System Design and Innovation: A Garbage Collection Case Study

Speaker
Kathryn S. McKinley, Google

Collaborators
Steve Blackburn, Google
Wenyu Zhao, ANU

Garbage Collection Design through the lens of

✓ Breaking the **Time & Space Tradeoff** with **Memory Performance**
Immix: Hierarchy of Lines and Blocks with judicious copying

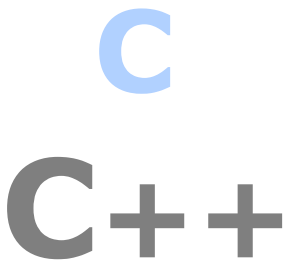
✓ Breaking Low Latency & GC Pause Time Tradeoff for **Parallel Performance**
LXR: Reference Counting with even less copying in pauses, concurrent tracing

✓ **Abstraction for Innovation**
MMTk & Work Packets

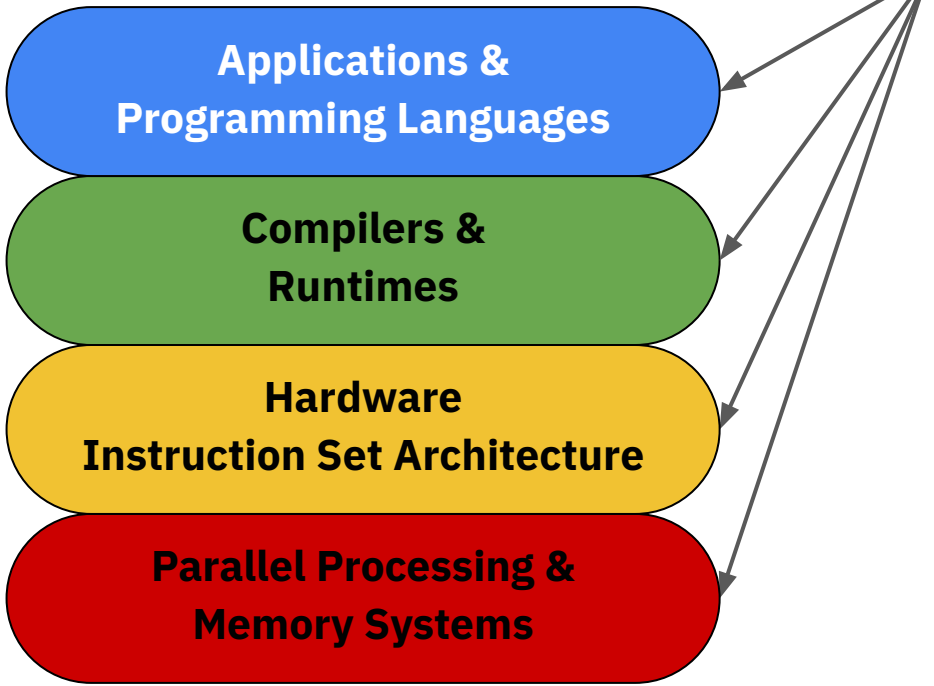
Benchmarks, Performance Methodologies & More Methodologies
Not today, but critical

Why should you care about Garbage Collection (GC)?

Programming Languages People Use Rely on Garbage Collection (GC)

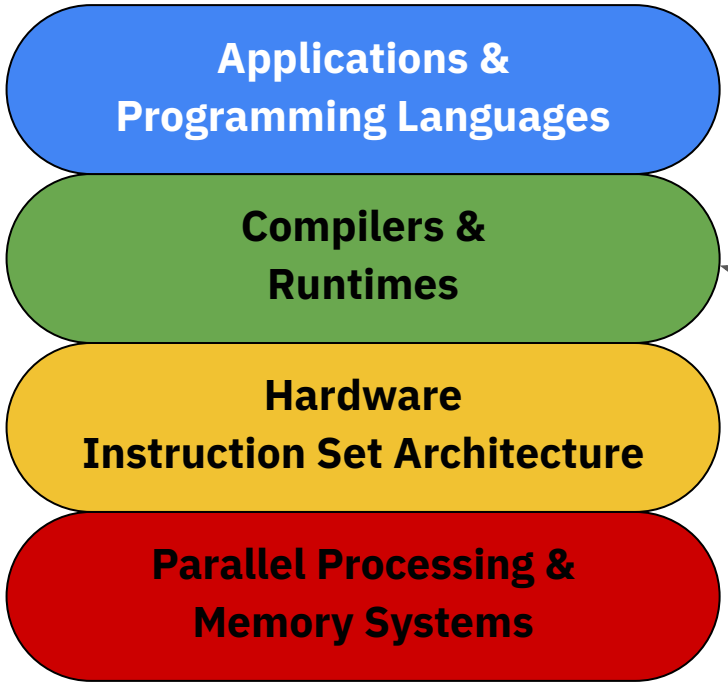


Innovation in applications, programming languages & hardware



Innovation is happening at all the layers of the stack

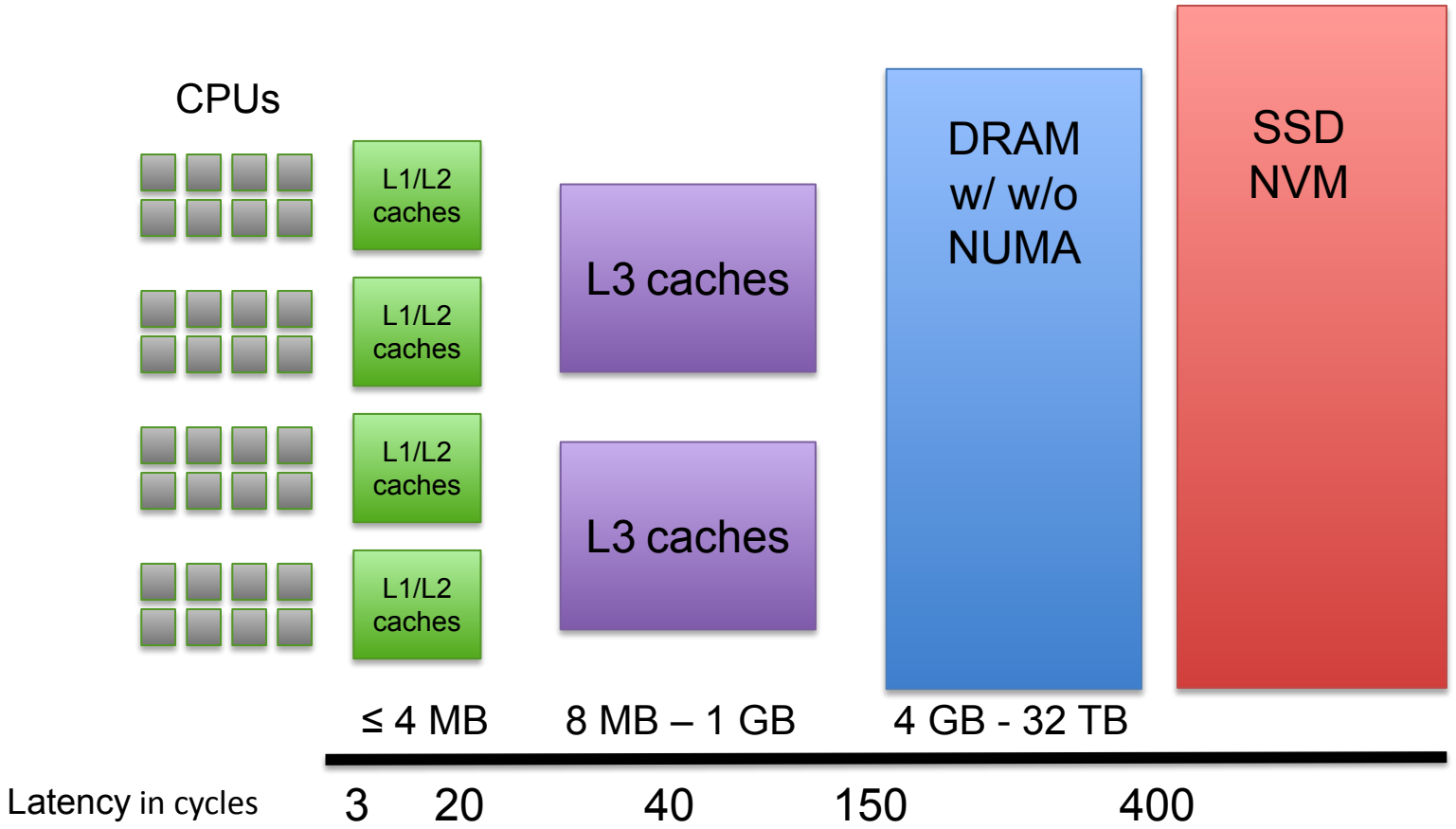
Garbage Collection Helps Glue it All Together



Innovation is happening at all the layers of the stack

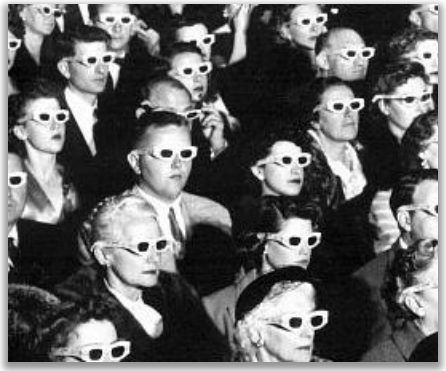
Compilers & Runtimes with Garbage Collection glue the layers together, dictating correctness & performance

