
ORACLE®

ORACLE®

Maxine: A JVM Written in Java

Michael Haupt
Oracle Labs
Potsdam, Germany

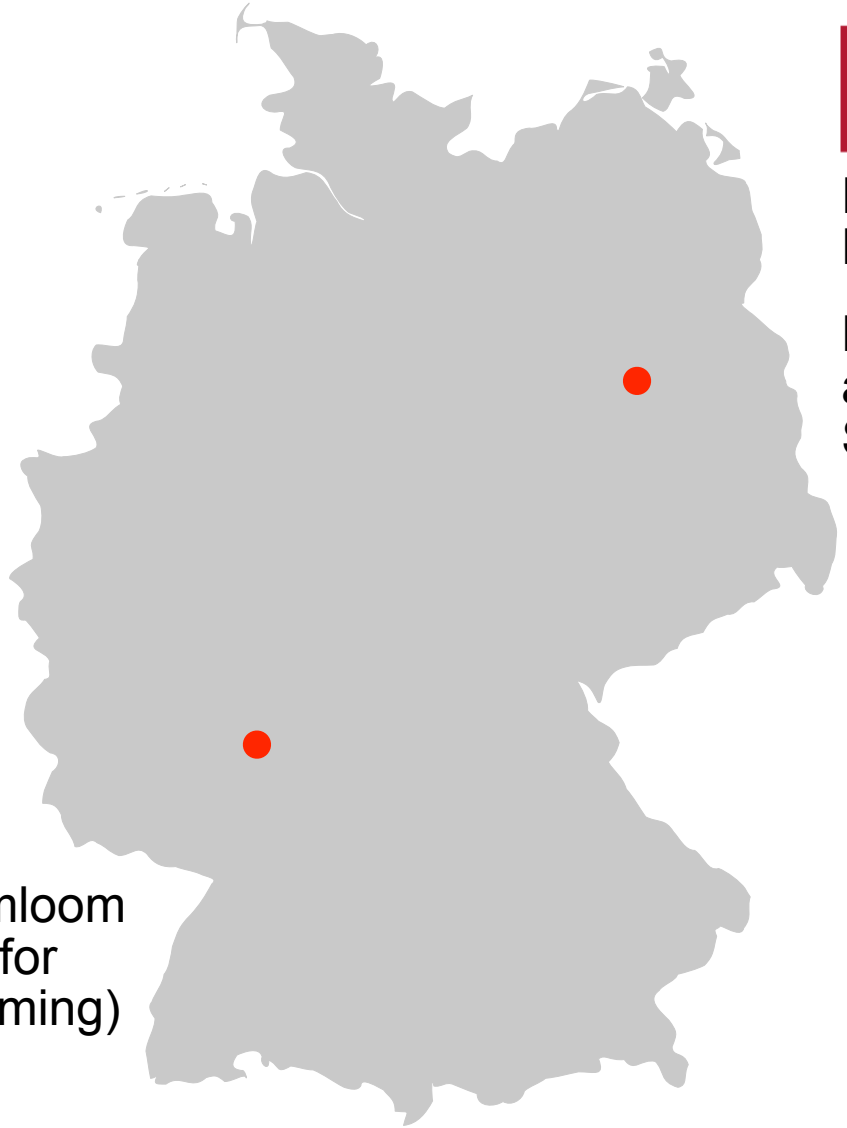
The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Speaker's Background

Technische Universität
Darmstadt, 2001–2006

Doctoral research, Steamloom
(virtual machine support for
aspect-oriented programming)

Speaker's Background



Technische Universität
Darmstadt, 2001–2006

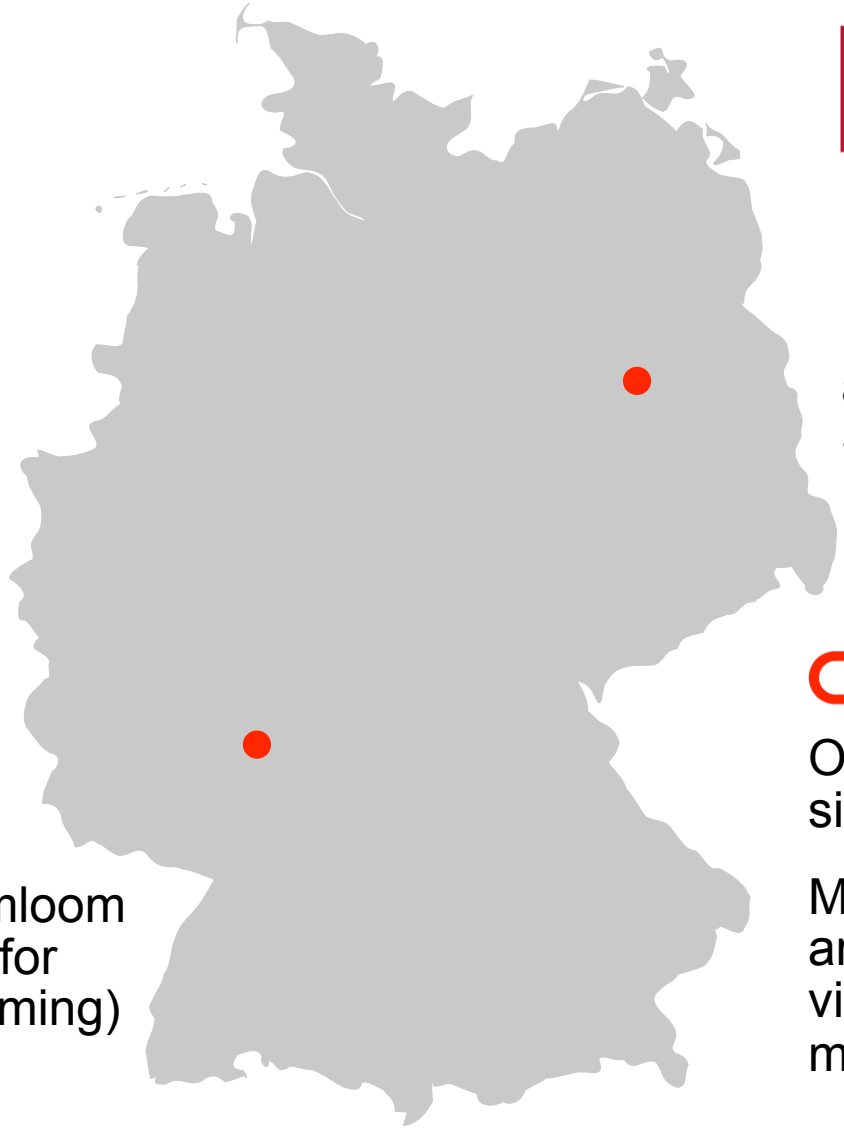
Doctoral research, Steamloom
(virtual machine support for
aspect-oriented programming)

HPI

Hasso-Plattner-Institut
Potsdam, 2006–2011

Postdoc, virtual machine
architecture disentangling,
SOM family, NXTalk

Speaker's Background



Technische Universität
Darmstadt, 2001–2006

Doctoral research, Steamloom
(virtual machine support for
aspect-oriented programming)

Hasso-Plattner-Institut
Potsdam, 2006–2011

Postdoc, virtual machine
architecture disentangling,
SOM family, NXTalk

ORACLE

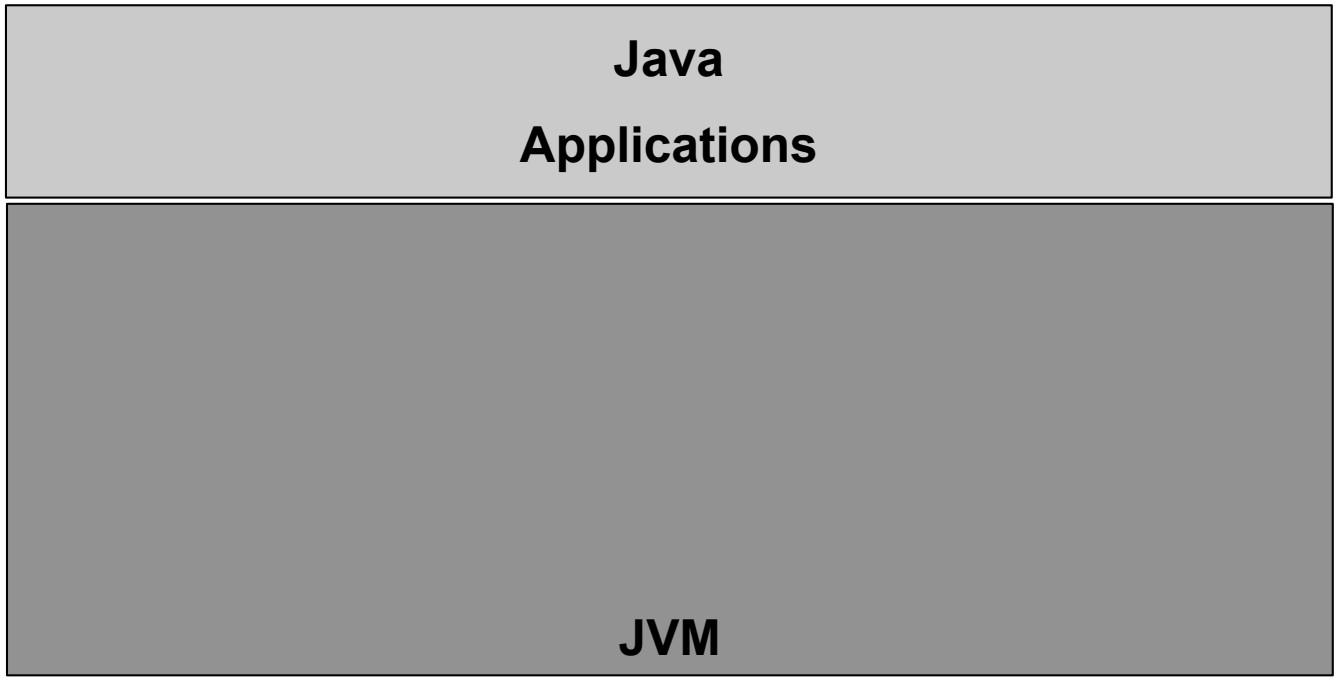
Oracle Labs, Potsdam,
since 2011

Maxine team, virtual machine
architecture, multi-language
virtual machines, code cache
management, JSR292

Who Needs a JVM?

Who Needs a JVM?

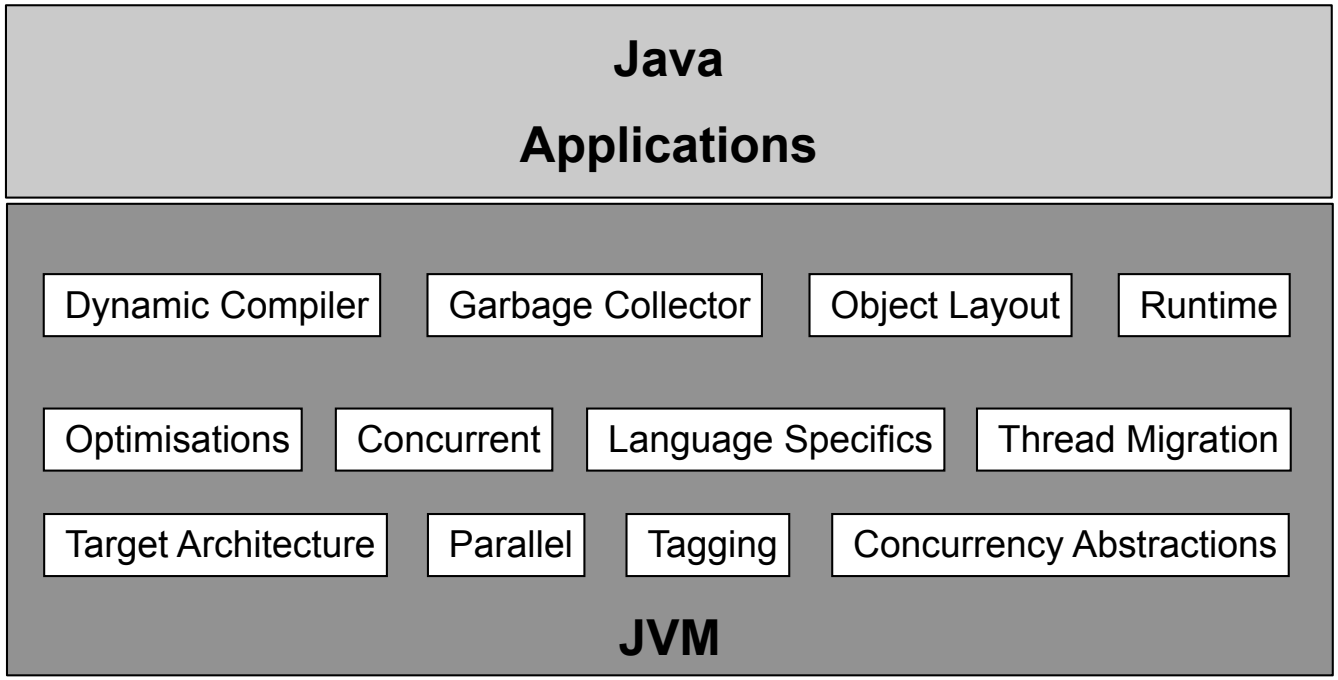
application developers, end users ✔



Who Needs a JVM?

application developers, end users ✔

researchers ✔

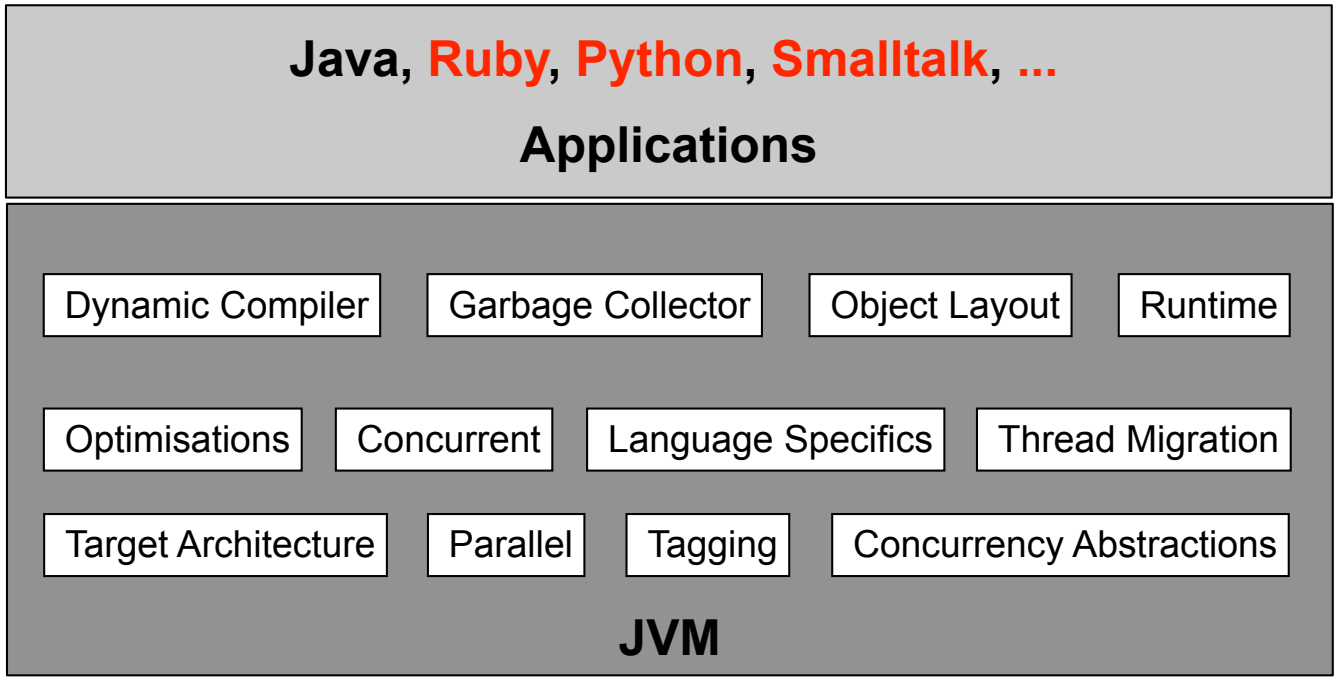


Who Needs a JVM?

application developers, end users ✔

researchers ✔

programming language implementers ✔

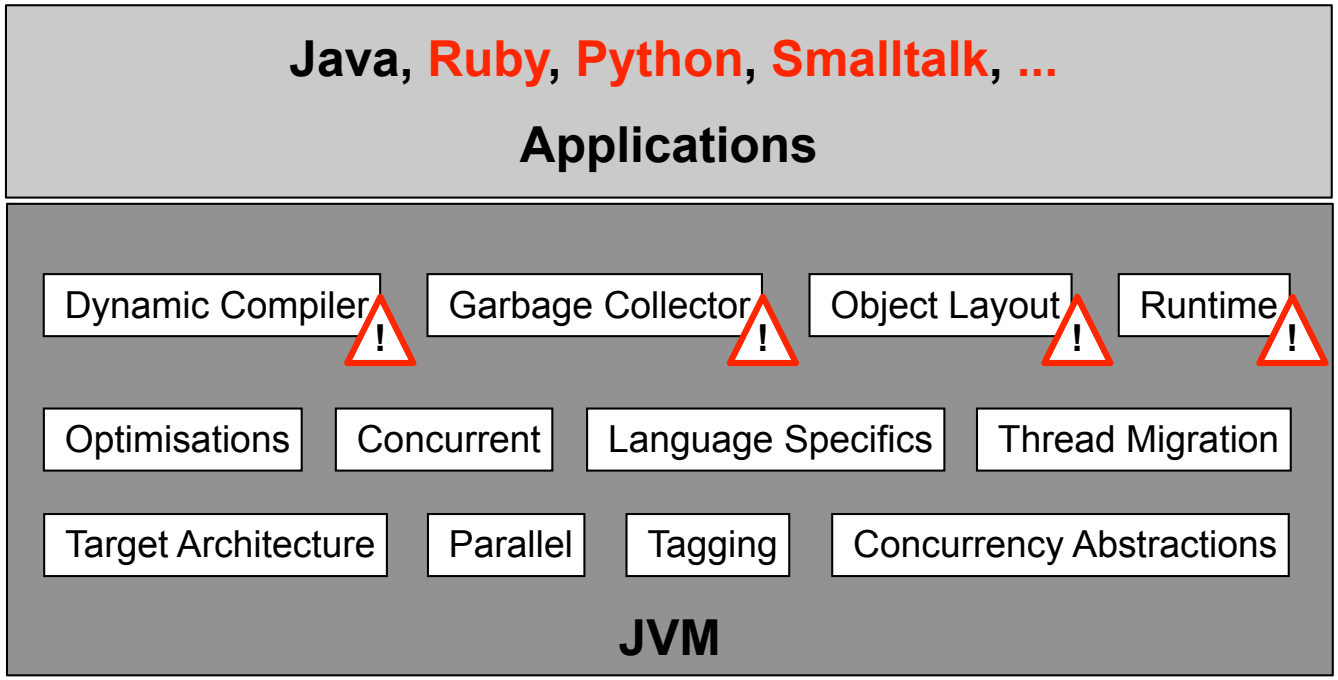


Who Needs a JVM?

application developers, end users ✔

researchers ✔

programming language implementers ✔

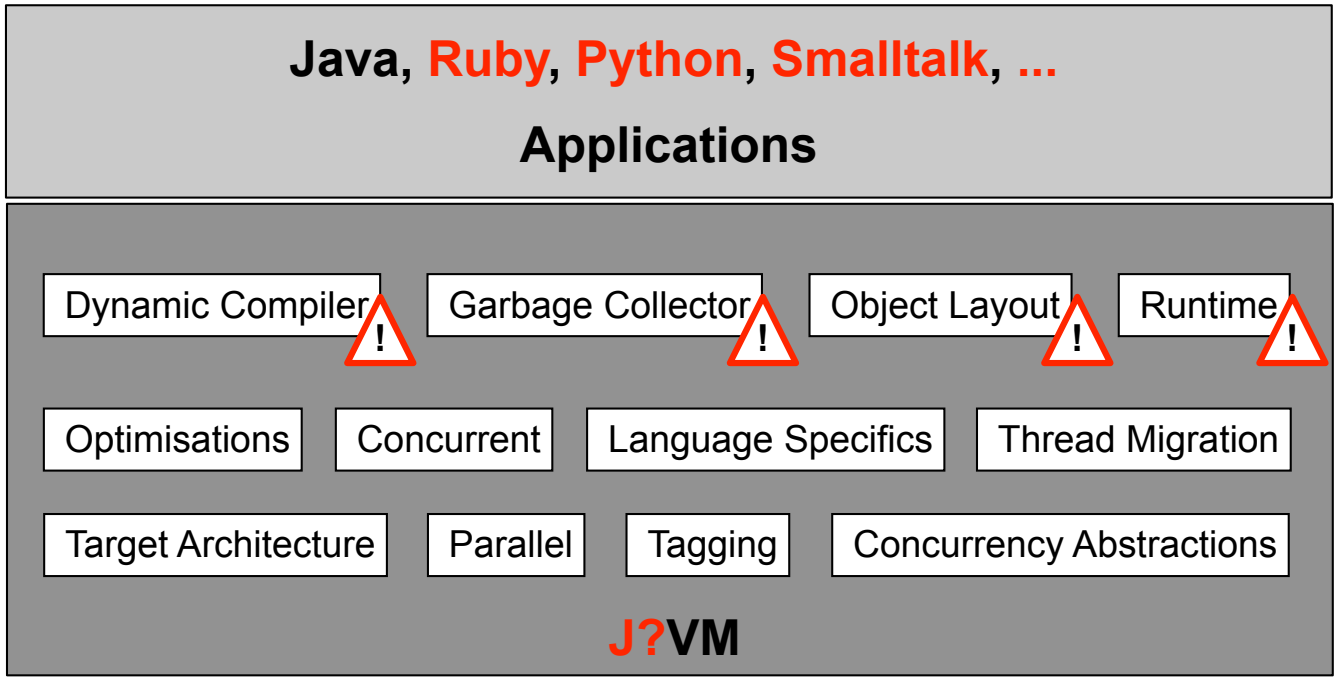


Who Needs a JVM?

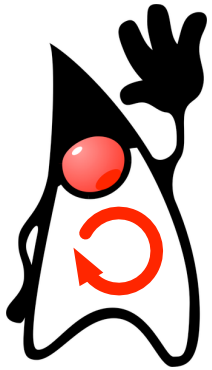
application developers, end users ✔

researchers ✔

programming language implementers ✔



Maxine



almost 100 % Java

use unmodified JDK

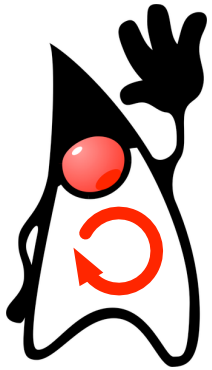
meta-circularity studies

Home page: http://wikis.oracle.com/display/MaxineVM/Home

Source (GPLv2): https://hg.kenai.com/hg/maxine~maxine

Mailing list via: http://kenai.com/projects/maxine

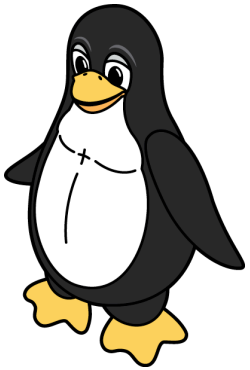
Maxine



almost 100 % Java

use unmodified JDK

meta-circularity studies



supported on 64-bit x86

Solaris, Mac OS X, Linux

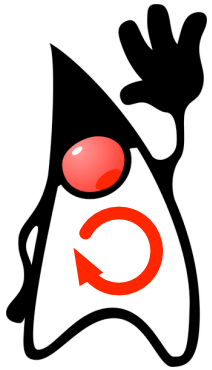
Virtual Edition: Xen

Home page: http://wikis.oracle.com/display/MaxineVM/Home

Source (GPLv2): https://hg.kenai.com/hg/maxine~maxine

Mailing list via: http://kenai.com/projects/maxine

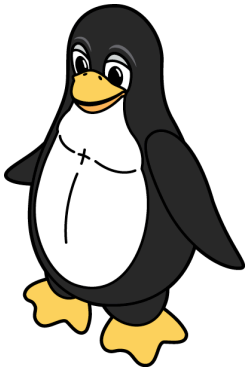
Maxine



almost 100 % Java

use unmodified JDK

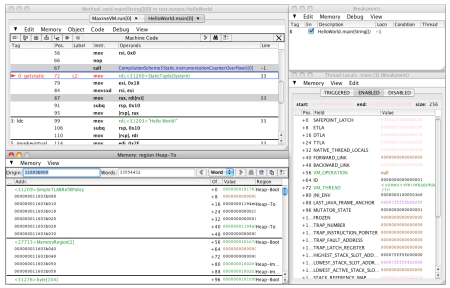
meta-circularity studies



supported on 64-bit x86

Solaris, Mac OS X, Linux

Virtual Edition: Xen



full IDE support

(Eclipse, NetBeans)

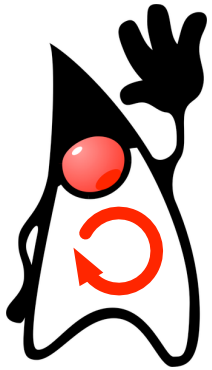
tooling: Inspector

Home page: http://wikis.oracle.com/display/MaxineVM/Home

Source (GPLv2): https://hg.kenai.com/hg/maxine~maxine

Mailing list via: http://kenai.com/projects/maxine

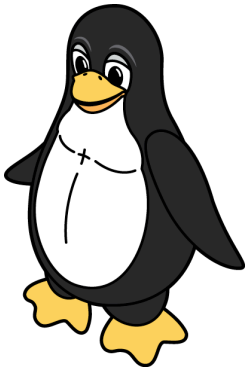
Maxine



almost 100 % Java

use unmodified JDK

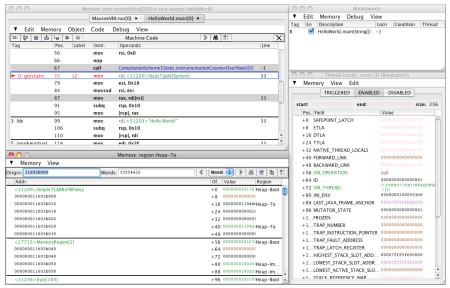
meta-circularity studies



supported on 64-bit x86

Solaris, Mac OS X, Linux

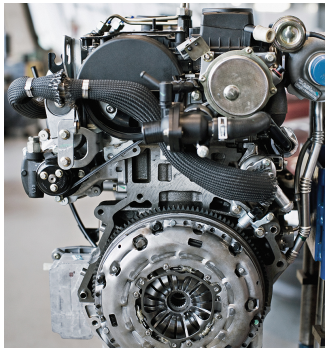
Virtual Edition: Xen



full IDE support

(Eclipse, NetBeans)

tooling: Inspector



dynamic compilation only

semi-space GC

modular architecture

Home page: http://wikis.oracle.com/display/MaxineVM/Home

Source (GPLv2): https://hg.kenai.com/hg/maxine~maxine

Mailing list via: http://kenai.com/projects/maxine

VM Research Group at Oracle Labs

Doug
Simon

Thomas
Würthinger

Michael
Haupt

Christian
Wimmer

Laurent
Daynès

Mario
Wolczko

Director

Mick
Jordan

Michael
Van De Vanter

Home page: http://wikis.oracle.com/display/MaxineVM/Home

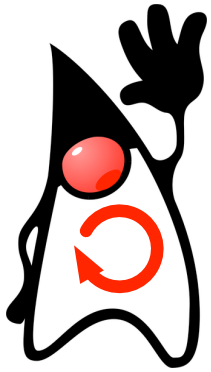
Source (GPLv2): https://hg.kenai.com/hg/maxine~maxine

Mailing list via: http://kenai.com/projects/maxine

Why Java?

language
- reflection
- annotations

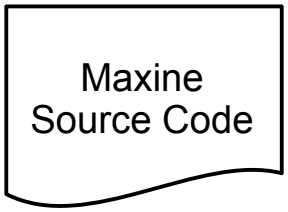
JDK
- rich and powerful API
- inclusion in VM



