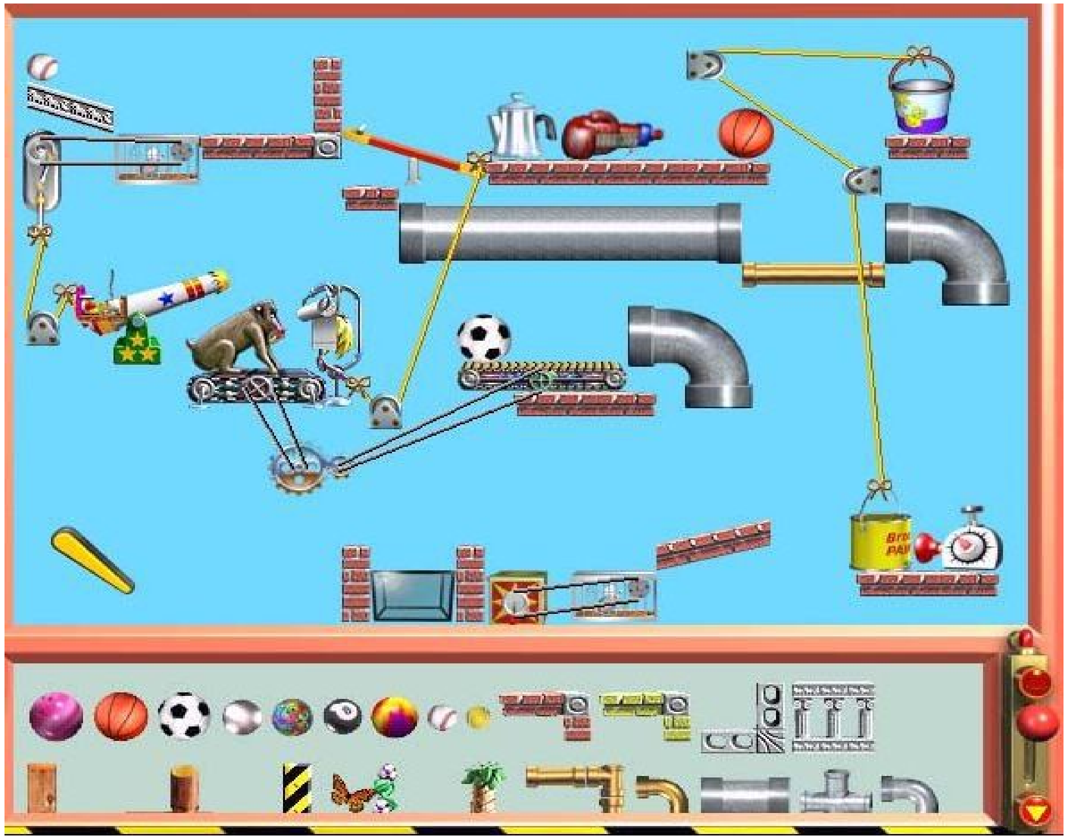
The History of Monty VM Project

Design choices made,Technologies developed, Lessons learned



Oleg Pliss
Oleg.Pliss@Oracle.com

July 2016

Agenda

- Introduction to Monty VM
- History of The Project
- Interpeter
- Multithreading
- Native Interfaces
- Dynamic Adaptive Compiler
- Garbage Collection
- Initial State Memorization
- Multitenancy
- Built-in and Distributed Tooling
- Shared Multilanguage Runtime
- Q&A

About the speaker

- Wrote first computer program in 1978
    - On extended Algol-60 for original M-222 Soviet-made computer
- In 1985 implemented Smalltalk-80 interpreter for a clone of IBM System 370
    - Text stream in, text stream out
    - Punched cards would work too
- In 1987 designed and implemented Object Forth
    - For a terminal station, 8-bit CPU, 48K RAM, 16K shadow ROM
    - Class based, single inheritance, reflection, garbage collection, reversed Polish notation, open stack access
- In 1992 defended a Ph.D. thesis on implementation techniques for object-oriented languages
- Compilers, converters, public lectures on CS subjects
- In 2003 joined SUN Microsystems as CLDC VM engineer

What is Monty VM?

- Internal name of SUN Microsystems CLDC HI JVM
    - First release: October 2003
- CLDC profile
    - Connected Limited Device Configuration
    - Java profile for small connected mobile and embedded devices
- HI — HotSpot Implementation
    - Profiler-driven dynamic compilation
    - Optimistic speculative code specialization
    - Dynamic deoptimization (when necessary)
- Not derived from any other JVM
    - KVM
    - Squawk
    - CVM
    - HotSpot SE/OpenJDK
- Open-sourced as a part of phoneME project @ java.net
    - Not updated since October 2009

Small Mobile & Embedded Devices

- Slow processor and memory
- Variety of processors
    - ARMv3, ARMv4, ARMv7-M with coprocessors
    - x86 (Intel Spark)
    - Renesas SuperH-3, 4
    - AndesCore
    - MIPS32
- Large variety of peripherals
- Constrained memory (16K-16M RAM)
- May not have a fully capable OS
    - Single process
    - Single native thread (in some cases, no threads)
    - May or may not support page protection and memory-mapped files
    - May have no OS («bare metal»)
- Large variety of RTOSes

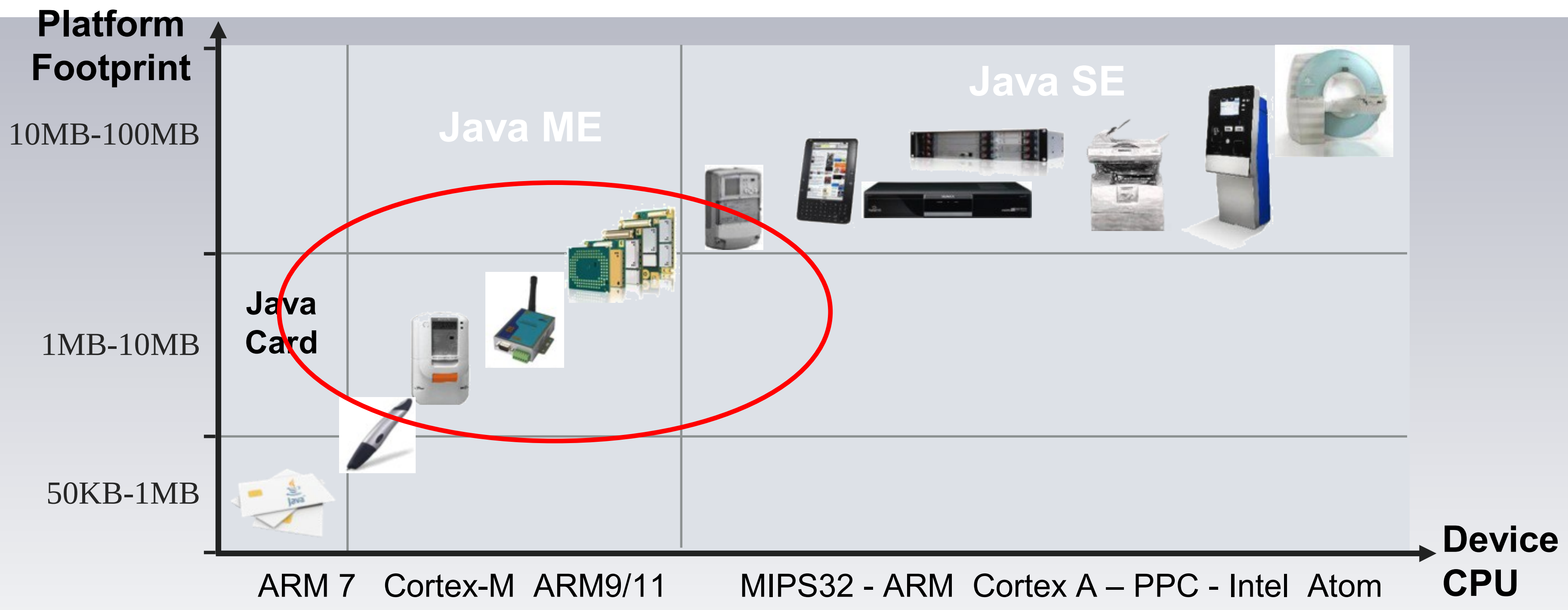
Evolution of CLDC Profile

- CLDC 1.0, 1.1, 1.1.1
  - Subsets of J2SE (JDK 1.3)
  - No user-defined class loaders
  - No reflection (except for Class.forName)
  - No serialization
  - No JNI and native code in applications
  - No user-defined finalizers
  - Requires 32K RAM, 160K ROM for VM and class library
- CLDC 8 (JSR 360)
  - Subset of Java SE 8, released April 2014
  - Supports modern language features (Generics, Annotations...)
  - Retains all limitations of the older CLDC profiles
  - No invokedynamic
  - No annotations with RUNTIME retention policy
  - Requires 128K RAM, 512K ROM for VM and class library

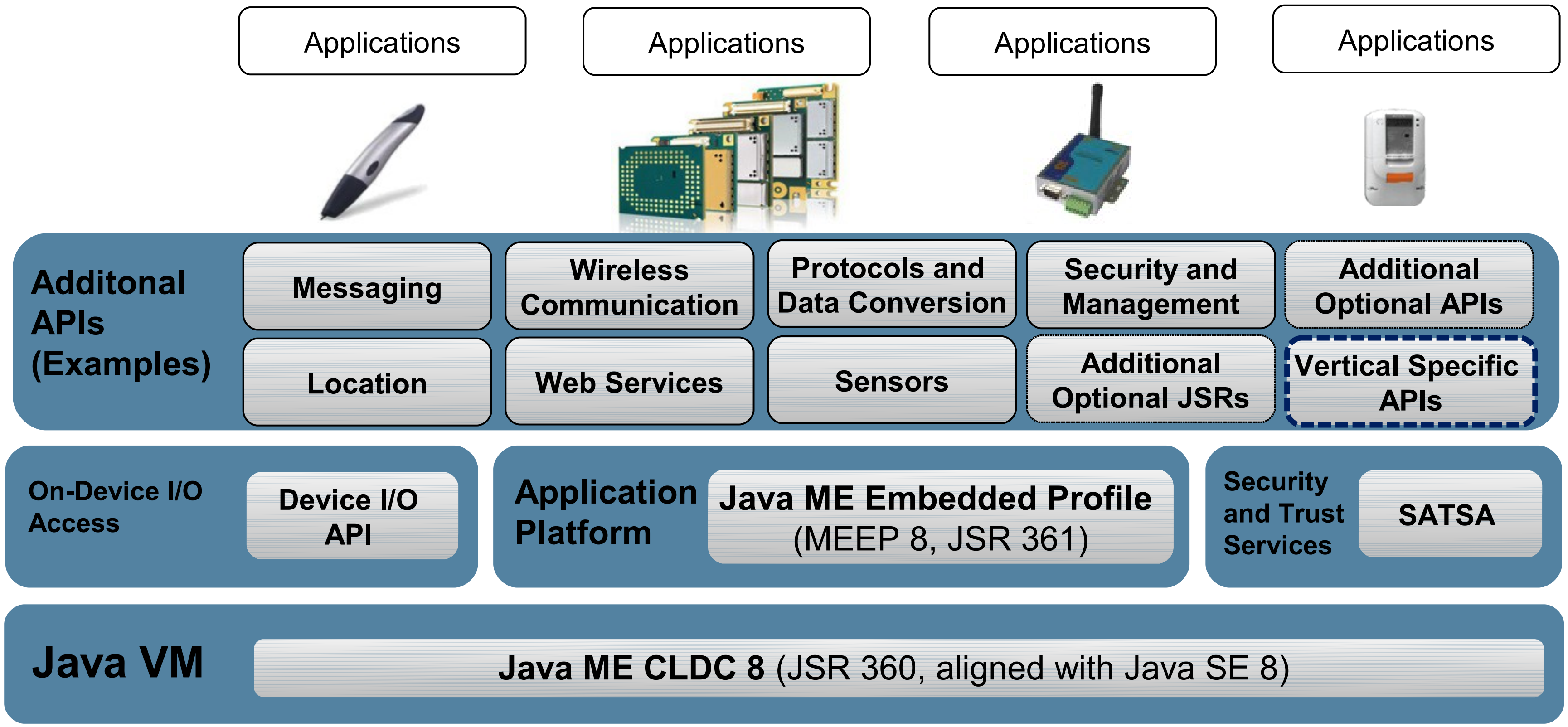
The History of the Project

- Lab Years (2001-2002)
  - Initiated by Lars Bak and Kasper Lund
  - Small language-agnostic VM featuring a Hot-Spot technology developed by Animorphic/LongView Tech.
  - Generational GC
  - Single-pass dynamic adaptive compiler by Bernd Mathiske
- Mobile Years (2003-2010)
  - R&D team of 8 engineers created, including Bernd Mathiske & Frank Yellin
  - Target devices: Java-enabled «feature» phones and PDAs
  - Multitenancy, Initial State Memorization, major improvements in dynamic adaptive compiler
  - Replaced KVM as a reference implementation of CLDC 1.1+
- Small Embedded Years (2010-Present)
  - Feature phone market collapsed
  - The technology re-focused on small embedded MCUs
  - Code generation for ARM Cortex M3,4
  - Build-time modularity, CLDC 8, access to peripherals, closed-world optimizations, distributed tooling

Focus of Java ME Embedded



Java ME 8 Platform Overview



Distributable and Executable Formats

- Distributable Format
  - File format, intended mostly for software distribution
  - Standardized
  - Machine-independent
  - Extendable
  - Compact
  - Java: class files, usually packed
- Executable Format
  - Internal in-memory format, intended mostly for code execution
  - Generated from the distributable format
    - Can supplement or replace it
    - Statically or dynamically
    - On the same or other device
  - Monty VM: extended set of standard Java bytecodes
    - Performance-critical code compiled to native instructions
  - Other option considered: subroutine threaded code
- Difficulties of conversion
  - Lazy linkage and error reporting
  - Tooling APIs

Interpreter

- Java Interpreters in Monty VM
- Techniques to Make Interpreters Faster
- Implementation of Execution Stacks
- Implementation of Java Threads

Interpreters in Monty VM

- C Interpreter
    - Pure C, architecture-neutral implementation
    - Executes bytecodes on the host during build process
- Hand-crafted assembly interpreters for all target ISAs
    - Statically generated by a special version of VM at build time
    - The generator is built from the same sources
        - correctly represents runtime structures in the assembly
        - correctly binds assembly code to runtime global variables
    - The syntax of the generated assembly may vary depending on the target (cross-)assembler
    - For any ISA, about 2x faster and a bit smaller than C interpreter
    - VM could bootstrap with assembly interpreter only but it would take one extra step to run interpreter-less generator of the assembly interpreter for the host ISA

Why Interpreters are Slow

- For most programming languages, interpretation:
  - 2.5-50 times slower than similar native code generated by optimizing compiler
  - Less predictable
  - Consumes more power for the same work
- Decoding of virtual instructions
  - Computed indirect branch to a large set of labels cannot be predicted by the CPU
    - Use direct or subroutine threaded code for lower overhead
- Access to virtual registers
  - Map globals to physical registers using non-portable pragmas
  - Rewrite the interpreter in assembly
- Superfluous checks on every instruction
  - Class initialization, null check...
  - Dynamic rewriting of interpreted code by «quick» versions of instructions solves the problem only partially
    - Code space is limited (not for threaded code)
    - Analysis and replacement can only be applied to a single instruction or a short sequence of instructions

Why Interpreters are Slow (2)

- Operands of virtual instructions are located on stack
  - Cache 1-2 upper stack elements in physical registers

Number of cached top elements	iadd	iload_0

0	`pop(tmp1);`
	`pop(tmp2);`
	`tmp2 += tmp1;`
	`push(tmp2);`
		`get_local(tmp1, 0);`
		`push(tmp1);`

