

Algorithms for Dynamic Memory Management (236780)

Lecture 13

Lecturer: Erez Petrank

Last Week

- Parallel GC
- Allocation methods

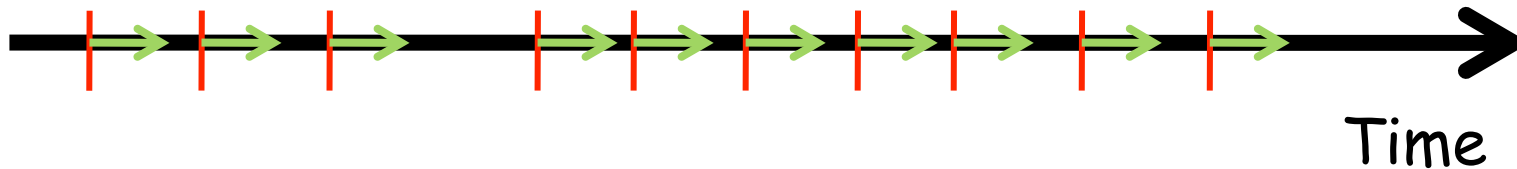
- Home exercise number 3 was published...

This Week

- Real-Time GC
- Agenda:
 - Real-time requirements.
 - What's bad with Baker's collector ? (Work-based versus time-based collection)
 - IBM's Metronome.
 - Microsoft's concurrent real-time collectors

Real-Time

- Respond **on time** to events.
 - Flight control, telecommunication, audio processing, stock commerce, network switches, etc.
- Predictable response more important than efficiency.
 - But it would be nice not to act slowly.
 - Main parameters: latency of response, throughput overhead.



Real-Time for Managed Code

- Historically, real-time was written in low-level code, requiring some sort of verification.
 - Difficult to create meaningful software.
- Garbage collection is a major obstacle for C# & Java.
 - So are Jitting and dynamic class loading.
- Real-Time Specification for Java (RTSJ): real-time threads do not use the garbage collected heap.
- With time, real-time programs get more complicated.
 - A reliable and scalable development environment is needed.

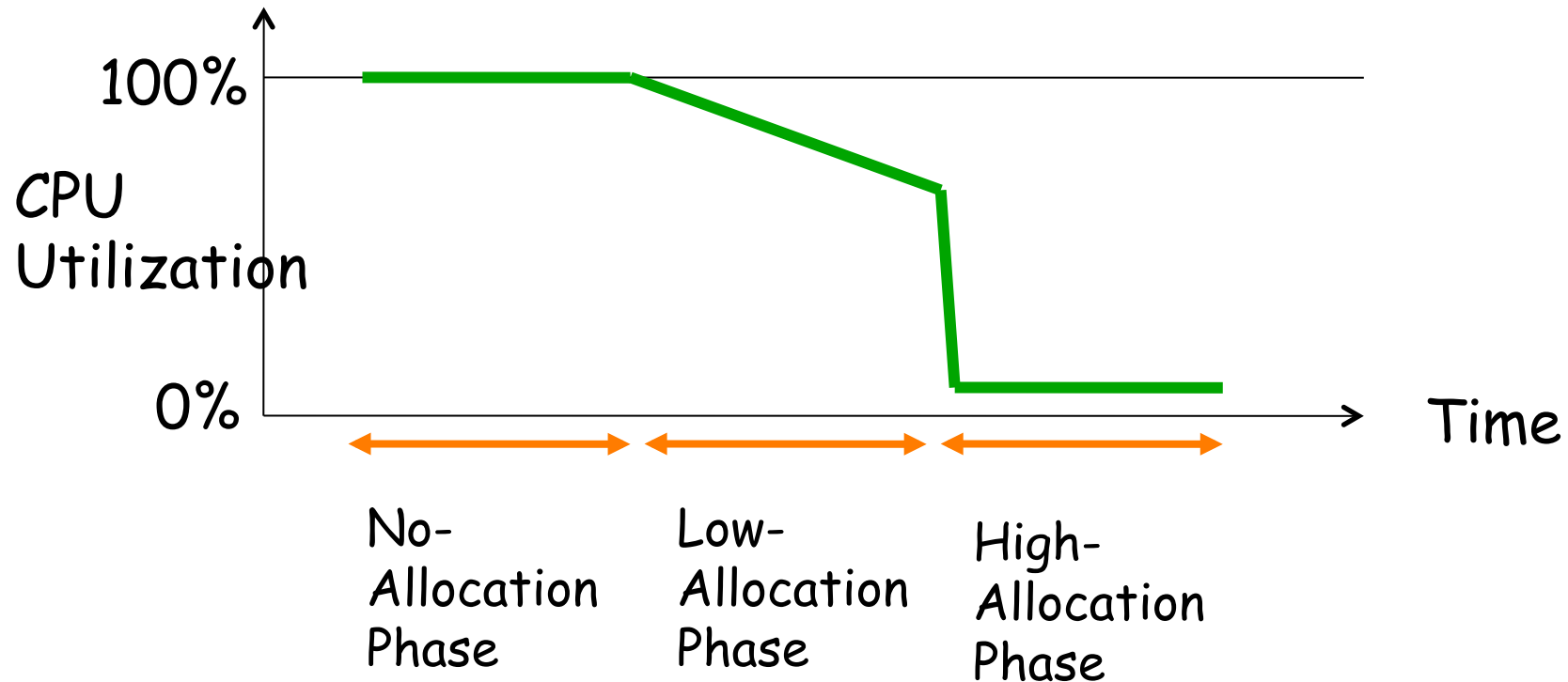
Real-Time Garbage Collection

- Must be either incremental or concurrent.
- Pauses must be bounded !
- Pauses should be evenly distributed to allow mutator to respond.
- Memory overhead should be within provided resources
 - Cannot just allocate without collecting
 - Cannot ignore fragmentation.

Work-Based versus Time-Based Triggering

- Baker designed a **work-based** incremental copying collector for real-time purposes.
- The idea: perform some incremental collector work during each allocation.
- **Advantages:**
 - Fairness: a thread that allocates a lot pays more.
 - Termination: The work of the collector ends before heap exhausted
- **Disadvantage:**
 - In program phases that allocate frequently, mutator gets a low percentage of the CPU.

Mutator CPU Utilization

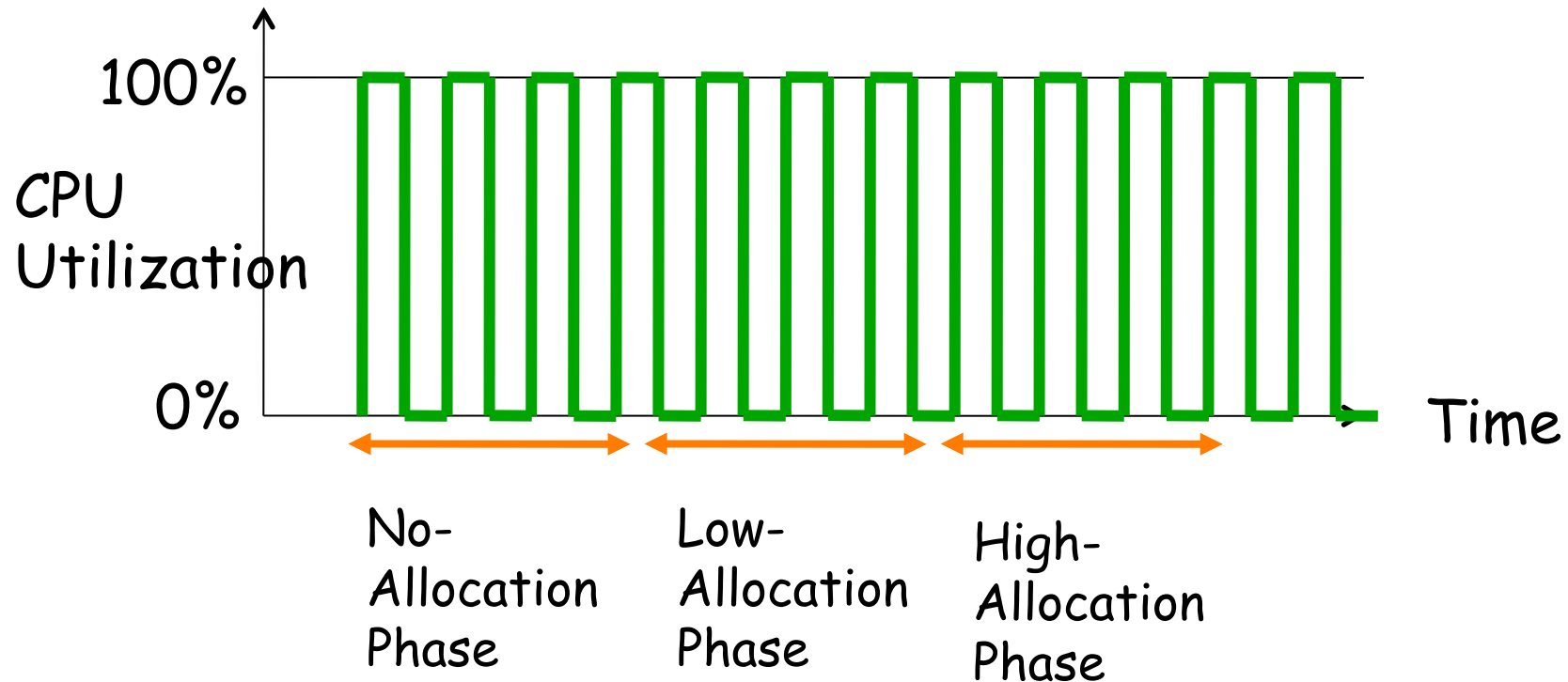


During high allocation phase, a program may miss a real-time deadline !