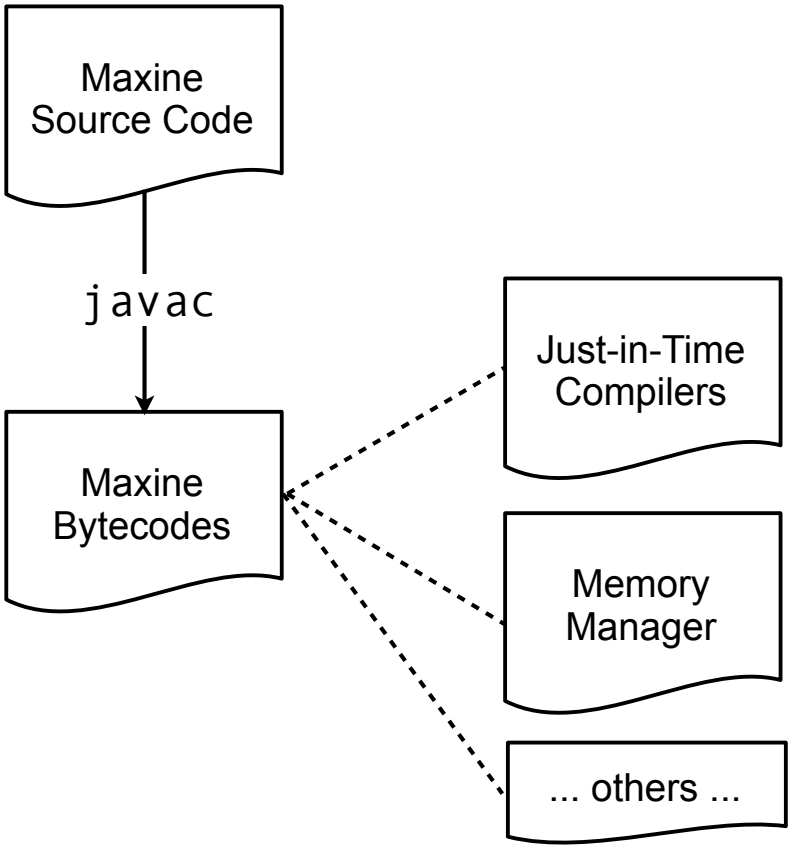
VM architecture
- uniform representation
- same compiler chain

