
ORACLE®

Safe and Efficient Hybrid Memory Management for Java

Codruț Stancu
Christian Wimmer

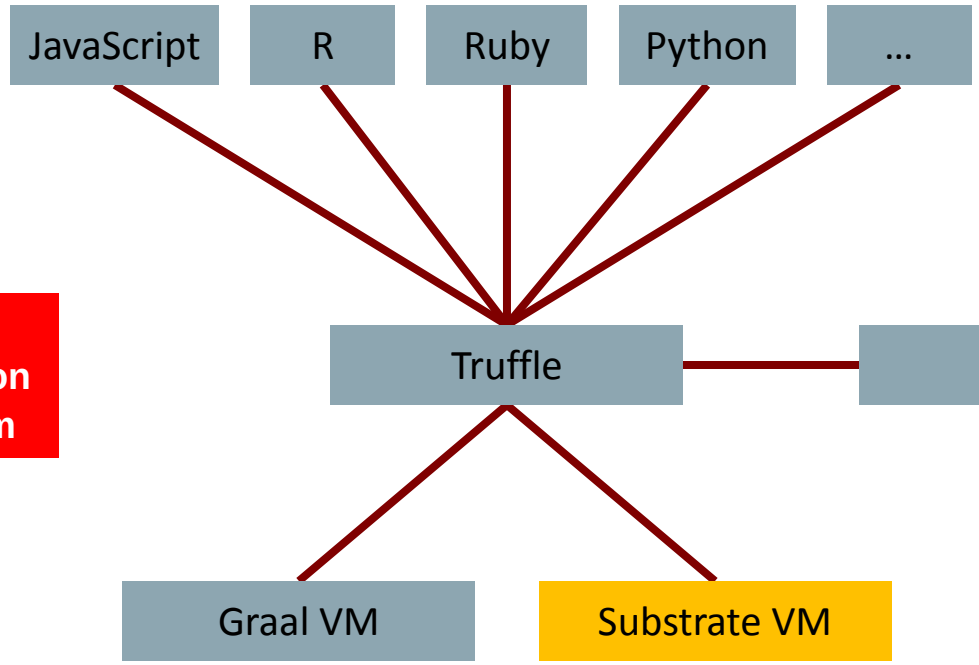
Oracle Labs

Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

Truffle System Structure

AST Interpreter for every language



Your language should be here!

Common API separates language implementation and optimization system

Language agnostic dynamic compiler

Integrate with Java applications

Low-footprint VM, also suitable for embedding

Substrate VM is …

… an **embeddable** VM

for, and written in, a **subset of Java**

optimized to **execute Truffle** languages

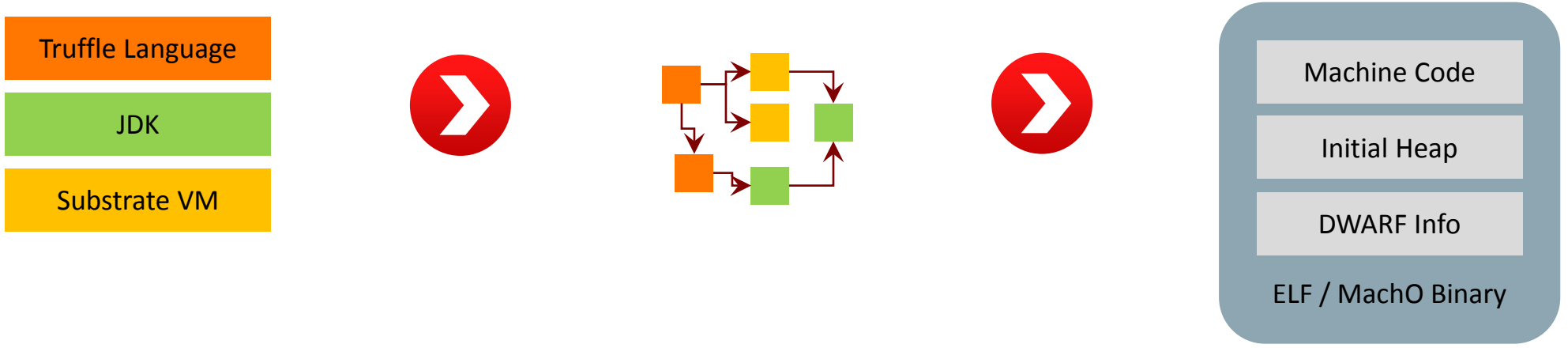
ahead-of-time compiled using Graal

integrating with **native development tools.**

Substrate VM: Execution Model

Static Analysis

Ahead-of-Time Compilation



All Java classes from Truffle language (or any application), JDK, and Substrate VM

Reachable methods, fields, and classes

Application running without dependency on JDK and without Java class loading

Microbenchmark for Startup and Peak Performance (1)

