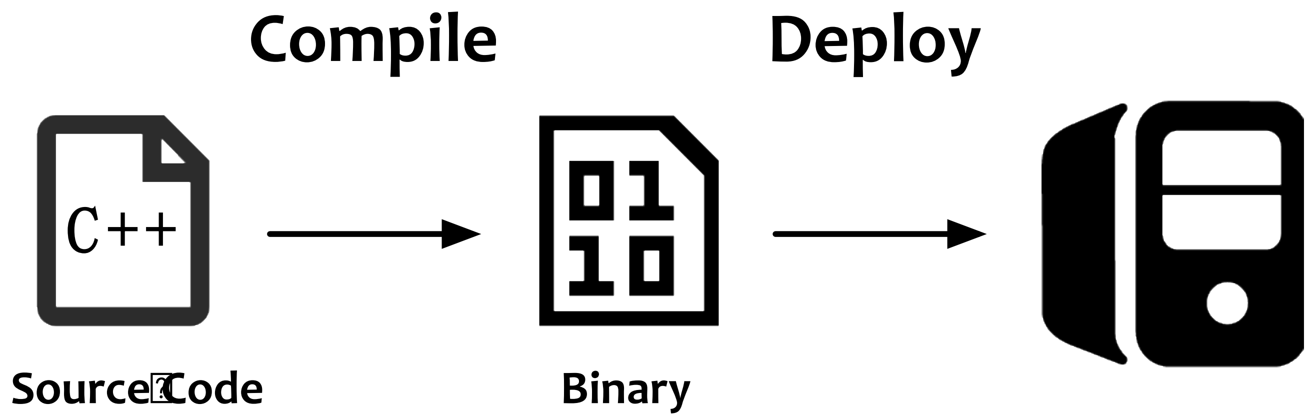
Language Virtual Machines

Mingyu Wu

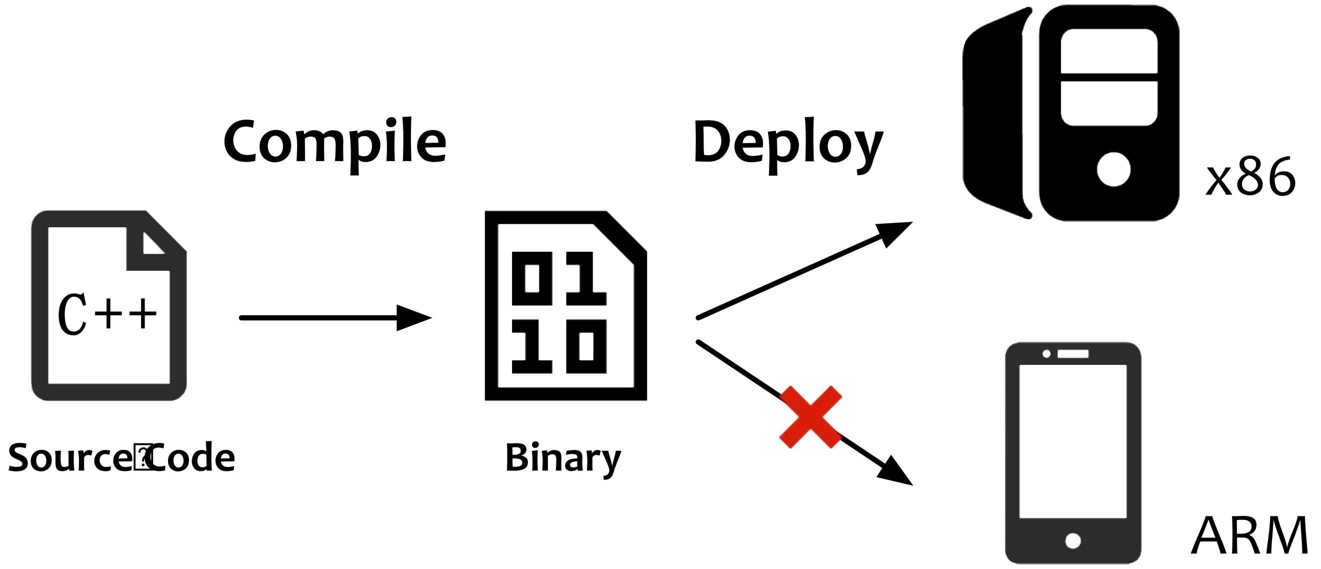
INTRODUCTION

-- Why do we need a language virtual machine?

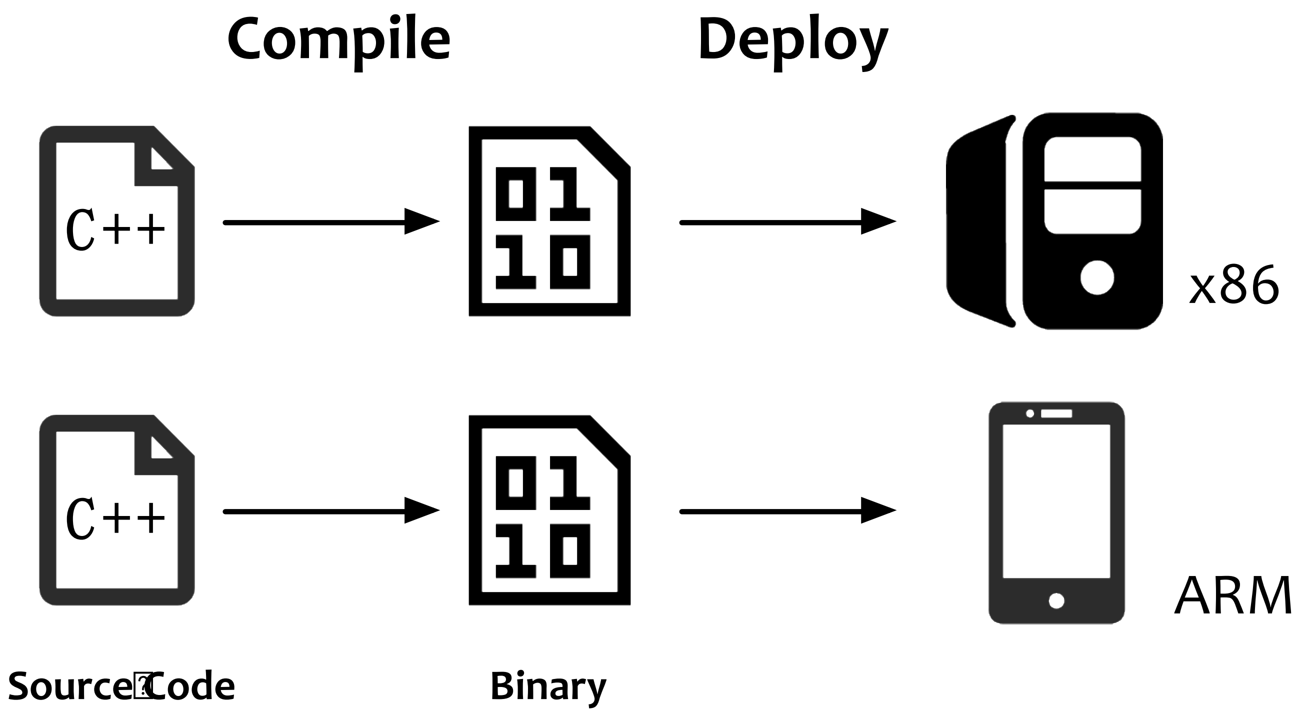
Workflow for Native Languages



Problem: Multiple ISAs

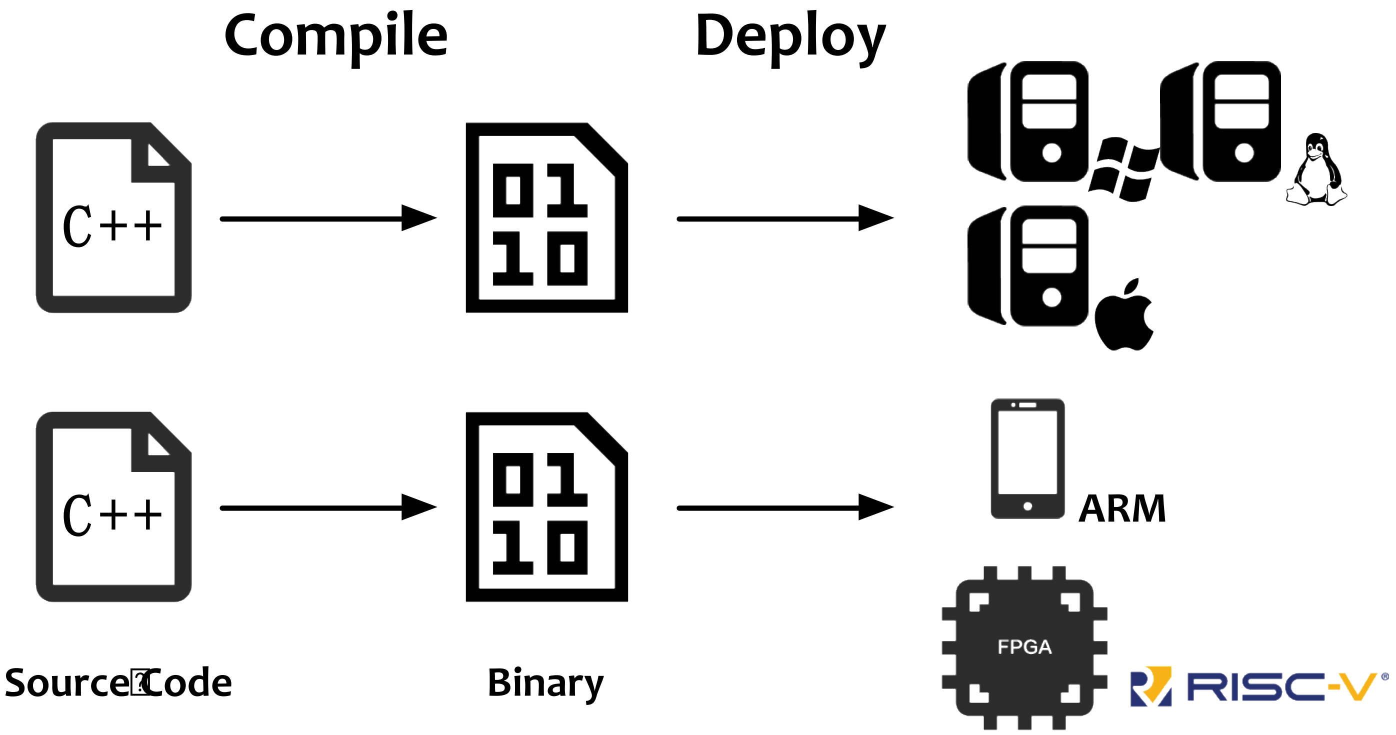


Solution: “One for Each”



However...

- The deployment environment is complicated!



Another Consideration: Security

- **Out-of-bound access**

```
char * chs = malloc(10);
Char ch = chs[10]; // oops!
```

- **Use-after-free**

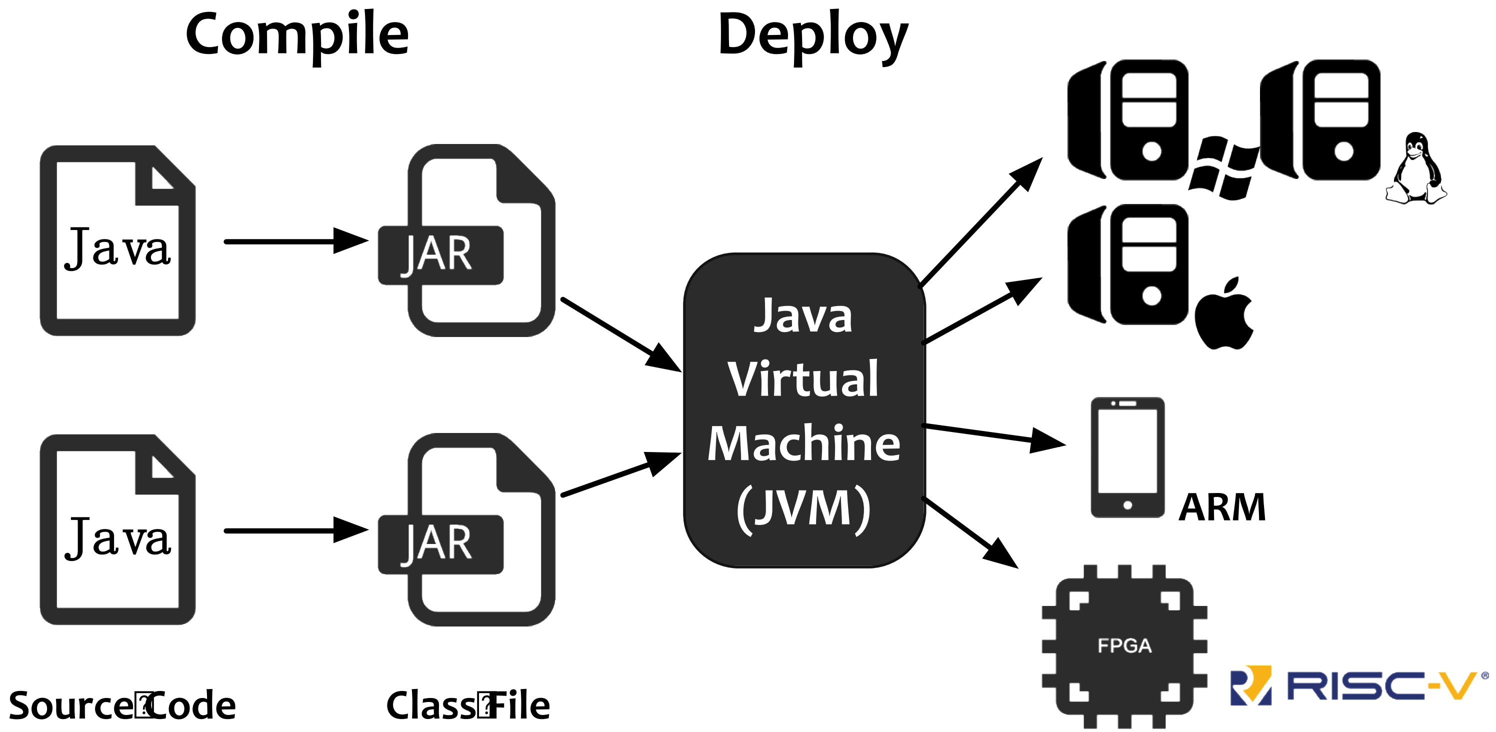
```
free(chs);
Char ch = chs[5]; // oops!
```

“All problems in computer science can be solved by another level of indirection”

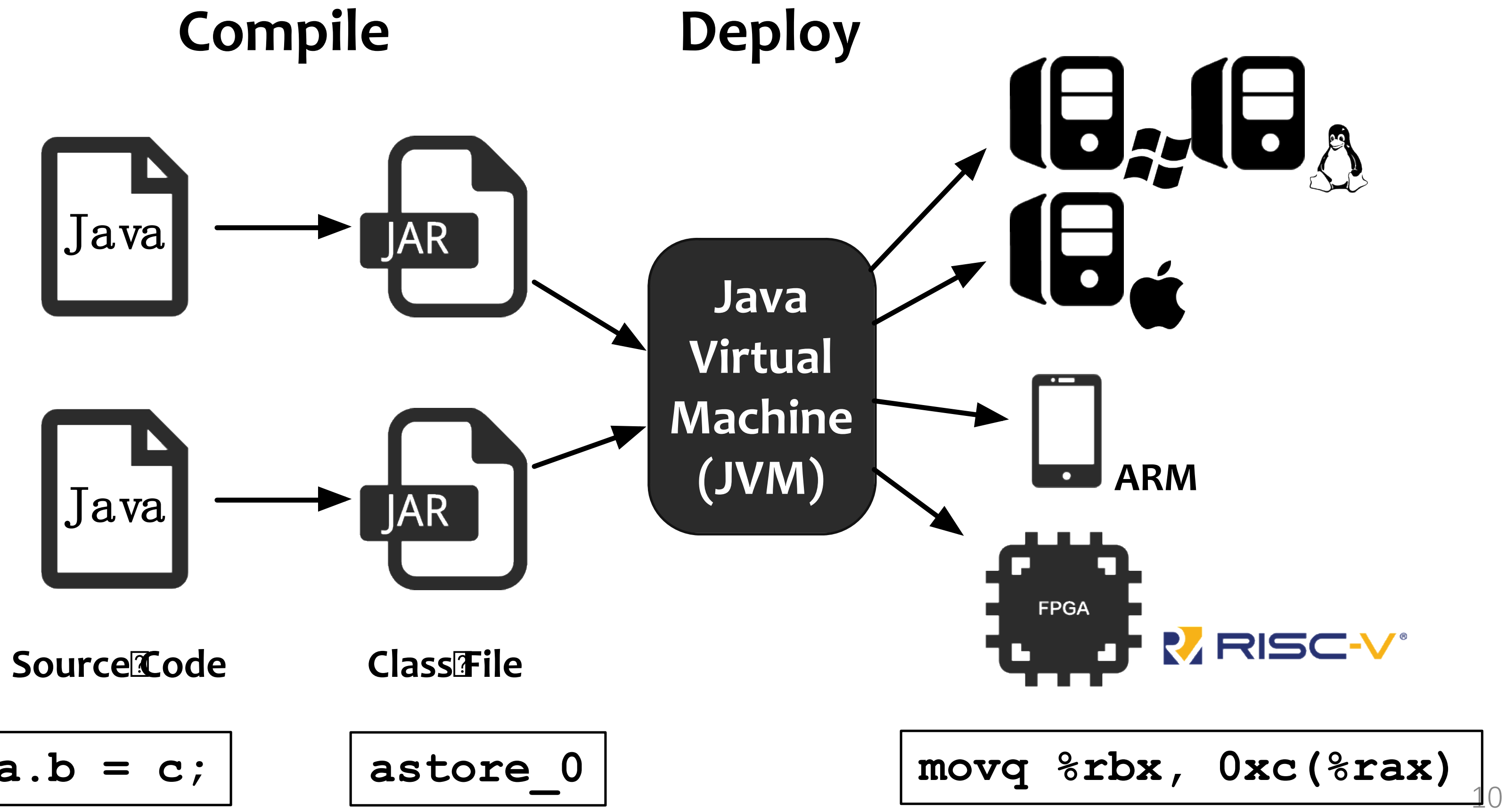
David Wheeler

University of Cambridge

JVM: The "Indirection"



JVM: The "Indirection"



```
a.b = c;
```

```
astore_0
```

```
movq %rbx, 0xc(%rax)
```

Security Enhancement

- **Out-of-bound access ← runtime check**

```
char * chs = malloc(10);
Char ch = chs[10]; // ArrayOutOfBoundException
```

