18-600 Foundations of Computer Systems

Lecture 19: "Virtual Machine Design & Implementation"

John P. Shen

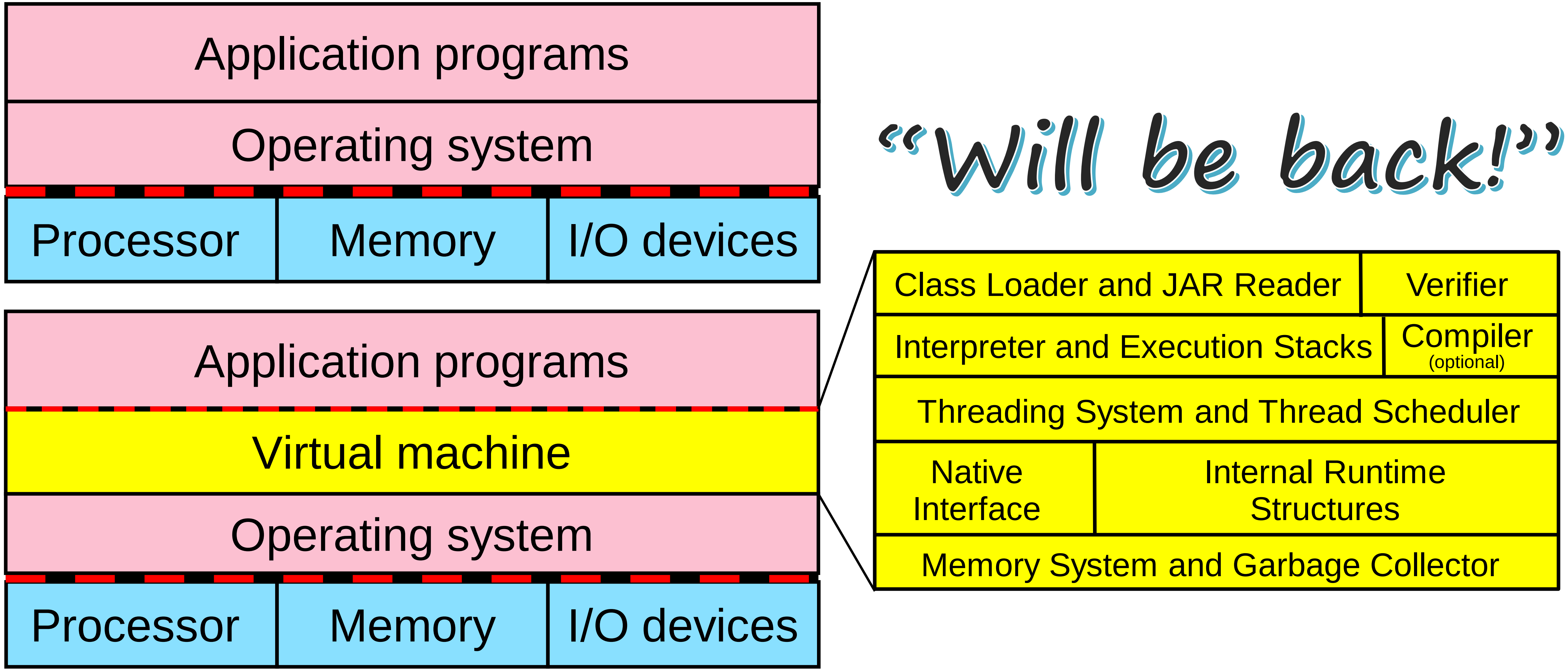
November 6, 2017

(Based on an 18-640 guest lecture given by Antero Taivalsaari, Nokia Fellow)

- Recommended References:
  - Jim Smith, Ravi Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*, Morgan Kaufmann, June 2005.
  - Matthew Portnoy, *Virtualization Essentials*, Sybex Press, May 2012

Electrical & Computer ENGINEERING

Java Programming & Java Virtual Machine



Goals of this Lecture

- Introduce you to the world of virtual machine (VM) design.
- Provide an overview of key technologies that are needed for constructing virtual machines, such as automatic memory management, interpretation techniques, multithreading, and instruction set.
- *Caveat: This is a very broad area – we will only scratch the surface in this lecture.*

Introduction

What is a Virtual Machine?

- A *virtual machine* (VM) is an “abstract” computing architecture or computational engine that is independent of any particular hardware or operating system.
- Software machine that runs on top of a physical hardware platform and operating system.
- Allows the same applications to run “virtually” on any hardware for which a VM is available.

Two Broad Classes of Virtual Machines

There are two broad classes of virtual machines:

1) ***System virtual machines*** – typically aimed at virtualizing the execution of an entire operating system.
    - Examples: VMware Workstation, VirtualBox, Virtual PC

2) ***Language virtual machines*** (process virtual machines) – typically aimed at providing a portable runtime environment for specific programming languages.
    - Examples: Java VM, Dalvik, Microsoft CLR, V8, LLVM, Squeak

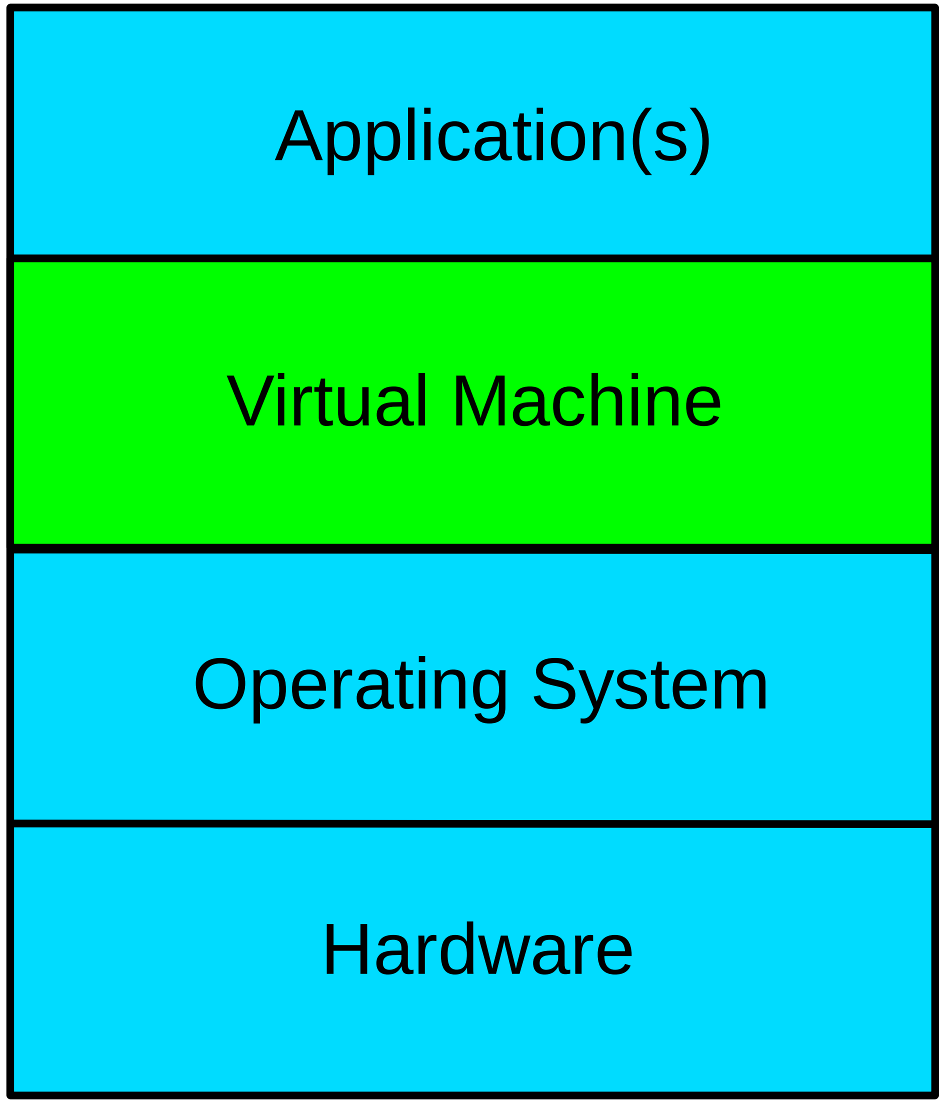
Focus in this lecture is on Language VMs

Why are Virtual Machines Interesting?

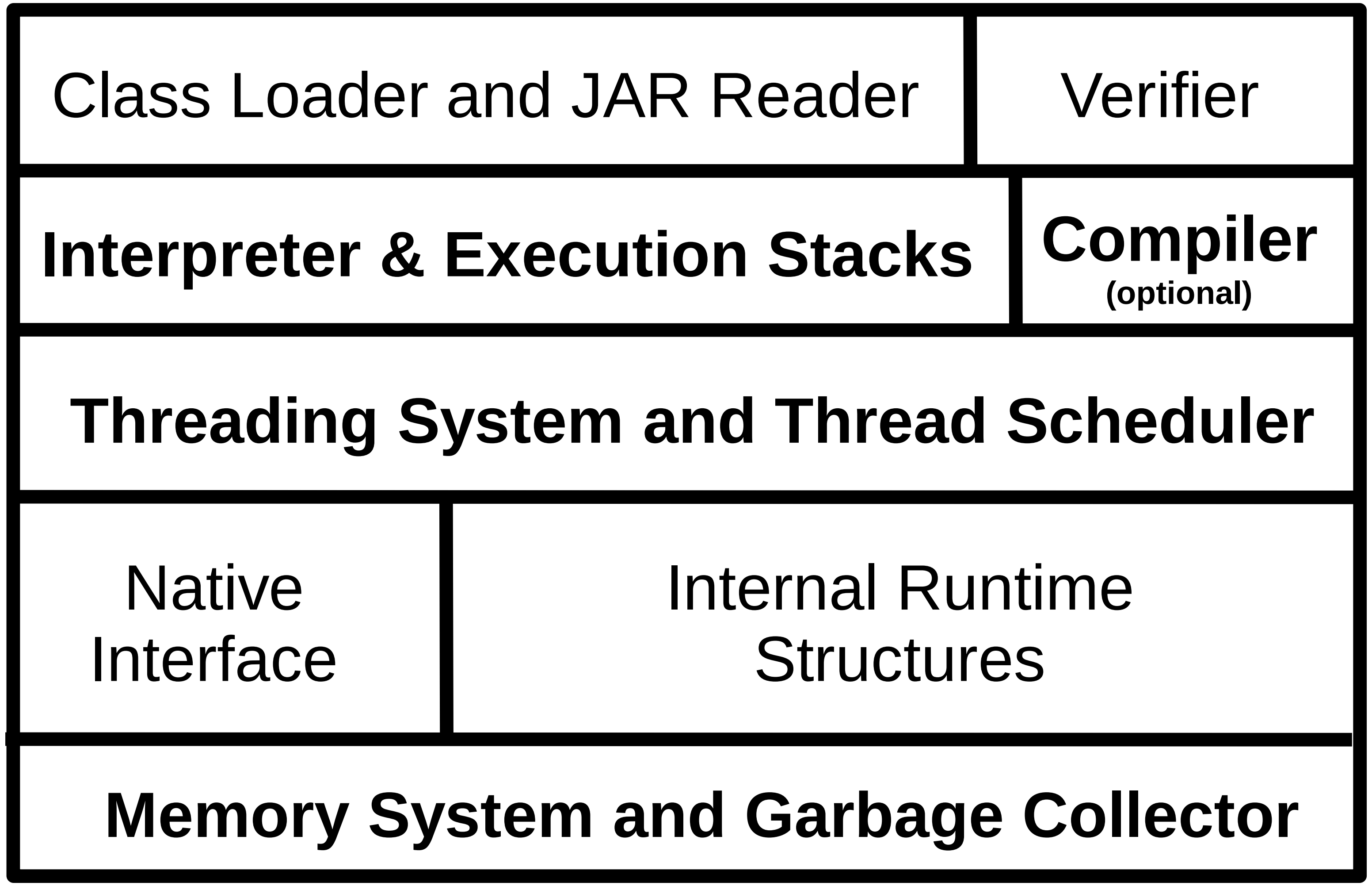
- Provide platform independence
- Isolate programs from hardware details
- Simplify application code migration across physical platforms
- Can support dynamic downloading of software
- Can provide additional security or scalability that hardware-specific implementations cannot provide
- Can hide the complexity of legacy systems

