Squawk Technology

Nik Shaylor
Doug Simon
Bill Bush

Overview

- The Project
- Technical Issues

The Squawk Project

The Squawk Project

- Technical Overview
- What we've done so far

The Squawk Project

- Goal: CLDC/KVM compliant implementation for small devices

- Goal: system written in Java for portability and ease of development

Small JVM Technology

- **Small Footprint**
  - The next generation of smart cards:
    - 8 K of RAM
    - 32 K of EEPROM
    - 160 K of ROM
    - 32 bit processor
  - Size issues include
    - Size of transmitted class files
    - RAM needed for class file loading
    - Size of loaded class files
    - RAM required during execution
    - Size of interpreter and memory system

Implemented in Java

- The whole VM is written in Java
  - Test: run under Hotspot
  - Deploy: translate to C and compiled using a C compiler
- The VM has two types of components
  - Core functionality (interpreter and GC)
  - System classes
    - Regular Java system classes
    - Loader
    - Verifier
    - Thread scheduler
    - Object synchronization

Split VM

- Off-Device Translator
  - Reads pre-verified class files
  - Writes space-optimized collections of classes called *Suites*

- Suites
  - Contain only inter-suite symbolic information
  - Internally linked
  - Compact class representation
  - Space-optimized bytecodes
  - Optimized to simplify verification and GC

Published Specification

- Squawk Specification -- Draft 2.1
  - Defines Architecture
  - Defines Suite file format
  - Defines Bytecode set
  - Defines Bytecode transformations for verification and GC
- Specification has two forms
  - A standard form in which all Java programs can be represented
  - A minimal form for small programs that limits quantities like the number of fields and methods in a class
- See http://sunlabs.eng/projects/squawk

Implementation Status

- A complete prototype implementation has been written
- Developed and tested in Java
- Translated to C and compiled on W32/x86

Conformance Tests

- Product-level testing is complete
  - All 4628 CLDC 1.0 TCK compatibility tests have been run
  - 4537 pass (98.05%)
  - 37 fail because VM constraints are exceeded (such as a class with more than 256 static fields)
  - 43 fail in the translator due to its current inability to handle esoteric or border case constructs
  - 11 fail due to the absence of complete runtime access control (the verifier currently ignores *private*, *protected*, and *public* access modifiers)

Results

- Results
  - Suites are between 35% and 45% size of J2ME class files
  - Less than 6K of RAM is needed to install small suites
  - The footprint of the core interpreter is about 16K (x86)
  - Execution speed is between 70% and 107% of KVM.

(These are all very early results)

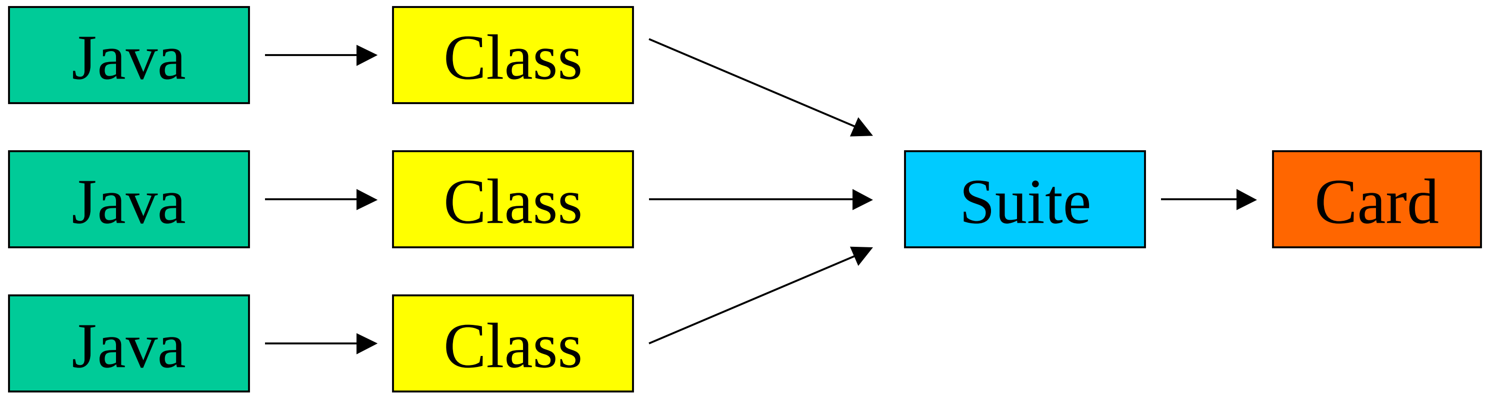
Next Generation Java Card

- Collaboration with Java Card group
  - Artifact
  - Standards work
- Java Card Forum
  - Three contenders: JEFF, standard CLDC, Squawk
  - Choice will not be technical
- Ongoing work to adapt Squawk to the Java Card platform

