The Squawk Java Virtual Machine

Doug Simon
Sun Labs

Overview

- Context and motivation
- VM architecture & implementation
- ARM port details
- Lessons learnt
- Future work

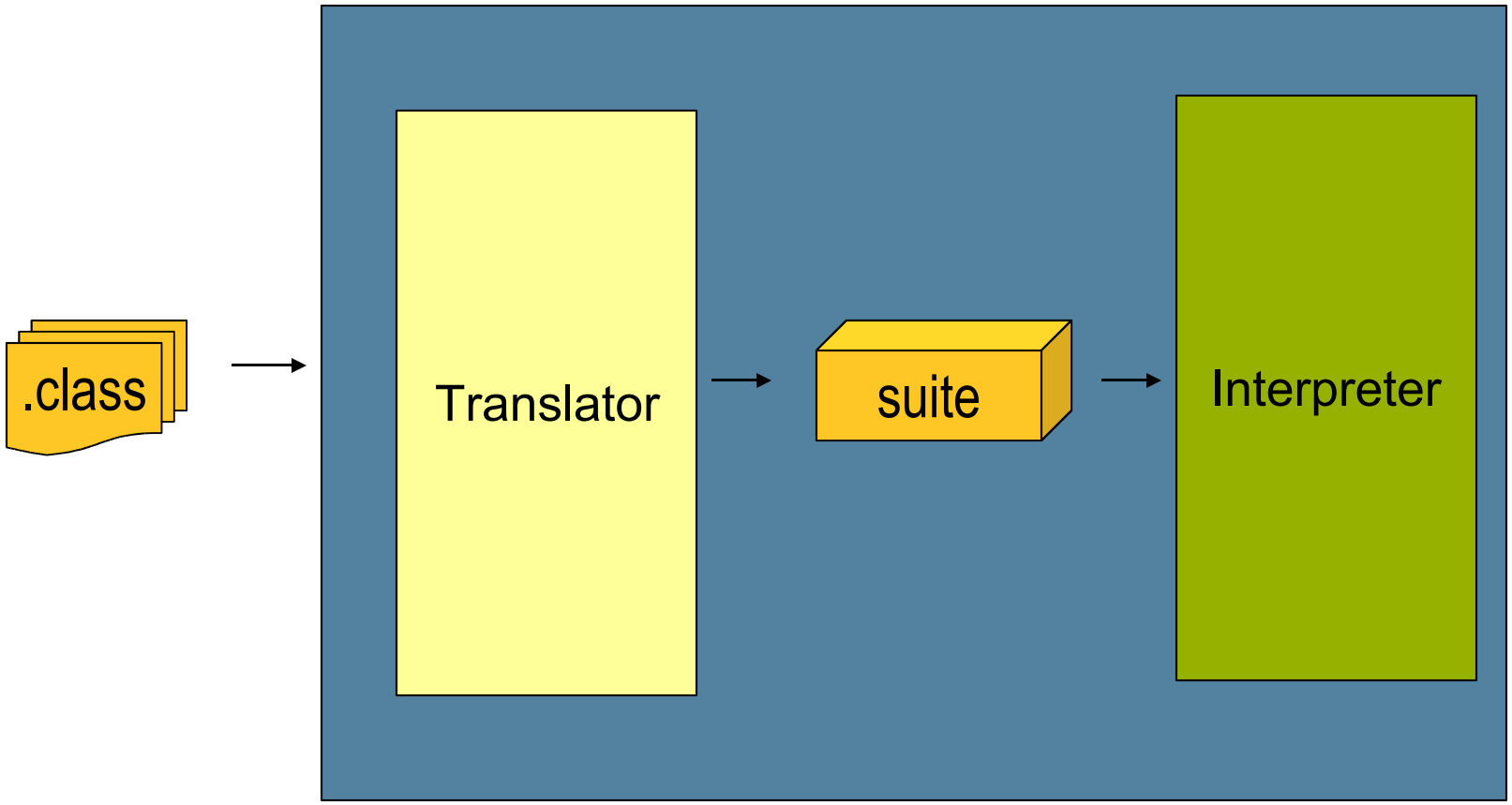
Context and motivation

- Spotless & KVM experience
- Alternative way to construct JVMs?
  - Do it in Java!
    - Pointer safety, exceptions, GC...
    - More portable
    - Ease of development
    - Extend loader/verifier to simplify execution
- What does this approach enable?
- Goals:
  - CLDC implementation for memory constrained devices
  - Can run on bare metal (i.e. no OS required)

