Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance

Stephen Blackburn and Kathryn S. McKinley

PLDI 2008

Youngjoon Jo

February 9th 2012

CS661

Outline

- Canonical tracing garbage collectors
  - Each sacrifice one objective
- Describe mark-region
- Immix
  - Combine mark-region and opportunistic defragmentation
  - Illustrate with figures!
- Implementation
- Results

Canonical Collectors

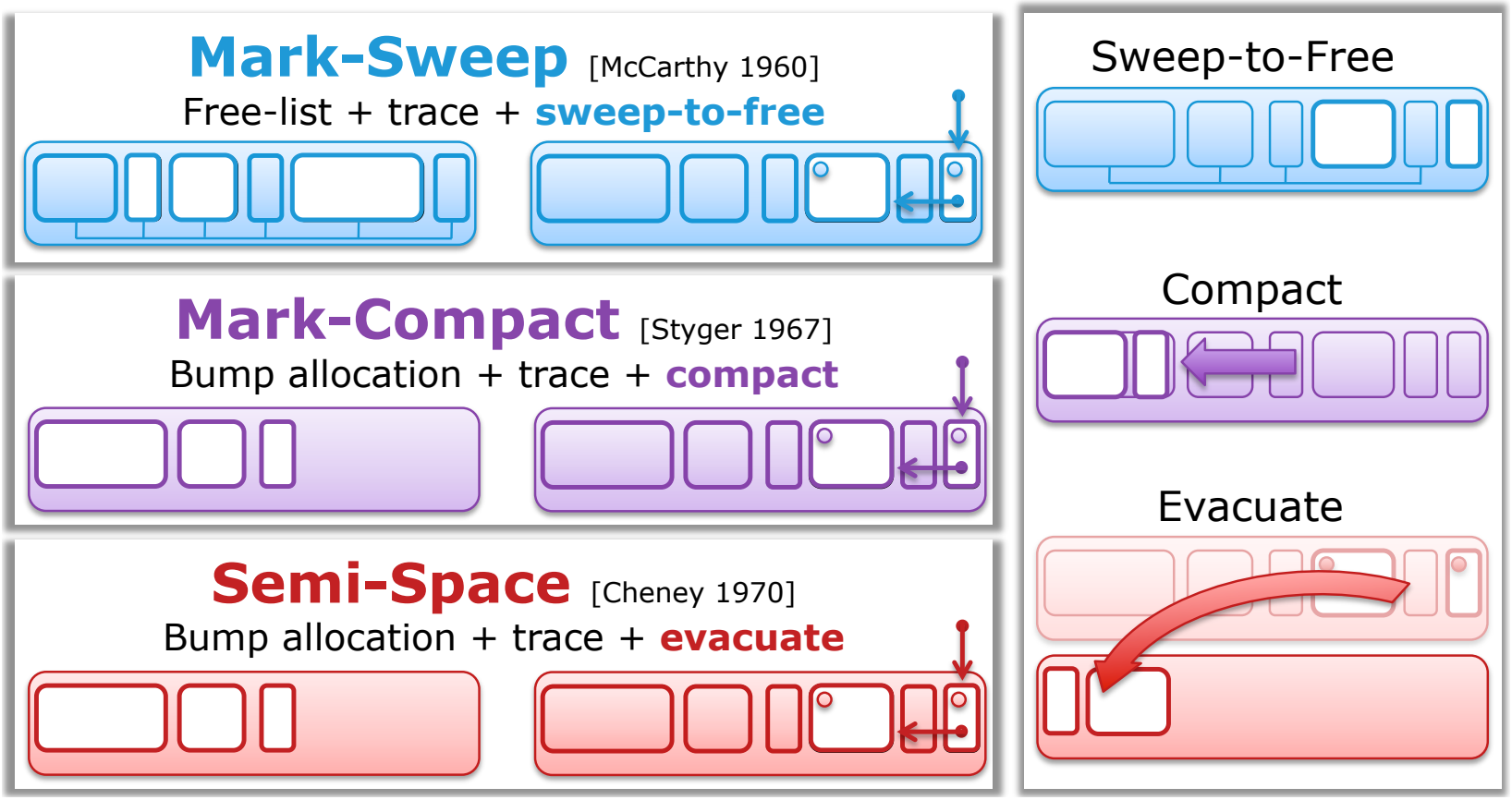


Image from Blackburn's slides

Canonical Collectors

	Space Efficiency	Fast Reclamation	Mutator Performance
Mark-Sweep	O	O	X
Mark-Compact	O	X	O
Semi-space	X	O	O

Canonical Collectors

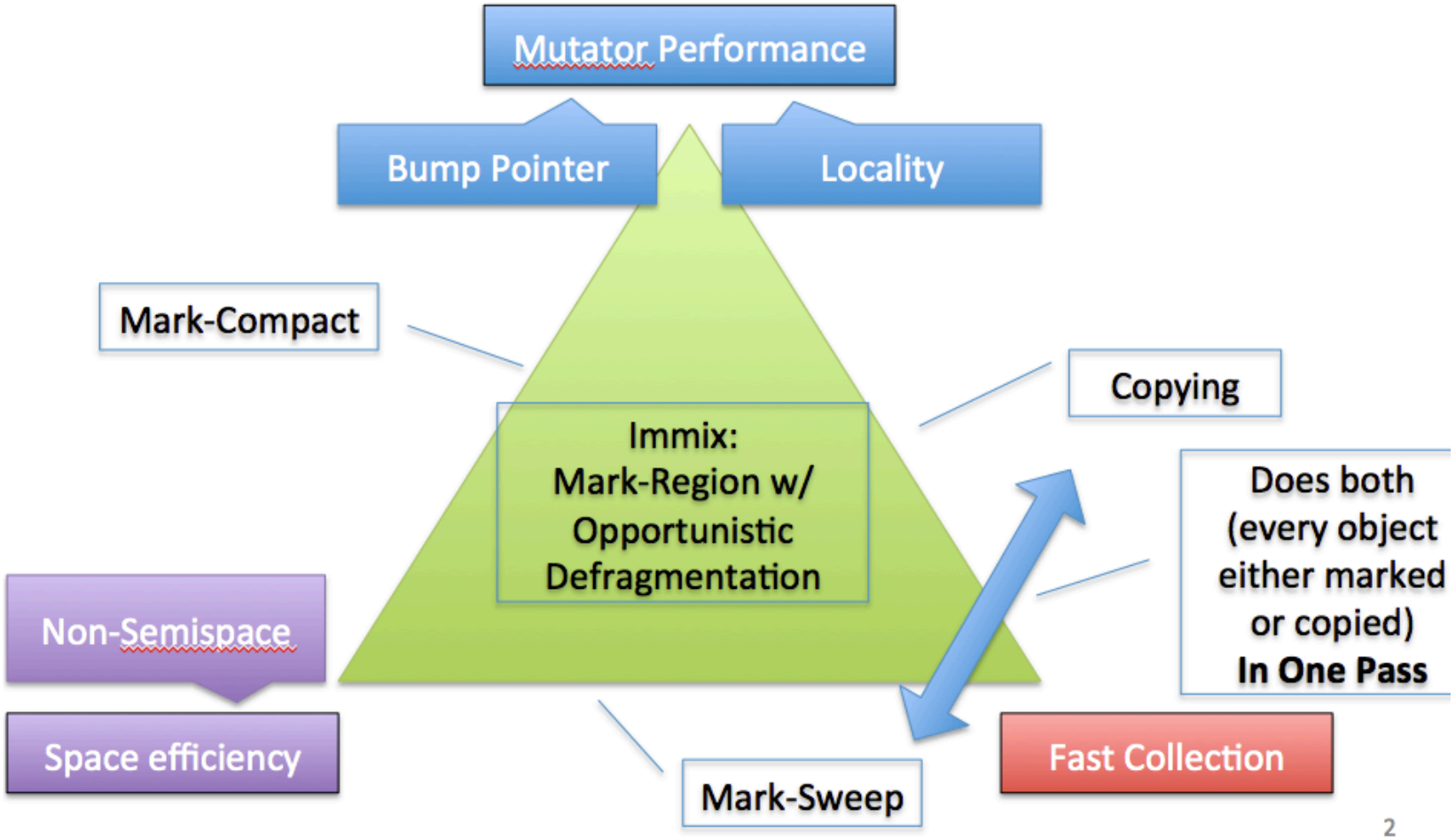


Image from Curtis Dunham's slides, CS 395T @ UTexas

Mark Region

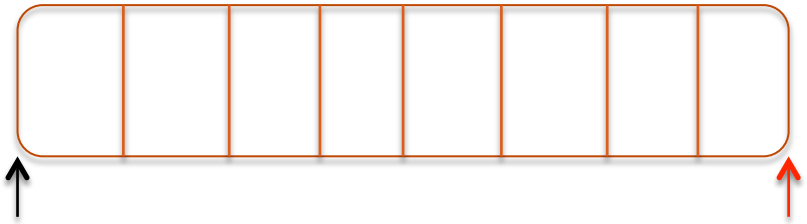
- Contiguous allocation into regions
  - ✓ Excellent locality
  - – For simplicity, objects cannot span regions
- Simple mark phase (like mark-sweep)
  - – Mark objects and their containing region
- Unmarked regions can be freed

Image from Blackburn's slides

Immix

- Two levels of region sizing
  - 32KB blocks (256 lines per block)
  - 128B lines
- Allocation policy
  - Recycle partially marked blocks first
  - Allocate into free blocks last
- Opportunistic defragmentation
  - Evacuate fragmented blocks (in order of most holes)
- Conservative line marking
  - Avoid looking up object size for small objects (< 128B)

