Concurrent Copying Garbage Collection

Filip Pizlo, Erez Petrank, Bjarne Steensgaard
Purdue, Technion/Microsoft, Microsoft

PLDI'08 - Tucson, AZ

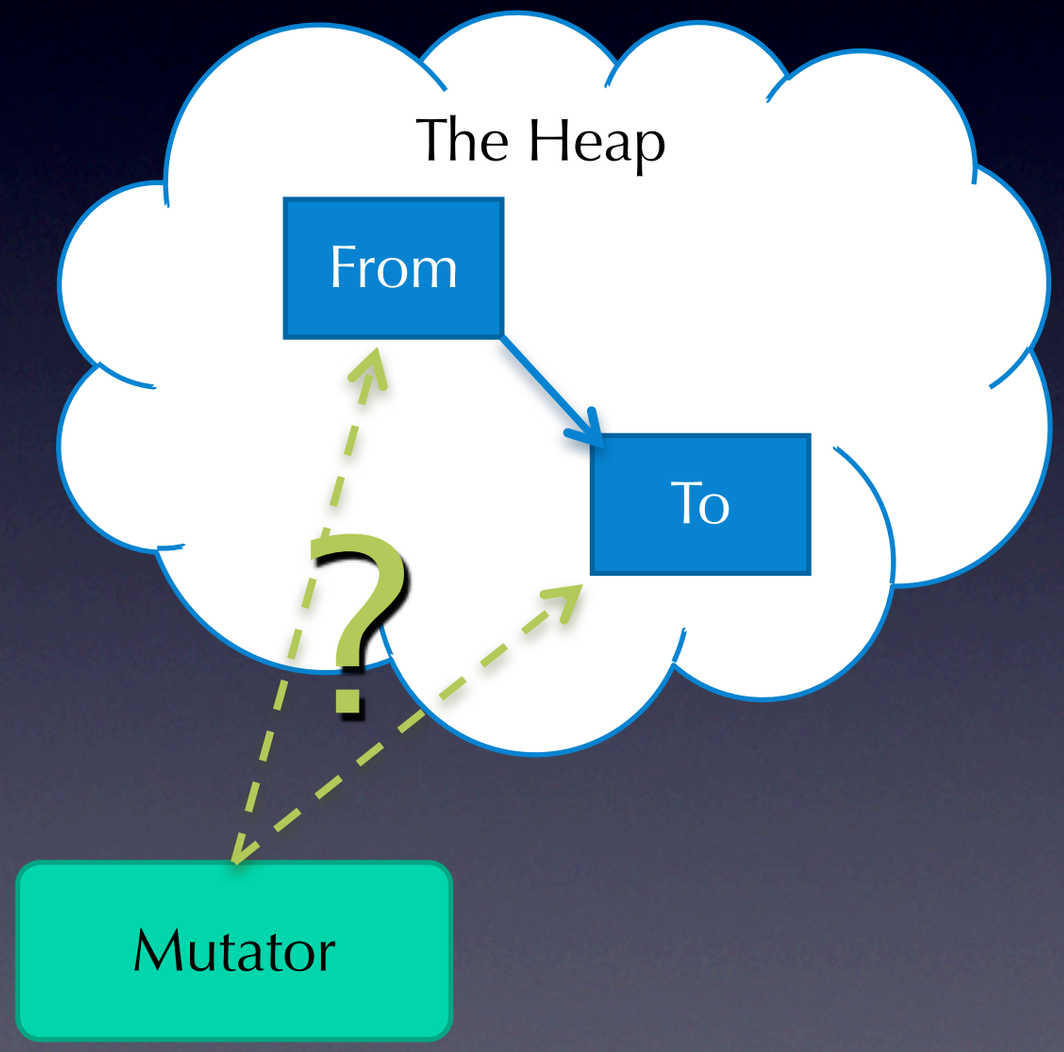
Introduction

- RTGC is gaining acceptance as an alternative to manual memory management for RT applications
- But:
  - Multiprocessor support is problematic
  - ... especially if defragmentation is required.

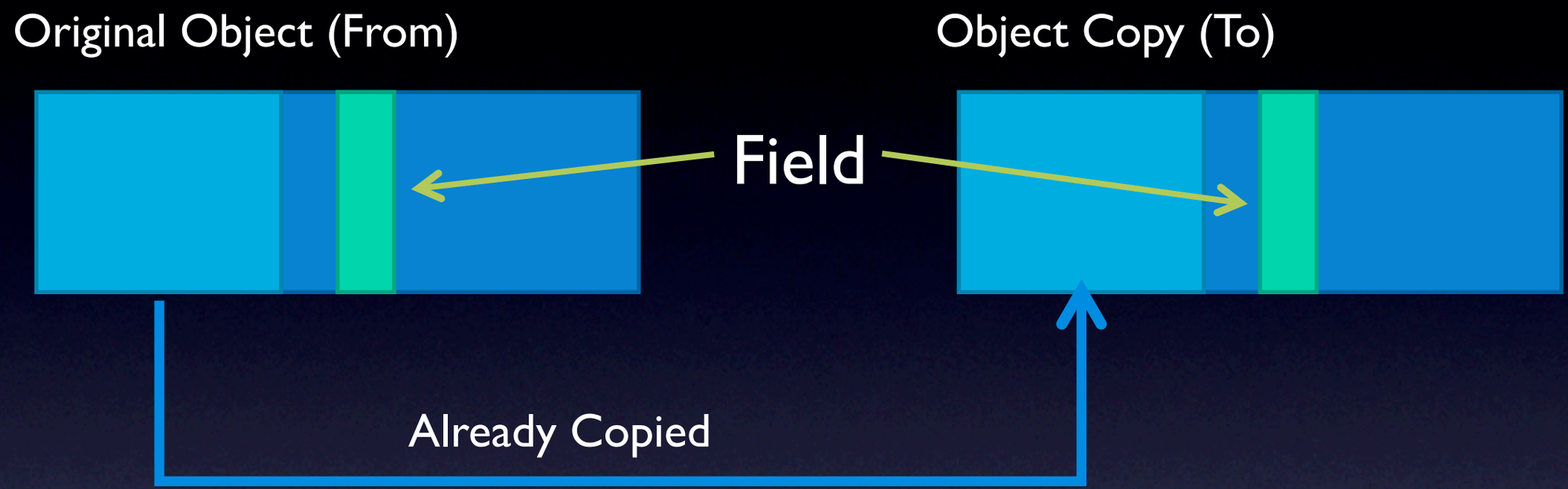
- What we deliver:
  - Compaction.
  - Concurrency.
  - Lock freedom.
  - Efficiency.

Why is it hard?

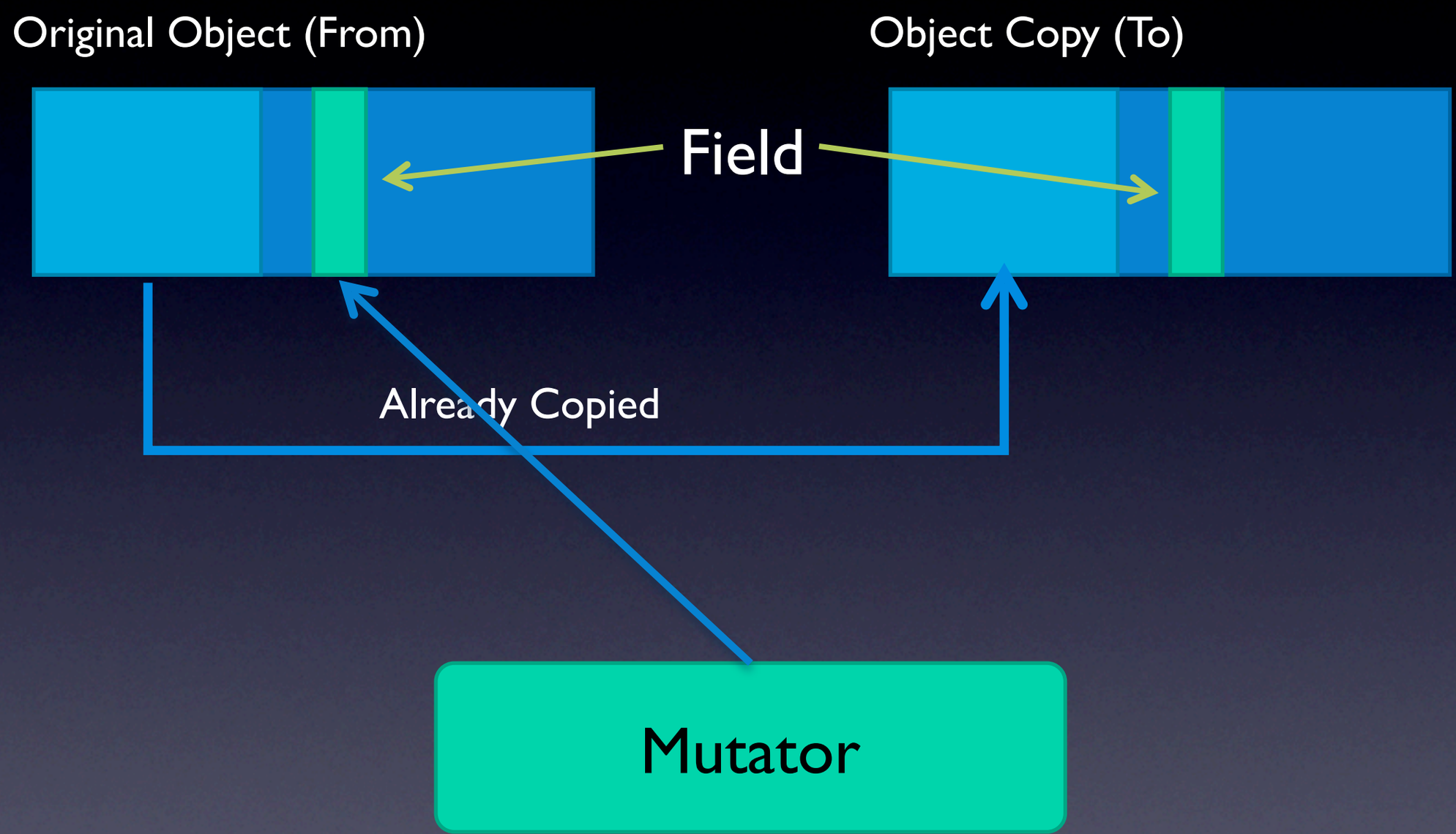
- At some point during defragmentation there will be two copies of the same object.
- Then: which version of the object should the mutator access?

