The Hotspot Java Virtual Machine

Paul Hohensee
paul.hohensee@sun.com
Senior Staff Engineer, Hotspot JVM, Java SE
Sun Microsystems

Agenda

- VM Overview: Java SE 6, aka 'Mustang'
- Compilation
- Synchronization
- Garbage Collection
- A Performance Future
- Research Directions

VM Overview

- Three major subsystems
- Two compilers, two VMs, same infrastructure
  - -client fast, small footprint
  - -server peak generated code performance
- Java heap management: three garbage collectors
  - Serial (client), Parallel (high throughput), Concurrent (low pause
- Runtime: everything else
  - Interpreter
  - Default class loader, thread management, synchronization, exception handling, ...

VM Overview

- Compiler focus on object-oriented optimizations
  - Deep inlining
  - Class hierarchy analysis
  - Virtual call inlining
  - Dynamic deoptimization
- No handles: direct object references, aka *oops*
  - Exact GC: Root set in globals, thread stacks, machine registers, VM data structures
- Reflection metadata stored in Java heap
  - Class, method objects become garbage when class is unloaded

Object Header

Mark Word
Class Metadata Address
Array Length

- Two words for ordinary objects
- Three for arrays

Mark Word

Bitfields			Tag	State
Hashcode	Age	0	01	Unlocked
Lock record address			00	Light-weight locked
Monitor address			10	Heavy-weight locked
Forwarding address, etc.			11	Marked for GC
Thread ID	Age	1	01	Biased / biasable

VM Overview

- Java stack frames interleaved with native frames
  - > Interpreted, compiled frames
  - > Inlining => compiled frame may include multiple Java methods' data
- Thread types
  - > Java, aka mutator
  - > One VM thread: GC, deoptimization, etc.
  - > Compiler
  - > Watcher, timer
  - > Low memory monitor
  - > Garbage collector, parallel collectors

VM Overview

- Interpreter starts execution, counts method entries and loop back-branches
- Counter overflow triggers compilation
  - Method, asynchronous
  - OSR == On-Stack Replacement == Loop, synchronous
- Compiler produces an *nmethod*
  - Contains generated code, relocation info, ...
- Transfer to compiled code on next method entry or loop iteration

VM Overview

- Compiled code may call not-yet-compiled method
  - Transfer control to interpreter
- Compiled code may be forced back into interpreter
  - Deoptimize and transfer control to interpreter
- Interpreted, compiled, and native code ABIs differ
  - Per-signature adapters
  - I2C, Interpreted-to-Compiled
  - C2I, Compiled-to-Interpreted
  - C2N, Compiled-to-Native (aka Native Wrapper)
  - OSR, Interpreted-to-Compiled-to-Interpreted

Agenda

- VM Overview
- Compilation
- Synchronization
- Garbage Collection
- A Performance Future
- Research Directions

Client Compiler

- Fast compilation, small footprint
- Compile triggered by method entry or loop back-branch counter overflow in interpreter
- Compile method or loop whose counter overflowed
- Inlining decisions based on CHA
  - Static info, essentially method bytecode size
- IR is CFG + per-BB SSA
  - Easier to analyze than trees
- Linear scan register allocator
  - Bin packer, single pass over virtual register lifetimes

Server Compiler

- 'Intermediate' optimizing compiler
  - Compile time still important
- Compile triggered by method entry and loop back-branch counter overflow in interpreter
- Compile scope may be a method whose frame is older than the one whose counter overflowed
  - Prefer larger methods with many inlining opportunities
- Optimization driven by interpreter profiling of all control transfers
  - 30% interpreter performance hit, hurts startup
  - Longer interpreter time => better generated code

Server Compiler

- Inlining decisions based on CHA plus profile data
- IR is SSA-like 'Sea of Nodes'
  - Control flow modeled as data flow
- Global code motion
- Loop optimization (invariant hoisting)
- Loop unrolling
- Instruction scheduling
- Graph-coloring register allocator

Server Compiler

- Contiguous code generated for most frequent path
- Uncommon traps in generated code
  > 'Trap' to interpreter when compile-time assumptions about run-time conditions are violated
  > Usually recompile without the violated assumption
- Generated for, e.g.,
  > Never-executed paths
  > Implicit to explicit null check transformation
  > Unloaded class accesses, e.g., method calls
- Deoptimization forced
  > Replace compiled frame with interpreter frame(s)
  > Safepoint required

Safepointing

- The means by which the JVM brings Java bytecode execution to a stop
- At a safepoint, Java threads cannot modify the Java heap or stack
- Native threads and Java threads executing native methods continue running
- GC, deoptimization, Java thread suspension / stop, certain JVMTI (JVM Tools Interface: JSR 163) operations (e.g., heap dump) require safepoints
- nmethods contain per-safepoint oopmaps
  - Describe oop locations in registers and on stack

Safepoint Polling

- Java threads stop cooperatively, no forced suspension
  - Thread suspension unreliable on Solaris and Linux, e.g., spurious signals
  - Suspend-and-roll-forward problematic: must catch *all* escapes from compiled code
  - Exception handling messy
  - Locating, copying and patching expensive, difficult, buggy
- Global polling page
  - Readable during normal execution
  - VM thread makes unreadable (poisons) when trying to reach a safepoint

Safepoint Polling

- VM thread poisons polling page, then acquires safepoint lock
- Threads poll for safepoint, fault if polling page poisoned
- If at return
  - Pop frame to simulate return to caller: don't want unexpected partial frame
  - Create handle for oop return value
  - Block on safepoint lock
- After VM operation, VM thread releases safepoint lock, Java threads resume

Safepoint Polling

- VM runtime and JNI code poll or check suspension state on transitions back into Java
- Interpreter polls at bytecode boundaries
  - Dispatch vector replaced
- Compiled code polls at method returns and loop back branches, but not at calls
  - x86 and x64: tstl eax, <polling page address>
  - SPARC: ldx [reg=<polling page address>], g0
- Minimal impact in server vm, most loops are 'countable' or have method calls, thus no safepoints
- Largest impact in client vm (but < 0.5%), occurs once per loop iteration

Deoptimization

- Convert compiled frame for an nmethod into one or more interpreter frames (because of inlining)
- Cooperative or uncooperative (i.e., preemptive)
- Uncommon trap is cooperative
  - Generated code calls uncommon trap blob
  - Blob is a handler of sorts, written, naturally, in assembly
- Vanilla deoptimization is uncooperative
  - Class unloading (inlining)
  - Async exceptions (incomplete compiler model)
  - JVMTI: debugging, hotswap

'Lazy' Uncooperative Deoptimization

- Walk thread stacks
- nmethod is patched at return address(es) to redirect execution to deoptimization blob
  - Deoptimization blob is also written, naturally, in assembly
- Safepoint required, since patch(es) might otherwise trash live code paths
- Thread PCs are not updated at the safepoint
  - Deoptimizing a compiled frame is synchronous
  - Compiled method frame(s) remain on thread stack until deopt occurs

Youngest Frame Deoptimization

- Thread calls uncommon trap blob or returns to, jumps to or calls deoptimization blob
- Blob calls into VM runtime, which returns an UnRollBlock describing sizes of frame(s) to remove (I2C or OSR adapter may be present) plus number, sizes and PCs of interpreter frame(s) to be created
- Blob removes its own frame plus frame(s) it's directed to remove

Youngest Frame Deoptimization

- Blob creates skeletal interpreter frame(s) and a new frame for itself
- Blob calls into VM runtime to populate skeletal interpreter frame(s), oldest to youngest: locals, monitors, expression stack, etc.
  - Virtual frame array created by VM runtime
  - Virtual frame contains mappings from compiled frame data locations (including registers) to interpreted frame equivalents
- Blob removes its own frame and resumes in interpreter

Compiler Futures

- Server compiler escape analysis
  - Object explosion
  - Scalar replacement
  - Thread stack object allocation
  - Eliminate synchronization
  - Eliminate object zero'ing
  - Eliminate GC read / write barriers

Compiler Futures

- Tiered compilation
  - Improve startup, time-to-optimized and ultimate performance
  - Single VM contains both client and server compilers
  - Client compiler generates profiled code
    - Feeds server compiler
    - Replaces interpreter profiling
    - Narrows profiling focus
  - Server compiler guided by more accurate profile data
  - Can afford longer compile time, heavier-weight optimizations
  - Aggressive uncommon trap use: trap to client-compiled code

Agenda

- VM Overview
- Compilation
- Synchronization
- Garbage Collection
- A Performance Future
- Research Directions

Java Locking

- Every Java object is a potential monitor
  - **synchronized** keyword
- All modern JVMs incorporate light-weight locking
  - Avoid associating an OS mutex / condition variable (heavy-weight lock) with each object
  - While uncontended, use atomics to enter / exit monitor
  - If contended, fall back to heavy-weight OS lock
- Effective because most locking is uncontended

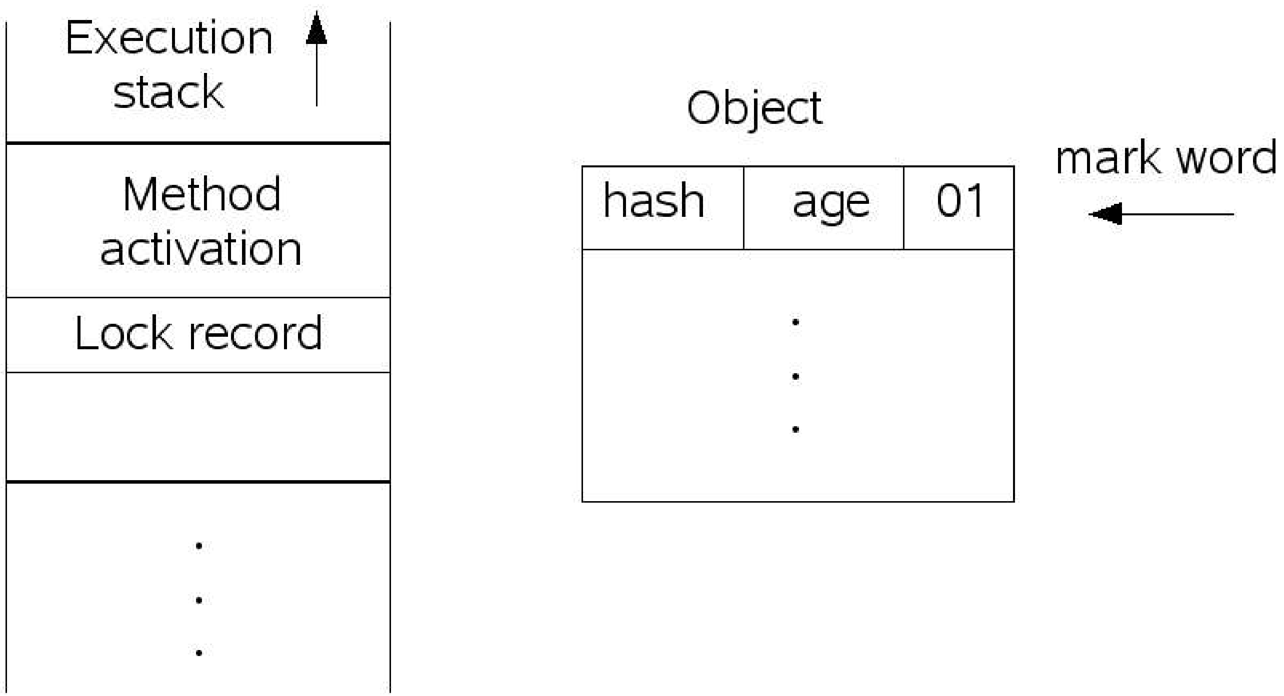
Light-weight Locking

- First word of each object is the *mark word*
- Used for synchronization and GC
  - Also caches hashcode, if previously computed
- Recall that low two bits of mark word contain synchronization state
  - 01 => unlocked
  - 00 => light-weight locked
  - 10 => heavy-weight locked
  - 11 => marked for GC

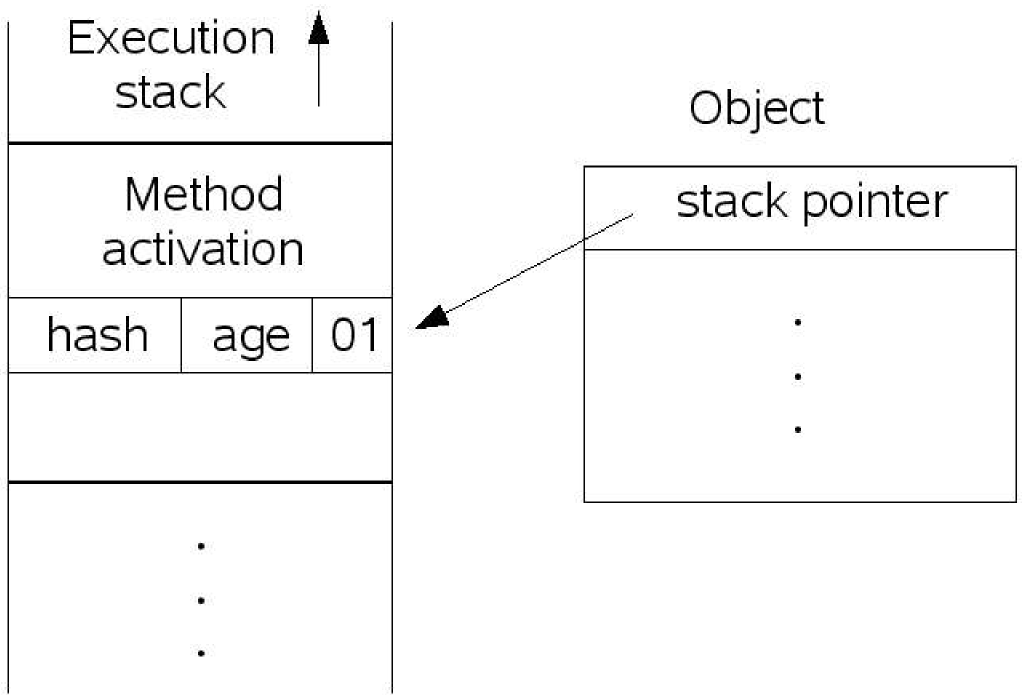
Light-weight Locking

- When object is locked, mark word copied into *lock record* in frame on thread stack
  - Aka, *displaced mark*
- Use atomic compare-and-swap (CAS) instruction to attempt to make mark word point to lock record
- If CAS succeeds, thread owns lock
- If CAS fails, contention: lock *inflated* (made heavy-weight)
- Lock records track objects locked by currently-executing methods
  - Walk thread stack to find thread's locked objects

Light-weight Locking: Before



Light-weight Locking: After



Light-weight Locking

- Unlock uses CAS to put displaced mark back into object mark word
- If CAS fails, contention occurred: lock inflated
  - Notify waiting threads that monitor has exited
- Recursive lock stores zero in lock record
  - Inflation replaces zero with displaced mark
  - Unlock checks for zero, no CAS if present

Observations

- Atomic instructions can be expensive on multiprocessors
- Most locking not only uncontended, but performed repeatedly by the same thread
  - cf Kawachiya et al, “Lock Reservation”, OOPSLA 2002
- Make it cheap for a single thread to reacquire a lock
  - Note: *reacquire*, not recurse
- Tradeoff: will be more expensive for another thread to acquire the lock

Biased Locking

- First lock of an object *biases* it toward the acquiring thread
  - > Add a bit to the mark word tag
  - > 001 => unlocked
  - > 101 => biased or biasable (thread ID == 0 == unlocked)
  - > Bias obtained via CAS
- Subsequent locks / unlocks by owner very cheap
  - > Test-and-branch, no atomics
- Bias *revoked* if another thread locks biased object
  - > Expensive for single objects

Bias Revocation

- Revert to light-weight locking
- Stop bias owner thread
- Walk bias owner thread stack to find lock records, if any
  - > Write displaced mark into oldest lock record, zero into younger ones
- Update object's mark word
  - > If locked, point at oldest lock record
  - > If unlocked, fill in unlocked value
- Expensive
  - > VM cannot stop a single thread, must safepoint

Bulk Rebiasing and Revocation

- Detect if many revocations occurring for a given data type
- Try invalidating all biases for objects of that type
  - Allows them to be rebiased to a new thread
  - Amortizes cost of individual revocations
  - Multiple such operations possible
- If individual revocations persist, disable biased locking for that type
  - Revert to light-weight locking for all currently locked objects of that type

Agenda

- VM Overview
- Compilation
- Synchronization
- Garbage Collection
- A Performance Future
- Research Directions

Garbage Collection

- Generational, two generations
  - Objects allocated in young generation (aka 'nursery')
  - Promoted to old (aka 'tenured') generation if they live long enough
- Young generation copying collectors (aka 'scavengers')
  - Each mutator has its own young generation allocation buffer (TLAB == Thread-Local-Allocation-Buffer)
  - Vast majority of collections are young gen only
  - Most objects die quickly, so young gen collections are fast
- Three collectors

Serial Collector

- Default client and restricted-platform server collector
- Good throughput on smaller heaps (~1gb)
- Young generation collection
  - Stop-the-world serial scavenger
  - Usually short pause time
- Full collection (old + young)
  - Stop-the-world serial Mark-Sweep-Compact
  - Potentially long pause time

Parallel Collector

- Server class machine collector
  - 2 or more cores + 2gb or larger physical memory
- High throughput, large (up to 100's of gb) heaps
- Young generation collection
  - Stop-the-world parallel scavenger
  - Short pause time
- Full collection (old + young)
  - Stop-the-world parallel Mark-Compact
  - Usually short pause time
  - SPECjbb2005, high warehouse counts

