Harmony GCv5 Overview

Xiao-Feng Li

2007-4-22

Outline

Harmony GCv5 goal and progress

Current status

Parallel load balance

Runtime adaptations

Parallel compactors

Miscellaneous

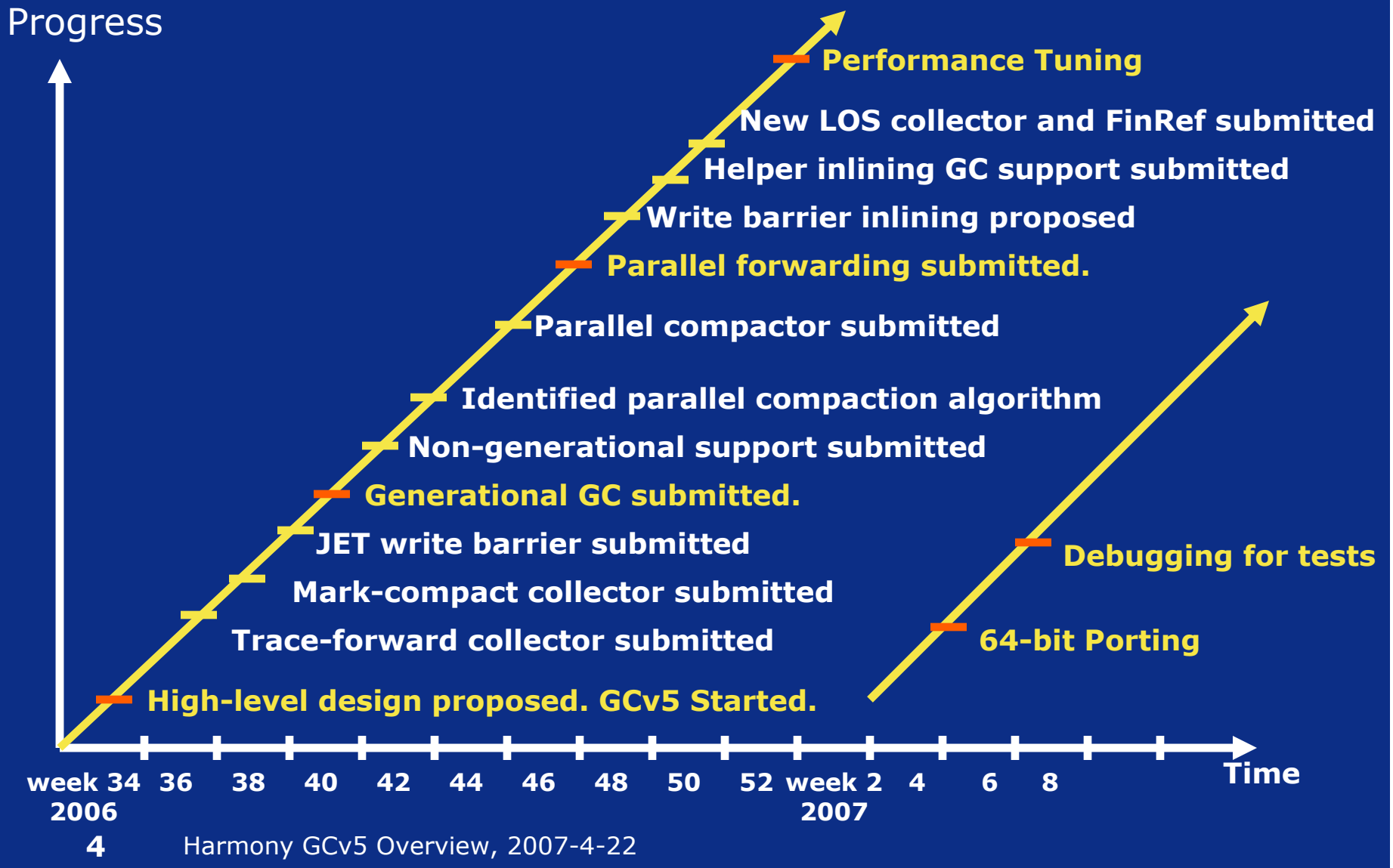
Harmony GCv5 Design Goal

Product-quality GC in both performance, robustness and flexibility

- Performance: Scalability and throughput
- Robustness: modularity and code quality
- Flexibility: configurability and extensibility

Proposal posted on August 22, 2006 in Harmony, and started then as a Harmony community project