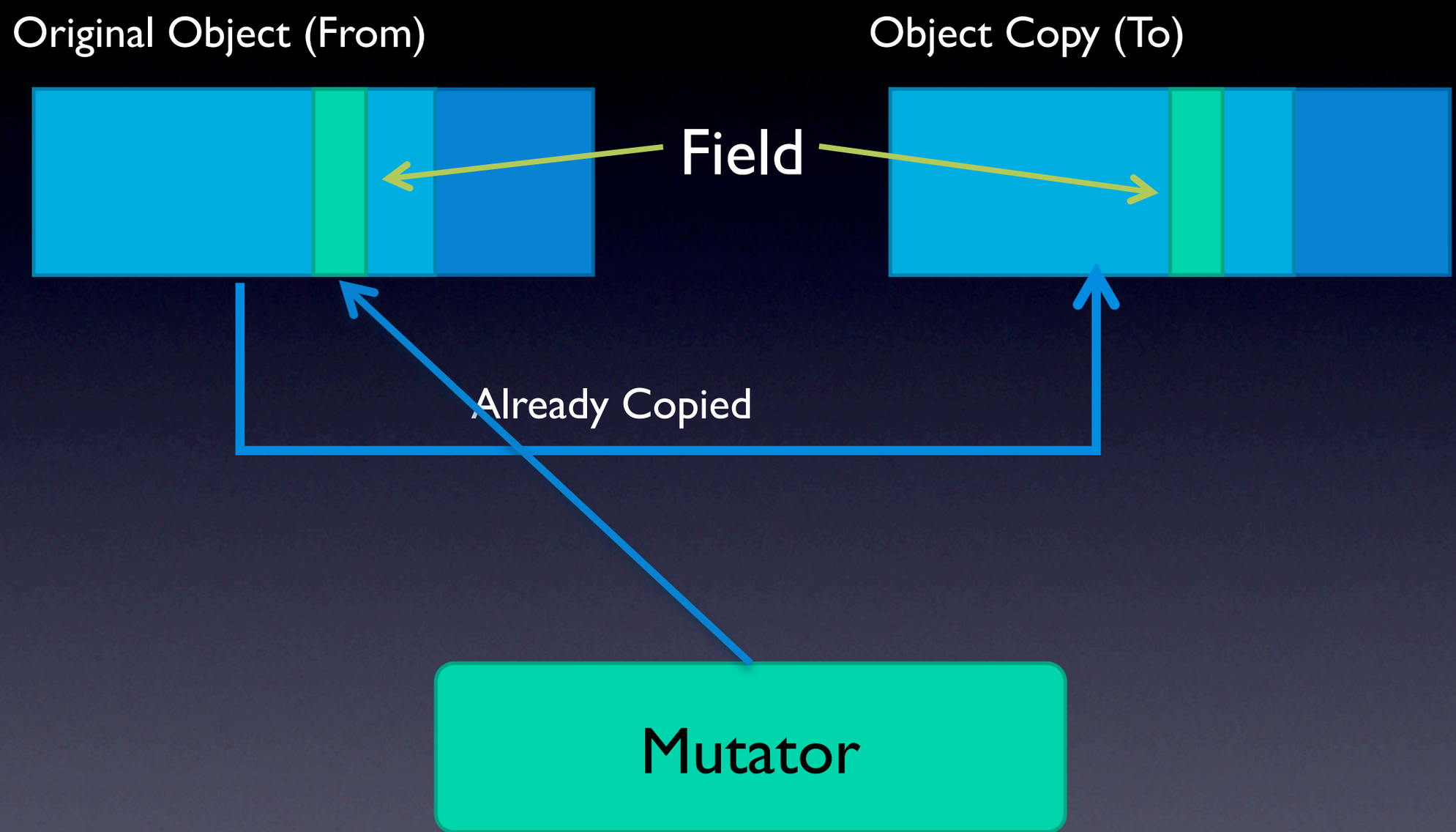


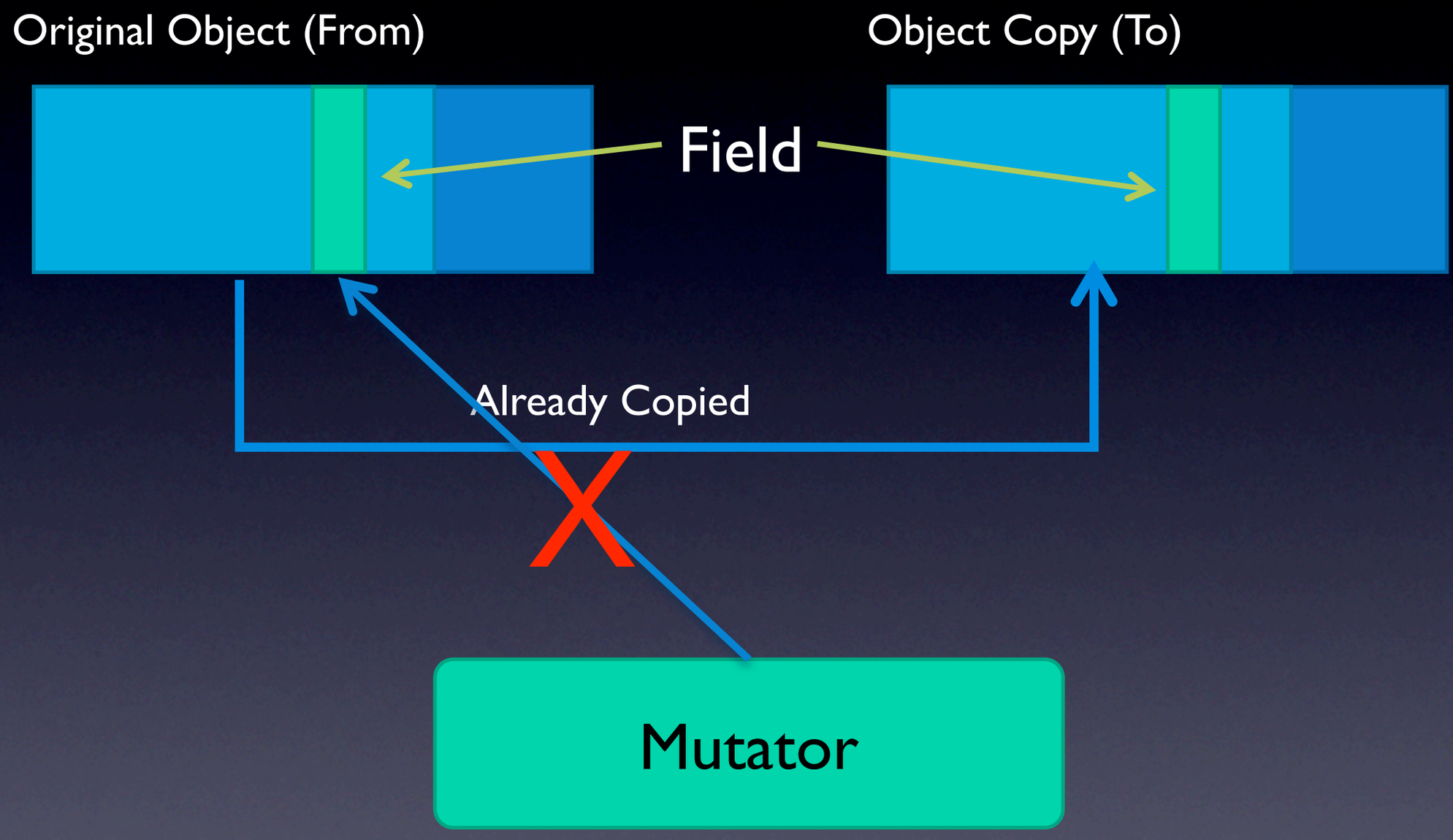
Mutator

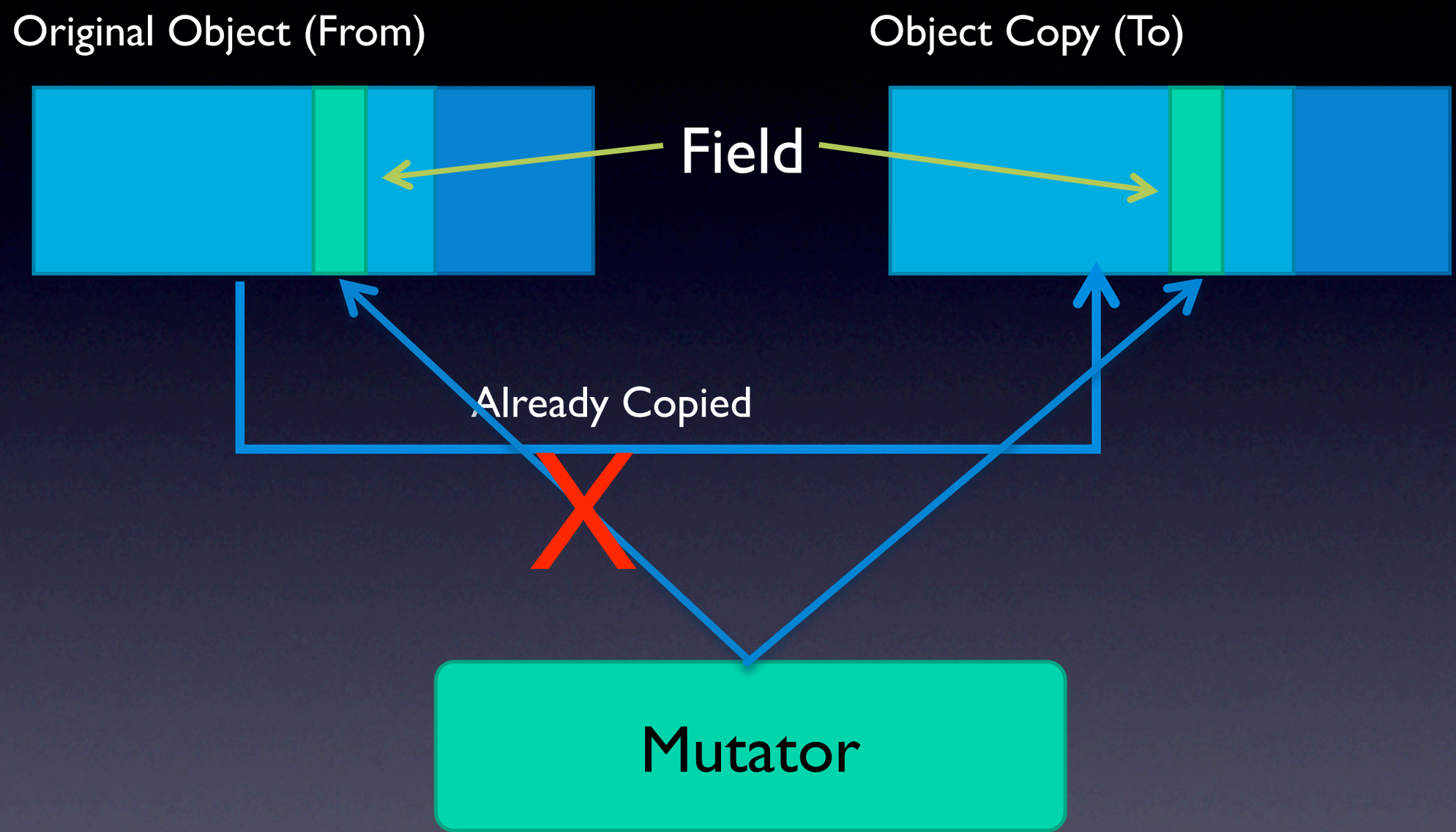


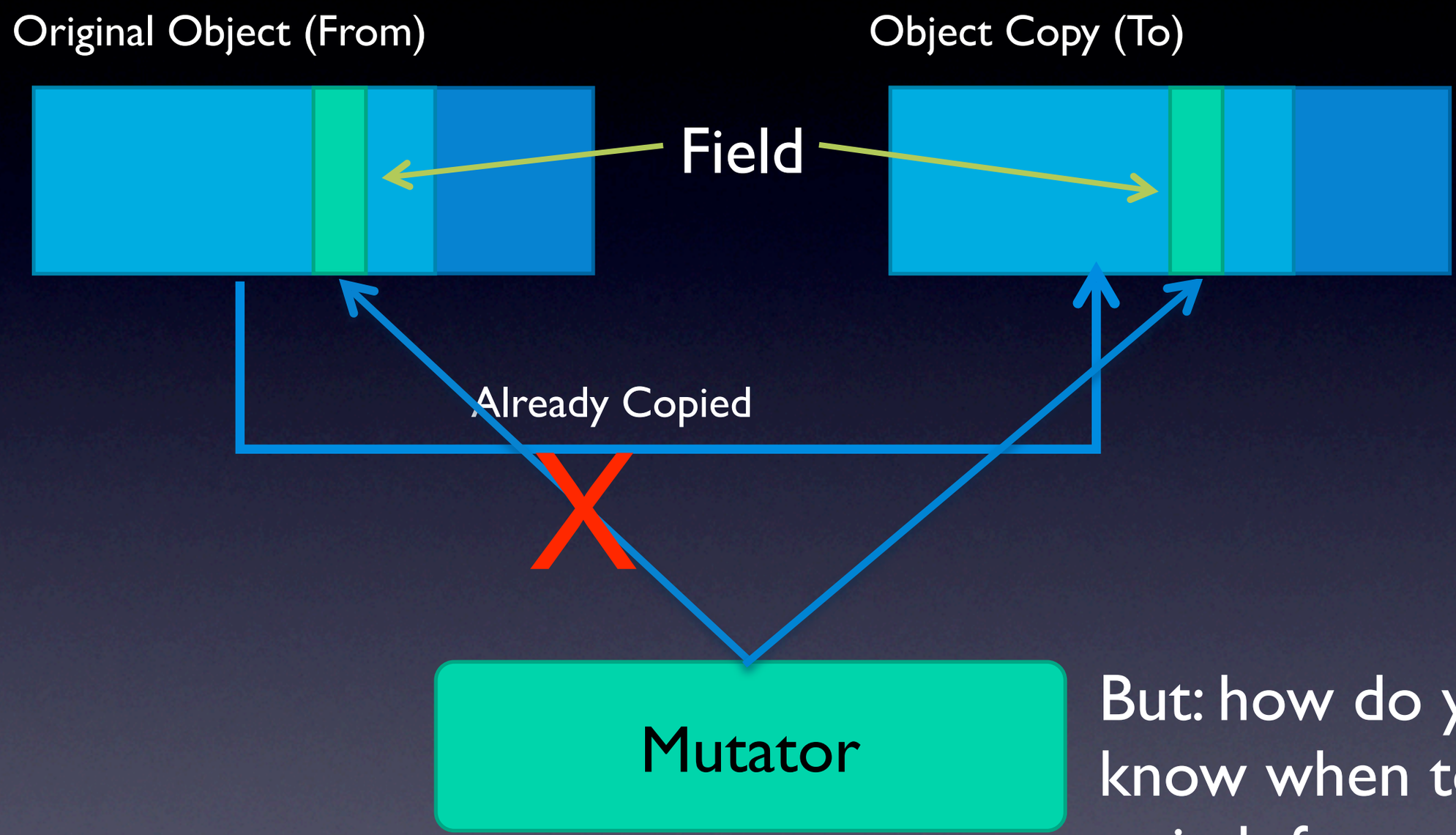
Mutator


Original Object (From)
Object Copy (To)
Field
Already Copied
Mutator

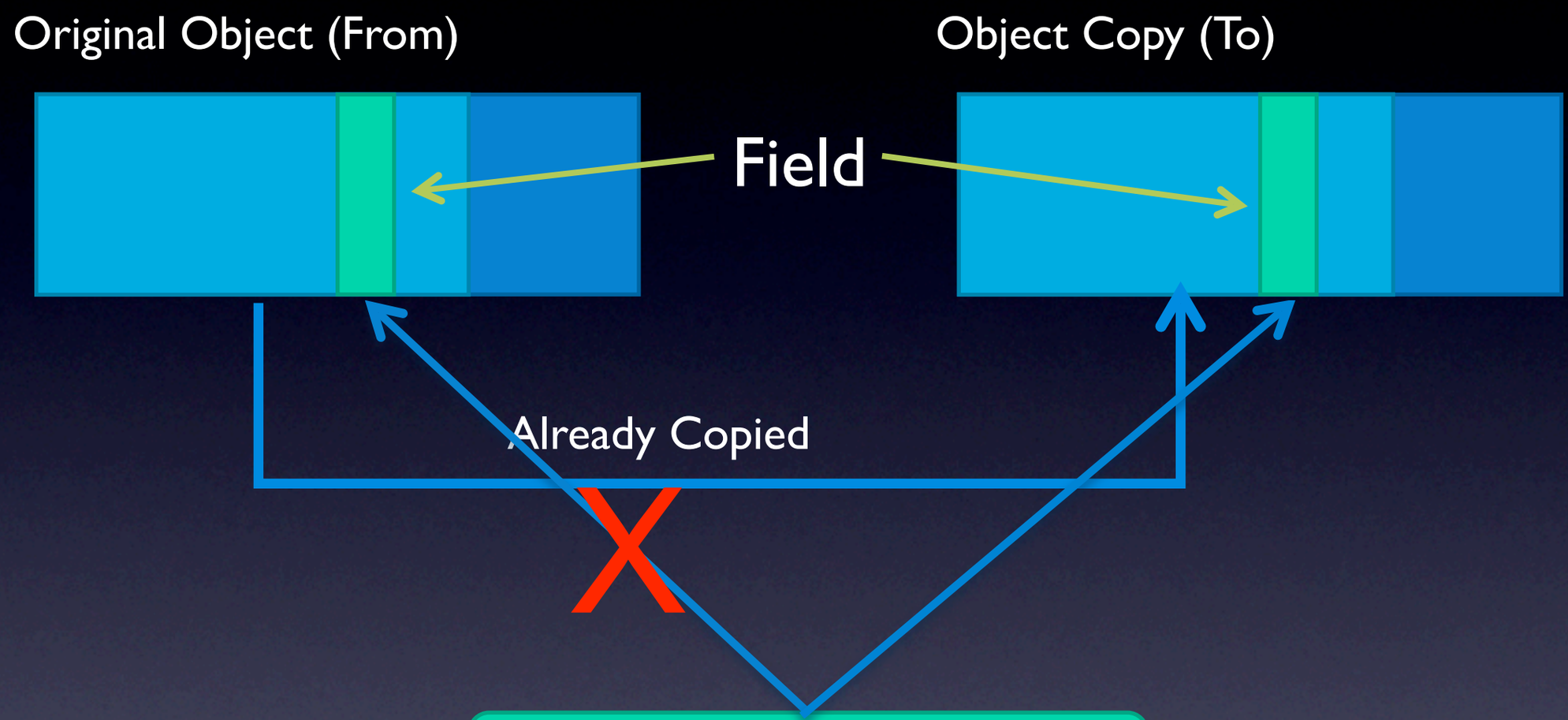

Original Object (From)
Object Copy (To)
Field
Already Copied
Mutator


Original Object (From)
Object Copy (To)
Field
Already Copied
Mutator


Original Object (From)
Object Copy (To)
Field
Already Copied
X
Mutator



But: how do you know when to switch from the original to the to-space object?



Immediately after you check which version of the field to use, the copier may advance past it.

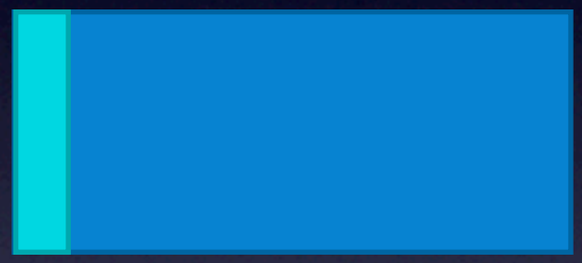
- Previous techniques:
  - Hudson & Moss ’01, Cheng & Blelloch ‘01
  - Stopless (Pizlo et al ‘07)
- Our New Techniques:
  - Chicken
  - Clover

- Chicken:
  - Really fast
  - Does not guarantee that all objects are copied
- Clover:
  - Probabilistic!
  - Guarantees that all objects get copied

- Both Chicken and Clover are simple to implement
  - (simpler, we argue, than any previous proposed concurrent copying technique).
- Both Chicken and Clover preserve the underlying hardware memory model - no JMM tricks are necessary.

Chicken

- Design Principles:
  - Use the cheapest barriers possible.
  - Don’t guarantee that objects tagged for copying will actually be copied.
  - Anytime the mutator writes to an object as it is being copied, abort the copying of the respective object.



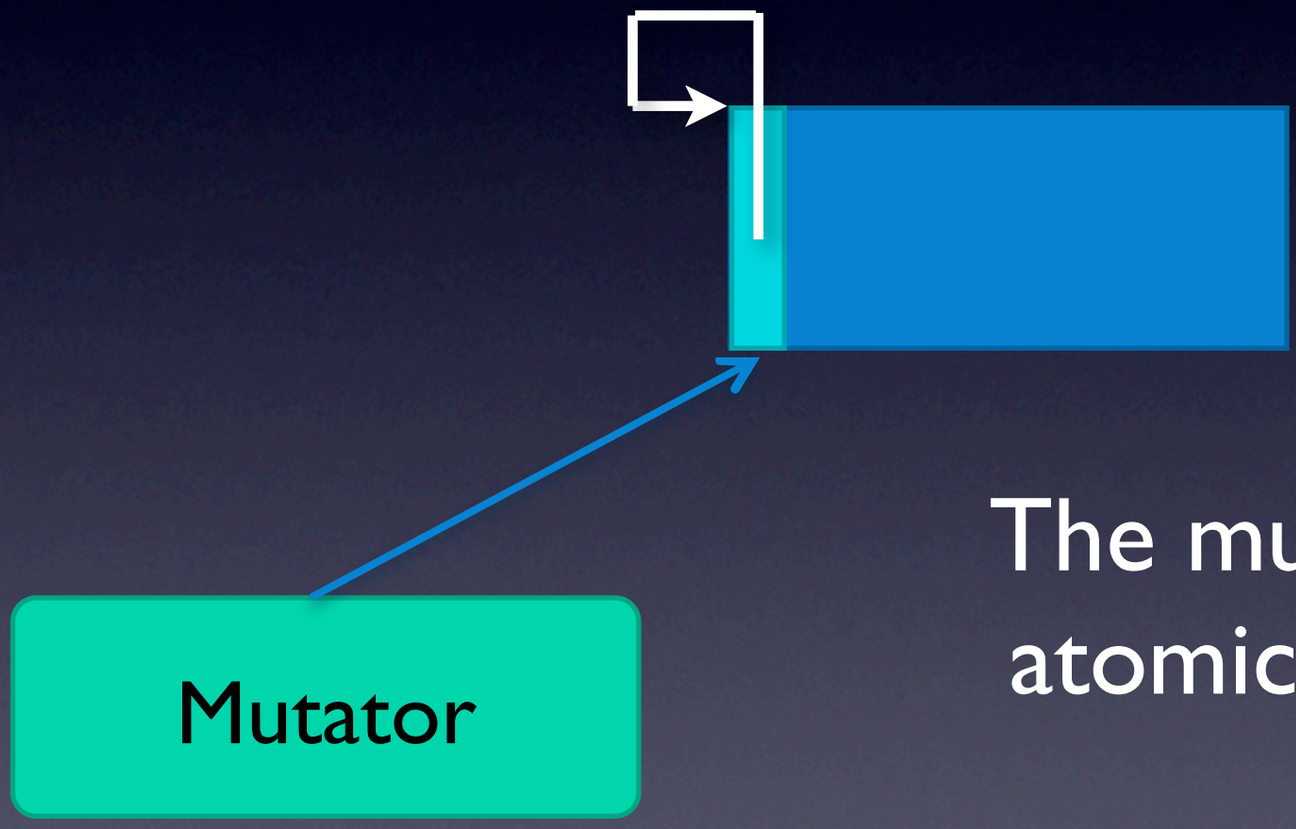
Use a Brooks-style forwarding pointer

Use a Brooks-style forwarding pointer

To copy the object, first "tag" the forwarding pointer (set a low order bit)

Use a Brooks-style forwarding pointer

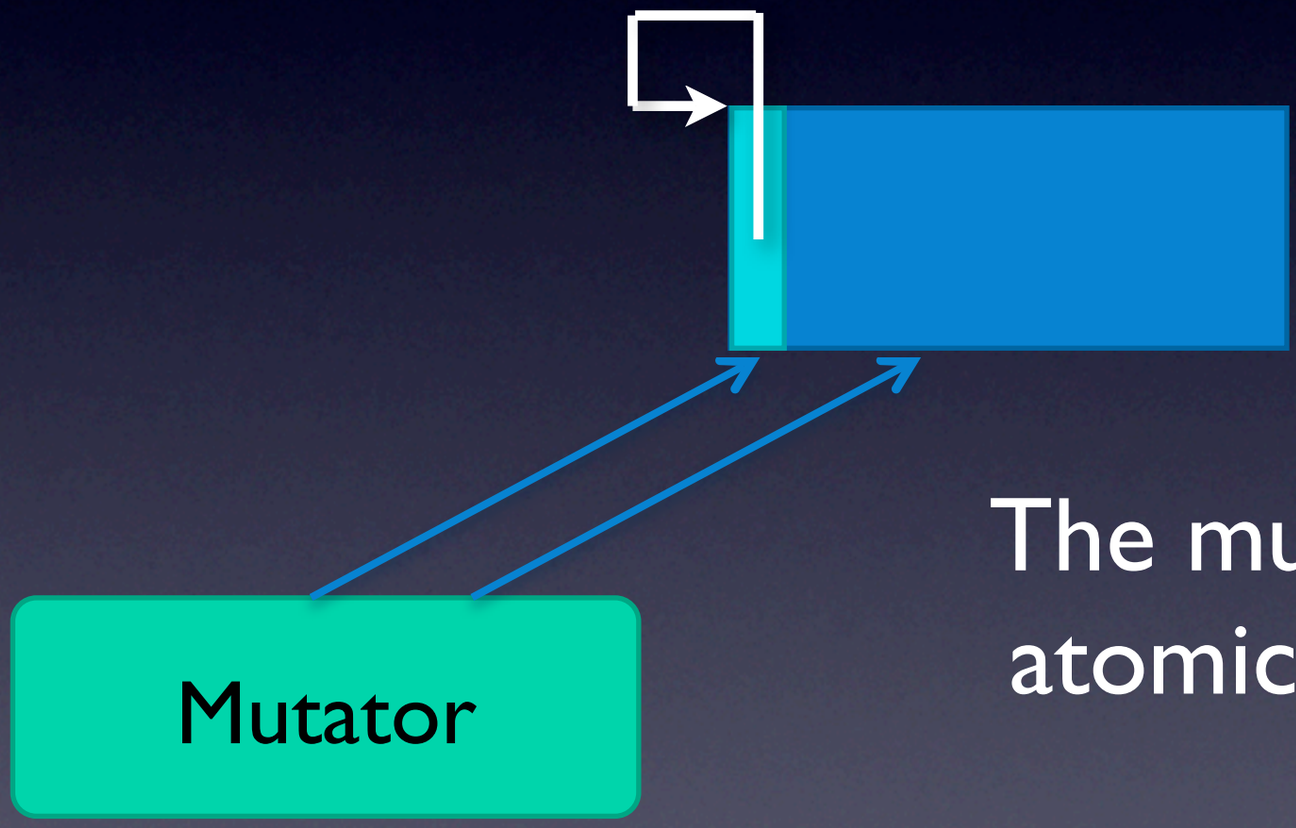
To copy the object, first "tag" the forwarding pointer (set a low order bit)



The mutator writes by first atomically clearing the tag.

