Concurrent GC Leveraging Transactional Memory

Phil McGachey	Purdue University
Ali-Reza Adl-Tabatabai	Intel
Richard L. Hudson	Intel
Vijay Menon	Intel
Bratin Saha	Intel
Tatiana Shpeisman	Intel

Introduction

- Multi-cored desktops now standard
- Look to simplify concurrent programming
- Two emergent technologies
  - Transactional memory
  - Concurrent garbage collection
- We exploit the overlap in mechanisms

Our Goals

- Leverage transactions infrastructure for GC
- Target desktop applications
  - Games, multimedia, VOIP
- Aim to keep 90% of pauses under 1ms
  - Competitive with modern real-time GCs

Copying GC

Copying GC

Java Heap

Copying GC

Java Heap

Copying GC

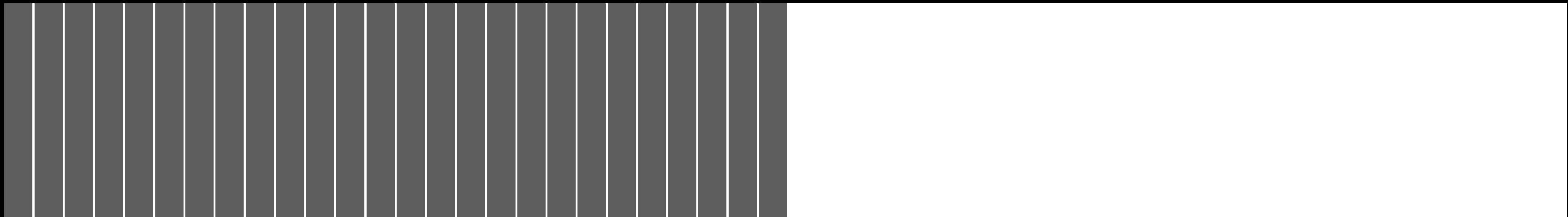
Java Heap

Copying GC

Java Heap

Copying GC

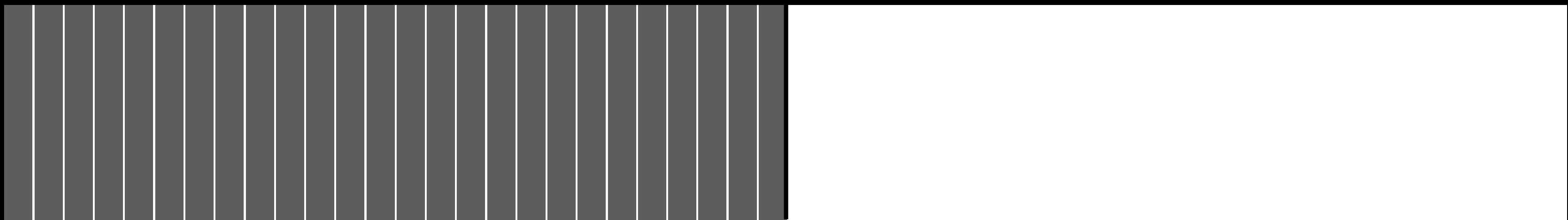
Java Heap



Copying GC

Java Heap

From Space To Space

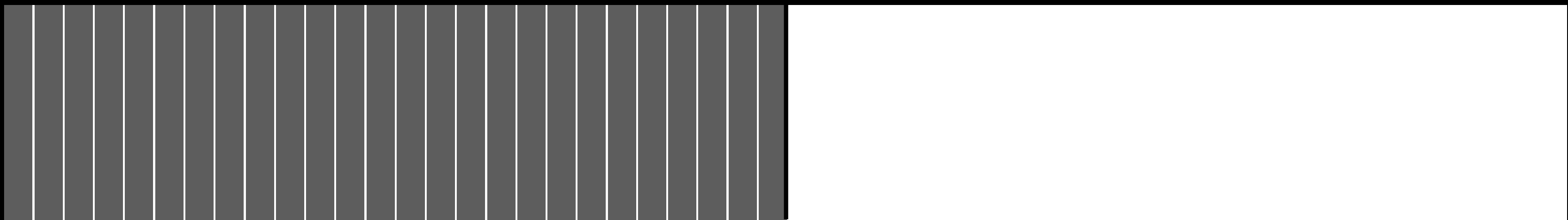


Copying GC

Java Heap

From Space

To Space