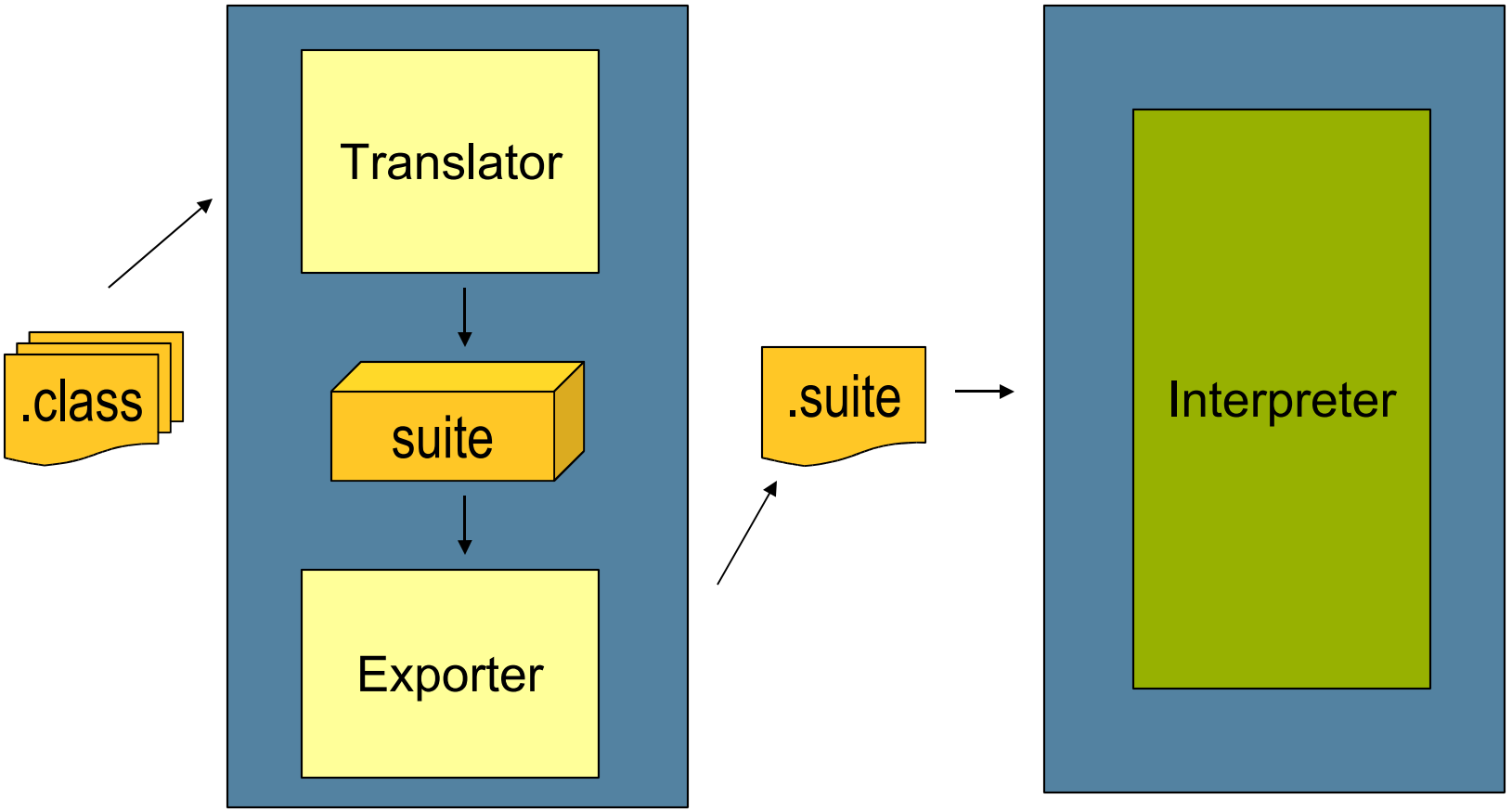
Architecture

- Native
  - Interpreter
  - Platform IO
- Java
  - Translator
    - Loader, verifier & bytecode transformation
  - Core
    - Thread scheduler
    - Exception handling
    - Synchronization
    - Garbage collector
  - Exporter
  - JDWP Debug Agent
  - CLDC API

