
Squawk: A Java™ Virtual Machine for Small (and Larger) Devices

Cristina Cifuentes

**Joint work with Bill Bush, Doug Simon,
Nik Shaylor**

Sun Microsystems Laboratories

IFIP WG2.4, Jan 3-7, 2005



Overview

- Problems with Current Virtual Machines
 - Squawk Principles
 - The Squawk Virtual Machine
 - Compilation Technology
 - Some Results
 - Applications
-
- Work in Progress



The Problem

- Most virtual machines are
 - Complex
 - Hard to understand
 - Hard to modify
 - Hard to maintain and
 - Hard to port

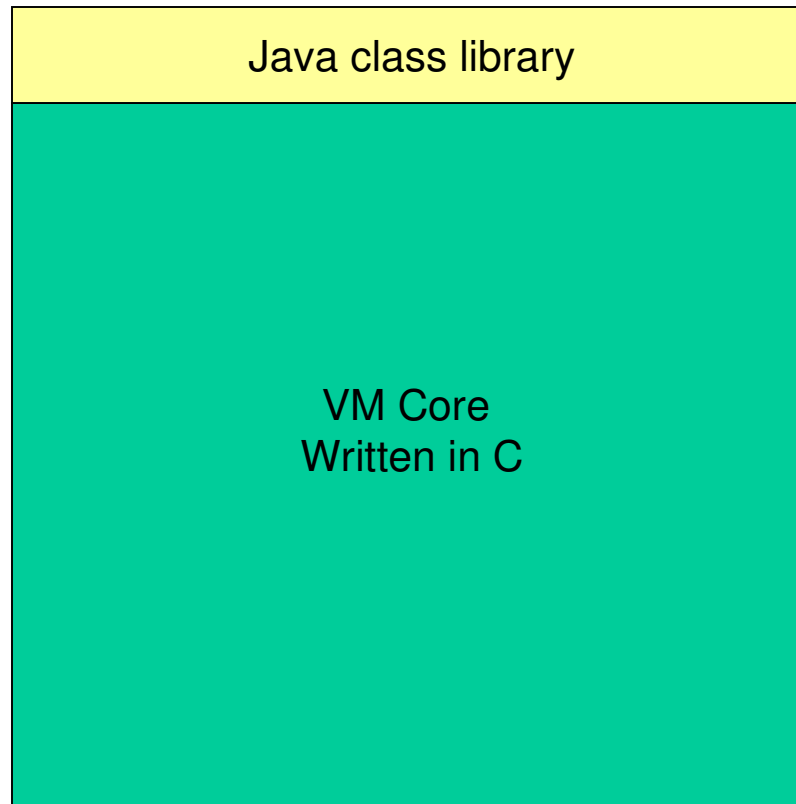


The Reason

- **Architectural failures**
 - Interaction between compiled, interpreted, and native code
 - Inadequate modularization
- **C/C++ are not good languages for virtual machine construction**
 - C is too low level a language
 - C has no provision for garbage collection



