picoJava™: A Hardware Implementation of the Java Virtual Machine

Marc Tremblay and Michael O'Connor

Sun Microelectronics

The Java – picoJava Synergy

- **Java's origins lie in improving the consumer embedded market**
- **picoJava is a low cost microprocessor dedicated to executing Java™-based bytecodes**
  - –Best system price/performance
- **It is a processor core for:**
  - –Network computer
  - –Internet chip for network appliances
  - –Cellular phone & telco processors
  - –Traditional embedded applications

Sun microsystems

Java in Embedded Devices

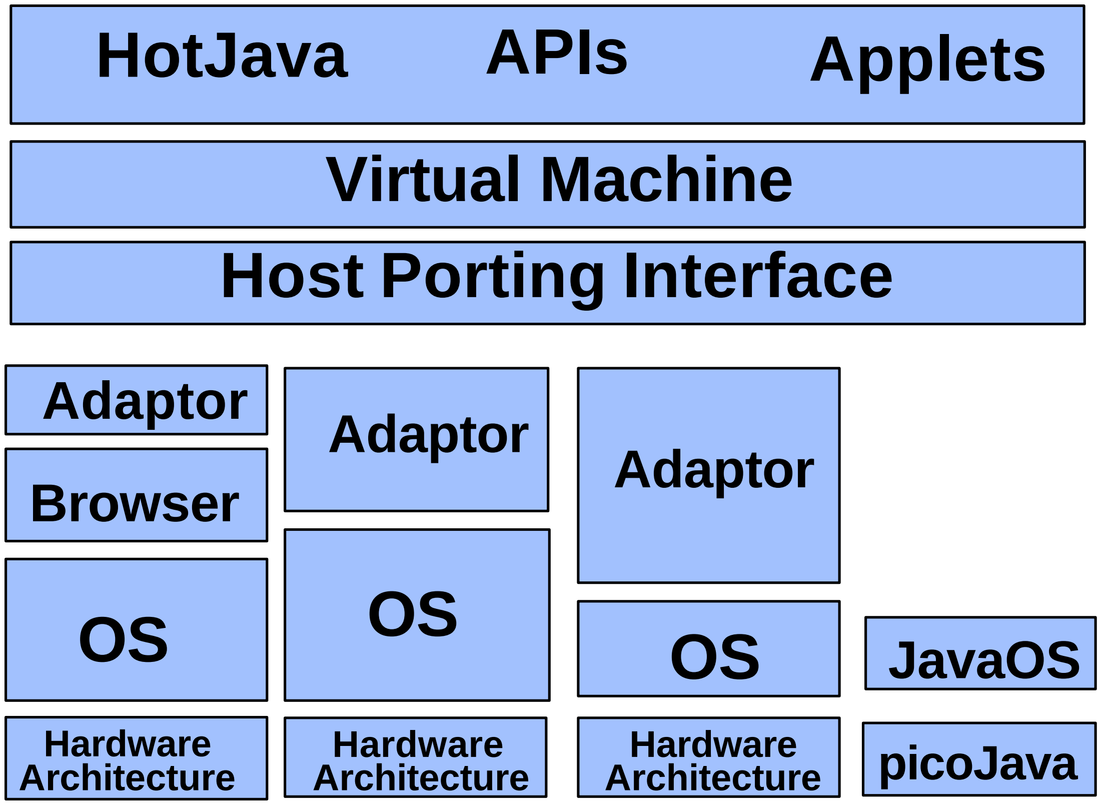
Products in the embedded market require:

- **Robust programs**
  - –Graceful recovery vs. crash
- **Increasingly complex programs with multiple programmers**
  - –Object-oriented language and development environment
- **Re-using code from one product generation to the next**
  - –Portable code
- **Safe connectivity to applets**
  - –For networked devices (PDA, pagers, cell phones)

Important Factors to Consider in the Embedded World

- **Low system cost**
  - Processor, ROM, DRAM, etc.
- **Good performance**
- **Time-to-market**
- **Low power consumption**