Parallel Hardware has Deep Memory Systems



History Foundational GC Heap Organizations

Mark-Sweep
McCarthy
1960



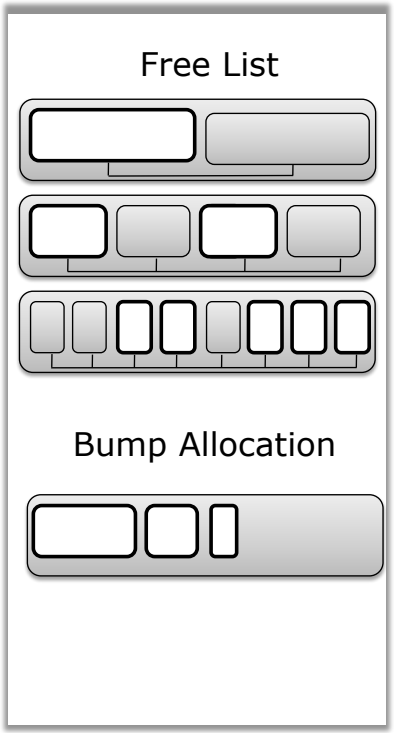
Mark-Compact
Styger
1967

Semi-Space
Fenichel & Yochelson
1969

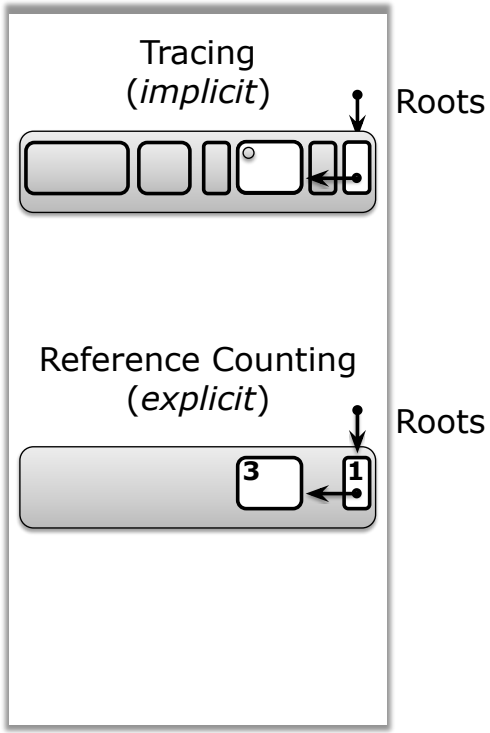
Mark-Region
Blackburn & McKinley
2008

GC Fundamentals

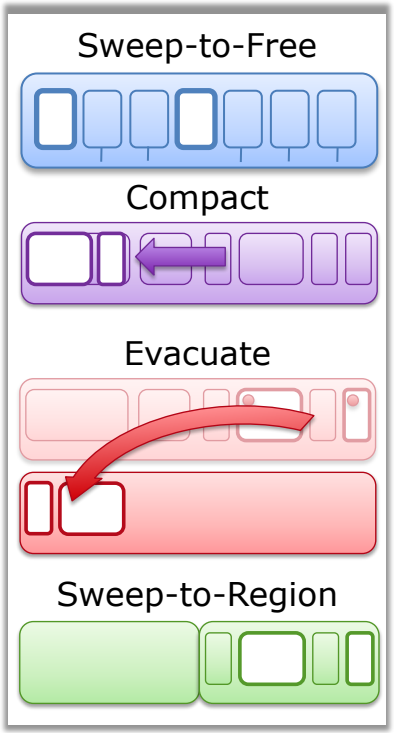
Allocation



Identification



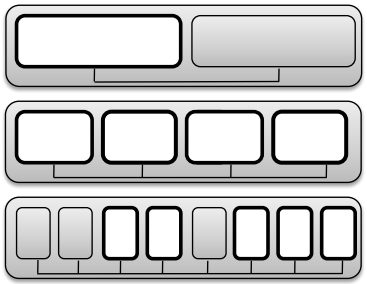
Reclamation



Mark Sweep

Allocation

Free List

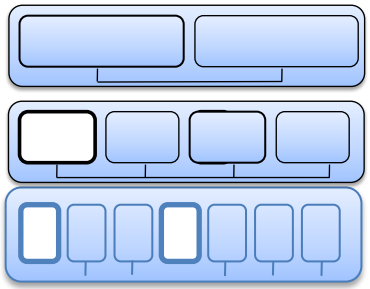


Identification

Tracing
(*implicit*)

Reclamation

Sweep-to-Free

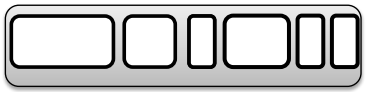


- Allocation produces **poor locality** in multiple size-based free lists.
- **Some fragmentation memory overhead**
- **Low computational overhead** due to super fast tracing & non moving reclamation

Mark Compact

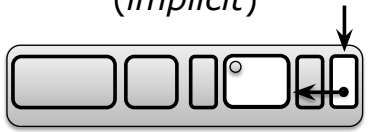
Allocation

Bump Pointer



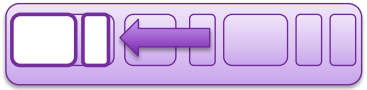
Identification

Tracing
(*implicit*)



Reclamation

Compact

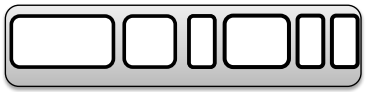


- Super fast allocation with **good locality**
- **No fragmentation**
- Tracing is fast
- Multipass algorithm to compacting adds **high computational overhead**

Semi Space

Allocation

Bump Pointer

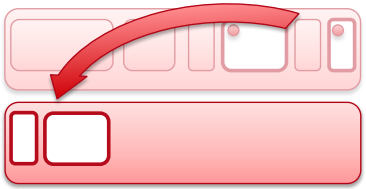


Identification

Tracing
(*implicit*)

Reclamation

Evacuate



- Super fast allocation with **good locality**
- **No fragmentation, but doubles the memory overhead** because all objects could survive!
- Tracing and evacuation (copying) adds computational overhead to copy objects

Mark-Region

Allocation

Bump Pointer

Identification

Tracing
(*implicit*)

Reclamation

Sweep-to-Region

- Super fast allocation with **good locality**
- **Low fragmentation**
- Tracing with occasional copying yields
  low computational and low memory overheads

Mark-Region Immix: Lines and Blocks

Large Regions

✓ More contiguous allocation

✗ Fragmentation (false marking)

Small Regions

✗ Constrained object sizes

✗ Increased metadata o/h

✗ Fragmentation can't fill blocks

Lines & Blocks

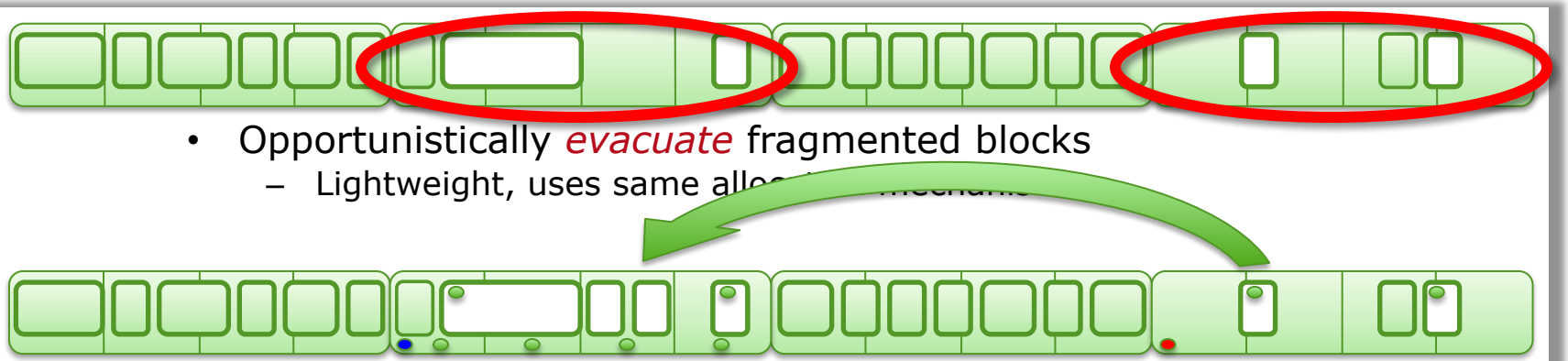
☐ TLB locality, cache locality
☐ Objects span lines

✓ Less fragmentation

☐ Block > 4 X max object size
☐ Lines marked with objects

✓ Fast common case

Opportunistic Defragmentation



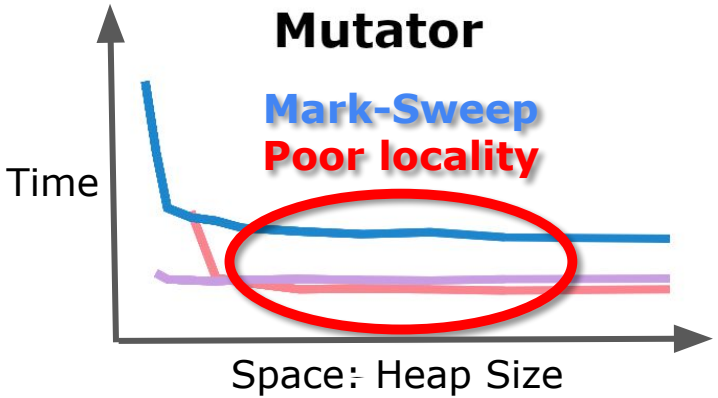
- Opportunistically *evacuate* fragmented blocks
  - Lightweight, uses same allo[illegible]

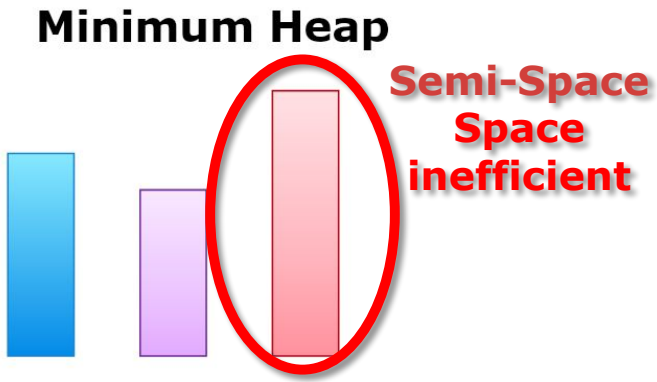
- Identify source and target blocks
- Evacuate objects in source blocks
  - Allocate into target blocks
- Opportunistic
  - Leave in place if no space, or object pinned

Mark-Region Immix Collection Broke the Space Time Tradeoff

Stop-the-World Collectors Pause all the Threads

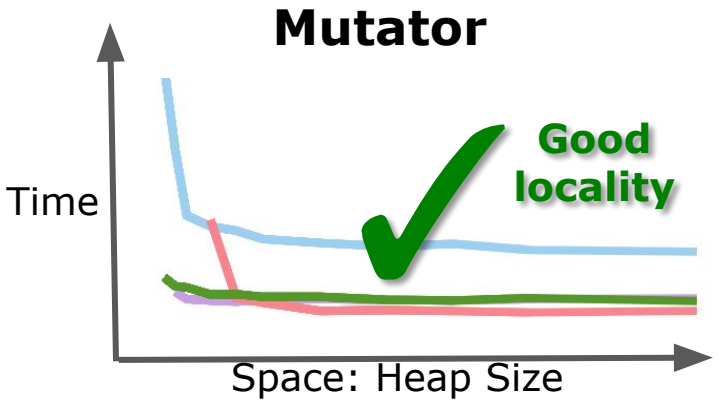
Memory System & Computation Performance Tradeoffs