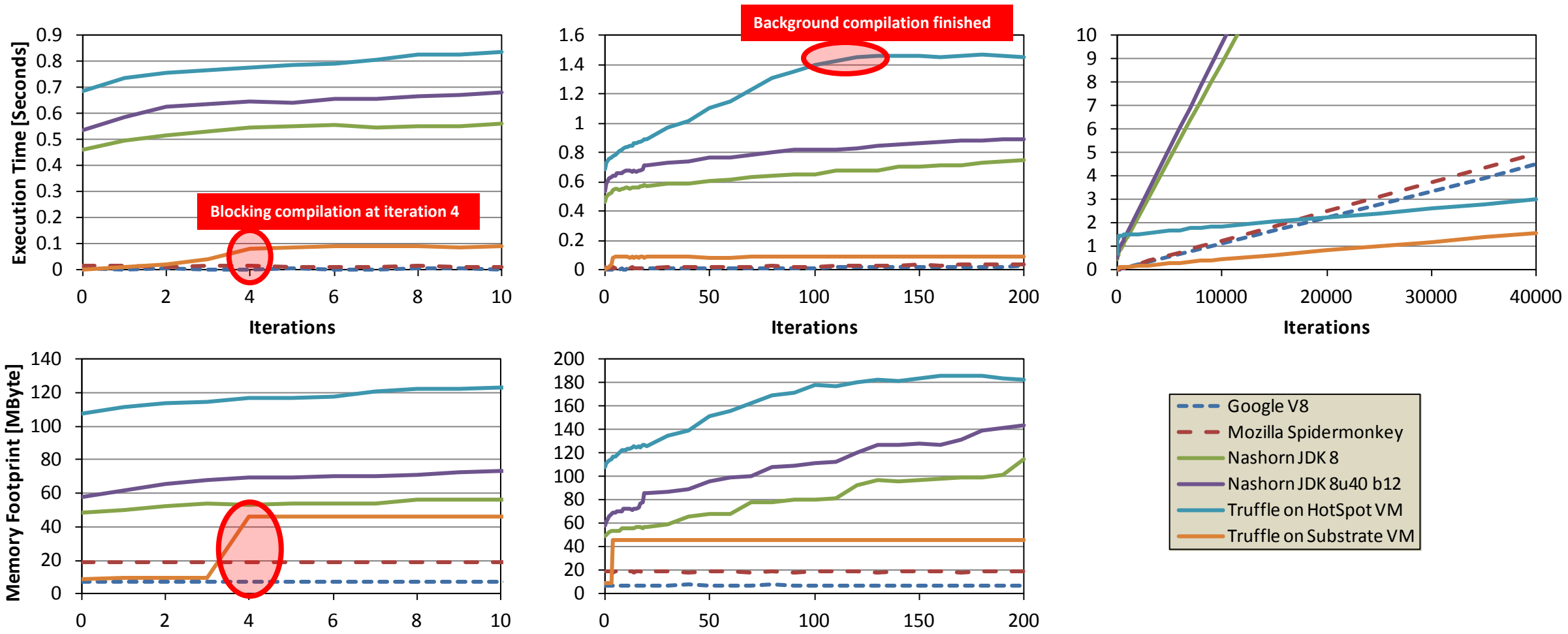
```
function benchmark(n) {
   var obj = {i: 0, result: 0};
   while (obj.i <= n) {
      obj.result = obj.result + obj.i;
      obj.i = obj.i + 1;
   }
   return obj.result;
}
```

Function benchmark is invoked in a loop by harness (0 to 40000 iterations)

n fixed to 50000 for all iterations

JavaScript VM	Version	Command Line Flags
Google V8	Version 3.26.31	[none]
Mozilla Spidermonkey	Version JavaScript-C36.0a1	[none]
Nashorn JDK 8	build 1.8.0-b132	-J-Xmx256M
Nashorn JDK8 update 40	build 1.8.0_40-ea-b12	-J-Xmx256M
Truffle on HotSpot VM	Changeset bb0f4716f830 from Nov 14, 2014	-J-Xmx256M
Truffle on Substrate VM	SVM changeset 1015c4a5458e from Nov 14, 2014 based on Graal and Truffle 0.5	[none]

