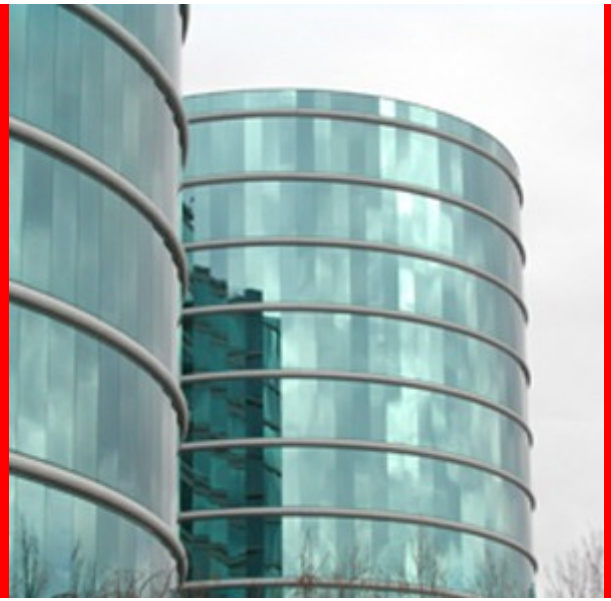


ORACLE®

Symmetric multilanguage VM architecture: Running Java and JavaScript in Shared Environment on a Mobile Phone

Oleg Pliss
Pavel Petroshenko

Agenda

- Introduction to Monty JVM
- Motivation for JavaScript support
- Symmetric multilanguage VM architecture
- Shared runtime components
- Shared compiler components
- Known problems
- Q & A

CLDC HI VM Overview

- Connected Limited Device Configuration JVM
- HotSpot Implementation
  - Profiler-driven dynamic compilation
  - Optimistic speculative optimizations
  - Dynamic deoptimization (when necessary)
- Targeted to small mobile devices
  - Slow processor and memory
  - Constrained memory (500K-16M RAM)
    - In interpreted mode can work in 100K
  - Limited OS capabilities
    - Single process
    - Single native thread
    - May or may not support page protection and memory-mapped files

Target processors

- ARM 9, 11 with optional coprocessors and instruction set extensions
  - Thumb, Thumb 2
  - JazelleRCT/Thumb2EE
  - JazelleDBX (HW bytecode interpreter)
  - ARM VFP (Vector Floating Point coprocessor)
- Intel x386+
  - For debugging and cross-compilation
- SuperHitachi SH3, SH4
- SPARC
  - For cross-compilation only

Target operating systems

- Linux
- Symbian
- Nucleus
- WinCE, Win Mobile
- Win32
- Brew

VM features

- Generational mark & compact garbage collector
  - Distinguished areas for compiled code, temporary compiler data and large immutable objects
- Single-pass dynamic adaptive compiler
- Multiple virtual threads over single native thread
- Multitasking within single OS process
  - With task priorities, memory quotas and shared libraries
- System class preloading (*ROMization*)
  - To speed up VM start-up and to reduce static footprint
- Binary conversion of libraries and applications
  - To speed up application start-up

VM Build Stages

- Generate asm interpreter source code
  - On the host, build and run special version of VM
  - Not necessary if the interpreter is implemented in C/C++
- ROMize system classes
  - Run special version of VM to load, verify, optimize and initialize «system classes»
    - Non-trivial class initializers are run only if requested by configuration file
  - Compile selected methods ahead-of-time
    - AOT-compiled code is slower than the dynamically compiled
  - Save the snapshot as C++ file (mostly arrays of constants)
- Compile and link together VM sources, the interpreter and the snapshot

JavaScript Support: Motivation

- Emerging Web for Mobile JavaScript API standards
  - JIL, BONDI, WAC
- Calling Java from JS and vice versa
  - Access to all the phone features from Web applications through Java API's
    - Standard JSR APIs
    - Platform-specific Java API extensions
- Sharing system resources between JavaScript engine and Java VM

