JavaOne℠

Sun's 2002 Worldwide Java Developer Conference℠

Project Code-named 'Monty': A High-Performance Java™ Virtual Machine for Small Devices

Lars Bak
Kasper Verdich Lund
Jakob Roland Andersen
Demo by Kay Neuenhofen
Sun Microsystems, Inc.

Purpose of the Presentation

To present the design and implementation of the next generation high-performance Java™ virtual machine for the CLDC platform

Learning Objectives

As a result of this presentation, you will be able to:

- Learn why speed will be important in mobile devices
- Understand the key technologies behind a high-performance VM for memory constrained systems
- Anticipate upcoming performance improvements for CLDC

Why Is Speed Important in Mobile Devices?

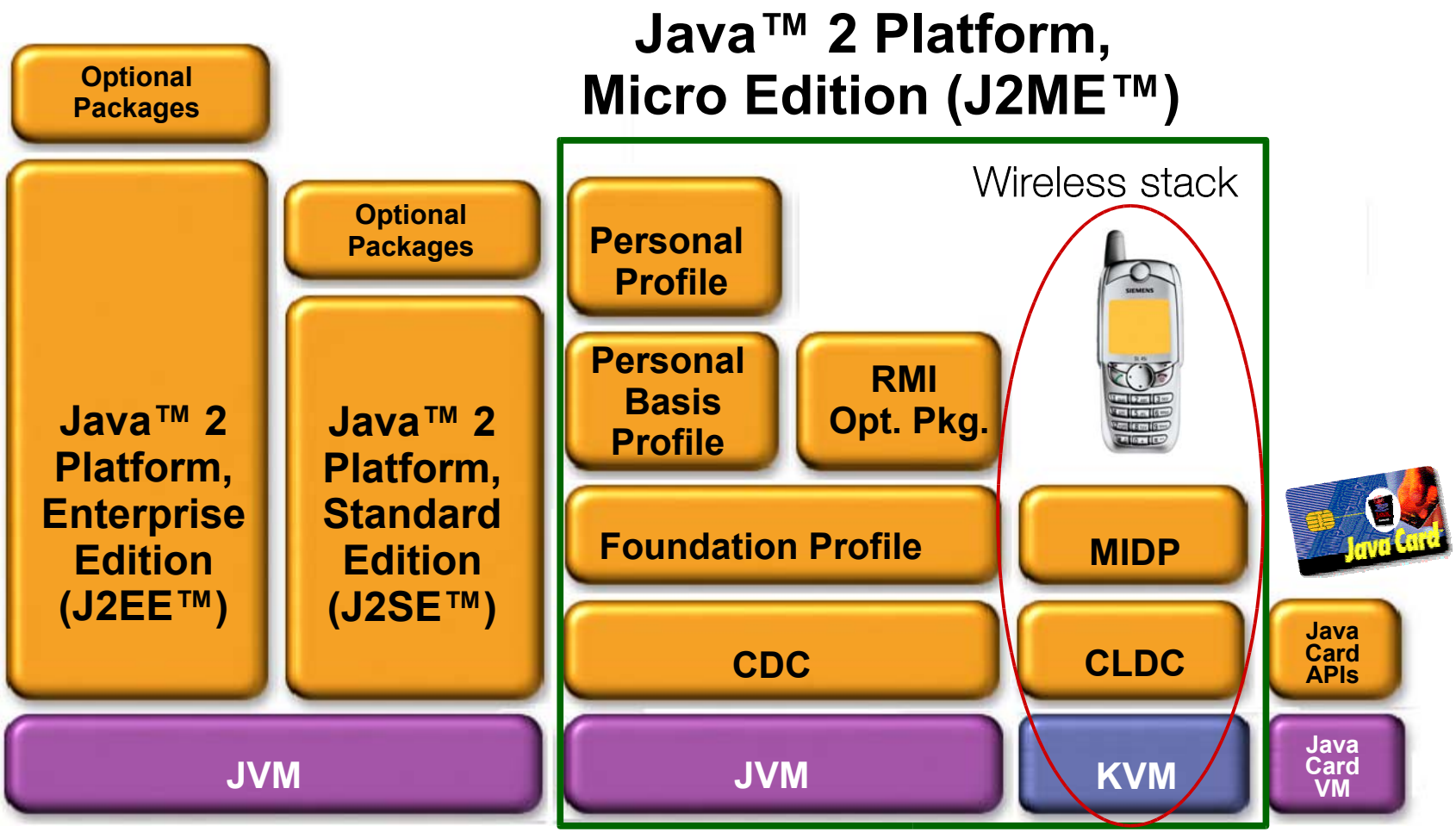
Bandwidth will approach 2Mbits/s

Battery drain is crucial

New compute-intensive application types

- E-commerce applications
- Banking applications
- System software
- Location-based services
- Multimedia applications
- 3D games and gambling applications

Java™ Platform



Java™ Virtual Machine Challenge

How do we design a high-performance Java virtual machine that preserves battery life?

- Fast byte code execution
- Low memory footprint
- Flexible memory system

Why Should You Listen to Us?

Lars Bak is a virtual machine architect at Sun Microsystems, Inc.

Lars Bak has 16 years of experience in designing and implementing object-oriented virtual machines:

Java HotSpot™, StrongTalk, Self, and Beta

Kasper Verdich Lund and Jakob Roland Andersen are graduate students at Aarhus University, Denmark