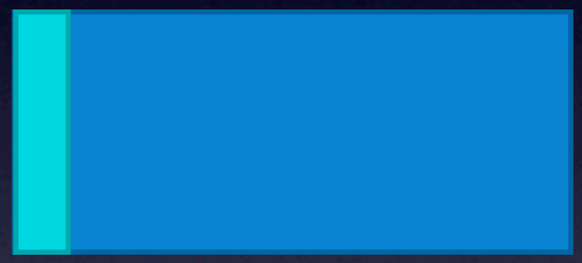
Use a Brooks-style forwarding pointer

To copy the object, first "tag" the forwarding pointer (set a low order bit)

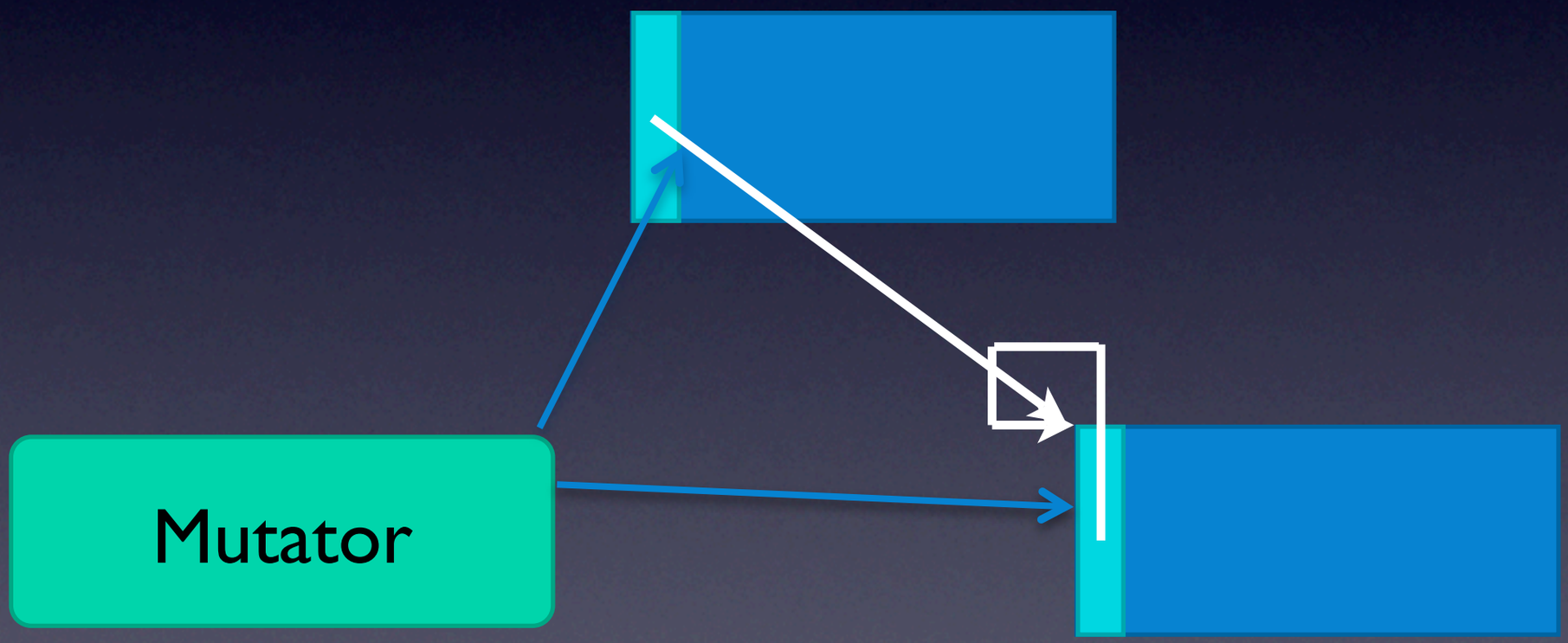


The mutator writes by first atomically clearing the tag.

... and then performing the write



If the object is already copied, the mutator writes to the new object via the forwarding pointer



Write barrier

```
write(object, offset, value) {
    if object is tagged
        CAS(object.forward, tagged → untagged)
    object.forward[offset] = value
}
```

Write barrier

Clears the tag bit that we stole from the Brooks forwarding pointer

```
write(object, offset, value) {
    if object is tagged
        CAS(object.forward, tagged → untagged)
    object.forward[offset] = value
}
```

Write barrier

Clears the tag bit that we stole from the Brooks forwarding pointer

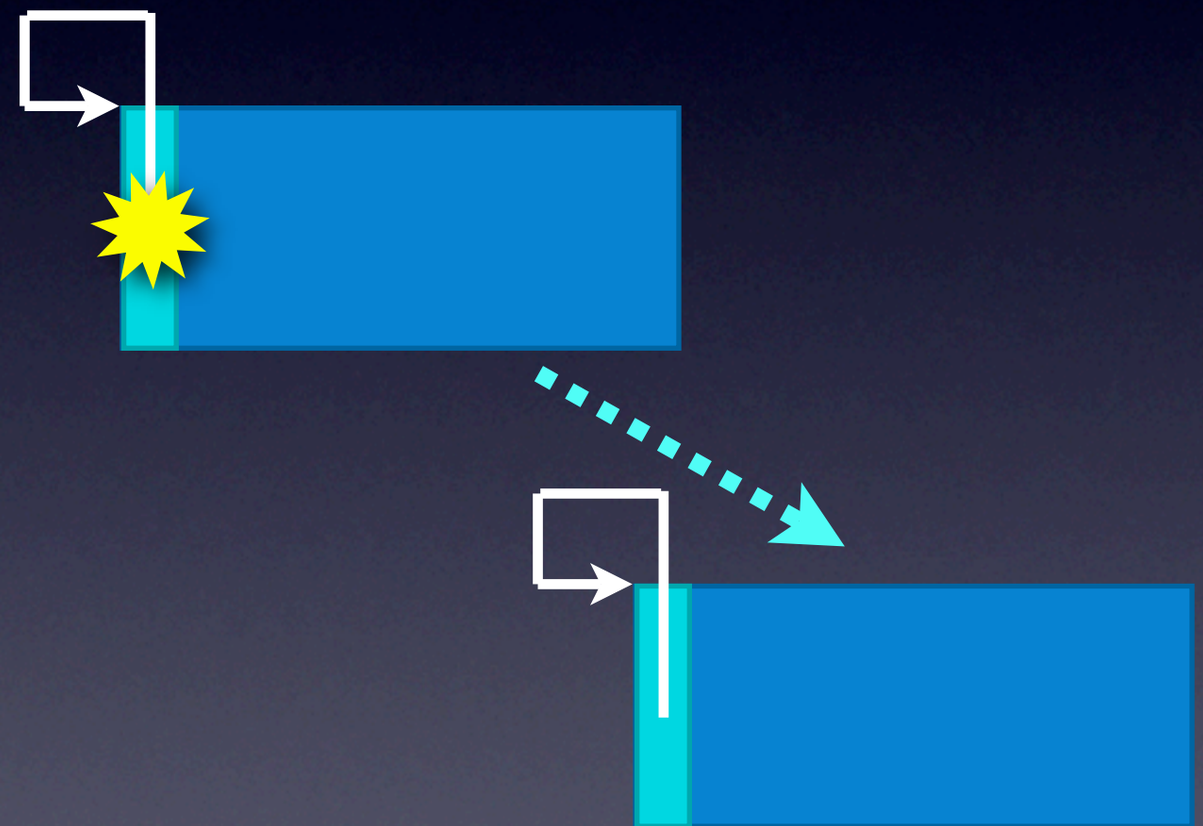
```
write(object, offset, value) {
    if object is tagged
        CAS(object.forward, tagged → untagged)
    object.forward[offset] = value
}
```

Writes to the field via the Brooks forwarding pointer

The collector starts by tagging objects that it wishes to copy.

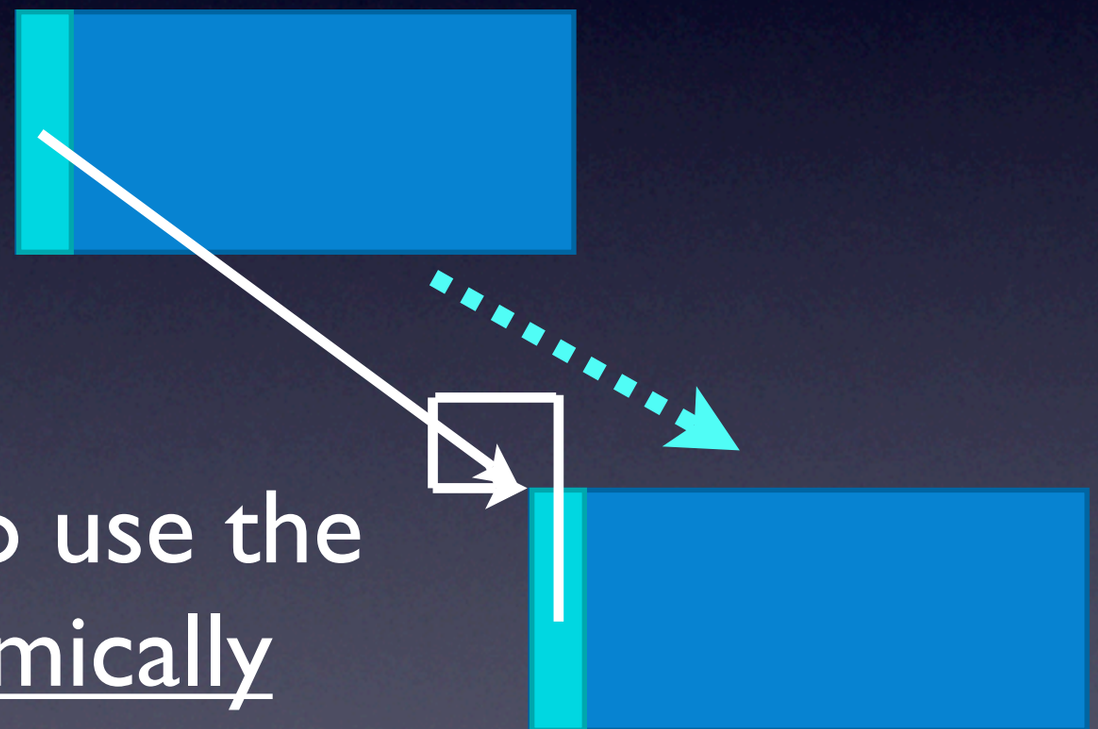
The collector starts by tagging objects that it wishes to copy.

The object is then copied.



The collector starts by tagging objects that it wishes to copy.

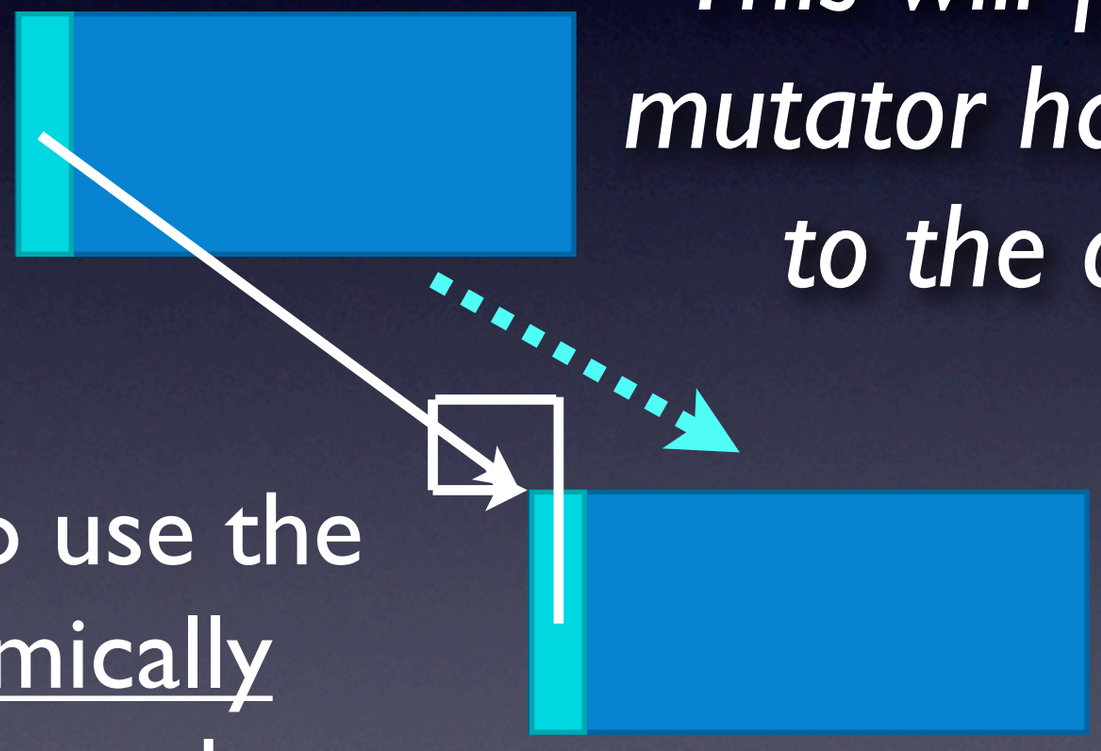
The object is then copied.



To get the mutator to use the new object, we atomically remove the tag and set the forwarding pointer.

The collector starts by tagging objects that it wishes to copy.

The object is then copied.



This will fail, if the mutator had written to the object!

To get the mutator to use the new object, we <u>atomically</u> remove the tag and set the forwarding pointer.

- Why this is good:
  - Read barrier is a wait-free Brooks barrier
  - Write barrier is a branch on the fast path, and a branch+CAS on the slow path (either way it's wait-free)
- Copying is simple and fast
- In practice only ~1% of object copying gets aborted.
  - Abort rates can be easily reduced (see paper).

- Things that could be improved:

- Things that could be improved:
  - Eliminate object copy abort entirely.

- Things that could be improved:
  - Eliminate object copy abort entirely.
- Segue into Clover...

Clover

Clover

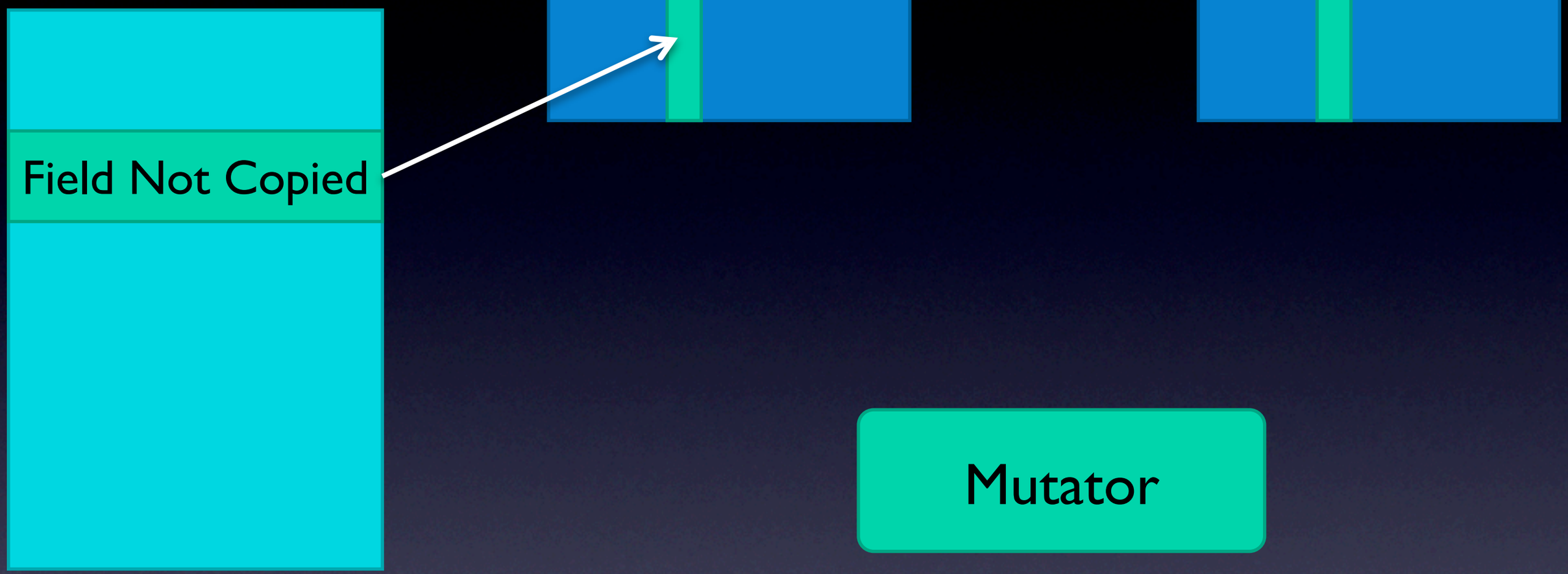
- What if each field had a status field that indicated, if the field was copied?
- And what if - you could CAS the field's value, as well as the status field, in one atomic, lock-free operation?