Technical Issues

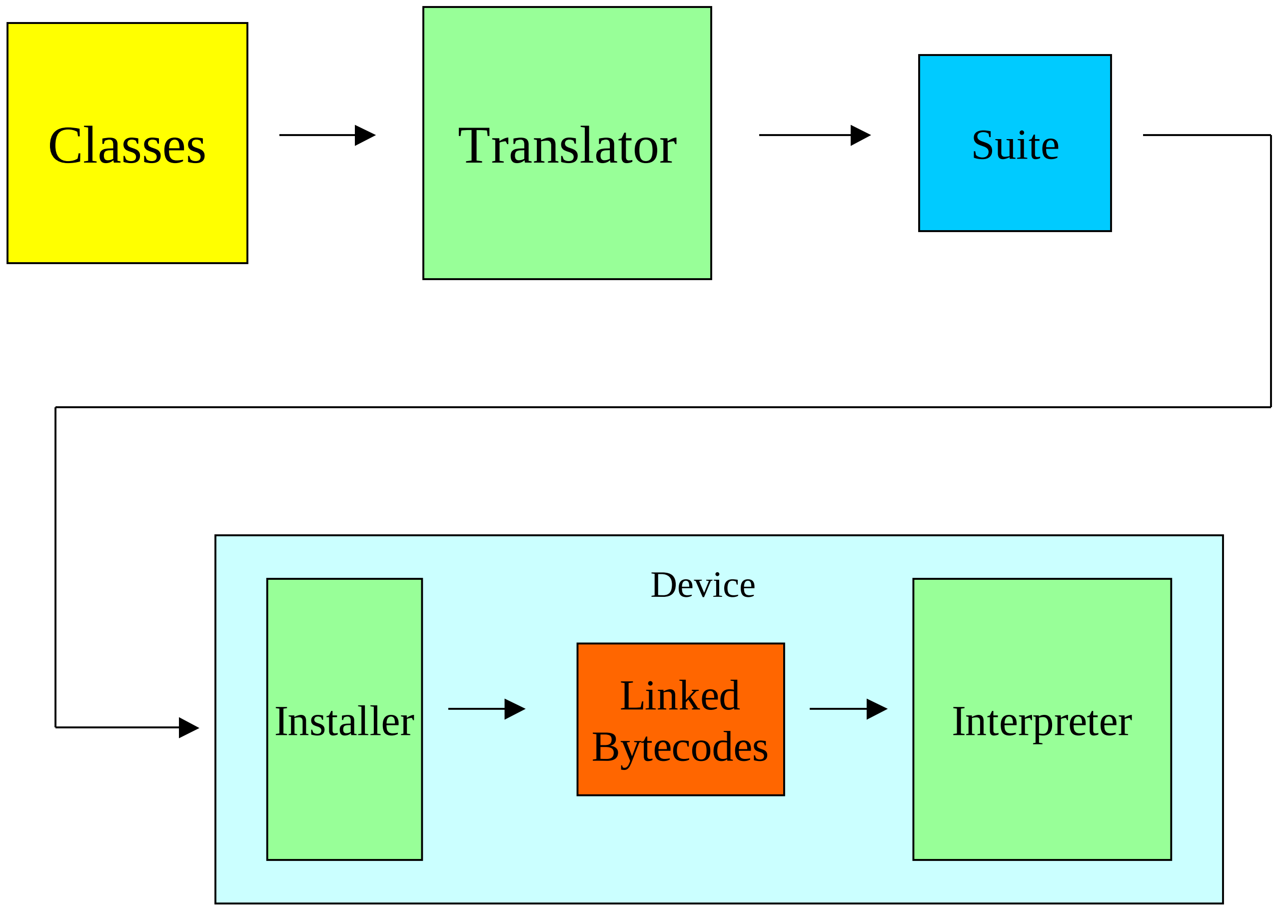
Application architecture

Suite construction



- Java source is converted into class files
- Class files are converted into a suite
- Suites are installed onto the Java card