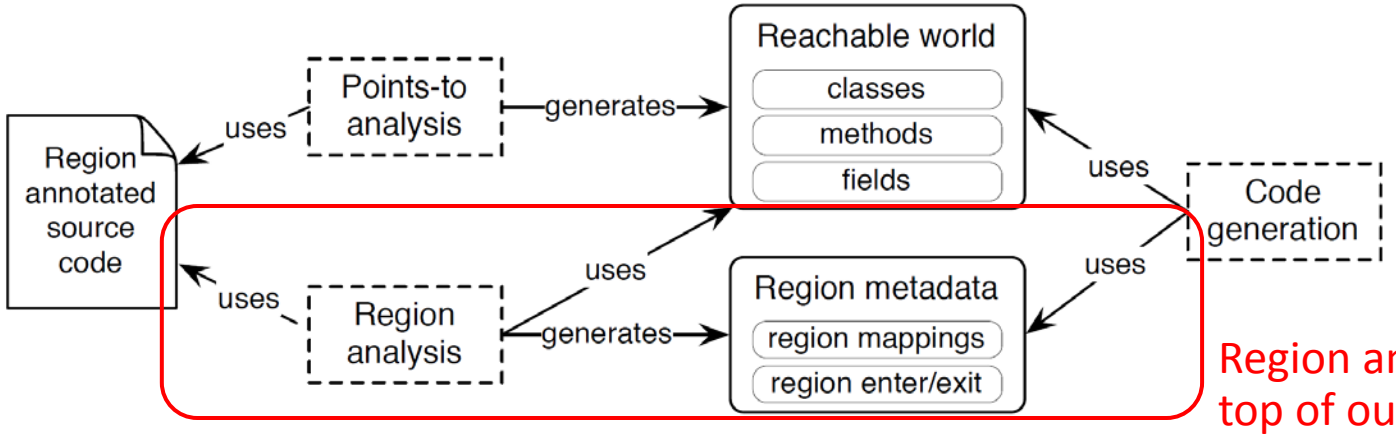
Microbenchmark for Startup and Peak Performance (2)



How can
execution pattern knowledge
be exploited to
optimize memory utilization?