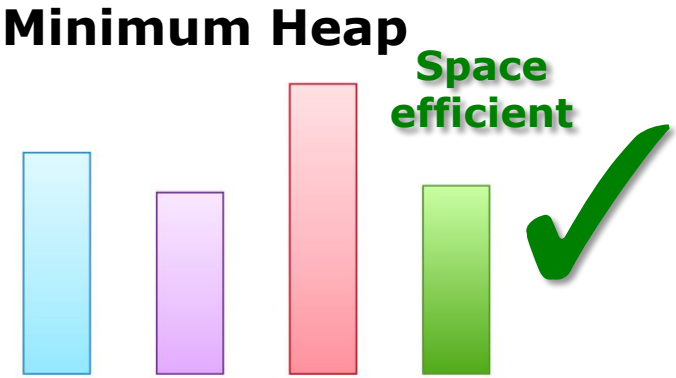


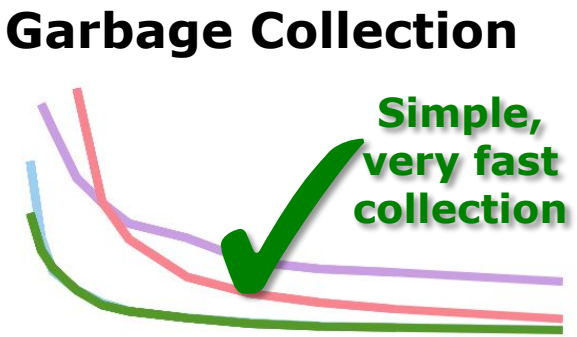


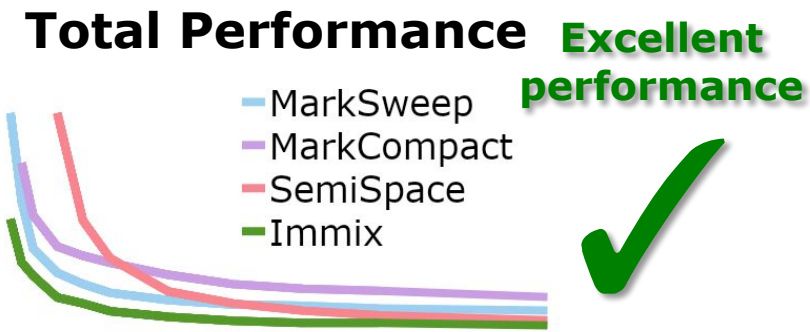
Actual data, taken from geomean of DaCapo, jvm98, and jbb2000 on 2.4GHz Core 2 Duo

Performance Tradeoffs









Actual data, taken from geomean of DaCapo, jvm98, and jbb2000 on 2.4GHz Core 2 Duo

Impact of Immix

LXR (PLDI 2022, OOPSLA 2025)

Exploits Immix to avoid expensive copying; a novel approach to achieving high throughput and low latency, significantly outperforming the state-of-the-art (11% throughput advantage over G1).

Iso (PLDI 2025)

Exploits Immix's pinning to solve 30-year old problem of how to do thread-local garbage collection without exploitable immutability (pin public objects, move private).

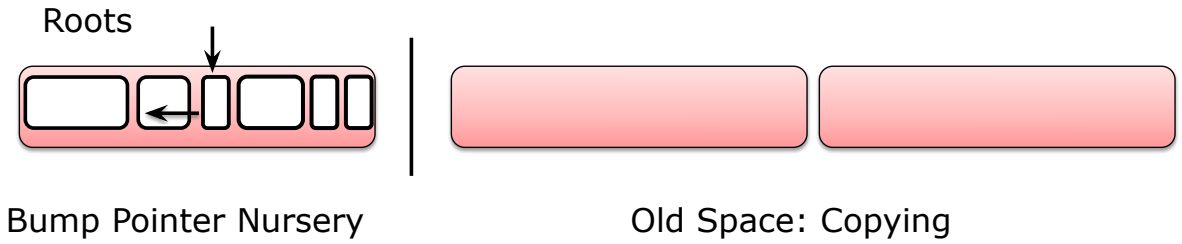
CRuby, **Julia** (ISMM 2025)

Exploit Immix's pinning to allow legacy languages with low-overhead C interfaces to have a copying GC for the first time. C-reachable objects are pinned.

Generational GC [1983, 1984]

Bump Pointer Nursery + Any Heap Organization for the Old Space

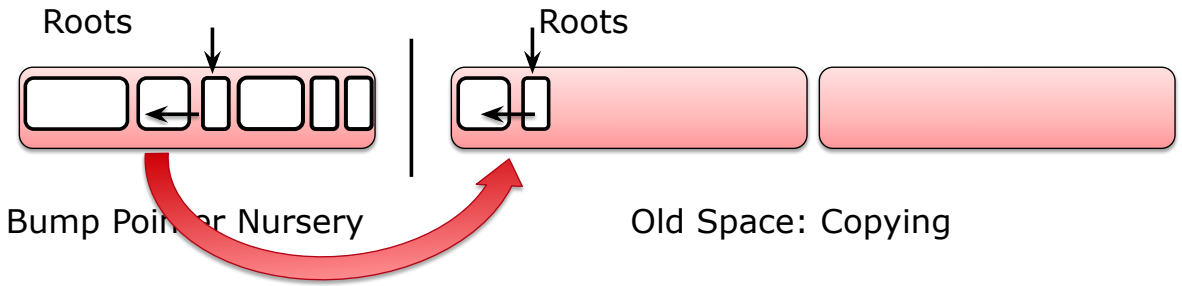
- Repeatedly fill the nursery, copy out, when old space is full perform a whole heap collection



Generational GC [1983, 1984]

Bump Pointer Nursery + Any Heap Organization for the Old Space

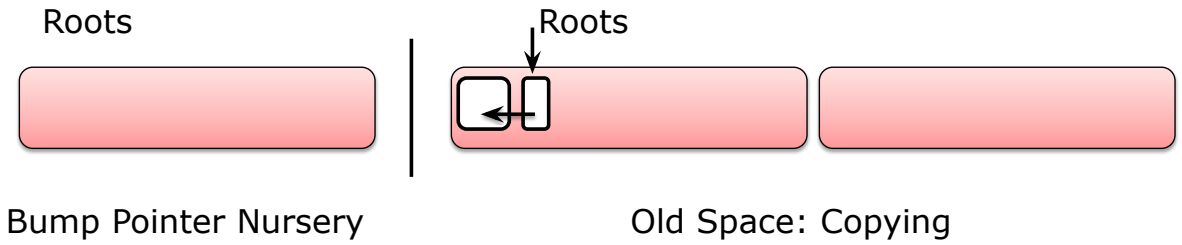
- Repeatedly fill the nursery, copy out, when old space is full perform a whole heap collection



Generational GC [1983, 1984]

Bump Pointer Nursery + Any Heap Organization for the Old Space

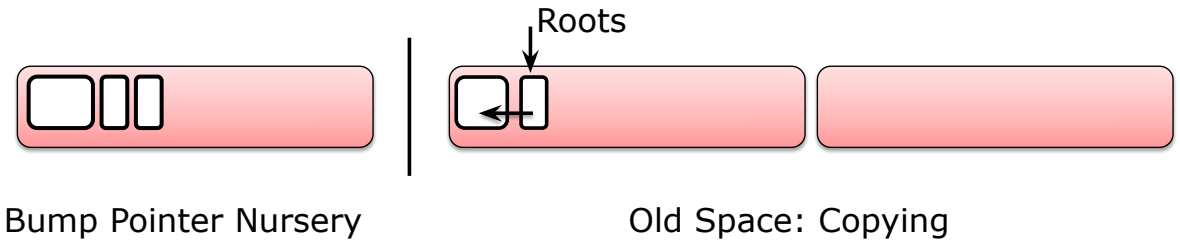
- Repeatedly fill the nursery, copy out, when old space is full perform a whole heap collection



Generational GC [1983, 1984]

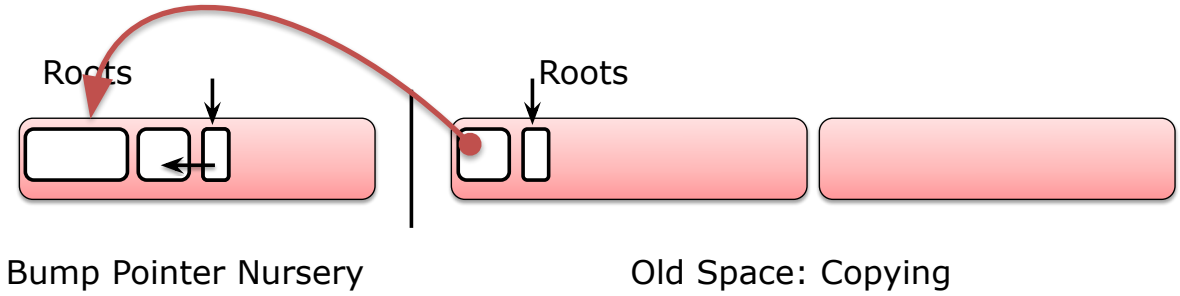
Bump Pointer Nursery + Any Heap Organization for the Old Space

- Repeatedly fill the nursery, copy out, when old space is full perform a whole heap collection



Generational GC [1983, 1984]

Bump Pointer Nursery + Any Heap Organization for the Old Space

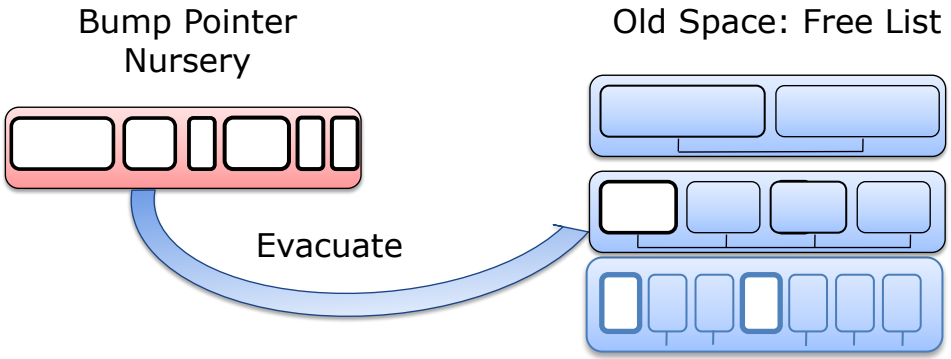


- Requires a write barrier to independent collect the nursery
  If source pointer in old space, and target is in the nursery then store the source pointer in a “cross generation pointer buffer”

G1: Popular Default Open JDK Production GC [2004]

Garbage First:

Generational GC Stop-the-World, with Free List for the Old Space

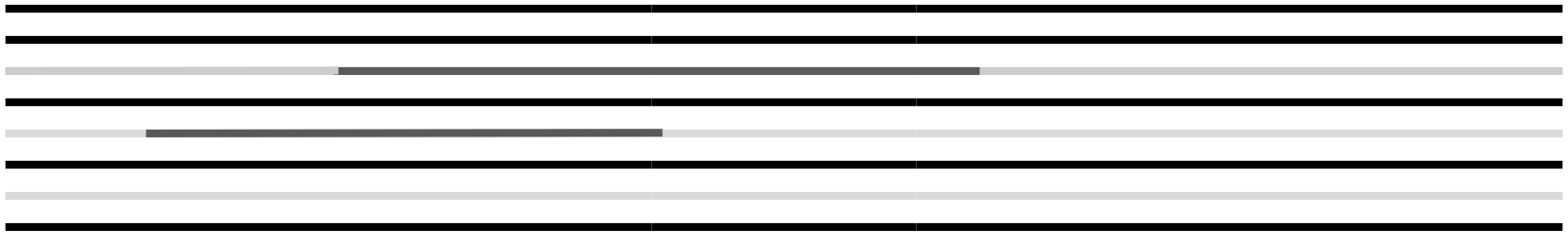


On to Concurrent & Parallel GC!

