Schism: Fragmentation-Tolerant Real-Time Garbage Collection

Fil Pizlo[†]
Tony Hosking[*]
Luke Ziarek[†]
Ethan Blanton[†]
Peta Maj[*]
Jan Vitek[†*]

Fiji Systems Inc.[†]

Purdue University[*]

Why another Real Time Garbage Collector?

Why another Real Time Garbage Collector?

- Real-time programmers want *hard bounds* on both ***Space*** <u>and</u> ***Time***.

Why another Real Time Garbage Collector?

- Real-time programmers want *hard bounds* on both ***Space*** and ***Time.***
- Previous RTGCs either:
  - **Fail to bound space**, or

Why another Real Time Garbage Collector?

- Real-time programmers want *hard bounds* on both ***Space*** and ***Time***.
- Previous RTGCs either:
  - **Fail to bound space**, or
  - **Cause large slow-downs**.

Why another Real Time Garbage Collector?

- Real-time programmers want *hard bounds* on both ***Space*** <u>and</u> ***Time***.
- Previous RTGCs either:
    - **Fail to bound space**, or
    - **Cause large slow-downs**.
- *We propose a new RTGC called Schism, which*
    - ***bounds space*** *while*

Why another Real Time Garbage Collector?

- Real-time programmers want *hard bounds* on both ***Space*** <u>and</u> ***Time***.
- Previous RTGCs either:
  - **Fail to bound space**, or
  - **Cause large slow-downs**.
- *We propose a new RTGC called Schism, which*
  - ***bounds space*** *while*
  - ***running faster*** *than other RTGCs.*

What **Schism** Real-Time GC provides:

What **Schism** Real-Time GC provides:

- executes **concurrently**
- guarantees **progress** for heap accesses
- **minimizes** heap access overhead
- gives uniformly good **throughput**
- minimizes external **fragmentation**

What **Schism** Real-Time GC provides:

- executes **concurrently** *preemptible at any time*
- guarantees **progress** for heap accesses
- **minimizes** heap access overhead
- gives uniformly good **throughput**
- minimizes external **fragmentation**

What **Schism** Real-Time GC provides:

- executes **concurrently** *preemptible at any time*
- guarantees **progress** for heap accesses *wait-free*
- **minimizes** heap access overhead
- gives uniformly good **throughput**
- minimizes external **fragmentation**

What **Schism** Real-Time GC provides:

- executes **concurrently** *preemptible at any time*
- guarantees **progress** for heap accesses *wait-free*
- **minimizes** heap access overhead *O(1), a few instructions*
- gives uniformly good **throughput**
- minimizes external **fragmentation**

What **Schism** Real-Time GC provides:

- executes **concurrently** *preemptible at any time*
- guarantees **progress** for heap accesses *wait-free*
- **minimizes** heap access overhead *O(1), a few instructions*
- gives uniformly good **throughput** *fastest RTGC*
- minimizes external **fragmentation**

What **Schism** Real-Time GC provides:

- executes **concurrently** *preemptible at any time*
- guarantees **progress** for heap accesses *wait-free*
- **minimizes** heap access overhead *O(1), a few instructions*
- gives uniformly good **throughput** *fastest RTGC*
- minimizes external **fragmentation** *proven space bounds (see appendix)*

Real Time Garbage Collection: state of the art

Real Time Garbage Collection: state of the art

- Baseline: **HotSpot 1.6** collector: Fast, hard space bounds.

Real Time Garbage Collection: state of the art

- Baseline: **HotSpot 1.6** collector: Fast, hard space bounds.
  - *but*: **not concurrent**, not suitable for hard real-time

Real Time Garbage Collection: state of the art

- Baseline: **HotSpot 1.6** collector: Fast, hard space bounds.
  - *but*: **not concurrent**, not suitable for hard real-time
- Java RTS: hard space bounds, concurrent, wait-free.

Real Time Garbage Collection: state of the art

- Baseline: **HotSpot 1.6** collector: Fast, hard space bounds.
    - *but*: **not concurrent**, not suitable for hard real-time
- Java RTS: hard space bounds, concurrent, wait-free.
    - *but*: **60% slow-down**, **logarithmic** heap access

Real Time Garbage Collection: state of the art

- Baseline: **HotSpot 1.6** collector: Fast, hard space bounds.
  - *but*: **not concurrent**, not suitable for hard real-time
- Java RTS: hard space bounds, concurrent, wait-free.
  - *but*: **60% slow-down**, **logarithmic** heap access
- J9 SRT (Metronome): only 30% slow-down, concurrent, wait-free.

Real Time Garbage Collection: state of the art

- Baseline: **HotSpot 1.6** collector: Fast, hard space bounds.
  - *but*: **not concurrent**, not suitable for hard real-time
- Java RTS: hard space bounds, concurrent, wait-free.
  - *but*: **60% slow-down**, **logarithmic** heap access
- J9 SRT (Metronome): only 30% slow-down, concurrent, wait-free.
  - *but*: **susceptible to fragmentation**

Real Time Garbage Collection: state of the art

- Baseline: **HotSpot 1.6** collector: Fast, hard space bounds.
  - *but*: **not concurrent**, not suitable for hard real-time
- Java RTS: hard space bounds, concurrent, wait-free.
  - *but*: **60% slow-down**, **logarithmic** heap access
- J9 SRT (Metronome): only 30% slow-down, concurrent, wait-free.
  - *but*: **susceptible to fragmentation**

We want something as fast as Metronome, but fragmentation-tolerant like Java RTS.

Previous Approaches to Minimizing Fragmentation in RTGC

On-demand Defragmentation

On-demand Defragmentation

- Stop-the-world or incremental: simple, but causes pauses.
    - *we don't want pauses.*

On-demand Defragmentation

- Stop-the-world or incremental: simple, but causes pauses.
  - *we don’t want pauses.*
- Concurrent: still has draw-backks

On-demand Defragmentation

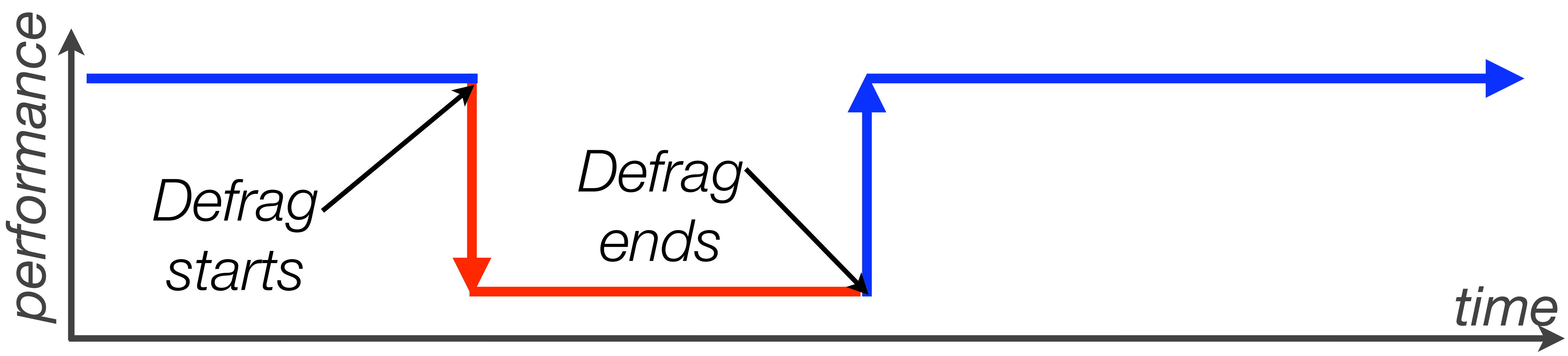
- Stop-the-world or incremental: simple, but causes pauses.
  - *we don’t want pauses.*
- Concurrent: still has draw-backs
  - Custom hardware? [Click et al ’05]

On-demand Defragmentation

- Stop-the-world or incremental: simple, but causes pauses.
  - *we don't want pauses.*
- Concurrent: still has draw-backs
  - Custom hardware? [Click et al '05]
  - throughput penalty during defrag is 5x or more. [Pizlo et al '07], [Pizlo et al '08]

On-demand Defragmentation

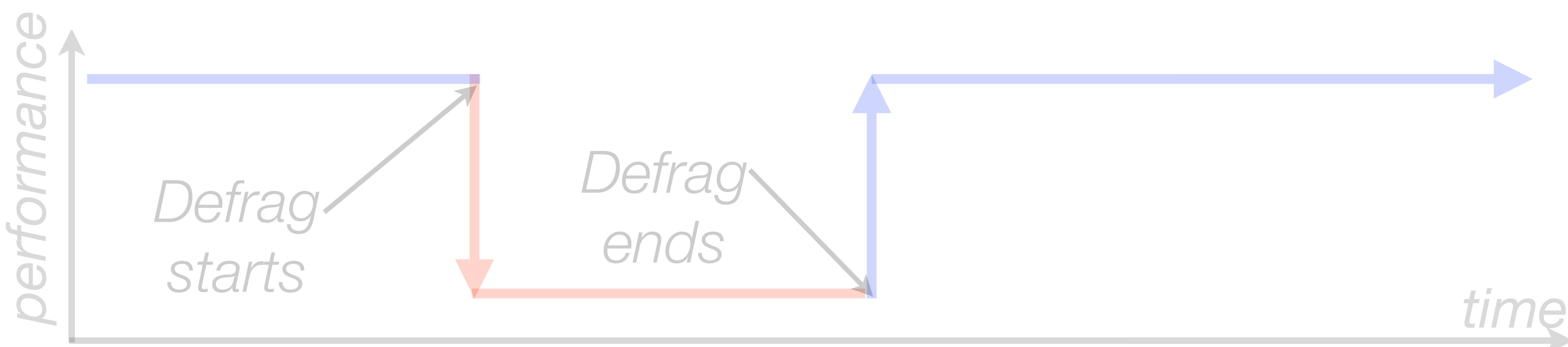
- Stop-the-world or incremental: simple, but causes pauses.
  - *we don't want pauses.*
- Concurrent: still has draw-backs
  - Custom hardware? [Click et al '05]
  - throughput penalty during defrag is 5x or more. [Pizlo et al '07], [Pizlo et al '08]



On-demand Defragmentation

- Stop-the-world or incremental: simple, but causes pauses.
  - *we don't want pauses.*
- Concurrent: still has draw-backs
  - Custom hardware? [Click et al '05]
  - throughput penalty during defrag is 5x or more. [Pizlo et al '07], [Pizlo et al '08]

Worst-case throughput penalty is too large.



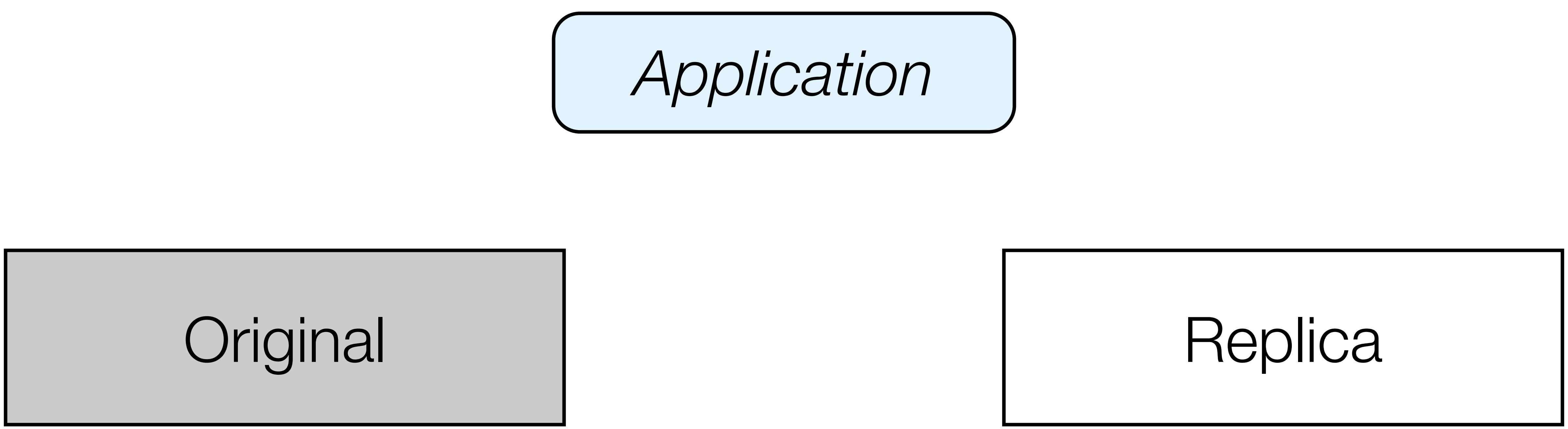
Replication-based GC

Replication-based GC

- See: [Nettles-O'Toole '93], [Cheng-Blelloch '01]
- Allows concurrent defragmentation

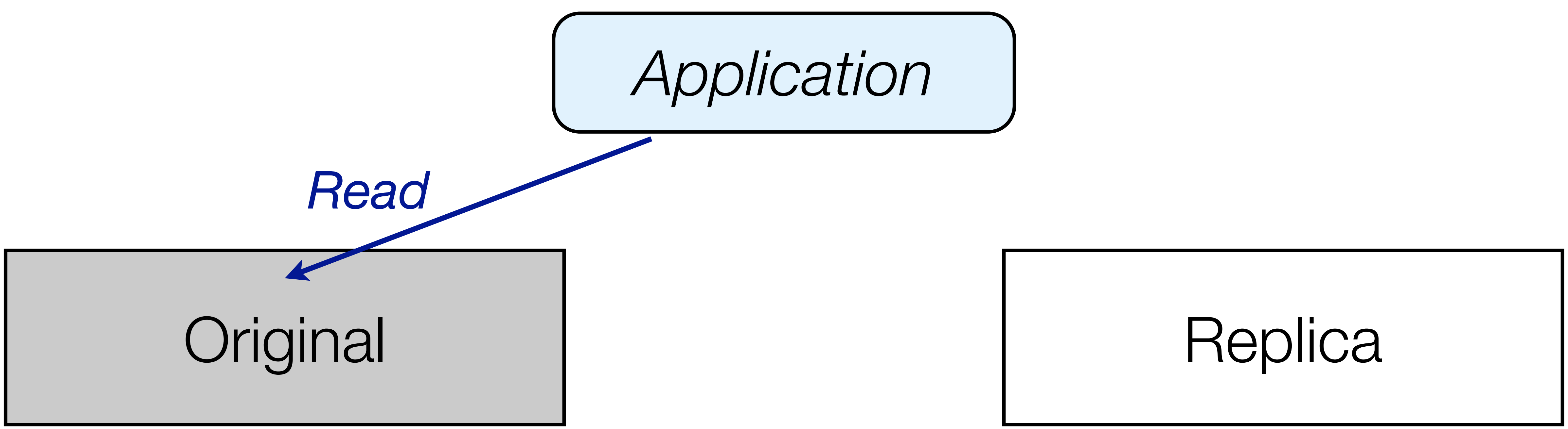
Replication-based GC

- See: [Nettles-O'Toole '93], [Cheng-Blelloch '01]
- Allows concurrent defragmentation
- Two spaces: one space for reads; writes "replicated" to both spaces



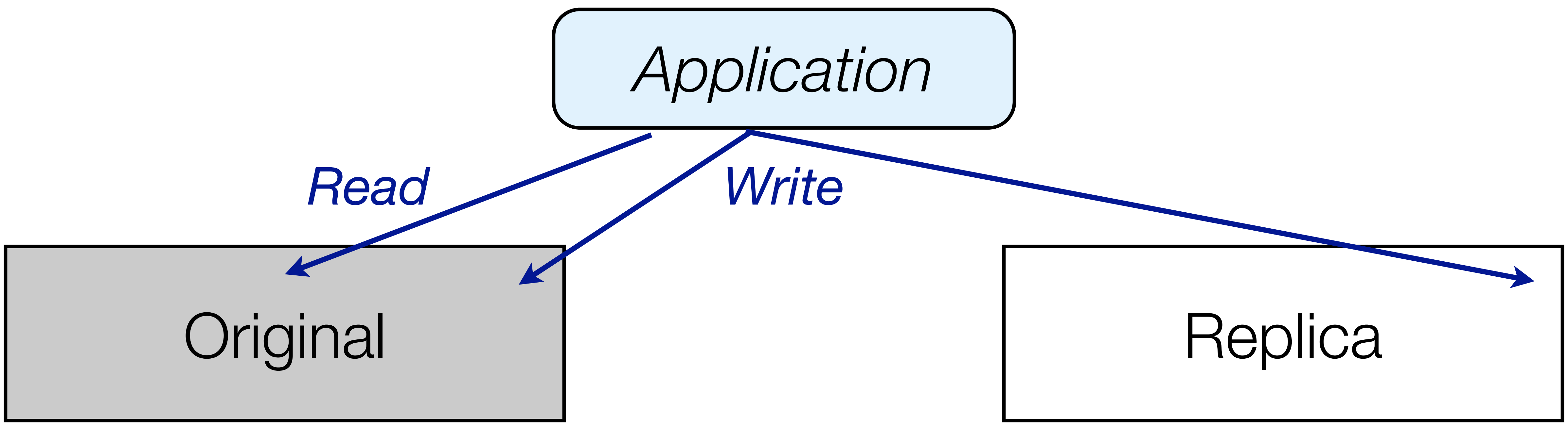
Replication-based GC

- See: [Nettles-O'Toole '93], [Cheng-Blelloch '01]
- Allows concurrent defragmentation
- Two spaces: one space for reads; writes "replicated" to both spaces



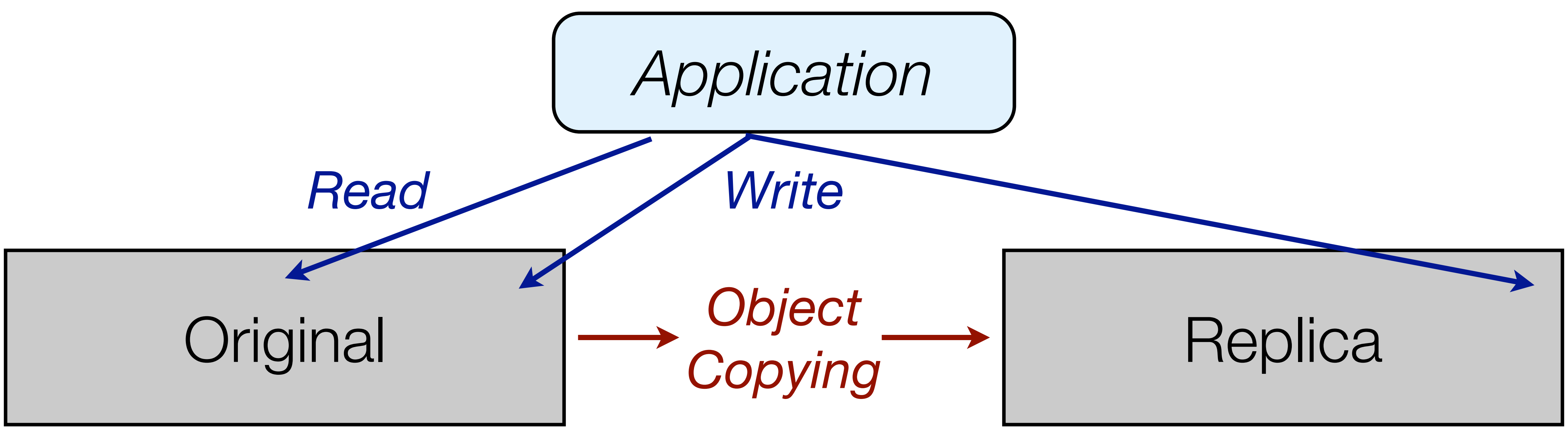
Replication-based GC

- See: [Nettles-O'Toole '93], [Cheng-Blelloch '01]
- Allows concurrent defragmentation
- Two spaces: one space for reads; writes "replicated" to both spaces



