An Emprical Evaluation *of* Memory Management Alternatives *for* Real Time Java

Filip Pizlo, Jan Vitek
Purdue University
West Lafayette, IN, USA

RTSS

Dec 2006

Motivation

- Real Time Java programmers are forced to choose between two memory management styles:
  - ***Scoped Memory***
  - ***Real Time Garbage Collection***
- To date, no direct performance comparison exists.

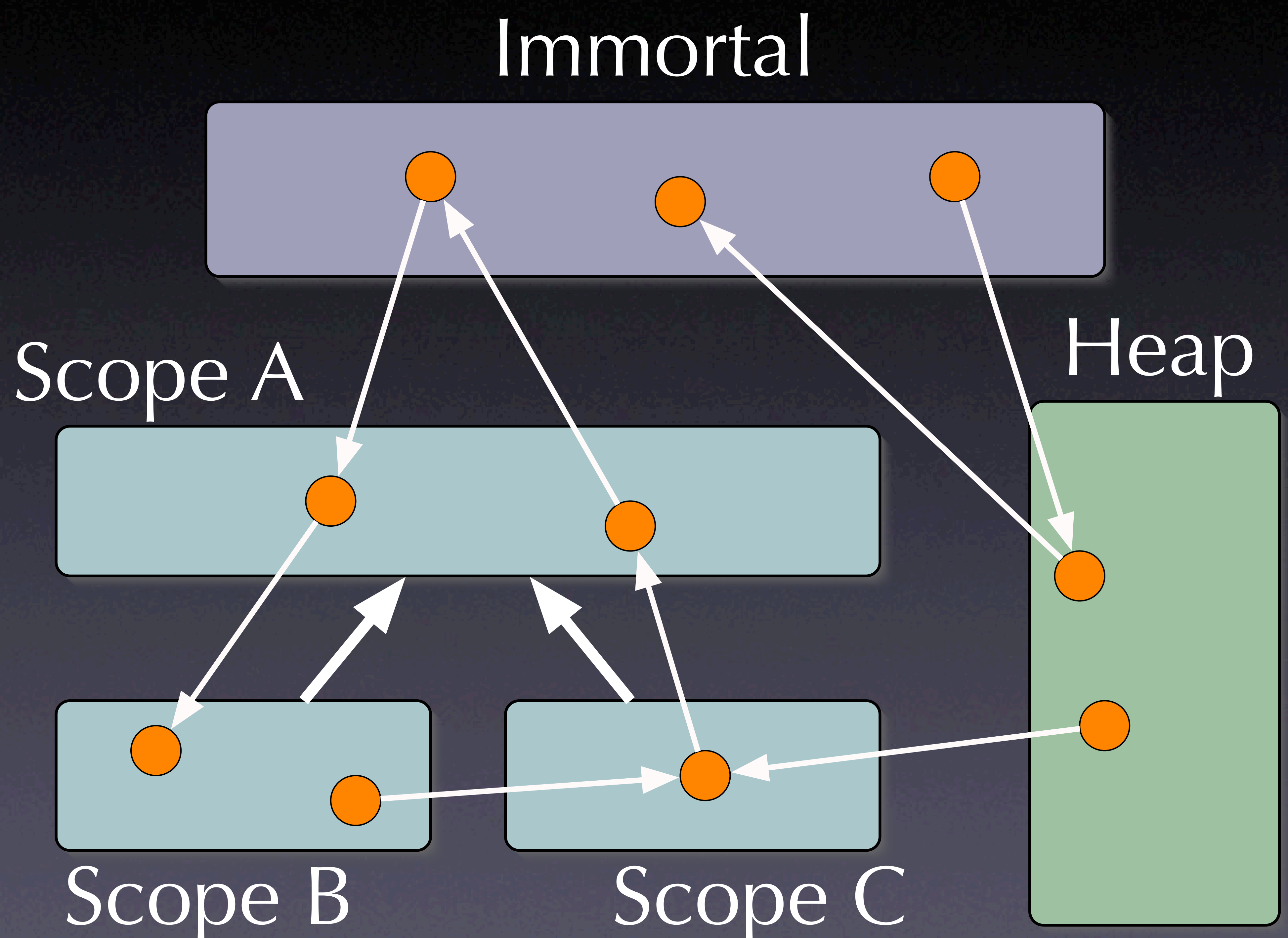
Contribution

- We present the first open-source implementation of both scoped memory and RTGC in one VM
- A discussion of software engineering benefits and dangers of scoped memory versus RTGC*
- An empirical performance evaluation using two realistic Real Time Java applications

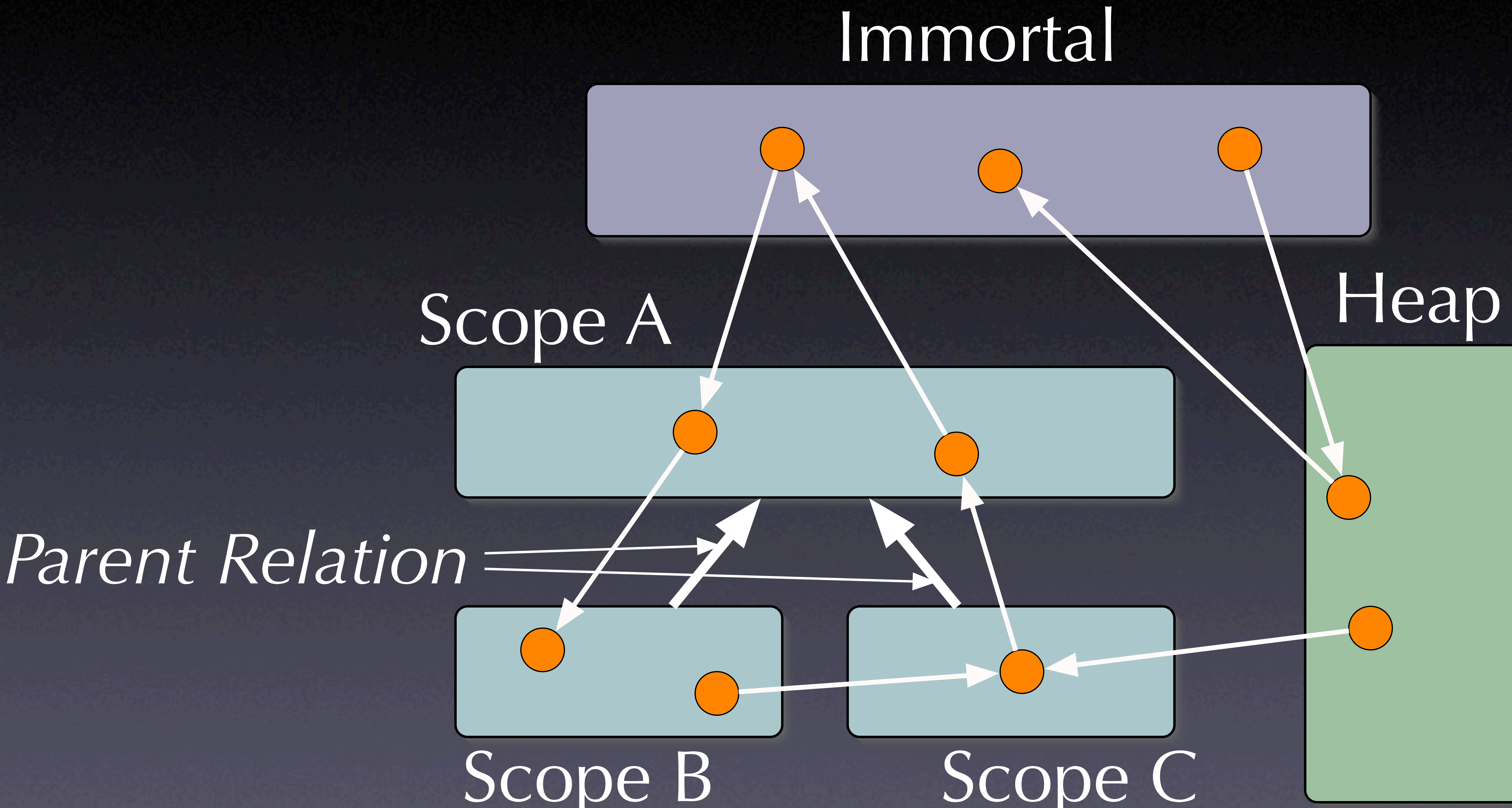
Talk Overview

- Summary of Scoped Memory
- Summary of RTGC (Metronome Style)
- Software Engineering Issues
- Evaluation