Various Ways of Implementing the Java Virtual Machine



picoJava

- **Directly executes bytecodes**
  - –Excellent performance
  - –Eliminates the need for an interpreter or a JIT compiler
  - –Small memory footprint
- **Simple core**
  - –Legacy blocks and circuits are not present
- **Hardware support for the runtime**
  - –Addresses overall system performance

Java Virtual Machine

- **What the virtual machine specifies:**
  - –Instruction set
  - –Data types
  - –Operand stack
  - –Constant pool
  - –Method area
  - –Heap for runtime data
  - –Format of the class file

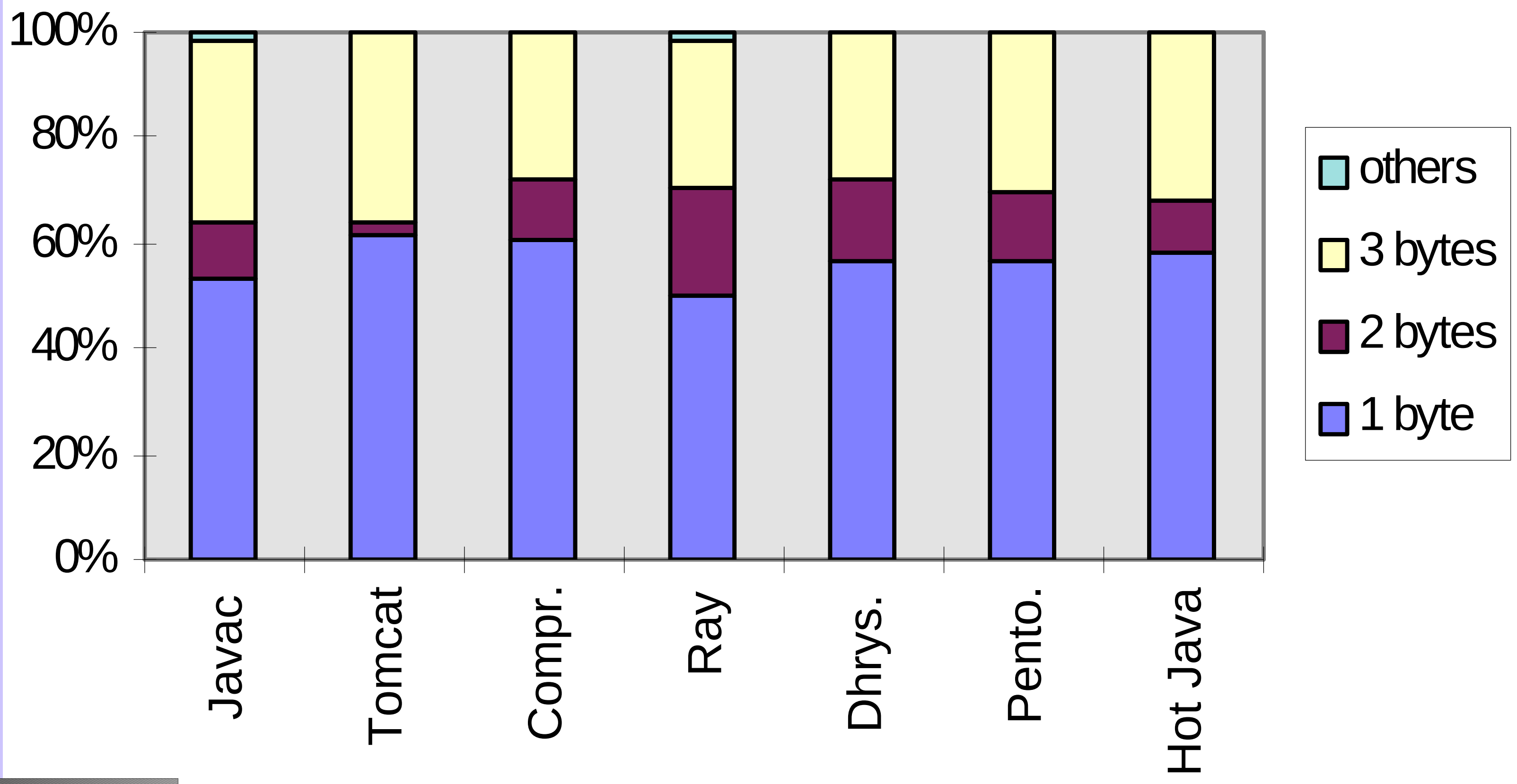
Virtual Machine —Instruction Set

- Data types: byte, short, int, long float, double, char, object, returnAddress
- All opcodes have 8 bits, but are followed by a variable number of operands (0, 1, 2, 3, …)
- Opcodes
  - –200 assigned
  - –25 quick variations
  - –3 reserved

Java Virtual Machine Code Size

- **Java™-based bytecodes are small**
  - –No register specifiers
  - –Local variable accessed relative to a base pointer (VARS)
- **This results in very compact code**
  - –Average JVM instruction is 1.8 bytes
  - –RISC instructions typically require 4 bytes

Instruction Length



Java Virtual Machine Code Size

- Java bytecodes are about 2X smaller than the RISC code from the C++ compiler
- A large application (2500+lines) coded in both the C++ and Java languages

JVM – Instruction Set – RISCy

- Some instructions are simple

```
bipush value      :push signed integer
iadd              :integer add
fadd              :single float add
ifeq              :branch if equal to 0
iload offset      :load integer from
                  :local variable
```

JVM – Instruction Set – CISCy

- **Some instructions are complex**

`lookupswitch`: “traditional” switch statement

byte 1	byte 2	byte 3	byte 4
opcode (171)	0..3 byte padding		
default offset			
numbers of pairs that follow (N)			
match 1			
jump offset 1			
match 2			
jump offset 2			
...			
...			
match N			
jump offset N			

Interpreter Loop

```
loop: 1: fetch bytecodes
      2: indirect jump to
         emulation code      ──────►
```

Emulation Code

```
1: get operands
2: perform
    operation
3: increment PC
4: go to loop
```