Concurrent Collector

- Low pause time collector
  - Soft real time, more or less
- Pause times on the order of 100 to 200 ms
- Young generation collection
  - Stop-the-world parallel scavenger
  - High throughput, large heaps
- Old generation collection
  - CMS (Concurrent Mark-Sweep)
  - Good throughput, medium (up to 4gb) heaps

Parallel Scavenge

- Stop-the-world
- Divide root set among GC threads
  - > Default thread count = # hardware threads, up to 8
  - > Above 8, 5/8 * # hardware threads
  - > -XX:ParallelGCThreads=<n>
- Copy live objects into survivor space PLABs (Parallel Local Allocation Buffers)
- Promote objects into old generation PLABs
- Ergonomics
  - > Adaptive size policy
  - > Work stealing to balance GC thread workloads

Concurrent Mark Sweep

- GC thread(s) run concurrently with mutators
- Allocate from / sweep to free lists
- Pause (very short), mark objects reachable from roots
- Resume mutators, mark concurrently
  - And in parallel for Java SE 6
- Remark objects dirtied during mark
  - Iterative concurrent (and parallel in Java SE 6) remark
  - Pause (short) for final parallel remark
- Resume mutators, sweep concurrently

Concurrent Mark Sweep

- Fragmentation can be a problem
  - > Free list coalescing and splitting
  - > Adaptive, based on time-aggregated demand for free blocks
- Worst case, compaction required
  - > Punt to serial MSC => potentially long pause time
  - > Future, punt to parallel MC => shorter pause time
- Incremental mode
  - > For low hardware thread count (e.g., one)
  - > GC thread runs at intervals rather than all the time

Parallel Mark Compact

- Stop-the-world
- Heap divided into fixed-size chunks
  > 2kb now, will likely increase or be subject to ergonomics
- Chunk is unit of live data summarization
- Parallel mark
  > Record live data addresses in external bitmap
  > Find per chunk live data size
  > Find dense chunks, i.e., ones that are (almost) full of live objects

Parallel Mark Compact

- Summarize over chunks
  - Serial, unfortunately, but quick
  - Will compact to the left (towards low addresses)
  - Find 'dense prefix' == part of heap that will not be compacted
    - Subset of dense chunk set
  - Determine destination chunks
- Parallel compact
  - GC threads claim destination chunks and copy live data into them
  - Heap ends fully compacted but for holes in dense prefix

Garbage Collection Futures

- Currently, each collector has its own ergonomics, e.g.,
  - When to promote from young to old generation
  - When to expand or contract committed memory
  - Survivor space sizing
  - Relative old and young generation sizing
    - Parallel collector can adjust dynamically
- Many knobs to turn!
- Create GC ergonomics framework
  - Driven by pause time and throughput goals
  - Switch and configure collectors dynamically
- Or, ...

Garbage First

- Concurrent and parallel mark-scavenge
- Low latency, high throughput, builtin ergonomics
- High probability of compliance with a soft real time goal
  - GC shall consume no more than X ms in any Y ms interval
  - Or, (Y – X) / Y == Minimum Mutator Utilization goal
- 'Regionalized' heap
  - Maintain estimated cost to collect each region
  - Prefer to collect regions containing lots of garbage
  - Mutators allocate in TLABs, which in turn are allocated from 'mutator allocation' regions

Garbage First

- Concurrent and parallel mark
- Per region, maintain
  - Live data size
  - Set of external (remembered) references into a region, thus,
  - Arbitrary sets of regions can be collected
- Parallel collection pauses as MMU spec allows
  - Stop-the-world for up to X ms within Y ms intervals
  - Determine set of desirable regions collectible in X ms
  - Evacuate live objects into per-GC-thread PLABs, compacting in the process
  - Work stealing to balance GC thread workloads

Garbage First

- Single generation, but,
- Remembered set maintenance => expensive write barrier (~12 RISC instructions each)
  - > Partially mitigated by compiler analysis
  - > 25 – 60% can be eliminated on a dynamic basis
- So, define 'young generation' as the set of uncollected mutator allocation regions
  - > Except for a prefix containing the youngest objects, young gen always evacuated during a collection, thus,
  - > No need to remember address of a write into young region

Garbage Collection Literature

- R. E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Ltd, 1996. With a chapter on Distributed Garbage Collection by R. Lins.
- P. R. Wilson. *Uniprocessor Garbage Collection Techniques.* In *Proceedings of the First International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1-42, St Malo, France, September 1992. Springer-Verlag.

Low Latency GC Papers

- D. L. Detlefs, C. H. Flood, S. Heller, and T. Printezis. *Garbage-First Garbage Collection*. In A. Diwan, editor, *Proceedings of the 2004 International Symposium on Memory Management (ISMM 2004)*, pages 37-48, Vancouver, Canada, October 2004. ACM Press.
- T. Printezis and D. L. Detlefs. A Generational Mostly-Concurrent Garbage Collector. In A. L. Hosking, editor, P*roceedings of the 2000 International Symposium on Memory Management (ISMM 2000)*, pages 134-154, Minneapolis, MN, USA, October 2000. ACM Press.

Online Articles

- Description of the GCs in HotSpot
  - > http://www.devx.com/Java/Article/21977
- Finalization and Memory Retention Issues
  - > http://www.devx.com/Java/Article/30192

Sun GC Blogs

- http://blogs.sun.com/jonthecollector
- http://blogs.sun.com/dagastine
- http://blogs.sun.com/tony

Mailing List

- hotspotgc-feedback@sun.com

Agenda

- VM Overview: Java SE 6, aka 'Mustang'
- Compilation
- Synchronization
- Garbage Collection
- A Performance Future
- Research Directions

A Performance Future

- Ergonomics writ large, dynamic adaptation
  - > Goal: no more switch mining
- Internal ergo: VM adapts to app demands
  - > GC ergonomics framework incorporating multiple collectors
- External ergo: VM adapts to system demands
  - > Decrease / increase heap size as memory demand increases / decreases
- Integrate VM ergonomics, performance analysis, monitoring and management
  - > Feedback between adaptive VM and app run histories
  - > Minimize time to optimized performance

Agenda

- VM Overview: Java SE 6, aka 'Mustang'
- Compilation
- Synchronization
- Garbage Collection
- A Performance Future
- Research Directions

Research Directions

- Soft Realtime Java Enterprise Stack
  - Requires pauseless GC
  - Garbage First still has STW pauses
  - Minimize GC-related mutator execution time
  - Cliff Click, Gil Tene and Michael Wolfe. *The Pauseless GC Algorithmn*. In *Proceedings of the 1st ACM/Usenix International Conference on Virtual Execution Environments (VEE 2005)*, pages 45-56, Chicago, Illinois, June 2005. ACM Press.
  - Is there an efficient way to do this without hardware assist, or with less hardware assist?

Research Directions

- Hard Real-Time GC
  - D. L. Detlefs. *A Hard Look At Hard Real-Time Garbage Collection*. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 23-32, Vienna, Austria, May 2004. IEEE Press. *Invited Paper*.
  - D.F. Bacon, P. Cheng and V.T. Rajan. *A Real-Time Garbage Collector with Low Overhead and Consistent Utilization*. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2003)*, pages 285-298, New Orleans, Louisiana, January 2003. ACM Press.
  - Requires that the user specify a maximum allocation rate and maximum live data size, which is problematic

Research Directions

- Identify and avoid marking long-lived, slowly / not-at-all changing portions of the Java heap
- NUMA-Aware Java
  - > Segmented (non-contiguous virtual address space) heaps?
  - > ???
- Cluster-Aware Java
  - > E.g., JSR 121 (Isolates, done), JSR 284 (Resource Management, in progress)
  - > Unified cluster-level JVM monitoring and management
  - > Autonomic computing, agent-based M&M
  - > http://www.research.ibm.com/autonomic/

Research Directions

- Inline native method machine code into compiled methods
  - L. Stepanian, A.D. Brown, A. Kielstra, G. Koblents and K. Stoodley. *Inlining Java Native Calls at Runtime*. In *Proceedings of the 1st ACM/Usenix International Conference on Virtual Execution Environments (VEE 2005)*, pages 121-131, Chicago, Illinois, June 2005. ACM Press.
- Debugging
  - Currently, deoptimize methods with breakpoints and such
  - Can we do better? Reverse execution? Deterministic replay of parallel execution?
  - Feed into M&M?

Java SE Related Links

- Hotspot and JDK source and binary bundles
  - > http://www.java.net/download/jdk6/
  - > Java Research License (JRL)
- GC Tuning Guides (J2SE 5.0 / 1.4.2)
  - > http://java.sun.com/docs/hotspot/gc5.0/gc_tuning_5.html
  - > http://java.sun.com/docs/hotspot/gc1.4.2/index.html
- Explanation of GC Ergonomics
  - > http://java.sun.com/docs/hotspot/gc5.0/ergo5.html
- J2SE 5.0 Performance White Paper
  - > http://java.sun.com/performance/reference/whitepapers/5.0_performance.html

Questions?

Paul Hohensee
paul.hohensee@sun.com

Let's Talk Trash!

Tony Printezis
tony.printezis@sun.com
http://blogs.sun.com/tony
Staff Engineer, Garbage Collection Group, Java SE
Sun Microsystems

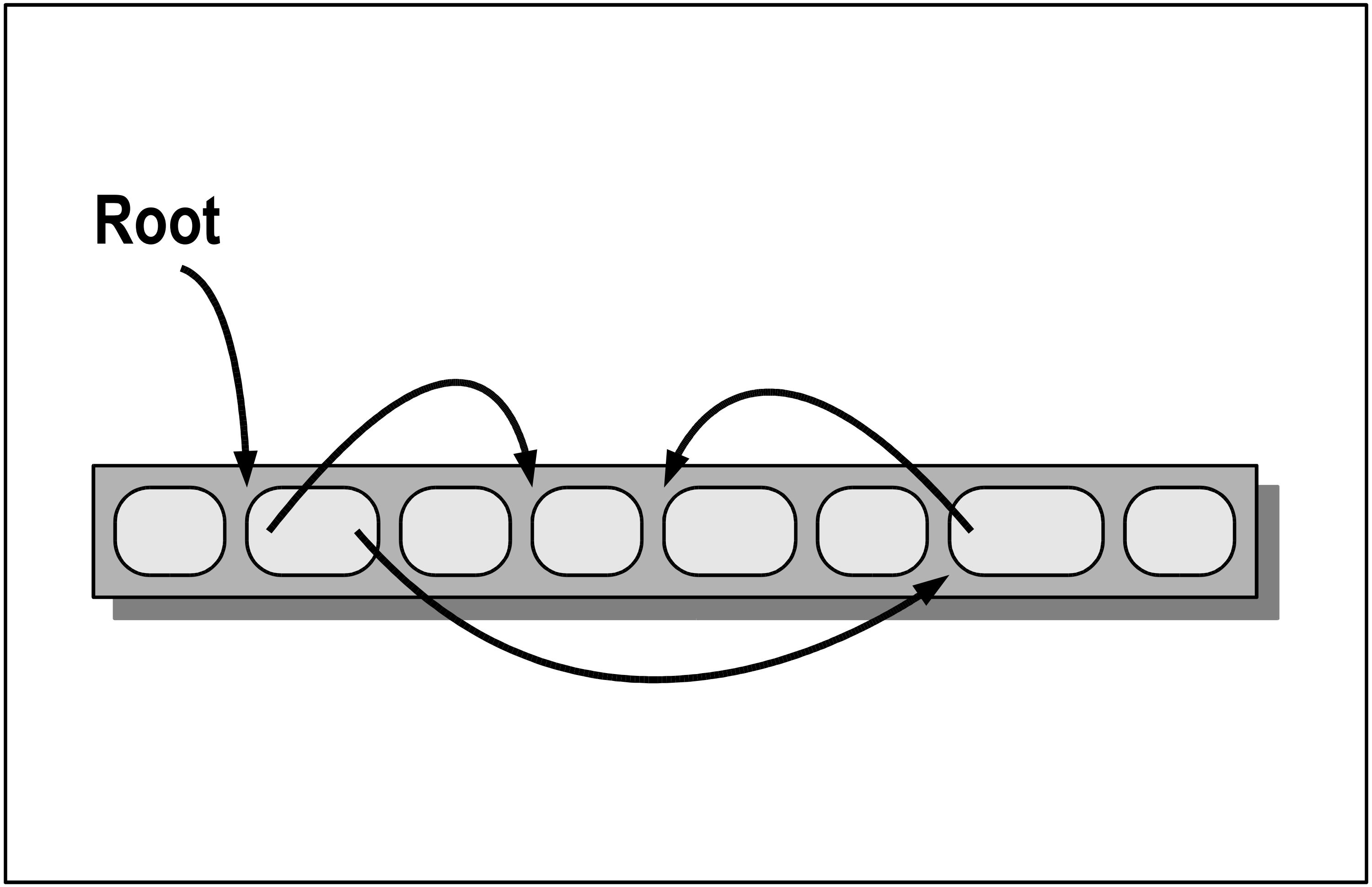
Outline

- **GC Background**
- GCs in the Java HotSpot™ Virtual Machine
- GC Ergonomics
- Latest / Future Directions
- Thoughts on Predictable Garbage Collection

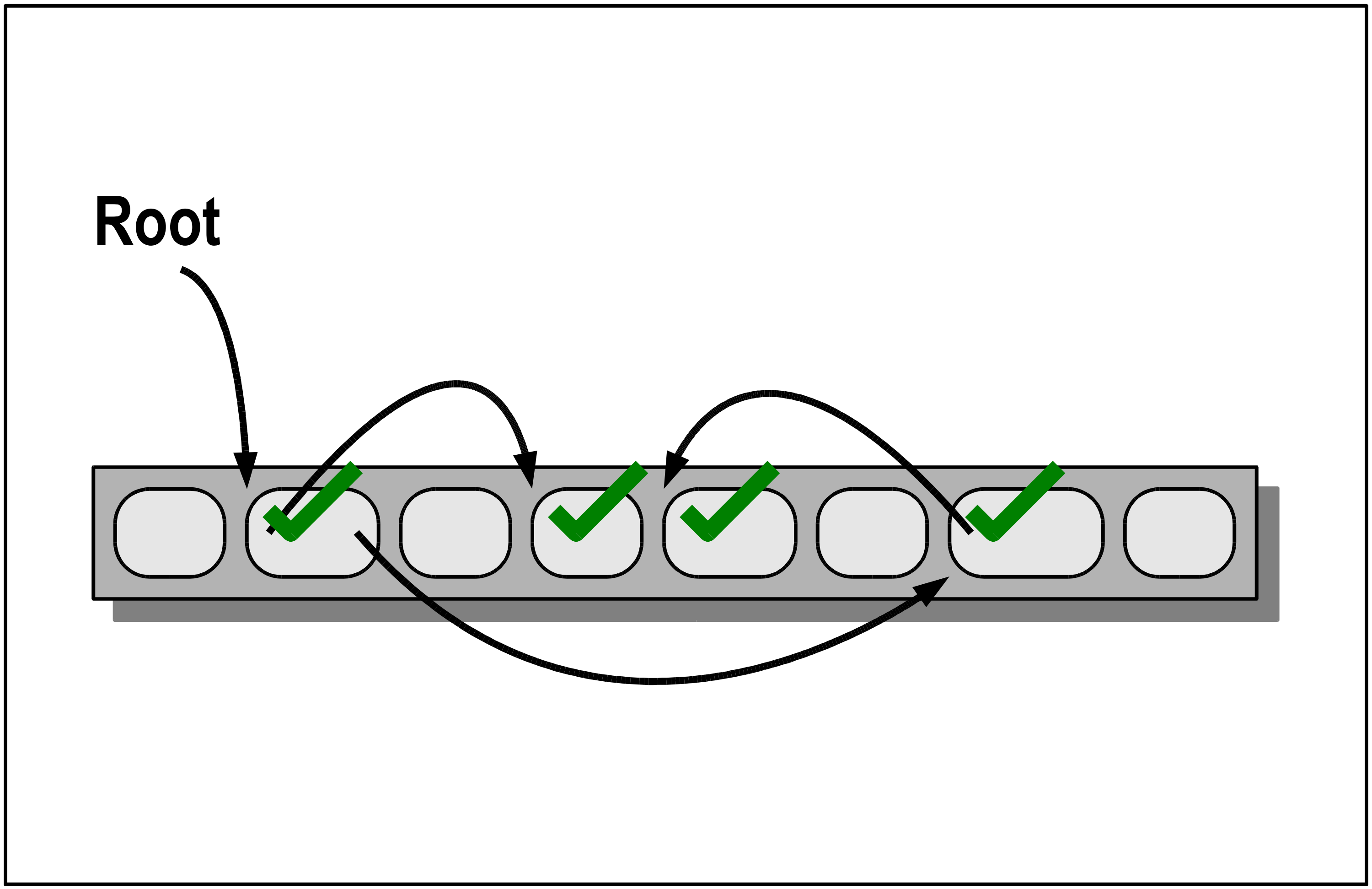
Garbage Collection Techniques

- **Indirect Techniques**
  - > *“Identify Live First, Deduce Garbage”*
  - > e.g., Mark-Sweep, Mark-Compact, Copying
- **Direct Techniques**
  - > *“Identify Garbage Directly”*
  - > e.g., Reference Counting