Harmony GCv5 Development Progress



Outline

Harmony GCv5 goal and progress

Current status

Parallel load balance

Runtime adaptations

Parallel compactors

Miscellaneous

GCv5 Current Status: Parallel and Generational GC

Parallel and Generational GC

- Allocation is always in parallel
- Parallel marking of live objects
- Parallel copying of nursery object space (NOS)
- Parallel compaction of mature object space (MOS)
- More than one algorithms for comparison study

Largely in C language for efficiency and conciseness, with OO design and modularity

Modular Design: Space and GC

One collection algorithm decides one space

- Fspace: trace & forward, or mark & copy
- Mspace: Lisp2-compaction, or 2-pass compaction
- Lspace: mark-sweep, or compaction
- All spaces inherit Space class

GC structure is:

- Heap manager, managing multiple spaces, or
- Collection coordinator, coordinate collections of different algorithms
- Easy to configure or build new GC algorithms
  - Any GC implementation should inherit GC class

Modular Design: Threading and Allocator

Threading is abstracted into Mutator and Collector

- Mutator and Collector inherit from Allocator class
  - Mutator allocates during application execution
  - Collector allocates during garbage collection (for copying GC)
- Collector number is equal to processor number or specified in command line
- Currently collectors and mutators not run concurrently
  - In other word: stop-the-world GC
  - Concurrent GC is planned next

GCv5 Design: Generational Control

Generational control is decoupled from spaces and threading

- Spaces and threading know only GC, not Generational GC (GC_Gen)
- Three spaces
  - NOS: nursery object space (managed as a Fspace)
  - MOS: mature object space (managed as a Mspace)
  - LOS: large object space (managed as a Lspace)
- Space sizes are variable at runtime
  - Adjust the space boundary dynamically

GCv5 Design: Flexibility

Multiple configurations

- Generational or non-generational collection
- Partial heap collection or full heap collection
- NOS supports half-space copying (elder-first collection)
- Space variable size or fixed size at runtime

GCv5 is a Superset of GCv4 and GCv4.1 (conceptually)

Platforms and other functionalities

Supported platforms:

- OS: MS Windows and Linux
- Arch: 32-bit and 64-bit (compressed pointer)

Finalizer and weak reference support

- Implemented in native separate threads

GC core data structures

- Root set, remember set, task pool, etc.
- Designed with parallelization support

Platform-specific APIs are in one header file

Outline

Harmony GCv5 goal and progress

Current status

Parallel load balance

Runtime adaptations

Parallel compactors

Miscellaneous

GCv5 Parallelization Load Balance

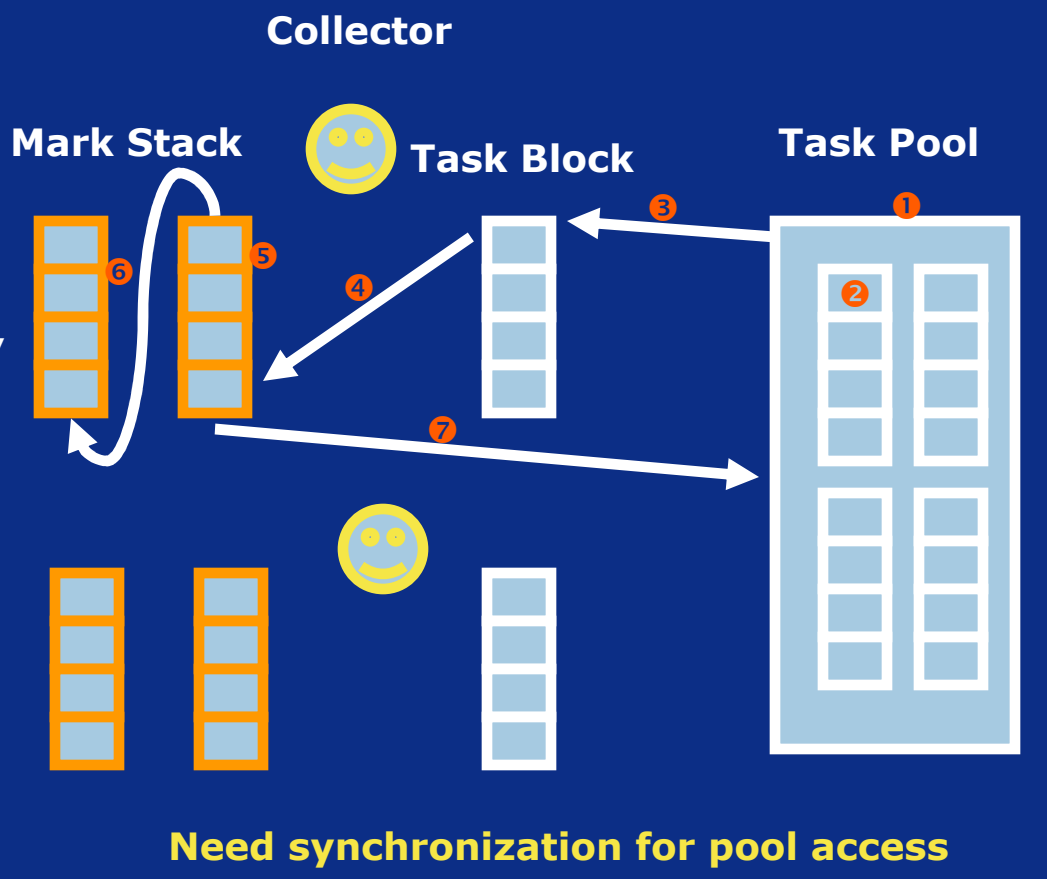
Three algorithms were experimented for marking and forwarding