Status

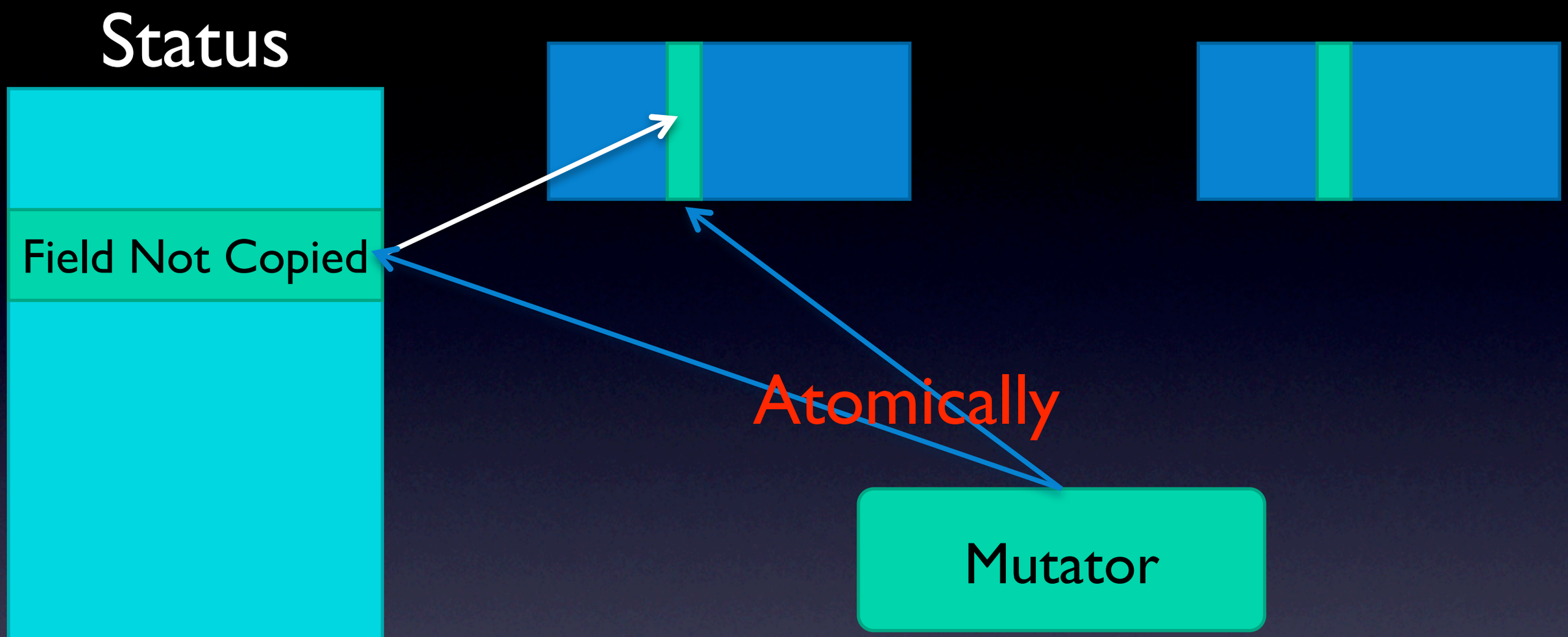
Field Not Copied

Mutator

Status

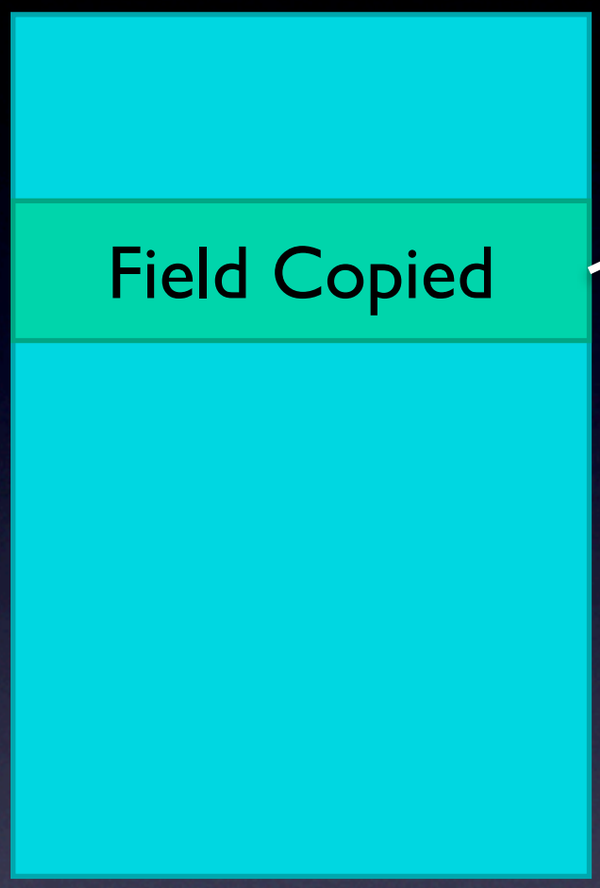


The idea is to allow the mutator to always write to the original object, and to have such writes force the collector to recopy the field at a later time.

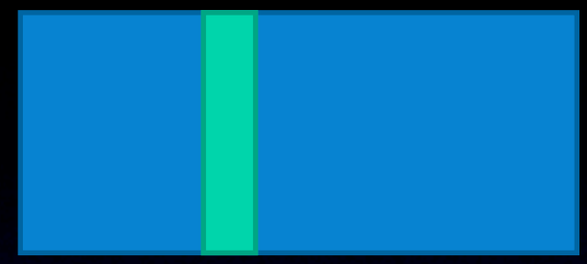


The idea is to allow the mutator to always write to the original object, and to have such writes force the collector to recopy the field at a later time.

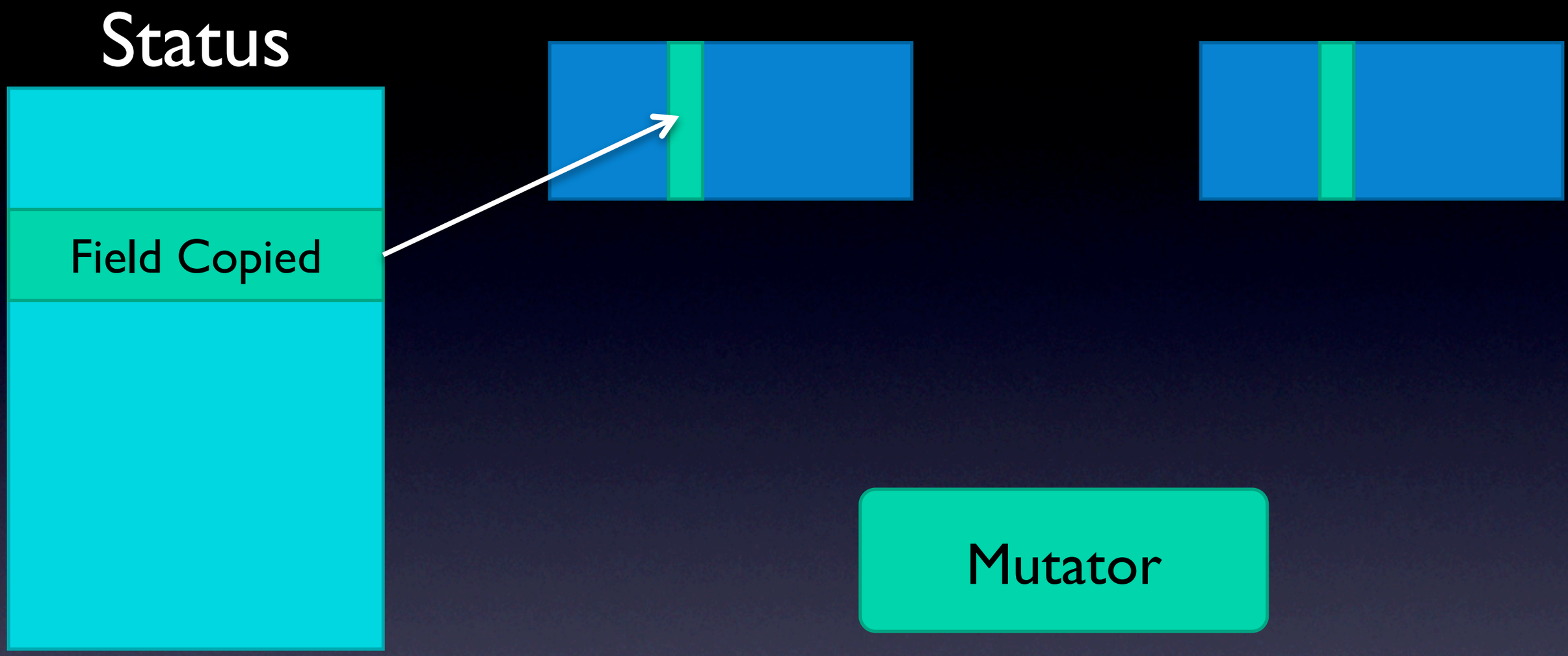
Status



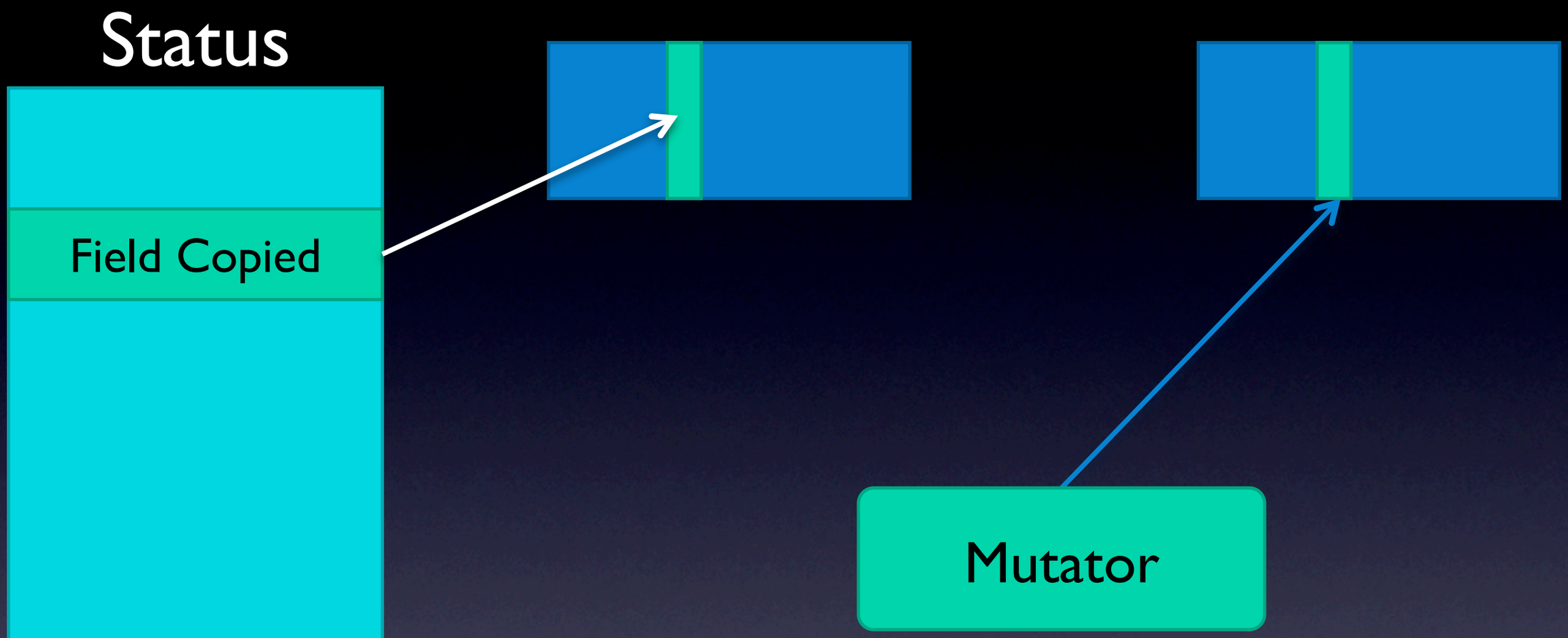
Field Copied



Mutator



If the field is already copied, access to-space.



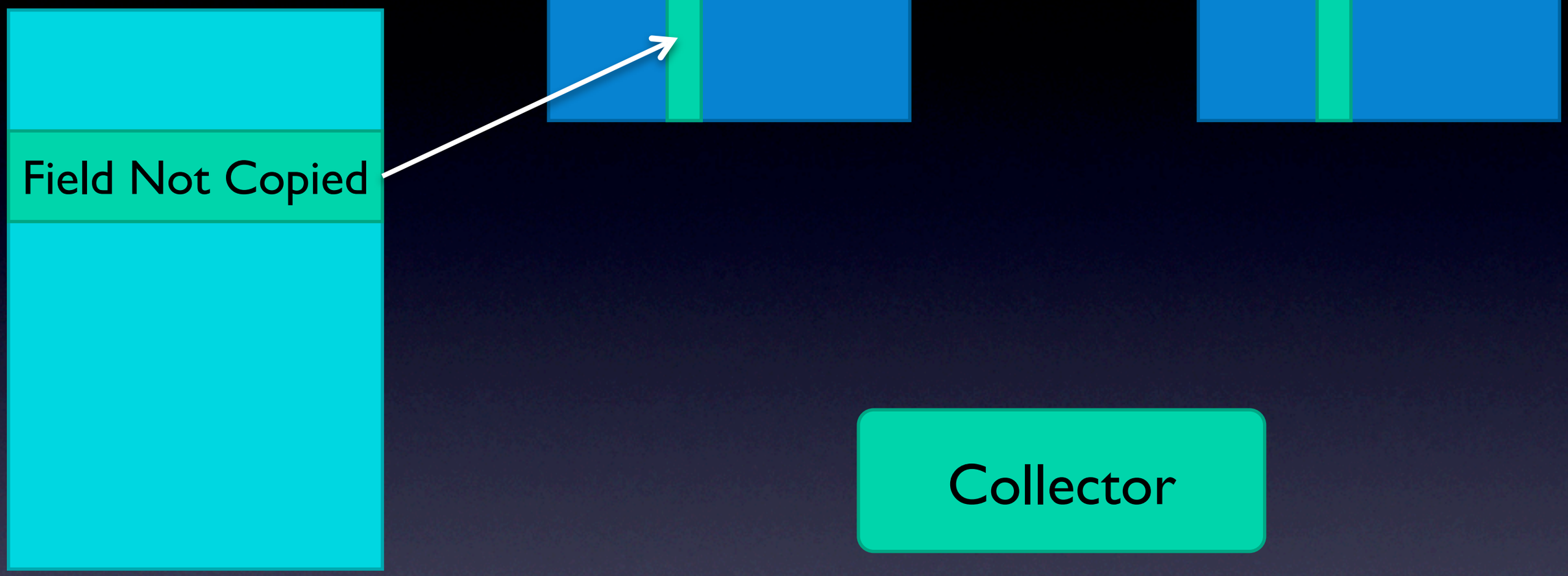
If the field is already copied, access to-space.