Time-Based Triggering

- A **time-based** collector provides a percentage of the time to the collector.
- E.g., mutator runs for 1ms, GC for 2ms, repeatedly.
- The idea: mutator guaranteed to get 1ms every 3 ms.
- **Advantages:**
 - Predictable response.
- **Disadvantage:**
 - No guaranteed termination. Must ask the user to supply parameters (such as rate of allocation and max live space)
 - Required mechanism for stopping threads on time and switching to collection.

Time-Based Collectors

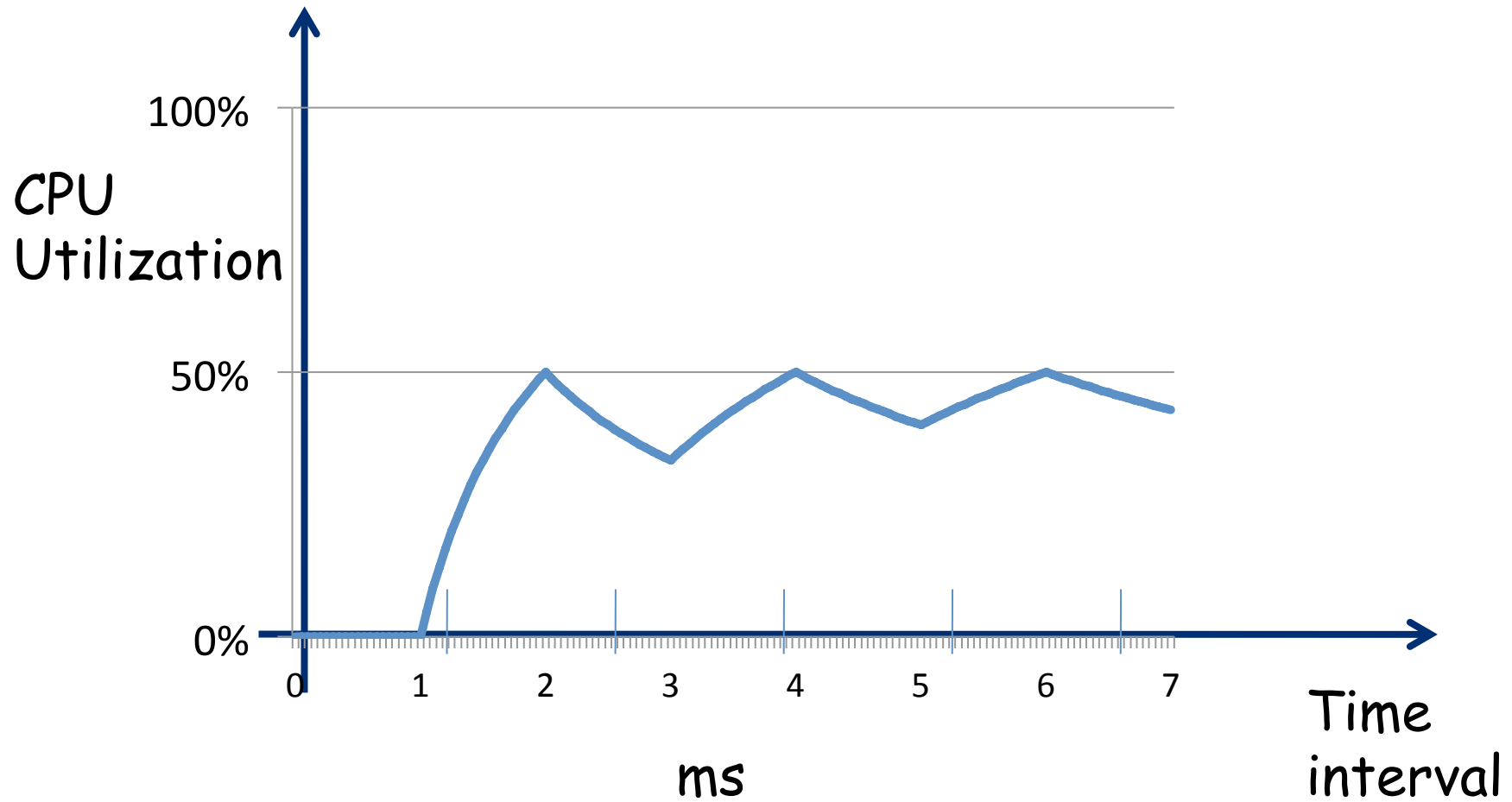


During any (sufficiently long) phase, a program gets a fixed percentage of the CPU.

Minimum Mutator Utilization

- Note that we said: “sufficiently long phase”.
- A more informative measure is the minimum mutator utilization graph (MMU).
- X-axis: time intervals
- Y-axis: given a time interval length, what is the worst mutator CPU utilization during the run.
- Example, if run is as in previous graph: collector and mutator get 1ms alternating.

The MMU Graph



Minimum Mutator Utilization

- Information that can be obtained:
 - Maximum pause time (how?)
 - For each possible time interval: what is the fraction of CPU time that the mutator is guaranteed to get in this interval length.
- A real run will not have such a simple line. E.g., the collection is not active all the time, and alternation ends, leaving the CPU to mutator.
 - If collection occurs half of the time, how will the above MMU graph change?

IBM's Metronome

David Bacon, Joshua Auerbach, Perry
Cheng, Dave Grove, Mike Hind
(2003-2009)

Overview

- A real-time garbage collector for Java.
- Initially (2003) designed for a uniprocessor, thus, incremental.
- Time-based triggering.
- Mark-sweep (incremental) collection.
- Infrequent partial compaction to avoid fragmentation.
- Extensions include generations, parallelism, and concurrency.

Allocation

- Segregated free-list allocator, block-oriented:
 - Fixed-size pages
 - Each page only allocates fixed-size spaces.
- Objects allocated in smallest block that fits
- Sizes grow geometrically, limiting the inner fragmentation according to the growth factor $(1+\rho)$.

Incremental Mark-Sweep

- Snapshot-oriented.
- Write-barrier saves overwritten (old values of) pointers in buffers.
- Buffered pointers become roots.
- Mark phase is used to fix pointers of objects that have moved (to be discussed later).

Handling Fragmentation

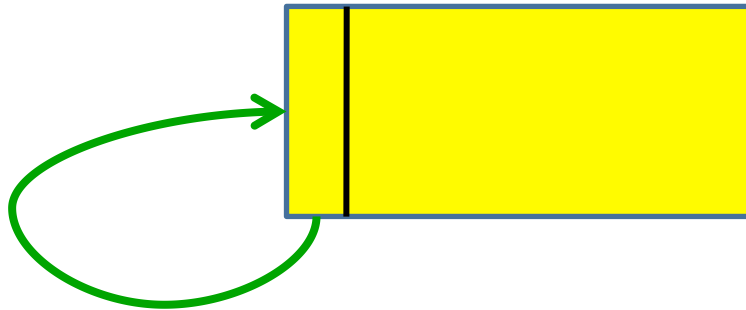
- A real-time garbage collector should handle worst-case scenarios.
- (This should be taken with a grain of salt because we cannot really handle all that can go wrong...)
- It must be able to handle fragmentation, otherwise, program will fail on space limit.
- Partial compaction: when fragmentation detected, move some of the objects.
- Hopefully, this happens seldom.
- Usually, no attempt is made to preserve objects order.

Partial Compaction

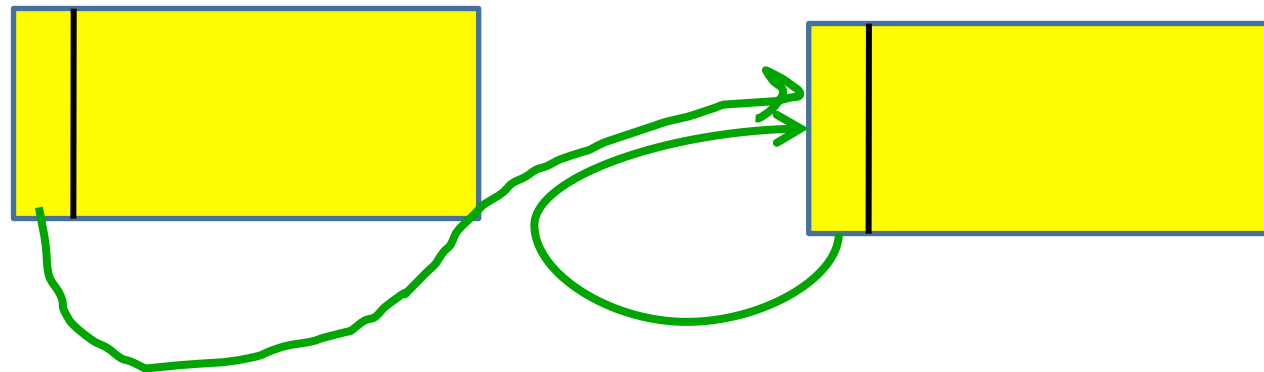
- Problem: when moving an object, a new copy is created.
 - At some point all program threads should start accessing the new copy.
 - If one thread writes the old copy after other threads read the new one, the write will be lost!
 - This switch is simpler on a uniprocessor.
- Main idea: indirect access to objects.
- Each object contains in its header a pointer either to itself or to its new copy.

Indirect Access (Brooks)

Case I: object not copied



Case II: object has a newer copy.



Object Access

- Each object access (read-barrier) goes through the indirect pointer.
- To move an object, the collector
 - Creates a new copy ;
 - Copies all data from old to new copy;
 - Sets the forwarding pointers (in both objects);
- The above operations must appear atomic to other threads (why?).
- Uniprocessor (or any non-concurrent) solution: CPU is not yielded in the middle of a move.