Architecture



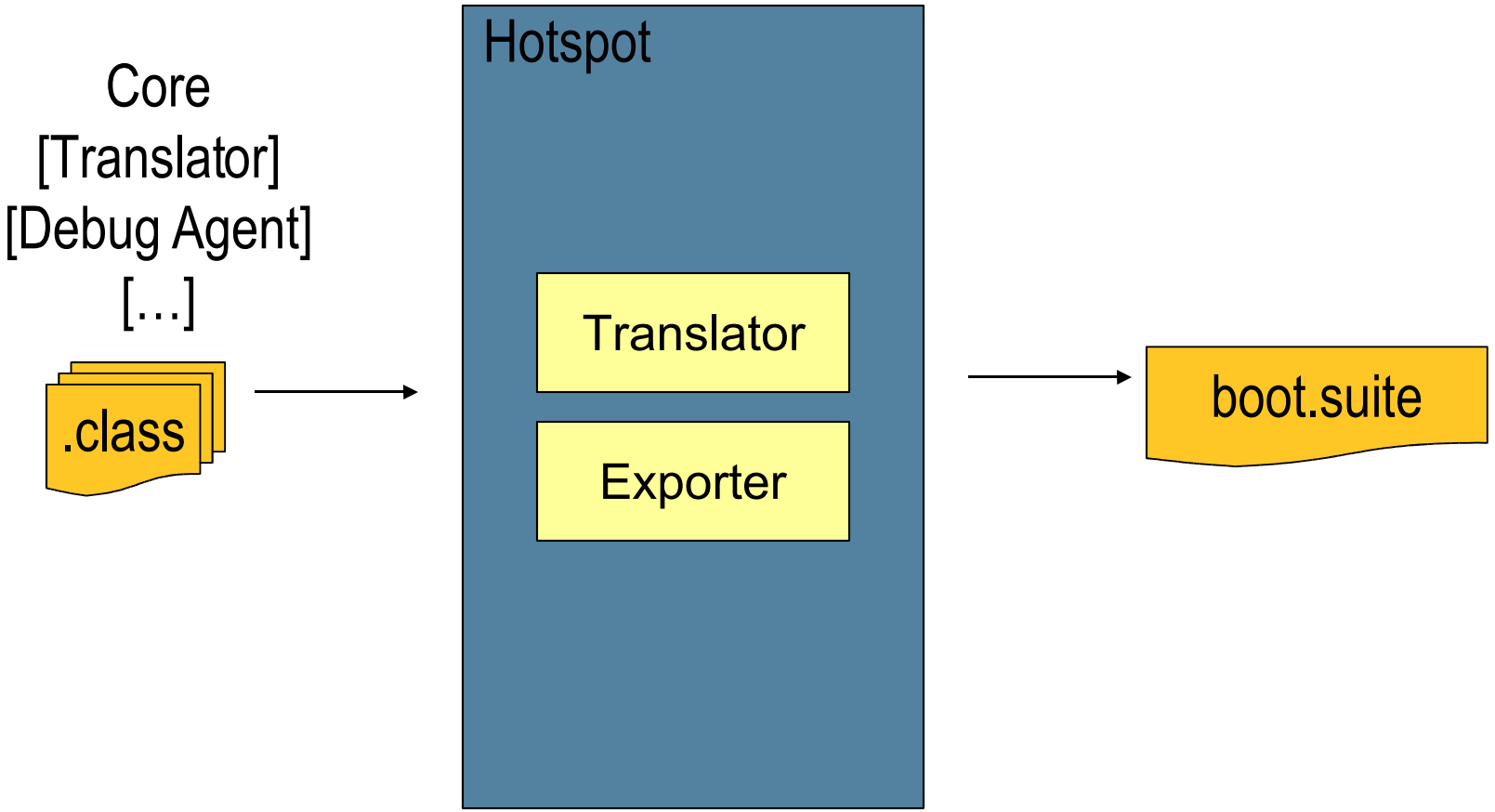
Split VM Architecture



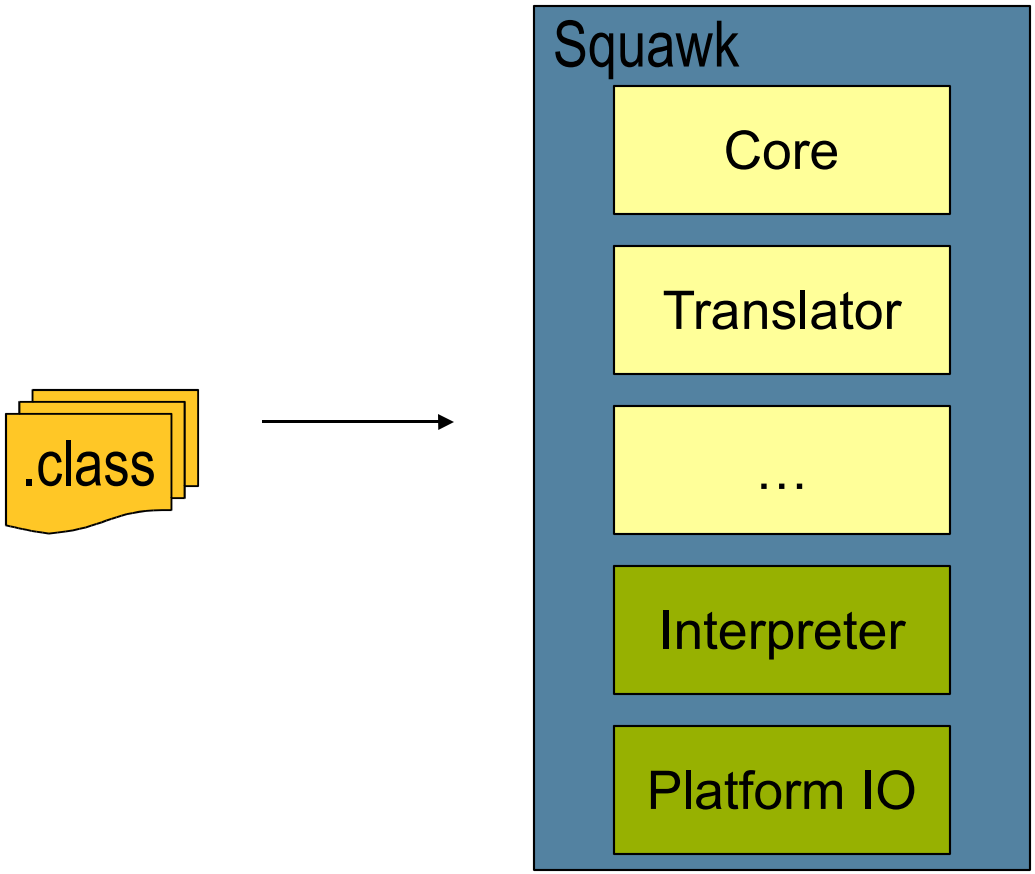
Architecture

- Different configurations for different needs
- Bootstrapping
  - Used to create boot .suite file
- Standard
  - CLDC compliant JVM
  - Used to create (non-boot) .suite files
- Minimal
  - Minimizes static and dynamic on-device memory requirements