JVM: Stack-Based Architecture

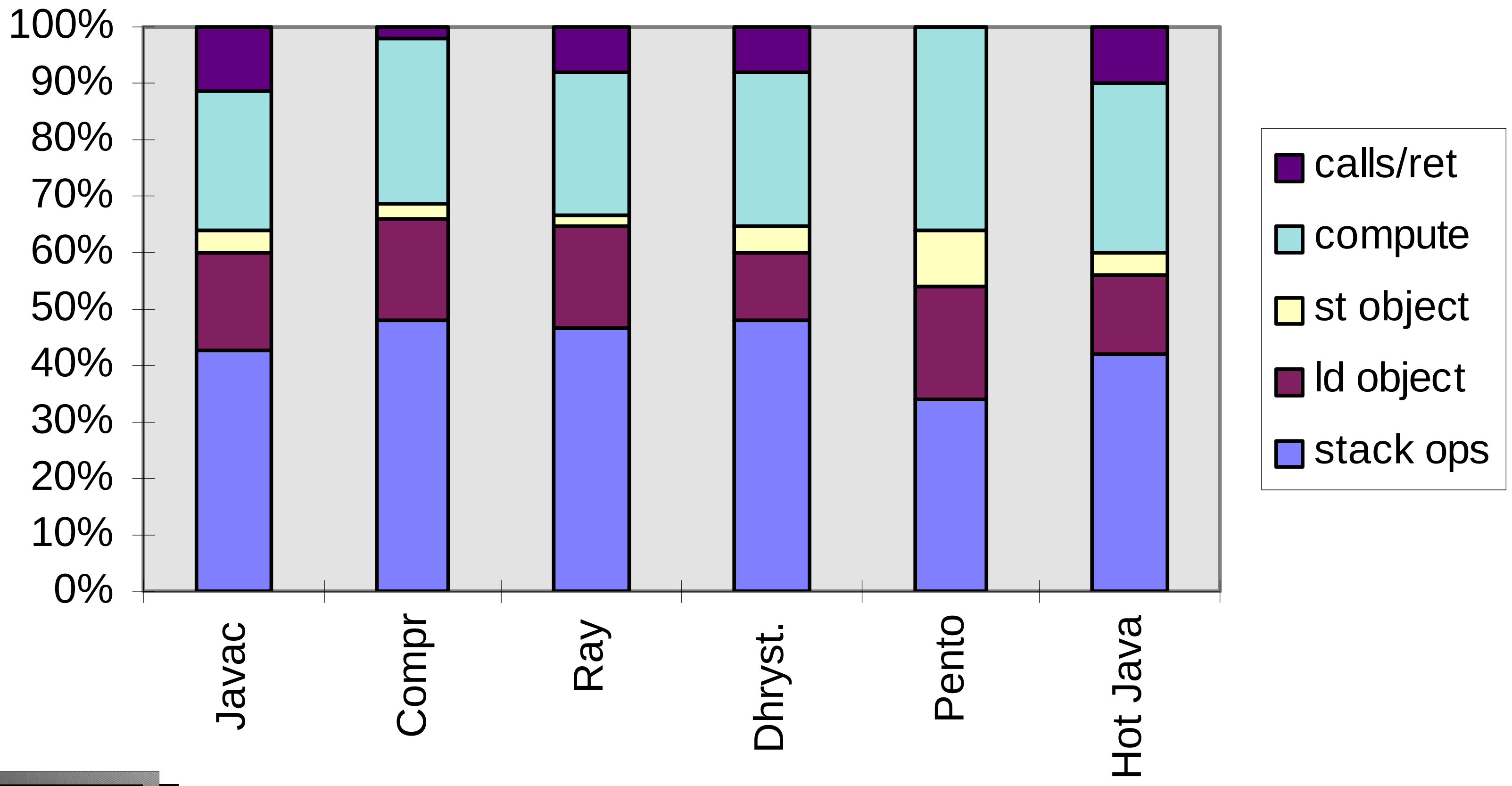
- **Operands typically accessed from the stack, put back on the stack**
- **Example — integer add:**
  - –Add top 2 entries in the stack and put the result on top of the stack
  - –Typical emulation on a RISC processor

```
1: load tos
2: load tos-1
3: add
4: store tos-1