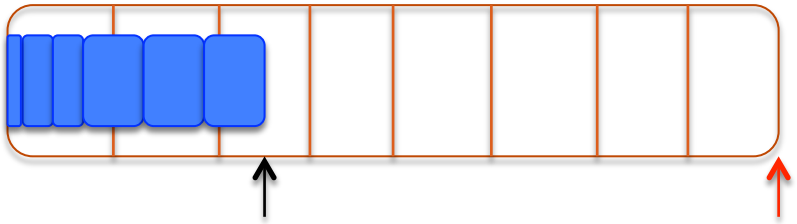
Immix Illustration



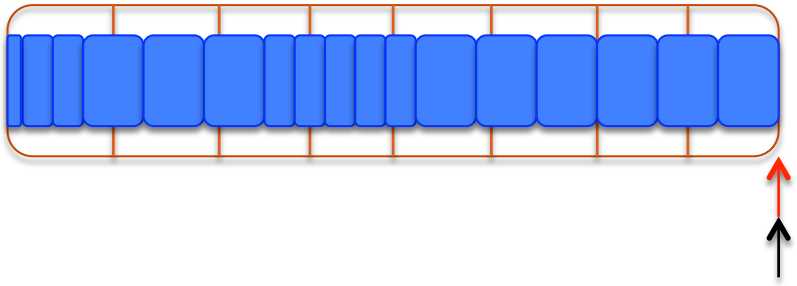
Bump Pointer

Limit Pointer

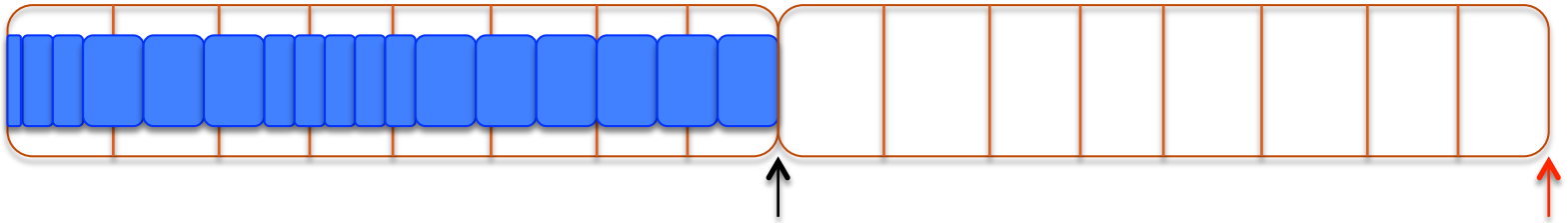
Immix Illustration



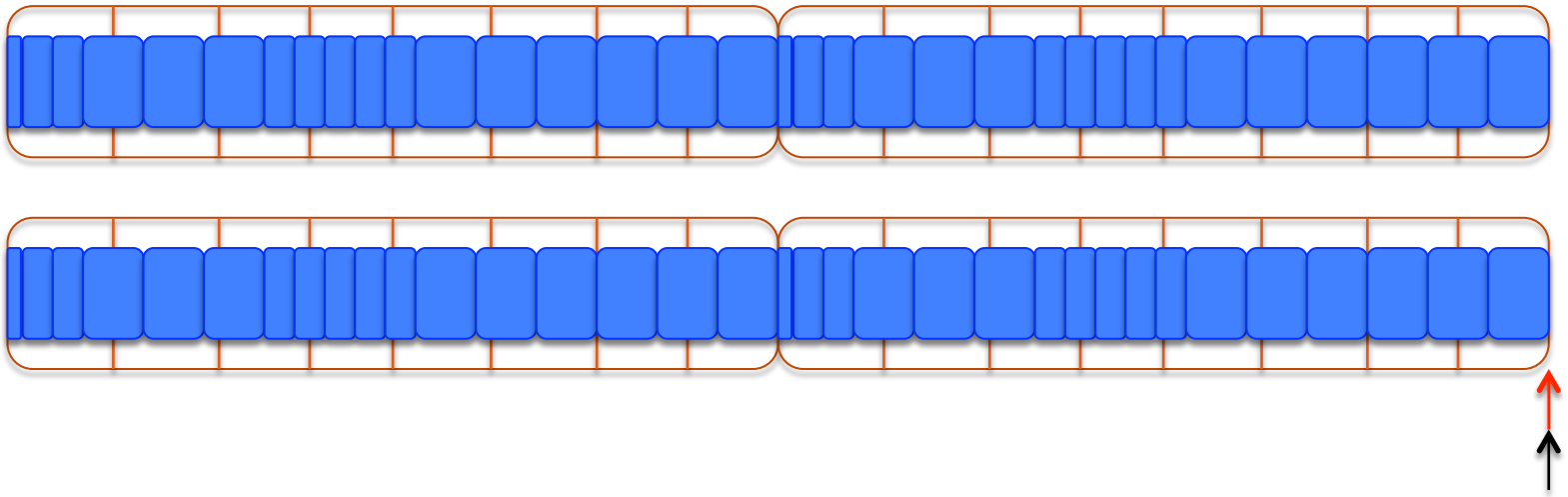
Immix Illustration



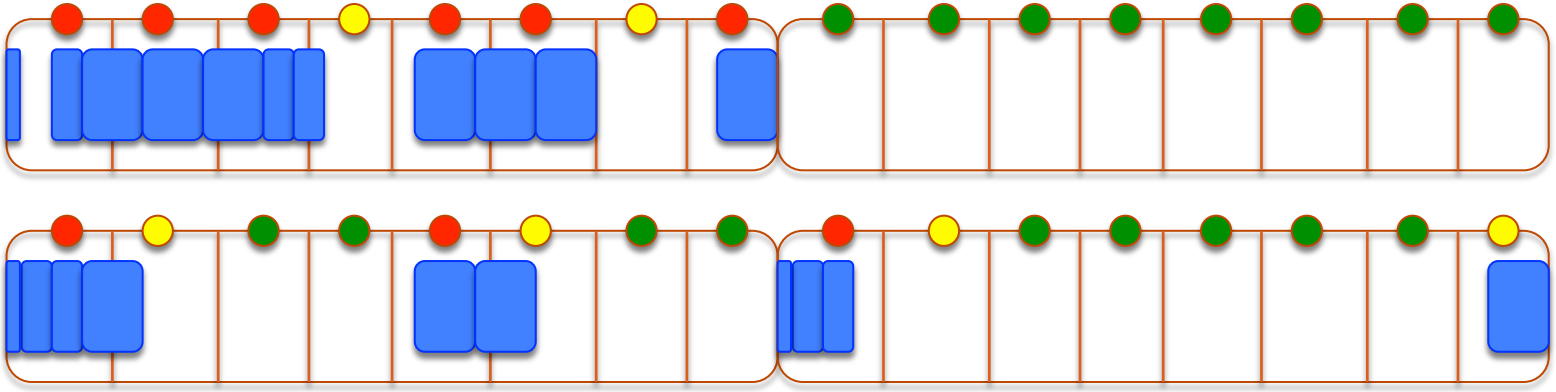
Immix Illustration



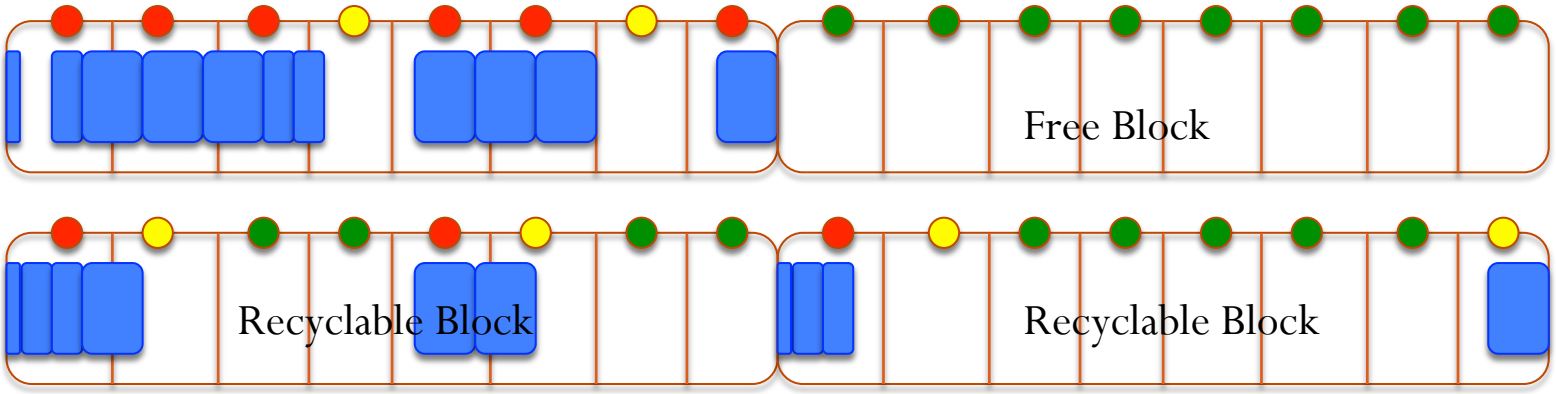
Immix Illustration



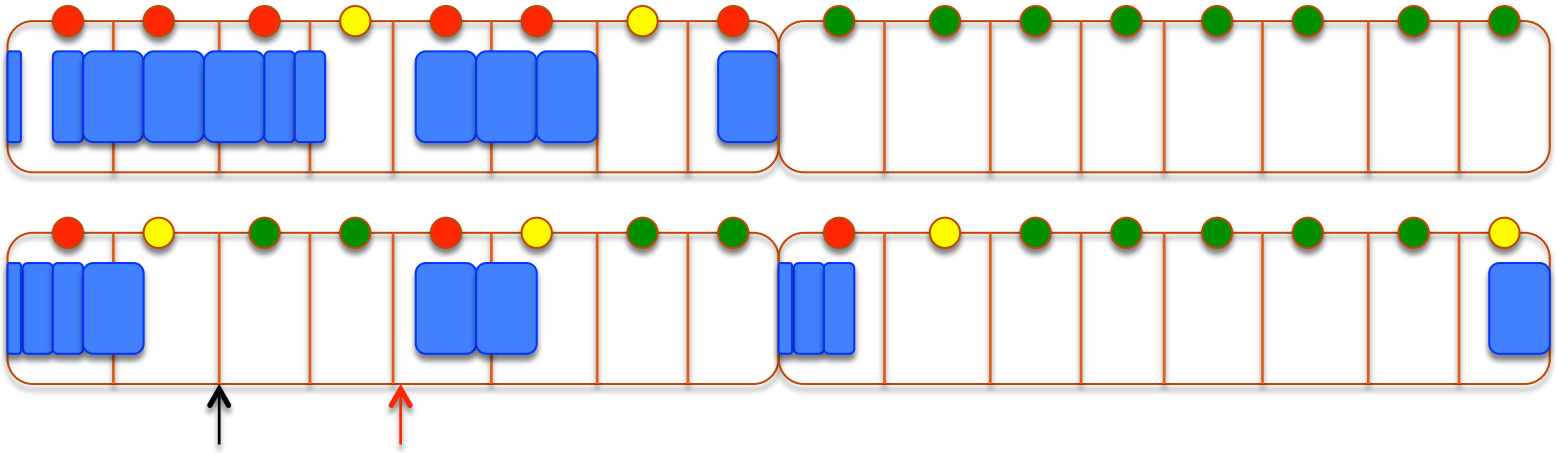
Immix Illustration