ecosystem
- IDE support
- tooling

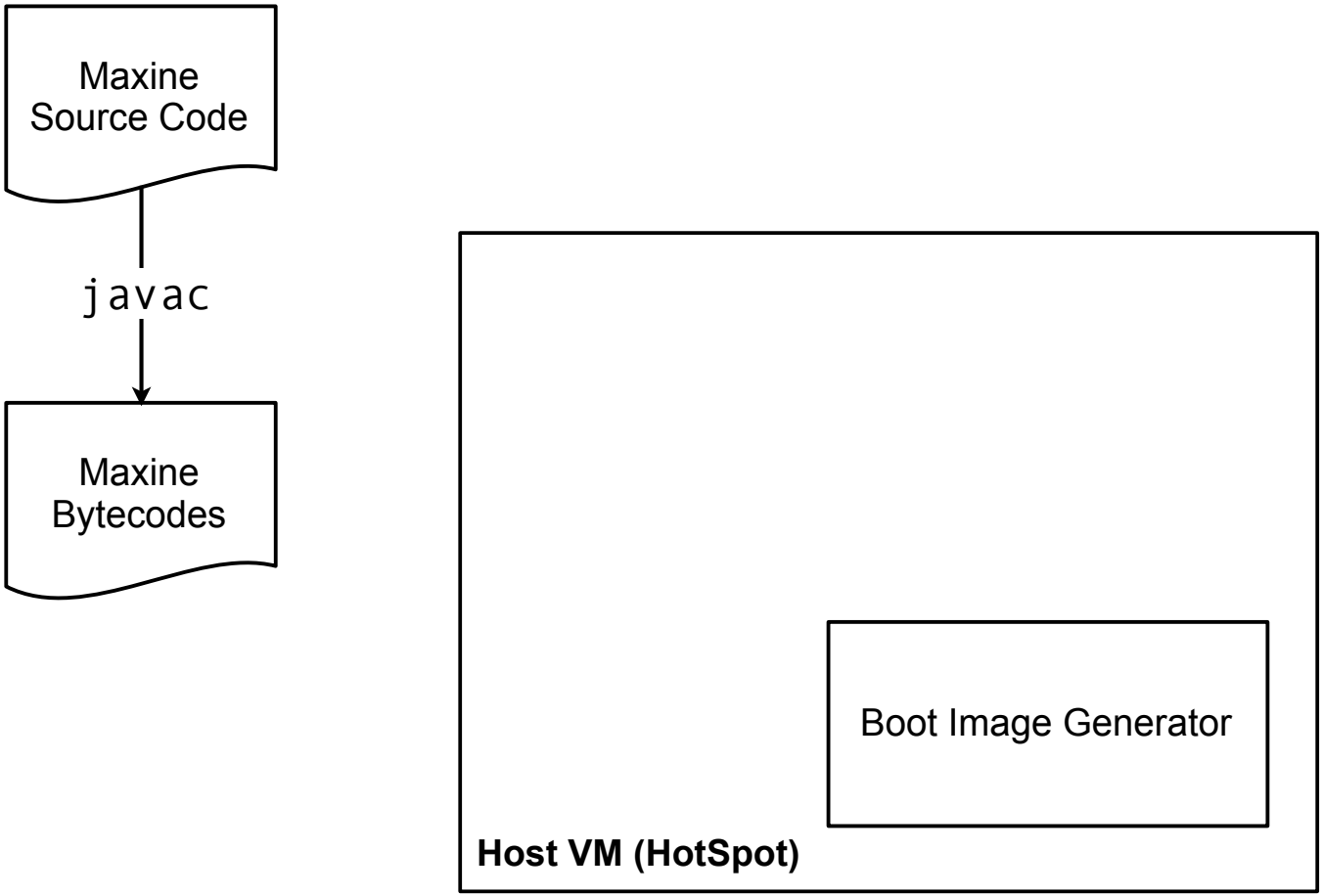
Bootstrapping Maxine



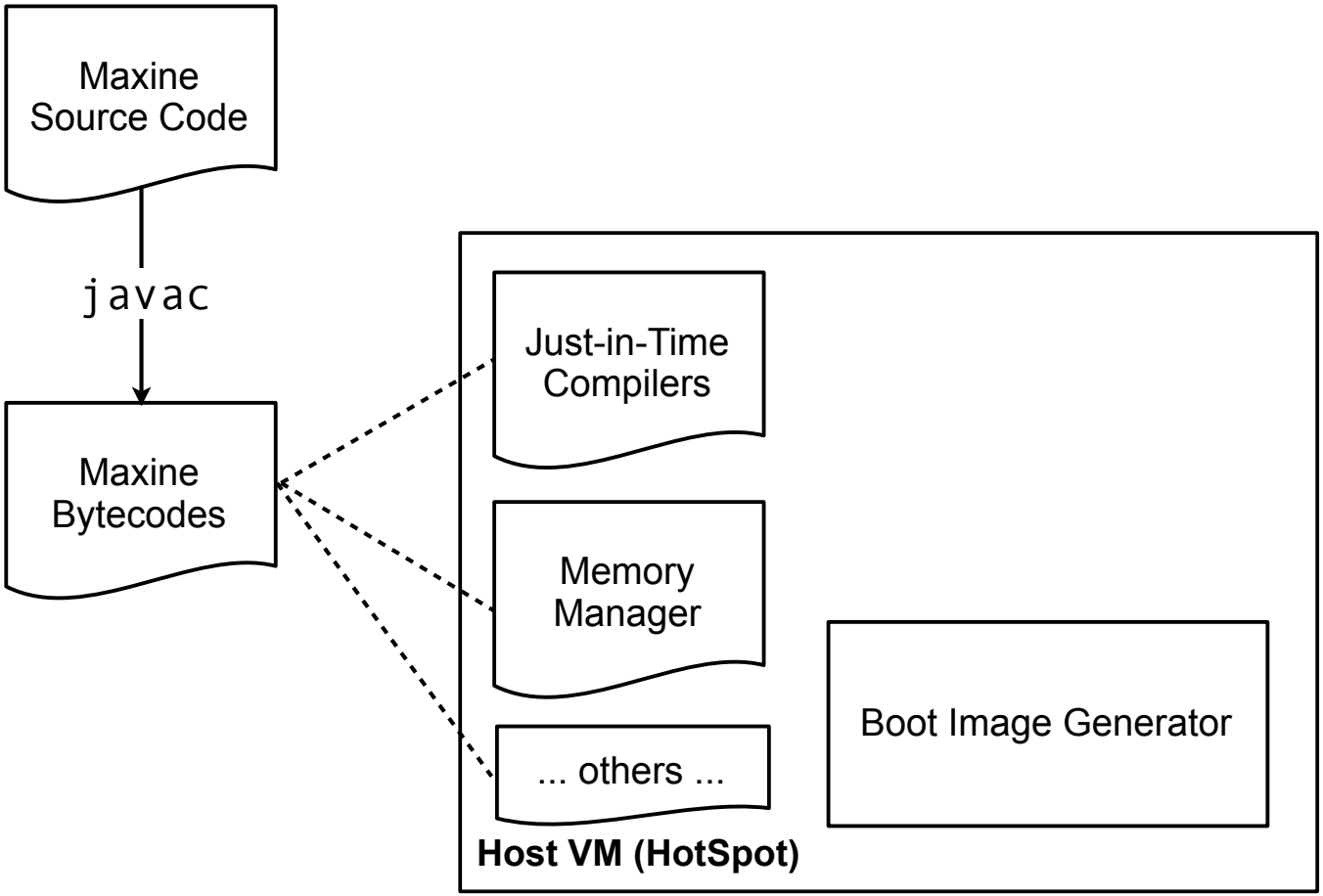
Bootstrapping Maxine



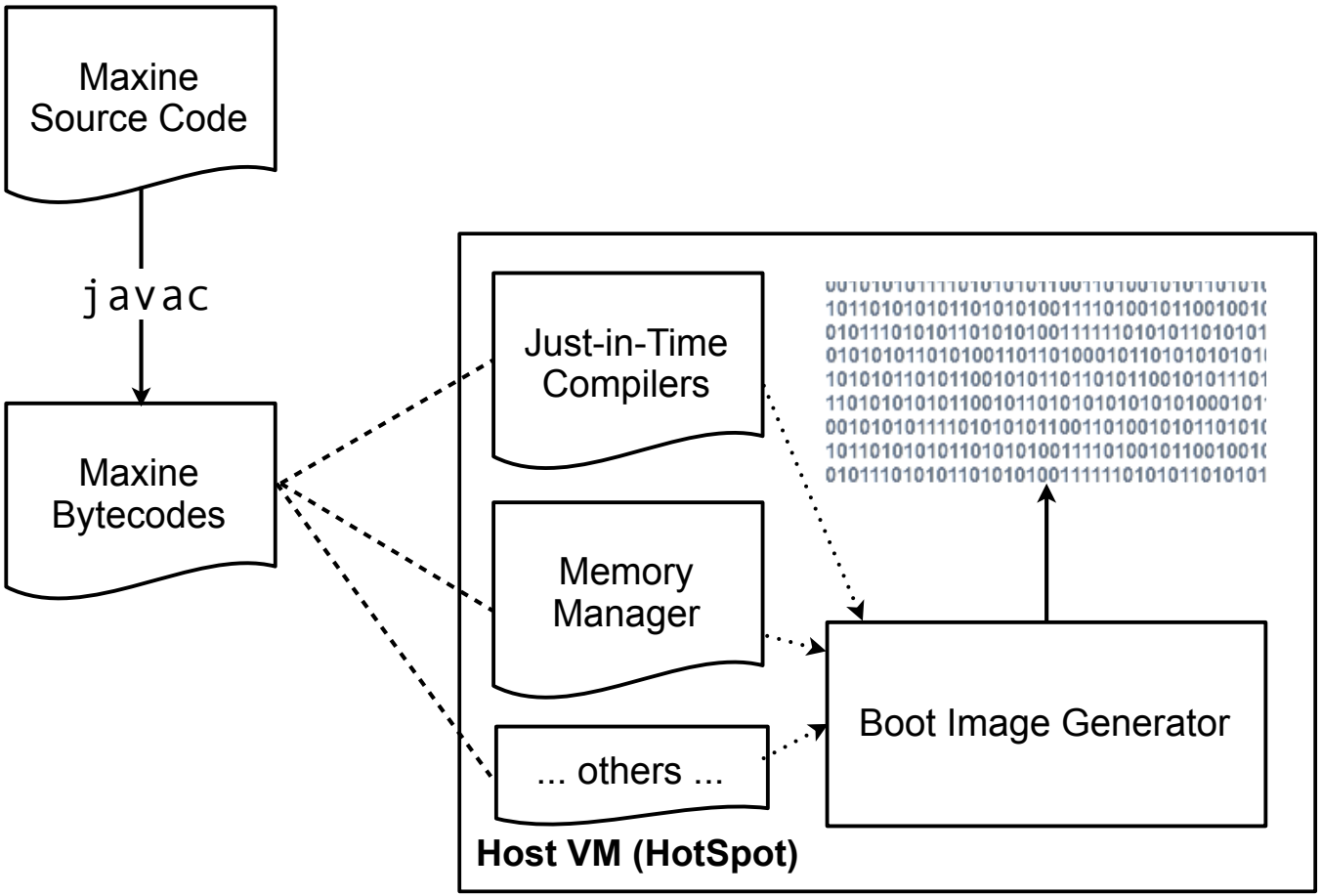
Bootstrapping Maxine



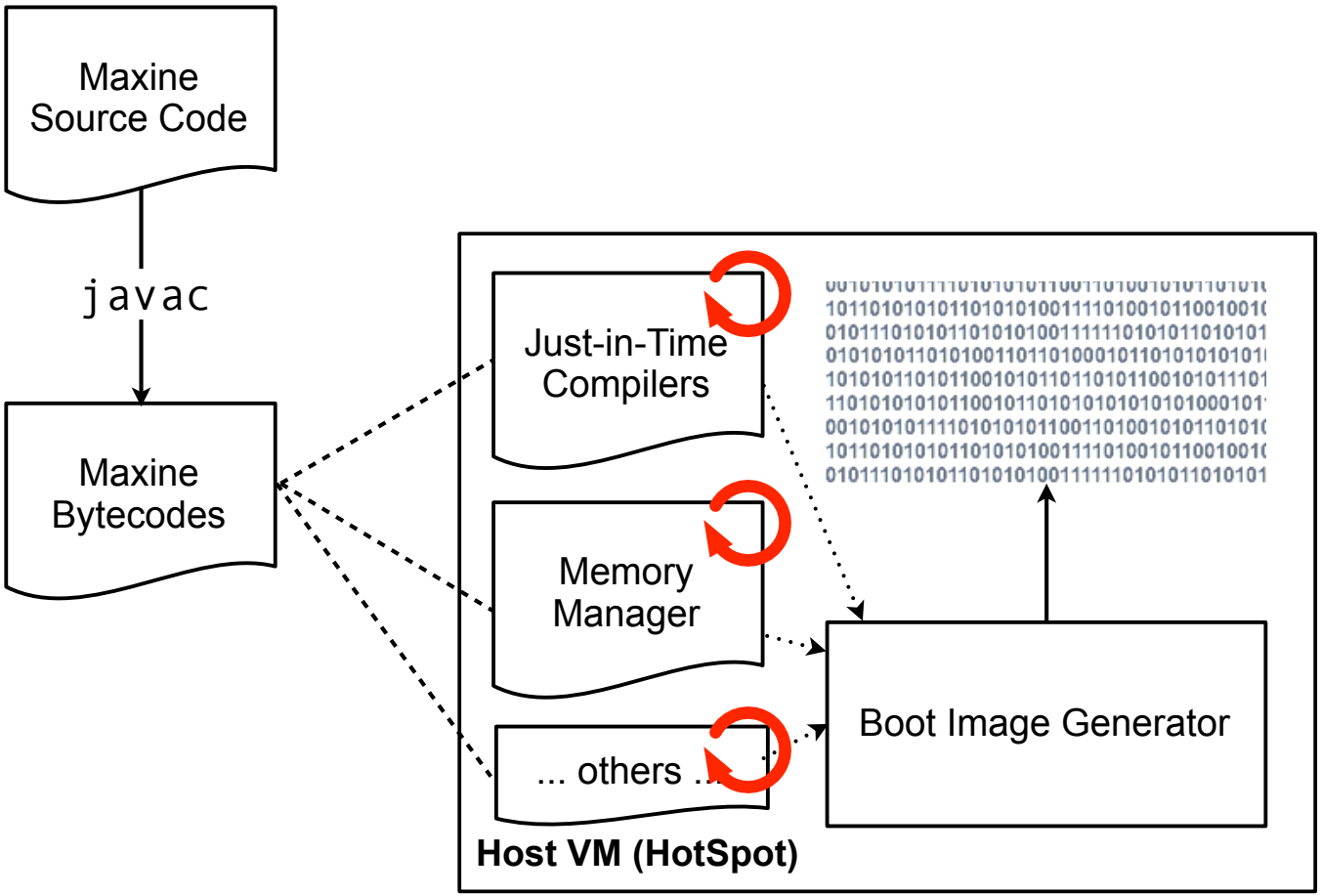
Bootstrapping Maxine



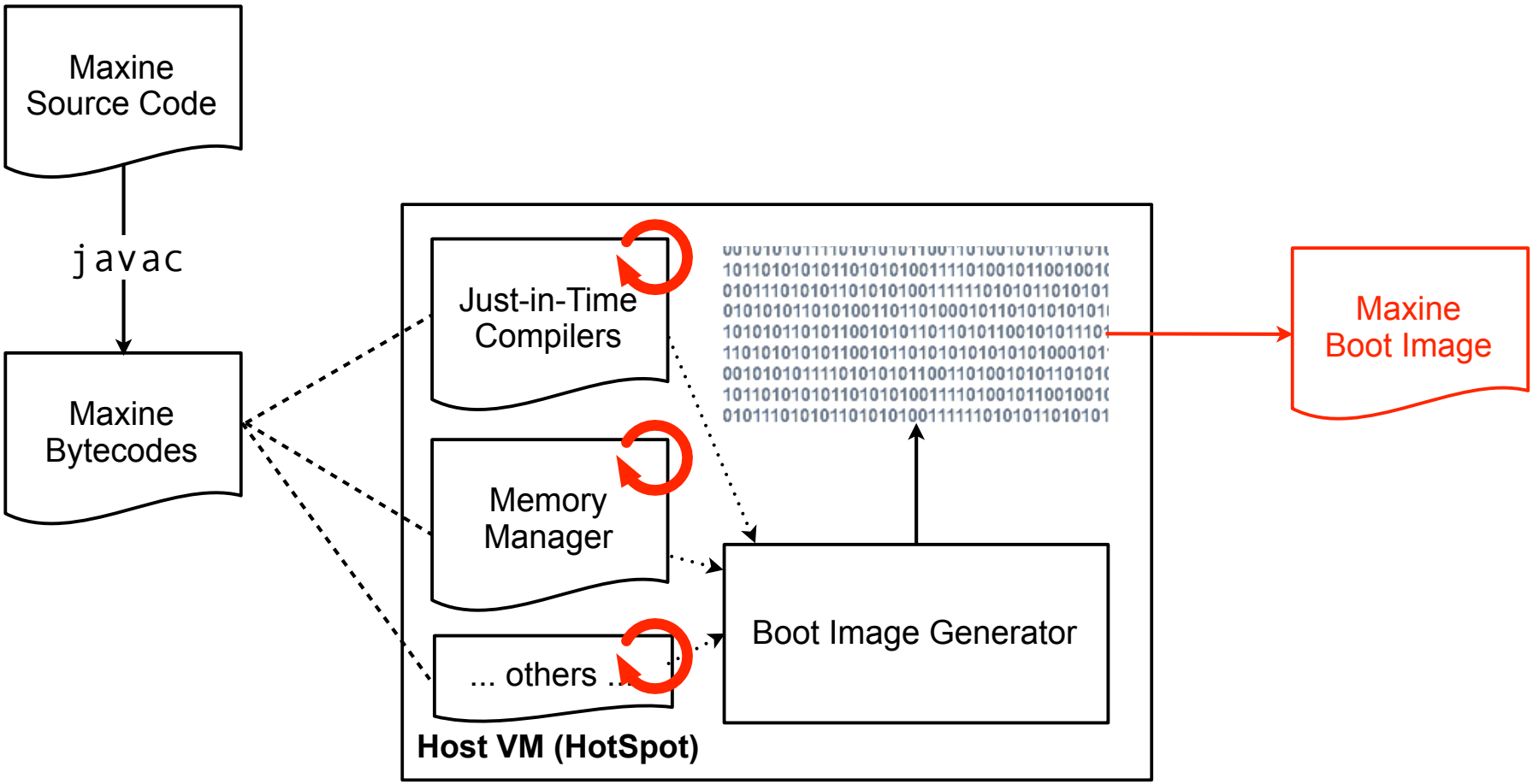
Bootstrapping Maxine



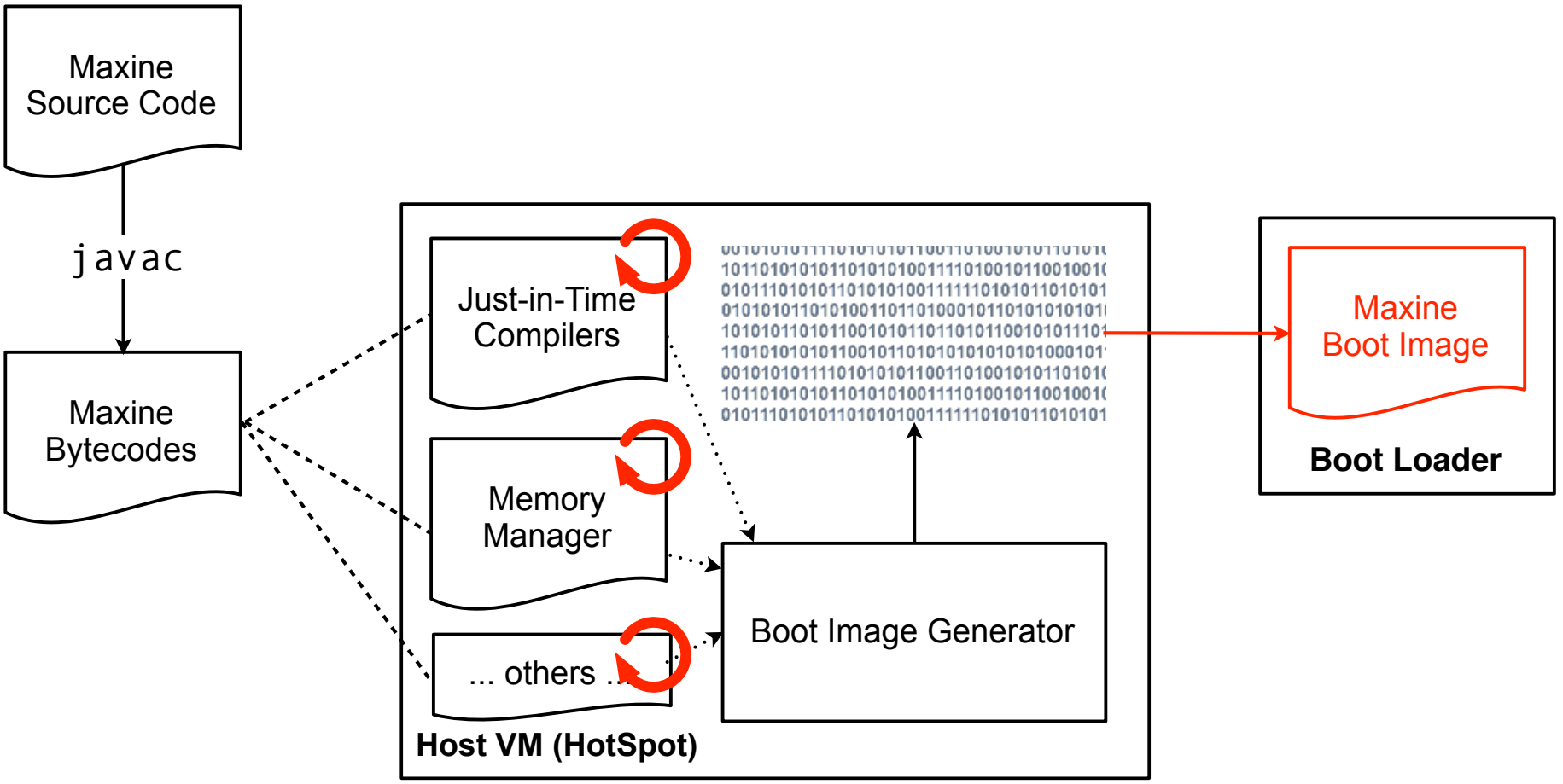
Bootstrapping Maxine



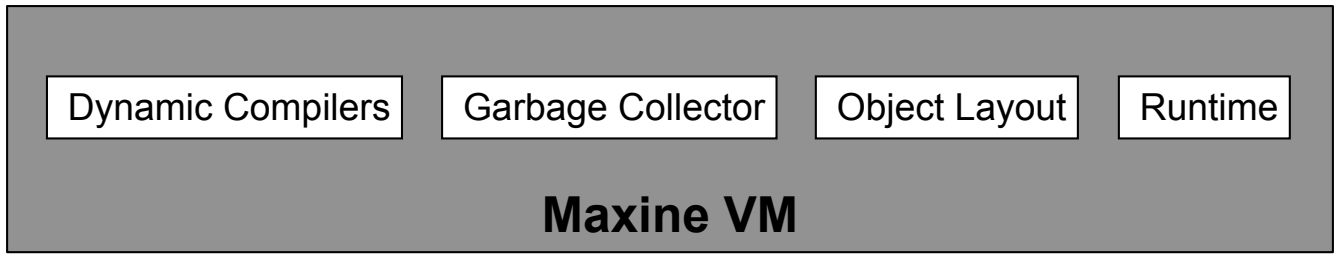
Bootstrapping Maxine



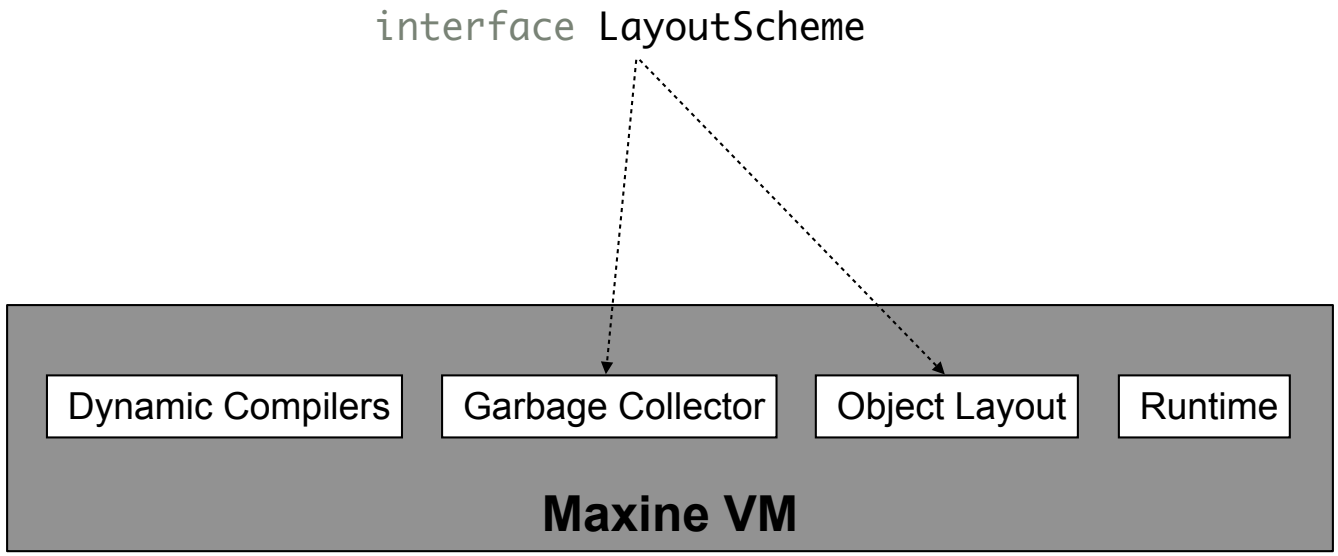
Bootstrapping Maxine



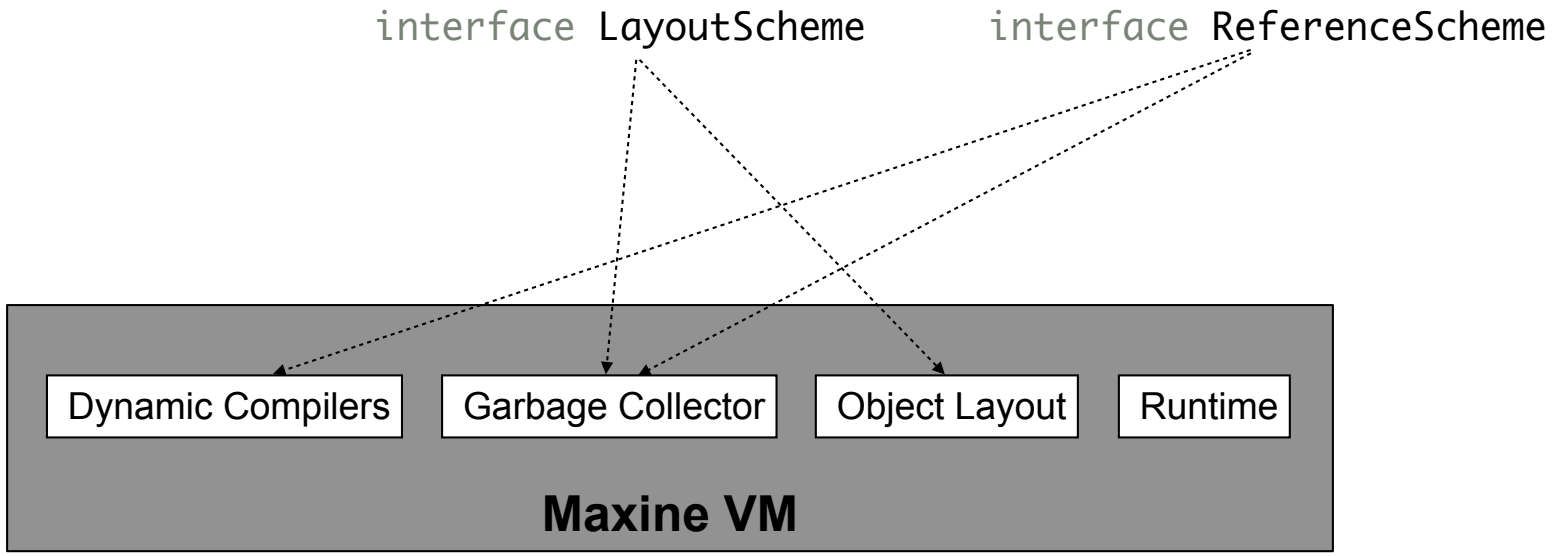
Maxine Scheme Abstractions



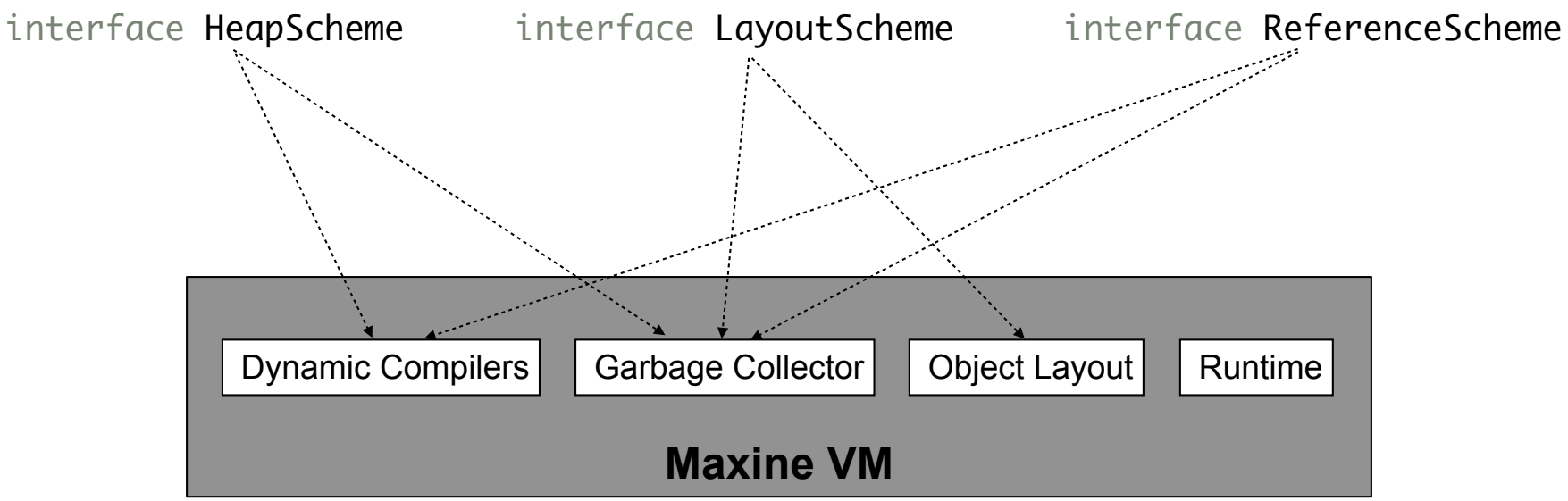
Maxine Scheme Abstractions



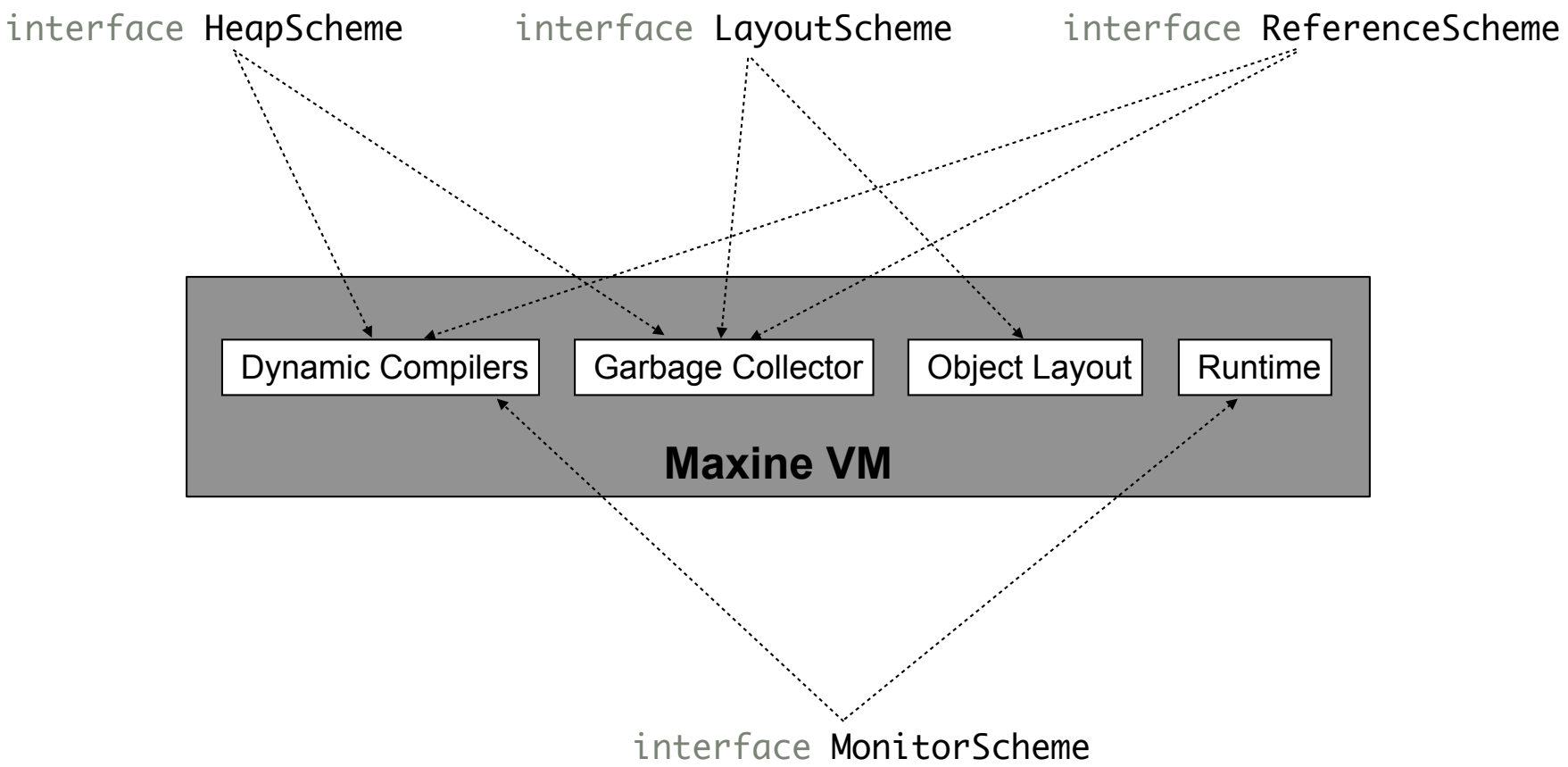
Maxine Scheme Abstractions



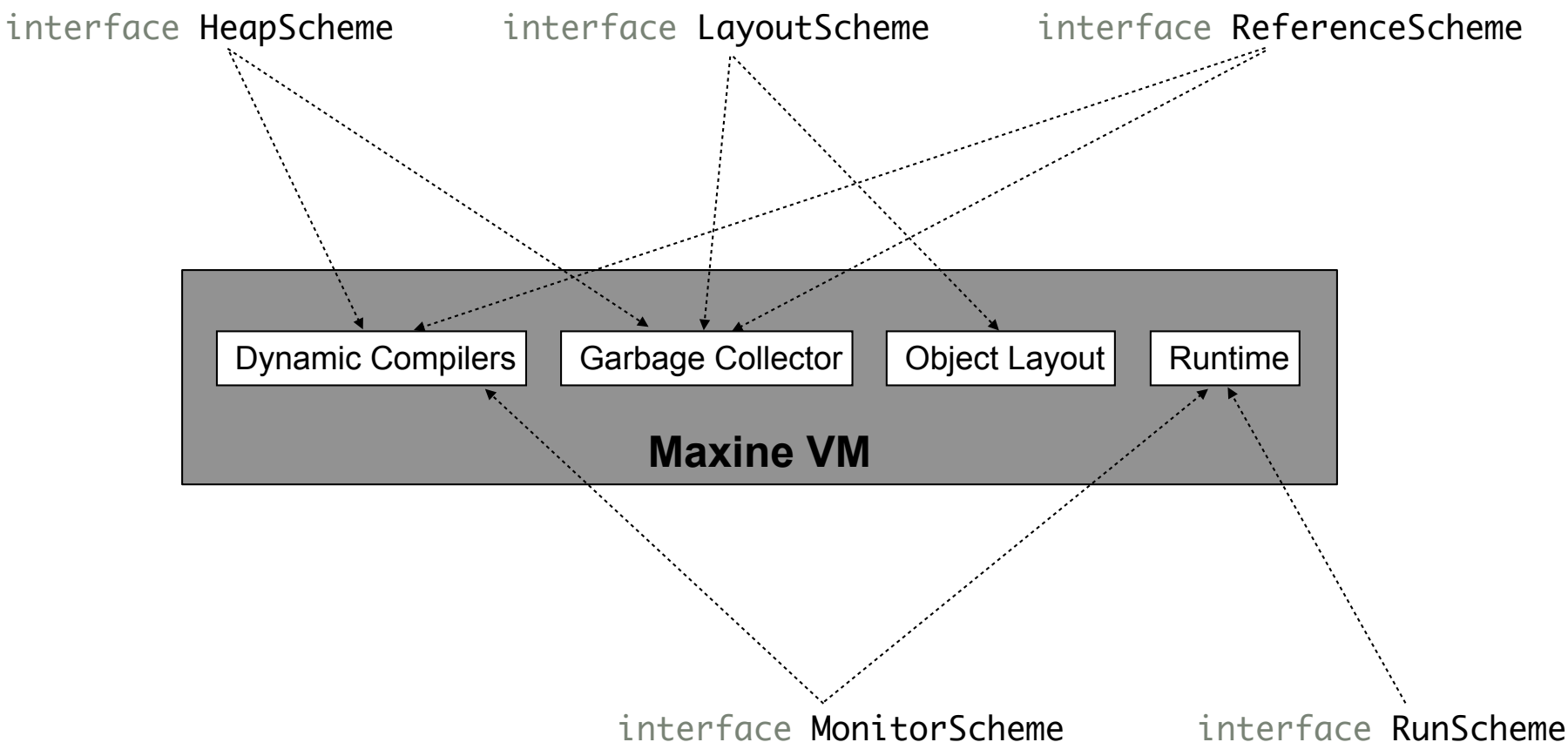
Maxine Scheme Abstractions



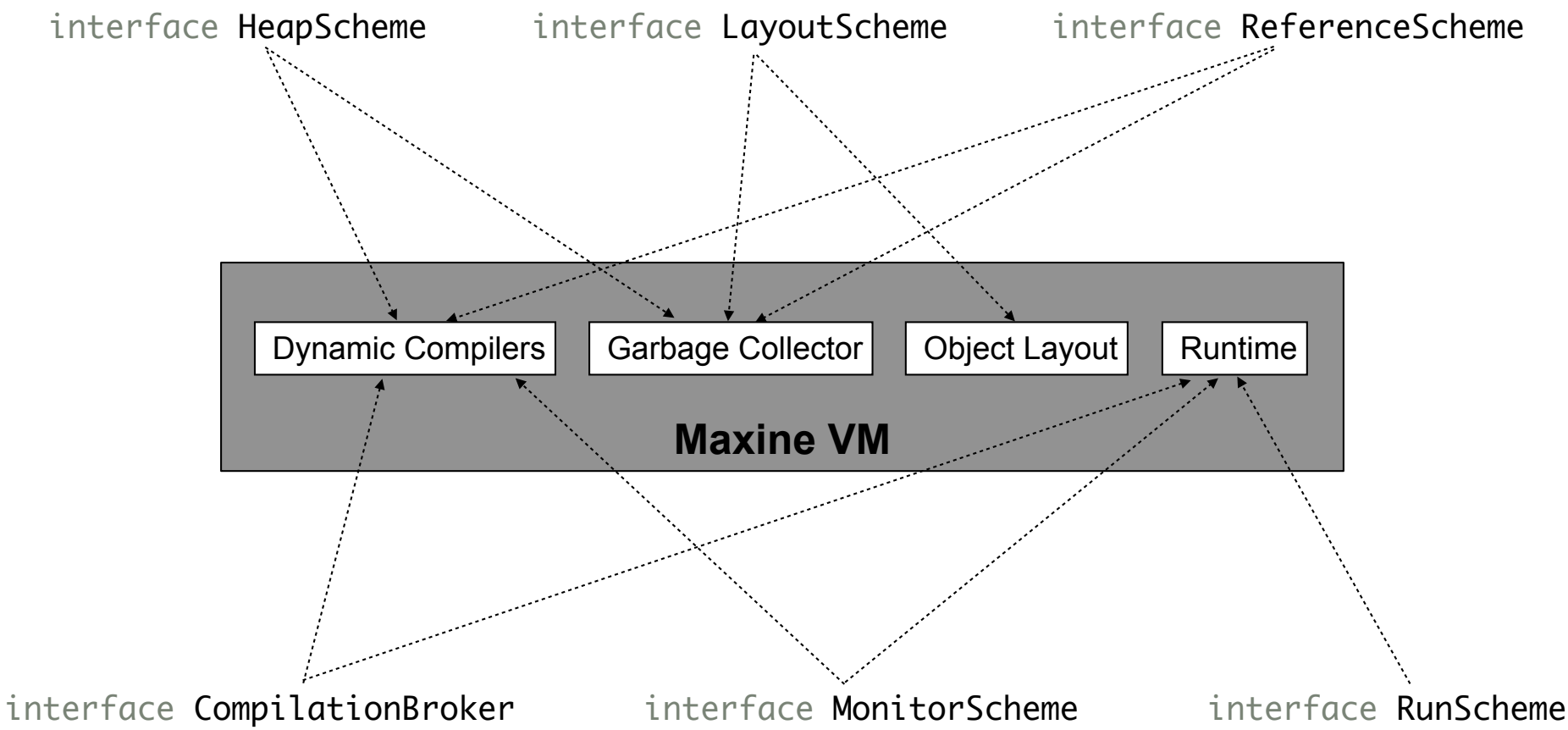
Maxine Scheme Abstractions



Maxine Scheme Abstractions



Maxine Scheme Abstractions



Example: ReferenceScheme

What you use in Maxine to access a field:

```
int x = Reference.fromJava(obj).readInt(24);
```

field offset (in bytes) is computed by `LayoutScheme`; we assume a constant offset in this example

Example: ReferenceScheme

What you use in Maxine to access a field:

```
int x = Reference.fromJava(obj).readInt(24);
```

field offset (in bytes) is computed by LayoutScheme; we assume a constant offset in this example

Actual call chain after forced inlining via @INLINE or @FOLD:

```
Reference ref = MaxineVM.vm.config.referenceScheme.toReference(obj);
Pointer ptr   = MaxineVM.vm.config.referenceScheme.toOrigin(ref);
int x = ptr.readInt(24);
```

Example: ReferenceScheme

What you use in Maxine to access a field:

```
int x = Reference.fromJava(obj).readInt(24);
```

field offset (in bytes) is computed by `LayoutScheme`; we assume a constant offset in this example

Actual call chain after forced inlining via @INLINE or @FOLD:

```
Reference ref = MaxineVM.vm.config.referenceScheme.toReference(obj);
Pointer ptr   = MaxineVM.vm.config.referenceScheme.toOrigin(ref);
int x = ptr.readInt(24);
```

All fields are marked @CONSTANT, so the compiler knows the concrete scheme implementation used:

```
class DirectReferenceScheme implements ReferenceScheme {
    @INTRINSIC(UNSAFE_CAST)
    native Reference toReference(Object origin);
    @INTRINSIC(UNSAFE_CAST)
    native Word toOrigin(Reference ref);
}
```

Example: ReferenceScheme

What you use in Maxine to access a field:

```
int x = Reference.fromJava(obj).readInt(24);
```

field offset (in bytes) is computed by LayoutScheme; we assume a constant offset in this example

Actual call chain after forced inlining via @INLINE or @FOLD:

```
Reference ref = MaxineVM.vm.config.referenceScheme.toReference(obj);
Pointer ptr   = MaxineVM.vm.config.referenceScheme.toOrigin(ref);
int x = ptr.readInt(24);
```

All fields are marked @CONSTANT, so the compiler knows the concrete scheme implementation used:

```
class DirectReferenceScheme implements ReferenceScheme {
    @INTRINSIC(UNSAFE_CAST)
    native Reference toReference(Object origin);
    @INTRINSIC(UNSAFE_CAST)
    native Word toOrigin(Reference ref);
}
```

Final machine code:

```
movl  [rax + 24], rbx
```

Baseline Compiler: T1X

job: assemble pre-generated bytecode templates

- written in Java
- compiled by optimising compiler at VM build time
- very small amount of glue assembly

Baseline Compiler: T1X

job: assemble pre-generated bytecode templates

- written in Java
- compiled by optimising compiler at VM build time
- very small amount of glue assembly

template code for integer array load (IALOAD)

```
@T1X_TEMPLATE(IALOAD)
static int iaload(@Slot(1) Object array, @Slot(0) int index) {
    ArrayAccess.checkIndex(array, index);
    return result = ArrayAccess.getInt(array, index);
}
```

```
@INLINE
static void checkIndex(Object array, int index) {
    int length = Layout.readArrayLength(Reference.fromJava(array));
    if (UnsignedMath.aboveOrEqual(index, length)) {
        throw Throw.arrayIndexOutOfBoundsException(array, index);
    }
}

@INLINE
static int getInt(Object array, int index) {
    return Layout.getInt(Reference.fromJava(array), index);
}
```

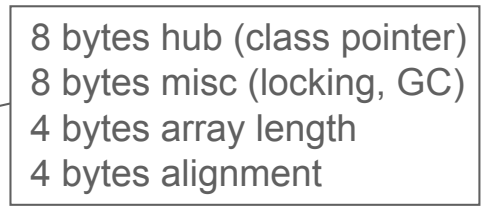
Baseline Compiler: T1X

instantiated template code for integer array load

```
    mov       rdi, [rsp + 16]
    mov       esi, [rsp]
-------------------------------------------
    mov       rax, rdi
    movsxd    rax, [rax + 16]
    cmp       esi, eax
    jb        L1
    call      Throw.indexOutOfBoundsException()
    nop
    mov       rdi, rax
    call      MaxRuntimeCalls.runtimeUnwindException
    nop
L1: movsxd    rsi, esi
    movsxd    rax, rdi[rsi * 4 + 24]
-------------------------------------------
    addq      rsp, 0x10
    mov       [rsp], eax
```

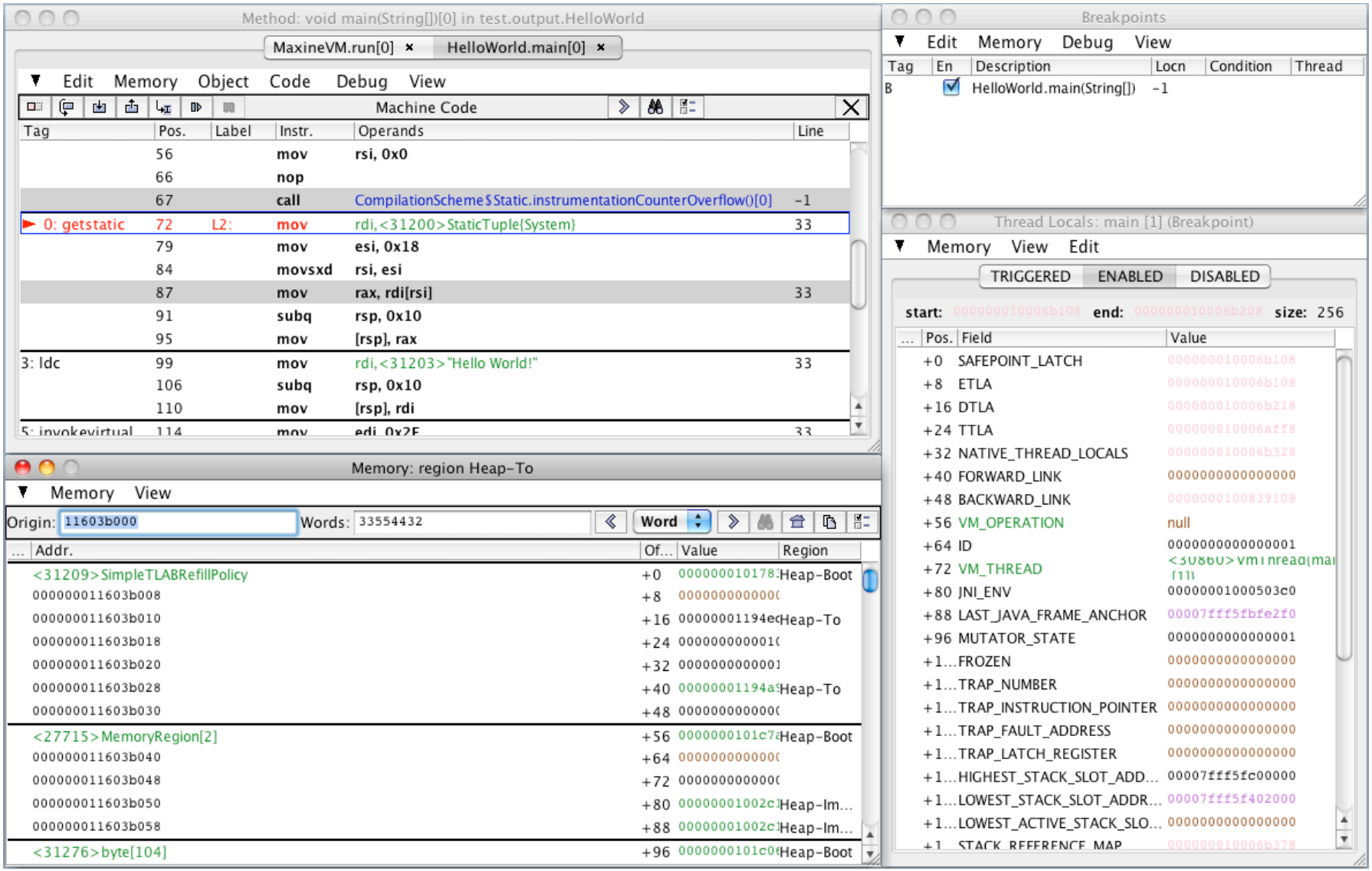
prologue
(loading of operand stack slots)

compiled template



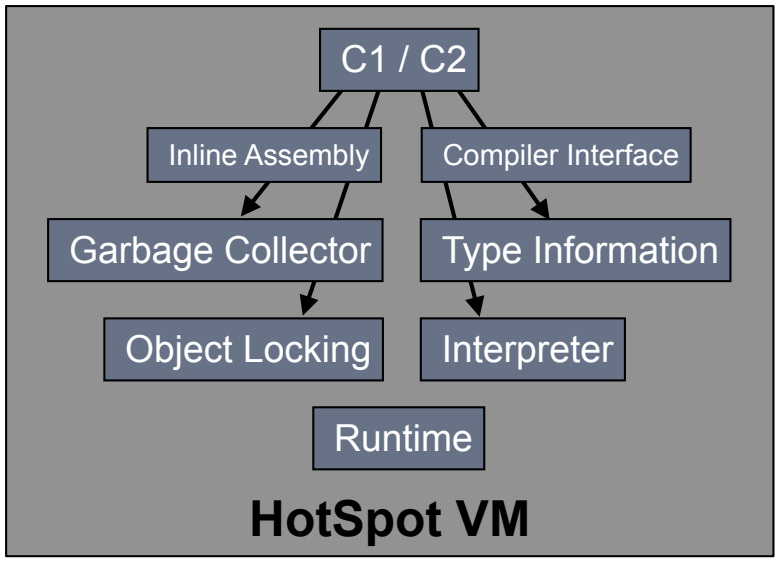
epilogue
(storing of operand stack slots)

Inspector

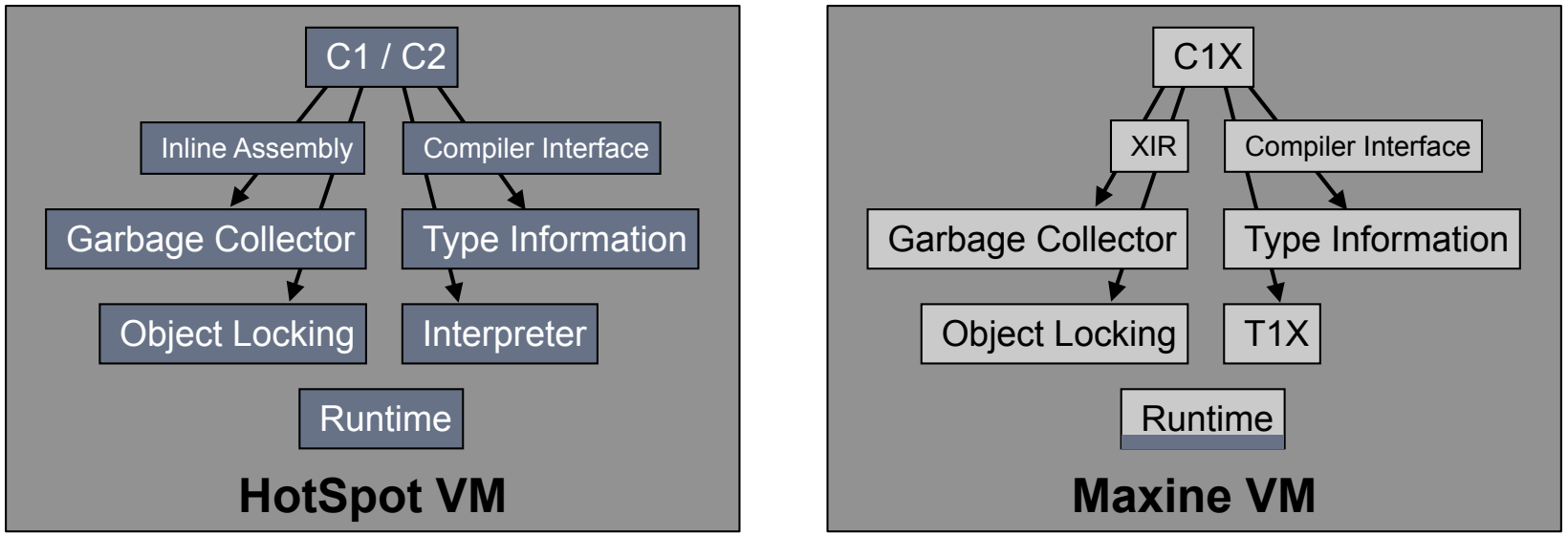

Method: void main(String[])[0] in test.output.HelloWorld
MaxineVM.run[0]
HelloWorld.main[0]
Edit Memory Object Code Debug View
Machine Code
Tag Pos. Label Instr. Operands Line
56 mov rsi, 0x0
66 nop
67 call CompilationScheme$Static.instrumentationCounterOverflow()[0] -1
0: getstatic 72 L2: mov rdi,<31200>StaticTuple{System} 33
79 mov esi, 0x18
84 movsxd rsi, esi
87 mov rax, rdi[rsi] 33
91 subq rsp, 0x10
95 mov [rsp], rax
3: ldc 99 mov rdi,<31203>"Hello World!" 33
106 subq rsp, 0x10
110 mov [rsp], rdi
Memory: region Heap-To
Memory View
Origin: 11603b000 Words: 33554432 Word
Addr. Of... Value Region
<31209>SimpleTLABRefillPolicy +0 Heap-Boot
000000011603b008 +8
000000011603b010 +16 Heap-To
000000011603b018 +24
000000011603b020 +32
000000011603b028 +40 Heap-To
000000011603b030 +48
<27715>MemoryRegion[2] +56 Heap-Boot
000000011603b040 +64
000000011603b048 +72
000000011603b050 +80 Heap-Im...
000000011603b058 +88 Heap-Im...
<31276>byte[104] +96 Heap-Boot
Breakpoints
Edit Memory Debug View
Tag En Description Locn Condition Thread
B HelloWorld.main(String[]) -1
Thread Locals: main [1] (Breakpoint)
Memory View Edit
TRIGGERED ENABLED DISABLED
start: 000000010006b108 end: 000000010006b208 size: 256
... Pos. Field Value
+0 SAFEPOINT_LATCH 000000010006b108
+8 ETLA 000000010006b108
+16 DTLA 000000010006b218
+24 TTLA 000000010006aff8
+32 NATIVE_THREAD_LOCALS 000000010006b328
+40 FORWARD_LINK 0000000000000000
+48 BACKWARD_LINK 0000000100839108
+56 VM_OPERATION null
+64 ID 0000000000000001
+72 VM_THREAD
+80 JNI_ENV 00000001000503c0
+88 LAST_JAVA_FRAME_ANCHOR 00007fff5fbfe2f0
+96 MUTATOR_STATE 0000000000000001
+1... FROZEN 0000000000000000
+1... TRAP_NUMBER 0000000000000000
+1... TRAP_INSTRUCTION_POINTER 0000000000000000
+1... TRAP_FAULT_ADDRESS 0000000000000000
+1... TRAP_LATCH_REGISTER 0000000000000000
+1... HIGHEST_STACK_SLOT_ADD... 00007fff5fc00000
+1... LOWEST_STACK_SLOT_ADDR... 00007fff5f402000
+1... LOWEST_ACTIVE_STACK_SLO... 0000000000000000