1	`pop(tmp1);`
	`top += tmp1;`
	
	Saved 2 memory accesses
		`push(top);`
		`get_local(top, 0);`
		
		No instructions saved

2	`top1 += top2;`
	`pop(top2);`
	
	No further savings
		`push(top2);`
		`mov(top2, top1);`
		`get_local(top1, 0);`
		
		1 extra register move

  - Optimal number of cached elements varies for virtual instructions
  - Number of extra register moves per virtual instruction grows with the number of top stack elements cached
  - Operand stack is local in a stack frame — more registers to save and restore on every method invocation

Smart operand cache

- Optimal number a cached operands varies for different virtual instructions
- The number of extra register moves can be reduced
- Generate multiple interpreter loops
  - One interpreter loop for each number of cached operands
  - Implementations of virtual instructions can be shared between the loops
    - The sharing can break alignment of labels
  - The implementation ends with a branch to the same or other loop

```
iadd0:                  // Initial state: no operands cached
      pop(top1);
iadd1:                  // Initial state: 1 operand cached
      pop(top2);
iadd2:                  // Initial state: 2 operands cached
      top1 += top2;
      goto Loop1; // Final state: 1 operand cached (top1)
```

- Larger interpreter code
  - Higher chances of code cache misses

Idioms and super-instructions

- *Idiom* — frequently used sequence of virtual instructions
  - Get a field of *this* object: `aload_0; getfield(cp_index);`
  - Here *getfield* can be «quickened»
    - Presence of the field in the class is verified
    - Field offset and type can be permanently bound
    - Short offsets are most frequently used

      `quick_get_ifield_4, quick_get_ifield_8,`
      `quick_get_afield_4, quick_get_afield_8...`
    - Quick *getfield* still needs to check the argument for *null* and to throw the exception
  - The quickened idiom: `aload_0; quick_get_ifield_4`
- The sequence may provide a context for better implementation
  - *this* object cannot be *null*
  - So the null check in quick *getfield* is superfluous
    - The invocation of the method would throw an exception
- Introduce a super-instruction for the idiom
  - `aload_0_quick_get_ifield_4`
  - Code space is limited (except for threaded code)

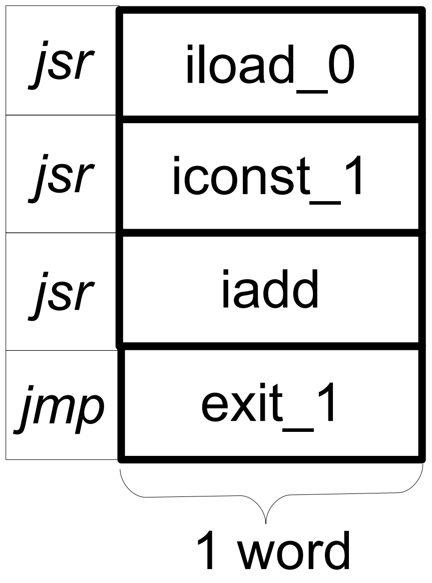
Register Allocation in C Interpreter

- C Interpreter loop

```
for(;;) {
  switch (*ip++) {    // Instruction decoder
    ...
    case iadd:
    ...
  }
}
```

- Complexity of implementation Java bytecodes varies
  - Some bytecodes are very simple, some are much more complex
  - Complex bytecodes need more locals
  - Complex bytecodes are relatively rare executed
    - But C compiler is not aware of this
  - Higher register pressure in the complex branches may cause spilling in the interpreter loop and unoptimal register allocation in simple branches
- Move complex bytecodes to a separate instruction decoder
  - Inspect the code generated for C interpreter loop — some compilers are already clever enough, and some may need profiler feedback

Subroutine threaded code



jsr — native subroutine call instruction
jmp — native jump instruction
(minor optimization of the tail *jsr*)

- Bell, J.K., Threaded code, Communications of the ACM vol 16, nr 6 (Jun) 1973, pp.370-372

- No interpreter loop is necessary
- Virtually inlimited extendable set of virtual codes / subroutines
- Many of subroutine calls can be inlined
- Generation of threaded code is a very primitive form of compilation

Execution Stacks

- Allocated within object heap
    - Execution stack is a relocatable non-Java object
    - Heap unifies memory management for all kinds of objects
- Continuous
    - To simplify most operations
- Elastic
    - Grow on overflow
    - Shrink on heap saturation
- Other options:
    - Continuous stacks outside the heap
        - Preallocate address space
        - Commit/uncommit memory pages as necessary
        - mmap-capable H/W and OS are required
    - Chunky stacks
        - Linked fixed-size chunks allocated outside the heap
        - Grow as necessary, shrink with some slack
        - Operations have to handle chunk boundaries

Java Threads

- Threading models
- Cooperative thread switching techniques

Threads Implementation

- Java threads *behavior* is defined by the Java Language Specification
- Java threads *implementation* options
  - Preemptive native threads
  - Cooperative native threads
  - Cooperative virtual threads
  - Generalized M:N threading
    - M Java threads map to N native threads
    - N is controlled to match h/w capabilities
    - M:M - preemptive native threads
    - M:1 - cooperative native threads
    - M:0 - cooperative native threads, time-shared (a.k.a. slave) mode
- Lesson learned
  - If the runtime is not designed to support multiple options for the threads implementation, the choice quickly becomes implicitly hard-wired into various components
  - Late switching between preemptive and cooperative native implementations is hard and error-prone

Preemptive Native Threads

- Every Java thread runs on its own native thread
- OS provides:
  - Preemptive thread switching
  - Thread scheduling
  - Thread synchronization (critical sections, mutexes, semaphores...)
- OS capabilities may not match the language spec
  - Number of priorities
  - Scheduling policies
  - Resources (native threads, semaphores...) can be scarce
- OS synchronization can be expensive
- VM itself needs to synchronize threads
  - Critical sections inside VM
  - Global garbage collection
  - Thread-Local Allocation Buffers (TLABs) to minimize contention on memory allocation

Cooperative Native Threads

- “Green Threads” in Java
- VM provides
  - Threads or coroutines
  - Thread scheduler
  - Thread synchronization
- Every Java thread runs on its VM-provided native thread
  - Has its own native context (registers and stack)
  - Java threads are preemptive
  - Native threads are cooperative
- Interpreter periodically calls Thread Scheduler
- Thread Scheduler selects next thread to run
- If the next thread is not current
  - Save current native context (*setjmp*)
  - Restore the next thread context (*longjmp*)

Cooperative Native Threads (2)

- Native code must be aware of cooperative threads
    - Can contain long-running loops, but must yield to the scheduler
    - Can recursively call the interpreter
- System calls block the interpreter
    - Non-blocking I/O is strongly preferred
- Native stack of every thread
    - May overflow
        - The overflow has to be detected and handled
    - Position-dependent
        - Dynamic call chain
    - Contains native frames of unknown layout
        - Native compiler is not aware of garbage collection
        - Native frames are not designed for relocation
        - Layout of native frames may vary depending on the compiler, compiler options, the callee qualifiers
    - Relocation of native stacks must be avoided
        - Native stacks to be allocated outside of the regular heap

Virtual Threads

- Java thread has no native context
- Interpreter periodically
    - Saves virtual registers to the context of current Java thread
    - Calls Thread Scheduler
    - Restores virtual registers from the context of current Java thread
- Thread Scheduler
    - Selects next thread to run and sets it current
- No native context switching (*setjmp*/*longjmp*) is involved
- No *yield* is provided for native context
- No long-running native loops
- No recursive interpreter calls
- No native stacks of unknown layout
    - VM defines layout of Java stacks, can resize and relocate them
    - Java stacks can be allocated in the regular heap

Threads in Monty VM

- Virtual threads
    - No multicore targets
    - Target OS often does not support threads
    - Time-shared (“slave”) mode support
- A pool of native threads for asynchronous calls
    - To simulate non-blocking I/O when only blocking I/O is available
    - To offload any long-running native activity from the interpreter
- Multiple copies of VM can run on different native threads
    - Mutable data replicated, immutable data shared
- Low-level real-time code simplified
    - Real-time code and data need to be separated from ordinary Java
    - Real-time data to be allocated in pools, may refer to the heap
    - Real-time code is written in C++, not available to applications
    - Direct access to ports, memory-mapped buffers
    - If Java syntax and type consistency are preferred, bytecodes of annotated methods can be converted to C++ source
    - No threads and heap allocations in real-time code

Thread Switch Points

- The interpreter has to be in consistent state
    - Not every native state maps to consistent interpreter state
- Thread switch points must be regularly visited
    - Calls and loops have to contain thread switch points
        - Additional points may be placed inside long linear blocks, loop and method exits
    - Better to switch threads inside method prologue than inside the method call site
        - Usually number of method calls sites >= number of methods
    - Better to switch threads at loop entry than at backward branch
        - A loop may contain multiple backward branches (see *continue*)
    - External events can wake up threads at switch points only
        - The more switch points, the lower latency, the higher overhead
    - Compilation can make code more responsive and predictable
        - Compiled code can visit switch points more frequently
- Thread switch point is a GC-safe point
    - Next thread can cause a GC
    - Not every GC-safe point has to be a thread switch point

Thread Switch Techniques

- Count virtual instructions
    - Makes possible deterministic thread switching
- Poll high-resolution timer
- Timer interrupt + poll the flag
- Timer interrupt + page protection
- Patch hot loops
    - Supplementary technique for compiled code in RAM

Virtual Instruction Counting

- Thread Scheduler sets *virtual instruction quota* to Interpreter
- Virtual instruction execution consumes the quota
    - Different weights can be assigned to different instructions
    - Fortunately, no contention on decrement of the global variable in natively single-threaded code
- Once the quota becomes negative, yield to Scheduler
- To force the yield at the closest thread switch point, set the quota to a negative value
- No OS support required
- Deterministic thread switching
- Duration of runtime function calls is ignored
    - Unless the function decrements the quota
- Compiled code has to count virtual instructions
    - Series of decrements can be hoisted or sunk

Naive Virtual Instruction Counting

```
    …
    if (--VirtualInstructionsQuota < 0) goto slow_path_1;
fast_path_1:
    Code compiled for virtual instruction 1;

    if (--VirtualInstructionsQuota < 0) goto slow_path_2;
fast_path_2:
    Code compiled for virtual instruction 2;
    …
    return;

// Sink slow-path code
slow_path_1:
    Save caller-saved registers;
    call Yield;
    Restore caller-saved registers;
    goto fast_path_1;

slow_path_2:
    Save caller-saved registers;
    call Yield;
    Restore caller-saved registers;
    goto fast_path_2;
```

Hoisted Virtual Instruction Counting

```
    …
    if ((VirtualInstructionsQuota -= 2) < 0) goto slow_path;
fast_path:
    Code compiled for virtual instructions 1, 2;
    …
    return;

// Sink slow-path code
slow_path:
    // Naive virtual instruction counting
    Save caller-saved registers;
    call Yield;
    Restore caller-saved registers;
    goto fast_path;
```

- Deterministic thread switching
    - Threads are switched at exactly the same positions for any program run regardless of interpereted or compiled code
    - Check quota at method prologue, epilogue, loop body entry, loop exit
    - Slow-path code must check the quota before each compiled instruction
    - High compiled code size overhead

Polling High-Resolution Timer

- H/W and OS must provide access to a high-resolution timer
- Thread Scheduler sets a *limit* for execution time
- At thread switch points Interpreter and compiled code
    - Read the designated high-resolution timer
    - If the value exceeds the limit, yield to Scheduler
    - No globals modified
- To force the yield at the closest thread switch point, set the limit to zero
- Thread switching is not deterministic

Polling High-Resolution Timer in Compiled Code

```
    …
    tmp_reg = TickCounterRegister;
    if (tmp_reg >= CurrentTickLimit) goto slow_path;
fast_path:
    …
    return;

slow_path:
    Save caller-saved registers;
    call Yield;
    Restore caller-saved registers;
    goto fast_path;
```

Programmable Timer Interrupt + Polling the Flag

- H/W and OS must provide access to a programmable timer interrupt
    - Or a high-resolution native thread *sleep*
- Define timer interrupt handler to raise *TimerTicked* flag
    - Cannot immediately switch the threads as Interpreter can be in inconsistent state
- Program the timer interrupt to call the handler with the desired period
- At thread switch points Interpreter and compiled code yield to Scheduler if the flag is raised
- Scheduler clears the flag
- To force the yield at the closest thread switch point, raise the flag
- Thread switching is not deterministic

Polling TimerTicked flag in Compiled Code

```
    if (TimerTicked) goto slow_path;
fast_path:
    …
    return;

slow_path:
    Save caller-saved registers;
    call Yield;
    Restore caller-saved registers;
    goto fast_path;
```

- If conditional call instruction is available:

```
    if (TimerTicked) call timer_tick_handler_1;
    …
    return;

timer_tick_handler_1: // Shareable between multiple points
    Save caller-saved registers;
    call Yield;
    Restore caller-saved registers;
    ret;
```

Programmable Timer Interrupt + Page Protection

- TimerTicked flag is usually clean. Can we exploit it to speed-up the check or to save the code?
  - The slow-path code is already sunk
  - The conditional branch is already predicted as non-taken
  - Eliminate the conditional branch instruction
- In addition to programmable timer interrupt, H/W and OS have to support dynamic memory protection
  - Usually available as h/w support of virtual memory (*mprotect*)
  - Static memory protection is not sufficient
- Define exception handler for invalid memory access (*SIGSEGV*)
  - To lookup a compiled method by current *pc*
  - To get a table of thread switch handlers from the method
  - To lookup the table of the handlers by the offset of current pc
  - To simulate a call of the handler

Programmable Timer Interrupt + Page Protection (2)

- Compiler has to create a table of thread switch handlers for each compiled method
  - One handle can serve multiple thread switch points in multiple methods
- Define timer interrupt handler to protect a memory page
- Program the timer interrupt to call the handler with the desired period
- At thread switch points Interpreter and compiled code access the page that may be protected
  - One memory access instruction
  - A read needs a spare register, a write does not but takes more cycles
  - The exception handler simulates a call of the thread switch handler upon return to the program context
- Scheduler unprotects the protected page
- Code size reduction with a minor performance improvement
  - Table of shared handlers vs multiple handlers & conditional branches

Programmable Timer Interrupt + Page Protection (3)

```
thread_switch_1:
    *ProtectablePage = any_reg;
    …
    return;

local_table_of_thread_switch_handlers: {
    (offset_of(thread_switch_1), global_timer_tick_handler_1)
    …
}

global_timer_tick_handler_1:
    Save caller-saved registers;
    call Yield;
    Restore caller-saved registers, common subexpressions
            and cached literals;
    ret;
```

- Local table of thread switch handlers can be compressed
    - Eliminate subsequent entries with the same handler
    - Or describe the saves and restores and interpret the description...

