Concurrent Compaction in JVM Garbage Collection

Jacob P. Opdahl

University of Minnesota, Morris

opdah023@morris.umn.edu

December 5, 2015

Automatic Memory Management

Implicit allocation and deallocation of memory

Languages: Java, C#, Python, and more

- We focus on the Java Virtual Machine and languages it supports

Abstracts details away from the developer

Implicit Deallocation

Memory is a finite resource

Garbage: objects that are no longer reachable

Garbage Collection (GC): detecting and removing garbage

Stopping the World

GC requires processing resources

When only one processor is used, collectors *stop the world*

Problem: applications today are subjected to increasing pauses

- More memory
- More strenuous applications

Use parallel processing to solve!

Outline

Outline

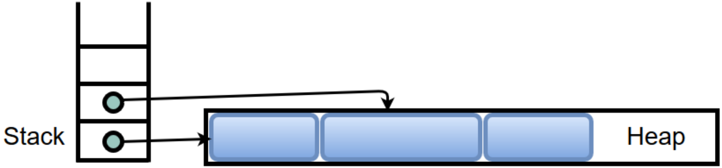
Memory

Heap: contiguous memory location used by the JVM

- Objects are stored here

Stack: memory for short-lived, method-specific values

- Stores *references*: memory addresses of objects



GC Cycle

Set Condemnation: determine which objects are garbage

Compaction: reclaim memory while fighting heap fragmentation

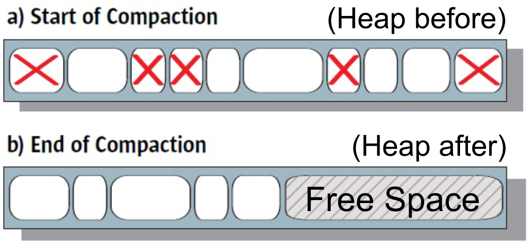
Set condemnation done by *tracing*

- Detect all reachable objects by chaining references

Compaction

Consists of two steps

- *Relocation*: move objects
  - *from-space* and *to-space*
- *Remapping*: update object references



Processes and Threads

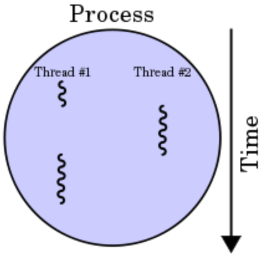
Process: instance of a program being run

- Examples: JVM, word processor
- Has its own memory space

Thread: sequence of independent instructions that can run on its own

- Component of a process
- Possible to run multiple in parallel

Parallel Processing: running multiple threads simultaneously with multiple processors



wikipedia.org/wiki/Thread_%28computing%29

Synchronization

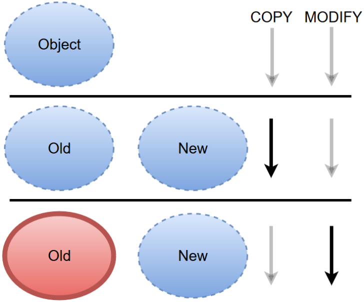
Threads do not coordinate automatically

- Independent instructions!

Poses new challenges

- Example: losing object modifications

Need to keep threads synchronized



Concurrency

We distinguish between application threads and GC threads

Concurrent GC: collector runs at the same time as the application

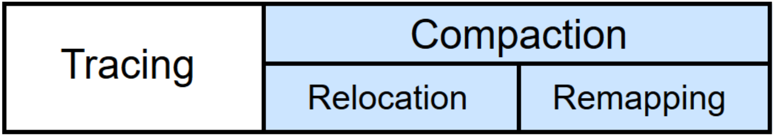
- Does not stop the world

Our focus: concurrent compaction!