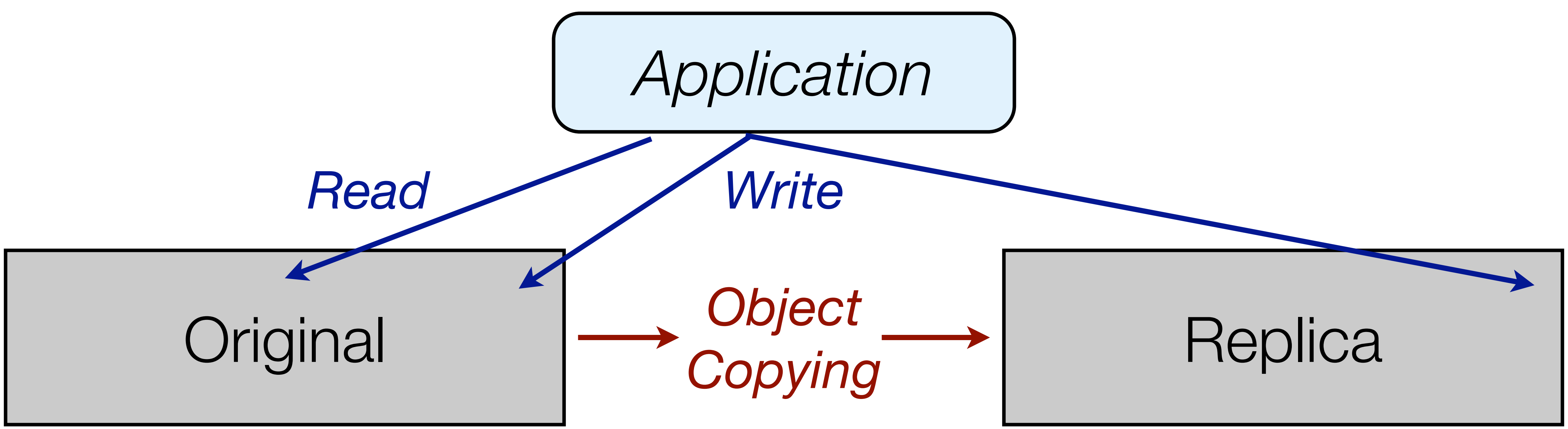
Replication-based GC

- See: [Nettles-O'Toole '93], [Cheng-Blelloch '01]
- Allows concurrent defragmentation
- Two spaces: one space for reads; writes "replicated" to both spaces



Replication-based GC

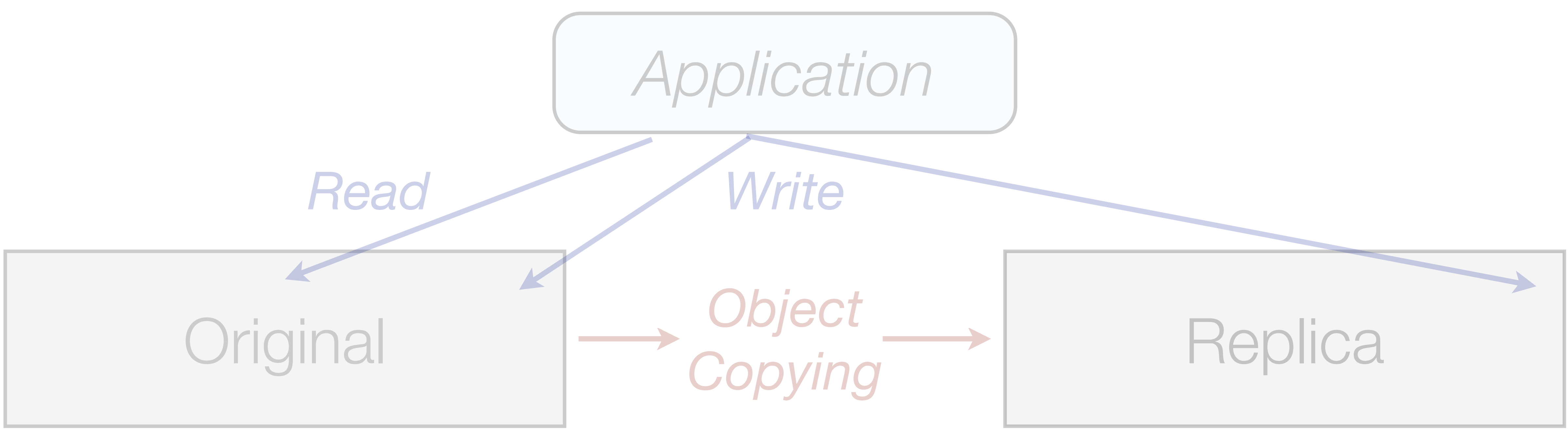
- See: [Nettles-O'Toole '93], [Cheng-Blelloch '01]
- Allows concurrent defragmentation
- Two spaces: one space for reads; writes "replicated" to both spaces
- Problem: **Writes not atomic**! Loss of coherence!



Replication-based GC

- See: [Nettles-O'Toole '93], [Cheng-Blelloch '01]
- Allows concurrent defragmentation
- Two spaces: one space for reads; writes "replicated" to both spaces
- Problem: Writes not atomic! Loss of coherence!

Works best for immutable objects.



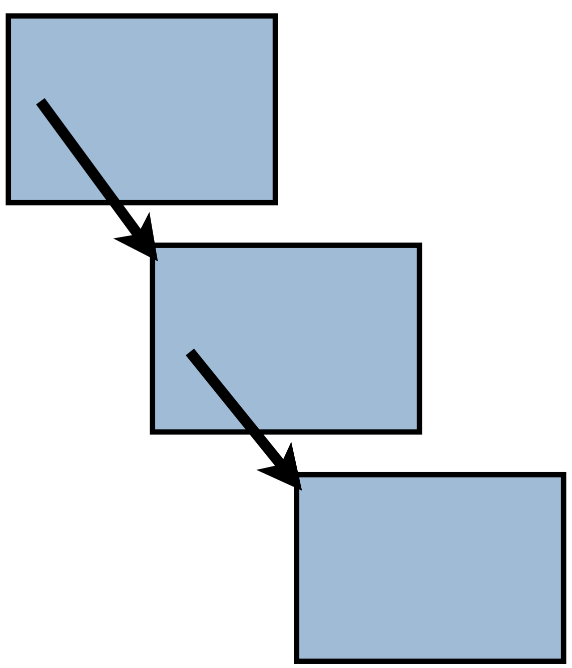
Allocate in fragments [Siebert '99]

- All objects split into small fragments.
- Fragment size is typically fixed at 32 bytes.
- Fragments are linked, application must follow links on object access.

Allocate in fragments [Siebert '99]

- All objects split into small fragments.
- Fragment size is typically fixed at 32 bytes.
- Fragments are linked, application must follow links on object access.

Plain Object



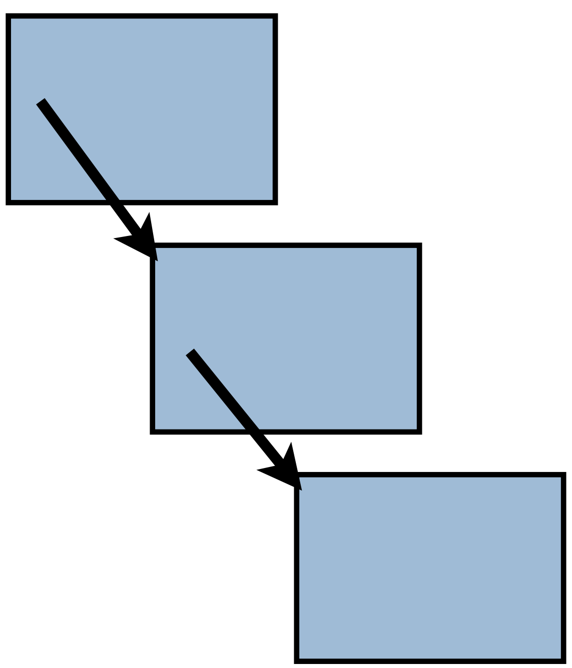
Access cost is known statically, does not vary.

Most objects require only two fragments.

Allocate in fragments [Siebert '99]

- All objects split into small fragments.
- Fragment size is typically fixed at 32 bytes.
- Fragments are linked, application must follow links on object access.

Plain Object



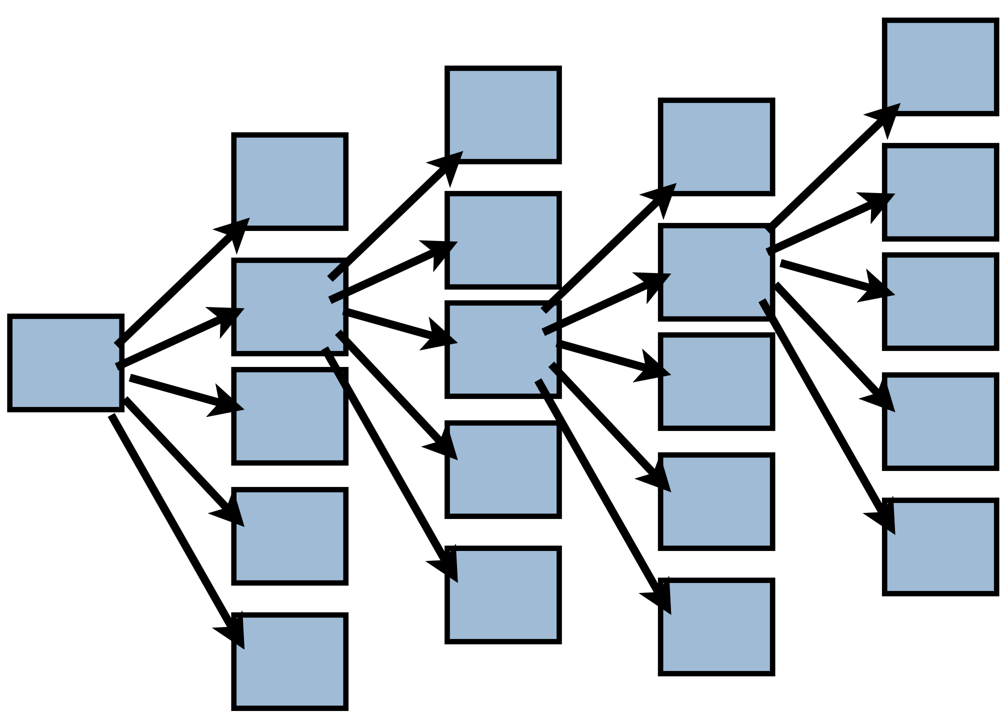
Access cost is known statically, does not vary.

Most objects require only two fragments.

Allocate in fragments [Siebert '99]

- All objects split into small fragments.
- Fragment size is typically fixed at 32 bytes.
- Fragments are linked, application must follow links on object access.

Array



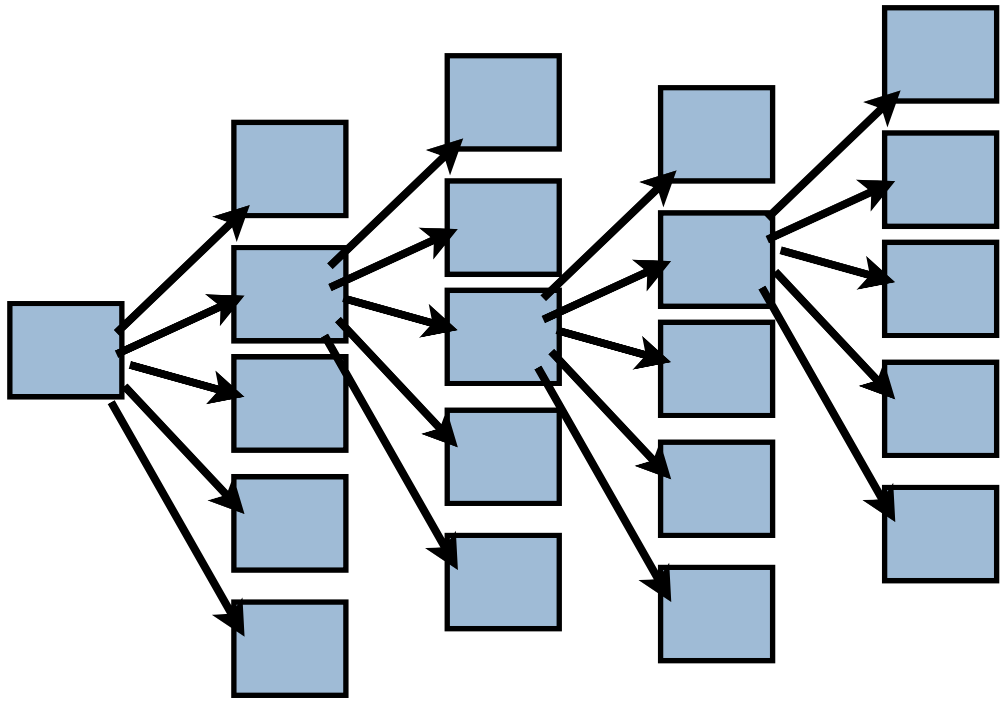
Access cost is logarithmic.

Array accesses will see significant slow-down!

Allocate in fragments [Siebert '99]

- All objects split into small fragments.
- Fragment size is typically fixed at 32 bytes.
- Fragments are linked, application must follow links on object access.

Array



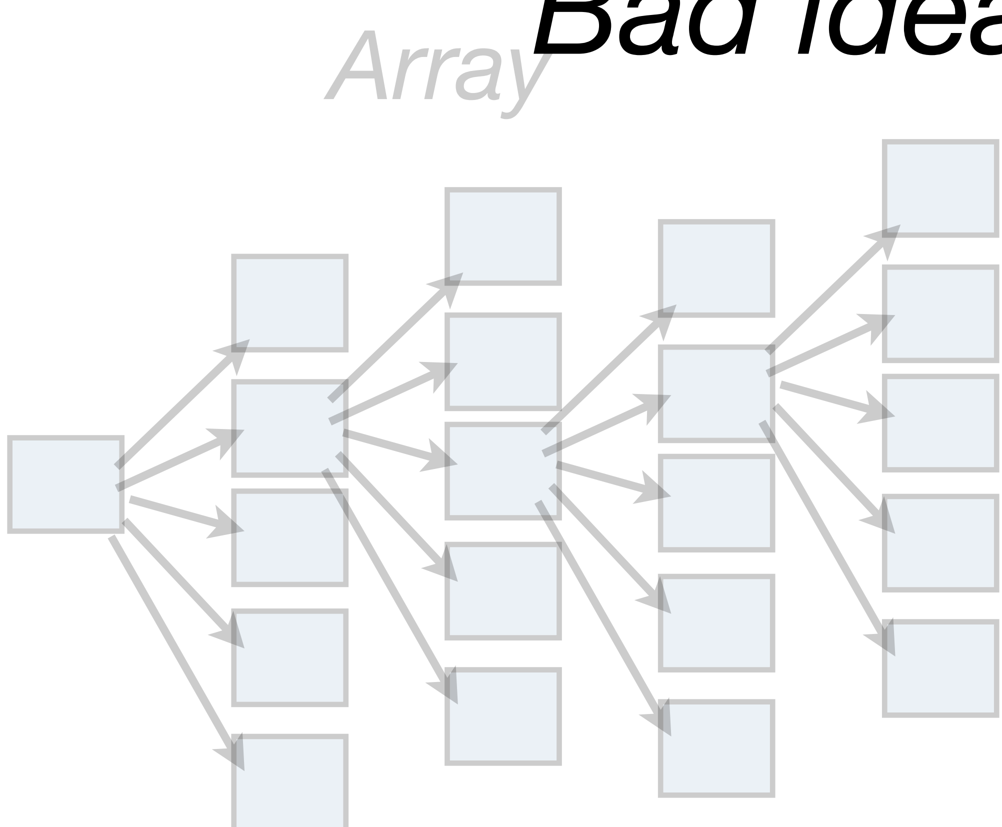
Access cost is logarithmic.

Array accesses will see significant slow-down!

Allocate in fragments [Siebert '99]

- All objects split into small fragments.
- Fragment size is typically fixed at 32 bytes.
- Fragments are linked, application must follow links on object access.

Bad idea for large arrays.



Access cost is logarithmic.

Array accesses will see significant slow-down!

Synopsis

- Replication-copying Collection:
  - *great, but only for immutable objects*
- Fragmented Allocation:
  - *great, unless you have large arrays*

Synopsis

- Replication-copying Collection:
  - *great, but only for immutable objects*
- Fragmented Allocation:
  - *great, unless you have large arrays*

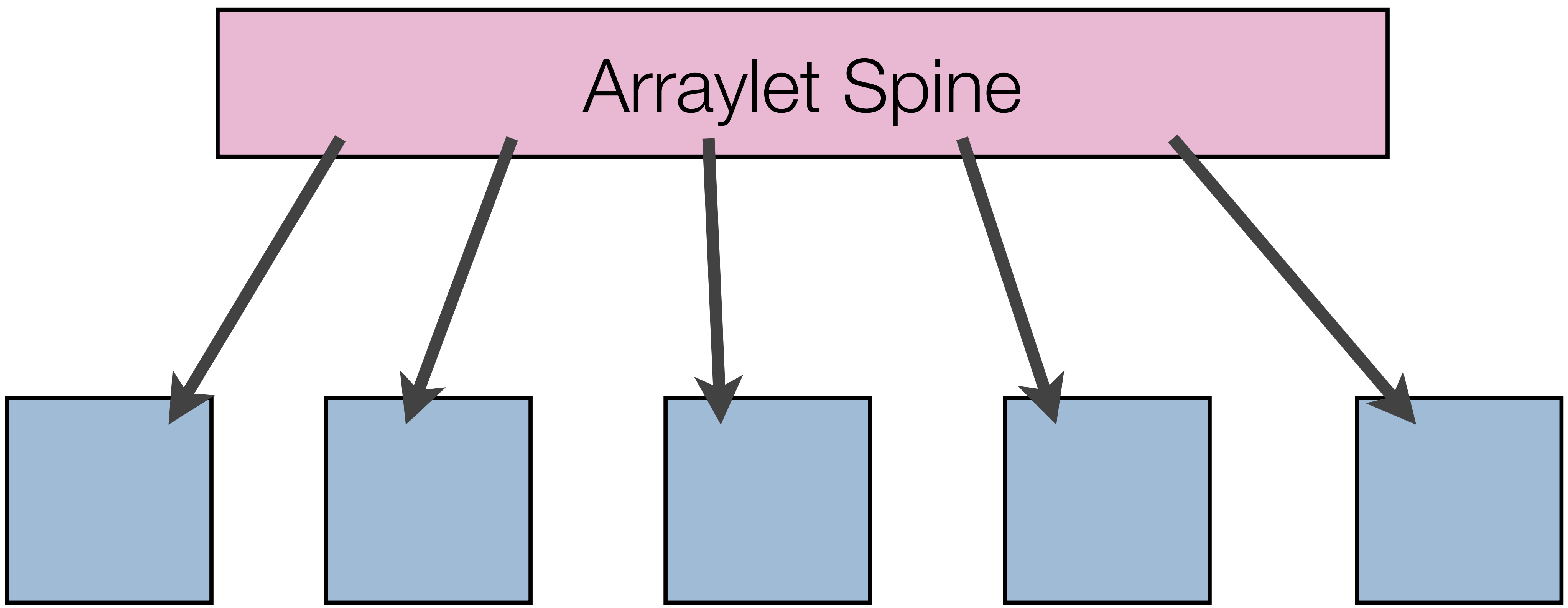
Can we combine the two?