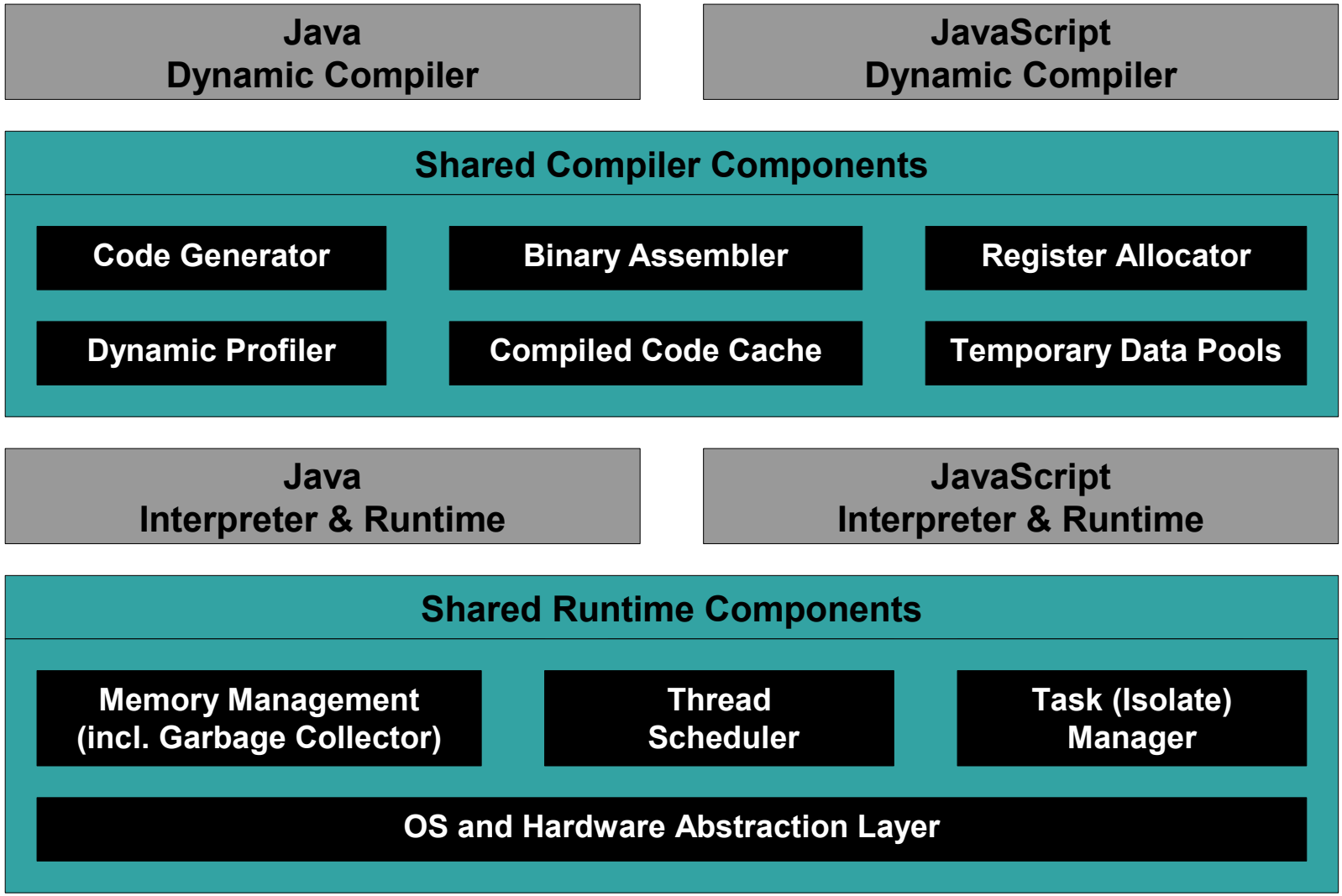
Java+JavaScript Integration Options

- NP Runtime plugin API
  - JVM + JSRs = browser plugin
- Public JS engine API
  - JVM + JSRs + JS engine = external JS engine for the browser
- Tight integration of JVM, JS engine and the browser
  - Direct calls between JS and Java
  - Avoid unnecessary argument and result conversions
  - Reuse JVM components (JIT, GC, etc)