Optimising Compilers: C1X and Graal

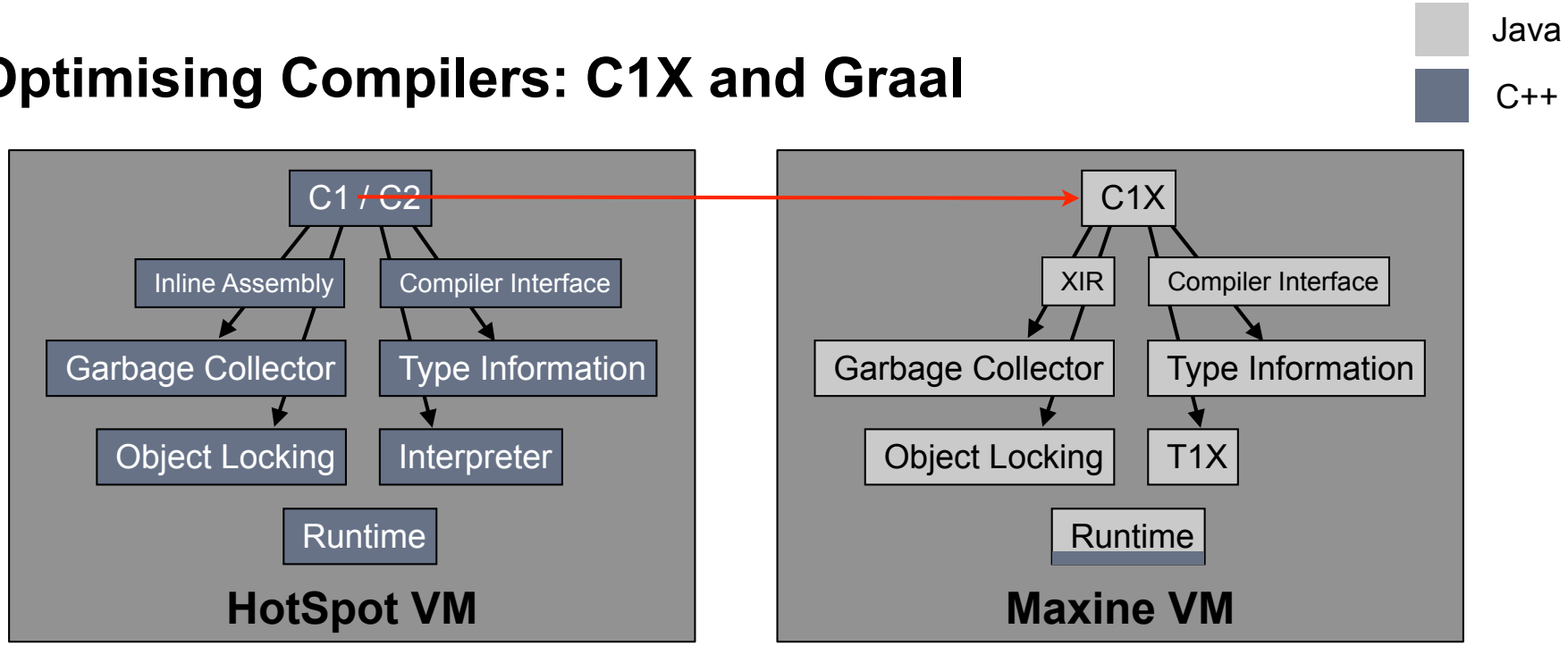
C++



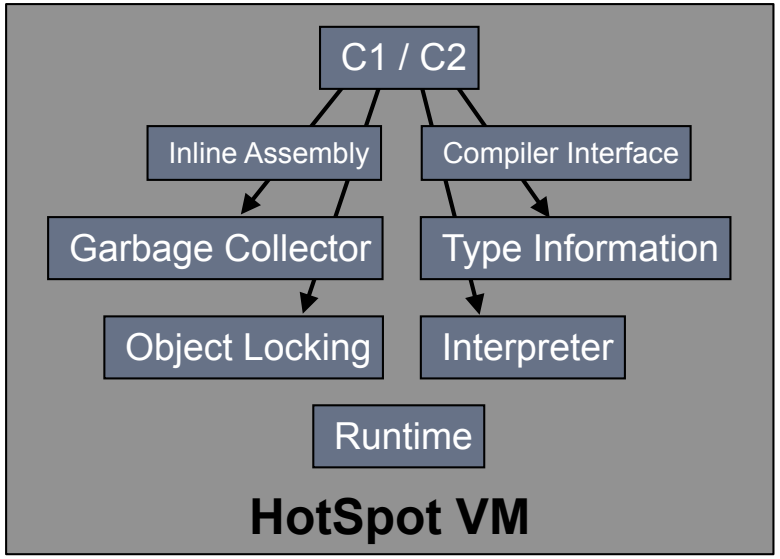
Optimising Compilers: C1X and Graal

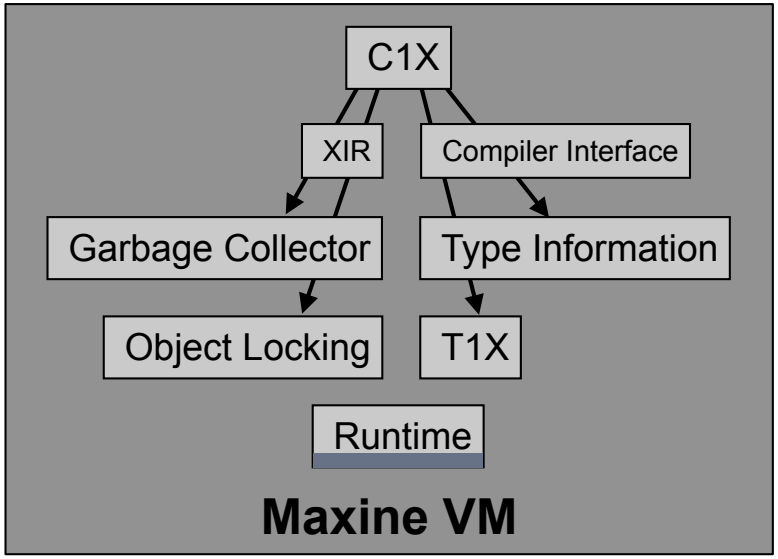


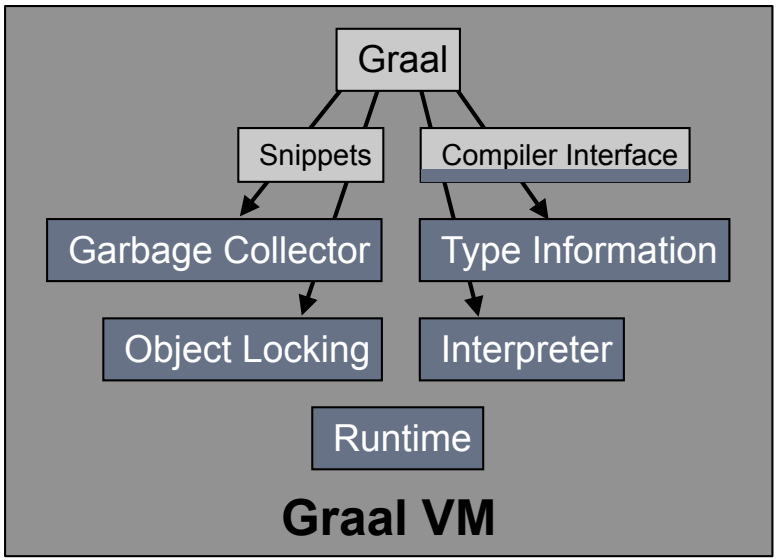
Optimising Compilers: C1X and Graal



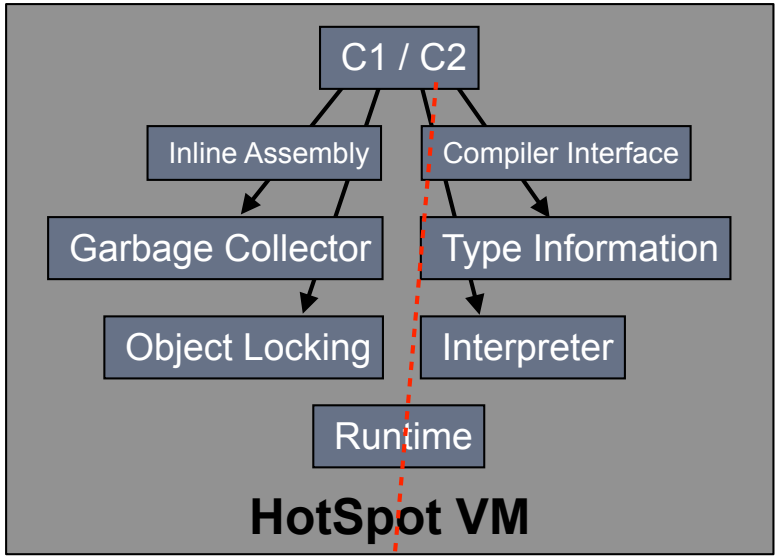
Optimising Compilers: C1X and Graal

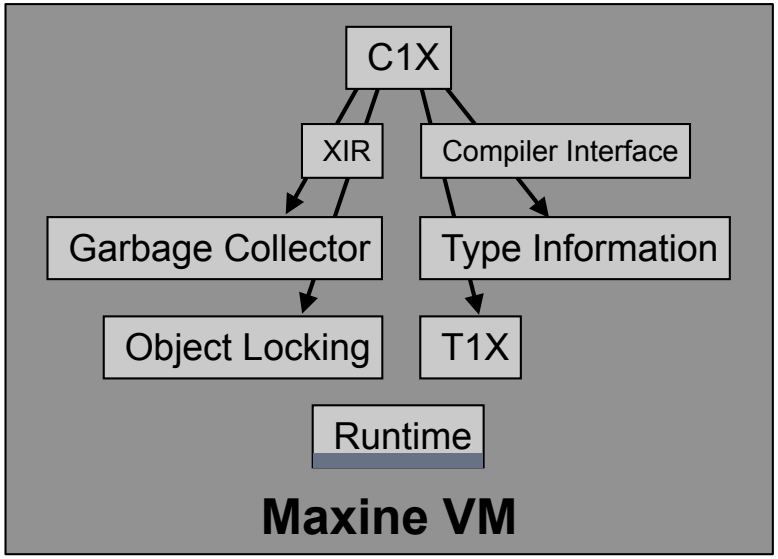


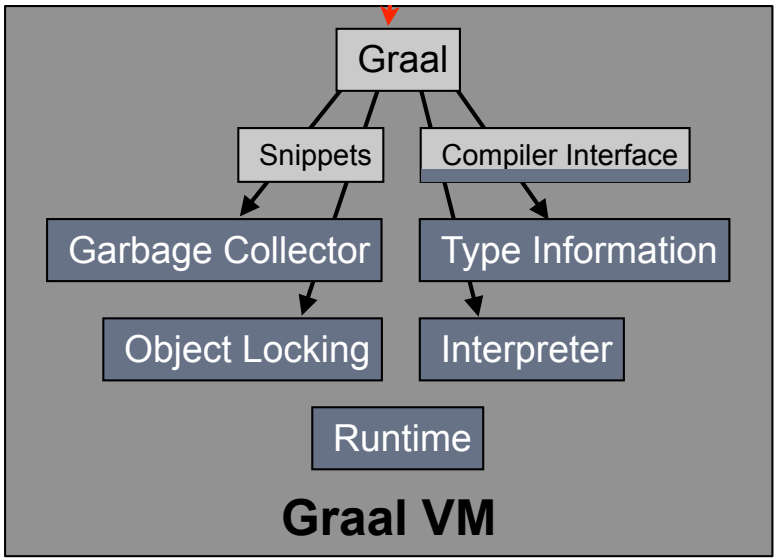




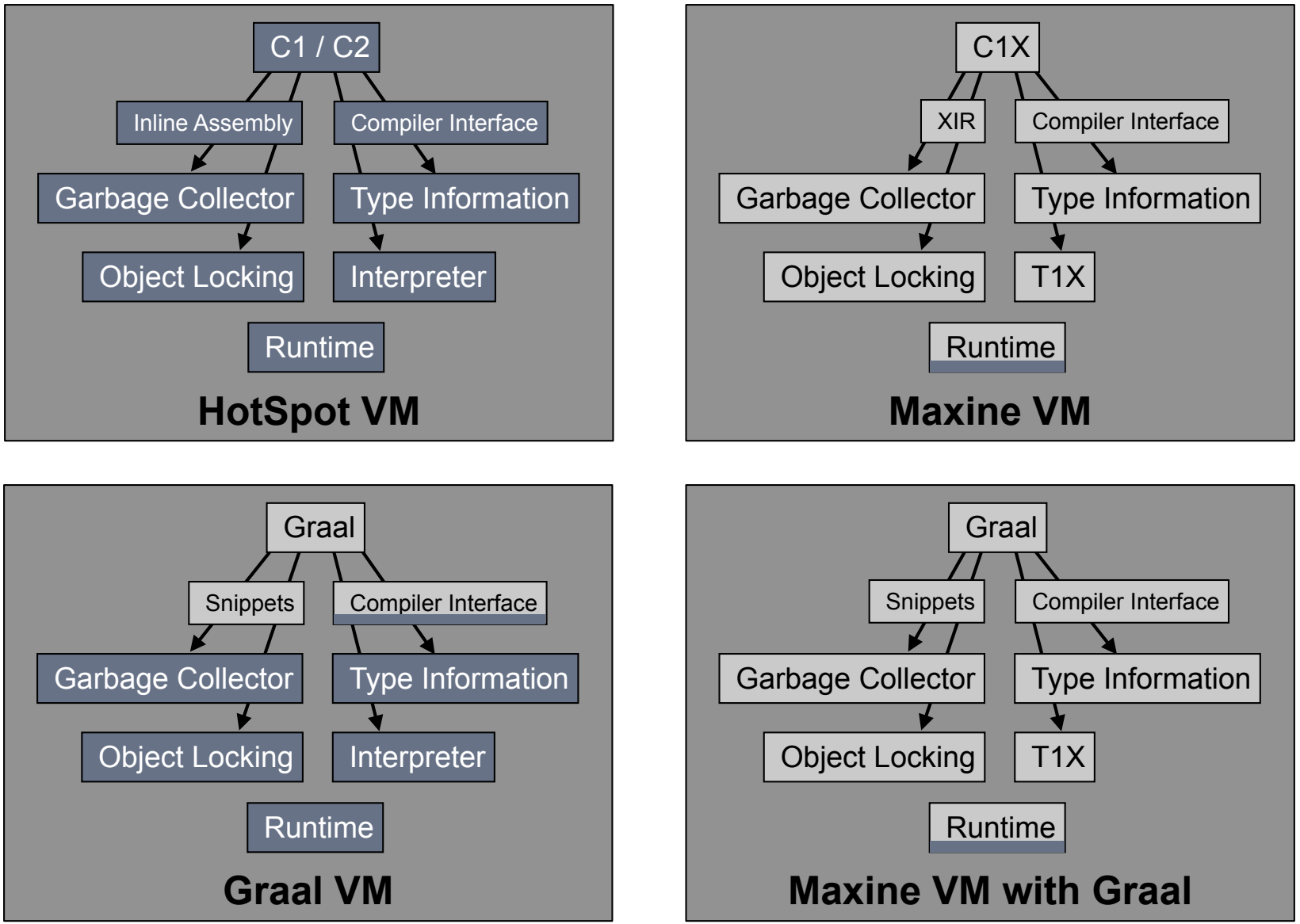
Optimising Compilers: C1X and Graal



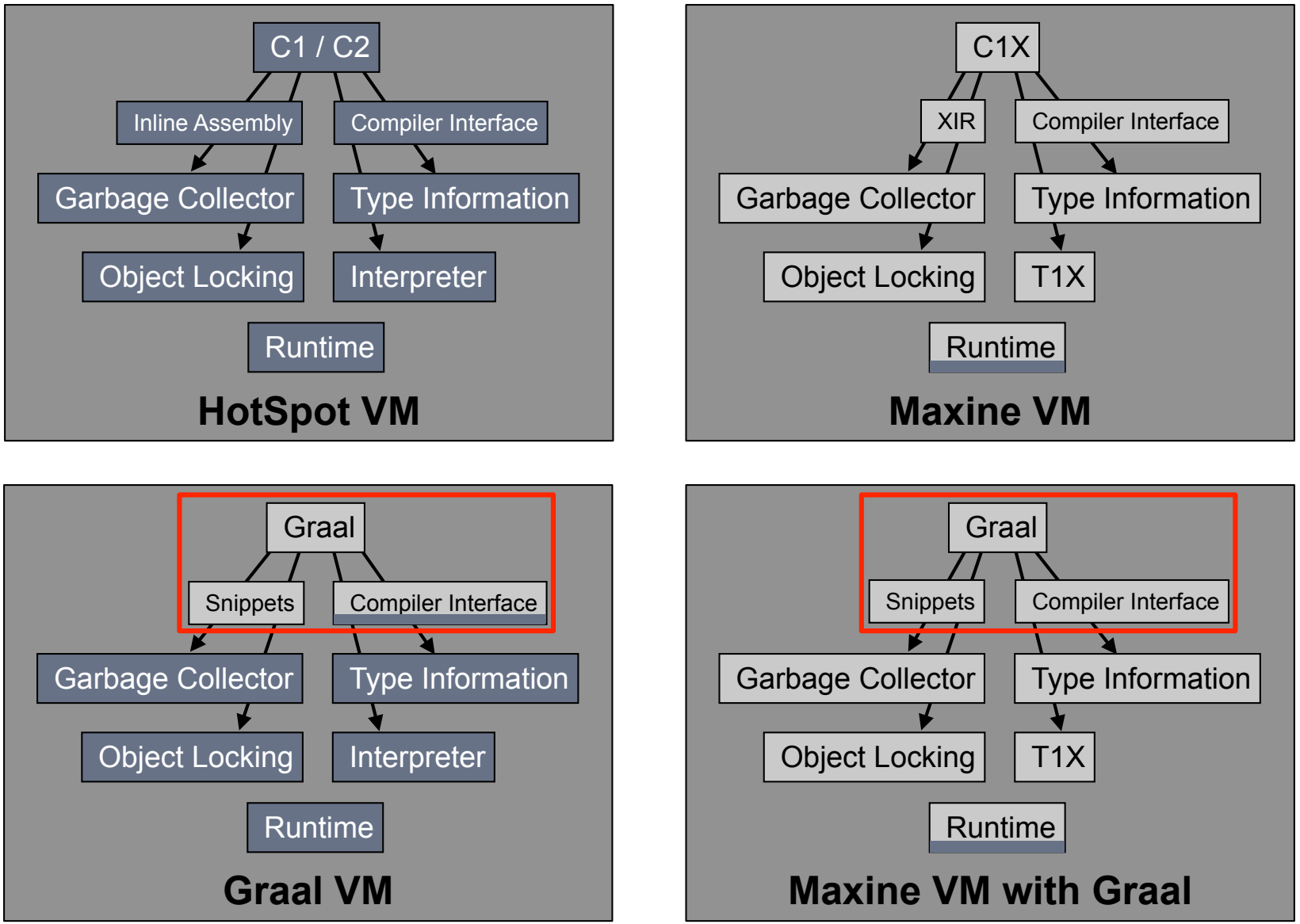




Optimising Compilers: C1X and Graal



Optimising Compilers: C1X and Graal



Graal

Vision Statement

Create an ***extensible***, ***modular***,
dynamic, *and* ***aggressive*** *compiler using*
object-oriented *and* ***reflective*** *Java programming,*
a ***graph-based***
and ***visualizable*** *intermediate representation,*
and Java ***snippets***.

-- Thomas Würthinger

Collaboration with Johannes-Kepler-Universität Linz, Austria

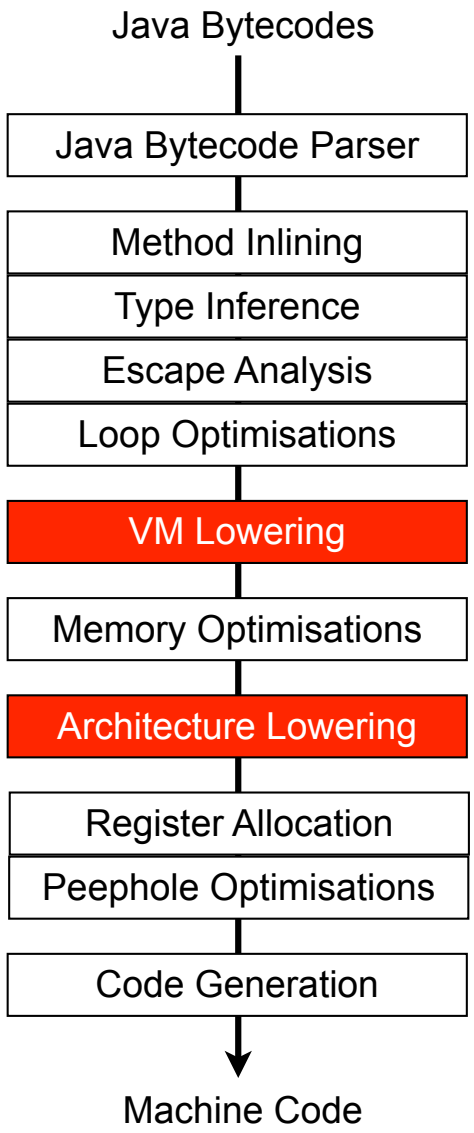
Lukas Stadler, Gilles Duboscq, Christian Häubl, Prof. Hanspeter Mössenböck

Home page: http://openjdk.java.net/projects/graal/

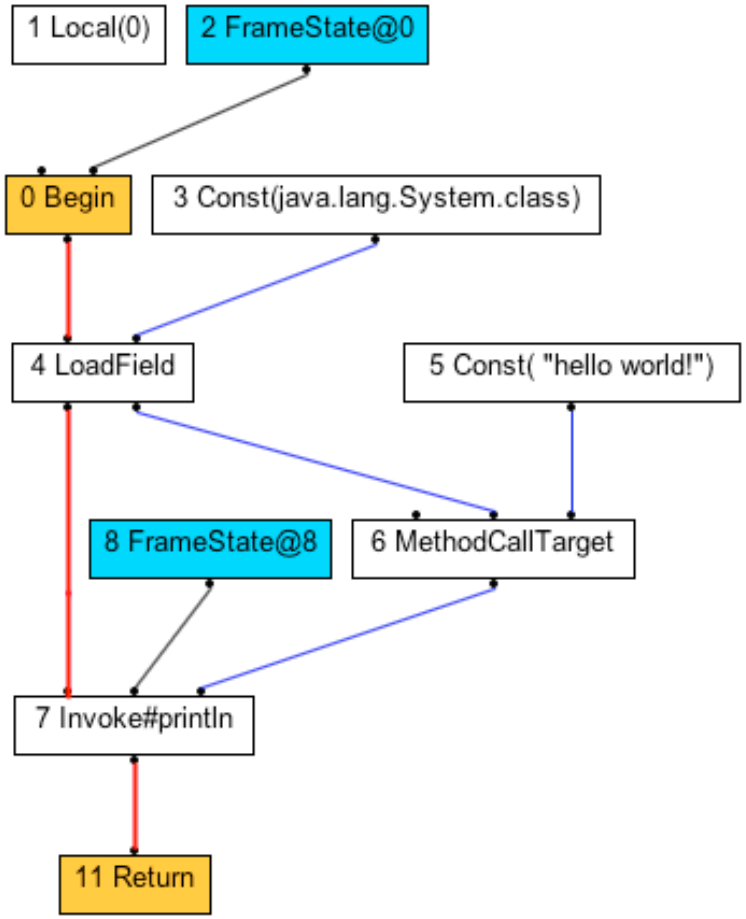
Source (GPLv2): http://hg.openjdk.java.net/graal/graal/

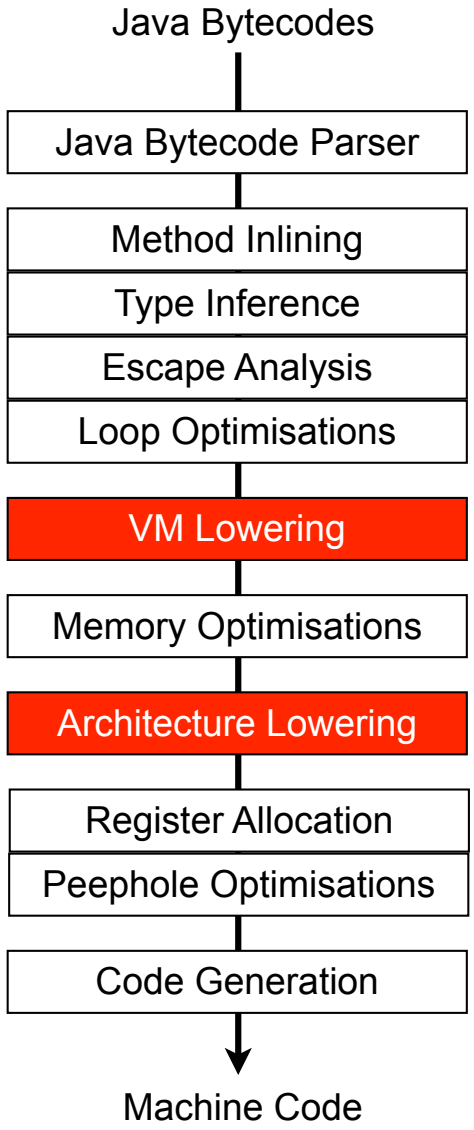
Mailing list: graal-dev@openjdk.java.net