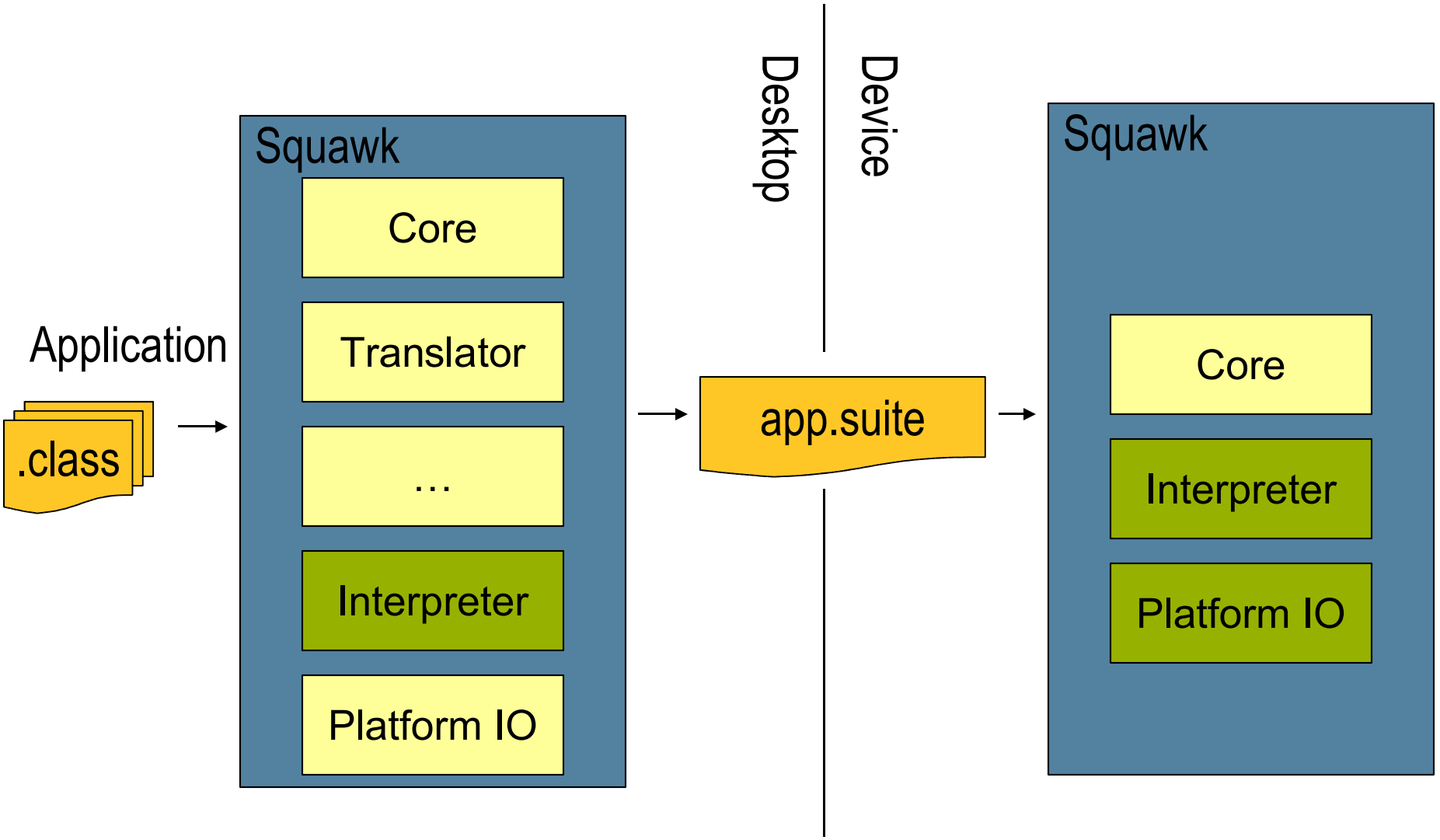
Bootstrapping



Standard



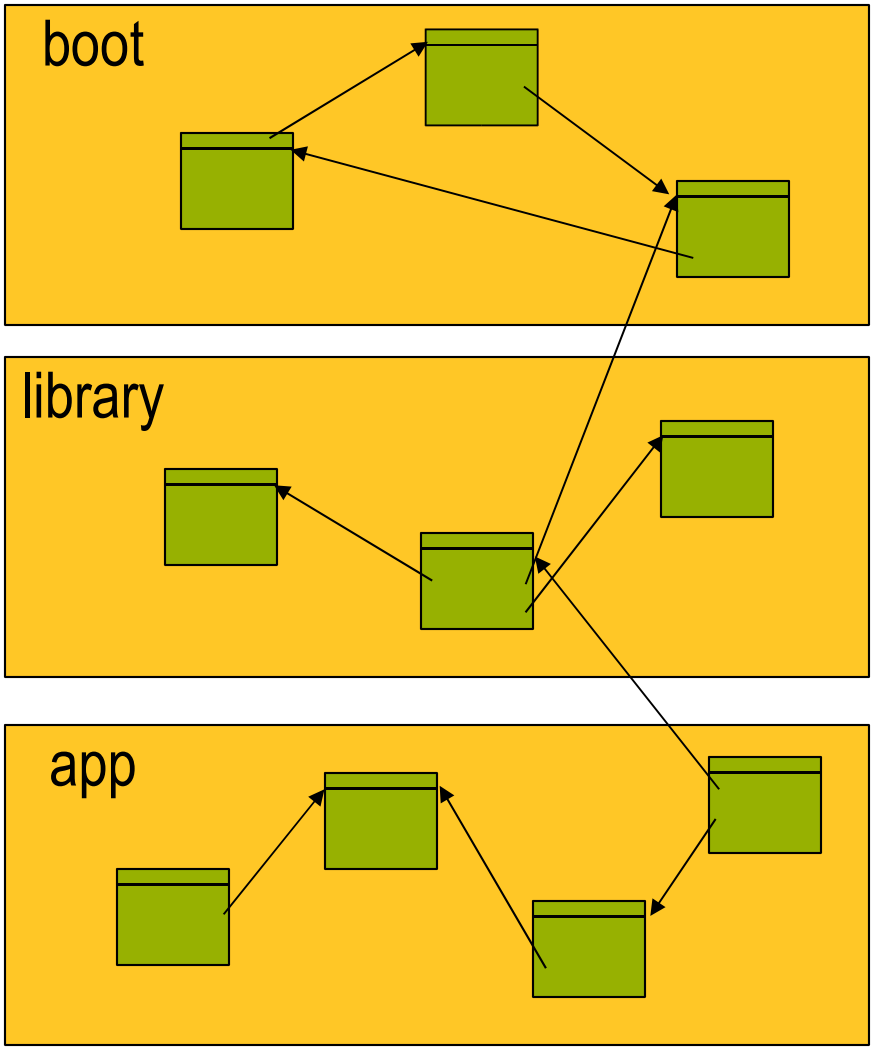
Minimal



Suites

- A suites is a preprocessed set of class files
- Internally fully linked
  - Pointers to other classes in suite or parent(s) only
  - Chain of suites is a transitive class closure
- Internal bytecode set optimized for:
  - Space
  - Simplified garbage collection
  - Compilation
  - In-place execution
    - Requires pre-linking
    - Enables deployment in ROM
    - Improves startup time