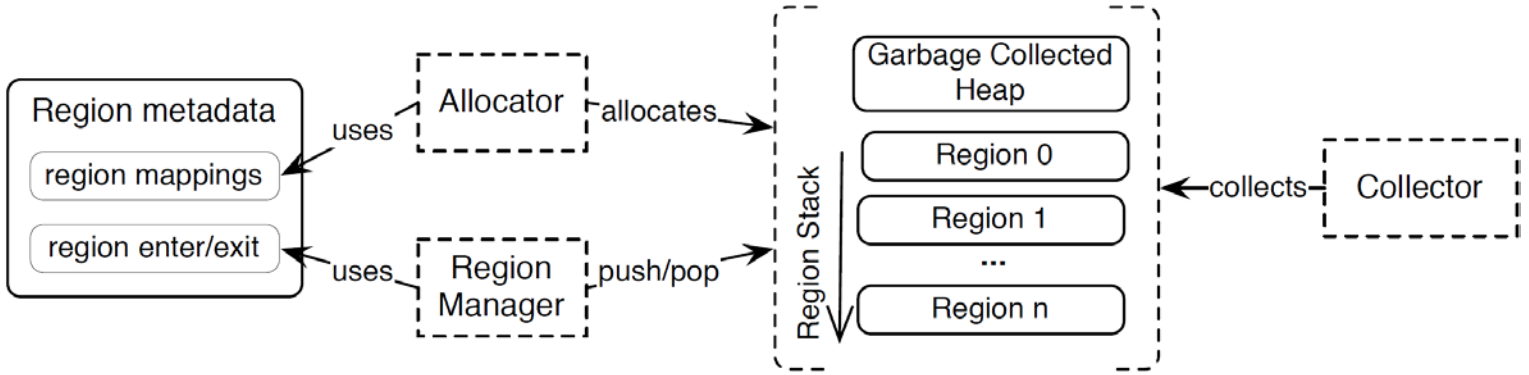
Region-Based Memory Management

Static Analysis



Region analysis is built on top of our static analysis

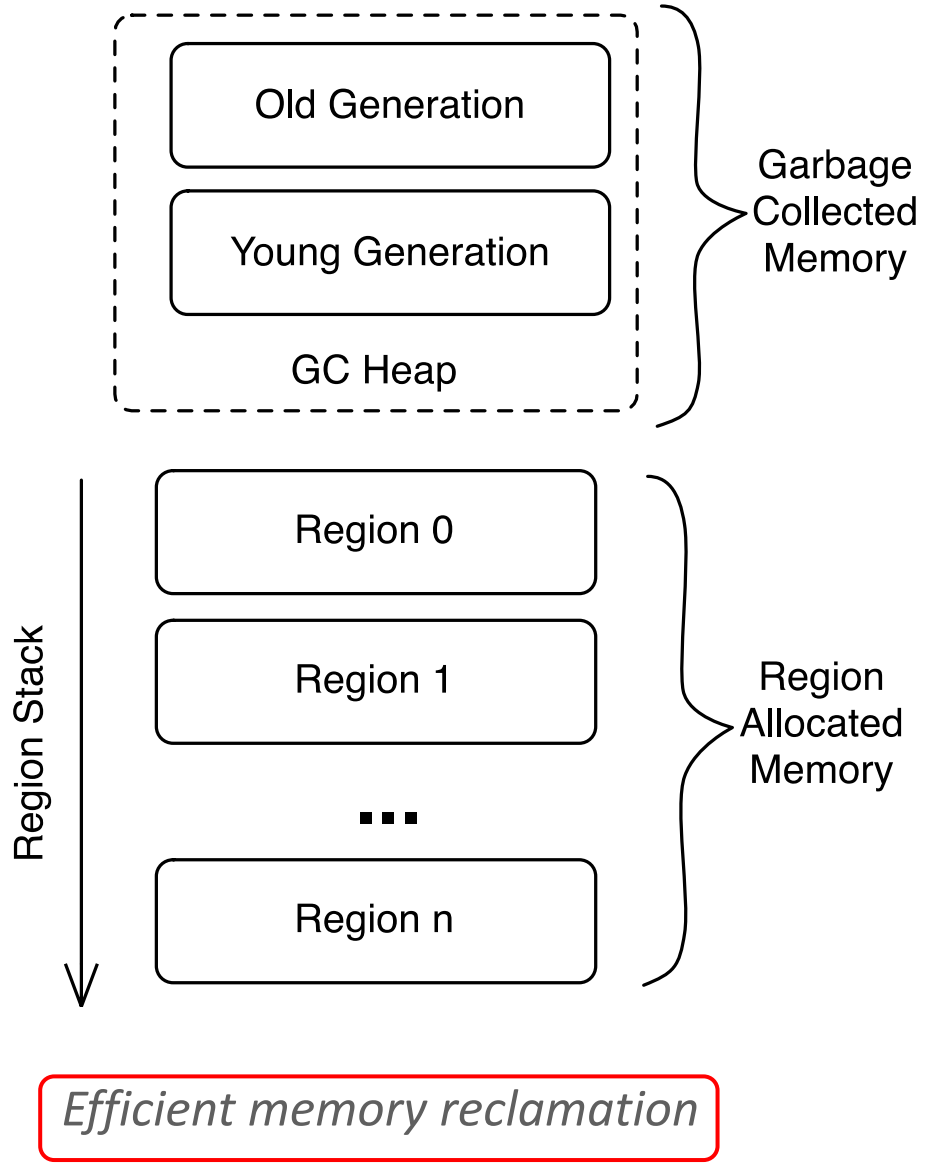
Runtime Data Structures



Automatic Memory Reclamation

- Modern VMs approach: one size fits all Garbage Collection
- GC has performance and memory footprint overhead

- **Region allocation**
  - Divide application in scoped phases
  - Release memory when phase exits
- GC as backup: hybrid memory management



Region Allocation

- Coarse grain regions, matching method borders
- Annotation based

```
@RegionScope(name = "R0")
public void foo() {
   // ...
}
```

- Static Region Analysis
  - Determine allocation region for each allocation site

Points-to Analysis

- Context sensitive
- Object abstraction: type & allocation site
- Invocation context is the receiver object abstraction
  - 2obj + 1H (*2-object-sensitive with 1-context-sensitive heap)*
- Generates:
  - Call graph
  - Points-to sets for reference variables and fields

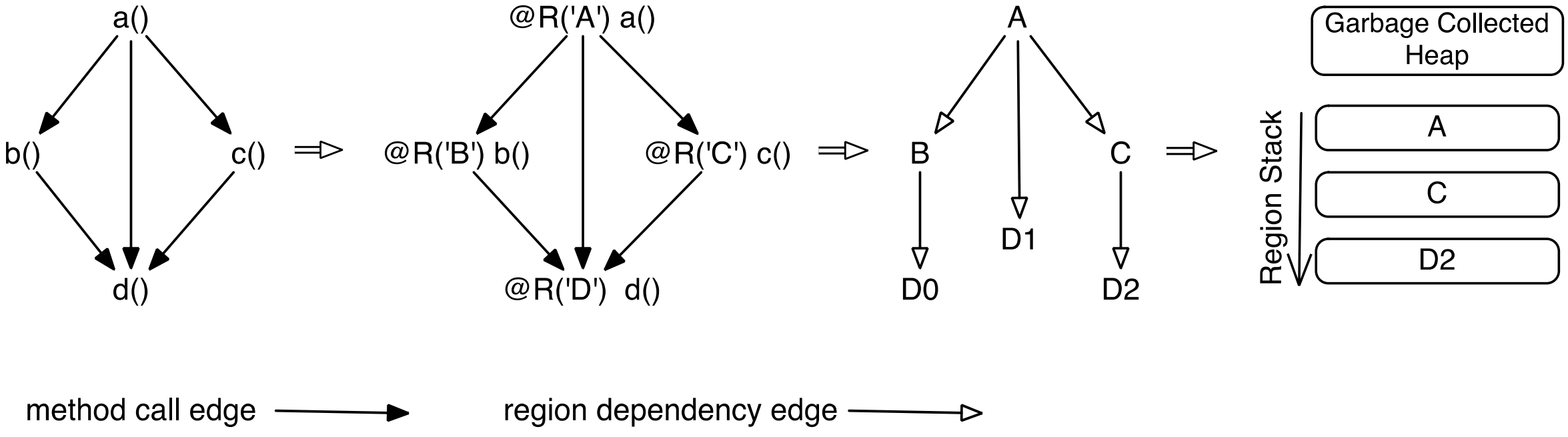
Region Analysis

- The points-to analysis uses the **region annotations as additional context elements**
- A method is analyzed separately for each different region scope from which it is invoked
- **2obj + 1H** analysis becomes **2obj + 1H + 1R**