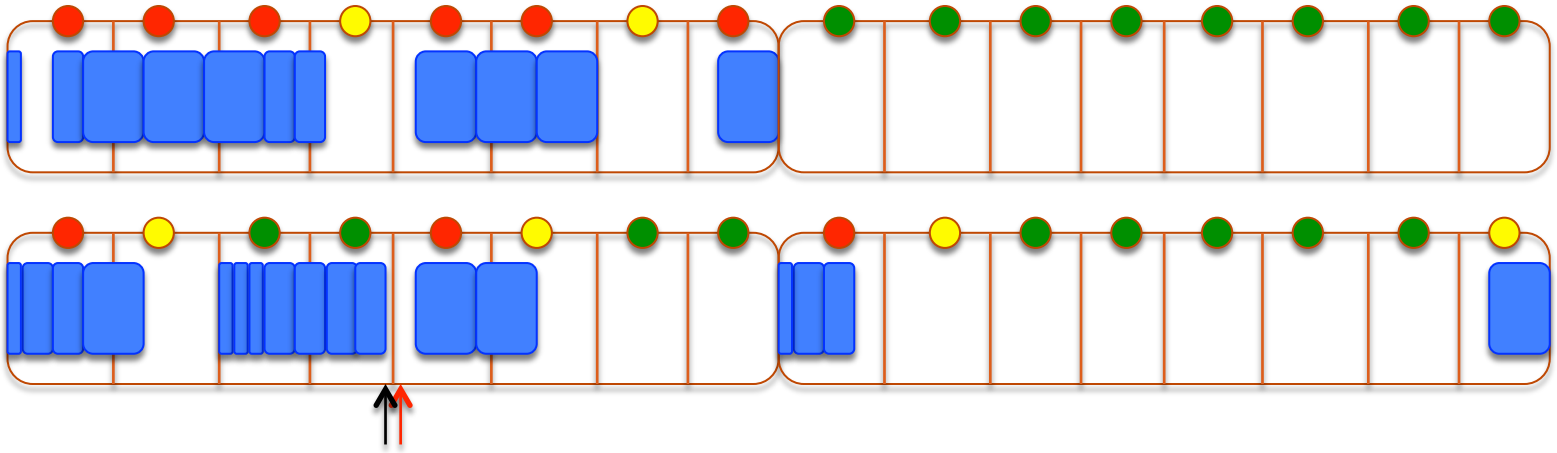
Immix Illustration



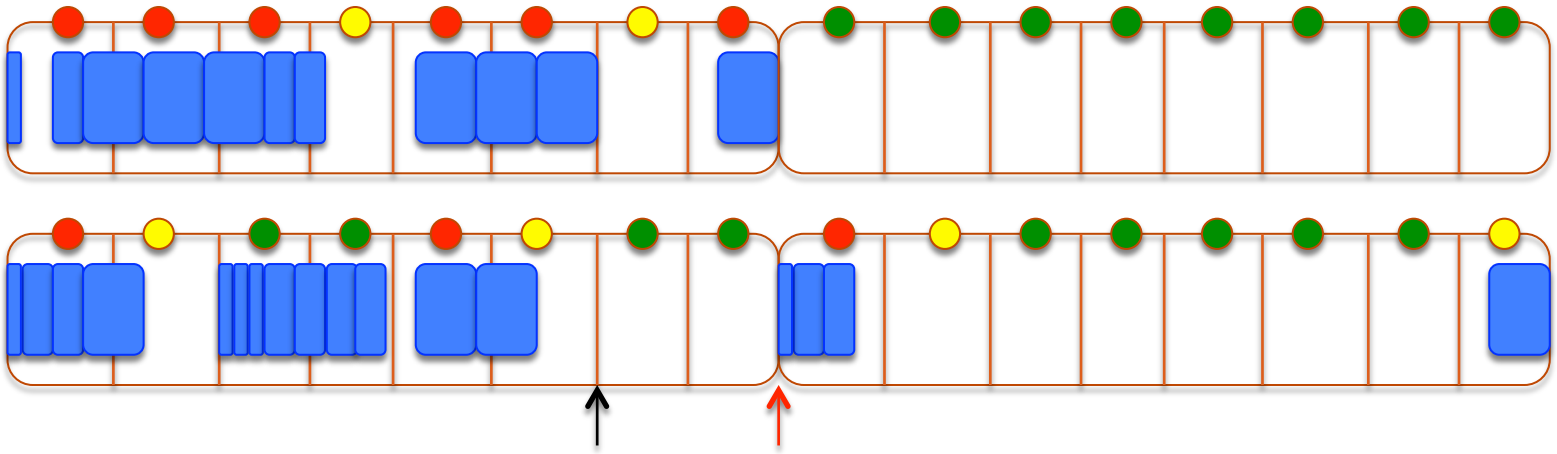
Immix Illustration



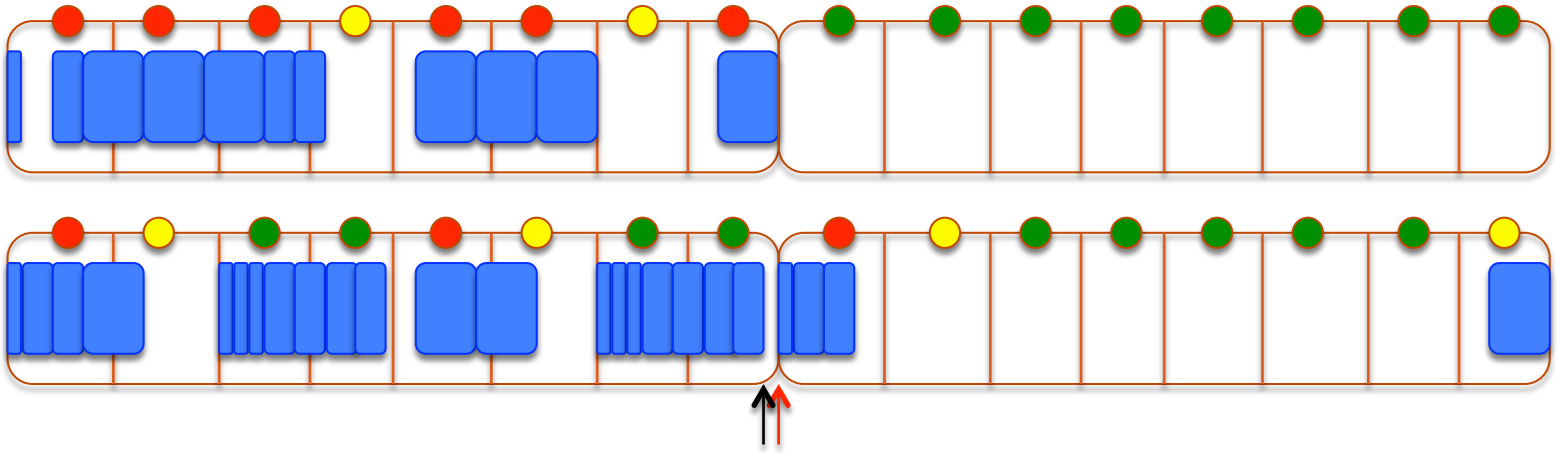
Immix Illustration



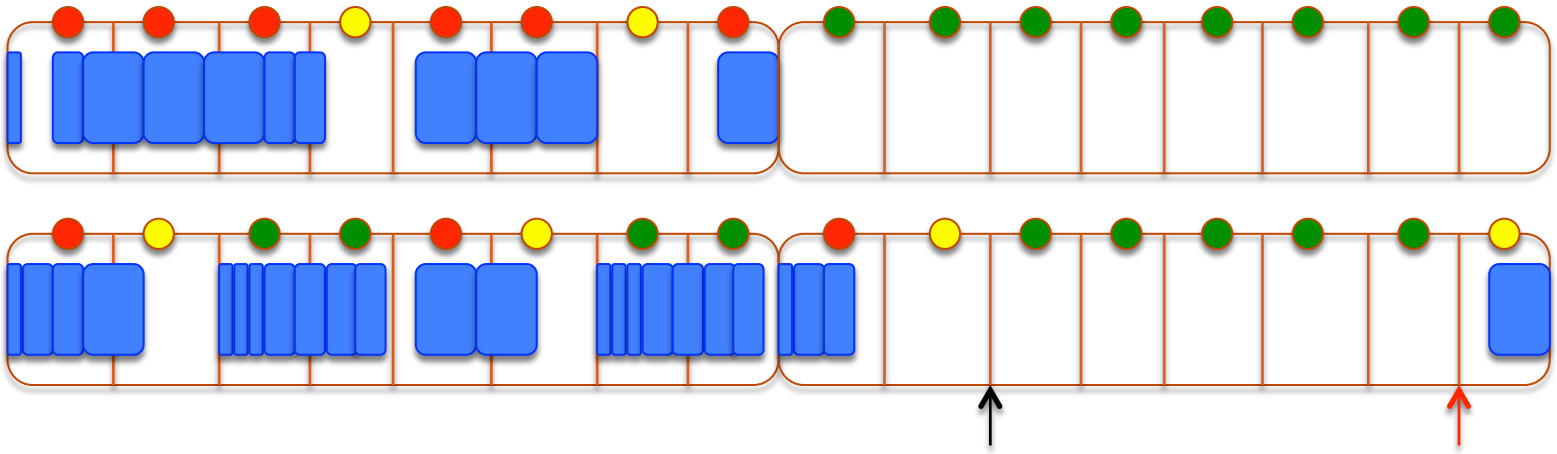
Immix Illustration



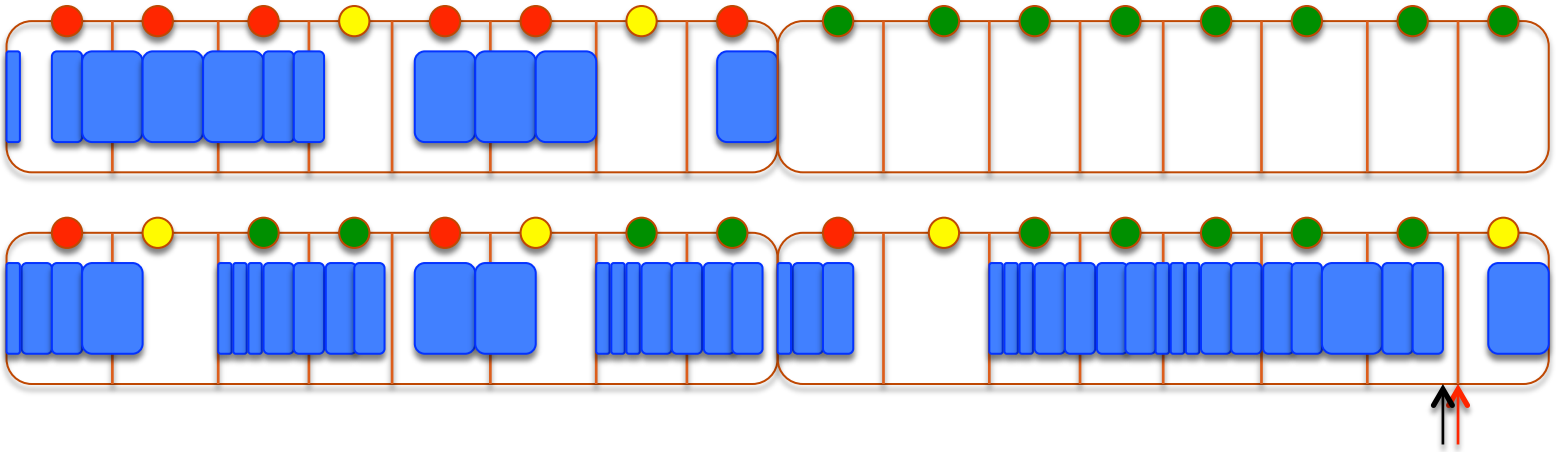
Immix Illustration



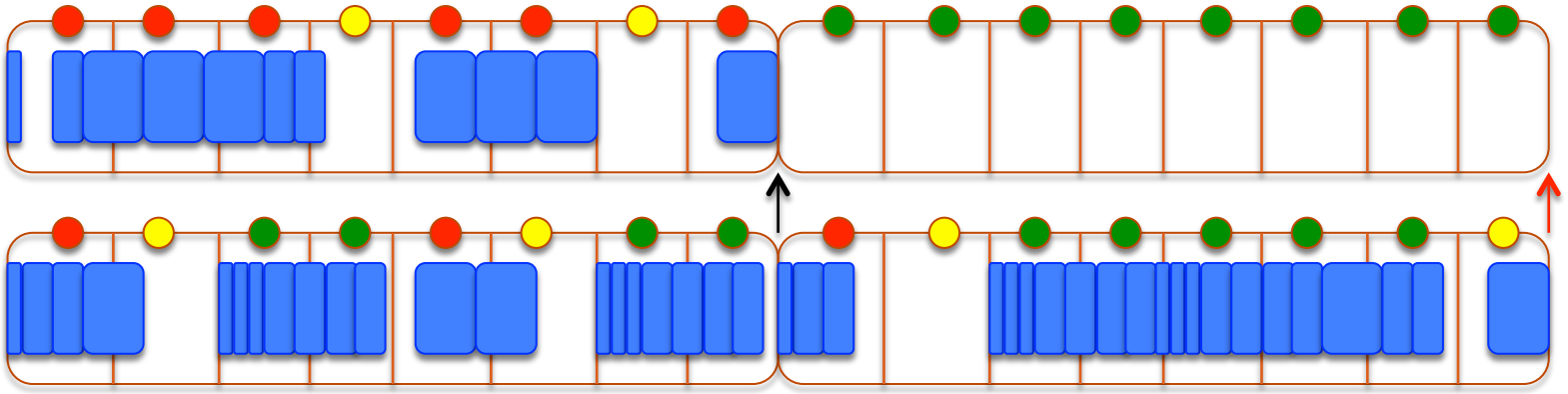
Immix Illustration



Immix Illustration



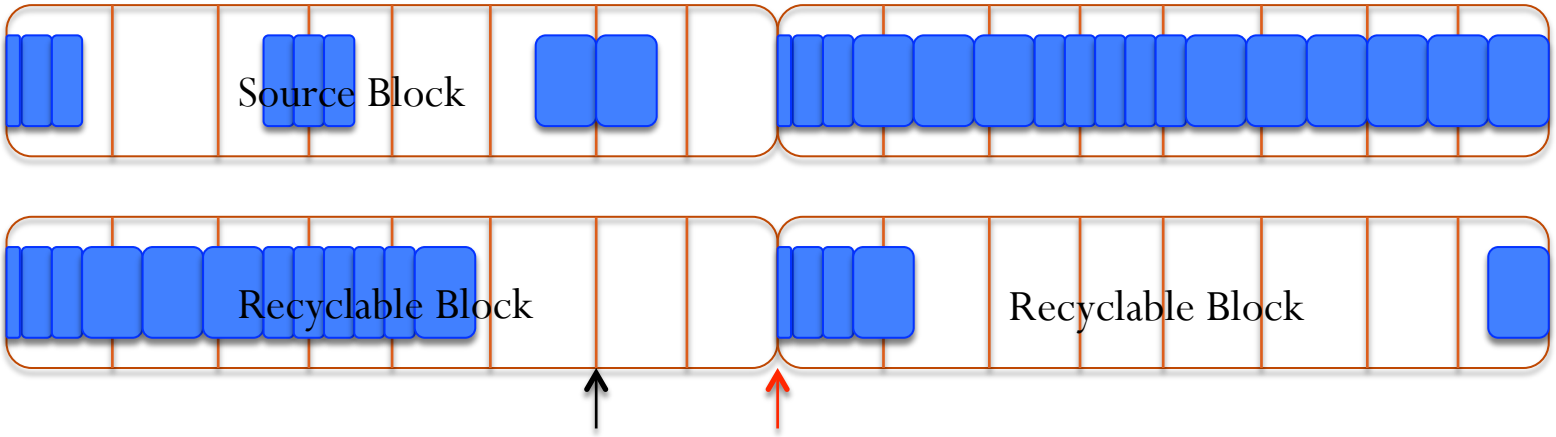