Reclamation of Old Copies

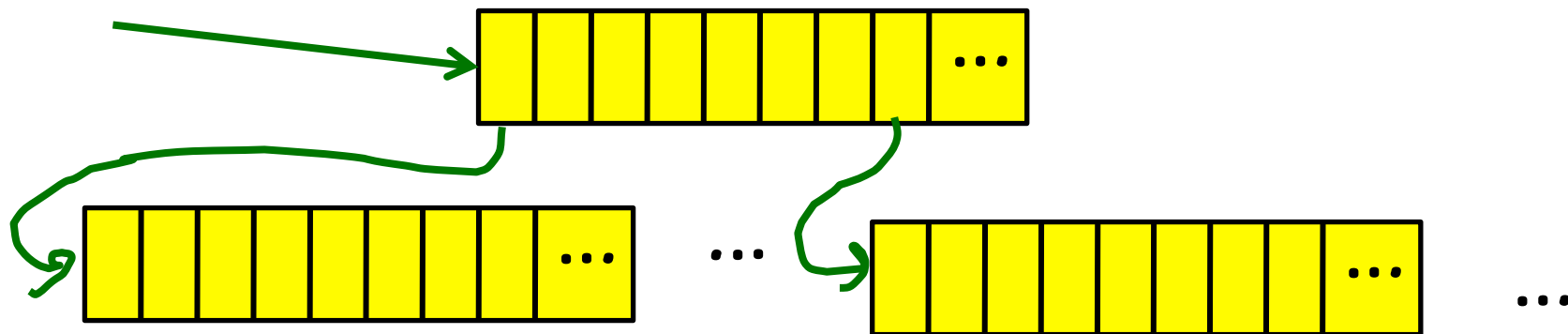
- An old copy can be reclaimed after nothing points to it.
- The mark procedure updates all reachable pointers.
- After that, old objects are not reachable and are automatically reclaimed during the next collection.

Picking Objects to Relocate

- Fragmentation indicators:
 - Free pages fall below limit for performing a GC
 - Dead slots are not re-used for a GC cycle
- In practice: 2-3% of reachable data is moved, allows heaps of size $< 2 \times$ size of live data.

Large Arrays and Objects

- Large objects are impossible to move in a short quantum.
- Solution: Arraylets.
 - Each array broken into arraylets.



Arraylets

- Advantage:
 - Can move bounded-size pieces.
- Disadvantage:
 - More indirection: time and space overhead.

Triggering

- Triggering is a major technical issue.
- Time triggering: after application runs time $t(app)$, collector runs time $t(col)$.
- To guarantee termination, $t(app)/t(col)$ is set according to allocation rate, live space, heap size, etc.
- The quantum $t(col)$ is upper bounded by the response time of the application, it is also lower bounded by technical constraints
 - At time required for the atomic sections (scan the stack, move a large object) + time to stop threads, if made parallel.
- We do not get into the (tedious) formulas.

Microsoft's Real-Time Concurrent Garbage Collection

Filip Pizlo, Daniel Frampton,
Gabi Kliot, Erez Petrank, Bjarne
Steendsgaard

Real-Time Garbage Collection

- Mtronomie's response time within $\sim 1\text{ms}$ (at 2007, now hundreds of $\mu\text{-sec}$).
- Can we respond within $10 \mu\text{-sec}$ and less?

Real-Time GC on Parallel Platforms: What's Expected?

- Short (or no) GC pauses
 - support responsiveness.
- Program never waits for the collector
 - lock freedom support: GC does not add locks.
- Partial compaction
 - to avoid worst-case fragmentation.
- Low overhead and multiprocessor scalability
- This work was the first to provide all of these requirements simultaneously.

Improving Responsiveness

- How do we get responsiveness faster than Metronome?
- A simple idea: **use concurrent GC**.
- Run GC on one processor/core and the (parallel) program on the others.

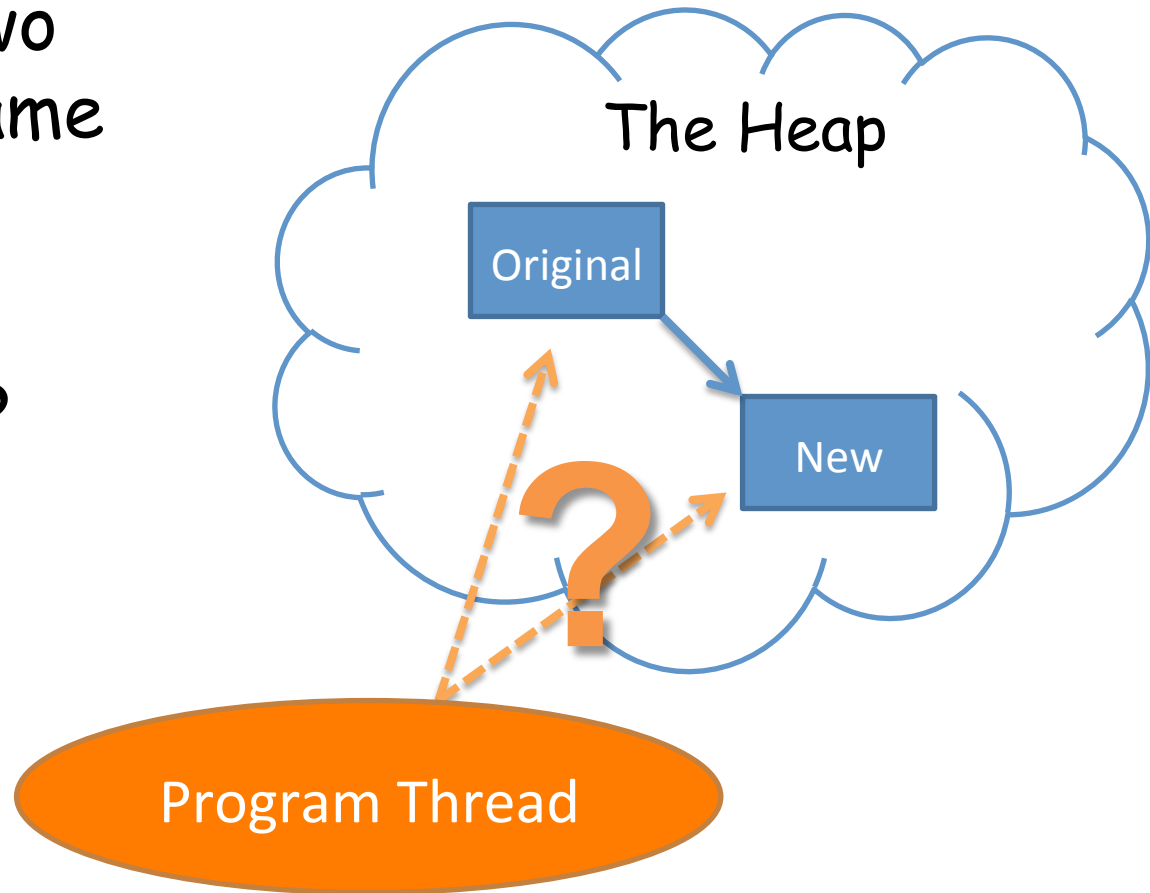
😊 Obtain high responsiveness.

😞 Problem: need new technique.

- Main problem: move objects concurrently

Moving Objects Concurrently

- The problem: two copies of the same object.
- *Which version should be used?*

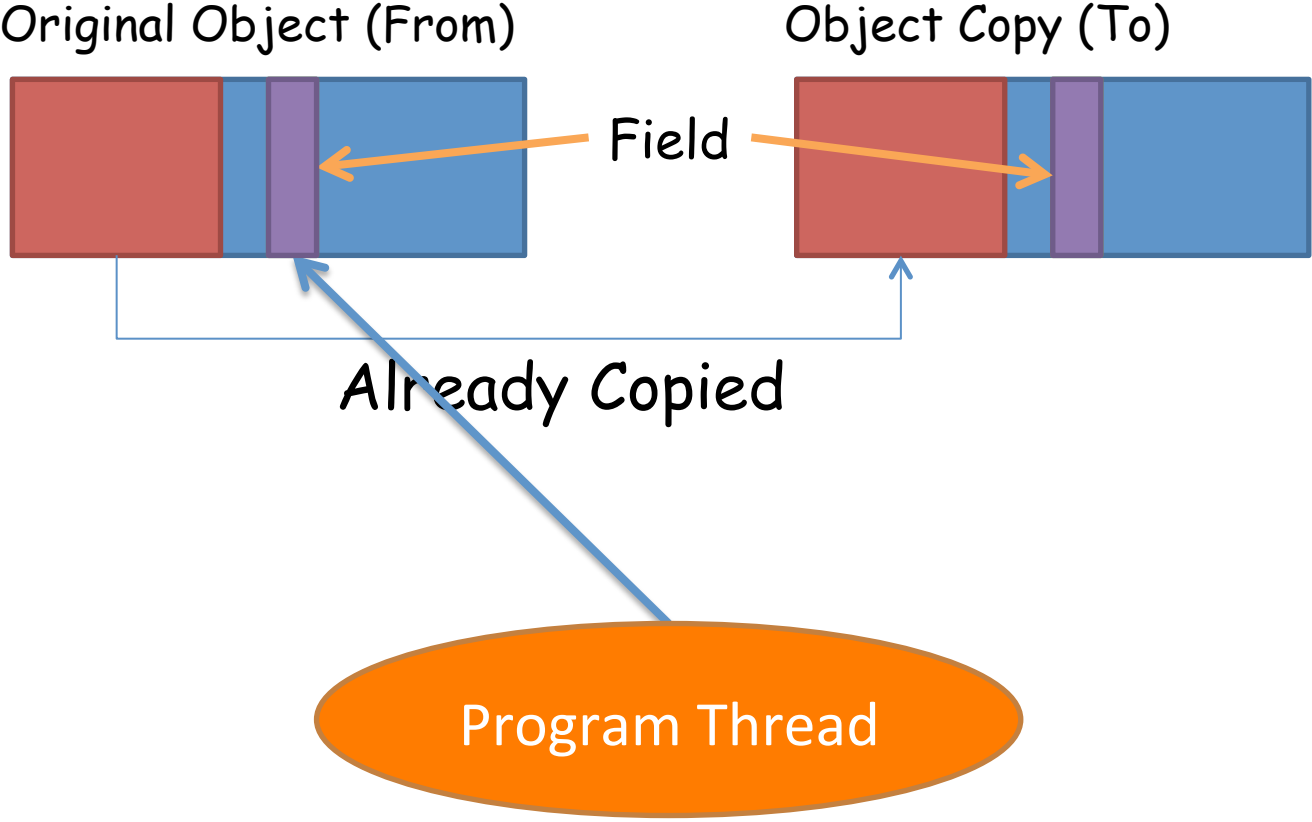


Original Object (From)