  (2-object-sensitive with 1-context-sensitive heap and 1-region-context)

Region Analysis

- Iterates annotated call graph, depth first, and builds a region tree.



Region Analysis

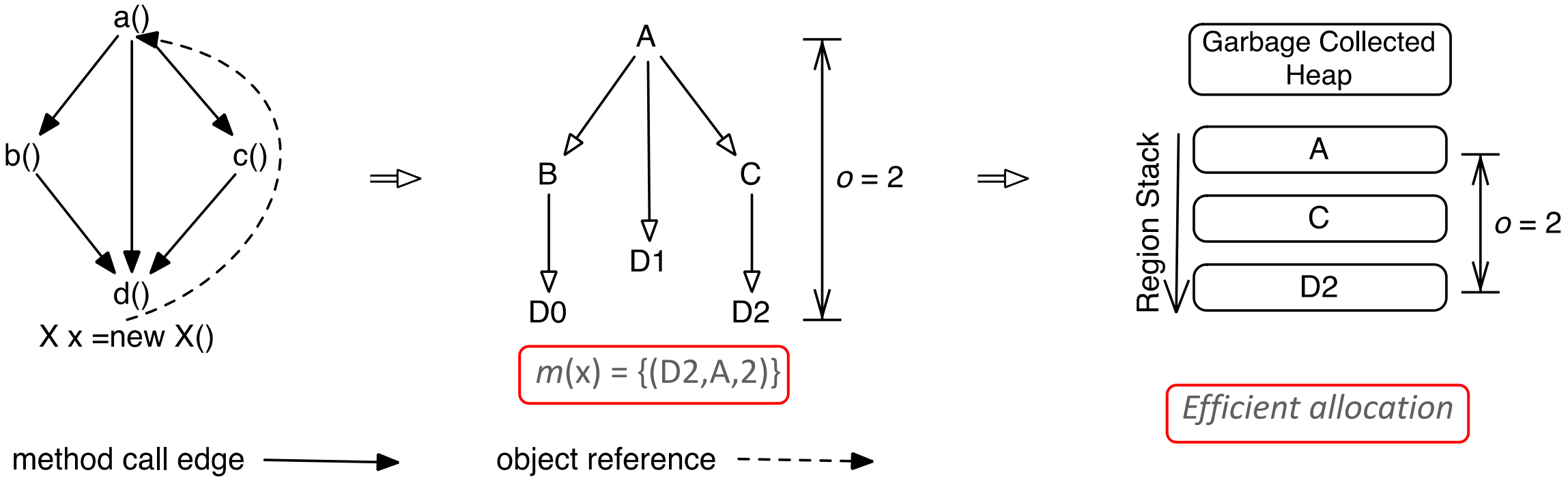
- Iterates over program statements
- For each object records **definitions** and **uses**
  - Definitions: new instance, new array
  - Uses: store, load, invoke, etc.
  - Return causes object to escape to caller
  - Store to static causes object to be allocated in the GC heap