Presentation Outline

- The Project
- The Technology
  - Compact object layout
  - Fast synchronization
  - Explicit type tags
  - Dynamic compilation
  - Unified memory management
- Benchmark results
- Demo on iPaq

Project Monty

Focused on delivering the next generation VM for the CLDC platform

- Fast execution
- Low memory footprint
- Long-lived VM
- From scratch design

Goals

- 10 times the speed of KVM 1.0.3
- Small footprint

Where Is Performance Important?

Project Monty Technology Overview

- Clean 32-bit Java virtual machine
- Compliant with JLS and JVM™ specification
- Approaches Java™ technology desktop performance
- Precise generation-based garbage collection
- No restrictions on:
  - Number of loaded classes
  - Size of object heap
- Target memory for the Java platform: 1Mb
  - JVM + CLDC + MIDP + Applications

Project Monty Design Rules

Keep it simple, stupid!

Complexity results in

- Increased footprint
- Fragile implementation

No premature optimizations

Examples:

- Stack maps for activations
- Special cache for compiled code

Object Requirements in the Java™ Programming Language

Must carry reflective information

java.lang.Object.getClass()

Can be synchronization targets

synchronized (x) { ... }

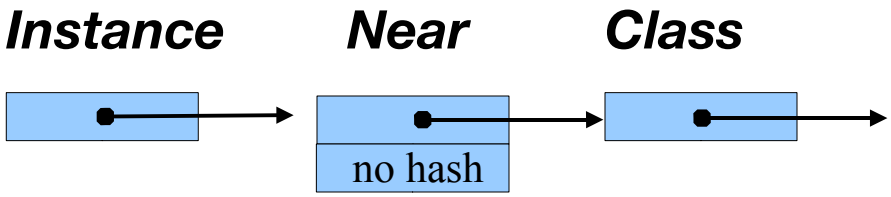
Support immutable hash code

java.lang.Object.hashCode()

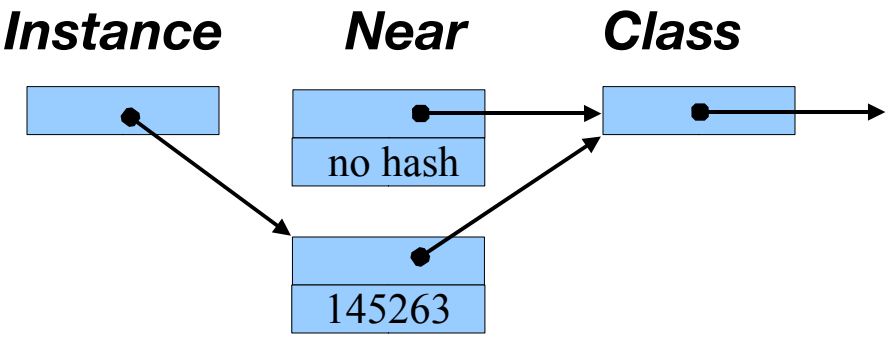
VM must provide default implementation

Compact Object Layout

1 word object header (class pointer)