- **Use-after-free ← no free at all!**

```
~~free(chs);~~
Char ch = chs[5]; // No problem!
```

An Introduction to JVM: Outline

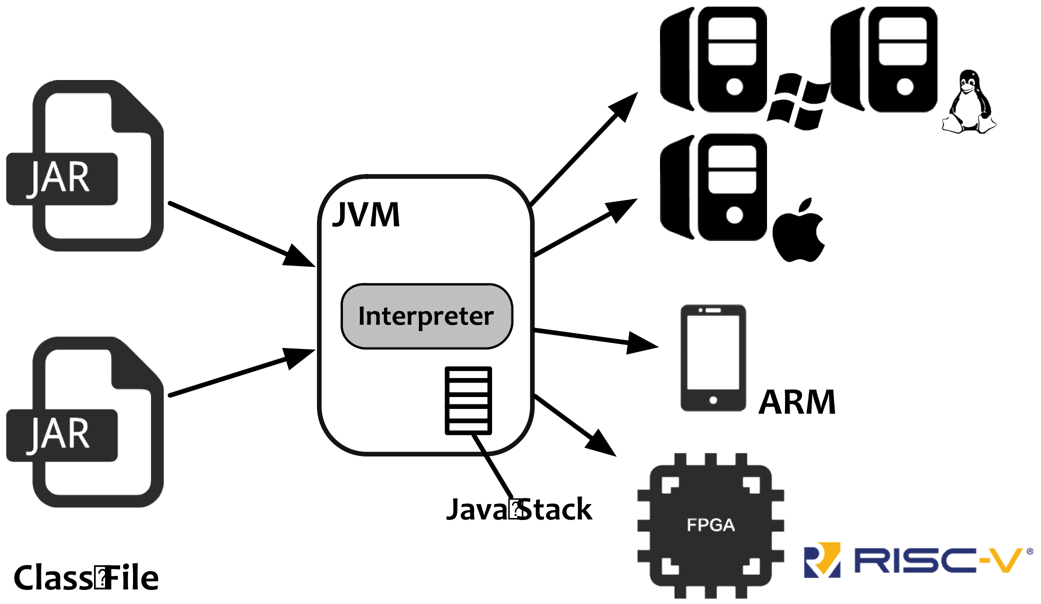
- **Code Execution**
  - Interpreter
  - JIT compiler
- **Memory management**
  - Heap layout
  - Basic GC algorithms
  - Modern garbage collectors

EXECUTION

-- Interpreter & compiler

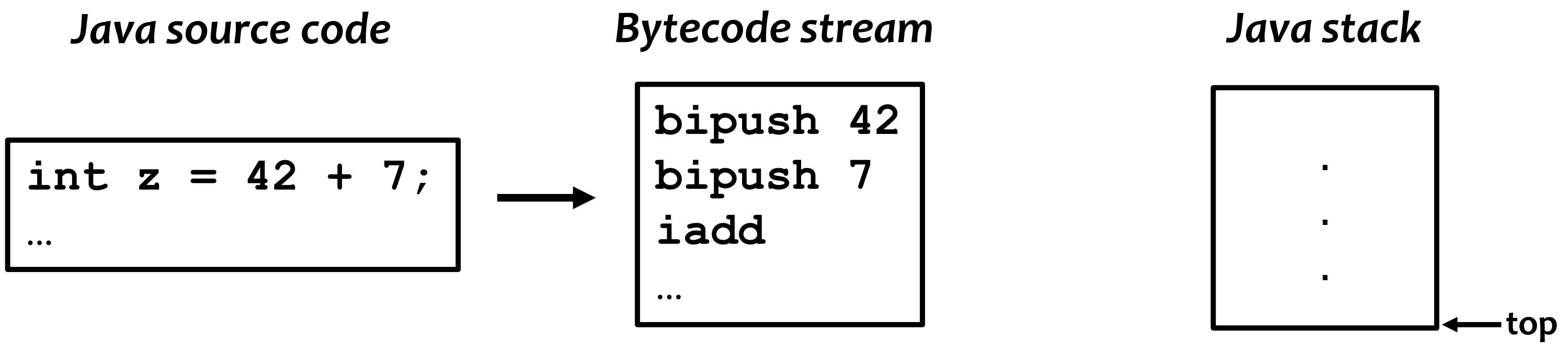
Execution Mode: Interpretation

- JVM has a **interpreter** to execute those class files with a **stack**



Stack-based Interpretation

- **All instructions (bytecode) are operated on the stack**

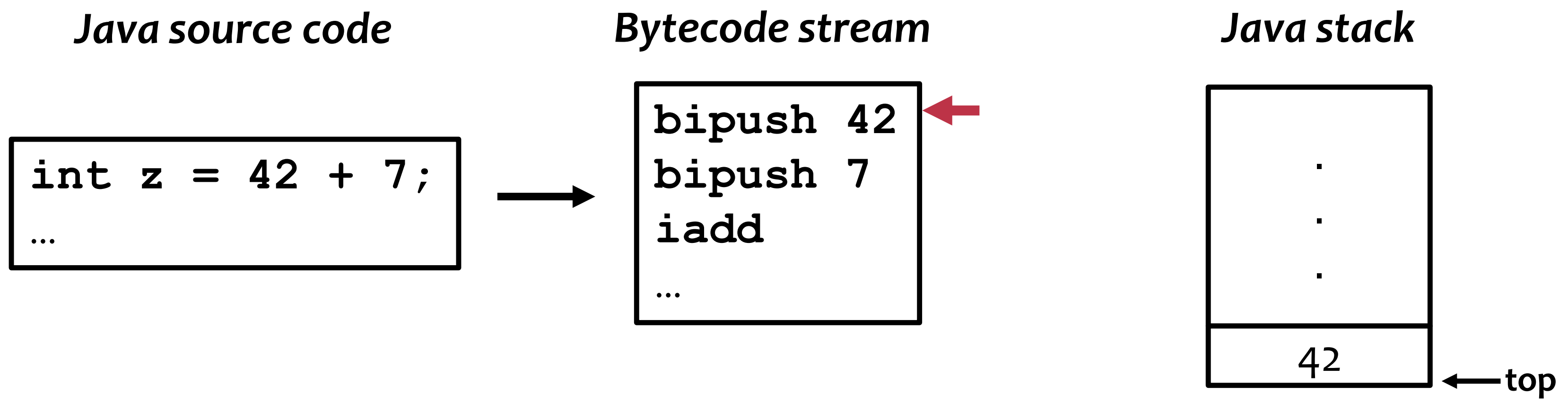


```
int z = 42 + 7;
…
```

```
bipush 42
bipush 7
iadd
…
```

Stack-based Interpretation

- All instructions (bytecode) are operated on the stack

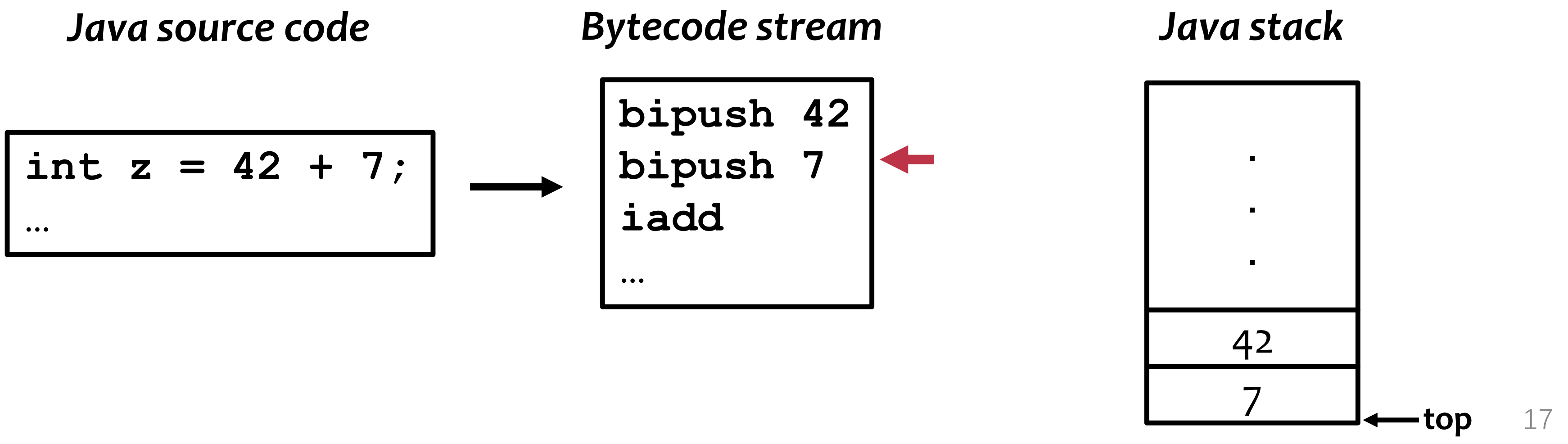


```
int z = 42 + 7;
...
```

```
bipush 42
bipush 7
iadd
...
```

Stack-based Interpretation

- **All instructions (bytecode) are operated on the stack**

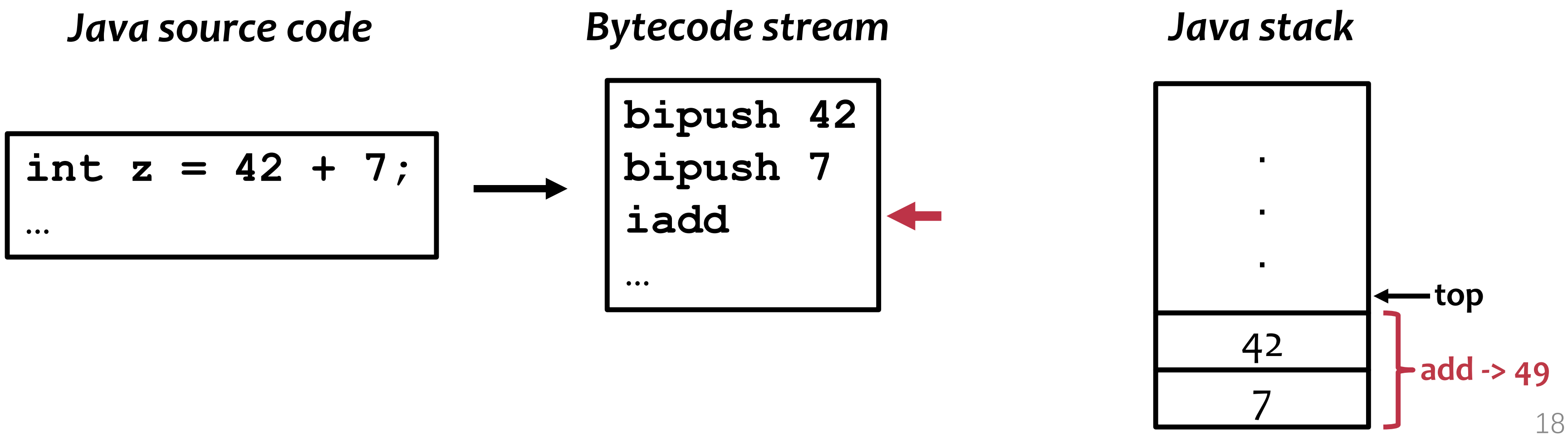


```
int z = 42 + 7;
…
```

```
bipush 42
bipush 7
iadd
…
```

Stack-based Interpretation

- **All instructions (bytecode) are operated on the stack**

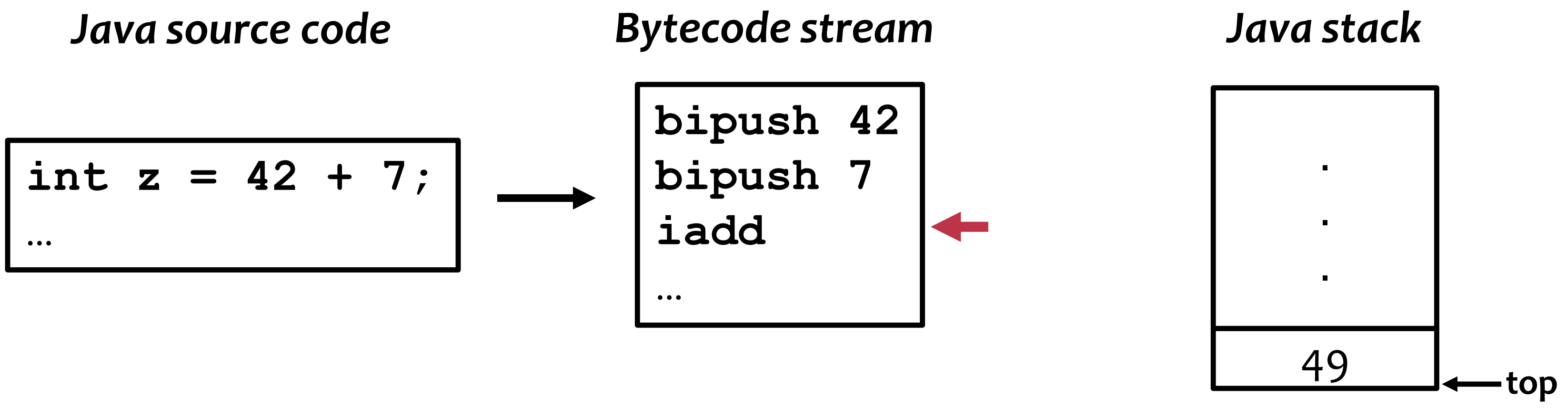


```
int z = 42 + 7;
…
```

```
bipush 42
bipush 7
iadd
…
```

Stack-based Interpretation

- All instructions (bytecode) are operated on the stack

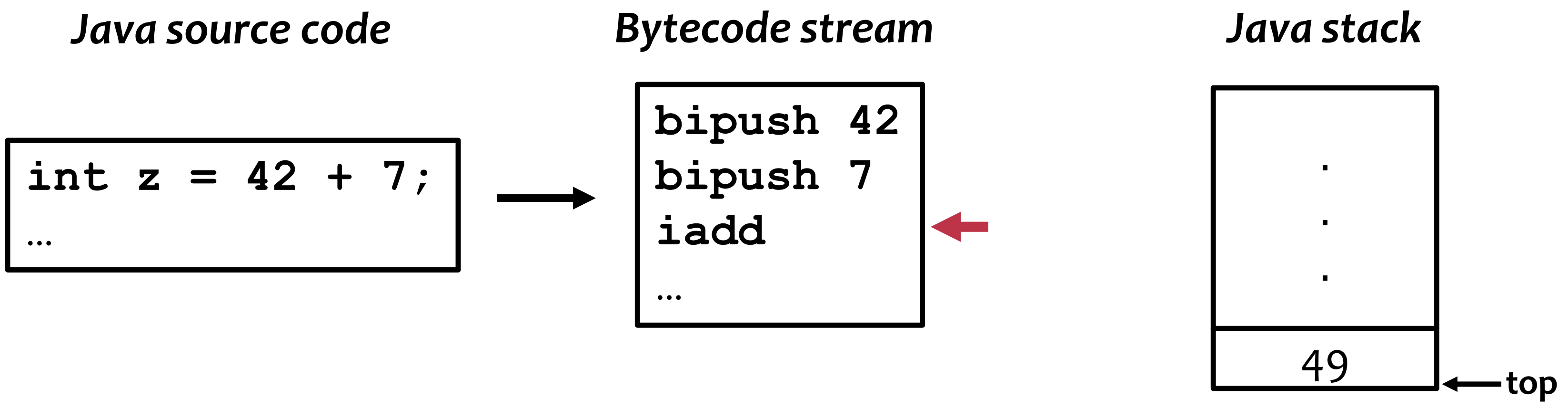


```
int z = 42 + 7;
…
```

```
bipush 42
bipush 7
iadd
…
```

Stack-based Interpretation

- All instructions (bytecode) are operated on the stack
- Other languages (like Python) have similar design

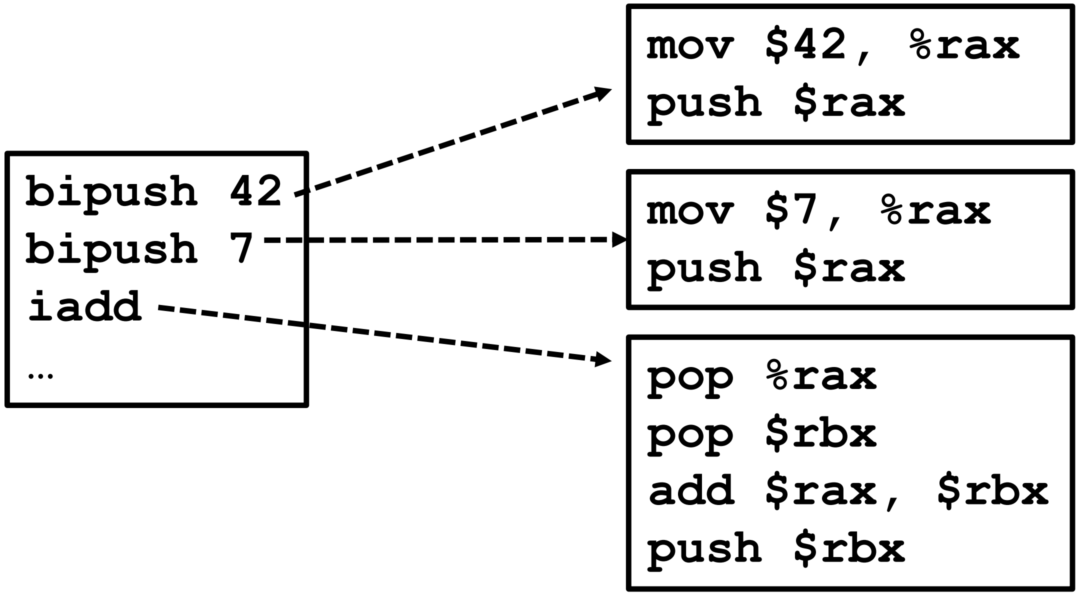


```
int z = 42 + 7;
...
```

```
bipush 42
bipush 7
iadd
...
```

Problem: Efficiency

- **Too many memory operations (>=1 per bytecode)**
  - Low utilization of registers
- **Missing optimization opportunities**



```
bipush 42
bipush 7
iadd
…
```

