a single collector across all evaluated heap factors.

G1's time overhead is 62.9% with the smallest 1.3× heap and reduces to 32.3% with a moderate 2× heap. As the heap headroom increases, G1's overhead further decreases, reaching 14.6% with the 6× heap. Contemporary collectors like Shenandoah have higher overhead than G1 at all heap sizes.

In contrast, LXR_F and LXR_O deliver lower overhead than all OpenJDK collectors across all heap sizes. LXR_F's overhead is 61.2% for the 1.3× heap, 1.7% lower than G1's overhead. The minimal overhead observed for LXR_F is 13.3%, achieved with the largest 6× heap. LXR_O further reduces overhead compared to LXR_F and has an overhead of only 37.1% on the 1.3× heap. With the largest 6× heap, LXR_O incurs an overhead of 10.1%, the lowest among all collectors evaluated. With a 2× heap, LXR_O's overhead is 18.7%, close to G1's overhead with a 4× heap (17.8%). This indicates that LXR_O is sufficiently efficient to achieve a similar GC overhead as G1 while operating on a significantly smaller heap — nearly half the size. These findings influenced our decision to use half of G1's heap size for both LXR_F and LXR_O, to address virtual space fragmentation issues on LSJ (Section 4.2) without incurring a significant performance penalty.

6.4 Comparison with OpenJDK 21

Blackburn et al. [3] found that G1 in OpenJDK 21 (G1_{21}) achieves the best performance and lowest GC overhead on most DaCapo benchmarks when compared to other OpenJDK 21 collectors. (Remember that LXR is not yet ported to OpenJDK 21.) Consequently, we include G1_{21} in Table 4 and Figure 6 as the reference production state-of-the-art OpenJDK collector.

As shown in Table 4, G1_{21} delivers throughput slightly better than in OpenJDK 11, just a 0.8% improvement. Individual benchmarks show significant variations. For instance, xalan has a 18.8% improvement, while lusearch shows a 61.8% slowdown. Figure 6 shows a similar trend — the average total time overhead of G1_{11} and G1_{21} are similar across all heap sizes.

When compared to $G1_{21}$, LXR_O still achieves a notable 10.2% throughput improvement on the 2× min-heap. LXR_O also consistently outperforms $G1_{21}$ on all heap sizes in terms of total time overhead. On the largest 6× heap, $G1_{21}$'s overhead is 15.2%, while LXR_O's overhead is only 10.1%.

We also evaluated the total time for G1 in OpenJDK 24 on the same 2× min-heap. At the time of writing, OpenJDK 24 is the most recent available version. Three out of the 22 benchmarks — cassandra, tradesoap, and tradebeans— do not complete with $G1_{24}$ due to their use of deprecated APIs. On the remaining 19 benchmarks, $G1_{24}$ achieves a geometric mean throughput improvement of 3.8% over $G1_{11}$, whereas $G1_{21}$ provides only a 0.9% improvement. Nonetheless, LXR_O continues to surpass $G1_{24}$, with a 9.1% throughput improvement.

These results indicate that LXR_O, implemented on OpenJDK 11, delivers performance competitive or superior to the latest available OpenJDK collectors. Furthermore, the many non-GC-related performance enhancements introduced in OpenJDK 24 will benefit applications using LXR_O once MMTk is ported to OpenJDK 24.

6.5 LSJ in Production

We evaluated all of the optimizations and fixes on LSJ. The most impactful fix was addressing the virtual space fragmentation, as discussed in Section 4.2, which efficiently eliminated out-of-memory errors. Subsequent correctness fixes further enhanced stability, allowing LSJ to use LXR_O for days under load without incident.

We observed a significant reduction in LSJ's tail request latency with LXR_O compared to LXR_F. Following the implementation of the concurrent marking performance fix discussed in Section 3, LXR_O's tail latency approached that of the modified baseline G1, with only a minor slowdown. This minor regression was attributed to the less-optimized lazy RC decrements, which could occasionally slow down and block other GC tasks. By disabling the lazy processing of RC decrements, LXR_O's tail latency became competitive with the modified G1.

Given that we compared our LSJ results with a highly customized and optimized G1 that only performs young GC, LXR's final delivered performance was very encouraging. Our findings show a productization pathway that delivers on correctness and performance while advancing performance on standard benchmarks. Nonetheless, some issues discussed in Section 4 are not fully resolved. We believe that resolution of these issues will likely yield even better performance, both on benchmarks and in our production environments. We believe that resolving these remaining tail latency issues, coupled with the significant memory savings and the elimination of the need for server redirection, will be sufficient to launch in production.

