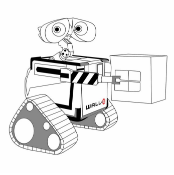
Immix: A Mark-Region Garbage Collector with Space Efficiency, Fast Collection, and Mutator Performance

Stephen M. Blackburn Kathryn S. McKinley

PLDI '08 Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation



presented by
Michael Lippautz
Concurrency and Memory Management Seminar
WS 2010
University of Salzburg

Why do we need GC? Where are we?

2008

- Programmers are more and more choosing managed languages for modern applications (safety)
- Lots of short/medium lived objects

Problem

- $\Rightarrow$ GC has a direct impact on program performance
- Tradeoff between time and space

Goal

Improve existing collection strategies

Immix

(let's figure out what this is about)

Mark-Region Garbage Collector

- Space efficiency - **space**
- Fast collection - **time**
- Mutator performance - **latency**

$\Rightarrow$ 3 dimensions; design space

Immix

(let's figure out what this is about)

How does this work?

Mark-Region Garbage Collector

Which strategies, how, why?

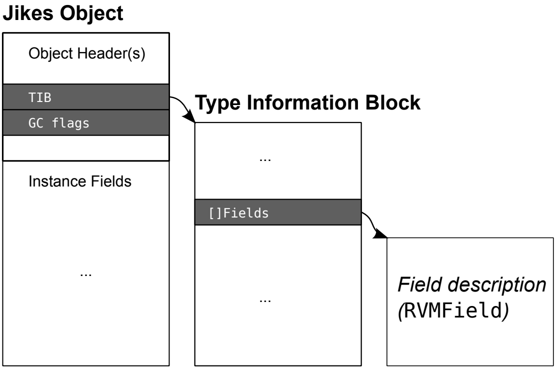
- Space efficiency - **space**
- Fast collection - **time**
- Mutator performance - **latency**

$\Rightarrow$ 3 dimensions; design space

Contents

Environment - the bigger picture

- Java
- Jikes RVM
- GC on object level (not chunks of memory)
  - Object model



A *very brief* overview

Only tracing GCs are discussed, but there do exist others, such as:

- Reference counting GCs
- Mixtures

Memory management utilizing a tracing GC

Basic approach

Determine which objects are reachable, discard the other ones

A *very brief* overview (cont)

Terms

- *Allocation* of new objects
- *Identification* of live objects (reachability analysis)
- *Reclamation* of free memory

Types

- Moving vs non-moving
- Stop-the-world vs incremental
- Precise vs conservative (pointers)

Reclamation strategies

Sweep-to-free-list

1. Allocate from a free list
2. Mark live objects
3. Sweep-to-free-list

Good: Time/space efficient
Bad: Locality
Examples: Mark-sweep (Naïve, tri-color)

Reclamation strategies (cont)

Evacuate

1. Move live objects to new space
2. Reclaim old space

Good: Locality, contiguous allocation
Bad: 2x space, slow (copy)
Example: Semi-space

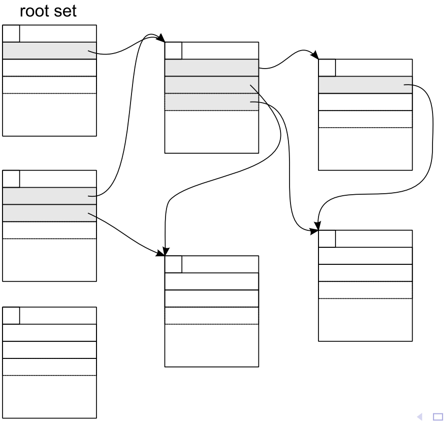
Compact

1. Move live objects to one end of the same space

Good: Locality, contiguous allocation
Bad: Multiple passes over heap, slow (copy)
Example: Mark-compact