Near is copied if hash code is assigned



Compact Object Layout

- Only few objects get hash code assigned
- Saves a word for most objects
  - Occupies less space
  - Speeds up allocation

Object Sizes

Sample “Java classes”

```
class Number {
    int value;
}
```

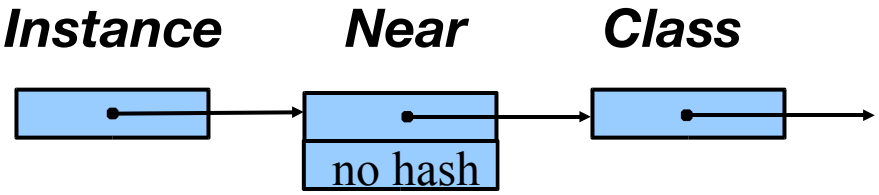
```
class Node {
  byte    code;
  Node    next;
  short   index;
  boolean hasIndex;
}
```

Byte sizes

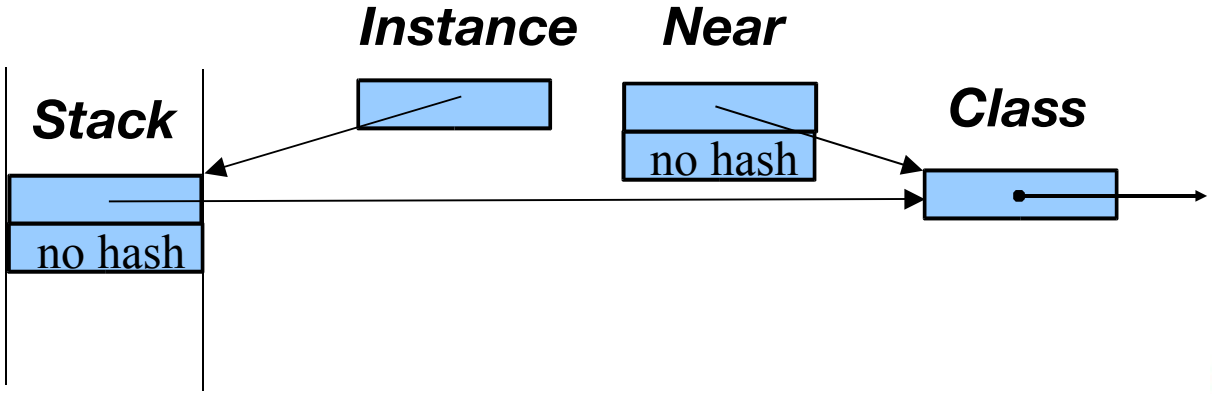
VM	Classic	KVM	HotSpot	CVM	Monty
Number Class	16	16	16	12	8
Node Class	32	28	24	24	12

Object Synchronization

Object locking



Object unlocking



Object Synchronization

Applied block-structured locking as in the Java HotSpot™ virtual machine

No space is needed in the object

Used the location of the near object to indicate locking

Pointers on the Execution Stack

Precise garbage collection requires the system to locate all pointers on the stack