Object Copy (To)



Program Thread



Original Object (From)

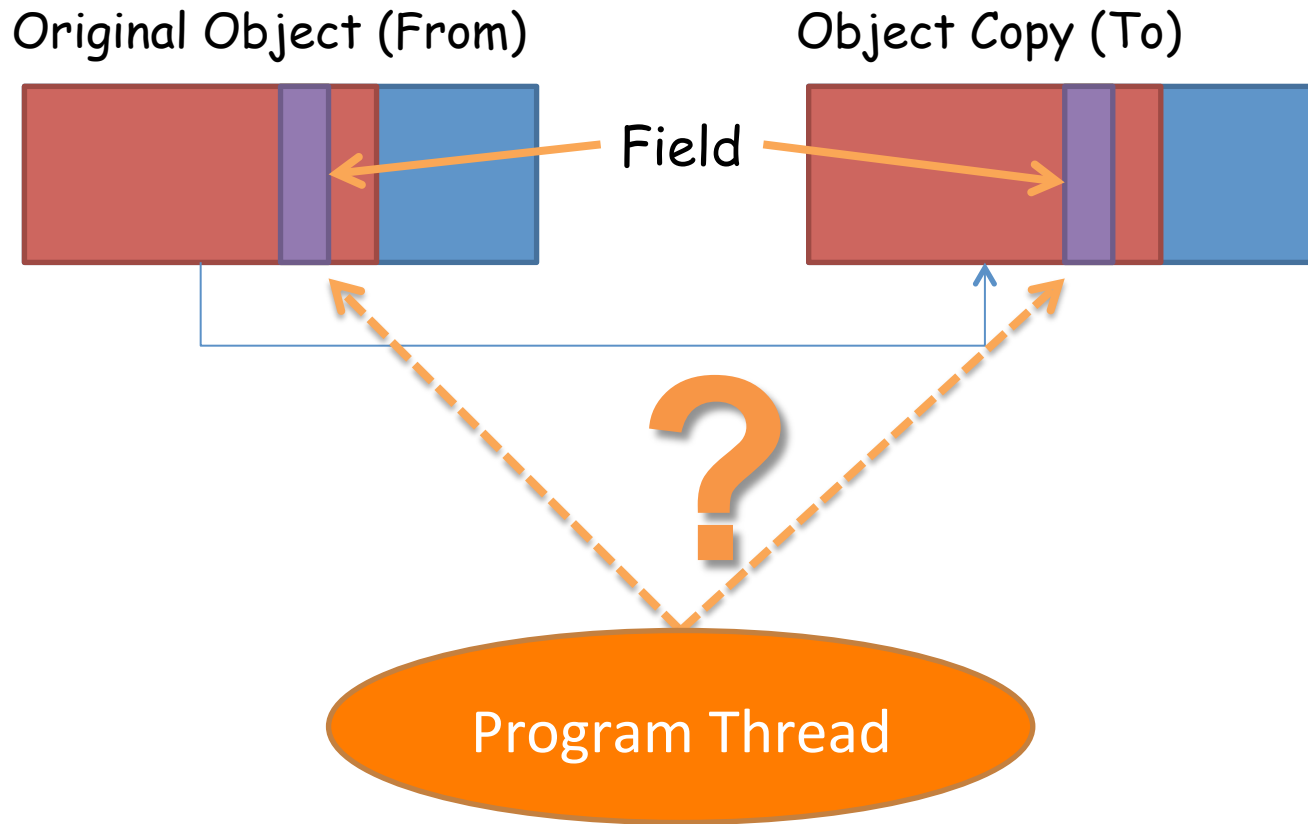
Object Copy (To)



Already Copied

Maybe, after we check which version, the status changes.

Program Thread



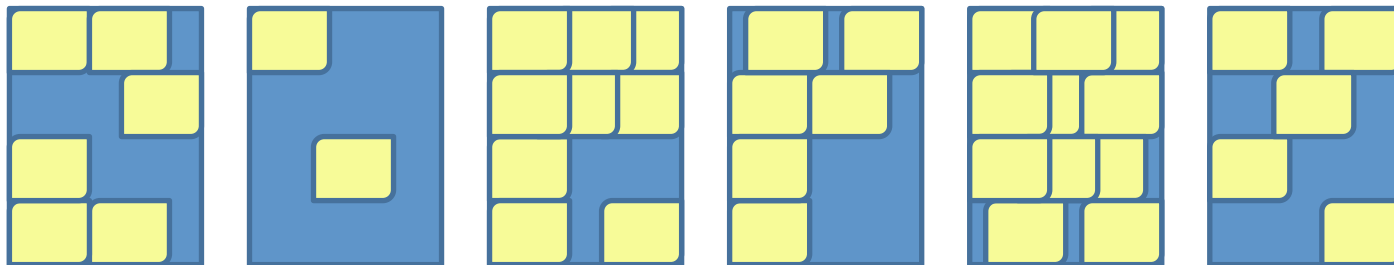
Need to
atomically access
two locations:
field and status.

The Three Collectors

- **Stopless**: employs an intermediate long object.
- **Clover**: locks with extremely small probability.
- **Chicken**: aborts copying when program interferes.
- Concentrate on concurrently moving objects.

The Overall Picture

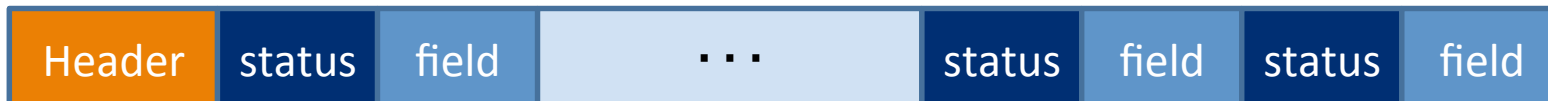
- **Concurrent mark-sweep**
 - Lock-free data structures & care for mark-stack overflow.
- **Partial Concurrent Compaction**
 - Move objects only when heap fragmented.
 - Evict fragmented pages.
 - Concurrently move objects using Stopless, Clover, or Chicken.
 - Fix pointers during next trace, or via special pass.
- **Both tasks can run concurrently and in parallel**



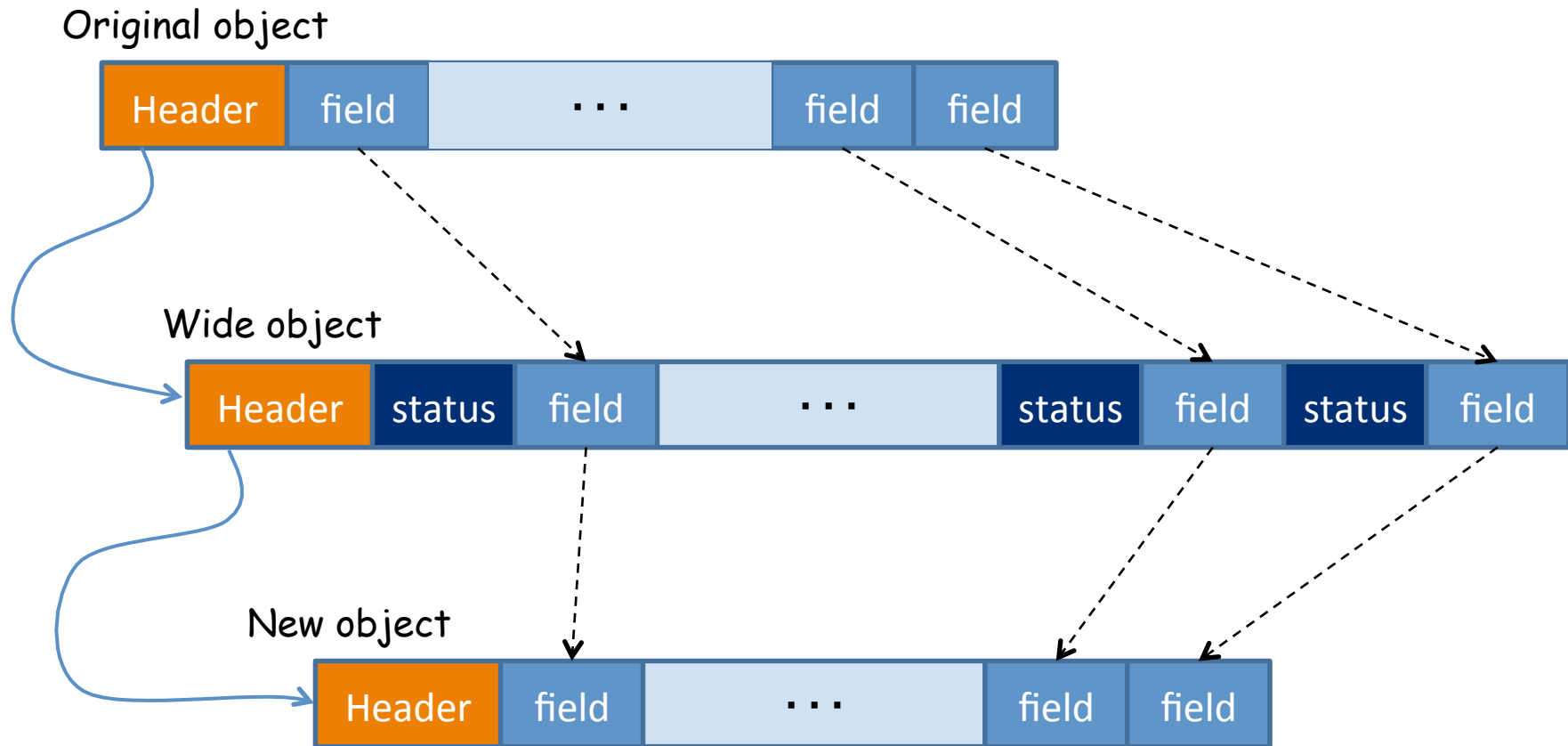
Stopless



- The basic missing item: atomic simultaneous update of
 - Data field
 - Copying status
- Solution: create a temporary "wide object".
- Each field has adjacent status field.
- Use (wide) CAS to atomically modify status & field.
 - Available on common architectures (in particular, on X64)