Graal

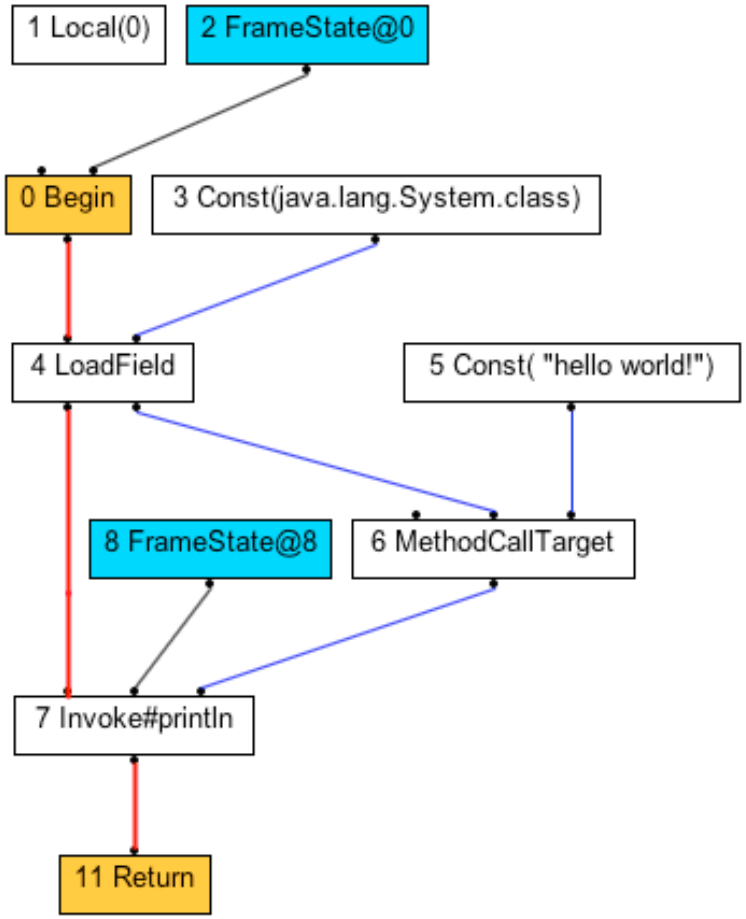

Java Bytecodes
Java Bytecode Parser
Method Inlining
Type Inference
Escape Analysis
Loop Optimisations
VM Lowering
Memory Optimisations
Architecture Lowering
Register Allocation
Peephole Optimisations
Code Generation
Machine Code

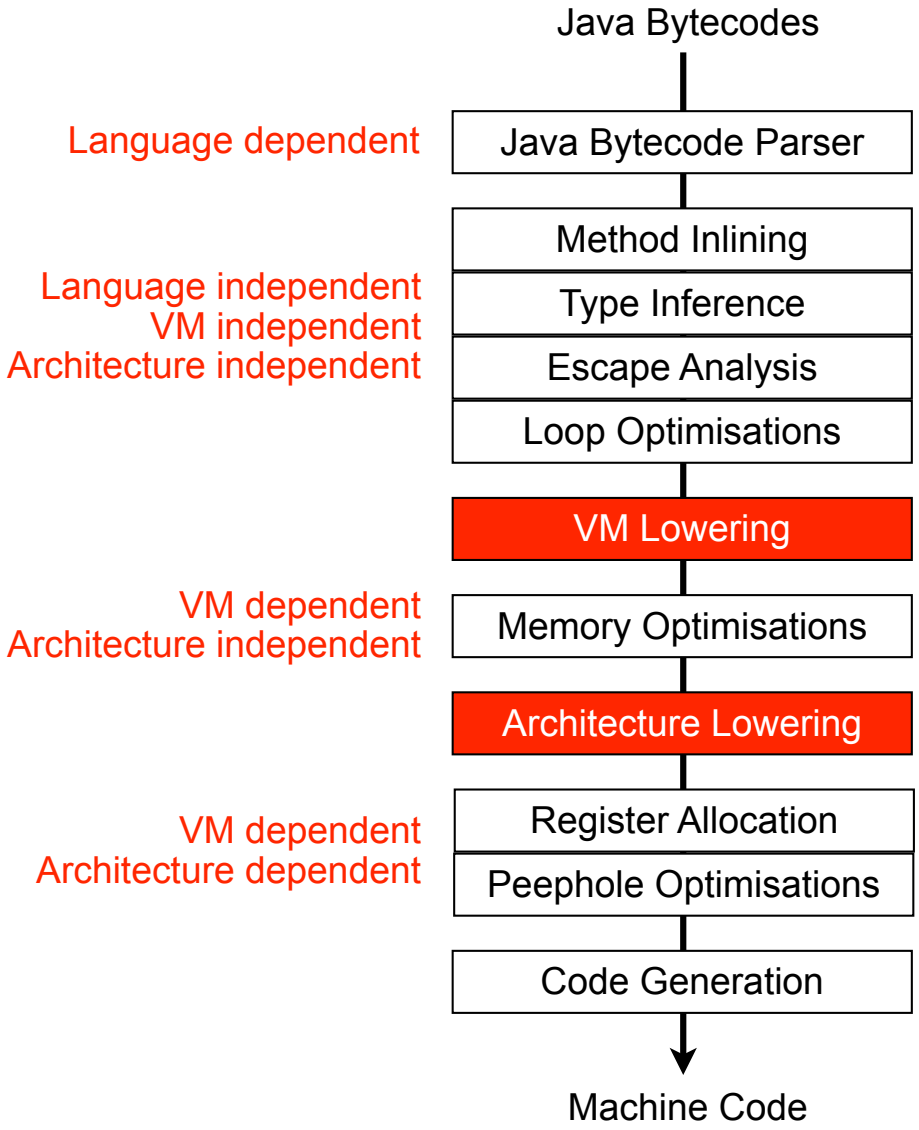
Graal


1 Local(0)
2 FrameState@0
0 Begin
3 Const(java.lang.System.class)
4 LoadField
5 Const("hello world!")
8 FrameState@8
6 MethodCallTarget
7 Invoke#println
11 Return

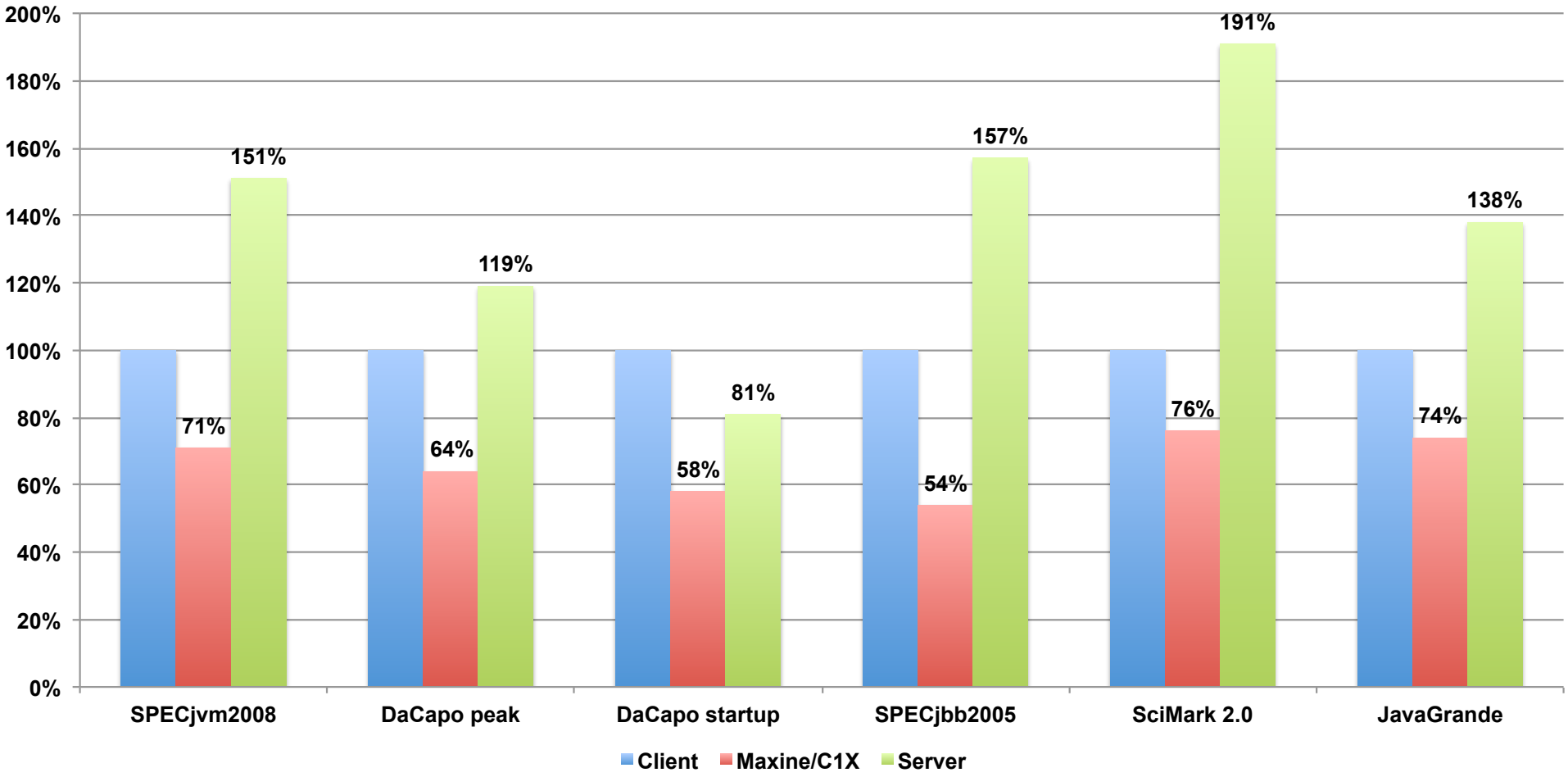

Java Bytecodes
Java Bytecode Parser
Method Inlining
Type Inference
Escape Analysis
Loop Optimisations
VM Lowering
Memory Optimisations
Architecture Lowering
Register Allocation
Peephole Optimisations
Code Generation
Machine Code

Graal


1 Local(0)
2 FrameState@0
0 Begin
3 Const(java.lang.System.class)
4 LoadField
5 Const("hello world!")
8 FrameState@8
6 MethodCallTarget
7 Invoke#println
11 Return

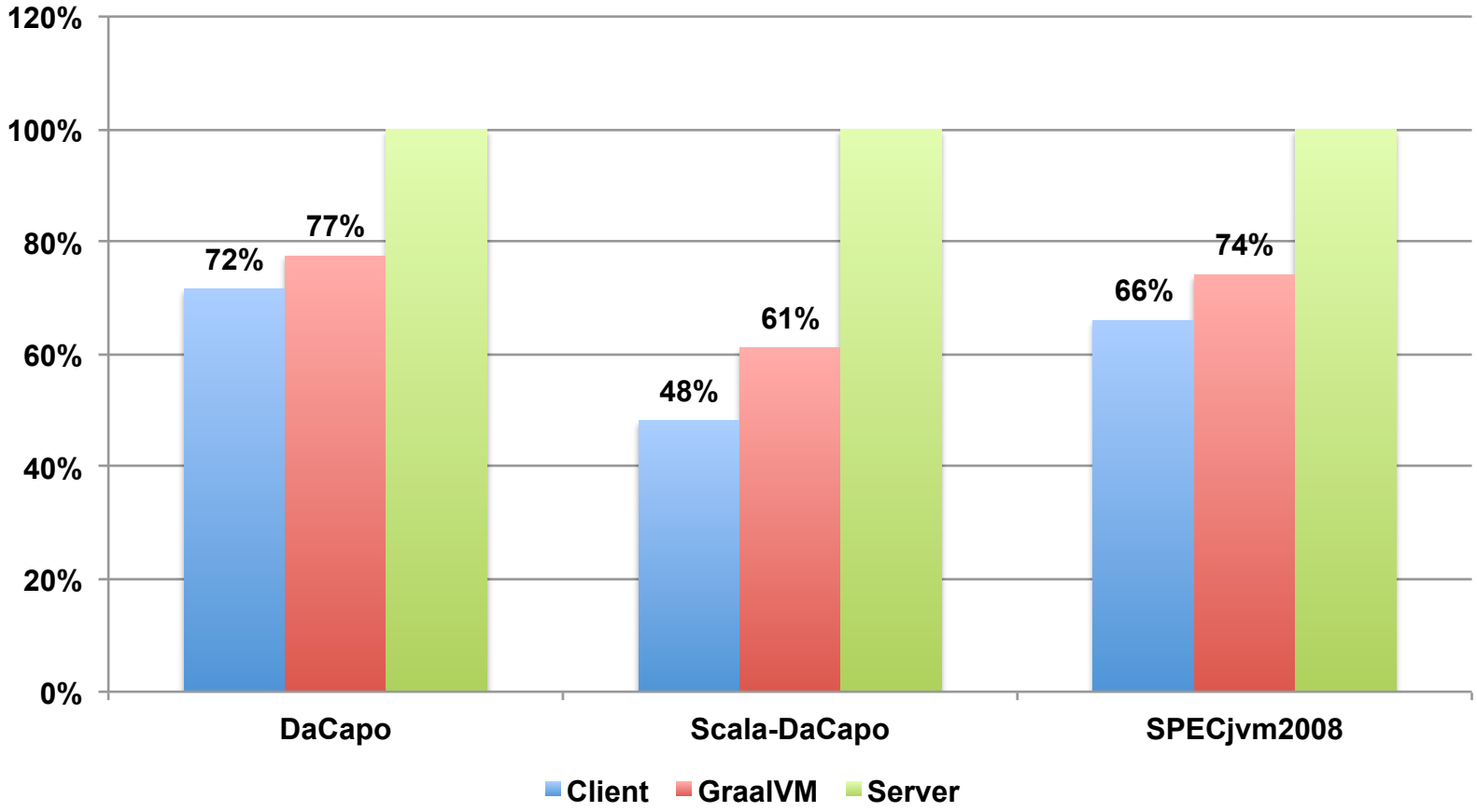

Java Bytecodes
Language dependent
Java Bytecode Parser
Method Inlining
Language independent
VM independent
Architecture independent
Type Inference
Escape Analysis
Loop Optimisations
VM Lowering
VM dependent
Architecture independent
Memory Optimisations
Architecture Lowering
VM dependent
Architecture dependent
Register Allocation
Peephole Optimisations
Code Generation
Machine Code

Performance: Maxine/C1X



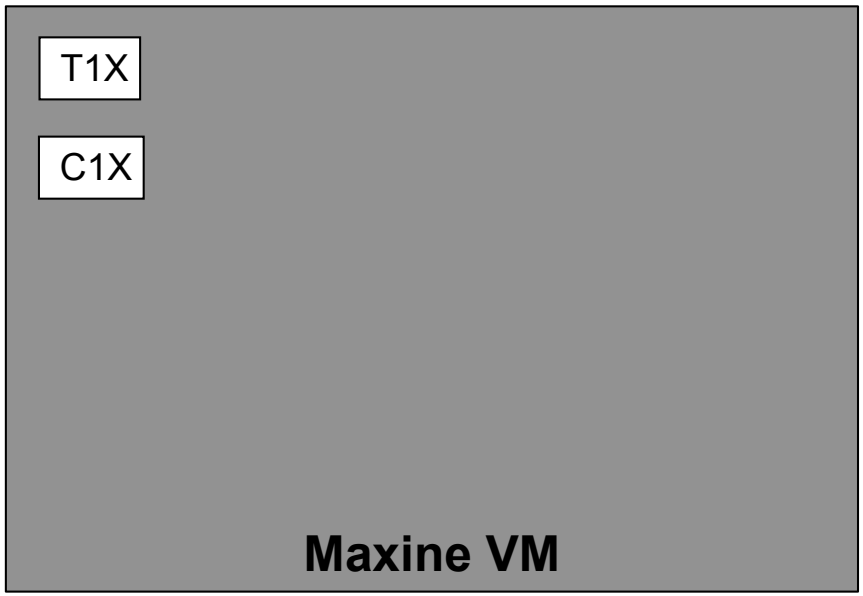
2-socket dual core AMD Opteron 2214, 2.2 GHz, 4 total cores
4 GByte main memory
Oracle Enterprise Linux, version 2.6.18-238.9.1.0.1.e15

Performance: GraalVM

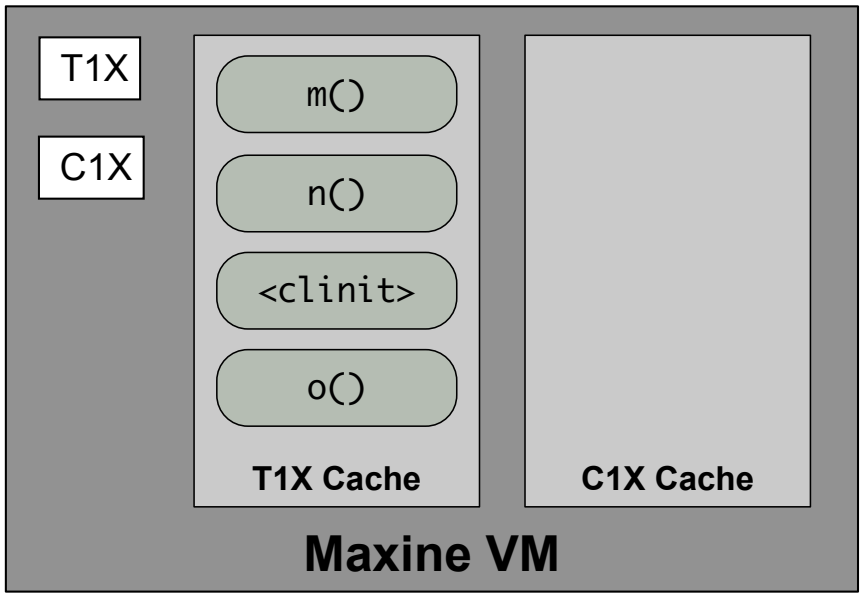


Intel(R) Core(TM) i5-750 @ 2.67GHz (4 cores)
8 GByte main memory
Ubuntu Linux 11.10, kernel 3.0.0-12