Parallelism

Application threads

Parallelism is imperfect



Application threads

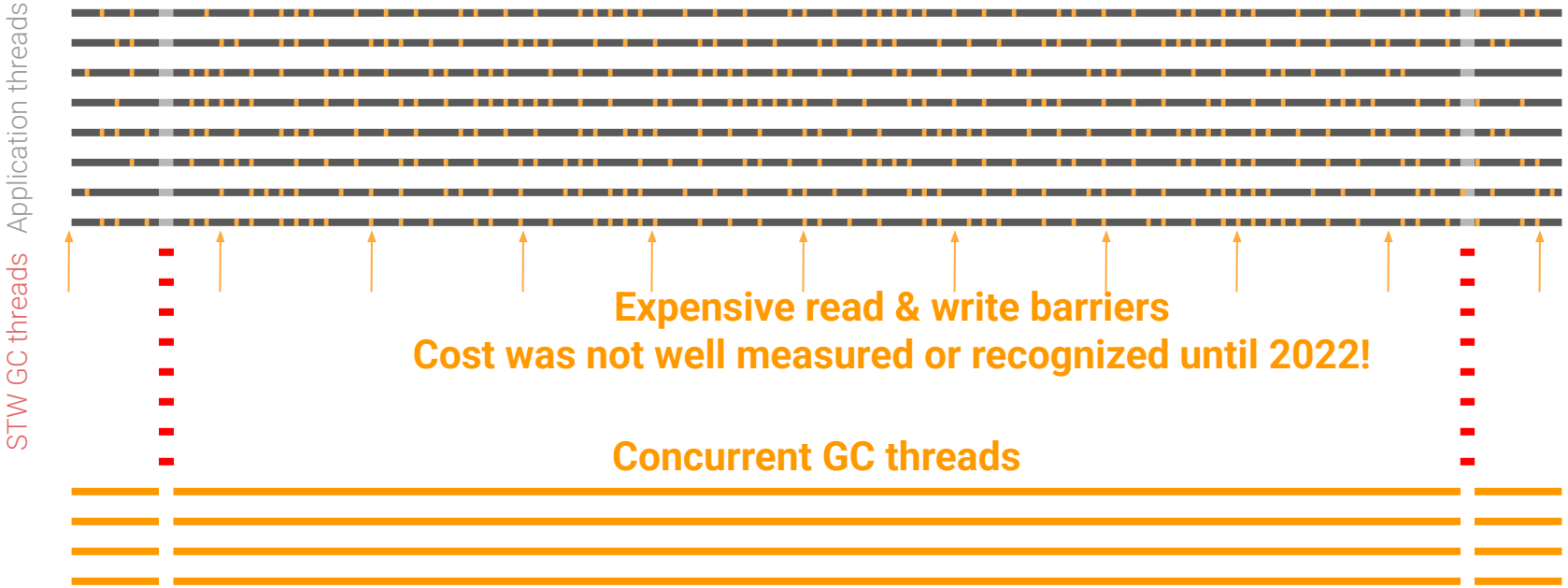
Stop-the-World

Periodic lengthy pauses, adds latency and queuing delay

Add Concurrency to Make Pauses Short!?

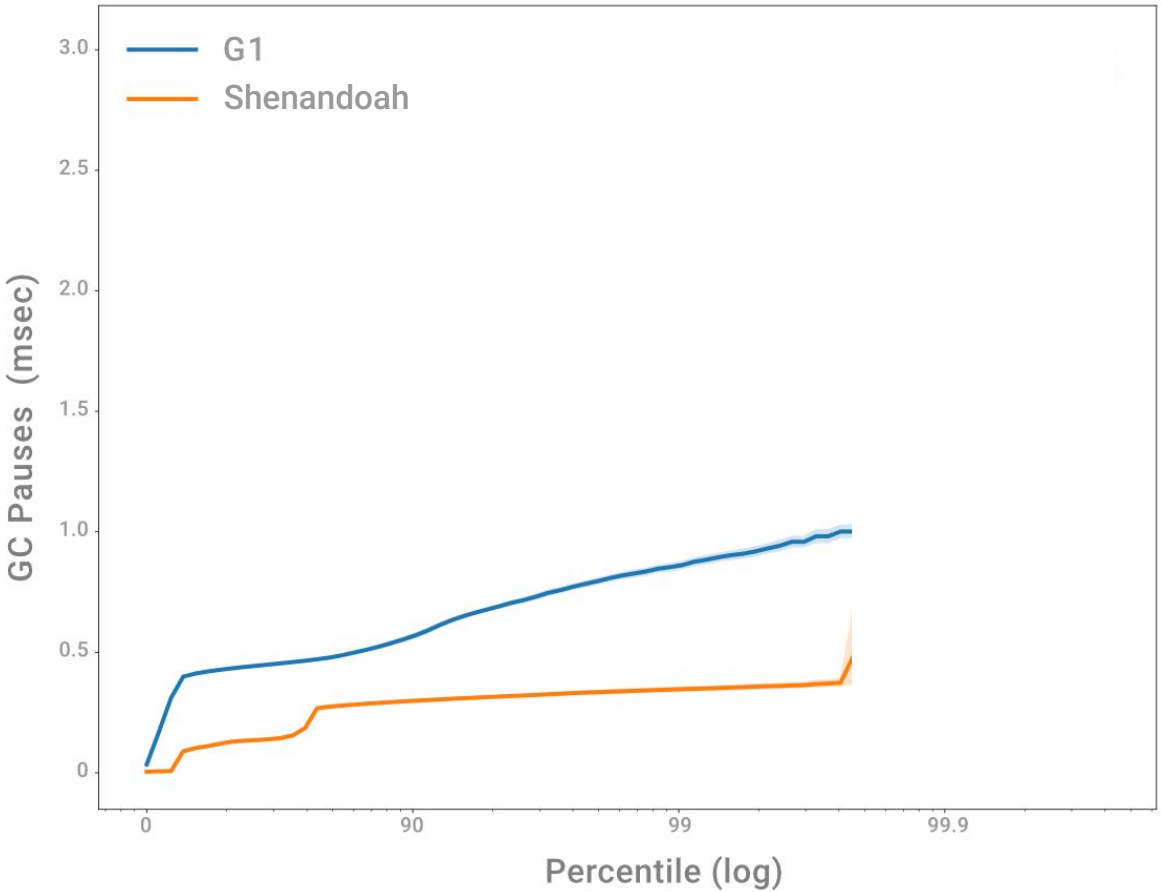
Shenandoah [2016], C4 [2011] & ZGC [2017]

Short pauses with tracing, but expensive concurrent copying barriers



DaCapo dev-Chopin (github 202204) using our fork of OpenJDK JDK 11.0.11+6 +35, running on a 16/32 AMD Ryzen 5950X with 64GB DDR4-3200 memory.

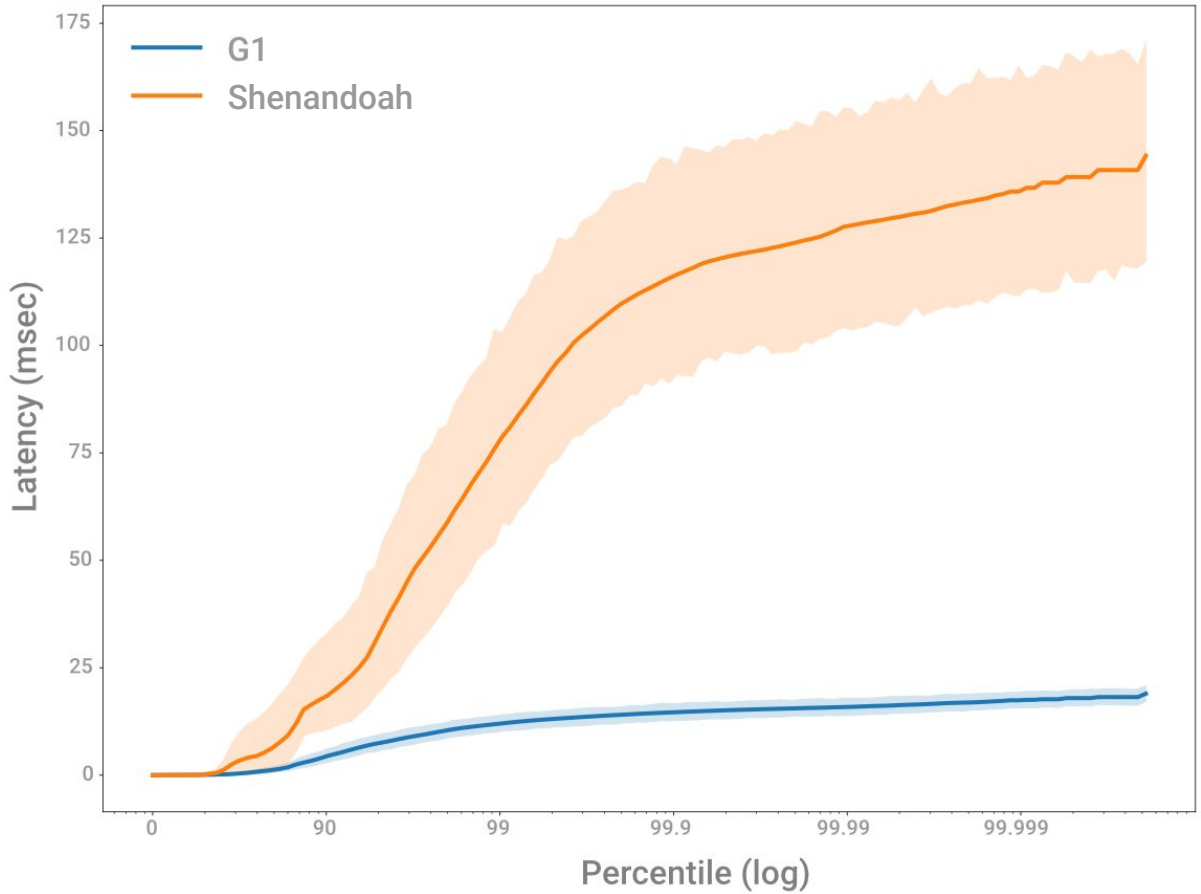
GC Pauses Lusearch



512K search queries running over Apache lucene 8.5.2., with a heap 1.3X G1's minimum.