Idea:

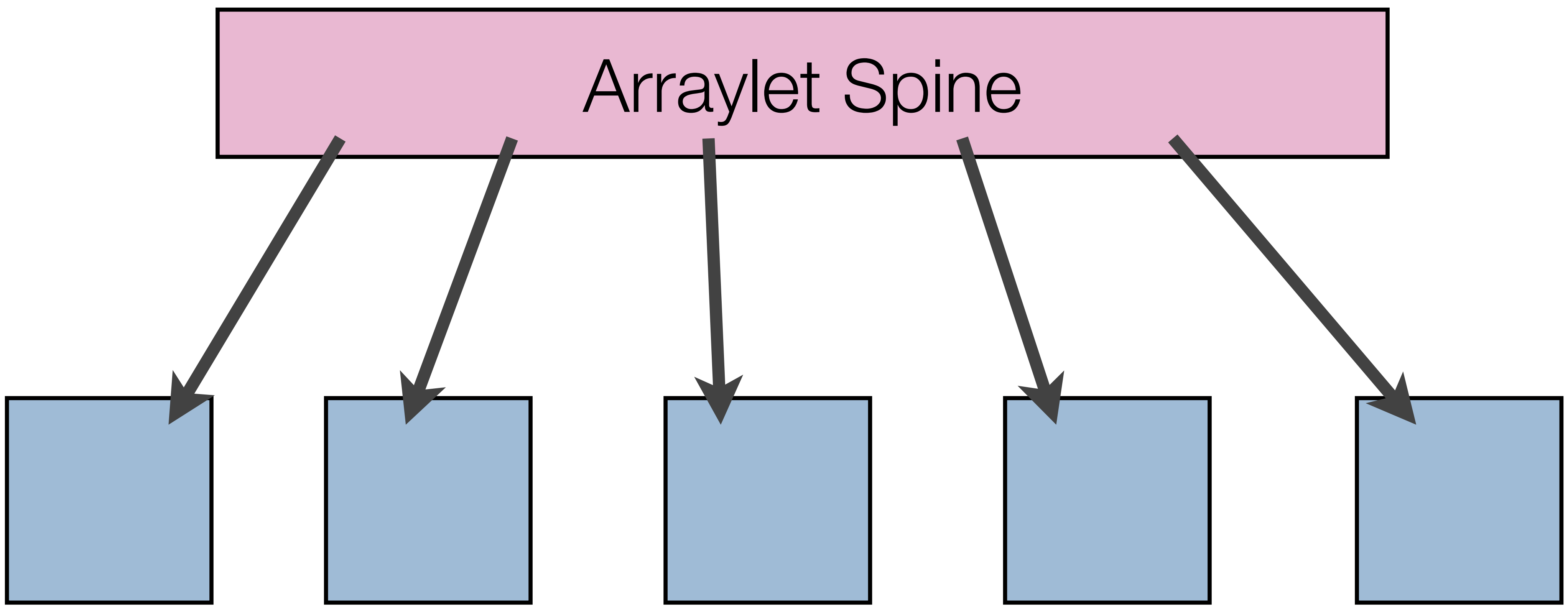
combine *Fragmented Allocation*
with *Replication-Copying*
using *Arraylets*

A new way of exploiting Arraylets

A new way of exploiting Arraylets



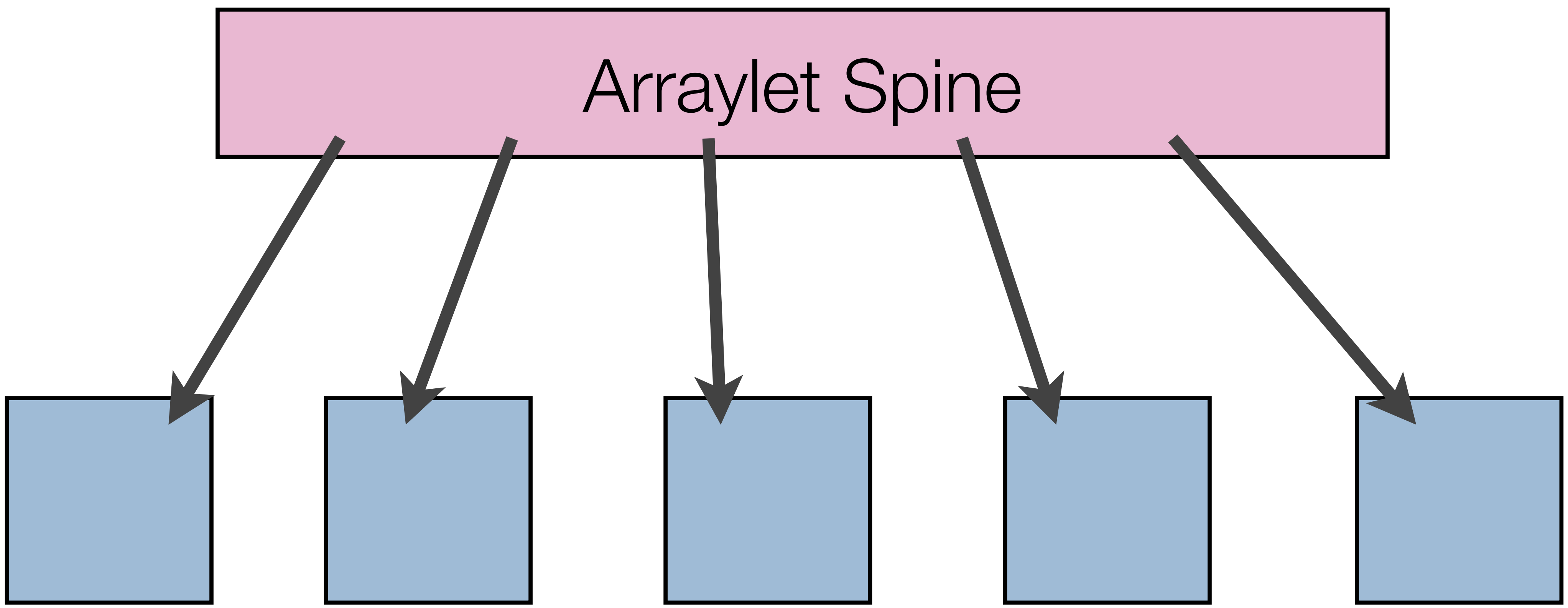
A new way of exploiting Arraylets



Fragments have fixed size - no external fragmentation

A new way of exploiting Arraylets

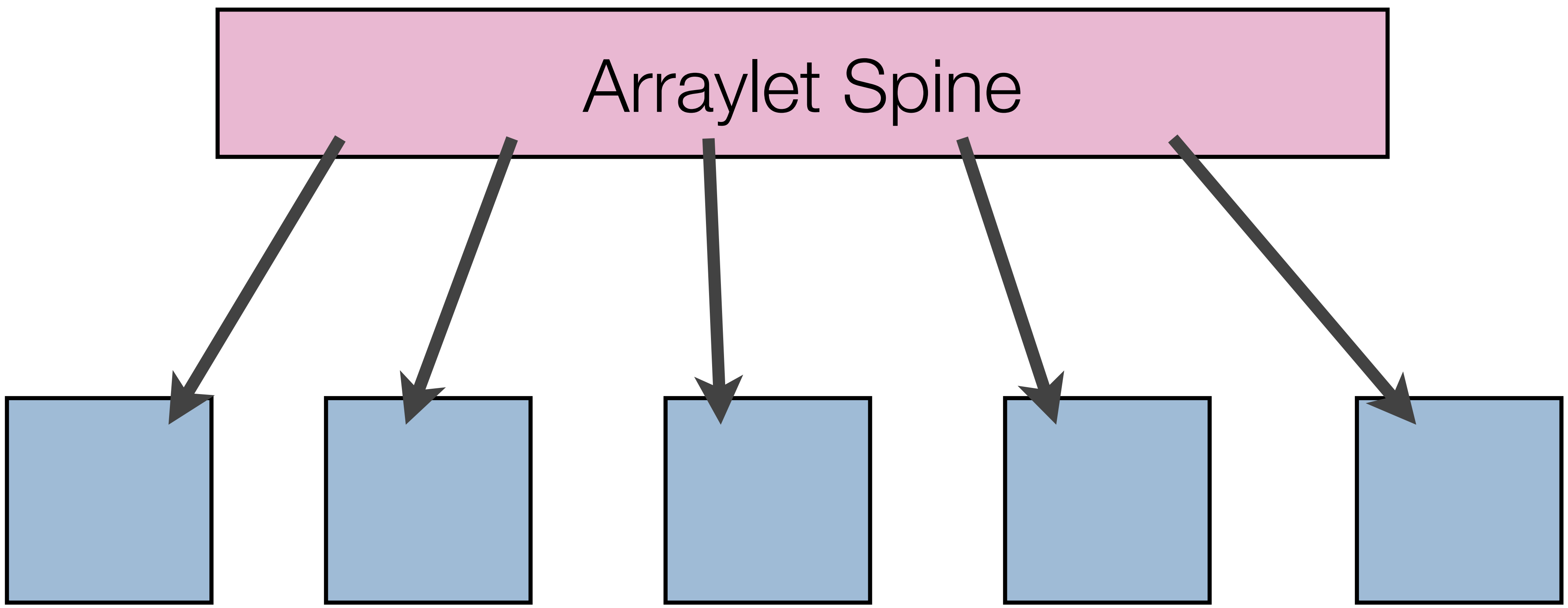
The Arraylet Spine has variable size, which can lead to fragmentation!



Fragments have fixed size - no external fragmentation

A new way of exploiting Arraylets

But the spine is immutable ...

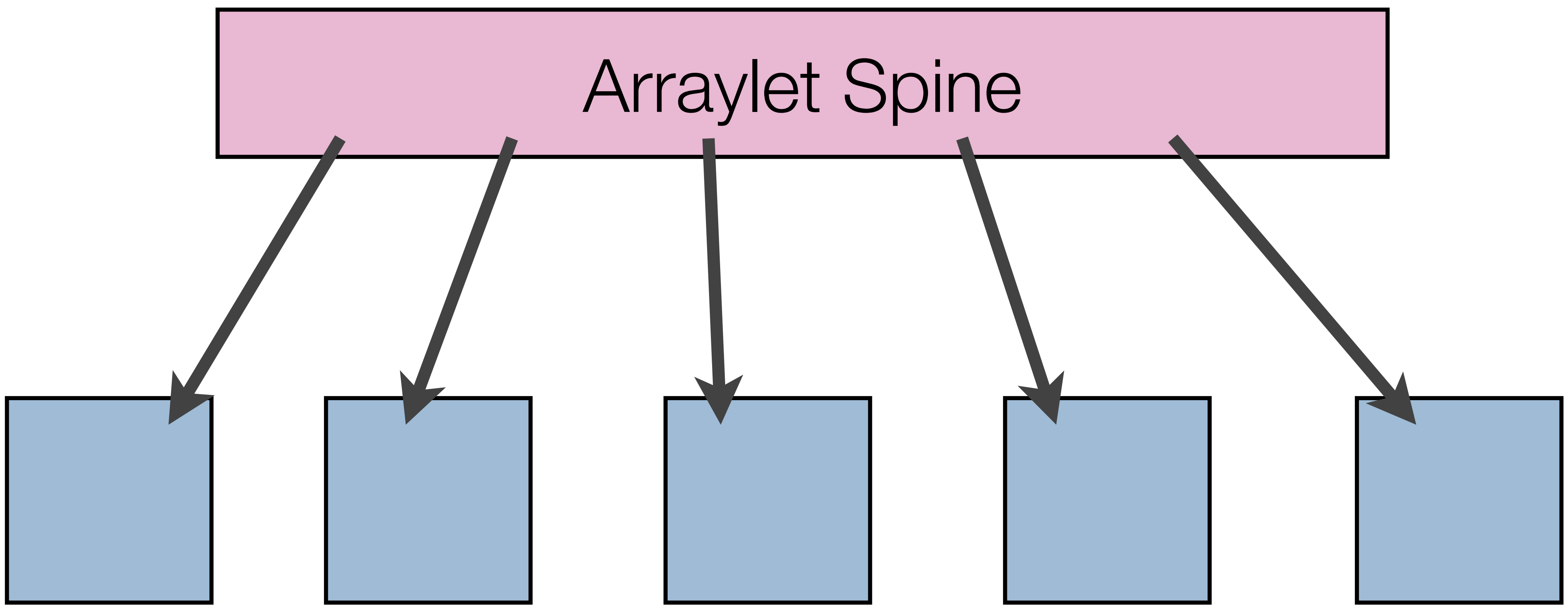


Fragments have fixed size - no external fragmentation

A new way of exploiting Arraylets

But the spine is immutable ...

... and <u>replication</u> is ideal for immutable objects

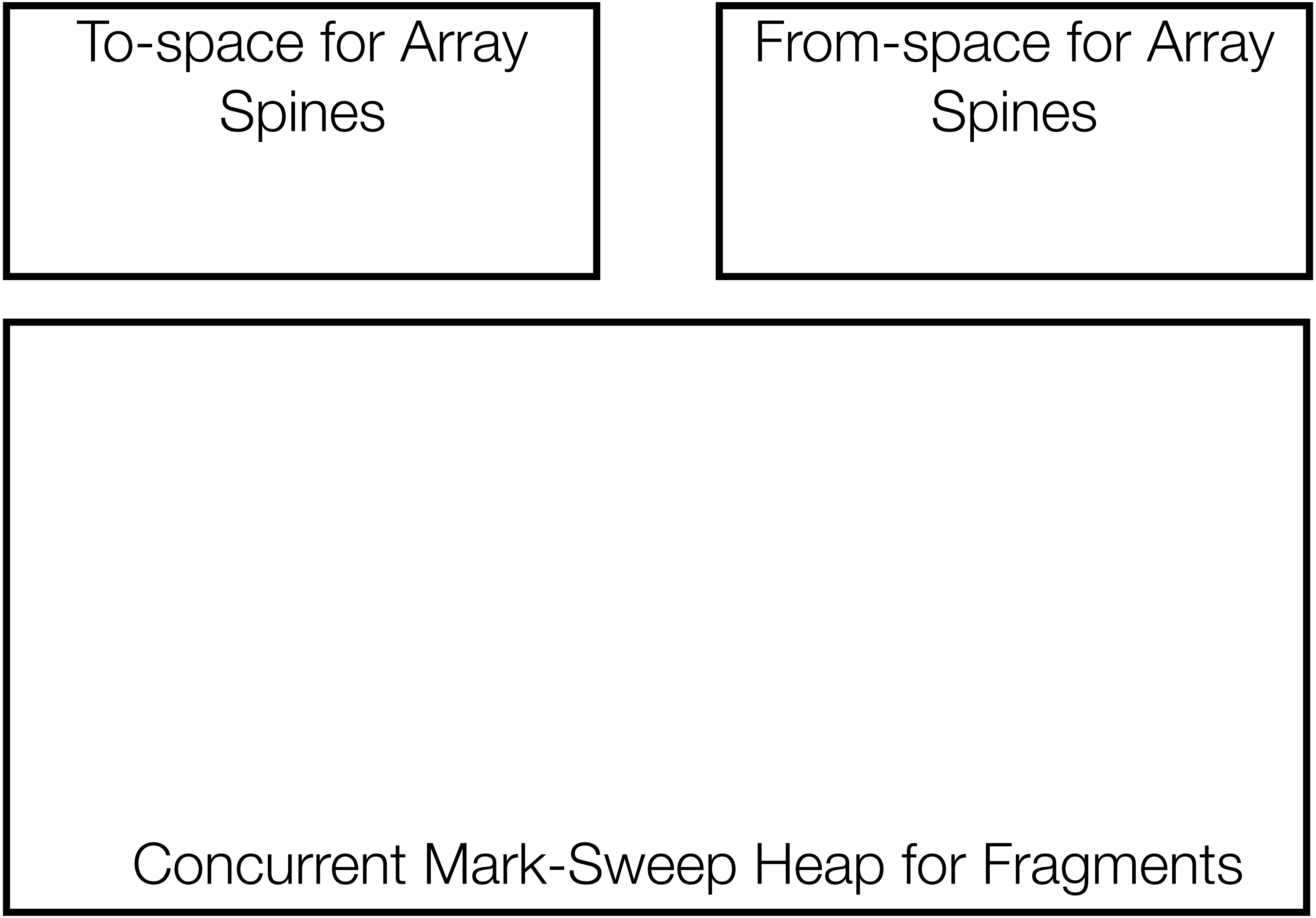


Fragments have fixed size - no external fragmentation

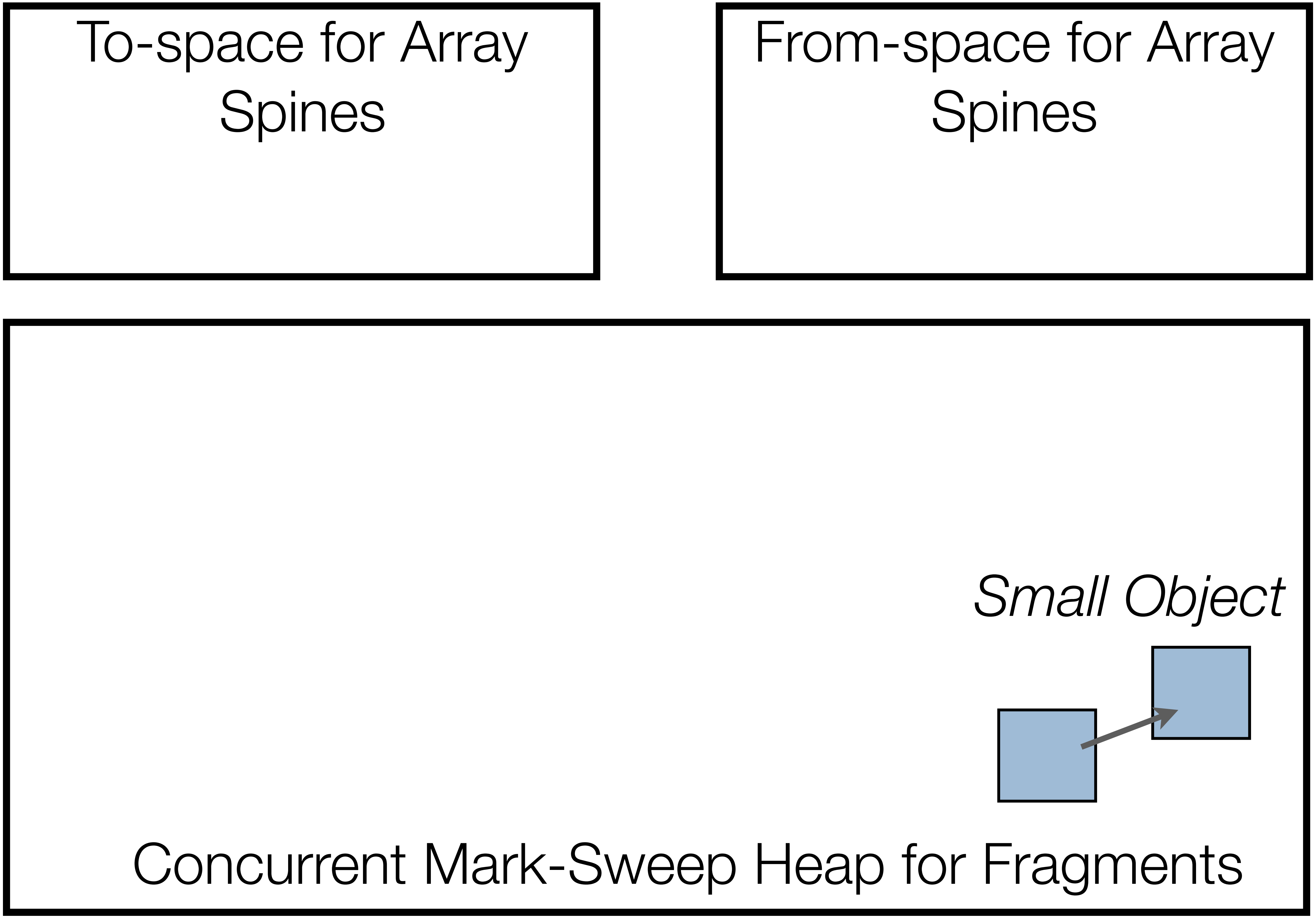
Schism = arraylets + replication + fragments

- Combination:
  - Concurrent **mark-sweep GC** for fixed-size **fragments**
  - **Replication copying** for variable-size **arraylet** spines
- *No external fragmentation for either fragments or spines*
- *Heap access is O(1), wait-free, and coherent.*