Suite chain



Space optimizations

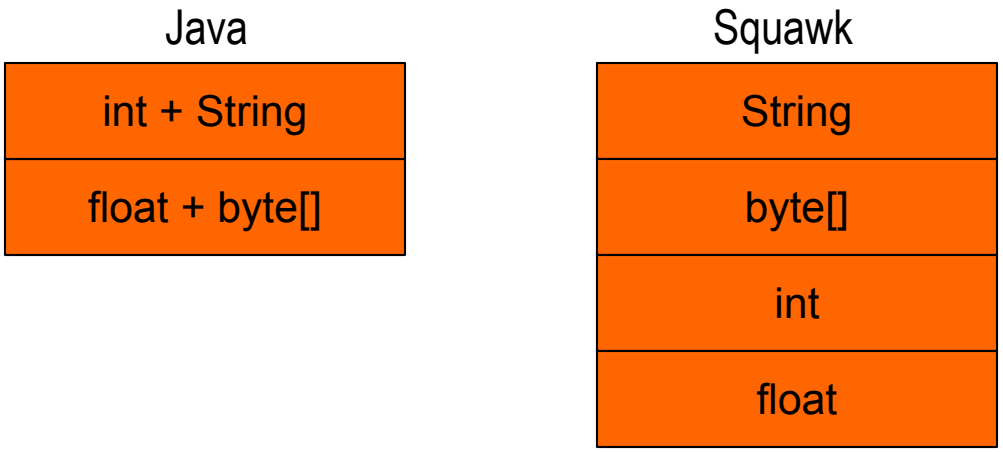
- 2 byte branch
- More 1 byte `load`/`store`/`const` instructions
- 2 byte field access
  - Operand is object offset, not constant pool index
- 2 byte invoke
  - Operand is vtable index, not constant pool index
- Escape mechanism for:
  - Float and double instructions
  - Large operands (i.e. widened operands)

Uncompressed JAR vs Suite File Size Comparison

Application	JAR	Suite	Suite/JAR
CLDC	458,291	149,542	0.33
cubes	38,904	16,687	0.42
hanoi	1,805	835	0.46
delta blue	30,623	8,144	0.27
mpeg	100,917	54,888	0.54
manyballs	12,017	6,100	0.51
pong	17,993	7,567	0.42
spaceinvaders	50,854	25,953	0.51
tilepuzzle	18.516	7,438	0.40
wormgame	23,985	9,131	0.38
Total	753,905	286,285	0.38

Simplified garbage collection

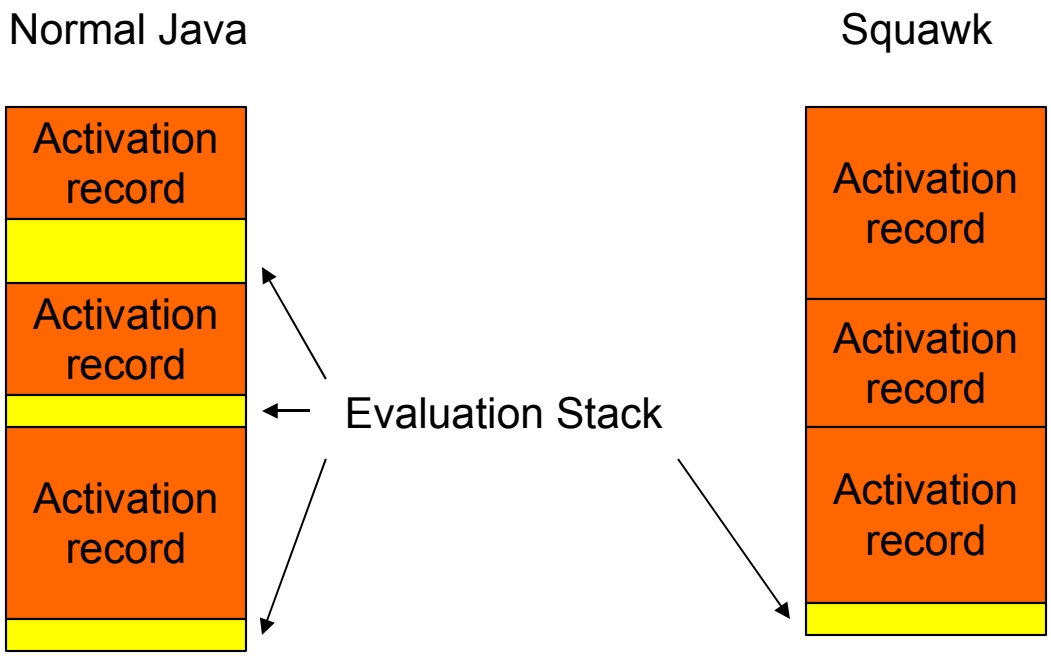
- One pointer map per method
  - > Obviates need for stack maps and analysis during collection
  - > Requires extra slots
    - > Can be mitigated by slot re-allocation based on liveness analysis (instead of lexical scoping)



Activation records

Simplified garbage collection

- Empty operand stack at GC points
  - > Same pros and cons

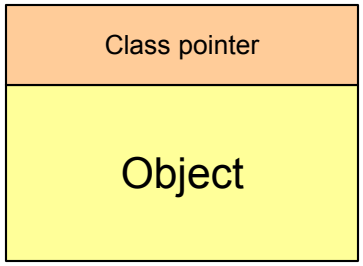


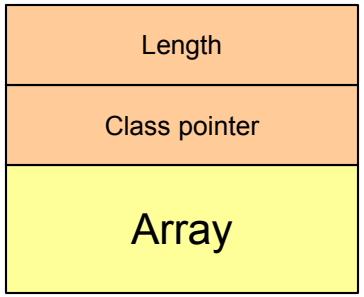
Object memory serialization

- A graph of objects can be serialized to a connection (e.g. file, socket, ...)
- Implemented as an extension to the garbage collector
- Pointers in serialized suites are normalized
- Can swap endianess on load and/or save
- Used by exporter and…