Bytecode quickening



Translation

Source example

A method from java.lang.Object

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

javac output

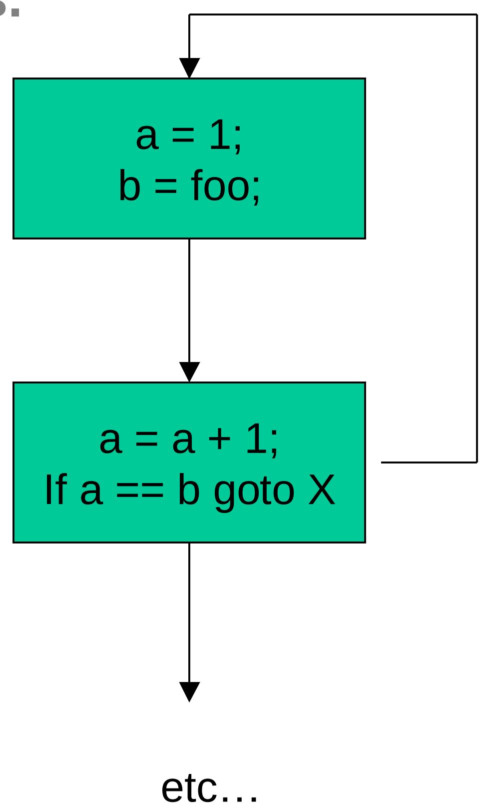
```
Method boolean equals(java.lang.Object)
   0 aload_0
   1 aload_1
   2 if_acmpne 9
   5 iconst_1
   6 goto 10
   9 iconst_0
  10 ireturn
```

After translation (XML output)

```
<method_body>
  <type>1</type>
  <entry>7</entry>
  <locals>
    <type>1</type>
    <type>1</type>
  </locals>
  <stack>2</stack>
  <code>
    <load_0/>
    <load_1/>
    <if_icmpne/><byte>2</byte>
    <const_1/>
    <return/>
    <const_0/>
    <return/>
  </code>
</method_body>
```

Bytecode manipulation

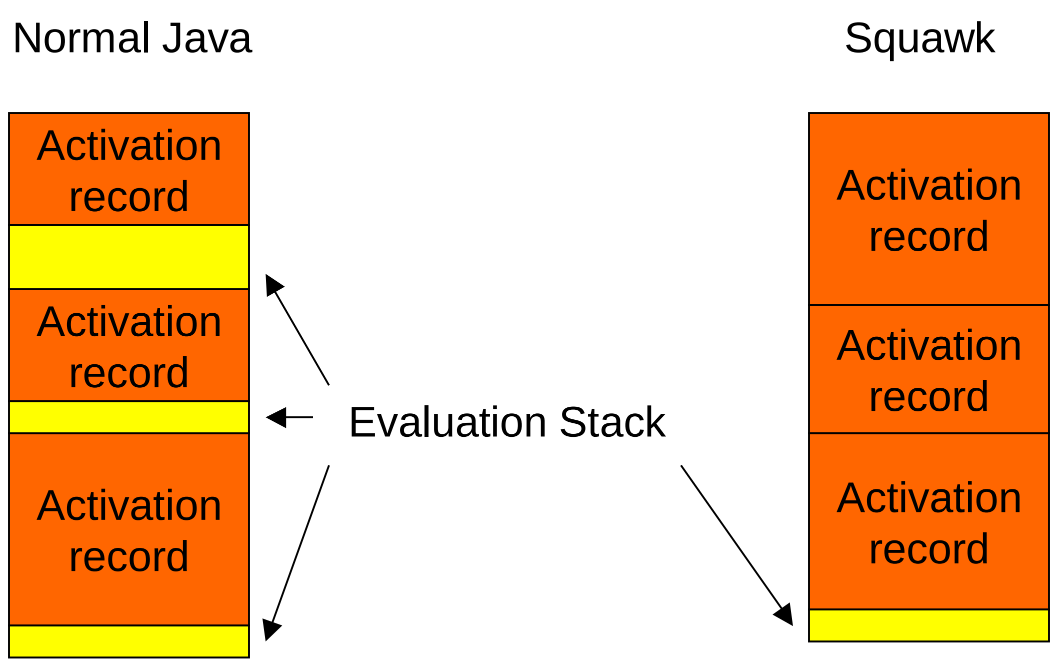
- **The Java stack must be empty at basic block boundaries.**



This is an issue that simplifies verification.