The Traditional Virtual Machine



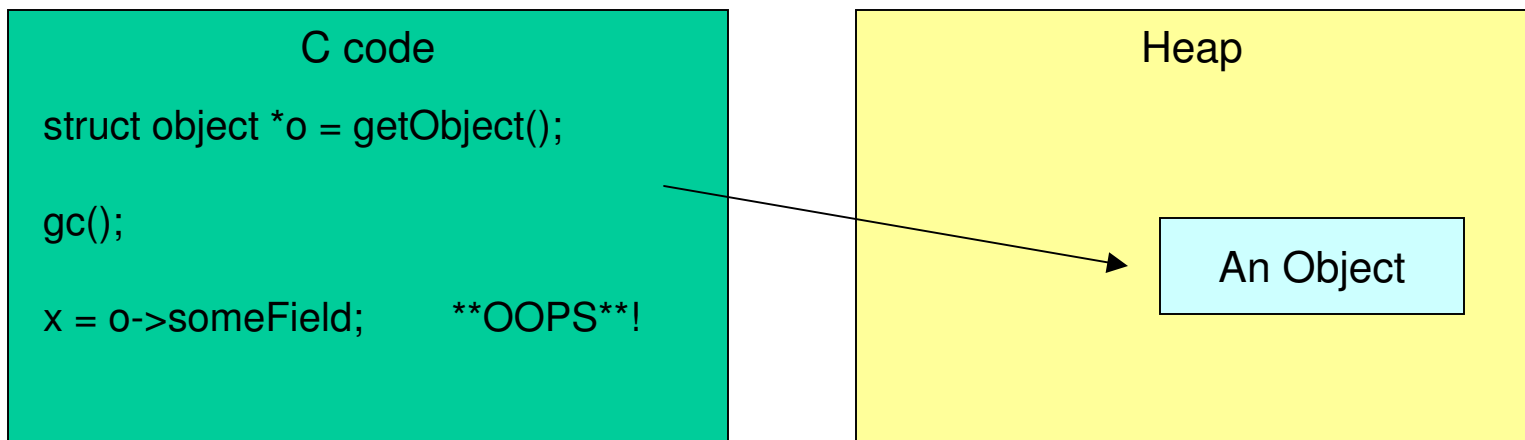
Most of the VM is written
In C or C++.

There is little that these
languages have to offer
VM construction.



The Pointer Problem

There is no support in the C language for compacting garbage collectors.



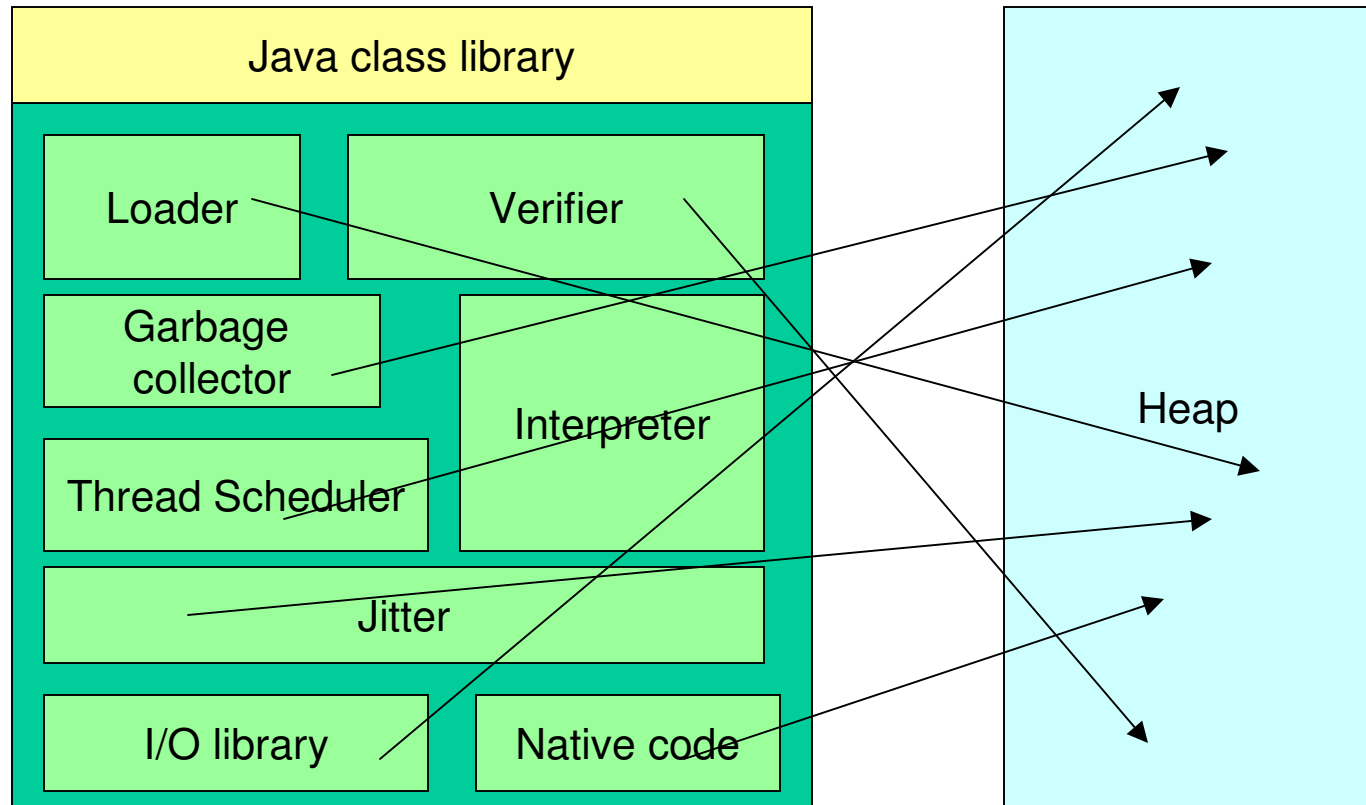
Calling the garbage collector moved the object and the local variable that pointed to it in the C code was not updated.