Object layout

- 1 word objects headers, 2 words for arrays





- Interposed ObjectAssociation for hashcode/monitor
  - > Created on demand
- Associative monitor used instead for ROM objects
  - > Rarely needed
  - > Hashcode is address

Method objects

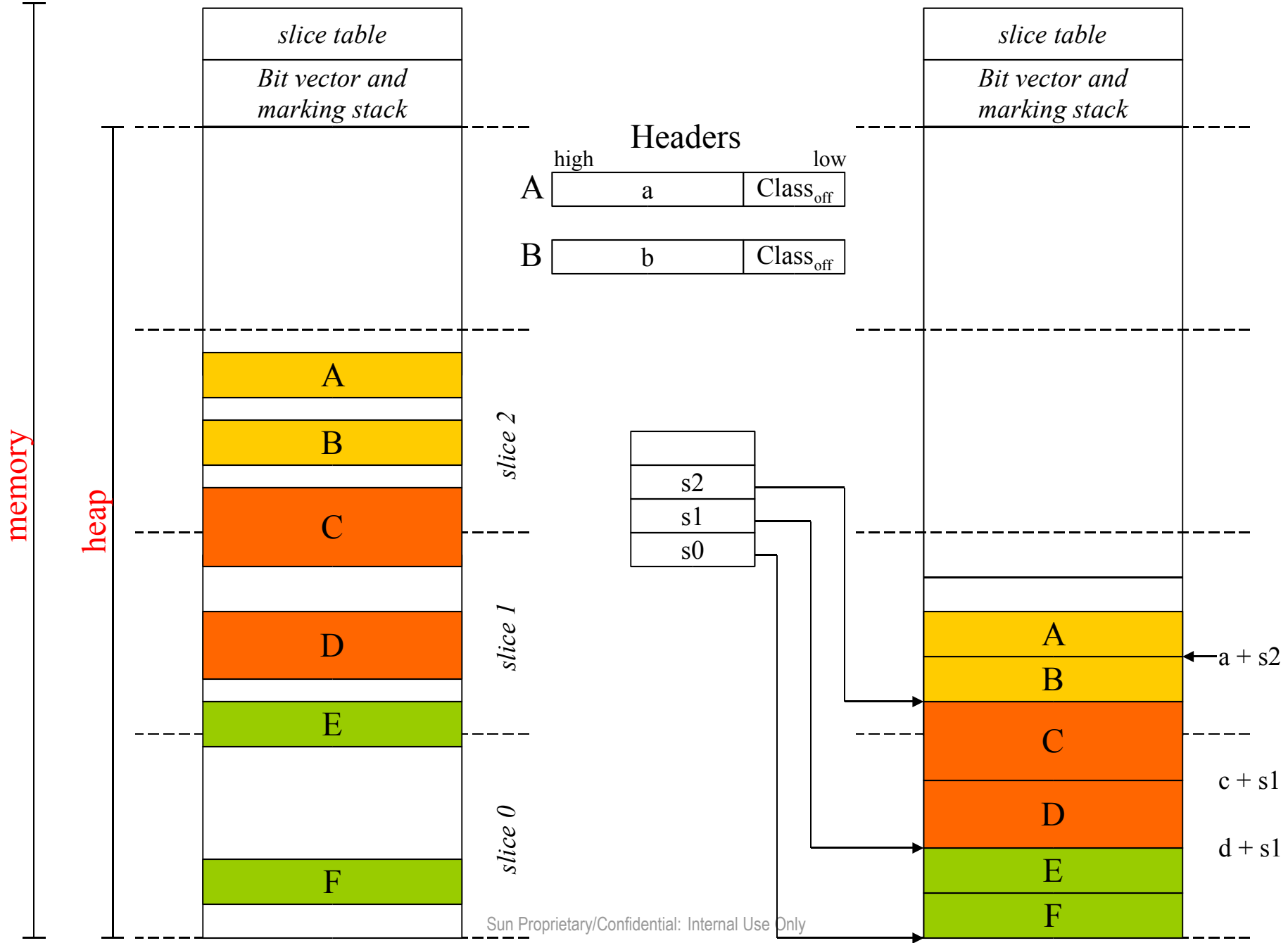
- Specialized byte array with variable length header
- Header encodes:
  - Defining class pointer
  - Pointer map
  - Type map (for non-object and non-int slots)
    - Used by compiler
  - Exception handler table
  - Slots required for parameters, locals, operand stack
- Tighter encoding for common case:
  - No handlers, parameters<32, locals<32, stack<32
  - 4 words

Space optimized symbols and metadata

- Symbols for a class encoded in a byte array
  - Method signatures
  - Field signatures
  - Access flags
- Method body metadata stored separately
  - Line number tables
  - Local variable tables
- Can strip:
  - Everything
  - Private and package private members (library)
  - Private (extendable library)