Status

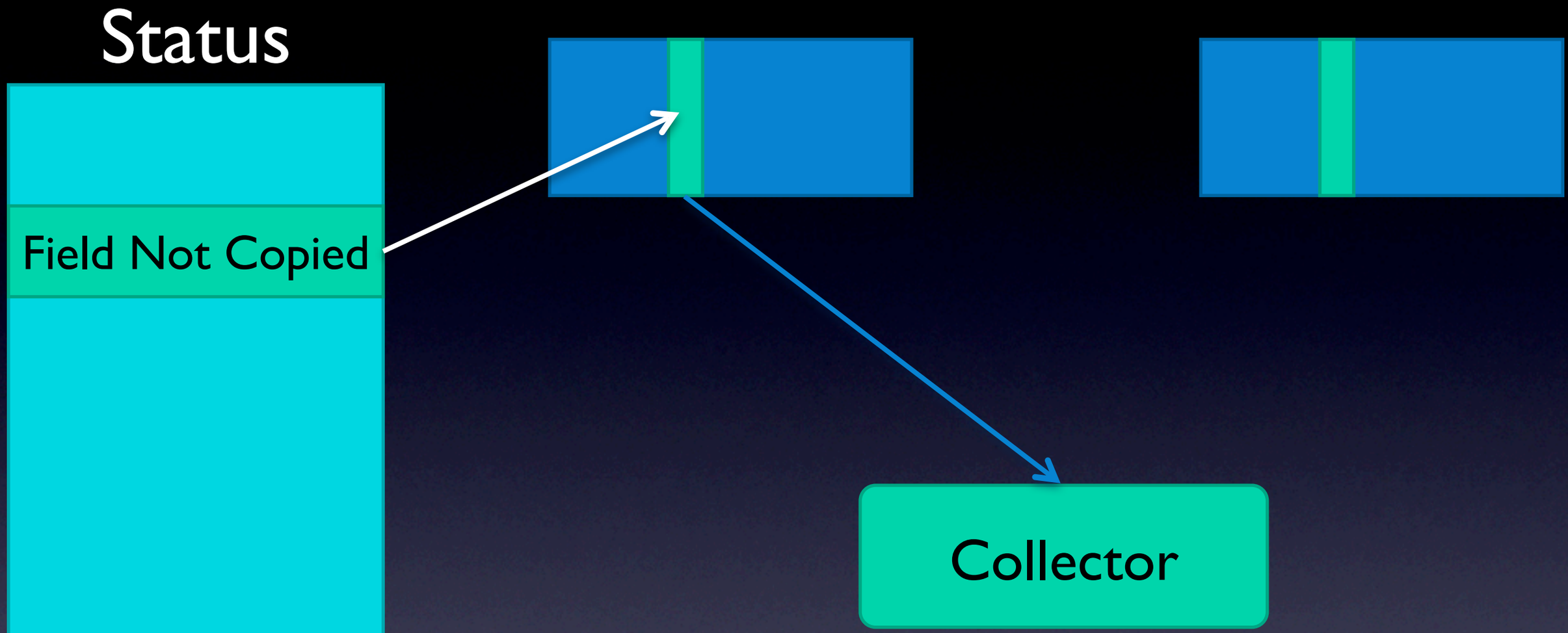
Field Not Copied

Collector

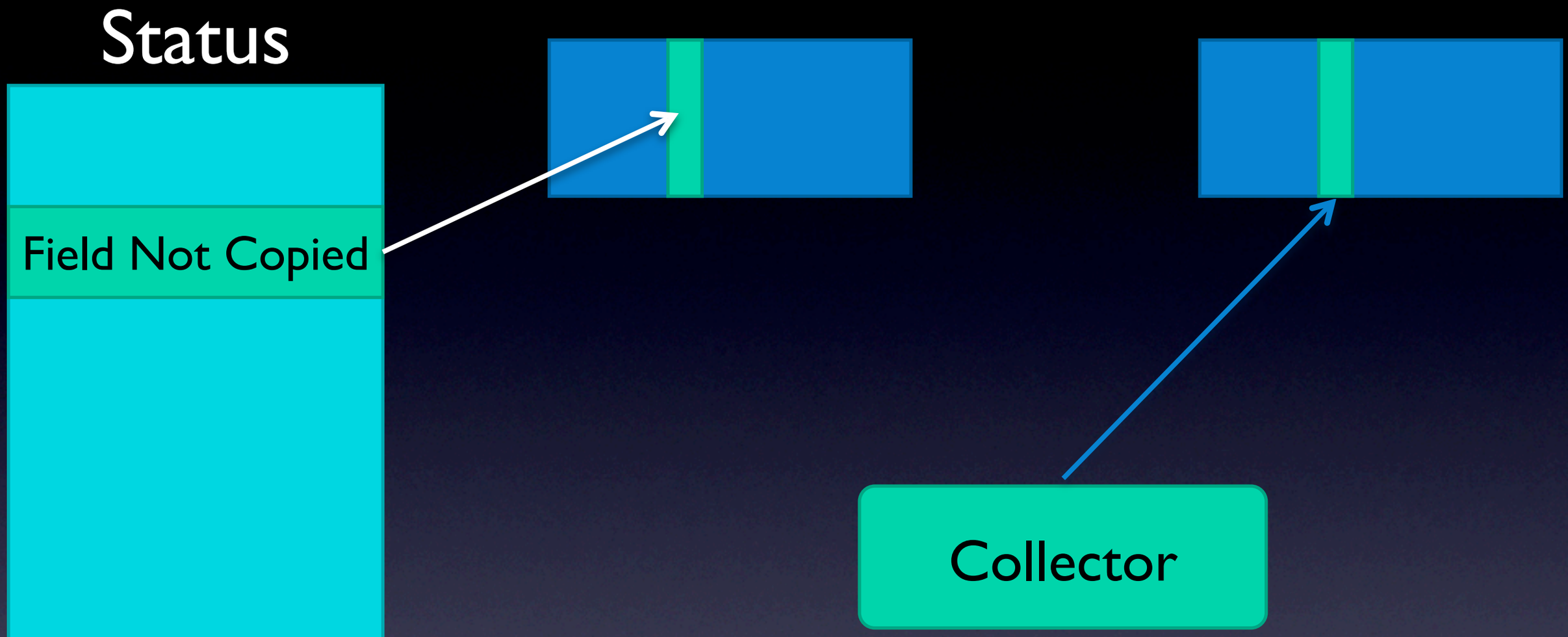
Status



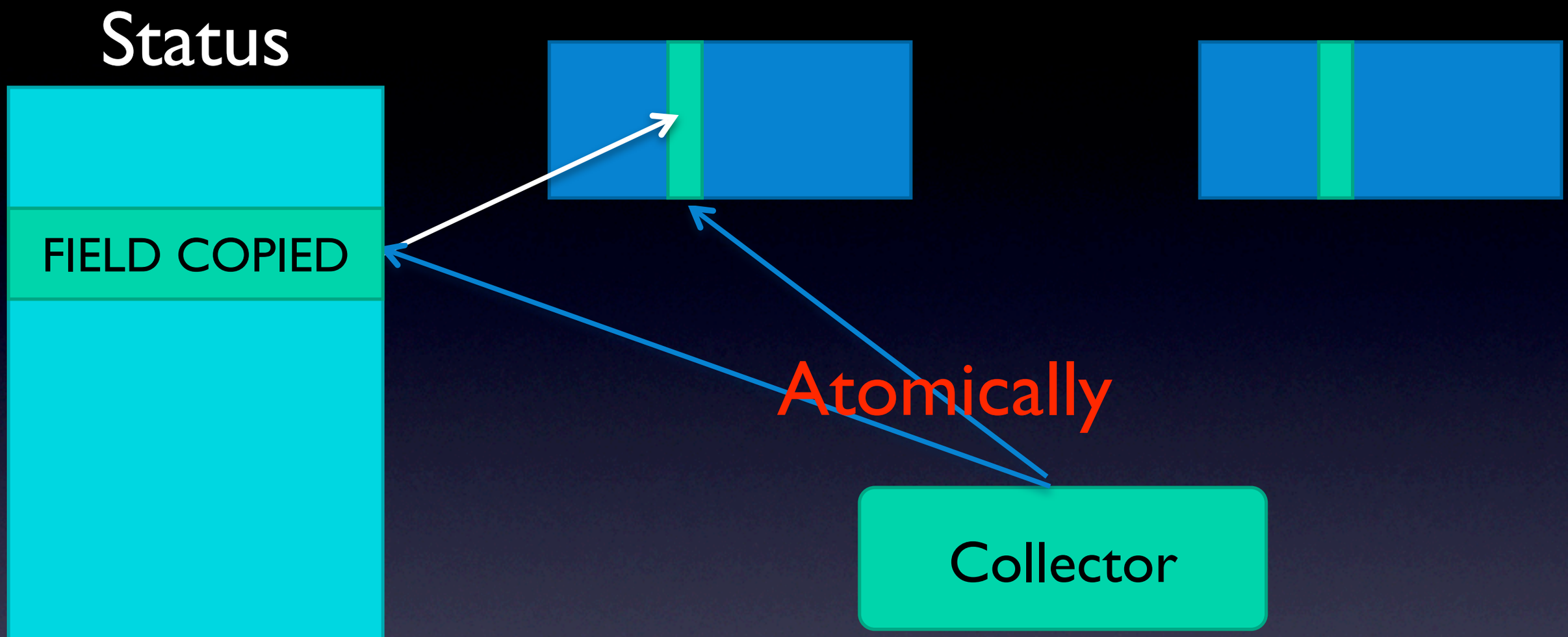
Collector repeatedly attempts to copy and assert the field as copied until it does so without the field's value changing.



Collector repeatedly attempts to copy and assert the field as copied until it does so without the field's value changing.



Collector repeatedly attempts to copy and assert the field as copied until it does so without the field's value changing.



Collector repeatedly attempts to copy and assert the field as copied until it does so without the field's value changing.

Problem: cannot CAS two *separate* fields in hardware

If you could steal a bit in the field, this would be easy...

But where do you get the bit?

Easy for reference fields - but really hard for integer fields!

Use a random number!

I.e. we steal 2^{-128} bits!

Let R = random bits

R can be huge - it can be the largest CAS-able word - 128 bits on Intel!

- The random number is used to mark fields as copied.
- This is correct, if the mutator does not use R.
- But R is selected at random, independently of the program - with R having 128 bits, the probability of "failure" is 2^{-128}.

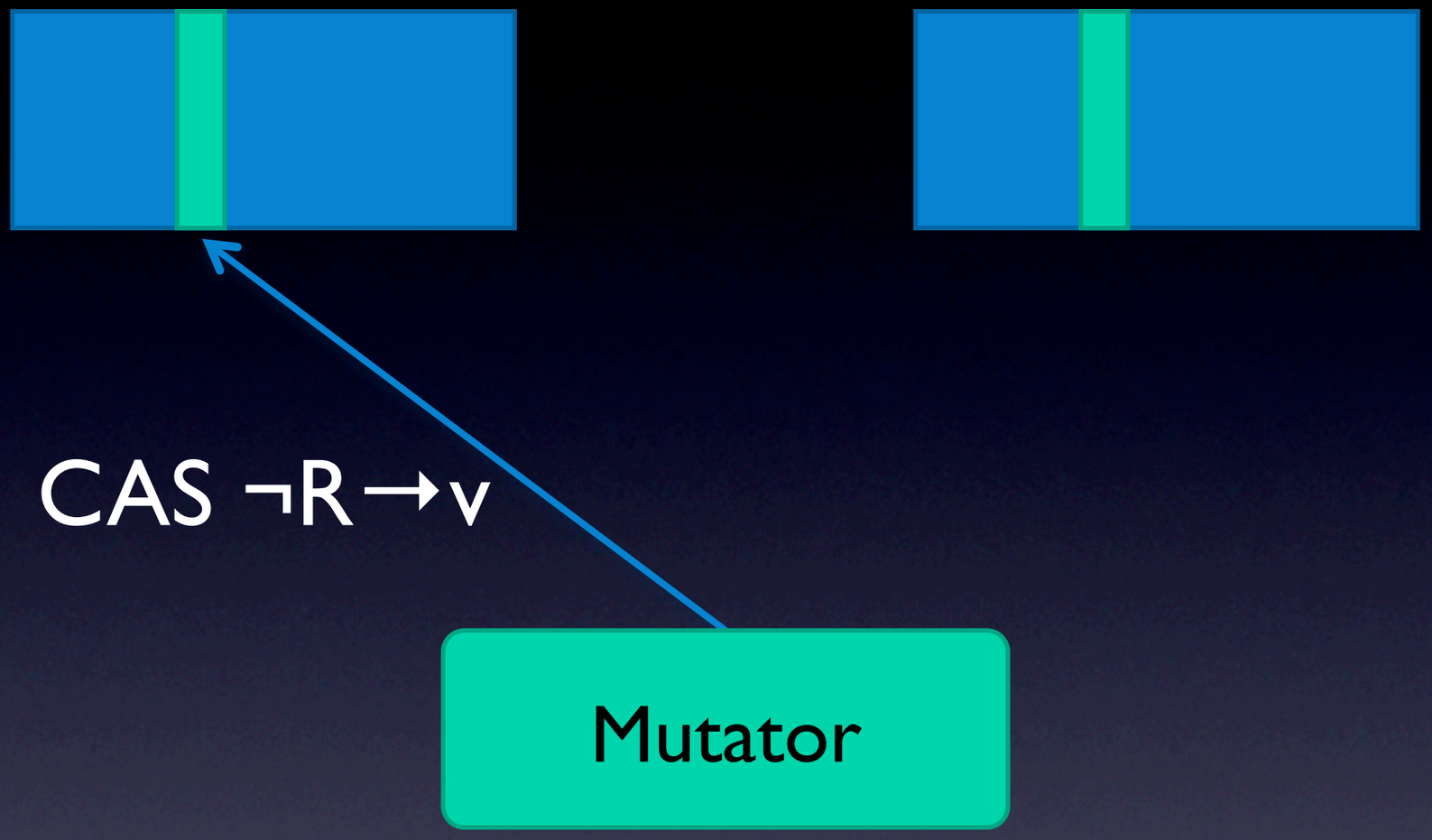
- Put this in perspective:
  - Probability that a person dies from a car crash in a single day in the US is higher than 1/300,000
  - Even if we stored a random value into a field once a nanosecond since the Big Bang, the probability of ever colliding with Clover would be 1/1,000,000,000,000

So - how does it work?

Mutator

Mutator

The mutator writes to the from-space using a CAS that asserts that the field is not copied (does not equal R).

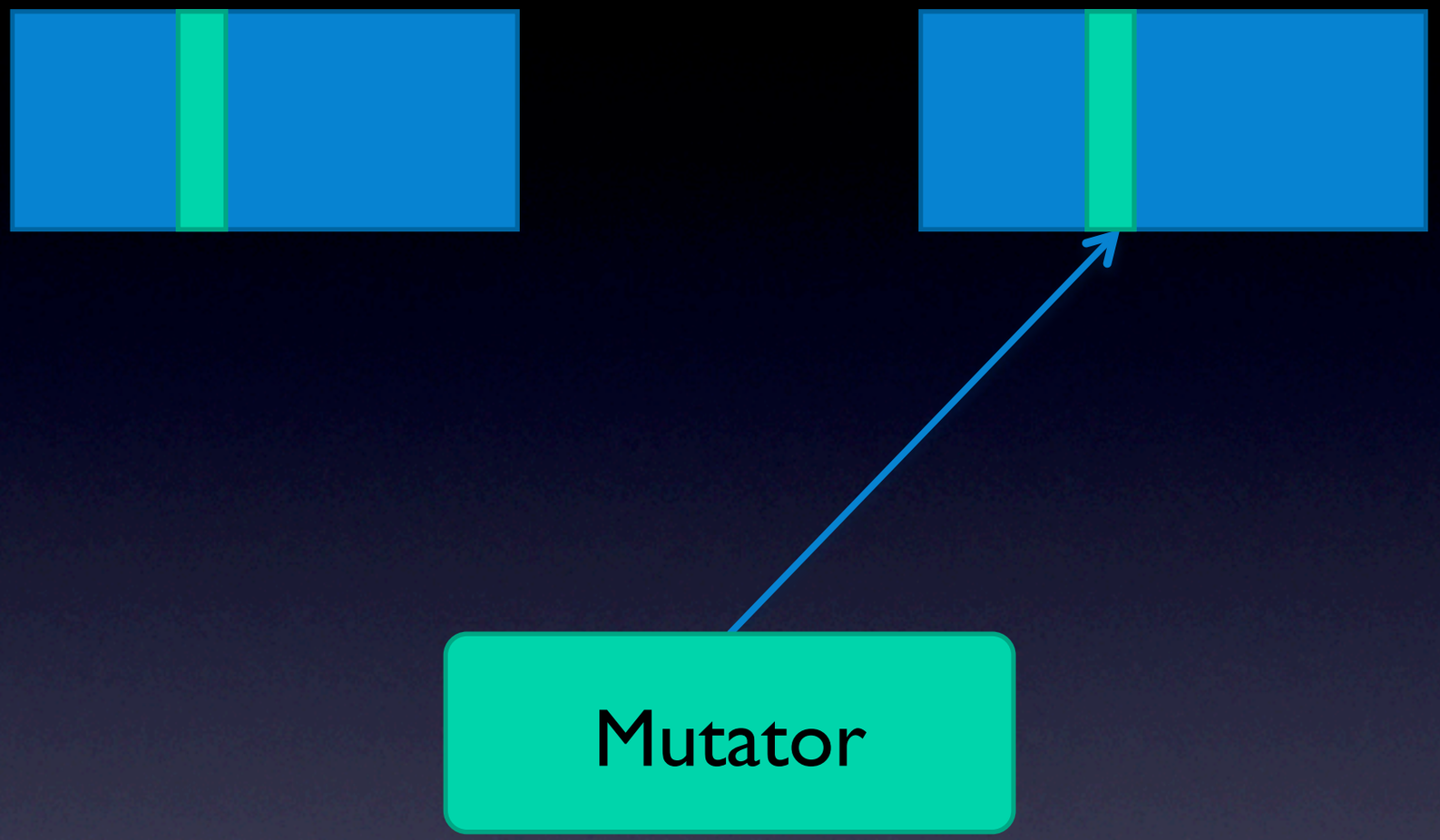


The mutator writes to the from-space using a CAS that asserts that the field is not copied (does not equal R).

Mutator

Mutator

If the CAS fails, the mutator just writes to to-space.