Bytecode manipulation

- **The stack must only contain the operands for certain operations such as invoke, getstatic etc.**



This is an issue that greatly simplifies GC.

Bytecode manipulation

- **Local variables can only be used for one data type.**

Normal Java
Header
int/Object
float/byte[]

Squawk
Header
Object
byte[]
int
float

This is an issue that simplifies verification and GC.

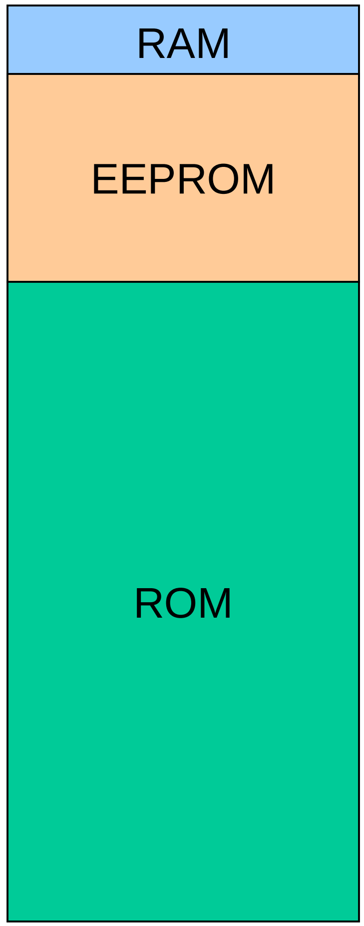
Cost are minimal:

~5% increase in local variables.

~3% increase in code size.