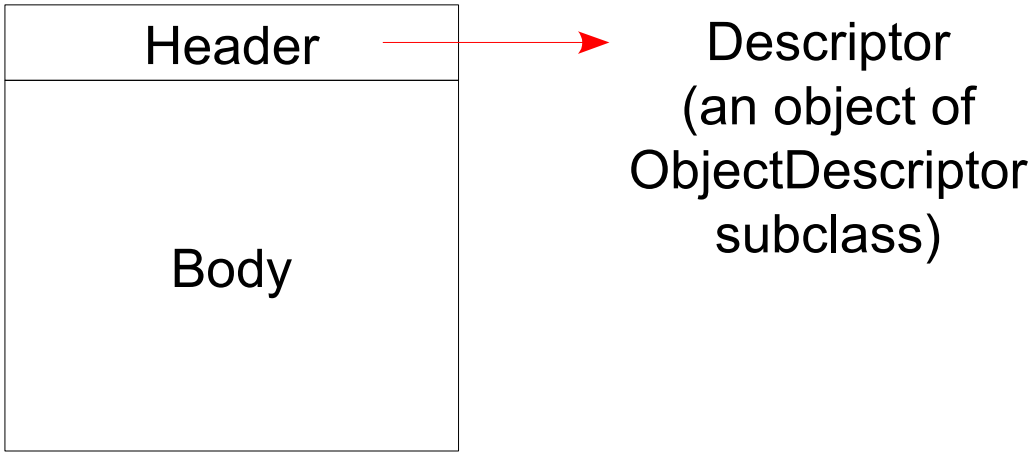
Why Native JavaScript Engine?

- Not enough system resources for JIT compiler to optimize Java implementation well enough
  - Static footprint
  - Dynamic footprint
  - Available JIT optimizations
- High-performance native JS engines exist
  - Google V8 is written in similar style and in similar moderate subset of C++
- Save footprint
  - Integration with existing VM runtime and GC
  - Share infrastructure with VM JIT

Symmetric Java+JavaScript VM architecture

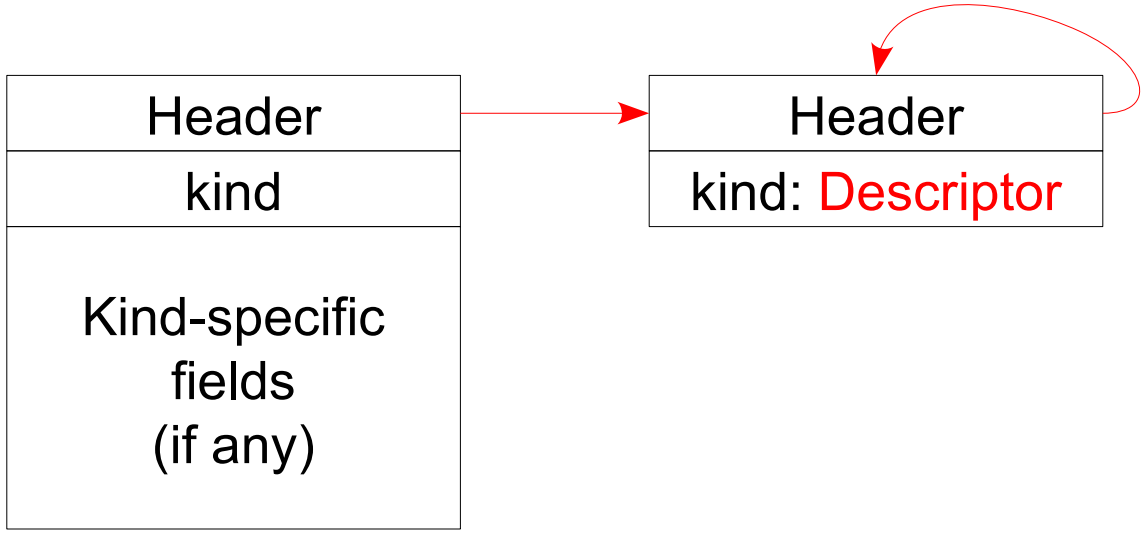


Object Layout



- Objects are aligned at word boundary
  - Can be located in heap or in ROM
- Single-word object header
  - During GC contains compressed forwarding pointer and pointer to Descriptor
  - Otherwise direct pointer to Descriptor

