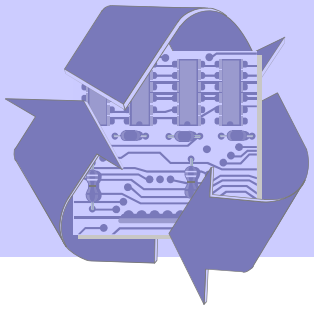


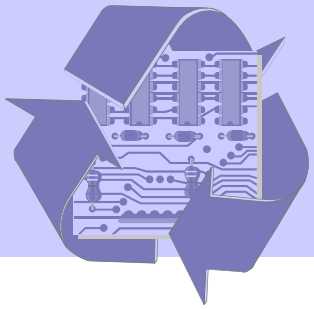
November 2010, London

Dynamic Memory Management

Challenges for today and tomorrow

Richard Jones
School of Computing
University of Kent
http://www.cs.kent.ac.uk/~rej

University of Kent



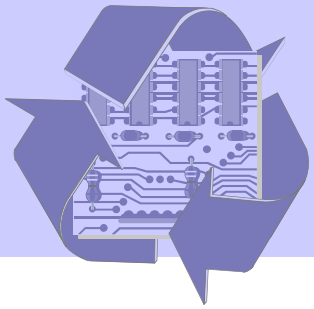
Overview

Part 1: Introduction, basic algorithms, performance

Part 2: Parallel: allocation, tracing and moving

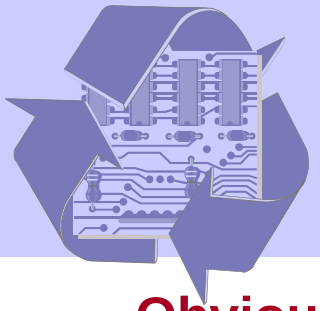
Part 3: Concurrent: reference counting, tracing and moving

Part 4: Real-time GC



PART 1: Introduction

- Why garbage collect?
- Basic algorithms
- Performance v. malloc/free



Why garbage collect?

Obvious requirements

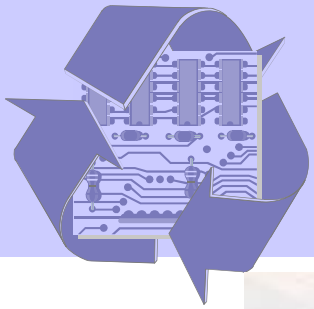
- Finite, limited storage.
- Language requirement — objects may survive their creating method.
- The problem — hard/impossible to determine when something is garbage.

Programmers find it hard to get right.

- Too little collected? memory leaks.
- Too much collected? broken programs.

Good software engineering

- **Explicit memory management conflicts with the software engineering principles of abstraction and modularity.**



Basic algorithms

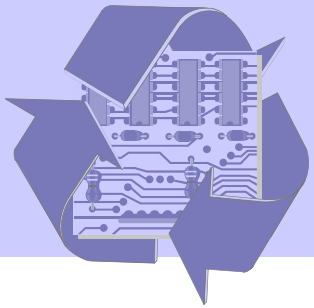
—The garage metaphor—

Reference counting: Maintain a note on each object in your garage, indicating the current number of references to the object. When an object's reference count goes to zero, throw the object out (it's dead).

Mark-Sweep: Put a note on objects you need (roots). Then recursively put a note on anything needed by a live object.
Afterwards, check all objects and throw out objects without notes.

Mark-Compact: Put notes on objects you need (as above). Move anything with a note on it to the back of the garage.
Burn everything at the front of the garage (it's all dead).

Copying: Move objects you need to a new garage. Then recursively move anything needed by an object in the new garage.
Afterwards, burn down the old garage (any objects in it are dead)!



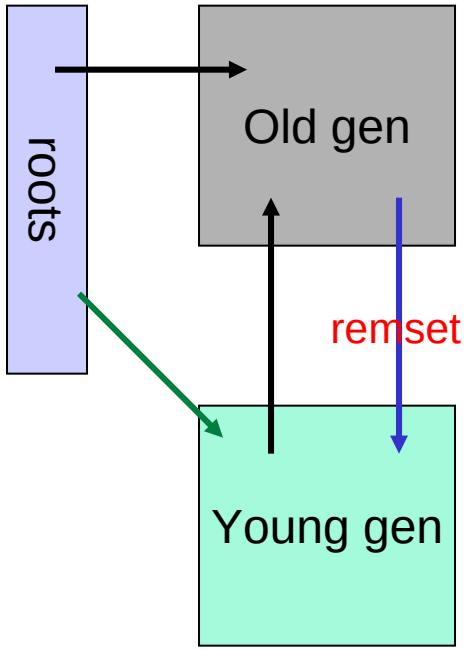
Generational GC

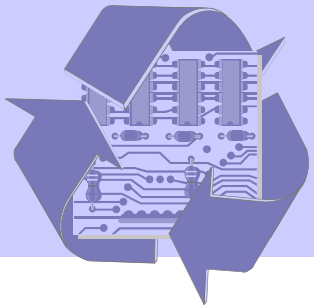
Weak generational hypothesis

- "Most objects die young" [Ungar, 1984]
- Common for 80-95% objects to die before a further MB of allocation.

Strategy:

- Segregate objects *by age* into **generations (regions of the heap).**
- Collect different generations at different frequencies.
  - so need to "remember" pointers that cross generations.
- Concentrate on the nursery generation to reduce pause times.
  - full heap collection pauses 5-50x longer than nursery collections.





Generational GC: a summary

Highly successful for a range of applications

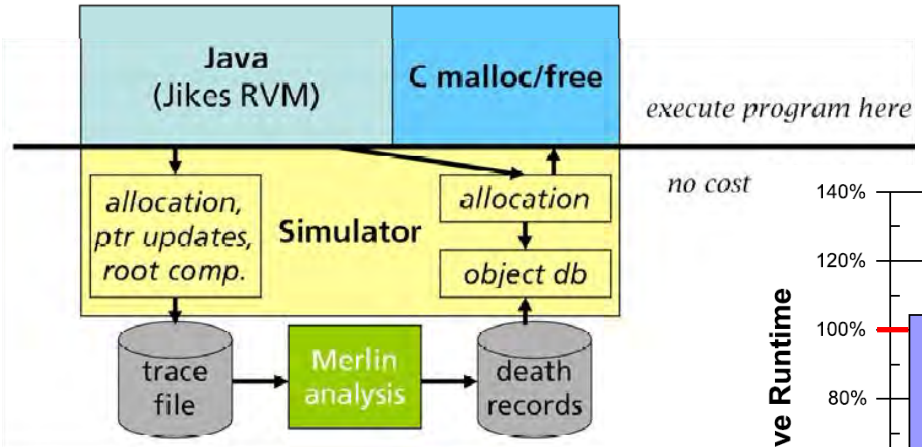
- ✓ acceptable pause time for interactive applications.
- ✓ reduces the overall cost of garbage collection.
- ✓ improves paging and cache behaviour.

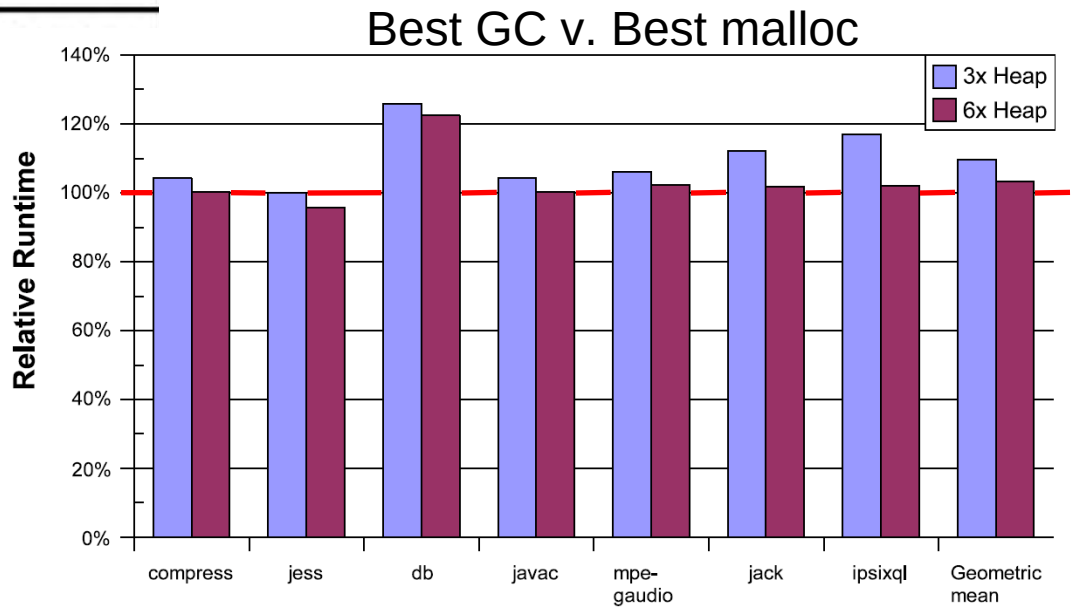
Requires a low survival rate, infrequent major collections, low overall cost of write barrier

But generational GC is not a universal panacea.
It improves expected pause time but not the worst-case.

- ✗ objects may not die sufficiently fast
- ✗ applications may thrash the write barrier
- ✗ too many old-young pointers may increase pause times
- ✗ copying is expensive if survival rate is high.

Can GC perform as well as malloc/free?

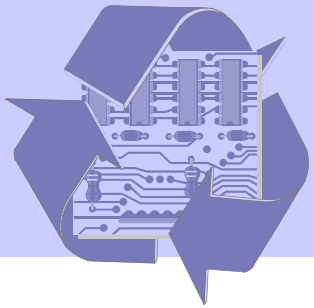




Test environment

- Real Java benchmarks + *JikesRVM* + real collectors + real malloc/frees.
- Object reachability traces: provide an 'oracle'.
- *Dynamic SimpleScalar* simulator: count cycles, cache misses, etc.

Quantifying the Performance of Garbage Collection v. Explicit Memory Management
Hertz and Berger. OOPSLA'05.



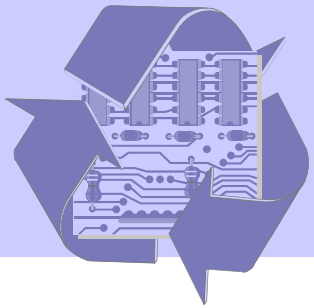
PART 2: The multicore, multiprocessor challenge

The modern environment:

- Multicore processors are ubiquitous
- Multiprocessors are common
- Heaps are large

Exploit parallel hardware

- Concurrent mutator threads (allocation)
- Parallel GC threads
- Concurrent mutator and GC threads



Parallelism

Avoid bottlenecks

Heap contention — allocation.

Tracing, especially contention for mark stack.

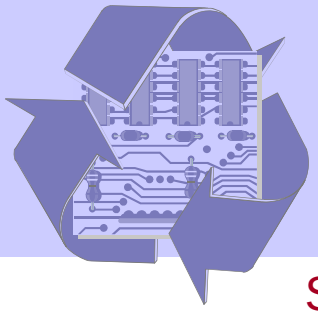
Compaction — address fix-up must appear atomic.