- Many interesting programming languages and systems are built around a virtual machine

Language VMs – Typical High-Level Architecture



Example: Components of a Java Virtual Machine (JVM)



VM vs. OS Design

- There is a lot of similarity between VM and operating system design.
  - The key component areas are pretty much the same (memory management, threading system, I/O, ...)

- A few key differences:
  - Operating systems are language-independent extensions of the underlying hardware. They are built to facilitate access to the underlying computing architecture and maximize the utilization of the hardware resources.

  - In contrast, language VMs implement a machine-independent instruction set and abstract away the details of the underlying hardware and the host operating system pretty much completely.

Languages that Use Virtual Machines

- Well-known languages using a virtual machine:
  - *Lisp* systems, 1958/1960-1980s
  - *Basic, 1964-1980s*
  - *Forth*, early 1970s
  - *Pascal* (P-Code versions), late 1970s/early 1980s
  - *Smalltalk*, 1970s-1980s
  - *Self*, late 1980/early 1990s
  - *Java*, late 1990s (2000's for Android)

- Numerous other languages:
  - ... *PostScript, TCL/TK, Perl, Python, C#*, ...

Designing and Implementing Virtual Machines

How are Virtual Machines Implemented?

- Virtual machines are typically written in "portable" and "efficient" programming languages such as C or C++.
- For performance-critical components, assembly language is used.
  - The more machine code is used, the less portability
- Some virtual machines (Lisp, Forth, Smalltalk) are largely written in the language itself.
  - These systems have only a minimal core implemented in C or assembly language.
- Most Java VM implementations consist of a mixture of C/C++ and assembly code.

The Common Tradeoffs

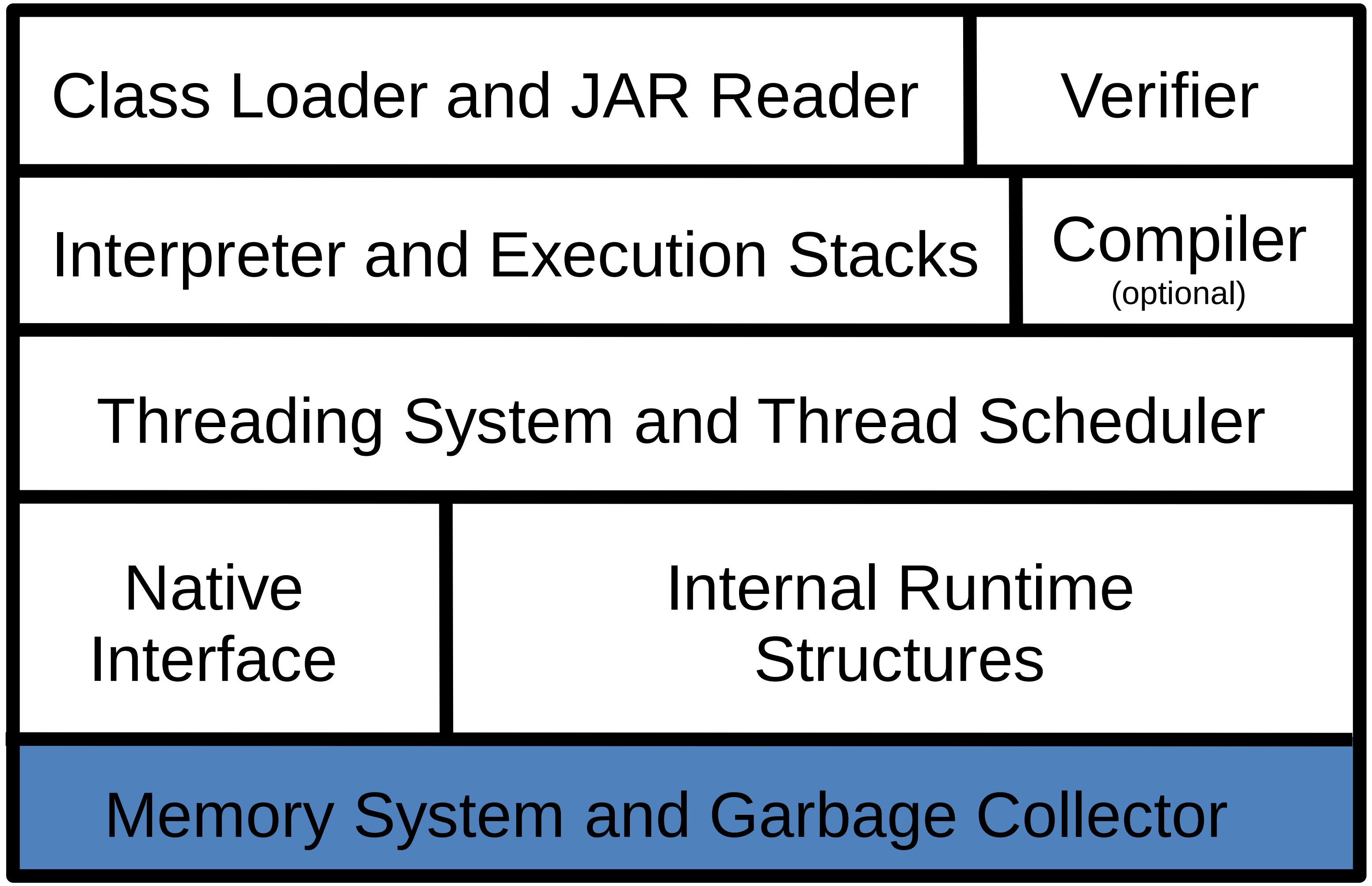
- Unfortunately, for nearly all aspects of the VM:
    - Simple implies slow
    - Fast implies more complicated
    - Fast implies less portable
    - Fast implies larger memory consumption

Examples of areas with significant tradeoffs:

- Interpretation
- Memory management
- Locking/Synchronization, exception handling
- Dynamic compilation, debugging

There are two “camps” of language VM designers:
(1) *speed enthusiasts* and (2) *portability enthusiasts*

Walkthrough of Essential Component Areas



Memory Management

Basic Memory Management Strategies

1) Static memory management
   – Everything allocated statically.

2) Linear memory management
   – Memory is allocated and freed in Last-In-First-Out (LIFO) order.

3) **Dynamic memory management**
   – Memory is allocated dynamically from a large pre-allocated “heap” of memory.

- Dynamic memory management is a prerequisite for most modern programming languages

Dynamic Memory Management

- In dynamic memory management, objects can be allocated and deallocated freely.
  - Allows the creation and deletion of objects in an arbitrary order.
  - Objects can be resized on the fly.
- Most modern virtual machines use some form of dynamic memory management.
- Depending on the implementation, dynamic memory management can be:
  - *Manual*: the programmer is responsible for freeing the unused areas explicitly (e.g., malloc/free/realloc in C)
  - *Automatic*: the virtual machine frees the unused areas implicitly without any programmer intervention.

Automatic Memory Management: Garbage Collection

- Most modern virtual machines support *automatic dynamic memory management*.
- Automatic dynamic memory management frees the programmer from the responsibility of explicitly managing memory.
- The programmer can allocate memory without having to worry about deallocation.
- The memory system will automatically:
  - Reclaim unused memory using a *Garbage Collector (GC)*,
  - Expand and shrink data in the heap as necessary,
  - Service weak pointers and perform finalization of objects (if necessary).

Benefits of Automatic Memory Management

- Makes the programmer’s life much easier
  - Removes the problems of explicit deallocation
  - Decreases the risk of memory leaks
  - Simplifies the use of abstract data types
  - Facilitates proper encapsulation
- Generally: ensures that programs are *pointer-safe*
  - No more dangling pointers
- Automatic memory management improves program reliability and safety significantly!!

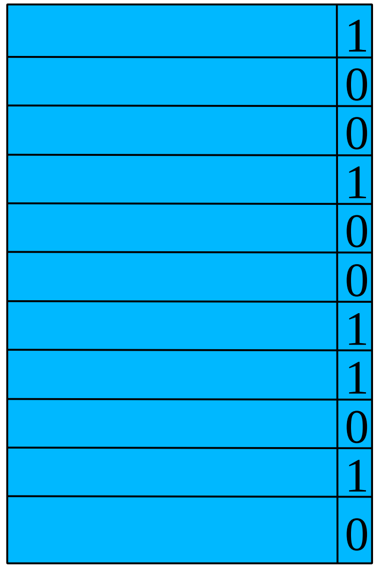
Basic Challenges in Automatic Dynamic Memory Management

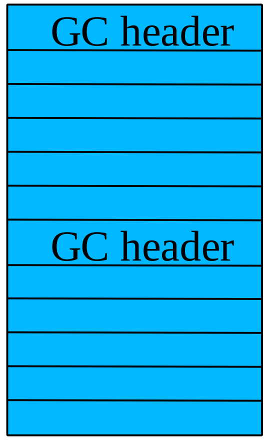
- How does the memory system know where all the pointers are?
- How does the memory system know when it is safe to delete an object?
- How does the memory system avoid memory fragmentation problems?
- If the memory system needs to move an object, how does the system update all the pointers to that object?