Wide Object Use

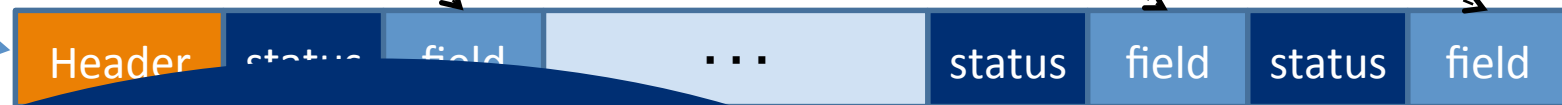


Wide Object Use

Original object



Wide object

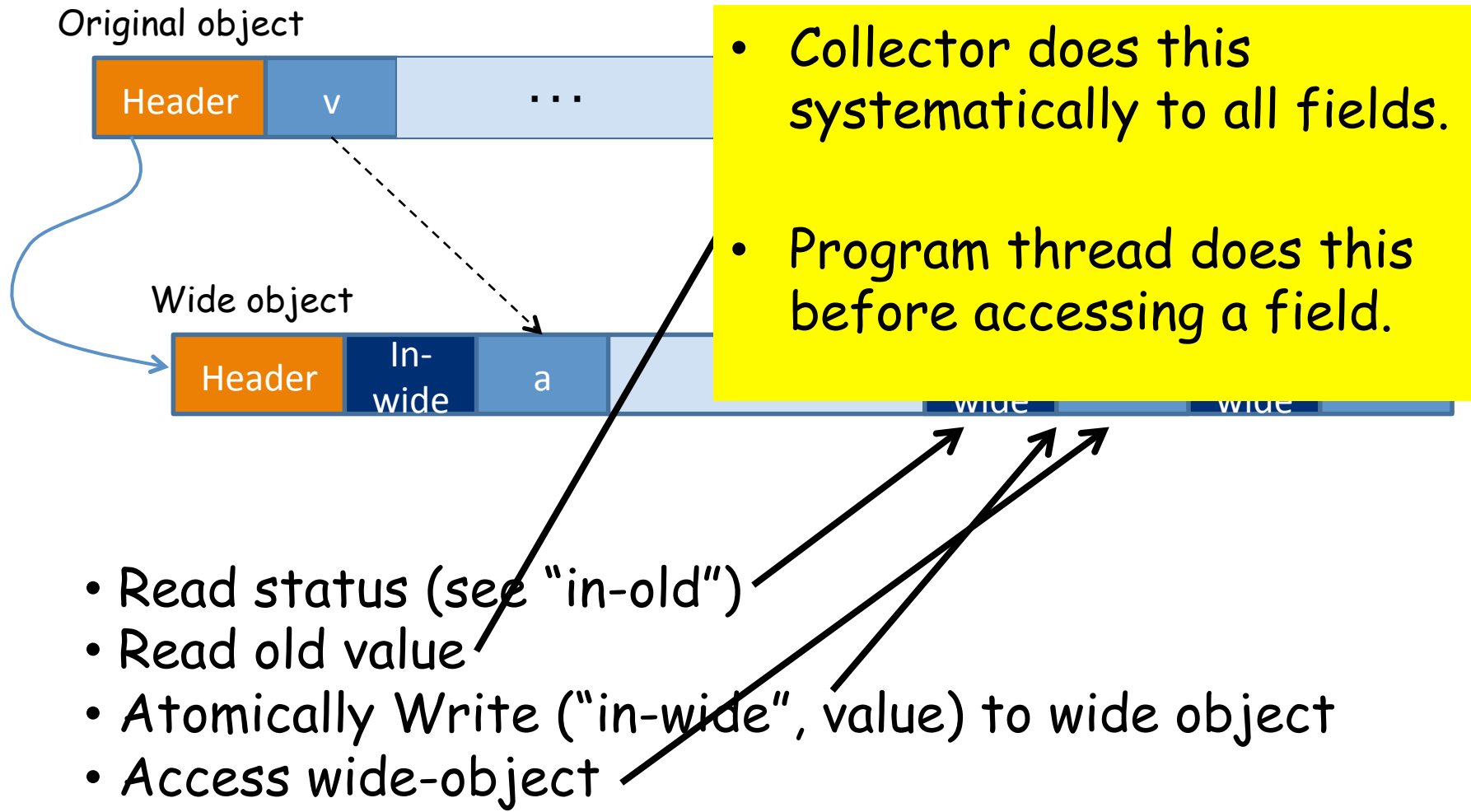


We need two concurrent copying algorithms:

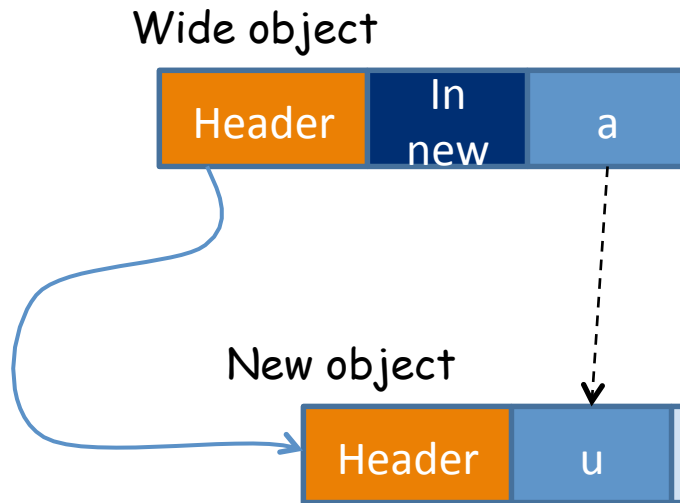
- 1) The target has a status field
- 2) The source has a status field.



Copying with Status in Target



Copying with Status in Source



- Only the collector copies fields.
- Program's writes:
 - If status "in-new" access new object.
 - Else, write to wide object.
 - If status changed to "in-new", rewrite in new.

- Copy value from wide to new object.
- Atomically Write ("in-new") assuming the value has not changed.
- If failed, try again.

Lock-Free Real-Time Copying

Status in target:

- Collector does this systematically to all fields.
- Program thread does this before accessing a field.

Status in source:

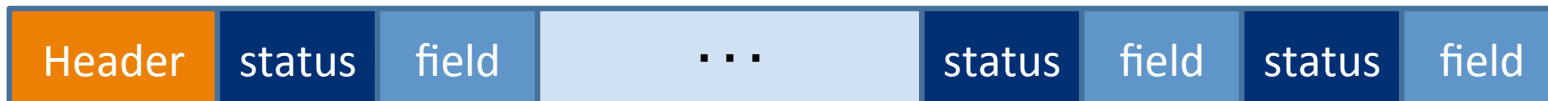
- Only the collector copies fields; and it does so systematically to all fields.

- A bounded number of operations for program barrier.
- Program never blocks (full lock-freedom).

Stopless Summary



- Employ an intermediate temporary wide object.
- Status word used to provide simultaneous access to data and status.
- Full lock-freedom, real-time (bounded barriers), and very short pauses.
- Disadvantages: time (20-80%) and space (5-45%) overheads.

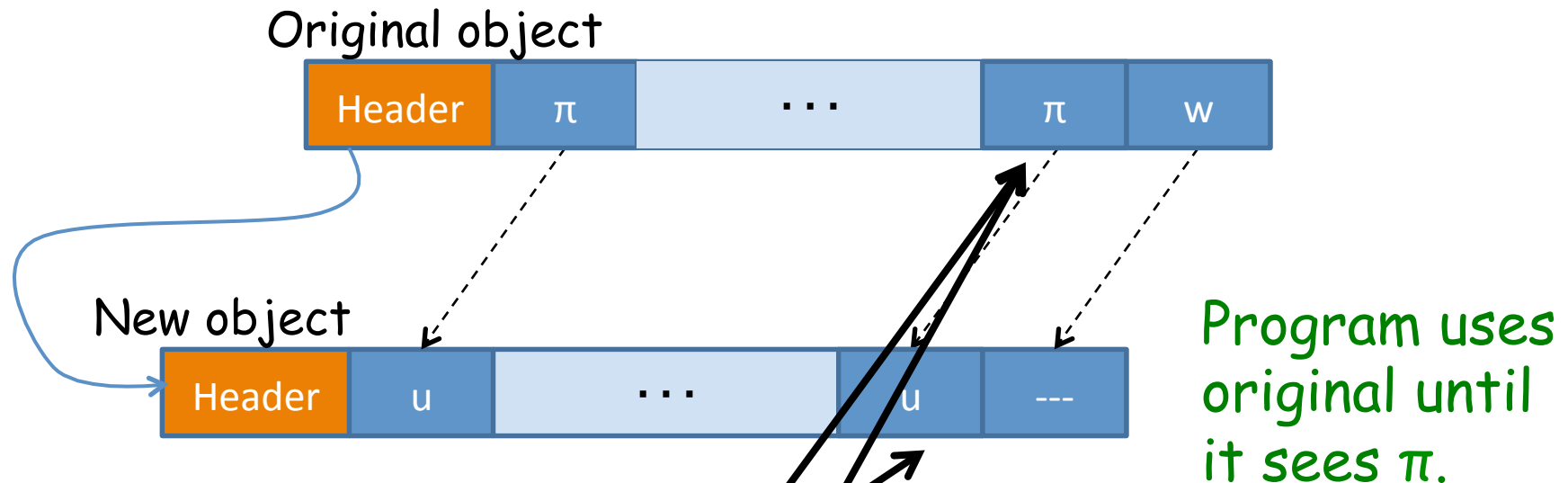


Clover



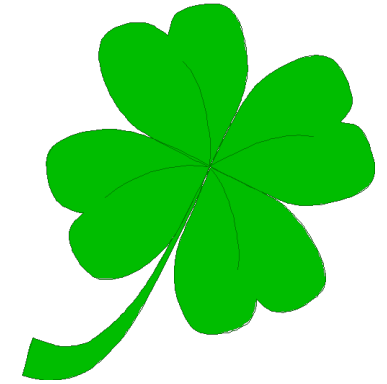
- Idea: Employ a (well defined) probabilistic approach.
- Choose uniformly at random a value π of double-word length, assume π is never written to the heap.
- For each write this happens with probability 2^{-128} or 2^{-64} .