Load balancing is critical

Work starvation *vs.* excessive synchronisation

Over-partition work, work-share

Marking, scanning card tables / remsets, sweeping, compacting



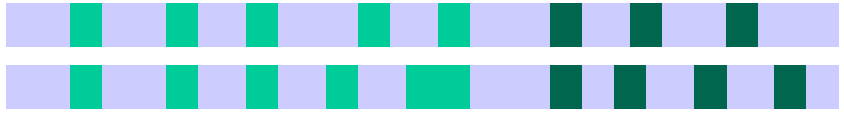
Terminology

Single threaded collection

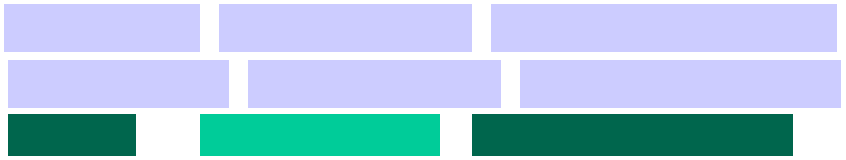
Parallel collection

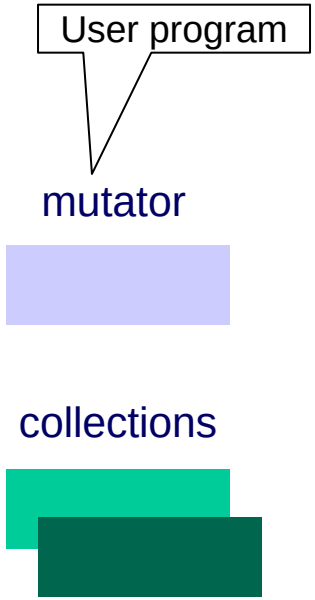
Concurrent collection

Incremental collection

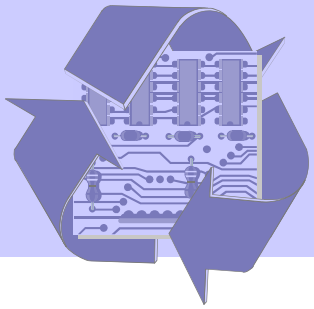


On-the-fly concurrent collection





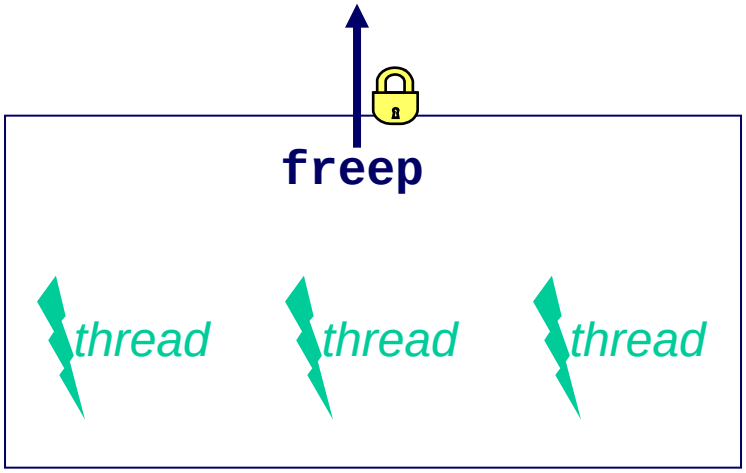
and all combinations…

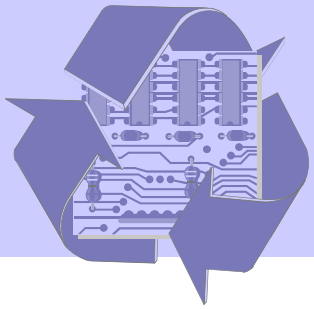


A. Concurrent allocation

Multiple user threads

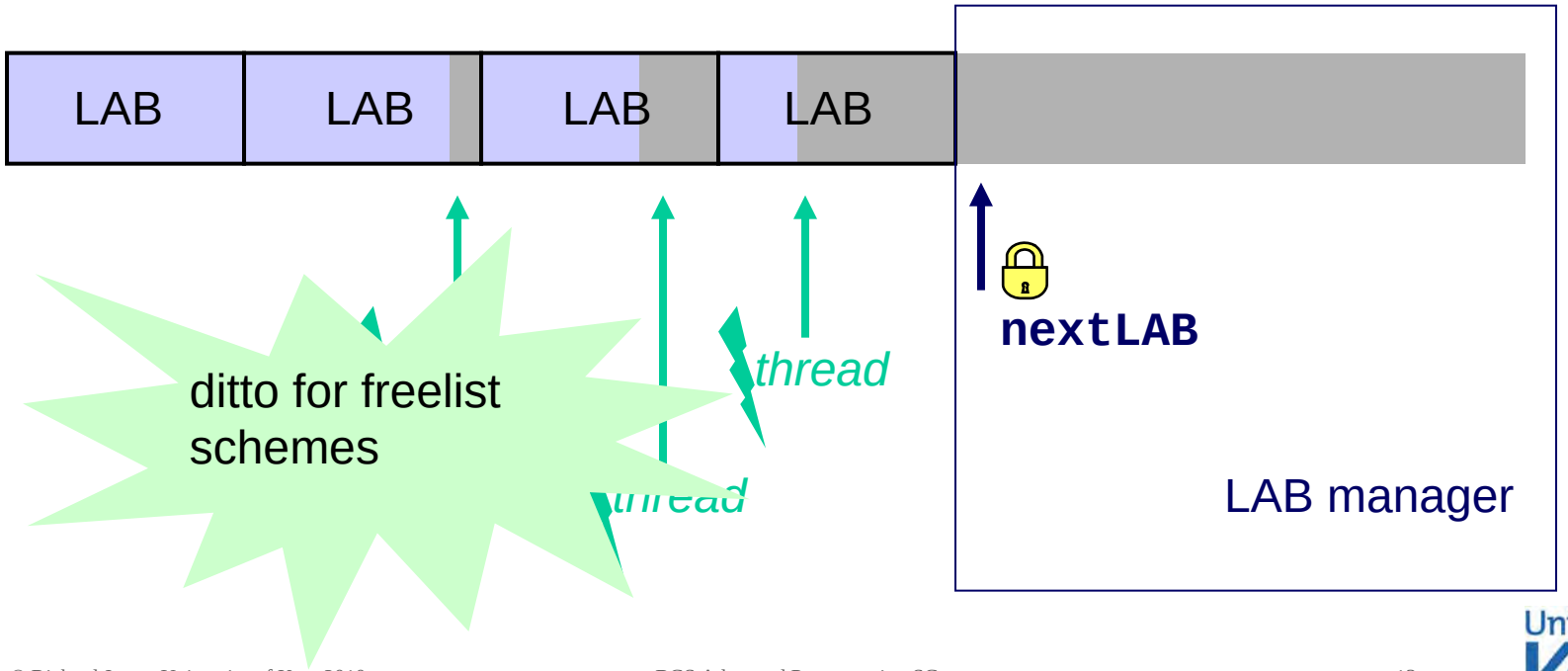
- Must avoid contention for the heap
- Avoid locks, avoid atomic instructions (CAS)

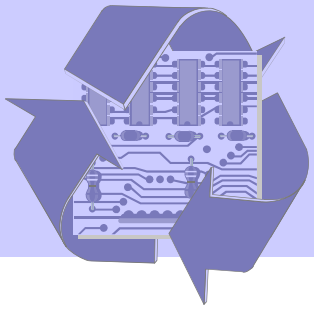




Local Allocation Blocks

- Thread contends for a contiguous block (LAB) — CAS.
- Thread allocates within LAB (bump a pointer) — no contention.
- Locality properties make this effective even for GCs that rarely move objects — needs variable-sized LABs.

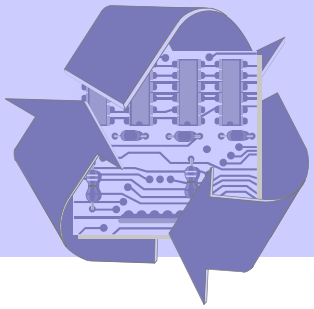




Exploiting parallel hardware

Issues

- Load balancing (dynamic)
- Synchronisation
- Processor-centric or memory-centric approach



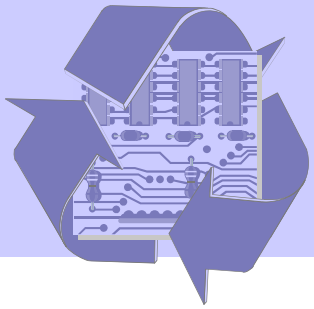
B. Parallel marking

The goal is always to avoid contention (e.g. for the mark stack) yet to balance loads.

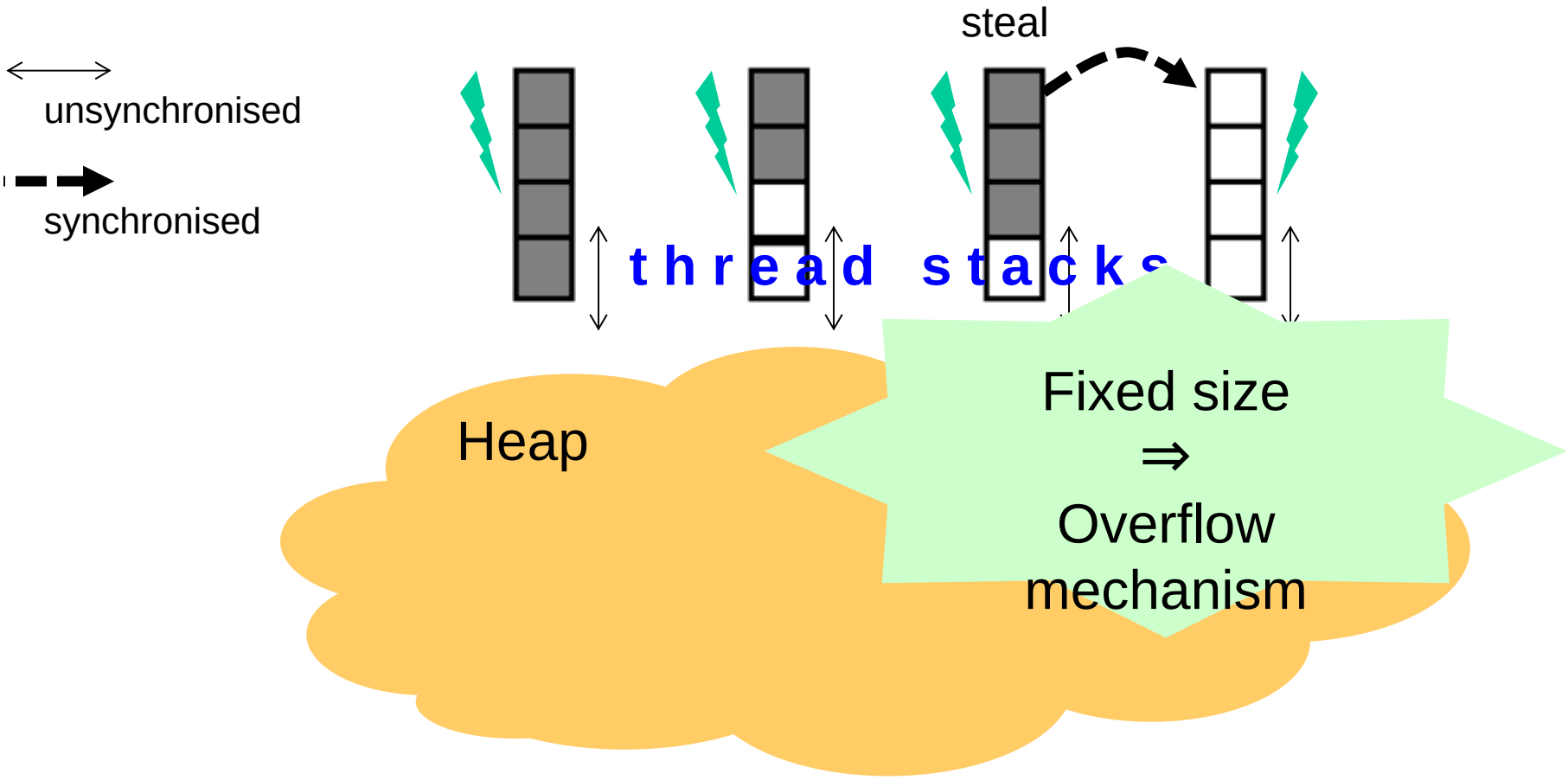
Thread-local mark stacks.