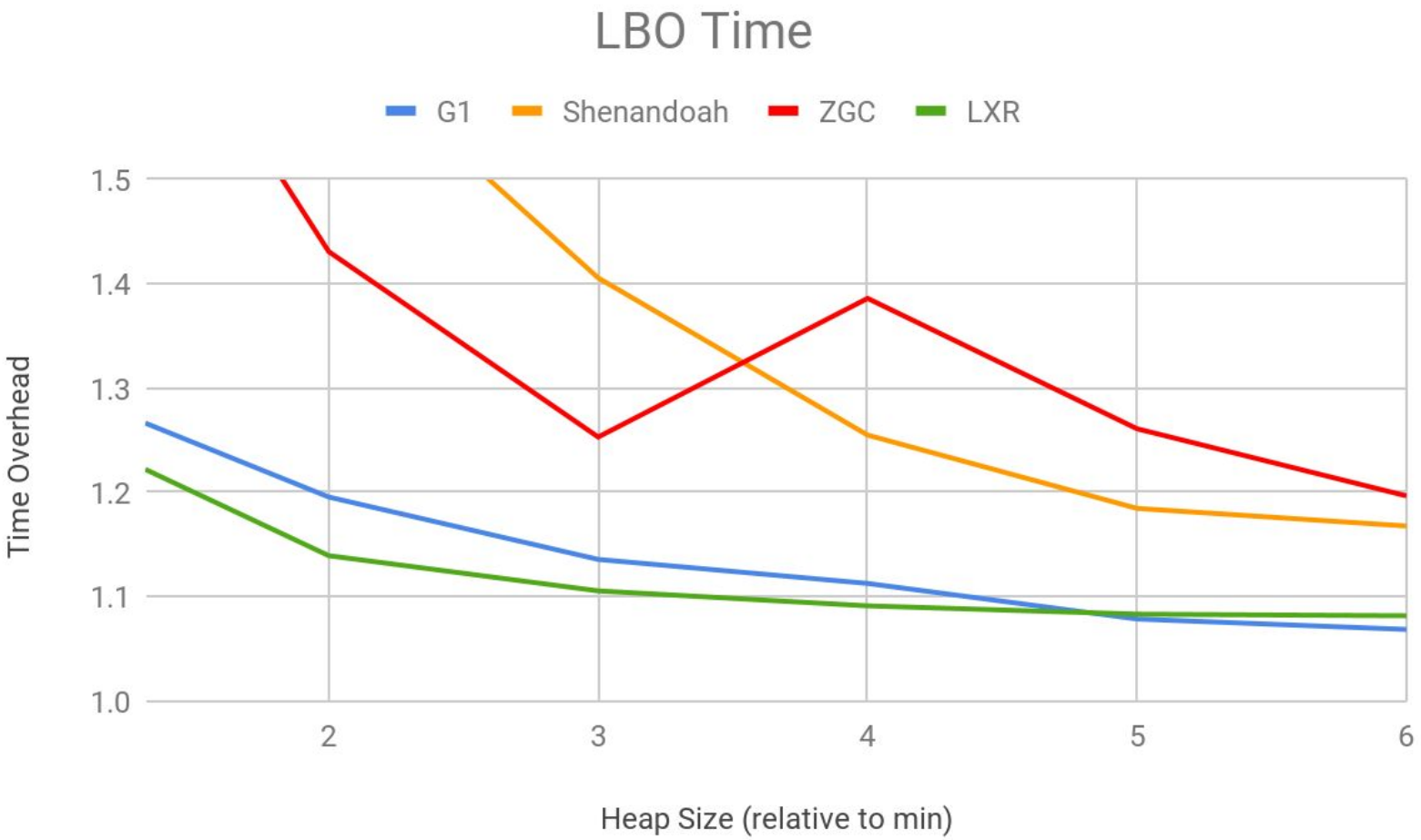
DaCapo dev-Chopin (github 202204) using our fork of OpenJDK JDK 11.0.11+6 +35, running on a 16/32 AMD Ryzen 5950X with 64GB DDR4-3200 memory.

Latency Lusearch



512K search queries running over Apache lucene 8.5.2., with a heap 1.3X G1's minimum.

DaCapo dev-Chopin (github 202204) using our fork of
OpenJDK JDK 11.0.11+6 +35, running on a 16/32
AMD Ryzen 5950X with 64GB DDR4-3200 memory.



DaCapo dev-Chopin (github 202204) using our fork of
OpenJDK JDK 11.0.11+6 +35, running on a 16/32
AMD Ryzen 5950X with 64GB DDR4-3200 memory.

LXR Design Insights

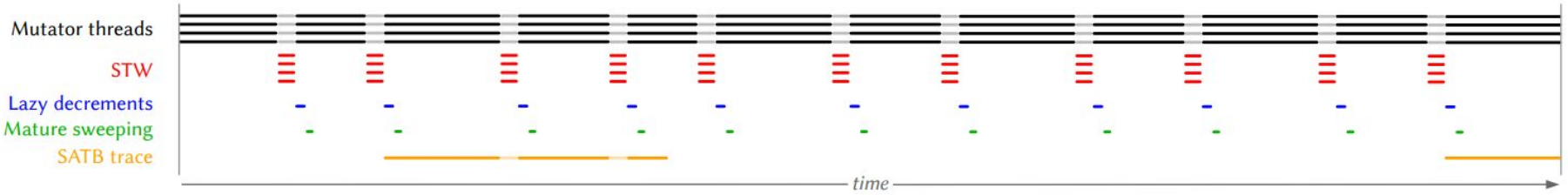
High throughput with

- ✓ **Immediacy** with reference counting in pauses
- ✓ **Avoiding copying** no concurrent copying

Low latency

- ✓ **Short frequent pauses** do reference counting & copying
- ✓ **Snapshot-At-The-Beginning (SATB)** fast write barrier (no read barrier)
- ✓ **Concurrent Cycle Tracing** since reference counting cannot reclaim cycles

LXR



STW Pause performs reference counting increments and some decrements
defragments heap with small amounts of copying

Not in Pause

Lazy decrements concurrent decrements in application slack

Mature sweeping when lazy decrements & trace completes, reclaim free memory

SATB Write Barrier captures reference counts and write during tracing

SATB Trace reclaims cycles

Methodology

OpenJDK 11 LTS at time of development

DaCapo Chopin 2022 dev snapshot

- 4 latency-sensitive workloads
- All 17 workloads report throughput

Three highly-tuned production collectors

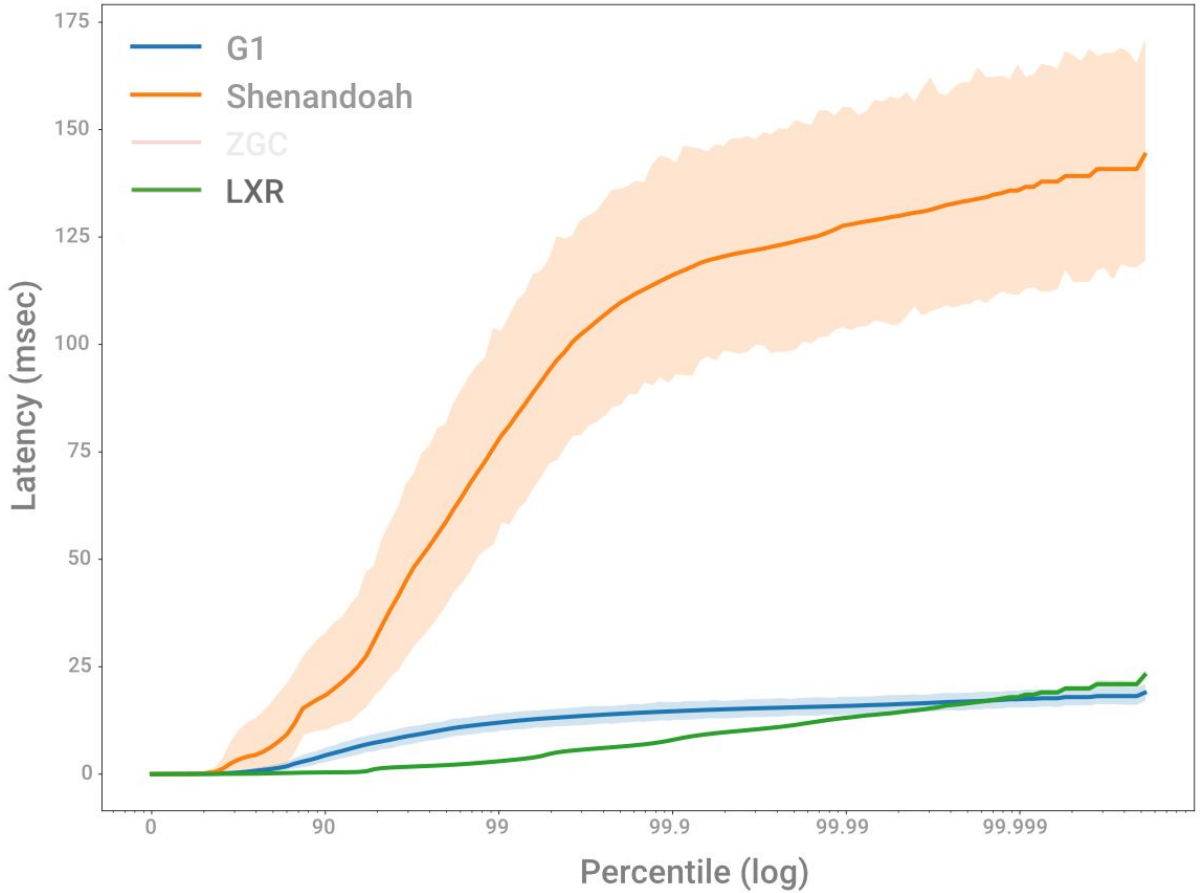
- G1, Shenandoah, ZGC

Three hardware platforms

- AMD Zen 3 5950X 16/32 3.4 GHz 64 MB LLC, 64 GB DDR4
- AMD Zen 2 3900X 12/24 3.8 GHz 64 MB LLC, 64 GB DDR4
- Intel Coffee Lake i9-9900K 8/16 4.6 GHz 16 MB LLC, 128 GB DDR 4

DaCapo dev-Chopin (github 202111) using our fork of OpenJDK JDK 11.0.11+6 +35, running on a 12/24 AMD Ryzen 3900X with 64GB DDR4-2133 memory.