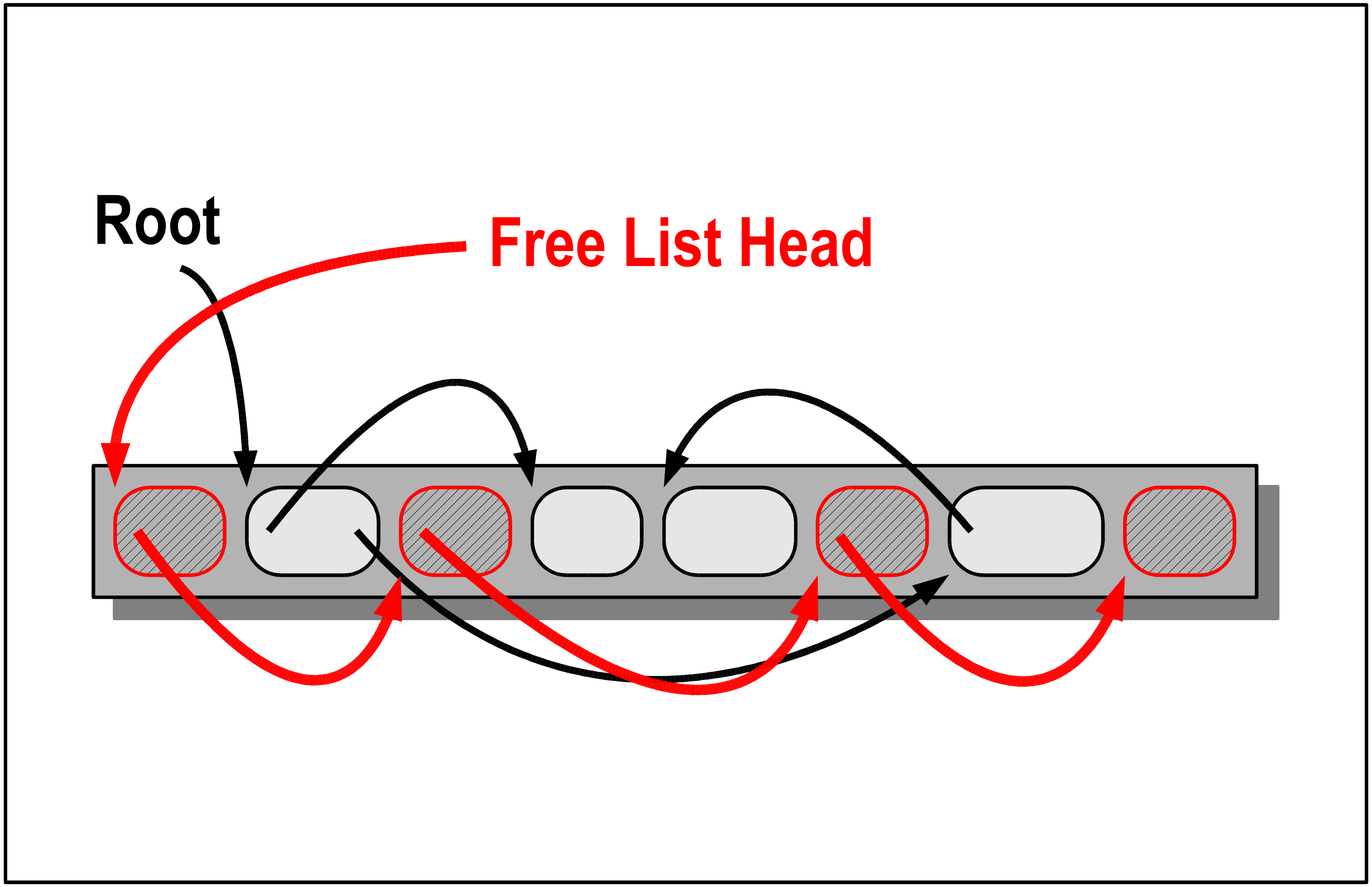
Mark-Sweep



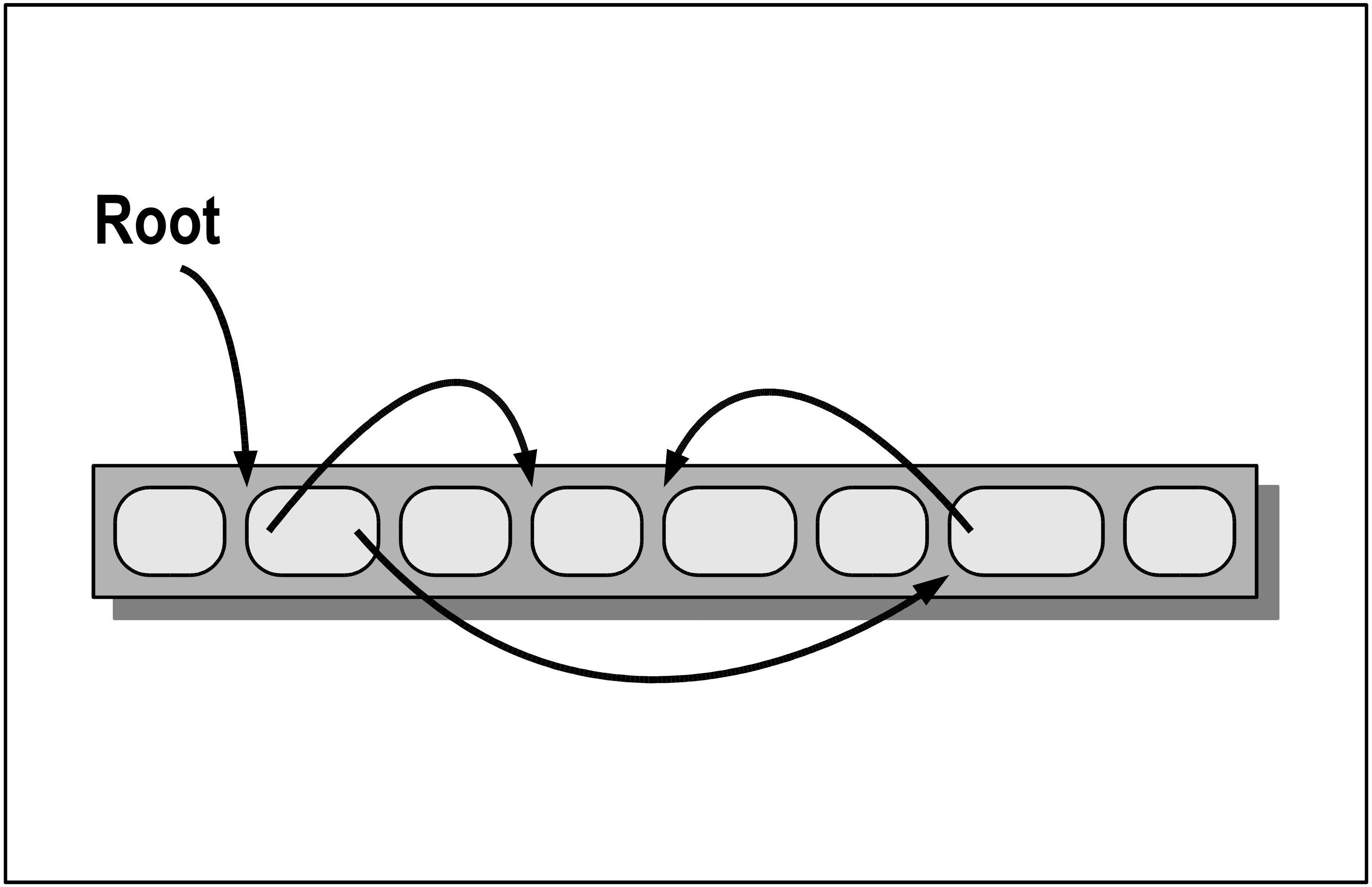
Mark-Sweep – Marking



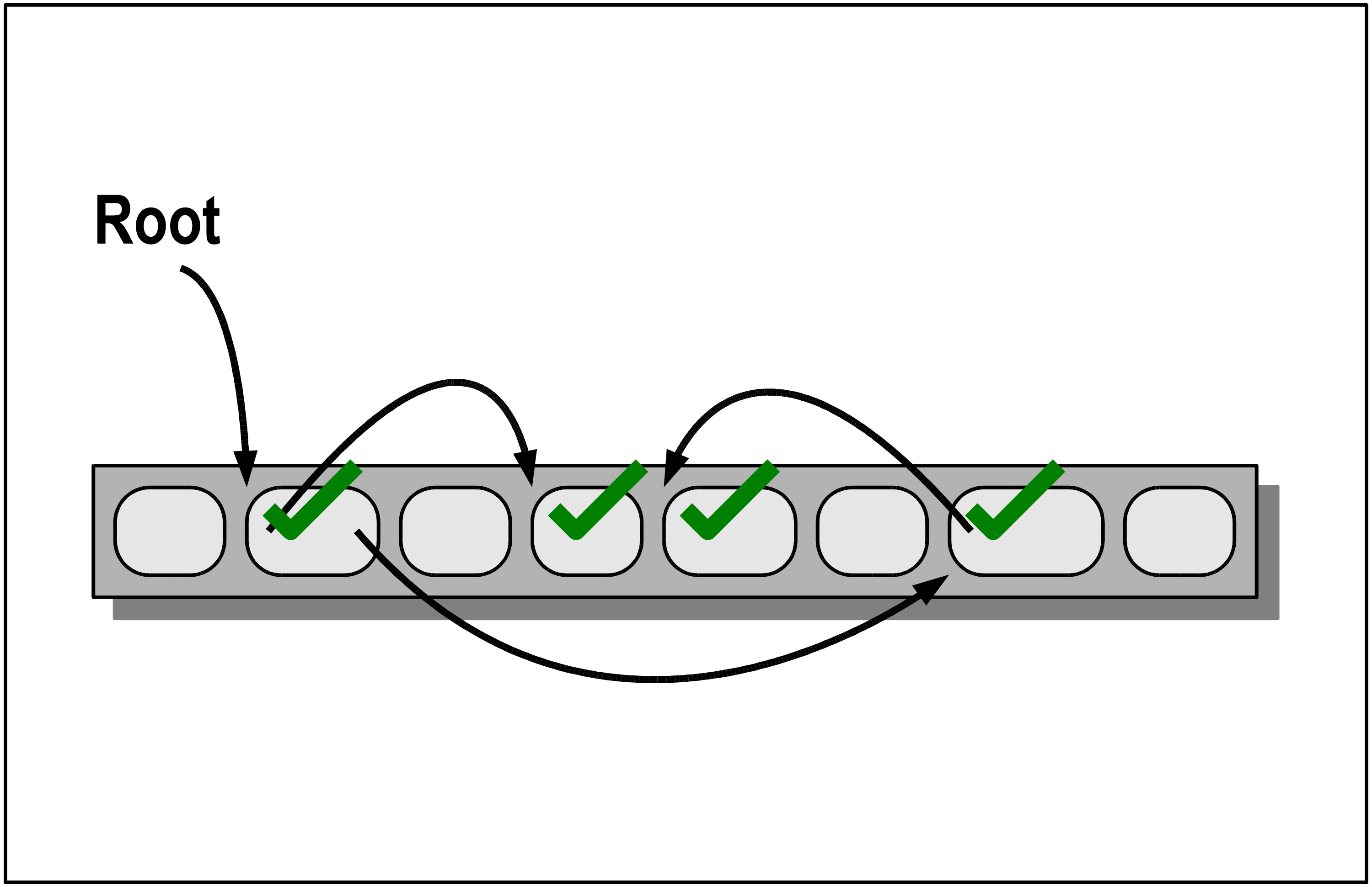
Mark-Sweep – Sweeping



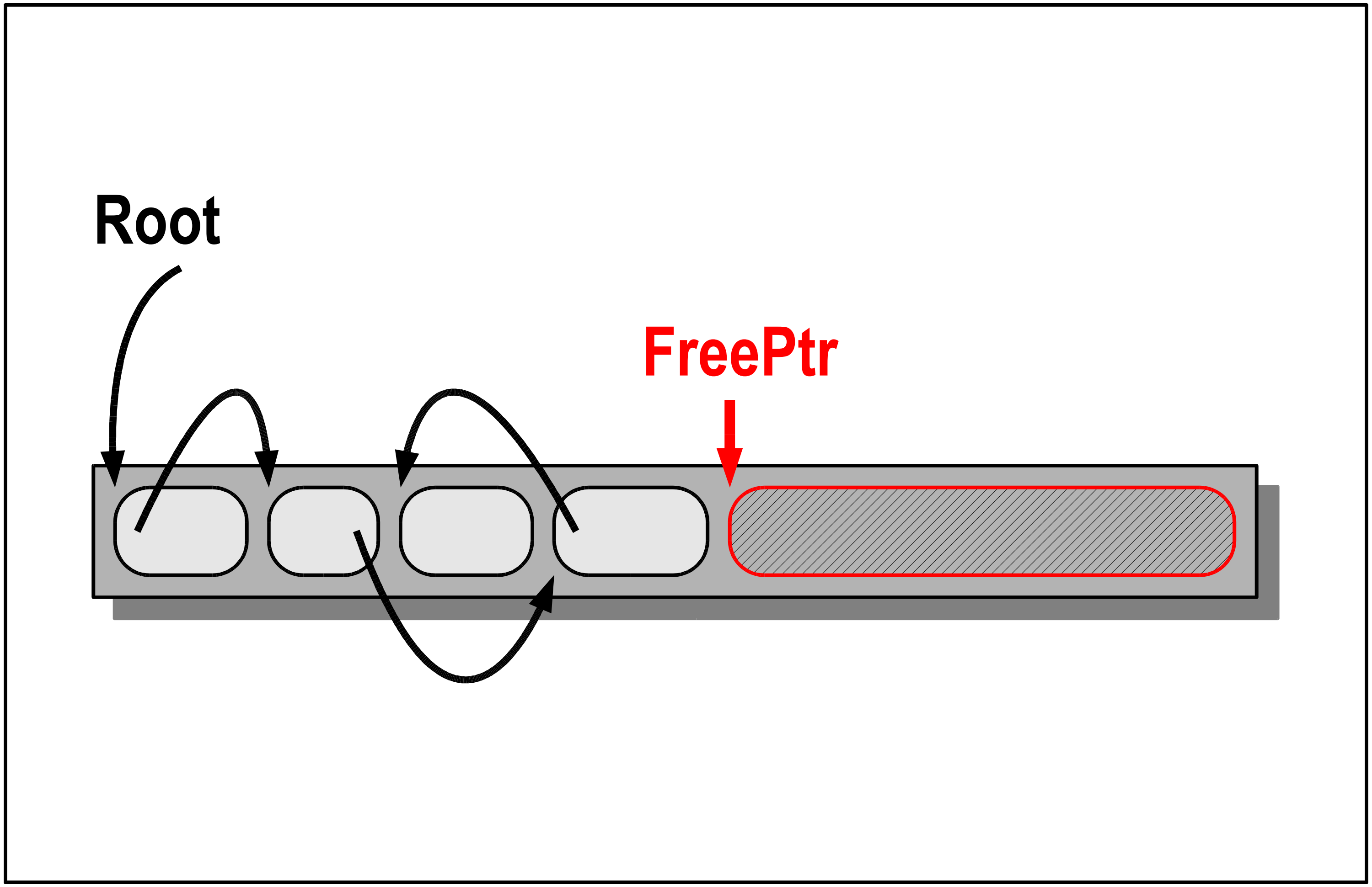
Mark-Compact



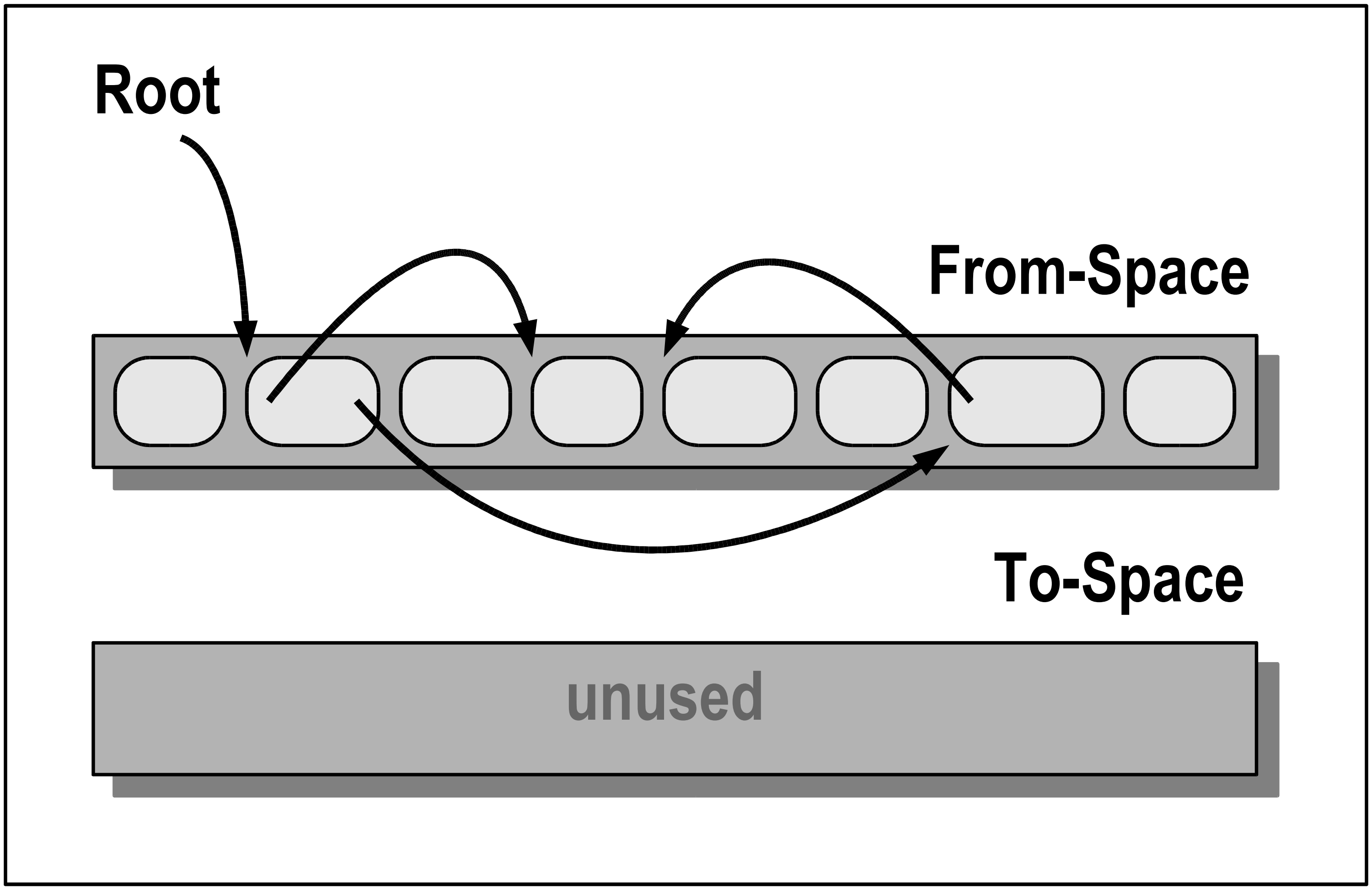
Mark-Compact – Marking



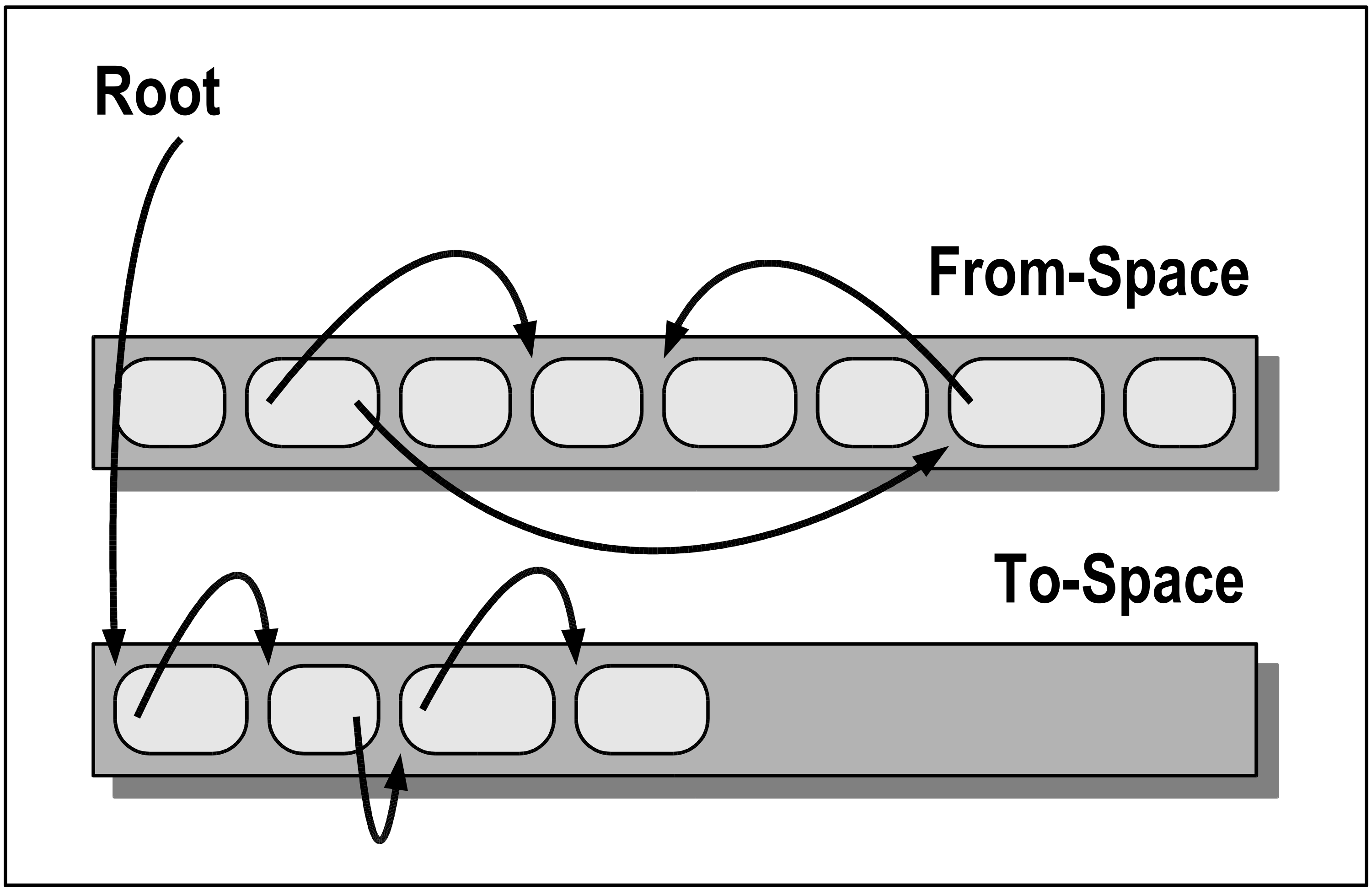
Mark-Compact – Compacting



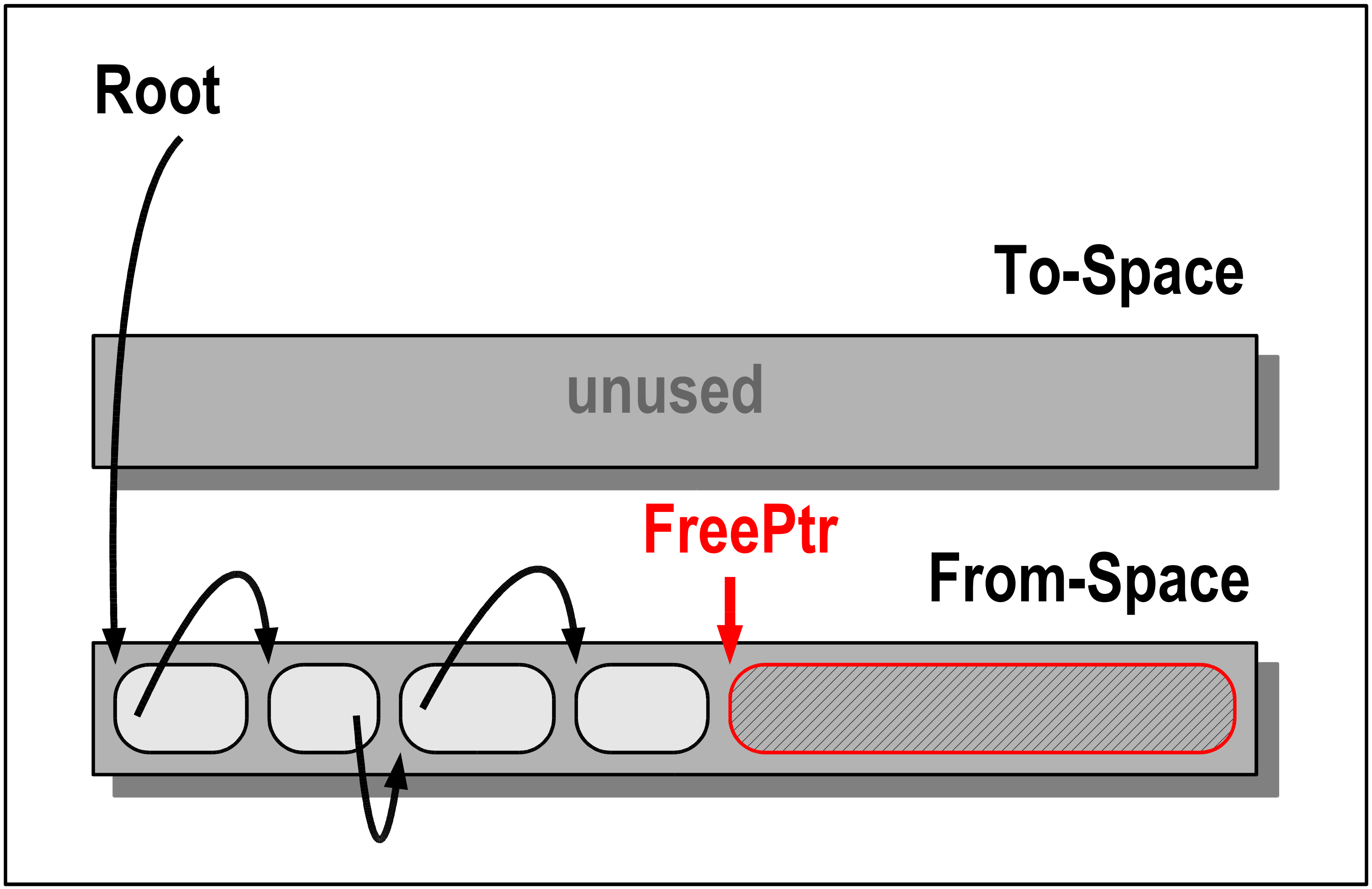
Copying



Copying – Evacuation



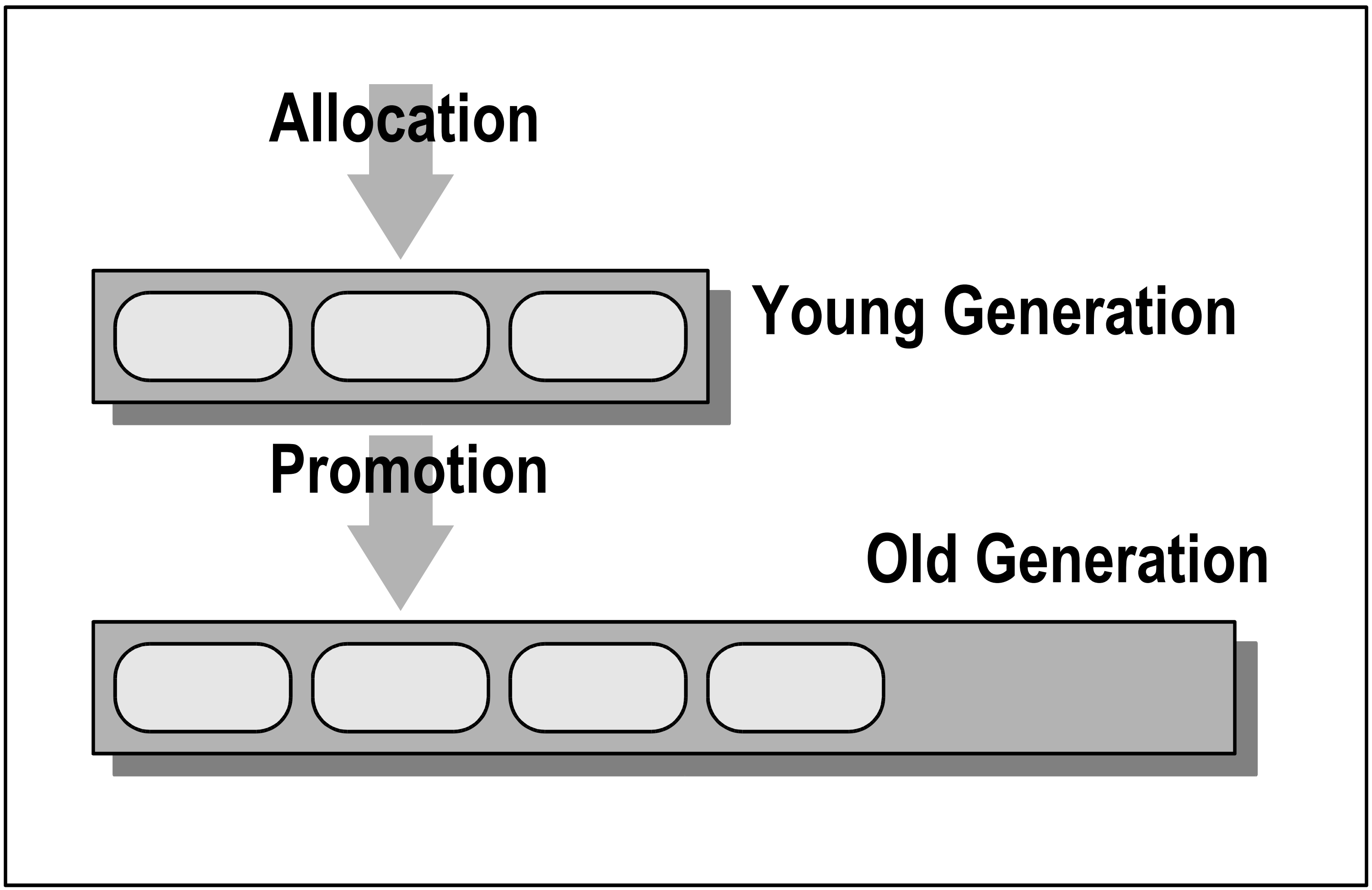
Copying – Flip



To Compact, Or Not To Compact?

- **Compaction**
  - e.g., in Mark-Compact, Copying
  - Battles fragmentation
    - Important for long running applications
  - Fast (linear) allocation
    - Bump-a-pointer
- No Free Lunch!
  - Not cheap
  - Incremental compaction is not trivial