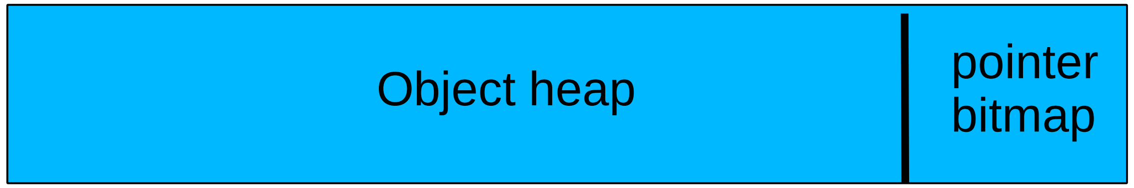
- When implementing a virtual machine, your VM must be able to handle all of this.

How to Keep Track of Pointers?

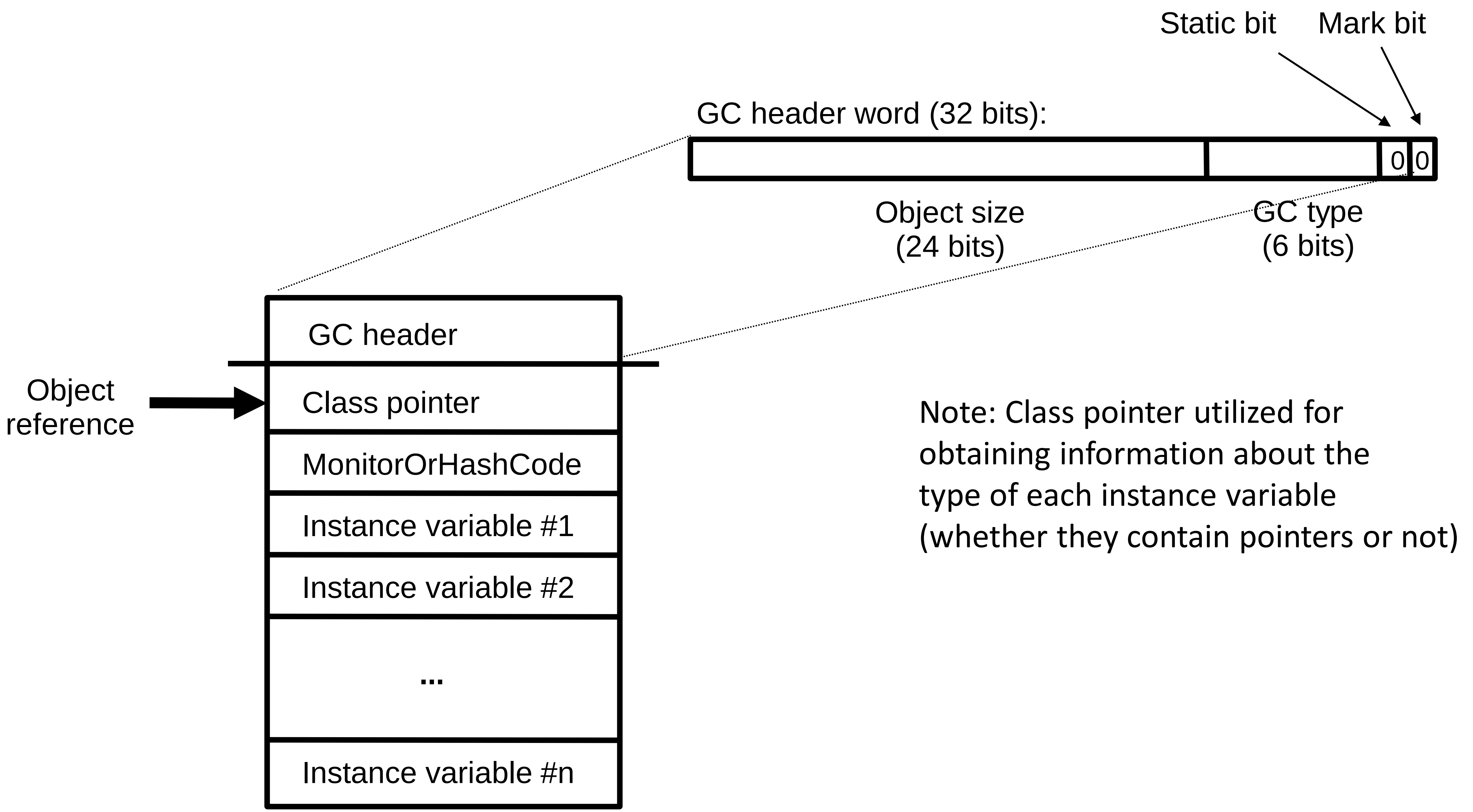
- The memory system must be able to know which memory locations contain pointers and which don't.
- Three basic approaches:
  - In some systems, all memory words are *tagged* with pointer/type information.
  - In some systems, objects have headers that contain pointer information.
  - In some systems, pointer information is kept in separate data structures.







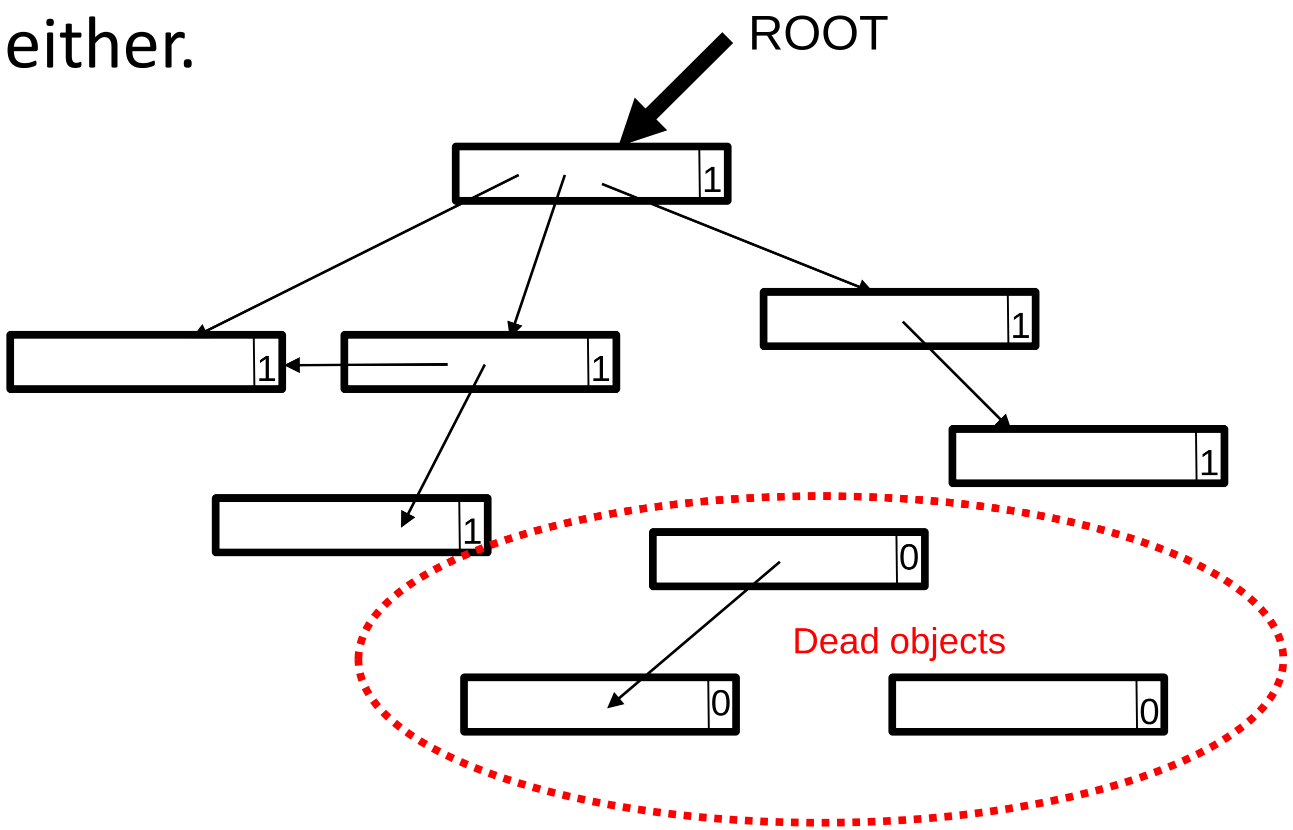
Example: Object Layout in the K Virtual Machine (with explicit GC headers)



Note: Class pointer utilized for obtaining information about the type of each instance variable (whether they contain pointers or not)

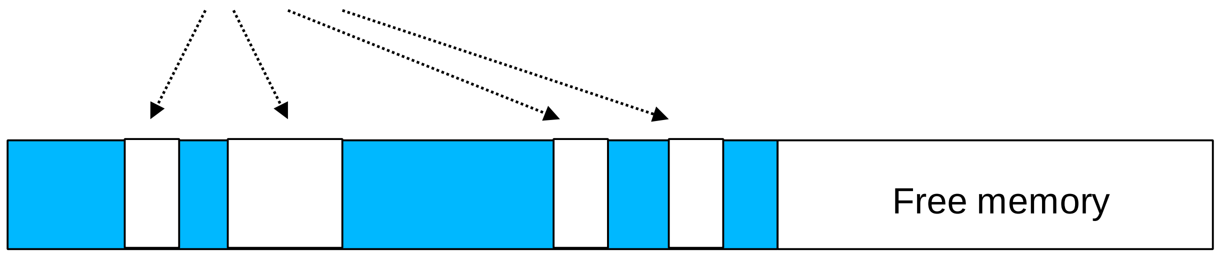
When Is It Safe to Delete An Object?

- Generally, an object can be deleted when there are no more pointers to it.
- All the dependent objects can be deleted as well, if there are no references to them either.



To Compact Memory or Not?

- When objects are deleted, the object heap will contain *holes* unless the heap is *compacted*.



- If a compaction algorithm is used, objects in the heap may move.
  - All pointers to the moved objects must be updated!
- If no compaction is used, the system must be able to manage free memory areas.
  - Often, a *free list* is used to chain together the free areas.
  - Memory allocation will become slower.
  - Fragmentation problems are possible!

Basic Heap Compaction Techniques

1) Two-finger algorithms

- Two pointers are used, one to point to the next free location, the other to the next object to be moved. As objects are moved, a forwarding address is left in their old location.
- Generally applicable only to systems that use fixed-size objects (e.g., Lisp).