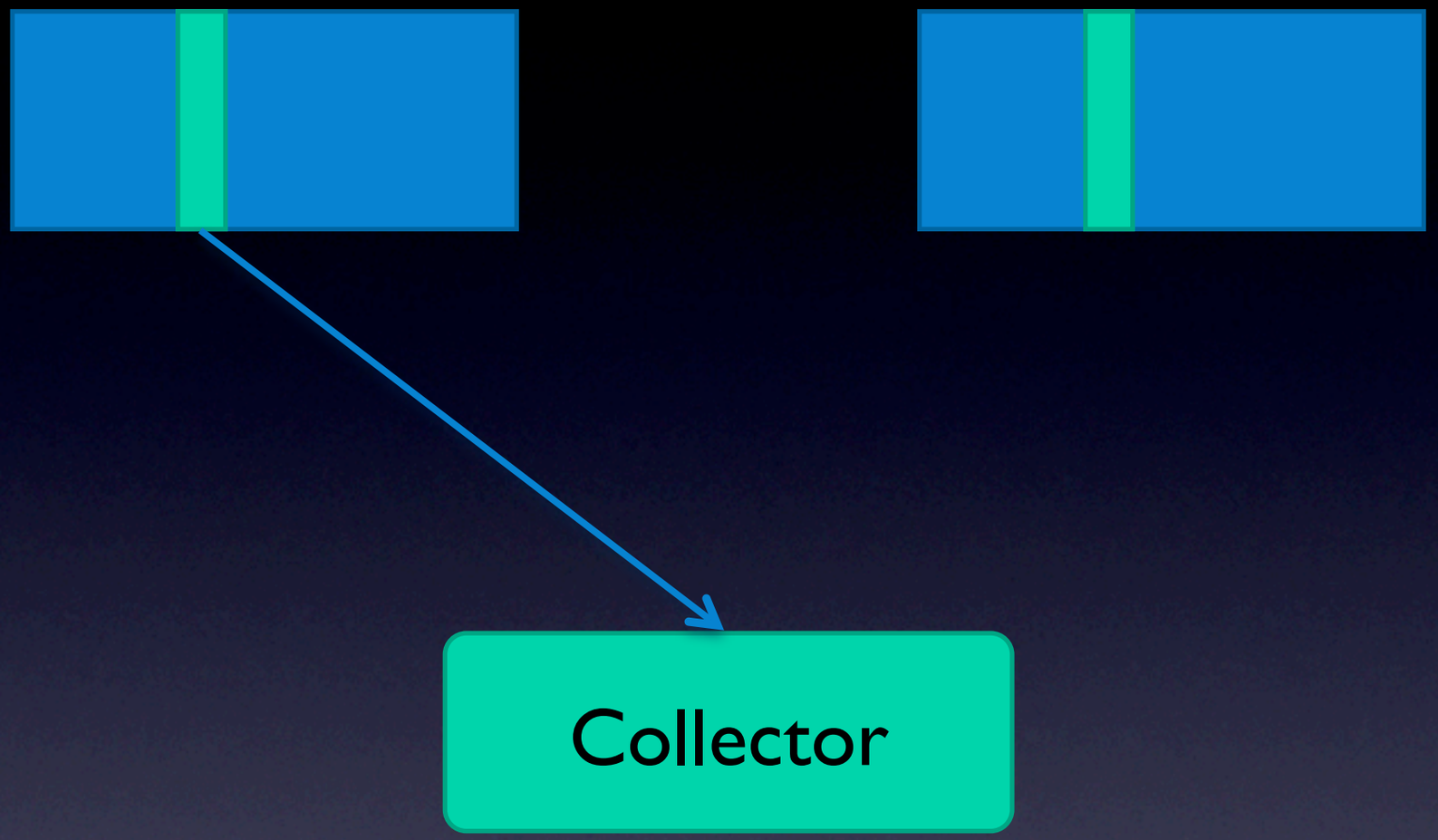


If the CAS fails, the mutator just writes to to-space.

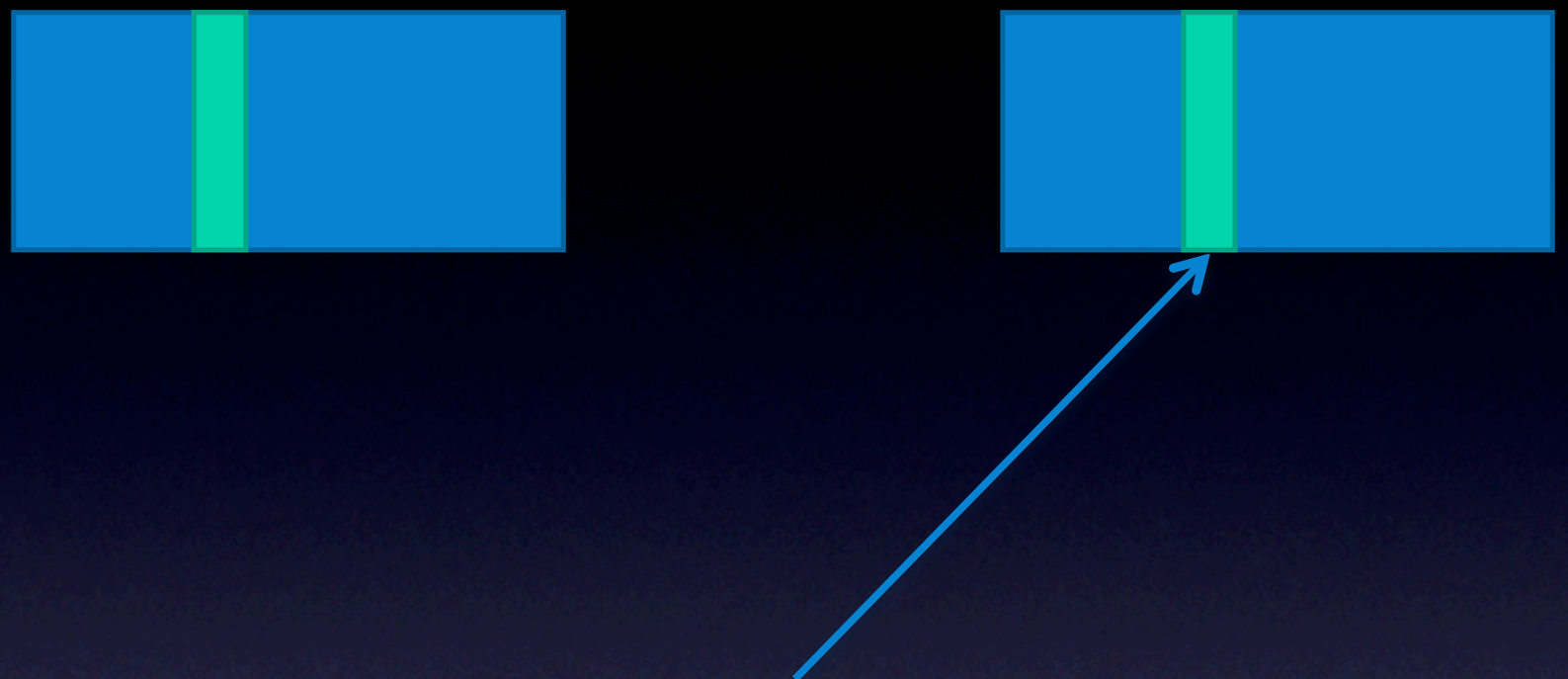
Collector

Collector

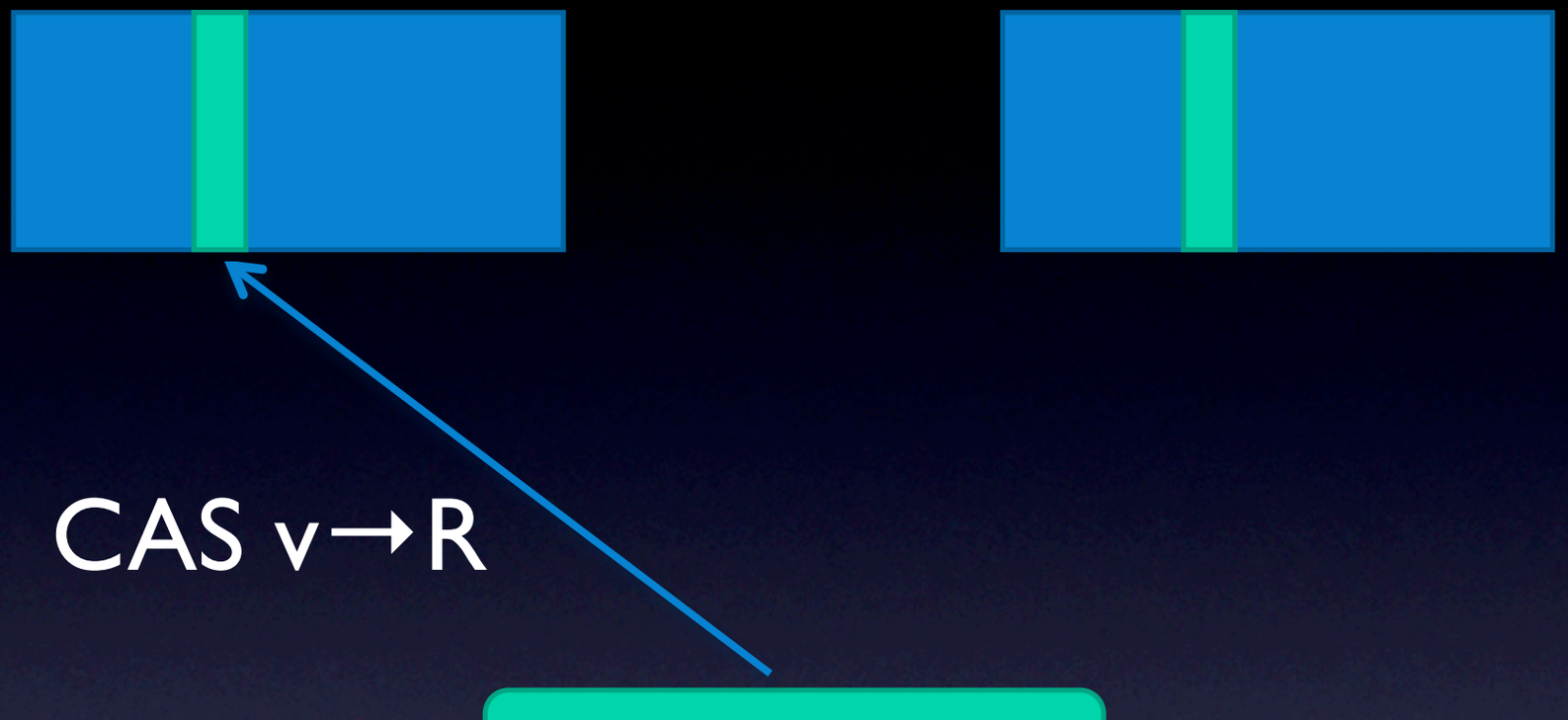
Collector repeatedly attempts to copy and assert the field as copied until it does so without the field's value changing.



Collector repeatedly attempts to copy and assert the field as copied until it does so without the field's value changing.



Collector repeatedly attempts to copy and assert the field as copied until it does so without the field's value changing.



Collector

Collector repeatedly attempts to copy and assert the field as copied until it does so without the field's value changing.

- What you just saw is a *probabilistically correct* concurrent copying algorithm.

- But we can:

- Make the algorithm *correct but probabilistically lock-free* by detecting when the user uses R.

Implementation

- Chicken and Clover are implemented in the same infrastructure as Stopless (ISMM'07)
- We use the Microsoft Bartok Research Compiler, and extend the lock-free concurrent mark-sweep collector.
- We use Path Specialization (ISMM'08) to optimize barrier performance.

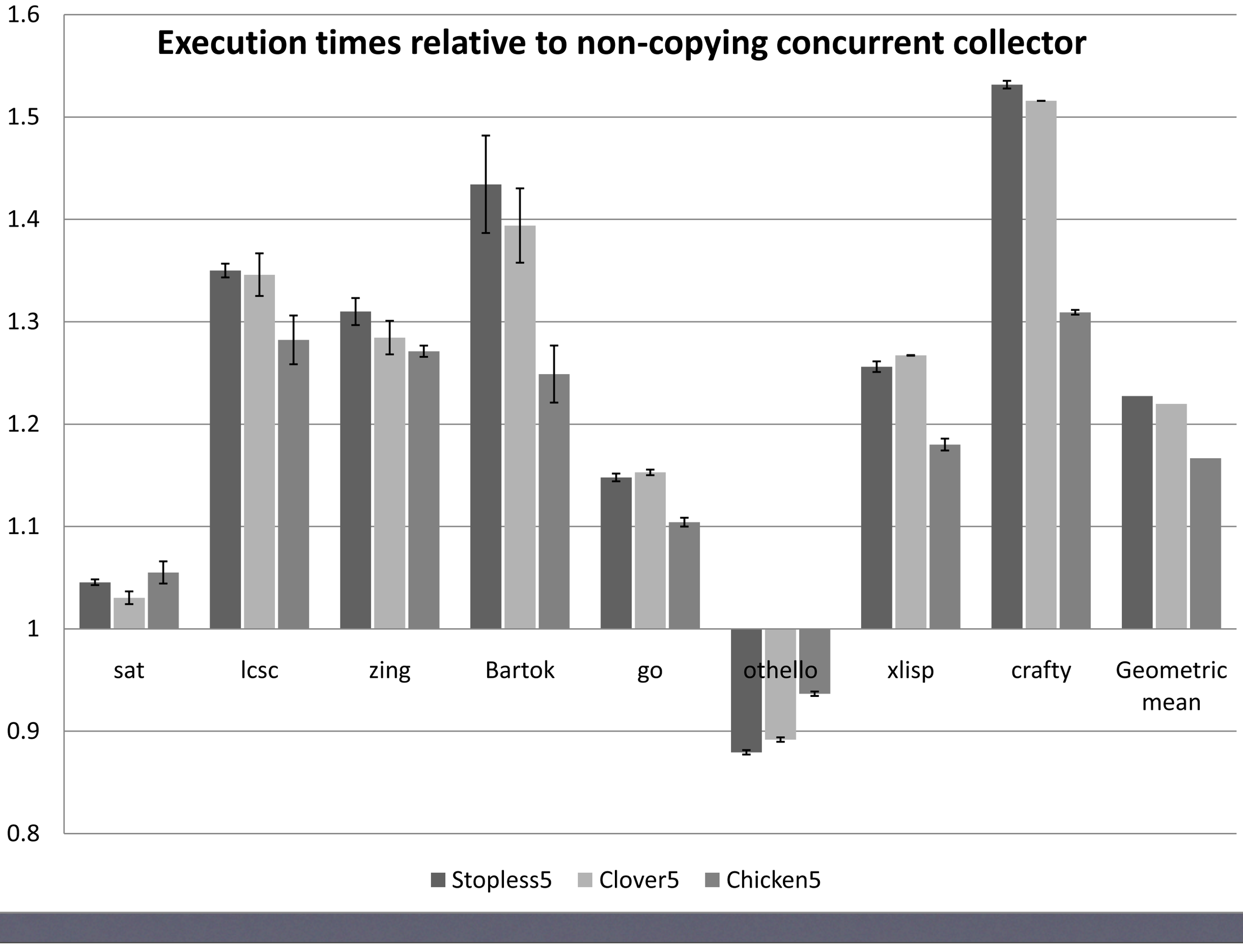
Results

Summary of Results

- Both schemes have ~20% throughput overhead
- Clover leads to a ~3x slow-down when executing with full barriers
- Chicken has almost no slow-down.

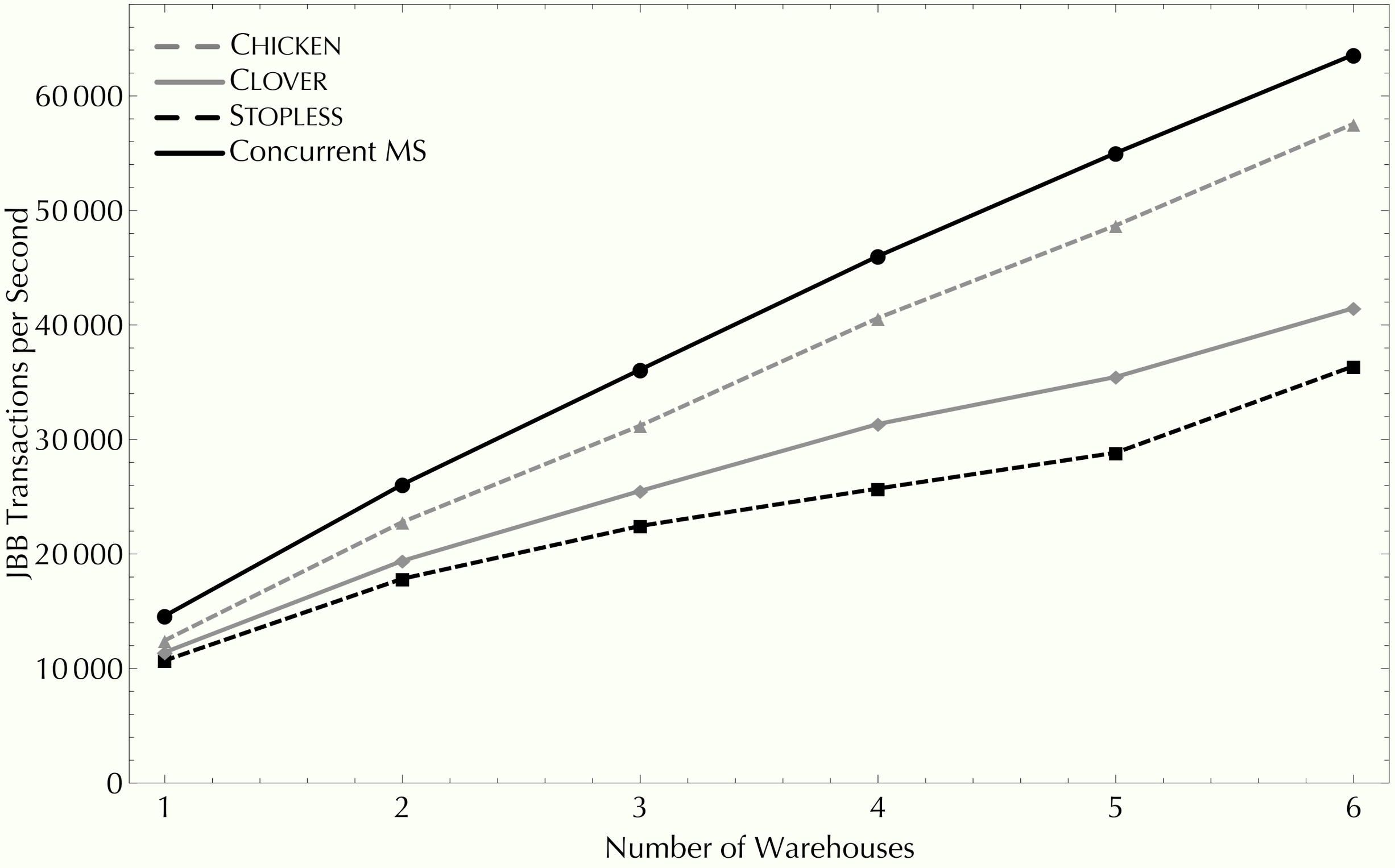
Detail: throughput

- MSR benchmark suite (four internal PL-type programs written in C#, VB, and C++, plus four traditional benchmarks ported to .NET)
- Compare concurrent mark-sweep (CMS), Stopless (ISMM'07), Chicken, and Clover