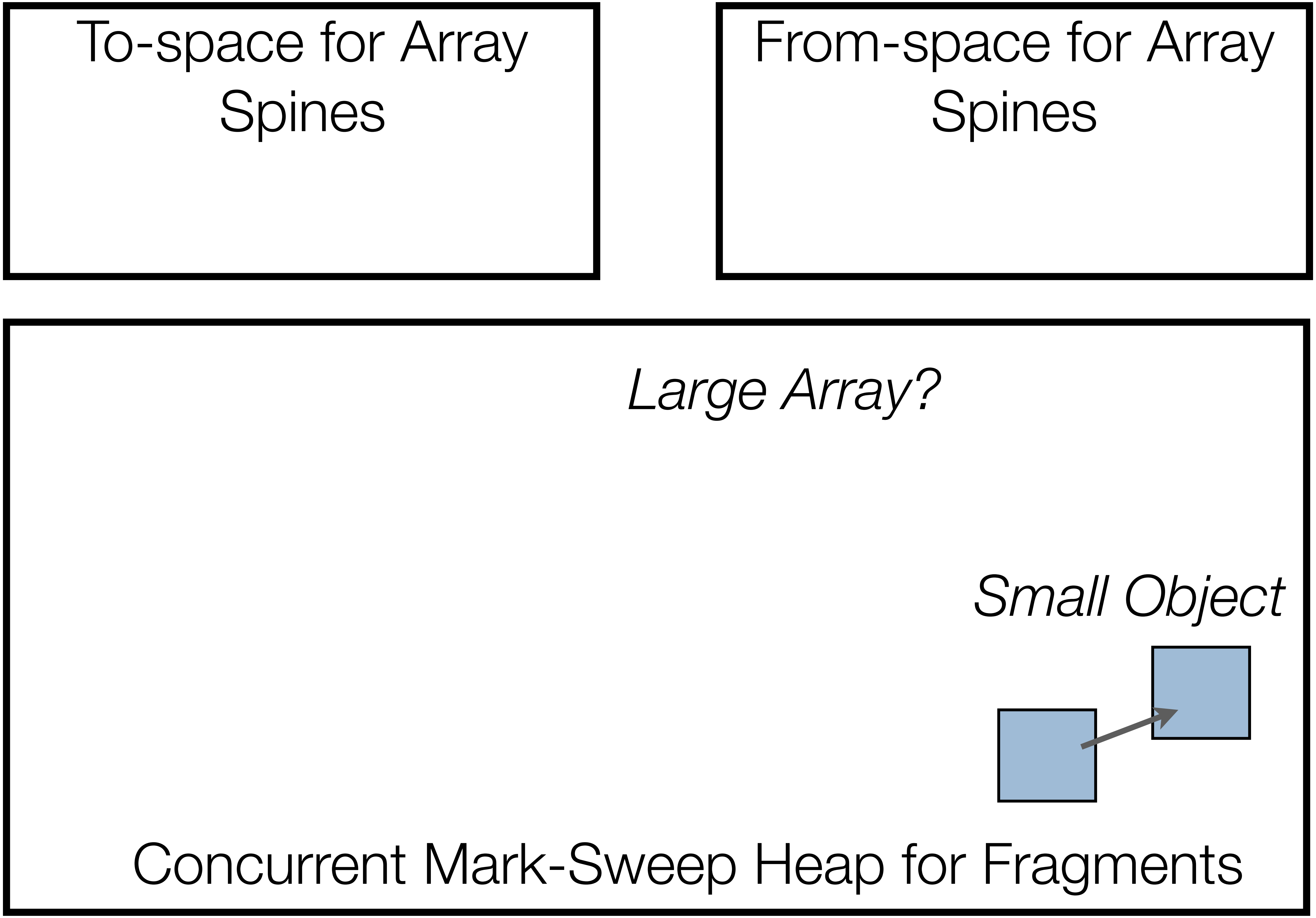
Concurrent Replication Heap for Spines



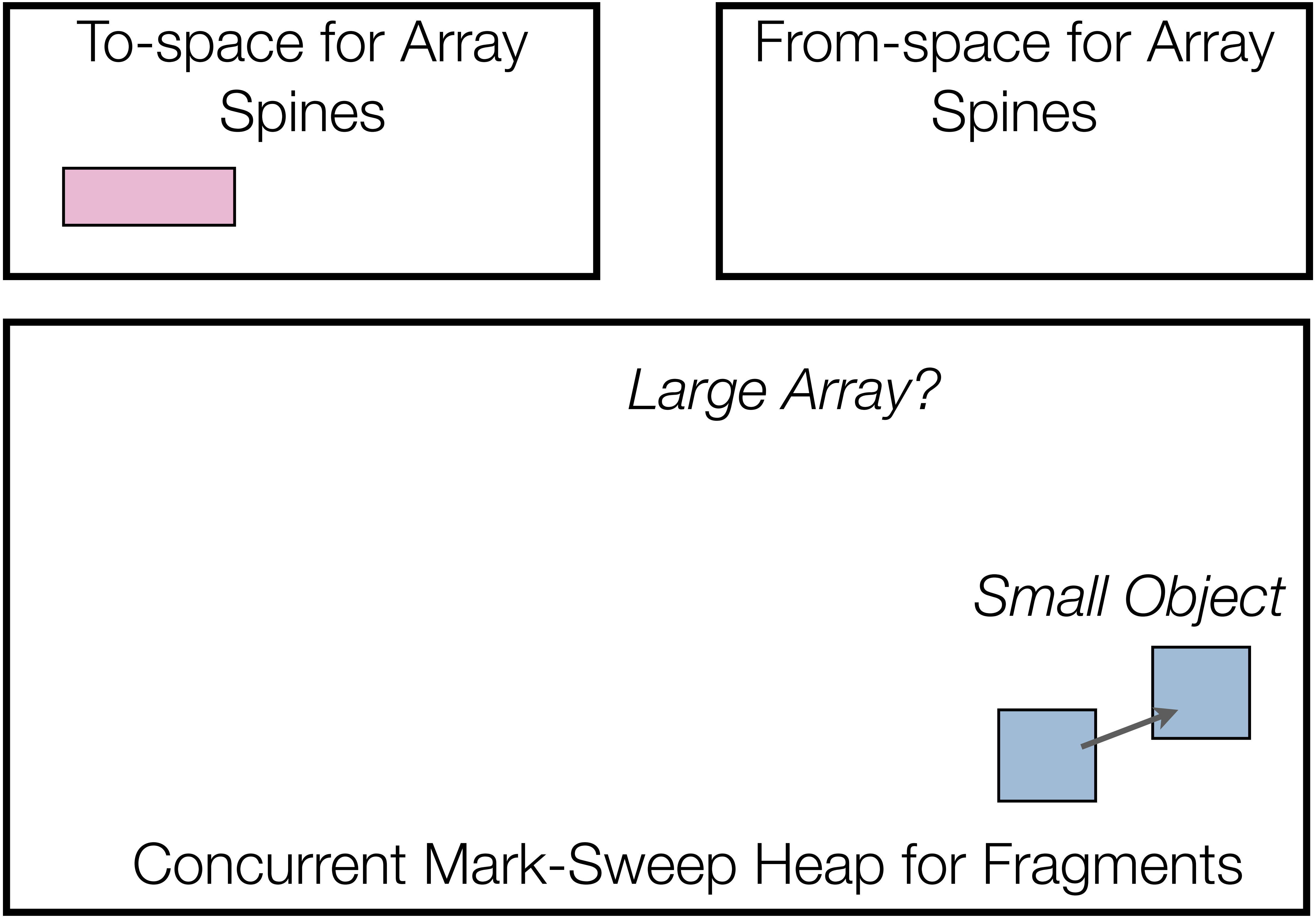
Concurrent Replication Heap for Spines



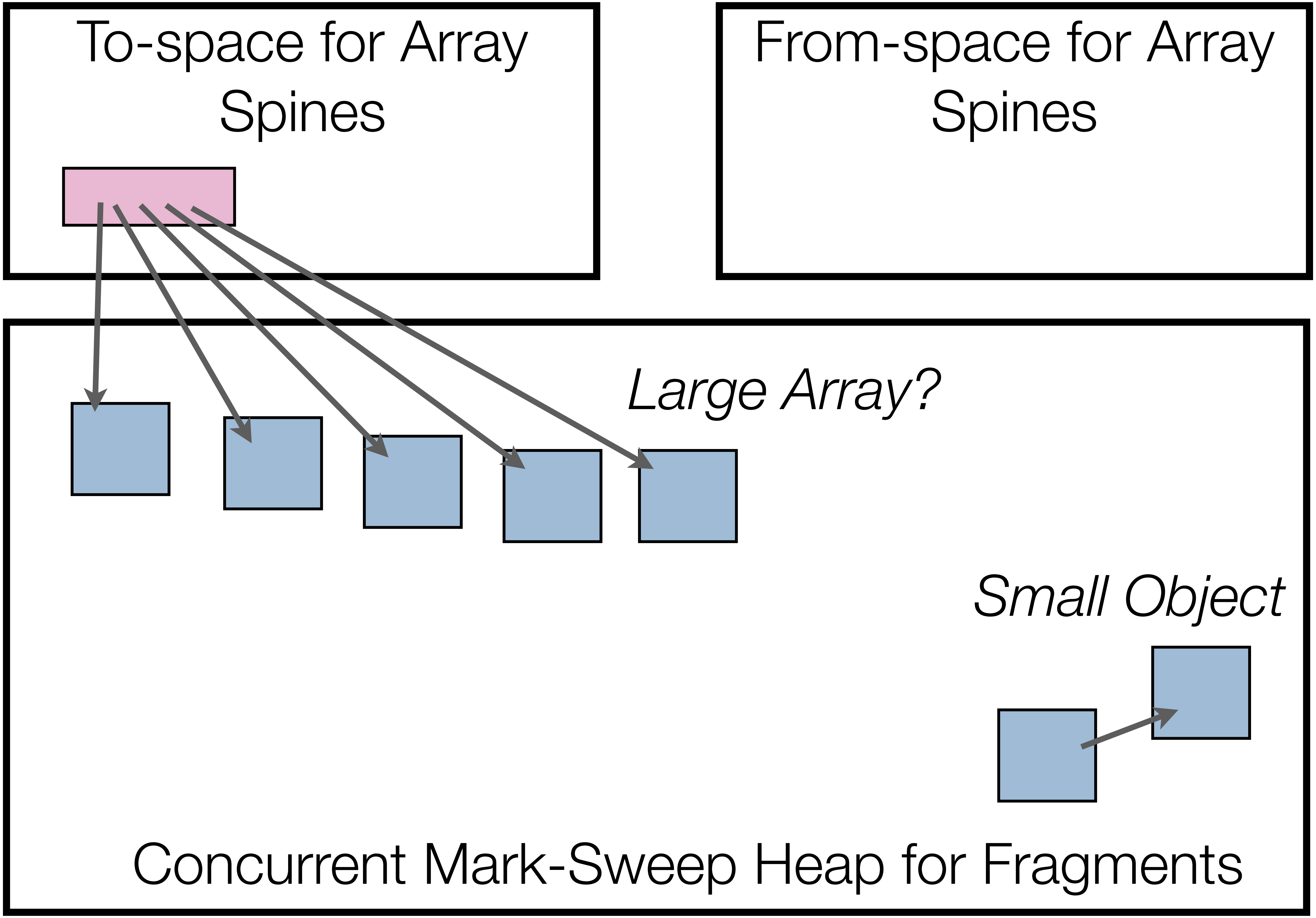
Concurrent Replication Heap for Spines



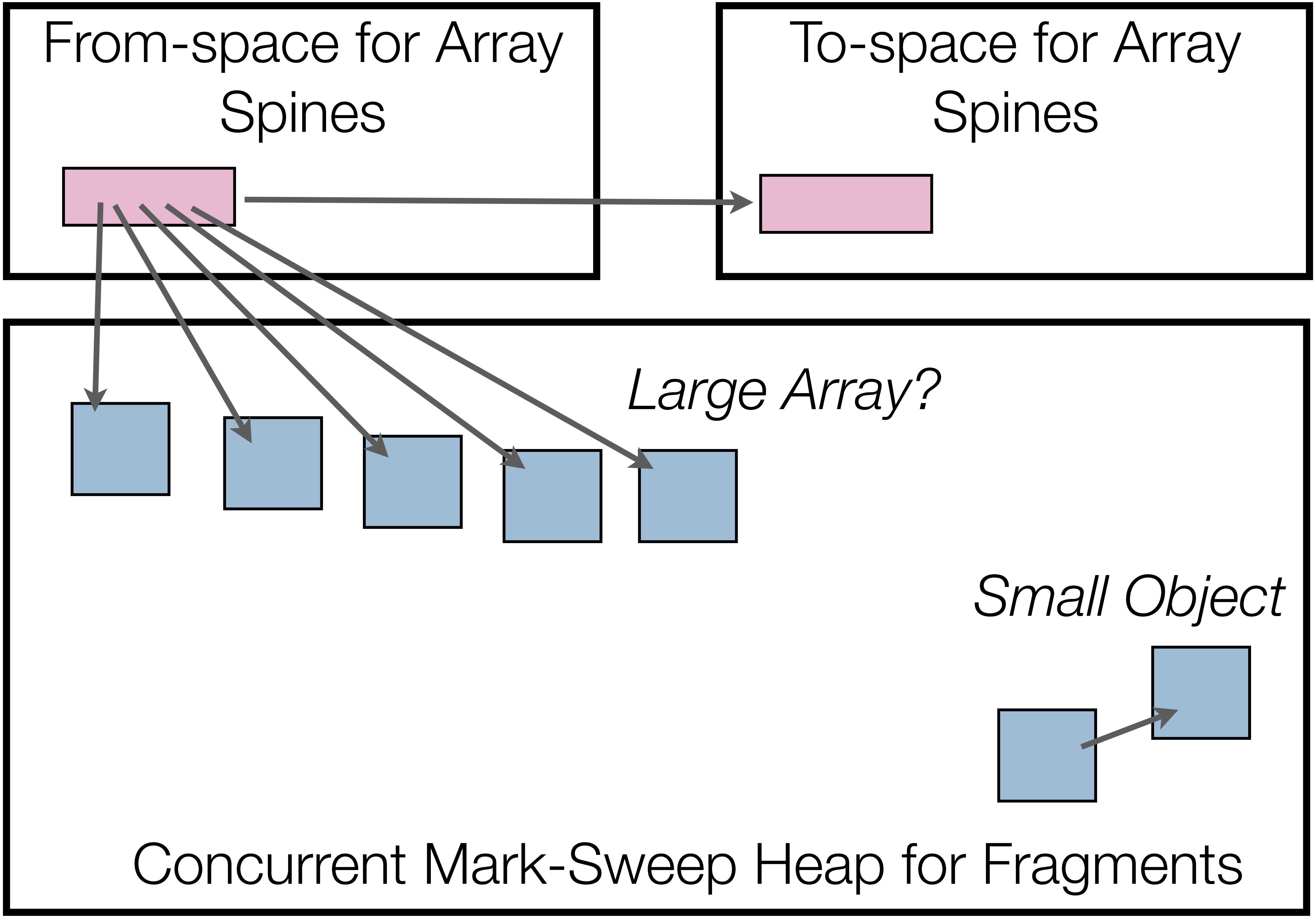
Concurrent Replication Heap for Spines



Concurrent Replication Heap for Spines



Concurrent Replication Heap for Spines



related work
- *or* -
how to make a complete RTGC

Cheng & Blelloch '01

related work

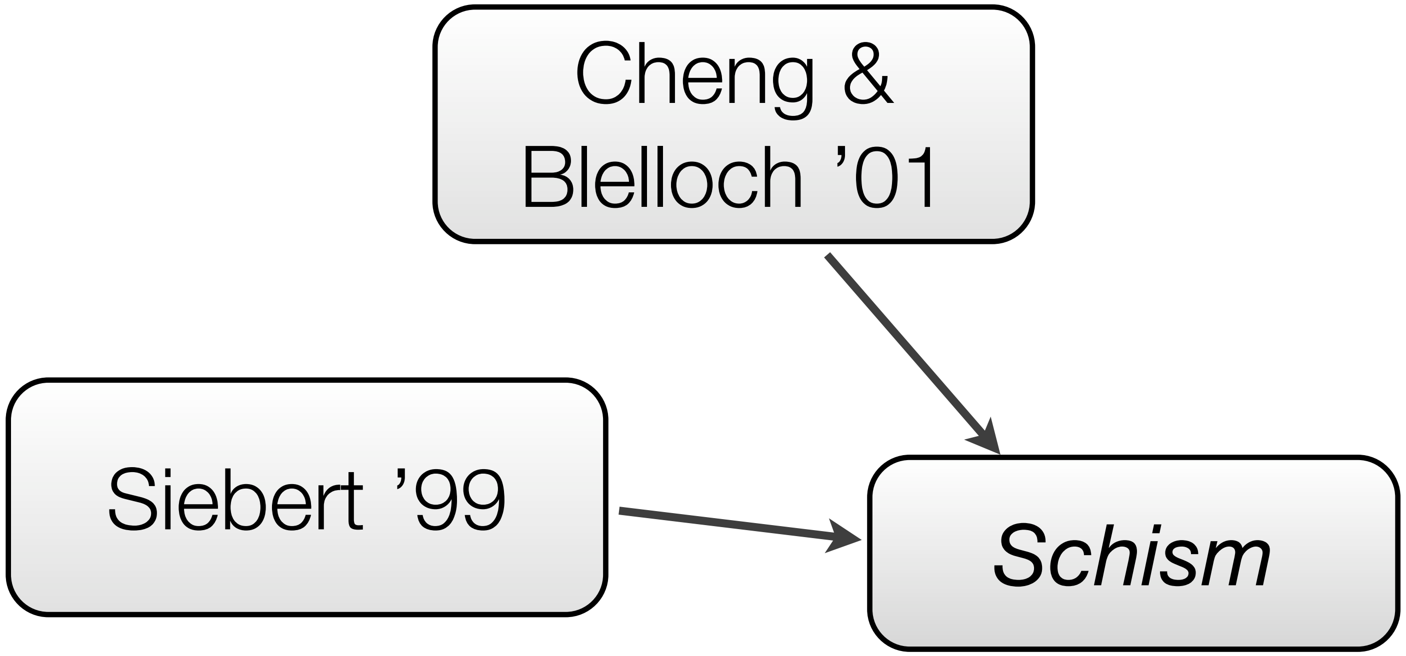
- or -

how to make a complete RTGC

Cheng & Blelloch ’01

Siebert ’99

related work
- or -
how to make a complete RTGC



related work

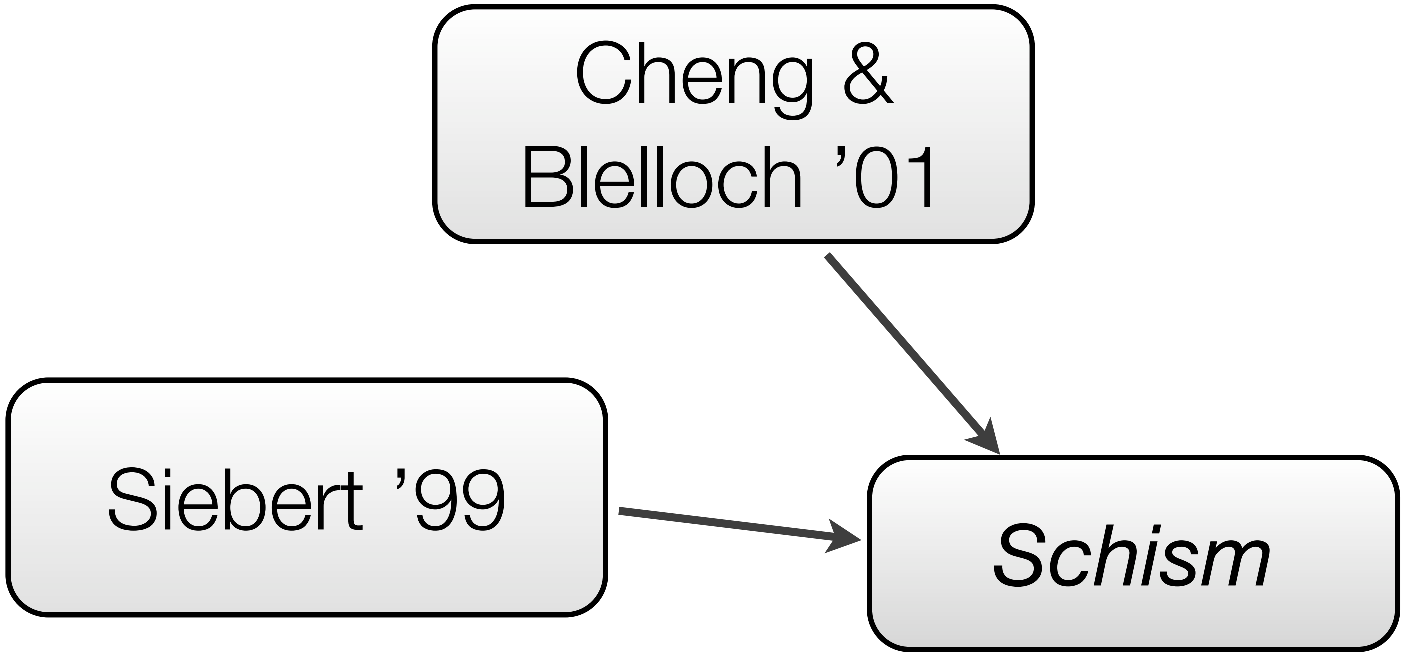
- *or* -

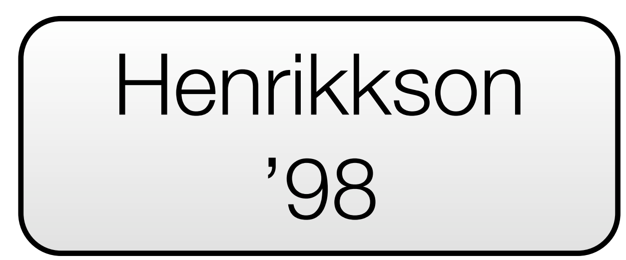
how to make a complete RTGC

related work

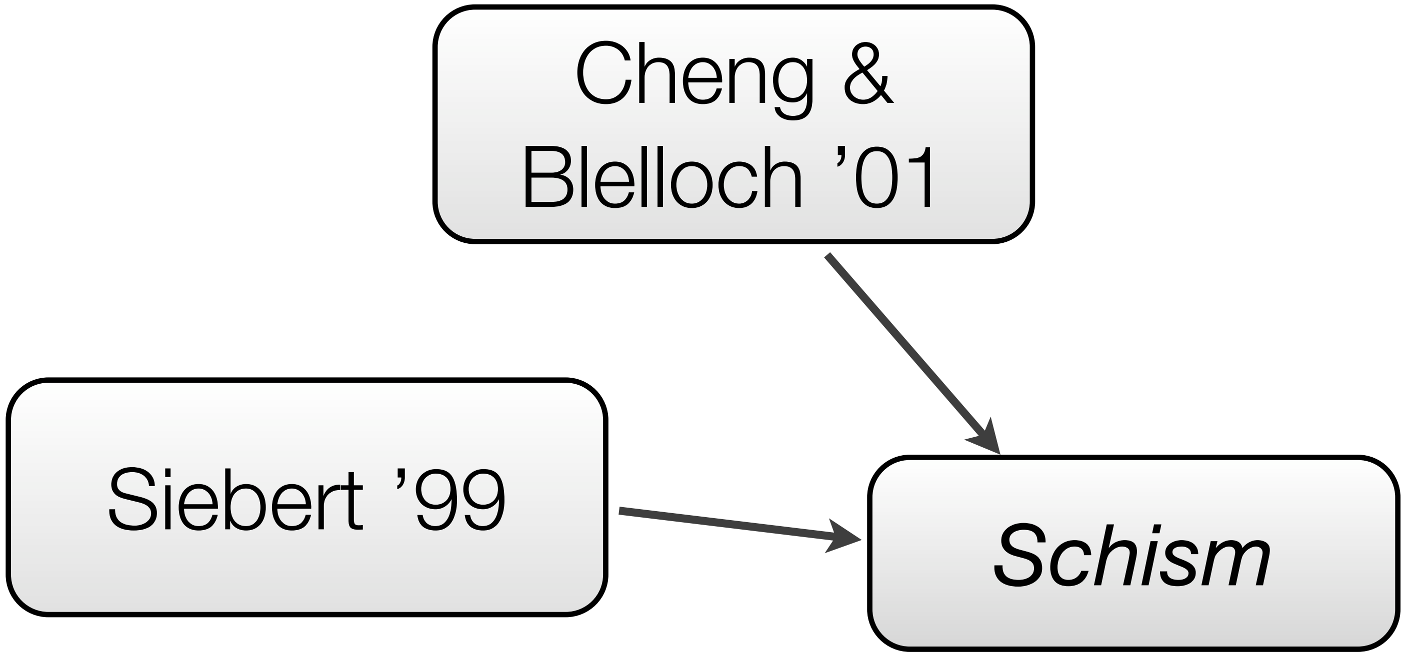
- *or* -

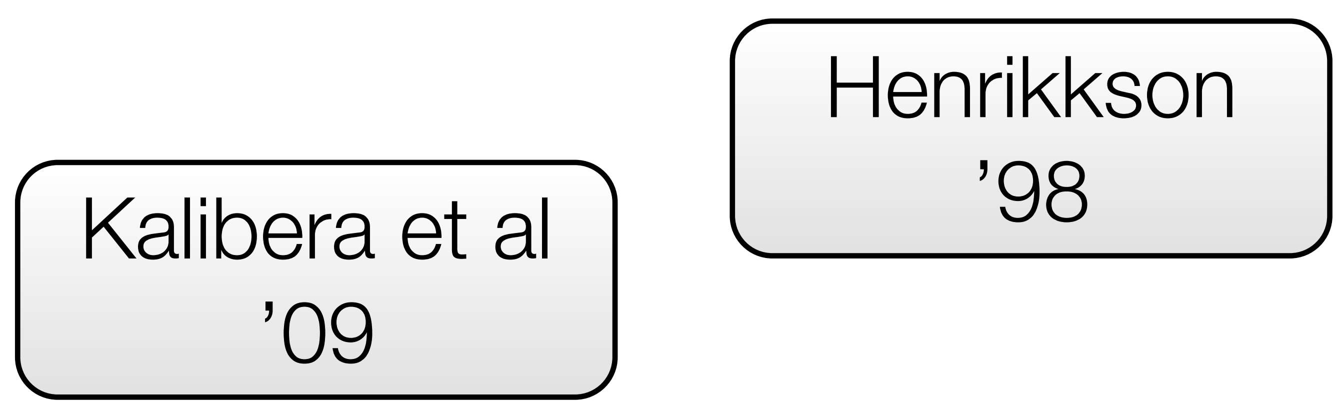
how to make a complete RTGC





related work
- or -
how to make a complete RTGC





related work
- *or* -