Hot Loops Patching

- In small frequently executed loops any extra instruction in the loop body counts
    - Virtual instruction counting: 2-4 extra instructions
    - High-resolution timer polling: 2-5 extra instructions
    - Timer interrupt + flag polling: 2-3 extra instructions
    - Timer interrupt + page protection: 1 extra instruction
    - Only very small loops need to be considered
    - No non-inlined calls in the loop body or multiple loop exits
- Can the overhead be fully eliminated?
    - Yes, if the code can be patched && programmable timer interrupt is available
    - Dynamically compiled code is located in RAM
- Compiler has to generate a table of hot loops for each method
    - Every entry defines:
        - range of code offsets for the loop body
        - original instruction(s) for backward branch at the loop end
        - offset of the sunk thread switch handler for the loop end

Hot Loops Patching (2)

- Timer interrupt handler
  - In addition to raising the flag or protecting the page:
  - To lookup the compiled method by current *pc*
  - To find the table of hot loops of the method
  - To remember the current *pc* is a global
  - To replace the backward branch with unconditional branch to the thread switch handler
    - May need to invalidate instruction cache & to flush data cache
- Thread switch handler
  - To save CPU flags, caller-saved registers
  - To yield to Thread Scheduler
  - To restore caller-saved registers, cached common subexpressions, cached literals
  - To restore CPU flags
  - To branch to the restored backward branch at the loop end
- Thread Scheduler
  - To restore all hot loops of the current compiled method

Native Interfaces

- Extended KNI
    - De facto standard introduced by KVM
    - Provides access to Java classes, instance and static fields from native code
    - Late binding by C string name
    - High overhead
- ROMStructs.h / ROMAccessors.hpp
    - Provides access to romized Java classes, instance and static fields from native code
    - Generated C/C++ representation of Java objects
    - Compile-time binding
    - Low overhead
- ANI (Asynchronous Native Interface)
    - Asynchronous native-to-native calls using a pool of native threads
    - Current Java thread gets blocked

Native Interfaces (2)

- Native method
  - Access to a native method from Java code
  - Link-time binding
  - Arguments pushed to native stack
  - Call performed on native stack
  - Result pushed to Java stack or exception is thrown
  - The code is written in C++
    - Could be written in extended Java subset ("System Java") as an annotated method in Java source
    - Statically converted to C++ source (and so harder to debug)
    - Improved type and pointer safety, less object handle declarations
    - Never implemented due to the lack of interest from the users
- Native method annotated as QuickNative
  - Lightweight version of a native method, cannot allocate memory
  - Call performed on current Java stack
- A primitive
  - Assembly implementation of performance-critical method specialized for the most common conditions
  - If a primitive returns failure, the Java method gets interpreted

Native Interfaces (3)

- Compiler macro
  - Compile-time version of a primitive to override regular method invocation or inlining
  - Executed when direct method invocation is compiled
  - If the macro returns failure, the compiler proceeds with regular method invocation or inlining
  - Implemented as a C++ function calling compiler API for register allocation, code generation and access to compile-time values
  - Lesson learned: compiler macros are extremely error-prone and require deep understanding of the particular compiler internals
    - i.e. Conditionals combined with one-pass register allocation are endless source of non-trivial bugs in the macros
  - Formally verifiable DSL would be a better choice for the macro implementation language

Compiler

- Compilation Strategies
- Compiled Code Cache Management
- Scheduled Concurrent Compilation
- One-pass Compilation by Abstract Interpretation
- Register Allocation
- Code Generation
- Optimizations for a Dynamic One-pass Compiler
- Connections of Design Choices in Dynamic Compilation

Dynamic Adaptive Compiler

- 10-20 times faster than the best assembly interpreter
    - Analyses and optimizes a compilation unit (method/function/procedure with inlined calls)
- Compiled code is longer than interpreted code
    - 5-6 times for ARM, 3.5-4 for Thumb2
    - Much longer than code generated by native compiler from similar C code
        - Method header
        - Compressed stackmaps for GC
        - Exception throwing and thread switching code
        - Relocation records
        - Deoptimization code
        - Dependencies of speculative opimizations ...
- Complexity
    - Compilers are harder than interpreters to develop, maintain, test and port to a new processor
        - Separate and minimize machine-dependent components
        - Derive assembler and disassembler from a formal description of the instruction set
        - Use a code generator generator (i.e. BURG-style)

Compilation Strategy

- When to compile?
  - Before the execution of the application
  - During the execution (a few options to choose from)
  - Convert to VM-specific LIR before the execution, run the backend during the execution
- What to compile?
  - All methods
  - Statically defined set of methods
    - Derived from profiling data of the training runs
  - The methods ever executed
  - “Hot” methods
    - Derived from current dynamic execution profile
  - Hot path within the methods
  - Execution traces (regardless of the method boundaries)

    A.Gal, C.Probst, and M.Franz. HotpathVM: an effective JIT compiler for resource-constrained devices. *Proceedings of the 2nd International Conference on Virtual Execution Environments*, pp. 144-153, 2006
- How to compile?
  - Stop-the-thread
  - Spawn a parallel native thread (from the compiler workers pool)
  - Concurrent or scheduled on the same thread to hide the pauses

Static Compilation

- Compile application before its execution
  - Conventional compilation (Ahead-Of-Time, AOT)
  - … or cross-compilation if the host differs the target
  - Well-studied conventional algorithms
  - No interpreter is necessary during execution
- Deep code analysis
  - Takes time and memory
  - Not effective for large libraries
  - Not effective for dynamic languages
    - Java: virtual and interface calls, dynamic class loading, reflection
  - Not effective for open-world models
    - A library is not aware of all its applications
    - An application is not aware of implementation of the used libraries
  - Limited availability of runtime statistics (hot/cold code, bindings of dynamic calls)
    - Training runs may not cover all scenarios
    - Application and the libraries profiles must be separated
  - Cannot exploit invariants of the dynamic state
    - i.e. an initialized class can never become uninitialized, so the initialization checks can be omitted

Dynamic Compilation

- Compile the entire application on its first run
    - … and cache the generated code to speed-up the subsequent runs
    - Delayed conventional compilation and linkage
    - Performed on the target
    - - Compiler needs to reside on the target
    - - The target is usually more resource-constrained than the host
    - - Duration of the first start is crucial for client applications
- Compile a program unit on its first execution
    - Just-In-Time (JIT)
    - A technique to spread compilations in time
    - ... and to exclude unreachable program units
    - + Dynamic state and execution statistics are available
    - - Compiler needs to reside within the VM
    - - Compilations spread unevenly
    - - Some code is executed just once (static class initializers in Java)
        - If it does not contain loops, the compilation effort never pays off

Dynamic Adaptive Compilation

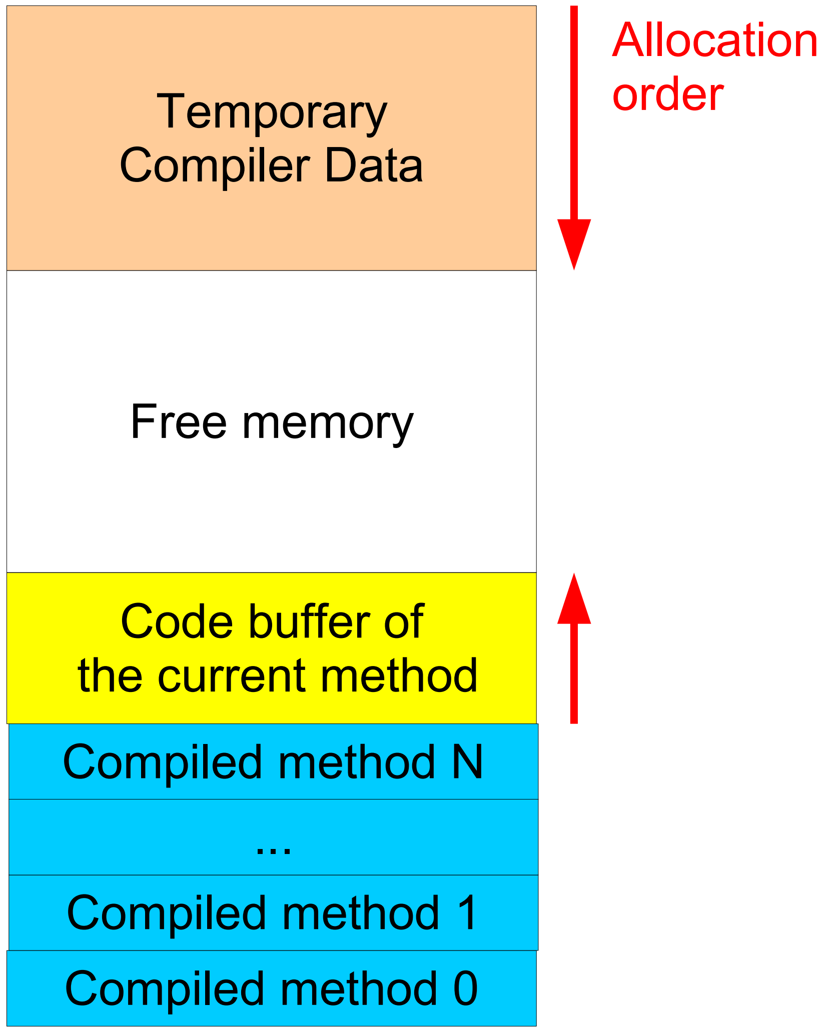
- Speculatively compile the program units that are likely to be actively used in the near feature
    - Dynamic code profiling
    - Prediction of activity in the near future by the activity in the near past
    - Speculative code specialization
    - On-Stack Replacement (OSR)
- To make a transition from the interpreted to the compiled code preserving the state
    - Replace the interpreted stack frame with the compiled one
- To make the opposite transition
    - Replace the compiled stack frame with the interpreted one
    - Frame conversion code has to be generated by the compiler
        - Available at certain points only
- First developed for Self programming language
    - Extremely dynamic prototype-based object-oriented language
    - Adopted by StrongTalk (Smalltalk with optional type declarations)
    - Animorphic/LongView Tech. (Urs Holzle, Dave Grisvold, Lars Bak) acquired by Sun Microsystems in 1997

Compiled Code Cache

- Dynamically compiled code is stored in *Compiled Code Cache*
  - Can be implemented as a special area within or outside of regular garbage-collectable heap
  - Must not be managed by the regular garbage collector
  - Liveness of compiled code is not solely defined by its reachability
- Compilation units *compete* for the cache storage
  - Hot interpreted methods to be compiled and inserted to the cache
  - Cold compiled methods to be evicted from the cache
- Reasons for compiled code eviction
  - Pressure of the hot code
    - Dispose cold code to free storage for hot code
  - Assumptions of speculative optimizations failed
    - The speculatively specialized code becomes invalid
  - Original code unloaded or replaced
    - Task terminated
    - Code instrumented by debugger or profiler
  - Compiled Code Cache has to shrink
    - Regular heap is saturated, needs to borrow memory from the caches

Compiled Code Cache in Monty VM

- Continuous area of the heap
- Dynamically adjustable size and location
- Not managed by the regular collector
- References between the heap and the cache are known to GC
- Watermarking to reduce the scope of scanned compiled methods for minor collections
    - Once the compilation complete, the compiled method cannot refer to newer objects
- Method code buffer is continuous, grows upwards in the natural direction
- Temporary data allocated downwards
- When the compilation completes:
    - Code buffer shrinks to the actual size of the code
    - Temporary data disposed



Compiled Code Size Prediction

- Before compiling a method, its compiled size has to be estimated
  - Is there enough spare memory in the cache to allocate the code?
  - Can it compete with other hot methods in the cache?
  - Large compiled code may become obsolete before the compilation completes
  - Large compiled code may cause a long pause if the compilation is not parallel or concurrent
    - Inlining decisions may require detailed call site info
- The estimate cannot be precise
  - Compilation unit includes all inlined methods
    - Inlining decisions may depend on detailed call site info
  - Generated code depends on the performed optimizations
  - If over-estimated, compilation of hot method may not start
  - If under-estimated, compilation may fail due to overflow
    - To avoid repeated overflows, a pair of the overflowed method and its code size prediction can be remembered in a small set
- Linear estimate is good enough when no inlining involved
  - A weight for every bytecode + fixed slack for the header
  - Bytecode weights are defined by code generator for the target ISA

Prediction of Interpreted Code Activity

- If a program unit was active in the nearest past, it is going to be active in the nearest future
  - Observation: many real-world programs spend most of their time in a small number of loops
  - What are exactly "the nearest past" and "the nearest future"?
  - The *time scale* may not be physical in a multitasking environment or when waiting for events
  - If we have a program in a steady state && the compiler stops the thread && the compilation speeds up the execution with a fixed factor, the compilation pays off in the execution time proportional to the compilation time
  - Compilation speed defines the duration of the nearest past
    - Predictors for slow and fast compilers may differ
- The heuristics may fail on program phase transitions and for counted loops with large number of iterations
  - Paradox 1: late compilation of an empty loop may slow down its execution
  - Paradox 2: execution of an empty loop may take longer than non-empty loop with the same number of iterations
    - Non-empty loop compiles at lower number of iterations
- Compile hot code early and fast

Prediction of Compiled Code Activity

- The same heuristics and the same time scale as for the interpreted code
  - Cold code is evicted by the pressure of the code compiled for hot interpreted methods
- Profiled events may differ
  - Different techniques for interpreted and compiled code
- Misprediction price is high
  - Prematurely evicted hot code has to be re-compiled
  - The slower the compiler, the higher the price

Code Profiling for Dynamic Compilation

- To provide data for code activity predictors
    - The infrastructure can be reconfigured for coarse performance profiling, context-sensitive memory profiling and debugging
- Hot code can be loop- or call-intensive
- Instrumentation
    - Profiling code inserted into the interpreter and compiled code
    - Precise but expensive
    - Bad for loops, good for calls
- Sampling
    - Profiling code is called periodically, scans execution stack(s)
    - Imprecise, can miss small methods
    - Event count within a sampling interval is lost
    - Good for loops, bad for calls
- Combine both techniques in a relevant time scale
    - To translate instrumentation events to sampling events, scan the recorded instrumentation events in sampling intervals
    - If necessary, event count or ordering can be taken in consideration

Profiling of Interpreted Code

- To detect loop-intensive code, on a sampling timer tick record current interpreted method
- To detect call-intensive code, record method *entries*
    - *Exits* refer to the earlier activity and are harder to instrument
- The recorded methods become *compilation candidates*
    - Current method immediately benefits from compilation and so has priority over the others
    - Small set of candidates is sufficient to keep the compiler busy
- Old compilation candidates become obsolete
    - Assign the *latest invocation age* to every candidate
        - Increment the age on every sampling timer tick
    - Remove old candidates from the set
    - The set can be implemented as a cyclic buffer with a few pointers to separate the ages
- Simpler alternative technique for a fast compiler
    - Immediately compile current method
    - Delay compilation of a call-intensive method until its next invocation
        - Variable method entry overwrites itself to simulate a sampling timer tick on the next execution

Basic Blocks Profiling

- Count basic block entries to distinguish hot and cold basic blocks during compilation
    - To perform hot path straightening or omission of cold blocks
    - Rough estimate is sufficient for the distinction
    - Allocate a byte array of counters for the method, one counter per each bytecode
    - Instrument implementation of local branches to increment the counters
- Not for the fast early compilation
    - Many basic blocks are not yet visited
    - The counts are low and less reliable than static branch prediction
        - Backward branches predicted as taken
        - Forward branches and switches predicted as non-taken
- Staged compilation
    - Do not profile interpreted basic blocks
    - For a hot interpreted method, first generate slightly optimized code instrumented to gather the execution statistics including block entries
        - No need to allocate a byte counter for every bytecode
    - Later use the gathered statistics for more optimized compilation

Profiling of Compiled Code

- To detect loop-intensive code, on a sampling timer tick record all compiled methods in active execution stacks
    - Do not scan threads waiting for events
- To detect call-intensive code, record method *entries*
    - Mark compiled methods invoked within current sampling interval
    - Bit marking is *idempotent*, no synchronization is required
    - The recency is more predictive than the frequency (count of the calls)
- Use the recorded activity for *exponential decay system*
    - Value of the activity decays with its age
    - Unsigned *temperature* is assigned to every compiled method
    - If the method activity is detected, set the highest bit of the temperature
        - Or a few highest bits as invocations count within current interval
    - Decay the temperatures every Nth sampling timer tick (*decay period*)
        - Logically shift them right by one bit
    - What is the optimal decay period?
        - The slower the compiler, the higher the decay period
        - On program phase transitions temporarily accelerate the decay
        - Monty VM samples at the thread switching rate (default 10 ms), the temperatures are bytes, two high bits are invocations counter, decay period is 1 (value of method activity decays to 0 in 8*10 = 80 ms)

Fast Eviction of Compiled Methods

- When starting a compilation or shrinking the cache, coldest compiled methods may need to be evicted
  - Given the size of memory to free, compute the temperature of the hottest method to evict
  - Given the size of memory to free, evict the coldest methods until enough memory is freed
- Sort compiled methods by growth of the temperature
  - Once sorted, the methods mostly keep the ordering
  - Just active methods have to be moved to the end
  - Doubly linked cyclic list needs 2 extra words per method
- Alternative technique: histogram of allocated memory for temperatures
  - Works well for narrow temperature values
  - Allocate and zero a local array of max_temperature counters
  - Scan the compiled methods, increment the counter for the temperature by the allocated size of the method
  - To find the temperature of the hottest method to evict, scan the counters by temperature until the sum reaches the given limit
  - Scan and evict methods with not higher temperatures until enough memory is freed