Outline

Continuously Concurrent Compacting Collector (C4)

Researchers: G. Tene, B. Iyengar, and M. Wolf at Azul Systems

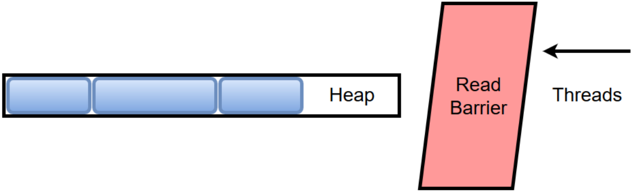


Loaded Value Barrier (LVB)

Read Barrier: instructions to run before a thread accesses memory

LVB protects from-space from application threads

- From-space: where objects were located before moving

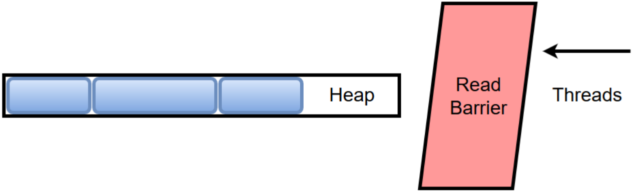


Loaded Value Barrier (LVB)

Rule: application can only use moved objects

- If a thread breaks this, the barrier will correct the situation

This facilitates concurrent relocation and remapping

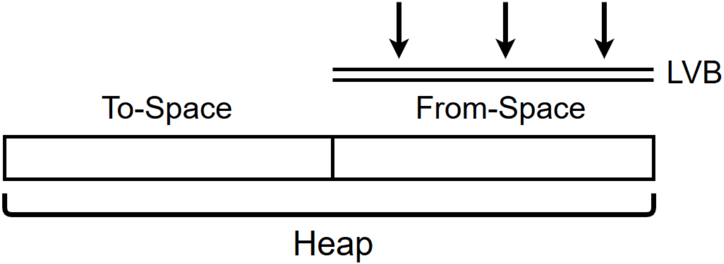


Concurrent Relocation

GC threads simply relocate objects

All references point to from-space!

- Application threads certain to trigger LVB



Concurrent Relocation

LVB instructions for applications threads

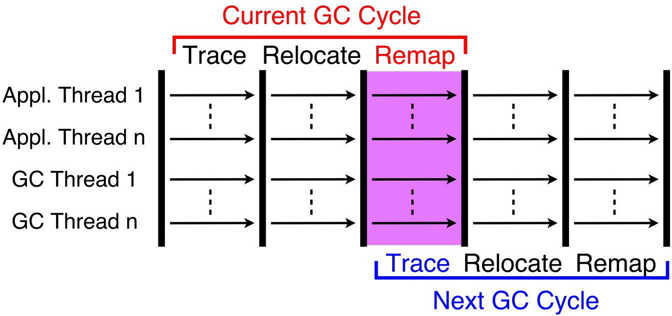
- If the object was moved, find it
- If the object is being moved, wait
- If the object is unmoved, move it

In all cases, update the reference after using the object in to-space

Concurrent Remapping

To update all references, need to traverse all reachable ones

Combine remapping with next tracing phase



Testing Environment

Tested against two collectors with non-concurrent compaction

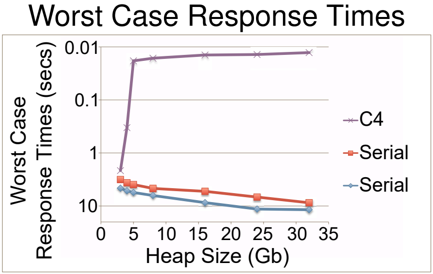
Improvements from concurrent compaction

Server environments used

Results

C4:

- Fastest response times
- Maintains them for largest range of heap sizes
- Least impact on application



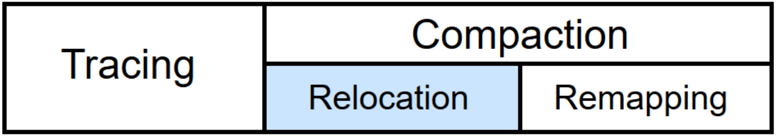
Outline

Field Pinning Protocol (FPP)

Implemented into a *host* GC algorithm

Differs from C4 - barrier-free!