Scoped Memory

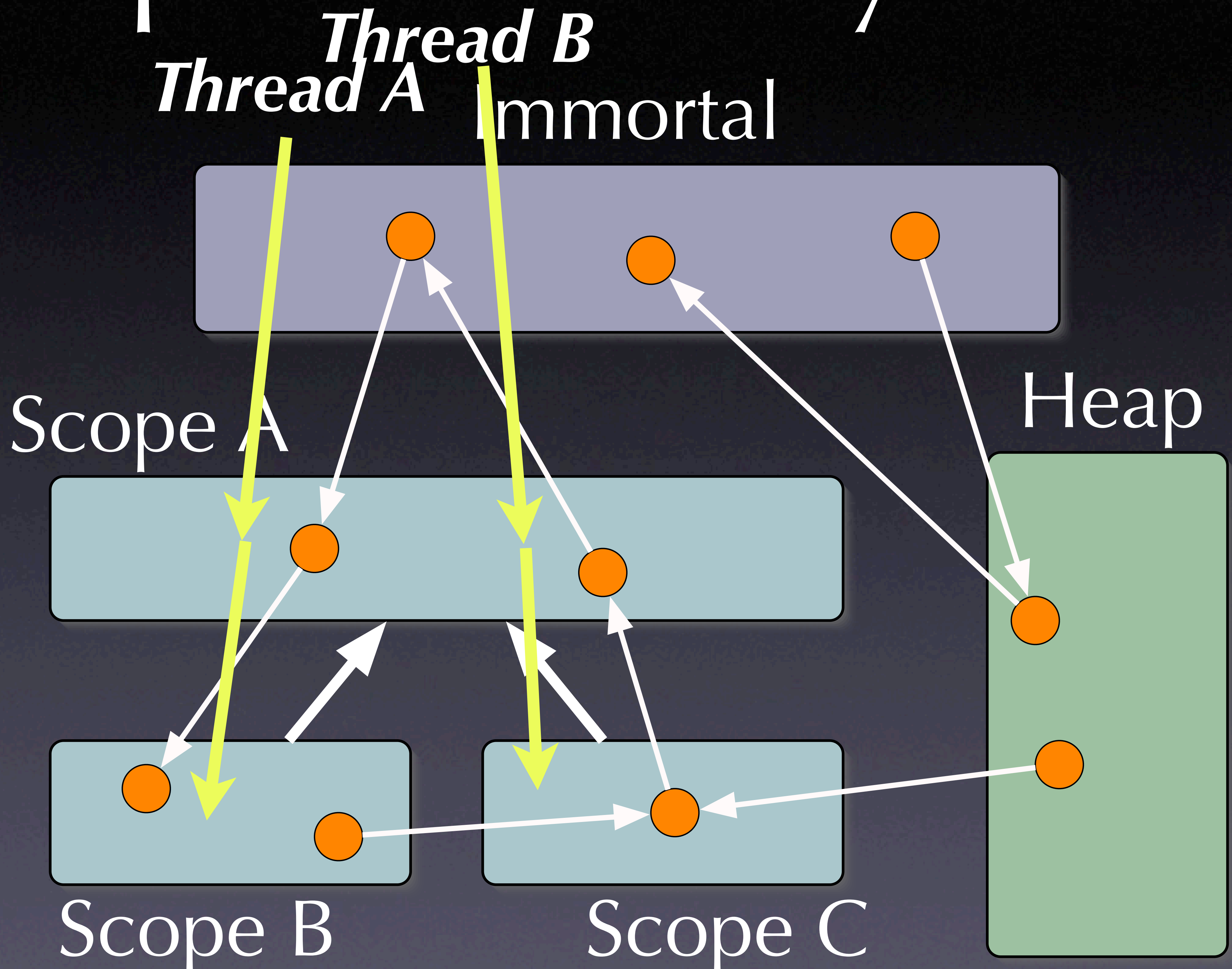


Scoped Memory

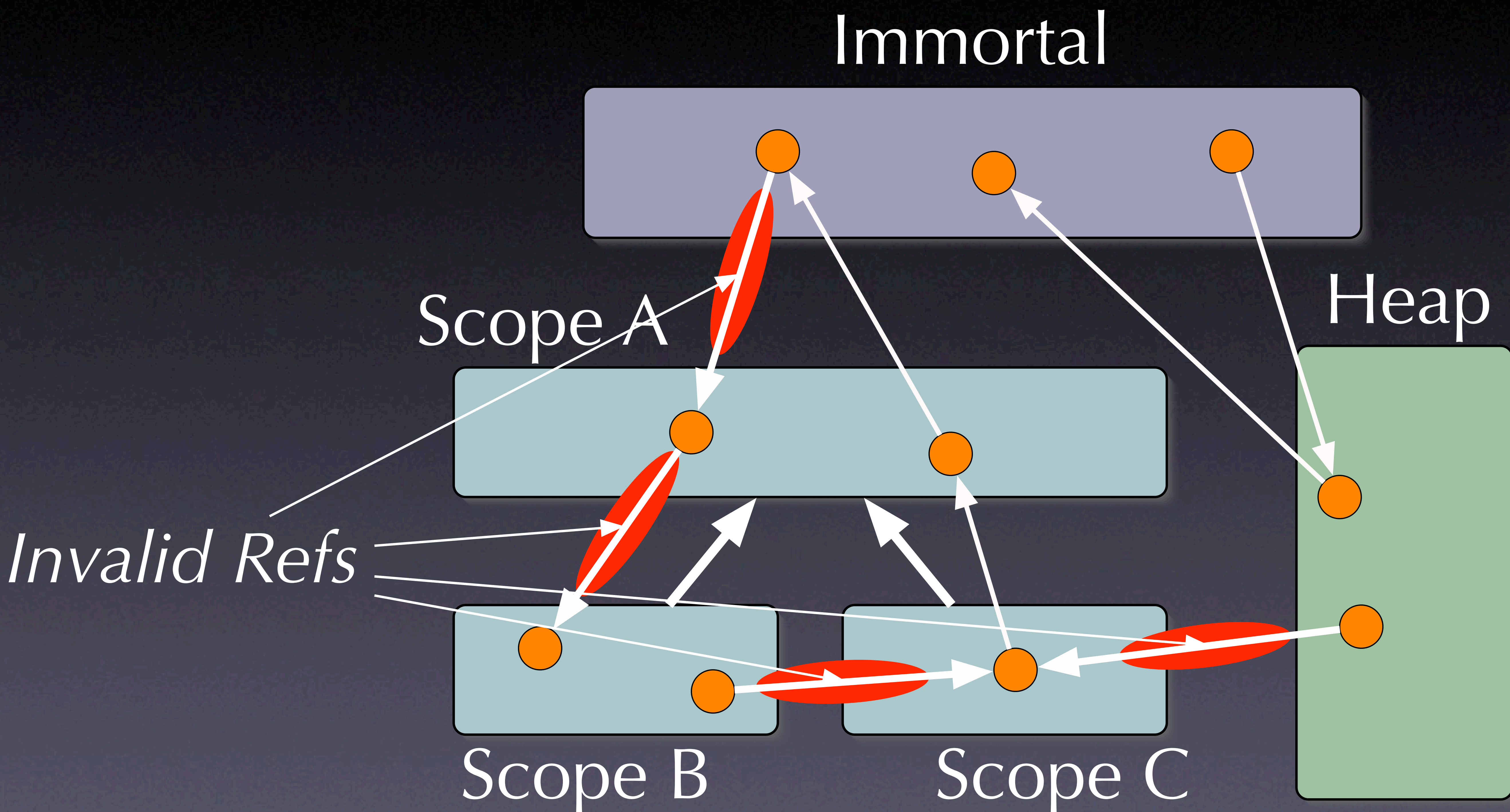


Scoped Memory

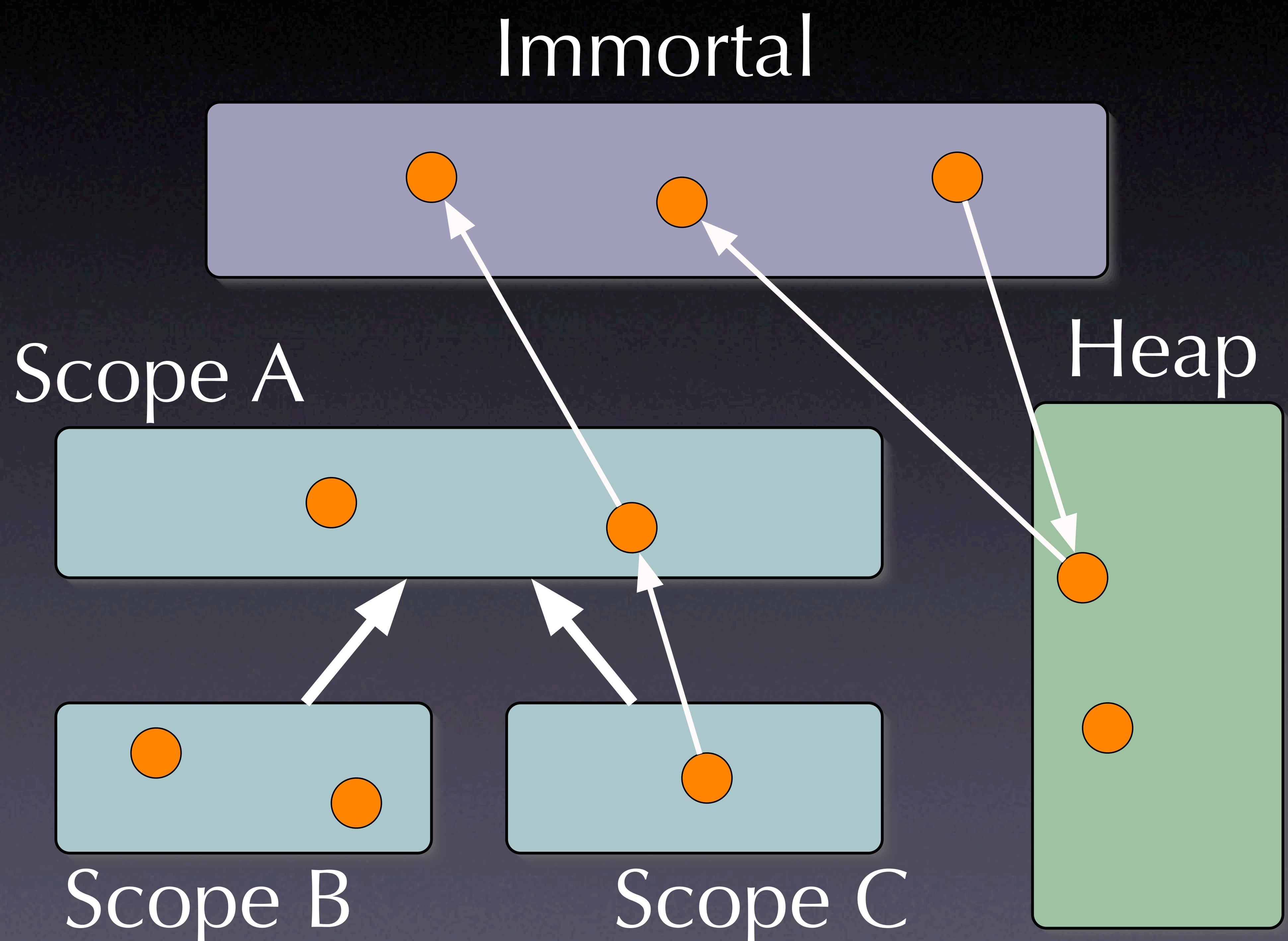
Threads create the scope hierarchy as they enter scopes.