1. Pool-sharing parallelization
2. Work-stealing parallelization [Flood, et al. JVM01]
   - Need deal with stack overflow
3. Task-pushing parallelization
   - No synchronization operations in GC marking
   - A paper published in IPDPS'07

Currently pool-sharing is selected for GCv5 code base

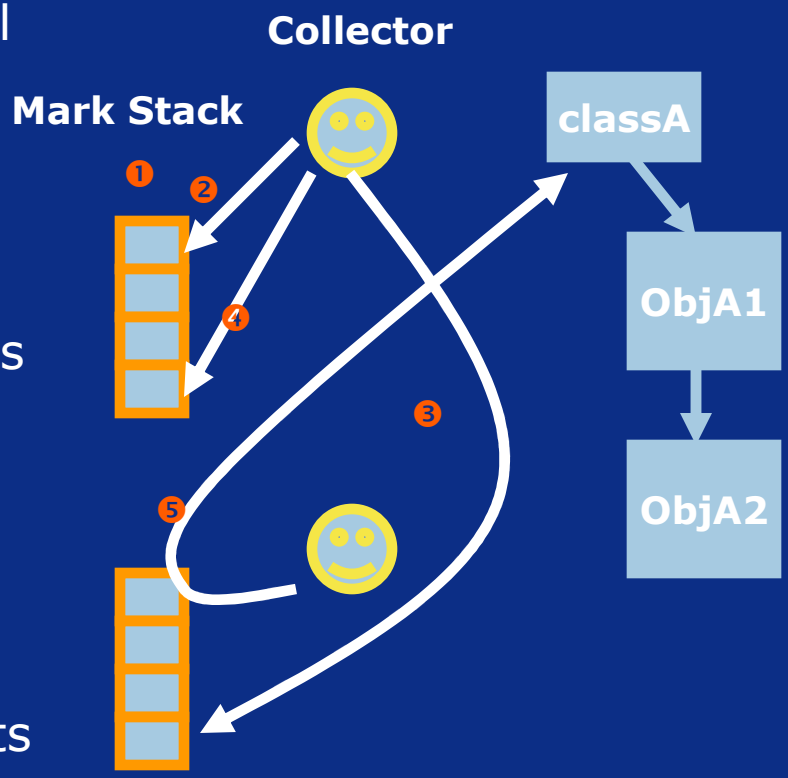
GCv5 Parallelization Load Balance: Pool-sharing in Parallel Marking

1. Shared Pool for task sharing
2. One reference is a task
3. Collector grabs task block from pool
4. Pop one task from task block, push into mark stack
5. Scan object in mark stack in DFS order
6. If stack is full, grow into another mark stack, put the full one into pool
7. If stack is empty, take another task from task block