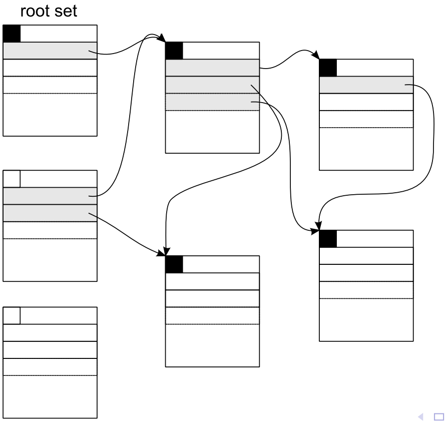
Naïve Mark-Sweep

- Mark phase
- Sweep phase



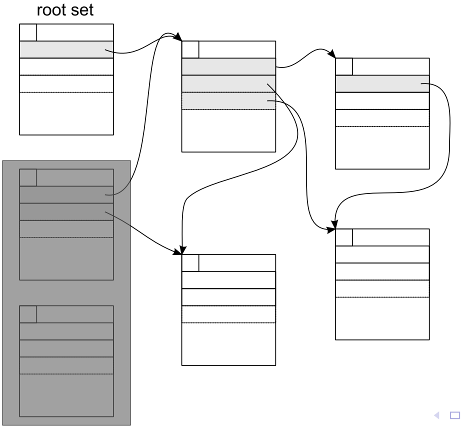
Naïve Mark-Sweep

- **Mark phase**
- Sweep phase



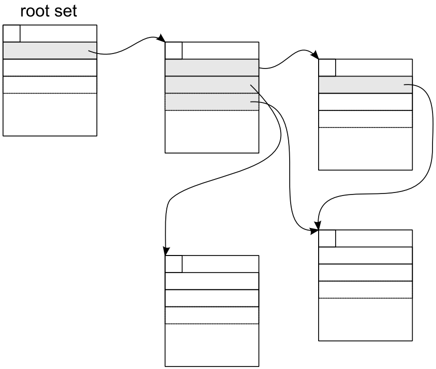
Naïve Mark-Sweep

- Mark phase
- **Sweep phase**



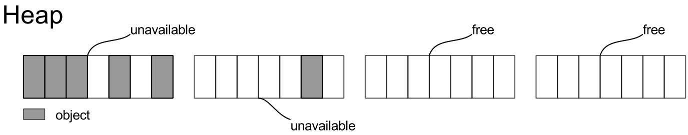
Naïve Mark-Sweep

- Mark phase
- Sweep phase



Naïve Mark-Region

- Memory is split into regions
- States:
  - *Free*
  - *Unavailable*
- Bump allocator in region
- Collector marks regions with at least one live objects as unavailable



Naïve Mark-Region(cont)

Questions

- How big can/should regions be?
- How does defragmentation work?

Immix

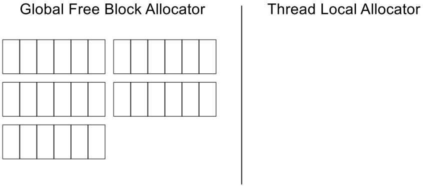
Sneak preview:

- Operates on a *block* (coarse grained) and *line* (fine grained) level
- Recycling of partially used blocks

Immix algorithm

Initial allocation

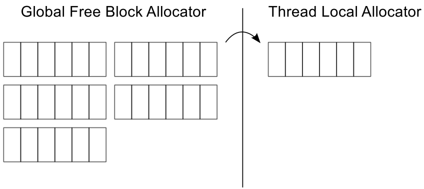
- **Initially, all blocks are empty**
- Thread-local allocator obtains a block from a global pool
- A full block triggers another request
- A full heap triggers a collection



Immix algorithm

Initial allocation

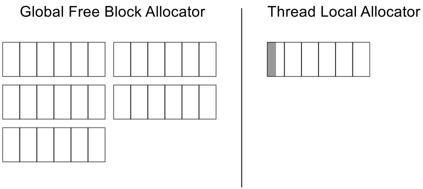
- Initially, all blocks are empty
- **Thread-local allocator obtains a block from a global pool**
- A full block triggers another request
- A full heap triggers a collection



Immix algorithm

Initial allocation

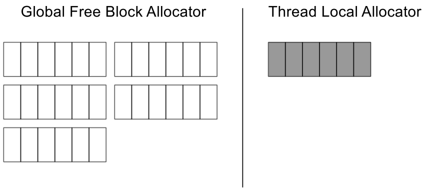
- Initially, all blocks are empty
- Thread-local allocator obtains a block from a global pool
- A full block triggers another request
- A full heap triggers a collection



Immix algorithm

Initial allocation

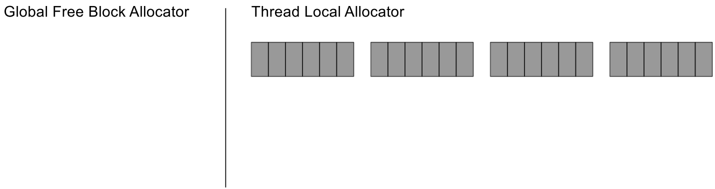
- Initially, all blocks are empty
- Thread-local allocator obtains a block from a global pool
- **A full block triggers another request**
- A full heap triggers a collection



Immix algorithm

Initial allocation

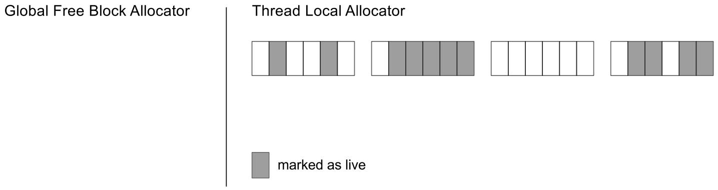
- Initially, all blocks are empty
- Thread-local allocator obtains a block from a global pool
- A full block triggers another request
- **A full heap triggers a collection**