Memory model

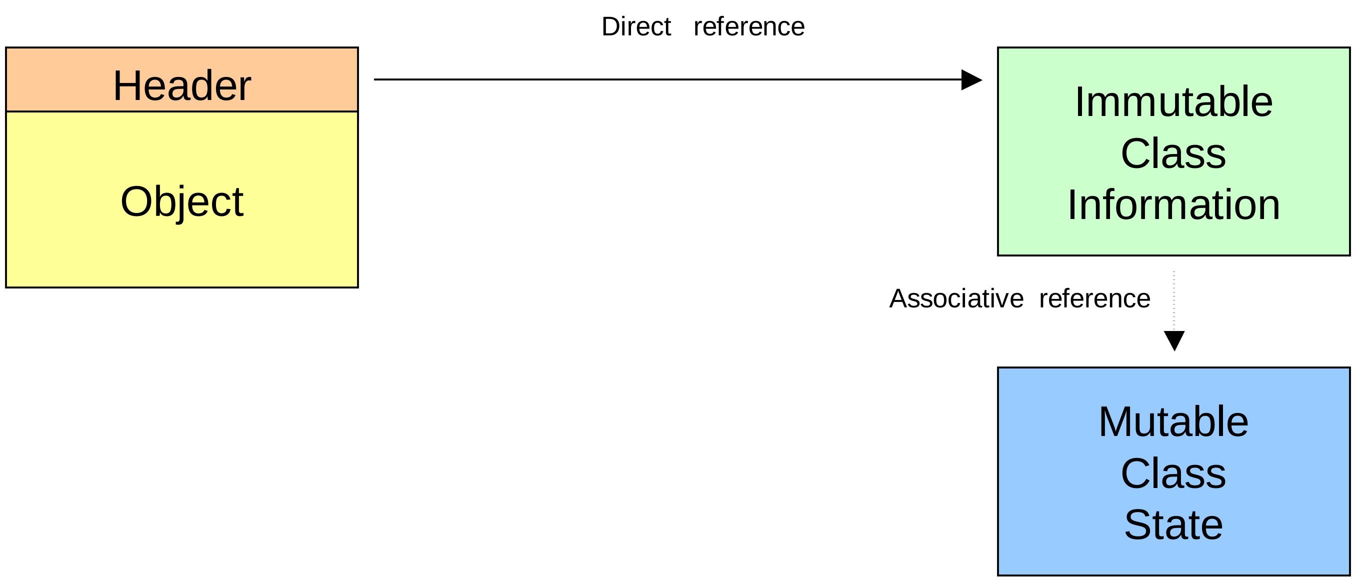
Memory model



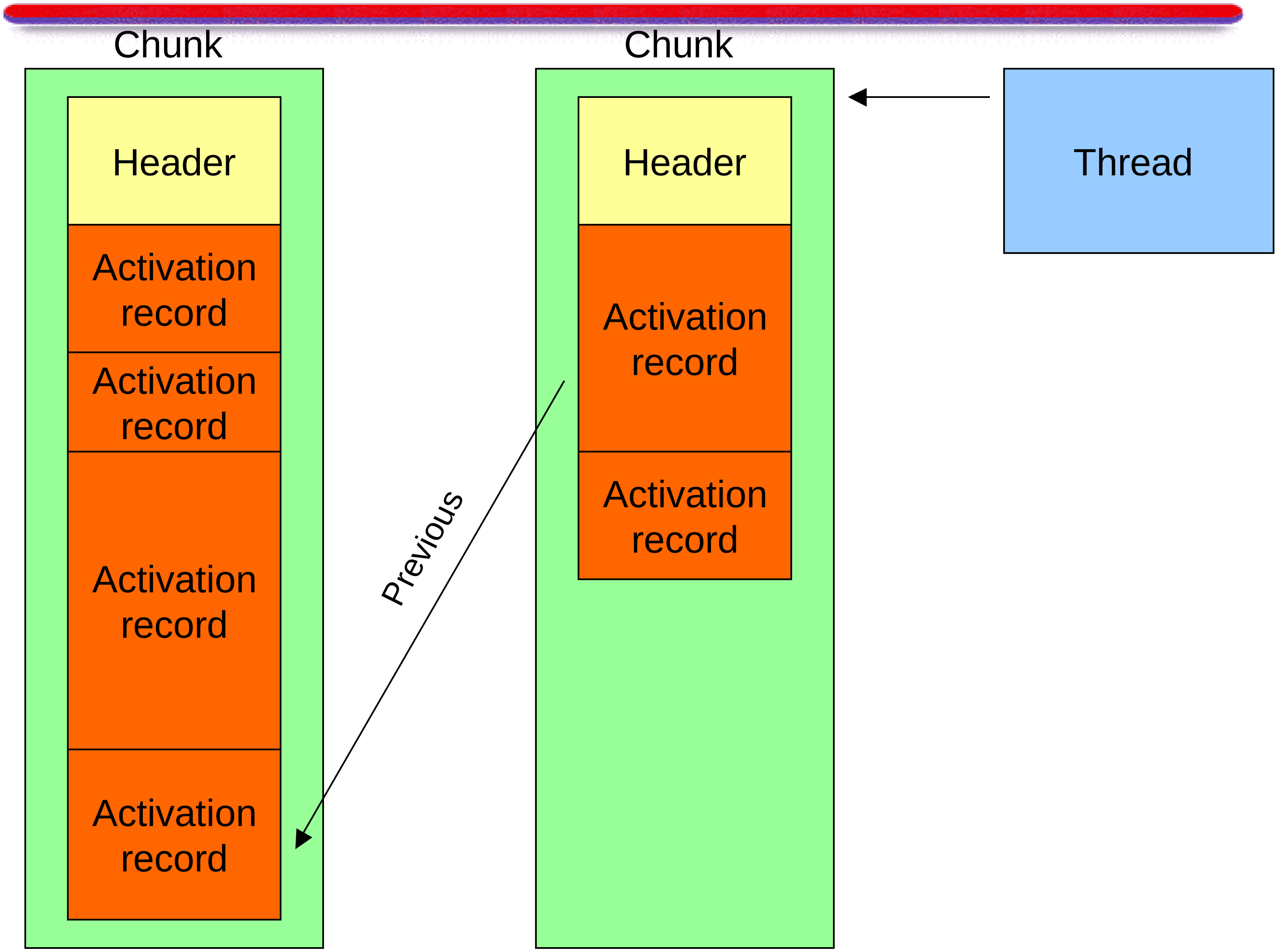
- Squawk uses three object memories.
- All Squawk data structures are Java objects.
- Different garbage collectors are used for the RAM and EEPROM.