```

How to Best Execute Bytecodes?

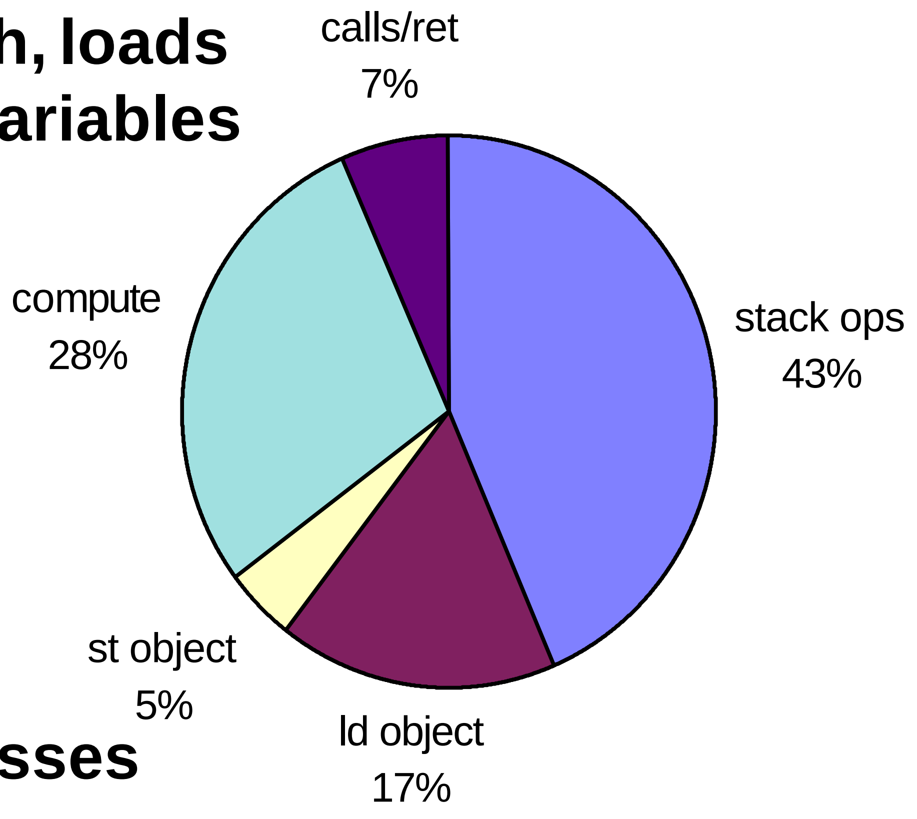
- **Leverage RISC techniques developed over the past 15 years**
- **Implement in hardware only those instructions that make a difference**
  - –Trap for costly instructions that do not occur often
  - –State machines for high frequency/medium complexity instructions