Using a Random Marker π



- Read value in original; write it to new object.
- Atomically Write π to original assuming the value has not changed.
- If failed, try again.

Clover -- Discussion



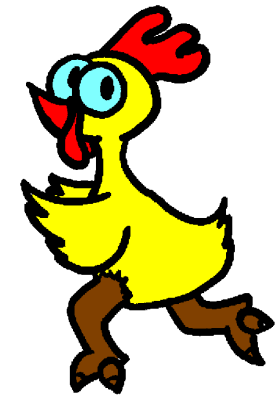
- Which probability is small enough to be acceptable for real-time systems?
- The probability that a person dies from a car crash in a single day is higher than $1/300000$ (in the U.S.).
- A hardware failure is much more likely.
- So is it enough that the program meets its deadlines with such probability?
- Clover fails to remain lock-free with prob. $\sim 10^{-20}$. Negligible for all practical purposes.
- Failure prob. is bounded independently of the input or of any other program run characteristics.

Clover -- Summary



- Advantage: no intermediate objects, quicker memory barriers.
- Disadvantage: may need to block if the program does write π to the heap.

Chicken

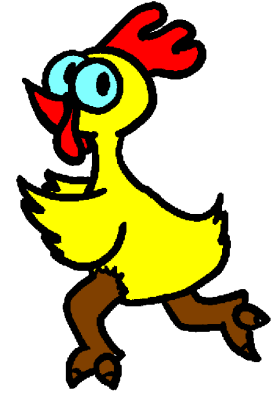


- Idea: if program touches an object during the move then the move is aborted.

- Collector copies objects one by one in the background.
- After copying an entire object, if it was not modified, atomically mark it "moved".
- Objects are copied as a whole.




- A program write starts by marking the object dirty.
- Then, checks if the object moved.
- Finally, it accesses it according to its determined location.

Chicken



- Advantage: no intermediate objects, no fall-back to locking, quick memory barriers.
- Disadvantage: may fail to move objects.

Comparison

	Non-Blocking	Time-Overhead	Space-Overhead	Fail to Copy
Stopless				 
Clover	 With very high prob.			
Chicken		 		 May fail to copy

Putting a Collector Together

The Overall Picture

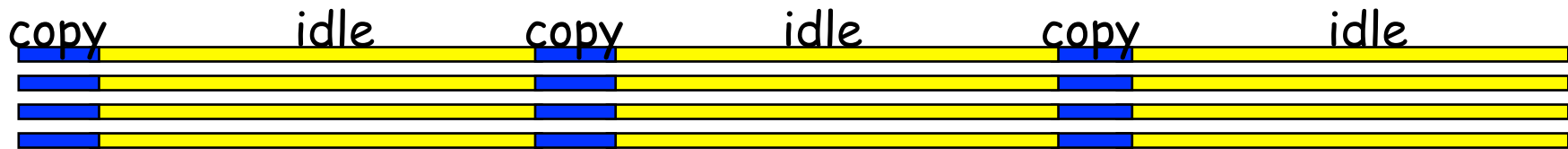
- **Concurrent mark-sweep**
 - Lock-free data structures & care for mark-stack overflow.
- **Partial Concurrent Compaction**
 - Move objects only when heap fragmented.
 - Evict fragmented pages.
 - Concurrently move objects using one of the above methods.
 - Fix pointers during next trace, or via special pass.
- **Both tasks can run concurrently and in parallel**

Real-Time Guarantees

- Collection activity is concurrent.
- Program only cooperates via:
 - Bounded read- and write-barriers.
 - Phase change cooperation (to be discussed).
 - Stack scanning (to be ignored for lack of time).
- Assumption: collector finishes on time
 - Scheduling issues which depend on program characteristics.

Phases

- In order to avoid always using a heavy barrier, the collector and program act in phases. E.g.,
 - Idle: no collection activity,
 - Copy: objects are being moved.



- Blocking option: stop all threads to notify them of phase change.
- Better option: handshakes.

Program & Collector Cooperation

- The program threads find about phase changes via **handshakes**.
- Once in a while, mutator checks if the phase has changed.

■ Prep phase
■ Copy phase
■ Idle phase



- *Gain*: no need for simultaneous blocking halt.
- *Loss*: fuzzy move from phase to phase; algorithm more involved.

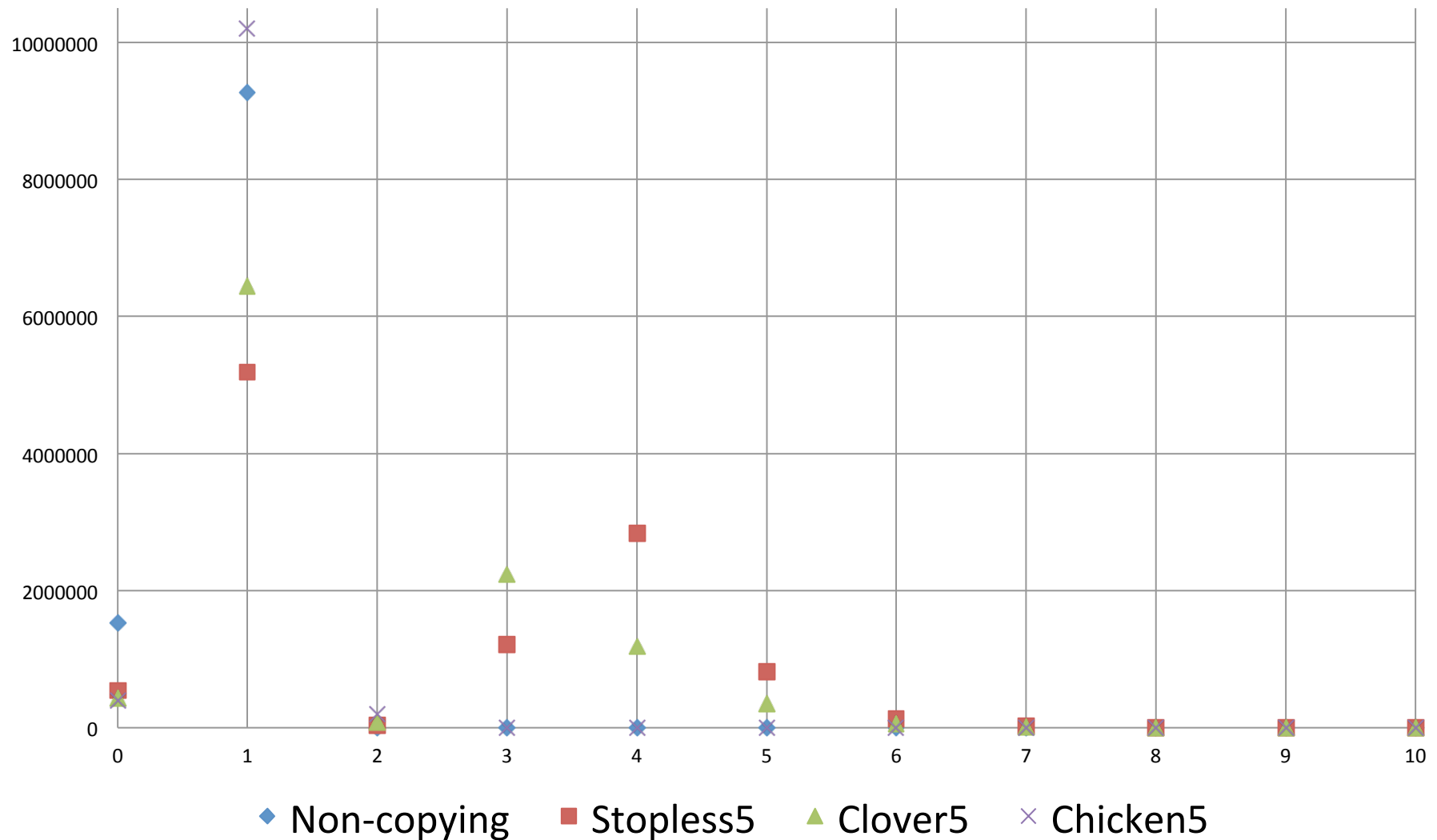
We do not cover these concurrency challenges in this lecture.