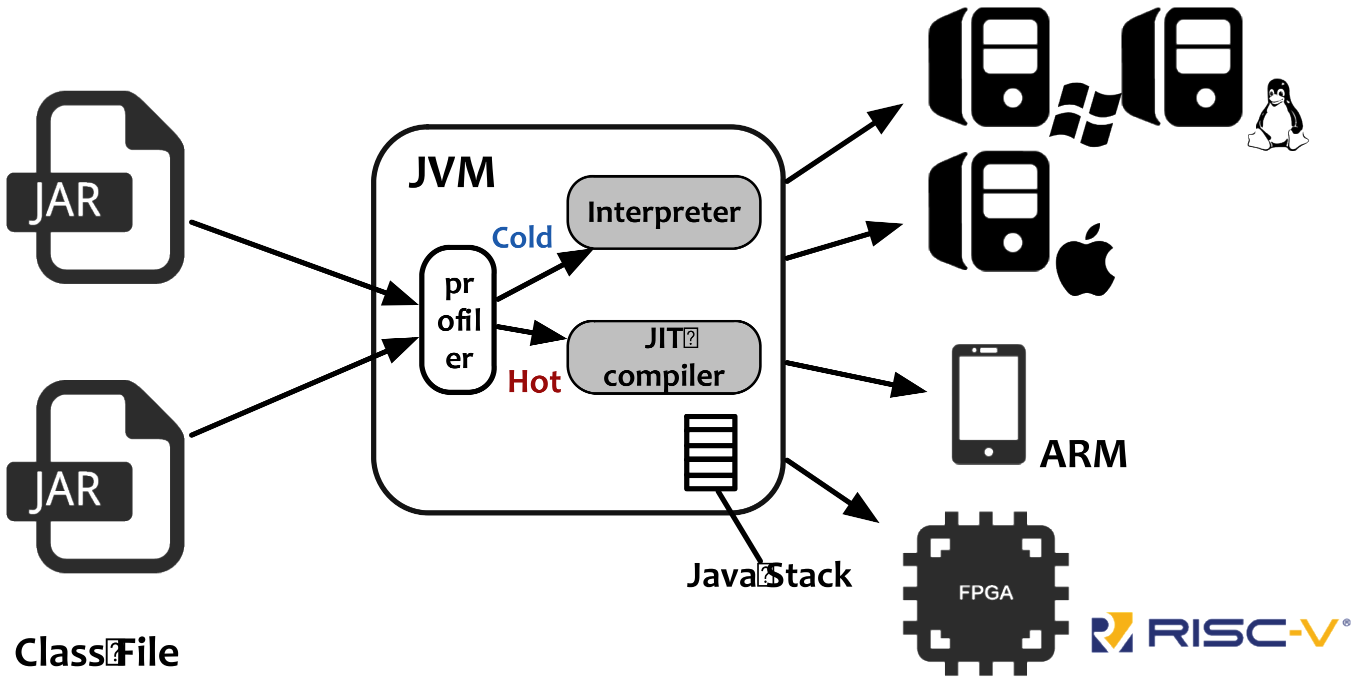
```
mov $42, %rax
push $rax
```

```
mov $7, %rax
push $rax
```

```
pop %rax
pop $rbx
add $rax, $rbx
push $rbx
```

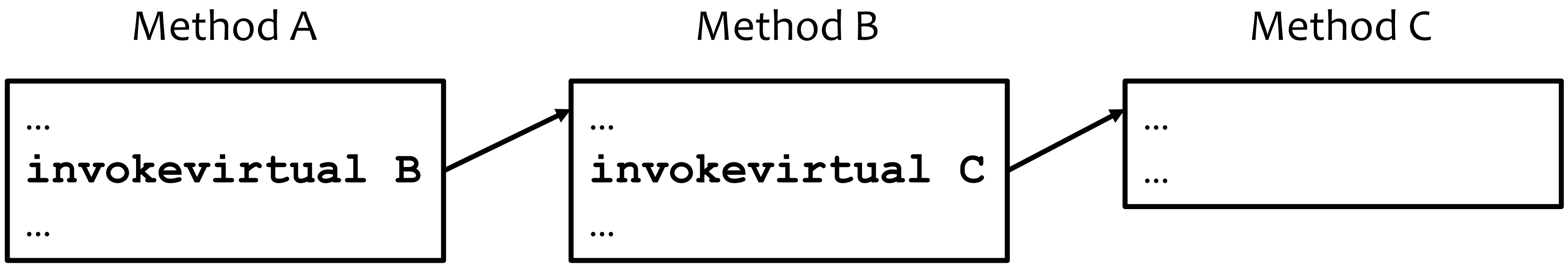
Solution: JIT (Just-In-Time) Compiler

- **Hot code will be compiled and executed**
  - "Hot" means that it has been executed for times



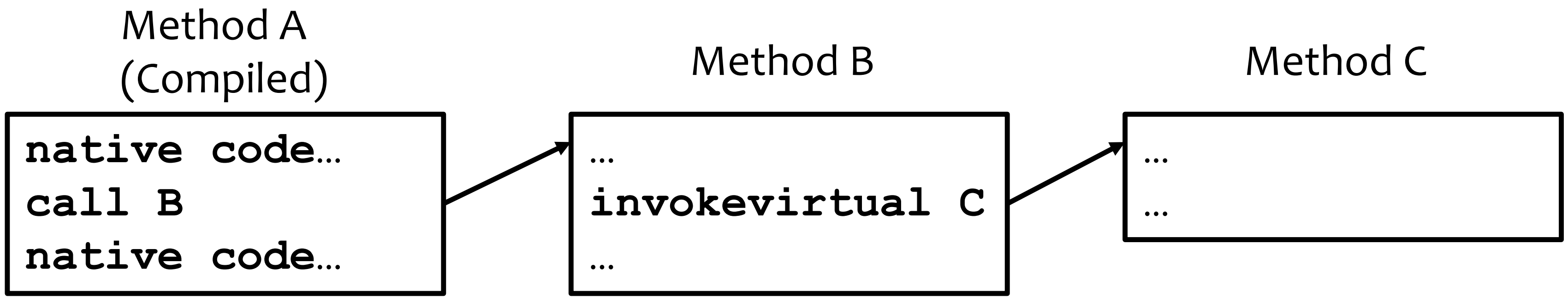
JIT: Aggressive Or Conservative?

- An example: inlining



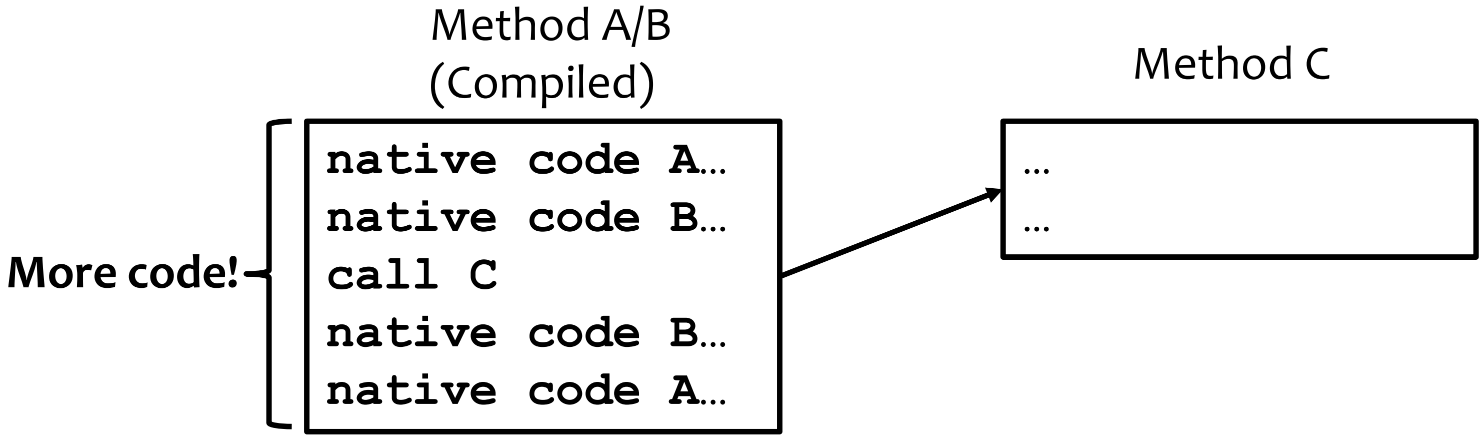
JIT: Aggressive Or Conservative?

- An example: inlining
- Choice 1: only compile A



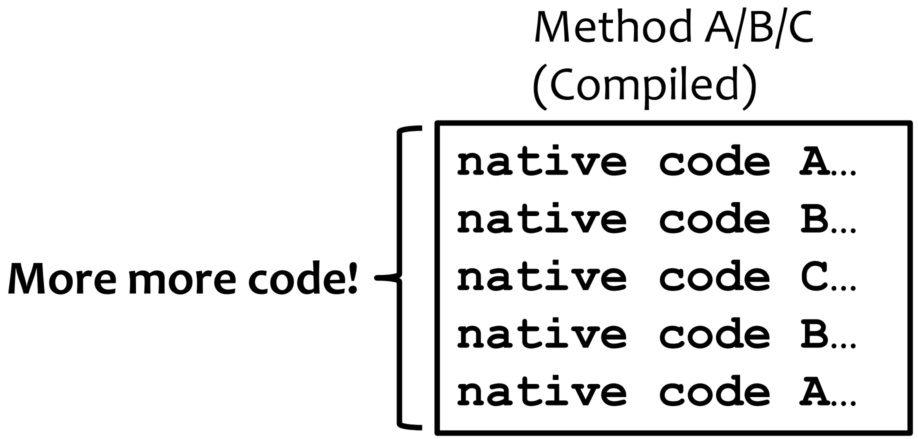
JIT: Aggressive Or Conservative?

- An example: inlining
- Choice 2: inline B



JIT: Aggressive Or Conservative?

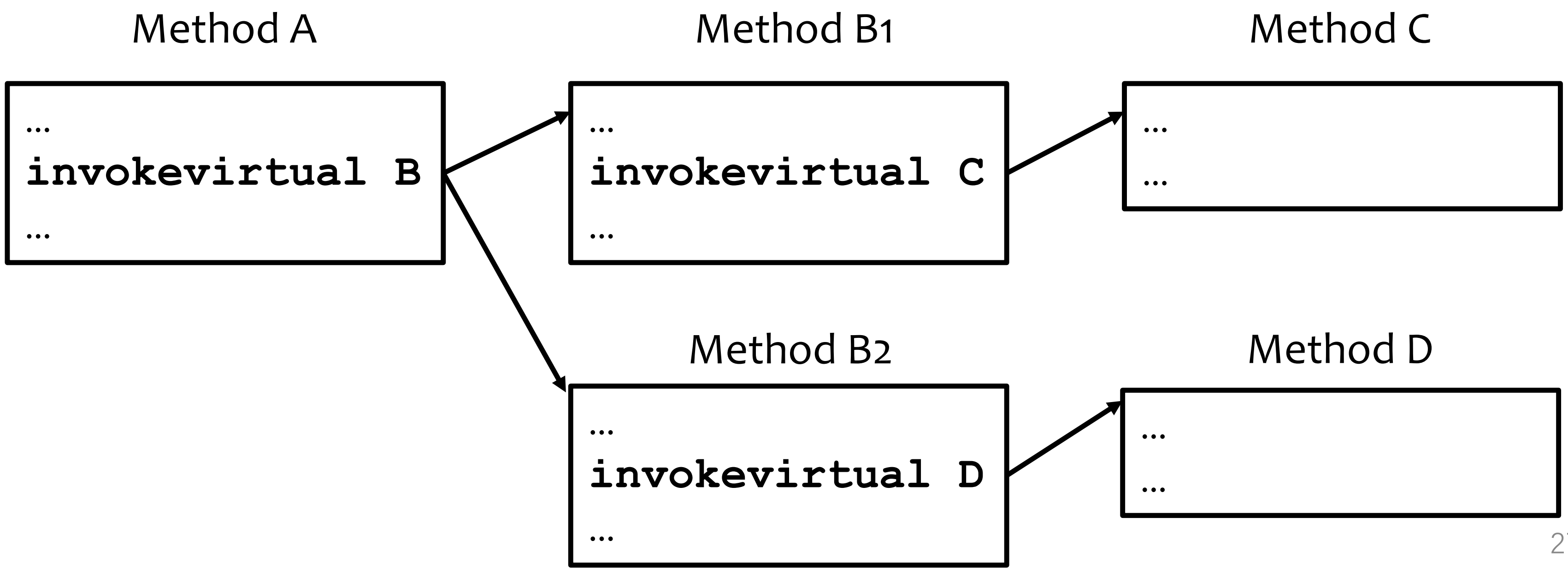
- An example: inlining
- Choice 3: inline all



```
native code A...
native code B...
native code C...
native code B...
native code A...
```

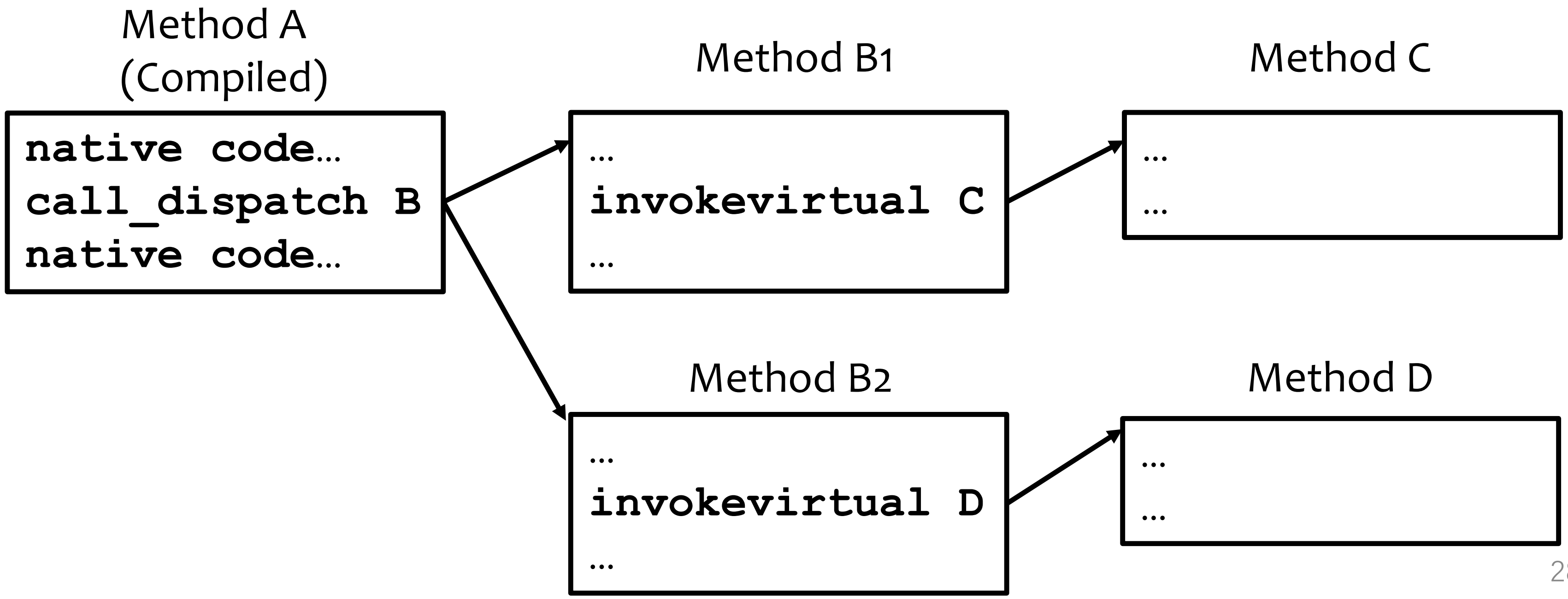
JIT: Aggressive Or Conservative?

- **What about polymorphism?**
  - More choices!



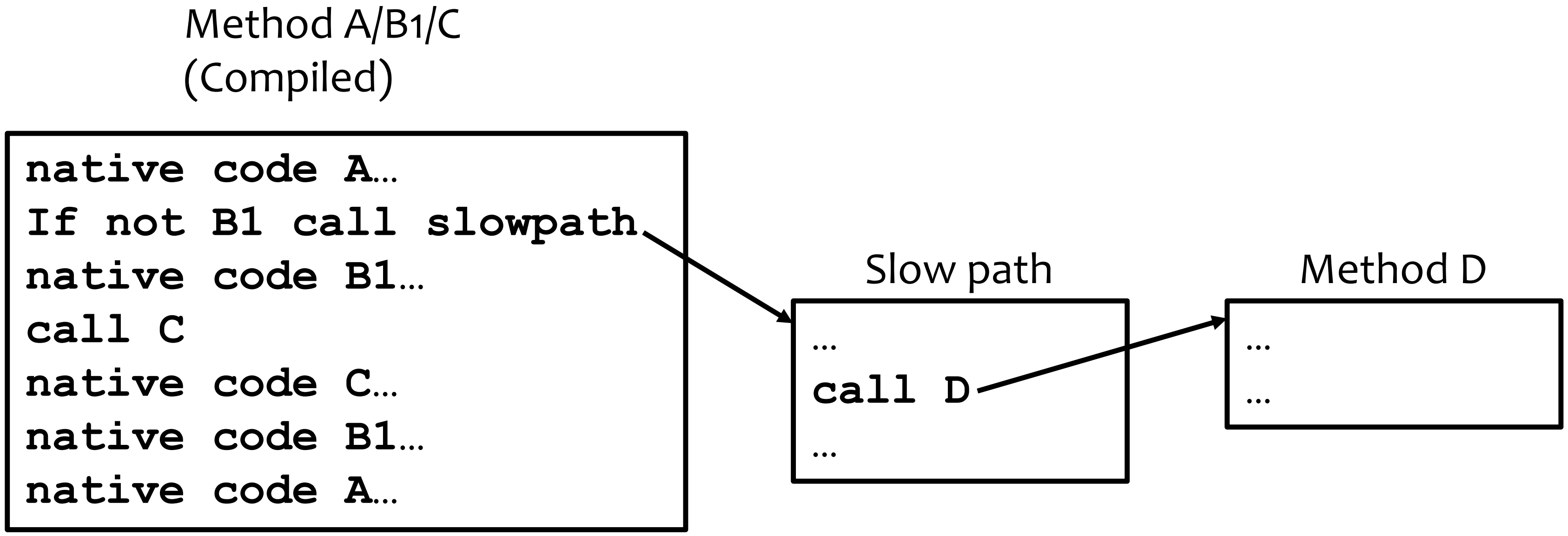
JIT: Aggressive Or Conservative?

- **Choice 1: compile A only**
  - Suitable for all cases where all methods are hot



JIT: Aggressive Or Conservative?