Object Descriptor Layout

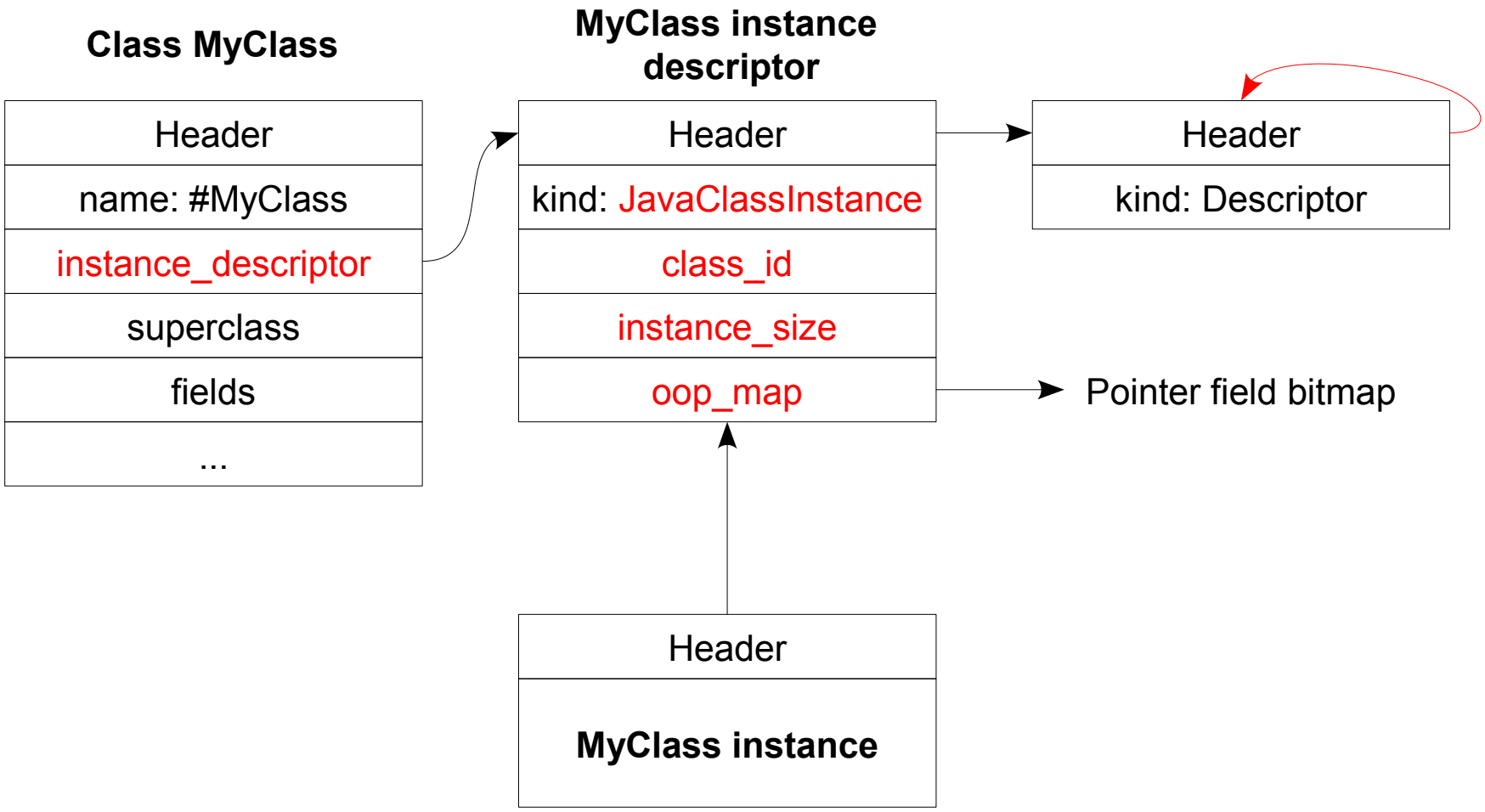


- Object Descriptor
  - Defines object kind
  - Computes object size (for given object)
  - Enumerates pointer fields (of given object)
  - Relocates pointer fields and the object itself
    - Objects may contain derived pointers (i.e. return address in stack frames)

Object Kinds

- All object kinds are known at VM build time
- Kind-specific behavior is implemented by a few switch statements
- Not every object kind has a representation in every language
  - Object descriptors, Execution stacks, Methods, arrays of unsigned integer types have no Java representation