Class references

- Objects directly refer to the immutable information of their class. This can therefore be kept in ROM.
- The mutable class state is held separately in an associatively mapped location.



Activation records, Stack chunks, and Threads



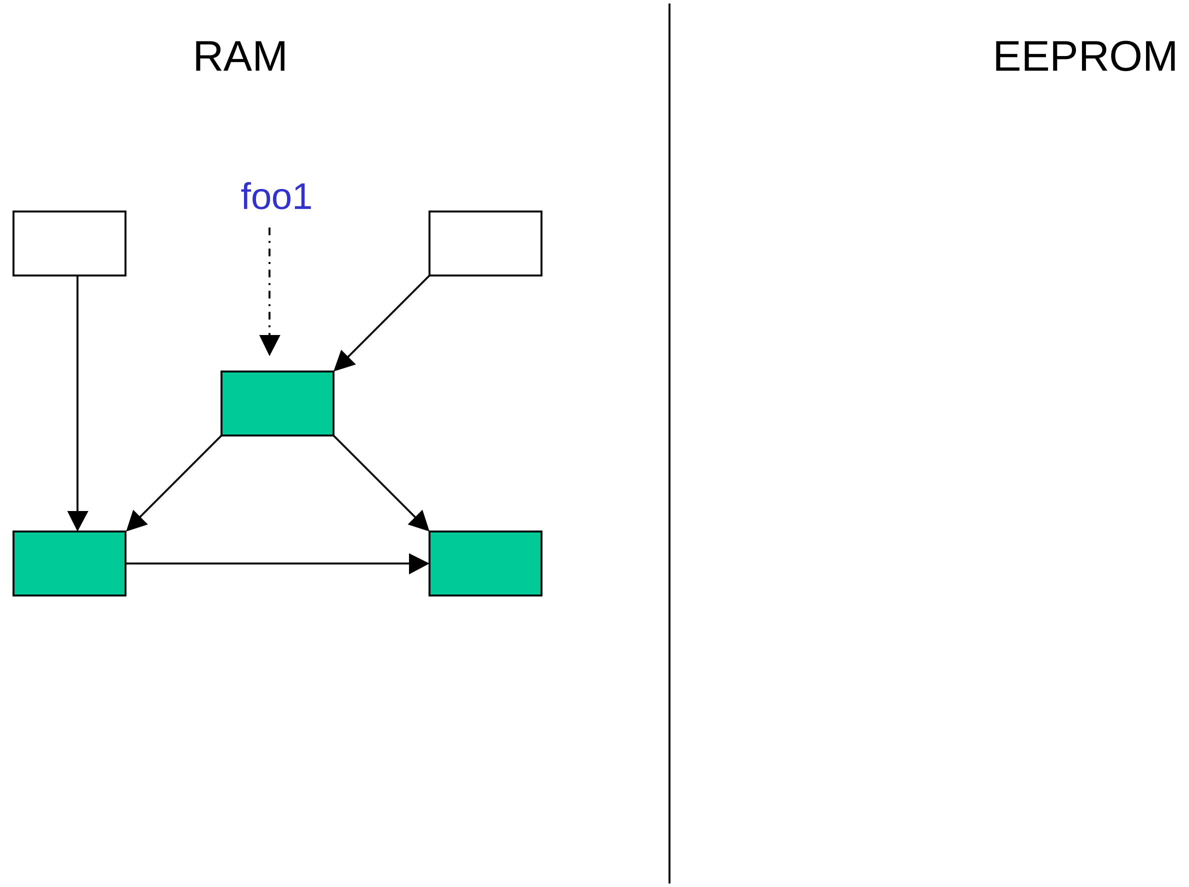
Garbage Collection

GC Features

- All the collectors are exact.
- The translator makes exact garbage collection much easier
  - Local variables are either pointers or non-pointers
  - Sections of evaluation stack are not found between activation records.
- RAM Collector
  - Cheney collector
  - Two generational “Lisp 2” mark/sweep/compact collector
- EEPROM Collector
  - Non compacting mark/sweep.
    - Minimizes EEPROM writes (which are both slow & finite)
- Support for
  - Finalization
  - Object migration