- **Choice 2: compile only for one path**
  - Suitable for cases where only one path is hot



```
native code A…
If not B1 call slowpath
native code B1…
call C
native code C…
native code B1…
native code A…
```

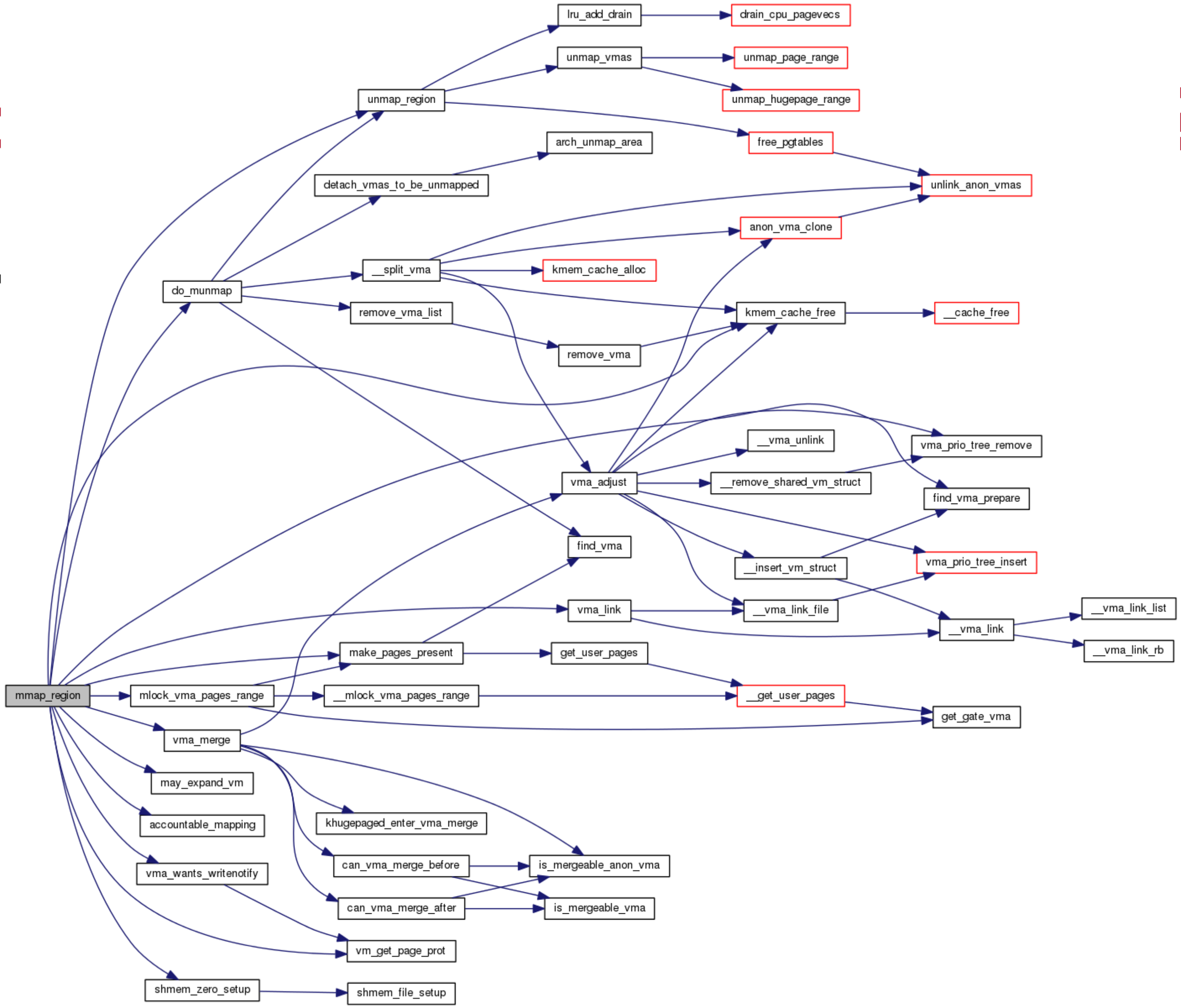
```
…
call D
…
```

JIT: Aggressive Or Conservative?

- The real call graph can be very complex…

JIT ?

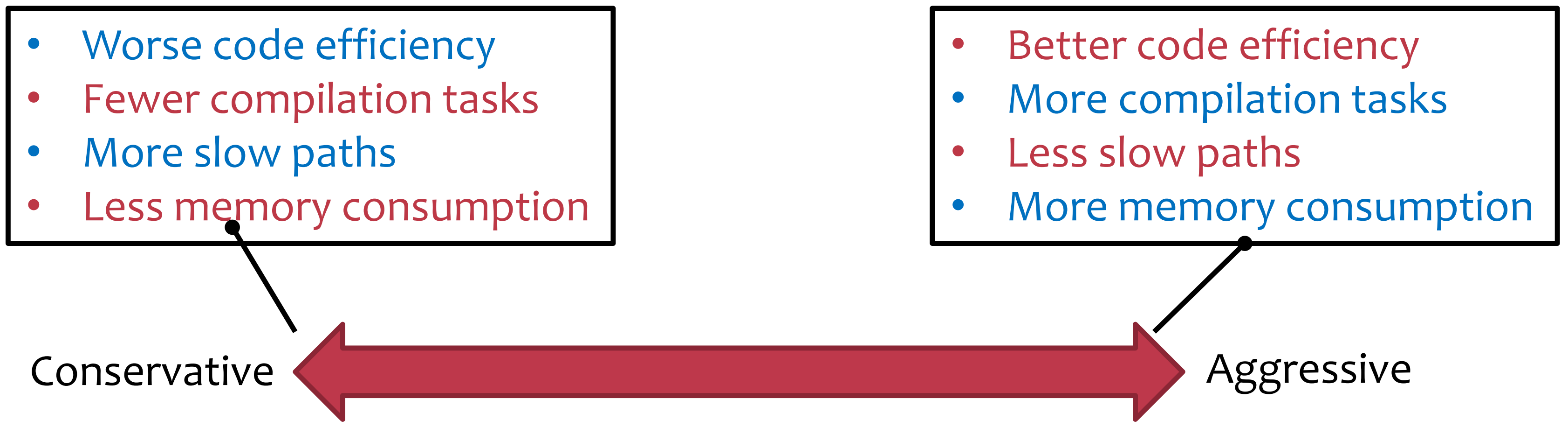
- Th



mmap_region
do_munmap
unmap_region
lru_add_drain
drain_cpu_pagevecs
unmap_vmas
unmap_page_range
unmap_hugepage_range
free_pgtables
unlink_anon_vmas
arch_unmap_area
detach_vmas_to_be_unmapped
anon_vma_clone
__split_vma
kmem_cache_alloc
kmem_cache_free
__cache_free
remove_vma_list
remove_vma
__vma_unlink
vma_prio_tree_remove
vma_adjust
__remove_shared_vm_struct
find_vma_prepare
find_vma
__insert_vm_struct
vma_prio_tree_insert
vma_link
__vma_link_file
__vma_link
__vma_link_list
__vma_link_rb
make_pages_present
get_user_pages
mlock_vma_pages_range
__mlock_vma_pages_range
__get_user_pages
get_gate_vma
vma_merge
may_expand_vm
khugepaged_enter_vma_merge
accountable_mapping
can_vma_merge_before
is_mergeable_anon_vma
vma_wants_writenotify
can_vma_merge_after
is_mergeable_vma
vm_get_page_prot
shmem_zero_setup
shmem_file_setup

JIT: Aggressive Or Conservative?

- **Summary: A tradeoff between code efficiency and runtime overhead**



- Worse code efficiency
- Fewer compilation tasks
- More slow paths
- Less memory consumption

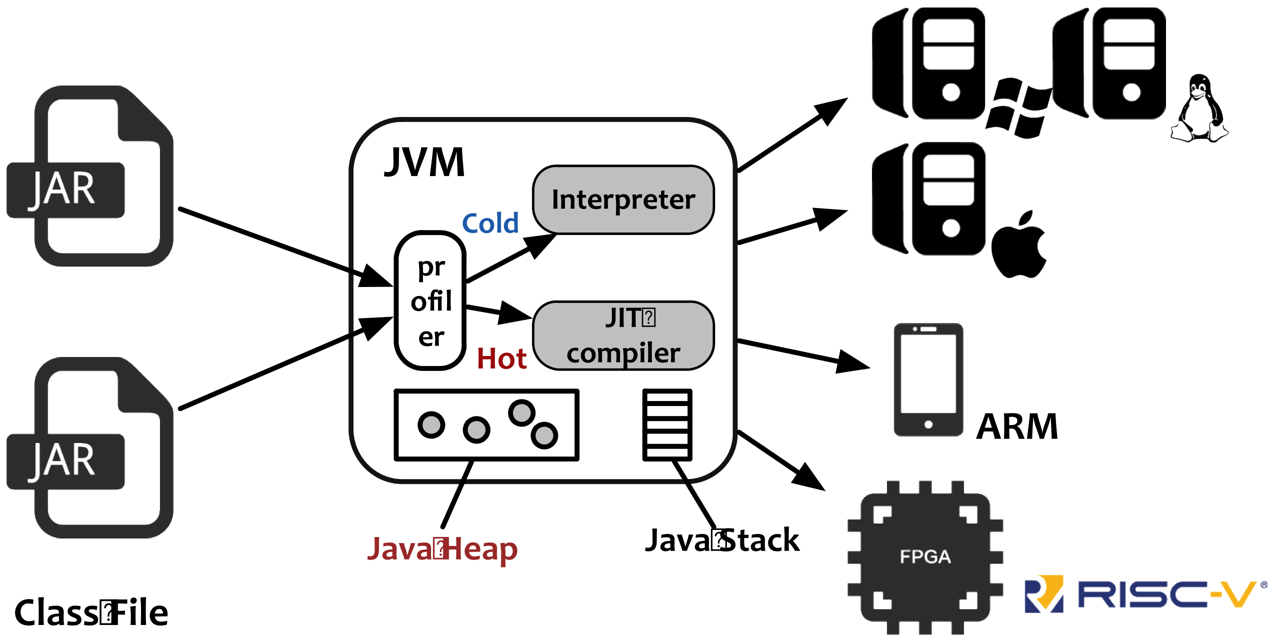
- Better code efficiency
- More compilation tasks
- Less slow paths
- More memory consumption

MEMORY MANAGEMENT

-- aka garbage collection

Where is data?

- Java objects are organized in *Java Heap*



"Free"-free Memory Management

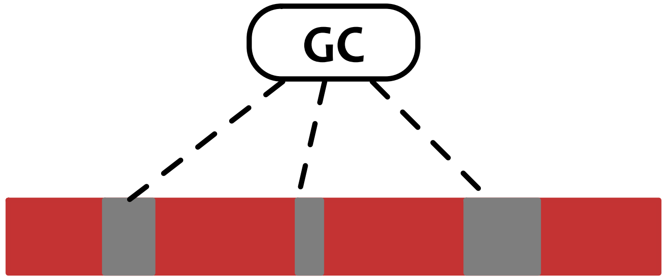
- **No *free()* is required in Java**
  - Only allocation (new) is required

- **JVM reclaims unused memory by garbage collection (GC)**

"Free"-free Memory Management

- **No *free()* is required in Java**
  - Only allocation (new) is required

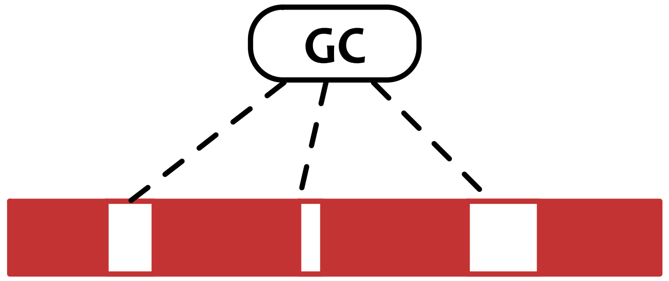
- **JVM reclaims unused memory by garbage collection (GC)**
  - Recognize live/dead objects



"Free"-free Memory Management

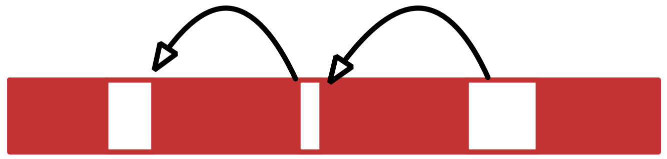
- **No *free()* is required in Java**
  - Only allocation (new) is required

- **JVM reclaims unused memory by garbage collection (GC)**
  - Recognize live/dead objects
  - Manage heap space



Heap Layout 1: Free List

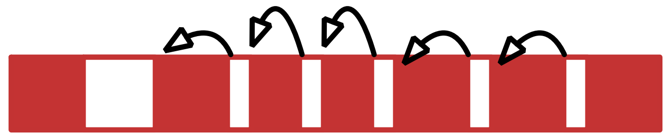
- Common and Straightforward
- Not welcomed in Java



Heap Layout 1: Free List

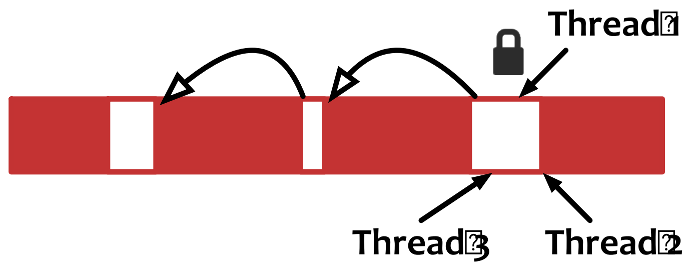
- **Common and Straightforward**

- **Not welcomed in Java**
  - Fragmentations



Heap Layout 1: Free List

- **Common and Straightforward**
- **Not welcomed in Java**
  - Fragmentations
  - Multi-threaded contention

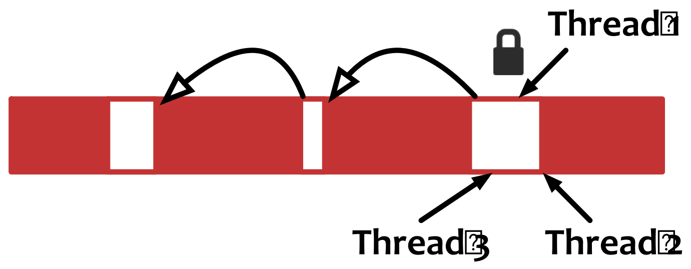


Heap Layout 1: Free List

- **Common and Straightforward**

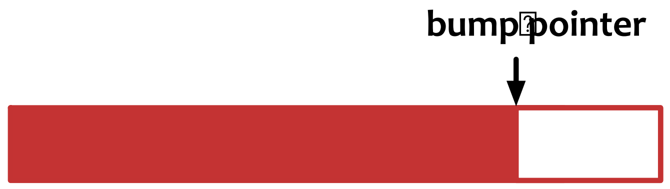
- **Not welcomed in Java**
  - Fragmentations
  - Multi-threaded contention

Java prefers fast allocation!



Heap Layout 2: Contiguous Space

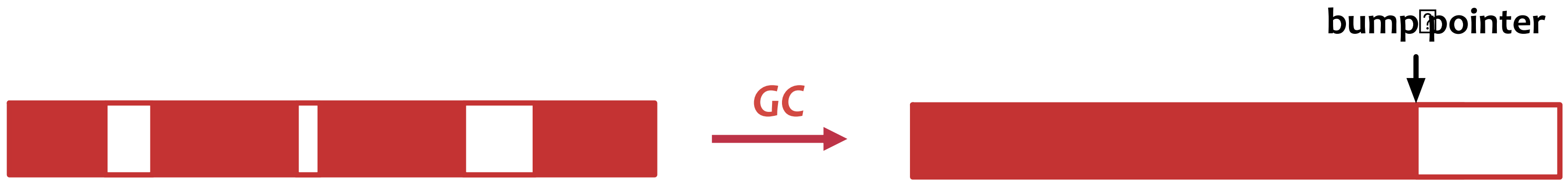
- **Free space is always contiguous**
  - A bump pointer to mark how much has been used
  - Allocation: Lock-free atomic instructions (CAS)



Heap Layout 2: Contiguous Space

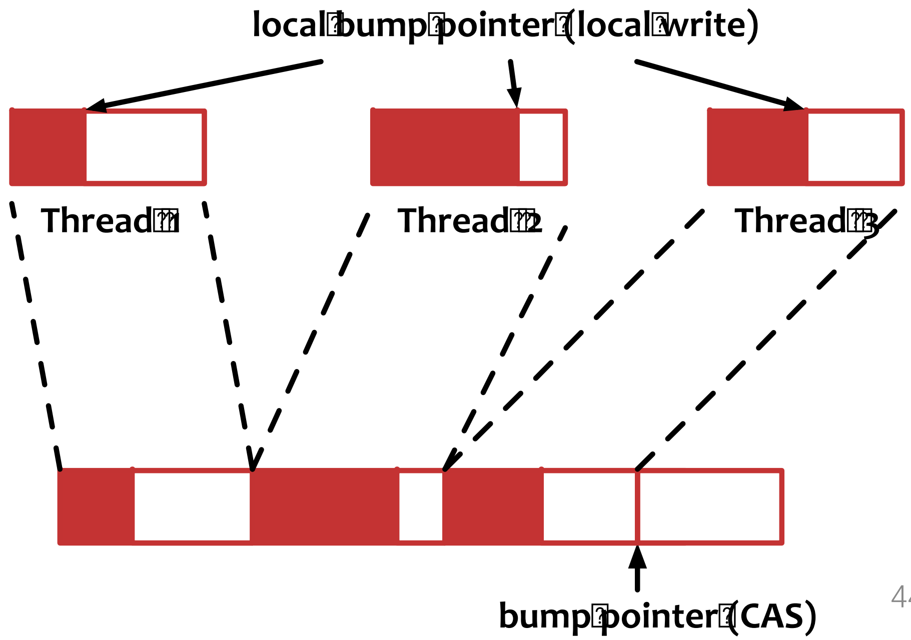
- **Free space is always contiguous**
  - A bump pointer to mark how much has been used
  - Fast allocation: Lock-free atomic instructions (CAS)

- **GC is responsible for compacting live objects**



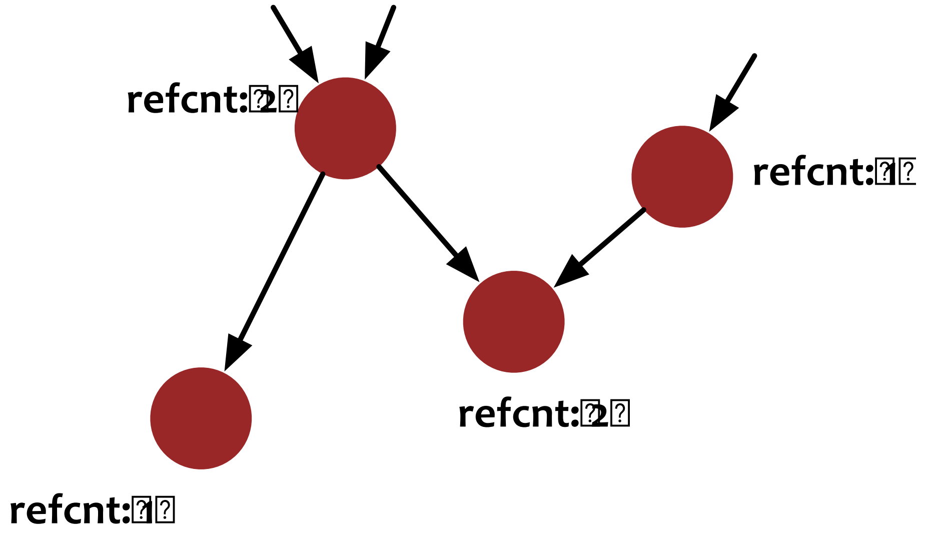
Heap Layout 2: Contiguous Space

- **Even faster allocation: local heap**
  - Allocate a large portion with CAS
  - Then allocate locally