Concurrent Compilation

- Compiler runs as a coroutine to the thread
  - Cooperative thread switching is assumed
  - Execution of the compiler and the thread is interleaved until the compilation complete
  - Compilation can be started, suspended, resumed and aborted
  - Once the compilation complete, replace the interpreted activation of the method with the compiled one
    - All threads are at their thread switch points
    - Topmost frame can (almost) always be replaced
    - Deeper frames often cannot be replaced as the compiled frame size may exceed the interpreted frame size
    - Interpreted-to-compiled transition options:
      - Replace topmost frame(s) only
      - Replace only the frames that are easy to replace
      - Replace deeper frames on return to them
      - If execution stacks are elastic, size of any frame can be changed
- Monty VM: due to constrained memory, no more than one compilation can be active at a time

Scheduled Compilation

- Concurrent compilation can be scheduled to hide pauses and to improve responsiveness
  - Suspend the compilation on external events to reduce the event handling latency
    - Compiler periodically checks if the event queue is non-empty
  - Limit duration of individual compilation pauses
    - Scheduler sets the limit
    - Compiler periodically checks current duration against the limit
  - Avoid pause clusterization in short time intervals
    - Dense series of pauses are perceived as a long pause
    - GC and class loading can also cause the pauses
    - Let the application to work at least for a given percentage in time intervals of given duration (*Minimal Mutator Utilization*, MMU)
  - Suppress compilation on application start and heap saturation

Partial Compilation

- Interpreted code is preserved after its compilation
- Compiled-to-interpreted code transition is supported
- So, the code can be compiled partially
    - Omitted code is replaced with the transition to interpreter (*UncommonTrap*)
- Omission of complex, slow and rarely used code fragments
    - Exception handlers
- «Lazy» development of the compiler
    - Gradually expand the set of compiled constructs

Compiler Architecture

- Multi-pass compiler
    - Build one or more convenient intermediate representations (IR)
    - Analyse, annotate, transform and generate lower level IR
    - Promotes cleaner modularity of the compiler
    - Convenient for development and testing of individual algorithms
    - IR construction consumes time and memory
- One-pass compiler
    - Do not build any IR for compiler
        - Use interpreted code
        - … or LIR statically generated from the interpreted code
    - Combine all analyses, transformations and code generation in one pass
    - Most of conventional algorithms require a non-trivial adaptation
- Trace compiler
    - Greatly simplified one-pass compiler for very specific kind of programs (a tree with backward branches in the leafs)
- Lesson learned
    - If the project started today for similar targets, a better choice would be to try trace compiler for DAC and multi-pass compiler for AOT
        - Trace compilation was not discovered when Monty project started

JVM Bytecode as IR

- Not a high-level IR
    - Some properties of the source code are lost during compilation to bytecode
    - Control structures are reduced to branches
    - Type arguments of generics are erased
- Not a low-level IR
    - All modern processors pass instruction arguments in registers
- Relatively convenient for interpretation
    - Too many codes require a large instruction decoder
- Inconvenient for code transformations
    - (Almost) not extendable
    - Hard to annotate
    - Instruction removal and insertion are complicated in a linear array with branches to indices

Dynamic Compiler in Monty VM

- One-pass
  - Memory is scarce and too slow for IR construction
  - Makes a fast pre-pass to mark basic block boundaries / destinations of branches in a byte array
  - Backward branches are recognized as potential loops
  - Can look ahead for several bytecodes to distinguish short patterns for conditional and combined instructions
- Abstract interpretation of bytecodes
- Compiled code is allocated in a continuous buffer
- Compiled code is relocatable
  - Mostly position-independent
  - Compiled-to-compiled code invocations require relocation after compiled code cache compaction
  - References from compiled code to the heap require relocation after garbage collection
- Complied code and temporary data are allocated in a separately-managed area of the heap
  - Compiled code lifetime is different from the application objects

Extended Basic Blocks

- *Basic Block*
  - Maximal sequence of instructions with single entry and single exit
- *Extended (Single Entry, Multiple Exits) Basic Block*
  - Maximal sequence of instructions with single entry
  - Multiple exits are allowed
- Exception Throwing in Monty VM
  - Implicit exception throwing is not an exit
  - Explicit throwing (bytecode *throw*) is an exit
  - Exception handler is not an entry (handled by the interpreter)
- Invocations in Monty VM
  - Do the invocations leave & entry the blocks?
  - In Monty basic blocks are not broken by invocations

Abstract Interpretation of an Extended Basic Block

- Execution context (*BytecodeCompileClosure*) simulates a stack frame (*VirtualStackFrame*) of the interpreter
- Run-time values are replaced with compile-time values
- Compile-time value attributes
    - Type
    - Offset in the stack frame (for arguments and locals)
    - Range or exact value (min=max=value) (for numbers)
    - May/must be null/non-null (for objects)
    - Exact class (for objects, propagated from constructors)
    - May be assigned to a register or register pair
- Interpretation of an instruction modifies the simulated stack frame
    - If the instruction semantics is not fully expressed by the change of compile-time state, code is generated to perform the respective change at run-time

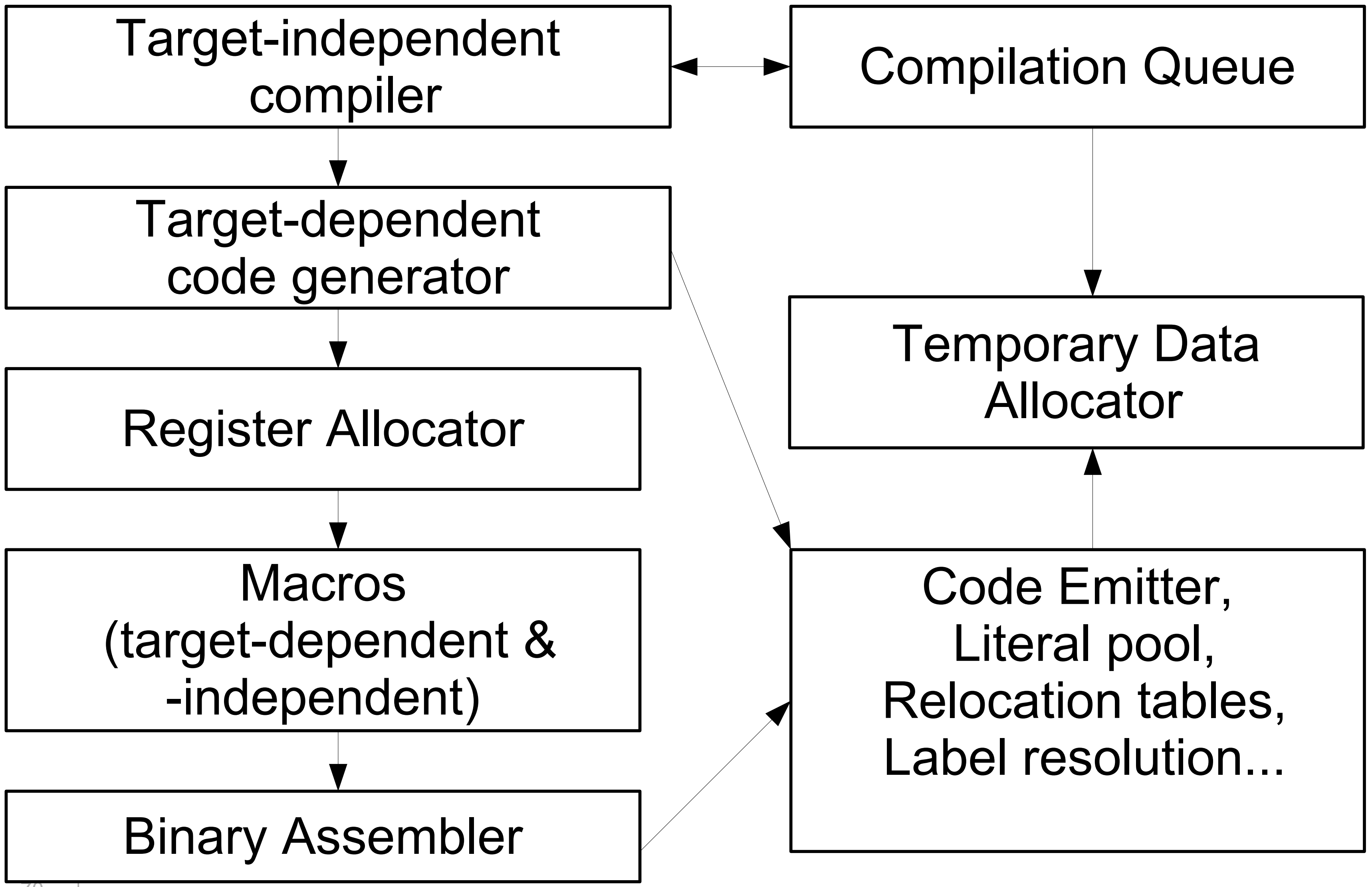
Abstract Interpretation of a Method

- *Entries* — array to map start of basic block to compilation context by bytecode index (*bci*)
  - *Null* if the context is not created yet
- *CompilationQueue* — priority queue of block compilations
  - Queue ordering defines the order of compilation
  - Slow code has to be compiled last (*Cold Code Sinking*)
  - In the beginning the queue contains single compilation of the method entry block
- When interpretation reaches a branch
  - For each target, check if its compilation context is created
  - If not created
    - If the branch is unconditional, continue interpretation from the target
    - Otherwise it is a *control flow split*, clone current state to the target. If the target is hot code, enqueue current compilation, continue with the target. Otherwise enqueue the target, continue with the current compilation.
  - Otherwise it is a control flow merge.
    - Depending on the target temperature, generate conforming code inline and branch to the target, or sink the generation of conforming code and continue compilation of the current block

Conformance of Compilation Contexts

- Compilation context of already compiled code is fixed
    - Context defines initial attributes of the compiled-time values and so the assumptions for the compilation
    - Cannot change the generated code
    - Actually, multiple copies of compiled code could be generated for the same block in different contexts (beware of combinatorial expansion of compiled code)
    - To guarantee conformance, derived attributes and common subexpressions have to be erased when the context is cloned
        - Local variables can have different values in different blocks
        - The loss of derived attributes leads to degradation of quality of generated code (superfluous null checks, dynamic casts...) until the attributes derived again
        - Short loops are the most sensitive to the loss of attributes
        - Handle short loops with a single backward branch specially – see *Loop Peeling*
- Type consistency and equal stack depth are guaranteed by bytecode verifier
- Memory and register allocations may vary
    - Conformance code may need to read, write locals and to move values between registers

Structure of Monty VM Dynamic Compiler


Target-independent compiler
Compilation Queue
Target-dependent code generator
Temporary Data Allocator
Register Allocator
Macros (target-dependent & -independent)
Code Emitter, Literal pool, Relocation tables, Label resolution...
Binary Assembler

Register Allocation

- FIFO
  - Build a queue of registers. When register is allocated, move it to the end. When disposed, move it to the beginning.
- Round Robin
  - Scan the array of registers from the current position, at the end jump to the beginning. Move current position to the found register.
- LRU
  - Build a queue of registers. When register is used, move it to the end. When disposed, move it to the beginning.
- Graph Coloring
- Linear Scan
  - The most practical algorithm
  - Requires low-level IR
- Puzzle Solving (for nested registers)
- Single-pass compilation limits the choice to FIFO, Round Robin or LRU
  - Modified Round Robin is used (unused > computable > spillable)

Code Generation

- Manual implementation
    - A set of routines to generate parametrized code patterns
    - The set is the same for most targets, the implementation varies
- Syntax-directed approach
    - Use a grammar to define the code patterns
- Bottom-Up Rewriting System (BURS)
    - LIR construction is required
    - Nymeyer, A. and Katoen, J.-P. Code generation based on formal BURS theory and heuristic search. 1997
    - Variety of implementations: BURG, IBURG, MBURG, WBURG, GBURG

Optimizations

- Should not be too expensive
    - Duration of compilations is added to the duration of the execution
        - No targets with excessive h/w parallelism
    - Temporary data of compiler are borrowed from the application memory
    - Implementation of the compiler increases VM code footprint
- Tiered compilation
    - Not used in Monty due to memory, time and footprint constraints
        - Otherwise would be easy to implement
    - Code either interpreted or compiled, no intermediate levels of optimization

Optimizations in Monty VM

- Unreachable Code Elimination
    - Natural side effect of abstract interpretation
- Unconditional Branch Straightening
    - If current basic block ends with unconditional branch to a destination block with the single predecessor, append target block to the current block
    - Reduces number of branches
    - Entry counts are computed at the preliminary bytecode scan
- Slow Code Sinking
    - Compilation queue with priorities
        - 2 or 3 priorities are enough (hot, cold and stubs)
    - Static prediction of cold code
- Branch-to-branch Straightening
    - Forward conditional or unconditional branch to a conditional or unconditional branch to its destination
    - … as long as the branch conditions can be combined for target CPU

Optimizations in Monty VM (2)

- Idiom Replacement
  - Look ahead a few instructions for frequently used instruction patterns
  - Signed or unsigned extension on memory load, shift & add, shift & mask, multiply & accumulate, conditional instructions, short loops
- Constant Folding
  - Natural side effect of abstract interpretation
- Local Property Propagation
  - Limited to extended basic blocks due to single-pass compilation
  - Constant Propagation
  - Copy Propagation
  - Null Check Elimination
  - Class Initialization Check Elimination
  - Dynamic Cast Elimination
- Common Subexpression Elimination
  - Dictionary compression of bytecode strings
  - Dictionary entry is a range of bytecodes associated with a register
  - On an expression start look ahead until the longest match is found, replace the computation with the register

Speculative Code Specialization

- Code specialization
  - Generation of more optimal code for the particular constraints
- Speculative code specialization
  - For constraints that are held at the moment of compilation && are likely but not guaranteed to hold in the future
  - If the assumed constraints are violated, the generated code is not valid and must not be executed
    - Initialization of a class can change results of class hierarchy analysis (CHA)
    - Instantiation of a class can change results of rapid type analysis (RTA)
    - Unexpected class of the receiver can change the result of dynamic binding
  - To avoid execution of invalid code
    - Switch to the interpretation
    - Switch to still valid generic version of the compiled code
  - Invalid code has to be disposed
    - Eagerly – as soon as it becomes invalid
    - Lazily – when the control leaves its last activation
    - Or when it becomes cold according to the profiler
  - Multiple specializations of the same code may exist

Speculative Devirtualization of Dynamic Calls

- Assume that past bindings of the call site extend to the future
    - Store statistics of previous bindings of a call site in an inline cache
    - First designed for Smalltalk-80 implementation

      Peter Deutsch, Allen Schiffman. Efficient Implementation of the Smalltalk-80 System. POPL'84
    - Most effective for typeless and dynamically typed languages

      Urz Holzle, Craig Chambers, David Ungar. Optimizing Dynamically-typed Object-oriented Languages with Polymorphic Inline Caches. ECOOP'91
    - Depending on the collected binding statistics, classify the call site as *monomorphic*, *polymorphic* or *megamorphic*
        - Generate different code accordingly
    - Most call sites are monomorphic
        - Guarded devirtualization: generate a *type guard*, replace the dynamic call with a direct or inlined call
        - Hoist and eliminate type guards, use *thin guards*
        - The guarded code is always correct and so can be deoptimized lazily

Speculative Devirtualization of Dynamic Calls (2)