Immix Illustration



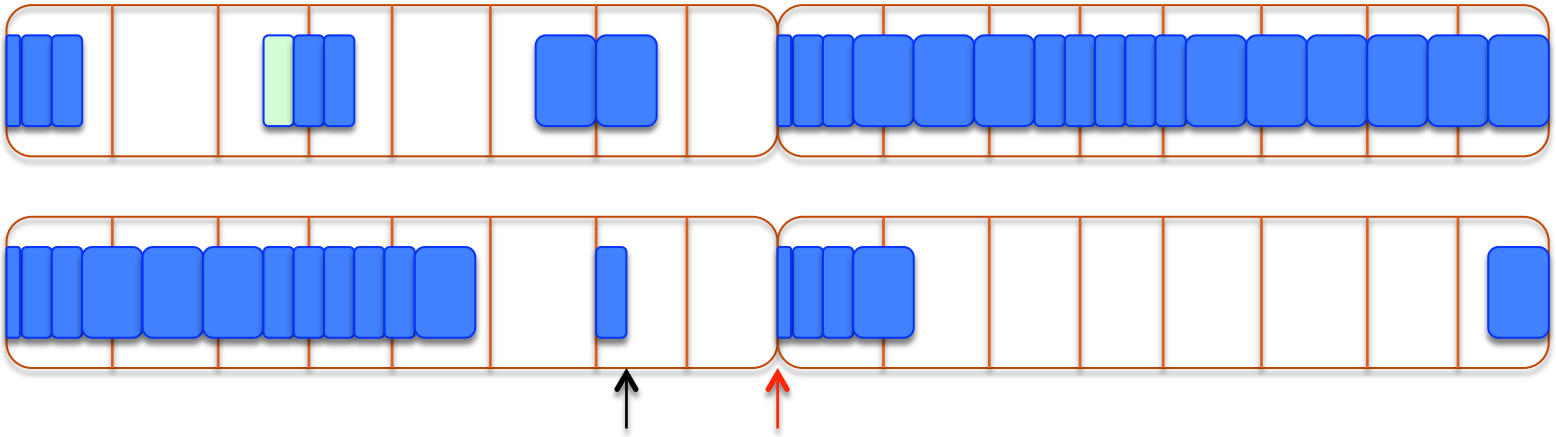
Opportunistic Defragmentation

- Apply opportunistically when
  - Unused recyclable blocks available
  - Previous collection did not yield enough space
- Mark source blocks at start of collection
  - Select blocks in order of most holes
  - Select as many blocks as possible based on space estimates
- Use same allocation mechanism as mutator
  - Evacuate during marking