2) Forwarding address algorithms

- Forwarding addresses are written into an additional field within each object before the object is moved.
- These methods are suitable for collecting objects of different sizes.

3) Table-based methods

- A relocation map, usually called a *breaktable*, is constructed in the heap either before or during object relocation. This table is consulted later to calculate new values for pointers.
- Best-known algorithm: Haddon-Waite breaktable algorithm; used in Sun's KVM.

4) Threaded methods

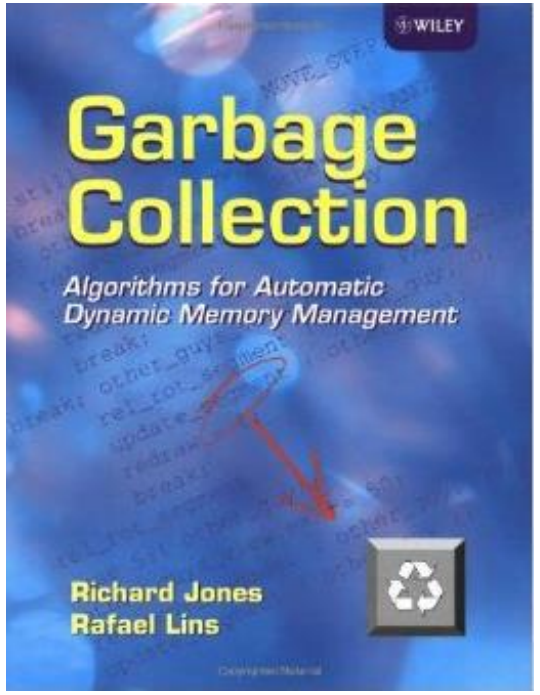
- Each object is chained to a list of those objects that originally pointed to it. When the object is moved, the list is traversed to readjust pointer values.

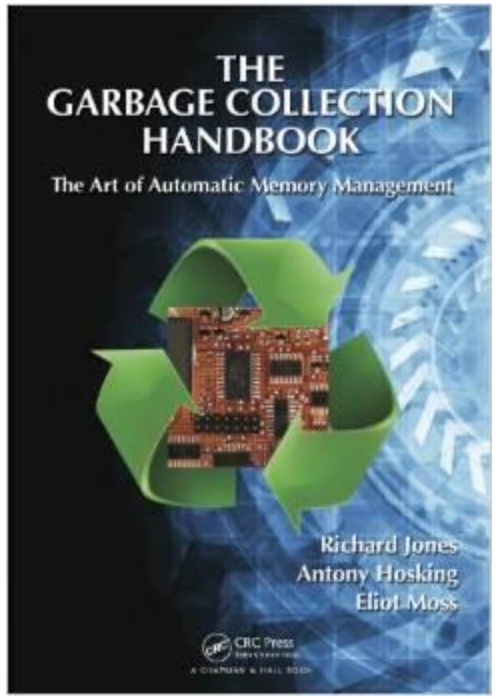
5) Semi-space (copying) compaction

- In copying collectors, compaction occurs as a side-effect to copying.

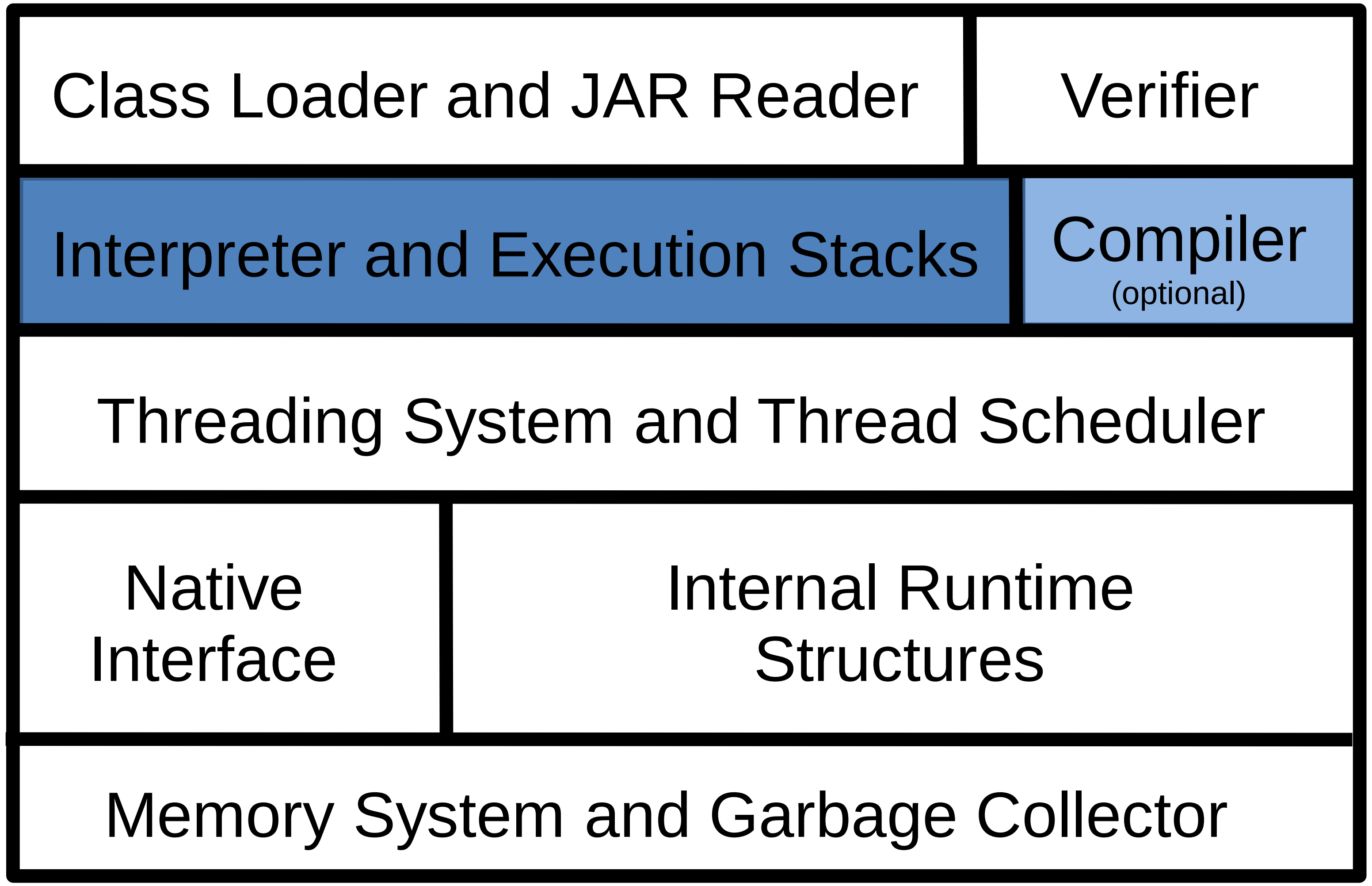
Further Reading on Memory Management and GC

- There are hundreds of garbage collection algorithms.
- For a great overview, read the “bible” of garbage collection (the original 1996 version & the 2012 update).
- http://www.gchandbook.org/





Walkthrough of Essential Component Areas



Interpretation and Execution

Background

- Executing source code directly can be very expensive and difficult.
  - Parsing of source code takes a lot of time and space.
  - In general, source code is intended to be human-readable; it is not intended for direct execution.
- Most virtual machines use some kind of an *intermediate representation* to store programs.
- Most virtual machines use an *interpreter* to execute code that is stored in the intermediate representation.

Two Kinds of Interpreters

Virtual machines commonly use two types of interpreters:

❶ Command-line interpreter ("outer" interpreter / parser)

- Reads and parses instructions in source code form (textual representation).
- Only needed in those systems that can read in source code at runtime.

❷ Instruction interpreter ("inner" interpreter)

- Reads and executes instructions using an intermediate execution format such as bytecodes.

> Parsers are covered well in traditional compiler classes;
> in this lecture we will focus on instruction interpretation

Basics of Inner Interpretation

- The heart of the virtual machine is the inner interpreter.

- The behavior of the inner interpreter:

    1) Read the current instruction,
    2) Increment the instruction pointer,
    3) Parse and execute the instruction,
    4) Go back to (1) to read the next instruction

A Minimal Inner Interpreter Written in C

```
int* ip; /* instruction pointer */
while (true) {
    ((void (*)())*ip++)();
}
```

Components of an Interpreter

- Interpreters usually have the following components:

Interpreter loop

```
while (true) {
    ((void (*)())*ip++)();
}
```

Instruction set

add, mul, sub, ...
load, store, branch, ...
...

Virtual registers

ip fp ...
sp lp

Execution stacks

Operand stack
Execution stack
...

Virtual Registers

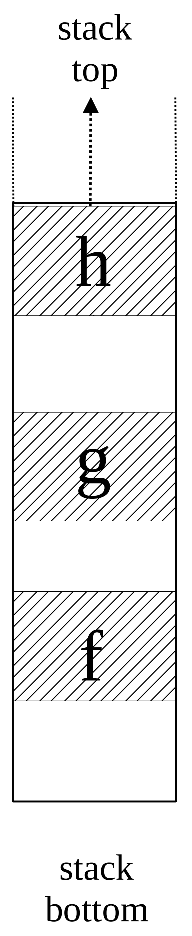
- *Virtual registers* hold the state of the interpreter during execution. Typical virtual registers:
  - *ip*: instruction pointer
    - Points to the current (or next) instruction to be executed.
  - *sp*: stack pointer
    - Points to the topmost item in the operand stack.
  - *fp*: frame pointer
    - Points to the topmost frame (activation record) in the execution stack (call stack).
  - *lp*: local variable pointer
    - Points to the beginning of the local variables in the execution stack.
  - *up*: current thread pointer (if multithreading is required)