Scoped Memory

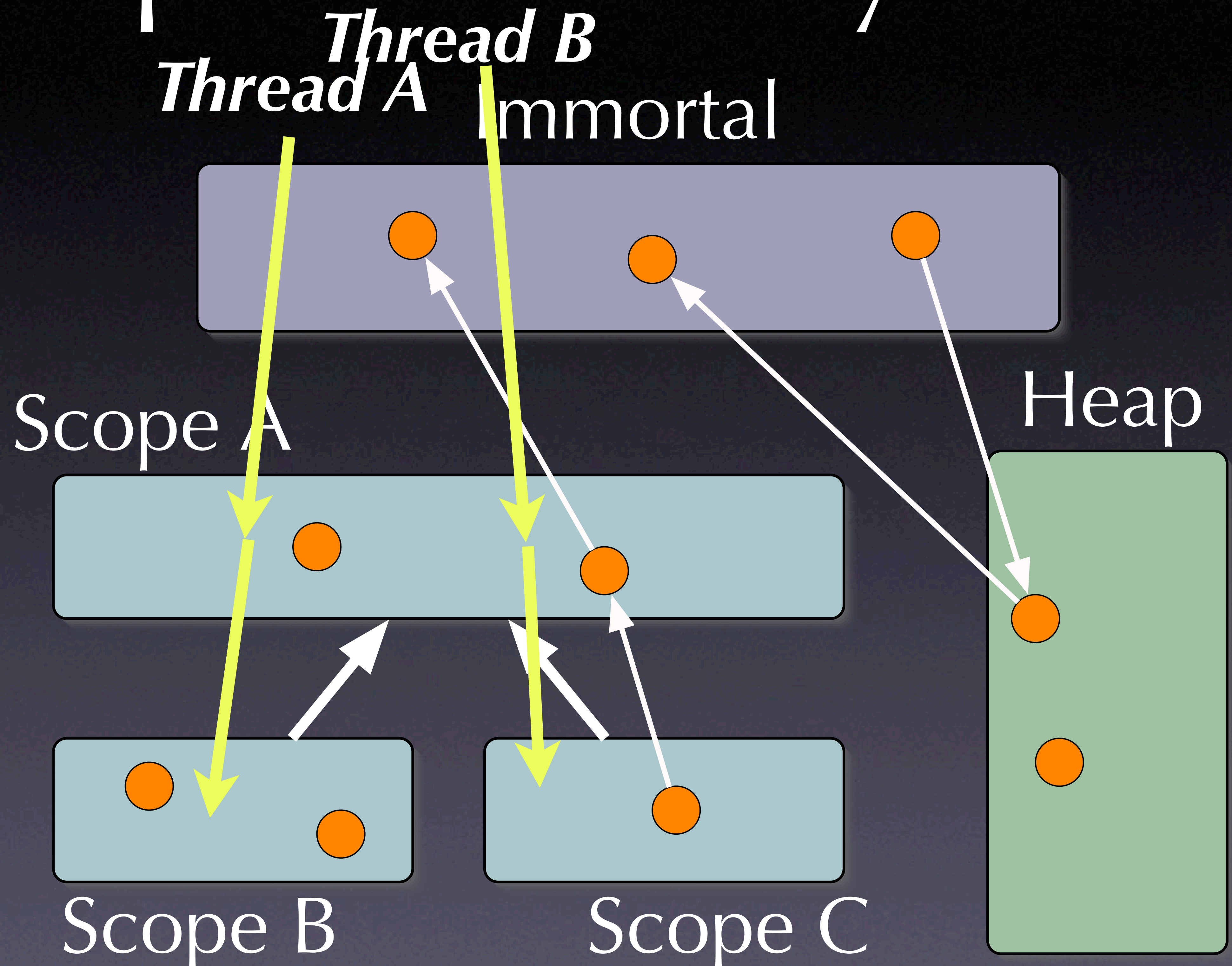


Scoped Memory



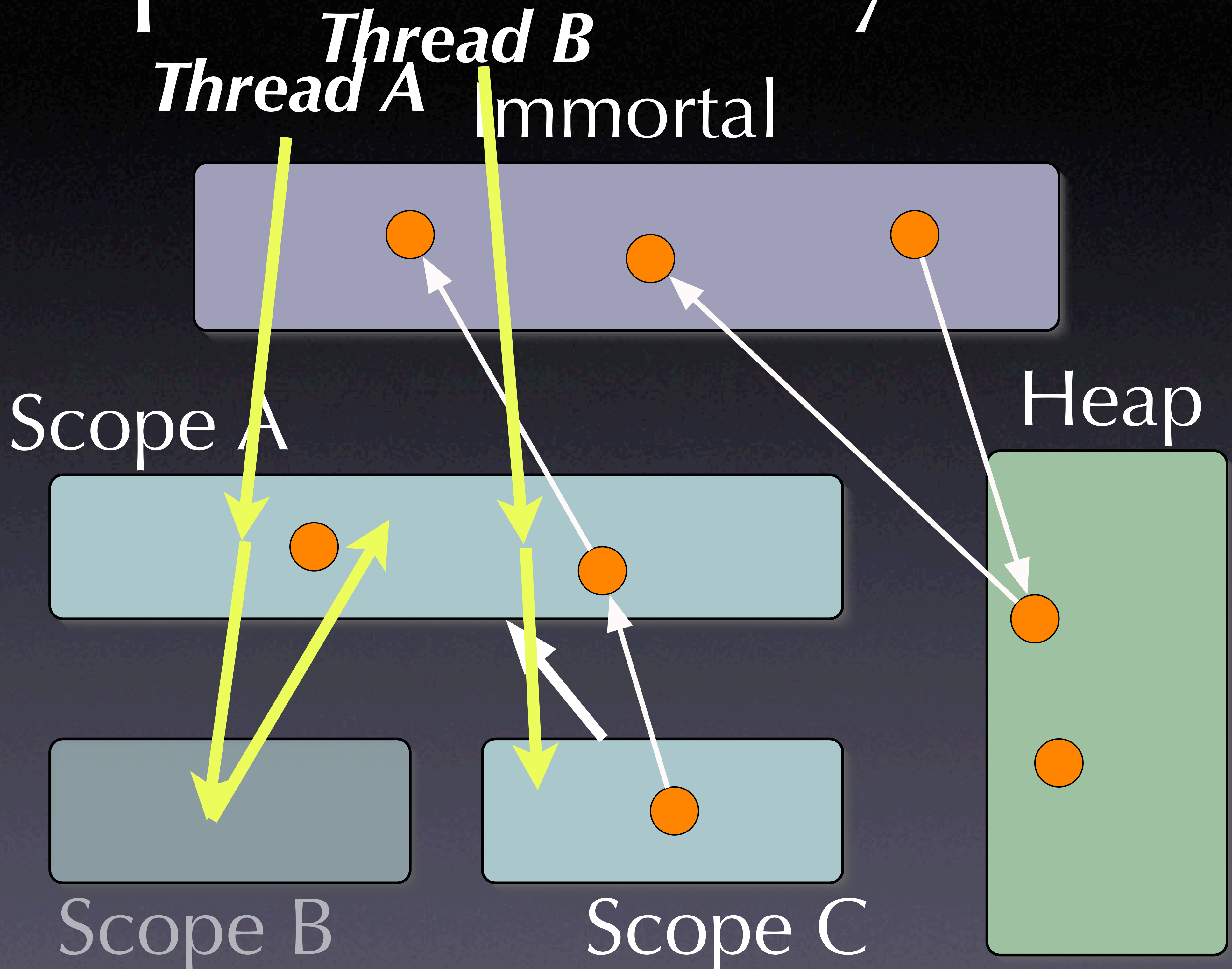
Scoped Memory

Objects in scopes are freed when the scope is exited.

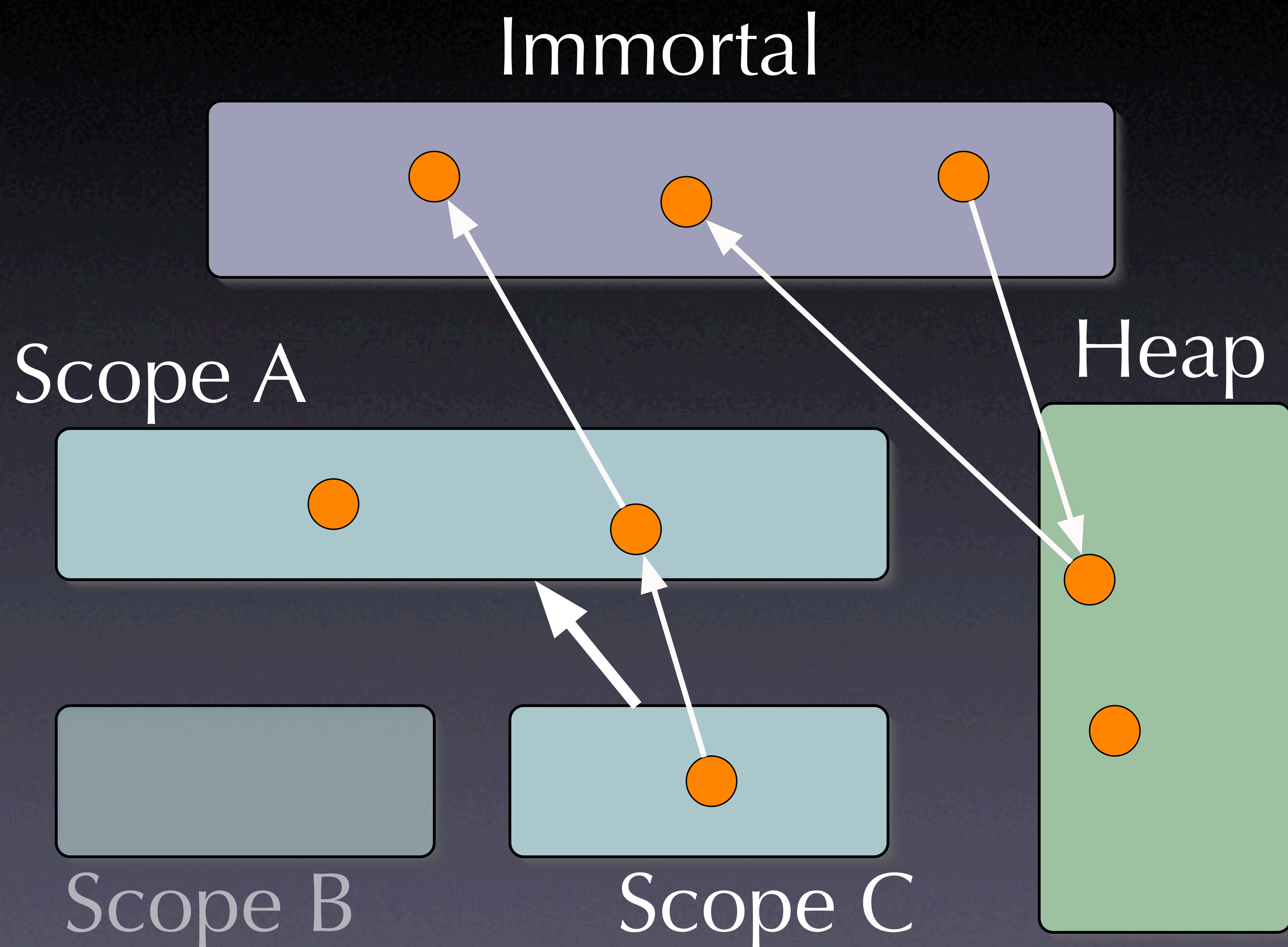


Scoped Memory

Objects in scopes are freed when the scope is exited.



Scoped Memory



Scoped Memory

- What we wanted: avoidance of GC interruptions.
- What scoped memory gives us:
  - Mostly-safe, somewhat-manual memory management
- To avoid GC interruptions we add ***no-heap threads***:
  - A no-heap thread cannot have references to the heap.

Scoped Memory Example

```
myScope = new LTMemory(65536, 65536);

myAction = new Runnable() {
        public void run() {
            new Object(); // allocated in scope
            // deallocated after we exit the scope
        }
};

// run myAction in myScope
myScope.enter( myAction );
```