GCv5 Parallelization Load Balance: Work-stealing in Parallel Marking

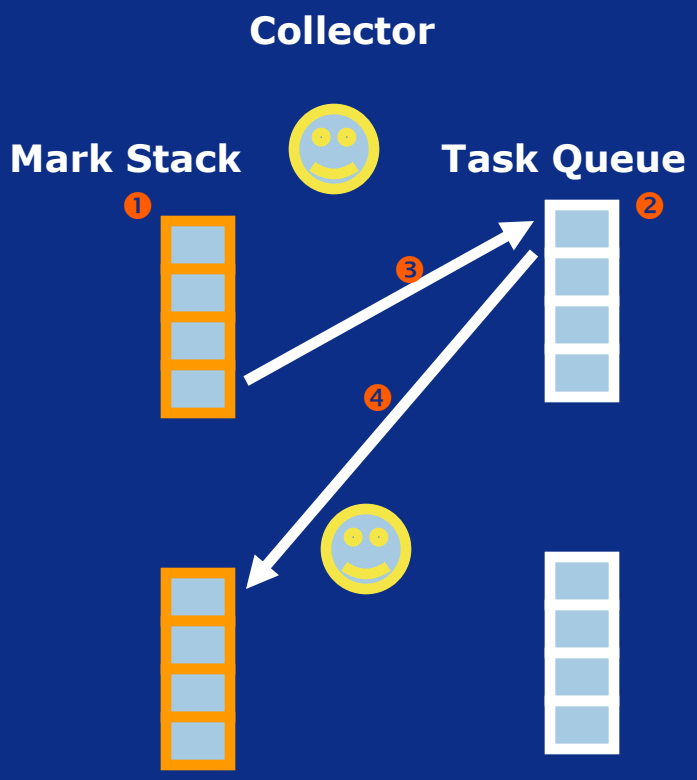
1. Each collector has a thread-local mark-stack, which initially has assigned root set references
2. Collectors operate locally on its stack without synchronization
3. If stack is empty, collector steals a task from other collector's stack's bottom
4. If stack has only one entry left, the collector need synchronization access
5. If stack is full, it links the objects into its class structure (should never happen in reality)



Need synchronization for task stealing

GCv5 Parallelization Load Balance: Task-pushing in Parallel Marking

1. Each collector has a thread local mark stack for local operations
2. Each collector has a list of output task queues, one for each other collector
3. When a new task is pushed into stack, the collector checks if any task queue has vacancies. If yes, drip a task from mark stack and enqueue it to task queue
4. When mark stack is empty, the collector checks if there are any entries in its input task queues. If yes, dequeue a task



No synchronization instruction !!

Outline

Harmony GCv5 goal and progress

Current status

Parallel load balance

Runtime adaptations

Parallel compactors

Miscellaneous

GCv5 Runtime Adaptation

Runtime adaptation is essential for good GCv5 performance

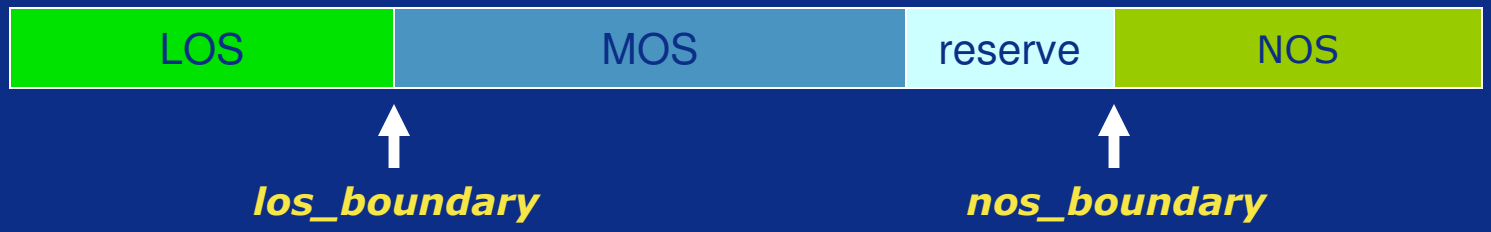
- Dynamic space size adjustment
  - So that no space wasted
- Dynamic major and minor collection switching
  - To achieve maximal throughput
- Dynamic switching between generational and non-generational mode
  - To leverage the advantages of both