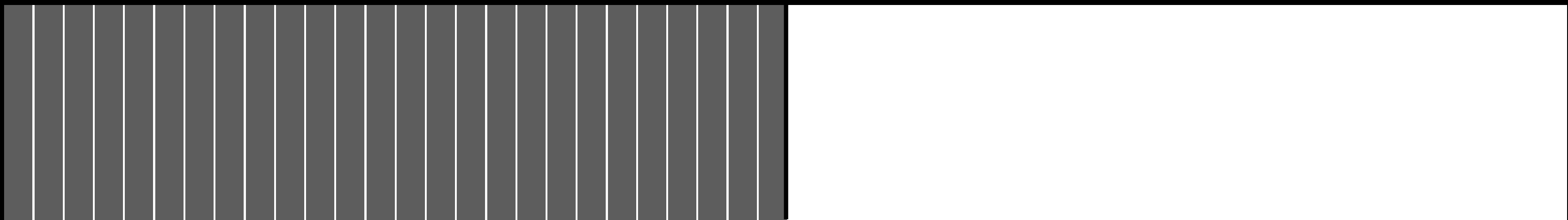
Thread Stacks

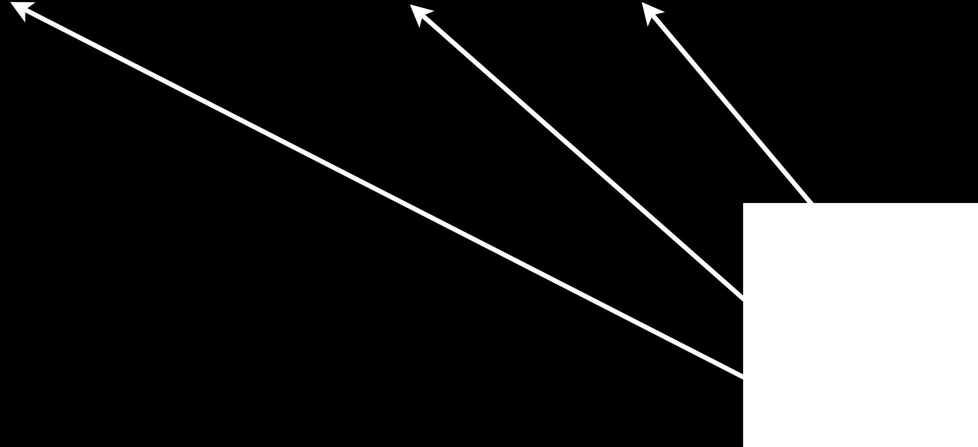
Copying GC

Java Heap

From Space

To Space



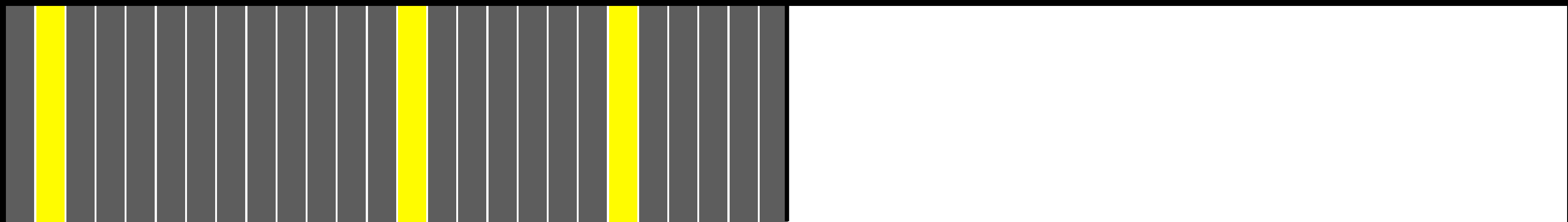


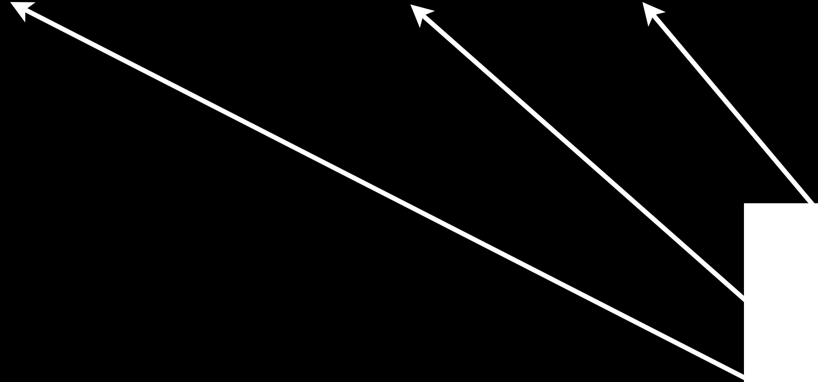
Thread Stacks

Copying GC

Java Heap

From Space | To Space



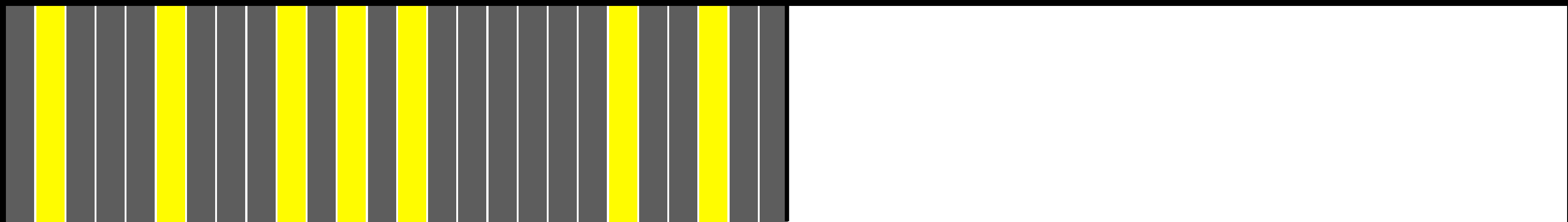


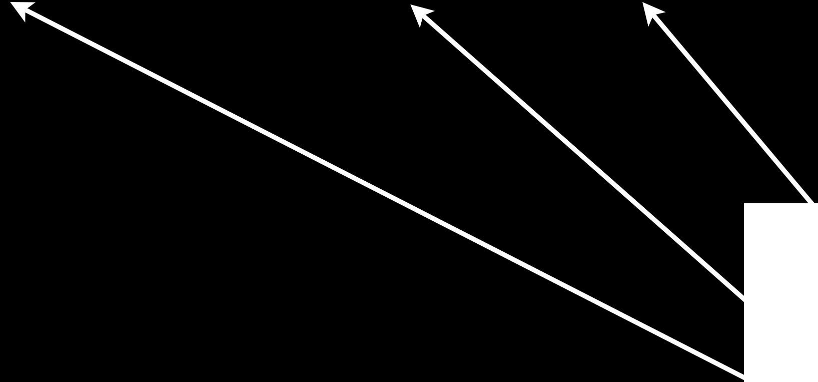
Thread Stacks

Copying GC

Java Heap

From Space To Space



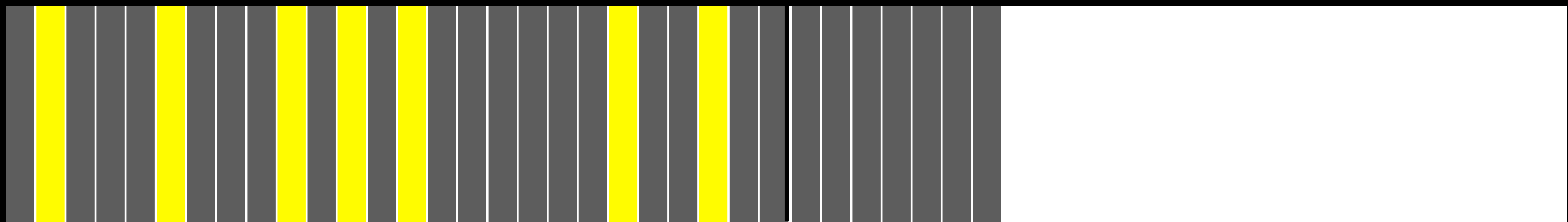


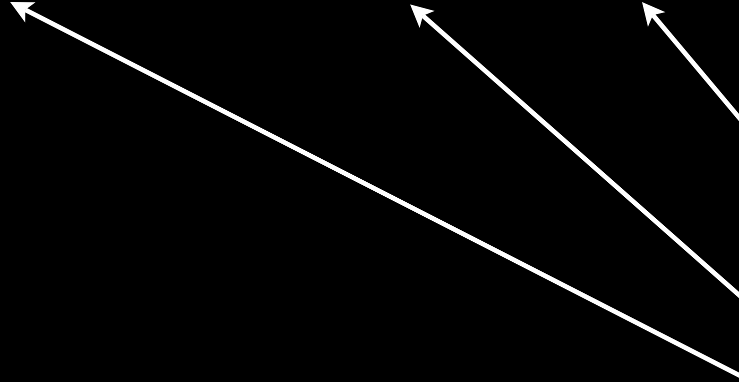
Thread Stacks

Copying GC

Java Heap

From Space   To Space





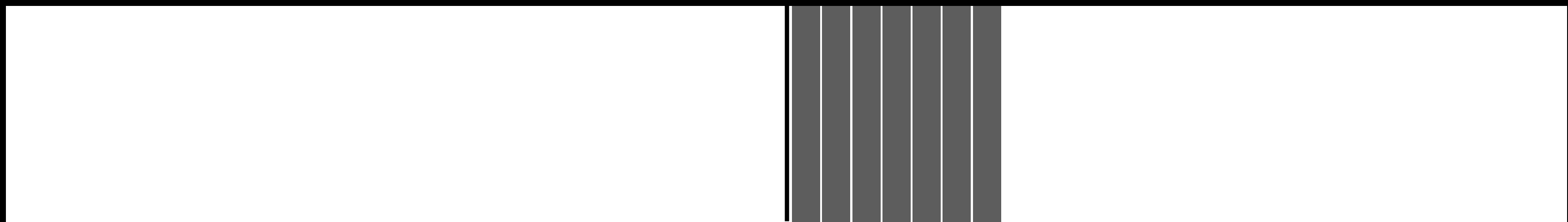
Thread Stacks

Copying GC

Java Heap

From Space

To Space



Thread Stacks

Multi-Cored GC

- Most current desktop GCs “Stop the World”
- Becomes unfeasible with many cores
  - Overhead of pausing the threads
  - Bottlenecks in the GC code
  - Large heaps increase GC time
- Concurrent GC requires synchronization