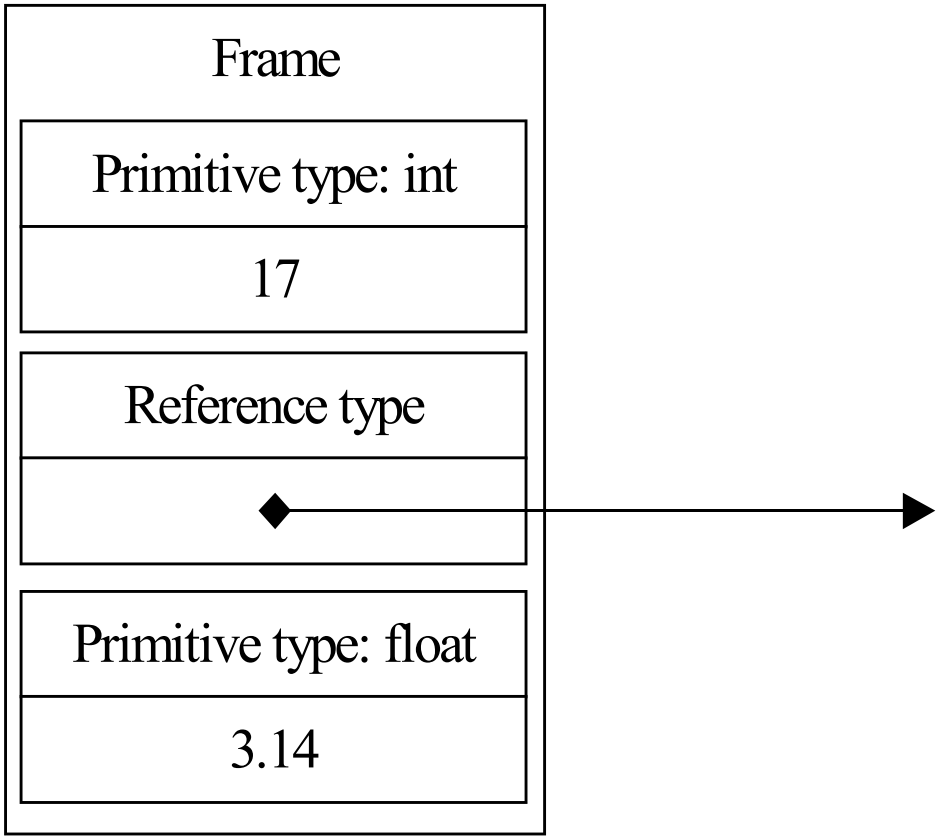
- ∀ `active execution frames`
  - ∀ `locals and expression stack elements`
    `is this an object pointer?`

Two approaches to solve this problem

- Abstract interpretation of bytecodes
- Carry explicit tags with bytecode execution