how to make a complete RTGC

Cheng & Blelloch ’01

Siebert ’99

Schism

Blackburn & McKinley ’08

Henrikkson ’98

Kalibera et al ’09

related work
- or -
how to make a complete RTGC

Cheng & Blelloch ’01

Siebert ’99

Schism

Blackburn & McKinley ’08

Doligez, Leroy, Gonthier ’93, ’94

Henrikkson ’98

Kalibera et al ’09

related work
- or -
how to make a complete RTGC

Cheng & Blelloch ’01

Siebert ’99

Schism

Blackburn & McKinley ’08

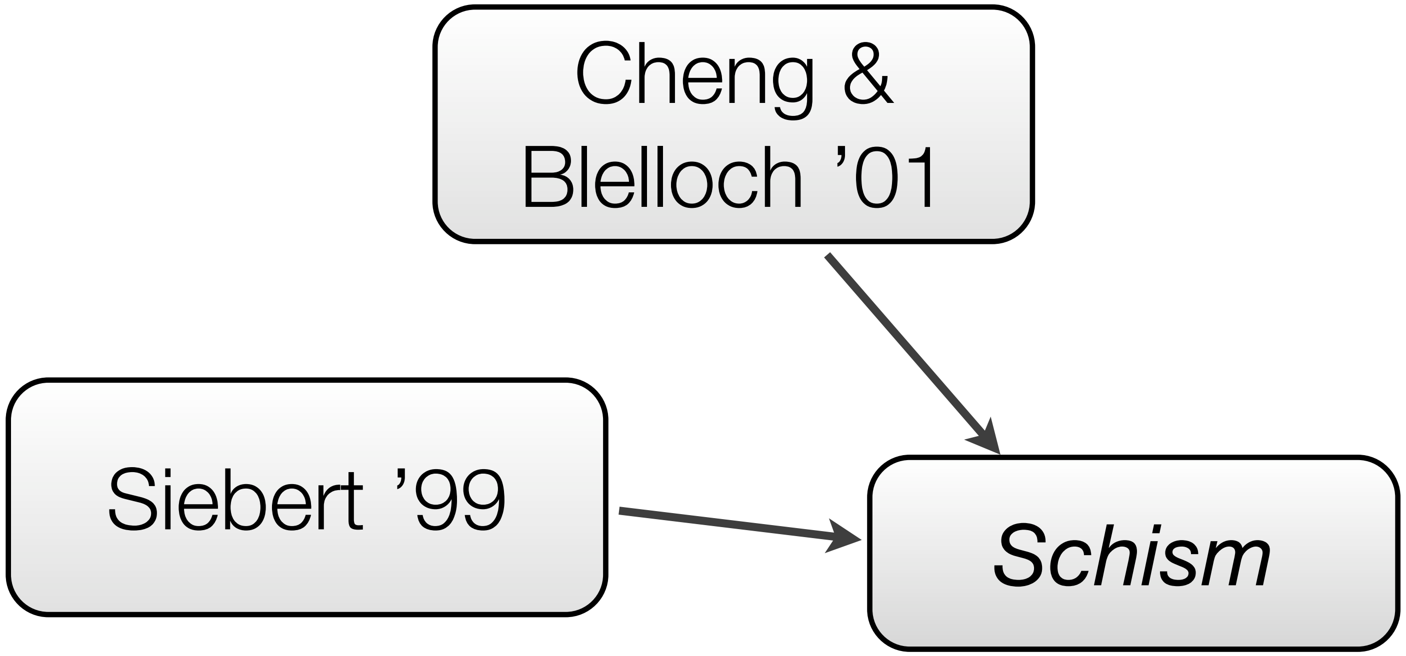
Doligez, Leroy, Gonthier ’93, ’94

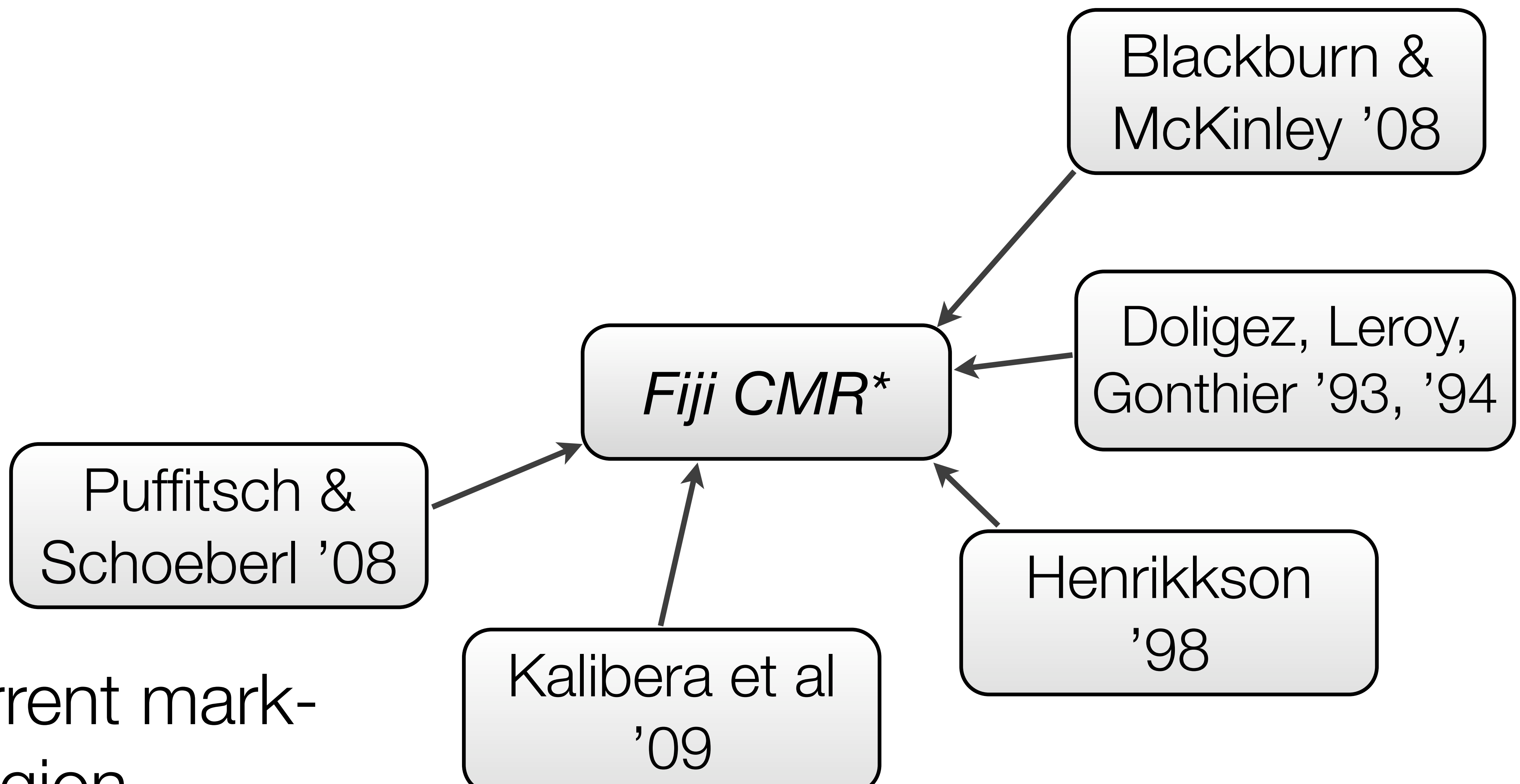
Puffitsch & Schoeberl ’08

Henrikkson ’98

Kalibera et al ’09

related work
- *or* -
how to make a complete RTGC



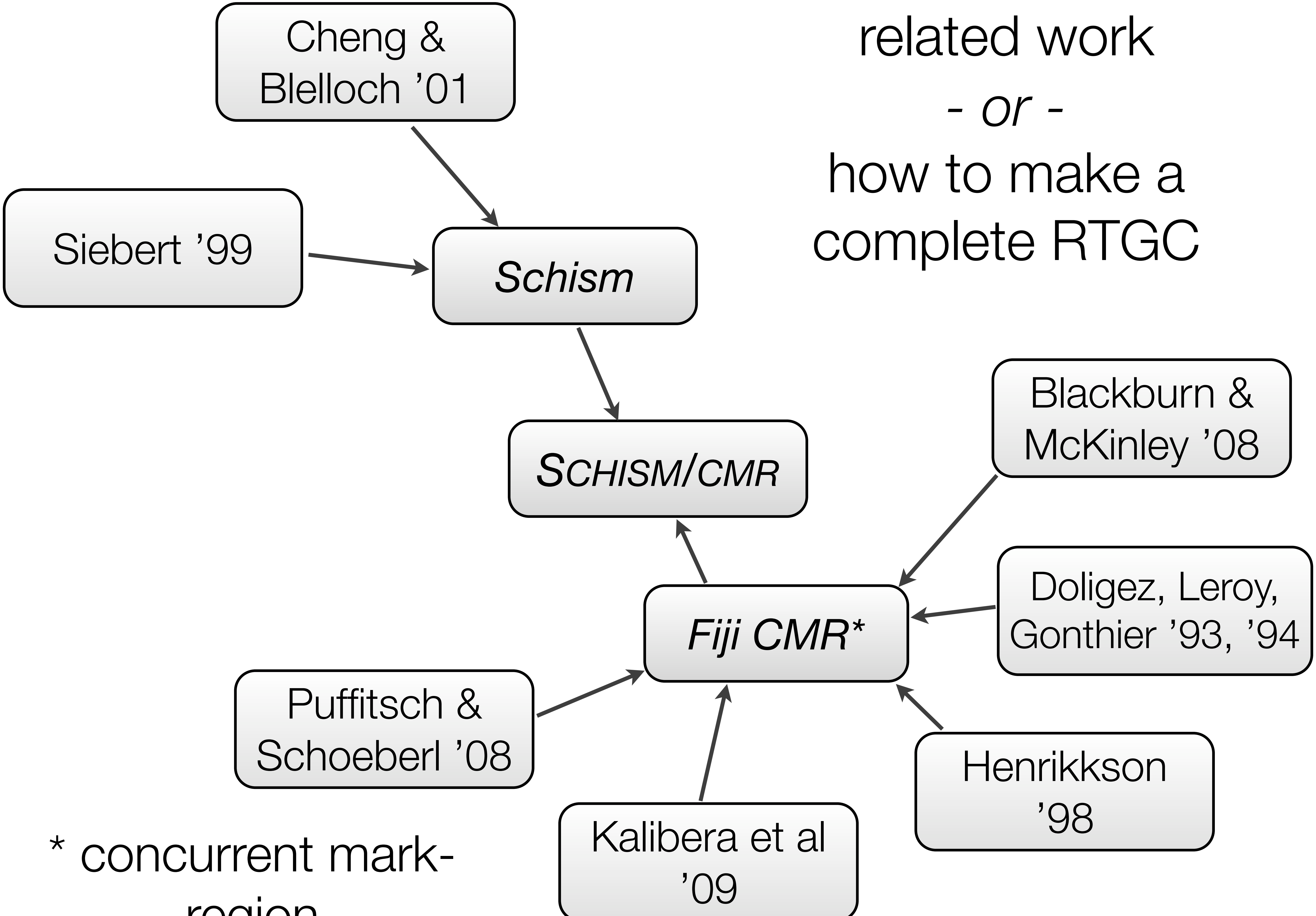


* concurrent mark-region

related work

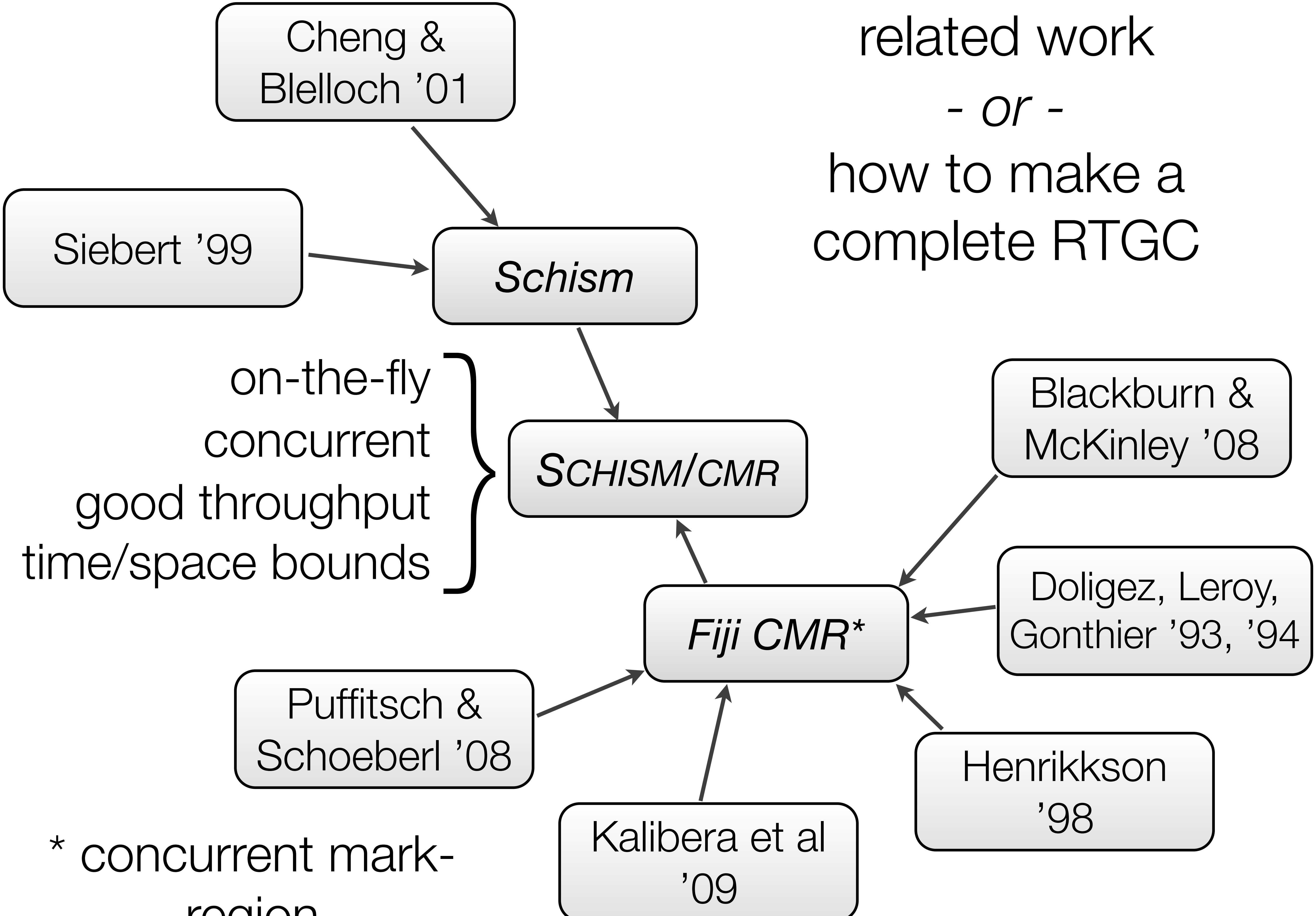
- *or* -

how to make a complete RTGC



* concurrent mark-region

related work
- or -
how to make a complete RTGC



* concurrent mark-region

Tunable throughput-predictability trade-off.

Tunable throughput-predictability trade-off.