Concurrent Copying

GC Thread

Application Thread

Concurrent Copying

GC Thread

Application Thread

Concurrent Copying

GC Thread

Application Thread

Concurrent Copying

GC Thread

Application Thread

Begin Copy

Copy Field A

Copy Field B

Concurrent Copying

GC Thread

Begin Copy

Copy Field A

Copy Field B

Copy Field C

Application Thread

Write to Field A

Concurrent Copying

GC Thread

Begin Copy

Copy Field A

Copy Field B

Copy Field C

Write Forwarding Ptr

Application Thread

Write to Field A

Concurrent Copying

GC Thread

Begin Copy

Copy Field A

Copy Field B

Copy Field C

Write Forwarding Ptr

Application Thread

Write to Field A

Read from Field A

Concurrent copying must be atomic

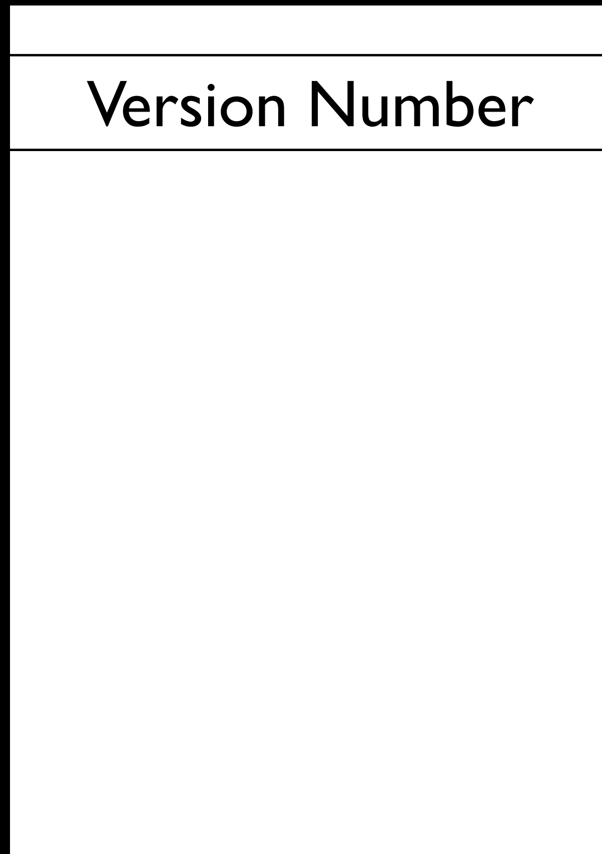
Implementing Atomicity

Implementing Atomicity

Object A

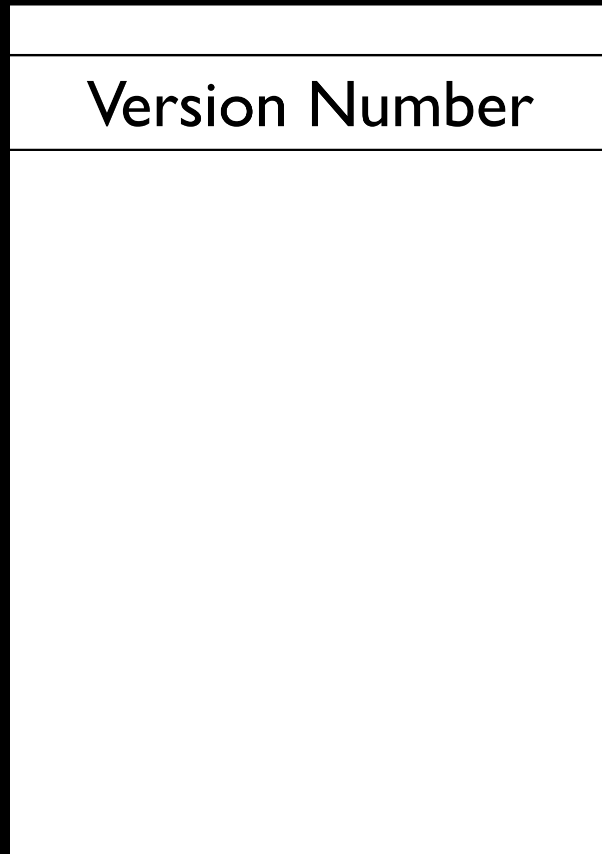
Implementing Atomicity

Object A



Implementing Atomicity

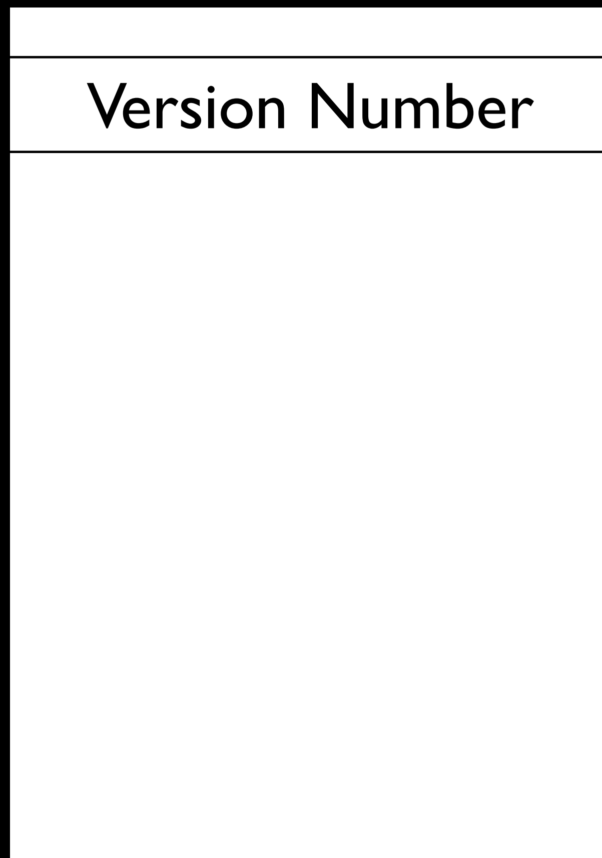
Object A

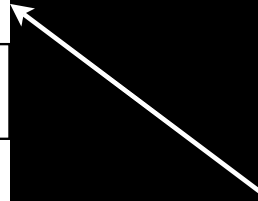


Tx1: Read

Implementing Atomicity

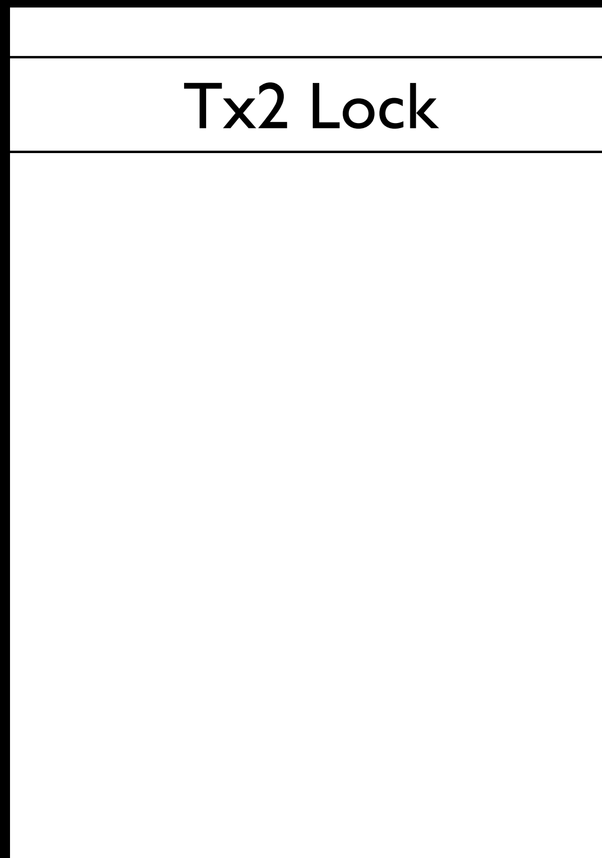
Object A

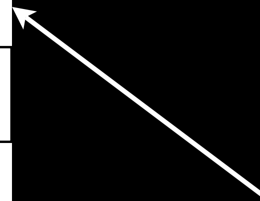




Implementing Atomicity

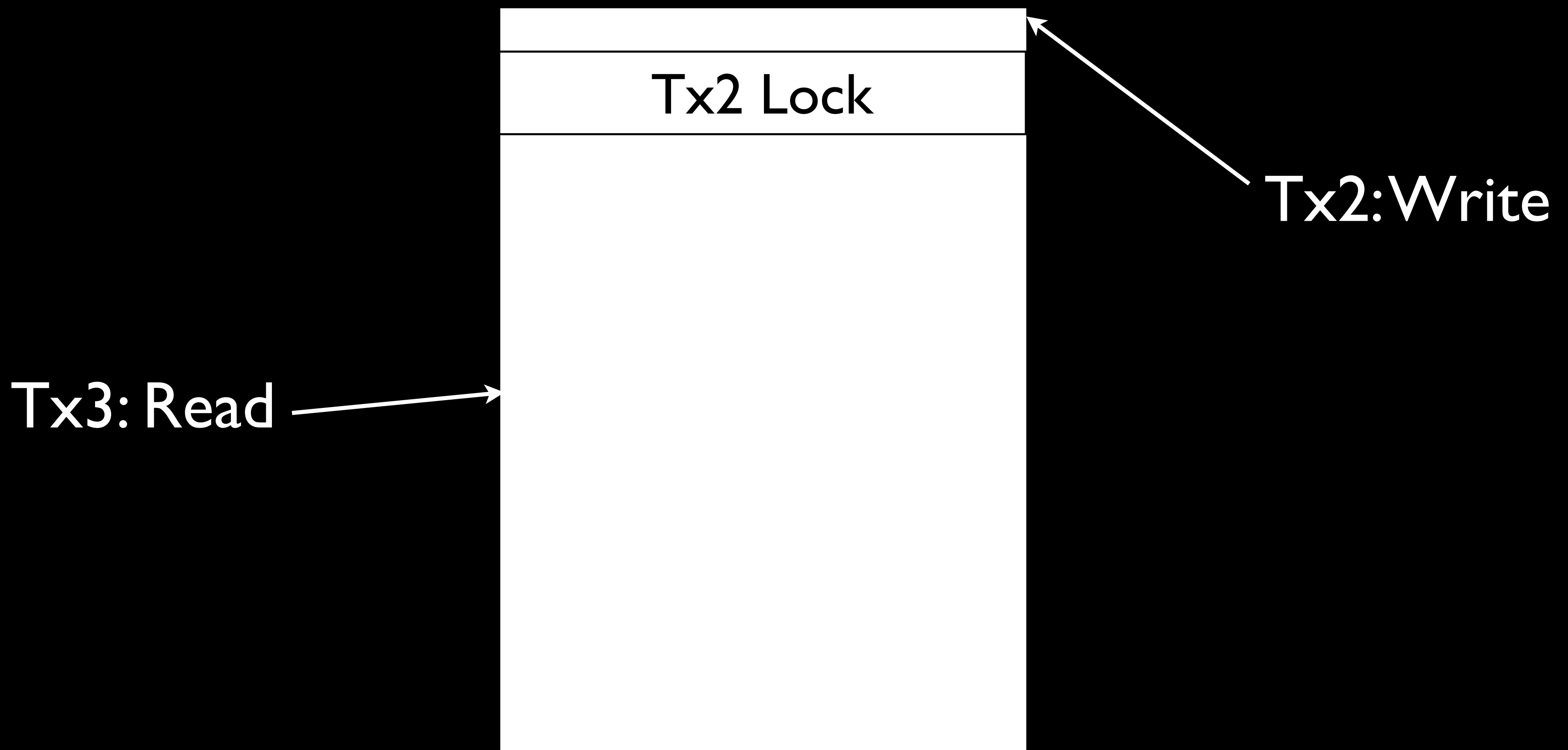
Object A





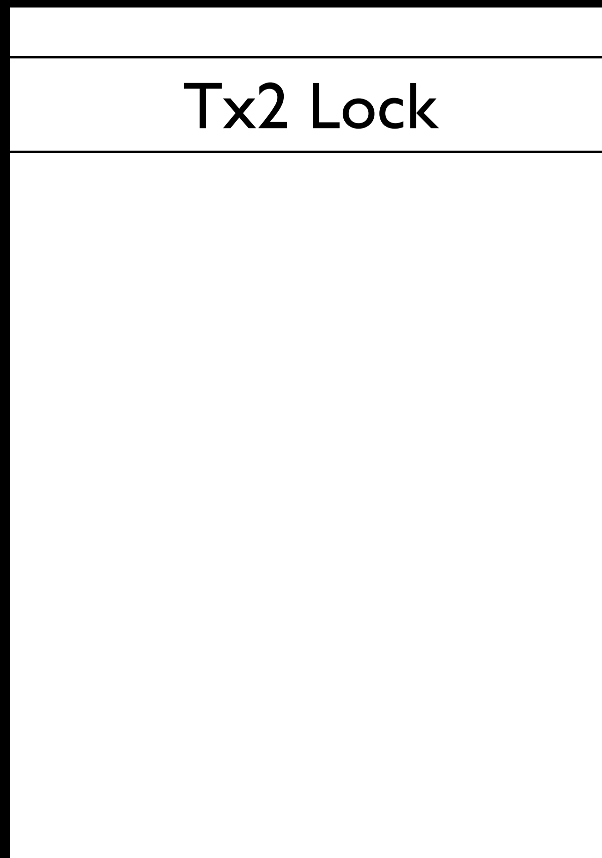
Tx2: Write

Implementing Atomicity



Implementing Atomicity

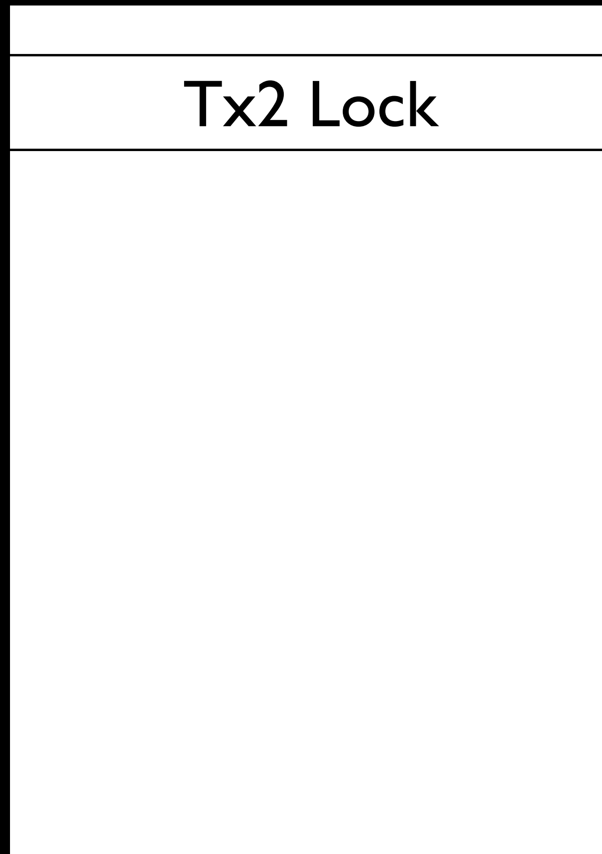
Object A



Tx2: Write

Implementing Atomicity

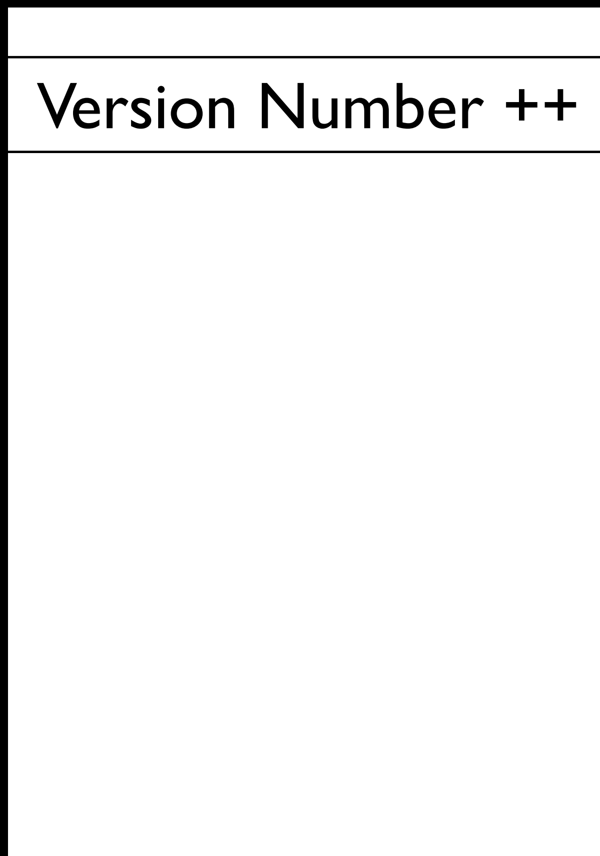