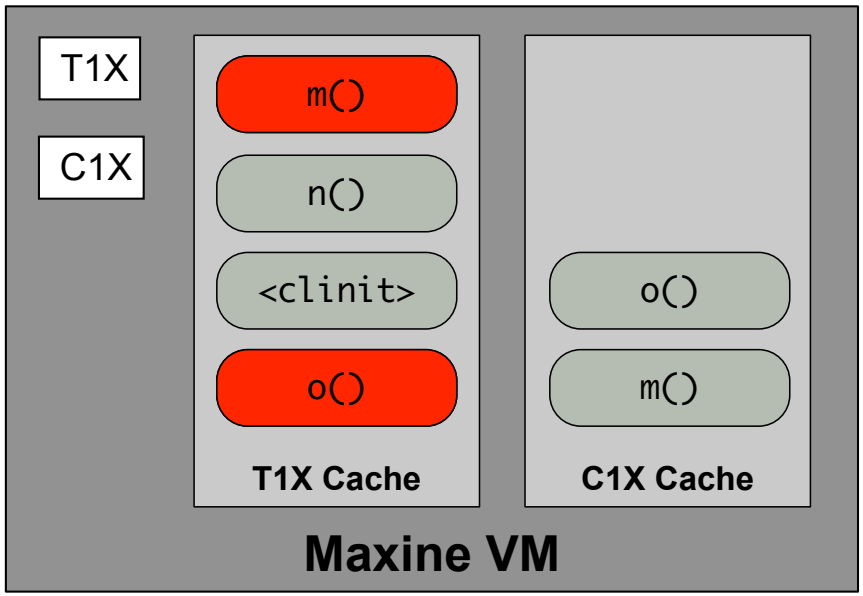
Maxine: Code Cache Management



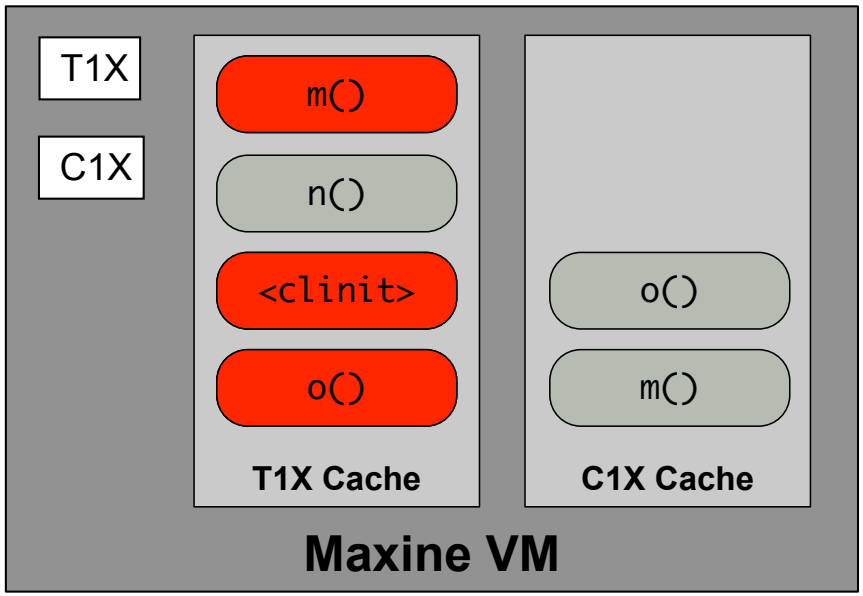
Maxine: Code Cache Management



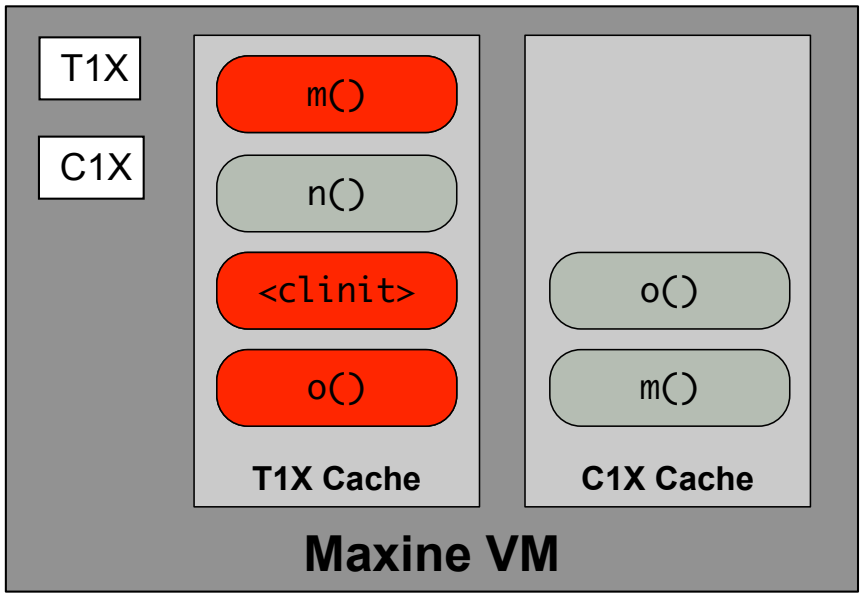
Maxine: Code Cache Management

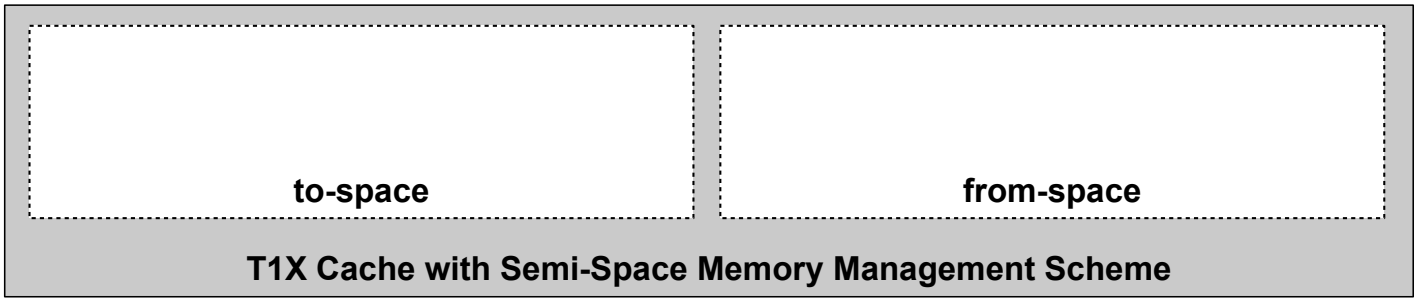


Maxine: Code Cache Management

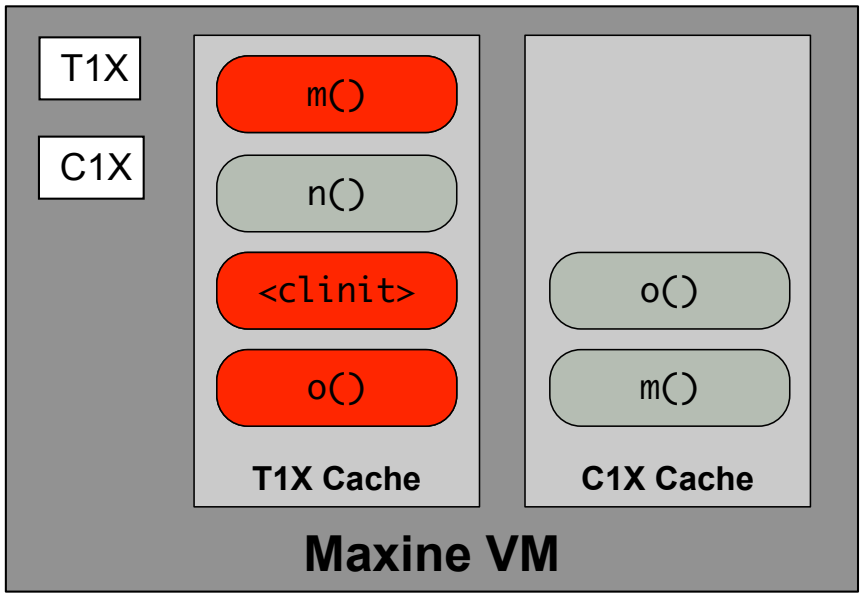


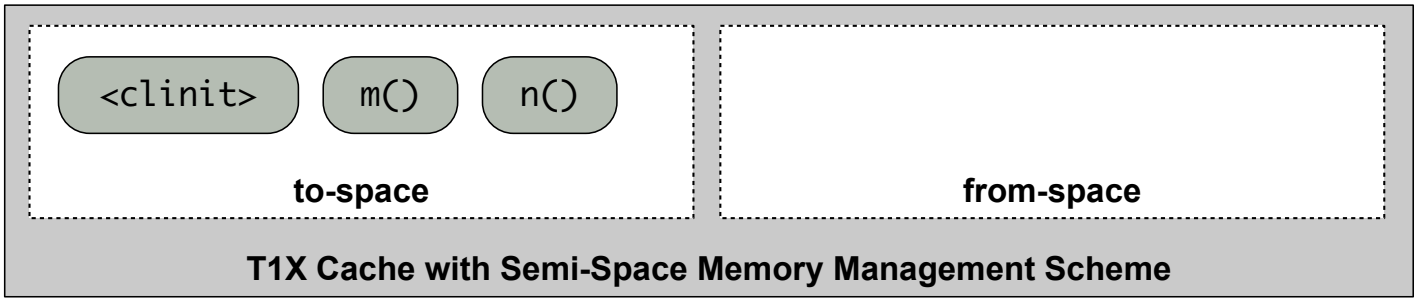
Maxine: Code Cache Management



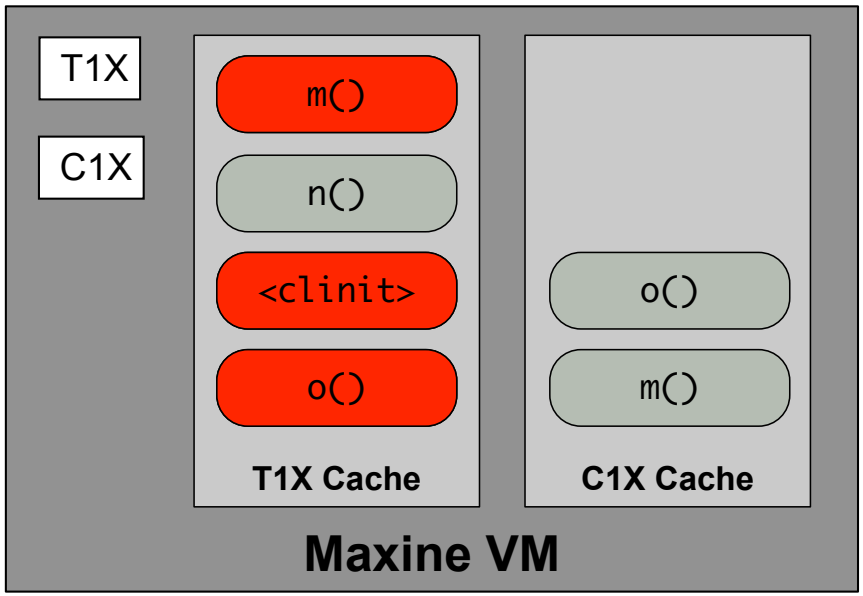


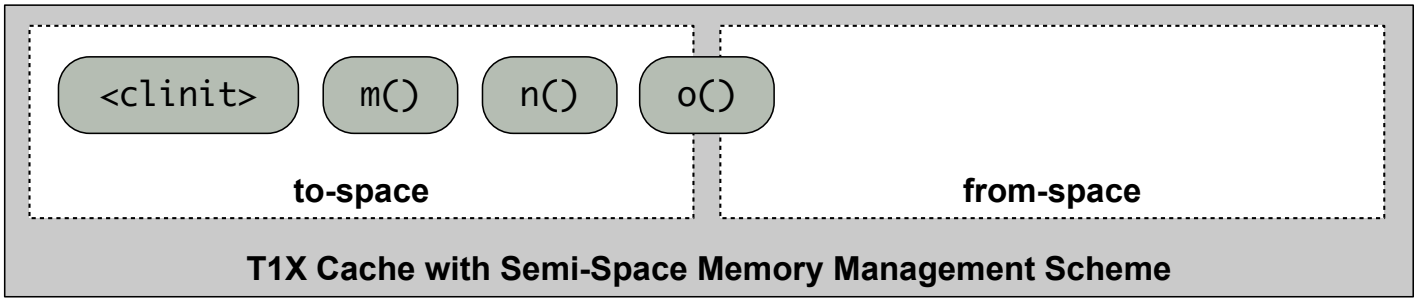
Maxine: Code Cache Management



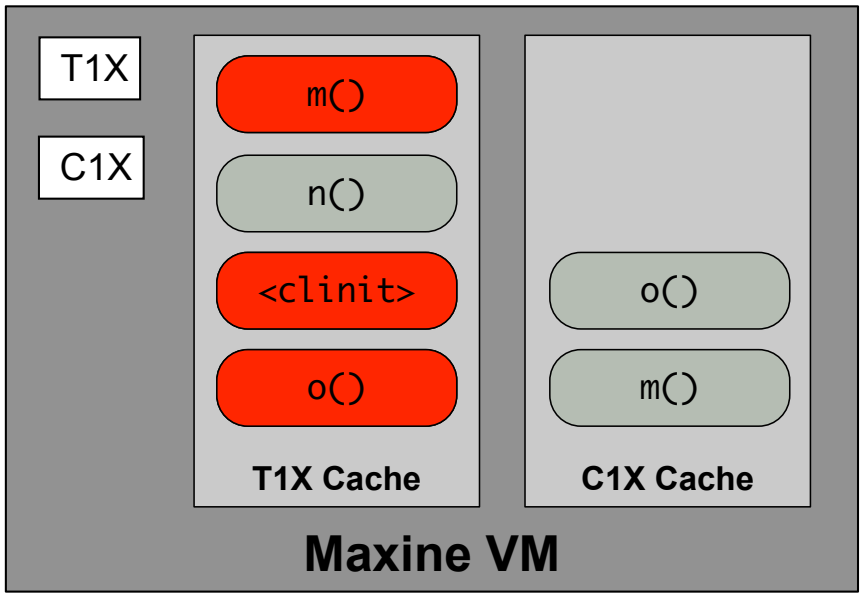


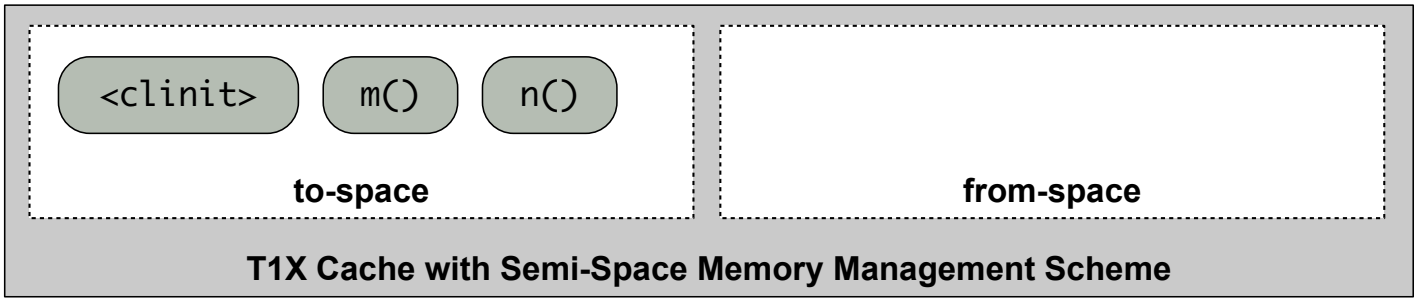
Maxine: Code Cache Management



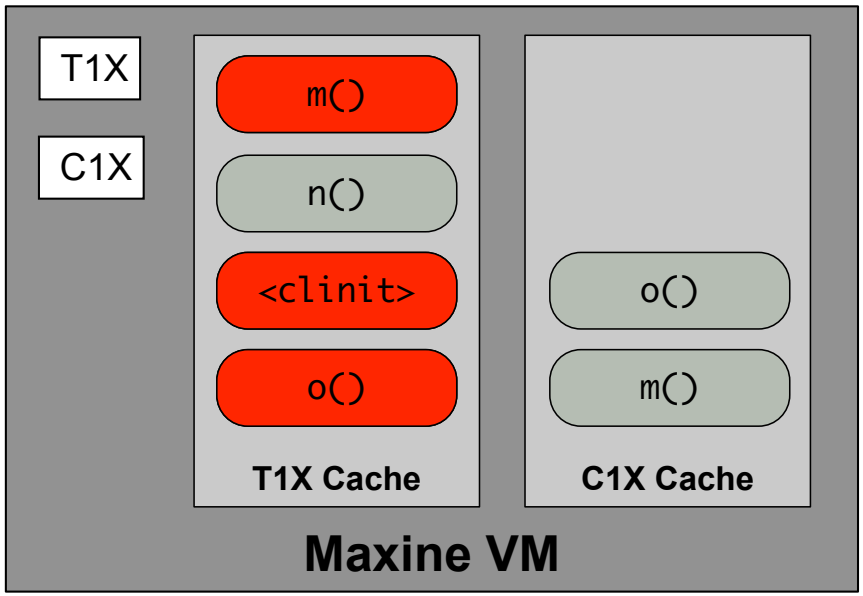


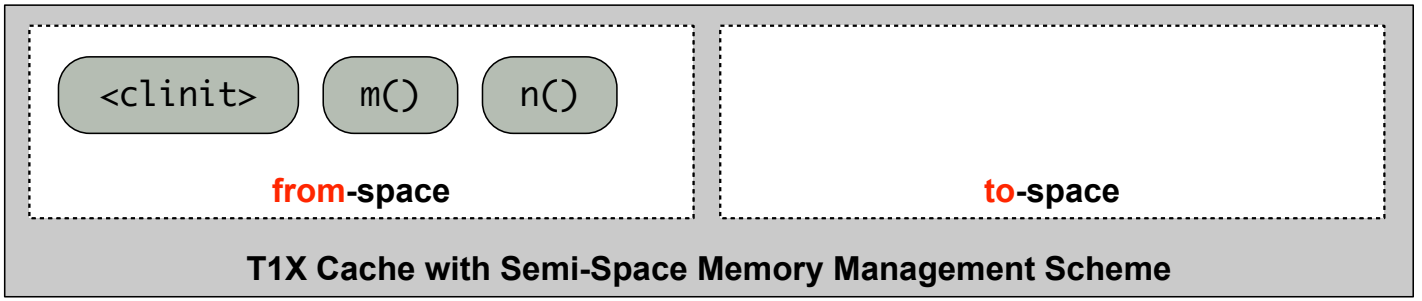
Maxine: Code Cache Management



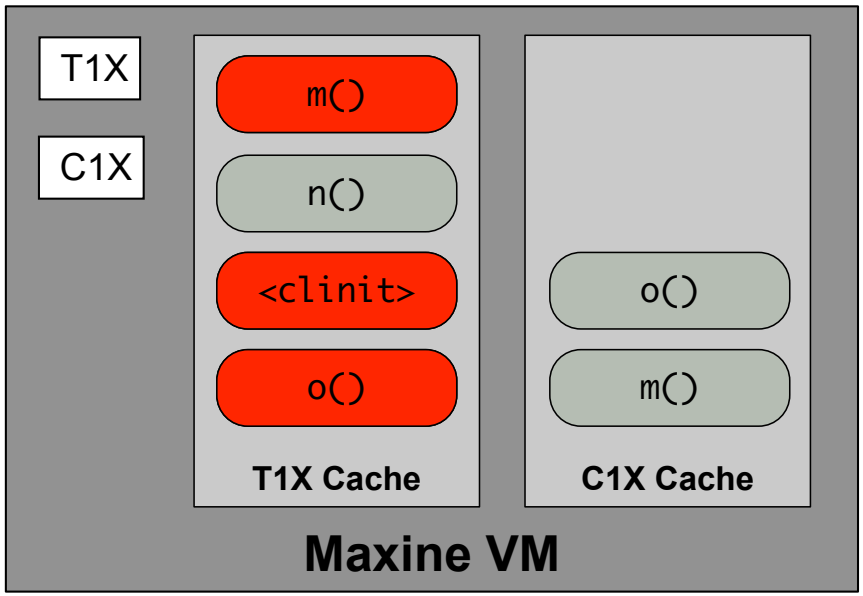


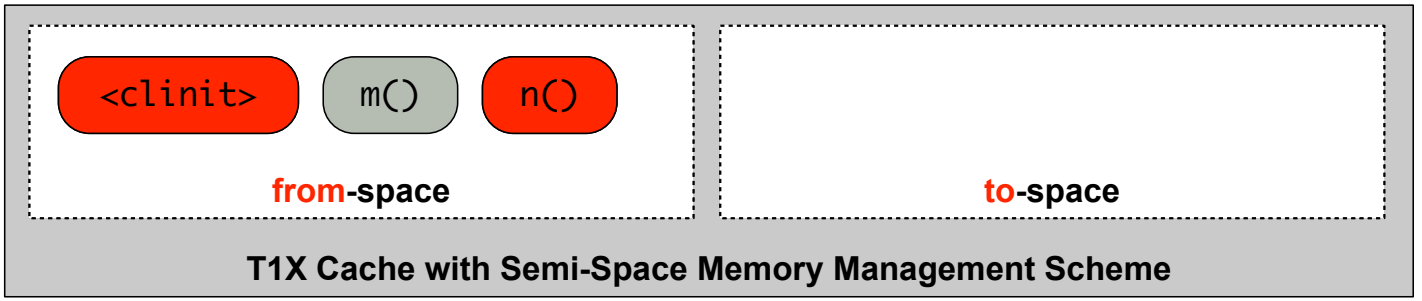
Maxine: Code Cache Management



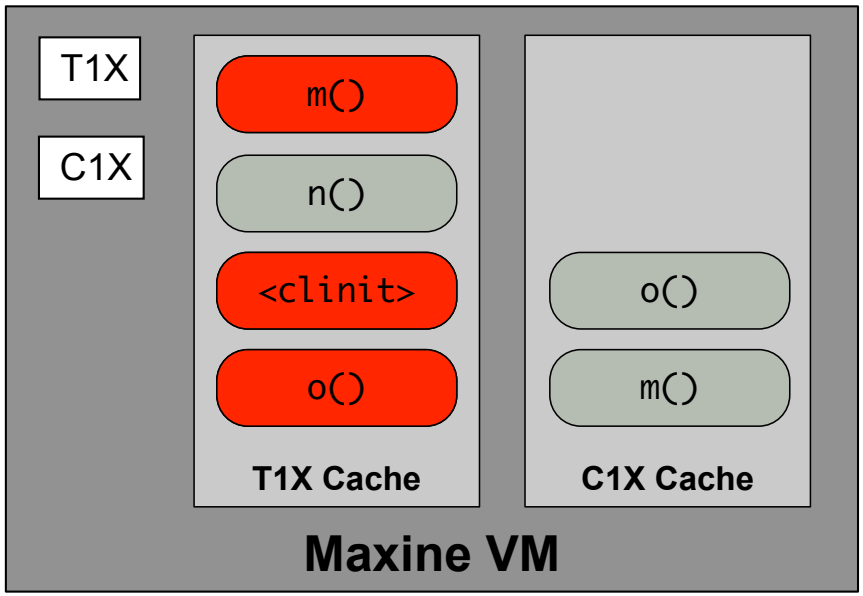


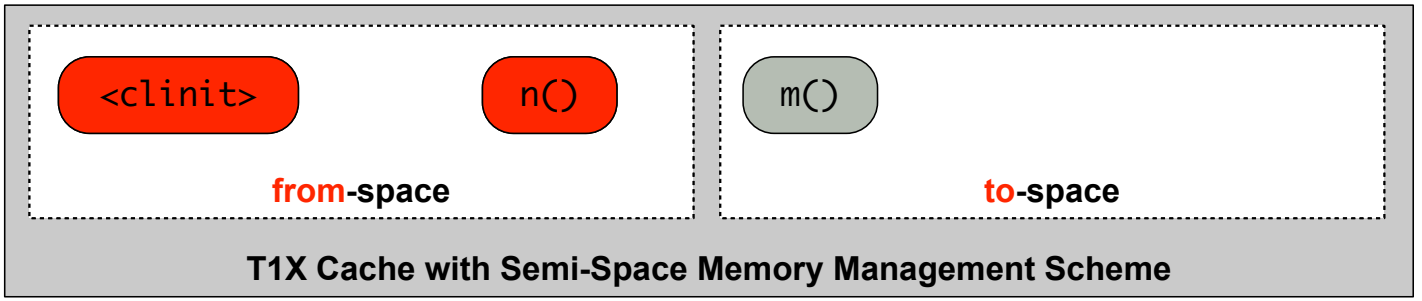
Maxine: Code Cache Management



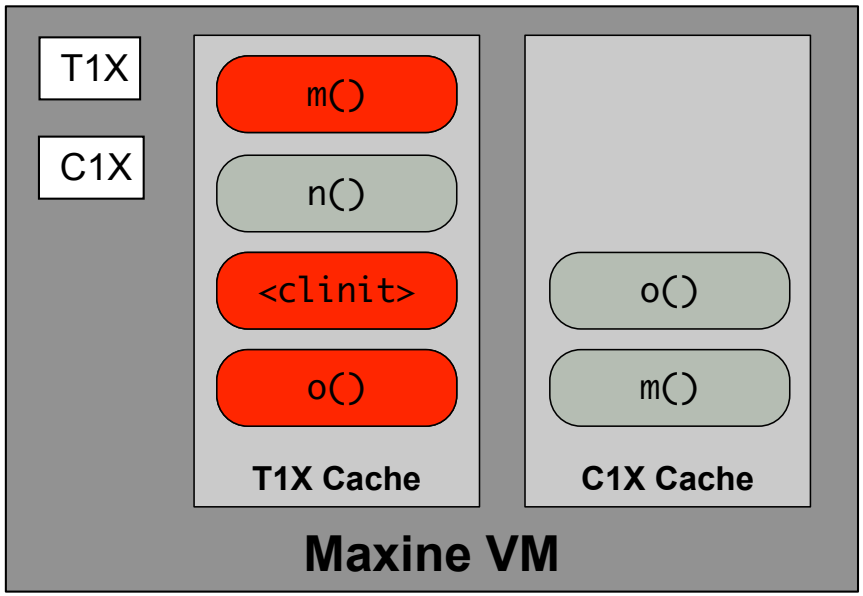


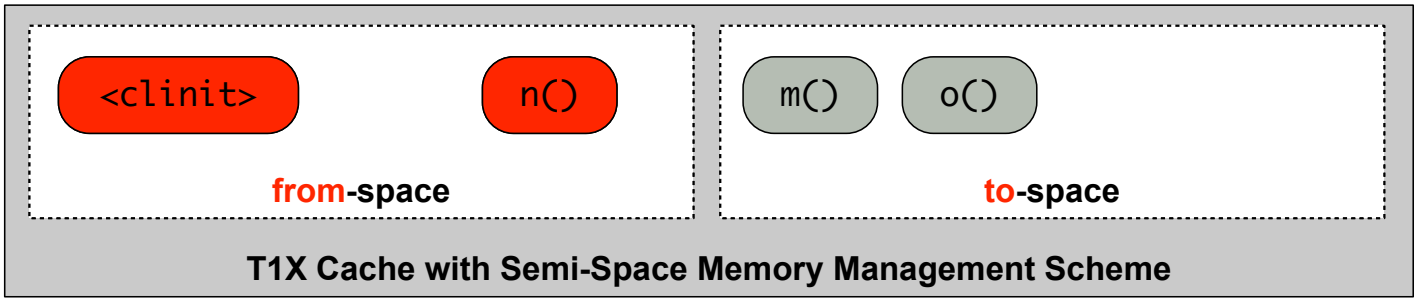
Maxine: Code Cache Management





Maxine: Code Cache Management



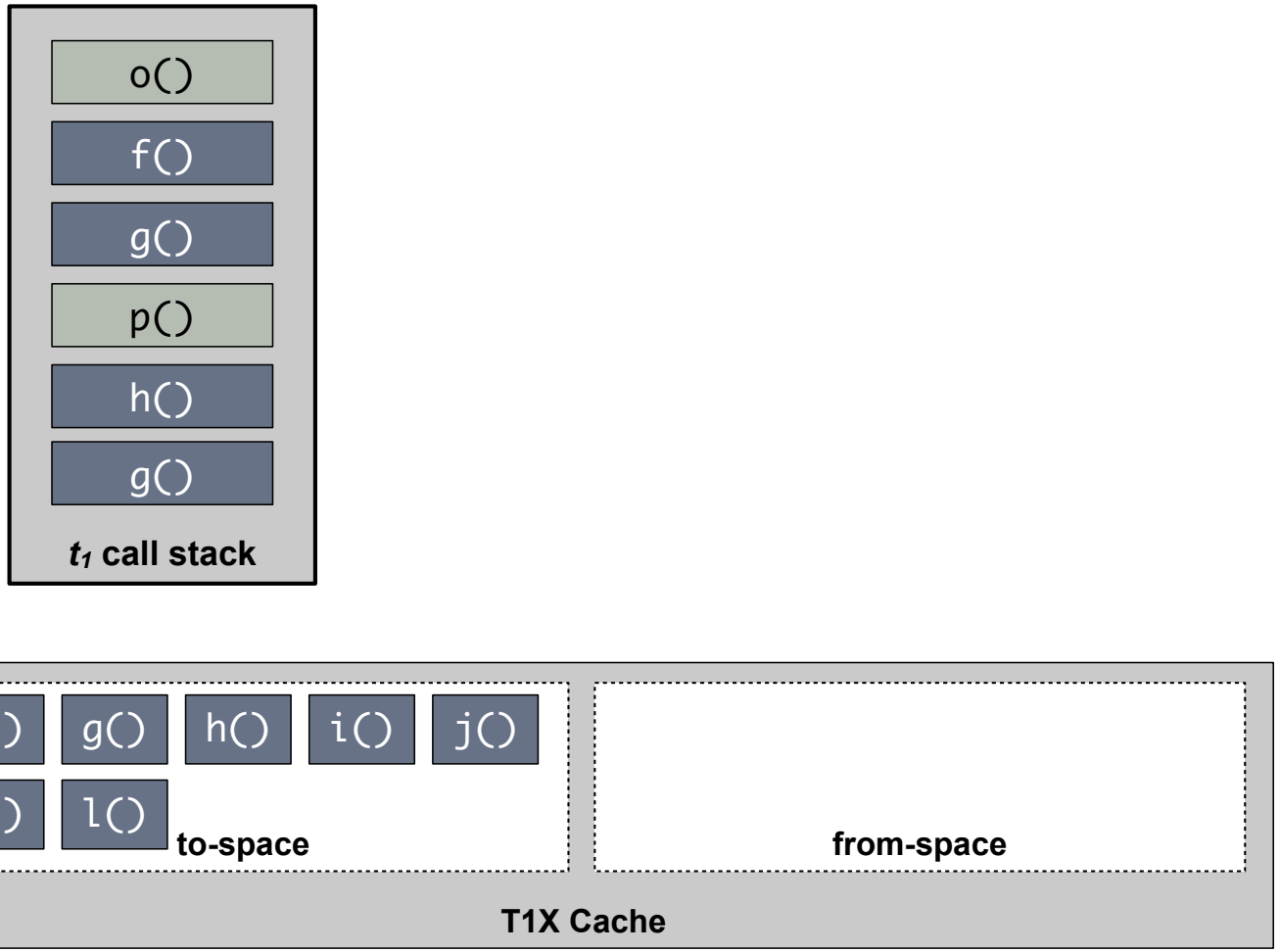


Code Cache Management: Which Methods are Live?

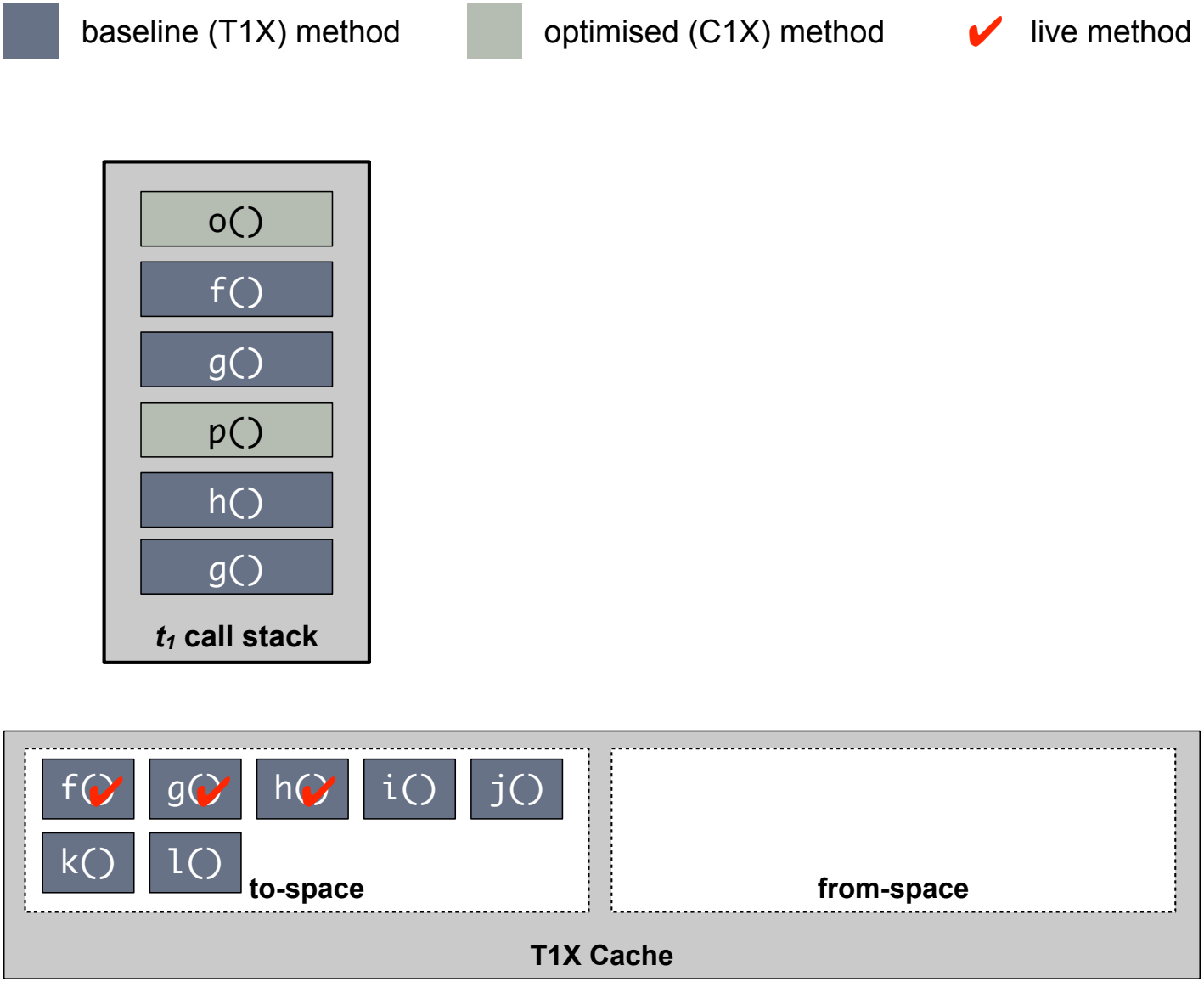
baseline (T1X) method

optimised (C1X) method

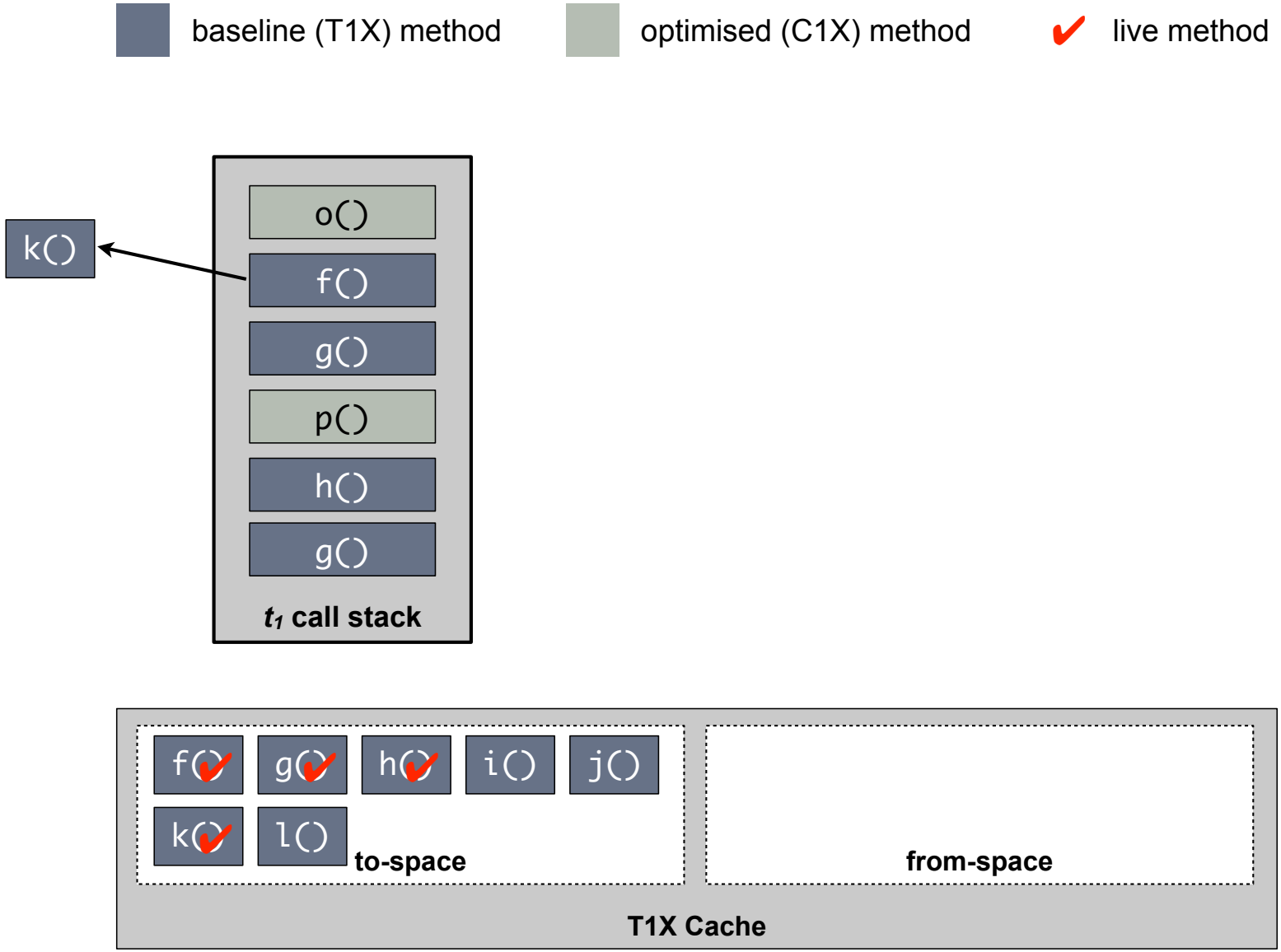
✔ live method



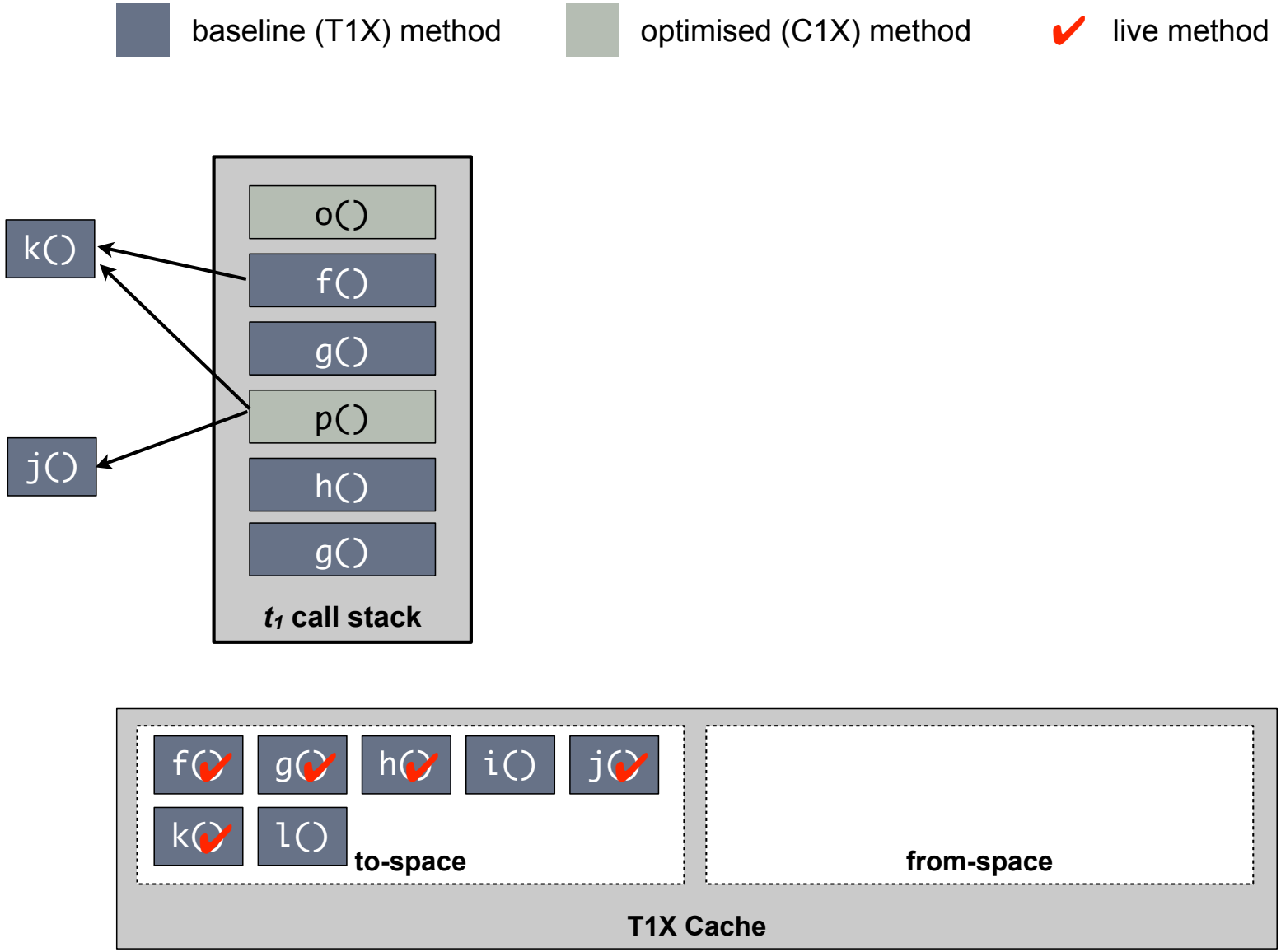
Code Cache Management: Which Methods are Live?