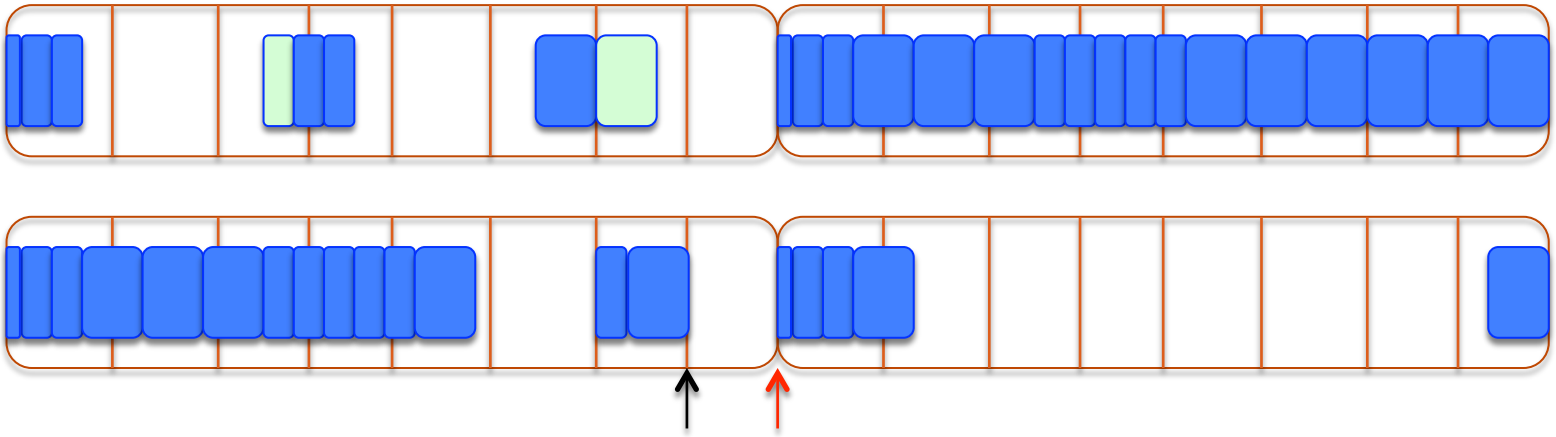
Defragmentation Illustration



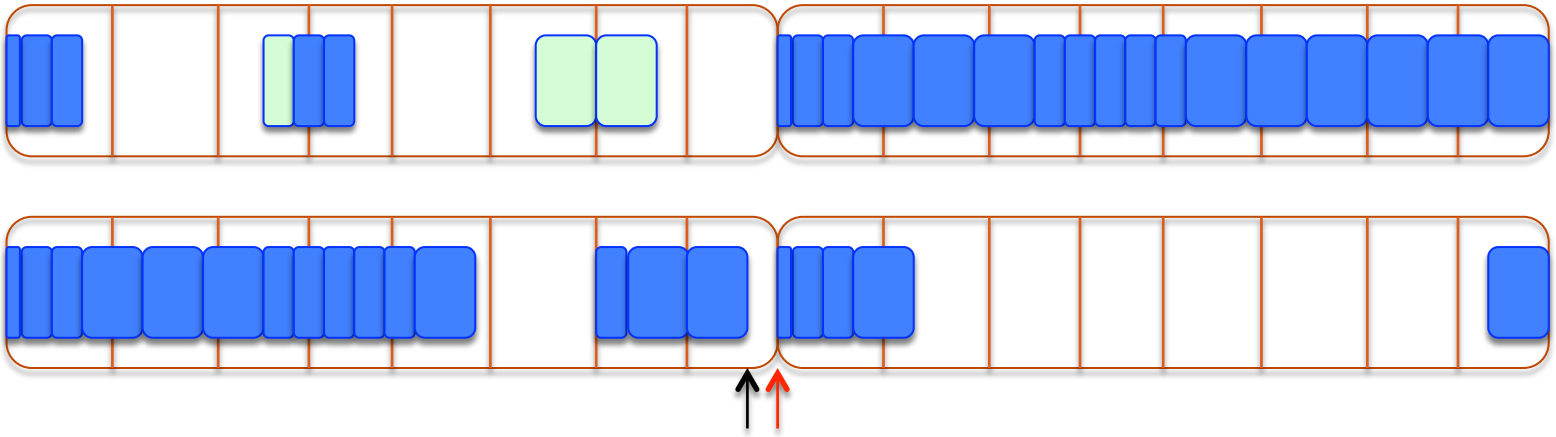
Defragmentation Illustration



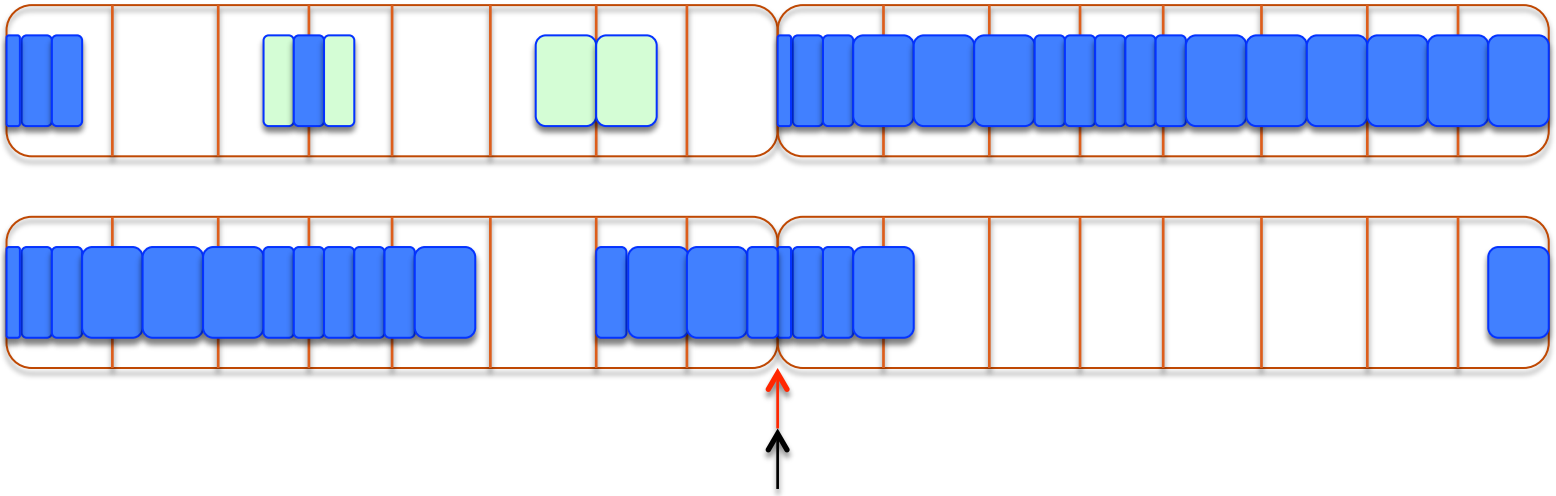
Defragmentation Illustration



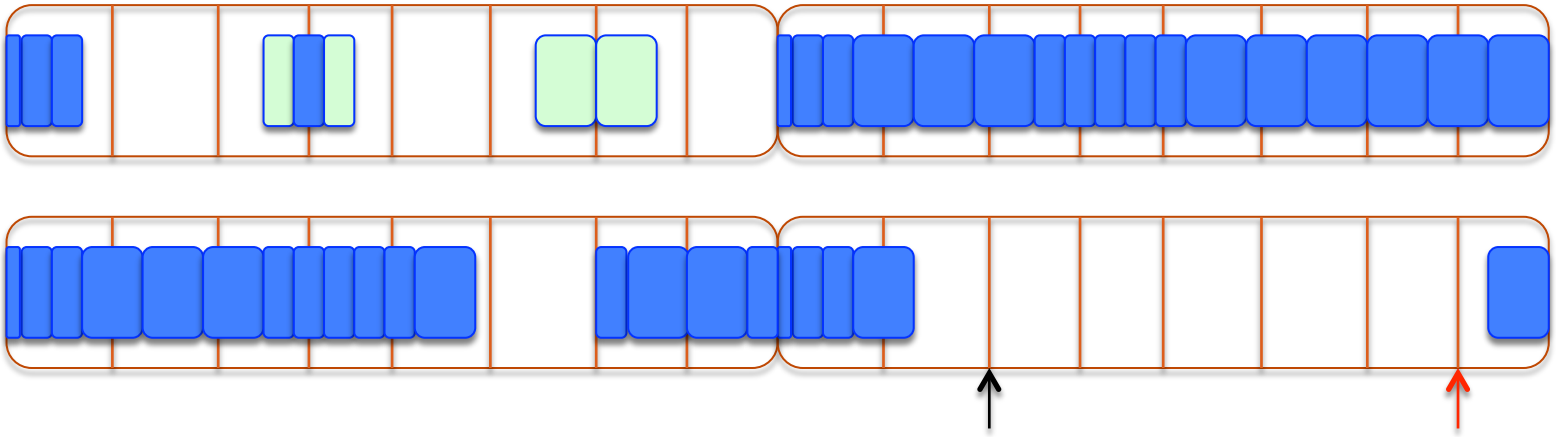
Defragmentation Illustration



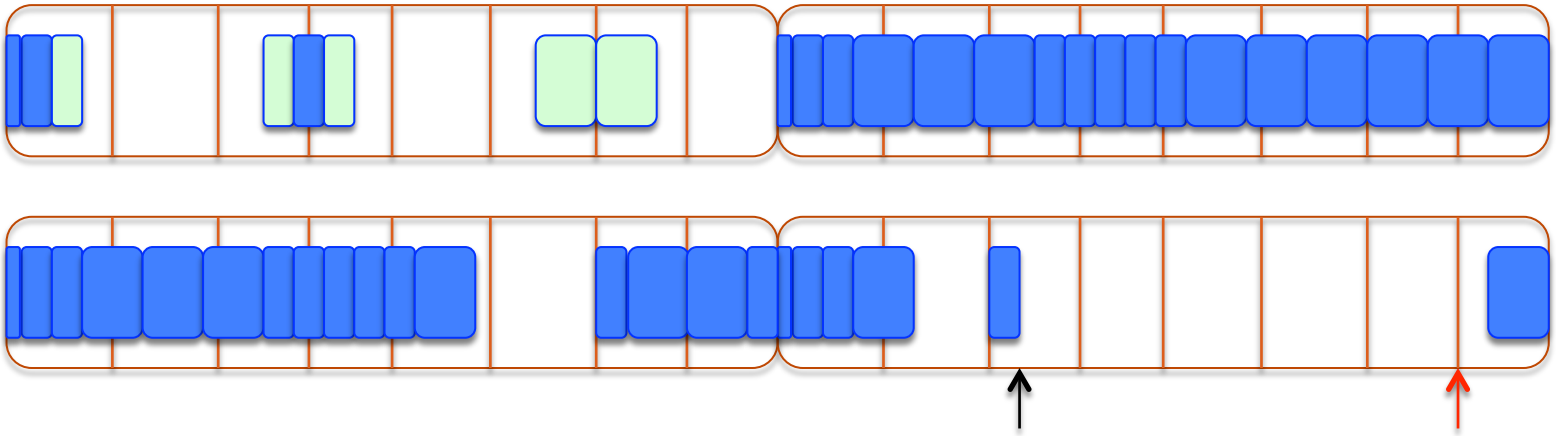
Defragmentation Illustration



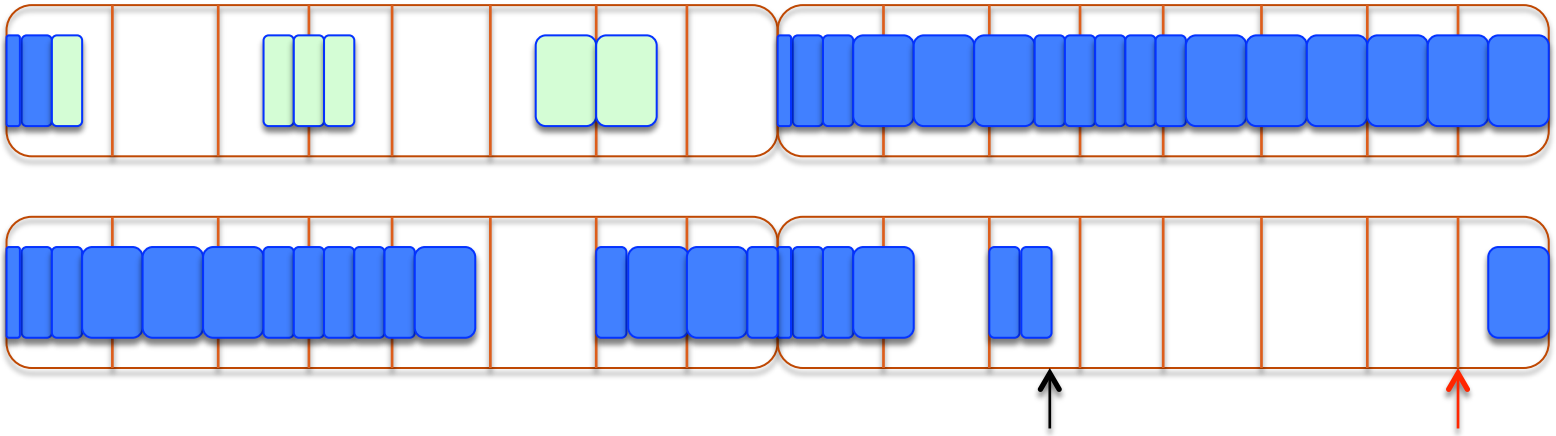
Defragmentation Illustration



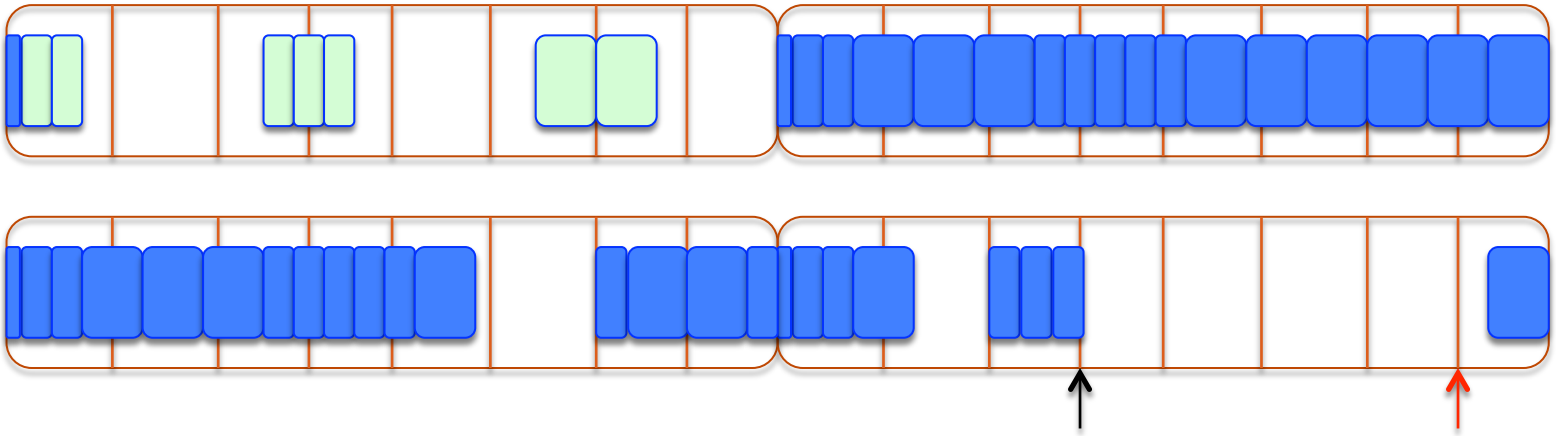
Defragmentation Illustration



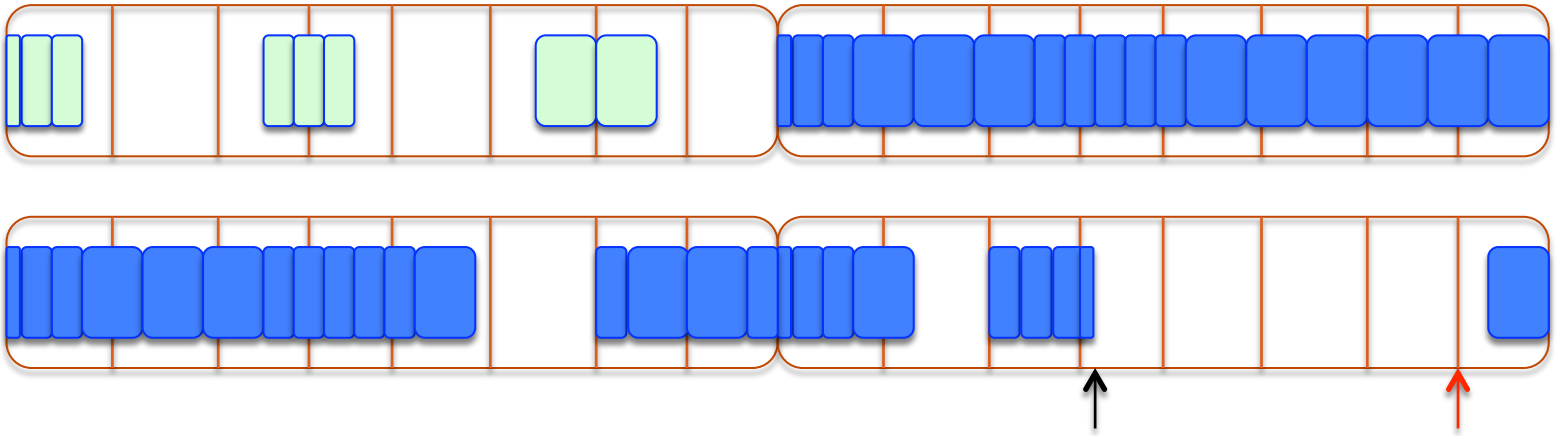
Defragmentation Illustration



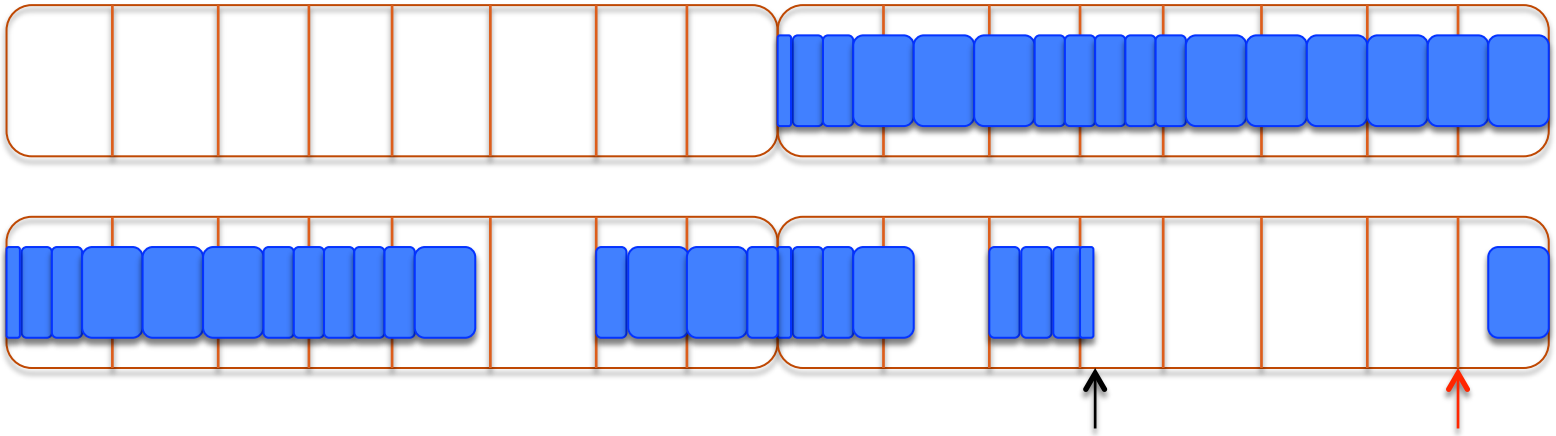
Defragmentation Illustration



Defragmentation Illustration



Defragmentation Illustration



More Implementation

- Overflow Allocation
  - Medium objects larger than line often skip holes