Object A



Tx2: Validate

Implementing Atomicity

Object A

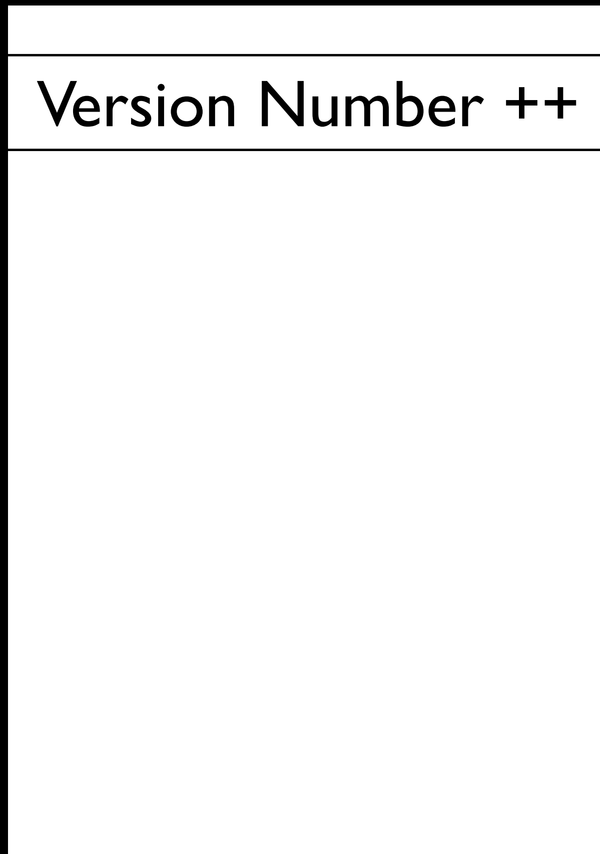


Tx2: Commit

Implementing Atomicity

Object A

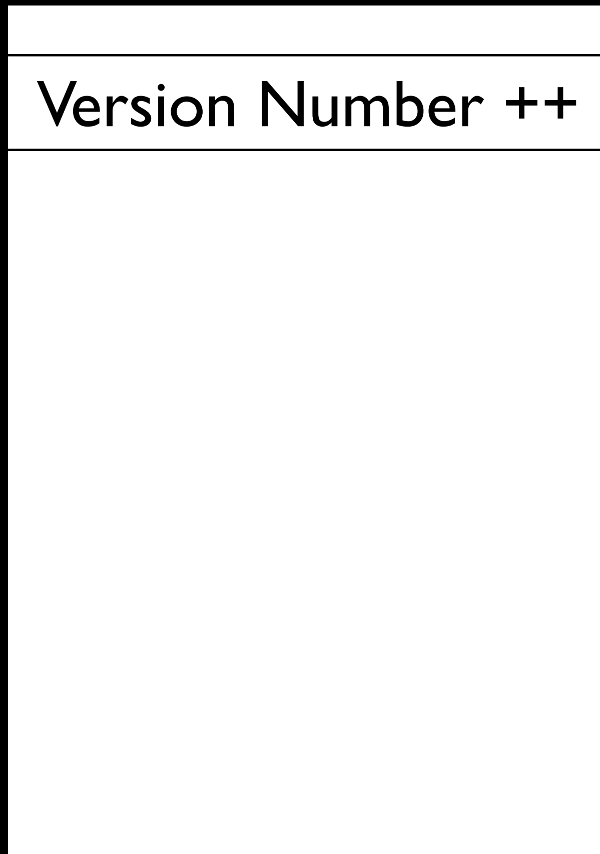
Tx1: Validate



Implementing Atomicity

Object A

Tx1: Abort



Transactional Integrity

- Two forms of transactional memory
- Weak atomicity
  - Programmer marks transactional accesses
  - Atomic / non-atomic conflicts possible
- Strong atomicity
  - System ensures that all accesses are atomic
- We build upon strong atomicity

Transactional Copying

- Concurrent copying must be atomic
- Wrap each copy operation in a transaction?
  - No lost updates due to strong atomicity
  - Each copy operation incurs overhead
- Perform all copies as one large transaction?
  - Increased probability of conflicts
- We leverage the underlying infrastructure

Synergy Part 1

- Shared mechanisms
  - GC must observe modifications to objects
  - TM must detect conflicts
- We leverage the overlap
  - Treat "to-space" objects as speculative

The GC Algorithm

- Don’t stop the world
  - Threads paused one at a time
  - Minimize work during each pause
- Work split over three phases
- Copy a portion of the heap per GC cycle
- Use TM infrastructure for copying

Pauses

- Phase changes
  - All threads must be aware of phase change
  - Pause threads to synchronize memory
- Mark and Flip phases
  - Pause each thread to scan stack
  - Pause to ensure all references processed