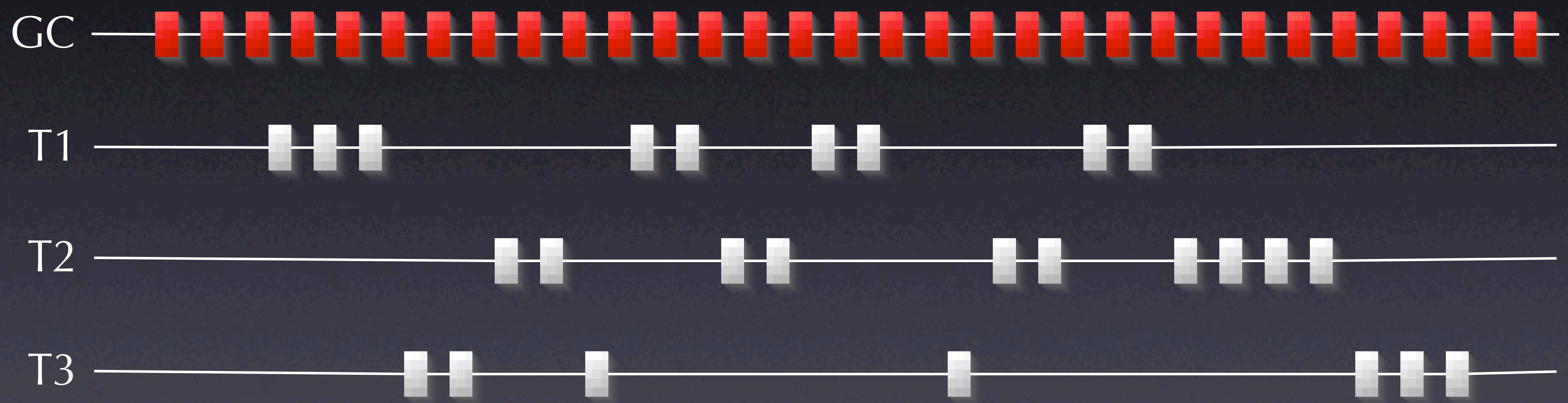
Scoped Memory Summary

- Threads enter/exit scopes following a stack discipline
- Objects deleted when scope exited
- Dynamic checks:
  - *Write Checks:* prevent dangling pointers
  - *Read Checks:* prevent no-heap threads from accessing the heap.

RTGC (The Metronome Way)

RTGC (The Metronome Way)

1) Control collector interruptions:



(*collector interruptions ~ 1ms*)

2) Insure that collector methods used by mutator are highly predictable (worst case ~ best case)

RTGC Implementation

- *"Insure that collector methods used by mutator are highly predictable (worst case ~ best case)"*
- We go to some trouble to make sure that the following are predictable:
  - Write Barrier
  - Allocation

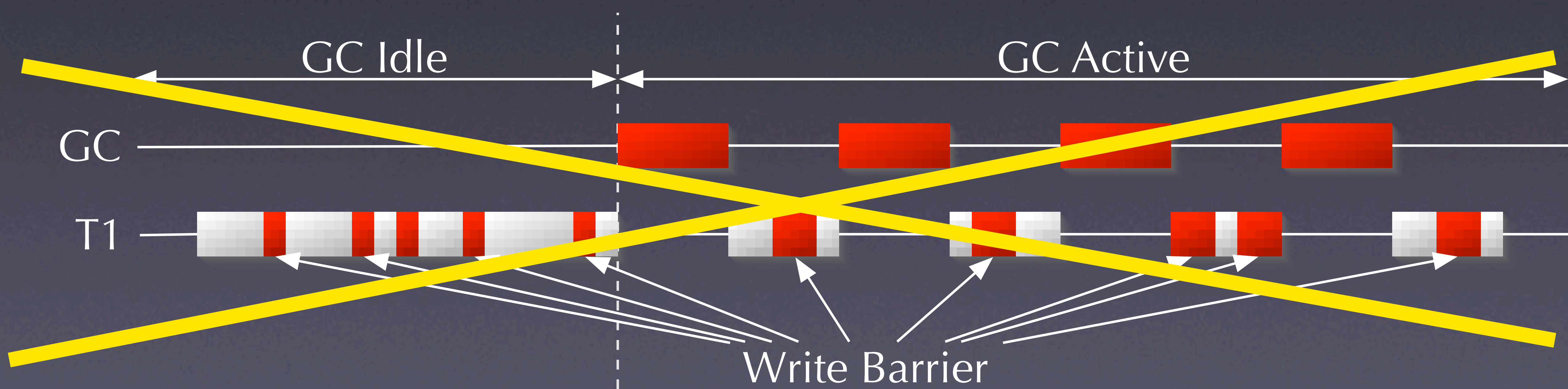
Write Barrier

- What it is:

A small piece of code inserted by the compiler at every write of a reference to memory. It guarantees that the collector does not lose track of objects.

- What we need it to do:

Do not exhibit worse performace during collection than when the collector is idle!



Write Barrier

- Idea: Whatever the worst case is, we need to simulate it.
- Solution: Our write barrier always performs at worst case when the GC is idle.

Allocation

- No slow path! Collector ensures that all free space is accounted for.
- Worst case: empty freelist, allocate new page, bump pointer in page

Software Engineering Issues

We now consider the software engineering impact of the two styles of Real Time Java memory management.

- Scoped Memory
- Real-Time Garbage Collection

Scoped Memory

Pros	*Cons*
Fast Alloc	Read Checks
Fast Free	Write Checks
Fail-Fast	Not Automatic

RTGC

Pros	*Cons*
Safe	Overhead
Automatic	Analysis Burden

Performance

- **Methodology**
- RTGC Overhead
- RTZen Performance
- CD Performance

Methodology

- We use the OpenVM virtual machine and the J2c ahead-of-time compiler.
- Our platform is an Pentium IV with 512MB RAM running Linux 2.6.
- Memory Management:
  - Java-GC (mostly-copying, semi-space)
  - Java-GC + Scopes
  - RTGC

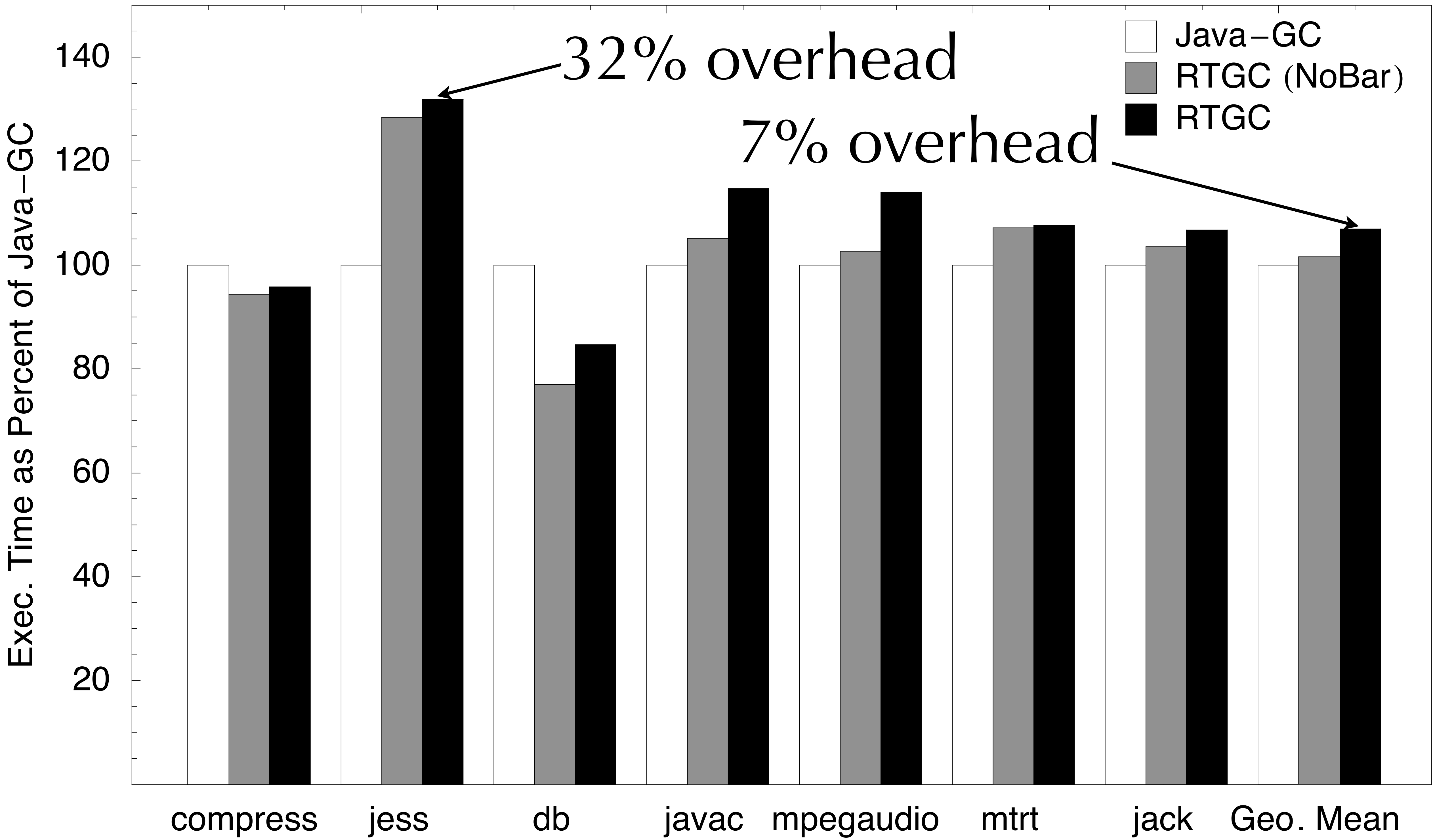
Performance

- Methodology
- **RTGC Overhead**
- RTZen Performance
- CD Performance

RTGC Overhead

- We use the industry standard SPECjvm98 benchmark suite.
- Three collectors:
  - Java-GC
  - RTGC w/o write barriers
  - RTGC