The first two are default in GCv5 now, the last one is not

GCv5 Runtime Adaptation: Harmony GCv5 Space Adjustment

Default with three spaces: LOS, MOS and NOS

- The boundaries are runtime adjustable (throughput driven)



nos_boundary is adjusted after every collection

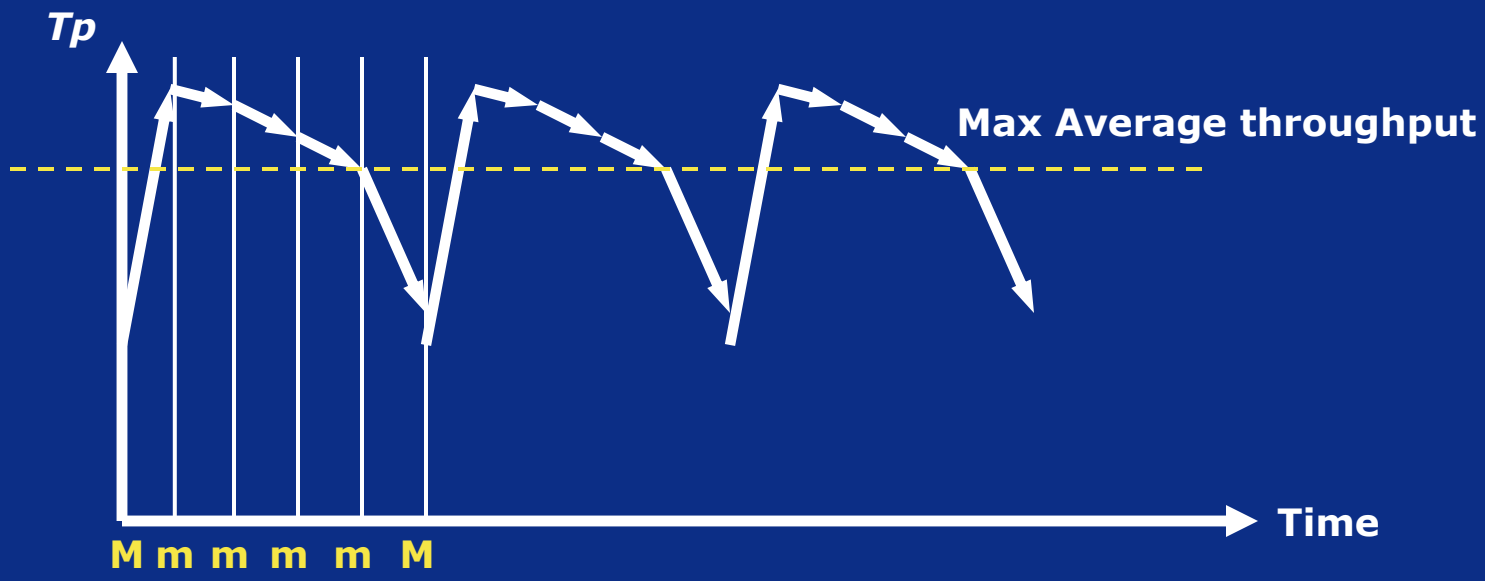
los_boundary is adjusted when necessary after major collection

If MOS reserve space is not enough to hold NOS survived objects, fall-back compaction happens.

GCv5 Runtime Adaptation: Minor and Major Collection Switch

Minor collection (**M**) is much faster than major collection (**m**)

- GC should use minor collection as more as possible
- Runtime adaptively switch (throughput driven) between **M** and **m**
  - Partial heap collection vs. full heap collection
  - A paper published in CAECW-10, 2007.



GCv5 Runtime Adaptation: Generational and Non-generational Switch

Gen mode is usually better than non-gen mode

- If application behavior matches with generational hypothesis, or
- If overhead for non-gen mode mark-scan is too high
- Write barrier has overhead in gen-mode
- Runtime adaptively switch between gen and non-gen mode
  - Hybrid mode gets better performance than either gen or non-gen
  - Real performance depends on workloads

Outline

Harmony GCv5 goal and progress

Current status

Parallel load balance

Runtime adaptations

Parallel compactors

Miscellaneous

GCv5 Compaction Algorithms

Three compaction algorithms developed

- Parallelized LISP2-basd mark-compactor
- Chained reference threaded-compactor
- 2-pass parallel move-compactor

Currently GCv5 use move-compactor and mark-compactor for respective collection scenarios

GCv5 Compaction Algorithm: Parallel Mark-Compactor

Parallelism granularity: block (default size is 32K)

- Source block: from which data are copied
- Destination block: to which data are copied