Transactional Copying

GC Thread

Store version #

Application Thread

Transactional Copying

GC Thread

Store version #

Transactional Copying

GC Thread

Store version #

Copy Field A

Copy Field B

Application Thread

Transactional Copying

GC Thread

Store version #

Copy Field A

Copy Field B

Copy Field C

Application Thread

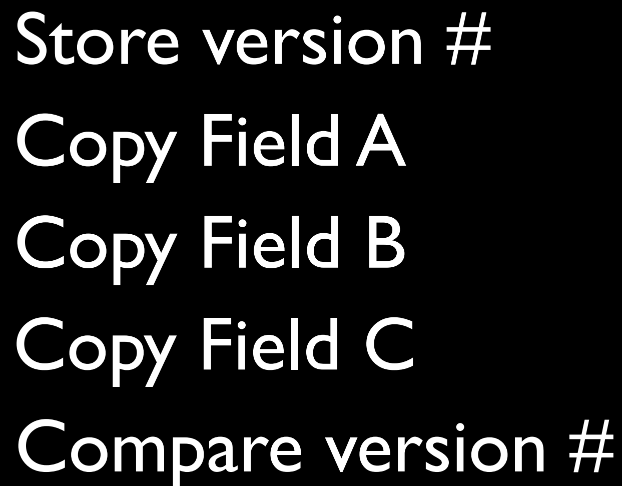
Write to Field A

- Increment Version #

Transactional Copying

GC Thread

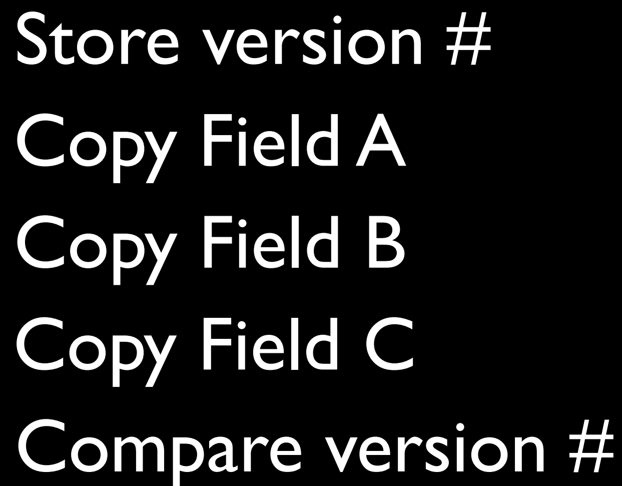
Application Thread



Transactional Copying

GC Thread

Application Thread



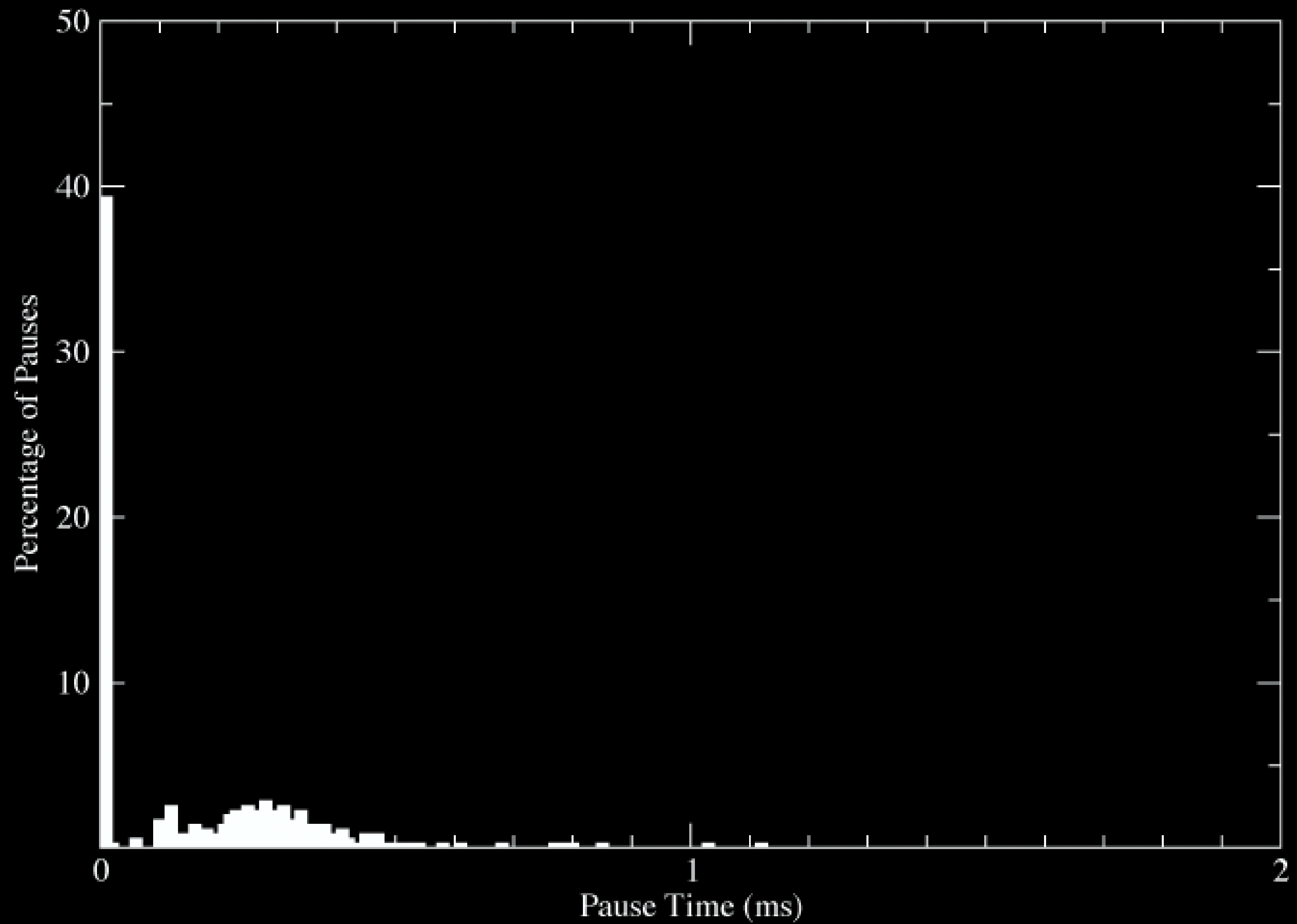
Synergy Part 2

- Both systems use read and write barriers
  - Code inserted around memory accesses
- Strong atomicity barriers:
  - Log transactional reads and writes
- Concurrent GC barriers
  - Prevents writes of unmarked pointers
  - Follow forwarding pointers