Some Results

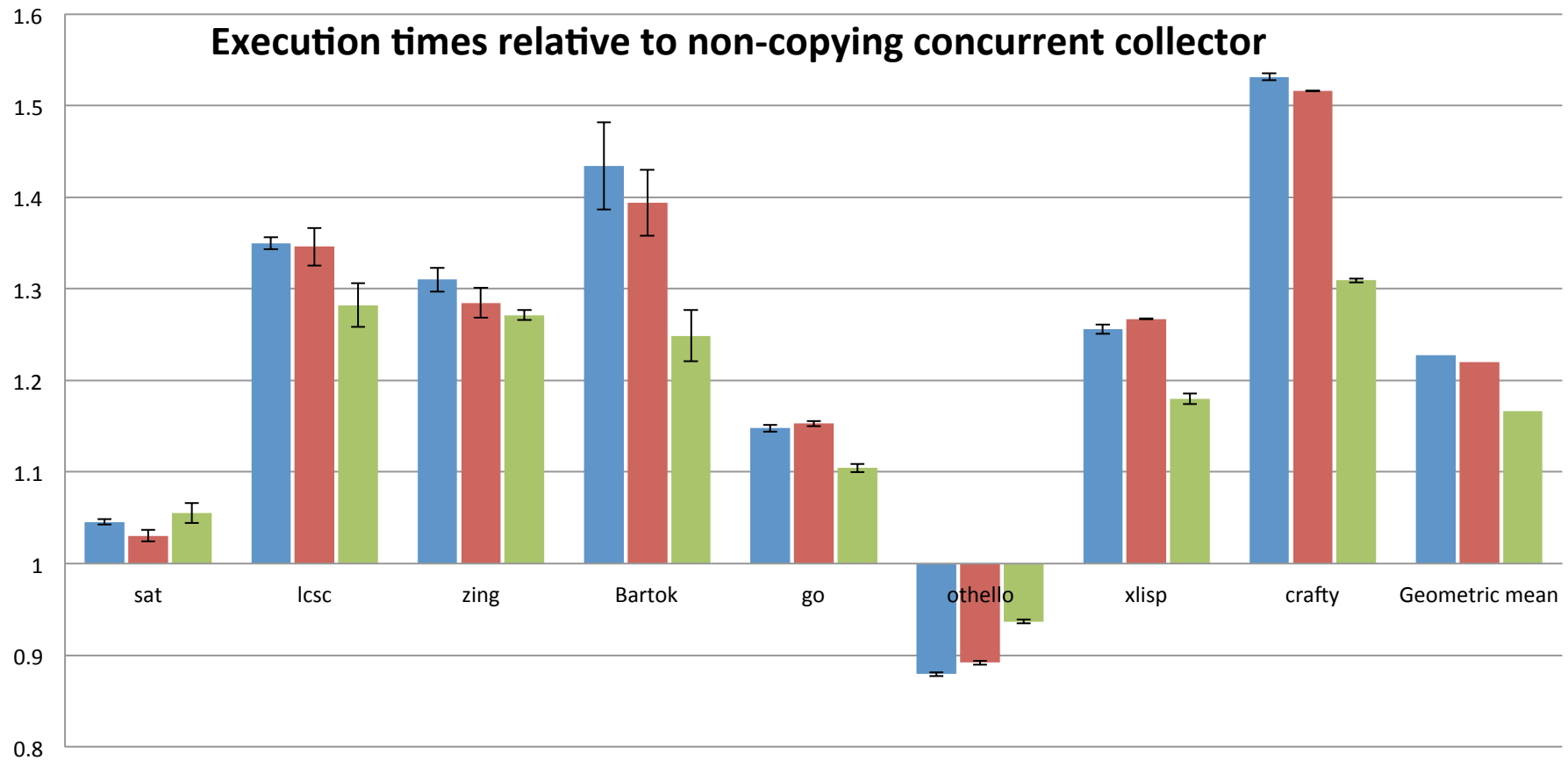
Checking Response Times

- We ran a task of copying 64 reference values.
- A competing parallel thread repeatedly allocates a 400MB data structure with over a million objects.
- A task was scheduled every $9.26\mu\text{s}$ (108KHz)