Immix algorithm (cont)

Identification

- Collector traces live objects by performing a transitive closure
- **Marks objects and their lines**

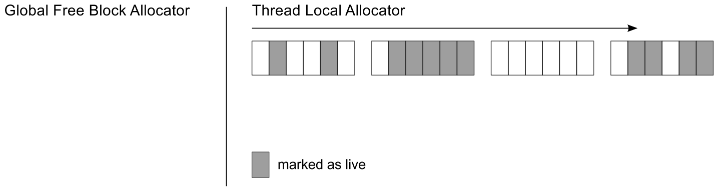


Immix algorithm (cont)

Reclamation

After trace completion the collector performs a *coarse-grained-sweep*.

- **Linearly scans line map to find free blocks and free lines**
- Returns completely free blocks to global allocator
- Recycles (marks) partially free blocks for the next phase

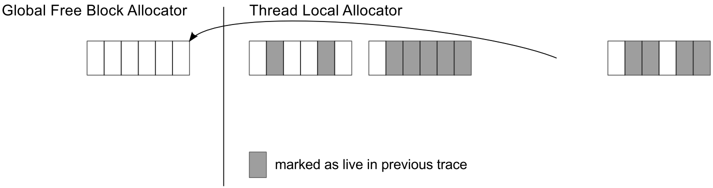


Immix algorithm (cont)

Reclamation

After trace completion the collector performs a *coarse-grained-sweep*.

- Linearly scans line map to find free blocks and free lines
- **Returns completely free blocks to global allocator**
- Recycles (marks) partially free blocks for the next phase

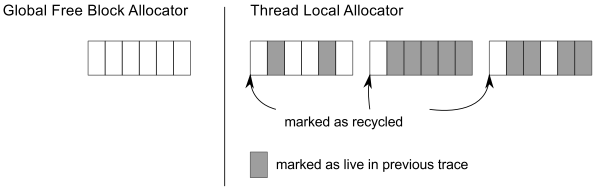


Immix algorithm (cont)

Reclamation

After trace completion the collector performs a *coarse-grained-sweep*.

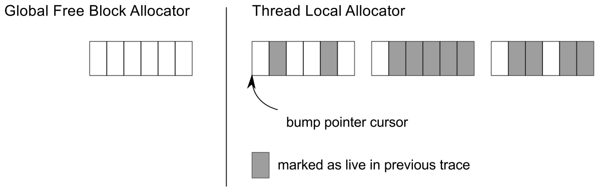
- Linearly scans line map to find free blocks and free lines
- Returns completely free blocks to global allocator
- **Recycles (marks) partially free blocks for the next phase**



Immix algorithm (cont)

Steady state allocation

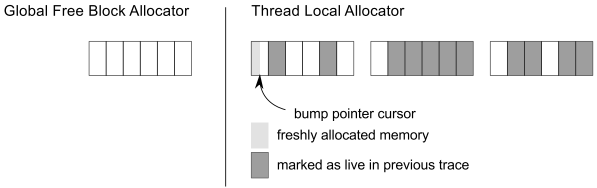
- **Thread-local allocator resumes allocation into recycled blocks**
- Bump allocates into free lines in a block
- Once there are no more recycled blocks available, a new one is requested from the global allocator
- Exhausted heap triggers another collection



Immix algorithm (cont)

Steady state allocation

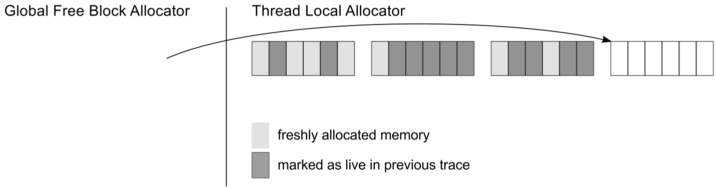
- Thread-local allocator resumes allocation into recycled blocks
- **Bump allocates into free lines in a block**
- Once there are no more recycled blocks available, a new one is requested from the global allocator
- Exhausted heap triggers another collection



Immix algorithm (cont)

Steady state allocation

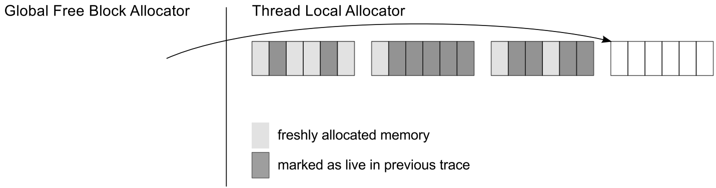
- Thread-local allocator resumes allocation into recycled blocks
- Bump allocates into free lines in a block
- **Once there are no more recycled blocks available, a new one is requested from the global allocator**
- Exhausted heap triggers another collection