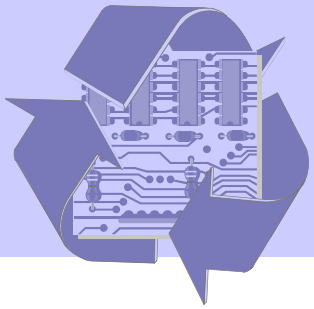
Work-stealing.

Grey packets.

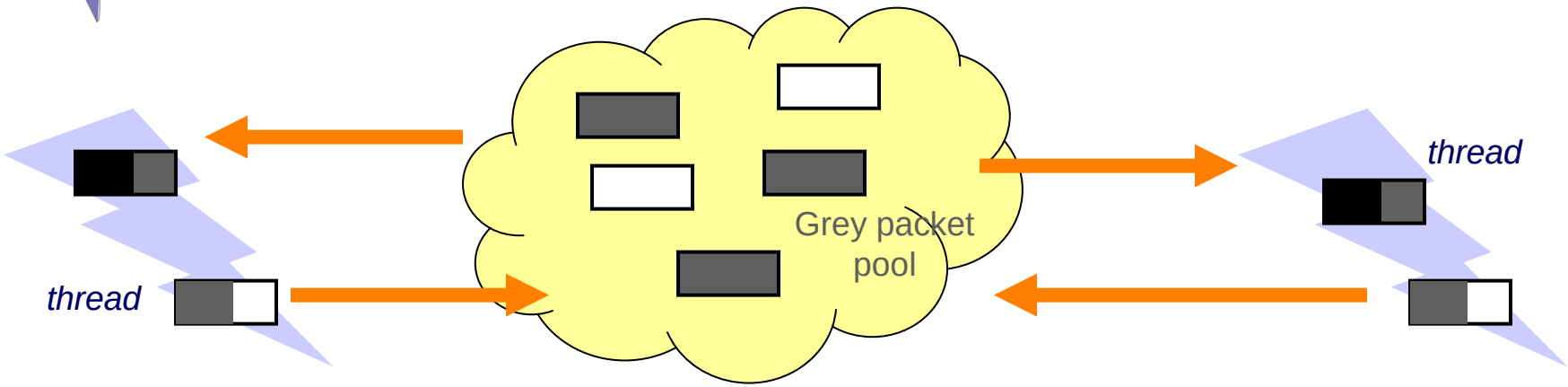


Work stealing

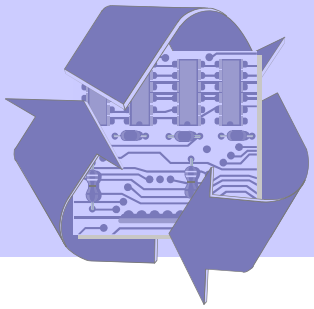




Grey packets



1. Acquire a full packet of marking work (grey references).
2. Mark & empty this packet and fill a fresh (empty) packet with new work.
3. Return full packets returned to the pool.
   - Avoids most contention
   - Simple termination.
   - Simplifies prefetching (cf. a traditional mark stack).
   - Reduces processor weak ordering problems (fences only around packet acquisition/disposal).



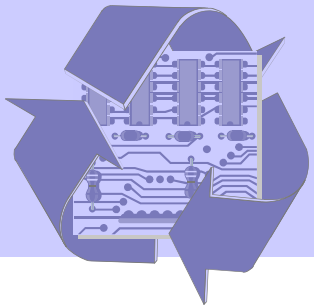
C. Parallel compaction

Without compaction

- Heaps tend to fragment over time.

Issues for compaction:

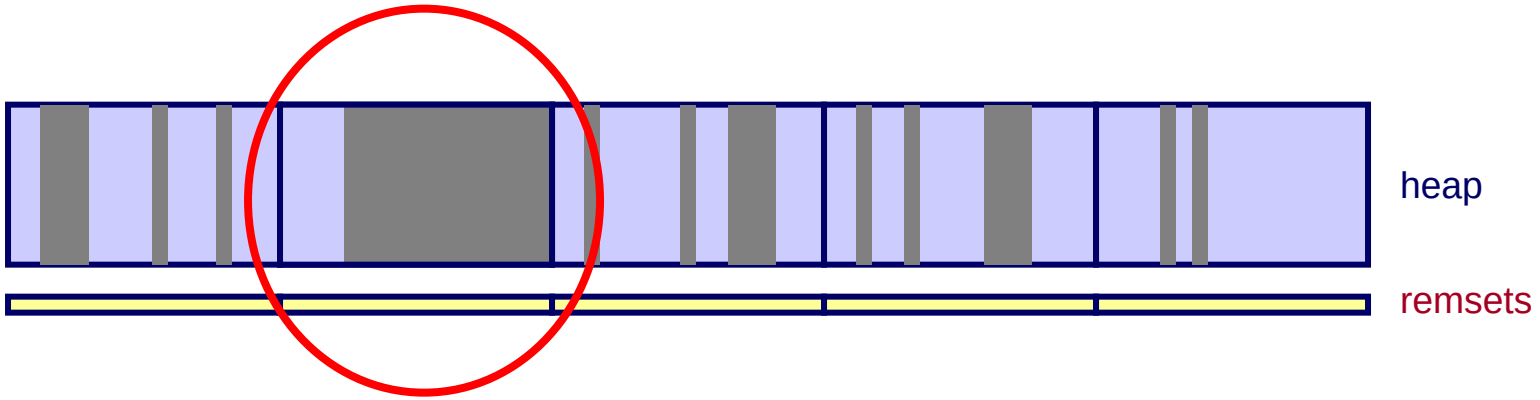
- Moving objects.
- Updating all references to a moved object…
- In parallel…
- (And concurrently…)



Compaction strategy

Old Lisp idea recently applied to parallel systems.

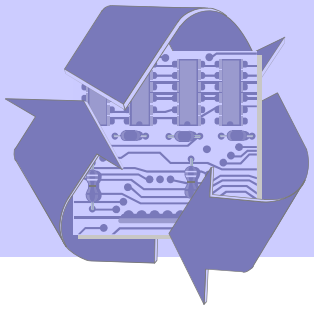
Divide the heap into a few, large regions:



- Marker constructs remsets
  - locations with references into each region.
- Heuristic for condemned region (live volume/number/references) .
- Use remsets to fix-up references.

e.g. *Garbage-First Garbage Collection*
Detlefs et al, ISMM04

University of Kent



Parallel compaction

Split heap into

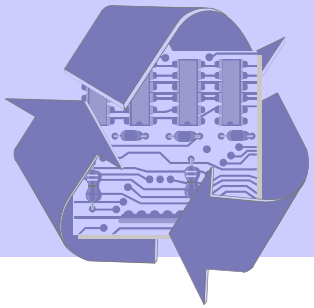
- Many small blocks (e.g. 256b).
- Fewer, larger target areas (e.g. 16 x processors, each 4MB +).

Each thread

1. increments index of next target to compact (e.g. CAS).
2. claims a target area with a lower address (e.g. CAS).
3. moves objects/blocks in the next block en masse into target area.

An Efficient Parallel Heap Compaction Algorithm,
Abuaiadh et al, OOPSLA'04

University of Kent



Results

Reduced compaction time (individual objects)

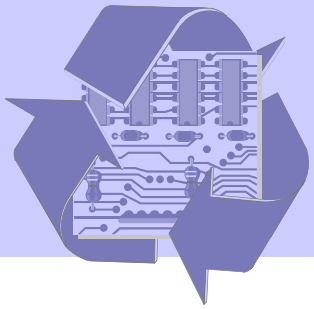
- from 1680 ms to 470 ms
  for a large 3-tier application suffering fragmentation problems.

Moving blocks rather than individual objects

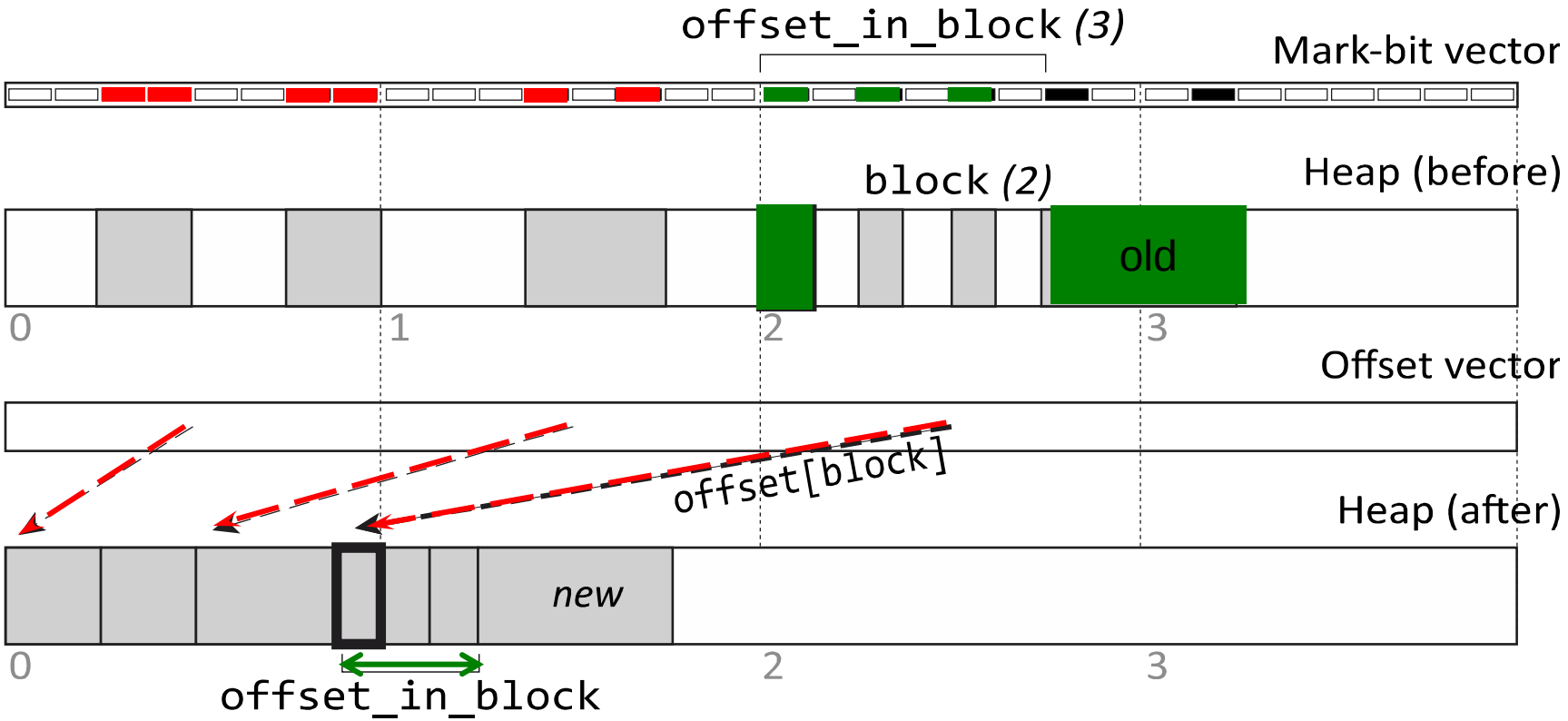
- further reduces compaction time by 25%
- at a small increase in space costs (+4%).