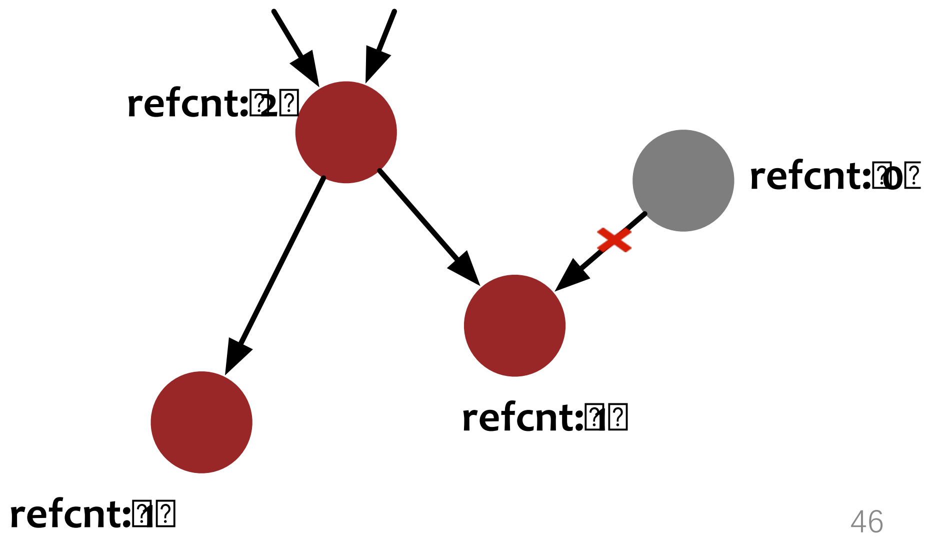
GC Algorithm 1: Reference Counting

- Counting the in-references for each objects



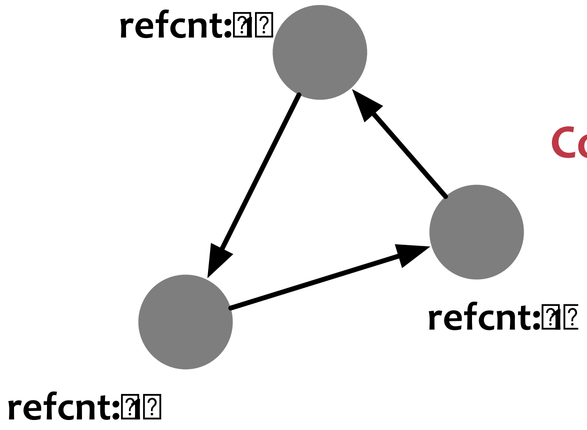
GC Algorithm 1: Reference Counting

- **Counting the in-references for each objects**
  - Immediate reclamation: reclaim when refcnt is 0
  - Used by some prototype systems (NVHeap, ASPLOS11)



GC Algorithm 1: Reference Counting

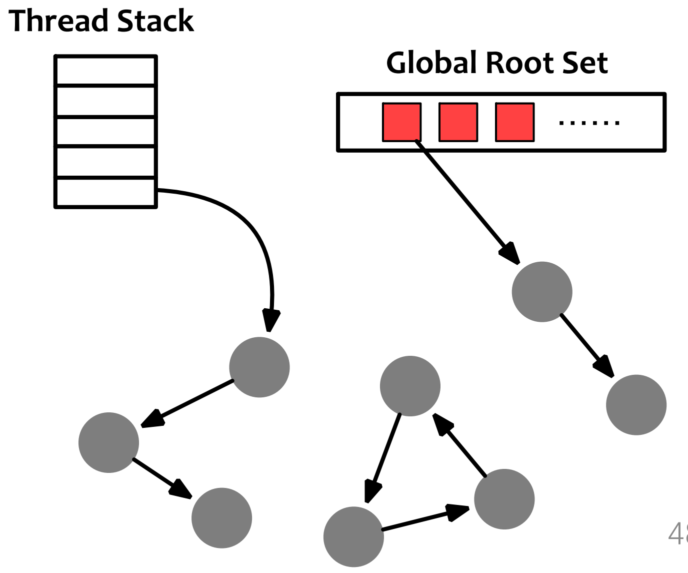
- **However, still no welcomed in Java**
  - Performance: 30%+ worse than others (Shahriyar, OOPSLA13)
  - Tightly integrated with free lists
  - Failing to resolve cycles



Collector: all objects are alive!

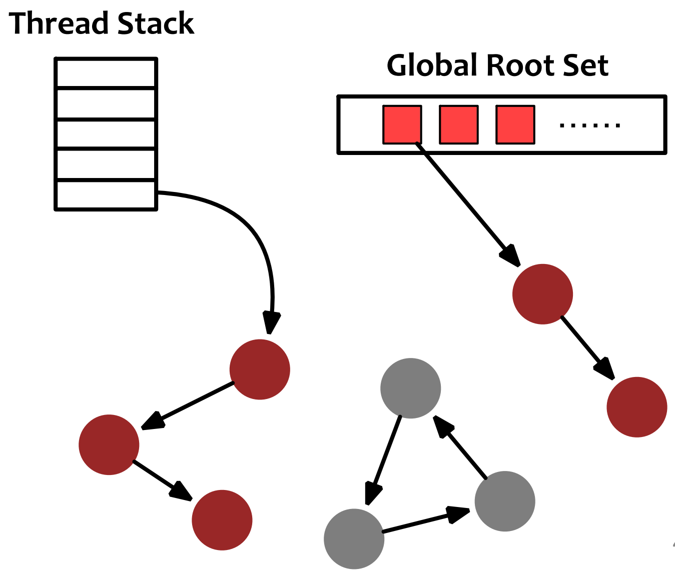
GC Algorithm 2: Tracing

- **Identifying live objects by traversing object graphs**
  - The start points are called "roots"



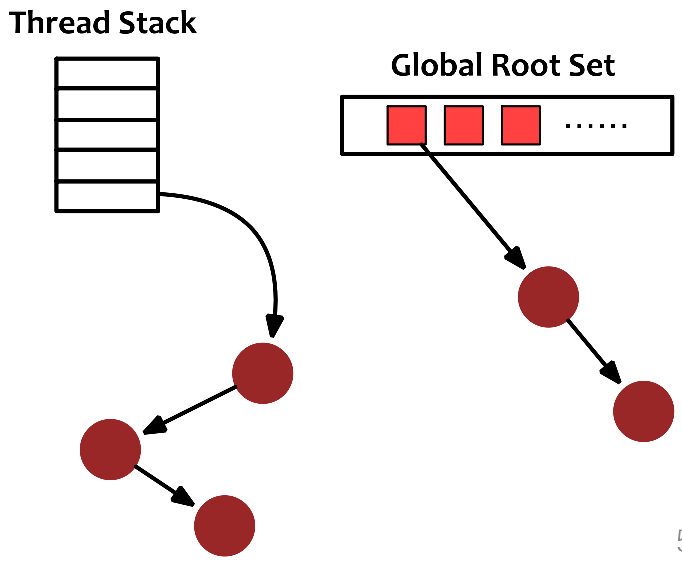
GC Algorithm 2: Tracing

- **Identifying live objects by traversing object graphs**
  - The start points are called "roots"



GC Algorithm 2: Tracing

- **Identifying live objects by traversing object graphs**
  - The start points are called "roots”
  - Solve the cycle problem



Tracing GC is Popular

- **Can be integrated with different layout**
  - Free-list: Mark-Sweep (Boehm GC)
  - Contiguous: Mark-Copy (PS, G1, Shenandoah…)

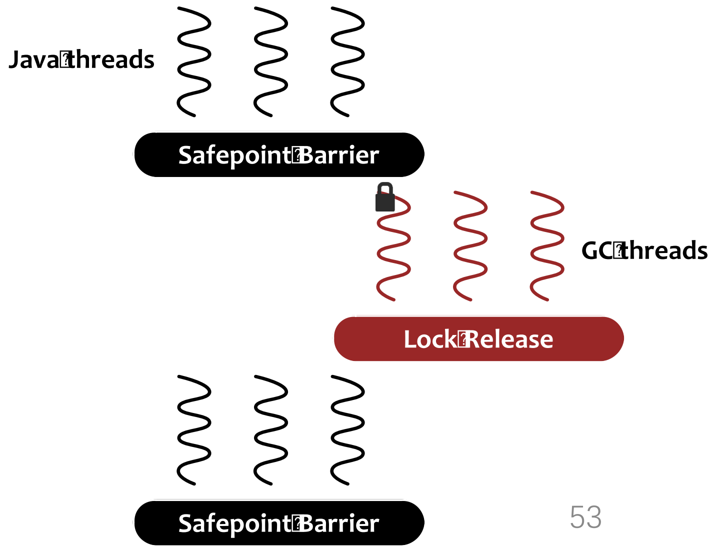
Case Study: PS

- **Design goal: high GC throughput**
  - Stop-the-world: Java threads must be paused during GC
  - Task-based parallelism: dividing collections into tasks

- **Also consider latency**
  - Generational

Stop-The-World

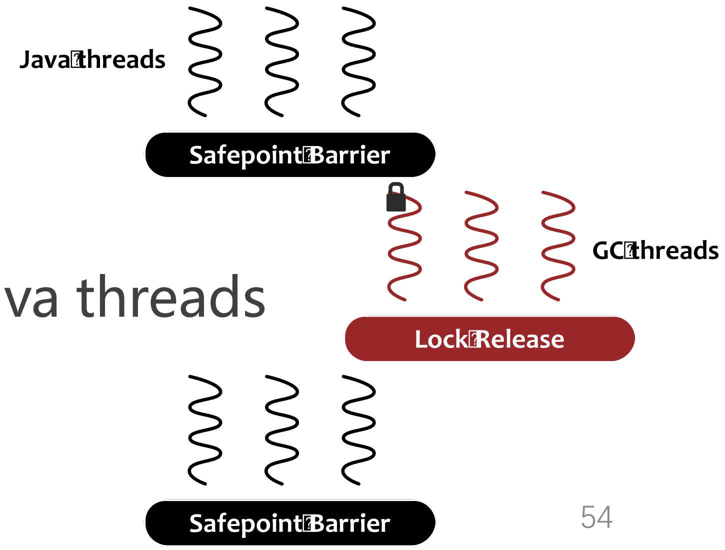
- **JVM leverages *safepoint* to pause all Java threads**
  - Java threads queue up for a lock held by GC threads



Stop-The-World

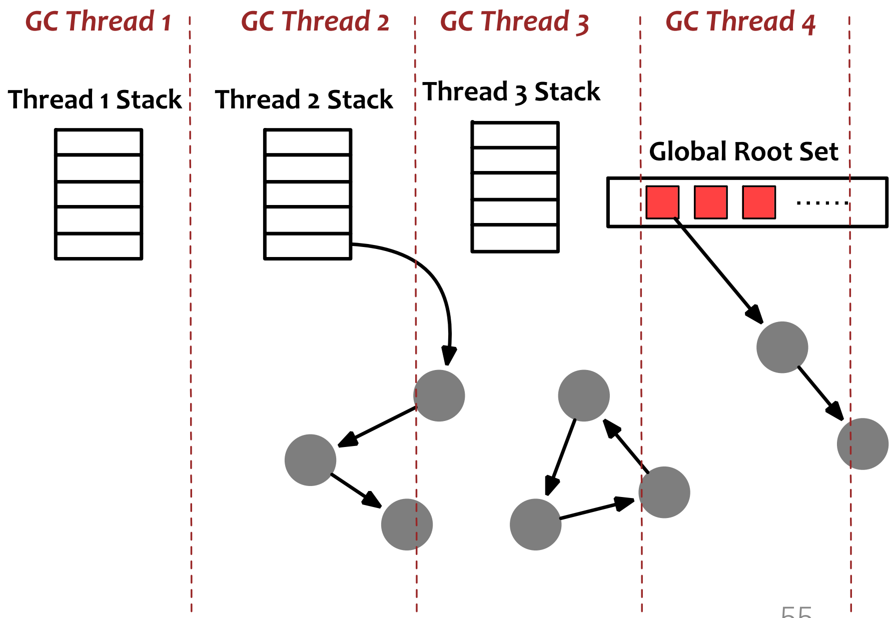
- **JVM leverages *safepoint* to pause all Java threads**
  - Java threads queue up for a lock held by GC threads

- **Aiming at high GC throughput**
  - CPU monopolized by GC threads
  - No coordination between GC and Java threads



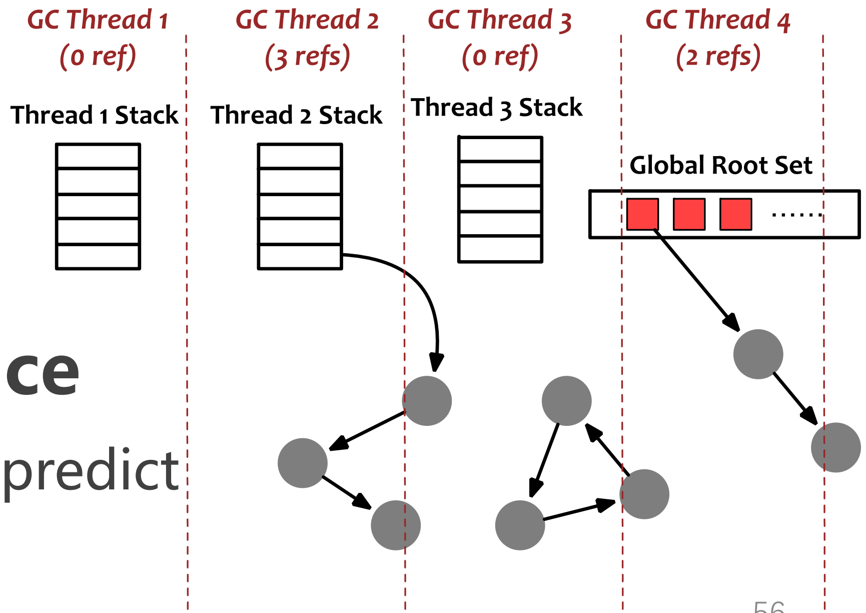
Task-based Parallelism

- **Dividing *roots* into different tasks**
  - Each GC threads can work independently



Task-based Parallelism

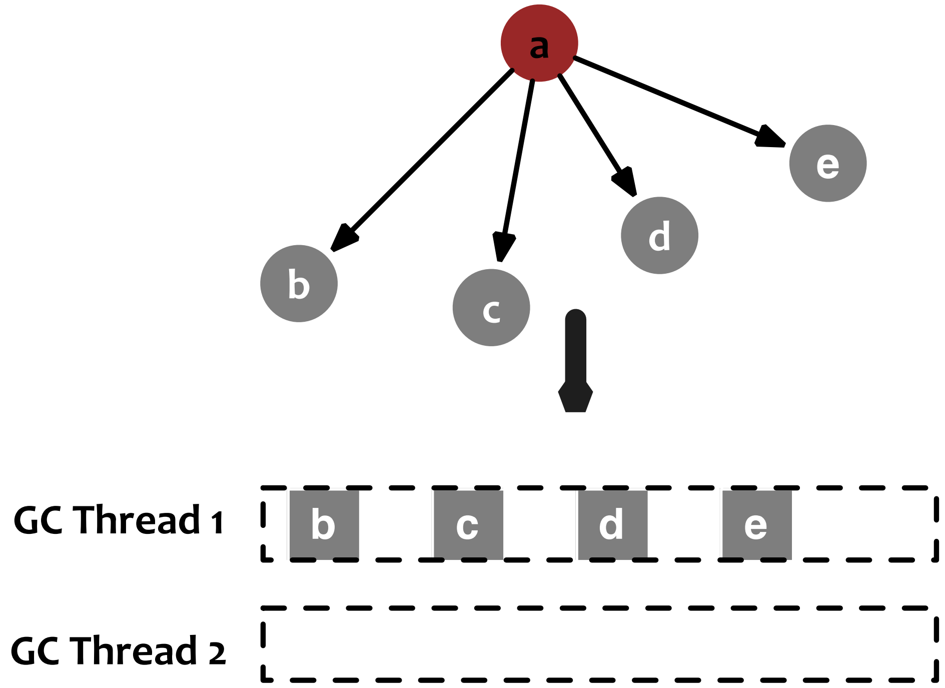
- **Dividing *roots* into different tasks**
  - Each GC threads can work independently



- **Problem: Load-imbalance**
  - Workload in graph is hard to predict

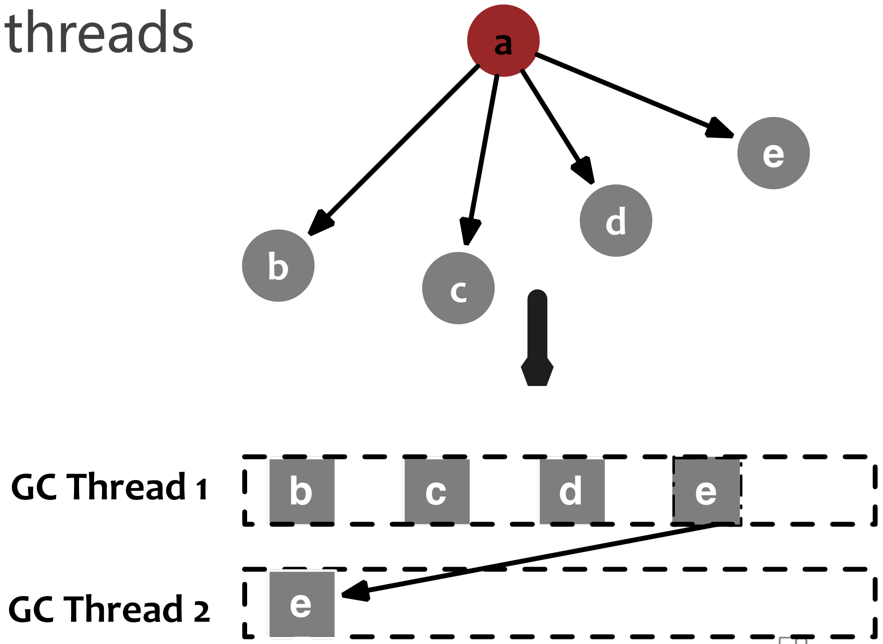
Task-based Parallelism

- **Solution: work-stealing (finer-grained tasks)**
  - Abstracting references into tasks



Task-based Parallelism

- **Solution: work-stealing (finer-grained tasks)**
  - Abstracting references into tasks
  - Allow task stealing between threads

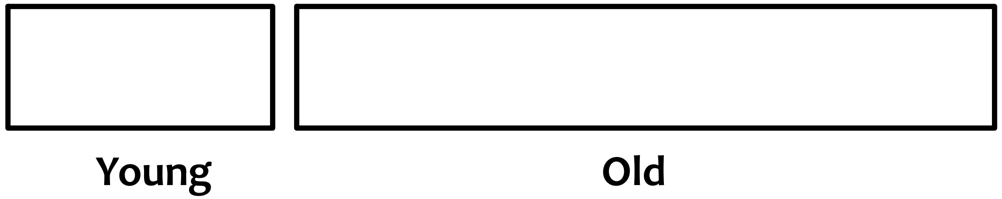


Latency Consideration

- **GC is usually triggered when memory is exhausted**
  - Pause time is proportional with live object size
  - 400MB live object size -> 100 ms pause
  - Not scalable: what about 10GB?