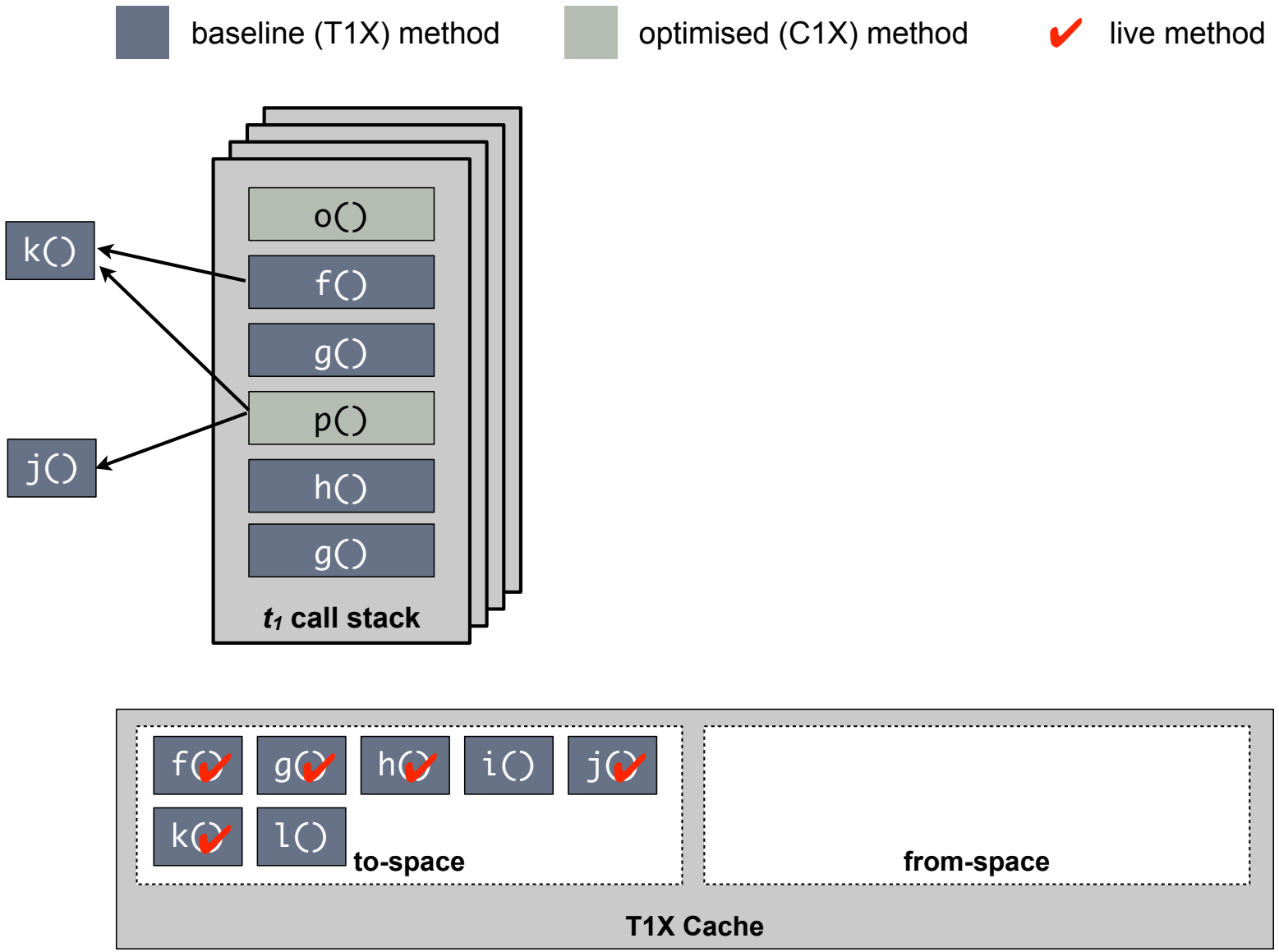
Code Cache Management: Which Methods are Live?



Code Cache Management: Which Methods are Live?

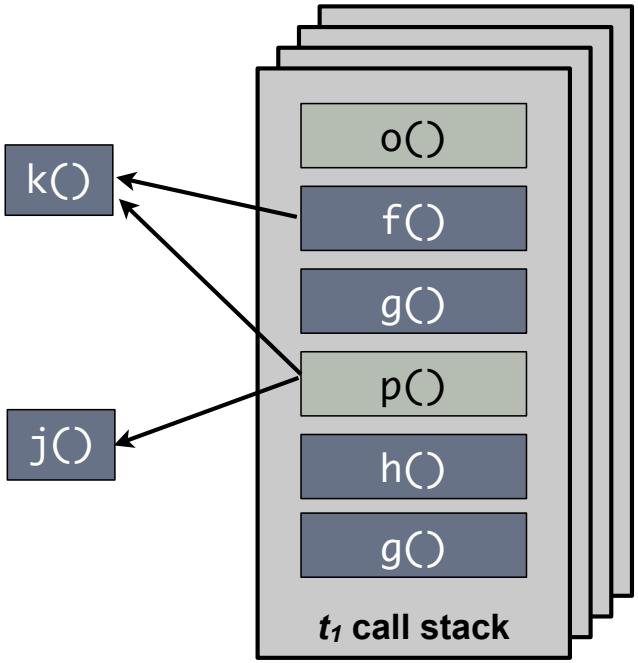


Code Cache Management: Which Methods are Live?



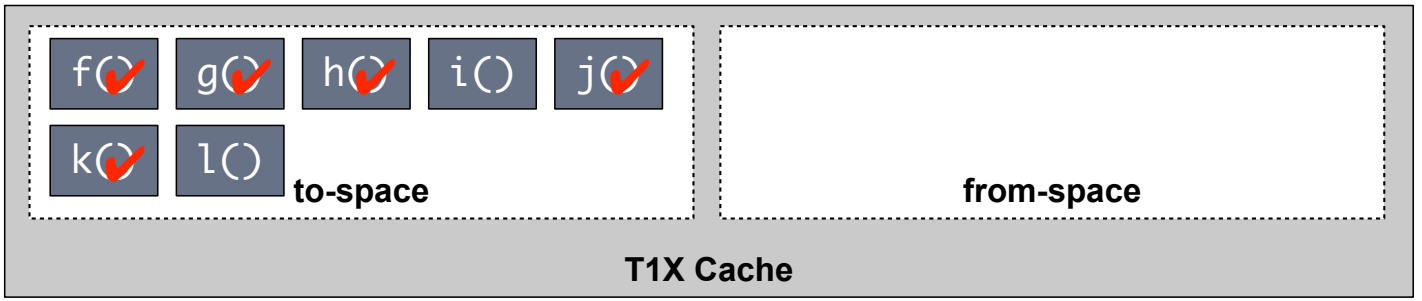
Code Cache Management: Which Methods are Live?

baseline (T1X) method optimised (C1X) method ✔ live method

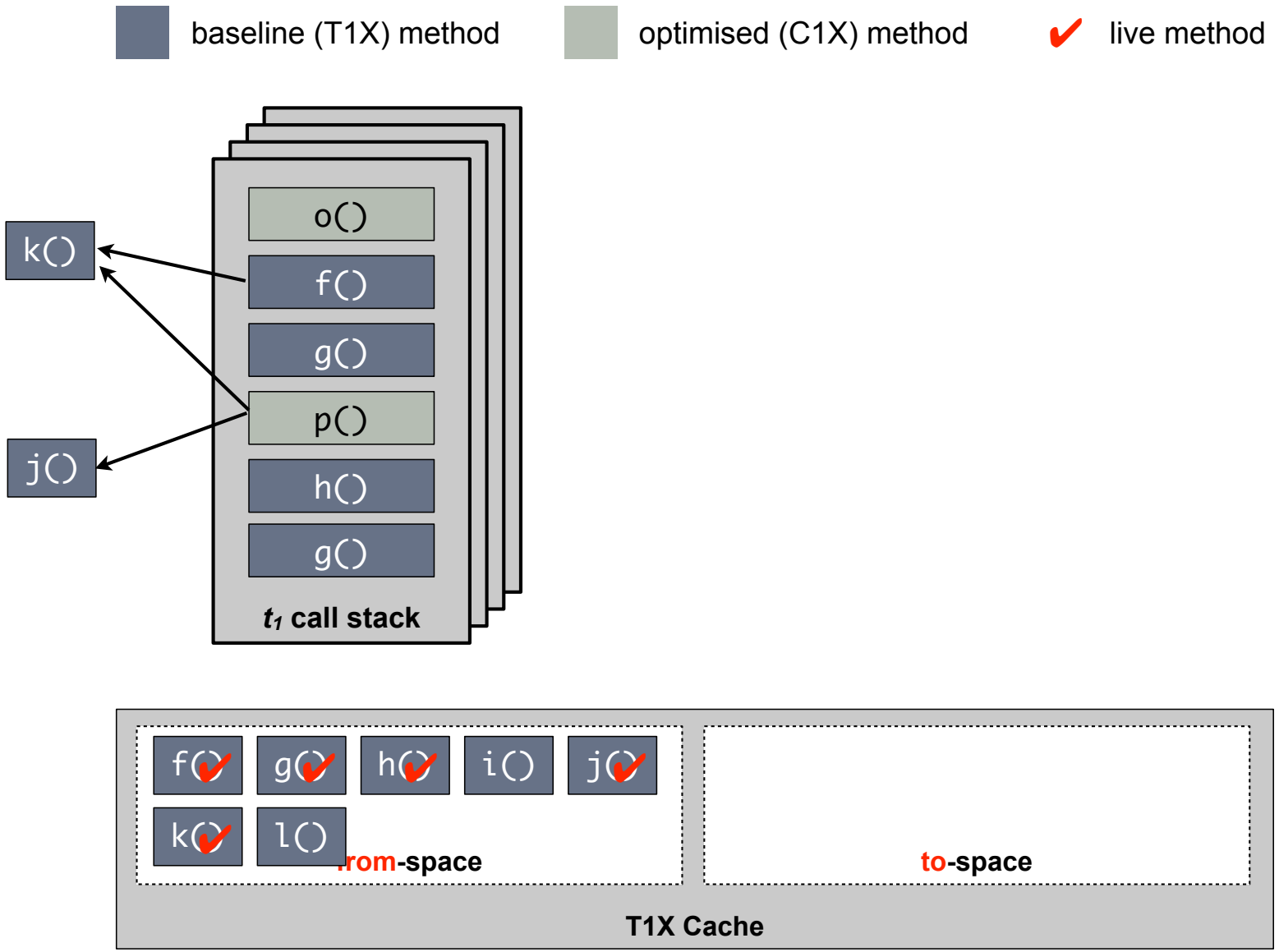


live methods
- methods on stack
- direct callees of on-stack methods

others to protect from eviction
- freshly compiled, not yet installed
- invocation counter within optimisation threshold
- existing type profile

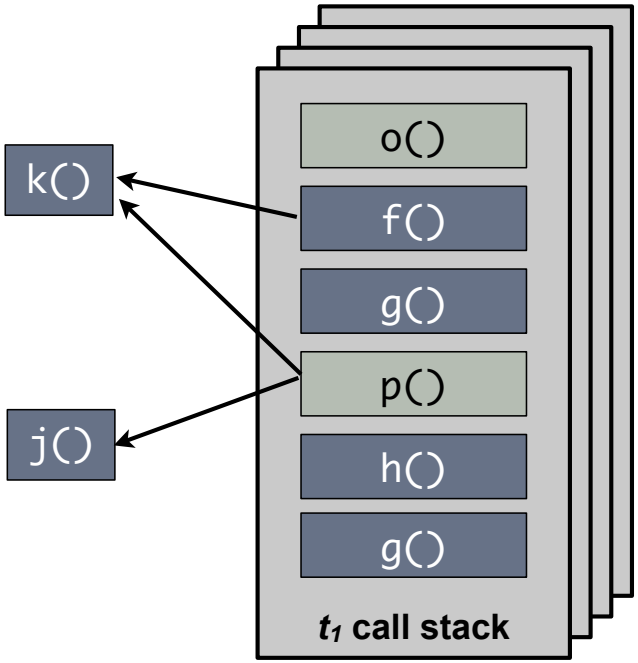


Code Cache Management: Eviction



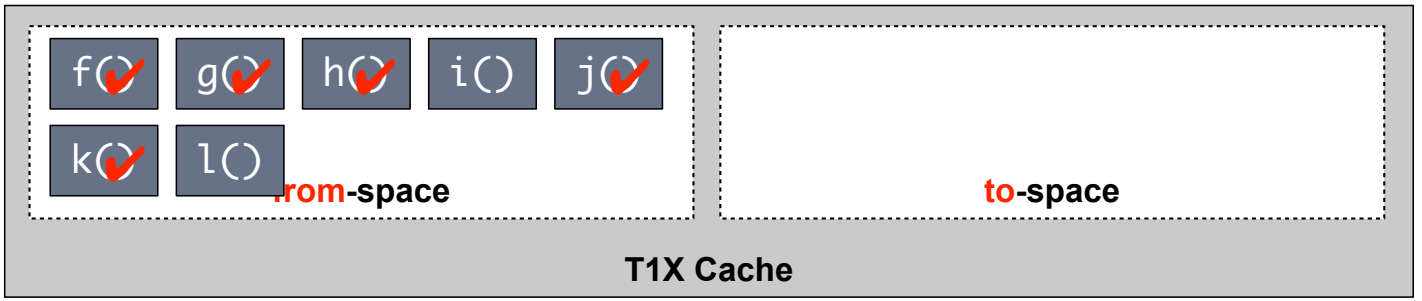
Code Cache Management: Eviction

baseline (T1X) method optimised (C1X) method ✔ live method



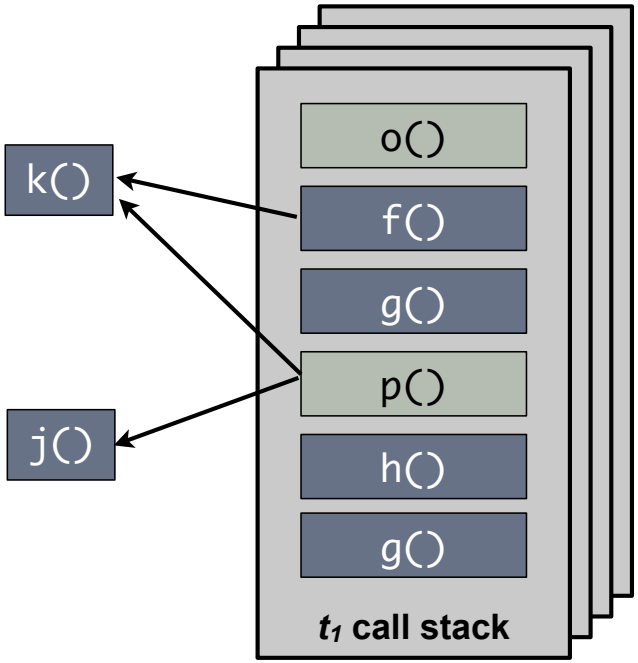
course of action

1. invalidate unmarked methods
   (and vtable entries, direct calls)



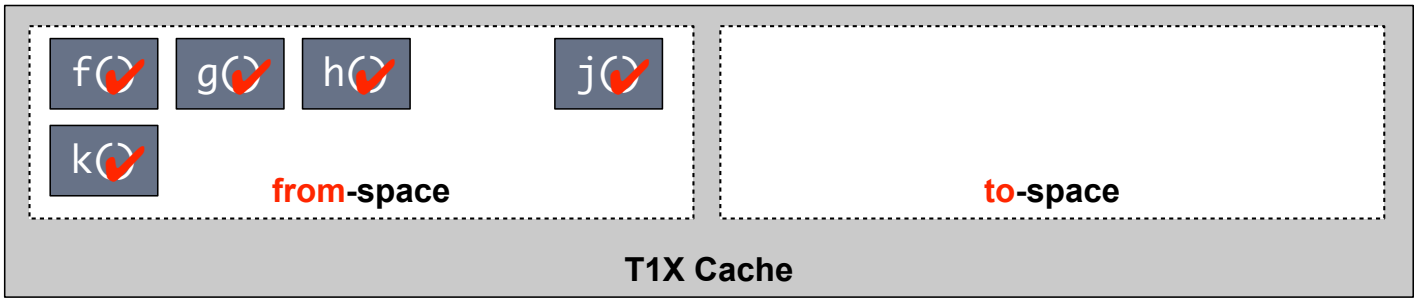
Code Cache Management: Eviction

baseline (T1X) method optimised (C1X) method ✔ live method



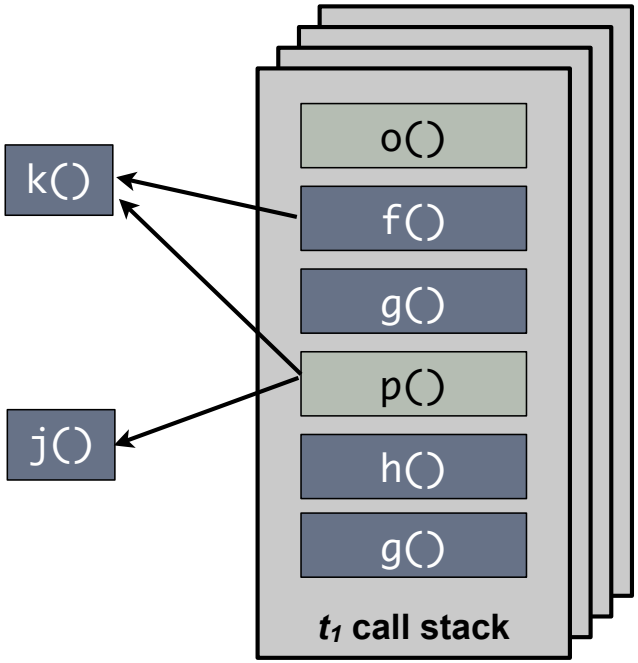
course of action

1. invalidate unmarked methods
   (and vtable entries, direct calls)



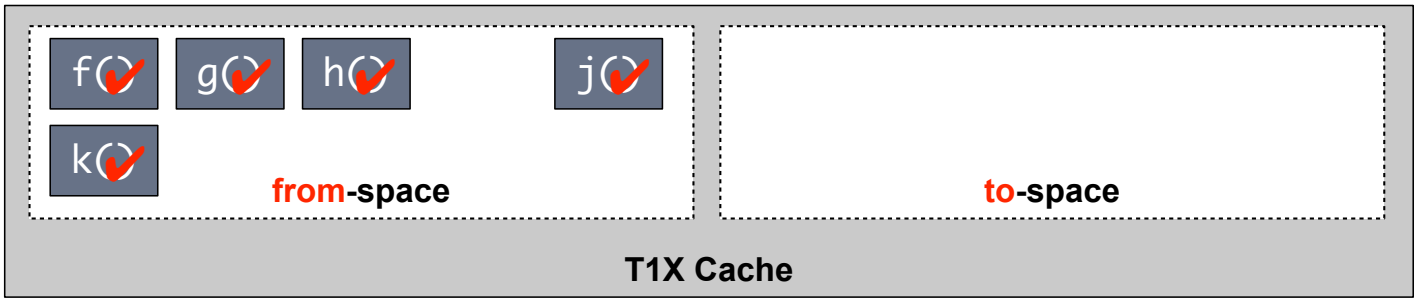
Code Cache Management: Eviction

baseline (T1X) method optimised (C1X) method ✔ live method



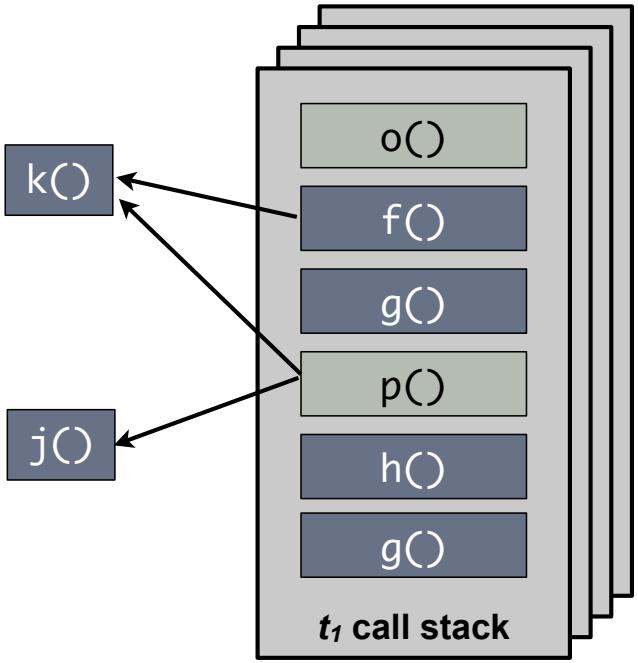
course of action

1. invalidate unmarked methods
   (and vtable entries, direct calls)
2. move marked methods
   (update vtable entries, direct calls)



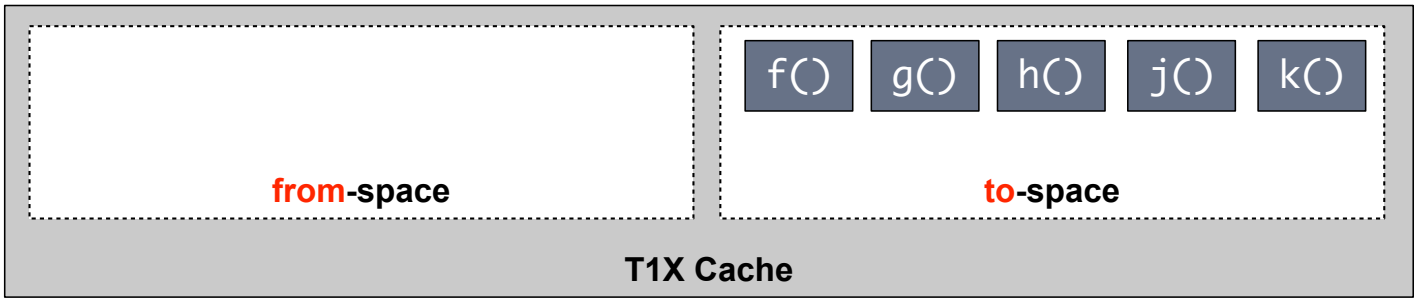
Code Cache Management: Eviction

baseline (T1X) method optimised (C1X) method ✔ live method



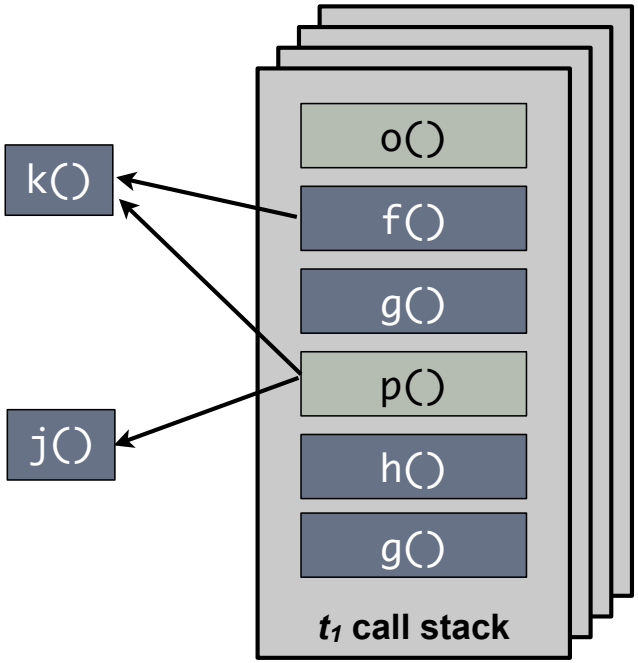
course of action

1. invalidate unmarked methods
   (and vtable entries, direct calls)
2. move marked methods
   (update vtable entries, direct calls)



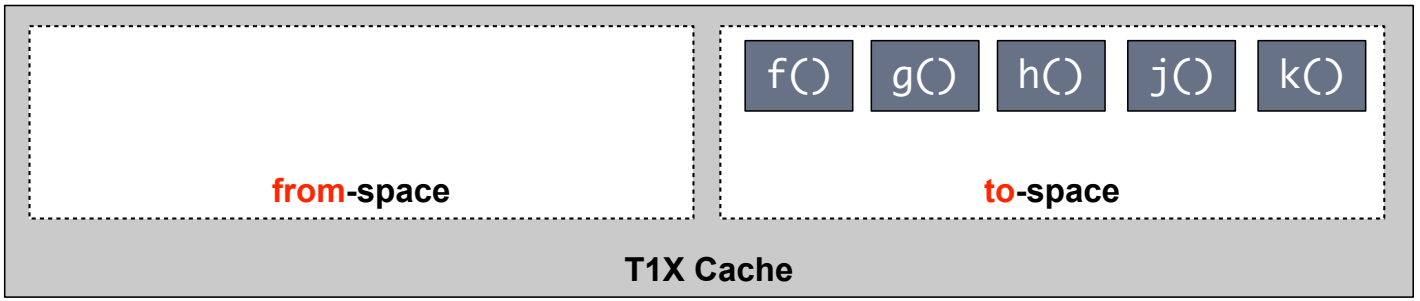
Code Cache Management: Eviction

baseline (T1X) method optimised (C1X) method ✔ live method



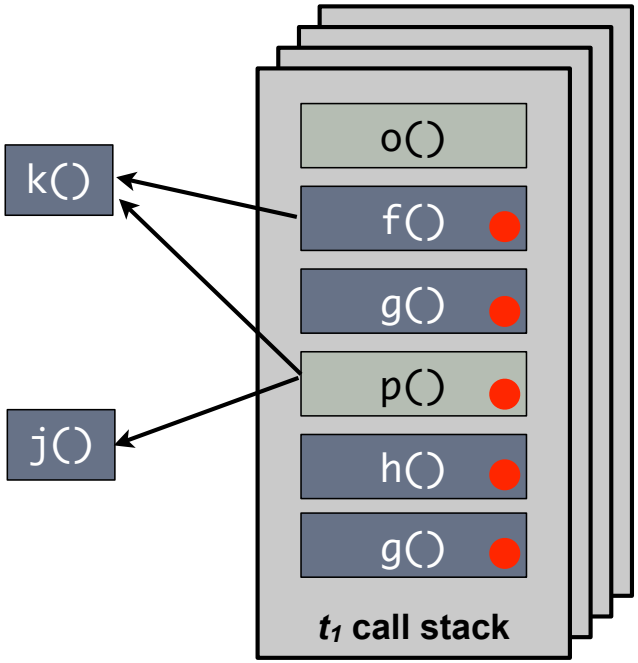
course of action

1. invalidate unmarked methods
   (and vtable entries, direct calls)
2. move marked methods
   (update vtable entries, direct calls)
3. patch return addresses