SPEC Performance



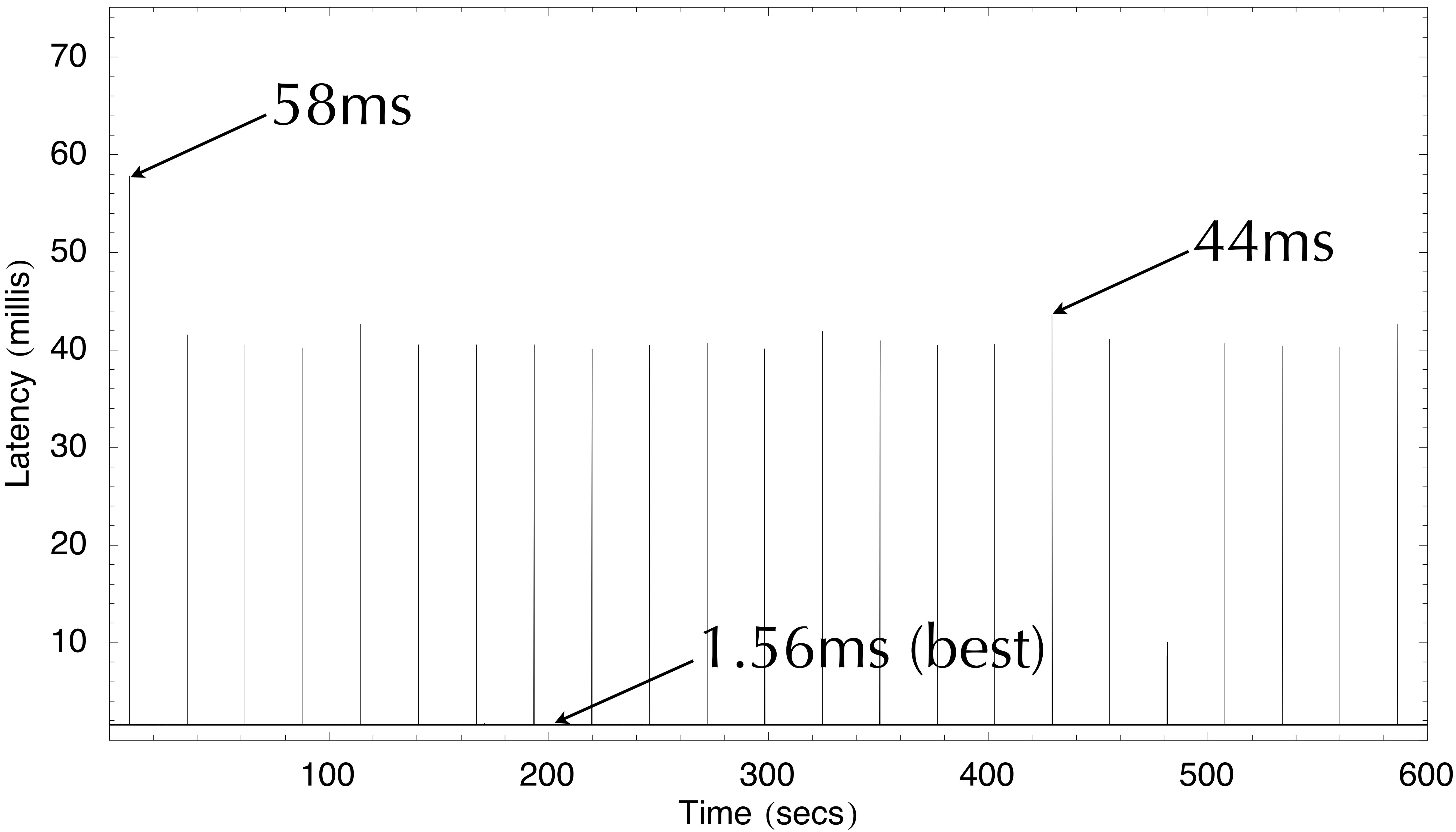
Performance

- Methodology
- RTGC Overhead
- **RTZen Performance**
- CD Performance

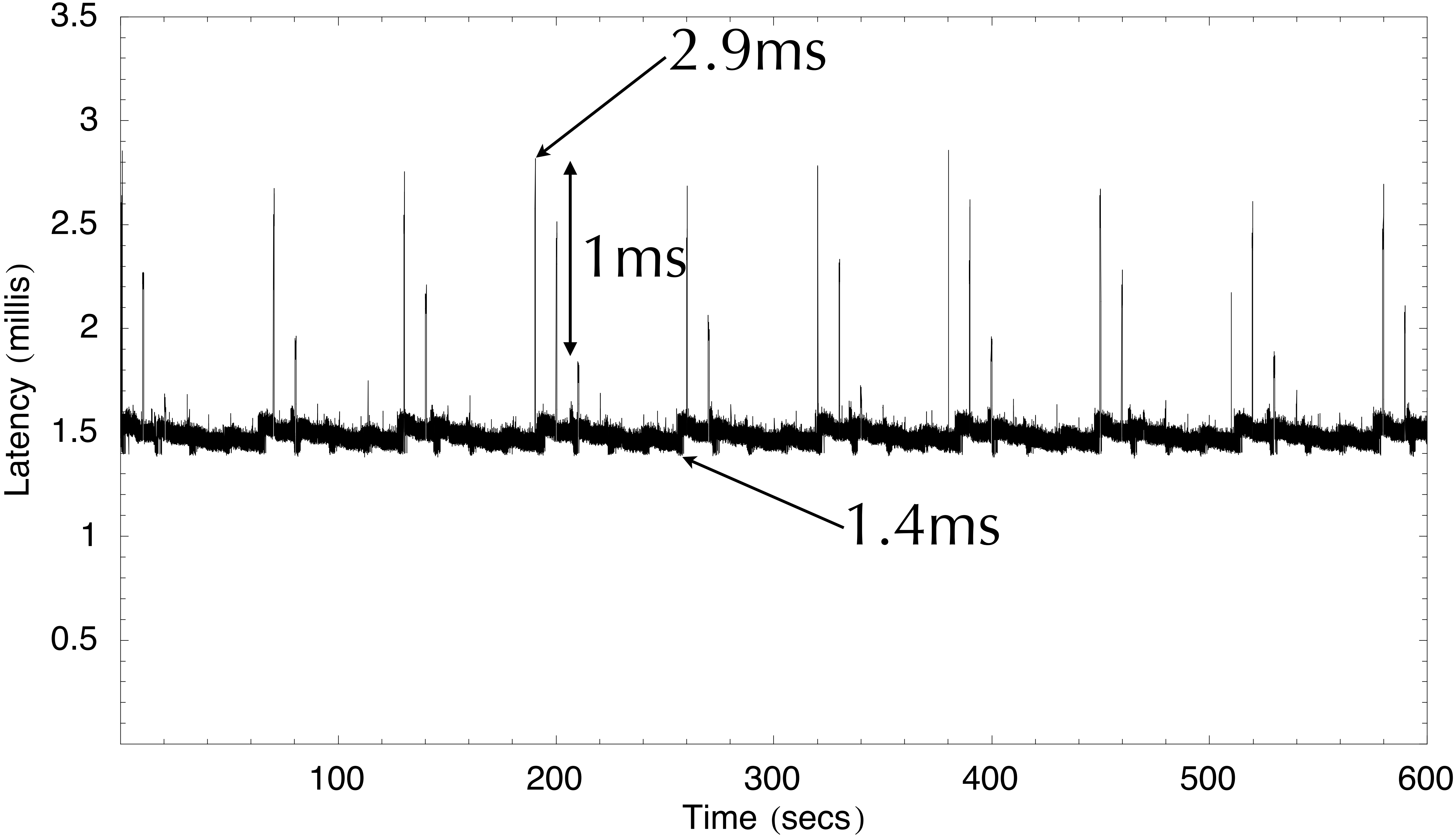
RTZen Performance

- RTZen is a real-time CORBA implementation.
- RTZen uses scoped memory. We run it with and without scopes.
- We test four memory management configurations:
  - Java-GC
  - RTGC
  - Scopes
  - Scopes w/o checks (see paper)