Researchers: E. Österlund and W. Löwe at Linnaeus University



Hazard Pointers

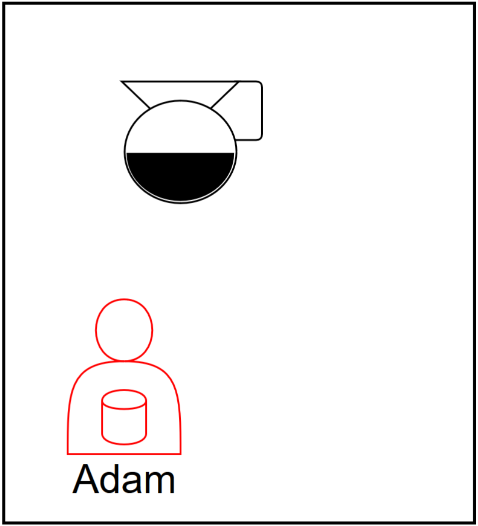
Hazard Pointers: values that show which objects an application thread is accessing

Inform other threads of objects that are in use

Main goal: safely access objects without worrying about relocation

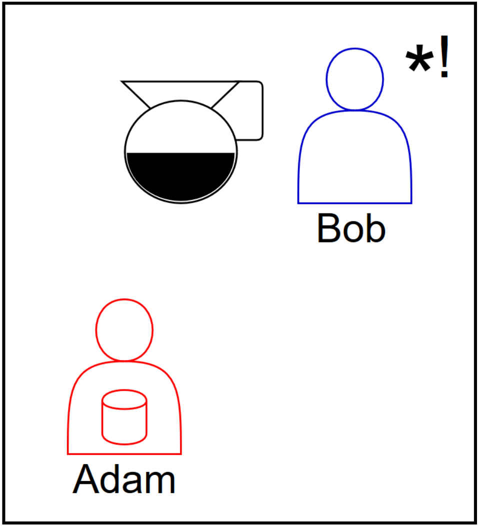
Example

- Object = Coffee
- Application Thread = Person w/ Coffee Cup



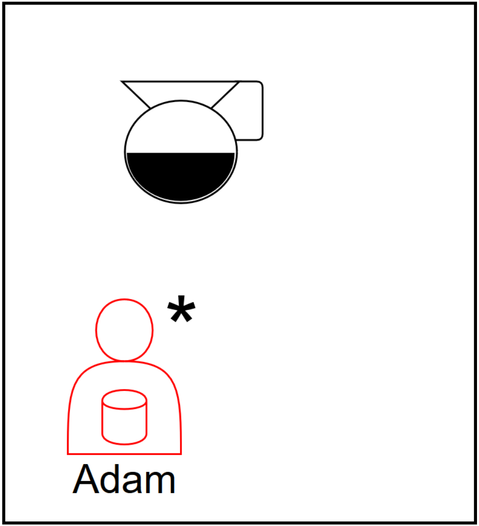
Example

- Object = Coffee
- Application Thread = Person w/ Coffee Cup
- Relocation Thread = Person
- * = Responsible
- ! = Impeded



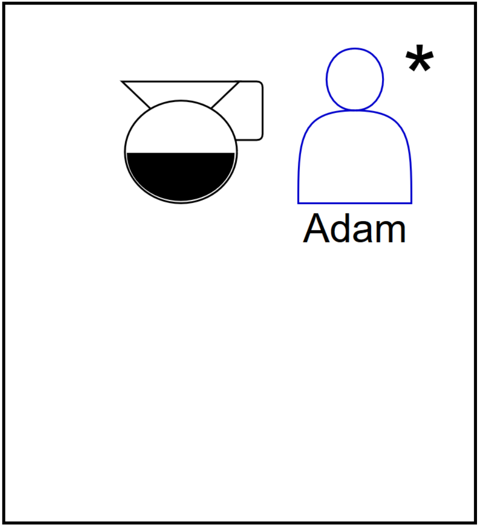
Example

- Object = Coffee
- Application Thread = Person w/ Coffee Cup
- Relocation Thread = Person
- * = Responsible
- ! = Impeded