Generational Garbage Collection



Generational GC Benefits

- Very efficient collection of dead young objects
  - Throughput / Short Pauses
- Reduced allocation rate in the old generation
  - Young generation acts as a “filter”
- Use best GC algorithm for each generation
  - Copying for young generation
    - Only visits live objects; good when survival rate is low
  - Mark-Sweep / Mark-Compact for old generation
    - More space efficient

Generational GC Benefits

- For most Java applications, Generational GC is hard to beat
  - > Many have tried,
  - > including Sun.
  - > Everyone went back to Generational GC.

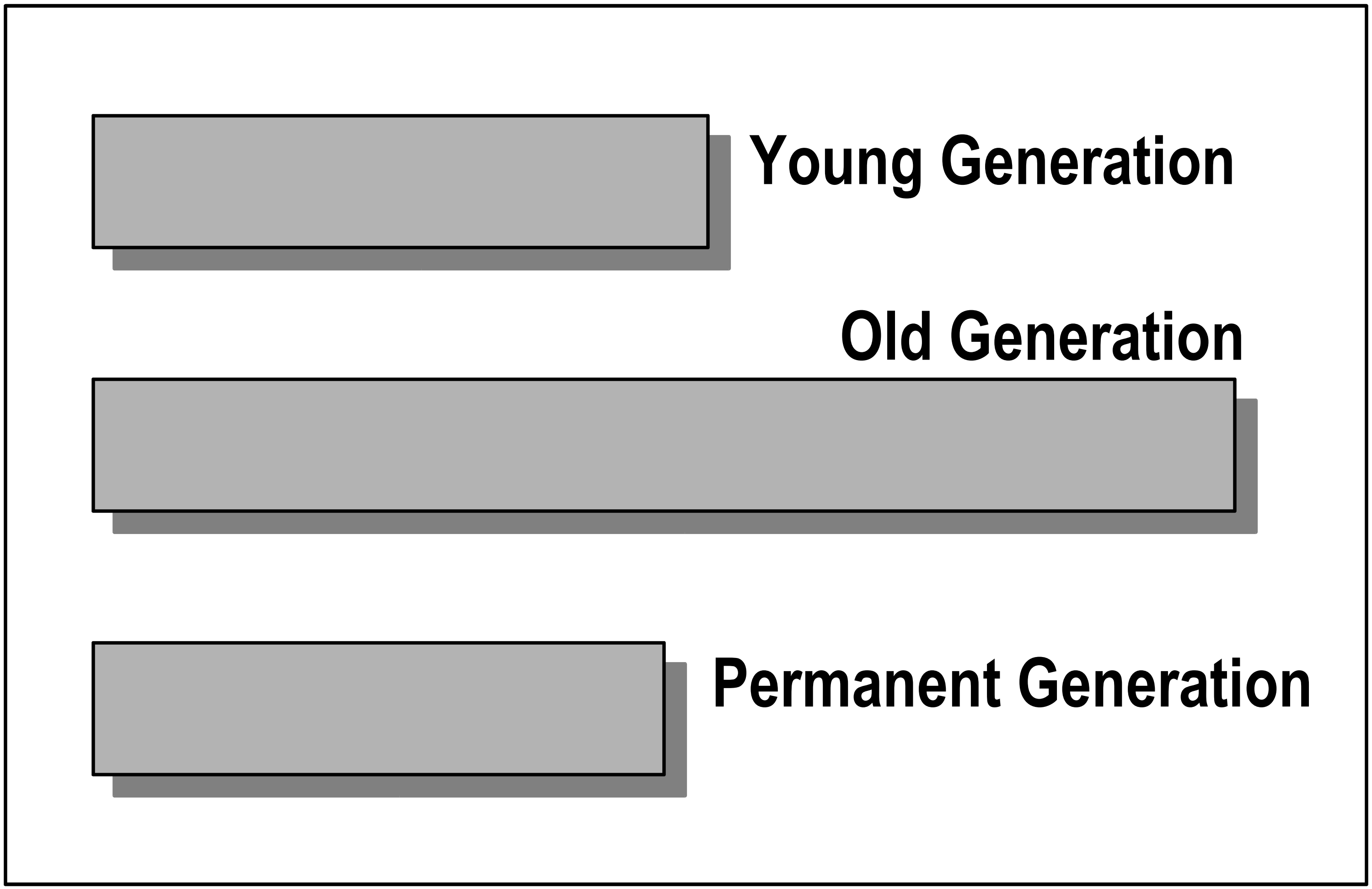
Outline

- GC Background
- **GCs in the Java HotSpot™ Virtual Machine**
- GC Ergonomics
- Latest / Future Directions
- Thoughts on Predictable Garbage Collection