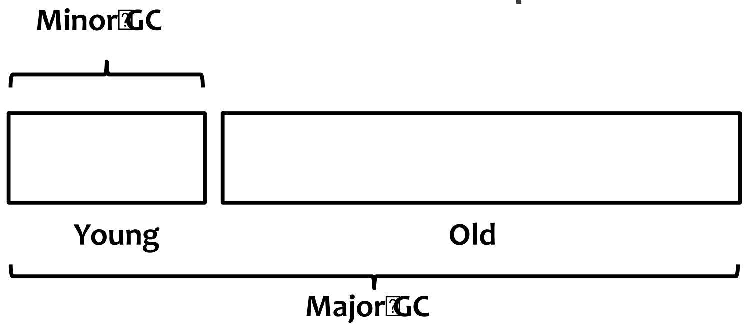
Solution: Generational GC

- **Dividing Java heap into multiple spaces**
  - In PS it has two spaces: *young* and *old*
  - The size of young gen can be small and fixed



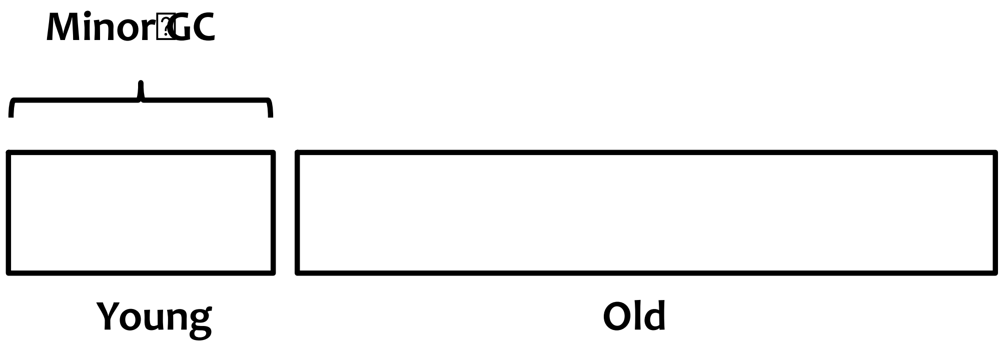
Solution: Generational GC

- **Dividing Java heap into multiple spaces**
  - In PS it has two spaces: *young* and *old*
  - The size of young gen can be small and fixed
- **GC is also two-fold**
  - Minor GC: only collecting young-space (fast and frequent)
  - Major GC: collecting both spaces (slow but infrequent)



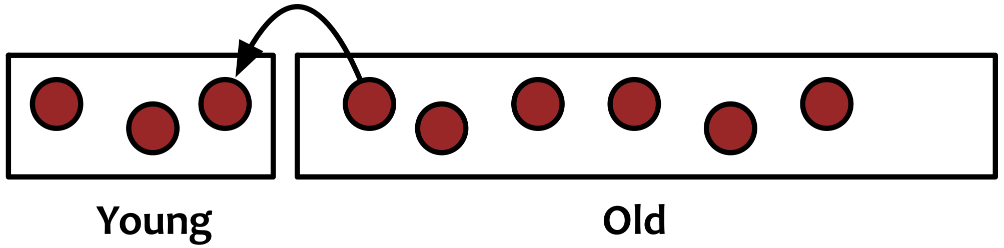
Minor GC: Collecting Young Only

- **Based on "generational hypothesis"**
  - Most objects have short life spans
  - Collecting young-space is efficient for memory reclamation
  - What if the hypothesis does not hold? (Yak, OSDI16)



Problem: Cross-Space References

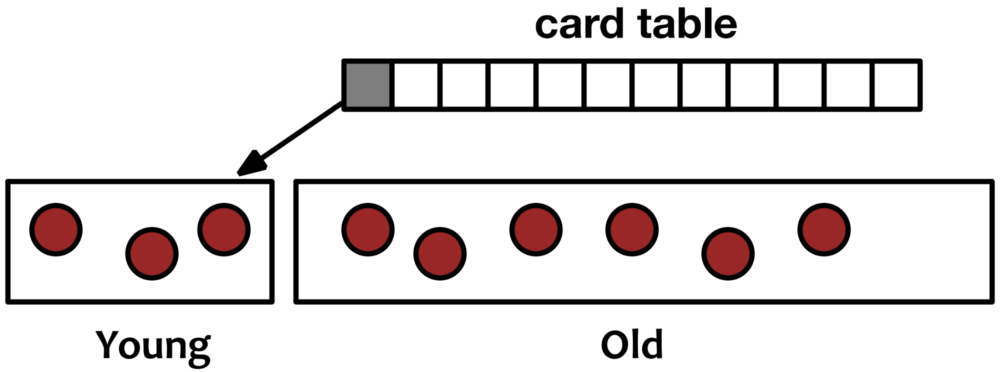
- **References can point from old-space to young-space**
  - They should be treated as roots during minor GC
- **How to identify cross-space references?**
  - Scanning the whole old space: why do we need a minor GC?
  - Remembering all references: large overhead



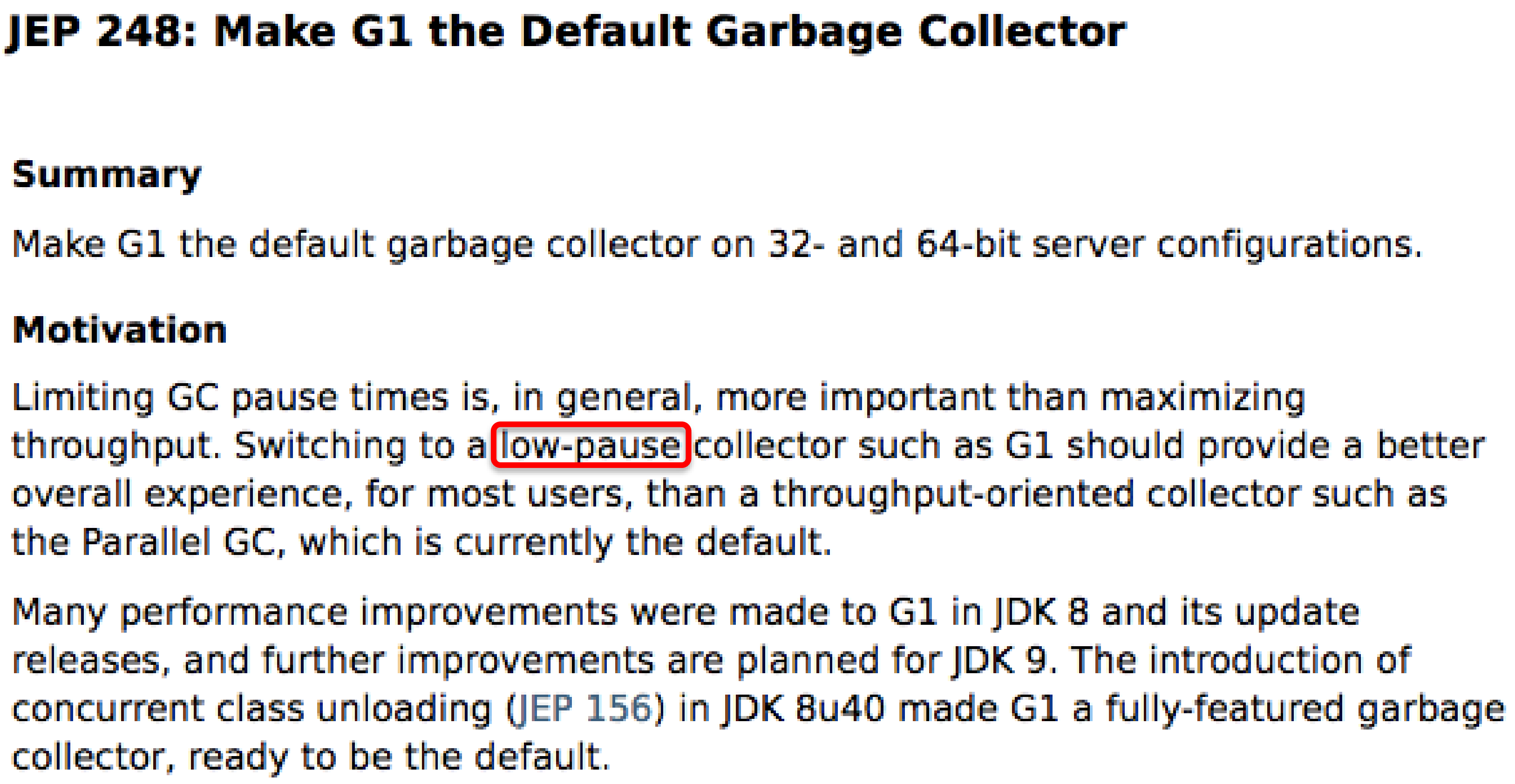
Solution: Card Table

- **Dividing old-space into many *regions***
  - Using 1 bit (card) in a table for each region

- **When a cross-space write happens, dirty the card**
  - Scanning dirty cards only during minor GC



Case Study: G1



JEP 248: Make G1 the Default Garbage Collector

Summary

Make G1 the default garbage collector on 32- and 64-bit server configurations.

Motivation

Limiting GC pause times is, in general, more important than maximizing throughput. Switching to a low-pause collector such as G1 should provide a better overall experience, for most users, than a throughput-oriented collector such as the Parallel GC, which is currently the default.

Many performance improvements were made to G1 in JDK 8 and its update releases, and further improvements are planned for JDK 9. The introduction of concurrent class unloading (JEP 156) in JDK 8u40 made G1 a fully-featured garbage collector, ready to be the default.

Design Highlight of G1

- **Controllable GC pauses**
  - Soft limits
  - Region-based heap layout
  - Mixed GC

- **(Partially) concurrent execution**
  - Concurrent marking

Controlling GC Pauses: Soft Limit

- **G1 allows applications to set a maximum GC pause time**
  - e.g. –XX:MaxGCPauseMillis=50

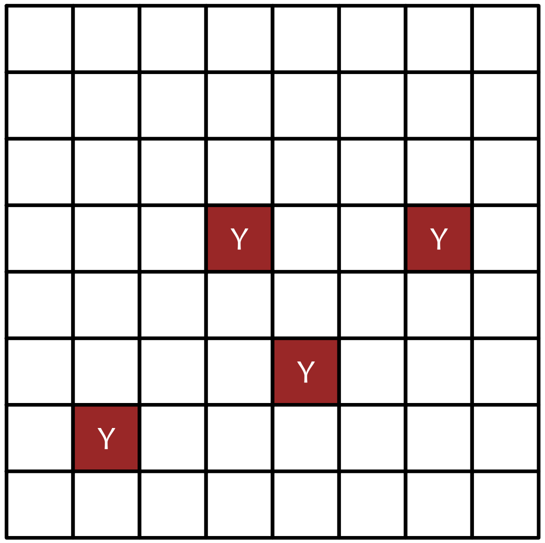
- **G1 will adjust the heap layout to meet the limit**
  - The layout must be adjustable and flexible
  - Two spaces are not enough!

Controlling GC Pauses: Soft Limit

- **G1 allows applications to set a maximum GC pause time**
  - e.g. –XX:MaxGCPauseMillis=50
- **G1 will adjust the heap layout to meet the limit**
  - The layout must be adjustable and flexible
  - Two spaces are not enough!

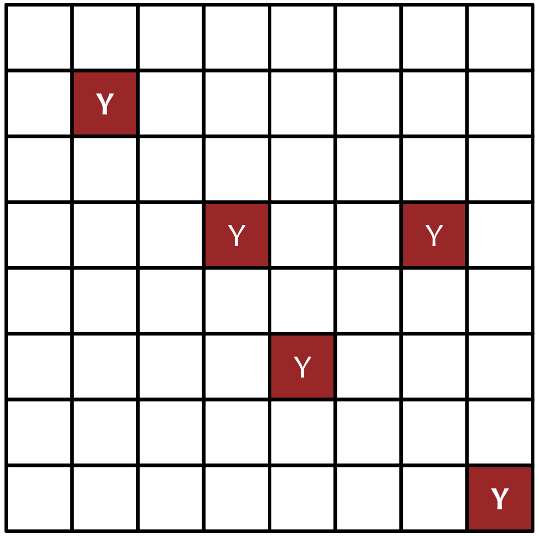
Region-based Heap Layout

- **Dividing the whole heap space into many *regions***
  - Young-space now can be segregated



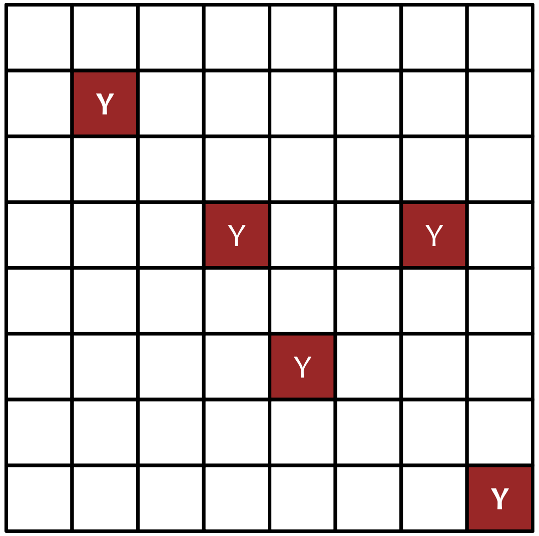
Region-based Heap Layout

- **Dividing the whole heap space into many *regions***
  - Young-space now can be segregated
  - Easy to enlarge/shrink



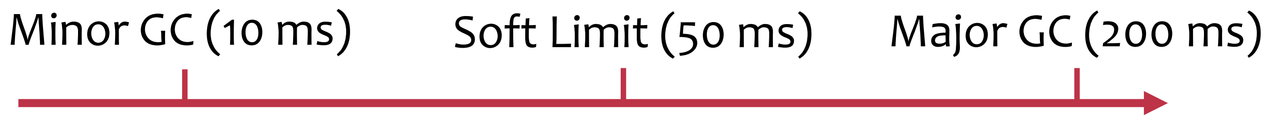
Region-based Heap Layout

- **Dividing the whole heap space into many *regions***
  - Young-space now can be segregated
  - Easy to enlarge/shrink
  - Also designed for **mixed GC**



Mixed GC: *Meeting* the Soft Limit

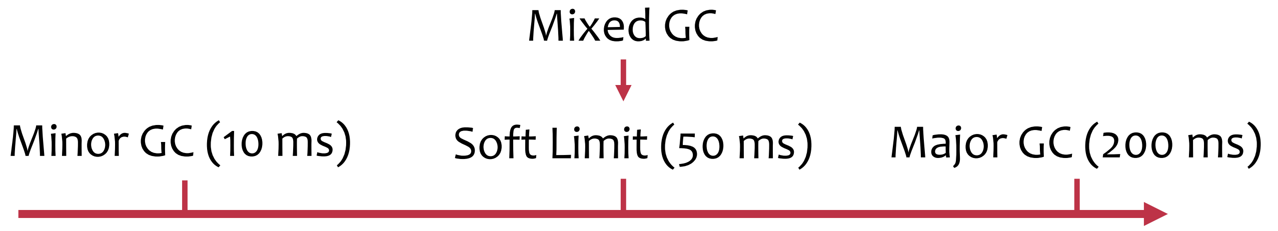
- **Prior GC (like PSGC) only has two inflexible GC algorithms**
  - Minor GC pause: too short
  - Major GC pause: too long



Mixed GC: *Meeting* the Soft Limit

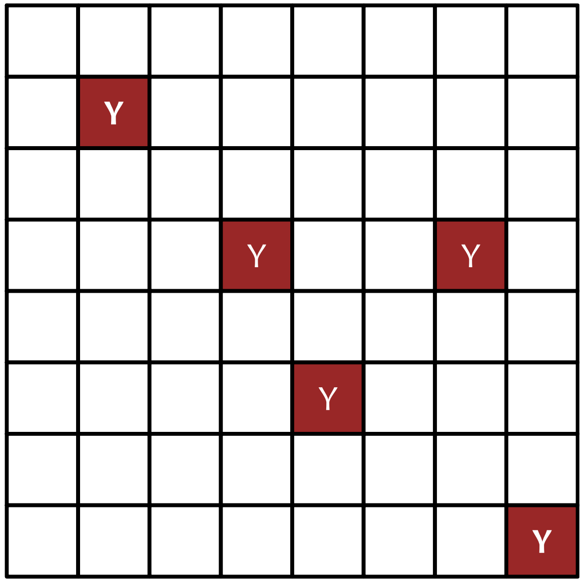
- **Prior GC (like PSGC) only has two inflexible GC algorithms**
  - Minor GC pause: too short
  - Major GC pause: too long