Latency Lusearch

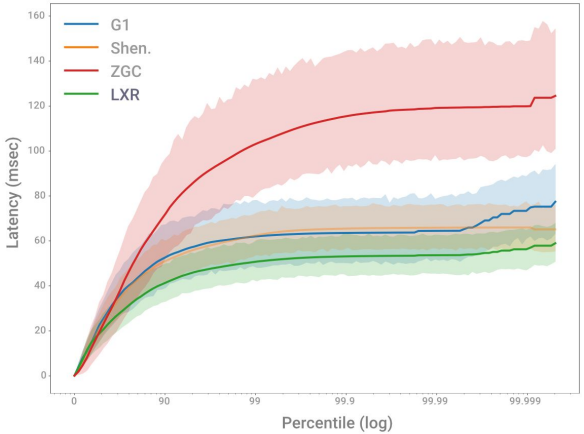


512K search queries running over Apache lucene 8.5.2., with a heap 1.3X G1's minimum.

DaCapo dev-Chopin (github 202204) using our fork of
OpenJDK JDK 11.0.11+6 +35, running on a 16/32
AMD Ryzen 5950X with 64GB DDR4-3200 memory.

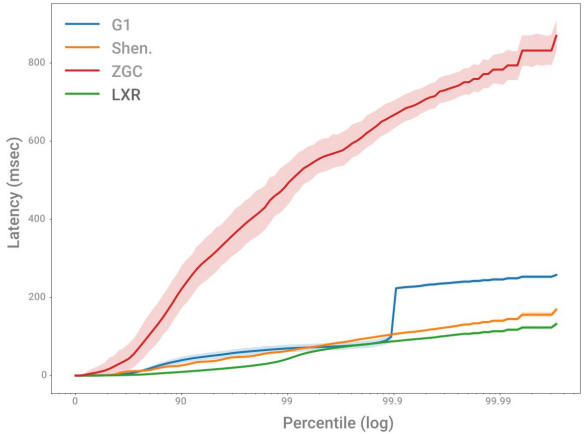
Latency

Cassandra



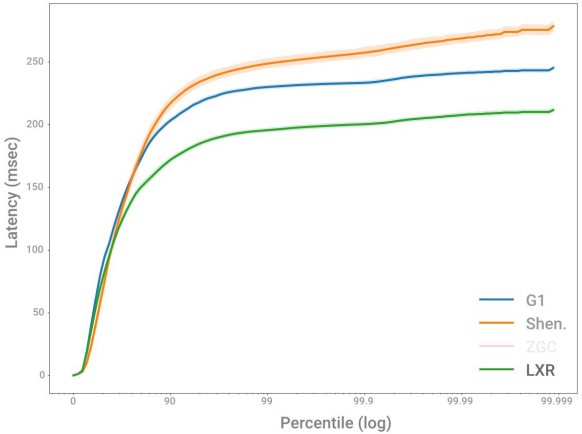
YCSB workloads running over Apache cassandra 3.11.10., with a heap 1.3X G1's minimum.

H2



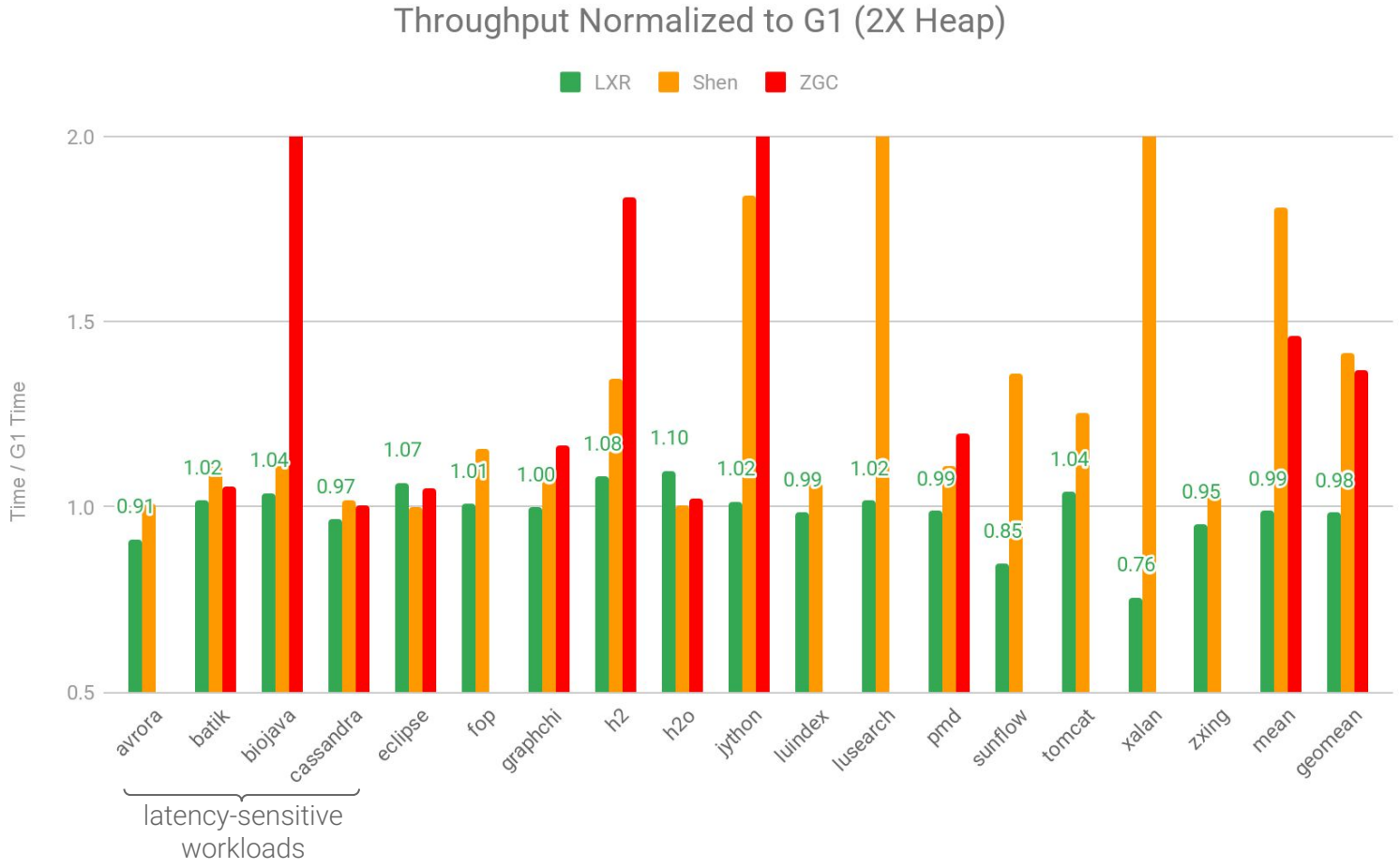
TCP-C-like workload running over Apache derby 10.14.2.0., with a heap 1.3X G1's minimum.

Tomcat



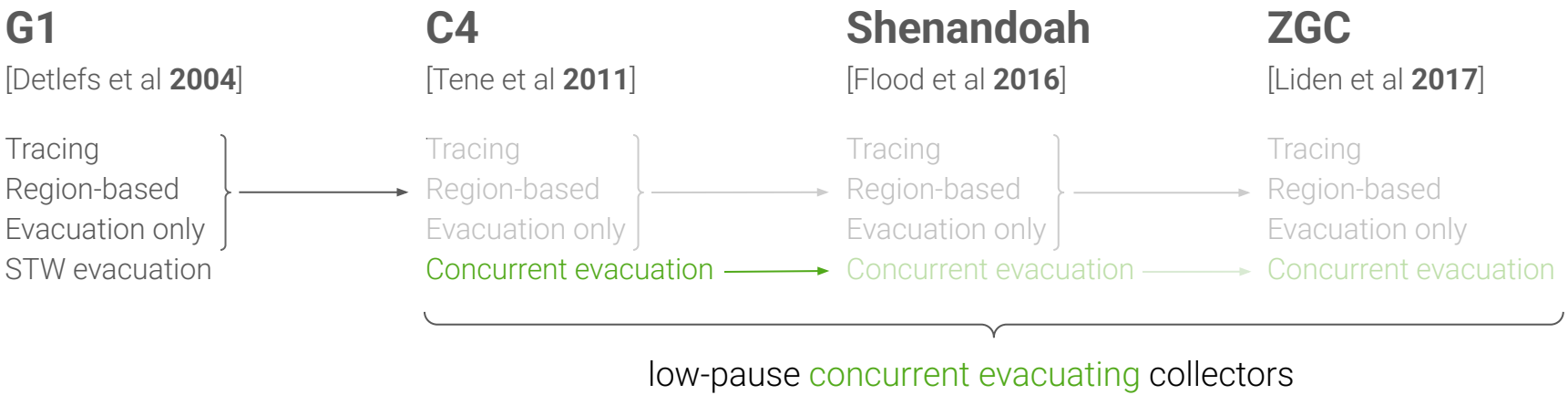
Tomcat's sample web application workload running over Apache tomcat 9.0.37., with a heap 1.3X G1's minimum.

DaCapo dev-Chopin (github 202204) using our fork of
OpenJDK JDK 11.0.11+6 +35, running on a 16/32
AMD Ryzen 5950X with 64GB DDR4-3200 memory.