Conclusion: C and compacting garbage collectors do not work well together.



The Pointer Problem

Somehow all these pointers must be updated by the garbage collector



Squawk Principles

- A Java VM all written in Java.
- A modular kit of parts.
- Simplicity is a goal.
- Performance is important.



Most Simple



Squawk



Maximum
Performance



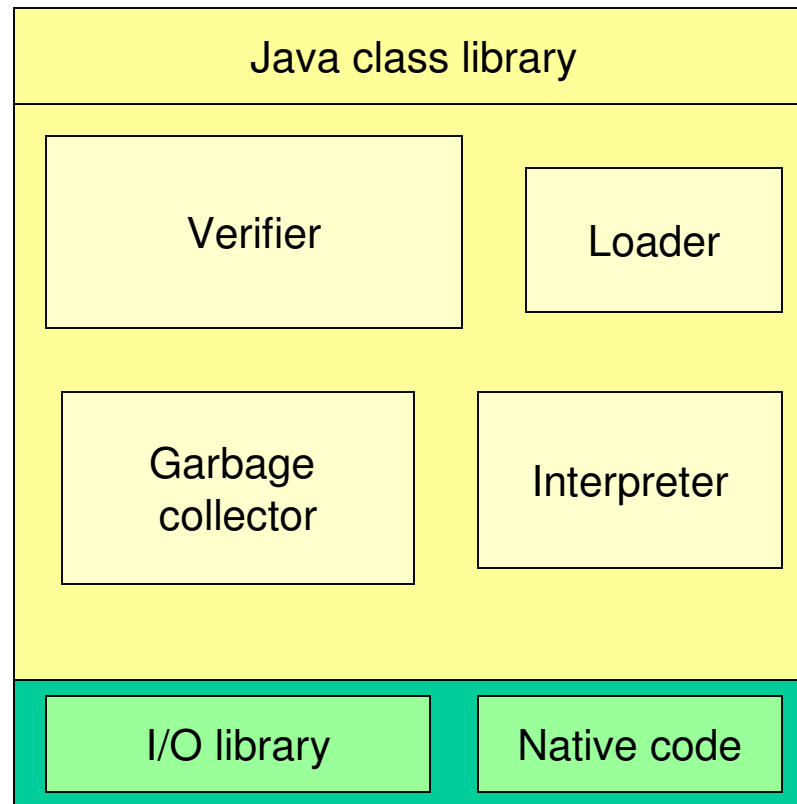
Sun Microsystems Laboratories

#8

Squawk Technology



Squawk for Java Card™



Squawk for Java Card Requirements

- Next generation smart card
- 32-bit processor
- 8 Kb RAM
- 32 Kb non-volatile memory (NVM, EEPROM)
- 160 Kb ROM



Squawk Features

- **CLDC compliant**
 - Dynamic class loading
 - Verification
 - Exact garbage collection
 - CLDC 1.0 Java APIs
- **Suites**
 - Off-device preprocessing to package classes into a smaller representation
- **Tri-partite memory**
 - RAM->ROM->NVM
 - Chunky stacks (allocated in the RAM heap)