Execution Stacks

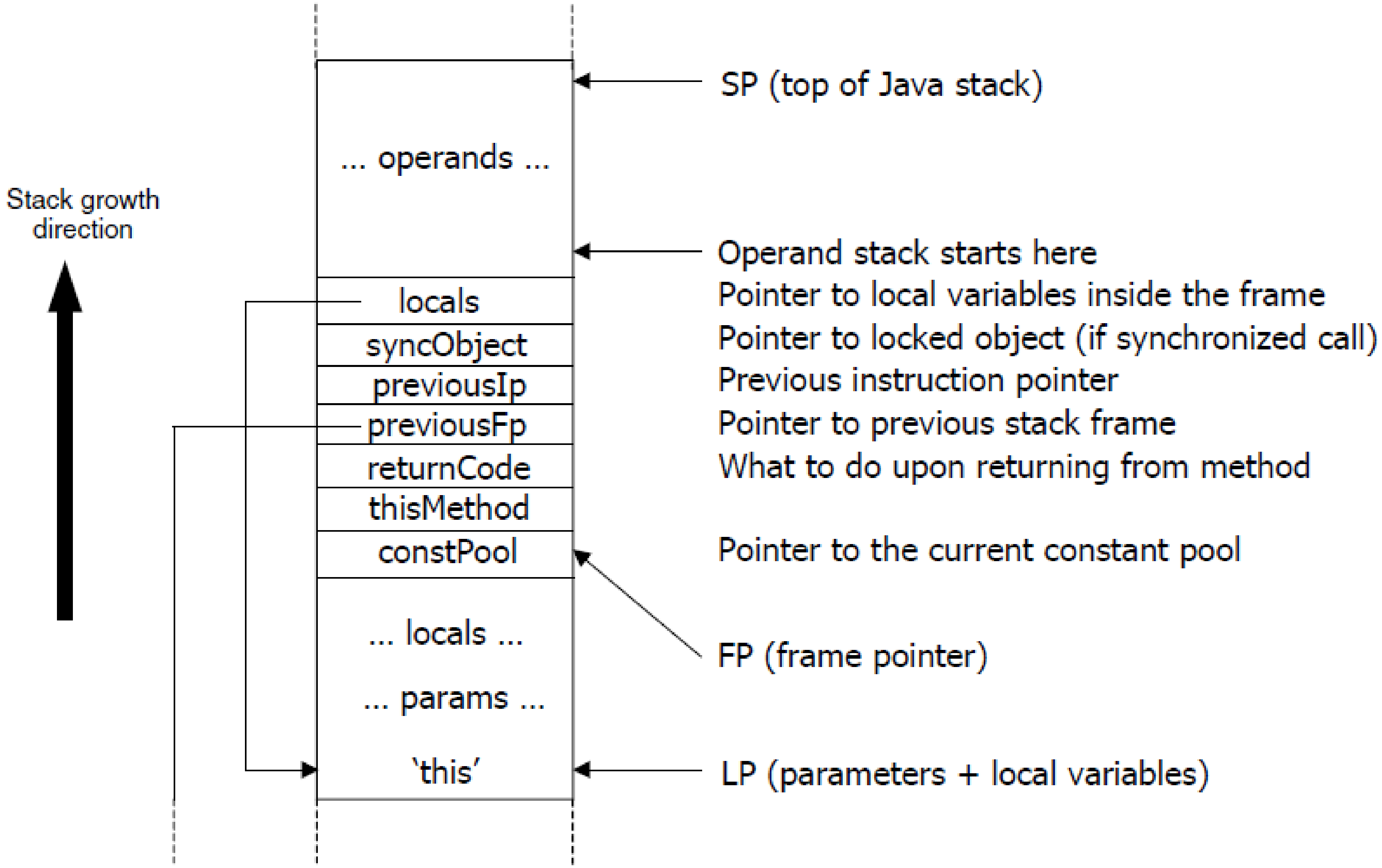
- In order to support method (subroutine) calls and proper control flow, an *execution stack* is typically needed.
  - Also known as the *call stack*.
- *Execution stack* holds the *stack frames* (activation records) at runtime.
  - Allows the interpreter to invoke methods/subroutines and to return to correct locations once a method call ends.
  - Each thread in the VM needs its own execution stack.
- Some VMs use a separate *operand stack* to store parameters and operands.

```
f() {
    g();
}

g() {
    h();
}
```

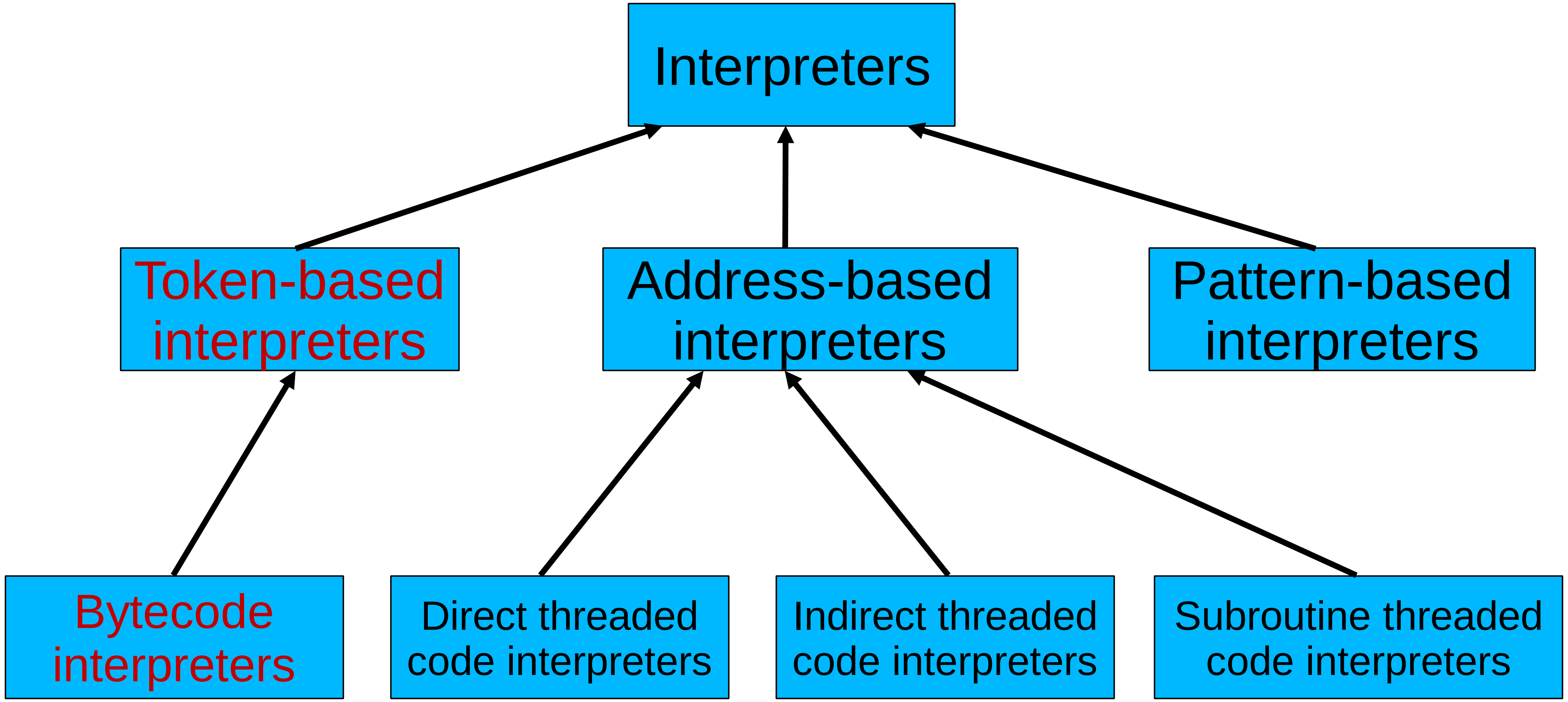


Concrete Example: Stack Frames in the KVM



Fundamental Interpretation Techniques

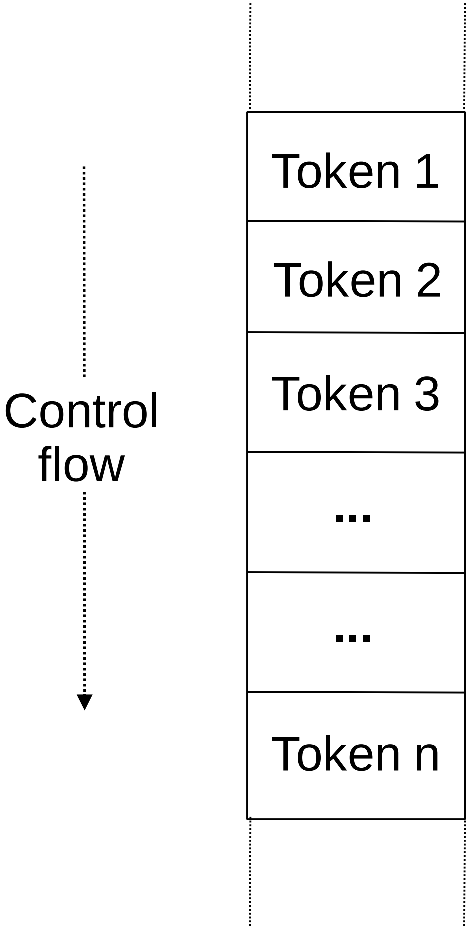
Taxonomy of Interpreters



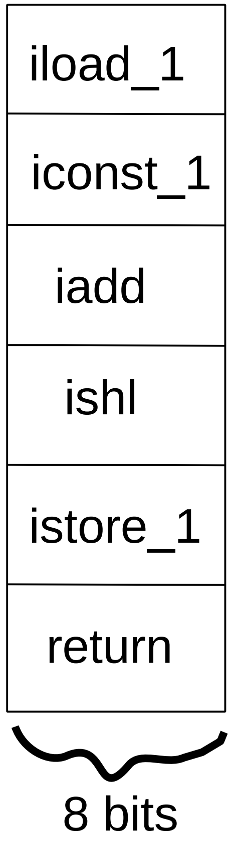
Token-Based Interpretation

- In token-based interpreters, the fundamental instruction unit is a *token*.
  - Token is a predefined numeric value that represents a certain instruction.
    - E.g., 1 = LOAD LITERAL, 2 = ADD, 3 = MULTIPLY, ...
  - Token values are independent of the underlying hardware or operating system.
- The most common subcase:
  - In a *bytecode interpreter*, instruction (token) width is limited to 8 bits.
  - Total instruction set limited to 256 instructions.
  - Bytecode interpreters are very commonly used, e.g.,
    for Smalltalk, Java, and many other interpreted programming languages.

Token-Based Code: Examples



Some Java bytecodes:



Code is represented as linear lists that contain fixed-size tokens. In bytecode, token width is 8 bits.

A Simple Bytecode Interpreter Written in C

```
void Interpreter() {
    while (true) {
        byte token = (byte)*ip++;

        switch (token) {
                case INSTRUCTION_1:
                    break;
                case INSTRUCTION_2:
                    break;
                case INSTRUCTION_3:
                    break;
        }
    }
}
```

Instruction Sets

Instruction Sets

- Each virtual machine typically has its own instruction set based on the requirements of the language(s) the VM must support.
- These instruction sets are similar to instruction sets of hardware CPUs.
- Common types of instructions:
  - Local variable load and store operations
  - Constant value load operations
  - Array load and store operations
  - Arithmetic operations (add, sub, mul, div, ...)
  - Logical operations (and, or, xor, ...)
  - Type conversions
  - Conditional and unconditional branches
  - Method invocations and returns
  - ...