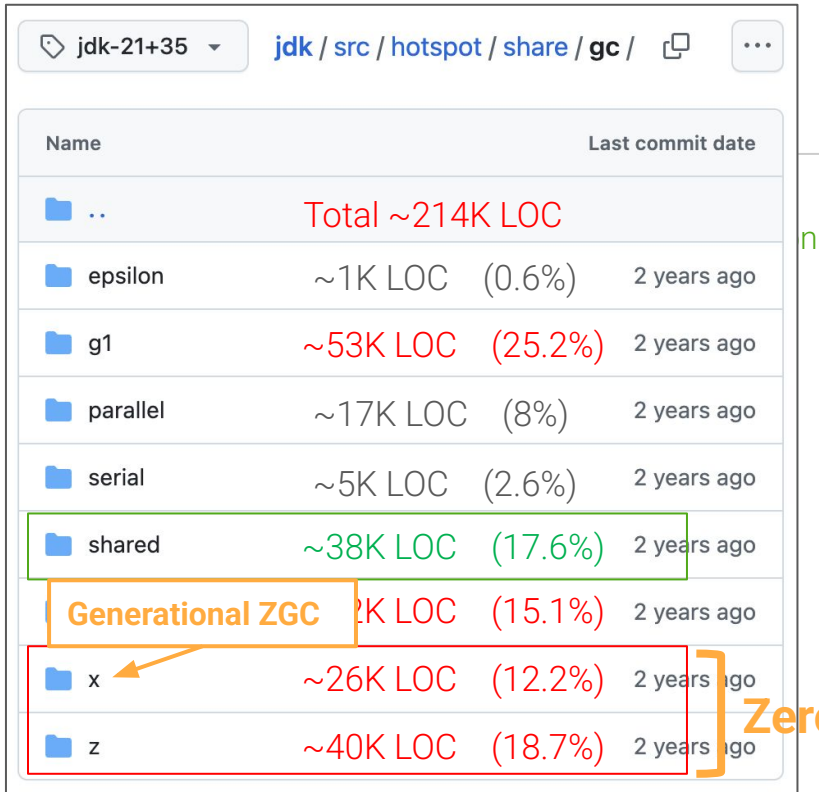
Abstraction for Innovation

Industry JVM Collectors 2004-today



Industry JVM Collectors 2004-today

G1 C4



Shenandoah
[Flood et al 2016]

ZGC
[Liden et al 2017]

Tracing
Region-based
Evacuation only
Concurrent evacuation

Tracing
Region-based
Evacuation only
Concurrent evacuation

- ✘ Low Code Reuse
- ✘ Low Maintainability
- ✘ Low Performance

Zero code reuse!

25 Years of Innovation from Modular, Extensible Design

2000 Steve refactored & built new Jikes RVM collectors, producing **JMTk**

- Pretenuring for Java, *S. M. Blackburn, S. Singhai, M. Hertz, K. S. McKinley, and J. E. B. Moss*, Proceedings of the ACM 2001 SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications. (OOPSLA), Tampa Bay FL, October 2001.
- Beltway: Getting Around Garbage Collection Gridlock, *S. M. Blackburn, R. Jones, K. S. McKinley, and J. E. B. Moss*, Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany, June 2002,.
- In or Out? Putting Write Barriers in Their Place, *S. M. Blackburn and K. S. McKinley*, International Symposium on Memory Management (ISMM), Berlin, Germany, June 2002.
- Older-first Garbage Collection in Practice: Evaluation in a Java Virtual Machine, D. Stefanovic, M. Hertz, S. M. Blackburn, K. S. McKinley, and J. E. B. Moss, Memory System Performance, Berlin, Germany, June 2002.
- Ulterior Reference Counting: Fast Garbage Collection without the Wait, *S. M. Blackburn and K. S. McKinley*, Proceedings of the ACM 2003 SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Anaheim, CA, October 2003.

2004 Steve, Perry, & I publish **MMTK** v1, 2004-2015, 100+ papers using Jikes + MMTK

- Oil and Water? High Performance Garbage Collection in Java with MMTk, *S. M. Blackburn, P. Cheng, and K. S. McKinley*, 26th International Conference on Software Engineering, pp. 137-146, Edinburgh, Scotland, May 2004. ***Citations from Google Scholar: 333***
- Myths and Realities: The Performance Impact of Garbage Collection, *S. M. Blackburn, P. Cheng, and K. S. McKinley*, ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems, pp. 25-36, New York, NY, June 2004. **Test of Time Award (June, 2014).** ***Citations from Google Scholar: 265***

2017-2025 Steve et al. (not me ;-) design and deliver **MMTk** in Rust for Jikes & **Open JDK**

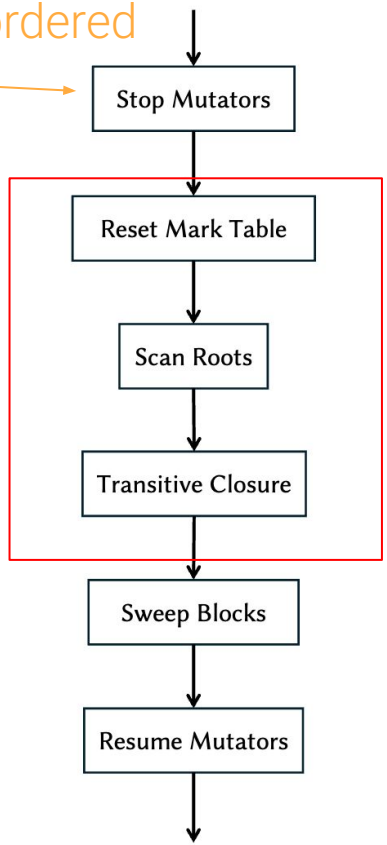
- **Rust as a language for high performance GC implementation**, Lin, Blackburn, Hosking, Norrish, in *Proceedings of the Sixteenth ACM SIGPLAN International Symposium on Memory Management, ISMM '16, Santa Barbara, CA, June 13, 2016.*
- **Reworking Memory Management in CRuby: A Practitioner Report**, Wang, Blackburn, Zhu, Valentine-House, in *ISMM, 2025.*
- **Reconsidering Garbage Collection in Julia: A Practitioner Report**, de Souza Amorim, Lin, Blackburn, Netto, Baraldi, Daly, Hosking, Pamnany, Smith, in *ISMM 2025,*

Work Packets

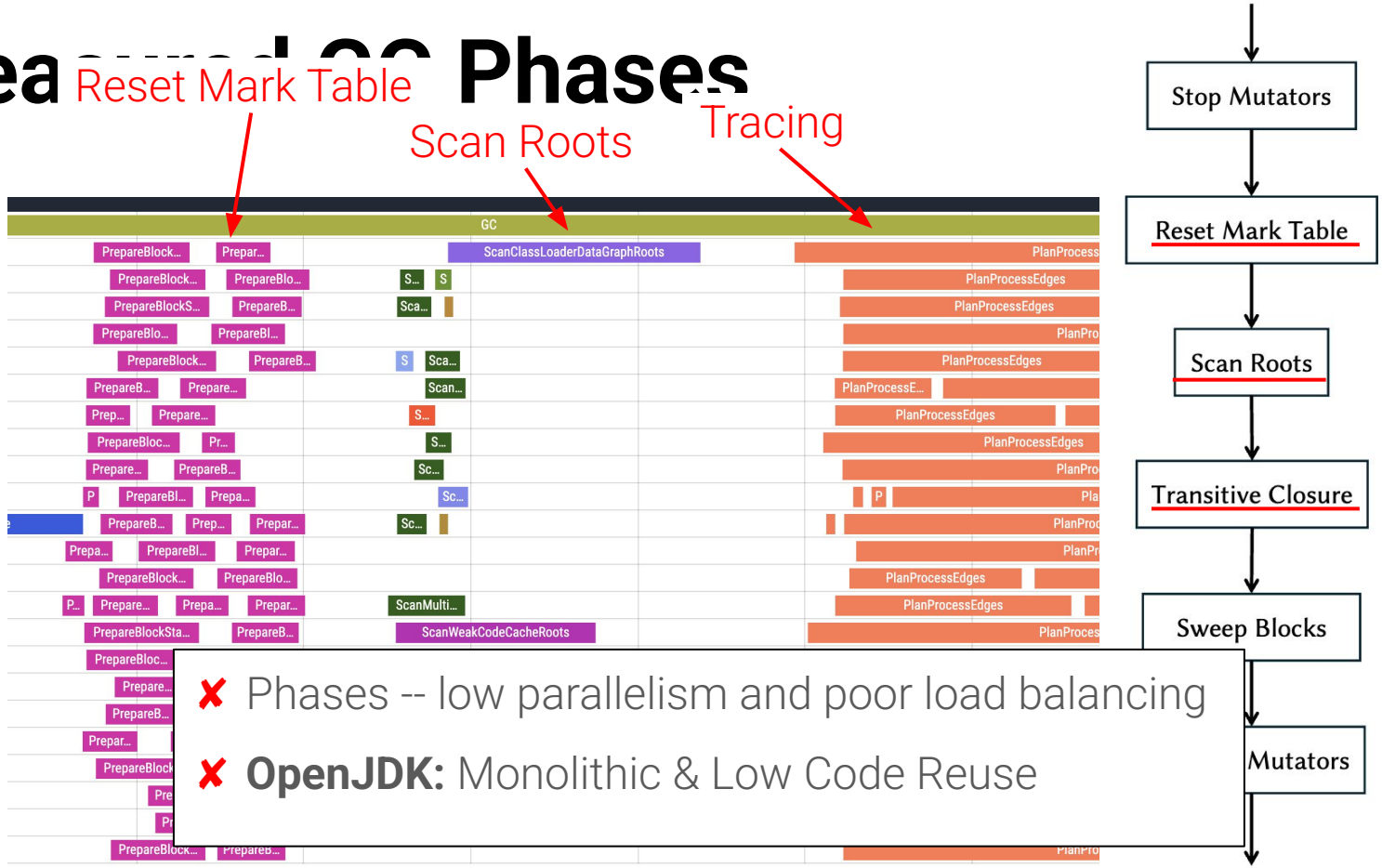
More Abstraction for
More Parallelism &
Even Better Software Engineering

Basic GC Phases

Sequentially-ordered Phases



Measured GC Phases



- ✘ Phases -- low parallelism and poor load balancing
- ✘ **OpenJDK:** Monolithic & Low Code Reuse

(Immix GC with Phases)

Work Items and Ker

Work Item: `Object obj`

Kernel:

```
void mark_objects(Object objects[]) {
  for (Object obj : objects) {
    if (attempt_mark(obj)) {
      scan_object(obj);
    }
  }
}
```

Work Items

- A smallest unit of work
  - A heap object to trace
  - A mutator stack to scan
- Work item stealing
  - Maximize parallelism