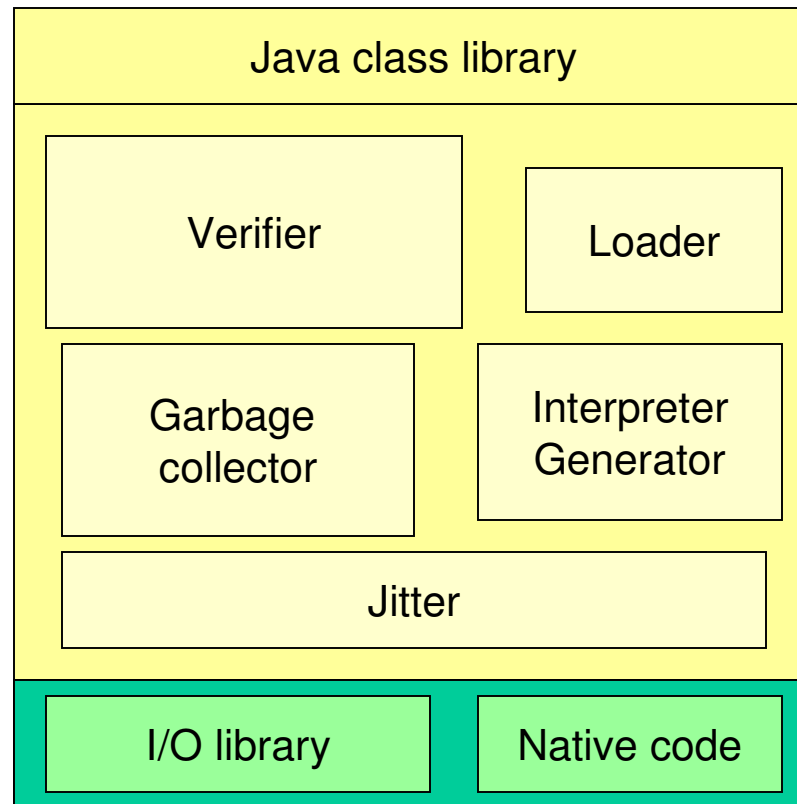


Squawk for Java Card Results (LCTES'03)

- Java->C interpreter, GC
- Java CLDC compliant
 - Passes 4537 of 4628 TCK tests
- Static footprint
 - 25 Kb (interpreter + 2 GCs (RAM and NVM)) on x86
- Minimum runtime footprint in RAM
 - 520 bytes for the Java heap
 - 532 bytes for the native stack and data (x86)
- Suites
 - 38% the size of JAR files
- Runtime performance
 - ~KVM (84-107% on benchmarks)