Explicit Type Tagging

Value on stack has associated tag



Explicit Type Tagging

Eliminates the need for stack maps

- KVM, 5-10% of reflective data

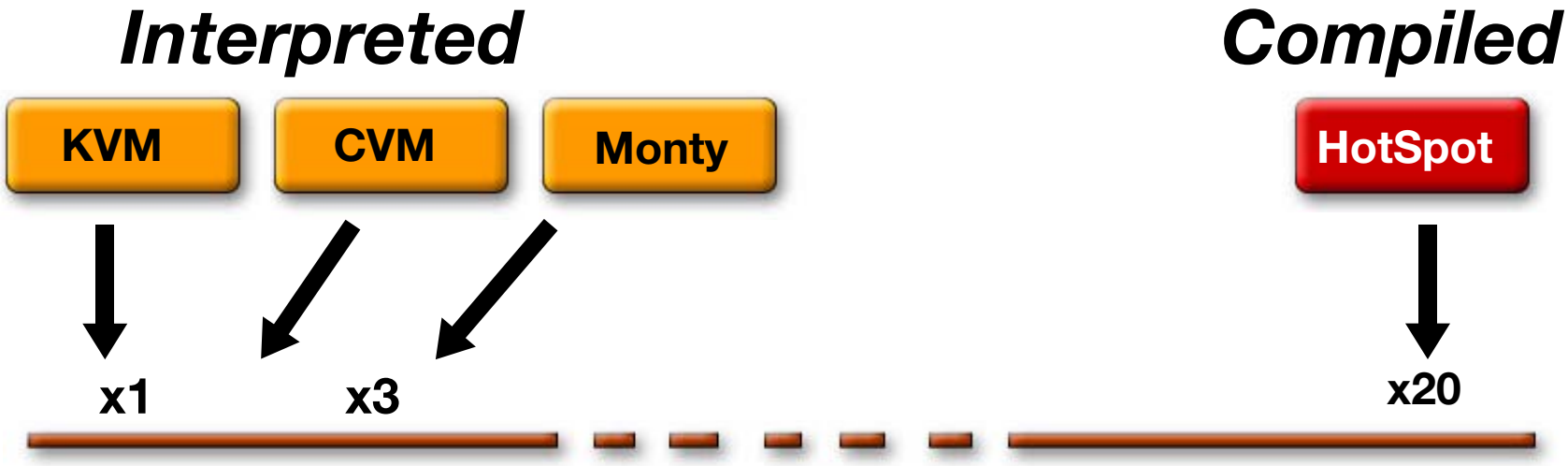
J2SE™ uses abstract interpretation

Makes scanning for references trivial

Doubles the needed stack space!

Is a Dynamic Compiler Needed?

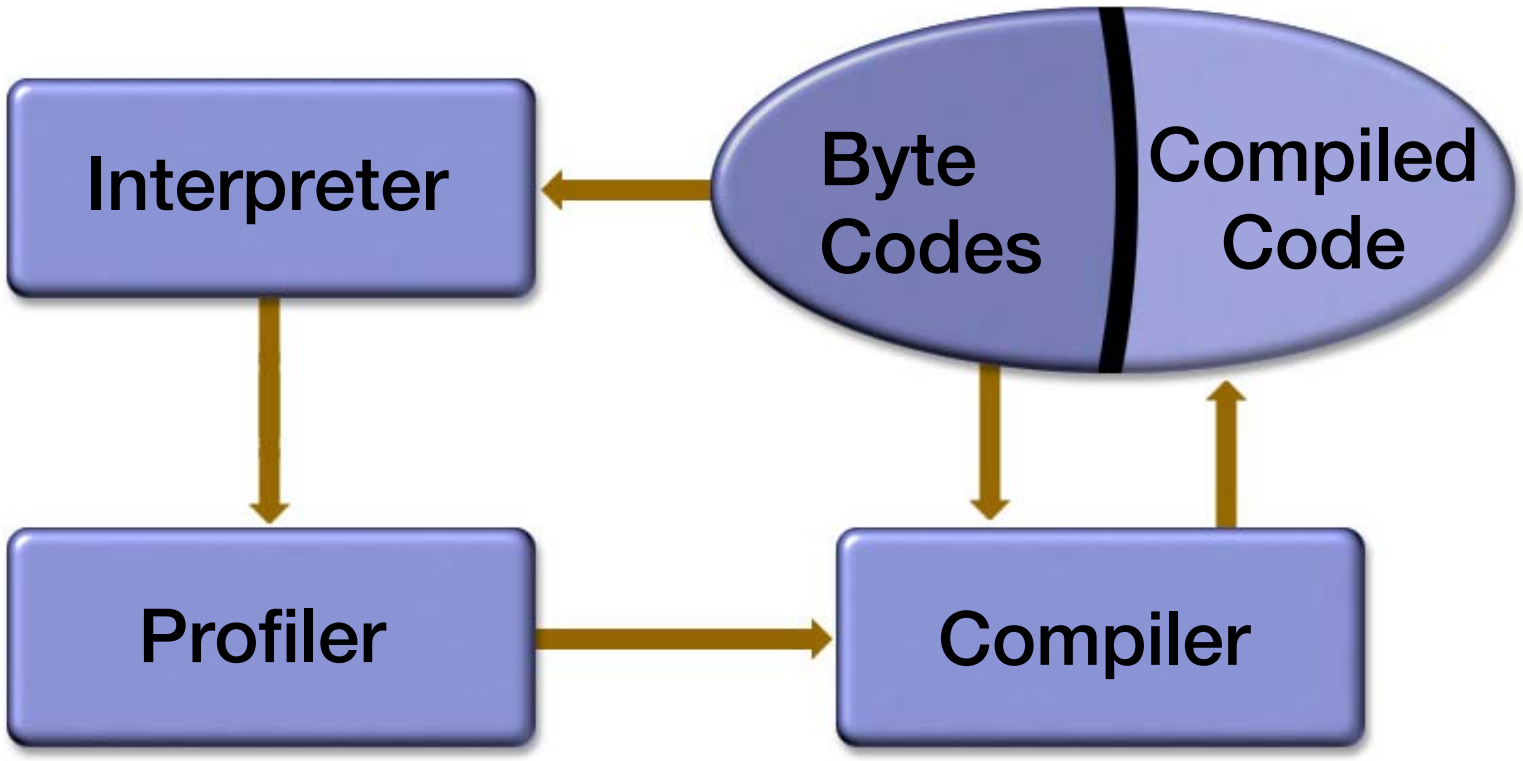
On average, interpreted VM performance is x10 slower than compiled VM performance