Dynamic Instruction Distribution

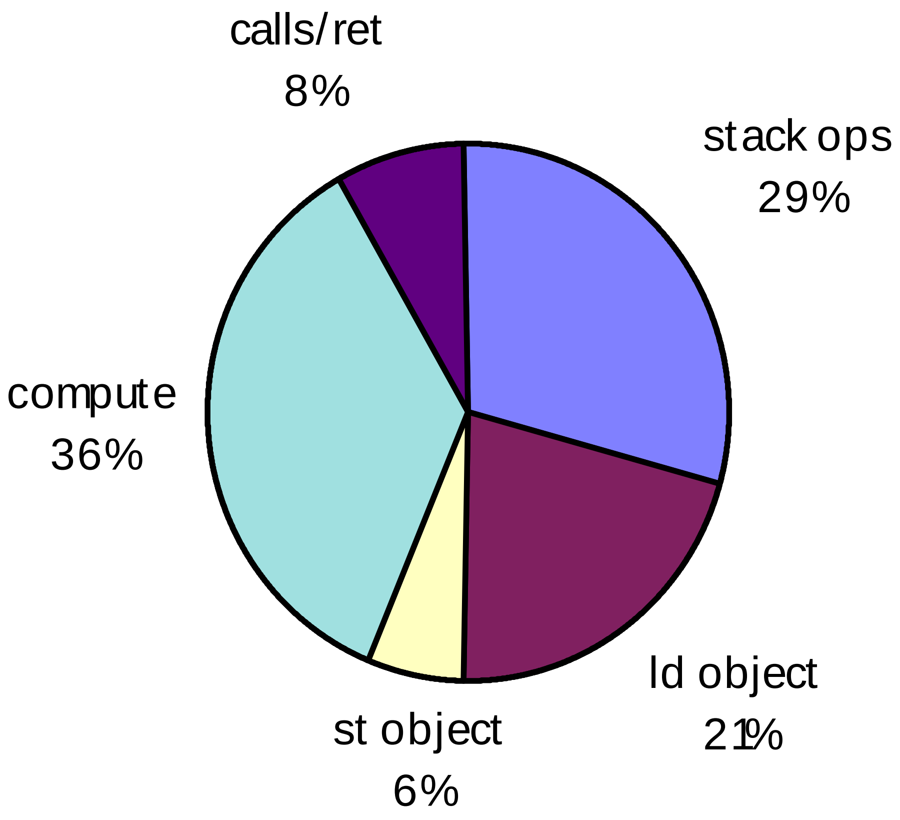


Composite Instruction Mix

- **Stack ops: dup, push, loads and stores to local variables**
- **compute: ALU, FP, compute branches**
- **calls/ret: method invocation virtual and non-virtual**
- **ld/st object: access to objects on the heap and array accesses**