Execution times relative to non-copying concurrent collector
1.6
1.5
1.4
1.3
1.2
1.1
1
0.9
0.8
sat
lcsc
zing
Bartok
go
othello
xlisp
crafty
Geometric mean
Stopless5
Clover5
Chicken5

Detail: scalability

- SpecJBB2000 ported to C# using the Microsoft Visual Studio Java to C# converter
- Compare CMS, Stopless, Clover, and Chicken

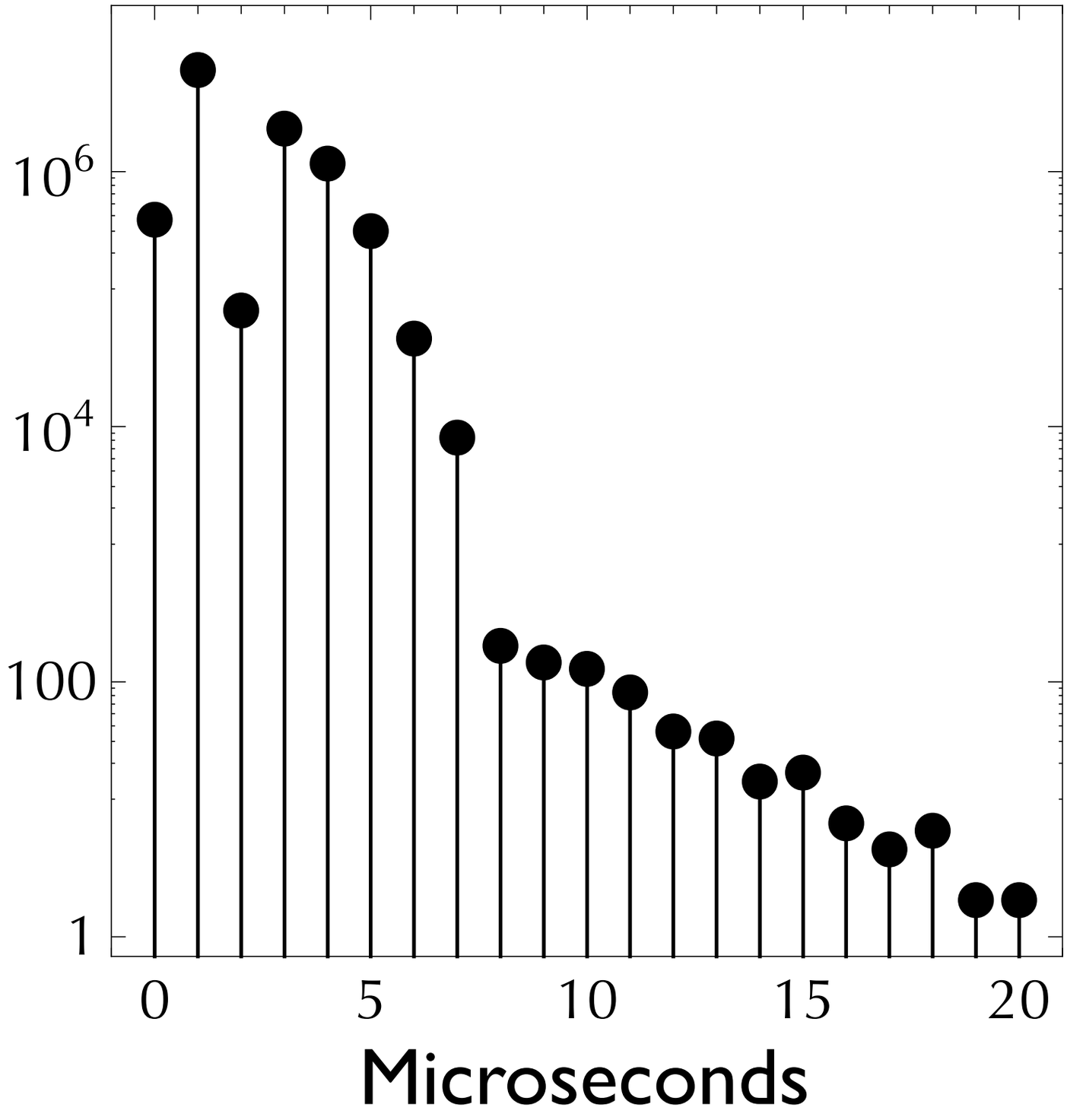

Chicken
Clover
Stopless
Concurrent MS
60 000
50 000
40 000
30 000
20 000
10 000
0
JBB Transactions per Second
1
2
3
4
5
6
Number of Warehouses

Detail: responsiveness

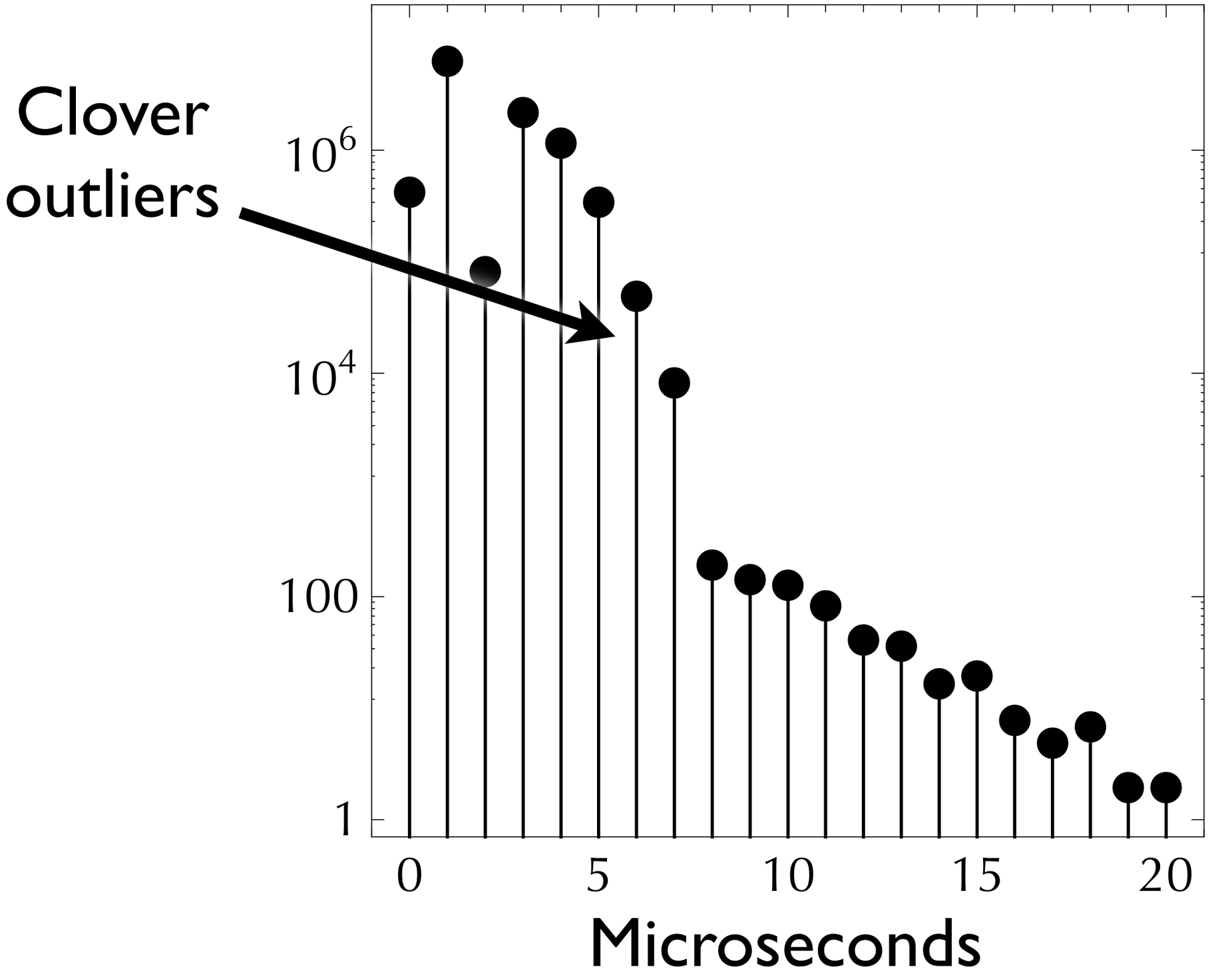
- Two benchmarls:
  - Microbenchmark measuring responsiveness for short-running interrupt handlers
  - Our JBB port (measure transaction time distribution)

- For the Interrupt Microbenchmark we measure:
  - concurrent mark-sweep (see paper)
  - Stopless (see paper)
  - Clover
  - Chicken