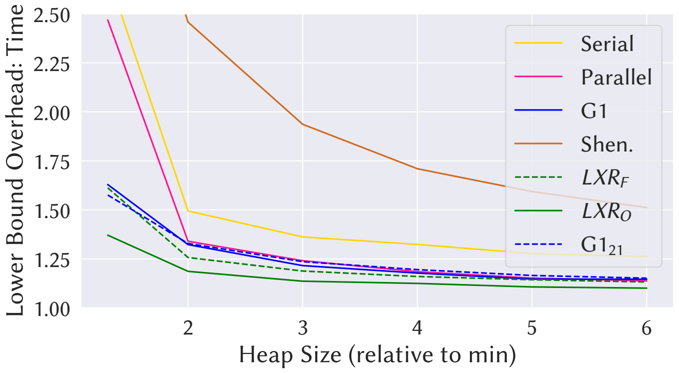


Fig. 6. **The average lower bounds on collector overheads across all benchmarks.** Following the LBO methodology [6], all results are compared to an estimated ideal collector with no GC overhead. Each line indicates the geometric mean LBO overhead across all 22 benchmarks on heap factors ranging from 1.3× to 6×.

7 Lessons

Beyond performance and correctness improvements in production, a valuable outcome of our work is transferable lessons for future productization processes.

Lesson 1. *A systematic combination of research and production methodologies can meet production objectives while simultaneously advancing the research state-of-the-art.*

Our productization process is iterative and each iteration ensures that neither production nor research objectives regress. We systematically employed a pathology mapping process to readily identify and address performance issues found in the production environment. This approach deviates significantly from the traditional practice of tuning an OpenJDK GC in production — besides benefiting the production workload, it can simultaneously improve the research algorithm and in our case, led to improvements in research benchmark suites.

Action 1. *When performance-tuning a research innovation for a production workload, use standard research benchmarks systematically to rapidly highlight inadvertent regressions.*

Lesson 2. *A benchmark suite need not contain facsimiles of production workloads. It is sufficient that the suite can replicate individual pathologies that manifest in production.*

Reproducibility is necessary for any debugging process, including production debugging. However, we demonstrate that simply replicating individual pathologies, rather than completely reproducing the issues, is sufficient and effective for production debugging, with the aid of our pathology mapping process. For example, high class unloading time is not a performance bottleneck in the DaCapo benchmark suite; thus, no benchmark can fully reproduce the pause time slowdown caused by slow class unloading. Once we captured the related pathology in some of the benchmarks, subsequent analysis and debugging became straightforward. Sometimes we configured the JVM differently to force a benchmark into the pathology. Our results illustrate that a systematic debugging methodology and a sufficiently diverse benchmark suite such as DaCapo are effective at capturing production performance issues without altering the workloads, even when reproducibility and observability are limited.

Action 2. *Benchmark suite contributors and maintainers should prioritize diversity and breadth of coverage when building suites, maximizing the space of performance pathologies the suite covers.*

Lesson 3. *Productization experiences can and should strengthen research workloads; we upstreamed one such improvement.*

Productization enhances both production and research workloads. In our case, we discovered that our pathology mapping process failed to immediately capture the virtual memory fragmentation issue (Table 1) on the DaCapo benchmarks. Consequently, we modified a benchmark (h2) to require substantially greater heap usage. As part of the mapping process, we also performed detailed analysis on each benchmark, enhancing our understanding of the benchmark behavior. These improvements have been upstreamed to the DaCapo benchmark suite, providing the research community with a strengthened and more comprehensive benchmark suite.

Action 3. *When benchmark users identify a gap in a suite's coverage, they should prioritize contributing a workload that addresses that gap.*

Lesson 4. *The requirements of a production environment are myriad and sometimes prosaic, extending well beyond those that can reasonably be addressed in a research paper.*

Research ideas and evaluations are built upon abstraction and simplification. They allow researchers to clearly shape a research idea within a less complex scope and work on the core problem with less interference from corner cases found in a production environment. Therefore, it is neither necessary nor feasible to address all production issues in a research paper — doing so would significantly complicate any research project and distract researchers from core problems. Our takeaways from this tension are that i) researchers need not address all production concerns in a research project, and should not be penalized for it, but they should clearly list the limitations of their research work. Thus, ii) a subsequent productization process will be necessary to fill these gaps, and based on our experience, may also result in further improvement over the original research.