Region analysis results: region mapping function

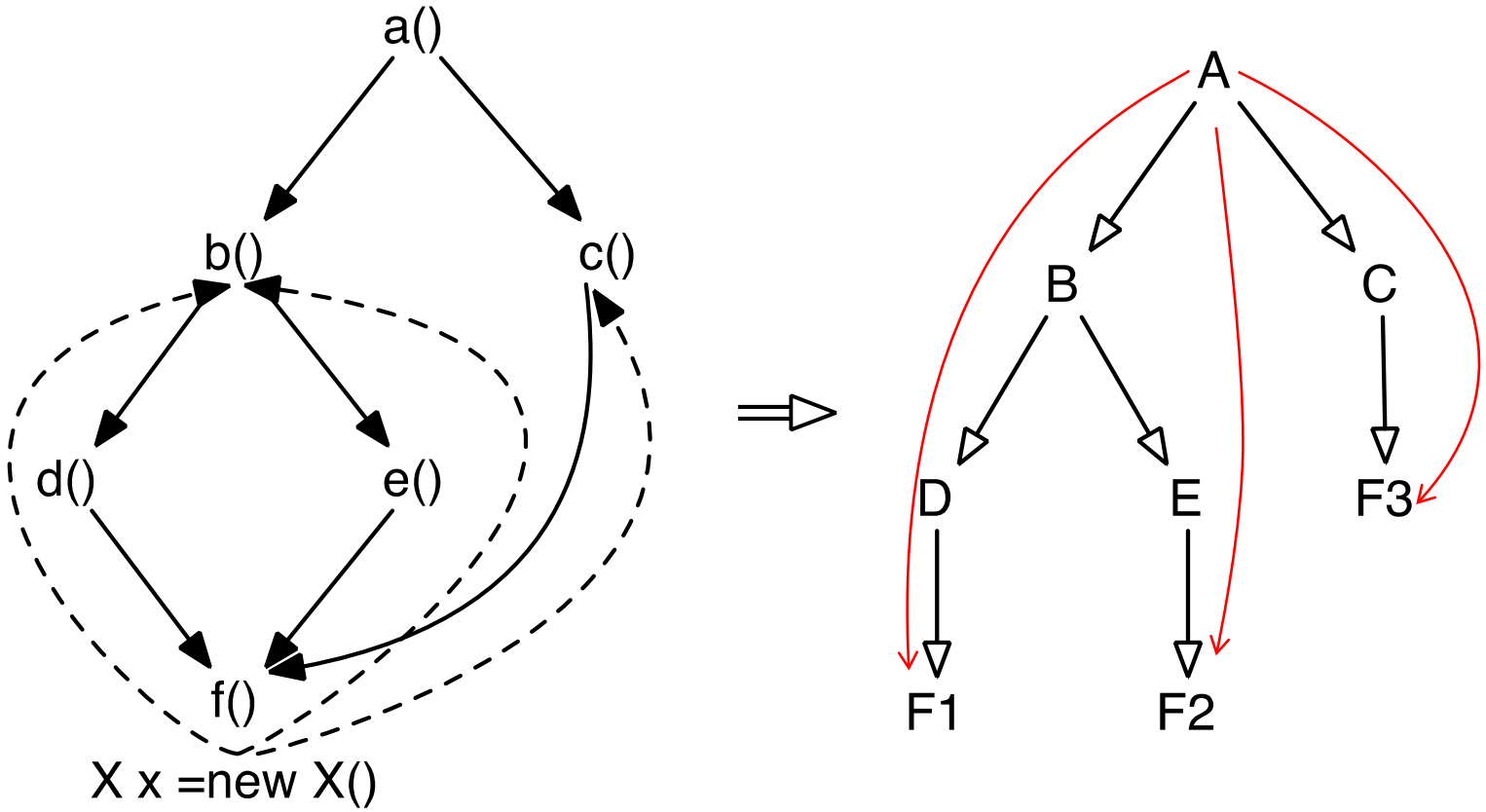
- ***m(A) = {(d, a, o)***, where

  m is the region mapping function, ***A*** is an allocation site,

  d is definition region context, ***a*** is allocation region, ***o*** is offset in runtime region stack}