- **Solution: collect young-space and part of old-space**



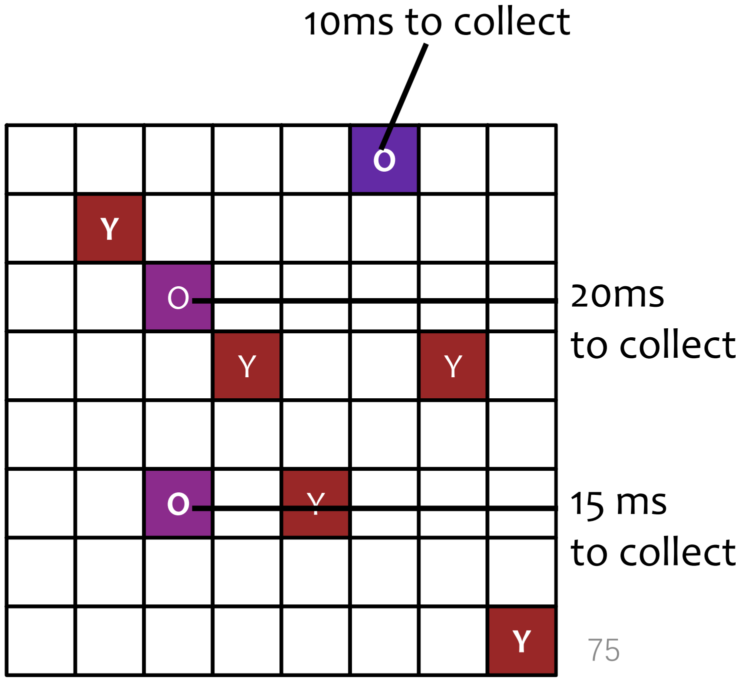
Mixed GC: Collection Set (CSet)

- **Mixed GC should construct a CSet to reach a close-to-soft-limit GC pause**
  - Including all young regions



Mixed GC: Collection Set (CSet)

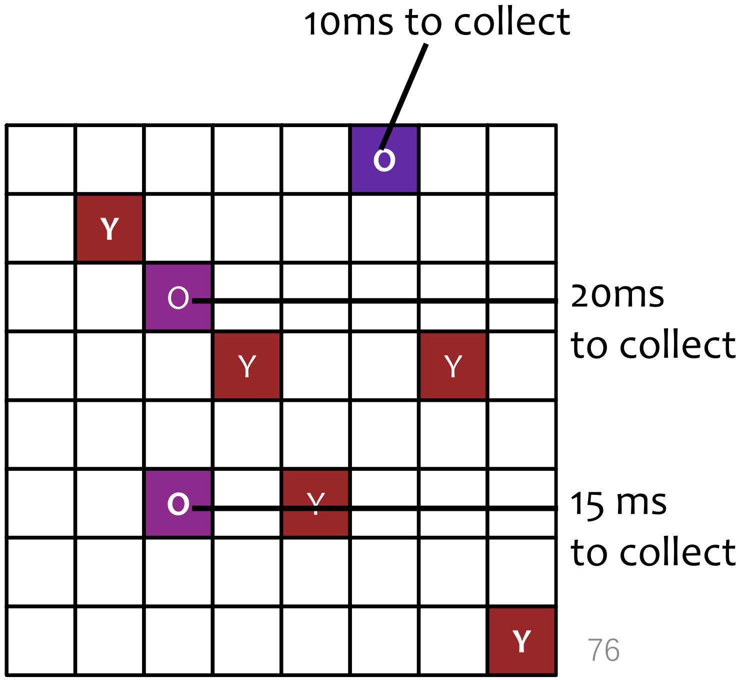
- **Mixed GC should construct a CSet to reach a close-to-soft-limit GC pause**
  - Including all young regions
  - Adding old regions and estimating the GC pause
  - Until the estimated pause time reaching the soft limit



Mixed GC: Collection Set (CSet)

- **Mixed GC should construct a CSet to reach a close-to-soft-limit GC pause**
  - Including all young regions
  - Adding old regions and estimating the GC pause
  - Until the estimated pause time reaching the soft limit

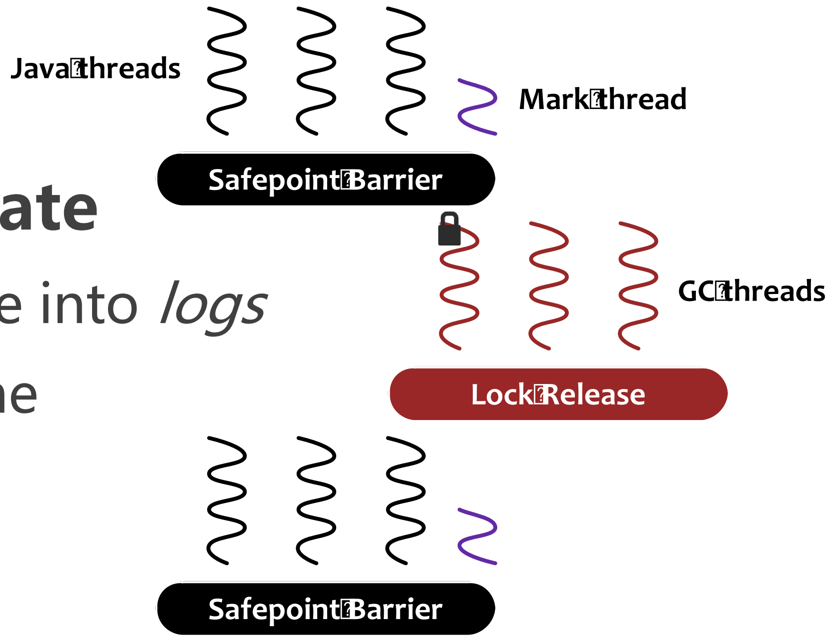
Problem: which regions to collect?



Concurrent Marking

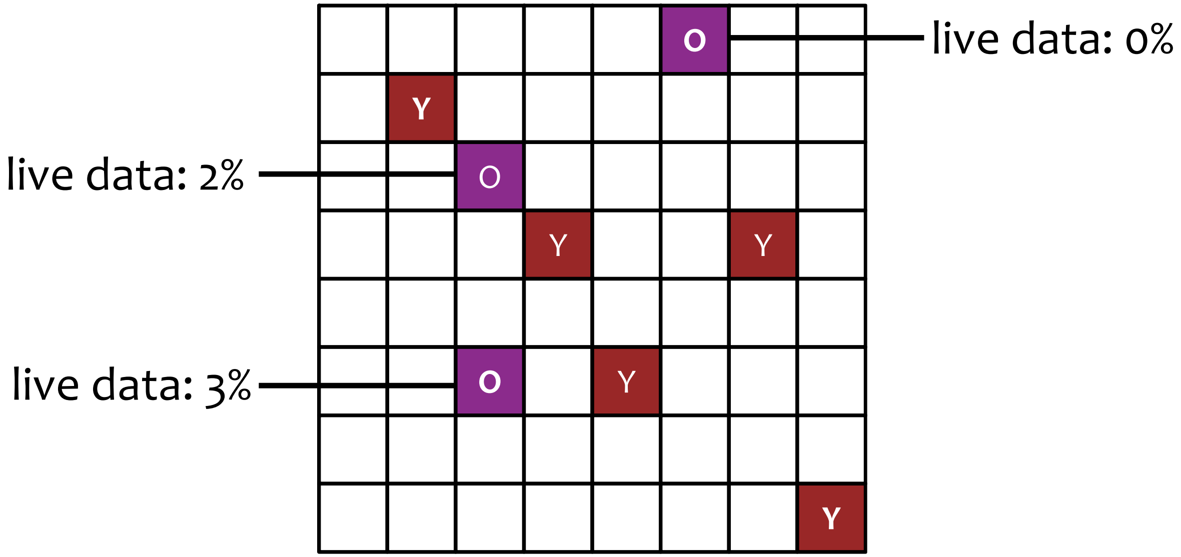
- **Before STW collection, mark live objects concurrently**

- **Handling concurrent update**
  - Java threads will write update into *logs*
  - Marking threads will consume the logs periodically



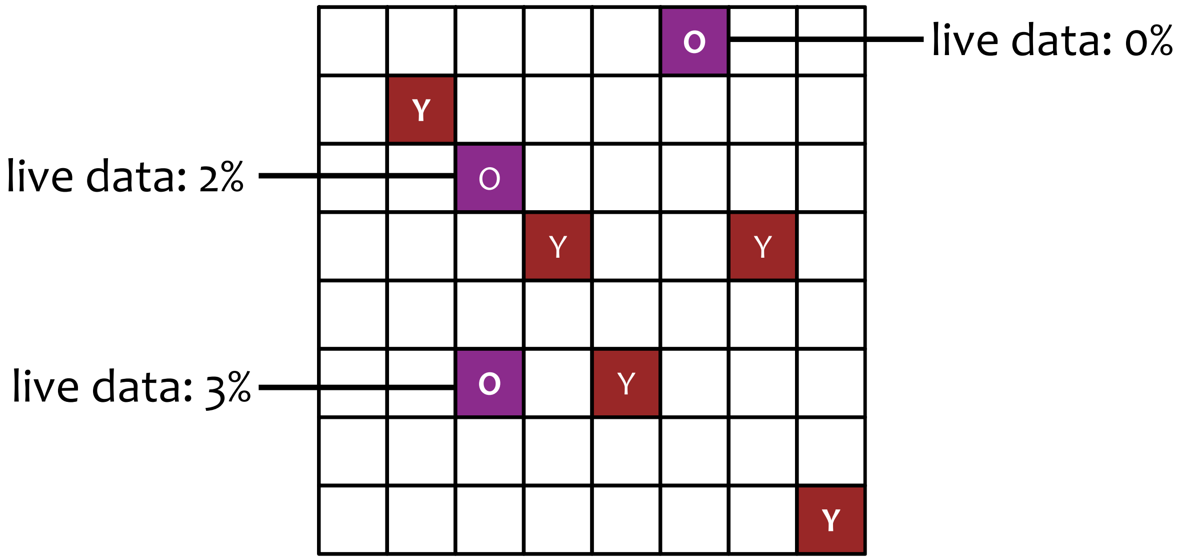
Live Data Profiling

- **After concurrent marking, count per-region live object size**
  - Choose regions with the least live object for collection



Live Data Profiling

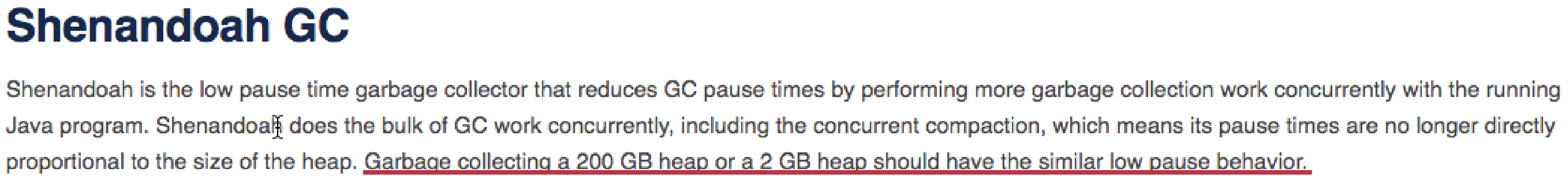
- **After concurrent marking, count per-region live object size**
  - Choose regions with the least live object for collection



Think about the name: "Garbage First"

Case Study: Shenandoah

- A ultra-low pause time GC regardless of heap size

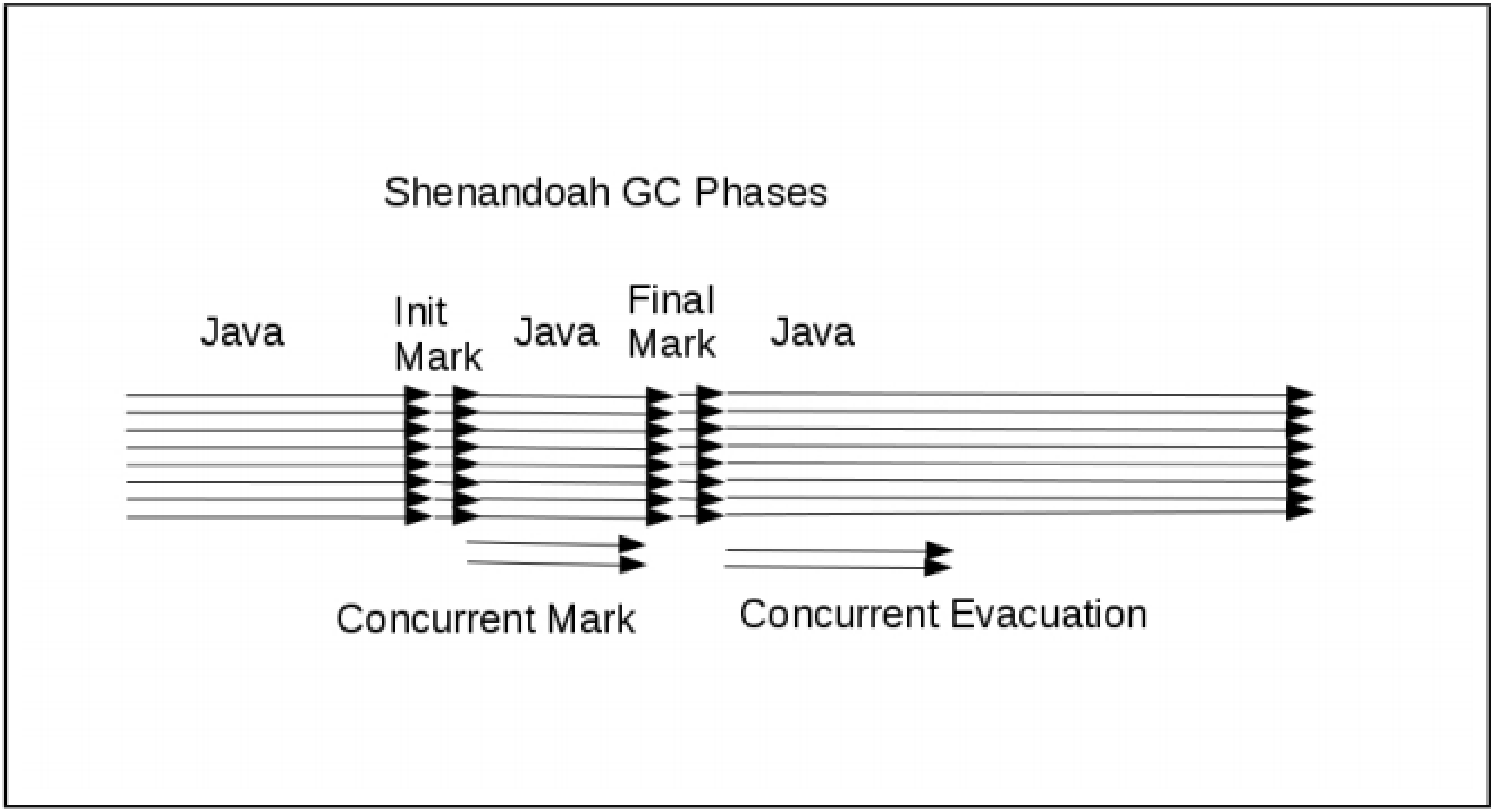


Shenandoah GC

Shenandoah is the low pause time garbage collector that reduces GC pause times by performing more garbage collection work concurrently with the running Java program. Shenandoah does the bulk of GC work concurrently, including the concurrent compaction, which means its pause times are no longer directly proportional to the size of the heap. Garbage collecting a 200 GB heap or a 2 GB heap should have the similar low pause behavior.

Workflow of Shenandoah

- **Only init/final mark require pauses**



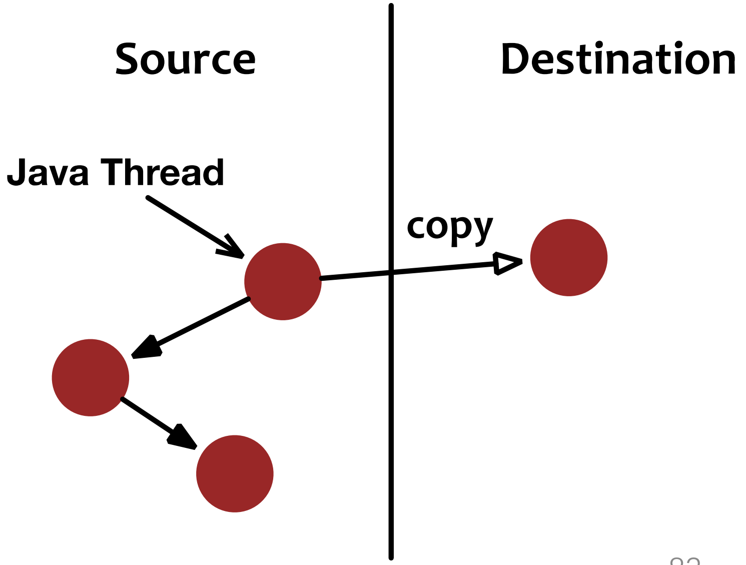
Design Highlight of Shenandoah

- **Concurrent collection**
  - Indirect references

- **Lazy reference update**
  - Piggy-backed with concurrent marking

Concurrent Collection

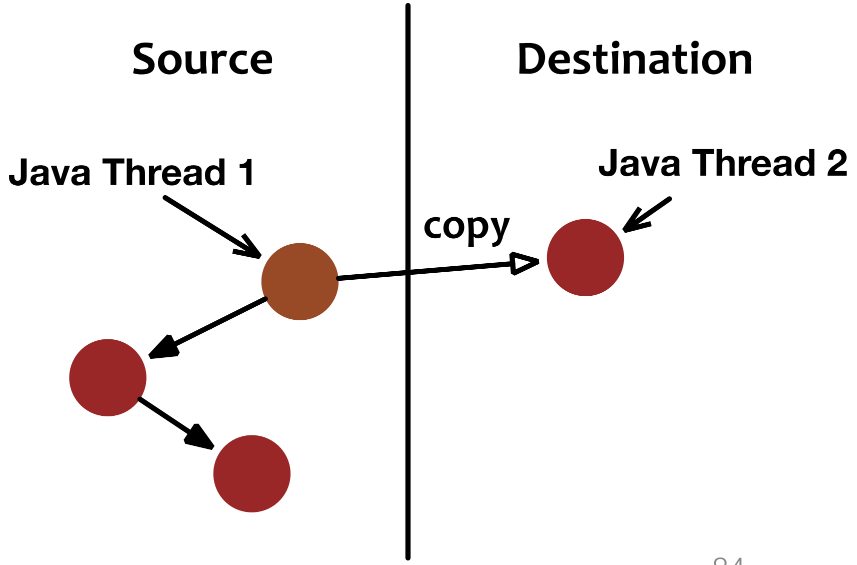
- **GC threads will generate a new copy of objects**
  - Two copies will co-exist