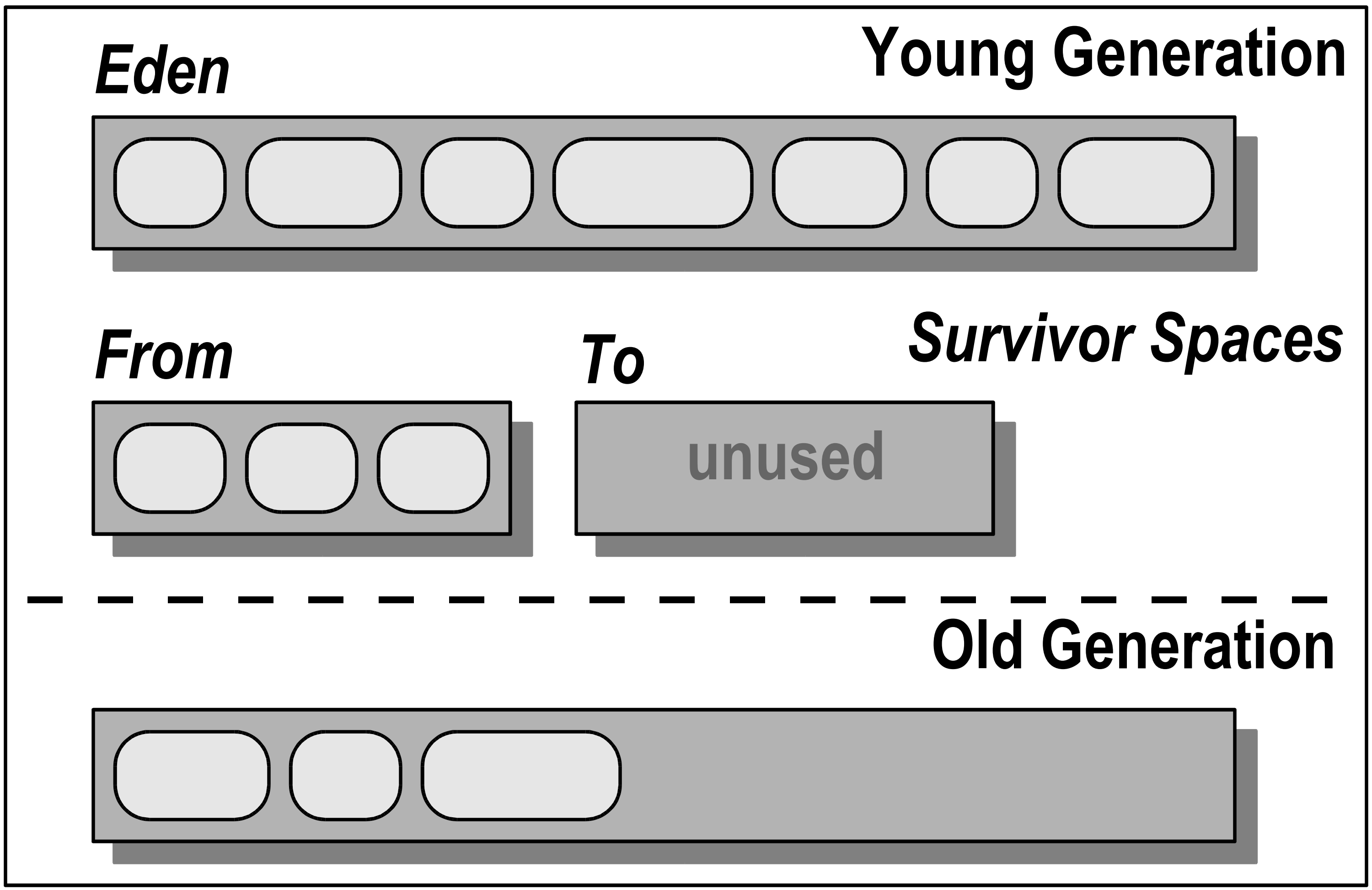
GCs In The HotSpot JVM

- Three GCs
  - **Serial GC**
  - **Parallel Throughput GC**
  - **Concurrent Mark-Sweep**
- All Generational

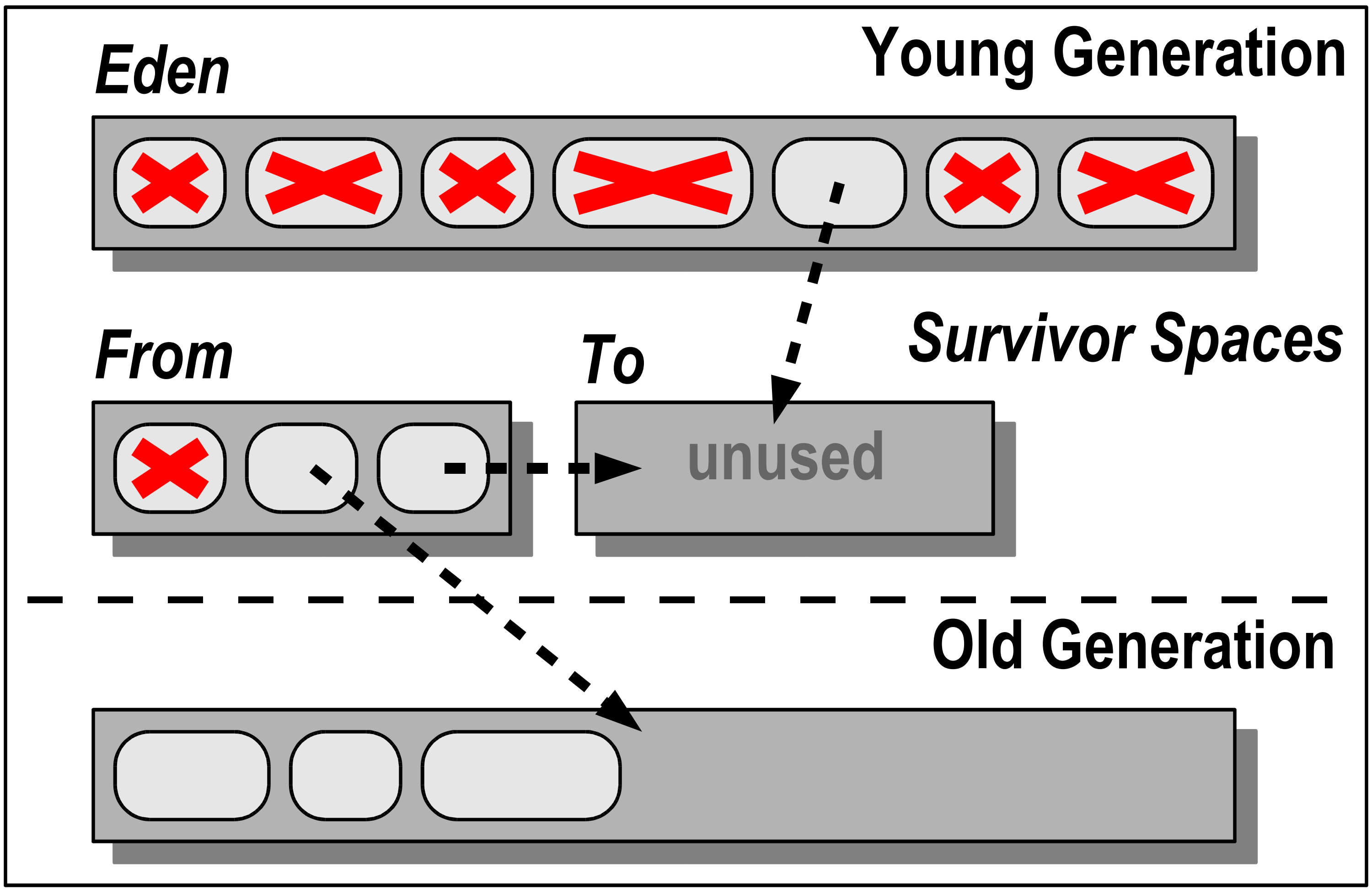
The Heap In The HotSpot JVM



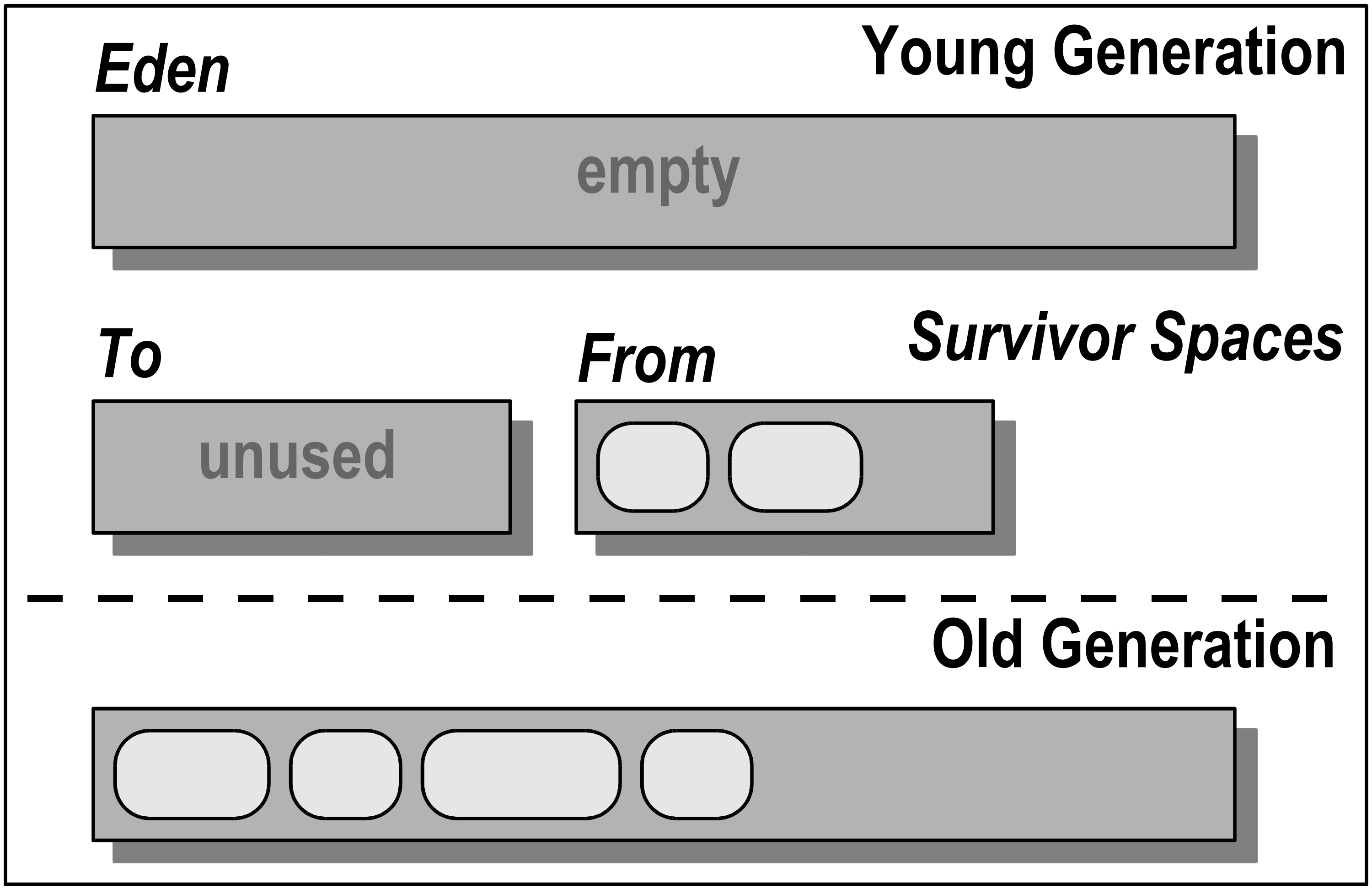
HotSpot Young Generation



Before Young Collection



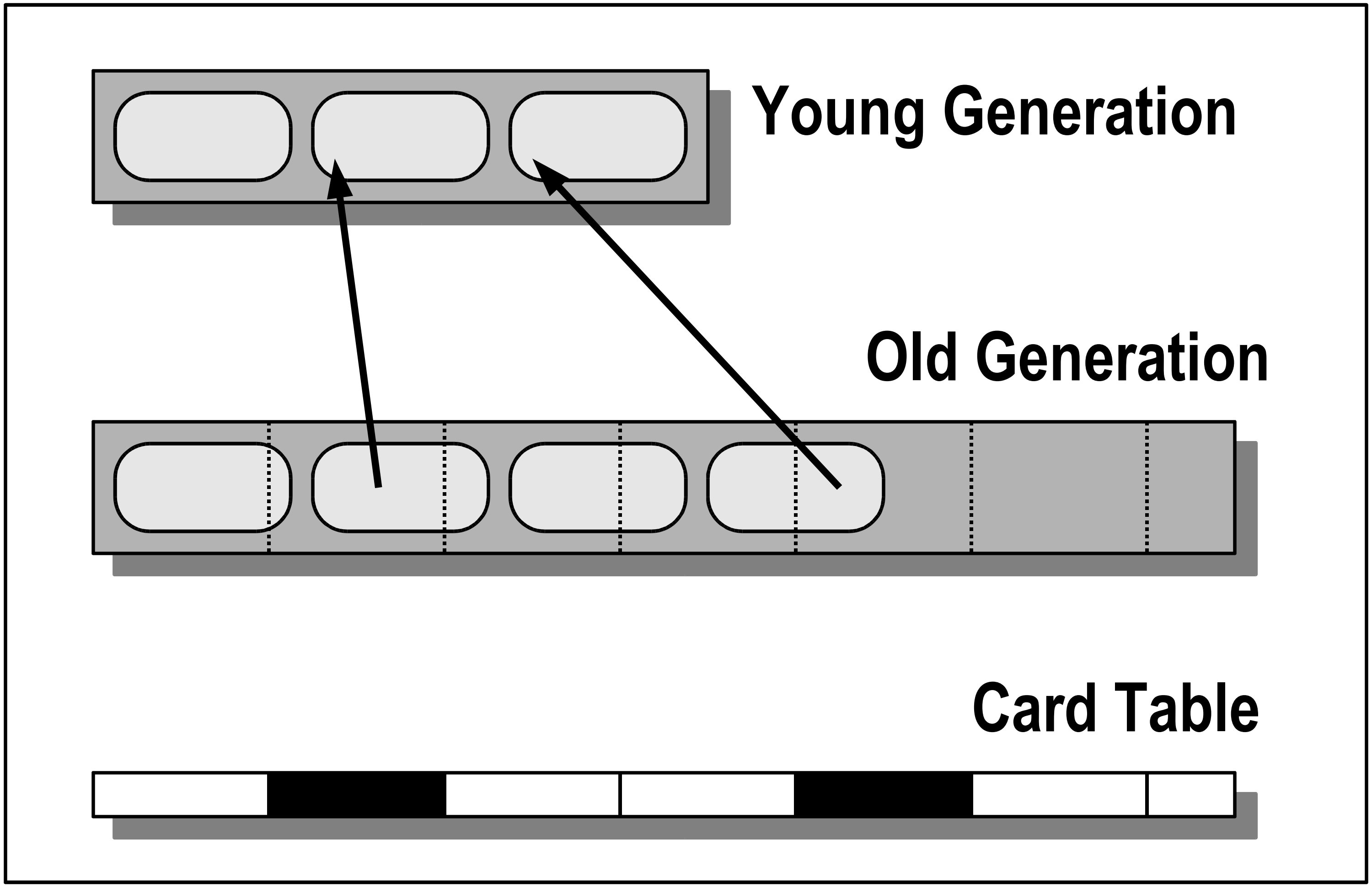
After Young Collection



TLABs

- *Thread-Local Allocation Buffers*
- Each application thread gets a TLAB to allocate into
  - > TLABs allocated in the Eden
  - > Bump-a-pointer allocation; fast
  - > No synchronization (thread “owns” TLAB for allocation)
- Only synchronization when getting a new TLAB
  - > Bump-a-pointer to allocate TLAB too; also fast
- Allocation code inlined
  - > Fast allocation path is around ten native instructions