Squawk for Larger Systems

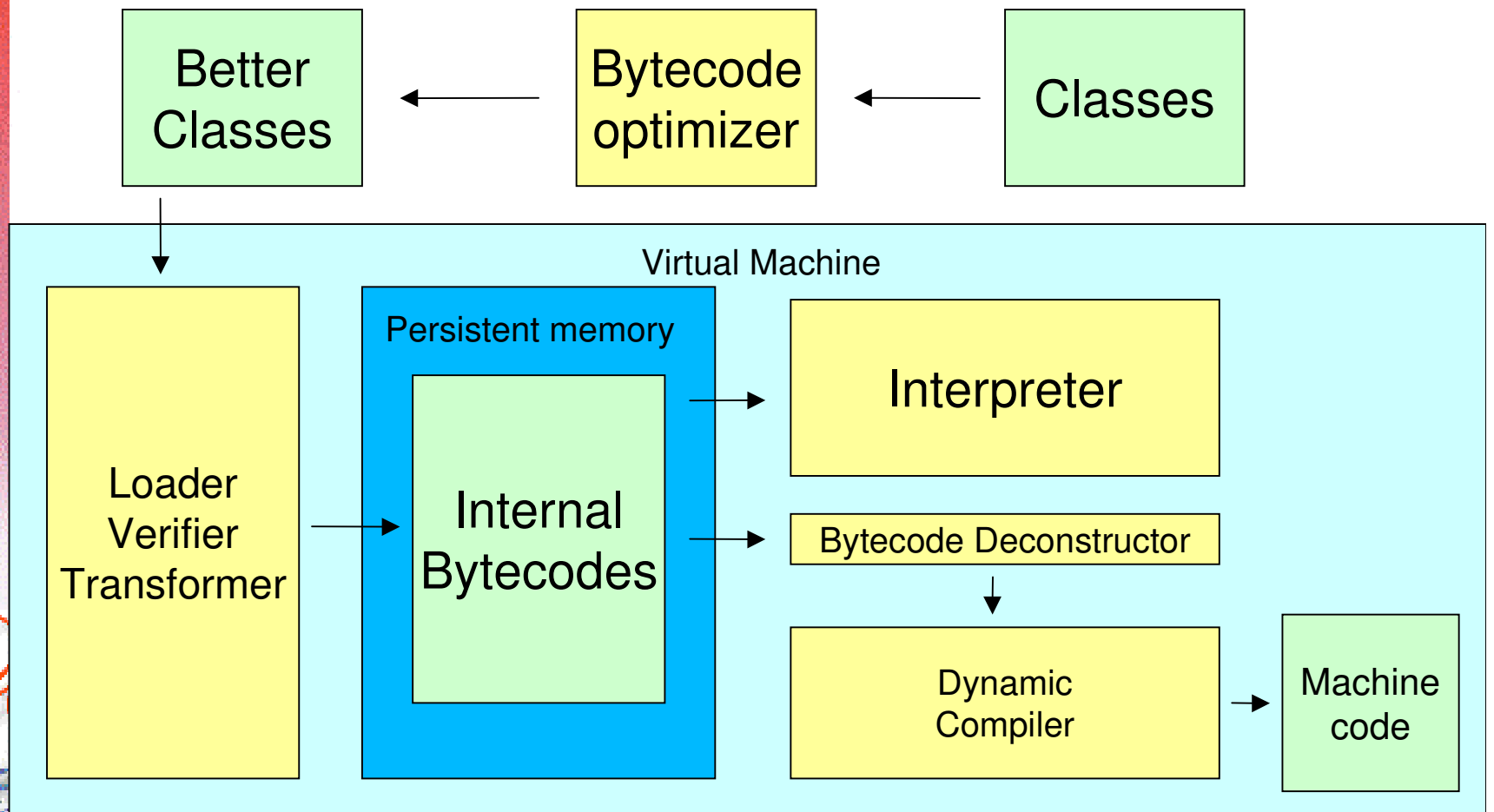


Squawk for Larger Systems - New Features

- Ahead-of-time bytecode optimization of system classes
- System built around a single compiler that can be used dynamically and ahead-of-time
 - AOT compilation of system classes
 - Dynamic compilation of user applications
- Isolates
 - Separate address spaces for multiple applications running on the one VM, with security guarantees
- Soft-real time support
- Everything written in Java