Concurrent Collection

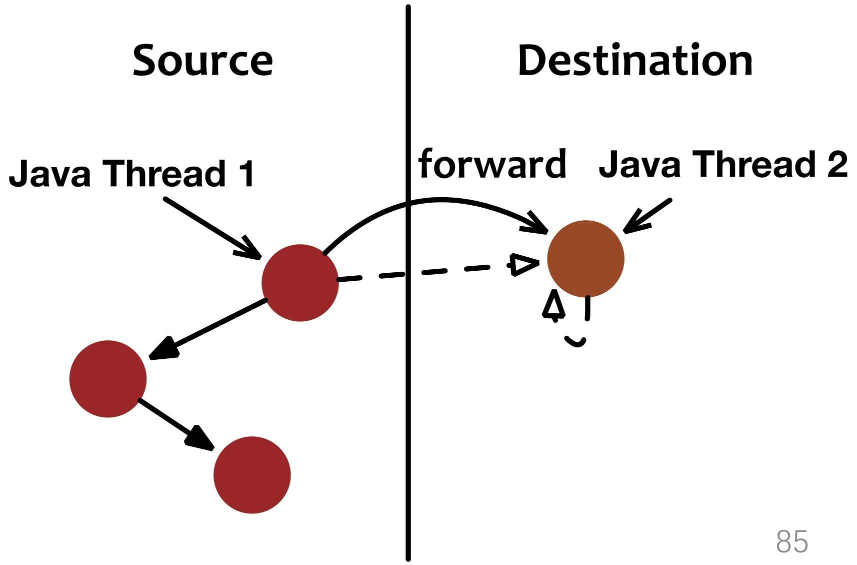
- **GC threads will generate a new copy of objects**
  - Two copies will co-exist

- **Direct update would cause problems**
  - Updates may not be observed



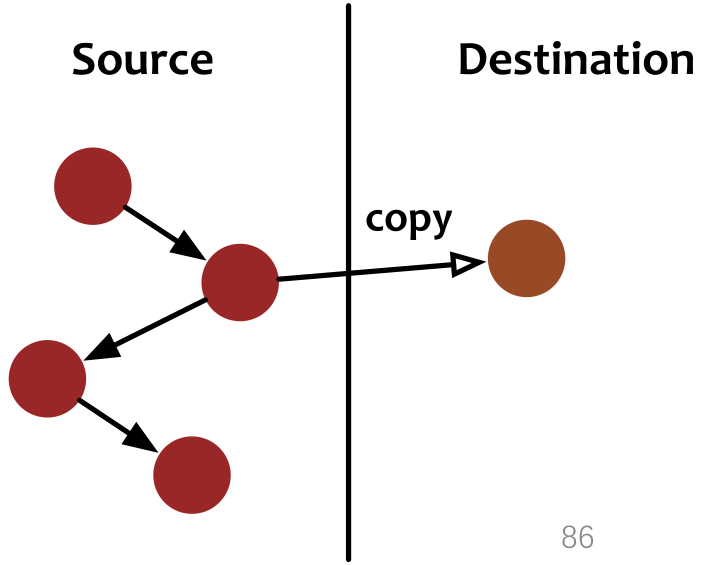
Concurrent Collection

- **Solution: Indirect pointer**
  - Always point to the newest version
  - All read/writes will be forwarded to the newest one



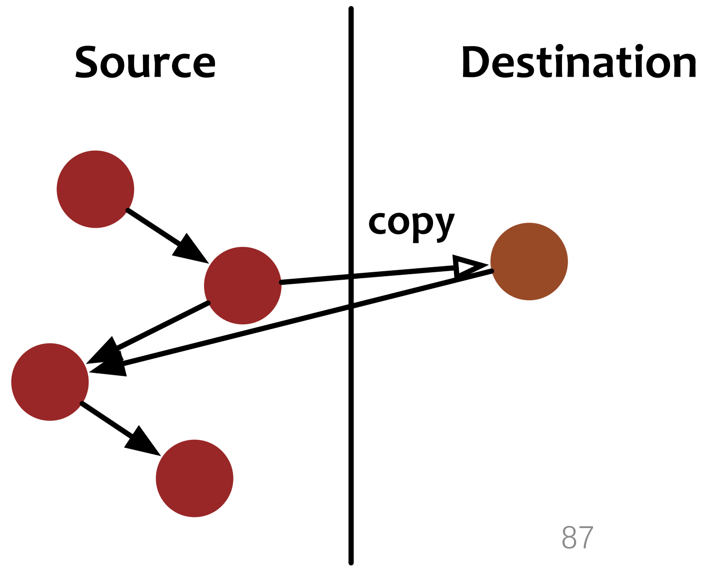
In-Reference Updates

- **Only part of heap space will be scanned and copied (similar to G1)**



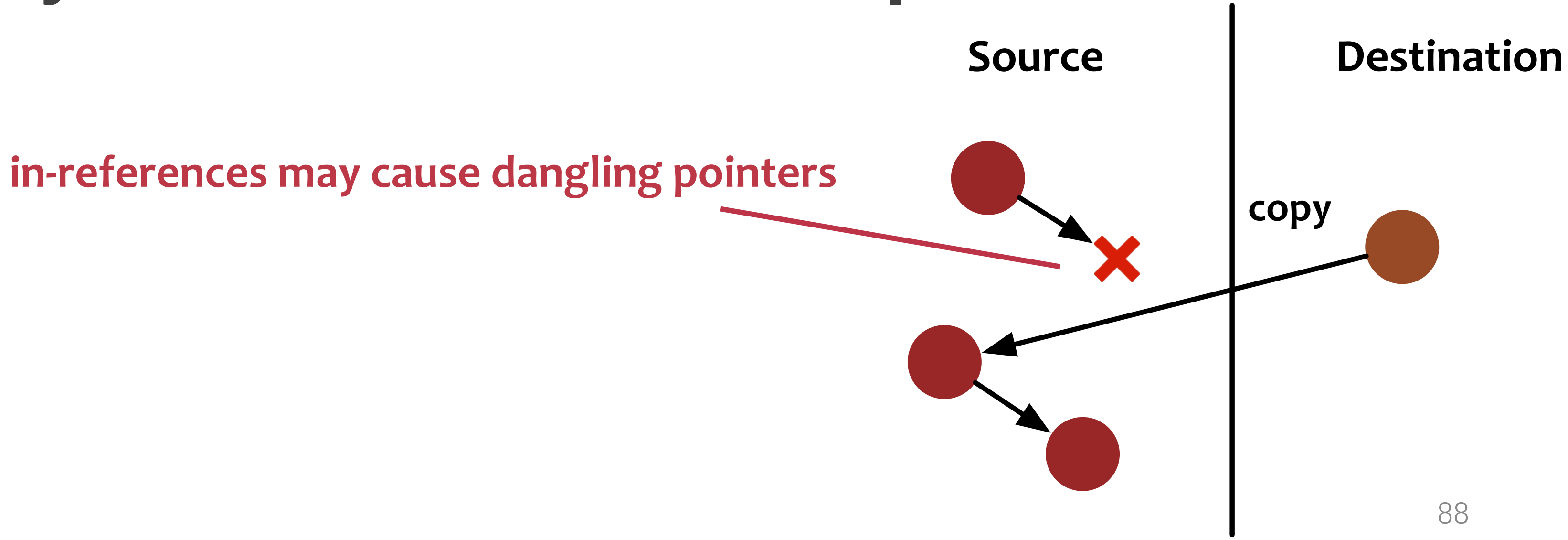
In-Reference Updates

- Only part of heap space will be scanned and copied (similar to G1)
- Only out-references will be updated



In-Reference Updates

- Only part of heap space will be scanned and copied (similar to G1)
- Only out-references will be updated



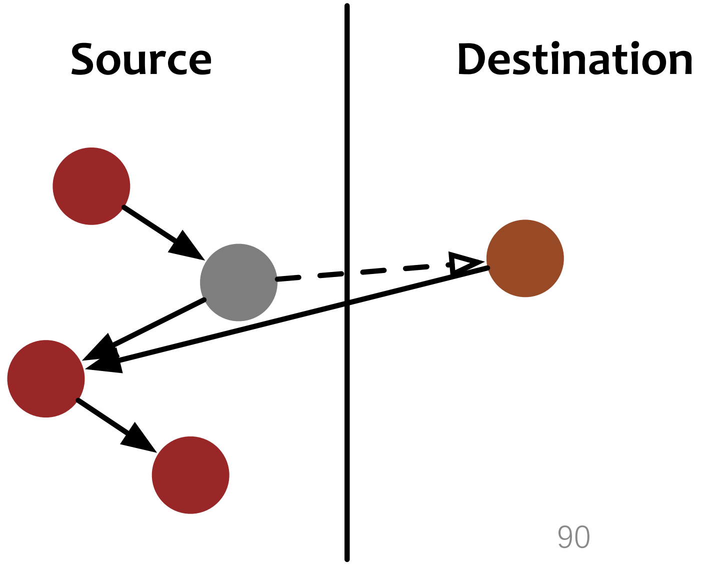
In-Reference Updates

- **Design 1: Stop-The-World update**
  - Large pause time (violating the goal of Shenandoah)

- **Design 2: piggy-backed with next marking phase**
  - Concurrent marking will scan the whole heap (similar to G1)

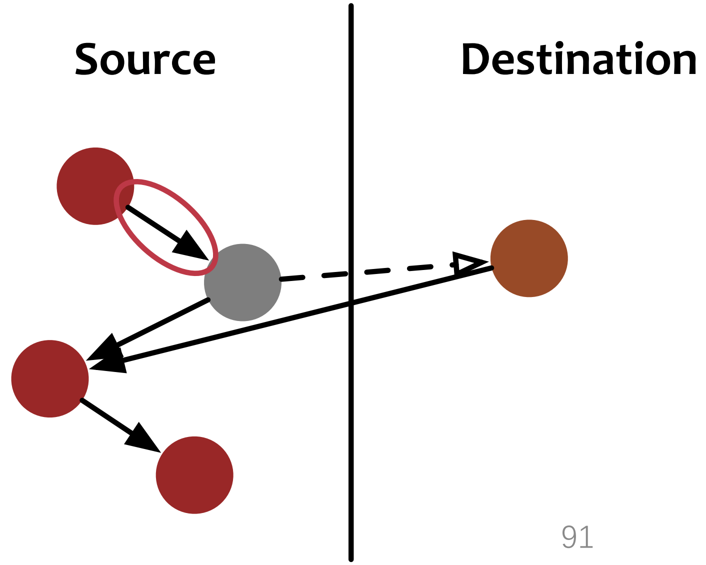
Lazy Reference Updates

- **Keeping a dummy object after collection**
  - Still using the indirect pointer



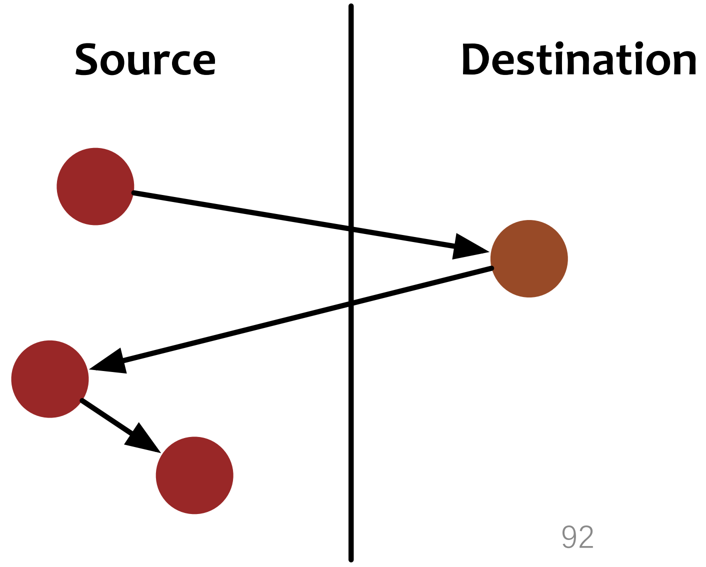
Lazy Reference Updates

- **Keeping a dummy object after collection**
  - Still using the indirect pointer
- **Correct references during marking**



Lazy Reference Updates

- **Keeping a dummy object after collection**
  - Still using the indirect pointer
- **Correct references during marking**
  - Now the object can be reclaimed



Results of Shenandoah

- **The pauses are really short!**

```
GC(3) Pause Init Mark 0.771ms
GC(3) Concurrent marking 76480M->77212M(102400M) 633.213ms
GC(3) Pause Final Mark 1.821ms
GC(3) Concurrent cleanup 77224M->66592M(102400M) 3.112ms
GC(3) Concurrent evacuation 66592M->75640M(102400M) 405.312ms
GC(3) Pause Init Update Refs 0.084ms
GC(3) Concurrent update references  75700M->76424M(102400M) 354.341ms
GC(3) Pause Final Update Refs 0.409ms
GC(3) Concurrent cleanup 76244M->56620M(102400M) 12.242ms
```

Results of Shenandoah

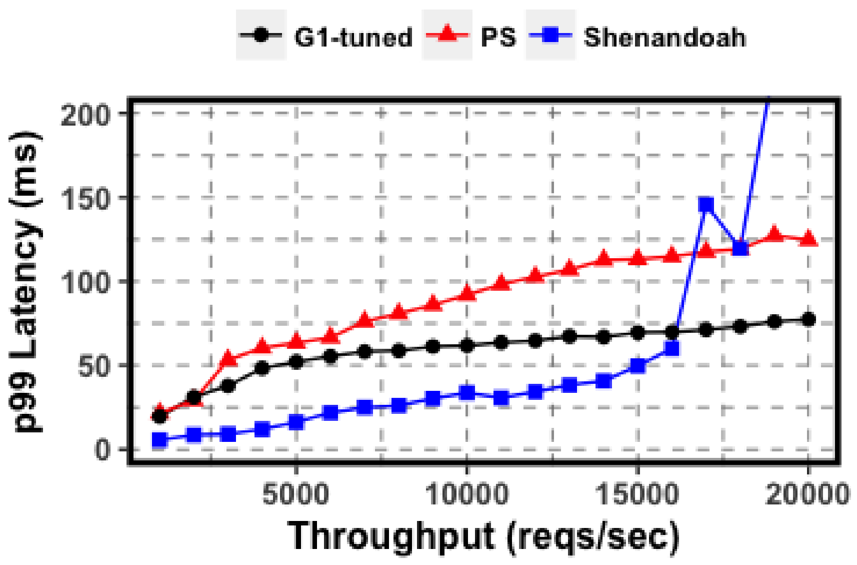
- **But the concurrent execution time is long**

```
GC(3) Pause Init Mark 0.771ms
GC(3) Concurrent marking 76480M->77212M(102400M) 633.213ms
GC(3) Pause Final Mark 1.821ms
GC(3) Concurrent cleanup 77224M->66592M(102400M) 3.112ms
GC(3) Concurrent evacuation 66592M->75640M(102400M) 405.312ms
GC(3) Pause Init Update Refs 0.084ms
GC(3) Concurrent update references  75700M->76424M(102400M) 354.341ms
GC(3) Pause Final Update Refs 0.409ms
GC(3) Concurrent cleanup 76244M->56620M(102400M) 12.242ms
```

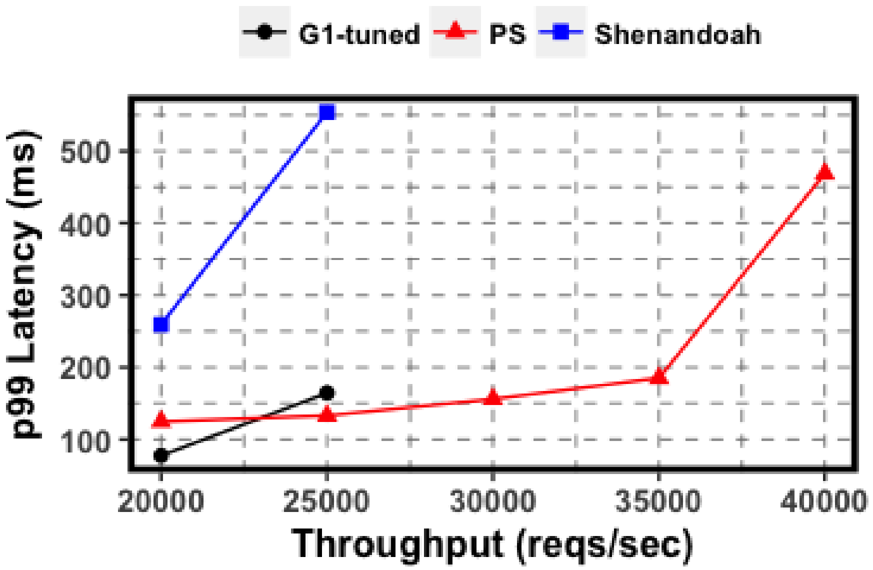
Comparison Among Modern GCs

- **SpecJBB2015 (online supermarket)**
  - 80 cores, 16GB heap

Low Throughput

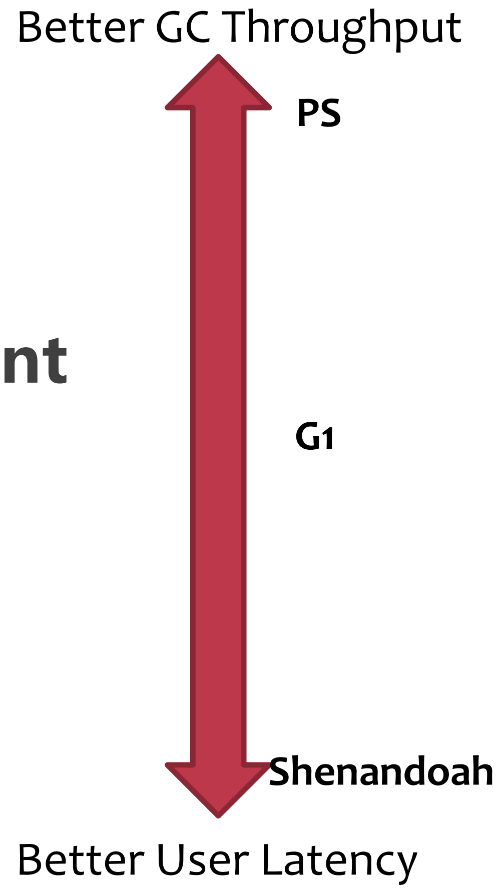


High Throughput



Summary: Throughput vs. Latency

- **PSGC: Stop-The-World**
  - Throughput-oriented
  - Large GC pauses
- **G1GC: adjustable & partially-concurrent**
  - Concurrent marking
  - Controllable GC pauses
- **Shenandoah: mostly-concurrent**
  - Concurrent collection
  - Ultra-low GC pauses
  - Hurting application throughput



Other Topics on JVM

- **Communication**
  - Reducing data transfer overhead (Skyway, ASPLOS18)
- **Multi-JVMs**
  - Coordination policies on GC (Taurus, ASPLOS16)
  - Elastic Java heap (ElasticMem, ATC17)
- **Non-volatile Memory**
  - Native support (Espresso, ASPLOS18)
  - Automatic persistence (AutoPersist, PLDI19)

Conclusion

- Language virtual machines: an "indirection" for portability and safety

- Introduction to JVM
  - Code execution: interpreter & JIT compiler
  - Memory management (GC): basics & modern GC designs

Thanks!

“All problems in computer science can be solved by another level of indirection… except for the problem of too many levels of indirection.”

Unknown