Card Table And Write Barrier



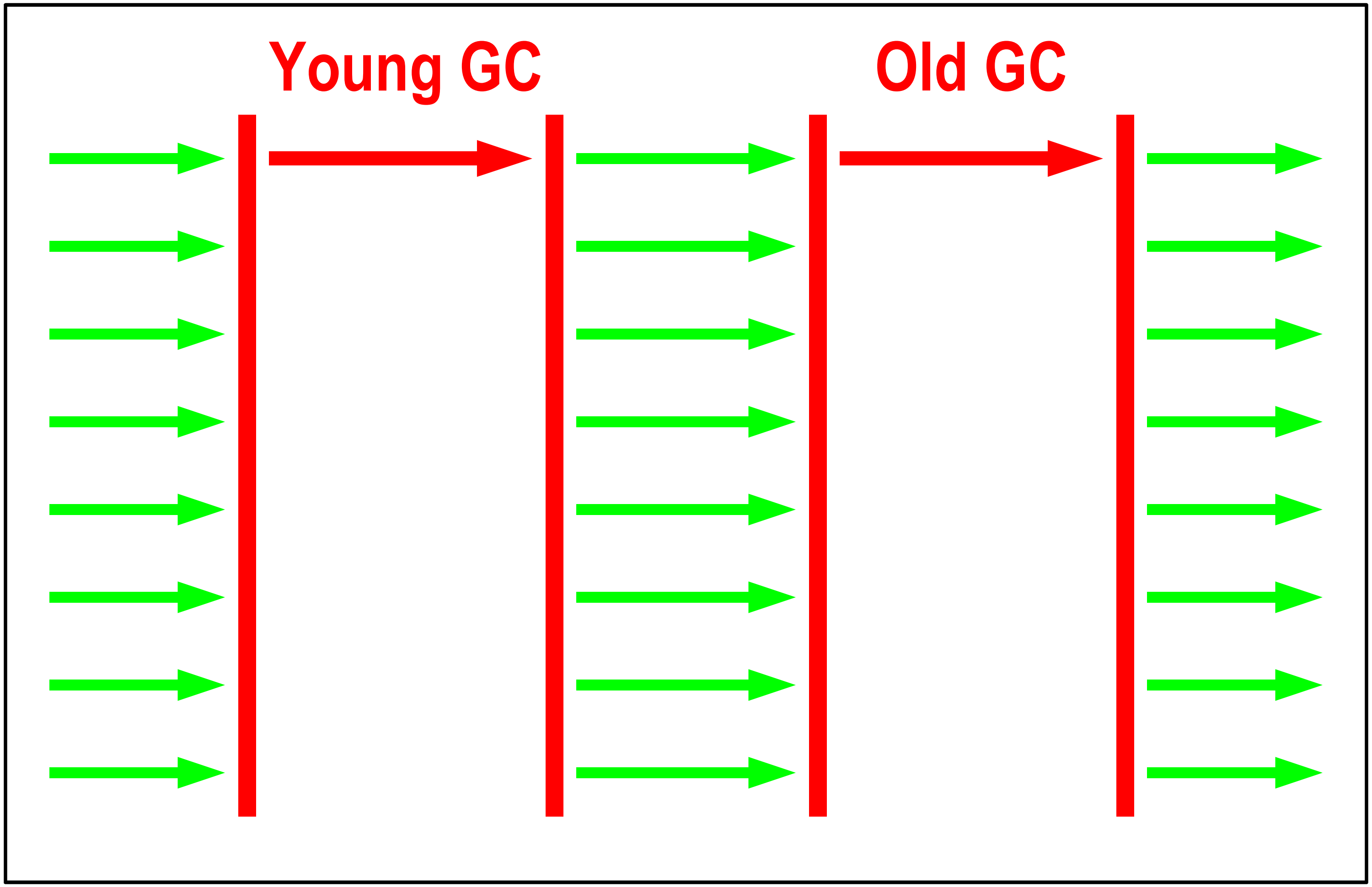
Card Table

- Cards: 512 bytes
- Card Table: byte array
- Identifies old-to-young references
  - Also used to identify reference mutation history in CMS
- Write Barrier
  - Inlined by the JIT
  - Two native instructions

Serial GC

- `-XX:+UseSerialGC`
- Serial Throughput GC
  - Serial STW copying young generation
  - Serial STW compacting old generation
- Default GC for client-type machines
- Simplest and most widely-used

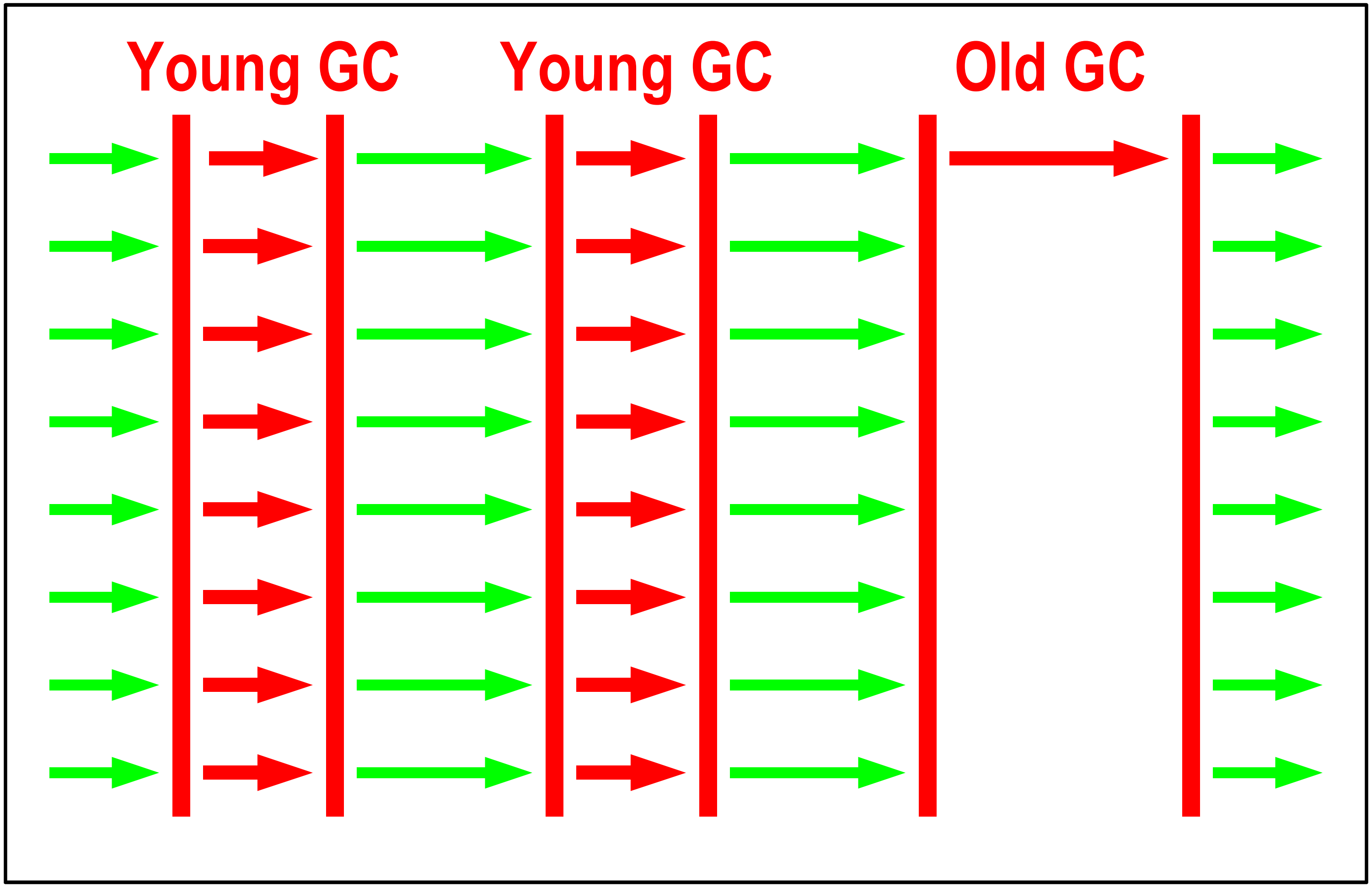
Serial GC



Parallel GC

- `-XX:+UseParallelGC`
- Parallel Throughput GC
  - Parallel STW copying young generation
  - Serial STW compacting old generation
- Default GC for server-type machines

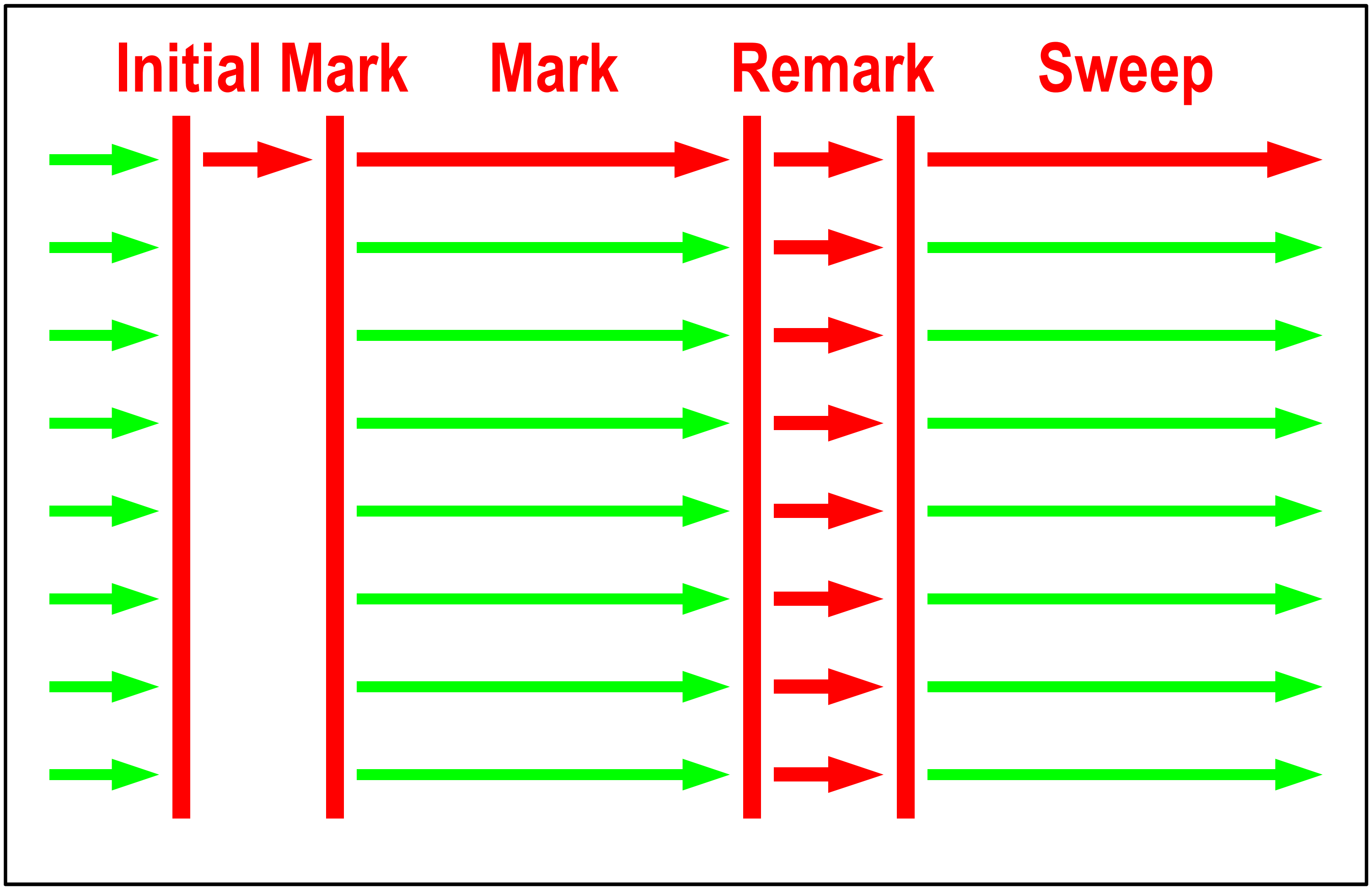
Parallel GC



Concurrent GC

- `-XX:+UseConcMarkSweepGC`
- Parallel and Concurrent Low-Latency GC
  - Parallel STW copying young generation
  - Concurrent and Parallel Mark-Sweep (non-compacting) old generation
  - Still STW pauses, but shorter
  - STW compacting GC, if GC not keeping up
- Low latency GC, best on server-type machines

Concurrent GC



GC Characteristics

- **Serial GC**
  - Most lightweight, good for uniprocessors, up to 1GB
- **Parallel GC**
  - Throughput-oriented for multiprocessors
  - Shortest young GC times
    - Efficient promotion due to compaction
  - Old generation GCs can be long
  - Parallel overhead on small-scale multiprocessors?

GC Characteristics

- **Concurrent GC**
  - > Low latency-oriented for multiprocessors
  - > Slower young GCs compared to Parallel GC
    - > Due to slower promotion
  - > Lower throughput compared to Parallel GC
  - > Higher memory footprint compared to Parallel GC
  - > ..., but much shorter old generation pauses!

When To Use Which?

- **Serial GC**
  - > Uniprocessors, multiple JVMs on Multiprocessors
- **Parallel GC**
  - > For Throughput-oriented Applications
  - > Multiprocessors
- **Concurrent GC**
  - > For Low Latency-oriented Applications
  - > Multiprocessors

Outline

- GC Background
- GCs in the Java HotSpot™ Virtual Machine
- **GC Ergonomics**
- Latest / Future Directions
- Thoughts on Predictable Garbage Collection

GC Ergonomics

- Tuning Parameter Hell!
- Can the GC do the right thing by itself?
  - Find a good starting configuration
  - Configure GC during execution

Starting Configuration

- Server-class Machine
  - 2+ CPUs, 2+ GB of memory
- Default configuration for server-class machines
  - Parallel GC
  - Initial heap size: 1/64th of memory, up to 1GB
  - Maximum heap size: 1/4th of memory, up to 1GB
  - Server compiler

GC Ergonomics

- The user specifies
  - Pause Time Goal
  - Throughput Goal
  - Maximum Footprint
- *Note:* first two are goals, not guarantees!

Pause Time Goal

- Measure young and old generation pauses
  - > Separately
  - > Average + Variance
- Shrink generation to meet goal
  - > A smaller generation is (usually!) collected faster
- Adapt to changing application behavior

Throughput Goal

- Measure throughput
  - Time spent in GC
  - Time spent outside GC
- Grow generations to meet throughput goal
  - A larger generation takes more time to fill up
- Adapt to changing application behavior

Survivor Spaces

- If survivor spaces overflow
  - > Decrease tenuring threshold
  - > Has priority over,
- Tenuring threshold adjustment
  - > Balance young / old collection times
  - > Higher → move GC overhead from old to young
  - > Lower → move GC overhead from young to old

Priorities

- Try to meet Pause Time Goal
- If Pause Time Goal is met
  - Try to meet Throughput Goal
- If both are met
  - Try to minimize Footprint

GC Ergonomics Status

- **Parallel GC**
  - > All the above
  - > Turned on by default
- **Concurrent GC**
  - > GC cycle initiation
    - > We've had this for a while
    - > Measure rate at which old generation is being filled
    - > Start GC cycle so it finishes before generation is full
  - > Rest
    - > Work in progress, incrementally available from 6.0 onwards

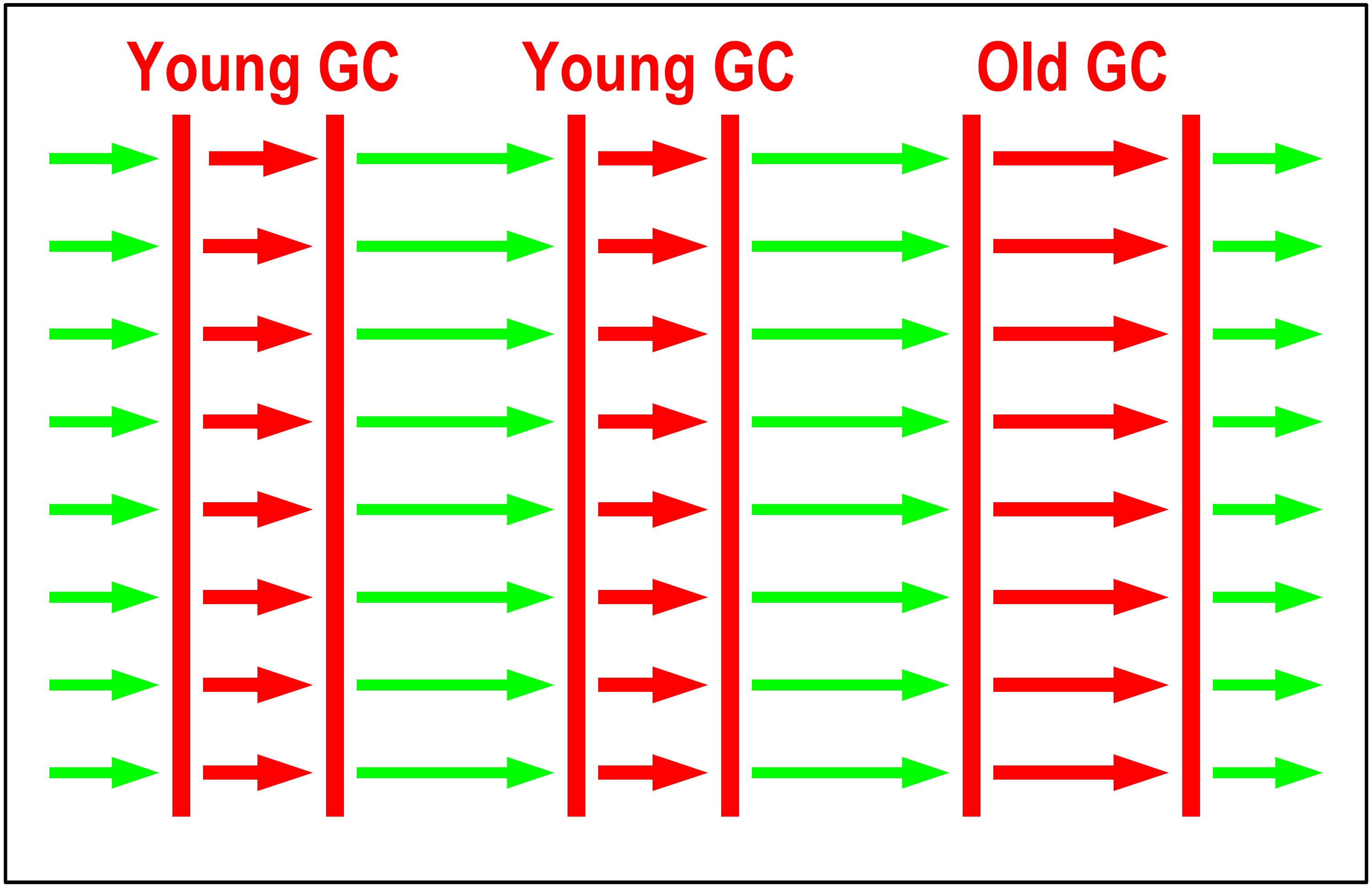
Outline

- GC Background
- GCs in the Java HotSpot™ Virtual Machine
- GC Ergonomics
- **Latest / Future Directions**
- Thoughts on Predictable Garbage Collection

Parallel Old GC

- `-XX:+UseParallelOldGC`
- Parallel Throughput GC
  - > Parallel STW copying young generation
  - > Parallel STW compacting old generation
- Eventual default in Parallel GC
  - > Maturing since 5.0_06
- Try it out! Tell us what you think!