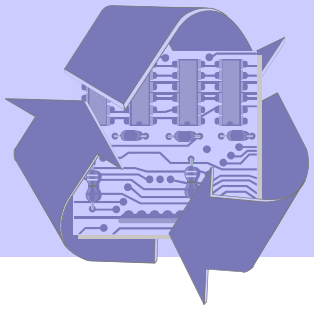
Compaction speed up is linear in number of threads.

Throughput increased slightly (2%).



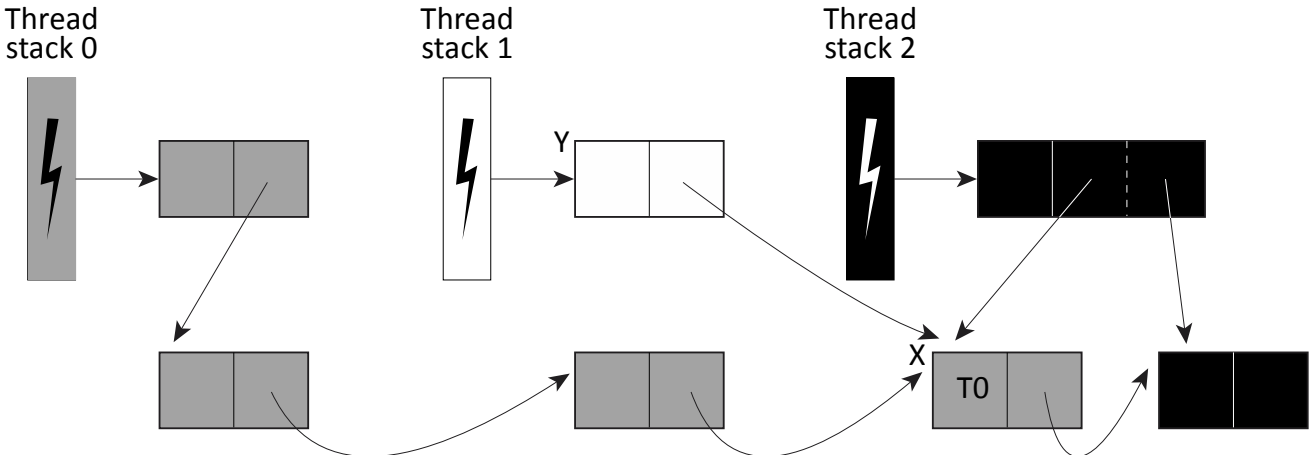
Compressor


offset_in_block (3)
Mark-bit vector
block (2)
Heap (before)
old
0
1
2
3
Offset vector
offset[block]
Heap (after)
new
0
2
3
offset_in_block



D. Parallel copying

A processor-centric approach



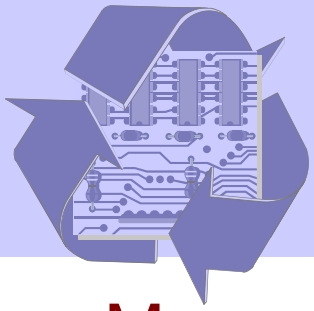
NUMA architecture

- Each memory segment mapped to a single processor
- Evacuate objects to preferred processor

‘Dominant’ thread

1. Reached from a thread stack
2. Locked/reserved by a thread
3. Same as parent

NUMA-aware memory manager with dominant-thread-based copying GC. Ogasawara OOPSLA’09



Parallel copying

Memory-centric approaches

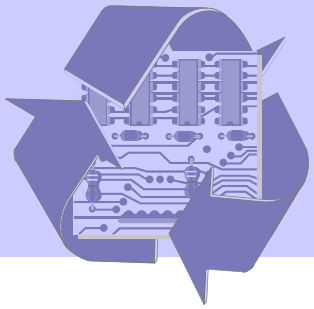
scan reaches
end of block

reset scan

global pool

Performance boost: Add a 1-block cache between each thread and the global pool.

Evaluation of parallel copying garbage collection on a shared-memory multiprocessor.
Imai and Tick. Transactions on Parallel and Distributed Systems, 1993



F: Reference counting

Problems

- Cycles
- Overheads
- Atomic RC operations

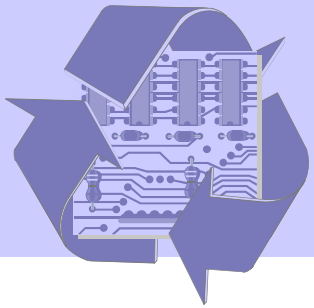
Benefits

- Distributed overheads
- Immediacy
- Recycling memory

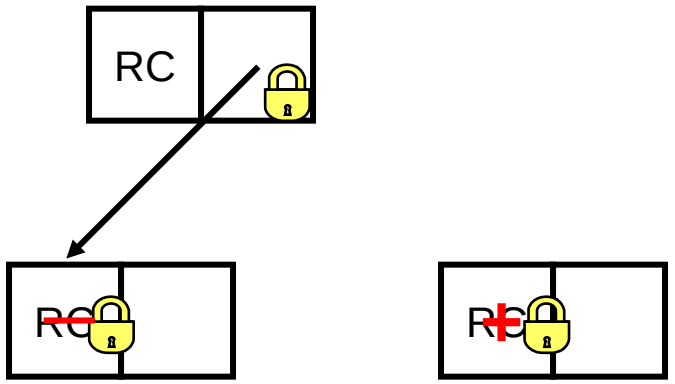
Observations

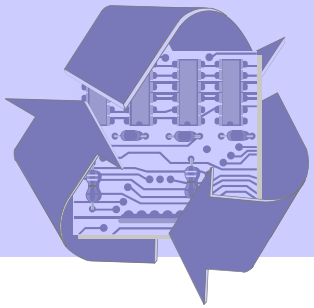
- Practical solutions use Deferred RC
  - don't count local variable ops
  - periodically scan stack

- RC and tracing GC are duals
  - GC traces live objects
  - RC traces dead objects
  - *Unified Theory of Garbage Collection*, Bacon et al, OOPSLA'04

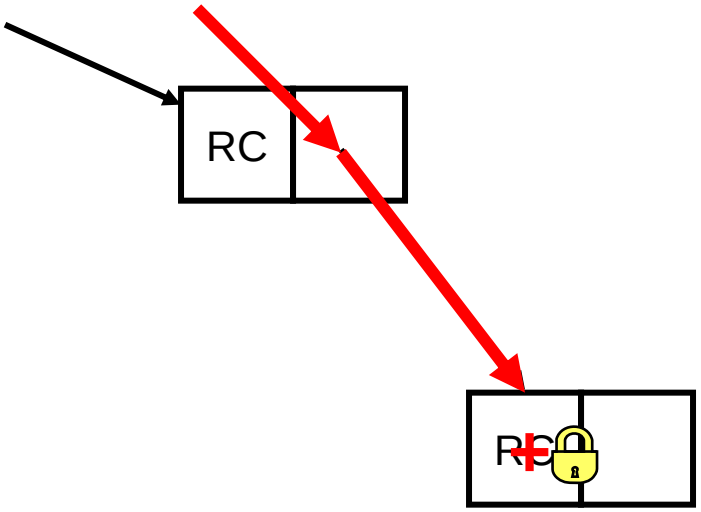


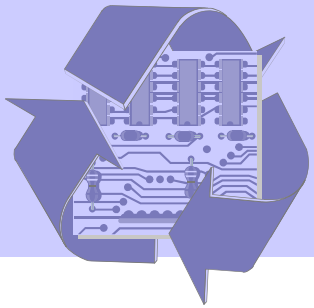
Atomic Write



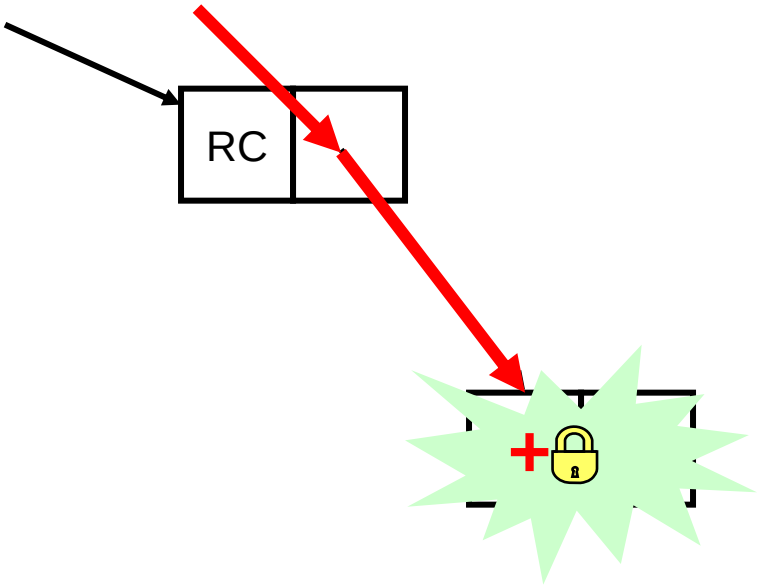


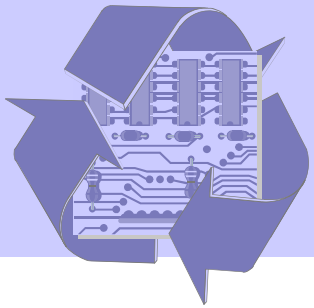
Atomic Read





Atomic Read – oops!





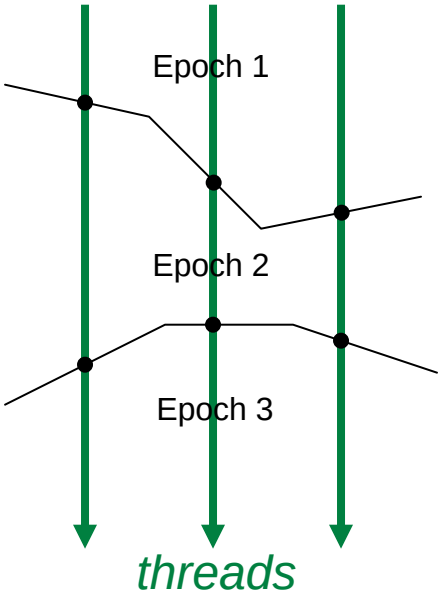
Concurrent RC

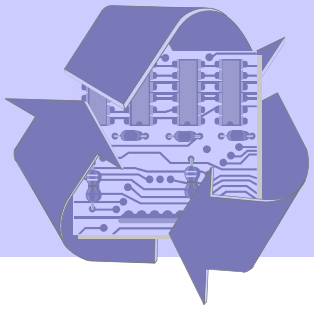
RC contention:

- Use a producer-consumer model
- Mutator threads add RC operations to local buffers.
- Collector thread consumes them to modify RC.