System Overview

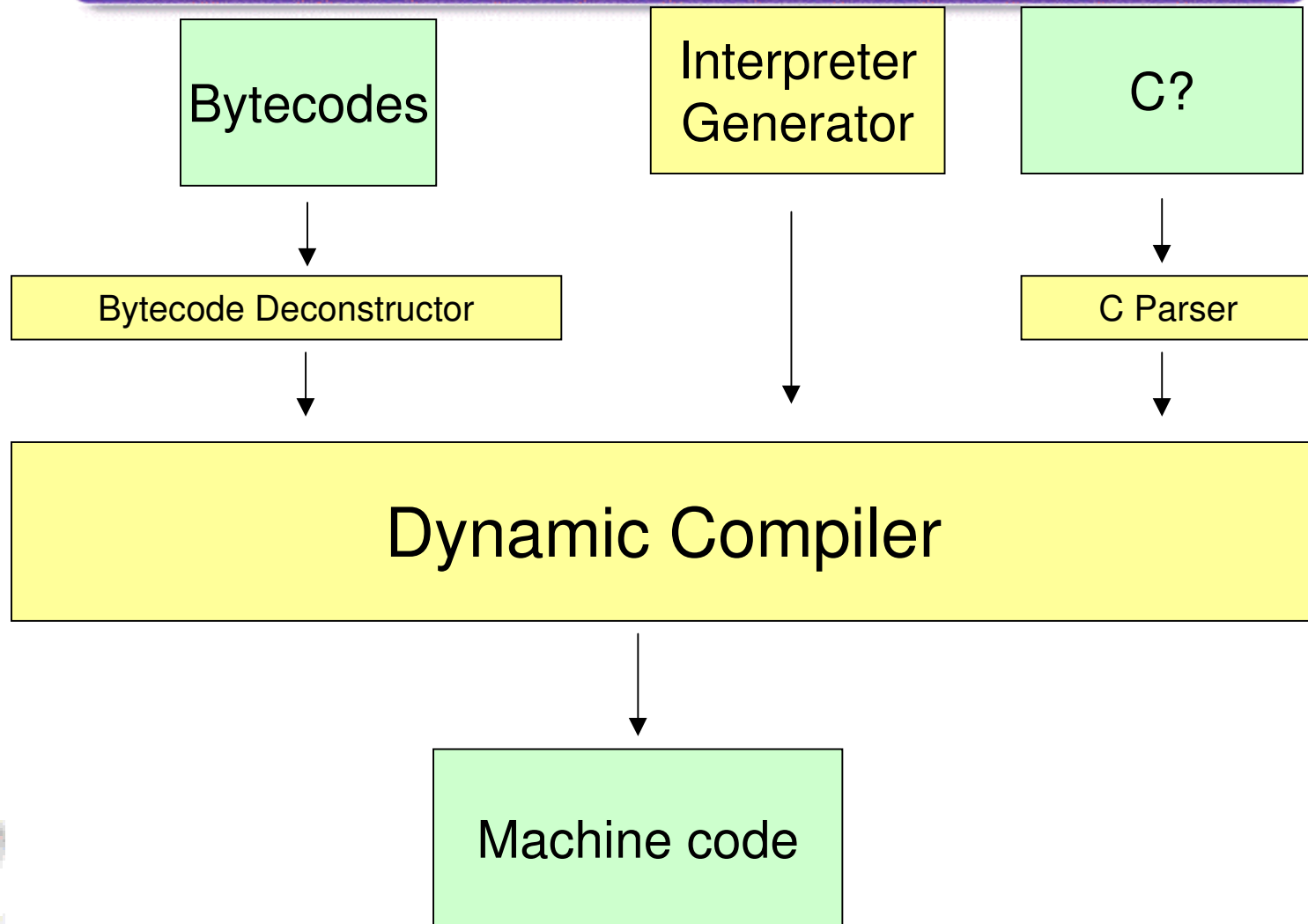


Bytecode Optimization

- Produces standard Java class files
- Optimizations
 - Inlining
 - Constant propagation
 - Copy propagation
 - Constant folding
 - Dead-code elimination
 - ...



The Dynamic Compiler



The Dynamic Compiler

- Needs to support a variety of target architectures
 - x86, ARM, PPC, SPARC
- Ease of porting
 - Target: 2 weeks
- Simple compiler
 - Fast, non-optimal, code generation
 - Small use of memory
 - Static optimizations supported by bytecode optimizer
 - => shadow stack representation



The Dynamic Compiler API - Example

```
public class Test implements Types {
    public static void main(String[] args) {

        Compiler c = Compilation.newCompiler();

        c.enter();
        Local x = c.parm(INT);        // x
        Local y = c.parm(INT);        // y
        c.result(INT);
        c.begin();
            c.load(x);                // x
            c.load(y);                // y
            c.add();                  // x + y
            c.ret();                  // return
        c.end();
        c.leave();

        c.compile();
        Linker linker = Compilation.newLinker(c);
        int entry = linker.link();
        int res = CSystem.icall(new Parm(entry).parm(1).parm(2));

        System.out.println("1 + 2 = "+res);
    }
}
```