  - If current hole cannot accommodate, allocate in new block
- Parallel but not concurrent
  - Synchronized global allocator gives blocks to unsynchronized thread-local allocator
  - Use bytes for line marks (instead of bits)

More Implementation

- Large objects (>8KB) handled separately
  - Each block accommodates at least four immix objects
- Metadata in heap
  - 1B per line, 4B per block = 260B/32KB = 0.8%
- Supports pinning
  - Important feature of C#
- Headroom for defragmentation
  - 2.5% of heap

Evaluation

20 Benchmarks

DaCapo
SPECjvm98
SPEC jbb2000

Methodology

MMTk
Jikes RVM 2.9.3
(*Perf ≈ HotSpot 1.5*)
Replay compiler
Discard outliers
Report 95th %ile

Collectors

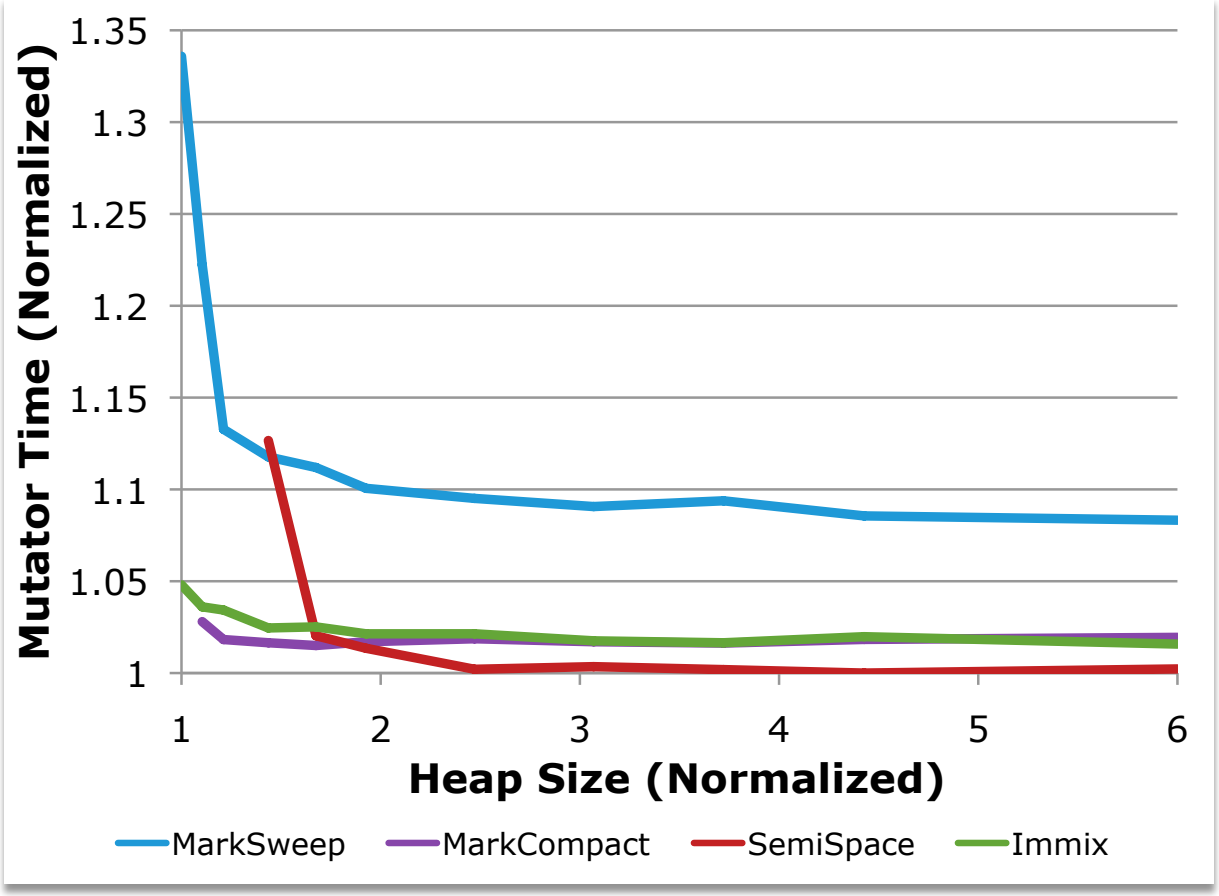
Full Heap
Immix
MarkSweep
MarkCompact
SemiSpace
Generational
GenIX
GenMS
GenCopy
Sticky
StickyIX
StickyMS