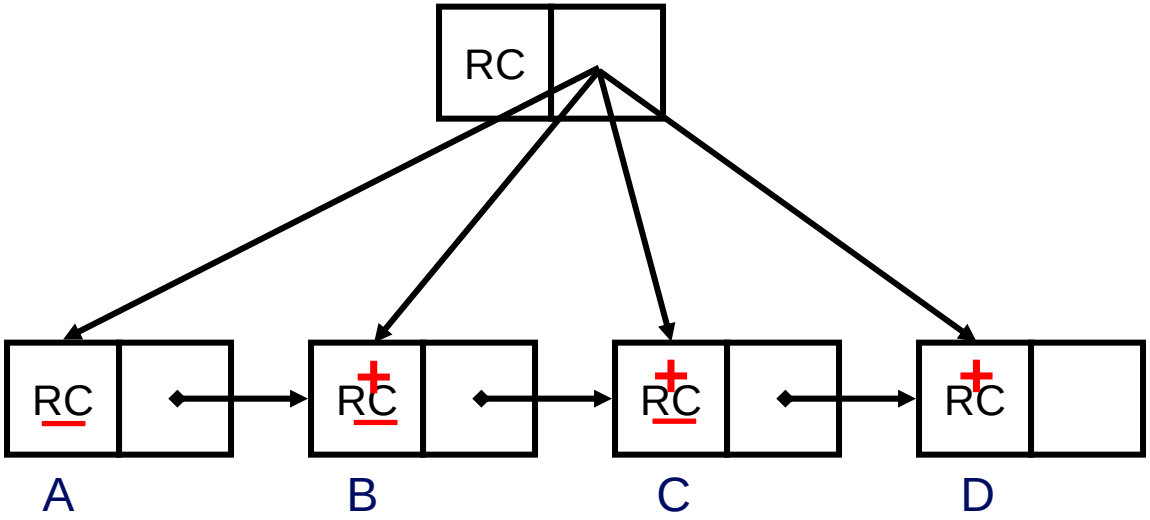
Prevent races between increments and decrements:

- Buffers periodically turned over to collector (epochs).
- Perform increments this epoch, decrements the next, or
- Use Sliding Views…





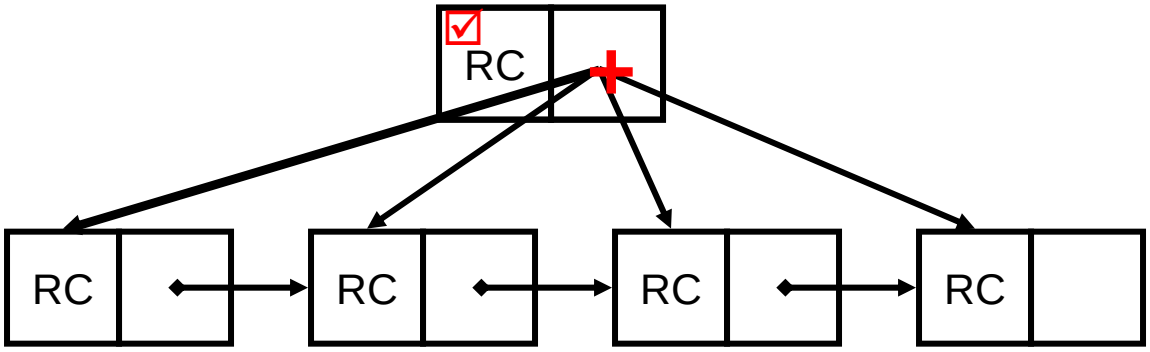
Coalesced RC



Only need

- A.rc--
- D.rc++

Coalesced RC



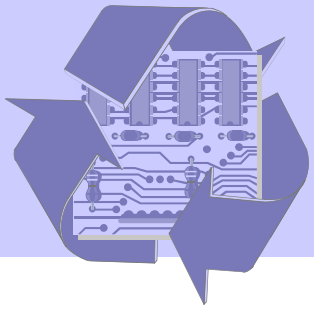
L
O
G

Reconcile

An on-the-fly reference counting garbage collector for Java

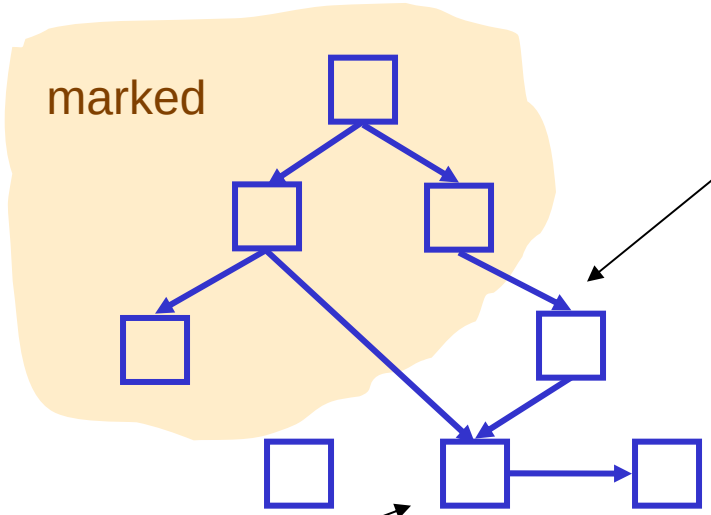
Levanoni and Petrank, OOPSLA'01.

University of Kent



C: Concurrent Tracing

Tracing concurrently with user threads introduces a coherency problem: the mutator might hide pointers from the collector.



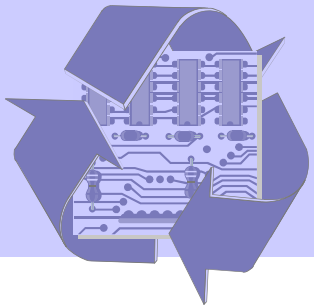
Deletion
Snapshot at the beginning write barrier

Record & revisit

Insertion
Incremental update write barrier

Record & revisit

IU catch changes to connectivity.
SAB prevent loss of the original path.



Write barrier properties

Tricolour invariants:

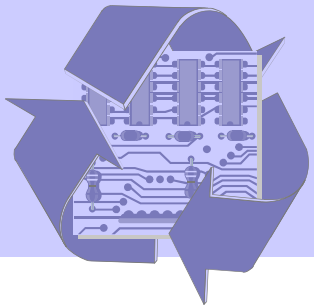
- **Weak**: live white objects are reachable from some grey object, either directly or through a chain of white objects.
- **Strong**: no black-white pointers.

Mutator colour:

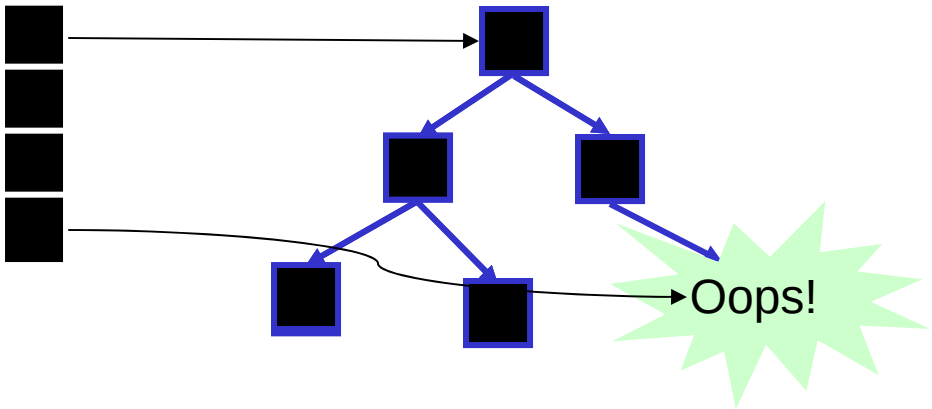
- **Black**: scans thread stacks just once
- **Grey**: revisit thread stacks

Colour of new objects

- Best allocated black?



Safe termination



Insertion/incremental update barrier

Needs a final, stop the world, tracing phase.

Deletion/snapshot at the beginning barrier

No stop the world phase required.

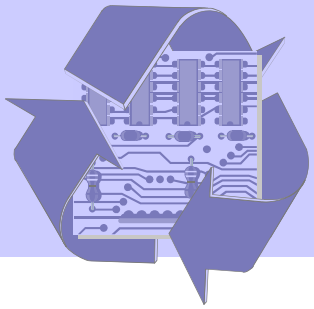
But more floating garbage retained.

Black mutator

- Read barrier to preserve the strong invariant (no black-white).
- Deletion write barrier to preserve the weak invariant (reachable from gray).

Grey mutator

- Deletion write barrier to preserve the strong invariant.

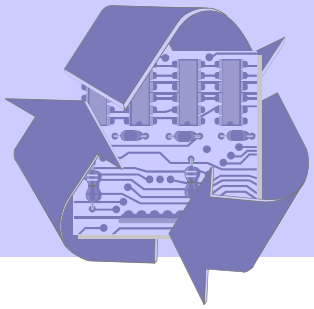


Read barrier methods

Alternatively: don't let the mutator see objects that the collector hasn't seen.

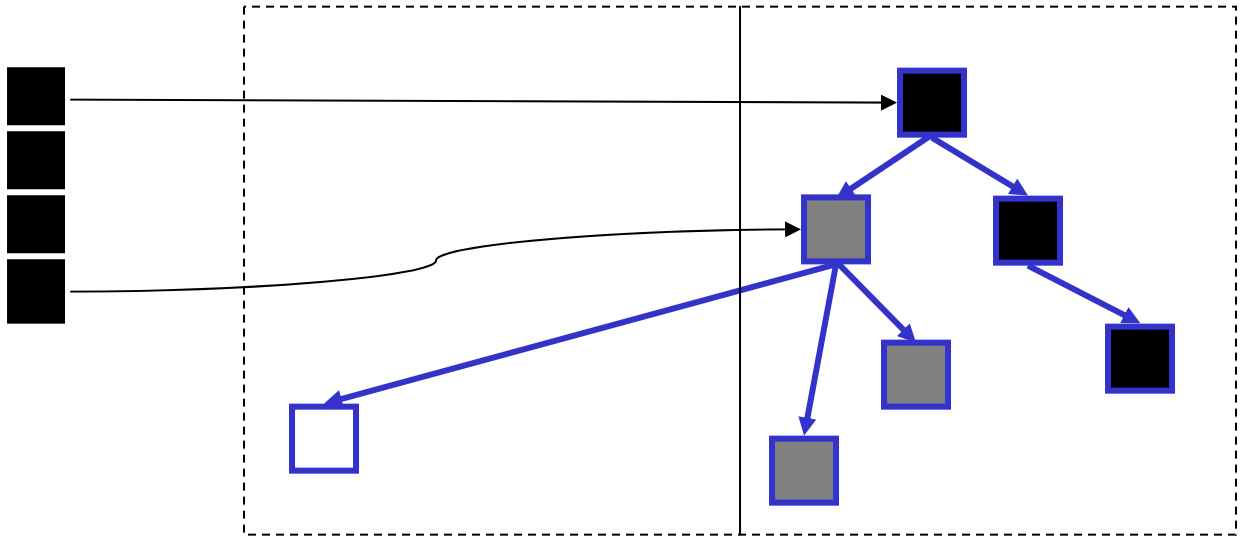
How?