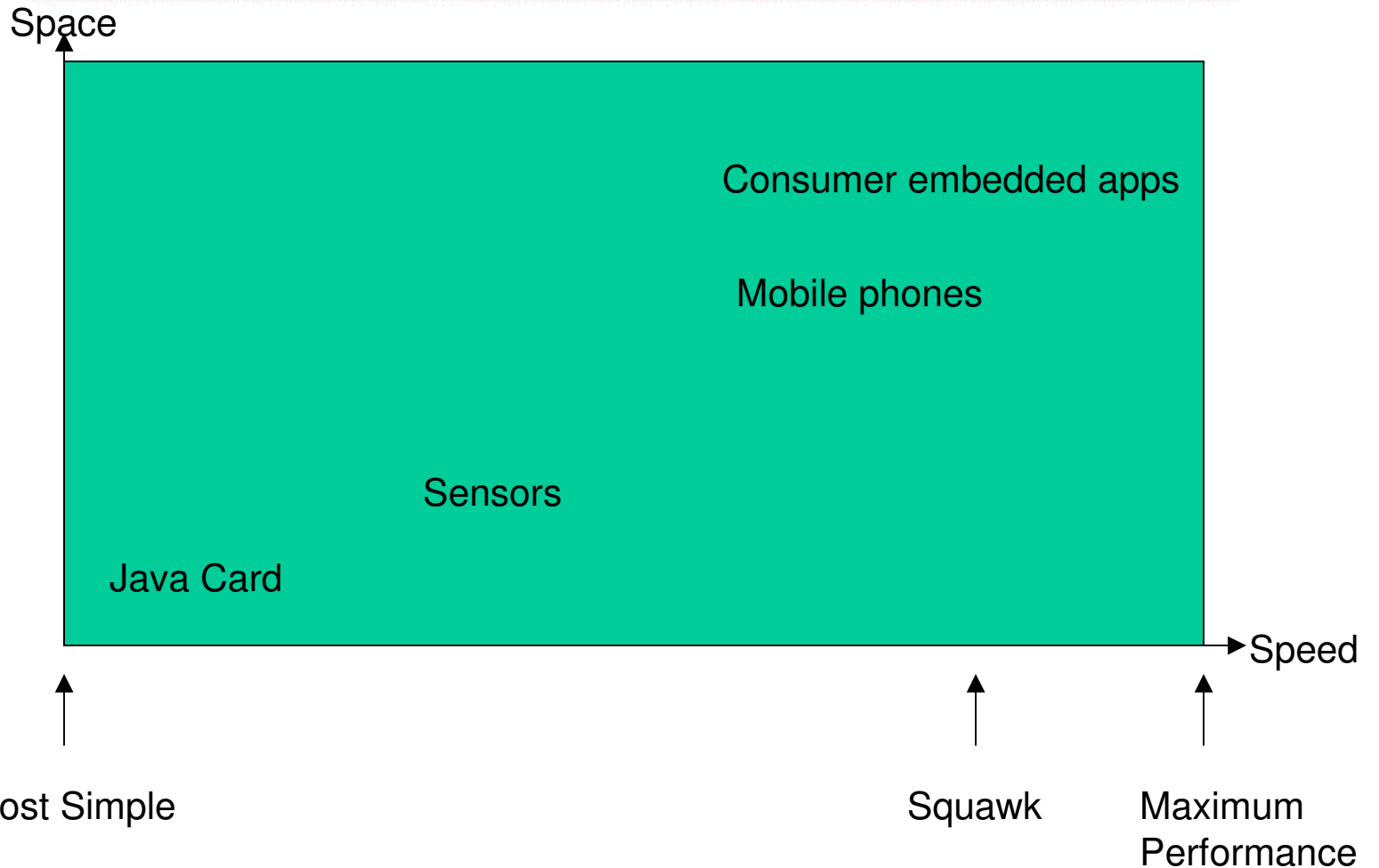


Squawk - Applications

- Java Card
- Sensors
- Mobile phones
- Consumer embedded applications



Squawk – Application Requirements



Expected Results Squawk for Larger System

- Static footprint target
 - 50 Kb (interpreter + GC + jitter)
- Simple compiler target
 - 75% of gcc -O2



Squawk: A Java™ Virtual Machine for Small (and Larger) Devices

`cristina.cifuentes@sun.com`

Sun Microsystems Laboratories

<http://research.sun.com/>