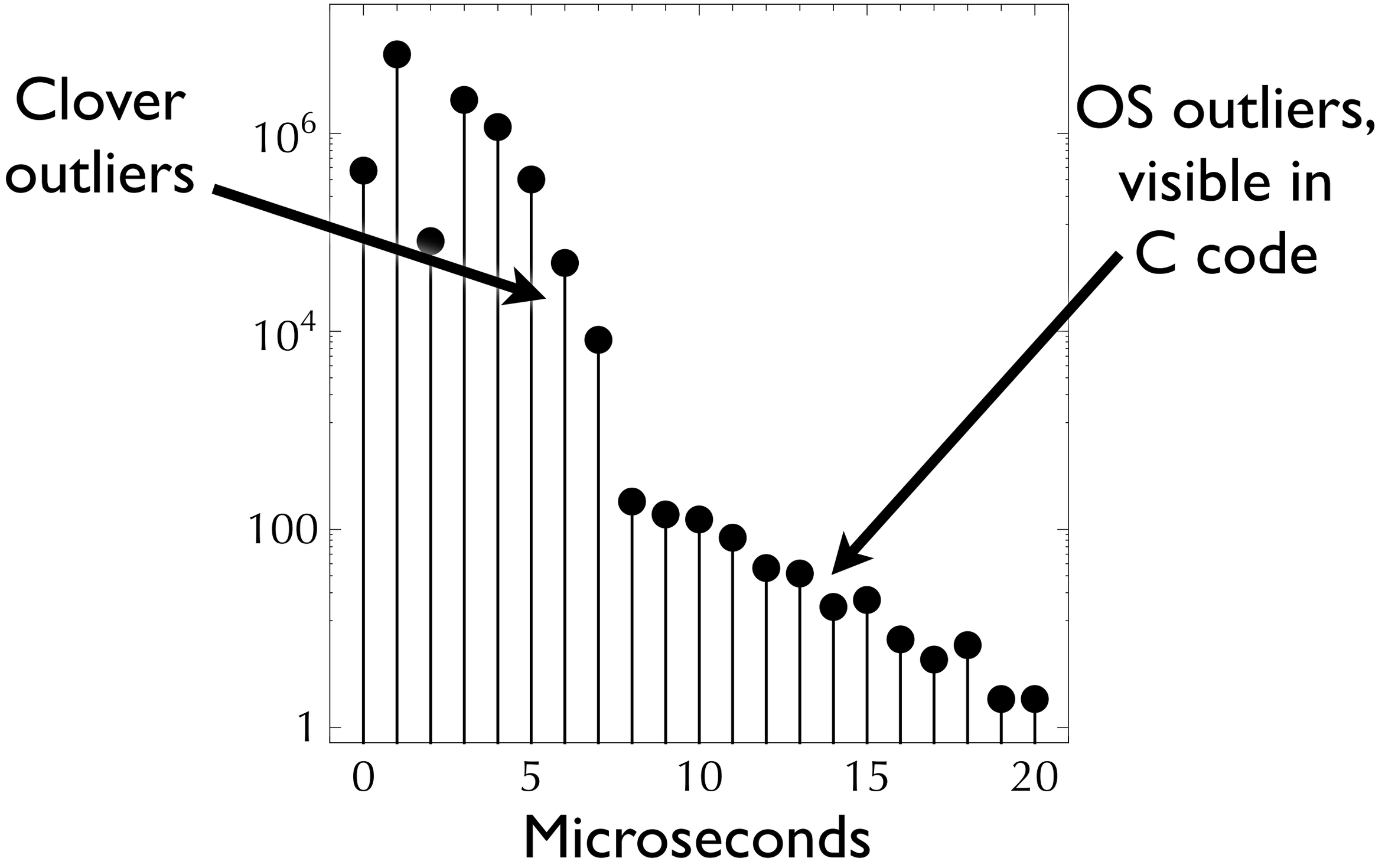
Interrupts: Clover



Interrupts: Clover



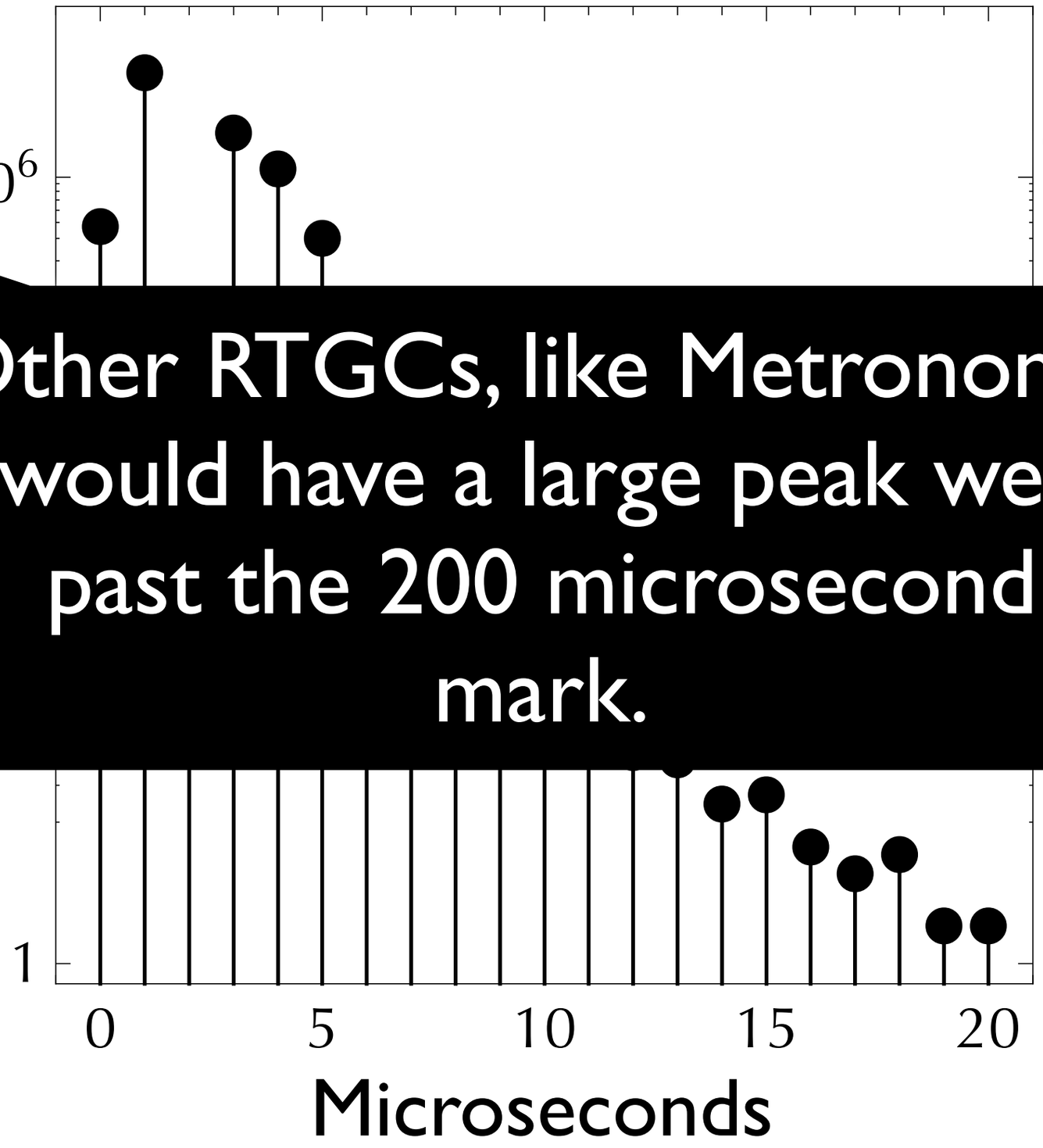
Interrupts: Clover



Interrupts: Clover

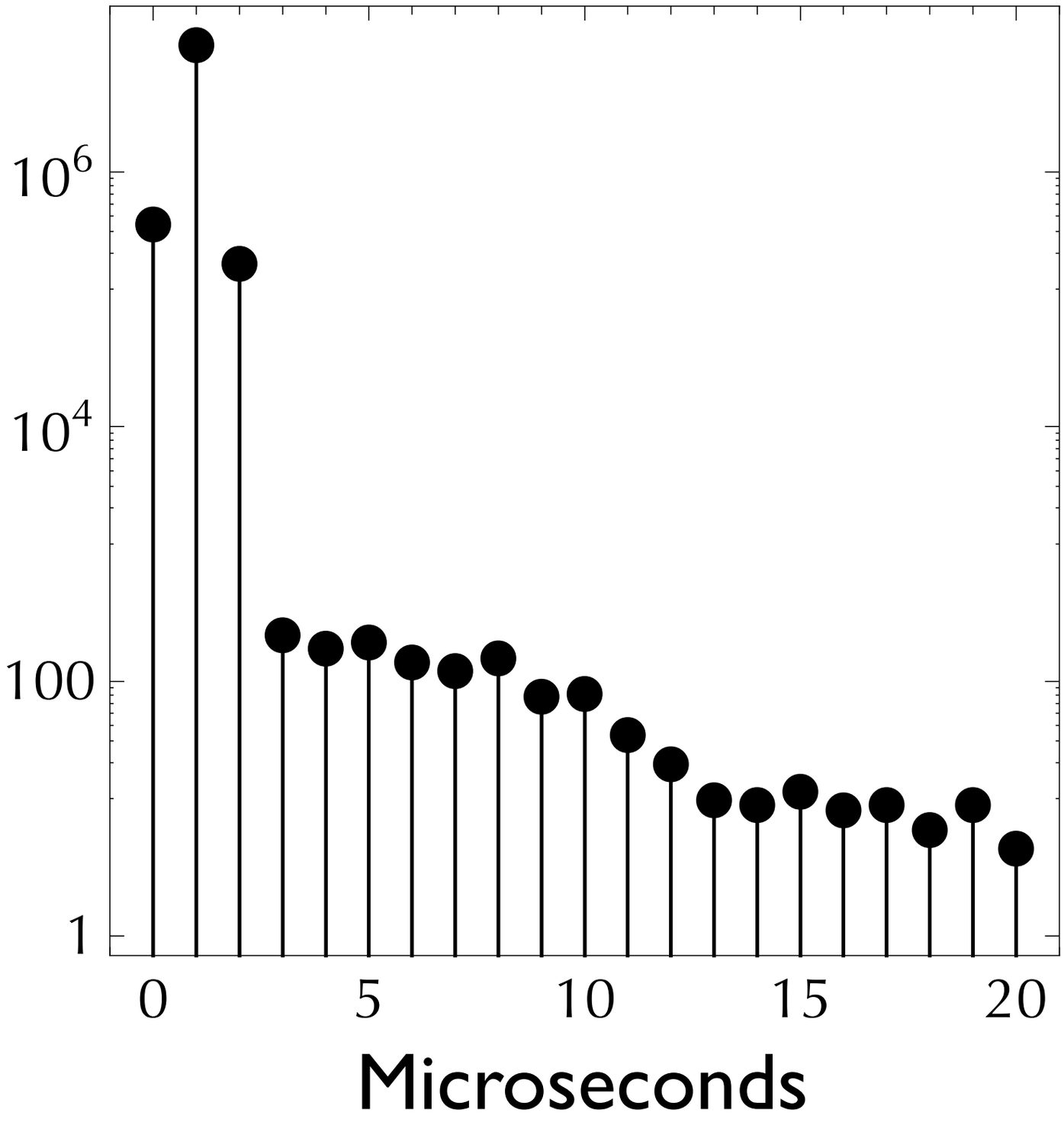
Clover outliers

OS outliers, visible in code

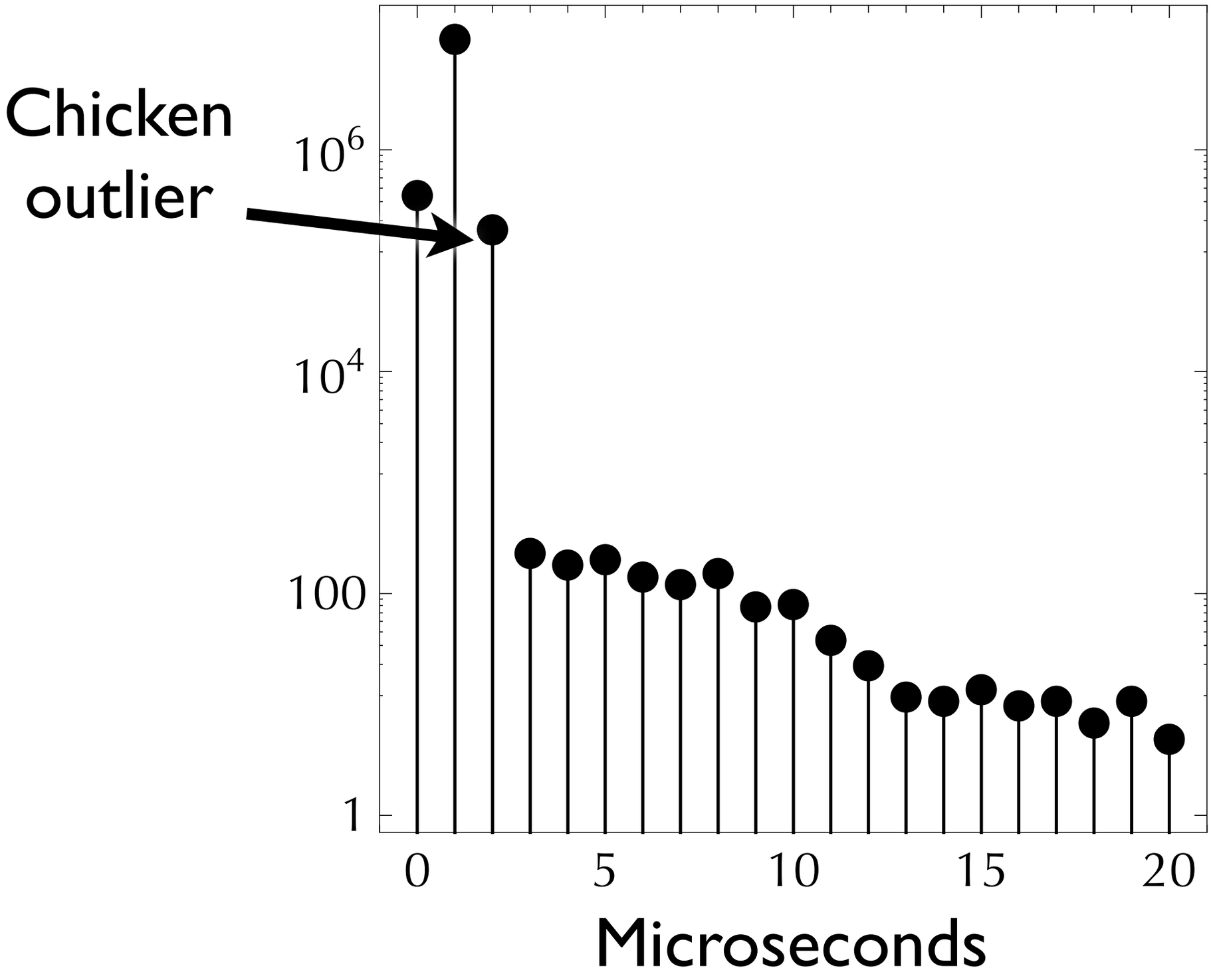


Other RTGCs, like Metronome, would have a large peak well past the 200 microsecond mark.

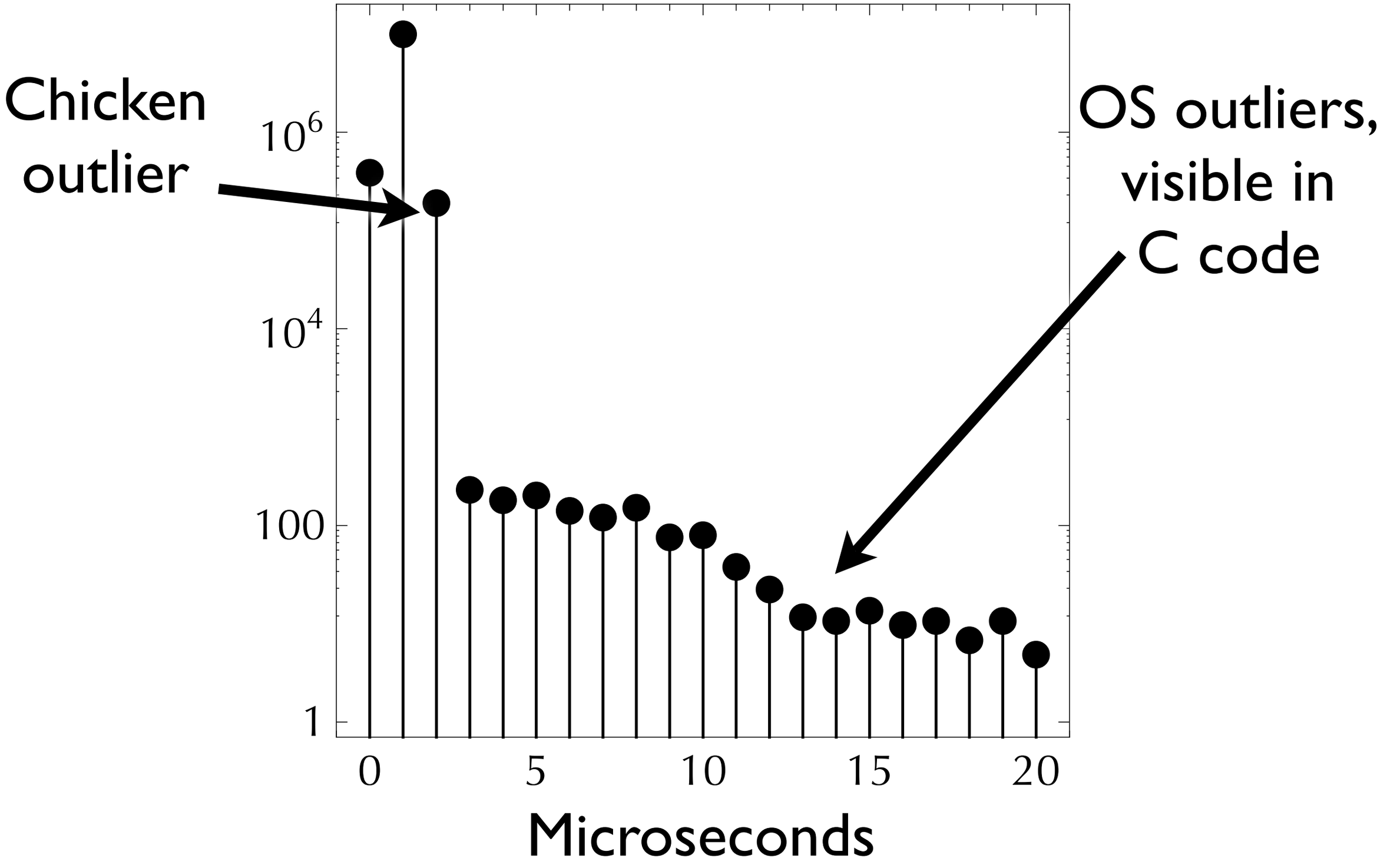
Interrupts: Clover



Interrupts: Clover

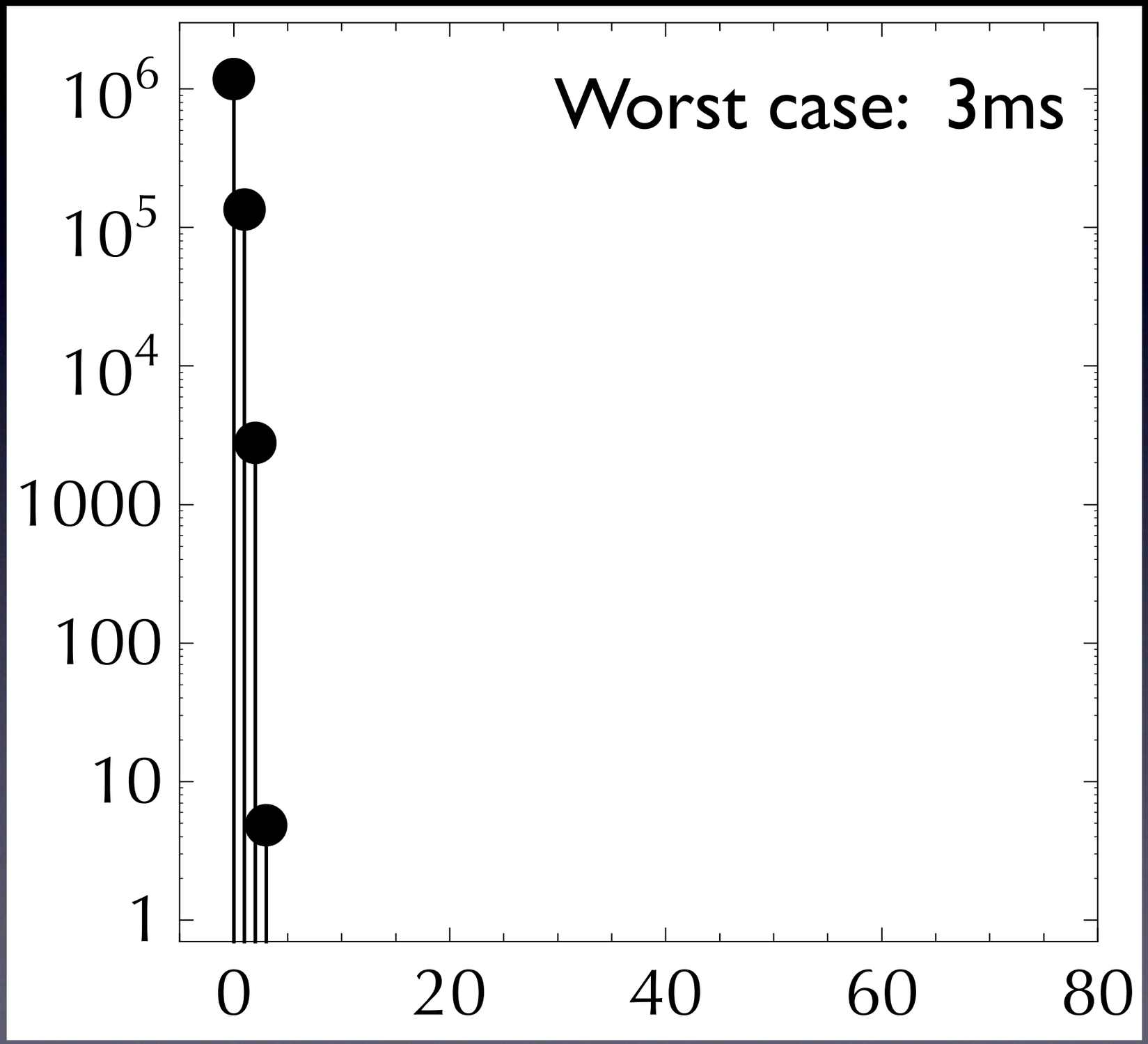


Interrupts: Clover

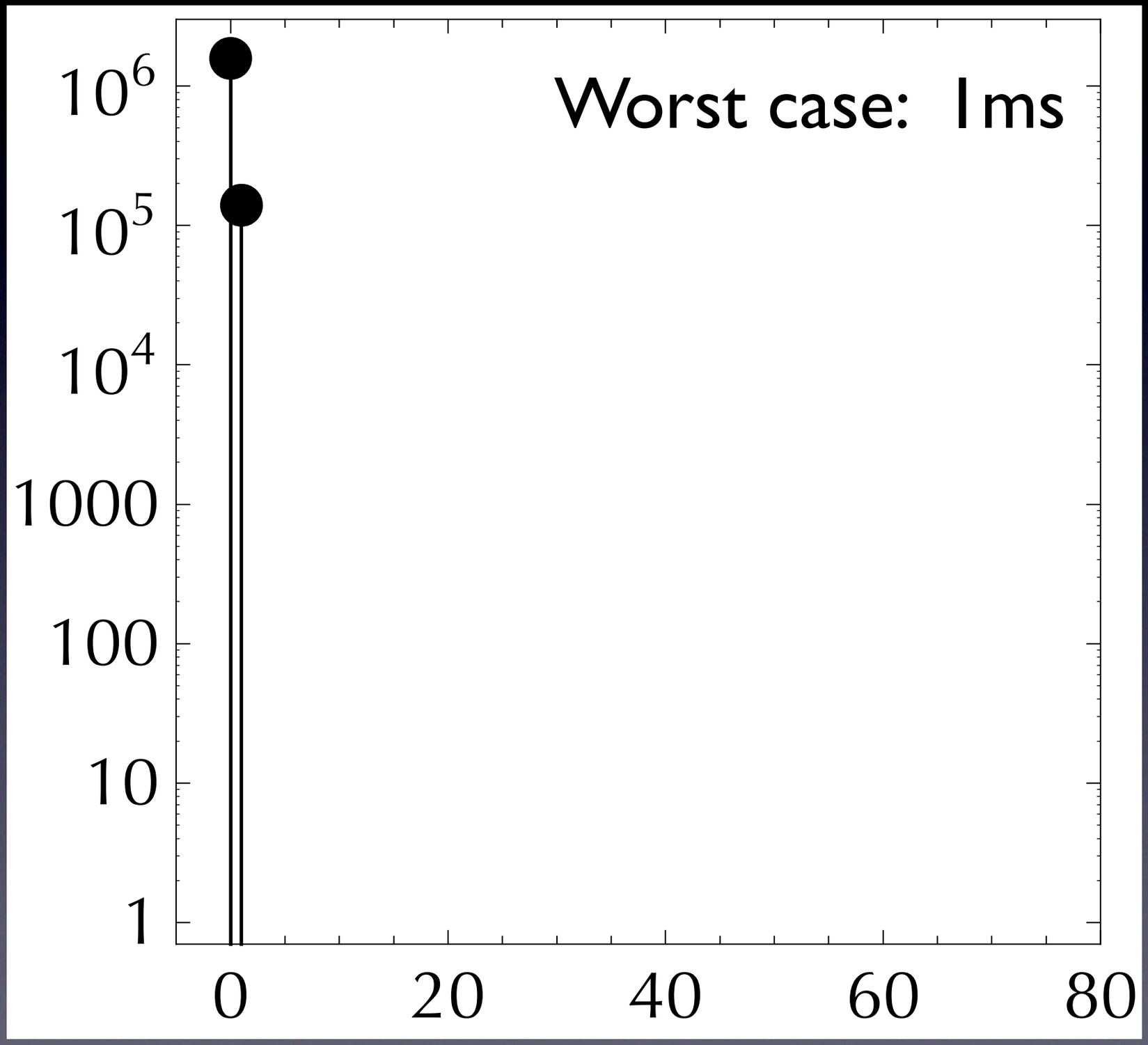


- For JBB we measure:
  - stop-the-world mark-sweep (see paper)
  - Stopless (see paper)
  - Clover
  - Chicken

JBB: Clover



JBB: Chicken



Summary

- Presented two new concurrent copying strategies - one that is very light-weight, and another with strong (but probabilistic!) guarantees.
- Both are simpler than previous techniques.
- Both provide good throughput and responsiveness.

Questions