- **Schism A**: *completely deterministic*:
  - arrays allocated fragmented
- **Schism C**: optimize throughput:
  - allocate contiguously if possible
- **Schism CW**: simulate worst-case execution of Schism C:
  - poison all fast-paths (array accesses, write barriers, allocations)

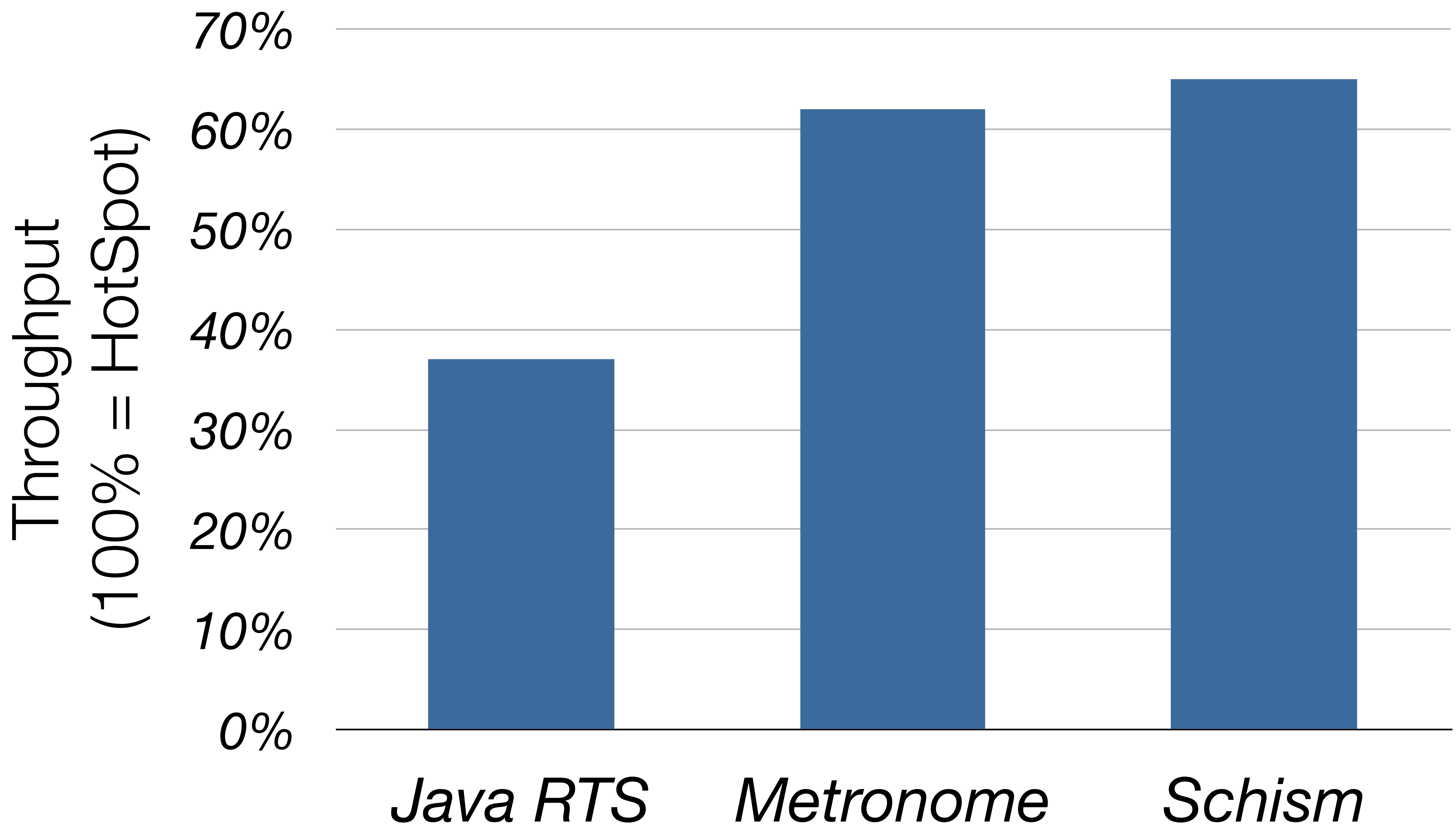
(very short) Summary of Results

- Goal: as fast as Metronome
- Goal: fragmentation tolerant like Java RTS
- Goal: deterministic

(very short) Summary of Results

- **Goal: as fast as Metronome**
- Goal: fragmentation tolerant like Java RTS
- Goal: deterministic

SPECjvm98 throughput summary



(very short) Summary of Results

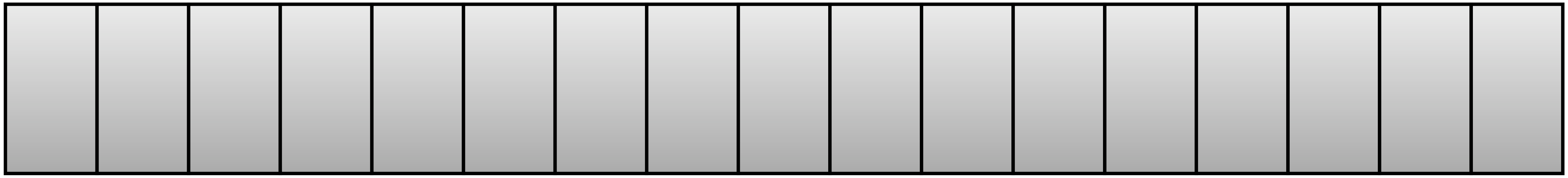
- Goal: as fast as Metronome
- **Goal: fragmentation tolerant like Java RTS**
- Goal: deterministic

(very short) Summary of Results

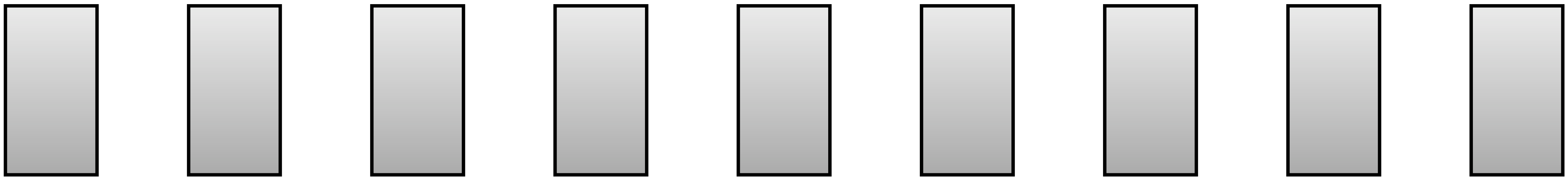
- Goal: as fast as Metronome ✓
- **Goal: fragmentation tolerant like Java RTS**
- Goal: deterministic

Fragger Results

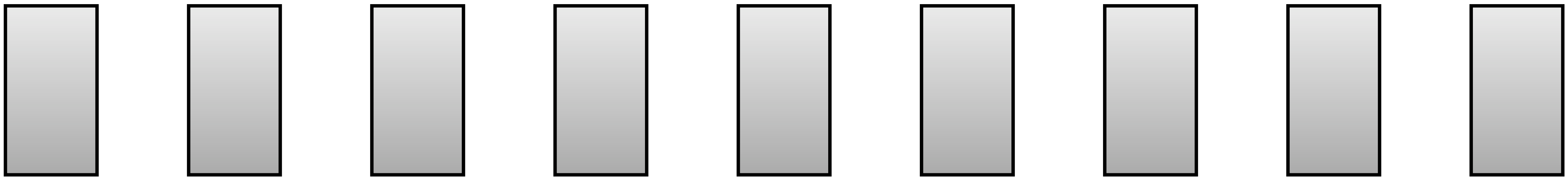
Fragger Results



Fragger Results



Fragger Results



- Amount of free memory successfully allocated under fragmentation:
  - *HotSpot*: ~**100%**
  - *Java RTS*: ~**80%**
  - *Metronome*: ~**1%**, unless using >10KB objects
  - *Schism*: ~**100%** (all objects)

(very short) Summary of Results

- Goal: as fast as Metronome ✓
- Goal: fragmentation tolerant like Java RTS
- **Goal: deterministic**

(very short) Summary of Results

- Goal: as fast as Metronome ✓
- Goal: fragmentation tolerant like Java RTS ✓
- **Goal: deterministic**

Schism predictability:
RTEMS* on 40MHz LEON3

Schism predictability:
RTEMS* on 40MHz LEON3

* Real Time Executive for Missile Systems

Schism predictability:
RTEMS* on 40MHz LEON3

The OS/hardware platform used for NASA & ESA space missions.

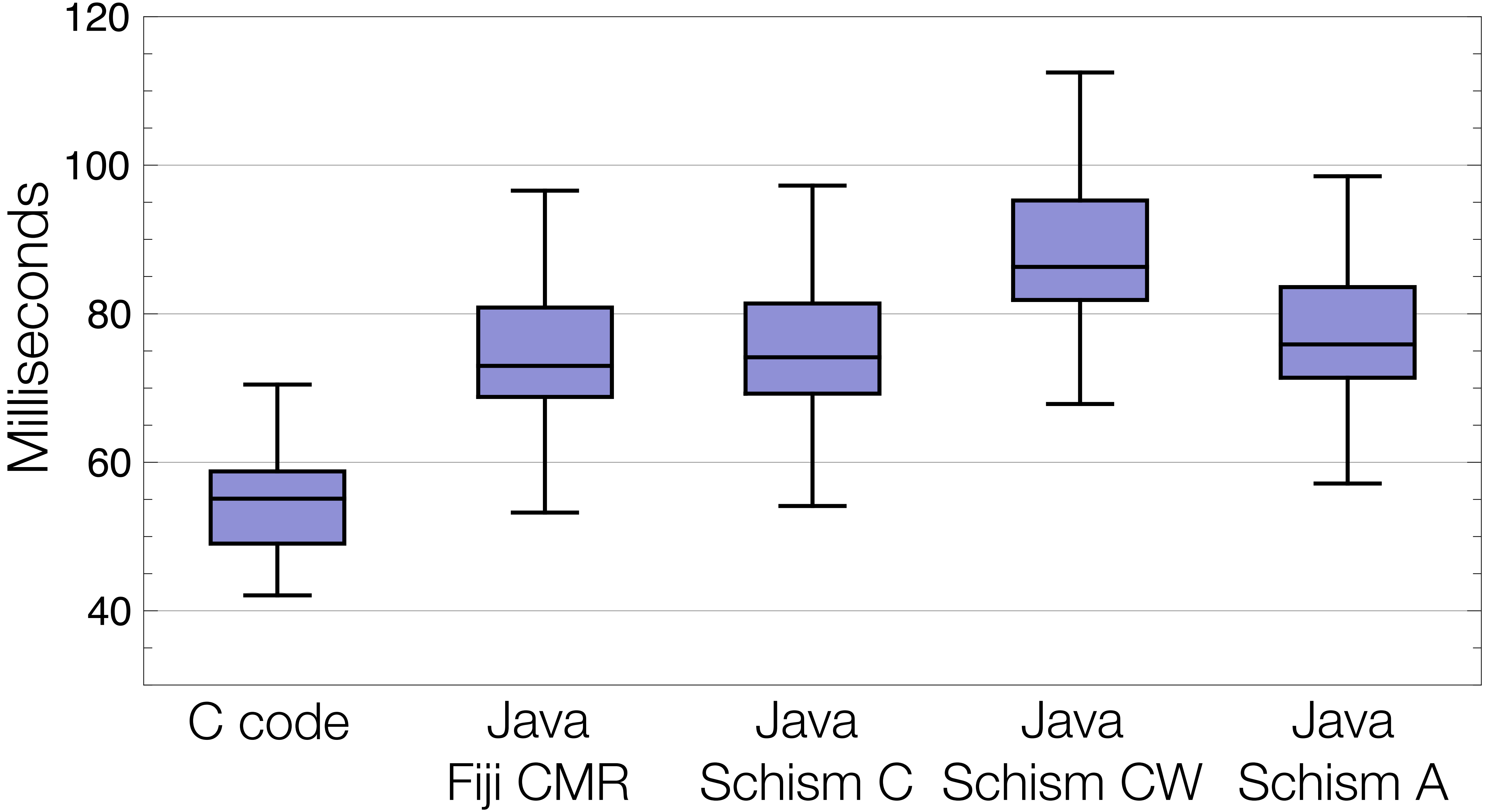
* Real Time Executive for Missile Systems

Performance baseline: C code.

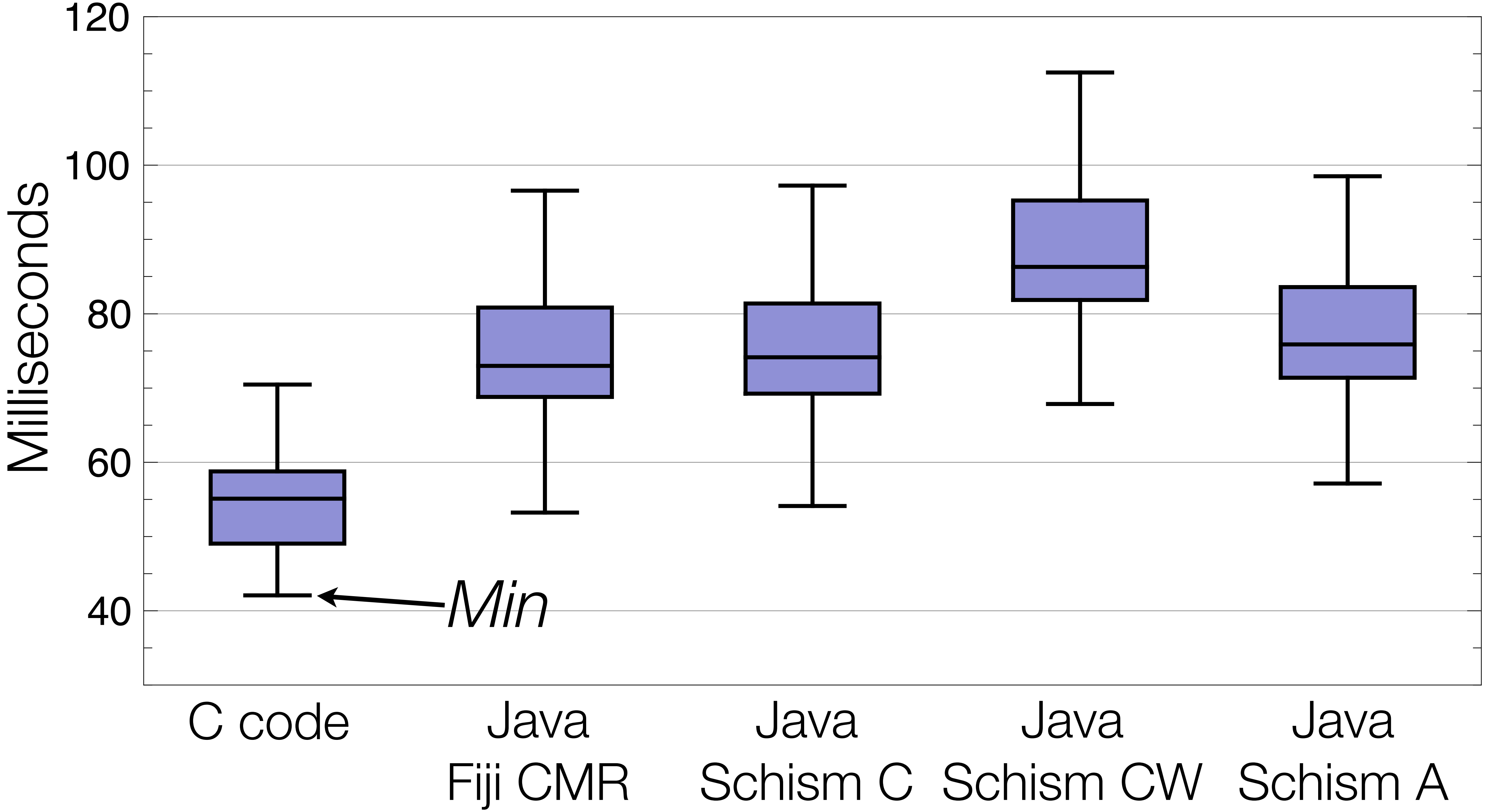
Performance baseline: C code.

Using both C and Java implementations of the ***CDx*** *real-time air traffic collision detection benchmark [Kalibera et al '09].*

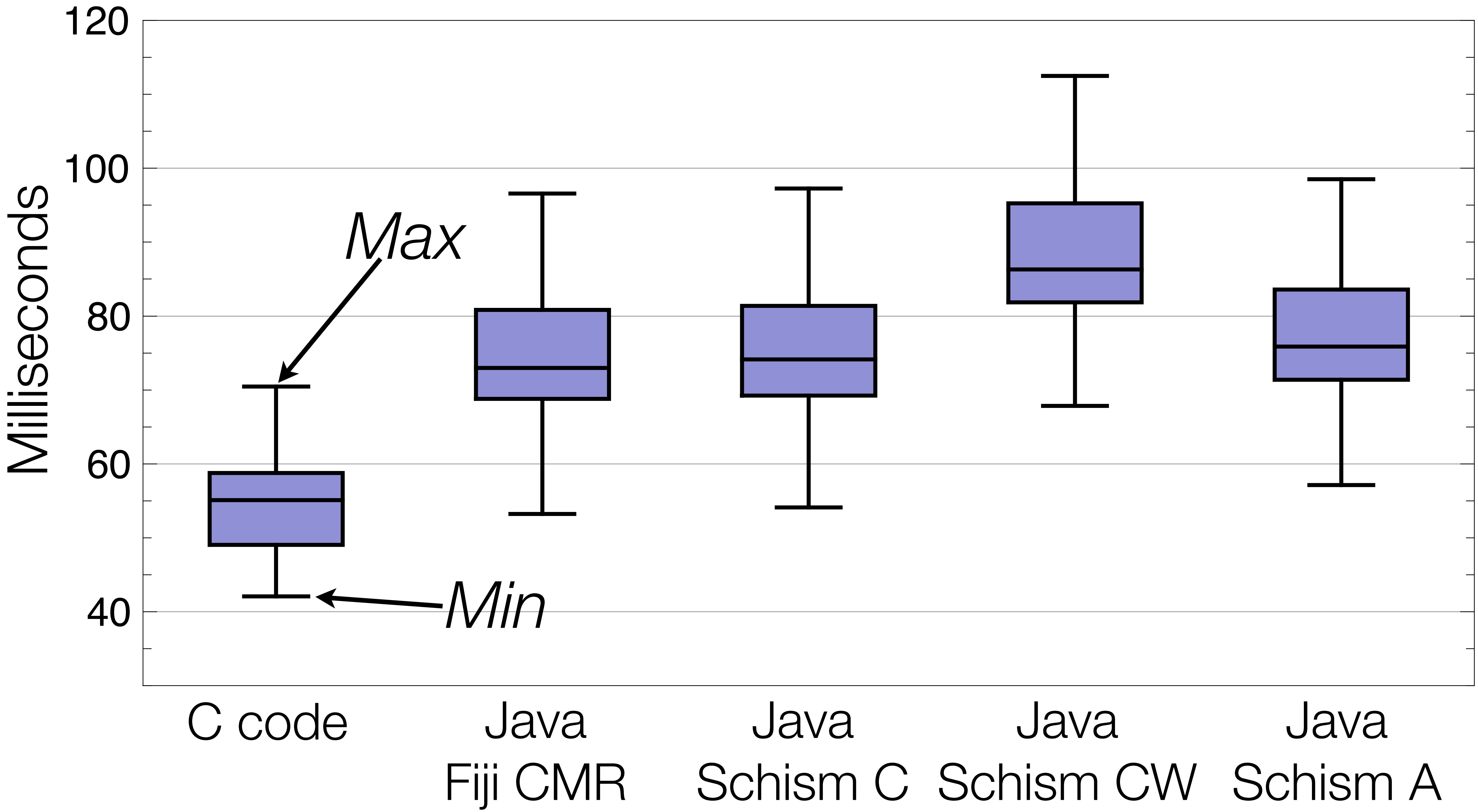
Java (CMR, Schism) versus C on **CDx** real-time benchmark