Hardware

Core 2 Duo
2.4GHz, 32KB L1, 4MB L2, 2GB RAM
AMD Athlon 3500+
2.2GHz, 64KB L1, 512KB L2, 2GB RAM
PowerPC 970
1.6GHz, 32KB L1, 512KB L2, 2GB RAM

Image from Blackburn's slides

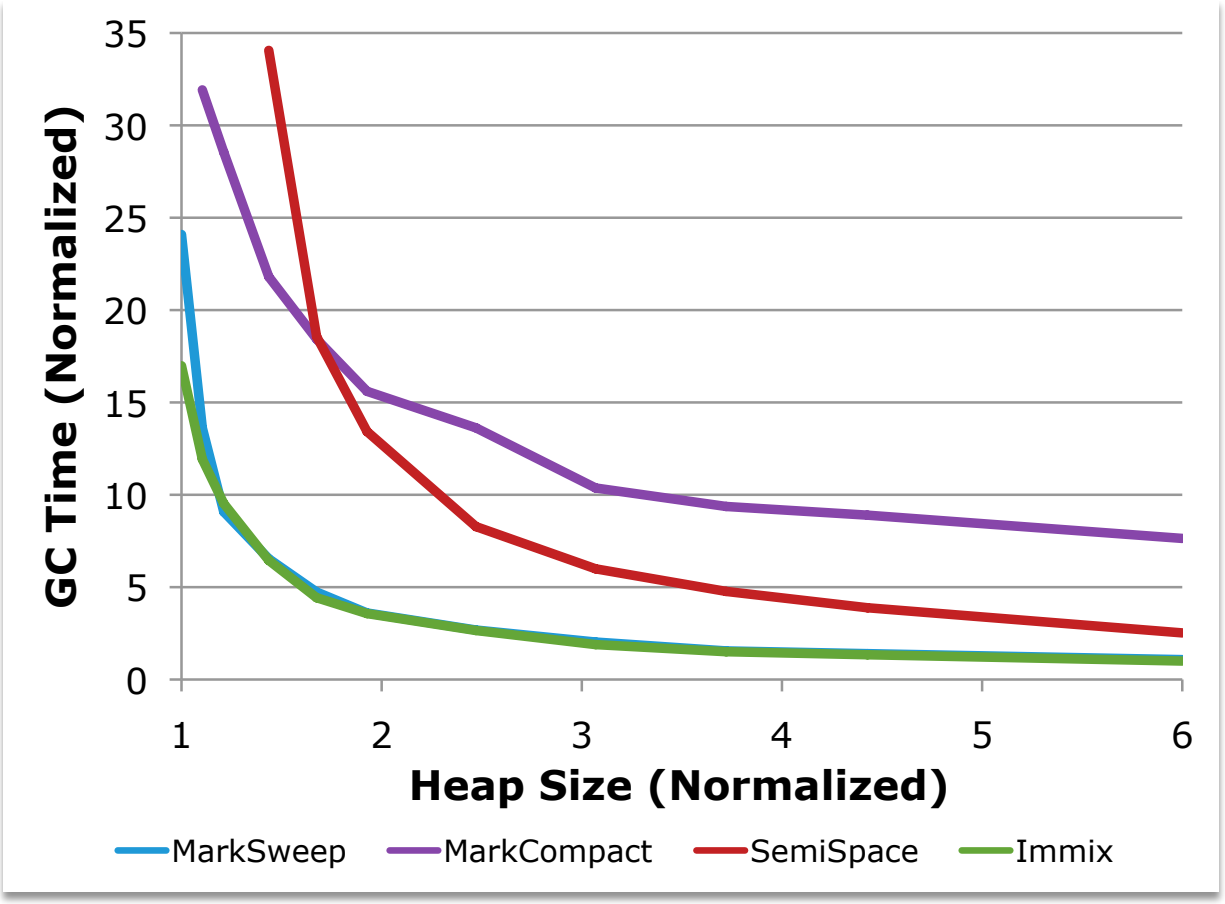
Mutator Time



Geomean of DaCapo, jvm98 and jbb2000 on 2.4GHz Core 2 Duo

Image from Blackburn's slides

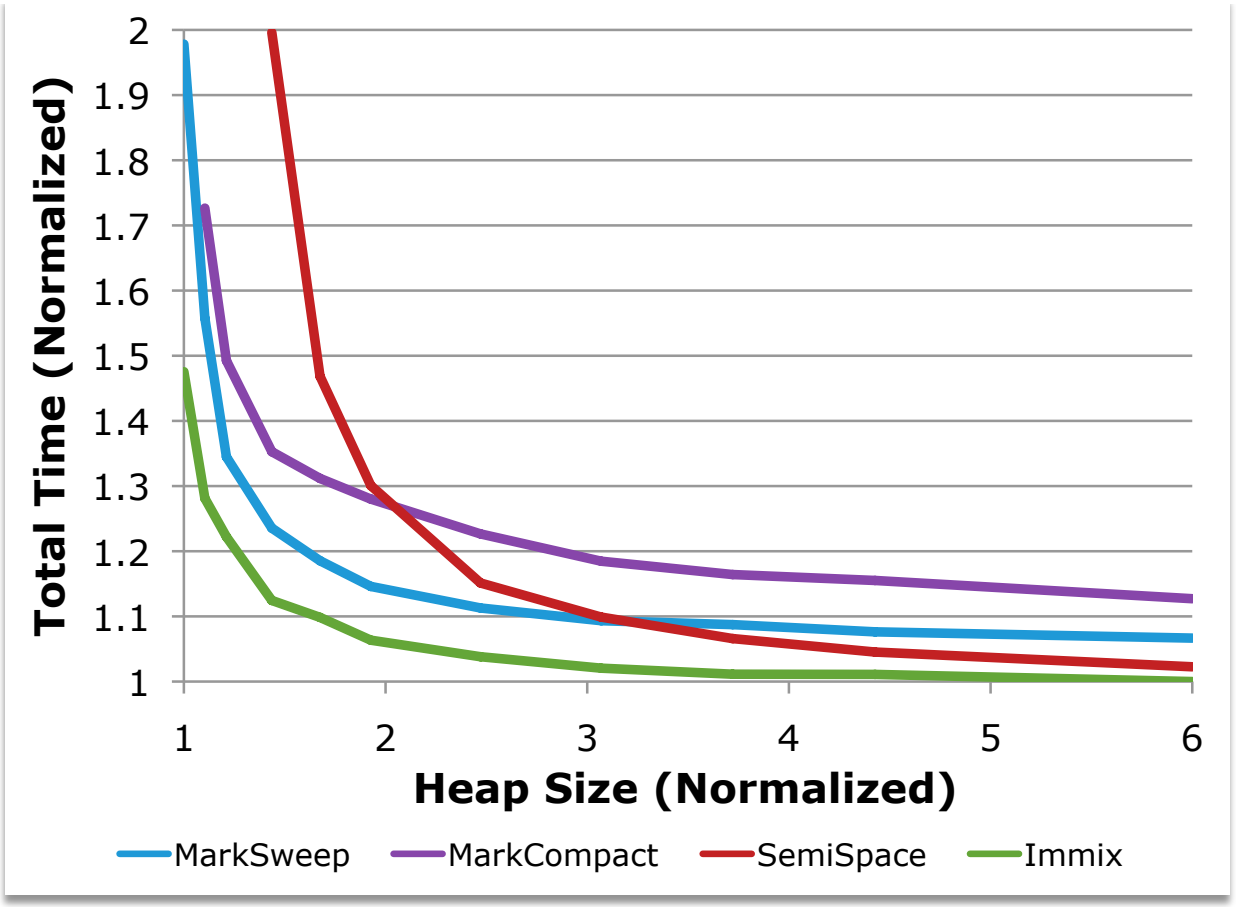
GC Time



Geomean of DaCapo, jvm98 and jbb2000 on 2.4GHz Core 2 Duo

Image from Blackburn's slides

Total Time



Geomean of DaCapo, jvm98 and jbb2000 on 2.4GHz Core 2 Duo

Image from Blackburn's slides

Minimum Heap

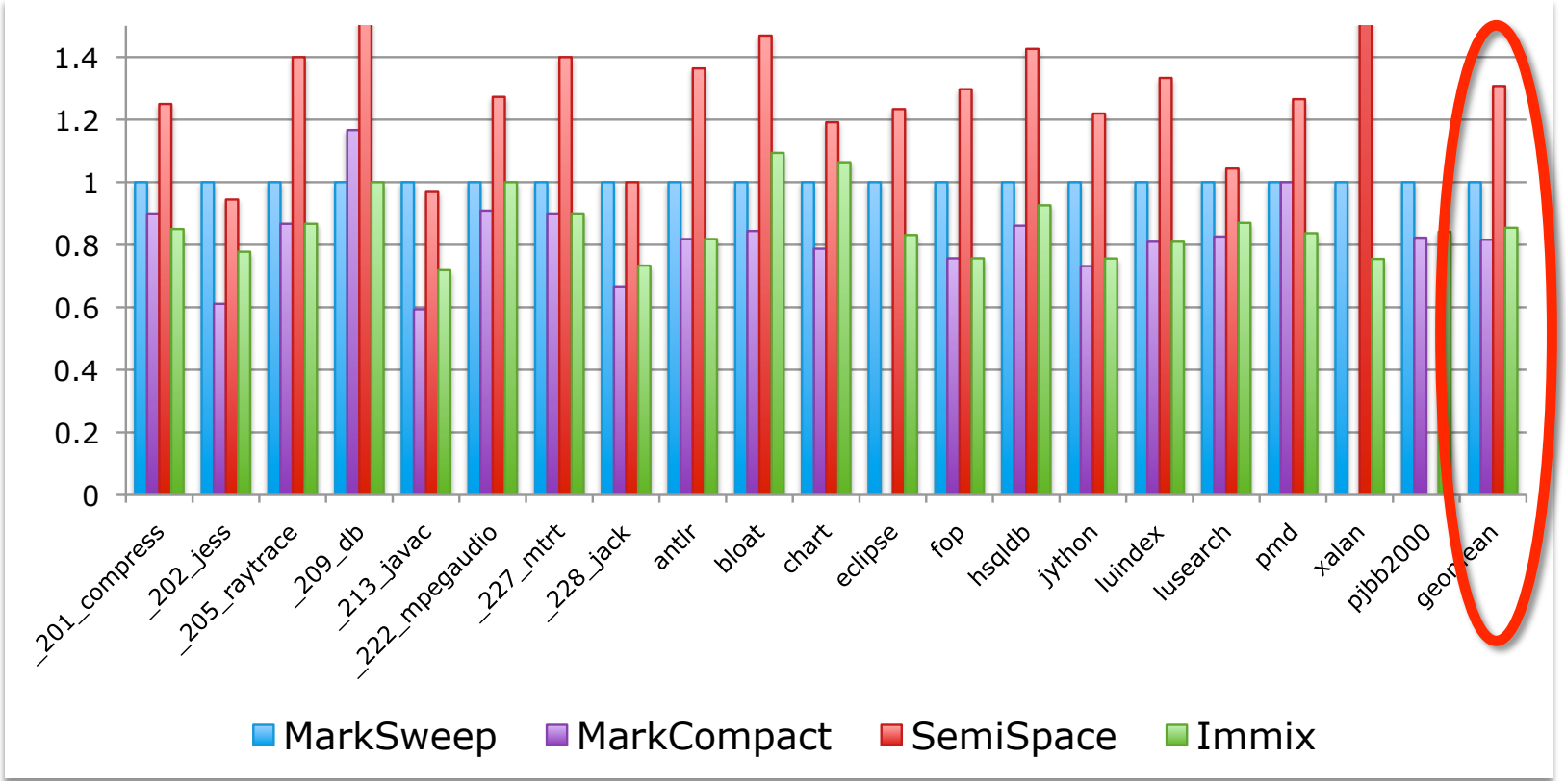