Garbage collector

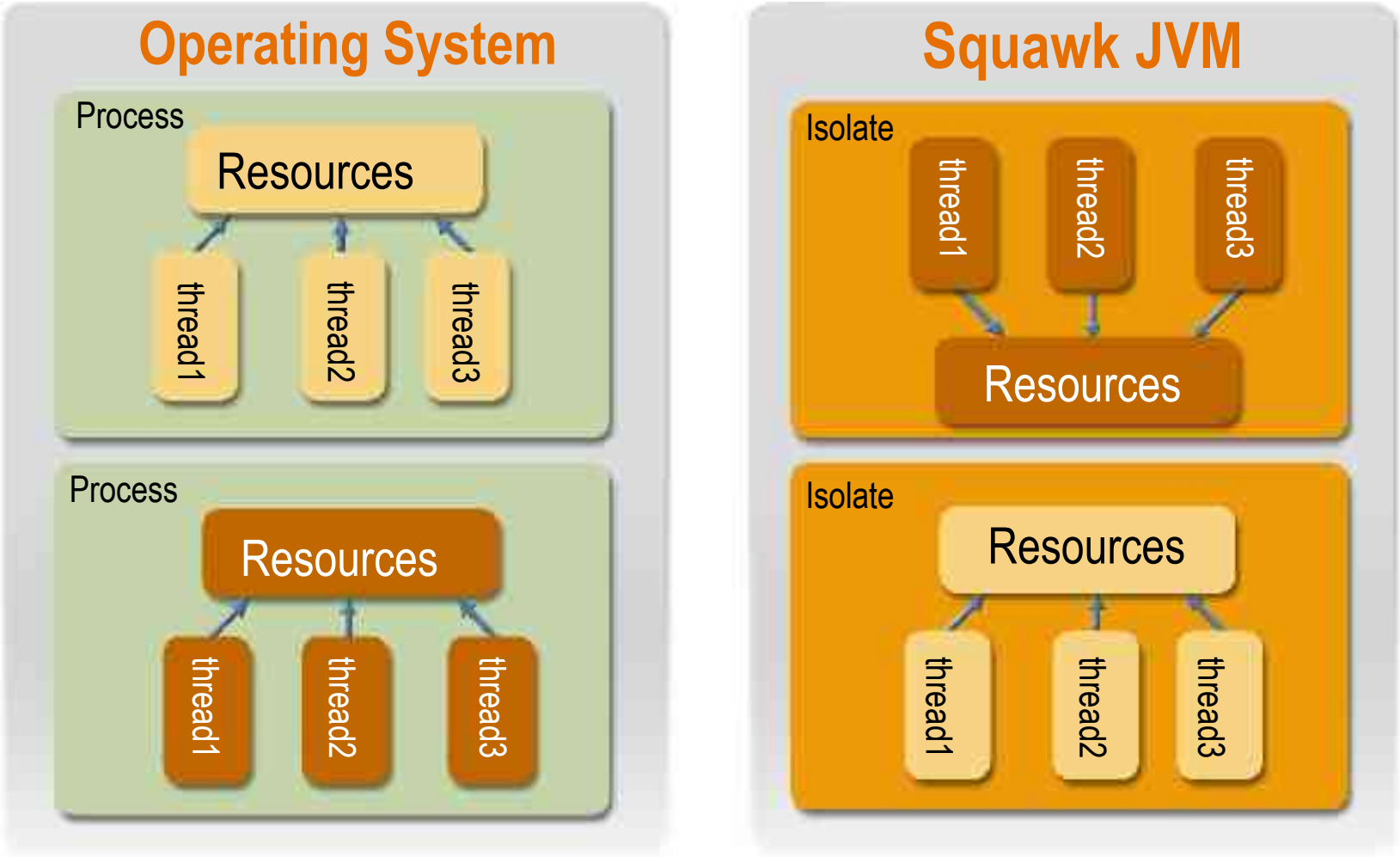
- M&C collector is based on lisp2 algorithm in Lins & Jones
- Generational - sliding window for young gen
- Slices (a la Monty) obviates need for extra header word for forwarding pointer
- Bit vector for entire heap (3% overhead):
  - > Mark bits for collection space
  - > Write-barrier bits for old generation
- Stop-the-world!
- Can copy an object graph into a contiguous block - object serialization