Region Analysis Example



$m(x) = \{(F1,B,2)\}$

$m(x) = \{(F1,B,2), (F2,B,2)\}$

$m(x) = \{(F1,B,2), (F2,B,2), (F3,C,1)\}$

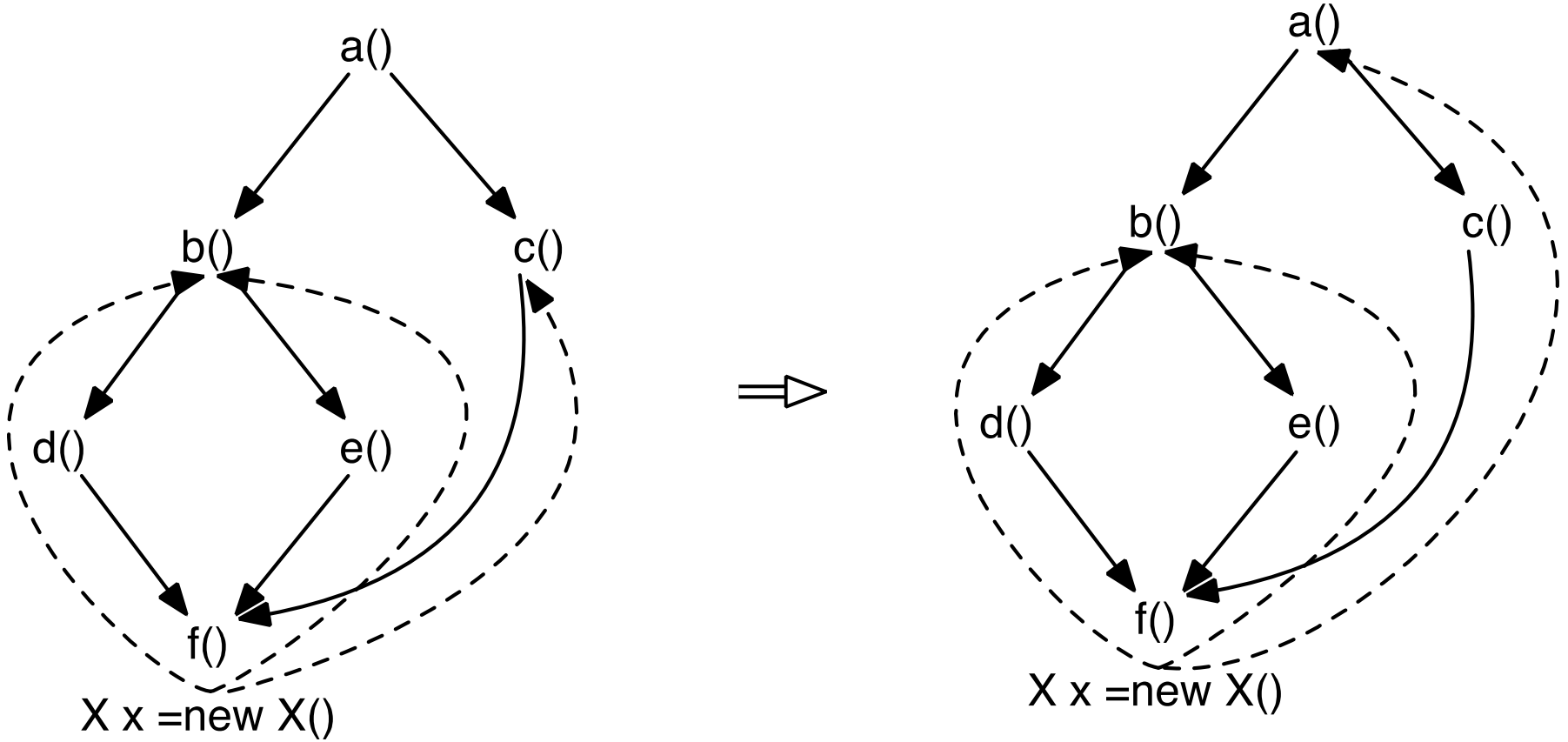
method call edge ——→ object reference - - - - → region dependency edge ——▷

Region Allocation

m(x) = {(F1,B,2), (F2,B,2), (F3,C,1)}

```
@Snippet
public static Object newInstanceSnippet(DynamicHub hub, RegionMapping mapping) {
    Region regionContext = RegionManager.regionStack().top();
    int offset = mapping.lookup(regionContext); // table lookup
    Region allocationRegion = RegionManager.regionStack().get(offset);
    return bump-pointer allocation in allocationRegion
}
```

Region Analysis Normalization Example



m(x) = {(F1,B,2), (F2,B,2), (F3,C,1)}

after normalization:

m(x) = {(F1,B,2), (F2,B,2), **(F3,A,2)**}

method call edge ——→ object reference - - - - →

Offset Normalization Optimization

m(x) = {(F1,B,2), (F2,B,2), (F3,C,1)}

```
@Snippet
public static Object newInstanceSnippet(DynamicHub hub, RegionMapping mapping) {
    Region regionContext = RegionManager.regionStack().top();
    int offset = mapping.lookup(regionContext); // table lookup
    Region allocationRegion = RegionManager.regionStack().get(offset);
    return bump-pointer allocation in allocationRegion
}
```

m(x) = {(F1,B,2), (F2,B,2), **(F3,A,2)**}

```
@Snippet
public static Object newInstanceSnippet(DynamicHub hub, int constantOffset) {
   Region allocationRegion = RegionManager.regionStack().get(constantOffset);
   return bump-pointer allocation in allocationRegion
}
```

Memory Management

Memory Chunks
Fixed size, aligned on size-granularity, contains all GC tables
e.g., 1 MByte, aligned on 1 MByte granularity

Young Generation (nursery space)
GC evacuates all live objects to old generation, no aging, live objects identified using card marks

Old Generation
Stop&Copy algorithm for first implementation

Regions (identified by static analysis)
No object referenced from outside
Can reference outside objects
Usually freed without GC
Processed by Young GC