Compilation Strategy

- Compiled code occupies 4 to 5 times more memory than byte codes
- The solution is adaptive compilation
  - Only compile the methods that are hot spots
  - Make the interpreter as fast as possible
- Fast and simple one-pass compiler

Adaptive Compilation Strategy



Speeding Up the Interpreter

Threaded assembly interpreter with static dispatch table

Interpreter is generated

Each bytecode template is described as an ordered set of instructions

Optimized by brute force instruction reordering

Is this really necessary?

Resource Management

Traditional approach

- Segment for user objects
- Segment for reflective data
- Segment for code cache

… results in

- Fragmentation
- Complexity
- Static segmentation

Unified Resource Management

The object heap contains all data

- Allocated objects
- Loaded classes
- VM internal data structures
- Compiled code

… everything is subject to garbage collection

Benefits of Unified Resource Management

No fragmentation at all

Compiled code can be removed to give space for application objects

More robust, smaller, and simpler memory management code

Compiled Code Removal

Least recently used (LRU) algorithm used for ranking compiled code for removal

Garbage collector determines how much code should be removed and removes the least recently used compiled code

Deoptimization is applied to frames described by compiled code to be removed

Battery Life and Cache Behavior

- Faster execution consumes less power
- Cache friendly design
  - Small objects
  - Generational garbage collector, touches memory locally
  - Compiled code in object heap, fully relocatable/flushable

Benchmarks Results

Selected Benchmarks

Richards	Simulates dispatch kernel of an operating system
DeltaBlue	Solves one-way constraint systems
Pentominoes	Mathematical puzzle benchmark
KXML	XML parser benchmark
BenchPress	Stanford integer benchmarks

Platforms

iPaq: 206Mhz StrongARM/WinCE

PC: 1.4Ghz P4/Windows2000

Benchmark Results

	Strong ARM/Win			x86/Win32		
	KVM 1.0.3	Monty	Ratio	KVM 1.0.3	Monty	Ratio
KXML	166.1	32.6	5.1	15.5	1.6	9.9
Richards	340.8	33.6	10.2	45.2	5.6	8.0
DeltaBlue	606.1	98.4	6.2	46.3	5.5	8.3
Pentominoes	938.6	117.2	8.0	170.1	11.4	14.9
Benchpress	1841.2	205.6	9.0	361.7	32.6	11.1
		Geomean	7.1		Geomean	10.0

* All numbers are seconds

Demo

Conclusion

Project Monty is a new high-performance Java virtual machine for the Java™ platform for CLDC with:

- Compact object layout
- Fast synchronization
- Explicit type tags
- Dynamic compilation
- Unified memory management

It provides high performance and small memory footprint

If You Only Remember One Thing…

Anticipate an order of magnitude performance boost for CLDC

How can **you** use the extra juice?

Q&A

JavaOne℠

Sun's 2002 Worldwide Java Developer Conference™

BEYOND BOUNDARIES

Session 2133