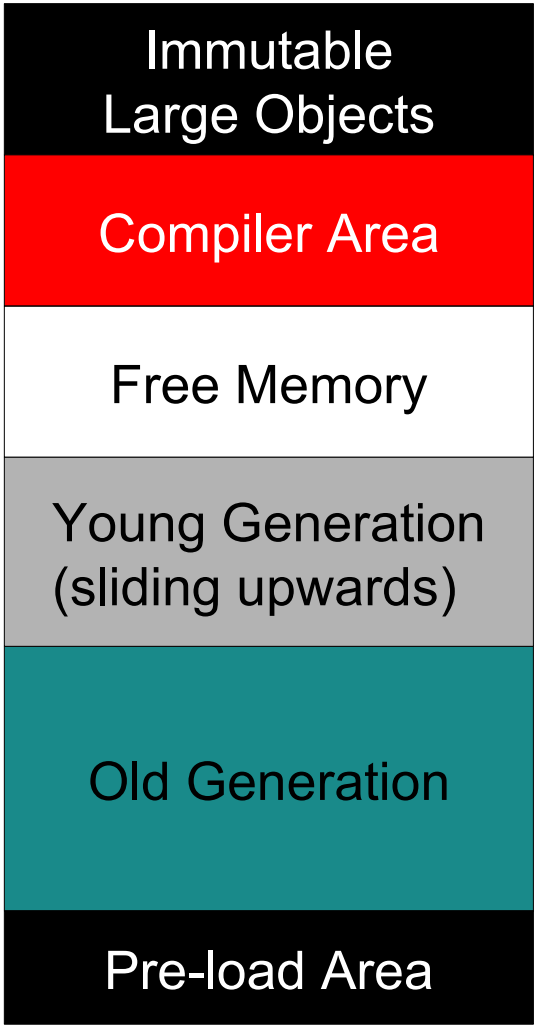
Java Classes and Object Descriptors



Runtime Access to Objects and Object Fields

- Object handles
  - Cannot use direct pointers in the scope where GC can happen
  - A few different kinds implemented as C++ «smart pointers»
- Object field access from C++
  - <type>_field (object, offset)
  - set_<type>_field (object, offset, value)
    - where <type> is one of:
      - [unsigned] {byte, short, int, long}
      - float, double
      - obj
  - clear_obj_field (object, offset)
  - A few write barriers for set_obj_field, arrays and use during GC
- Object field access from asm interpreter
  - Target-independent SourceAssembly macros API

Object Heap Layout



Dynamically loaded unloadable ROM modules

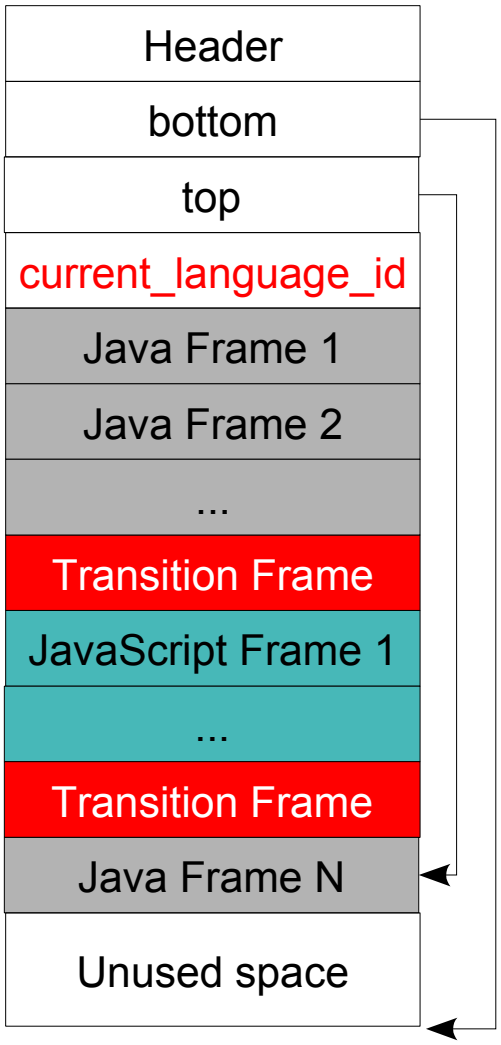
Compiled method cache,
temporary compiler data