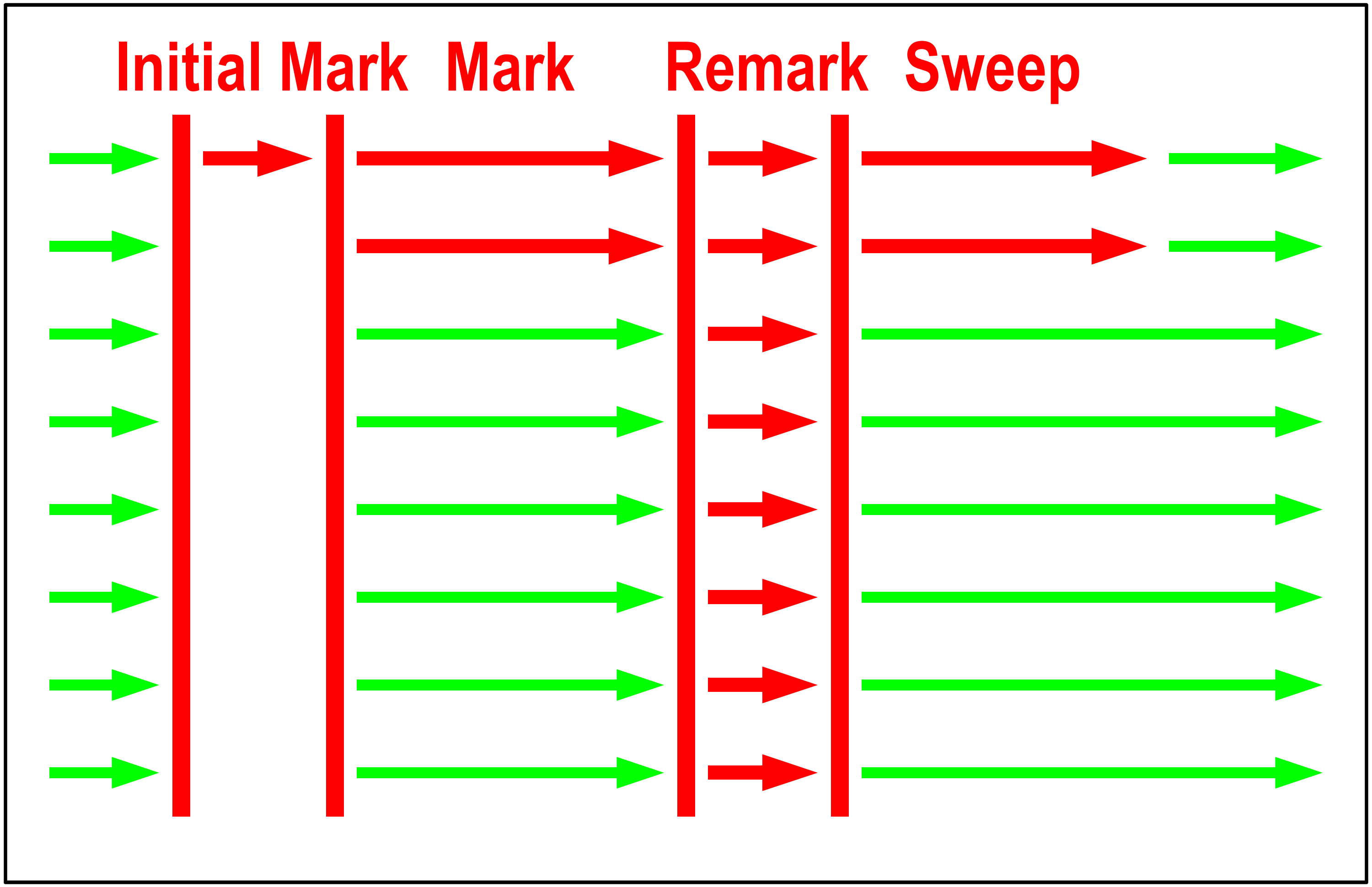
Parallel Old GC



Parallelized Concurrent GC

- Single concurrent GC thread good for 4-8 CPUs
- Cannot keep up with more
- Parallelized
  - > Concurrent Marking Phase
  - > Concurrent Sweeping Phase
- Targeting large multiprocessors
- Parallel Marking in 6.0, Parallel Sweeping to follow

Parallelized Concurrent GC



Outline

- GC Background
- GCs in the Java HotSpot™ Virtual Machine
- GC Ergonomics
- Latest / Future Directions
- **Thoughts on Predictable Garbage Collection**

Sample Customer Quote

"The garbage collector should not pause my application for more than 100 ms."

Pause Time Goal

- A Pause Time Goal alone is not very helpful
  - What happens if the application only runs for *1 ms* between *100 ms* pauses? :-)

- What the customer really meant:

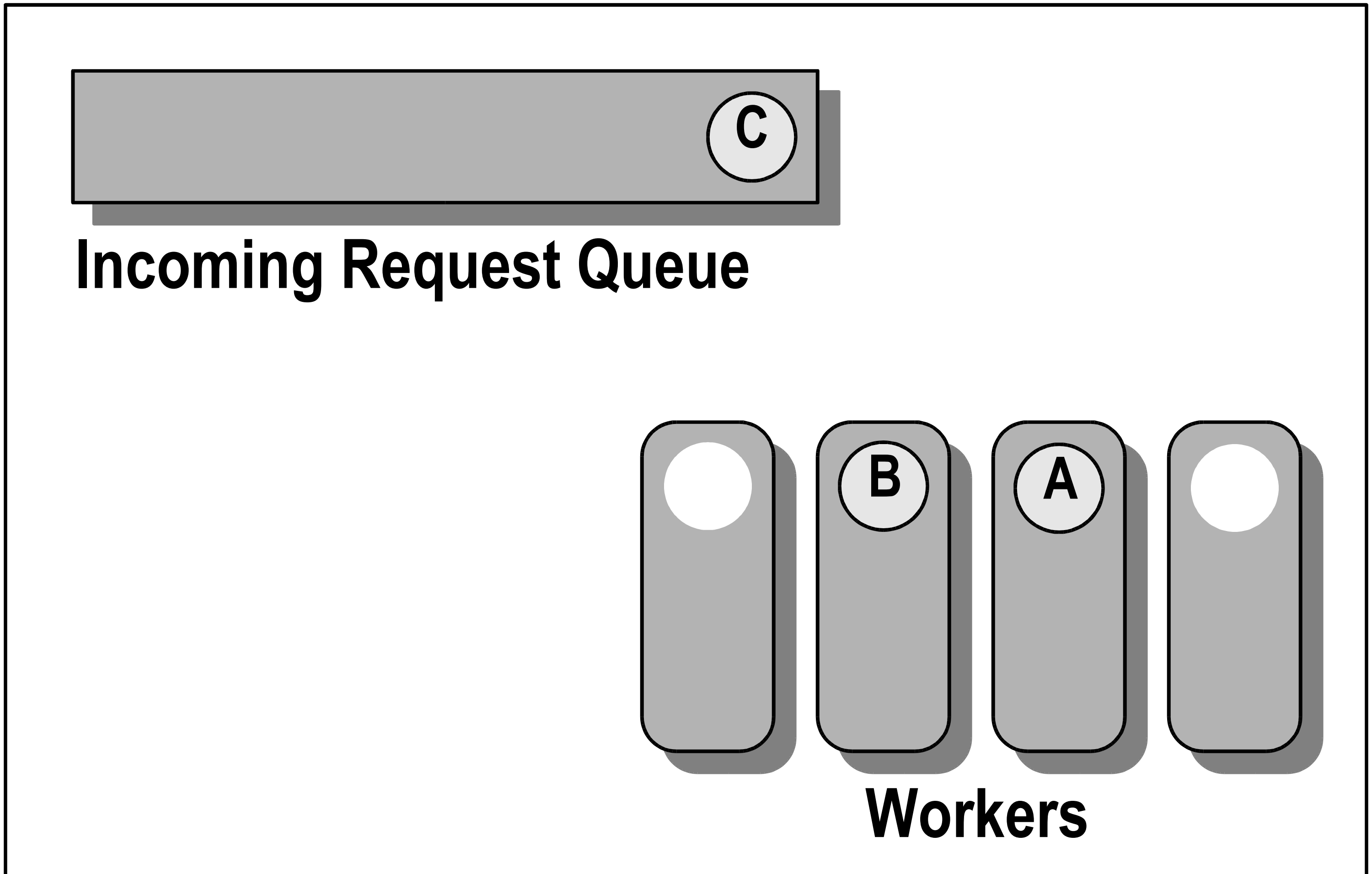
  “The garbage collector should not pause my application for more than 100 ms ***and*** *it should be scheduled reasonably infrequently.”*

More Accurate GC Goal

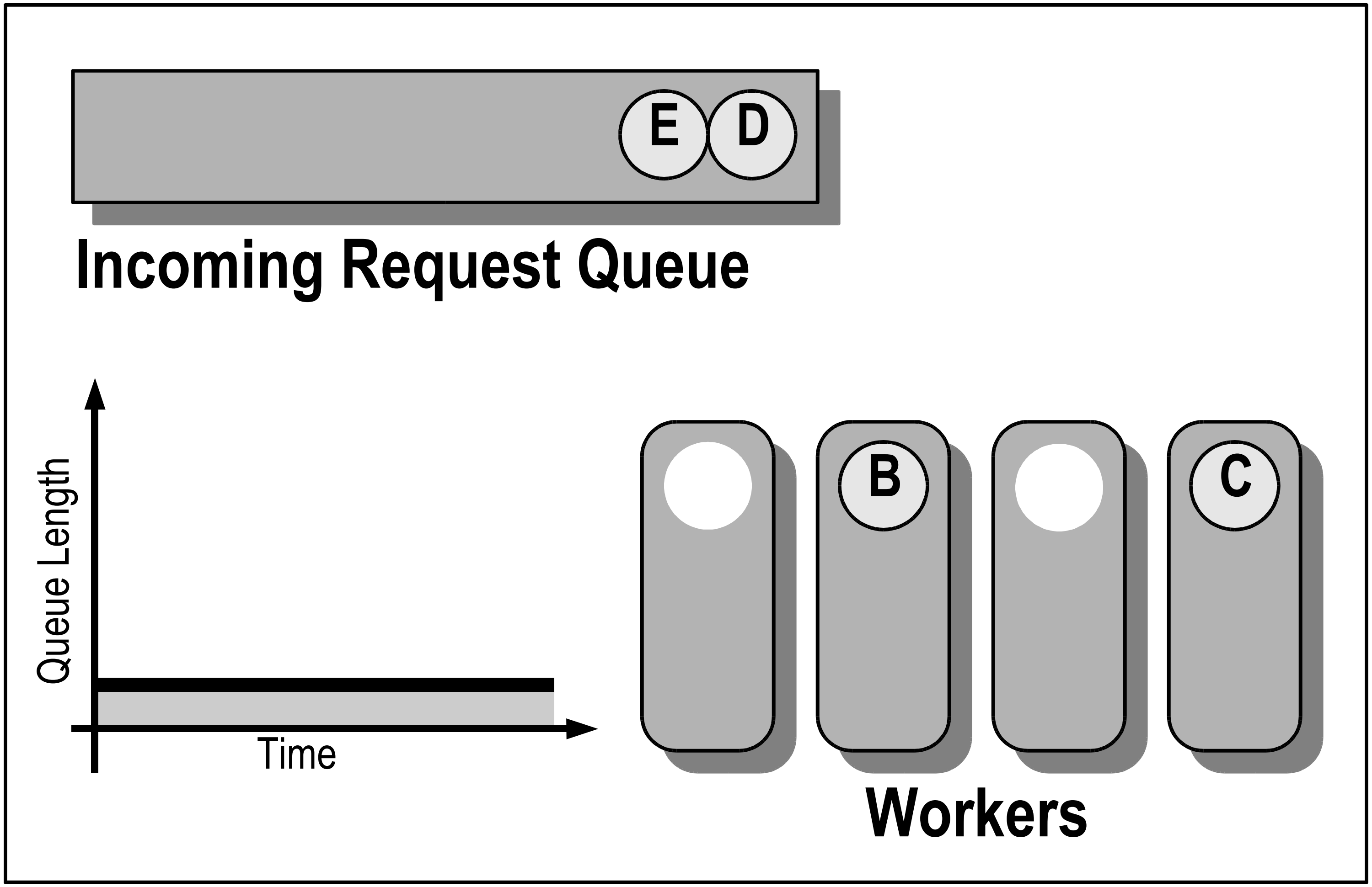
- A more accurate GC goal comprises
  - A Max GC Time *x ms*
  - A Time Slice *y ms*

"The garbage collector should not take more than x ms for any y ms time slices of the application."