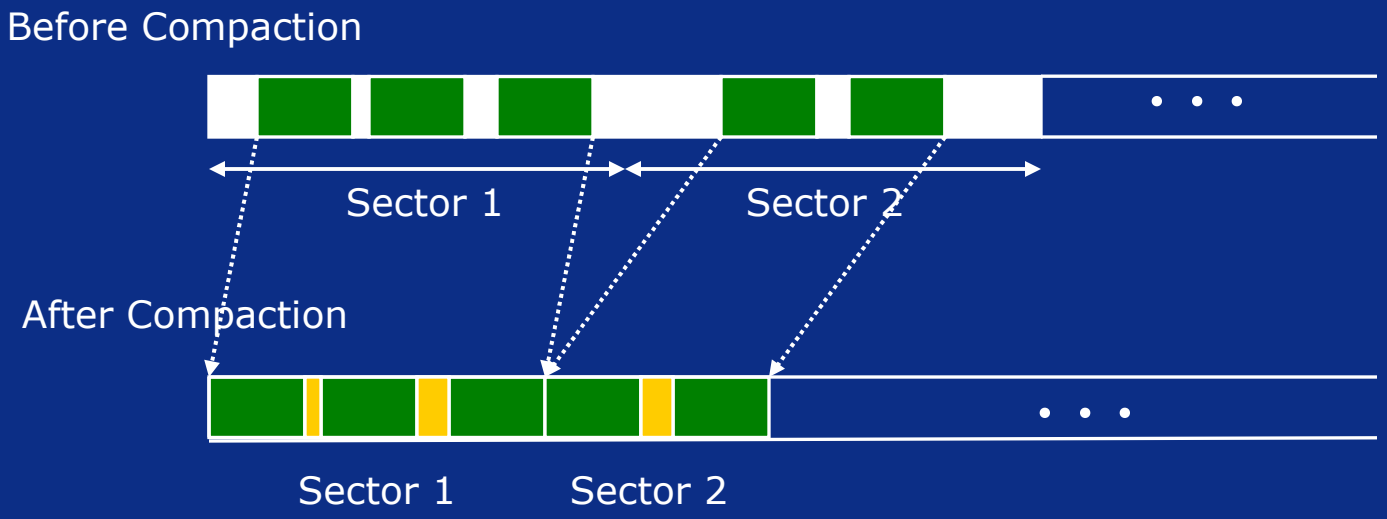
Key idea:

- During target address computing phase
  - Every target block maintains a list pointing to its source blocks
- During object moving phase
  - A collector grabs a source block from the lists one by one
  - A collector moves live objects from its src to dest block
- Guarantee data in one area has been moved before it is overwritten

GCv5 Compaction Algorithm: Parallel Move-Compactor

Has fewer passes than mark-compactor

- Idea based on Abuaiadh, et al. OOPSLA'04.
- Parallelization scheme in GCv5 is different



Free spaces between live objects in a sector are not compacted

Outline

Harmony GCv5 goal and progress

Current status

Parallel load balance

Runtime adaptations

Parallel compactors

Miscellaneous

Performance Tuning and Debugging

Performance tuning

- Parallelization
- Runtime adaptation
- Data prefetching
- Inlining of allocation and write barrier routines
  - Written as short methods in Java
  - Only for fast path, and fall back to native GC code

Debugging took time

- Mainly in finalizer interactions with DRLVM threading subsystem

Work in Progress

Harmony GCv5 Solidification for robustness/performance

Parallel scalability on large number of processors/cores

64-bit support tuning for large heap

Acknowledgement

Harmony GCv5 is a result of the community, just to name a few:

Chunrong Lai, Ligang Wang, Yunan He, Ji Qi, Ivan T. Volosyuk
Steve Blackburn, Washburn Weldon, Rana Dusgupta
Mikhail Fursov, Vladimir Ivanov

Sorry to the developers whose names are missing here.

Publications During GCv5 Development

Ming Wu, Xiao-Feng Li, *Task-pushing: a Scalable Parallel GC Marking Algorithm without Synchronization Operations*, In Proceedings of 21st IEEE International Parallel & Distributed Processing Symposium (IPDPS 2007), Long Beach, CA, March 26, 2007.

Chunrong Lai, Ivan T Volosyuk, and Xiao-Feng Li, *Behavior Characterization and Performance Study on Compacting Garbage Collectors with Apache Harmony*, In Proceedings of Tenth Workshop on Computer Architecture Evaluation using Commercial Workloads (CAECW-10) held with HPCA-13, Phoenix, AZ, Feburary 2007.