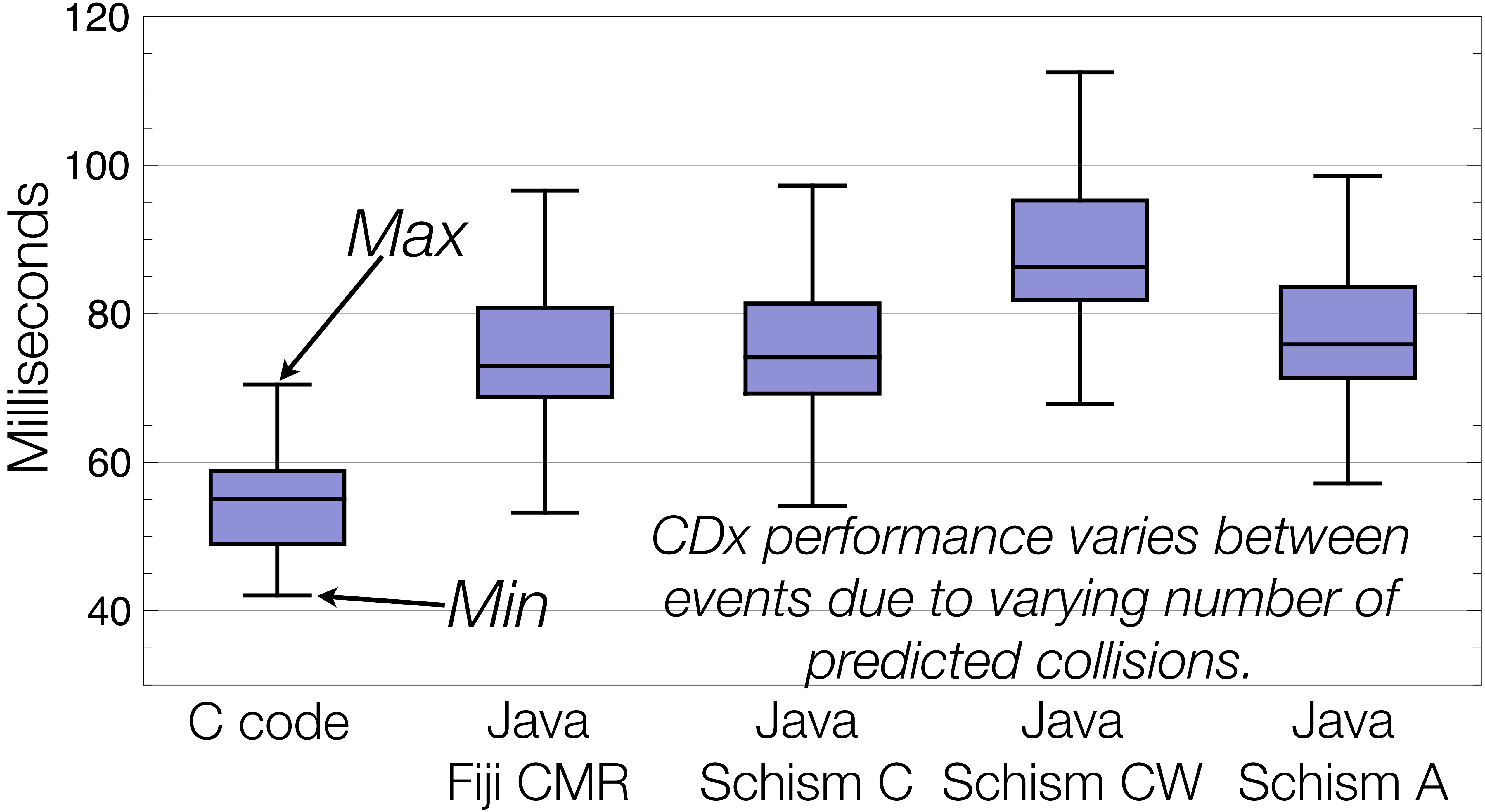
Java (CMR, Schism) versus C on **CDx** real-time benchmark



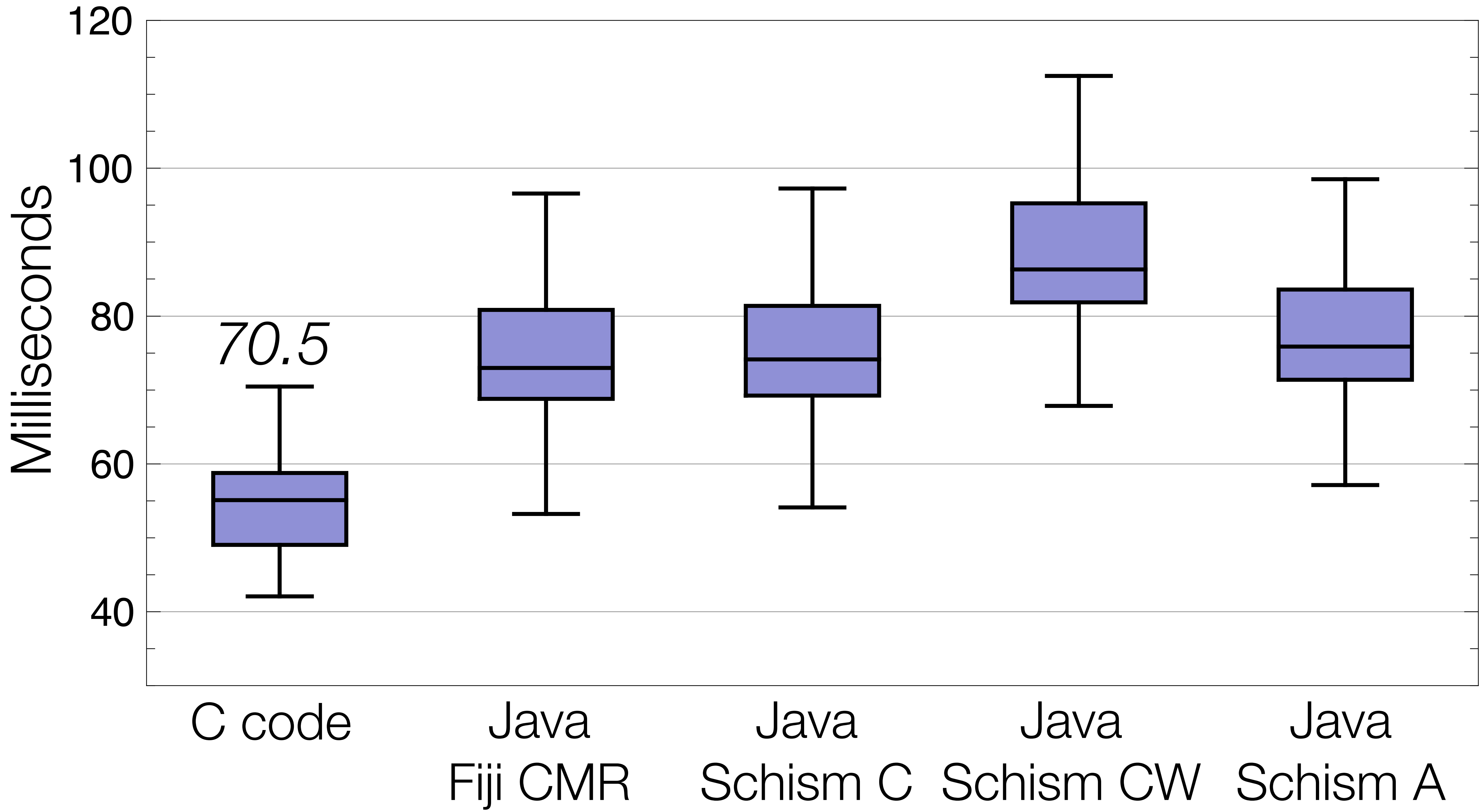
Java (CMR, Schism) versus C on **CDx** real-time benchmark



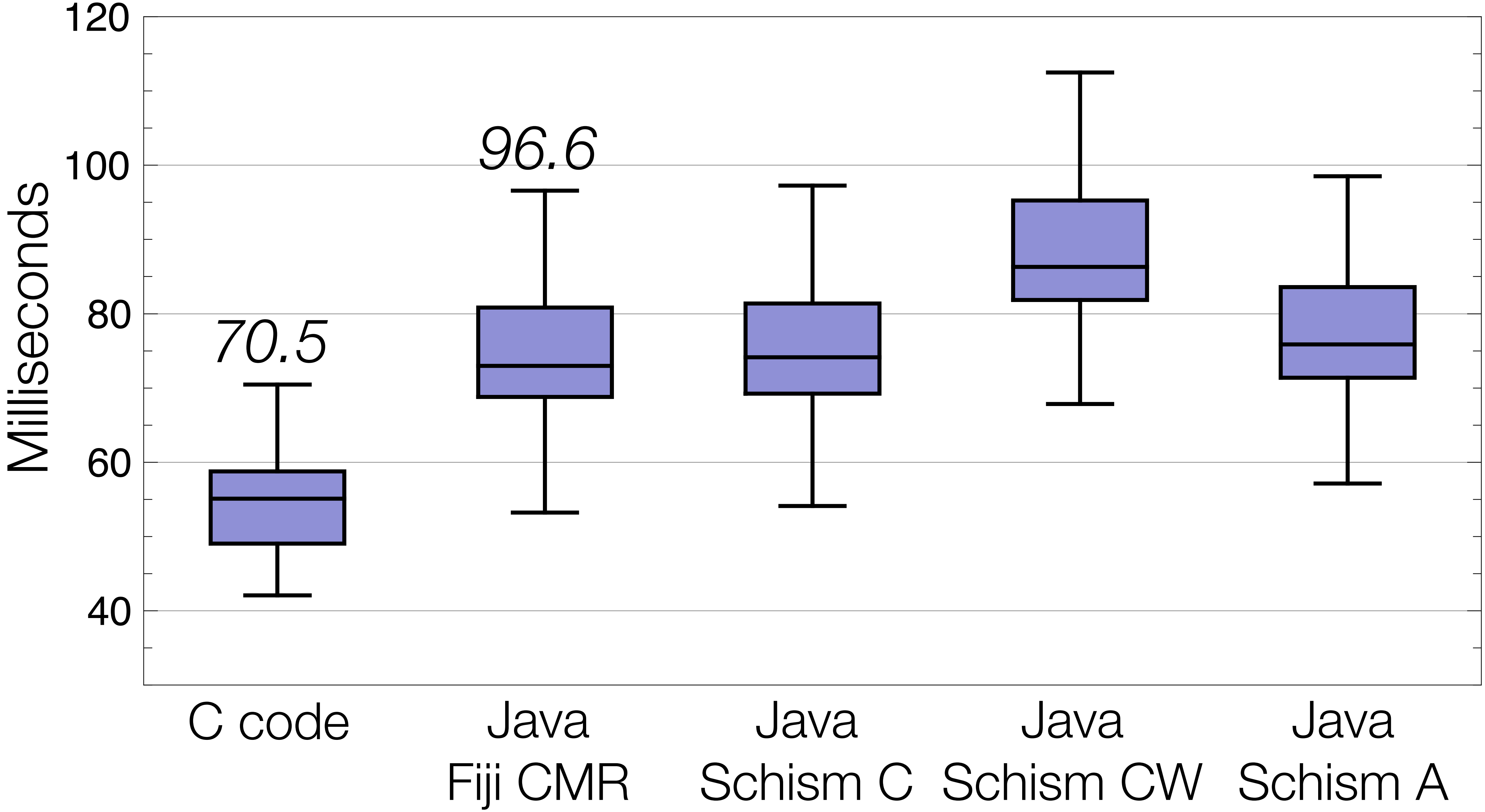
Java (CMR, Schism) versus C on **CDx** real-time benchmark



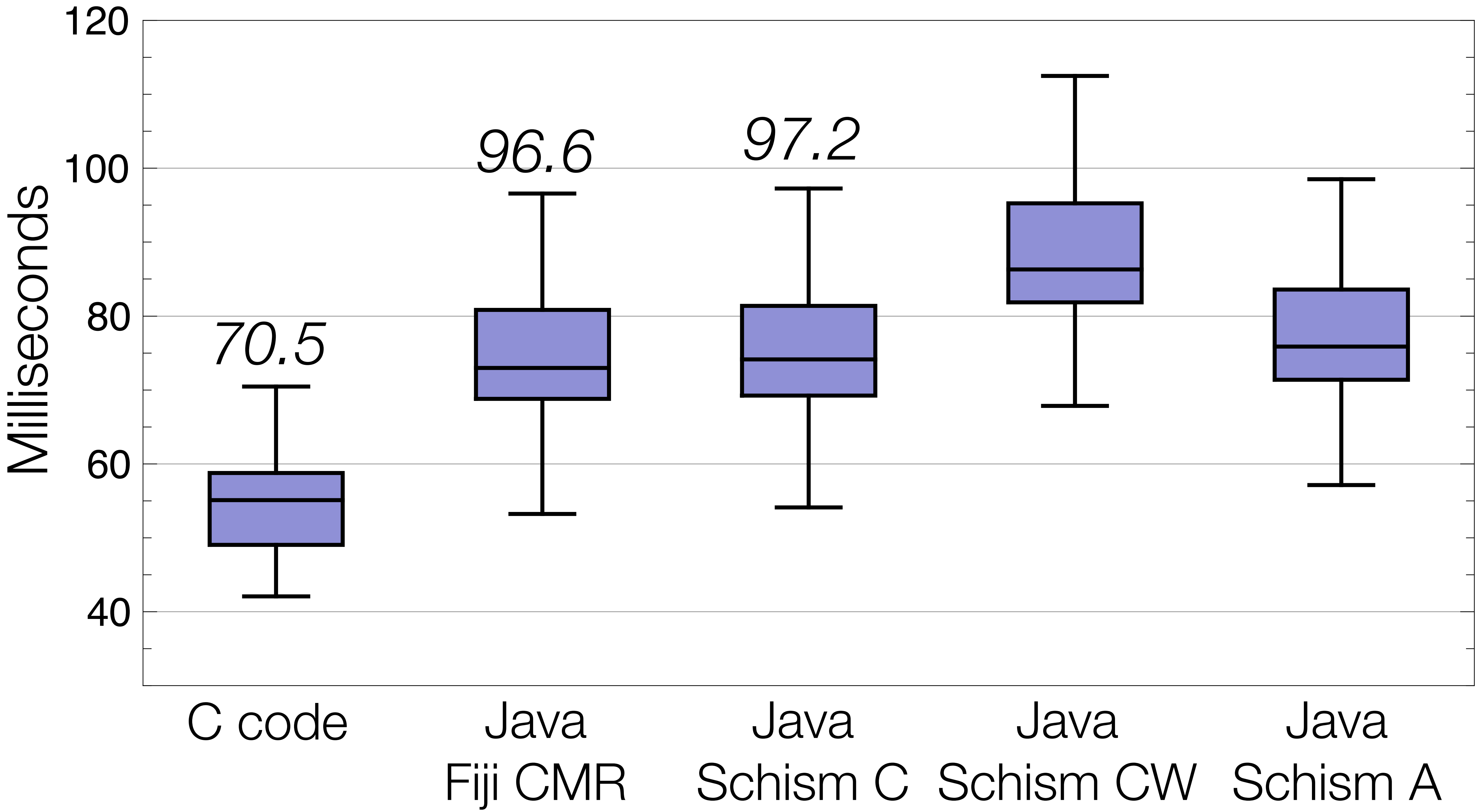
Java (CMR, Schism) versus C on **CDx** real-time benchmark



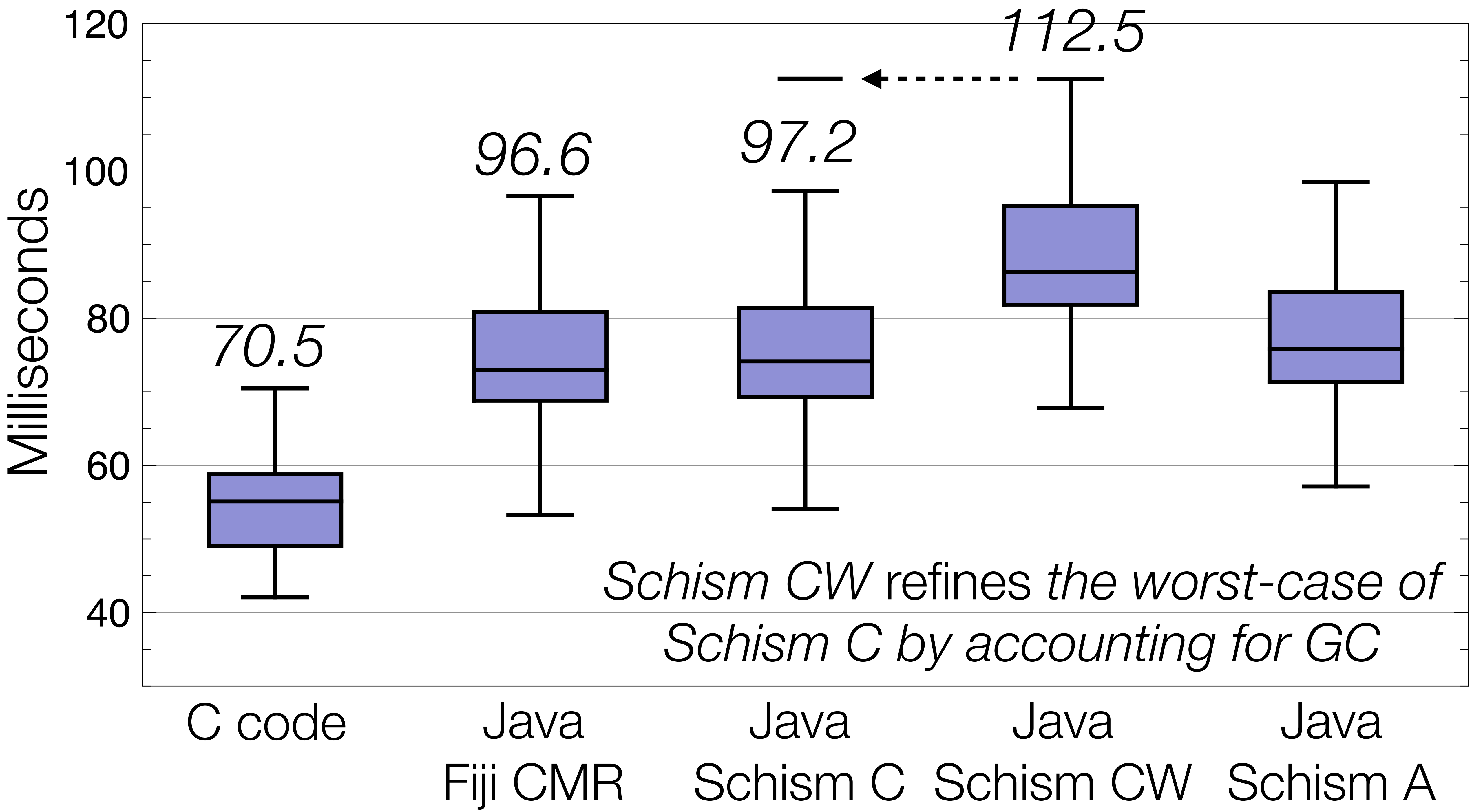
Java (CMR, Schism) versus C on **CDx** real-time benchmark



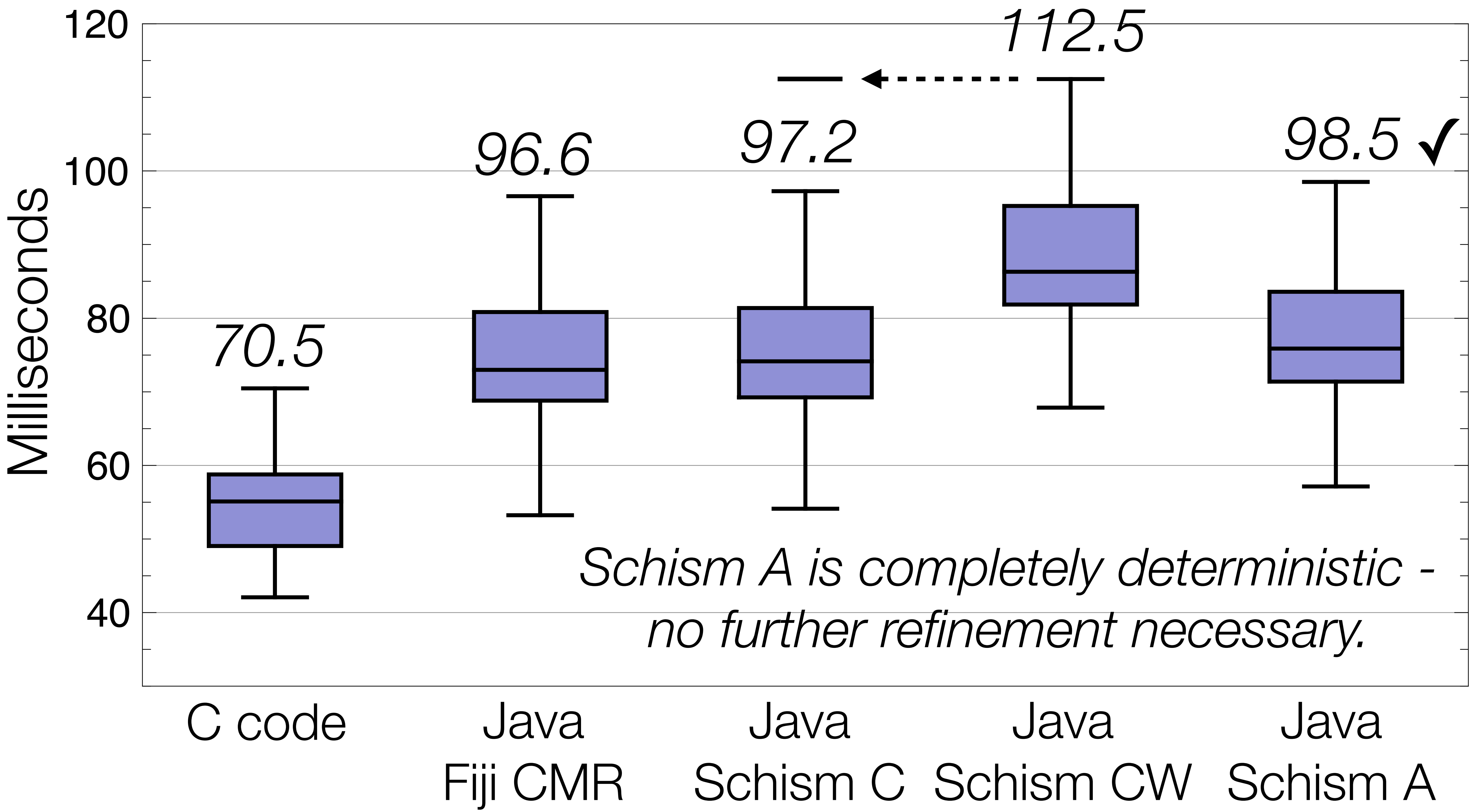
Java (CMR, Schism) versus C on **CDx** real-time benchmark