Loads from Local Variables



- Loads from local variables move data within the chip
- Target register is often consume immediately
- Up to 60% of them can be hidden
- Resulting instruction distribution looks closer to a RISC processor

Pipeline Design

- **RISC pipeline attributes**
  - Stages based on fundamental paths (e.g. cache access, ALU path, registers access)
  - No operation on cache/memory data
  - Hardwire all simple operations
- **Enhance classic pipeline**
  - Support for method invocations
  - Support for hiding loads from local variables

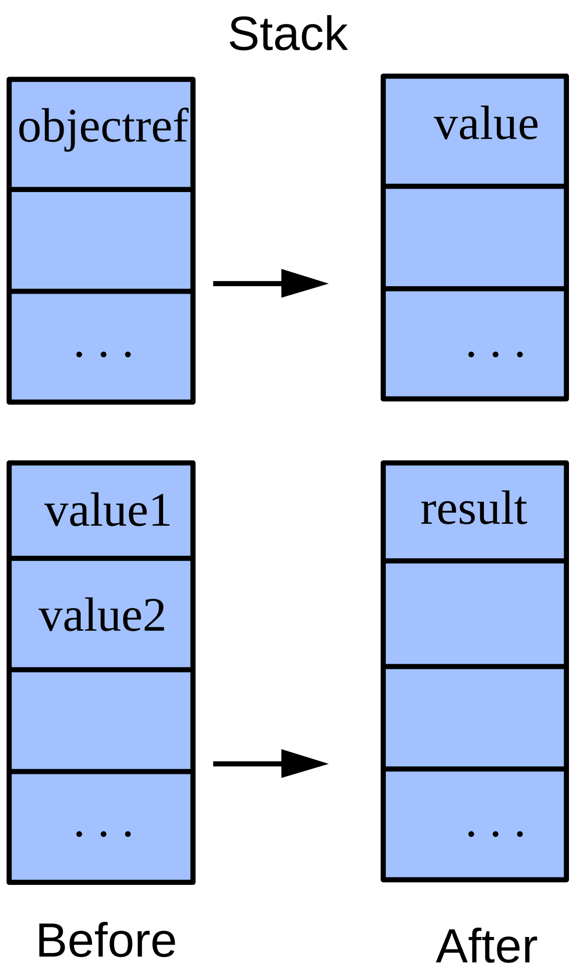
Implementation of Critical Instructions

`getfield_quick offset`

– Fetch field from object
– Executes as a “load [object + offset]” on picoJava

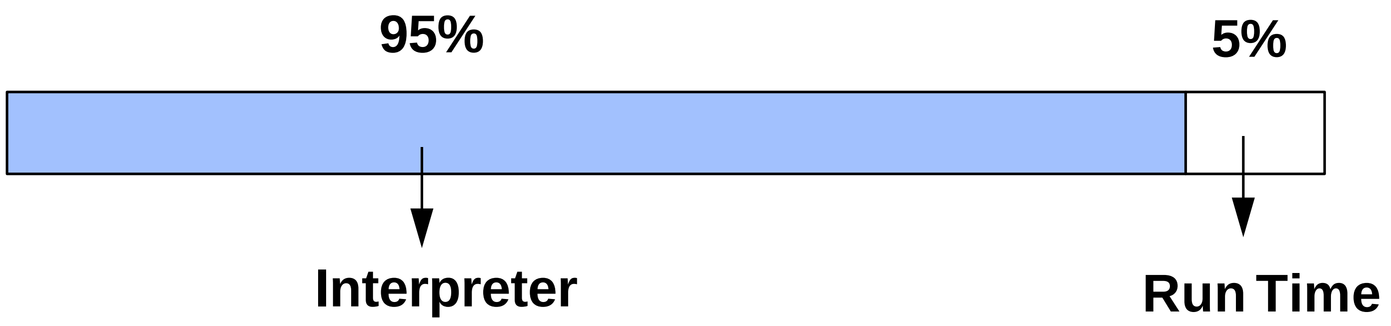
`iadd`

– Fully pipelined
– Executes in a single cycle



Typical Small Benchmarks (Caffeinemarks, Pentonimo, etc.)