Immix algorithm (cont)

Steady state allocation

- Thread-local allocator resumes allocation into recycled blocks
- Bump allocates into free lines in a block
- Once there are no more recycled blocks available, a new one is requested from the global allocator
- Exhausted heap triggers another collection



Details and policies

Basic algorithm works, but needs tuning to be competitive with existing GCs

Details and policies (cont)

Recycling policy

The allocator marks partly used blocks with atleast F lines as recyclable. Experimenting shows that $F = 1$ works best on average.

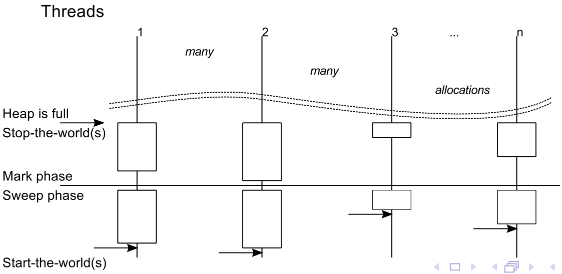
Allocation policy

Immix allocates into recyclable blocks first, before touching any free blocks. This is done to reduce competing for free blocks. I.e. thread-local allocators compete against each other and a large object space allocator for pages (indirectly over blocks).

Details and policies (cont)

Parallelism (detailed study is still open)

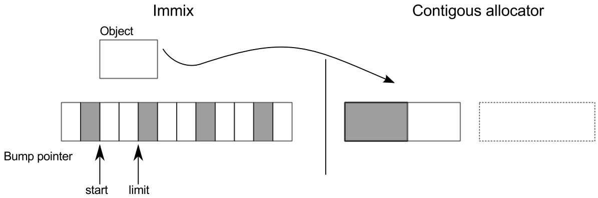
- Synchronization happens only when obtaining/returning a block from the global allocator
- TLAs can work unsychronized
- Transitive closure is performed parallel (worst case: marking an object multiple times)



Details and policies (cont)

Demand driven overflow allocation

- *Definition:* Medium sized objects are greater than one line
- *Problem:* Allocator wastes free space when searching for holes to store medium sized objects
- *Solution:* Each immix allocator is paired with a contigous allocator that uses empty blocks



(This is enough, since there are not that many medium sized objects [1])

Details and policies (cont)

In numbers...

- 128 Byte lines
- 32 KByte blocks

Conservative marking

- Again: Most object are below 128 byte of size (small)
- → Most objects fit in max. 2 lines
- Exact marking of lines is very costly
- Only the first line of an object is marked
- The first line of a hole is skipped upon allocation (implicit marking)
- Flags for small and medium ensure that this is still correct

Defragmentation

Lightweight opportunistic evacuation

- Pure mark-region would be non-moving and suffer from fragmentation
- *Evacuation* and/or *compaction*?

Candidate selection

- Round-robin policy like JRockit
- Defragmentation on demand like Metronome

Defragmentation

Lightweight opportunistic evacuation

- Pure mark-region would be non-moving and suffer from fragmentation
- *Evacuation* and/or ~~*compaction*~~?

Candidate selection

- Round-robin policy like JRockit
- Defragmentation on demand like Metronome

Defragmentation

Lightweight opportunistic evacuation

- Pure mark-region would be non-moving and suffer from fragmentation
- *Evacuation* and/or ~~*compaction*~~?

Candidate selection

- ~~Round-robin policy like JRockit~~
- Defragmentation on demand like Metronome

Defragmentation (cont)

Trigger

- If there exists one ore more recyclable blocks that have not been that have not been used by the allocator in the previous run
- If the previous collection did not yield enough free space

Which blocks?

Based on two histograms the collector decides which blocks should be used as source and targets

- Marked, calc. at the end of each run [marked lines/holes]
- Available, calc. on demand [free lines/holes]

Defragmentation (cont)

How?

Immix mixes marking and evacuation

- Candidate blocks have been selected before the collection run
- If an object in a candidate block is mark, it is also evacuated and a forward pointer is set

$\Rightarrow$ Single pass, evacuation based defragmentation

References

S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, *The DaCapo benchmarks: Java benchmarking development and analysis*, OOPSLA '06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programing, Systems, Languages, and Applications (New York, NY, USA), ACM Press, October 2006, pp. 169–190.

Stephen M. Blackburn and Kathryn S. McKinley, *Immix: A mark-region garbage collector with space efficiency, fast collection, and mutator performance*, PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation (New York, NY, USA), ACM, 2008, pp. 22–32.