Action 4. *Reviewers should respect the value of research which does not manifest in a system that is engineered to production standards.*

Lesson 5. *Replacing core technology in industry is a challenging sociotechnical problem.*

Adopting new technology in industry involves more than just technical challenges [7]. Many companies are reluctant to adopt new algorithms due to factors such as long-term maintenance costs, restricted and specialized production environments, real or imagined risks, or commercial concerns. Although we address key issues related to the specialized environment by introducing a pathology mapping process, our work is not yet deployed in production.

Action 5. *Researchers and reviewers should not interpret the failure of industry to embrace a research innovation as evidence of weakness of the idea.*

8 Discussion

Benchmarks. Many important languages including Python, JavaScript, Rust, and Go, lack widely-used, rich, representative benchmarks of the kind that were central to the productization process we undertook. On the one hand, one might say that this limits the transferability of our approach. On the other hand, we would argue that this highlights just how important it is that the community (academia and industry) invest in widely available, representative benchmark suites for all of these important languages. We hope that the community will note how well Java and C have been served by their benchmark eco-systems, and acknowledge the value in continued investment in keeping them up to date and representative.

Productization Dividend. We invite the community to think carefully about the ‘productization dividend’ and its effect on the peer review process. We are not aware of other accounts that quantify this effect, so we do not know how representative the 10% that we observe is. However, it is clear that in this domain, at least, the dividend is substantial enough to introduce bias into our research priorities. As we observed at the start of this paper, it seems plausible that researchers are implicitly advantaged if they can exploit the dividend already built into the highly tuned systems that define the status quo. If the status quo is engineered monolithically, in such a way that entirely new ideas cannot be readily built on it, then incremental work will tend to carry a substantial advantage over entirely new work, which does not bode well for innovation. Perhaps the research community should explicitly set performance expectations differently for entirely new ideas? We do not have answers to this dilemma, but we believe that the research community, particularly in areas where mature products define the state-of-the-art, would do well to consider whether the effect we describe is real and if so, how to mitigate it in such a way as to ensure innovation can thrive.

9 Conclusion

The goals of clarity and generality in research often render research outcomes less suitable for complex and specialized production environments. Consequently, a translation process is essential to bridge the gap between research advances and production-readiness. Even when this translation occurs, the process rarely appears in the literature, so the nature of the gap and techniques for bridging it tend not to be well known, likely impeding translation.

We use the LXR garbage collector as a case study to systematically discuss our experience in productizing a research artifact in a complex industry workload. We show that our problem resolution methodology efficiently addresses issues in production and show that it significantly improved the correctness and performance of the LXR collector, resulting in a runtime that outperforms the mature state of the art by more than 10%. Our methodology and lessons demonstrate how the translation process can feed back and enhance both the project itself and other research tools in the broader systems community, such as standard research benchmark suites. The productization dividend of 10% that we experienced is itself interesting, and raises questions about how the peer review process evaluates novel work when the state-of-the-art is mature and highly productized.

10 Data-Availability Statement

Our source code, the DaCapo benchmark suite, and detailed instructions for reproducing our results are publicly available [30, 31].

Acknowledgments

We thank Jonathan Joo, Tony Scelfo, and Amer Diwan for supporting this work.

References

[1] Bowen Alpern, C. Richard Attanasio, John J. Barton, Michael G. Burke, Perry Cheng, Jong-Deok Choi, Anthony Cocchi, Stephen J. Fink, David Grove, Michael Hind, Susan Flynn Hummel, Derek Lieber, Vassily Litvinov, Mark F. Mergen, Ton Ngo, James R. Russell, Vivek Sarkar, Mauricio J. Serrano, Janice C. Shepherd, Stephen E. Smith, Vugranam C. Sreedhar, Harini Srinivasan, and John Whaley. 2000. The Jalapeño virtual machine. *IBM Syst. J.* 39, 1 (2000), 211–238. doi:10.1147/SJ.391.0211

[2] Stephen M. Blackburn. 2022. Dacapo dev-branch, commit-b00bfa9. https://github.com/dacapobench/dacapobench/tree/b00bfa96b6db296bdc6f57a57e56a9a34b2d2d89

[3] 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

[4] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, and Applications (OOPSLA), Portland, OR, USA*. ACM, New York, NY, USA, 169–190. doi:10.1145/1167473.1167488