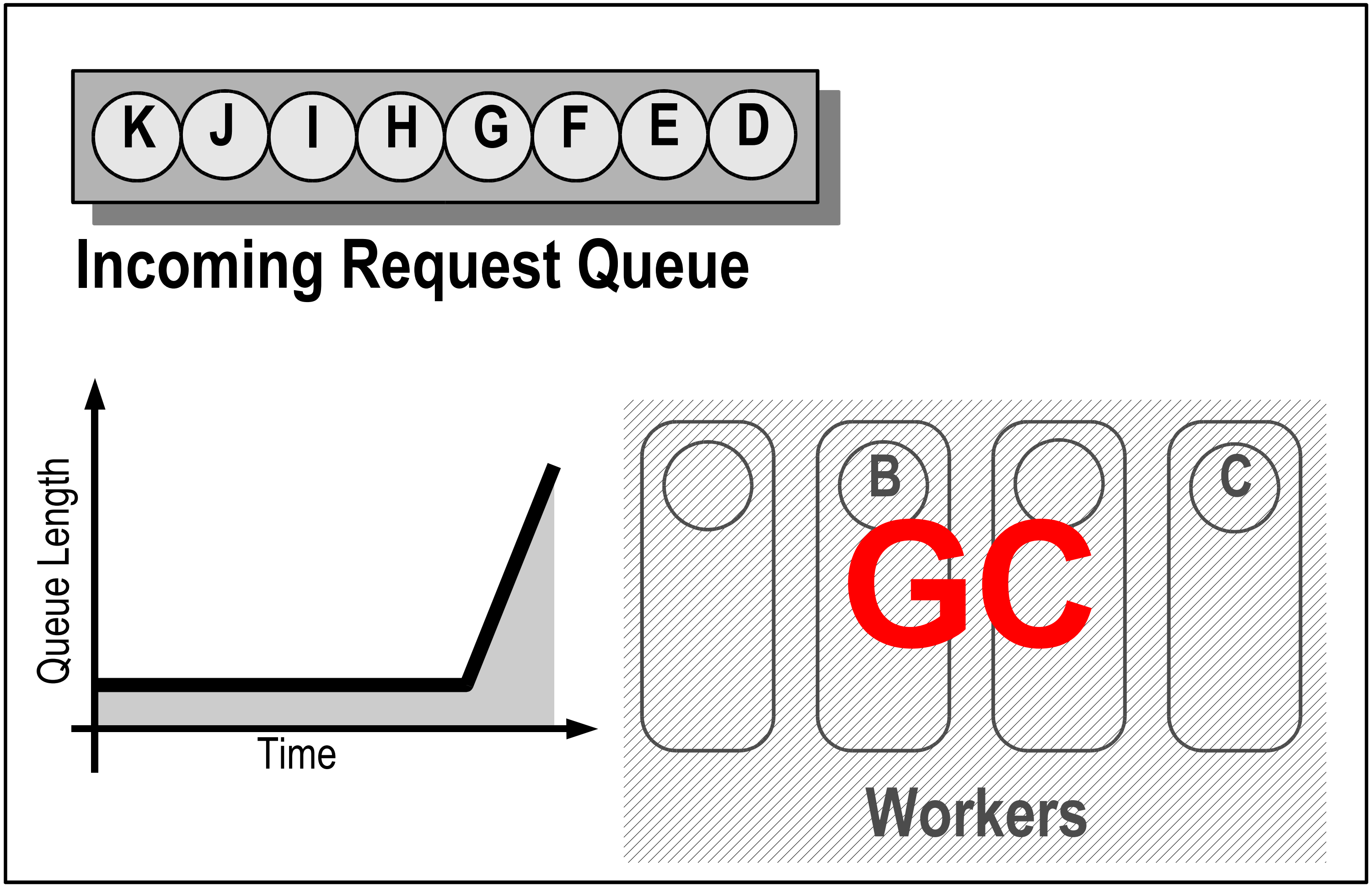
Example



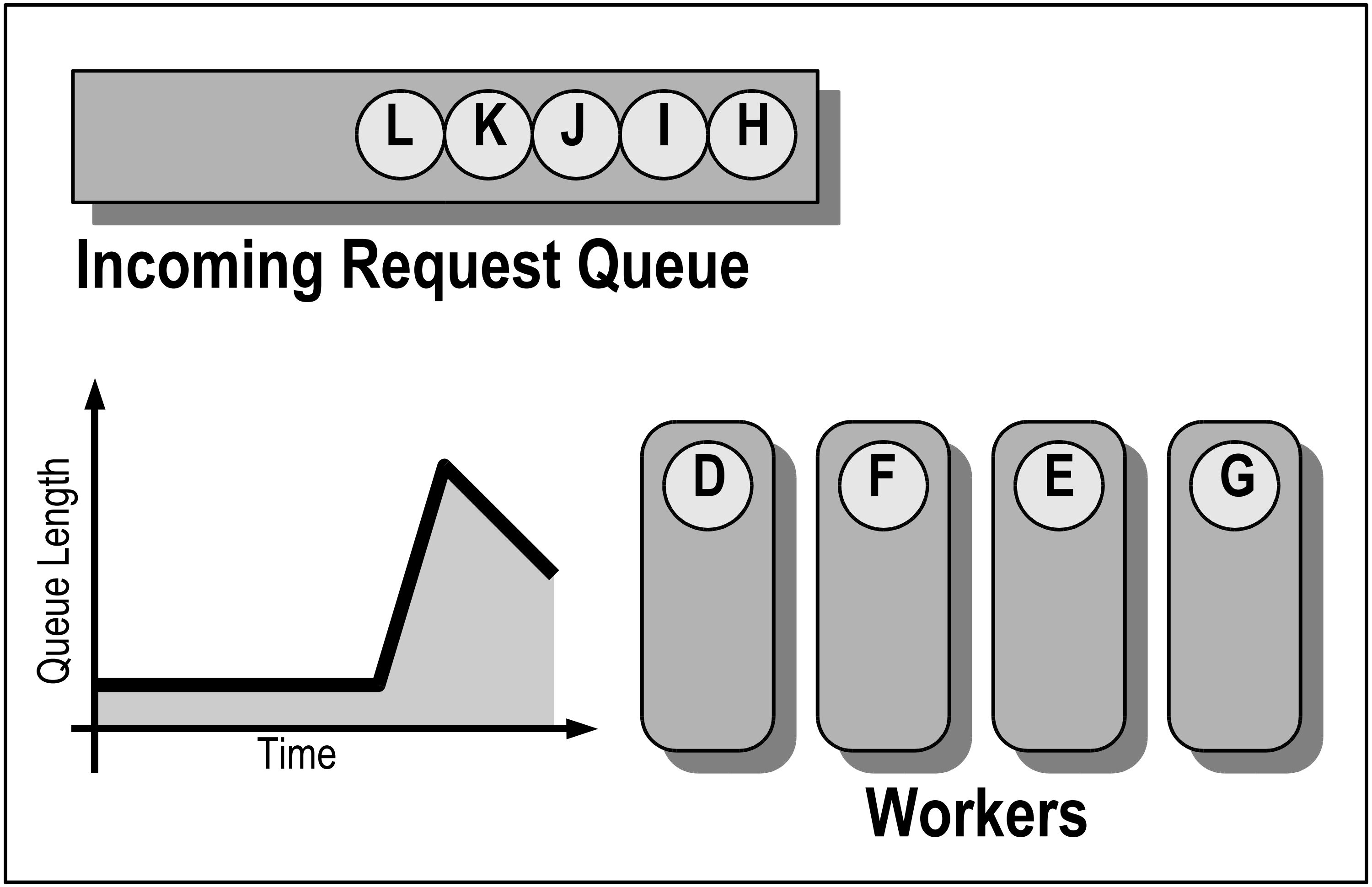
Example



Example

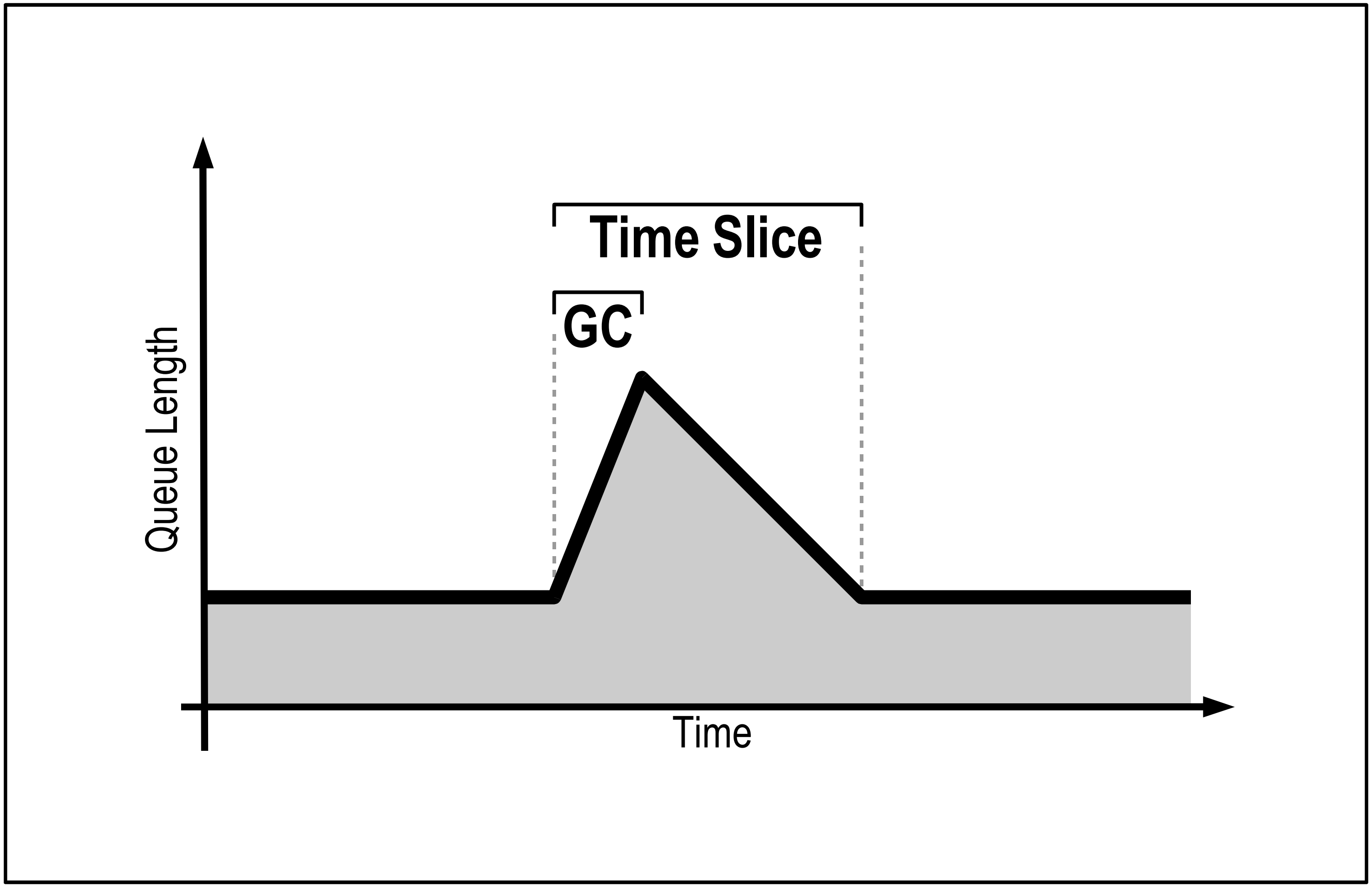

K
J
I
H
G
F
E
D
Incoming Request Queue
Queue Length
Time
B
C
GC
Workers

Example

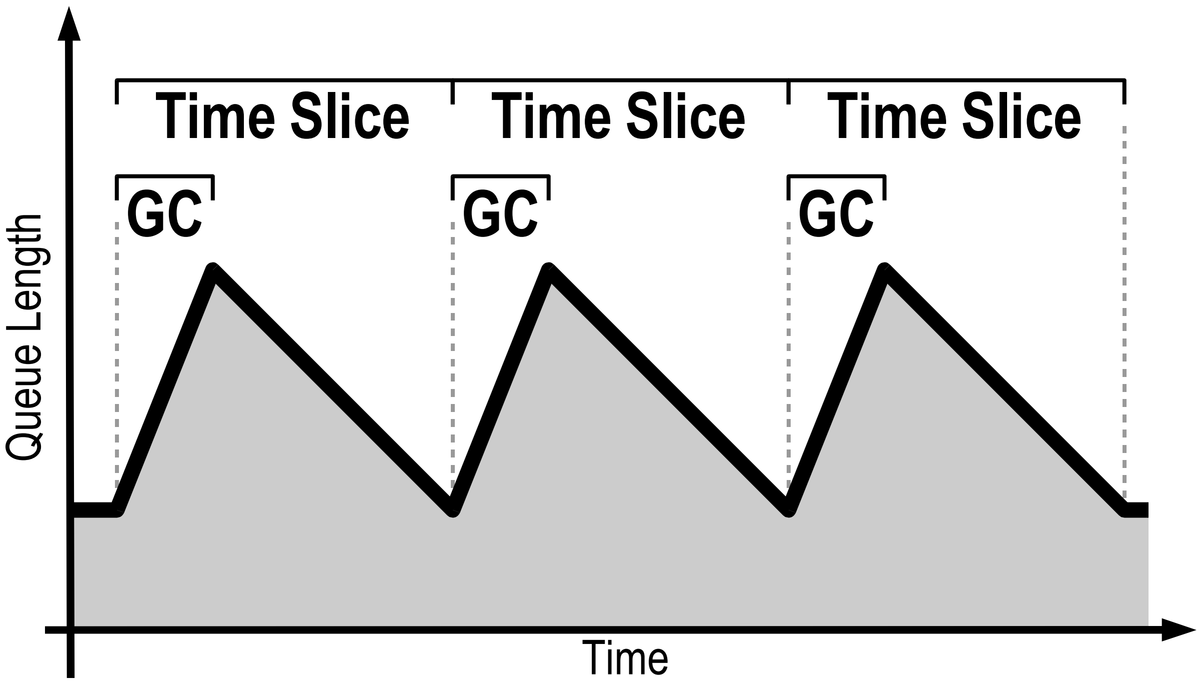


L
K
J
I
H
Incoming Request Queue
Queue Length
Time
D
F
E
G
Workers

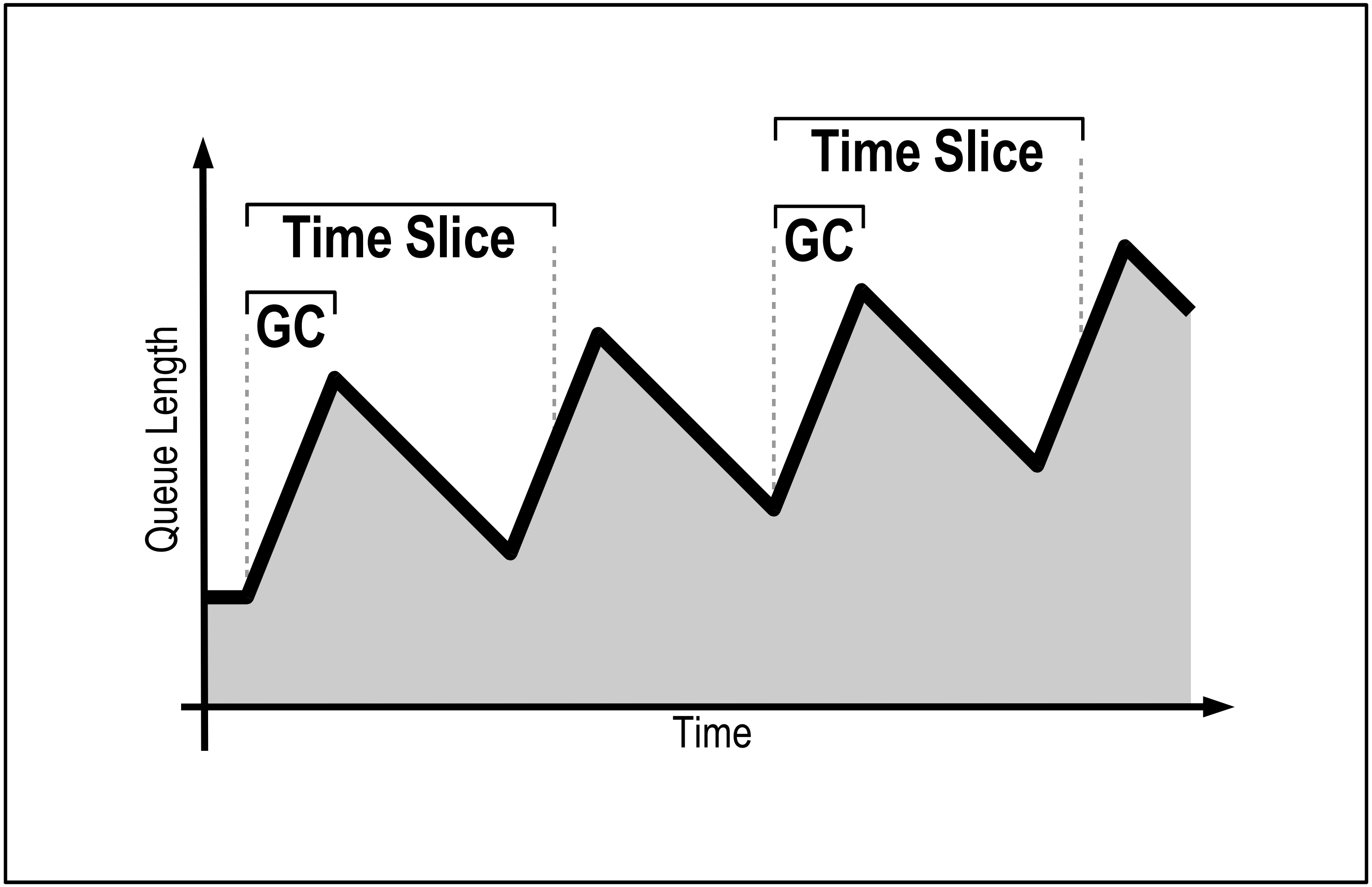
Example



Example



Example



Another Example

- Video Rendering Application
  - > It has to render one frame per *40 ms*
  - > Rendering takes up to *30 ms*
  - > This gives a GC Goal of *10 ms* out of *40 ms*
- Many other goals will cause frames to be dropped
  - > *1 ms* out of *3 ms*
  - > *10 ms* out of *20 ms*
  - > *50 ms* out of *10,000 ms*
  - > *100 ms* out of 4*00 ms*

The Point Of The Examples

- Bounding Pause Times
  - Is important
  - Does not guarantee required behavior
- Scheduling pauses (and other GC activity) is equally important
- A *useful* predictable GC has to do both

GC Goal Not Always Feasible

- Work-based GCs can inherently keep up with an application, e.g.,
  - Collect the young generation whenever it is full
- Time-based GCs might not
  - A given GC goal might not allow the GC to keep up with an application's allocation rate
  - A GC goal must be *realistic* for a particular application / GC combination

Is A GC Goal Realistic?

- Trial and Error
- Program Analysis
  - Required for hard real-time applications
  - Not realistic for most applications
- GC can also decide to miss the goal gracefully, not abruptly
  - Try to meet a slightly different goal, not fail

Predictable $\neq$ Fast

- Predictable / low latency GCs usually sacrifice
  - Some throughput
  - Some memory footprint
- GC cycles are longer in “application time”
- Some GC work needs to be redone
  - To synchronize with the application's operation

Possible Future Direction

- **Garbage-First Garbage Collection**
  - > SunLabs project
  - > Potential replacement for Concurrent GC
    - > ..., also a SunLabs project!
  - > Currently under evaluation

Garbage-First Garbage Collector

- Server-Style Garbage Collector
- Generational
- Concurrent / Parallel
- Low-Pause / Predictable (tries to meet GC goal)
- Compacting
- Ergonomic
- Very Good Throughput

Garbage-First Garbage Collection

- The heap is split into fixed size *Regions*
  - > The young generation is a set of regions
  - > Young survivors evacuated out of the young generation
    - > Evacuation Pauses
- Concurrent Marking Phase
  - > Identifies mostly-dead regions, cheapest to collect
  - > Collect mostly-dead regions during evacuation pauses
- Cost model
  - > For all GC activity
  - > Predict / Schedule to meet a given GC Goal
  - > Accurate, but no hard guarantees