- Few objects, few calls, few threads



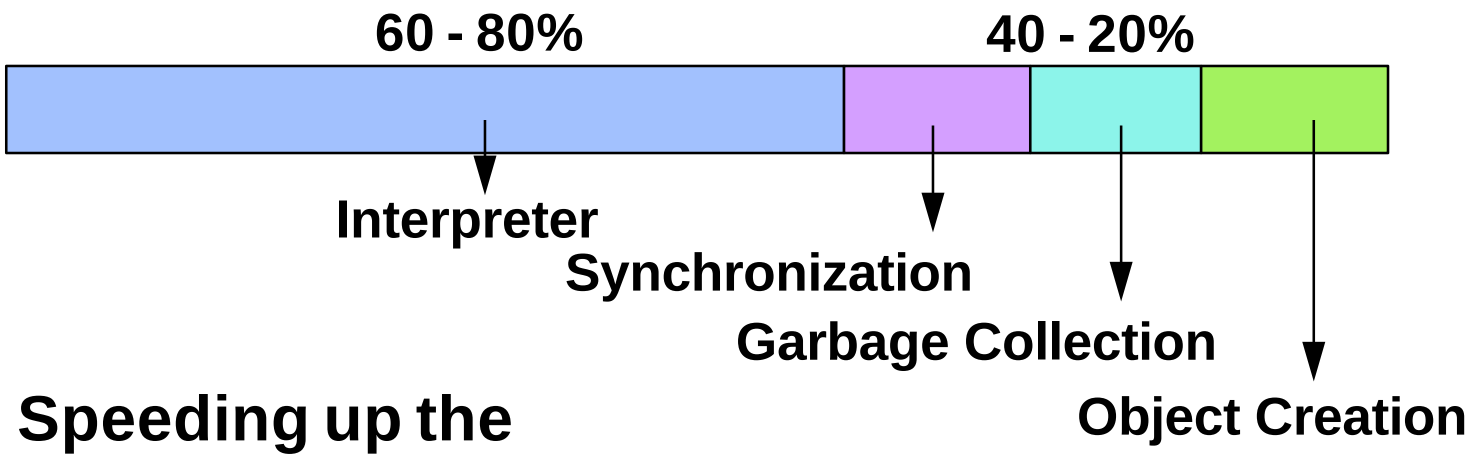
Speeding up the Interpreter by 30X results in:

$$\begin{array}{rcr} 95 & \longrightarrow & 3.2 \\ 5 & \longrightarrow & \underline{\;5\;} \\ & & 8.2 \end{array}$$