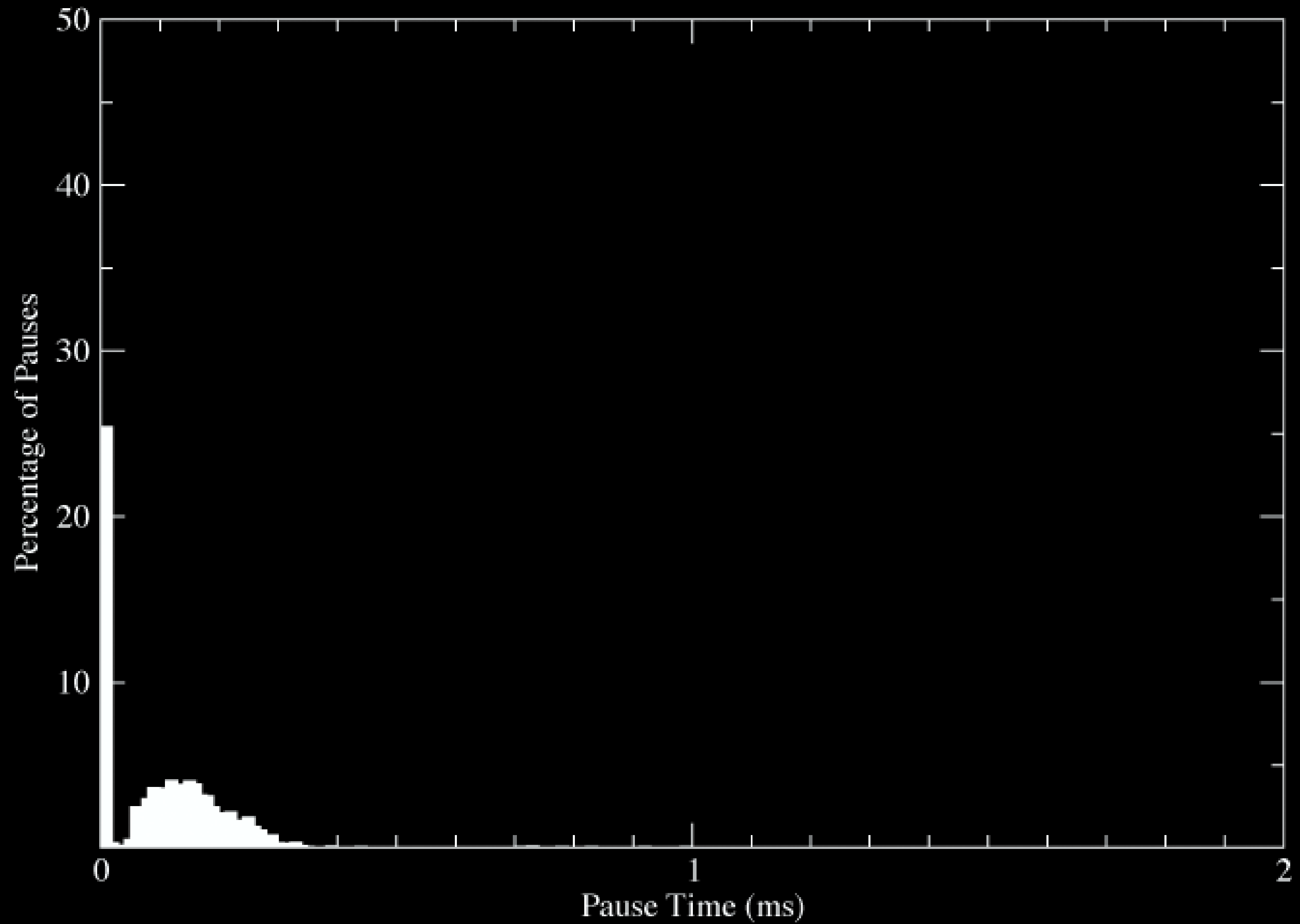
Experiments

- SPEC JVM98
- SPECjbb2000
- Atomicjbb
- AtomicTSP, AtomicOO7

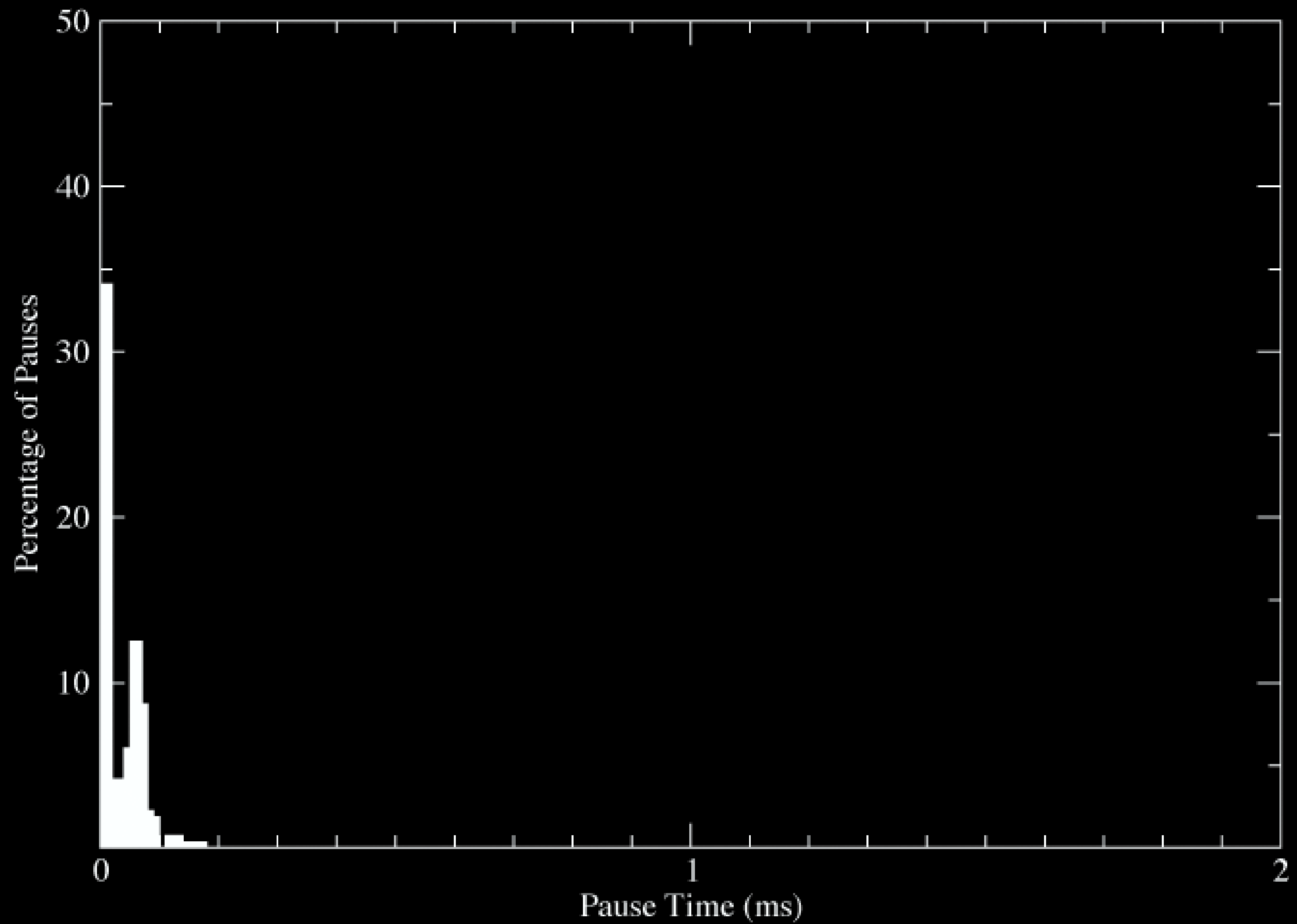
Javac



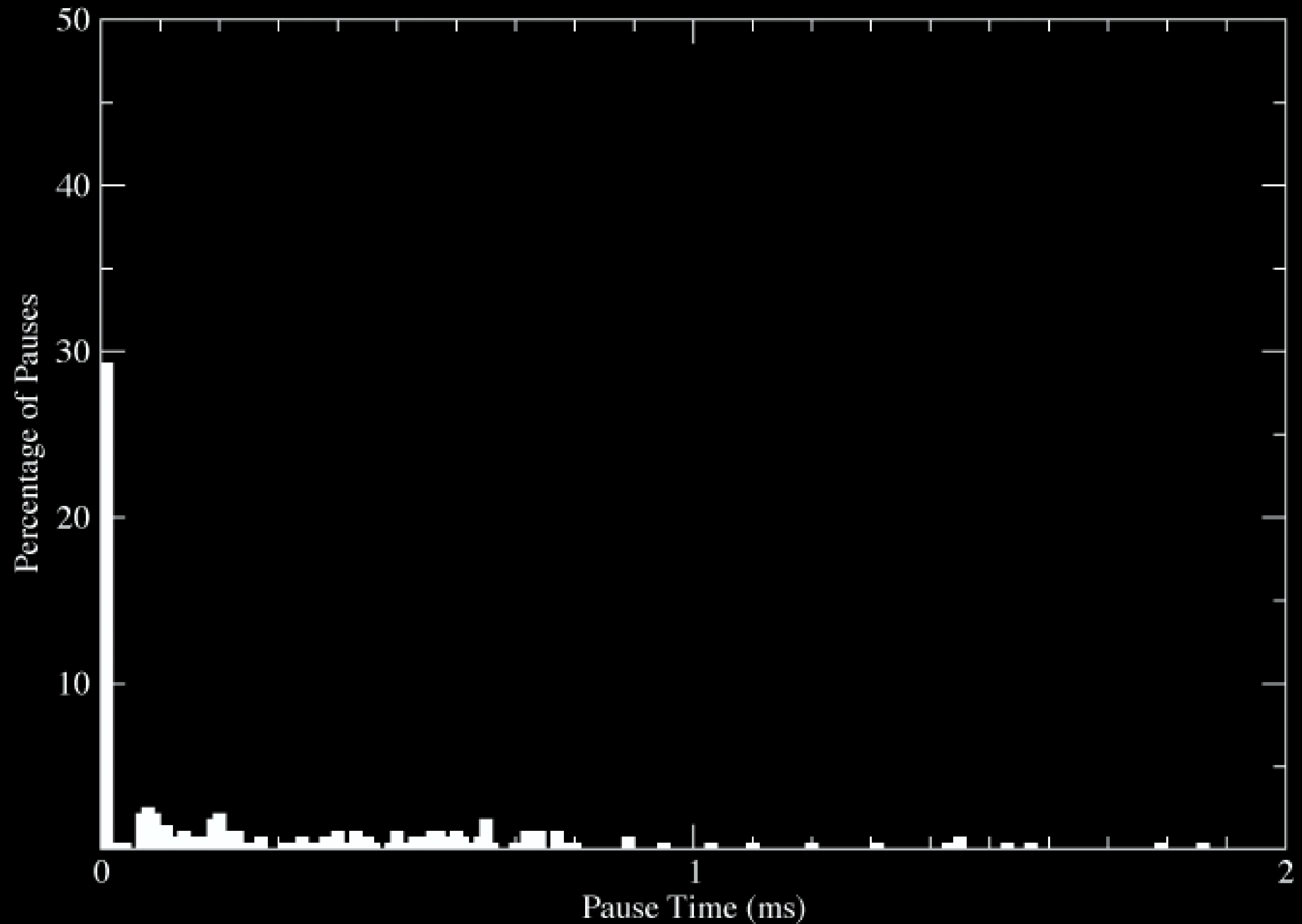
SPECjbb



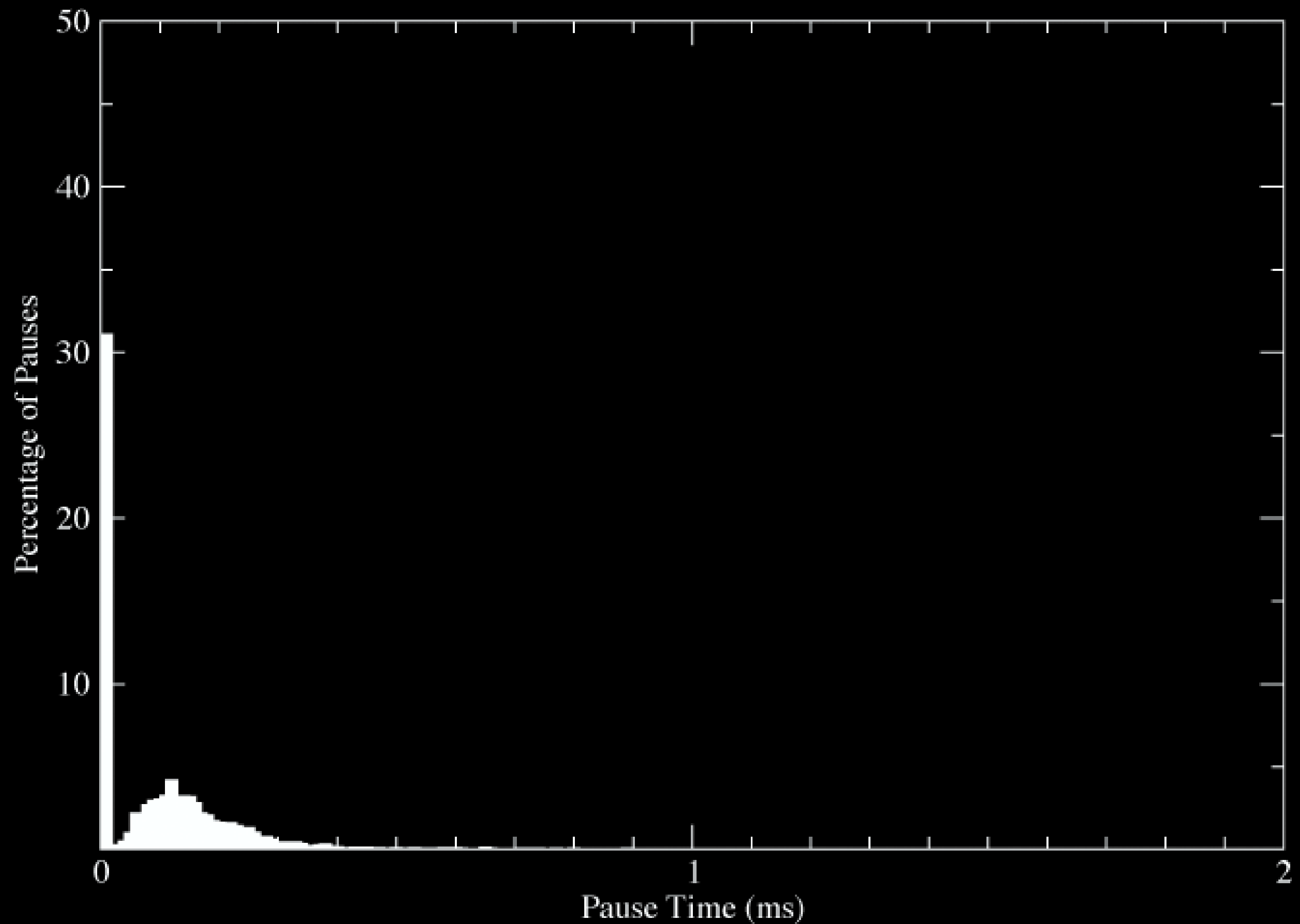
AtomicTSP



AtomicJBB

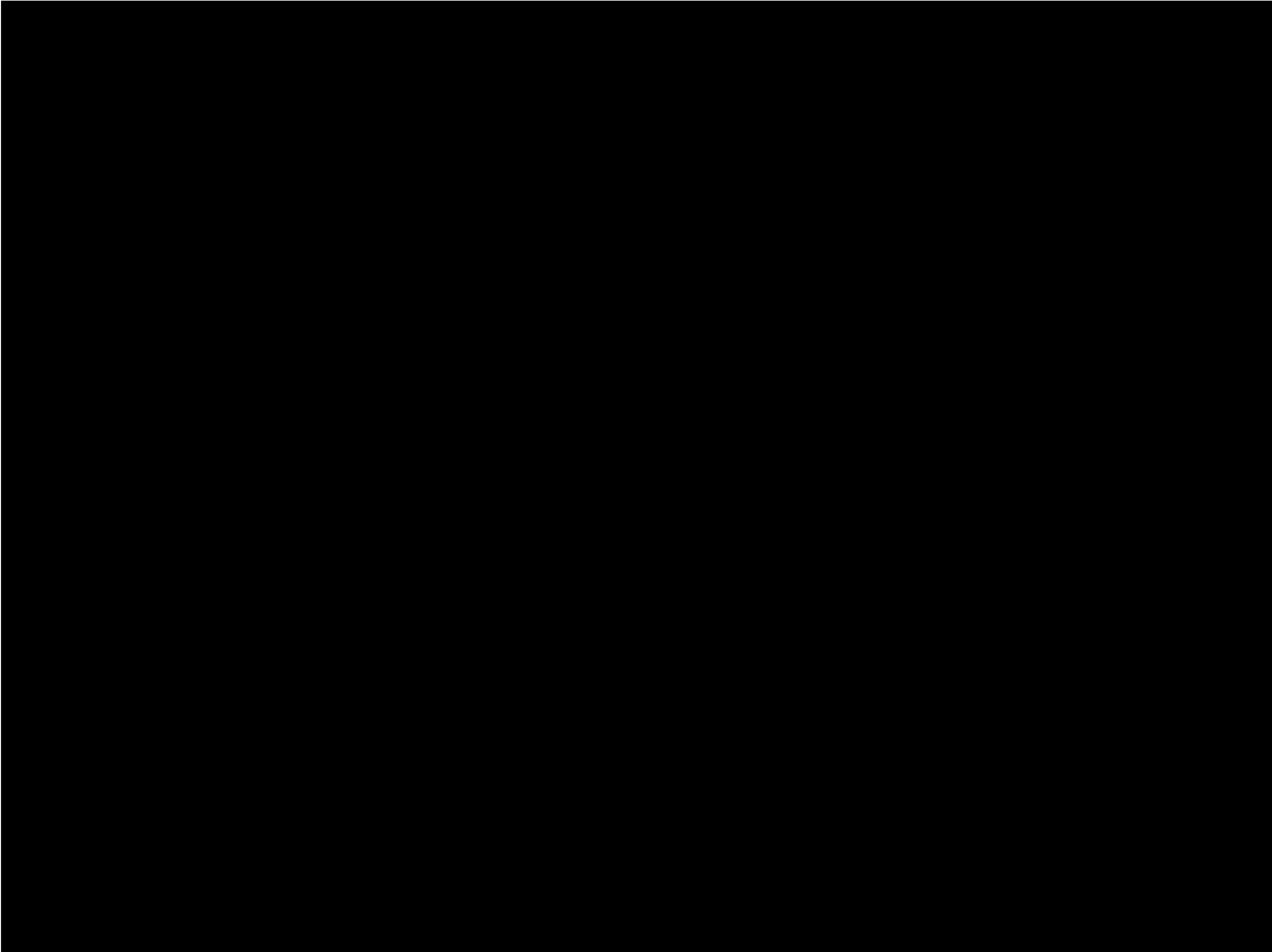


All workloads



Conclusion

- Identified overlap between GC and TM
- Leverage TM to safely copy concurrently
- Focus on pause times
  - Aim to keep 90% < 1ms
  - Result: 98.8% < 1ms, 96.9% < 0.5ms



Outliers

Benchmark	< 1ms	1..10 ms	10...100 ms	> 100 ms
201_compress	100.0%	0.00%	0.00%	0.00%