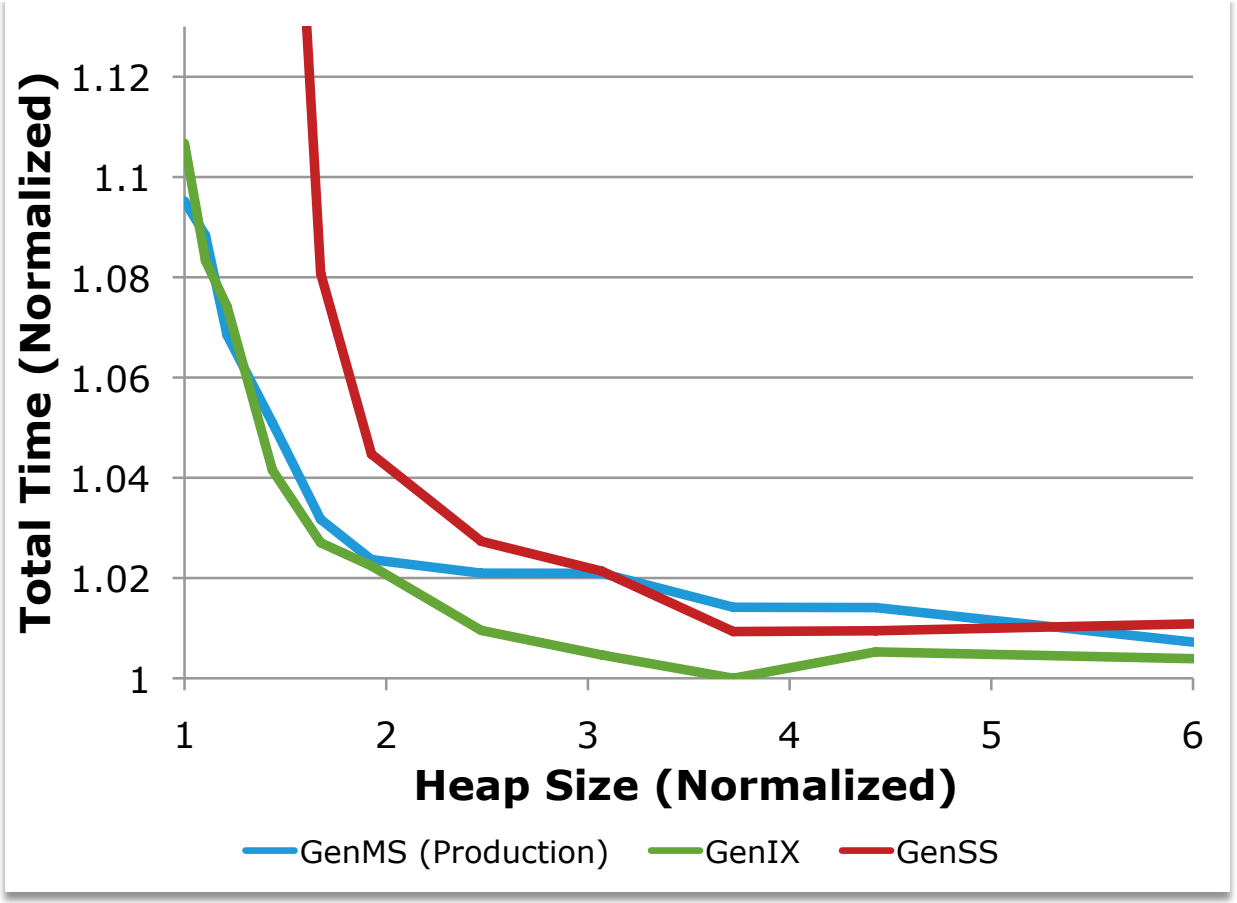


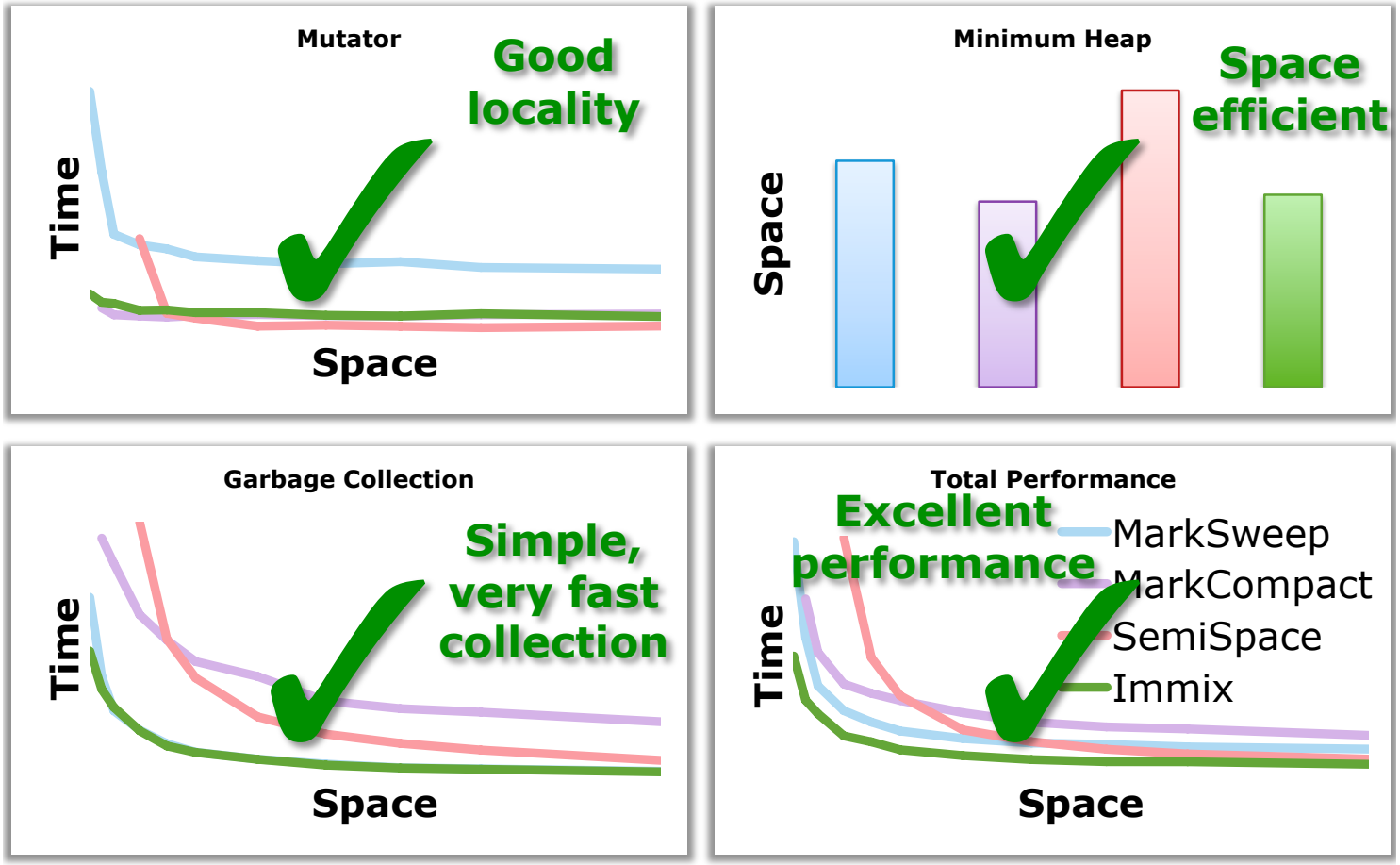
Image from Blackburn's slides

Generational Performance



Geomean of DaCapo, jvm98 and jbb2000 on 2.4GHz Core 2 Duo

Conclusion



Actual data, taken from geomean of DaCapo, jvm98, and jbb2000 on 2.4GHz Core 2 Duo

Thoughts

- Can Immix be made concurrent (e.g. “real time”)?
- What about longer benchmarks?
  - OLTP equivalent for Java?
- Defragmentation candidate selection?
  - What is the initial available space?
- Defragmenting more often should help mutator locality
  - Could it become a net win for total performance?