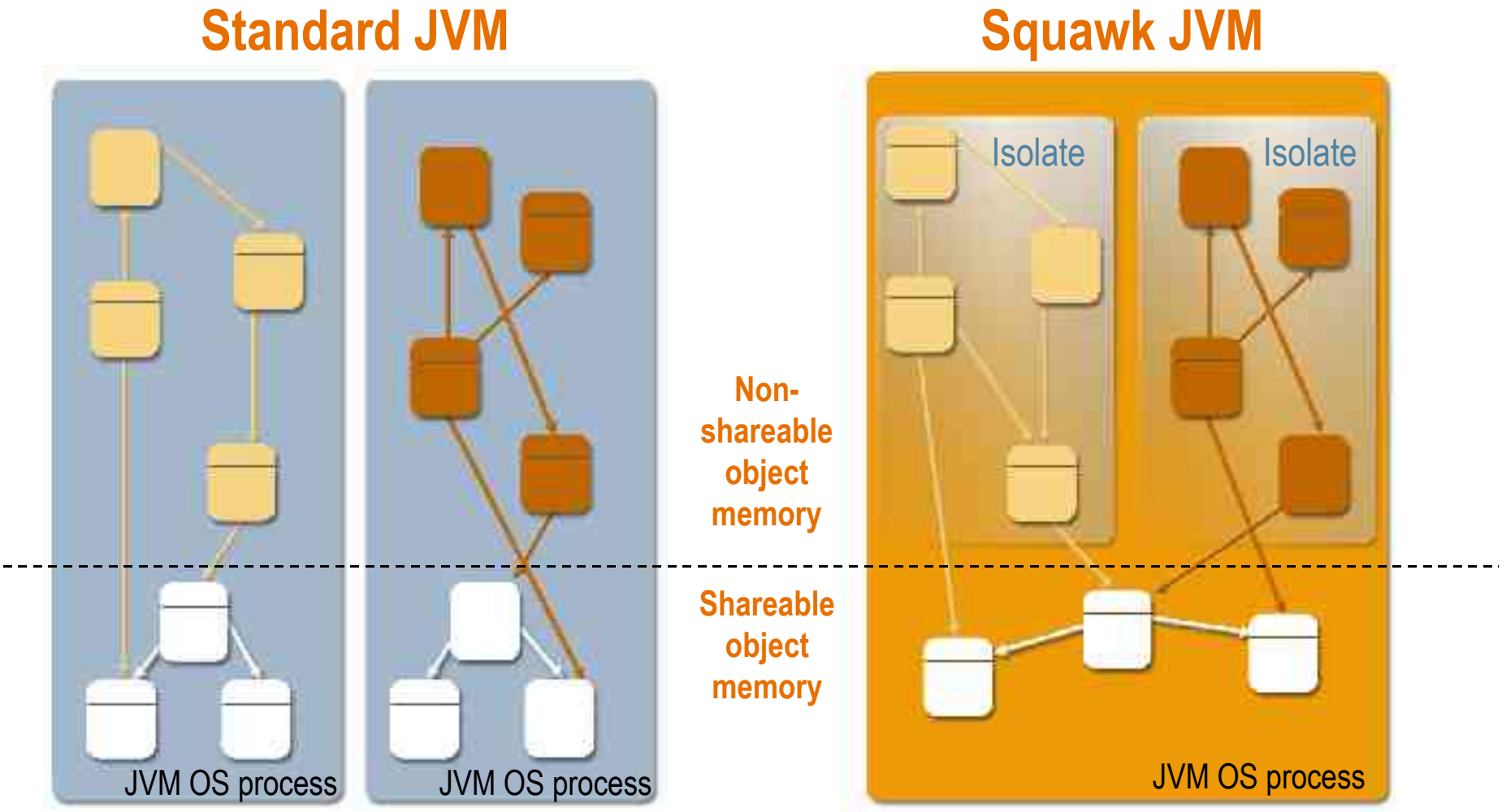
Isolates

- Abstraction for an application in a JVM
- Multiple isolates supported in single Squawk VM
- Objects are:
  - Logically isolated for each isolate
  - Physically interleaved on heap
- Non-shared class state:
  - Static fields
  - Class initialization state
  - Class monitors
- Immutable state is shared:
  - Methods, string constants, classes (apart from above)

JVM Isolates and OS Processes Analogy



Shared memory for multiple isolates



Squawk Isolates

- Similar to JSR 121
  - No support for inter-isolate communication
- Can be paused and resumed
- Can be migrated
  - Uses same object serialization mechanism as the Exporter
  - Constraints on external state
    - Must be none, OR
    - Must be homogenous at both ends, OR (Sun SPOT)
    - Must be serializable (desktop Squawk)

Thread scheduling

- Green threads
- Supports blocking in native methods
  - Thread blocked on event queue polled by scheduler
  - Events correspond to interrupt
- Rescheduling done at backward branches
  - `bbtarget` inserted by translator
- System code is non-preemptible
  - Simplifies VM greatly
  - Assumes most execution is in application code
- Requires explicit `Thread.yield()` in potentially long running system code
  - E.g. System.arraycopy()

Compiler Interface

- Stack based API with C-like semantics
- Two categories of instructions:
  - General purpose
    - `add(), sub(), eq(), push(), call()`
  - Interpreter generator specific
    - alloca(), getIP(), getFP()
- Intended to be used for:
  - AOT compiling of boot suite
  - JIT
  - Interpreter generator

Prototype Compiler Implementation

- One-pass compiler
  - > Uses shadow-stack representation
- Simple register allocator
- Implemented for x86 and ARM
- TBD: integration with rest of system
- Collaboration with U Sydney to develop optimizing implementation
  - > Probably too heavyweight for JIT but suitable for AOT
- Experimental approach: results pending!

Porting to the ARM

- Squawk ported to the ARM simulator in a week by someone external to VM team
- Within 3 months, 2 contractors had it running on the Atmel ARM board
  - > Including the ARM Platform IO support
- Complete system running on the Sun SPOT within 6 months
  - > Including sensor libraries, demo apps and build notes

Sun SPOT: Hardware

- Hardware
  - 32-bit ARM core
  - Chipcon CC2420 based wireless platform
  - 256K RAM, 2Mb FLASH
  - SPI based peripherals
  - Simple sensor board
    - 3D accelerometer
    - Light sensor
    - 3 color LEDs
    - 9 I/O pins
    - Temperature

Sun SPOT: Software

- Minimal Squawk VM configuration
  - 350K static footprint:
    - 80K - Native + PlatformIO
    - 270K - Core + CLDC 1.0 library (bootstrap suite)
- Device drivers written in Java
- Libraries for
  - Driving hardware: radio, sensor boards, ...
  - Basic 802.15.4 network functionality

Sun SPOT + Squawk: Why?

- Improve accessibility of sensor devices
- Motes + nesC was considered hard to program
- Java libraries alleviate a lot of infrastructure programming
- Can use standard Java tools including a debugger!

Lessons & Observations

- Problem: want to write highly factored VM code AND want great performance
- Solution: Need optimizing AOT compiler!

- Annotations useful for expressing VM specific constraints:
  - > method must be inlined (can't invokevirtual)
  - > method must never not do allocation
  - > public class must only be accessed by VM code

Lessons & Observations

- Writing GC in Java was a double-edged sword

  + No need to duplicate definitions in C

  - Hidden constraints: e.g. can't invoke virtual methods on object with corrupt header

  - Very slow without AOT!

- Don't couple VM code with standard core classes and namespaces
  - > Complicates bootstrapping
  - > Requires -Xbootclasspath and (even worse) -Dsun.boot.library.path (works differently in JDK 5.0!)

Future Work

- Optimize Sun SPOT deployment:
  - Improve interrupt handling latency
  - Integrate optimizations in the translator
- Complete the compiler and to validate (or otherwise) the compiler interface approach
- Investigate resource management based on isolates
- Port to a cell phone?
- Open source