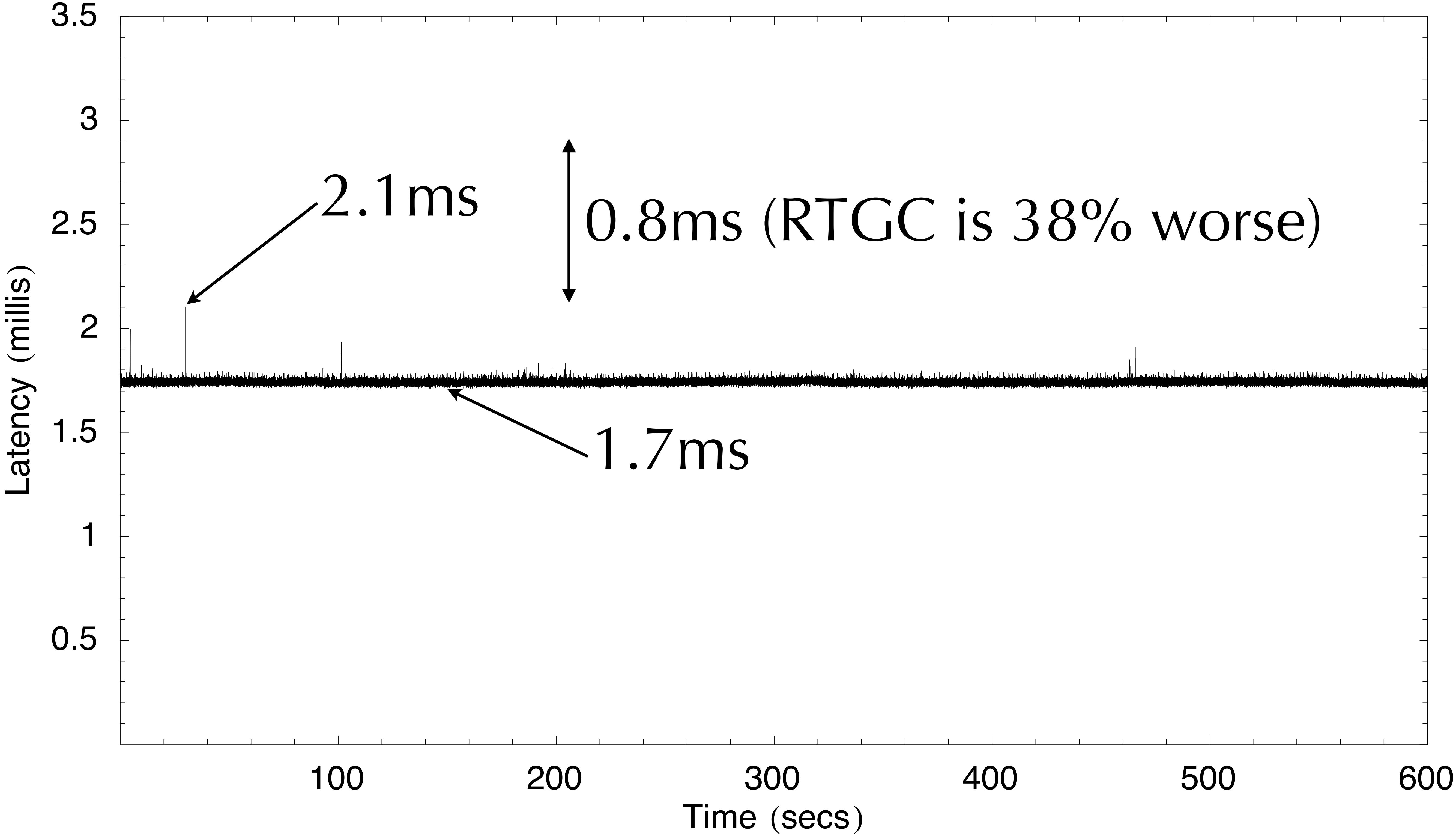
RTZen Latency v. Time, Java-GC



RTZen Latency v. Time, RTGC



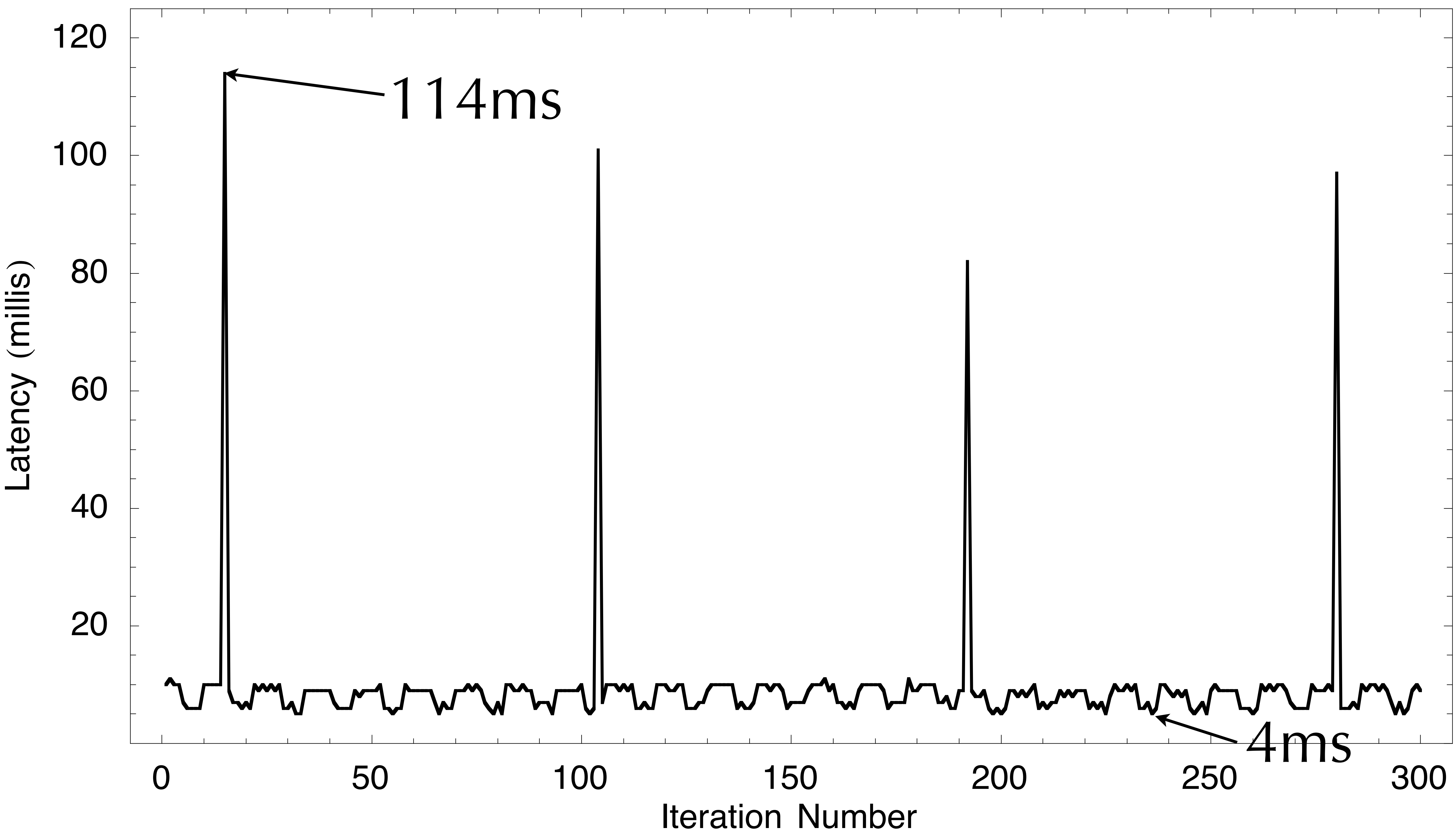
RTZen Latency v. Time, Scopes



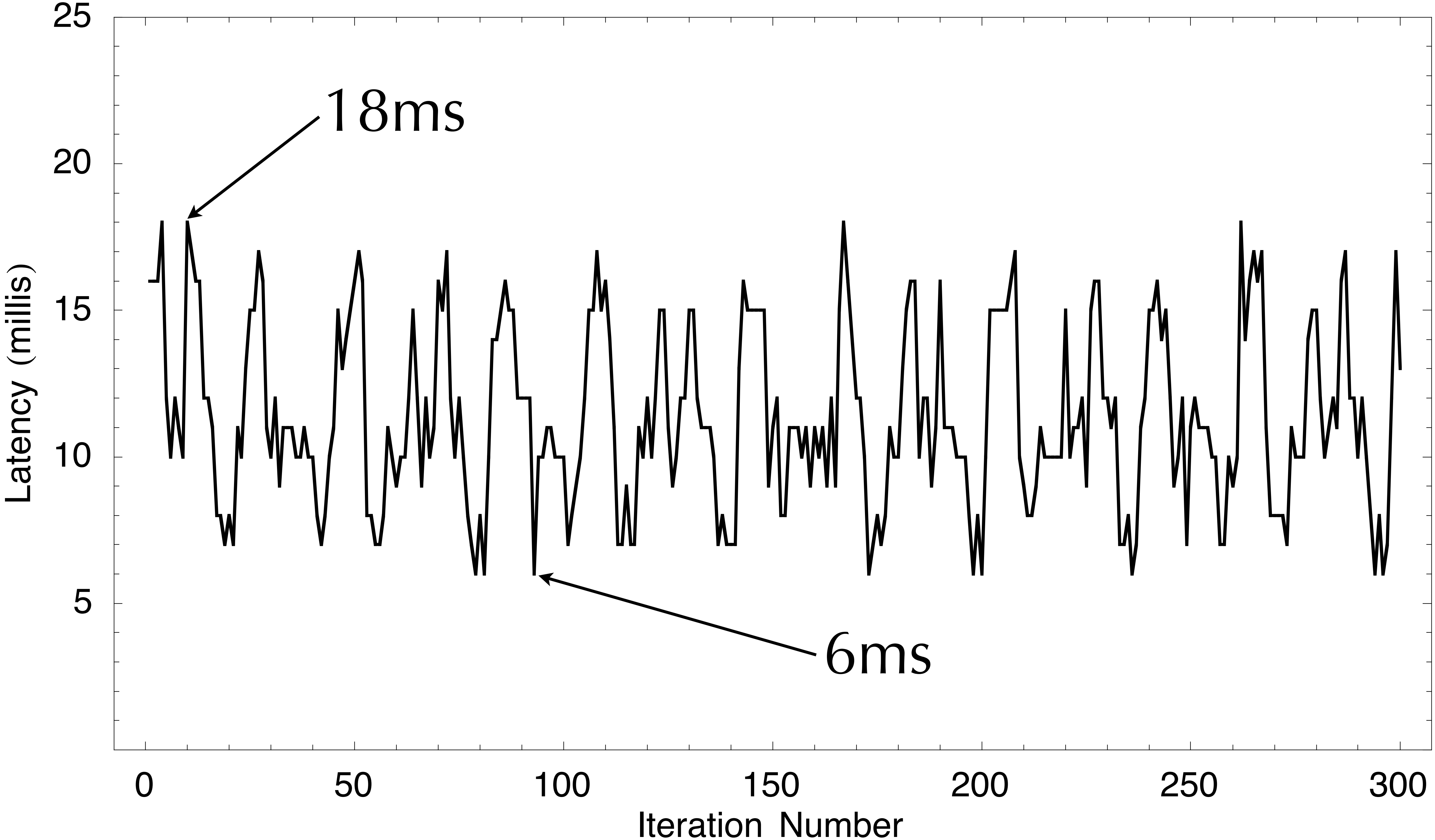
Performance

- Methodology
- RTGC Overhead
- RTZen Performance
- **CD Performance**

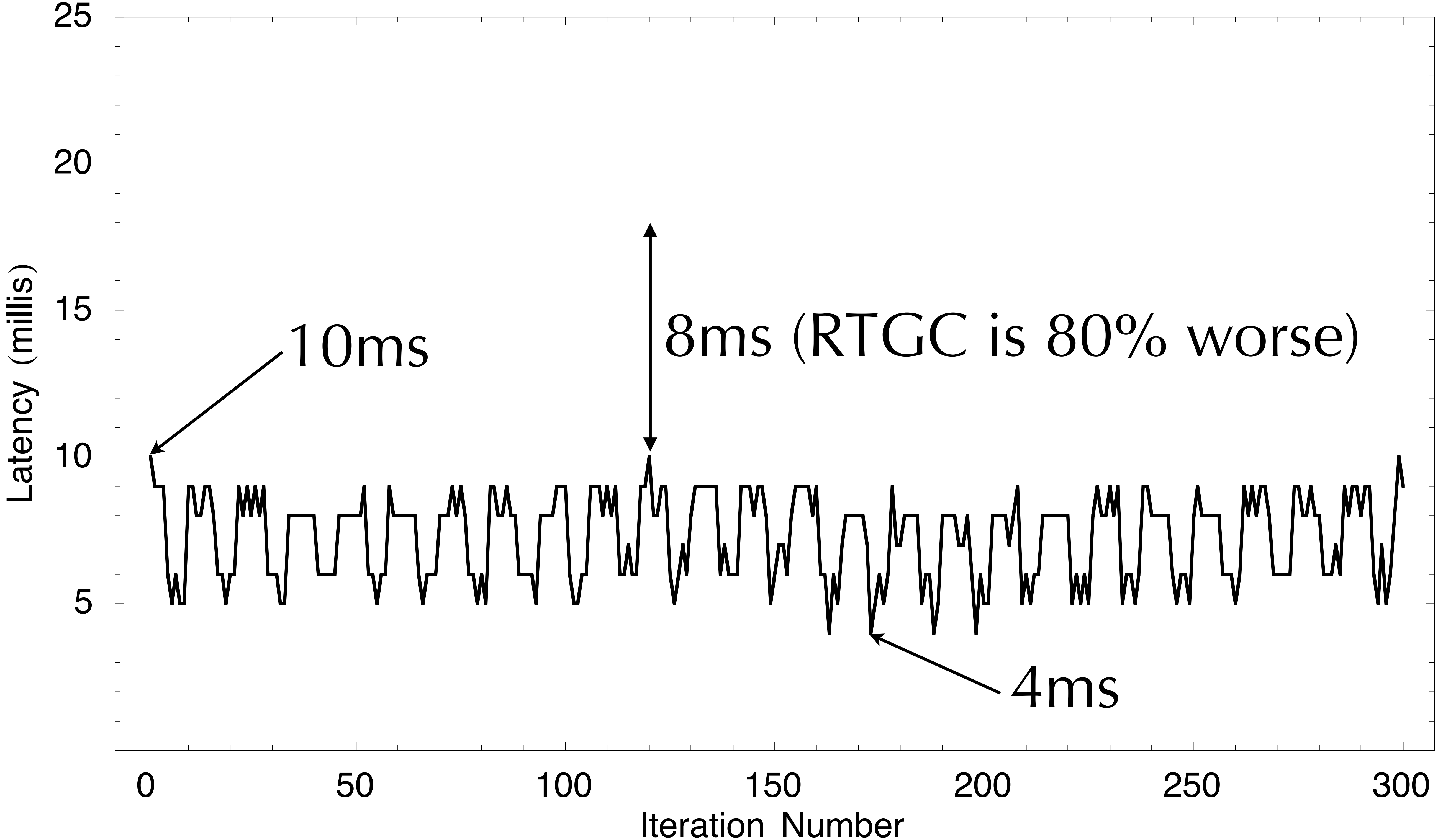
CD Latency v. Iteration, Java-GC



CD Latency v. Iteration, RTGC



CD Latency v. Iteration, Scopes



Conclusion

- In RTGC, raw throughput suffers only 7% for SPECjvm98 (though it is 32% worse in the jess benchmark).
- RTGC has between 38% (RTZen) and 80% (CD) worse latency in the worst case.
- Your Mileage May Vary, but:
  - If you can tolerate the overhead, RTGC is easier.
  - Scopes are still best if your specification is tight.
- Read the paper for a more in-depth evaluation!