- Assume that current class hierarchy remains mostly the same in the future
    - Consider only currently instantiated classes (RTA)
    - … or only currently initialized classes (CHA)
        - Less precise but easier to implement: no need to instrument object constructors or replace with an uncommon trap
    - Interface call with the only matching implementation is replaced by the virtual call
    - Virtual call of effectively final method is replaced by the direct call
    - Direct call can be inlined
    - Unguarded devirtualization
        - Annotate the entire speculatively specialized method with a set of made assumptions
        - Class instantiation (or initialization) may invalidate the code
        - Push the change to all speculatively compiled methods
        - If the assumptions are invalidated, eagerly deoptimize the method
- Monty VM: an educated choice not to use inline caches
    - Code is often immutable (stored in ROM, shared between tasks)
    - One-pass compiler cannot hoist the guards, deeply inline the calls
    - Due to early compilation the caches are often empty / unbound

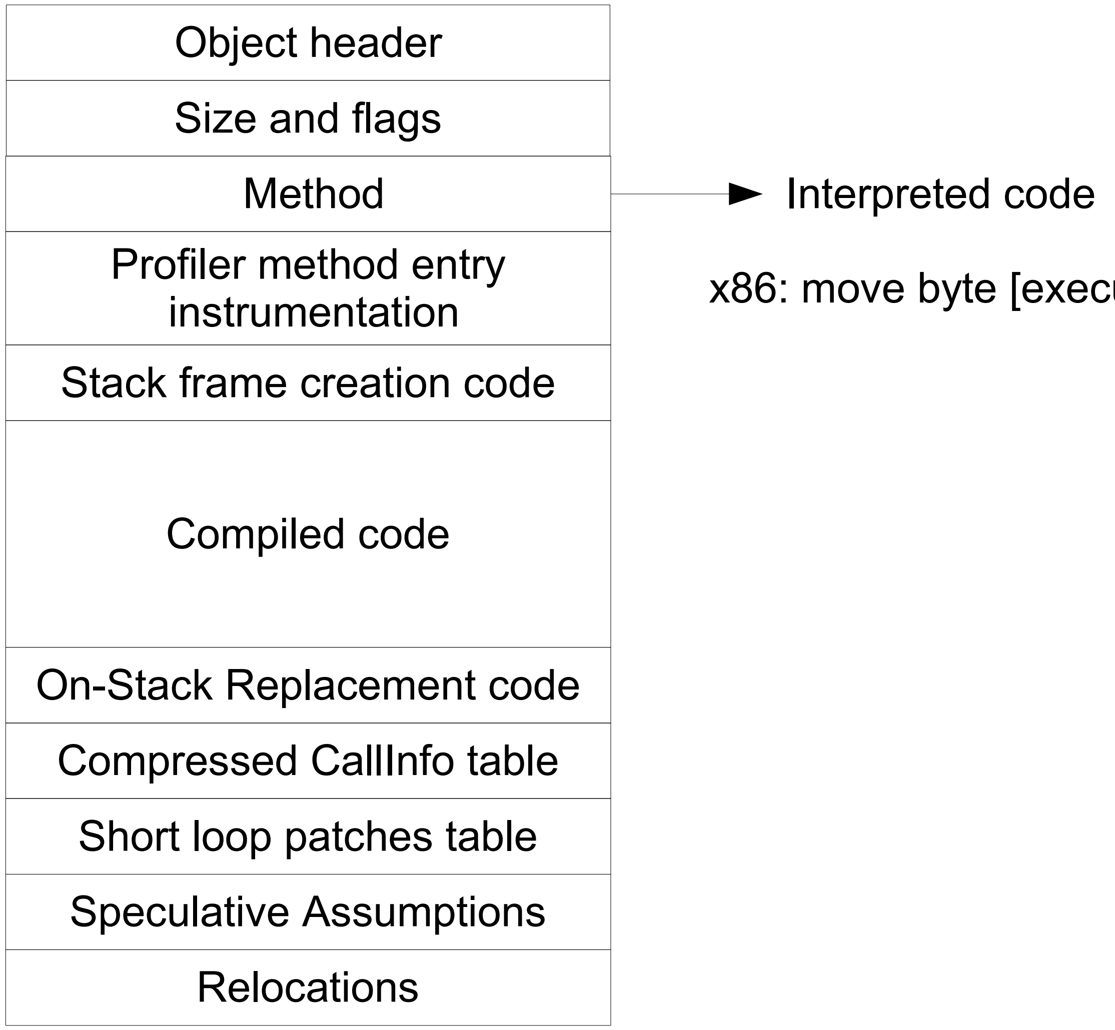
Optimizations in Monty VM (3)

- Instruction Strength Reduction
  - Replacement of some multiplications with additions and shifts
  - Replacement of division by a power of two with the shift
- Direct Call Inlining
  - Replace direct call of a short method with the code of the callee
  - Limitation: only leaf methods without loops can be inlined
  - Loops and nested calls require on-stack replacement entries
  - Replacement of a single compiled frame with multiple interpreted frames is complicated deep in the execution stack
- Speculative Unguarded Devirtualization
  - Assume that class hierarchy is limited to the initialized classes at the moment of compilation
  - Perform CHA to devirtualize the calls
  - Remember the assumptions used for devirtualization
  - Do not generate and hoist assumption guards in the code
  - Instead, on class initialization deoptimize methods with invalidated assumptions

Optimizations in Monty VM (4)

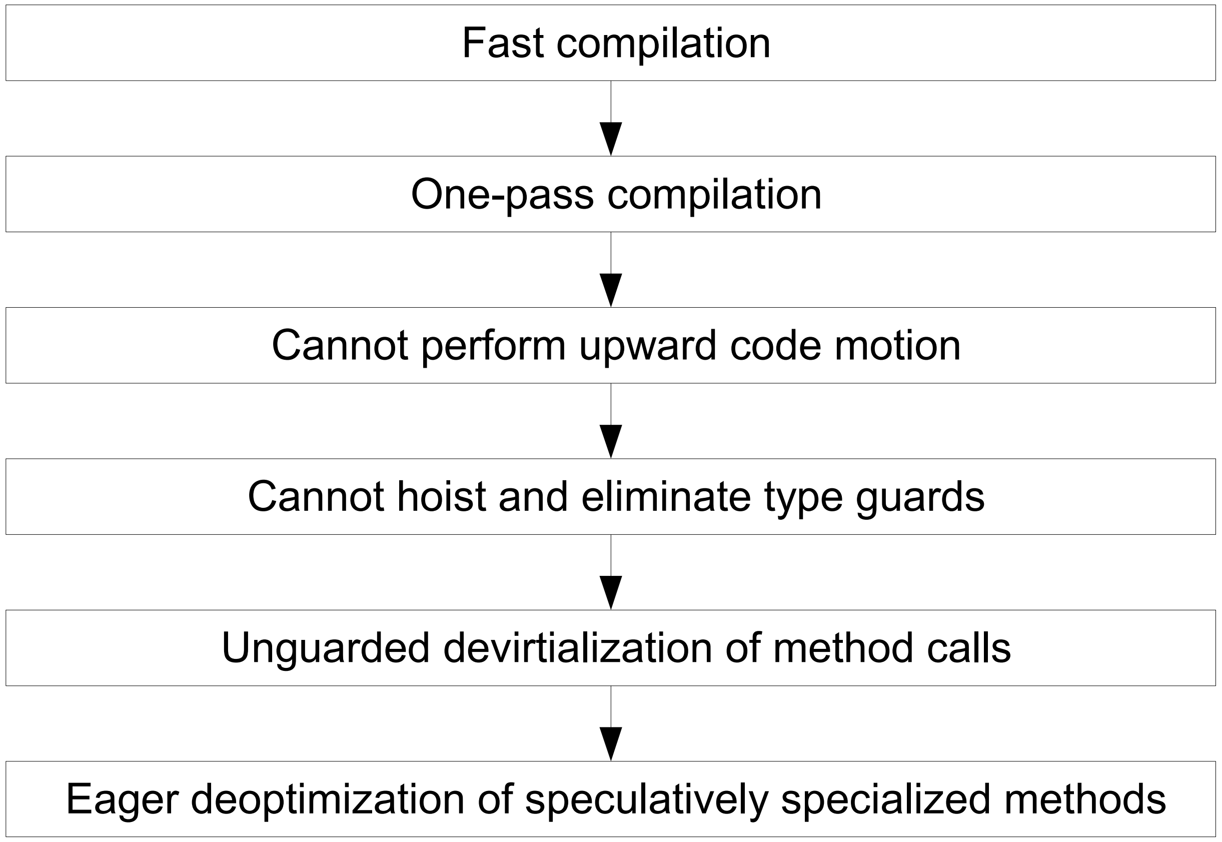
- Loop Peeling
  - A single-pass adaptation of loop invariant code hoisting
  - A copy of the loop body is appended to the loop preheader
    - Class initializations, null checks and dynamic casts are performed there and propagated to the loop
  - Other copy of the loop body is appended to the loop preheader for CSE and Round Robin register allocator
    - Register allocations and common subexpressions change after the hoisting of class initializations, null checks and dynamic casts
    - A special pass through the loop body has to be made to exclude killed subexpressions
  - The implementation is messy
    - Too complex for a one-pass compiler
    - Too far from abstract interpretation style
  - Often no performance improvement observed
    - The decision to peel a loop is made before the loop is analyzed
    - There can be no code to hoist
    - Array index range checks re not eliminated
- Local Instruction Scheduling (for ARM only)
  - Peep-hole optimization in code generator to reduce the stalls

Compiled Method Layout in Monty VM

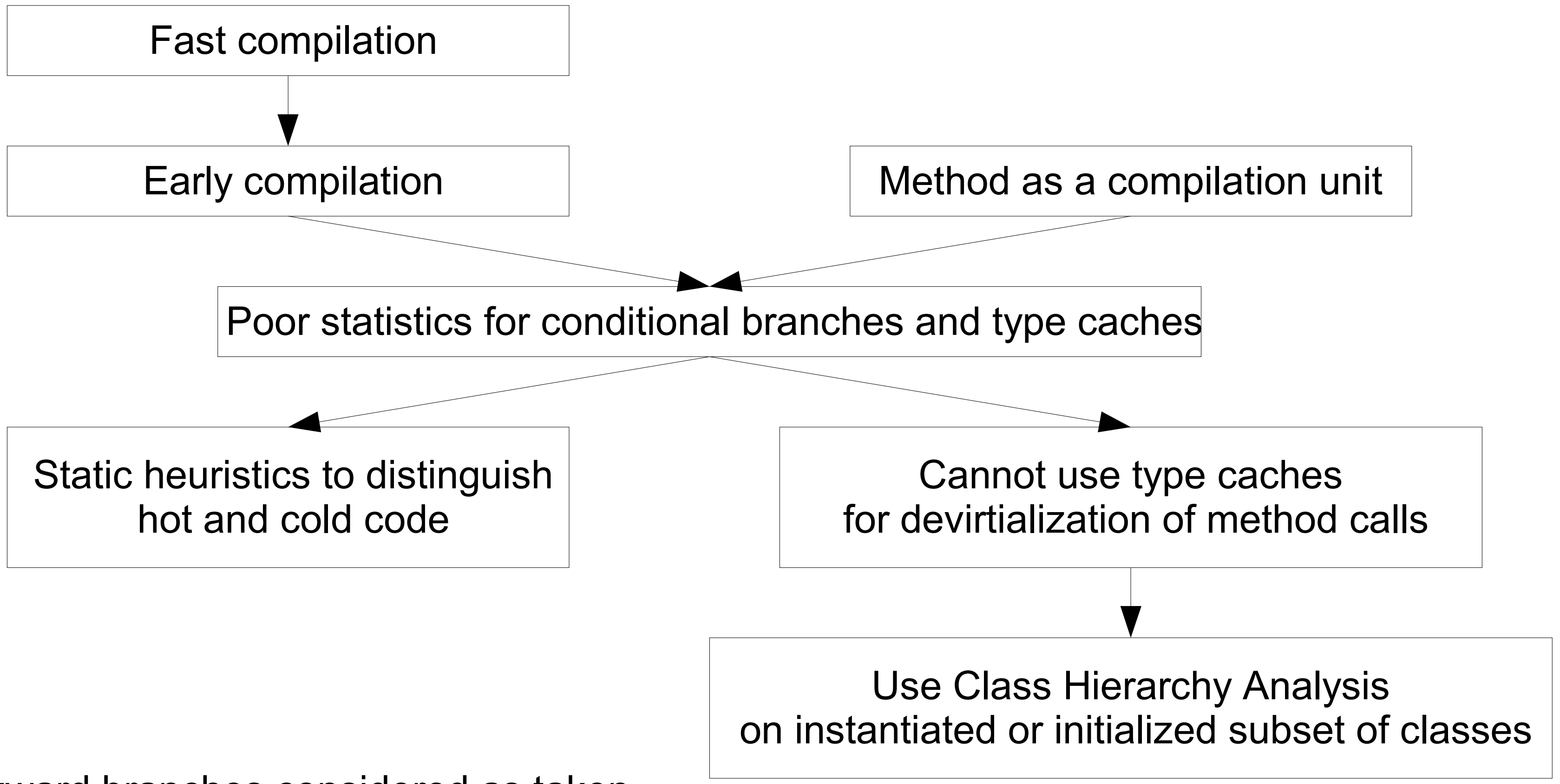


x86: move byte [execution_sensor+method_index], 0

Connections of Design Choices in Dynamic Compilation


Fast compilation
One-pass compilation
Cannot perform upward code motion
Cannot hoist and eliminate type guards
Unguarded devirtialization of method calls
Eager deoptimization of speculatively specialized methods

Connections of Design Choices in Dynamic Compilation (2)


Fast compilation
Early compilation
Method as a compilation unit
Poor statistics for conditional branches and type caches
Static heuristics to distinguish hot and cold code
Cannot use type caches for devirtialization of method calls
Use Class Hierarchy Analysis on instantiated or initialized subset of classes

Backward branches considered as taken,
and forward – as non-taken.