Response Time Histogram

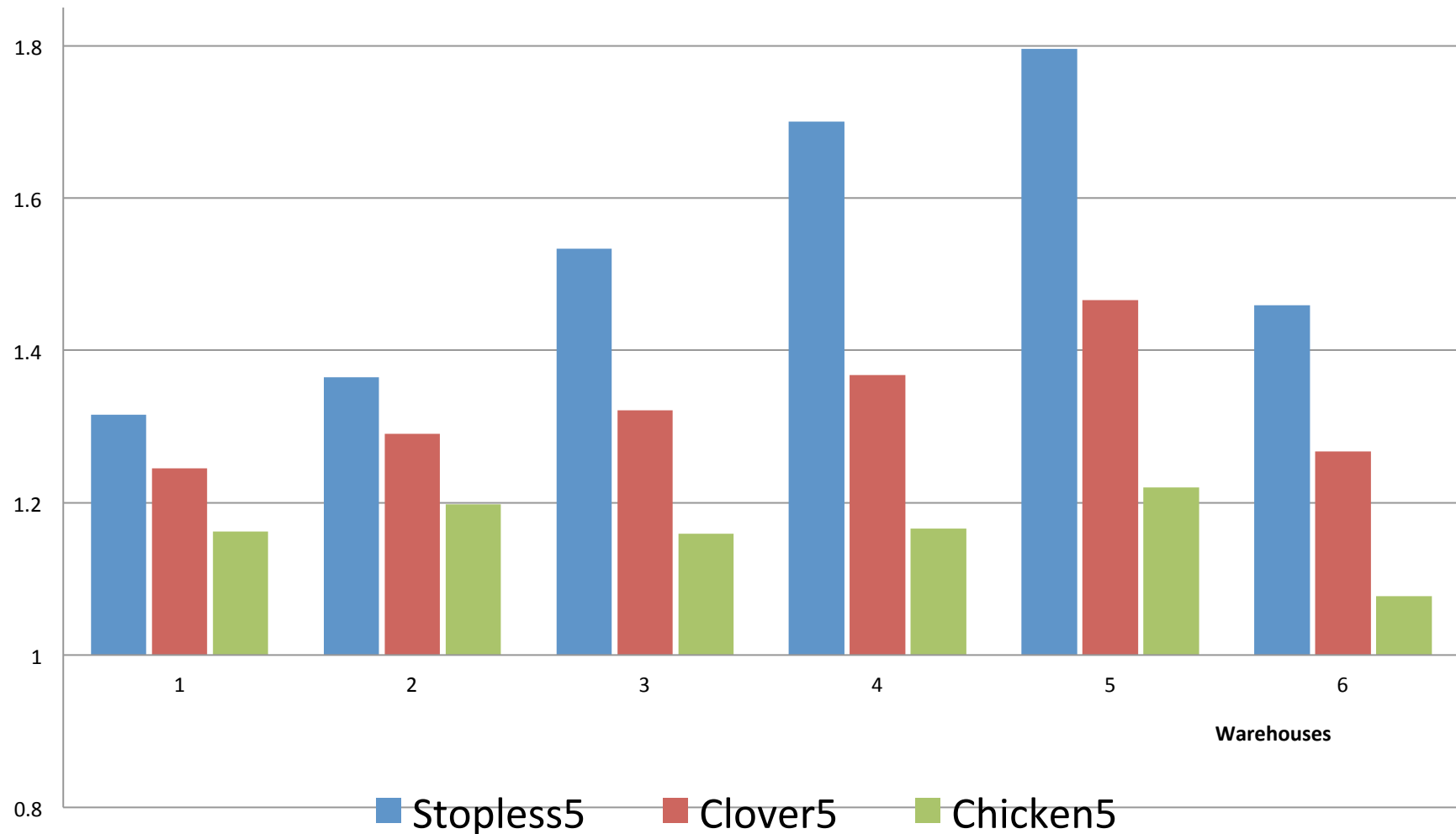


Throughput Overhead

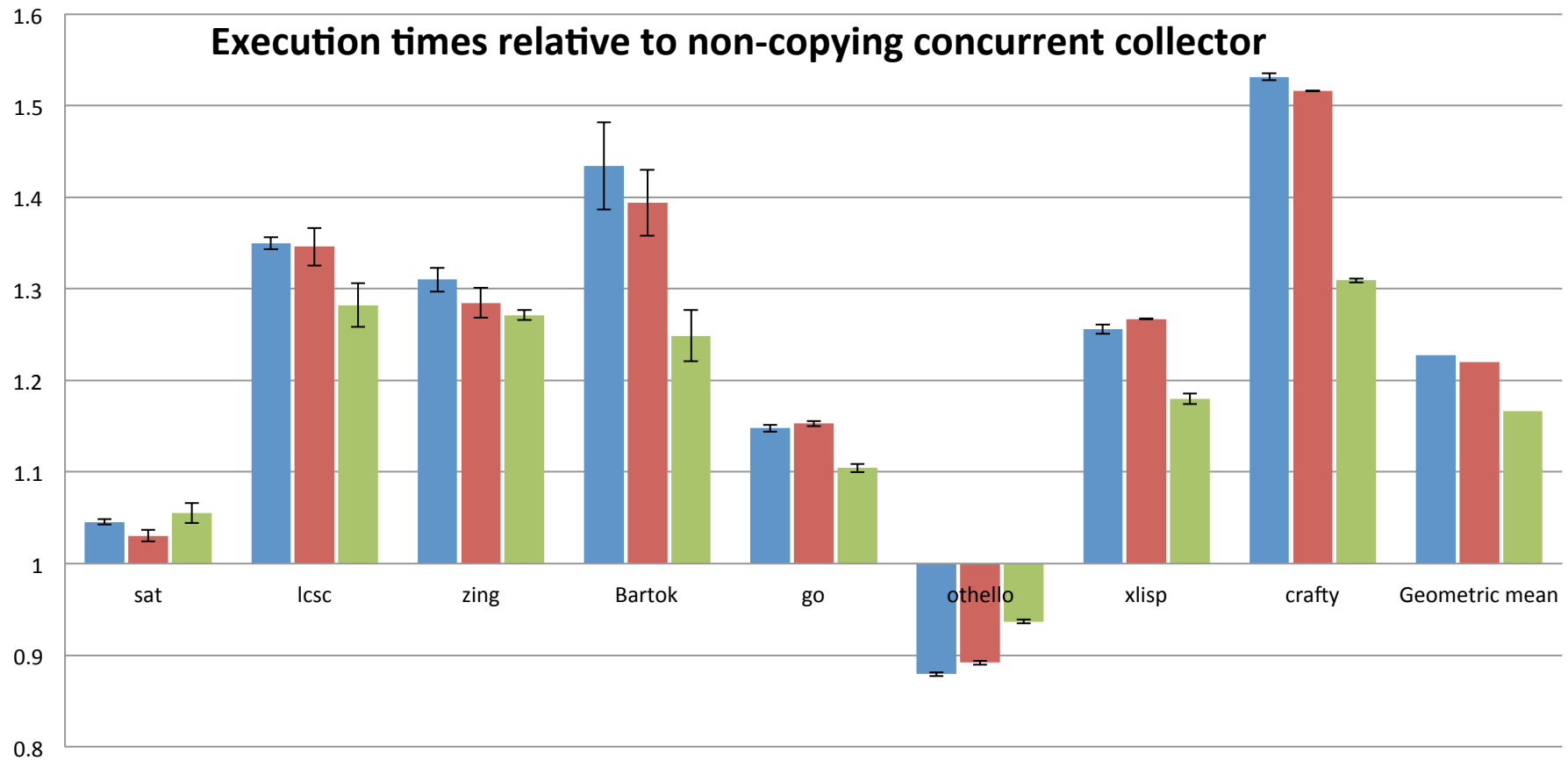


■ Stopless5 ■ Clover5 ■ Chicken5

Relative JBB Throughput

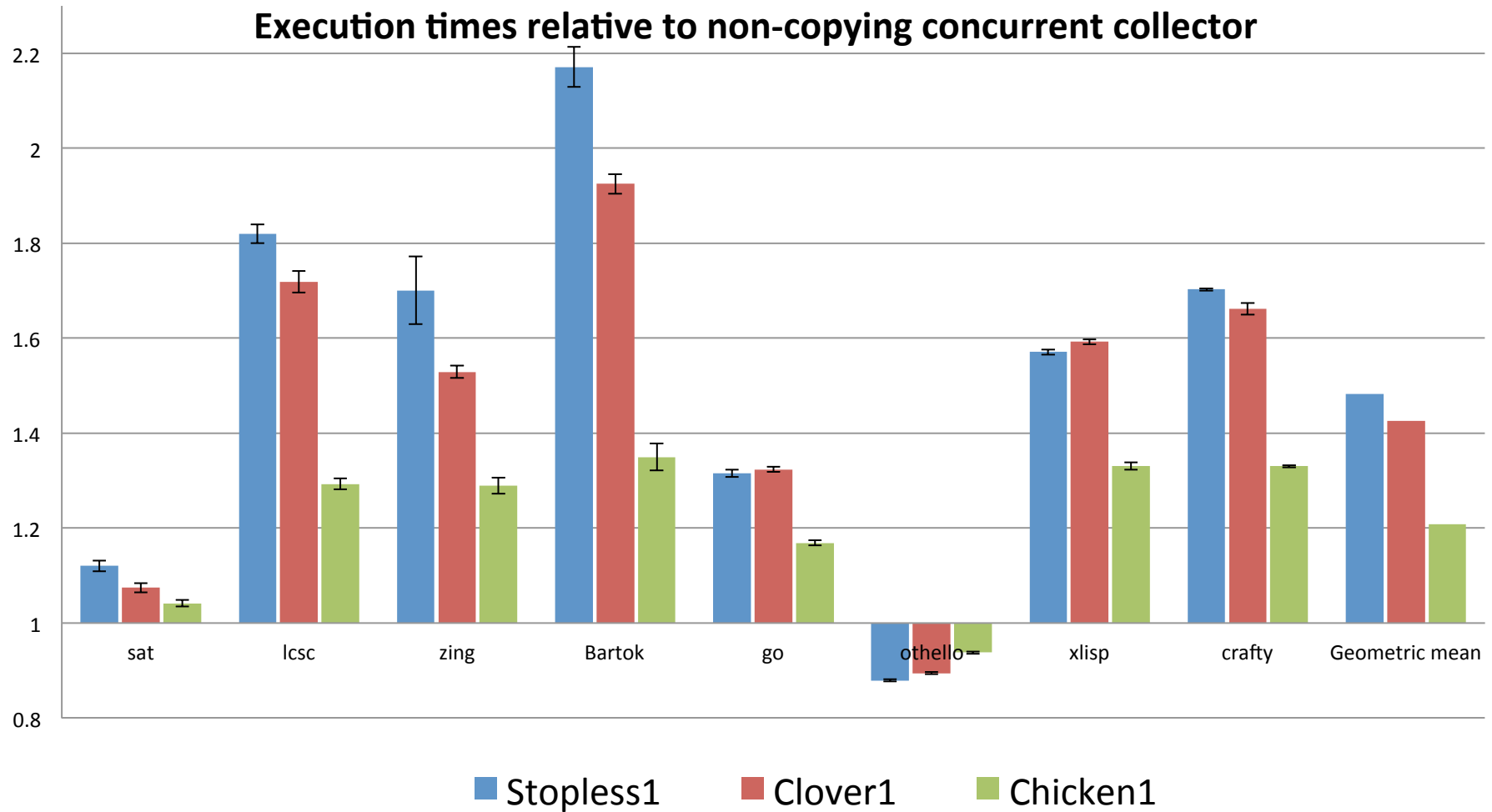


Throughput Overhead

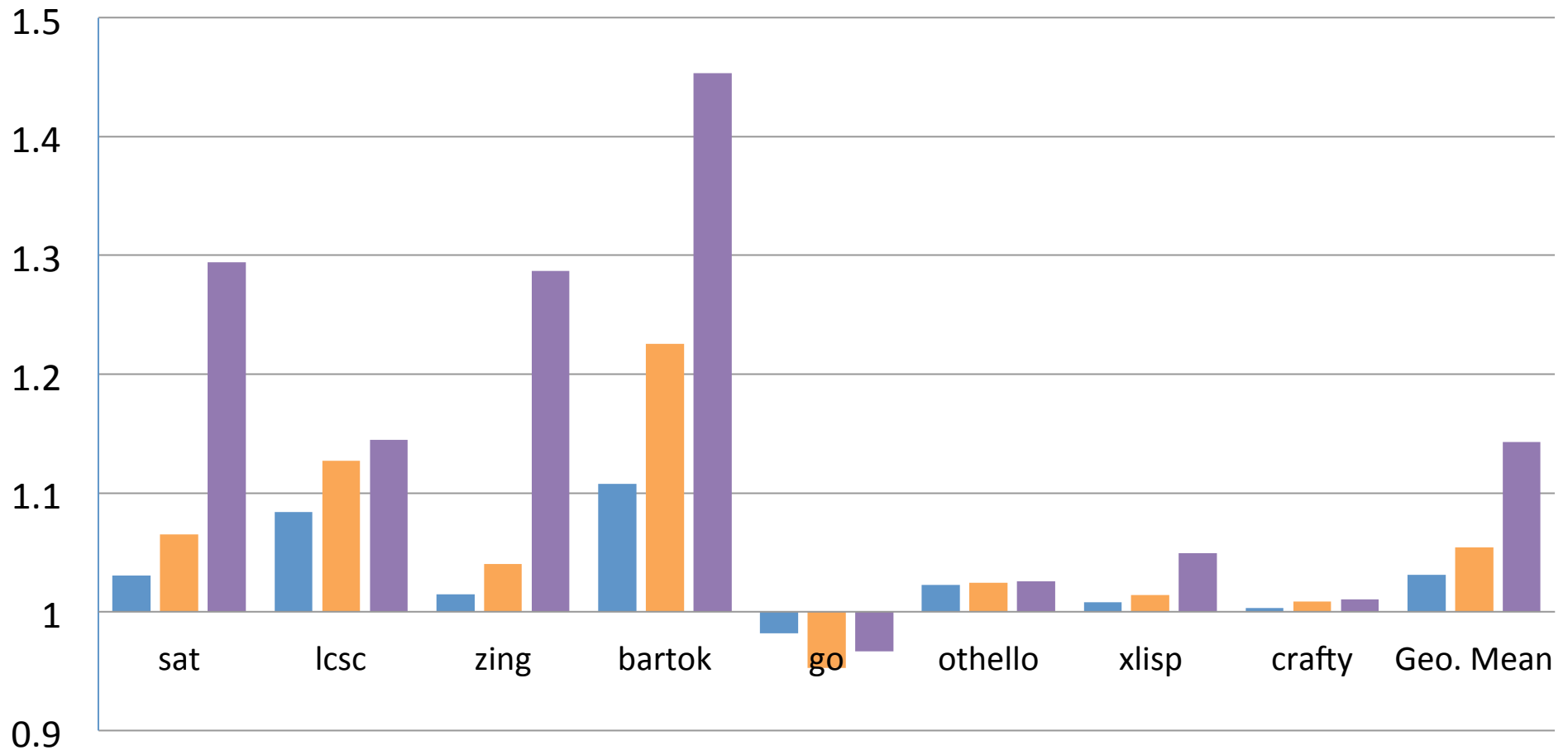


■ Stopless5 ■ Clover5 ■ Chicken5

Throughput Stressed



Space Overheads for Stopless



■ First Header Word Overhead

■ Second Header Word Overhead

■ Total Overhead, Including To-Space Objects, Wide Objects, and Meta-data

Related Work

- Various other solutions exist.
- It is possible to use lock-free concurrent mark-sweep collectors, but **fragmentation** is not solved.
- We mentioned Baker.
- [Herlihy-Moss 91]: Create a new copy of the object per update. (**Lock-free but with unbounded overhead.**)
- Sapphire and the Cheng-Blelloch collectors move objects concurrently but grab a lock when moving shared objects.
- Azul has a nice product, but pauses may get to 100 ms

Conclusion

- Allowing development of real-time software on high-level languages is important.
- A major obstacle is garbage collection pauses.
- Need partial compaction, lock-freedom, highly responsive collector.
- We've discussed IBM's Metronome and Microsoft's concurrent solutions.