Stack-Oriented vs. Register-Oriented Instruction Sets

- Two types of instruction sets:
  1. In *stack-oriented* instruction sets, operands to most instructions are passed in an operand stack; this stack can grow and shrink dynamically as needed.
  2. In *register-oriented* instruction sets, operands are accessed via “register windows”: fixed-size areas that are allocated automatically upon method calls.

- Historically, most virtual machines used a stack-oriented instruction set.
  - Stack machines are generally simpler to implement.
  - No problems with “running out of registers”; the instruction set can be smaller.
  - Less encoding/decoding needed to parse register numbers.

- Unlike JVM, Dalvik uses a register-based instruction set.

Criteria	Dalvik	JVM
Architecture	Register-based	Stack-based
OS-Support	Android	All
Executables	DEX	JAR
Constant-pool	per application	per Class

Example: The Java Bytecode Interpreter

- The JVM uses a straightforward stack-oriented bytecode instruction set with approximately 200 instructions.
  - Fairly similar to the Smalltalk bytecode set, except that in Java primitive data types are not objects.
- One execution stack is required per each Java thread.
  - No separate operand stack; operands are kept on top of the current stack frame.
- Four virtual registers are commonly assumed:
  - *ip* (instruction pointer): points to current instruction
  - *sp* (stack pointer): points to the top of the stack
  - *fp* (frame pointer): provides fast access to stack frame
  - *lp* (locals pointer): provides fast access to local variables

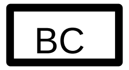
Example: The Java Virtual Machine Instruction Set

aaload	daload	f2l	getstatic	if_icmplt	invokevirtual_quick	ldc_w	new
aastore	dastore	fadd	getstatic	if_icmpne	invokevirtual_quick_w	ldc_w_quick	new_quick
aconst_null	dcmpg	faload	getstatic_quick	ifeq	invokevirtualobject_quick	ldc2_w	newarray
aload	dcmpl	fastore	getstatic2_quick	ifge	ior	ldc2_w_quick	nop
aload_0	dconst_0	fcmpg	goto	ifgt	irem	ldiv	pop
aload_1	dconst_1	fcmpl	goto_w	ifle	ireturn	lload	pop2
aload_2	ddiv	fconst_0	i2b	iflt	ishl	lload_0	putfield
aload_3	dload	fconst_1	i2c	ifne	ishr	lload_1	putfield
anewarray	dload_0	fconst_2	i2d	ifnonnull	istore	lload_2	putfield_quick
anewarray_quick	dload_1	fdiv	i2f	ifnull	istore_0	lload_3	putfield_quick_w
areturn	dload_2	fload	i2l	iinc	istore_1	lmul	putfield2_quick
arraylength	dload_3	fload_0	i2s	iload	istore_2	lneg	putstatic
astore	dmul	fload_1	iadd	iload_0	istore_3	lookupswitch	putstatic
astore_0	dneg	fload_2	iaload	iload_1	isub	lor	putstatic_quick
astore_1	drem	fload_3	iand	iload_2	iushr	lrem	putstatic2_quick
astore_2	dreturn	fmul	iastore	iload_3	ixor	lreturn	ret
astore_3	dstore	fneg	iconst_0	impdep1	jsr	lshl	return
athrow	dstore_0	frem	iconst_1	impdep2	jsr_w	lshr	saload
baload	dstore_1	freturn	iconst_2	imul	l2d	lstore	sastore
bastore	dstore_2	fstore	iconst_3	ineg	l2f	lstore_0	sipush
bipush	dstore_3	fstore_0	iconst_4	instanceof	l2i	lstore_1	swap
breakpoint	dsub	fstore_1	iconst_5	instanceof_quick	ladd	lstore_2	tableswitch
caload	dup	fstore_2	iconst_m1	invokeinterface	laload	lstore_3	wide
castore	dup_x1	fstore_3	idiv	invokeinterface_quick	land	lsub	xxxunusedxxx
checkcast	dup_x2	fsub	if_acmpeq	invokenonvirtual_quick	lastore	lushr	
checkcast_quick	dup2	getfield	if_acmpne	invokespecial	lcmp	lxor	
d2f	dup2_x1	getfield	if_cmpge	invokestatic	lconst_0	monitorenter	
d2i	dup2_x2	getfield_quick	if_icmpeq	invokestatic_quick	lconst_1	monitorexit	
d2l	f2d	getfield_quick_w	if_icmpgt	invokesuper_quick	ldc	multianewarray	
Dadd	f2i	getfield2_quick	if_icmple	invokevirtual	ldc_quick	multianewarray_quick	

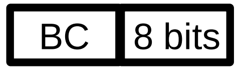
(Note: “_quick” bytecodes are non-standard and implementation-dependent)

JVM Instruction Formats

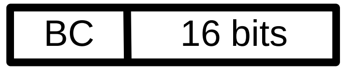
1. Most Java bytecodes do not require any parameters from the instruction stream.
   - They operate on the values provided on the execution stack (e.g., IADD, IMUL, ...)



2. Some bytecodes read an additional 8-bit parameter from the instruction stream.
   - For instance, NEWARRAY, LDC, *LOAD, *STORE



3. Many bytecodes read additional 16 bits from the instruction stream.
   - INVOKE* instructions, GET/PUTFIELD, GET/PUTSTATIC, branch instructions, ...



4. Three instructions are varying-length.
   - LOOKUPSWITCH, TABLESWITCH, WIDE

Example: IFNULL

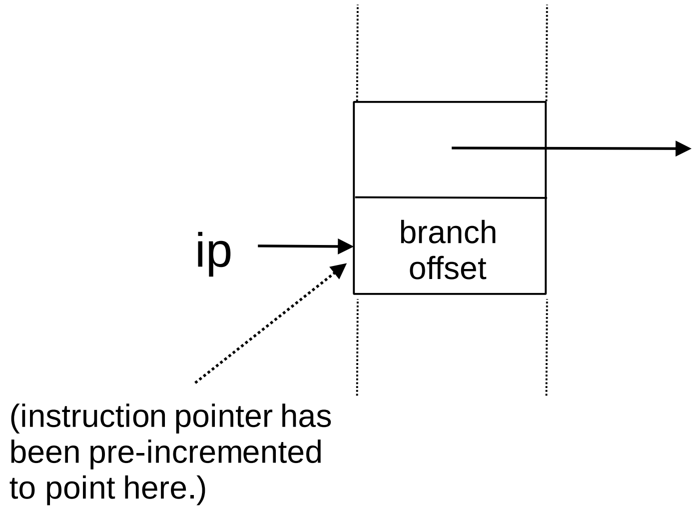
IFNULL	16-bit parameter

Operand stack: ..., *value* => ...

- IFNULL: Branch if reference is null.
- The instruction pops value off the operand stack, and checks if the value is NULL.
- The 16-bit parameter contains a branch offset that is added to the instruction pointer if value is NULL.
- Otherwise, execution continues normally from the next instruction.

Accessing Inline Parameters

- Inline parameters are generally very easy to access.
- Use the instruction pointer to determine the location.
- For instance, a bytecode for performing an unconditional jump could be written as follows:




```
void UnconditionalBranch() {
    int branchOffset = *ip;
    ip += branchOffset;
}
```

- Important: Keep in mind the endianness issues!
  - In Java, all numbers in classfiles are *big-endian*; if a machine-specific endianness was used, Java class files wouldn't be portable across different machines.

Remarks on Interpreter Performance

Interpretation Overhead

- Interpreted code is generally a lot slower than compiled code/machine code.
  - Studies indicate an order of magnitude difference.
  - Actual range is something like 2.5x to 50x.

- Why? Because there are extra costs associated with interpretation:
  - Dispatch (fetch, decode and invoke) next instruction
  - Access virtual registers and arguments
  - Perform primitive functions outside the interpreter loop

- *“Interpreter performance is primarily a function of the interpreter itself and is relatively independent of the application being interpreted.”*

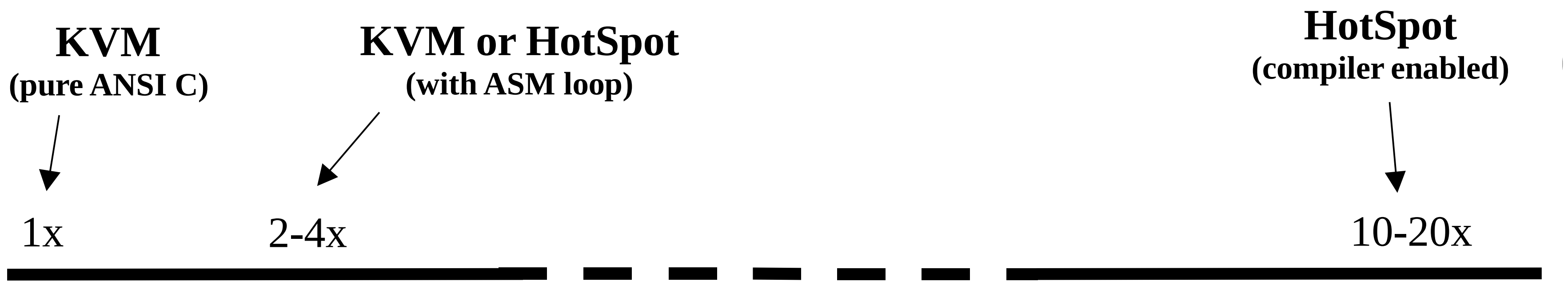
Interpreter Tuning

- Common interpreter optimizations techniques:
  - Writing the interpreter loop and key instructions in assembly code.
  - Keeping the virtual registers (ip, sp, ...) in physical hardware registers – this can improve performance dramatically.
  - Splitting commonly used instructions into a separate interpreter loop & making the core interpreter so small that it fits in HW cache.
  - Top of stack caching (keeping topmost operand in a register).
  - Padding the instruction lookup table so that it has exactly 16/32/64/128/256 entries.

- Actual impact of such optimizations will vary considerably based on underlying hardware.