202_jess	100.0%	0.00%	0.00%	0.00%
209_db	100.0%	0.00%	0.00%	0.00%
213_javac	99.43%	0.57%	0.00%	0.00%
222_mpegaudio	100.0%	0.00%	0.00%	0.00%
227_mtrt	100.0%	0.00%	0.00%	0.00%
_228_jack	100.0%	0.00%	0.00%	0.00%
SPECjbb	99.72%	0.14%	0.14%	0.00%
AtomicOO7	100.0%	0.00%	0.00%	0.00%
AtomicTSP	100.0%	0.00%	0.00%	0.00%
Atomicjbb	85.00%	12.50%	2.14%	0.36%
Total	98.92%	0.85%	0.21%	0.02%
Target	≥ 90%	≤ 9%	≤ 0.9%	≤ 0.1%

Pauses per GC

Benchmark	Mark Phase	Flip Phase	Total
201_compress	2.0	2.0	4.0
202_jess	2.6	2.0	4.6
209_db	2.0	2.0	4.0
213_javac	2.7	2.0	4.7
222_mpegaudio	2.0	2.0	4.0
227_mtrt	3.7	2.9	6.6
_228_jack	2.1	2.0	4.1
SPECjbb	5.7	2.7	8.4
AtomicOO7	3.6	2.0	5.6
AtomicTSP	2.0	2.0	4.0
Atomicjbb	4.0	2.0	6.0
Average	2.9	2.1	5.1

Time In Each Stage

Benchmark	Mark Phase	Copy Phase	Flip Phase	Total
201_compress	0.19%	0.08%	0.26%	0.53%
202_jess	0.91%	0.37%	1.18%	2.45%
209_db	0.43%	0.14%	0.44%	1.00%
213_javac	0.82%	0.17%	0.82%	1.81%
222_mpegaudio	0.22%	0.08%	0.27%	0.57%
227_mtrt	1.81%	0.61%	2.02%	4.44%
_228_jack	0.77%	0.36%	0.58%	1.71%
SPECjbb	1.27%	0.27%	1.02%	2.56%
AtomicOO7	0.04%	0.01%	0.03%	0.08%
AtomicTSP	0.51%	0.00%	0.00%	0.51%
Atomicjbb	1.37%	0.52%	2.39%	4.28%
Average	0.76%	0.24%	0.82%	1.81%