- Trap mutator accesses and mark or copy & redirect.
- Read barriers
  - software
  - memory protection.

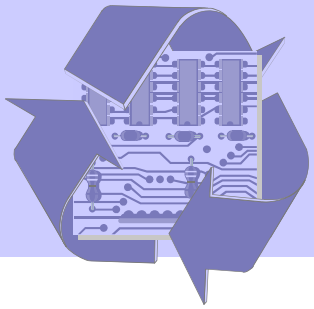


Concurrent copying

Read barrier: the original Baker collector

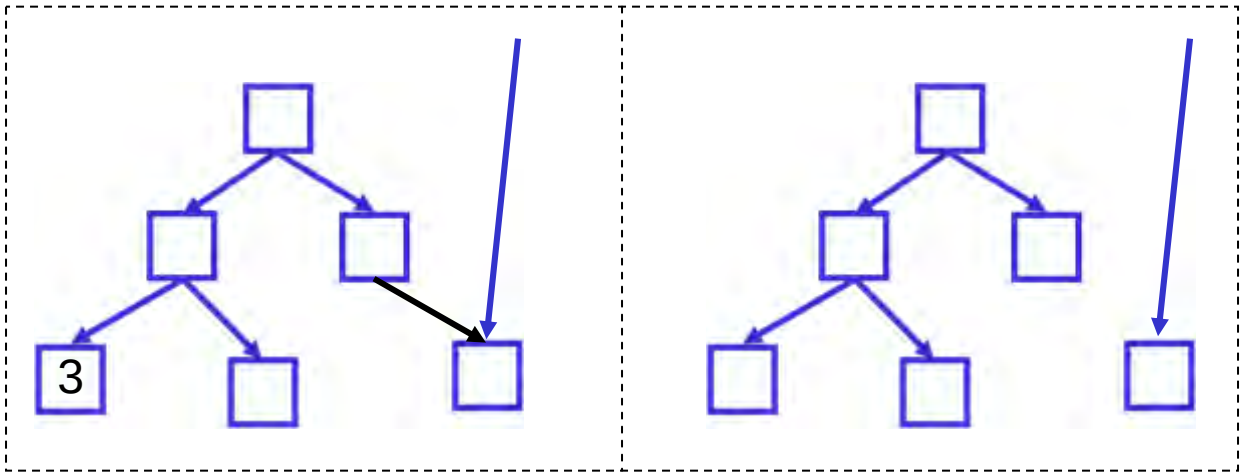


- Read barriers are expensive
- Trap storm at the start of a collection



Replicating collection

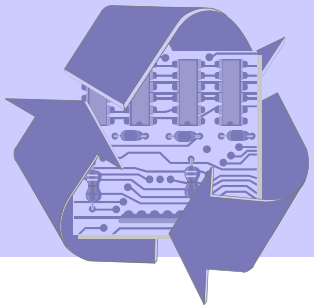
Use a write barrier rather than a read barrier



Options

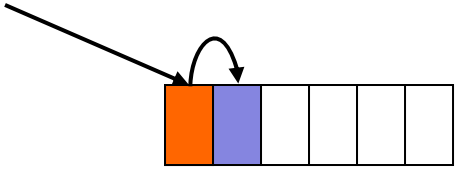
- Log the mutated objects and (later) reapply change to replica
- Use only the to-space replica
- Double-write to both original and replica

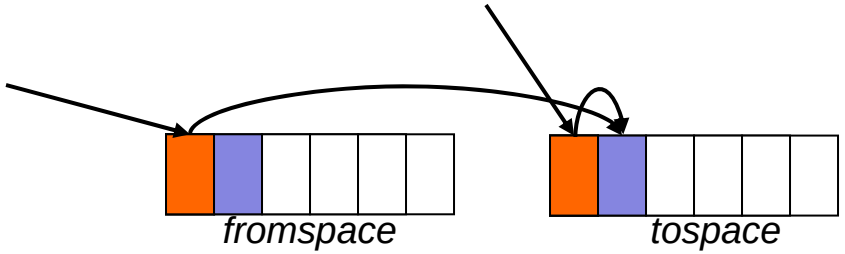
University of Kent



Brooks barrier

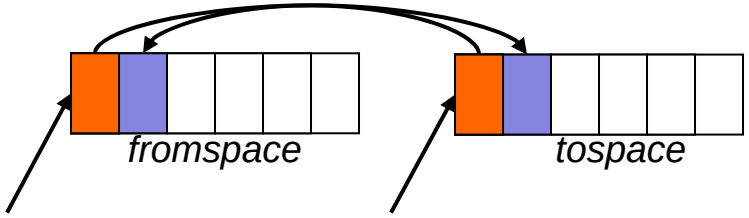
- Read from / write to replica only
- f.x = g.y ⇒ f.fwd.x = g.fwd.y

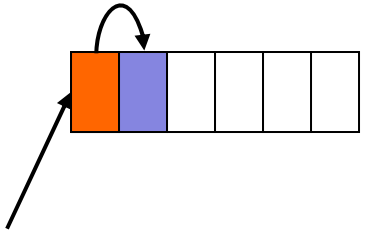




OVM barrier

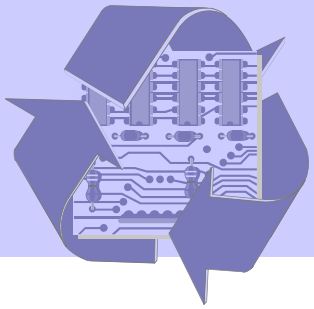
- Double write (but often to the same object); read from either
- f.x = g.y ⇒ f.x = g.y; f.wd.x = g.y





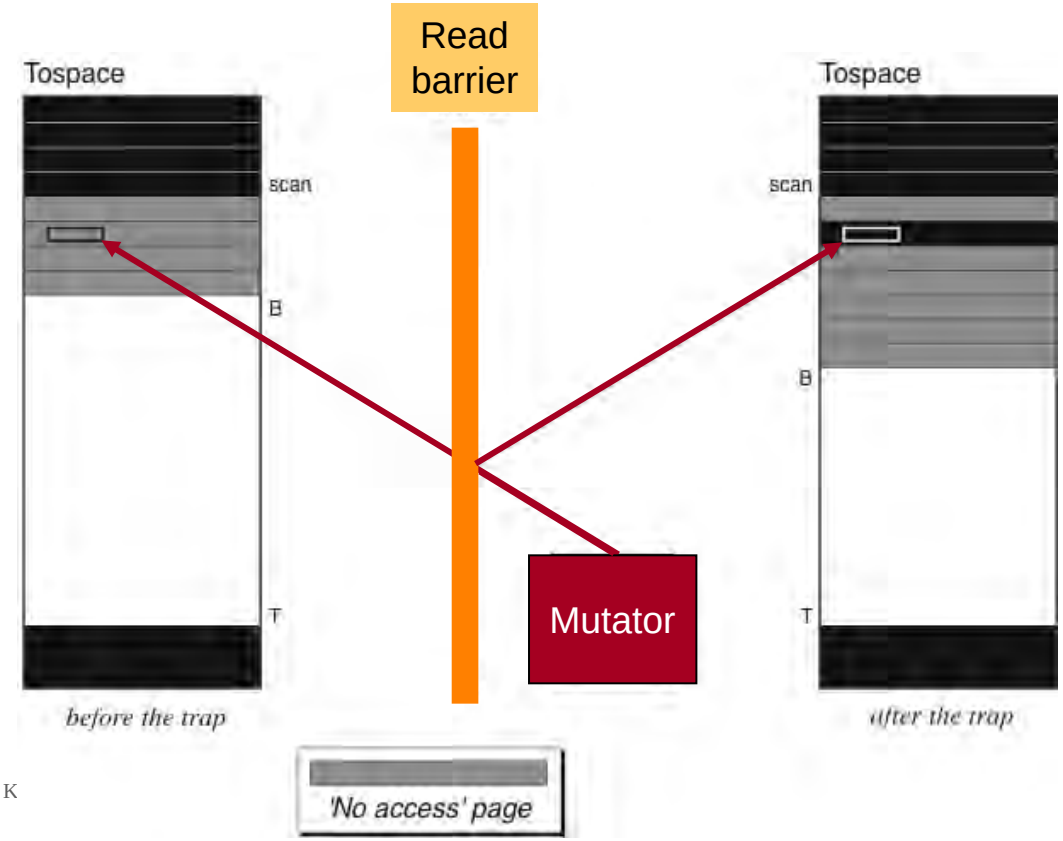
Scheduling Hard Real-time Garbage Collection
Kalibera et al, RTSS'09

University of Kent

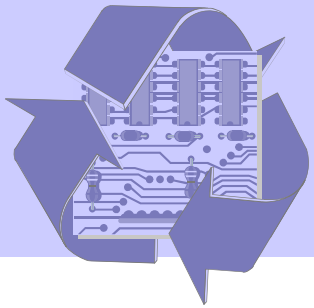


Memory Protection

By protecting a grey page, any access attempt by the mutator is trapped.



University of Kent



Compressor revisited

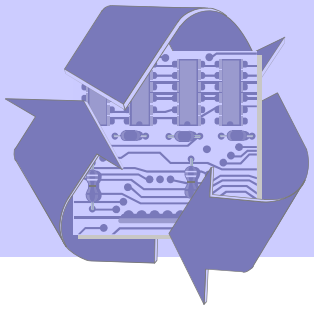
On-the-fly

- Fully concurrent = no Stop-the-world phase

Phases

1. Mark live objects, constructing a mark-bit vector.
2. From mark-bit vector, construct offset table mapping fromspace blocks to tospace blocks.
3. mprotect tospace pages (virtual not physical).
   - need to map each physical page to 2 virtual pages.
4. Stop threads one at a time and update references to tospace.
5. On access violations, move N pages and update references in moved pages. Need only 1 empty physical page / processor
6. Unprotect these pages and resume mutator