Large Arrays
Memory requested per object from OS
Never moved

Experiments

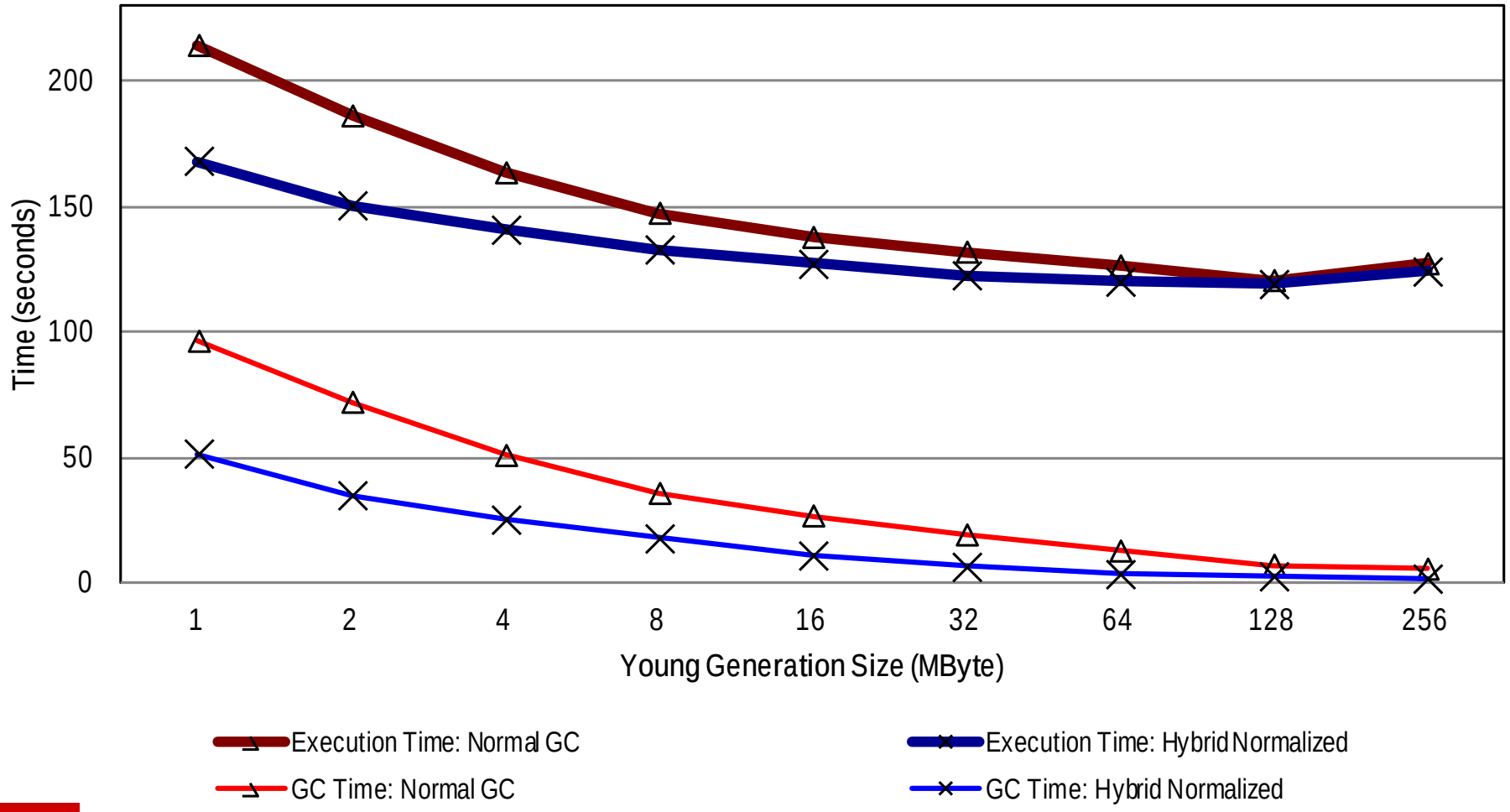
- Benchmark: SPECjbb2005
- Configuration:
  - Normal GC & hybrid normalized
  - 256MB young gen.

Lines of code	12,581
Annotations	7

- 78% of memory region allocatable
- 583 KB maximum region stack size

Region enter #	hybrid normalized	3000030
Region deallocation #	hybrid normalized	2999952
Max stack depth	hybrid normalized	3
Allocated memory (MByte)	all configurations	56777
GC freed memory (MByte)	normal GC	56497
	hybrid normalized	12124
Region freed memory (MByte)	hybrid normalized	43625
Region freed memory (%)	hybrid normalized	77.62%
Max region stack size (KByte)	hybrid normalized	583
Full GC #	normal GC	2
	hybrid normalized	0
Incremental GC #	normal GC	219
	hybrid normalized	49
Full GC time (s)	normal GC	0.95
	hybrid normalized	0.00
Incremental GC time (s)	normal GC	4.68
	hybrid normalized	1.64
Execution time (s)	normal GC	127.43
	hybrid normalized	124.12
Speedup (%)	hybrid normalized	3%

Execution and GC time, varying young generation size



Thank You!

- More information on hybrid memory management
  - Codrut Stancu, Christian Wimmer, Stefan Brunthaler, Per Larsen, Michael Franz: Safe and Efficient Hybrid Memory Management for Java. In Proceedings of the International Symposium on Memory Management, pages 81–92. ACM Press, 2015. doi:10.1145/2754169.2754185
- More information on Graal and Truffle:
  - https://wiki.openjdk.java.net/display/Graal/Publications+and+Presentations

Hardware and Software
Engineered to Work Together


ORACLE®