New objects are allocated here sequentially in upward direction

Heap size and area boundaries are adjusted automatically in run-time

Statically loaded permanent ROM modules

Execution Stack Layout



- Execution stacks are heap-allocated objects
  - Re-allocatable on overflow
  - Configurable growth direction
  - Each execution stack represents a virtual thread and belongs to a task
- Stack frame layout is defined by the language
  - Compiled frame may differ from interpreted frame
- On a language transition (if not inlined)
  - Create a transition frame
  - Save current_language_id in the known local variable of the transition frame
    - Not necessary for just two languages
  - Set current_language_id to the new language
  - Set return address to a special native entry point
  - Create new language frame, do necessary argument conversions...
  - Upon return the entry point converts the results and restores the language

Runtime Access to Locals & Arguments

- Stack frame is defined by its execution stack offset
  - Stack address may change
- Local variable/arg is defined by its offset in its frame
  - Add or subtract offsets depending on stack growth direction
- Local variable access from C++
  - <type>_local (stack, offset)
  - set_<type>_local (stack, offset, value)
    - where <type> is one of:
      - [unsigned] {byte, short, int, long}
      - float, double
      - obj
  - clear_obj_local (stack, offset)
  - No barriers on access to the locals
- Local variable access from asm interpreter
  - Target-independent SourceAssembly macros API

Virtual multithreading

- Multiple virtual threads over a single native thread
  - Preemptive at Java level, cooperative at native level
  - Timer interrupt sets a global flag
  - Native code checks the flag and yields to the scheduler
  - Page protection and tight loop code patching can be used
- Built-in advanced virtual thread scheduler
  - Provides independency of OS capabilities
- Slave mode is supported for time-slicing OS
- GC-safe points are method and runtime calls
  - Including yields to the scheduler
- Execution of native code is not preemptible
  - Asynchronous calls of long-running third-party native code are supported with a pool of native threads

Dynamic profiler

- Predicts code activity in the near future by the activity in the near past
- Monitors the activity of interpreted and compiled methods to maximize performance with compiled code of limited size
  - Frequently used interpreted code is compiled
  - Rarely used compiled code is evicted from the cache
- Combined stack sampling and code instrumentation
  - Code instrumentation is precise but expensive
  - Sampling is fast but can miss small methods
  - Combine sampling for loops with instrumentation for calls
- Integrates events in common time scale
- Dynamic adjustment of compiled code cache size