Cold code: exception throwing and catching,
class initialization, thread switching,
offline memory allocation

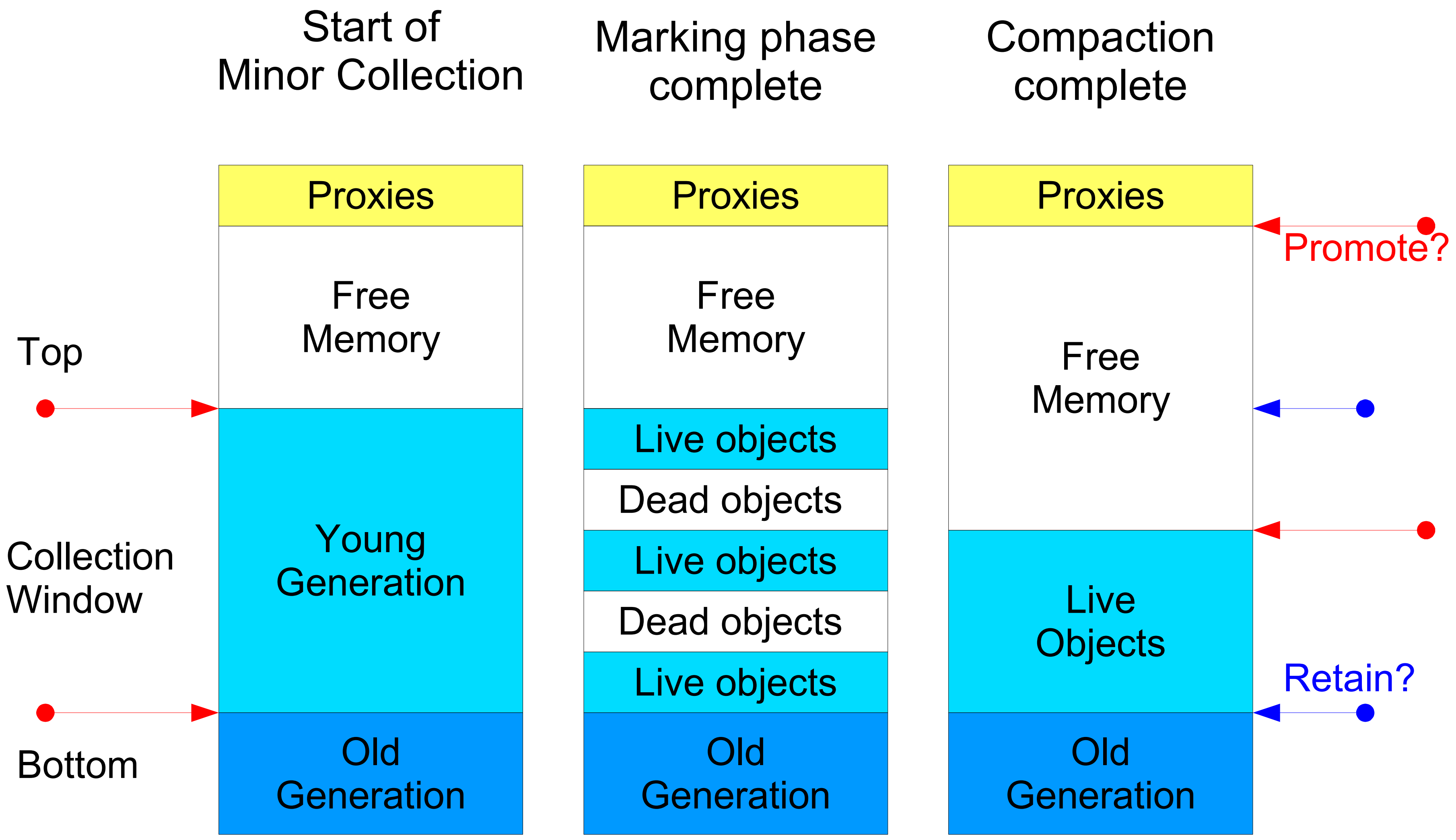
Garbage Collection

- Unified Memory Management
- Elastic Heap
- Generational Mark'n'Compact
- Benefits of Bumping Pointer Memory Allocation
- Design Choices in Garbage Collection

Garbage Collection

- A variant of generational mark'n'compact collection
    - One optional generation managed by the same algorithm
        - Can be disabled to lower memory and implementation footprint
    - Memory is scarce, so heap occupancy is high (over 75%)
    - Sliding young generation (with the size and the position controlled)
        - En masse promotion policy (either all or none survivors promoted)
        - Partial promotion could be implemented by the cost of scanning the promoted objects for references to retained objects to add to the remembered set
        - When survivors are promoted, young generation slides up by the total size of the promoted objects
        - When survivors are retained, young generation stays in place
        - When high amount of garbage expected, temporarily disable the promotion
    - Separate areas for large long-living objects and compiled code
- Elastic heap
    - Optional dynamic size adjustment based on workload
    - Custom policy handler can be installed
    - Requires OS support for memory mapping
        - Reserve address space for the entire heap
        - Commint/uncommit pages after major collection

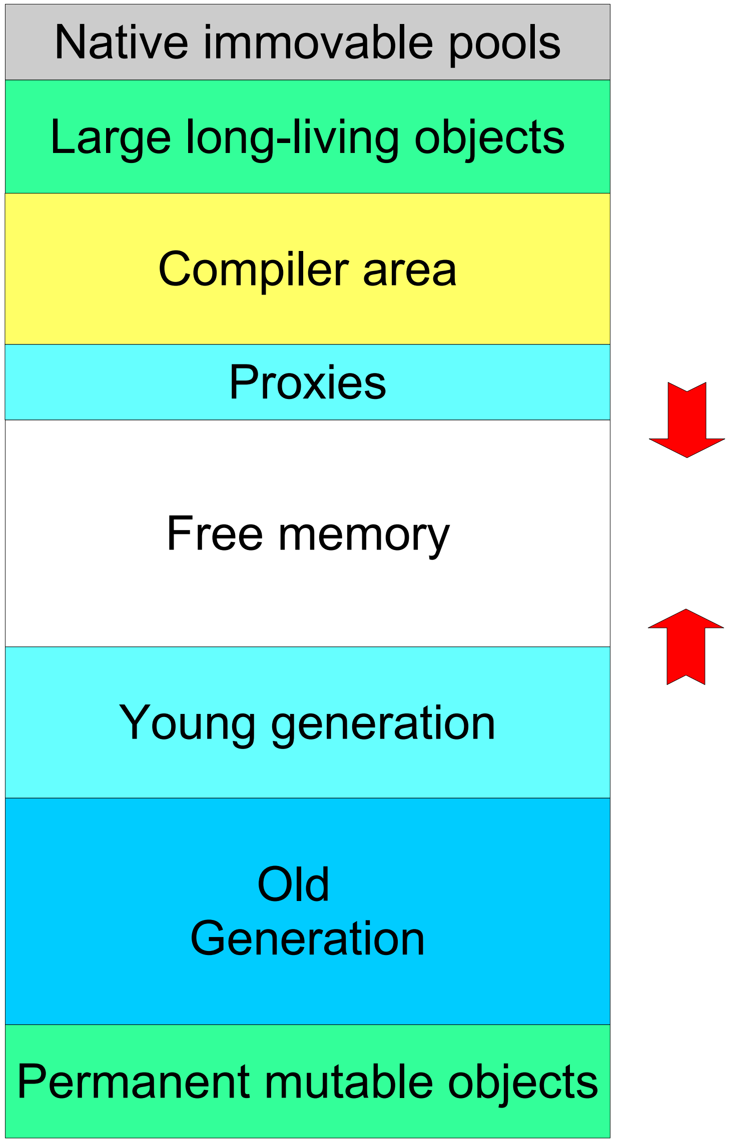
Object Promotion with Sliding Window



Garbage Collection (2)

- Unified memory management
    - VM manages all memory allocations of the runtime
        - No OS or C runtime allocations are allowed
    - All mutable objects reside in the heap
        - Execution stacks, classes, auxiliary non-language objects of VM
    - Complicates garbage collection
        - Object layout description can move during collection
    - Immutable objects can be allocated in ROM
- Static heap allocation
    - An option to define heap size and location at build time
    - The heap is represented by a static array assigned to a separate segment
    - The segment address can be set by the linker script
    - Helps to maximize heap size on devices with non-continuous memory address ranges
- Single-word object header
    - Represents object class, hash value and synchronization lock
    - Two words for an array

Heap Layout



- Proxy is an intermediate agent between the object and its class
- A proxy stores:
  - pointer to the class
  - hash code
  - reference to the lock
  - cached pointer to speed up the access to the *virtual method table* of the class
- All proxies have the same size and layout
- Initially all instances of a class share the same *prototypical* proxy of the class
- Proxy is created for an individual object when first hash code is computed or synchronization performed
- Separate proxy area is necessary to reduce number of heap scans during GC

Bumping Pointer Memory Allocation

- To allocate an object, advance allocation top by the aligned object size
    - Just a few instructions for a preliminary cleaned heap
    - No synchronizations or Thread-Local Allocation Buffers required
        - Single native thread
- Good initial locality of objects
- To allocate a fixed sequence of objects
    - Compute and request the total size
    - Split the chunk to individual objects
        - Together with preservation of allocation order during GC naturally supports weak speculative object inlining (not implemented)
        - Headers of inlined objects retained
        - Inlined object is addressed by fixed offset from the root object
        - If a reference to inlined object escapes, no conversion of heap objects is required
- To allocate a variable-size object of unknown size
    - Compute a conservative estimate
    - Request memory
    - Once the object size is known, shrink the object

Mark'n'Compact Algorithm

- Use largest continuous free memory area for marking stack
- Recursively mark all reachable objects from the roots
  - Derived pointers require special processing
- Perform language-specific semantic actions on the loss of reachability
  - Schedule execution of finalizers, handle special kinds of references
- Skip solid prefix
- Build the *break table*
  - Link dead intervals (*breaks*), hash for faster search by object
- Compact proxies, install forwarding pointers
- Update references in the roots, regular objects and proxies
  - Use Break Table to compute object offset from its current location
- Compact live intervals of regular objects
  - Individual objects are not decodable after the update of references
  - Use Break Table to iterate through the intervals

Derived Pointers

- A pointer to an object with a non-zero displacement
  - In general, the displacement can be negative or exceed the object size
  - Usually points to the object field or behind the end of the object
    - Running pointer for an array
- Break table updates pointers to the object start, inside the object and right behind the object equally well
  - Find live interval of the pointer, subtract the interval offset
- Derived pointers are rare, mostly located in execution stacks
  - Return address
  - Current instruction address
  - Dynamic chain – previous frame address
- Interior derived pointer
  - Stored in the object, points inside the object
    - Dynamic chain
  - No need to mark
  - Easier to update – compute the object offset once, subtract from each interior pointer of the object

Solid Prefix

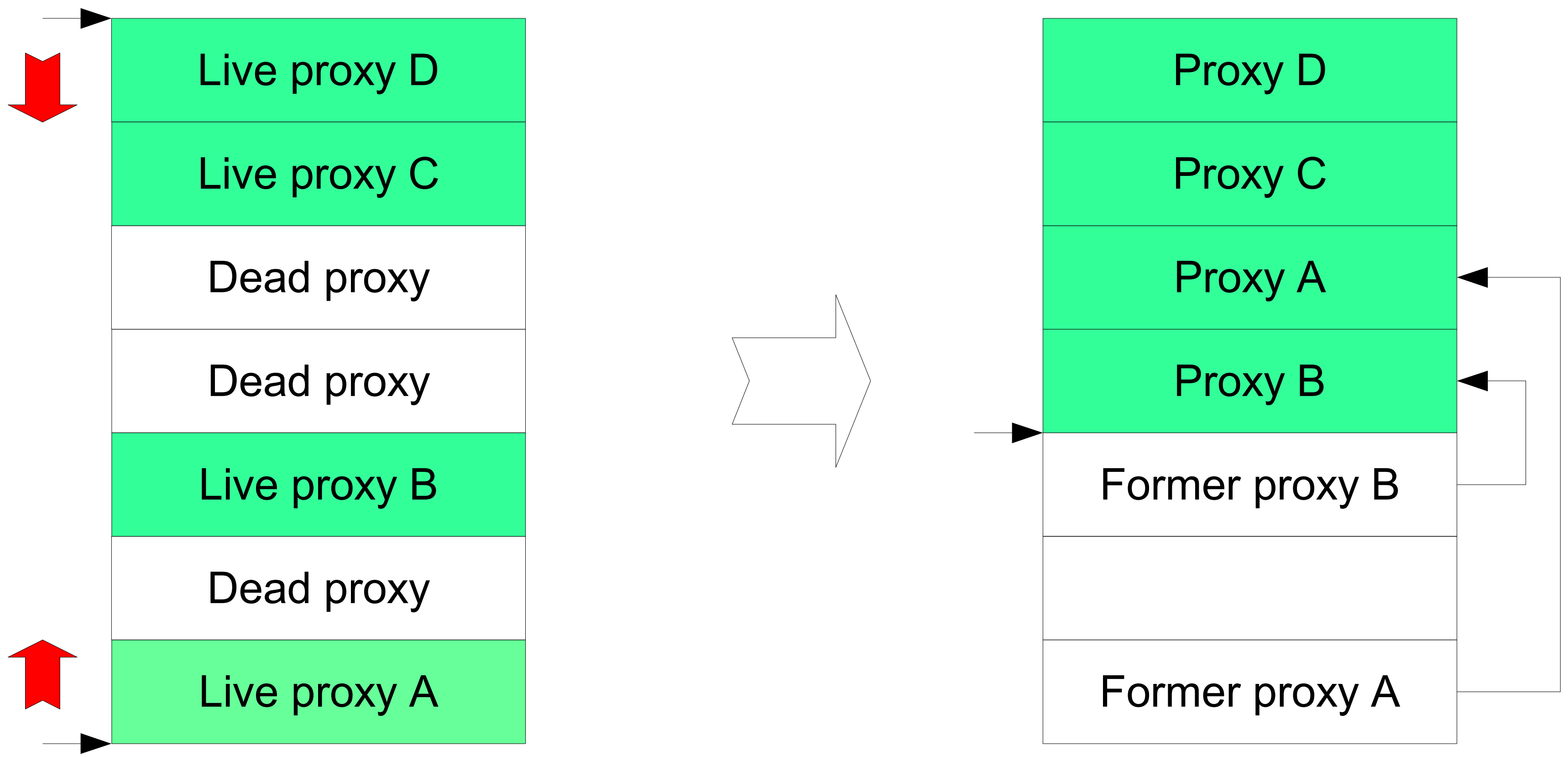
- The algorithm preserves the allocation order
- Solid Prefix is a maximal sequence of reachable objects at the bottom of collection area
- Scan the heap from the bottom until first unmarked object found
- Clean the marks of the scanned objects
    - Object header is already cached
- Solid Prefix does not move – adjust the bottom of compaction area
    - Update pointers to compaction area only
    - Save on unnecessary break table lookups

Linked Hashed Break Table

- Reachable (live) and unreachable (dead) objects form intermittent intervals
- Scan the compaction area, form an ascending list of dead intervals
  - Store next interval address in the first word of the dead interval
  - Store the dead interval size in the second word
  - In a single-word dead interval set the lowest bit of the next address
- The table is used for reference updates and compaction
  - During compaction live objects in compaction area move downwards
  - All objects of a live interval move by the same *relocation offset* of the interval
  - Relocation offset is a sum of lengths of the preceding dead intervals
    - Linear complexity
  - To speed-up the summarization, slice the compaction area by segments of a power of 2 words
  - For each segment store its initial relocation offset and address of its first dead interval
  - Marking stack is not used anymore – store the segments there

Compaction of Proxies

- The proxies are of fixed size, already marked
- Use "two finger" algorithm to compact proxies in one pass



Write Barrier & Remembered Set

- *Write barrier* records all stores of references to young objects to fields of old objects in the *remembered set*
- Compiled write barrier is simplified
  - No compiled check that the field is old and the object is young
  - No barrier is generated for stores to a newborn object
  - If the compiler can derive that the object is old or null, the record is removed from the remembered set
- The remembered set has pointer-size granularity
  - Records individual fields of old objects that may refer to young gen
  - Implemented as a bitset (one bit per aligned word of the heap)
  - Memory overhead is 1/32 of the heap size
  - Single native thread, no need to buffer the updates
  - Modern CPUs have special instructions for bit setting
  - Easy to scan remembered fields without decoding layout of the containing object
  - Two-level implementation tried, no benefits observed in absence of native multi-threading, rejected
    - Sequential Store Buffer as primary storage
    - Bitset as secondary storage (backup)

Design Choices in Garbage Collection

- Copying collection is not an option
  - Usually memory is scarce, heap liveness is high
  - Use copying collection for young generation only?
    - Could be an option for larger targets
    - Does not preserve allocation order, breaks expected invariants
- Reference counters are not practical
  - Memory and performance overhead is too high
  - Secondary collector required to dispose cyclic garbage
  - Java does not mandate prompt finalization on loss of the last reference to the object from the application
- Precise collection is strongly preferred over conservative
  - Runtime is designed to cooperate well with the collector
- Mark'n'compact vs mark'n'sweep
  - Mark'n'sweep with optional compact-after-sweep is indeed a practical alternative
    - Worked well in KVM
    - Object relocation is greatly reduced
    - But slower memory allocation, allocation order not preserved

Design Choices in Garbage Collection 2

- Generational collection
  - Lower average pauses, higher throughput
  - Hypothesis of high infant mortality
    - Young generation size must be sufficient for low promotion rate
  - Hypothesis of rare old-to-young references
    - Greatly reduces reference update overhead
    - Works even in very tight heaps with high promotion rate
  - Generational collection is always beneficial
  - Two generations are enough
- Chunky vs continuous heap
  - Continuous heaps are simpler
  - Chunky mark'n'compact collector is implemented and performs well
  - Never deployed due to the lack of customer demand
  - Can be more beneficial for parallel collection
- Parallel collection
  - No multicore/multiprocessor targets (but nice to have anyway)
  - All phases of the current algorithm can be easily parallelized