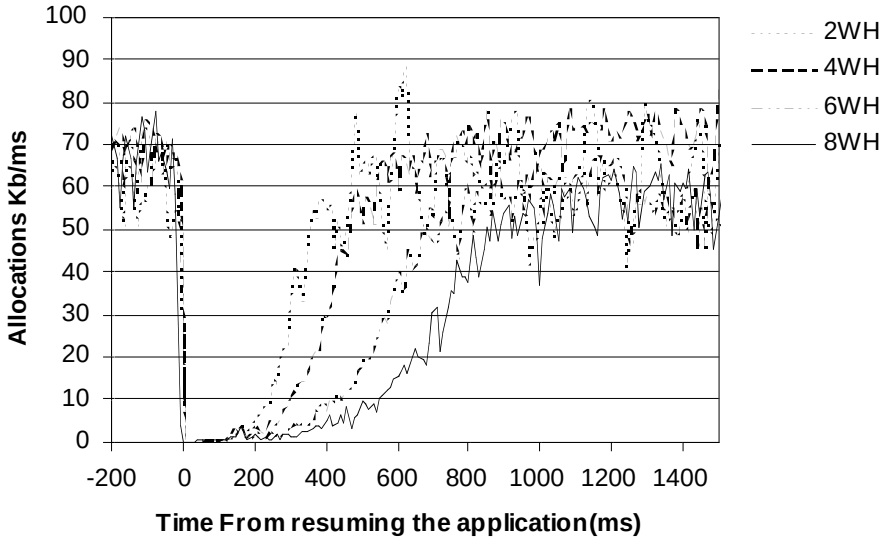
The Compressor: concurrent, incremental and parallel compaction, Kermany & Petrank, PLDI'06
cf. *Mostly Concurrent Compaction for Mark-Sweep GC,* Ossia et al, ISMM'04



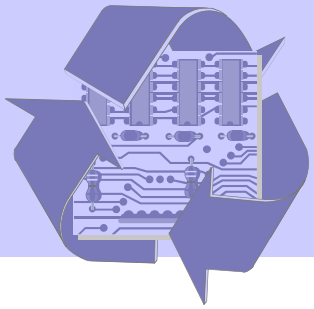
Compressor results

Throughput: better than generational mark-sweep for SPECjbb2000 server benchmark, worse for DaCapo client benchmarks

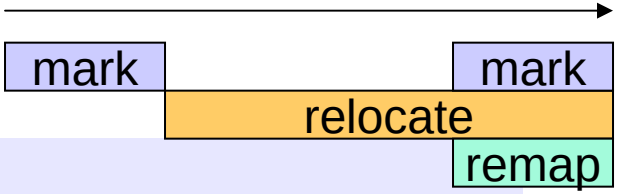
Pauses: delay before achieving full allocation rate.



SPECjbb2000: 2-way 2.4GHz Pentium III Xeon



Azul Pauseless GC



Mark (parallel and concurrent)

- Identifies pages to evacuate (sparse).
- Tagged pointers (*Not-Marked-Through* bit).

Relocate (parallel and concurrent)

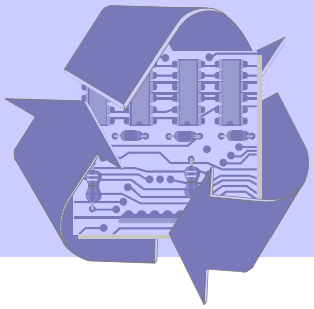
- Protect *from-space* pages and concurrently relocate.
- Hardware TLB has GC-mode privilege level.
- Fast traps: no null check, no memory access, etc., etc.

Remap (parallel and concurrent)

- GC threads traverse object graph, tripping RB to update stale refs.

Results

- Uses proprietary hardware (Vega) and/or modified kernel.
- Substantially reduced transaction times; almost all pauses < 3ms.
- Reports other (incomparable) systems had 20+% pauses > 3ms.



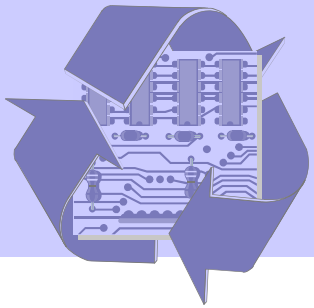
PART 4: Real-time GC

Region schemes

- Cumbersome model, hard to program?
- Requires rewriting libraries.
- Better to infer regions (à la ML-kit)?

Real-time GC issues

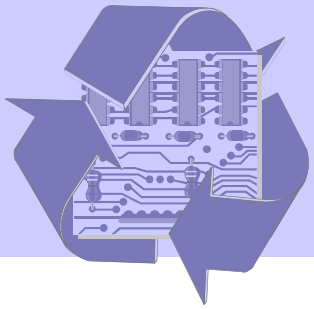
- Incrementality
- Schedulability
  - Guaranteed worst-case execution (pause) times.
  - Pause time distribution.
  - Time-based (slack or periodic) or work-based?
- Managing large data structures.
- Dealing with fragmentation.



Java for Real-time

US Navy DDG-1000 Zumwalt class destroyer by Raytheon, programmed in Real-time Java, running on IBM's WebSphere virtual machine

Boeing ScanEagle UAV, navigation system running on Purdue University's OVM, an open-source real-time Java virtual machine



Scheduling of RTGC

Work-based

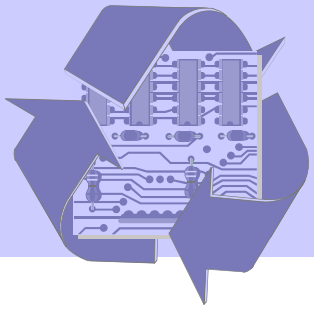
- GC work quanta based on mutator allocation rate
- Used by Baker

Slack

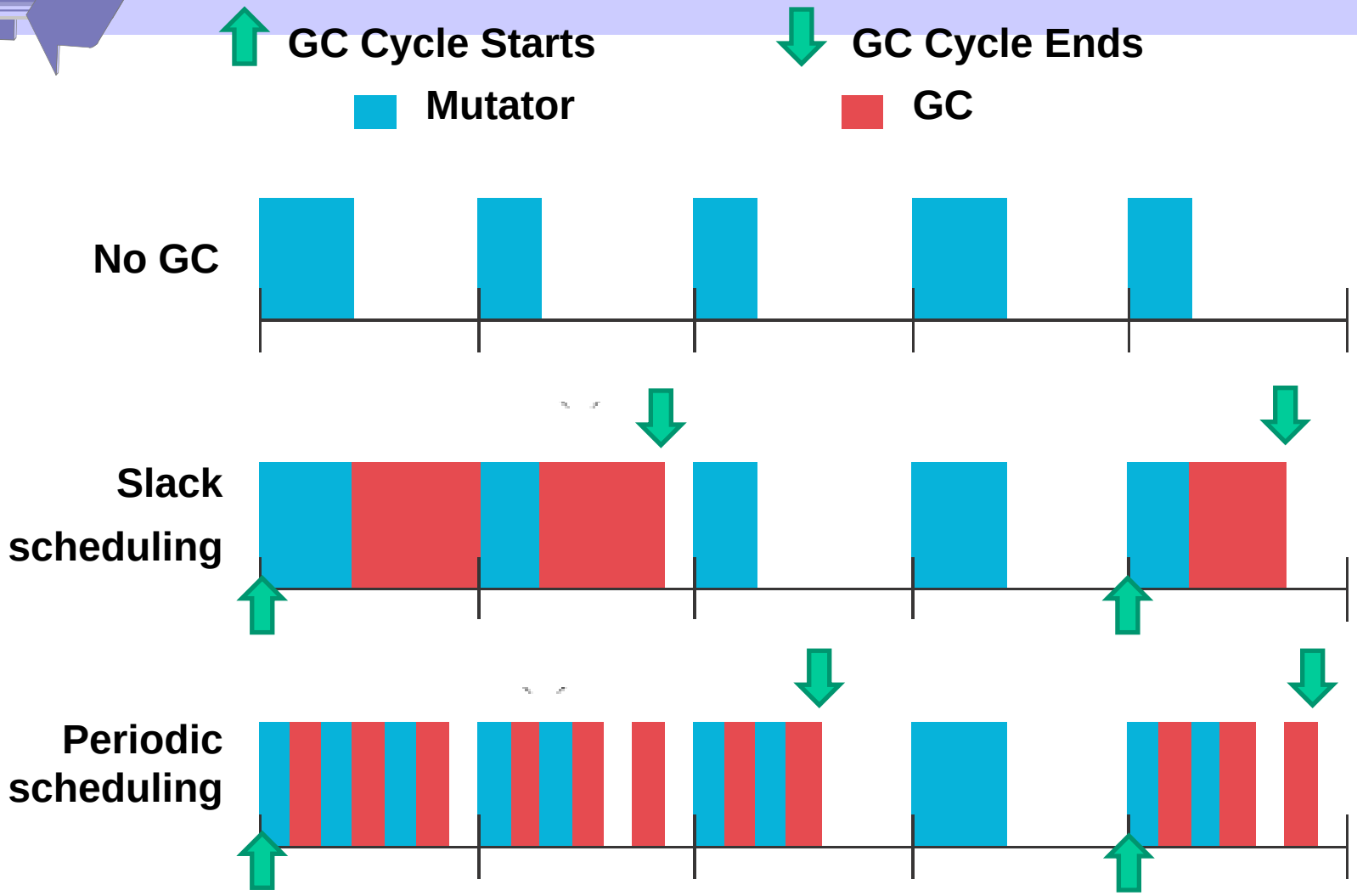
- GC thread scheduled only in slack time
- Used by SUN Real-time System for hard real-time threads

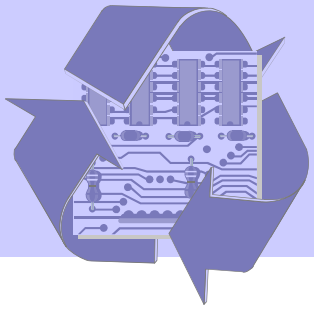
Periodic

- GC thread scheduled only in statically allocated time slots, preempting mutator tasks
- Time slots are allocated via a repeating pattern
- Used by IBM Web-Sphere Real-time



(Time-Based) Scheduling of RTGC

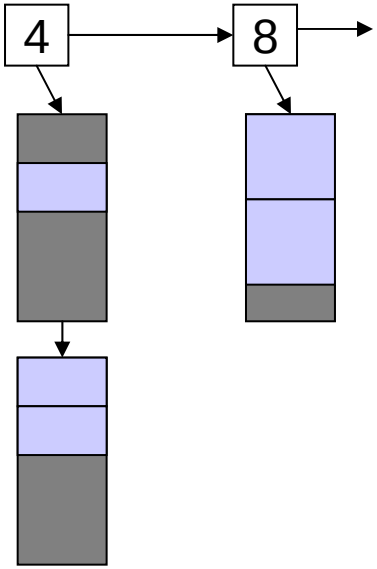




Metronome

Allocation

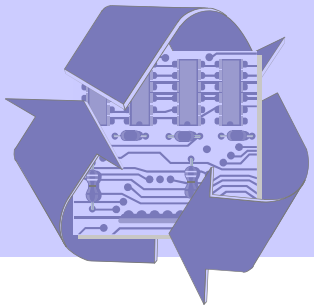
- Segregated free-lists, with large number of size classes
  - low internal fragmentation
  - external fragmentation removed by copying



Mostly non-copying collection

- Incremental mark
- Deletion barrier (snapshot at the beginning)
  - avoids rescanning.
- Lazy sweep.