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

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

[7] Michael A. Cusumano, Yiorgos Mylonadis, and Richard S. Rosenbloom. 1992. Strategic Maneuvering and Mass-Market Dynamics: The Triumph of VHS over Beta. *Business History Review* 66, 1 (1992), 51–94. doi:10.2307/3117053

[8] 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

[9] Robin Garner, Stephen M. Blackburn, and Daniel Frampton. 2011. A comprehensive evaluation of object scanning techniques. In *Proceedings of the 10th International Symposium on Memory Management, ISMM 2011, San Jose, CA, USA, June 04 - 05, 2011*, Hans-Juergen Boehm and David F. Bacon (Eds.). ACM, New York, NY, USA, 33–42. doi:10.1145/1993478.1993484

[10] Mikael Gerdin. 2014. Implement Class Unloading after completing a concurrent mark cycle. https://bugs.openjdk.org/browse/JDK-8051611

[11] Stefan Johansson. 2017. JEP 248: Make G1 the Default Garbage Collector. https://openjdk.org/jeps/248

[12] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004*. IEEE, IEEE Computer Society, USA, 75–86. doi:10.5555/977395.977673

[13] Chris Lattner and Vikram S. Adve. 2005. Transparent pointer compression for linked data structures. In *Proceedings of the ACM Workshop on Memory System Performance (MSP), Chicago, IL, USA*. ACM, New York, NY, USA, 24–35. doi:10.1145/1111583.1111587

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

[15] MMTk Team. 2024. MMTk: A high performance language-independent memory management framework. https://mmtk.io

[16] OpenJDK. 2024. Card marking barrier's special handling to slow-path allocated objects. https://github.com/openjdk/jdk/blob/aaaa86b57172d45d1126c50efc270c6e49aba7a5/src/hotspot/share/gc/shared/cardTableBarrierSet.cpp#L91-L144

[17] Oracle. 2009. Changes in 1.6.0_14 (6u14). https://www.oracle.com/java/technologies/javase/6u14.html

[18] Erik Österlund. 2018. ZGC: Concurrent Class Unloading. https://bugs.openjdk.org/browse/JDK-8214897

[19] John Rose. 2012. Compressed oops in the Hotspot JVM. https://wiki.openjdk.org/display/HotSpot/CompressedOops

[20] Rifat Shahriyar, Stephen M. Blackburn, and Daniel Frampton. 2012. Down for the count? Getting reference counting back in the ring. In *International Symposium on Memory Management, ISMM '12, Beijing, China, June 15-16, 2012*, Martin T. Vechev and Kathryn S. McKinley (Eds.). ACM, 73–84. doi:10.1145/2258996.2259008

[21] 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

[22] The LLVM Team. 2009. LLVM 2.6 Release Notes. https://releases.llvm.org/2.6/docs/ReleaseNotes.html

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

[24] The OpenJDK Team. 2019. G1 parallel cleaning task. https://github.com/openjdk/jdk/blob/jdk-21%2B0/src/hotspot/share/gc/g1/g1ParallelCleaning.cpp#L72-L77

[25] Cristian Velazquez and Vineeth Karayil Sekharan. 2024. Uber: GC Tuning for Improved Presto Reliability. https://www.uber.com/en-AU/blog/uber-gc-tuning-for-improved-presto-reliability

[26] Taiichi Yuasa. 1990. Real-time garbage collection on general-purpose machines. *Elsevier Journal of System Software* 11, 3 (1990), 181–198. doi:10.1016/0164-1212(90)90084-Y

[27] Wenyu Zhao. 2022. OpenJDK 11.0.11+6 with MMTk support. https://github.com/mmtk/openjdk/tree/f817e9d00b2850221bb9443443a123e38e81a129

[28] Wenyu Zhao. 2023. LXR as of April 08, 2022, commit-4d4e516. https://github.com/wenyuzhao/mmtk-core/tree/4d4e516c8789da184367676e4c7390411b3c8a8f

[29] 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

[30] Wenyu Zhao, Stephen M. Blackburn, Kathryn S. McKinley, Man Cao, and Sara S. Hamouda. 2025. Latest LXR Artifact. https://doi.org/10.5281/zenodo.15751334

[31] Wenyu Zhao, Stephen M. Blackburn, Kathryn S. McKinley, Man Cao, and Sara S. Hamouda. 2025. LXR Artifact. https://doi.org/10.5281/zenodo.15751335

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