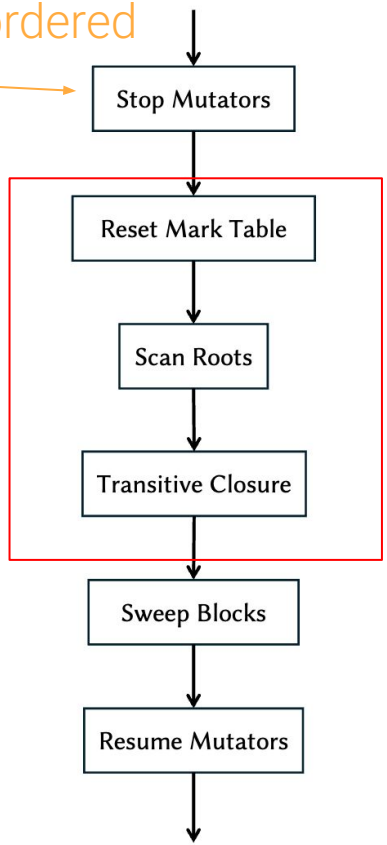
Kernels

- Small functions to process a list of same-typed work items
- Highly concentrated
  - Captures hot loops only
  - Better Locality
- Highly generic
  - Better code reuse
  - Easy to optimize, understand, & prove correct

Work Packet = < `WorkItem[]` × `Kernel` >

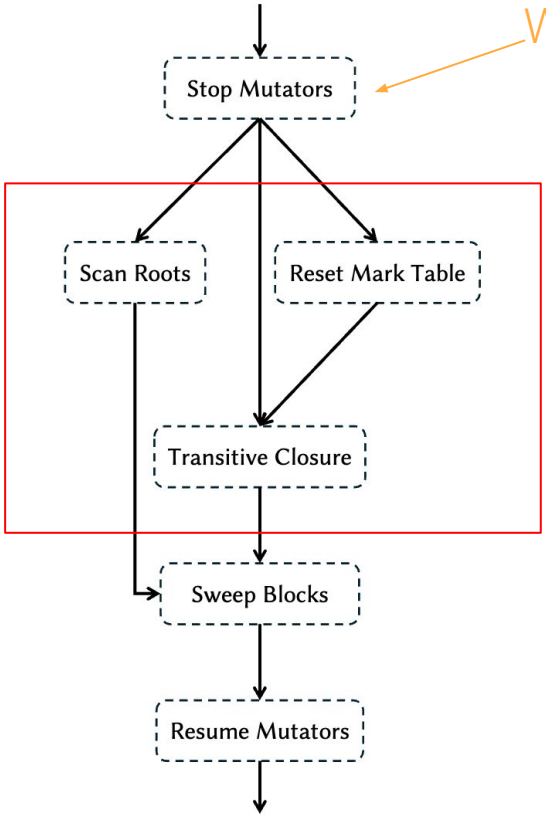
Immix with Phases

Sequentially-ordered Phases



Immix with Work Packets & Dependencies

Work Buckets



Scheduling

Work Packet Dependencies

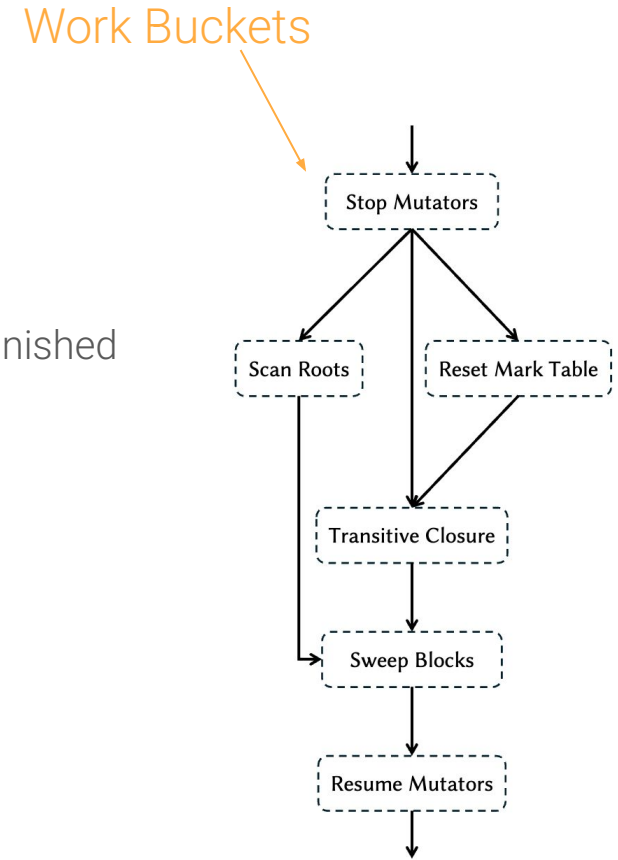
- e.g. Marking requires a properly initialized mark table
  - Marking cannot start before mark table zeroing is finished

Work Buckets

- A set of packets with same dependencies
- A dependency graph over buckets

Dependency-based scheduling

- A set of algorithm agnostic GC workers
- Workers pull work packets to execute
  - Update bucket status when necessary
- Performs both packet and item stealing



Evaluation

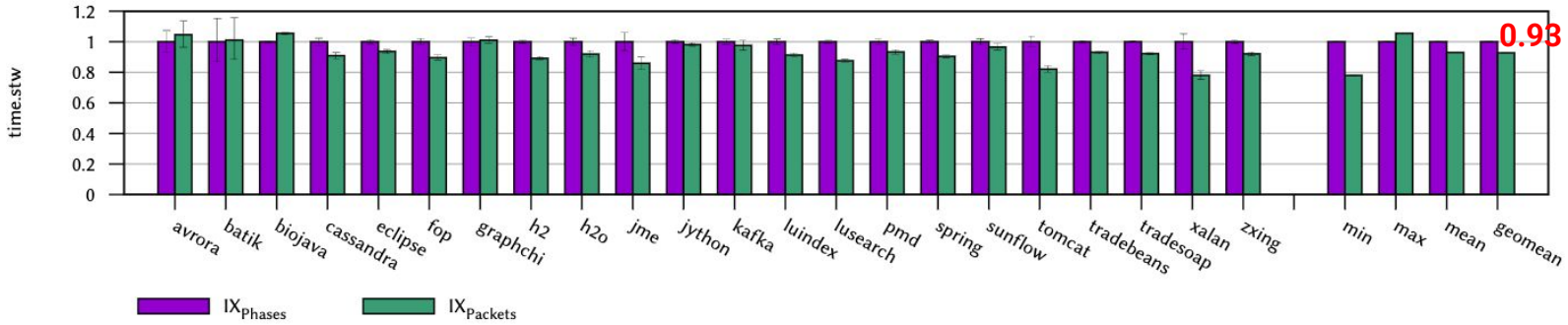
Code Reuse

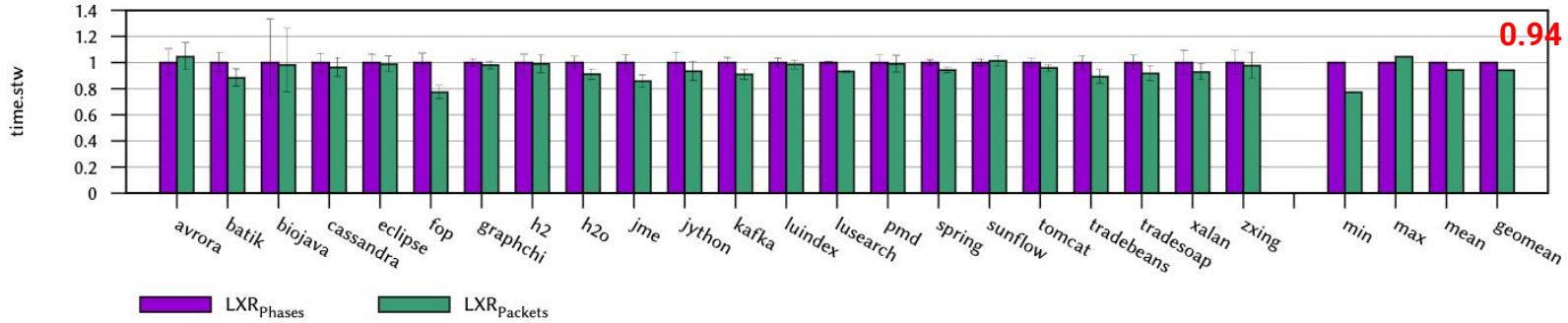
Most components are reusable (across 10 GCs)

- Algorithm agnostic scheduler
- Common work packets (28 / 49)
  - Root scanning
  - Marking & evacuation
  - Mark table zeroing
  - Sweeping
  - Stop/Resume mutators
  - ...
- GCs only need to declare their packets and dependencies
- Analysis and instrumentation code (Huang et al. 2023)

	LOC
LXR	3.8K (8.6%)
Immix	0.5K (1.1%)
10 GCs Total	45.1K (100%)

GC Pause Time





Lessons for Parallel Programming

- Programming model transferable to other parallel computation systems
  - A generic abstraction: Work packets and kernels
    - Elegantly handle multiple different types of work with dependencies
  - Two-level scheduling, generalizes work stealing
    - Efficient work stealing with low overhead
  - Reusable and user-friendly analysis and evaluation tools

25 Years of Garbage Collection Innovation

Breaking the **Time & Space Tradeoff** with **Memory Performance**

Immix: Hierarchy of Lines and Blocks with judicious copying

Breaking Low Latency & GC Pause Time Tradeoff for **Parallel Performance**

LXR: Reference Counting with even less copying in pauses, concurrent tracing

Abstraction for Innovation

MMTk & Work Packets

Benchmarks, Performance Methodologies & More Methodologies

A research journey

Thank you!

GC Algorithms

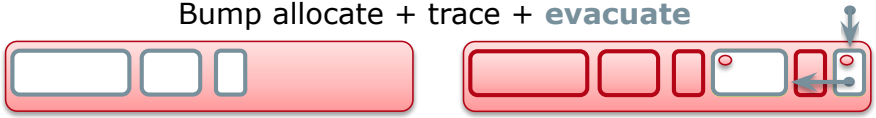
Mark-Sweep [McCarthy 1960]

Free-list + trace + **sweep-to-free**

- Allocation produces **poor locality** in multiple size-based free lists.
- **Some fragmentation**
- **Low computational overhead** due to super fast tracing & reclamation, as objects do not move

Semi-Space [Cheney 1970]

Bump allocate + trace + **evacuate**



- Super fast allocation with **good locality**
- **No fragmentation**
- Tracing and evacuation adds computation & **lots of memory overhead** by moving all the objects

Mark-Compact [Styger 1967]

Bump allocate + trace + **compact**

- Super fast allocation with **good locality**
- **No fragmentation**
- Tracing and compacting adds **high multipass computational overhead** with **low memory overhead**

Mark-Region

Bump allocate + trace + sweep-to-region

- Super fast allocation with **good locality**
- **Low fragmentation**
- Tracing with occasional copying yields **low computational and low memory overheads**