Design Choices in Garbage Collection 3

- Low-latency concurrent collection
  - Minor collection pause is below 5 ms
  - Major collections are rare but may take 10x time
  - Nice to have but:
    - The worst-case concurrent collection pause is 2x stop-the-world
    - If concurrent collection cannot yield enough free memory due to floating garbage, the world has to be stopped, the collection retried
    - The runtime and native code in libraries is written with assumptions:
      - No read barrier
      - GC can occur and objects can relocate only when memory is allocated
    - Concurrent variations on the current collector look terribly inefficient
  - The implementation cost is dominated not by the garbage collector itself but numerous changes required in the runtime and the libraries
  - Hardware-assisted barrier is very rarely an option
    - Most of the targets don't have h/w or OS support for dynamic memory protection
- Scoped memory backed by garbage collection
  - Nice to have if real-time code is written in Java
    - Monty is not Squawk, all low-level code is written in C++
  - Must be implemented without excessive annotations

Design Choices in Garbage Collection 4

- Object header vs separate bitset to store marking bits
    - The bitset to remember references for generational collection can be re-used to store marking bits
        - On major collections all objects promoted to old generation and so the entire remembered set is cleared
        - On minor collections only the old generation segment of the bitset stores the remembered set, the young generation segment can be used for marking
    - Both options implemented, object header is faster
    - Object header is preferred for parallel marking
        - Setting a fixed bit in otherwise immutable word is idempotent
    - Separate bitset is preferred for concurrent collection
        - Avoids synchronization of the collector and the mutator on object header updates

Design Choices in Garbage Collection 5

- Compressed forwarding pointers vs Break table
  - Original Lars Bak's GC implementation used compressed forwarding pointers
  - The rationale is to replace O(n) table lookup with O(1) forwarding pointer decoding (however see the table construction time below)
  - An extra word in object header would be necessary to store a forwarding pointer for a live object
  - Compression can be used to fit both proxy and forwarding pointers in one word for smaller heaps
    - Live objects can only move downwards
    - The distance between live objects cannot grow due to GC
    - Slice the heap into segments
    - Store forwarding pointer for the segment base in a *table of slices*
    - Store the offset from the base in the object header
  - Heap layout simplified - proxies allocated as ordinary objects
  - Additional memory is required for the table of slices
    - Table of slices size and construction time is O(n2) of the heap size
  - Additional heap scan is required to decompress the headers
  - Both options implemented, Break Table is always faster
    - If a very small marking stack is used, the speed-up due to the break table hashing diminishes but the collection is dominated by frequent recoveries of marking stack overflow

Initial State Memorization

- Introduction to the Initial State Memorization
- Difficulties of Initial State Memorization for Java
- Initial Image Optimizations
- Variants of Initial State Memorization in Monty VM

Initial State Memorization

- A technique to speed-up VM and application start and to reduce dynamic memory footprint
  - Also helps to share immutable data between multiple tasks
- Cold VM start
  - The heap is empty
  - Internal structures are not initialized
  - No classes exist
  - VM executes special bootstrap code
    - To initialize internal structures
    - To load a number of system classes, to resolve symbolic references and to initialize the classes
    - To create a number of objects and to connect them together
- The same code is executed at every VM start
- The same objects are created
  - As long as system classes are the same
  - … and VM behavior is not altered by command-line options
- Can we memorize the initial state and to restore it instead of the execution of this bootstrap code?

Initial State Memorization (2)

- It is a good old well-known technique
    - Lisp, Smalltalk, Forth...
- Run a special application (“zygote process”)
    - VM bootstraps
    - The application loads additional classes, creates additional objects and makes them reachable from the roots
    - The application traverses the heap and saves its state to a file
        - ROM image, DLL...
- Provide VM with an option to start from a given initial state

Initial State Memorization for Java

- Can this technique be adopted by Java?
- In common case, the answer is negative
  - Java classes may contain arbitrary class initializers <*clinit*>
  - Class initializer may depend on non-local variables
    - i.e. current time, locale...
  - … and to produce non-local side effects
  - The non-local side effects can spread anywhere
  - May change class initialization order
- Fortunately, non-trivial class initializers are rare

Conservative Class Initialization

- Recognize safe-to-initialize classes
  - Trivial static fields initialization is always safe
  - Hard-coded heuristics for frequently used code patterns
    - i.e. arrays of strings
  - Supplementary annotations to force or defeat the early initialization of a class
    - In the sources
    - In configuration files for the zygote process
  - Contradictory annotations is an error
    - Initialization of *InitAtBuild* class can require initialization of *DontInitAtBuild* class
- If you are not sure it is safe, do not force class initialization
  - If a class is not initialized early, it will be initialized in ordinary way
- Is it always possible to avoid unsafe class initialization?
  - In the common case, the answer is negative
  - Cannot avoid initialization of bootstrap classes
  - … and the classes used by zygote process
    - i.e. to write the image to a file
  - These classes are involuntarily initialized

Avoiding Involuntary Class Initialization

- Minimize number of bootstrap classes
- Carefully review implementation of bootstrap classes
- Implement zygote process mostly in native code
    - Alternative: implement Java in Java, run zygote process on the host
    - Possible alternative: run zygote process in other isolate
- There still can be a few exceptions
    - In CLDC StreamReader/Writer class reads character encoding from internal Helper class
    - … and Helper class static initializer reads the encoding from the system property
- Sometimes the problem can be solved by repeating the initialization of already initialized class at VM start
    - i.e. Helper class in CLDC
    - Generally it is not safe to repeat class initialization
    - It is against Java Language Specification
    - The effects can be observed if some values from the early initialization are not overwritten or accessed before the overwriting by the second initialization

Initial Image Optimizations

- Zygote process can edit and optimize the state before saving it to a file
- Eliminate unreachable classes, fields and methods
    - Class initializer becomes unreachable after the class initialization
    - Don't forget about dependencies from native code
- Eliminate non-public symbolic names
    - Reflection and stack trace may suffer
- Separate immutable objects
    - Place them off-heap in ROM or CONST data section
- Merge constant pools and string bodies
- Quicken bytecodes
- Devirtualize calls where applicable
- Infer *final* attribute for effectively final objects
- Inline small final methods (i.e. field accessors)
- Selectively compile code ahead-of-time

Closed- and Open-world Assumptions

- Closed world
    - All classes are present at the moment of memorization
    - Entry points are defined
        - i.e. public static method main of a given class
- Open world
    - Arbitrary classes can be loaded during execution
    - All public symbols are entry points
- Precision of analyses depends on the choice of the world
    - … and the effect of the optimizatons
        - The quantitative effects significantly vary depending on the code
    - Closed world for highly optimized stripped code
    - Open world for extensible, debuggable code

Reachable objects

- Initial image contains classes, their methods, fields, instances and arrays
- Reachable methods can be derived from the entry points
    - Virtual calls and class loading by name complicate the matter
    - RTA is fast and works well for devirtualization
        - Better but more resource-consuming techniques are also known
    - Class loading by variable name can be annotated to define the set of loadable classes
        - Constant names rarely occur but can be analyzed automatically
    - Native code also has to be analyzed and/or annotated
        - CLANG greatly simplifies the analysis
- Class is *instantiable* if a constructor of its instances is reachable
    - Native code also needs to be analysed
- Reachable fields are referenced by reachable methods
    - Only readable fields need to be considered
    - Write-only fields can be eliminated
        - The code reading this field can be unreachable

Reachable instances and classes

- A class instance or an array is reachable if it can be reached through reachable fields from the roots
  - Add class of reachable instance to instantiable classes
  - Add finalizer of reachable instance to reachable methods
  - Repeat the previous steps of the analysis as necessary
- A class is reachable if:
  - It is instantiable
  - Or it contains reachable methods or fields
  - Or a reachable class inherits from this class
  - Or a reachable class implements this interface
  - Or the class is referenced by a reachable method
    - Bytecodes ldc, instanceof, checkcast
    - Reflection (CLDC: Class.forName)
- Unreachable methods, fields, classes and their instances can be eliminated
- Additional optimizations can be performed over *effectively abstract* classes
  - Class hierarchy flattening

Elimination of unreachable fields in Java

- Can we eliminate unreachable fields in Java?
  - The answer is usually negative for object/reference fields
- Java programs may assign semantics on the loss of reachability via finalizers and special references
  - Reachability for the runtime is not reachability for the application
  - An object reachable through an unreachable for the application field is still reachable for the runtime
- Unreachable object field can be eliminated if it can never be the last reference on the ways from the roots to a finalizable or specially referenced object
  - It is hard or impossible to proof for practical libraries and applications unless special references are not reachable
  - Type analysis cannot help here:
    - Object of any class can be specially referenced
    - Type signature of generic invocations is erased
  - Global DFA for the entire runtime with the application is very resource-consuming
    - Large number of modeled object constructors and references
    - High algorithmic complexity

Initial State Memorization in Monty VM

- ROMization
    - Build-time conversion of system classes, performed by a stand-alone utility a.k.a. *Class Prelinker* or *Class Preloader*
    - Implemented as a special build of VM with native zygote process
    - Converts a set of classes to *ROMImage.cpp* file to (cross-)compile and to link with VM
    - Additional files are generated to simplify access, analysis and debugging of the image
    - The initial state with the classes becomes permanently compiled into VM
- "Monet"
    - Device-side binary conversion of applications and shared libraries
        - Work-in-progress: host-side binary conversion
    - Built-in VM feature to convert an application to a binary image which can be mapped to memory and executed
        - The image is not portable - can only be executed by VM that generated it
    - On AMS discretion, the conversion can be performed at:
        - application installation time
        - one of the application starts
        - as a background process

Multitenancy

- Isolates
- Thread Scheduling
- Native Resource Quotas
- Exact Memory Quotas for Compacted Shared Heap

Multitenancy

- Single native process, multiple Java *isolates*
  - Cannot rely on OS processes for:
    - Isolation of references
    - Task termination with the resource reclamation
    - Resource usage minitoring and quotas
- Extended subset of Isolates API (JSR 121)
  - Resource quotas
  - Synchronous termination of an isolate with all resources reclaimed
  - Lightweight inter-isolate communication with shared objects
- Shared heap with exact memory quotas
  - Instant redistribution of unused memory between isolates
  - However separate heaps would provide lower individual GC pauses
- Flat thread scheduling
  - No hierarchical scheduling for the tasks first, the threads second
  - Thread priority can still be computed depending on its task
- Multiple execution profiles
  - Isolates can run in different and incompatible environments

Isolates Implementation in Monty

- Every isolate has its own:
  - Class name space
  - Static variables
  - Synchronization monitors
  - Threads
  - Dynamically created objects
  - Resource quotas and usage counters
- When a thread switches to a thread of different isolate, the scheduler overloads task-specific static VM state
  - List of classes
  - Current task id and reference to the task object
  - Current resource quotas
- Resolution of a symbolic class reference to class index in the context of current task
- Access to static variables via class index in current class list
  - Class index cannot be resolved to class address in shared code
    - … and so the variable cannot be resolved to its address
  - The overhead of indirect access to static variables is minor

Referential Isolation

- Java code has no access to objects of other isolates
    - Unless the object is passed through inter-isolate call
- Native code can break the isolation
    - The same process, the same address space
- CLDC does not allow native code in applications
- The implementation supports native code in trusted libraries and applications
    - Trusted code must be ROMized

Resource Quotas

- Built-in resource quota management
  - Cannot rely on OS capabilities
  - Isolates compete for available resources
  - Java runtime competes with other components/processes
- Resource types defined at build time
- Resource quotas assigned to every isolate and to the entire runtime
- Monitoring of current resource usage
- The quotas are exact – cannot be temporarily exceeded
- Intelligent resource reclamation
  - Resources are wrapped into natively finalizable objects
  - VM tracks resource wrappers per isolate, per generation
  - When resource quota is exceeded, the minimum scope is selected for the collection
  - It is a safety net, too expensive and unpredictable for regular reclamation
    - Use try-with-resources, Closeable and AutoCloseable in Java

Memory Quotas

- Memory is a special resource
  - Performance of memory allocation is critical
  - Memory allocations are ubiquitous and inlined in compiled code
  - Zero performance and code size overheads are highly desirable
- Separate heaps vs Shared heap
  - Garbage collection
    - Separate heaps can be collected independently and in parallel
    - Lower GC pauses, higher the parallelism
  - Elastic heaps depending on the workload
    - Separate heaps need a mechanism to balance memory load
    - Chunky heap, memory mapping or relocation of objects during or after collection can be necessary
  - Memory quotas
    - Shared heap requires special memory quota management
    - For separate heaps the heap size is the memory quota

Exact Memory Quotas for Compacted Shared Heap

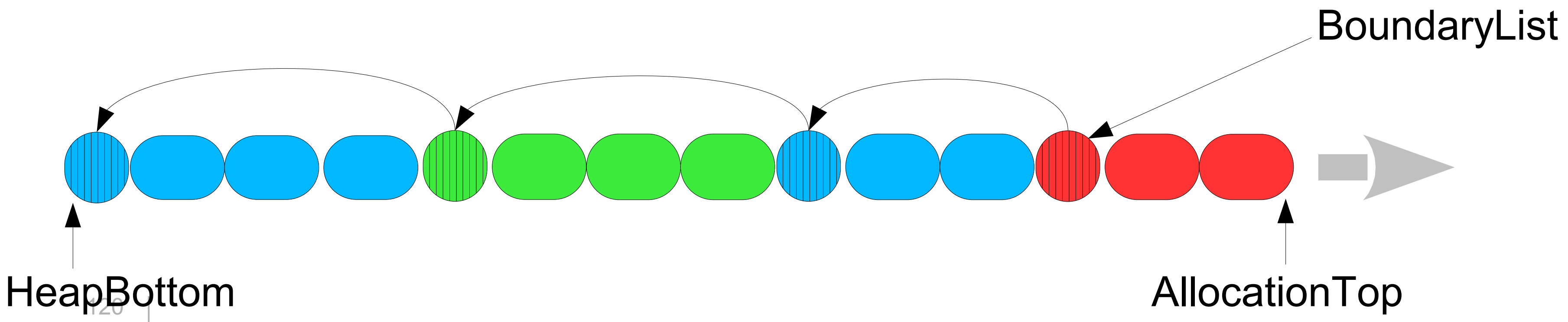
- The heap is shared between tasks
- Inlined bumping pointer allocation
  - Allocation top pointer is incremented by the object size, compared with the allocation end pointer, if exceeded the GC is called
  - Allocation end pointer can be used to limit allocations
    - … while the called GC can do the rest of work offline
  - If allocation end is null, the first allocation attempt calls the GC
  - Objects of different tasks form intermittent object ranges
- Allocation order is preserved by the collector
  - So are preserved the object ranges
- Generational collection

  -
    ```
    used_memory = used_memory_in_old_generation +
                  used_memory_in_young_generation;
    ```

  -
    ```
    used_memory_in_young_generation =
                  retained_memory_after_last_collection +
                  newly_allocated_memory;
    ```
  - Liveness of these conservative estimates can be determined by a garbage collection in the respective scope

Memory Usage Accounting by Ranges

- Object ranges vs instrumentation of the marking phase
  - Task memory usage can be computed by scanning the heap from the task roots, marking already visited objects
  - Marking phase of the collector can be instrumented to avoid additional passes
  - Implemented, tested the performance, rejected – the overhead is significant
    - The bottleneck is the size computation for every object
  - Separate object ranges with *boundary* objects
  - Let a boundary to define task *ownership* of the preceding range
  - Link boundaries in descending singly-linked list
  - Object range size is the distance between its boundaries
  - After garbage collection empty ranges to be removed
  - … and neighbor ranges with the same owner to be coalesced