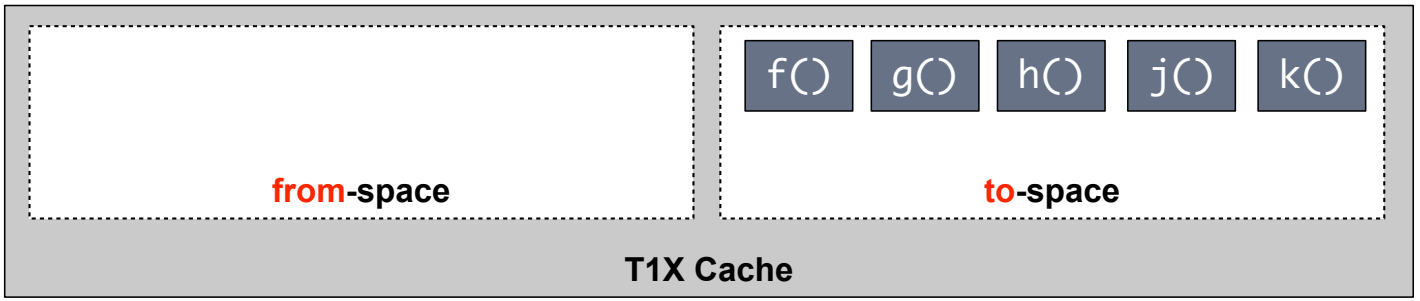
Code Cache Management: Eviction

baseline (T1X) method optimised (C1X) method ✔ live method



course of action

1. invalidate unmarked methods
   (and vtable entries, direct calls)
2. move marked methods
   (update vtable entries, direct calls)
3. patch return addresses



Where Do Pointers to Machine Code Occur, and How Are They Handled?

vtables/itables	✓	directly accessible via meta-level API (actors, hubs); invalidate (replace with trampoline calls) or relocate

Where Do Pointers to Machine Code Occur, and How Are They Handled?

vtables/itables	✓	directly accessible via meta-level API (actors, hubs); invalidate (replace with trampoline calls) or relocate
direct calls	✓	directly accessible via target method API (safepoints); invalidate (replace with trampoline calls) or relocate

Where Do Pointers to Machine Code Occur, and How Are They Handled?

vtables/itables	✓	directly accessible via meta-level API (actors, hubs); invalidate (replace with trampoline calls) or relocate
direct calls	✓	directly accessible via target method API (safepoints); invalidate (replace with trampoline calls) or relocate
return addresses	✓	accessible during stack walking; relocate accordingly

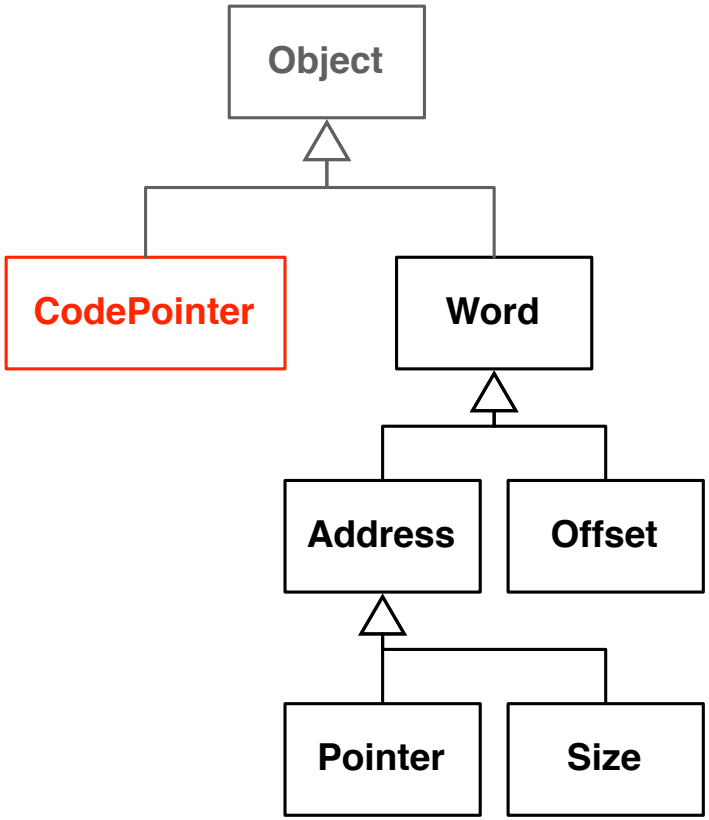
Where Do Pointers to Machine Code Occur, and How Are They Handled?

vtables/itables	✓	directly accessible via meta-level API (actors, hubs); invalidate (replace with trampoline calls) or relocate
direct calls	✓	directly accessible via target method API (safepoints); invalidate (replace with trampoline calls) or relocate
return addresses	✓	accessible during stack walking; relocate accordingly
local variables	✗	
member variables	✗	

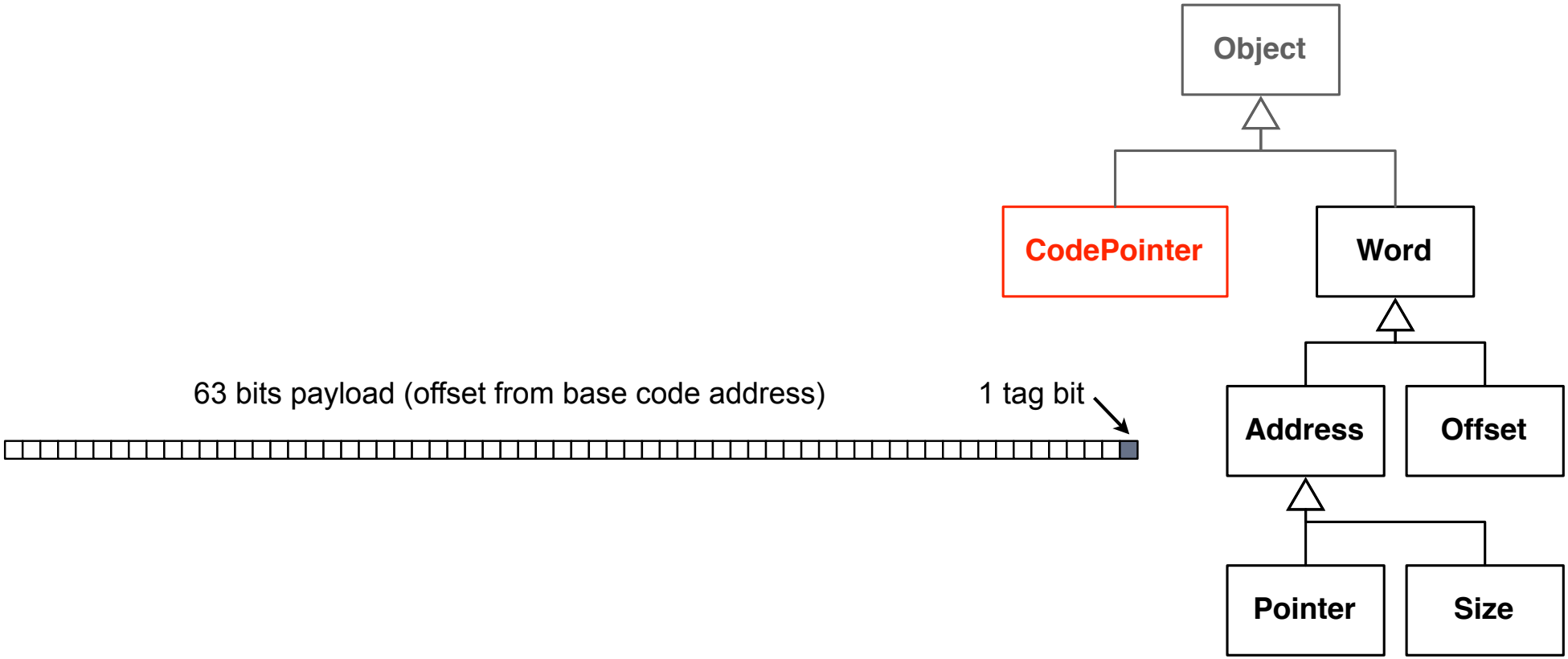
Where Do Pointers to Machine Code Occur, and How Are They Handled?

vtables/itables	✓	directly accessible via meta-level API (actors, hubs); invalidate (replace with trampoline calls) or relocate
direct calls	✓	directly accessible via target method API (safepoints); invalidate (replace with trampoline calls) or relocate
return addresses	✓	accessible during stack walking; relocate accordingly
local variables	✗	These might have to be relocated—how to determine which `Address` is a code pointer?
member variables	✗	

Tagged Code Pointers

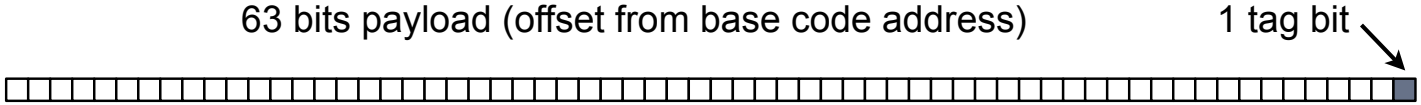


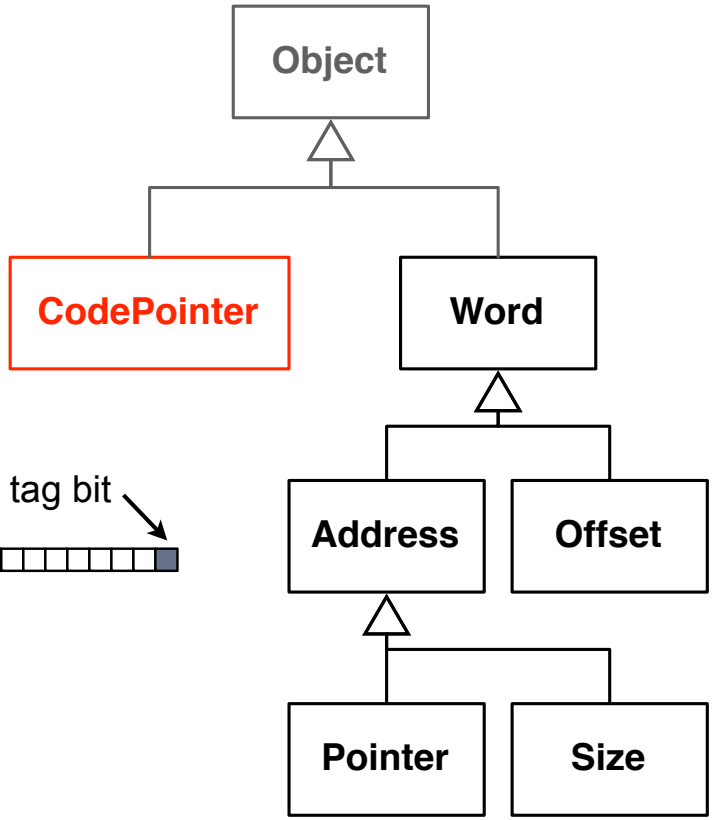
Tagged Code Pointers



Tagged Code Pointers

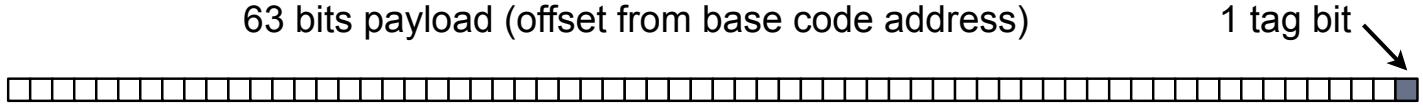
```
class CodePointer {
  ...
  @INLINE
  public static CodePointer from(long value) {
    if (isHosted()) {
      return new CodePointer(tag(value));
    }
    return UnsafeCast.asCodePointer(tag(value));
  }
  ...
}
```

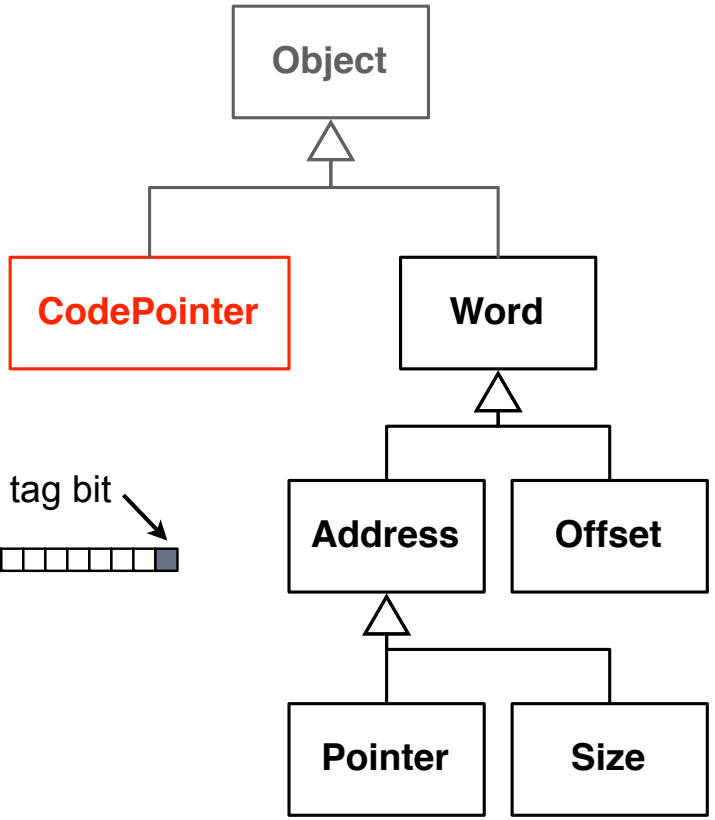




Tagged Code Pointers

```
class CodePointer {
  ...
  @INLINE
  public static CodePointer from(long value) {
    if (isHosted()) {
      return new CodePointer(tag(value));
    }
    return UnsafeCast.asCodePointer(tag(value));
  }
  ...
}
```






```
@INTRINSIC(UNSAFE_CAST)
public static CodePointer asCodePointer(long value) {
  return CodePointer.from(value);
}
```

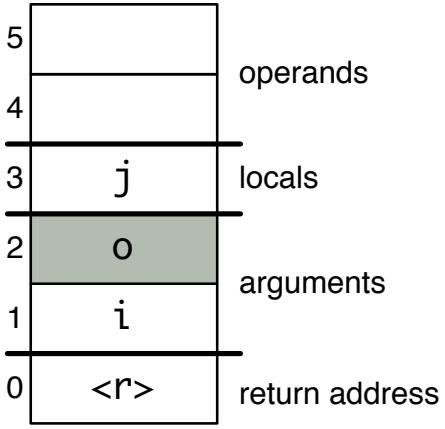
Stack Reference Maps and Tagged Code Pointers

```
static void m(int i, MyClass o) {
  int j = 2 * i;
  o.f(i);
}

0: iconst_2
1: iload_0
2: imul
3: istore_2
4: aload_1
5: iload_2
6: invokevirtual <f:(I)V>
9: return
```

Stack Reference Maps and Tagged Code Pointers

```
static void m(int i, MyClass o) {
  int j = 2 * i;
  o.f(i);
}

0: iconst_2
1: iload_0
2: imul
3: istore_2
4: aload_1
5: iload_2
6: invokevirtual <f:(I)V>
9: return
```

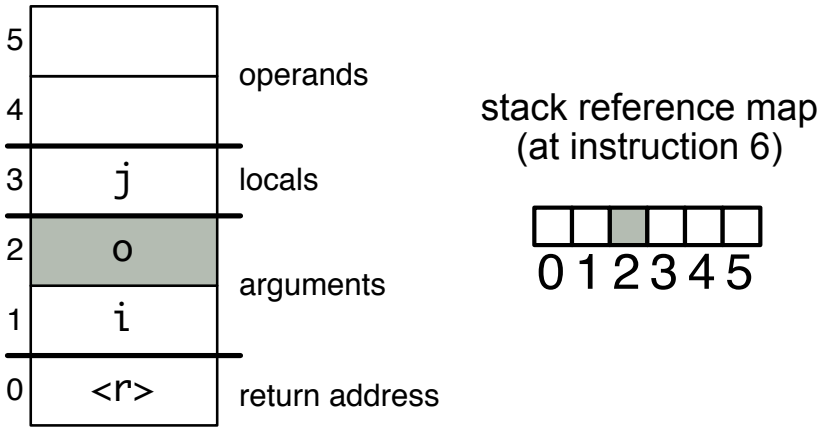
idealised stack frame layout



Stack Reference Maps and Tagged Code Pointers

```
static void m(int i, MyClass o) {
  int j = 2 * i;
  o.f(i);
}

0: iconst_2
1: iload_0
2: imul
3: istore_2
4: aload_1
5: iload_2
6: invokevirtual <f:(I)V>
9: return
```

idealised stack frame layout



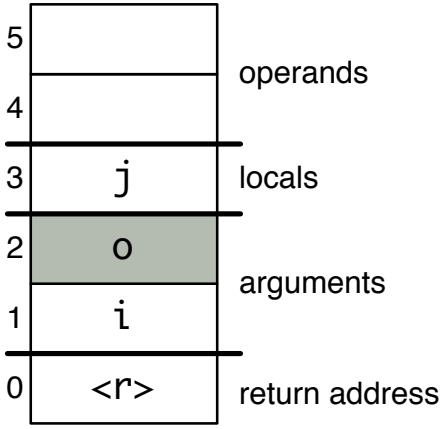
stack reference map
(at instruction 6)

Stack Reference Maps and Tagged Code Pointers

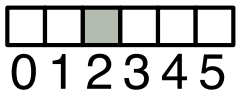
```
static void m(int i, MyClass o) {
  int j = 2 * i;
  o.f(i);
}
```

```
0: iconst_2
1: iload_0
2: imul
3: istore_2
4: aload_1
5: iload_2
6: invokevirtual <f:(I)V>
9: return
```

idealised stack frame layout



stack reference map
(at instruction 6)



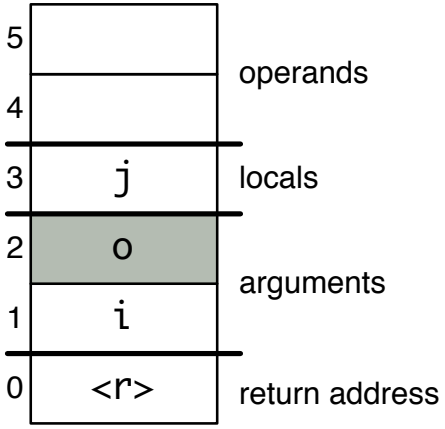

```
void x(...) {
  ...
  MyClass o = ...;
  CodePointer cp = ...;
  ...
}
```

Stack Reference Maps and Tagged Code Pointers

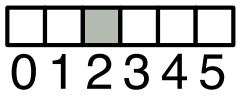
```
static void m(int i, MyClass o) {
  int j = 2 * i;
  o.f(i);
}
```

```
0: iconst_2
1: iload_0
2: imul
3: istore_2
4: aload_1
5: iload_2
6: invokevirtual <f:(I)V>
9: return
```

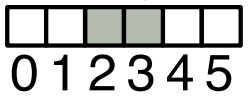
idealised stack frame layout



stack reference map
(at instruction 6)




```
void x(...) {
  ...
  MyClass o = ...;
  CodePointer cp = ...;
  ...
}
```

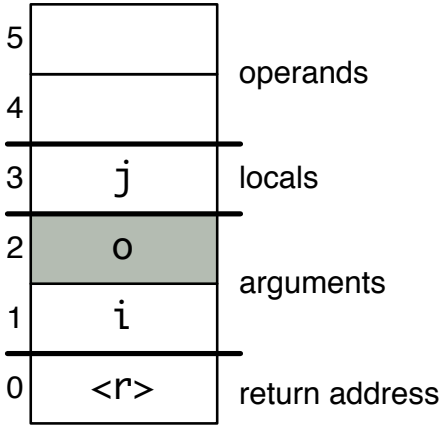


Stack Reference Maps and Tagged Code Pointers

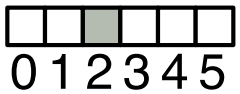
```
static void m(int i, MyClass o) {
  int j = 2 * i;
  o.f(i);
}
```

```
0: iconst_2
1: iload_0
2: imul
3: istore_2
4: aload_1
5: iload_2
6: invokevirtual <f:(I)V>
9: return
```

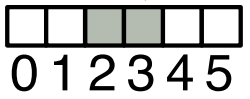
idealised stack frame layout

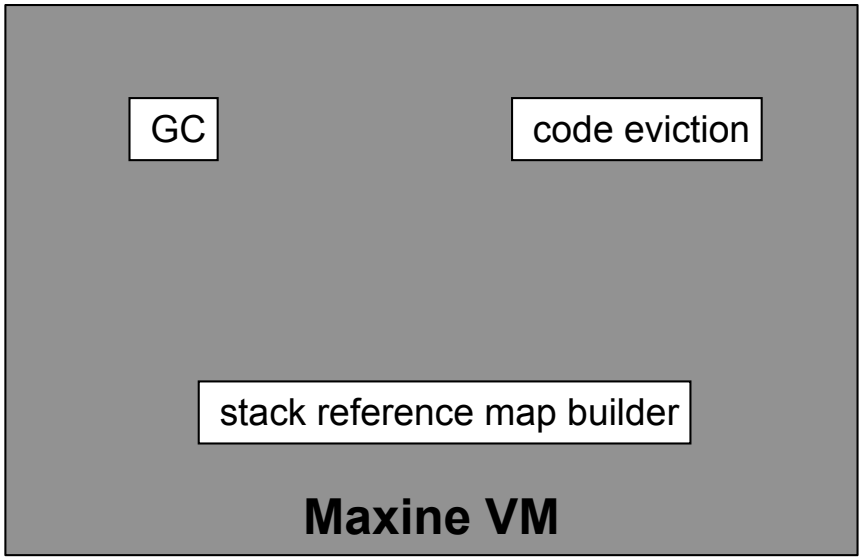


stack reference map
(at instruction 6)




```
void x(...) {
  ...
  MyClass o = ...;
  CodePointer cp = ...;
  ...
}
```



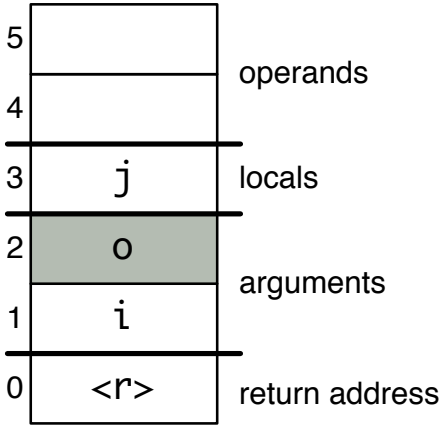


Stack Reference Maps and Tagged Code Pointers

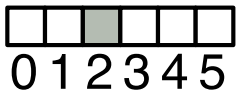
```
static void m(int i, MyClass o) {
  int j = 2 * i;
  o.f(i);
}
```

```
0: iconst_2
1: iload_0
2: imul
3: istore_2
4: aload_1
5: iload_2
6: invokevirtual <f:(I)V>
9: return
```

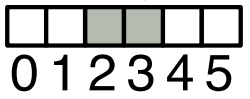
idealised stack frame layout

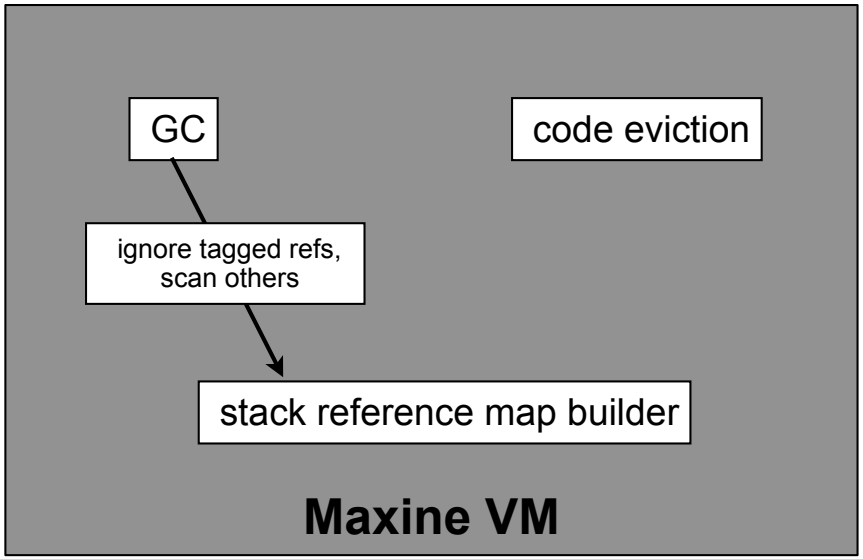


stack reference map
(at instruction 6)




```
void x(...) {
  ...
  MyClass o = ...;
  CodePointer cp = ...;
  ...
}
```



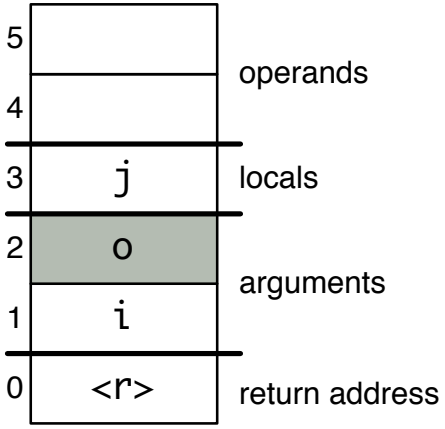


Stack Reference Maps and Tagged Code Pointers

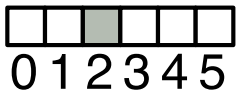
```
static void m(int i, MyClass o) {
  int j = 2 * i;
  o.f(i);
}
```

```
0: iconst_2
1: iload_0
2: imul
3: istore_2
4: aload_1
5: iload_2
6: invokevirtual <f:(I)V>
9: return
```

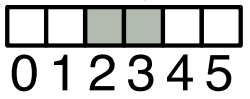
idealised stack frame layout

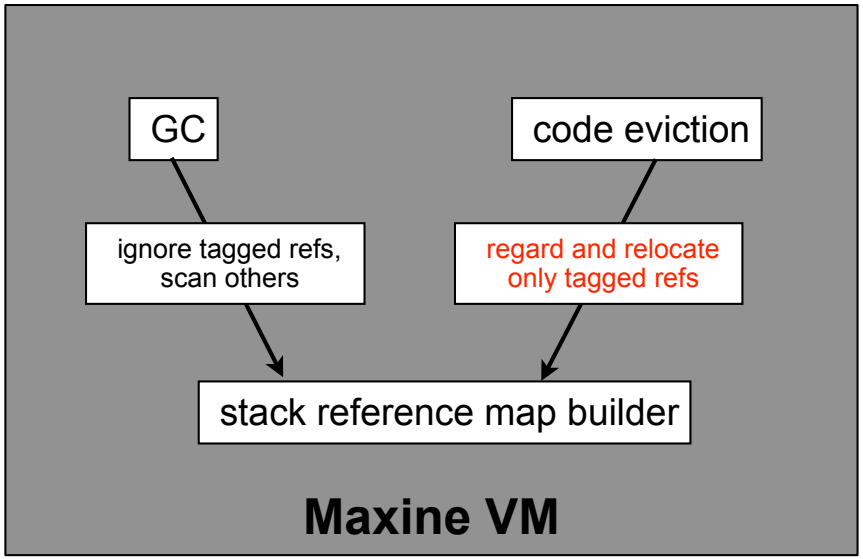


stack reference map
(at instruction 6)




```
void x(...) {
  ...
  MyClass o = ...;
  CodePointer cp = ...;
  ...
}
```





Thank you for your attention.

Acknowledgements: these slides are joint work of the VM research group.

Home page: http://wikis.oracle.com/display/MaxineVM/Home

Source (GPLv2): https://hg.kenai.com/hg/maxine~maxine

Mailing list via: http://kenai.com/projects/maxine


ORACLE®