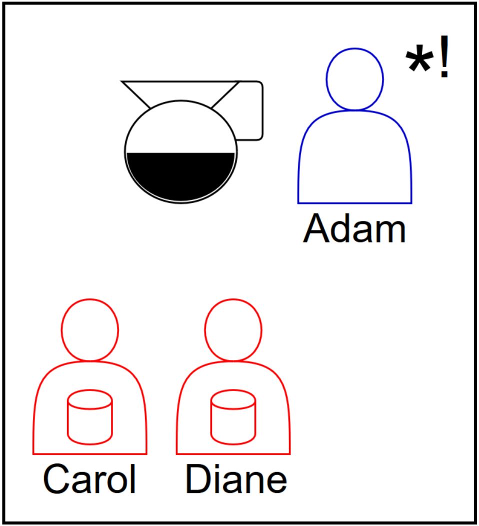
Example

- Object = Coffee
- Application Thread = Person w/ Coffee Cup
- Relocation Thread = Person
- * = Responsible
- ! = Impeded



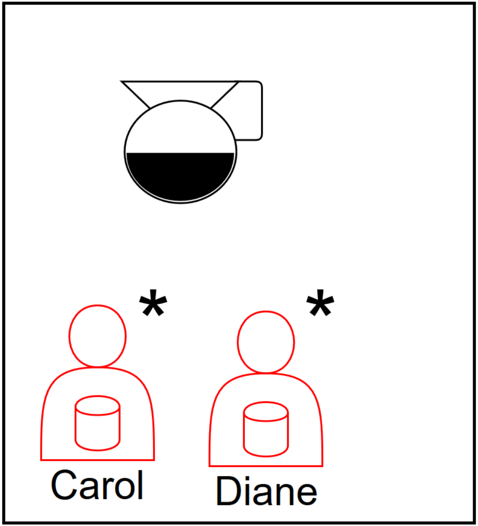
Example

- Object = Coffee
- Application Thread = Person w/ Coffee Cup
- Relocation Thread = Person
- * = Responsible
- ! = Impeded



Example

- Object = Coffee
- Application Thread = Person w/ Coffee Cup
- Relocation Thread = Person
- * = Responsible
- ! = Impeded



Concurrent Relocation and Responsibility

Responsibility: thread required to try relocating an object

- Comes from hazard pointers (coffee cups) impeding copying

Relocation with FPP

- GC threads attempt to relocate objects
- Impeding application threads are made responsible
  - When finished with the object, try to move
- Responsibility passed to impeding threads until relocation succeeds

Testing Environment

Implemented in the Garbage-First (G1) Garbage Collector

- Concurrent tracing and remapping
- Relocation requires stop-the-world pauses

Tested against the default G1 collector

Improvements from solely concurrent relocation

Results

G1 with FPP on average 50% shorter delays than standard G1

- Less impact on application performance

Concurrent relocation without barriers is feasible!

Benchmark	G1	G1 w/ FPP
pmd	40.82 ms	5.02 ms
lusearch	2.73 ms	2.72 ms
tomcat	12.31 ms	5.48 ms
tradebeans	31.73 ms	11.81 ms
fop	37.39 ms	13.22 ms

Table: Average GC Delays

Outline

Conclusions

Moving toward concurrent compaction without barriers

- C4 - heavily relies on barriers
- FPP - barrier-free

Tough to compare them directly

All tests showed that concurrency can improve application performance

- Approach used will depend on intended environment

Thanks for your time!

Questions?

Contact: opdah023@morris.umn.edu

References

G. Tene, B. Iyengar, and M. Wolf.
C4: the continuously concurrent compacting collector.
2011 ACM SIGPLAN International Symposium on Memory Management (ISMM 2011). ACM, New York, NY, USA, 79-88.

E. Österlund and W. Löwe.
Concurrent compaction using a field pinning protocol.
2015 ACM SIGPLAN International Symposium on Memory Management (ISMM 2015). ACM, New York, NY, USA, 56-69.

See the UMM Opdahl Fall ’15 paper for additional references.