High-Performance VMs

- Small, simple & portable VM == slow VM
- Interpreted code has a big performance overhead compared to native code:



- If you need speed, you need a compiler!
  - Unfortunately, compilers are always rather machine/CPU-specific.
  - This introduces a lot of additional complexity & requires a lot more manpower to implement.
- Almost always: Fast == more complex

Compilation: Basic Strategies

❶ Static / Ahead-Of-Time (AOT) Compilation

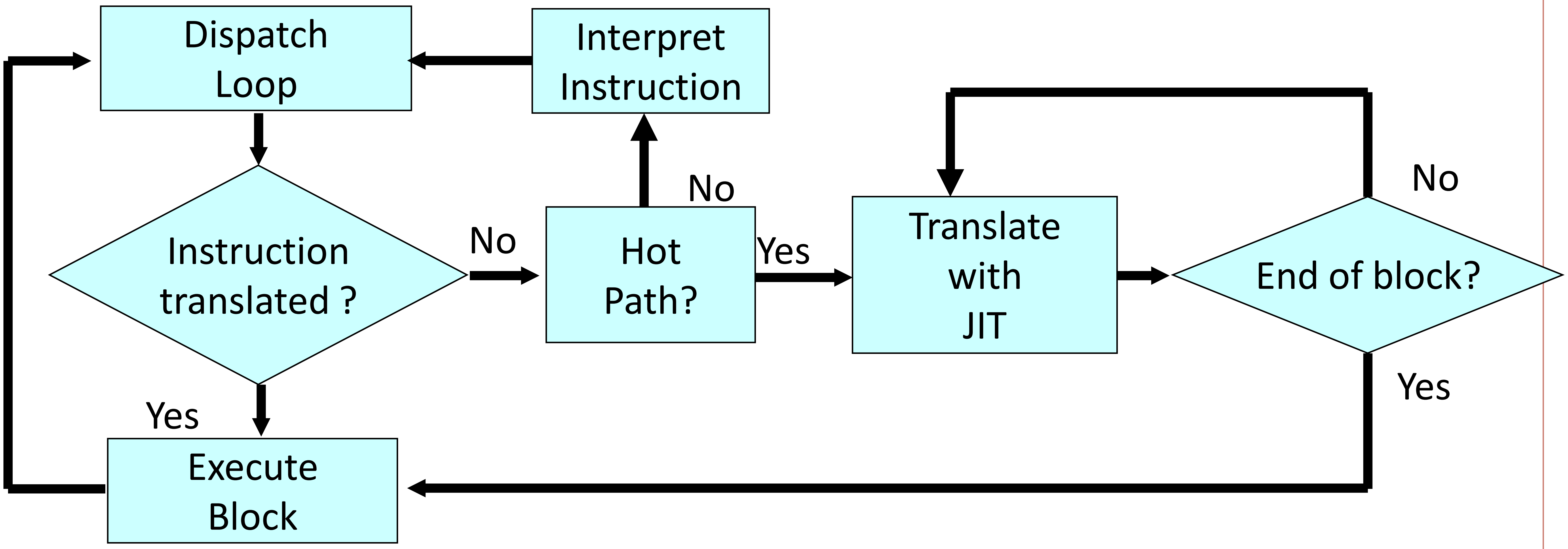
– Compile code before the execution begins.

❷ Dynamic (Just-In-Time, JIT) Compilation

– Compile code on the fly when the VM is running.

- Different flavors of dynamic compilation:
  - Compile everything upon startup (impractical)
  - Compile each method when executed first time
  - Adaptive compilation based on “hotspots” (frequently executed code)

Dynamic Compilation/Translation Loop in VM

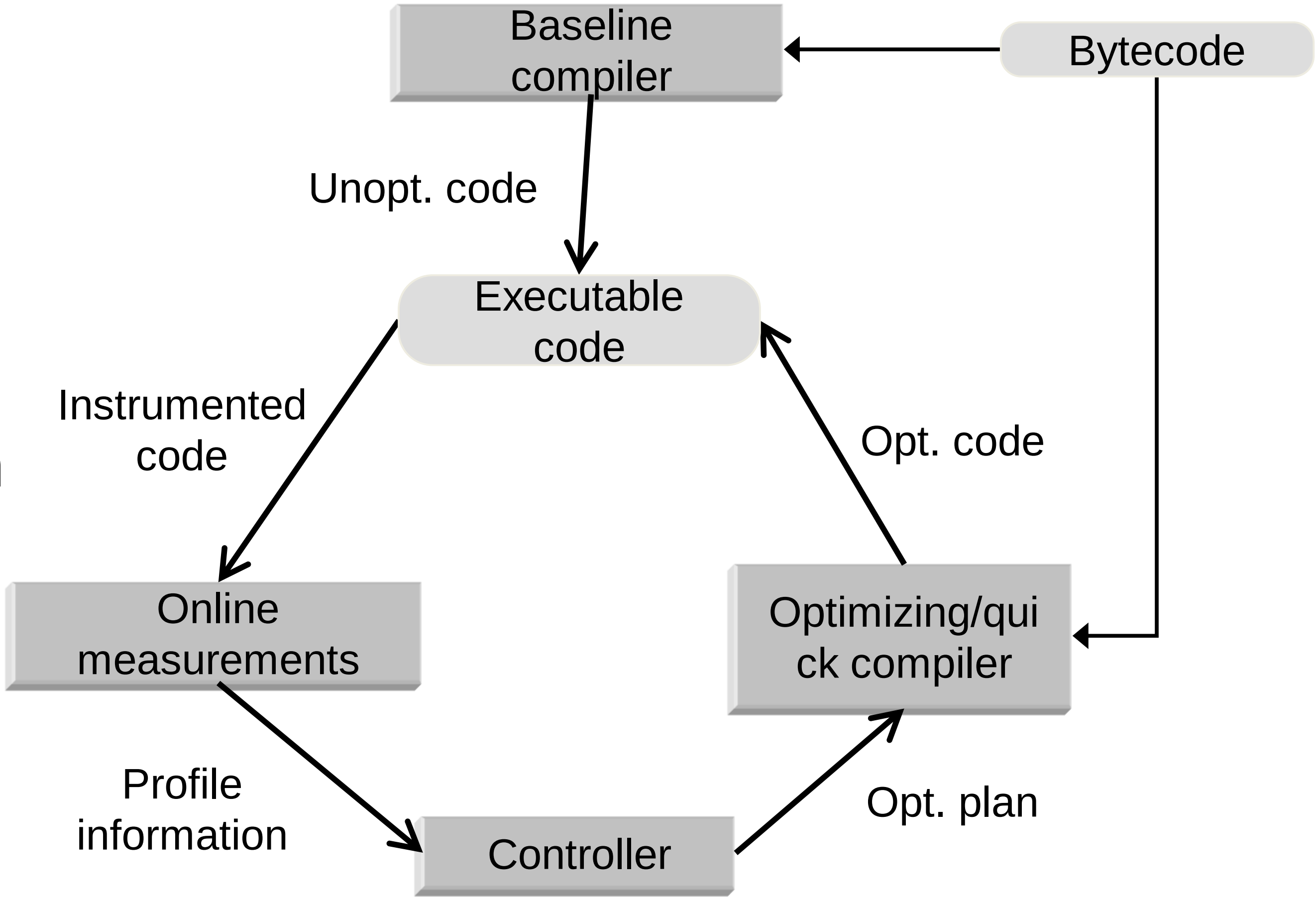


Sun Hotspot JVM

- Derived from Self VM
- Applies basic lazy compilation model
  - Code is initially run interpreted
  - Compile after certain number of invocations of a method
- Client compiler
  - Fast compiler performing minimal optimizations
- Server compiler
  - Aggressive SSA dataflow compiler
  - ~10x slower code generation, but 20%-50% faster code

IBM Jalapeno VM (aka Jikes VM)

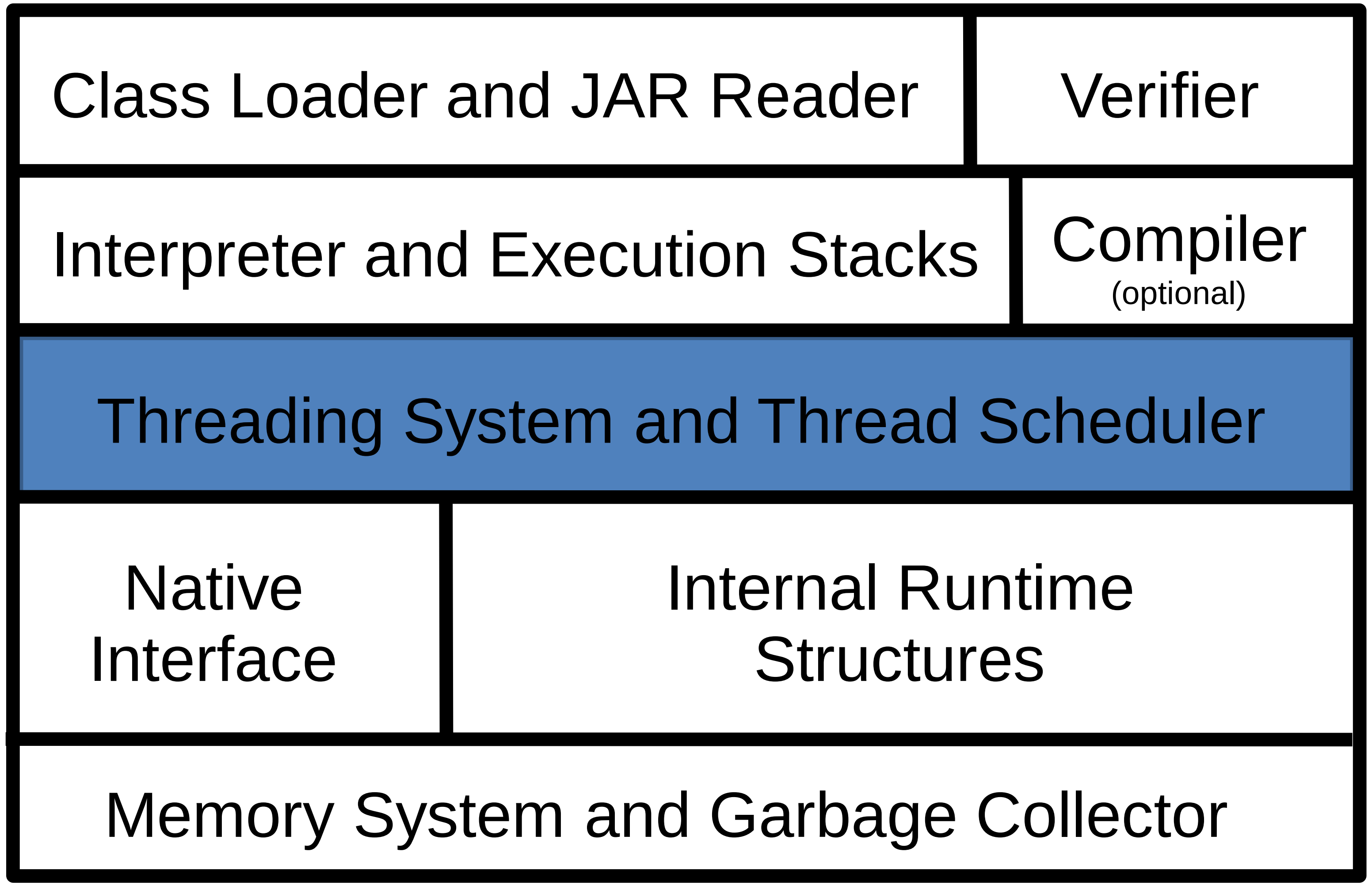
- Entire JVM written in Java
  - Allows VM code to be inlined into programmer code
- Basic lazy compilation
  - Uses three dynamic compilers
  - No interpreter engine
- Full support for adaptive optimization
  - Compiler directed edge sampling & instrumented profiling
  - Profiling directed by compiler optimization passes
- Successfully implemented many profile directed optimizations