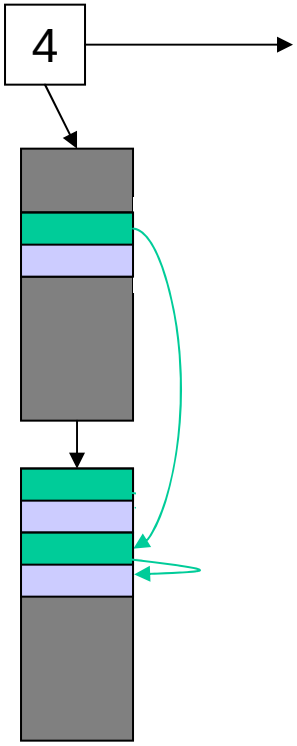
A Real-time GC with Low Overhead and Consistent Utilisation
Bacon et al, POPL'03



Metronome collection

Defragmentation

- If sufficient free pages
  - Allow mutator to continue for another GC cycle without running out of memory.
- Otherwise, copy objects:
  - From a fragmented page to another page of same class.
  - Break large arrays broken into a sequence of power-of-2 sized arraylets.
    - compiler optimisations for fast access.
  - Brooks indirection pointer in object headers,



Metronome performance

Scheduling

- *Time-based*.
- Work-based gave inconsistent utilisation, leading to infeasible scheduling.

Parameters

- Max live memory: 20-34MB live.
- Max allocation rate: 80-358MB/s max.

Results

- Min. mutator utilisation: 44.1%-44.6%.
- Pause time: 12.3-12.4ms max (10.7-11.4ms avg).
- Copied < 2% of traced data.

- JVM98 on 500MHz PowerPC.

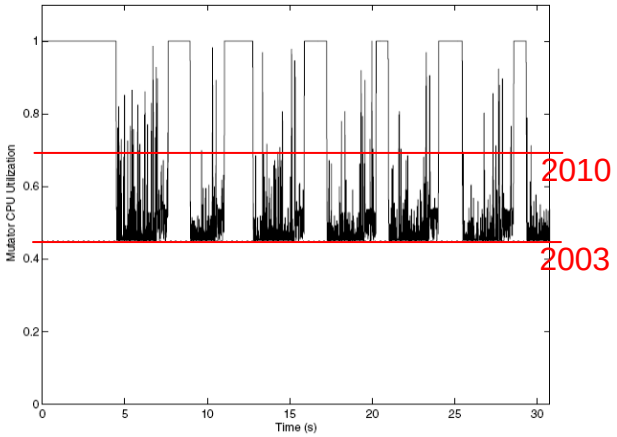


Figure 10: Time-based utilization for javac, $\Delta t = 22.2$ ms.

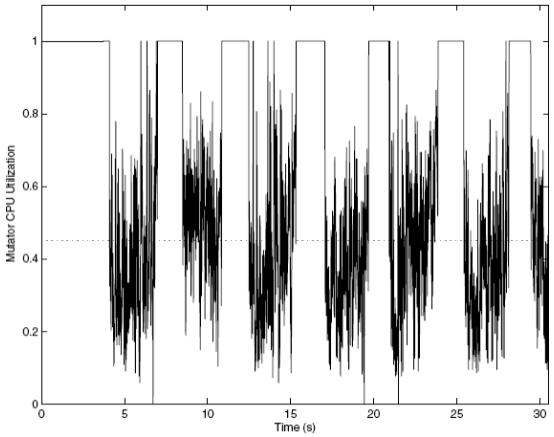
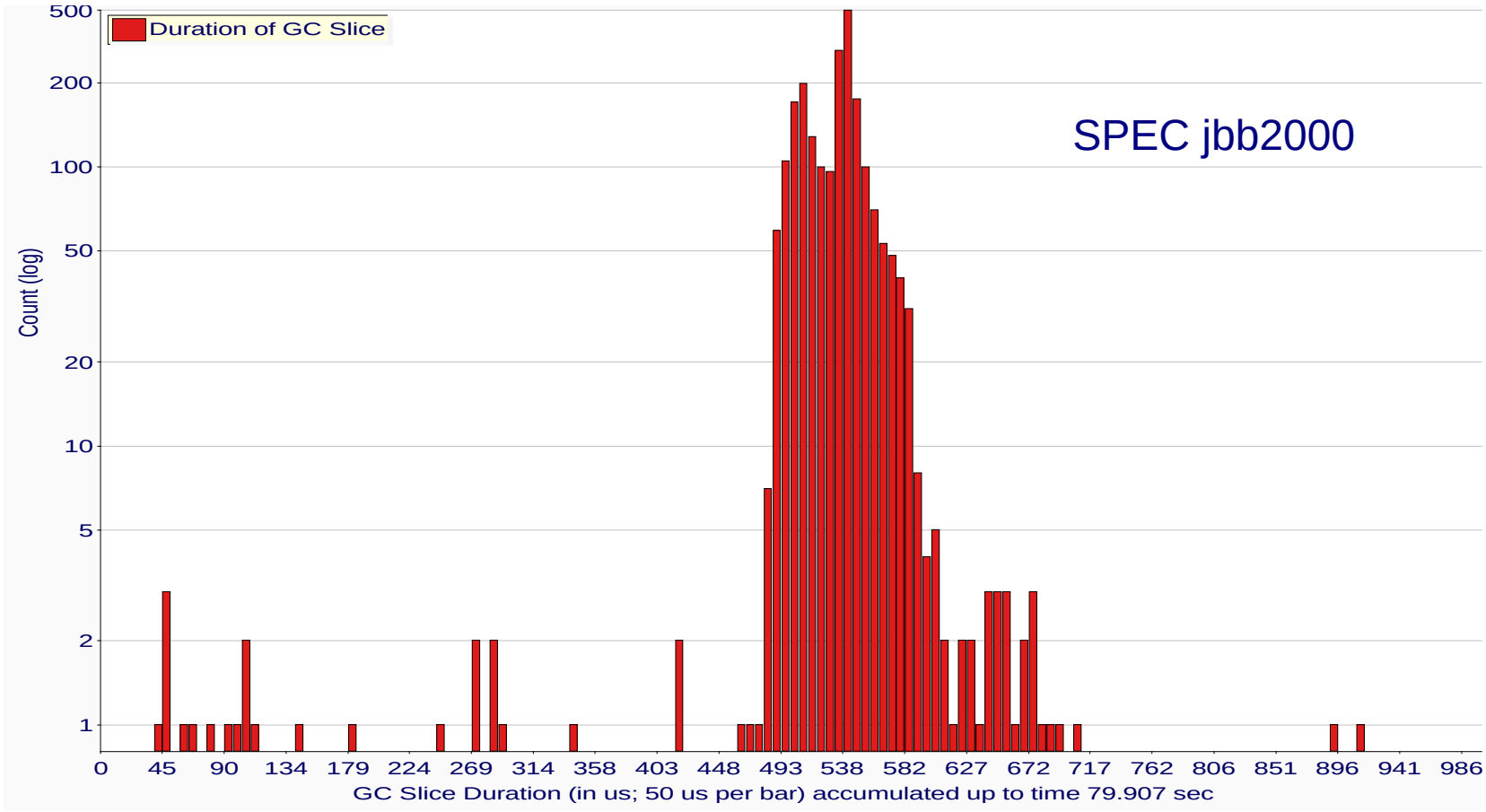


Figure 13: Work-based utilization for javac, $\Delta t = 22.2$ ms.

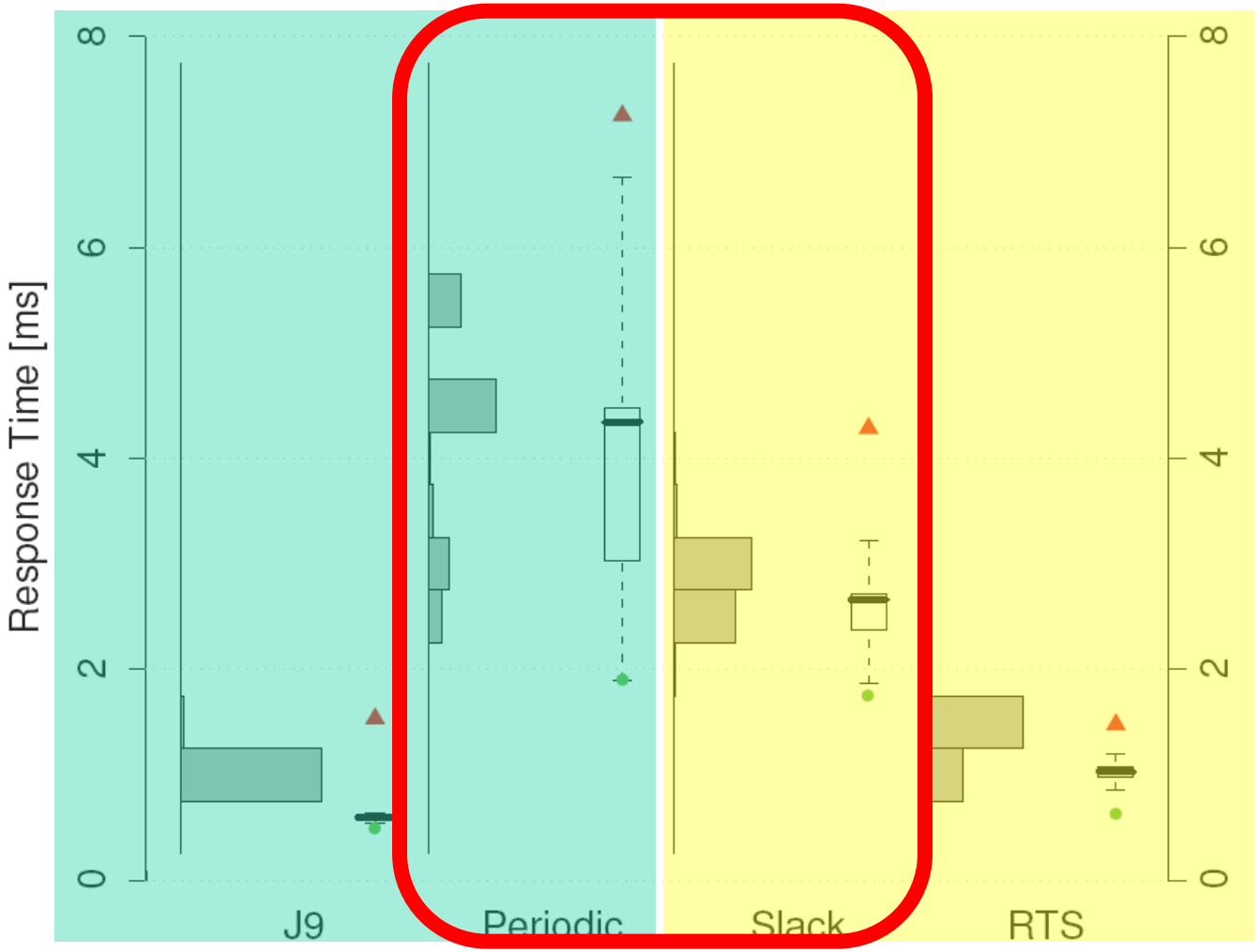
Metronome performance



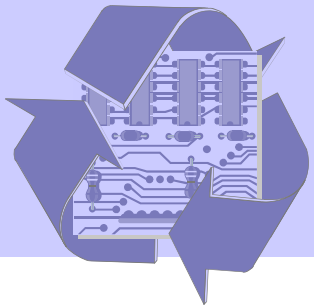
Design and Implementation of a Comprehensive Real-time Java Virtual Machine
Bacon et al, EMSOFT'07

University of Kent

Response Time



Scheduling Hard Real-time Garbage Collection
Kalibera et al, RTSS'09



Shameless plug!

The Garbage Collection Handbook: The Art of Automatic Memory Management

Richard Jones, University of Kent
Tony Hosking, Purdue University
Eliot Moss, University of Massachusetts

Chapman & Hall, 2011




Questions?