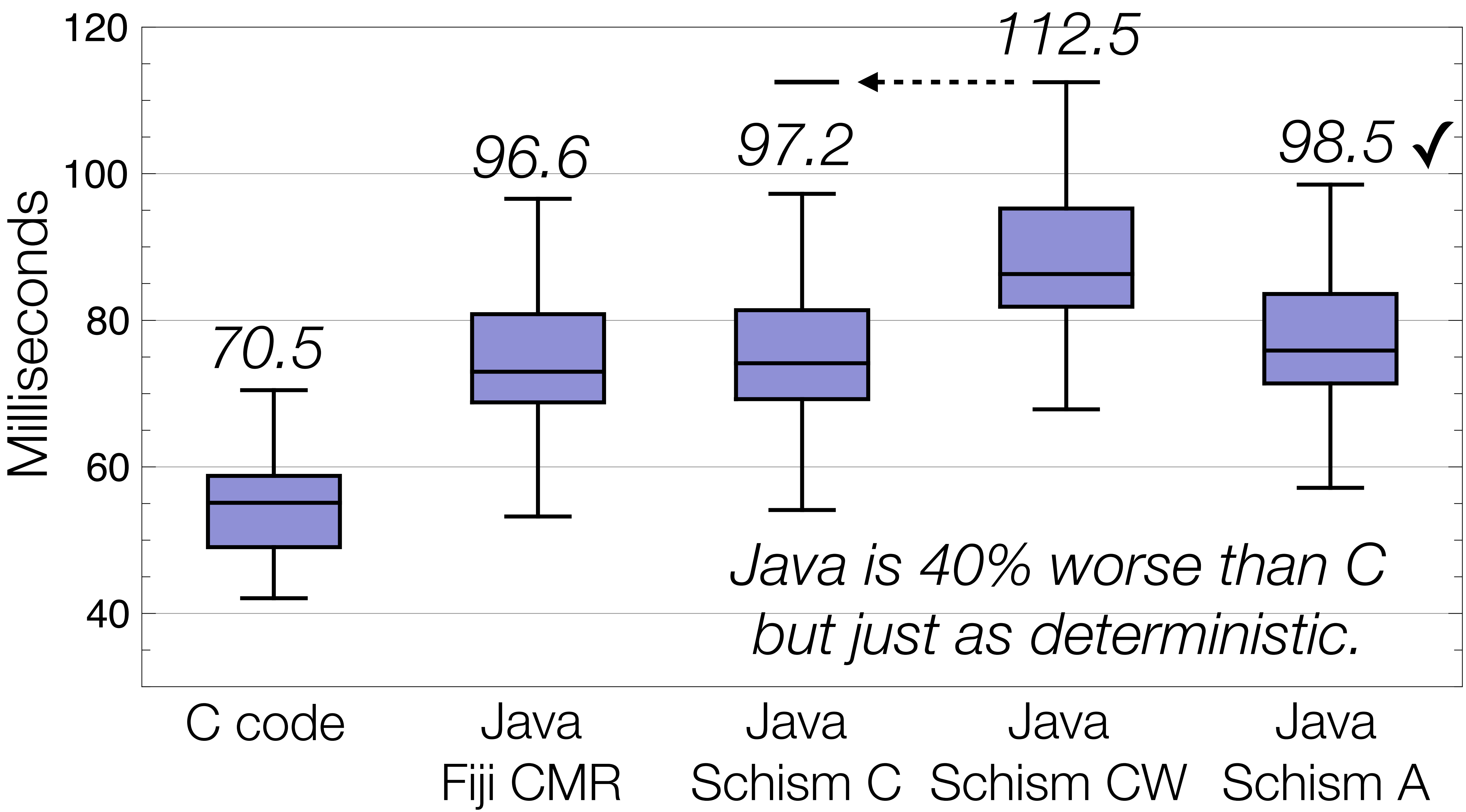
Java (CMR, Schism) versus C on **CDx** real-time benchmark



Java (CMR, Schism) versus C on **CDx** real-time benchmark

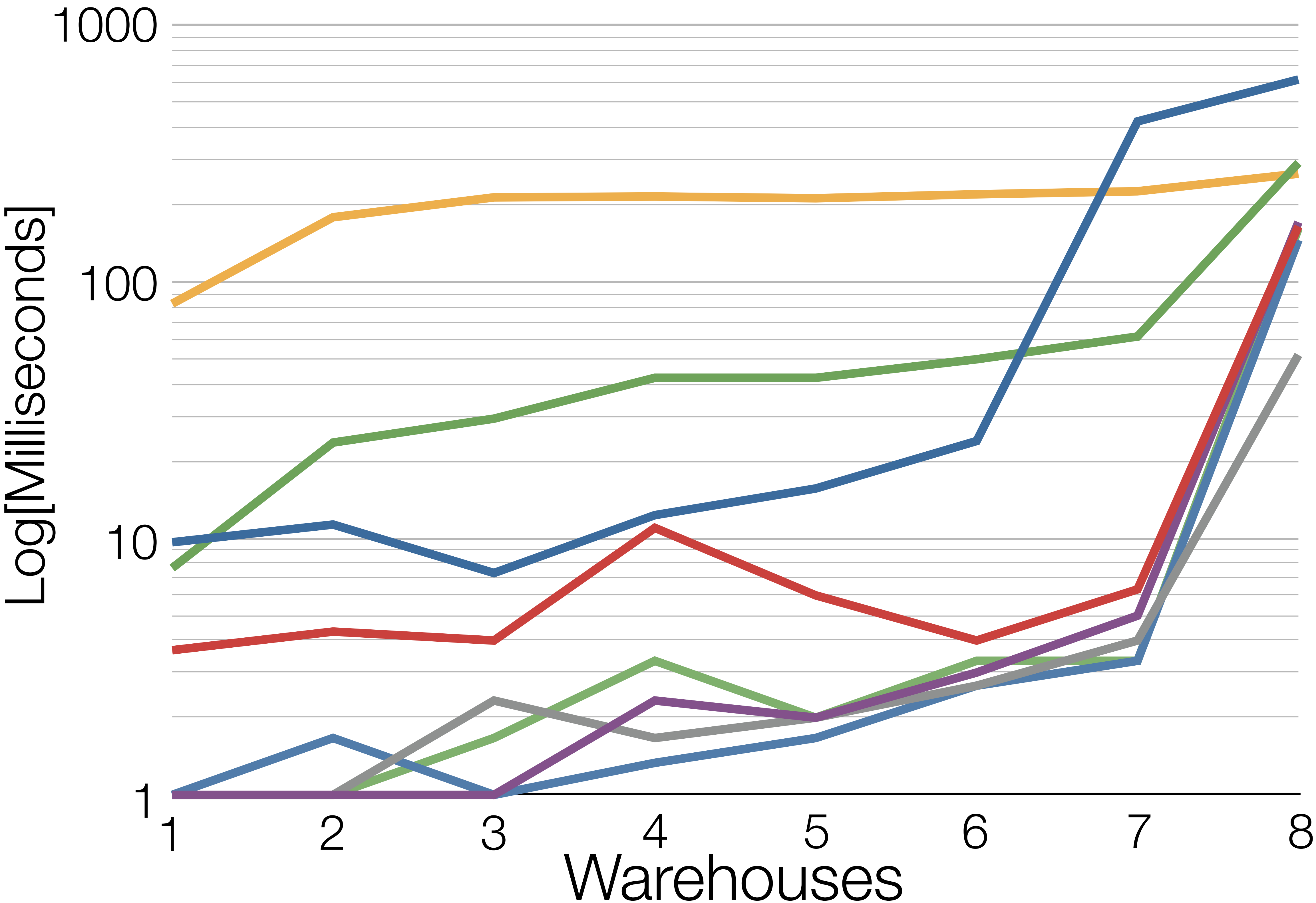


Java (CMR, Schism) versus C on **CDx** real-time benchmark

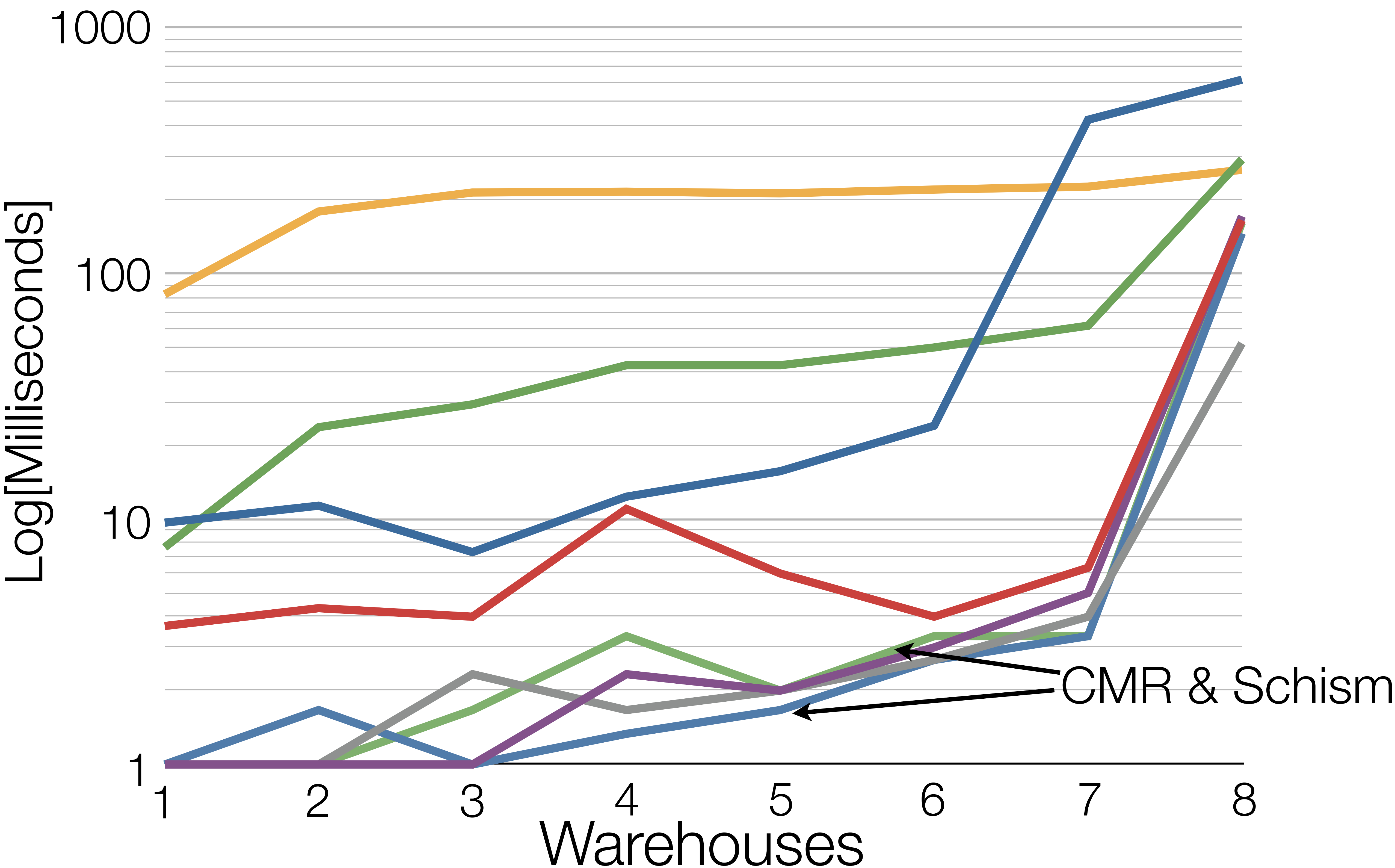


Schism Predictability:
SPECjbb2000 on Linux Xeon

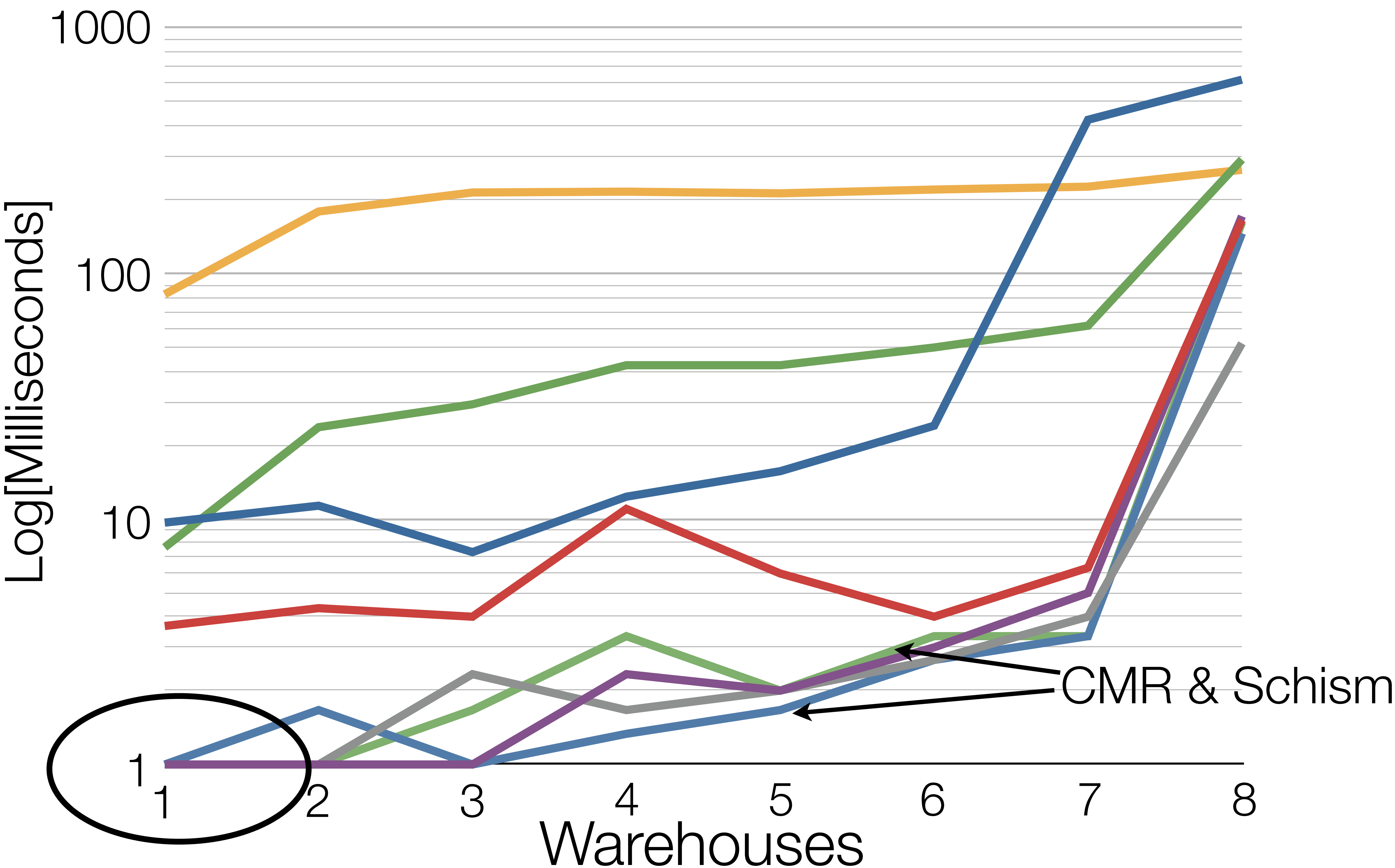
SPECjbb2000 Worst-case Transaction Times



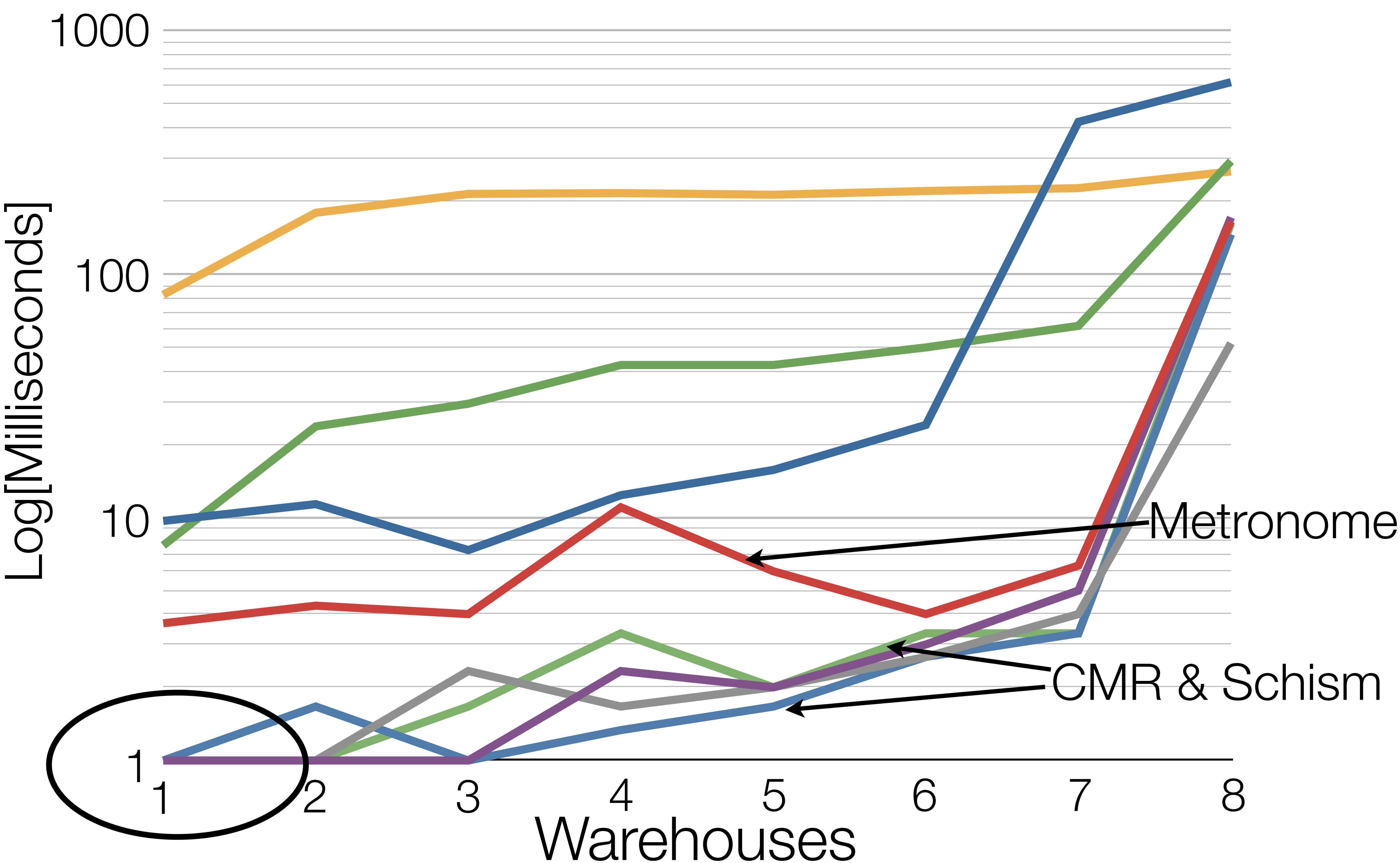
SPECjbb2000 Worst-case Transaction Times



SPECjbb2000 Worst-case Transaction Times



SPECjbb2000 Worst-case Transaction Times



- Additional experiments in the paper:
  - SPECjvm98 in detail
  - Worst-case-time v. memory for CDx on RTEMS/LEON3
  - MMU for CDx on RTEMS/LEON3
  - Detailed fragmentation numbers with Fragger
  - Array access performance under fragmentation
  - Scalability with SPECjbb2000
  - Analytical proof of space bounds
  - Experimental validation of analytical proof of space bounds

Read the paper for the most awesomely epic RTGC evaluation, ever.

Conclusion: A good Real-Time GC...

- executes concurrently with mutator threads
- guarantees progress for heap accesses
  - wait-free (per-thread progress)
- minimizes heap access overhead
  - few instructions
- gives uniformly good throughput
- is space efficient (minimizes external fragmentation)

Conclusion: A good Real-Time GC...

- executes concurrently with mutator threads
- guarantees progress for heap accesses
  - wait-free (per-thread progress)
- minimizes heap access overhead
  - few instructions
- gives uniformly good throughput
- is space efficient (minimizes external fragmentation)