Java Dynamic Optimization

- Inlining [Jalapeno]
  - Use profiling to identify hot paths for inlining
  - Reduce code size and compilation time
  - Provides slight performance addition to static inlining
- Speculative specialization [Hotspot]
  - Method target specialization
    - Inline most common target method with a check for expected target
  - Eliminate class check if virtual / interface method targets known given current class hierarchy
  - Decompile inlining if loaded class breaks assumption
  - Works very well for many programs since many methods declared virtual even if they are not subclassed

Walkthrough of Essential Component Areas



Adding Multithreading Support

Multithreading and Synchronization

- A key feature of many programming languages is *multithreading.*
  - *Multithreading*: the ability to create multiple concurrently running threads/programs.
  - Smalltalk, Forth, Ada, Self, Java, ...

- Each thread behaves as if it owns the entire virtual machine
  - ... except when the thread needs to access external resources such as storage, network, display, or perform I/O operations in general.
  - ...or when the thread needs to communicate with the other threads.

- *Synchronization/locking* mechanisms are needed to ensure controlled communication and controlled access to external resources.

Implementing Multithreading: Technical Challenges

- Each thread must have its own virtual registers and execution stacks.
- Critical places of the VM (and libraries) must use mutual exclusion to avoid deadlocks and resource conflicts.
- Access to external resources (e.g., storage, network) must be controlled so that two threads do not interfere with each other.
- I/O operations must be designed so that one thread's I/O operations do not block the I/O of other threads.
- Generally: All native function calls must be non-blocking.
- Locking / synchronization operations must be provided also at the application level.

Recap: Components of an Interpreter

- Interpreters usually have the following components:

Interpreter loop

```
while (true) {
    ((void (*)())*ip++)();
}
```

Instruction set

add, mul, sub, ...
load, store, branch, ...
...

Virtual registers

ip fp ...
sp lp

Execution stacks

Operand stack
Execution stack
...

Building a Multithreading VM

Interpreter loop

```
while (true) {
    ((void (*)())*ip++)();
}
```

Instruction set

add, mul, sub, ...
load, store, branch, ...
...

Virtual registers

ip fp
sp lp
...

...

Execution stacks

Operand stack
Execution stack
...

...

In principle, building a multithreading interpreter is easy: we must simply replicate the virtual registers and stacks for each thread!

Building a Multithreading VM: Supporting Interrupts

- In addition, we must modify the system so that it allows the current thread to be *interrupted*.
- When a thread is interrupted, a *context switch* is performed.

Example:

```
int* ip; /* instruction pointer */
while (true) {
        if (isInterrupted()) ContextSwitch();
        ((void (*)())*ip++)();
}
```

Building a Multithreading VM: Context Switching

- What happens during a context switch?
  1. The virtual registers of the current thread are stored (context save).
  2. Current thread pointer is changed to point to the new current thread.
  3. Virtual registers are replaced with the saved virtual registers of the new current thread (context load).

- Context switching must be performed as an uninterrupted operation!
  - No further interrupts may be processed until the context switch operation has been performed to completion.
  - Operating systems commonly have a “supervisor mode” for running system-critical code.

Avoiding Atomicity Problems Using Safepoints

- In operating systems, threads can usually be interrupted at arbitrary locations.
  - Interrupts may be generated by hardware at any time.
  - The entire operating system must be designed to take into account mutual exclusion problems!
  - Must use monitors or semaphores to protect code that can be executed only by one thread at the time.
- In VMs, simpler solutions are often used.
  - Threads can only be interrupted in certain locations inside the VM source code.
  - These locations are known as "*safepoints*".
  - In the simplest case, thread switching is only allowed in one place inside the VM.
  - Makes VM design a lot simpler and more portable!

Using the “One Safepoint” Solution

- No separate interrupt handler routine.
- All the checking for interrupts happens inside the interpreter loop.
- Even if the actual interrupts are generated asynchronously, the actual context switching is not performed until the interpreter loop gets a chance to detect and process the interrupt:

```
int* ip; /* instruction pointer */
while (true) {
        if (isInterrupted()) ContextSwitch();
        ((void (*)())*ip++)();
}
```

Making Thread Switching 100% Portable

- In a virtual machine, you don't necessarily need an external clock to drive the interrupts!
- In the simplest case, you can count the number of executed instructions using a “timeslice”:

```
int* ip; /* instruction pointer */
while (true) {
    if (--TimeSlice <= 0) ContextSwitch();
    ((void (*)())*ip++)();
}
```

  - Force a context switch every 1000 bytecodes or so.
- You can also enforce a thread switch at each I/O request (used in many Forth systems).

Thread Scheduling

- When you perform a context switch, which thread should run next?
- Various choices:
  - FCFS (First-Come-First-Serve)
  - Round robin approach
  - Priority-based scheduling
    - ... with fixed priorities or varying priorities
- In operating systems, thread scheduling algorithms can be rather complicated.
- There is no “ideal” scheduling algorithm.
- The needs of interactive and non-interactive programs (and client vs. server software) can be fundamentally different in this area.

Case Study: KVM

- The original KVM implementation used a simple, portable *round robin* scheduler.
  - Threads stored in a circular list; each thread got to execute a fixed number of bytecodes until interrupt was forced.
  - Thread priority only affected the number of bytecodes a thread may run before it gets interrupted.

- In the actual product version, thread scheduling based on Java thread priority.
  - Higher-priority threads always run first.

- Fully portable thread implementation; interrupts driven by bytecode counting; thread switching handled inside the interpreter loop.

Actual Code: Thread Switching in KVM

Code inside the interpreter loop:

```
if (--Timeslice <= 0) {
    do {
        ulong64 wakeupTime;

        /* Check if it is time to exit the VM */
        if (AliveThreadCount == 0) return;

        /* Check if it is time to wake up */
        /* threads waiting in the timer queue */
        checkTimerQueue(&wakeupTime);

        /* Handle external events */
        InterpreterHandleEvent(wakeupTime);

     } while (!SwitchThread());
}
```

Thread switching code (simplified):

```
bool_t SwitchThread() {
    /* Store current context */
    StoreVirtualRegisters();

    /* Obtain next thread to run */
    CurrentThread =
        removeQueueStart(&RunnableThreads);
    if (CurrentThread == NULL) return FALSE;

    /* Set new context and timeslice */
    LoadVirtualRegisters(CurrentThread);
    Timeslice = CurrentThread->timeslice;

    return TRUE;
}
```

Java

- Developed by James Gosling's team at Sun Microsystems in the early 1990s.
- Originally designed for programming consumer devices (as a replacement for C++).
  - Uses a syntax that is familiar to C/C++ programmers.
  - Uses a portable virtual machine that provides automatic memory management and a simple stack-oriented instruction set.
  - *Class file verification* was added to enable downloading and execution of remote code securely.
- Again, great timing: the development of the Java technology coincided with the widespread adoption of web browsers in the mid-1990s.

Why is Java Interesting from VM Designer's Viewpoint?

- Most people had never heard of virtual machines until Java came along!
- Java brought virtual machines to the realm of mobile computing.

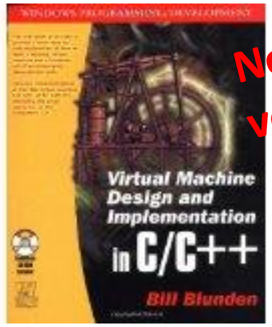
- Java combines a statically compiled programming language with a dynamic virtual machine.
- The Java virtual machine (JVM) is very well documented.
  - Tim Lindholm, Frank Yellin, *The Java Virtual Machine Specification*, Second Edition, Addison Wesley, Java Series, April 1999.
- A JVM is seemingly very easy to build.
- However, tight compatibility requirements make the actual implementation very challenging.
  - Must pass tens of thousands of test cases to prove compatibility.

Java for Android / Dalvik VM

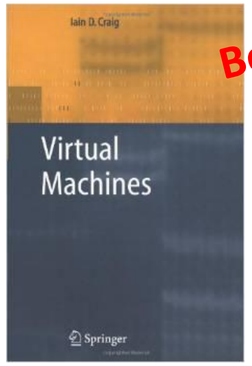
- Android resurrected interest in Java in the mobile space.
- *Dalvik* is an alternative runtime environment for executing Java programs, using an Android-specific application format (.dex files).
- Unlike JVM, which uses a stack-based architecture, Dalvik uses a register-based architecture and bytecode set.
- As of Android 5.0 (Lollipop), the Dalvik VM will be replaced by Android Runtime (ART) – an architecture based on ahead-of-time compilation.

More Information on Virtual Machine Design

- Bill Blunden, *Virtual Machine Design and Implementation in C/C++*, Wordware Publishing, March 2002
- Iain D. Craig, *Virtual Machines*, Springer Verlag, September 2005
- Jim Smith, Ravi Nair, *Virtual Machines: Versatile Platforms for Systems and Processes*, Morgan Kaufmann, June 2005
- Xiao-Feng Li, Jiu-Tao Nie, Ligang Wang, *Advanced Virtual Machine Design and Implementation*, CRC Press, October 2014
- Matthew Portnoy, *Virtualization Essentials*, Sybex Press, May 2012



Not that good; very narrow scope



Better

Worth buying!

Brand new – Haven't read yet

Good introduction to OS virtualization!

18-600 Foundations of Computer Systems

Lecture 20:
"Parallel Systems & Programming"

John Paul Shen
November 8, 2017

Next Time …

➢ Recommended Reference:
- "Parallel Computer Organization and Design," by Michel Dubois, Murali Annavaram, Per Stenstrom, Chapters 5 and 7, 2012.

Electrical & Computer ENGINEERING