=> Speedup of ~12X

Representative Applications

- Lots of Objects
- Threaded Code

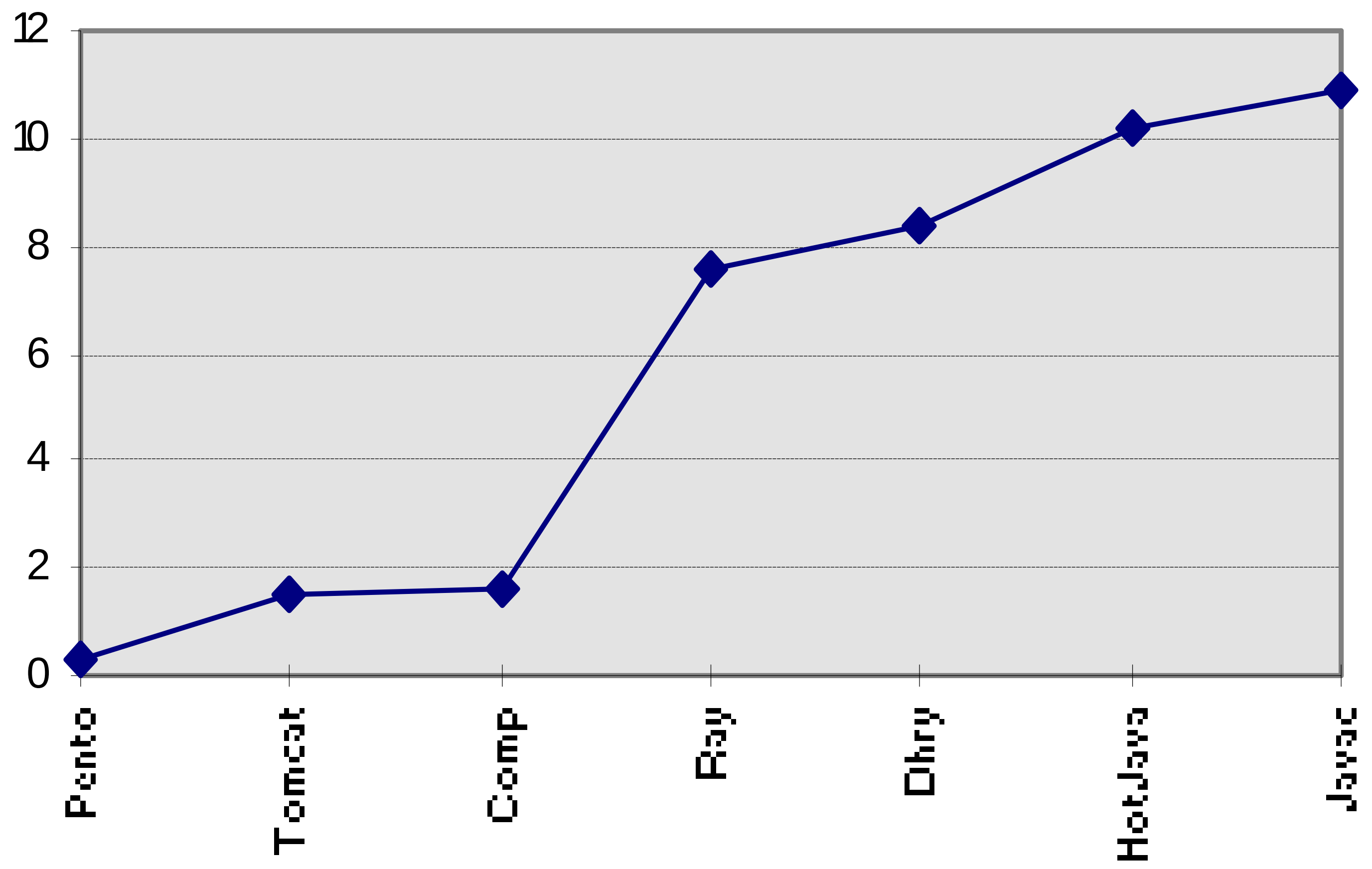


Speeding up the Interpreter by 30X results in:

$$\begin{array}{rcr} 60 & \rightarrow & 2 \\ 40 & \rightarrow & 40 \\ \hline & & 42 \end{array}$$

=> Speedup of ~2X

Percentage of Calls



Varies dramatically according to benchmark type

picoJava: A System Performance Approach

- **Accelerates object-oriented programs**
  - simple pipeline with enhancements for features specific to bytecodes
  - support for method invocation
- **Accelerates runtime**

  (gc.c, monitor.c, threadruntime.c, etc.)
  - Support for threads
  - Support for garbage collection
- **Simple but efficient, non-invasive, hardware support**

System Programming

- Instructions added to support system programming
  - available only “under the hood”
  - operating system functions
  - access to I/O devices
  - access to the internals of picoJava

picoJava - Summary

Best system price/performance for running Java™-powered applications in embedded markets

- **Embedded market very sensitive to system cost and power consumption**
- **Interpreter and/or JIT compiler eliminated**
- **Excellent *system* performance**
- **Efficient implementation through use of the same methodology, process and circuit techniques developed for RISC processors**