Tools

- Built-in tools
- Distributed Tooling Architecture

Built-in Tools

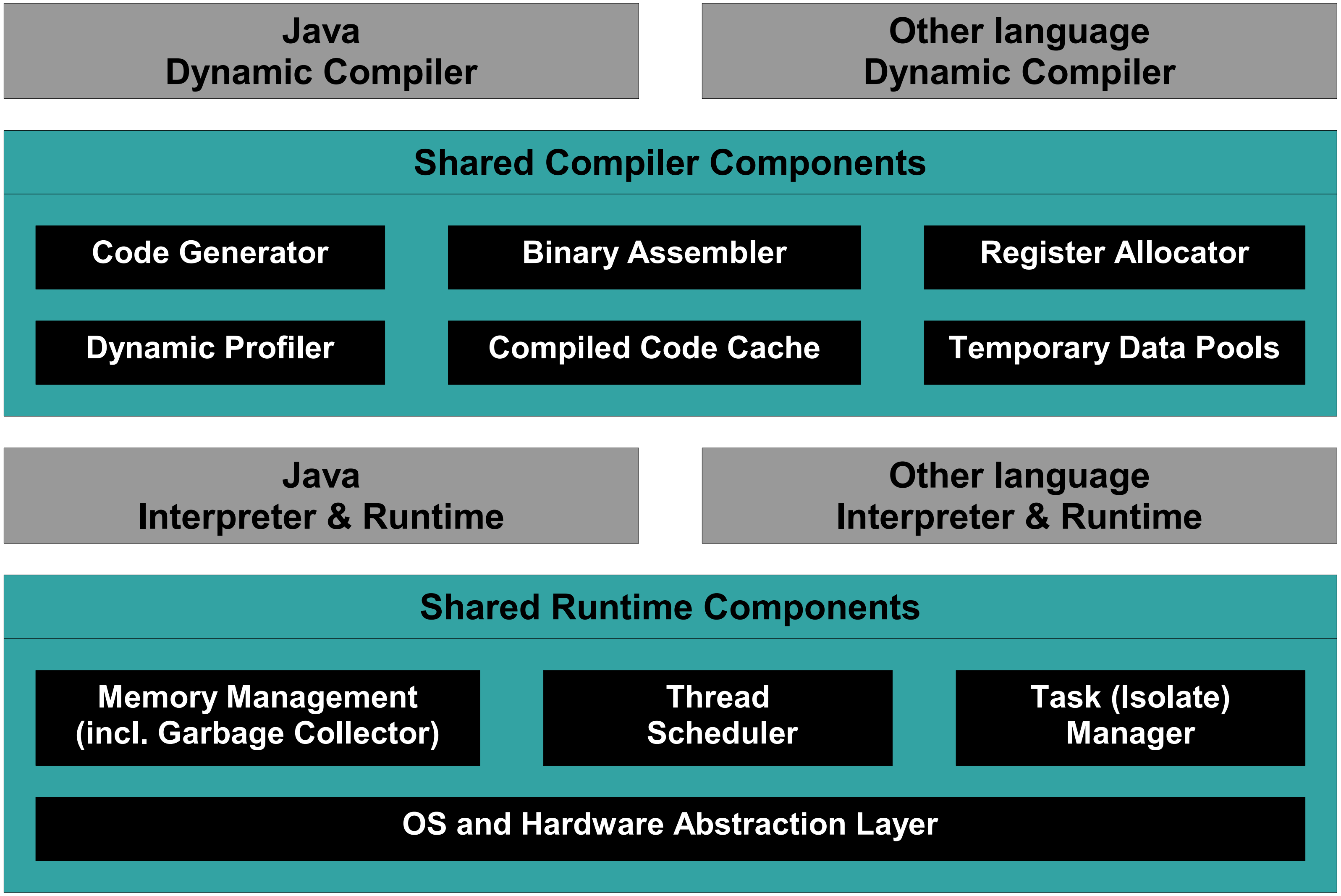
- VM provides a number of optional built-in tools
  - Java debugger
  - Sampling and instrumenting Java performance profilers
  - Runtime performance counters
  - Latency measurement and analysis
  - Lightweight memory usage monitor
  - Heap dump
  - Execution stack backtrace
  - Printout of individual objects
  - Compiled code disassembler
  - Logs and traces
  - Deadlock detector...
- These tools can be enabled individually at build time
  - Enabled by default in debug build
  - Consume RAM and ROM for implementation
- Built-in tools often cannot fit into small production devices
  - Simulation is rarely an option because of the peripherals, target-only OS and native libraries

Distributed Tooling Architecture

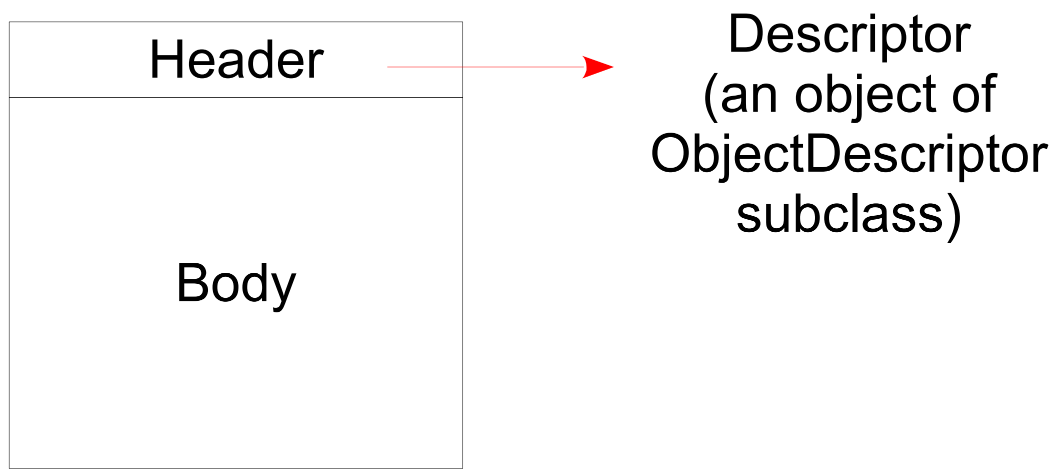
- The tooling functions are distributed between the device and the *device proxy*
- Small agent resides in the device
  - Multiplexes communication channel
  - For maximum throughput communicates with the proxy by proprietary protocol
    - Asynchronous exchange by timestamped variable-size buffers
  - Performs simple semantic actions on the proxy request or on specified events
    - Send or receive a block of data by given address and size
    - Send incremental updates on heap allocations and relocations
    - Send method entries and exits
  - Can redirect system and application output to proxy
- The proxy resides on the host
  - Provides standard tooling APIs for external tools (ME SDK, NetBeans, Mission Control, Eclipse MAT...)
  - Models computational state of the device with necessary for active tools details
  - Can request current state or state updates from the device

Symmetric Multilanguage Architecture

Symmetric Multilanguage Architecture

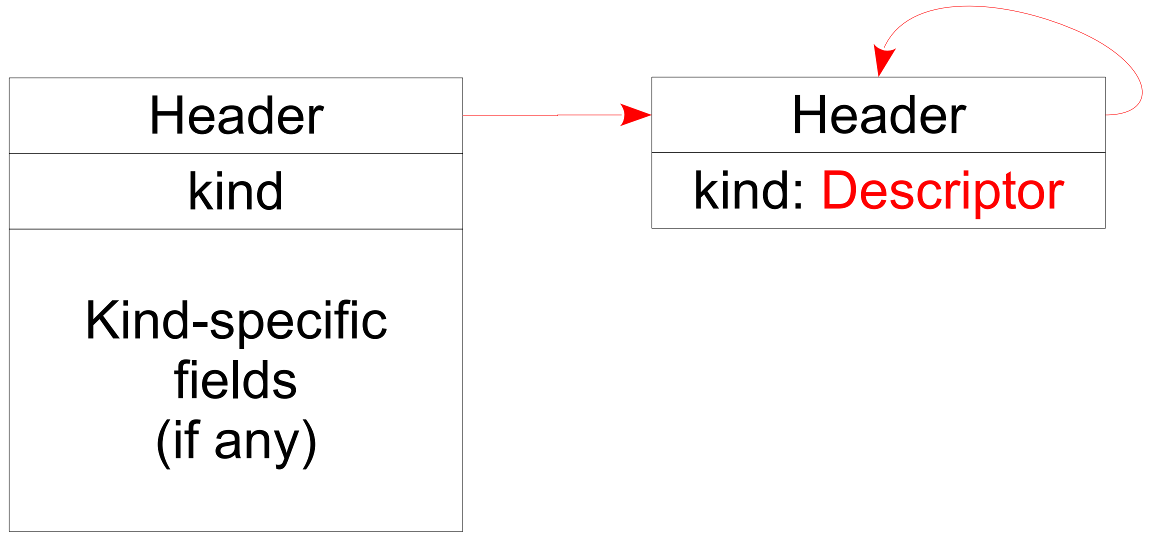

Java Dynamic Compiler
Other language Dynamic Compiler
Shared Compiler Components
Code Generator
Binary Assembler
Register Allocator
Dynamic Profiler
Compiled Code Cache
Temporary Data Pools
Java Interpreter & Runtime
Other language Interpreter & Runtime
Shared Runtime Components
Memory Management (incl. Garbage Collector)
Thread Scheduler
Task (Isolate) Manager
OS and Hardware Abstraction Layer

Object Layout



- Objects are aligned at word boundary
    - Can be located in heap or in ROM
- Single-word object header points to Descriptor
    - The proxy between the object and its descriptor is not shown
    - The proxy can store hash code and synchronization lock

Object Descriptor Layout

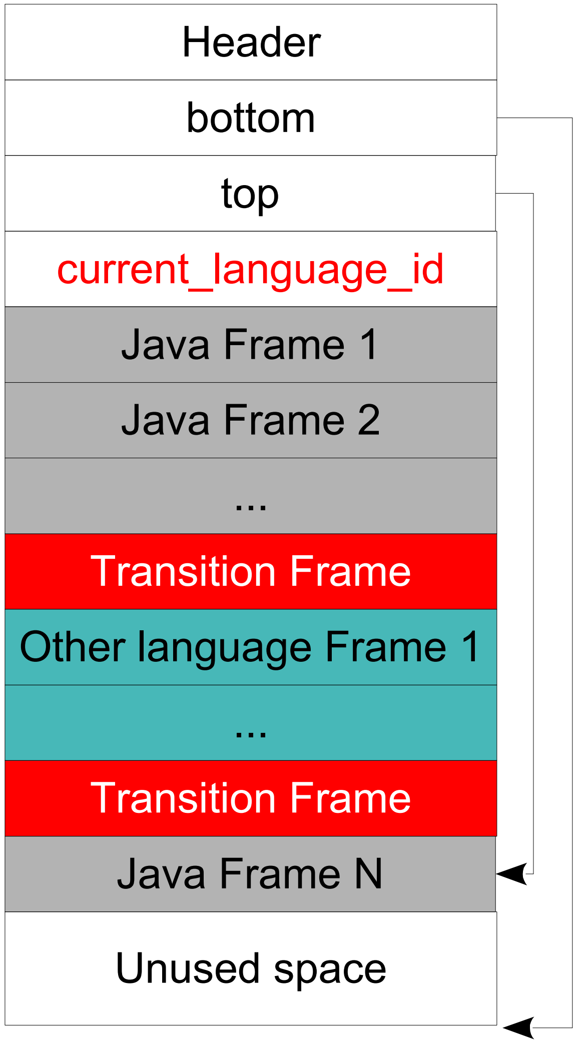


- Object Descriptor
  - Defines object kind
  - Computes object size (for given object)
  - Enumerates pointer fields (of given object)
  - Relocates pointer fields and the object itself
    - Objects may contain derived pointers (i.e. return address in stack frames)
  - Must not refer to other objects for object kind, size or relocation
    - Otherwise after update of the descriptor references the garbage collector will fail to decode objects with this descriptor

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

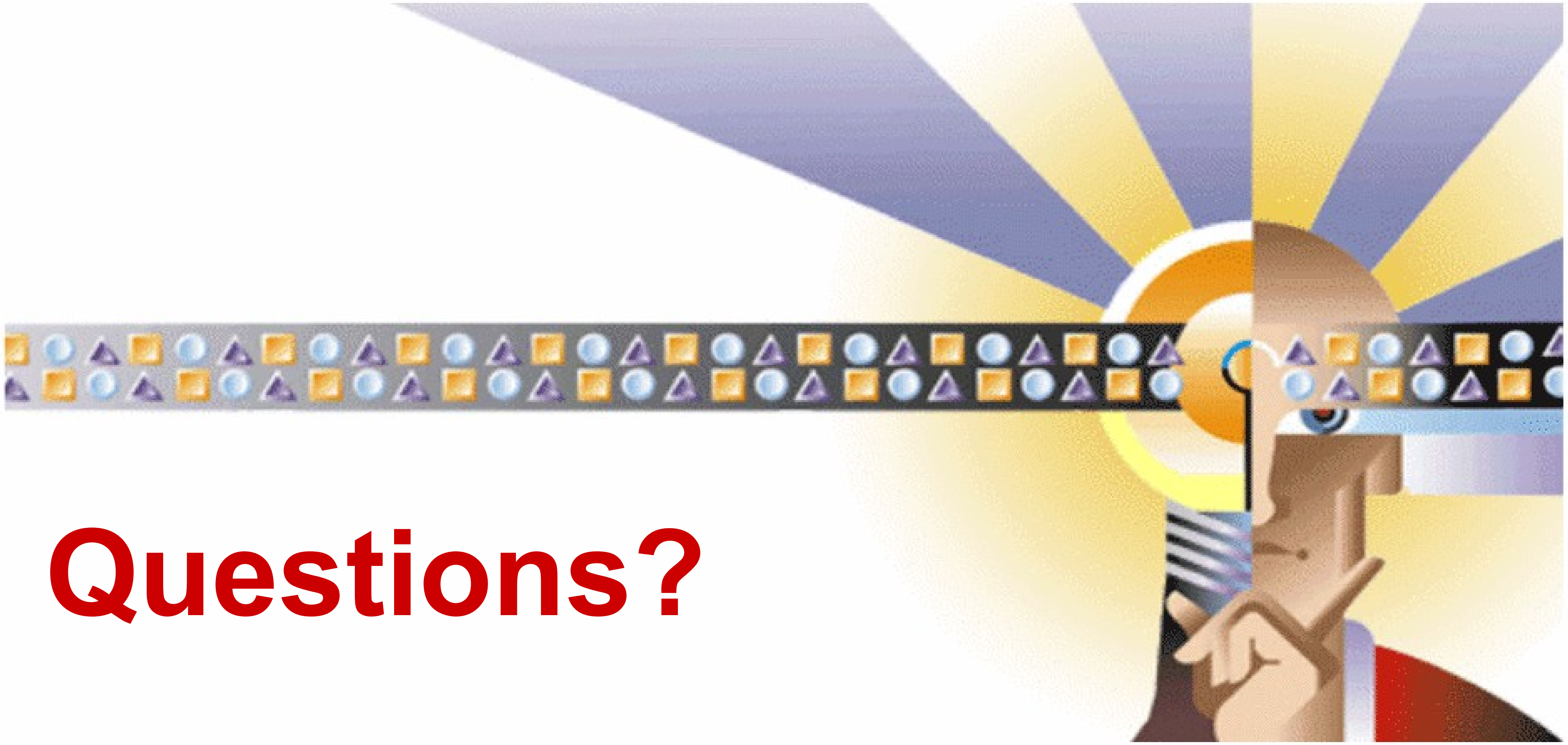
Execution Stack Layout



- Execution stacks are heap-allocated objects
    - Re-allocatable on overflow
    - Globally configurable growth direction
    - Each execution stack represents a virtual thread and belongs to a task
    - Compiled frame layout may differ from interpreted frame
- Stack frame layout is defined by the language
- On a language transition (if not inlined)
    - Create a transition frame
    - Save current_language_id in the known local variable of the transition frame
    - For just two languages it would not be necessary to save current_language_id
    - 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

Native VM Extension

- New language interpreter, compiler and its generated code must be trusted
    - A technique for VM developers, not for ordinary users
- 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?
    - Every target must implement this extension
- Native extension is not portable to any other VM



Questions?

microsystems

We make the net work.