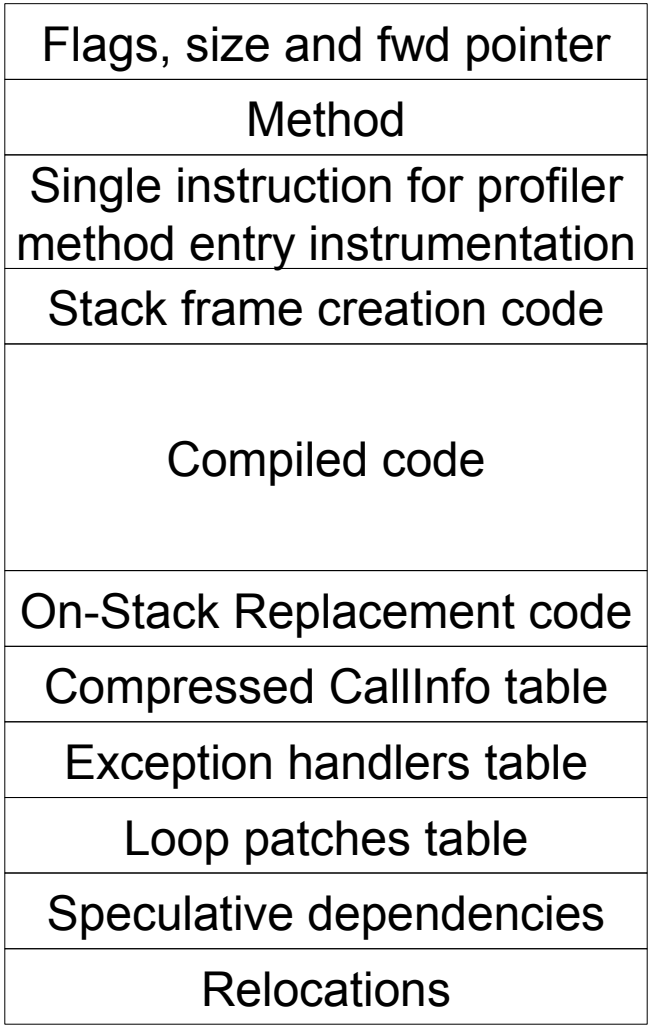
Schedulable Concurrent Compilation

- Dynamic compiler runs as a co-routine to any virtual thread
- Compilation can be suspended or aborted on request
- Suspended compilation can be resumed to run for no longer than given time interval
- Compilation can be temporarily disabled during program phase transitions
- Pause manager prevents clasterization of GC and compilation pauses

Compiler Area

- Predictability of GC pauses
  - No GC should happen when application does not allocate memory
  - Amortized cost of system object allocations over user object allocations should not be excessive
- GC performance
  - Temporary compiler objects are allocated during compilation and disposed altogether when the compilation terminates
  - Lifespan of compiled code is controlled by dynamic profiler
  - Prediction of High Infant Mortality fails for compiler objects

Compiled Method Layout



Single-pass dynamic compiler

- No intermediate representation
  - Memory is too scarce and too slow
- No explicit basic blocks
  - Quick preliminary scan to detect entry points and tight loops
- Continuation-style abstract interpretation of basic blocks
  - Compilation context mimics run-time stack frame
  - Compile-time values mimic run-time values
  - When a computation cannot be performed at compile-time, code is emitted to perform it at run-time
  - Compilation context is cloned when control flow forks
  - Compilation contexts are merged when control flow merges
- Multiple-pass compilation can be implemented
  - Any convenient kind of IR can be build in temporary compiler area
  - Currently there is no shared components for IR construction and transformations

Compile-time Value

- Belongs to Compilation Context
- Has compile-time type
- Can be a constant or an expression
- Can have a location in the stack frame
- Can be assigned to a register or a register pair

Example: Compiling bytecode iadd

```
void iadd (void) {
  PoppedValue b;
  PoppedValue a;
  Value result (T_INT);

  if (a.is_immediate() && b.is_immediate()) {
    const int x = a.int_value()+b.int_value();
    result.set_int_immediate(x);
  } else {
    codegen->int_binary_add(a, b, result);
  }
  push (result);
}
```

Java Compiler Optimizations

- Constant folding
- Local constant and copy propagation
  - In extended basic blocks (single entry, multiple exits)
- Null-pointer & class initialization check elimination
- Local common subexpression elimination
  - Dictionary of registers annotated with bytecode ranges
- Multi-level inlining of small methods
- Branch and loop optimizations
  - Bytecode pattern matching for common cases
- Speculative unguarded devirtualization
  - Analise only initialized/instantiated classes in the hierarchy
  - Compile a method, record the made assumptions
  - If new class initialization/instantiation invalidates the assumptions, deoptimize the dependent compiled code
    - Any compiled stack frame can be deoptimized

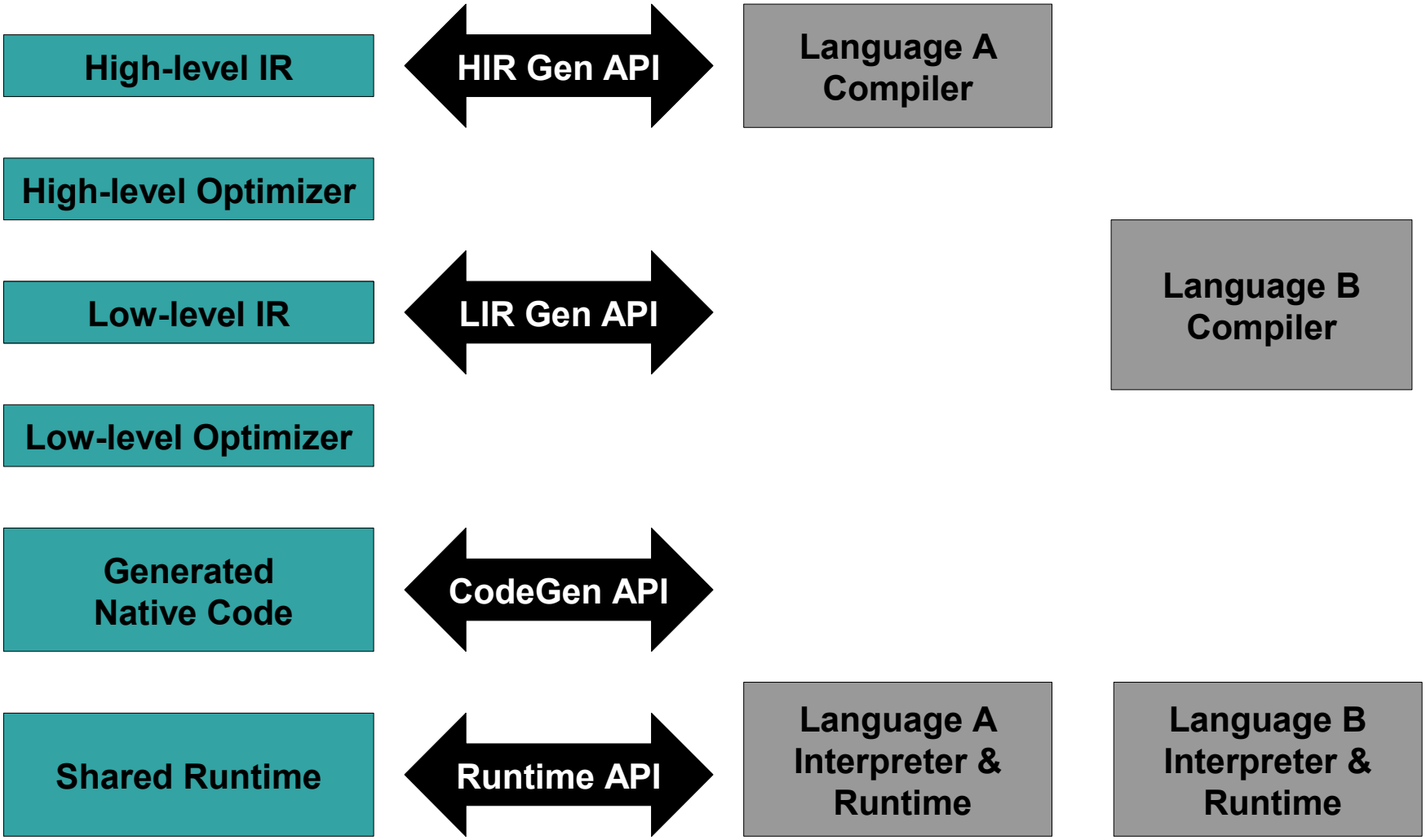
Register Allocation

- Registers can be allocated, deallocated, pinned and unpinned
  - pin (Register reg) increments pin_count of the reg, unpin decrements it
  - Pinned register cannot be re-allocated
- Modified Round-Robin strategy
  - First, an empty register
  - Next, the least recently cached re-computable value (literal or a subexpression)
  - Otherwise the least recently allocated unpinned register

Code Generator

- Target-independent set of virtual functions
  - Some of them are abstract and must be implemented for any target CPU
  - Others have default target-independent implementation that can be overridden
  - Oriented to 2-3-address load-store architecture
    - <binary_op> (a, b, result, bool must_set_flags = false);
    - <unary_op> (a, result, bool must_set_flags = false);
- Compile-time values and registers passed as arguments
- Field access for arrays and objects
  - <type>_field (Value& object, Value& index, int offset, Value& result);
  - set_<type>_field (Value& object, Value& index, int offset, Value& value);
  - clear_obj_field (Value& object, Value& index, int offset);
    - field_address = object + index*sizeof(<type>) + offset

Symmetric Multilanguage VM Architecture



Known Problems

- Native interpreter and compiler must be trusted
  - Write compiler in one of languages supported by VM (Java?)
  - And what about the generated asm interpreter?
- Dynamically generated native code must be trusted
  - Validate the code?
- Language-specific code generator extension
  - Code generator API is supposed to be language- and target-independent
  - What if it is not sufficient for the new language?
  - This extension has to be implemented for every target CPU
- IR generation interfaces are not standardized
  - May not fit other VMs

phoneME Feature Software

- Open source JavaME platform for “feature phone” devices
- https://phoneme.dev.java.net/content/phoneme_platforms.html

Q&A