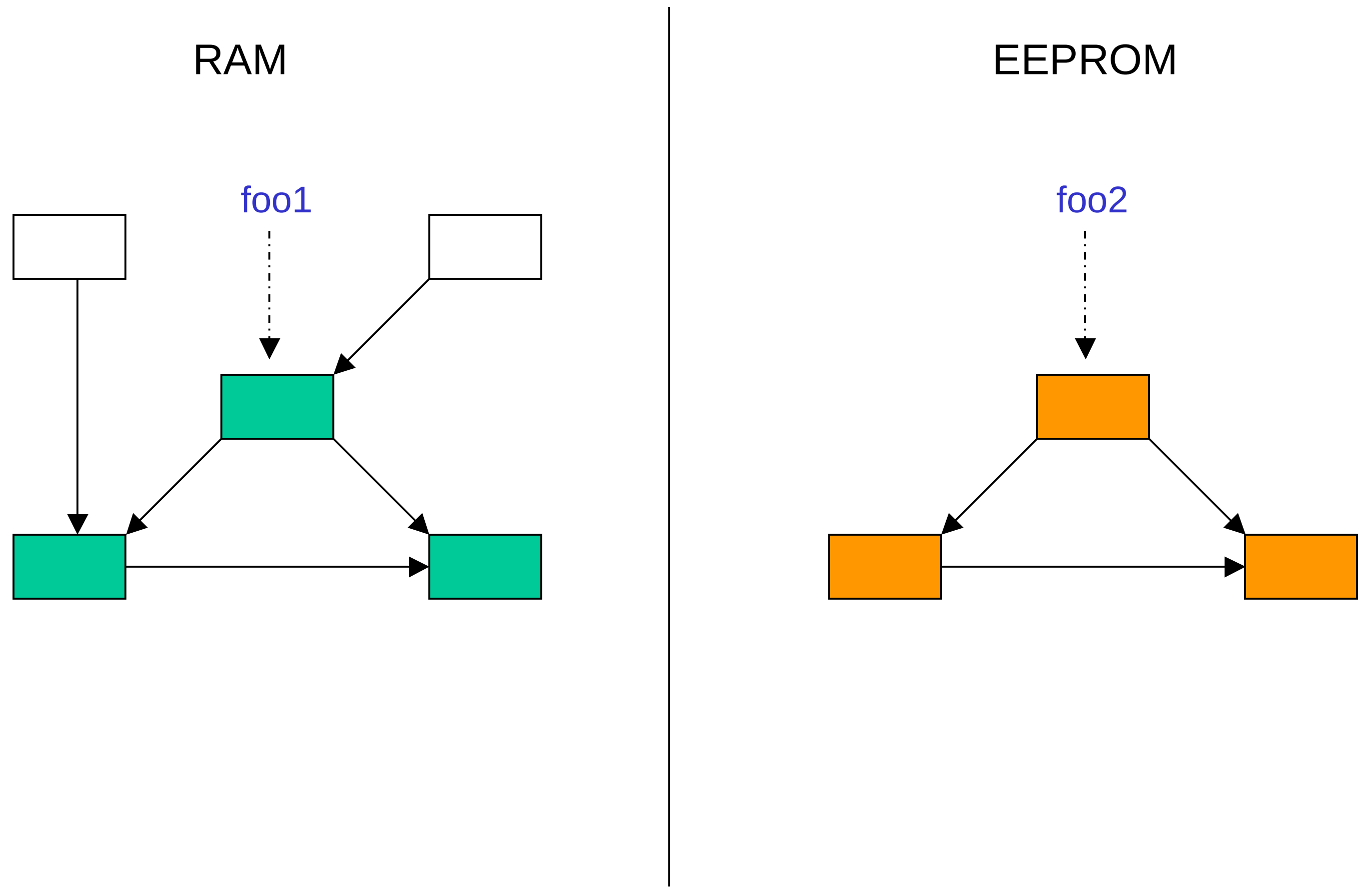
Copying to EEPROM

Foo foo2 = (Foo)PersistentMemory.makePersistentCopy(foo1);



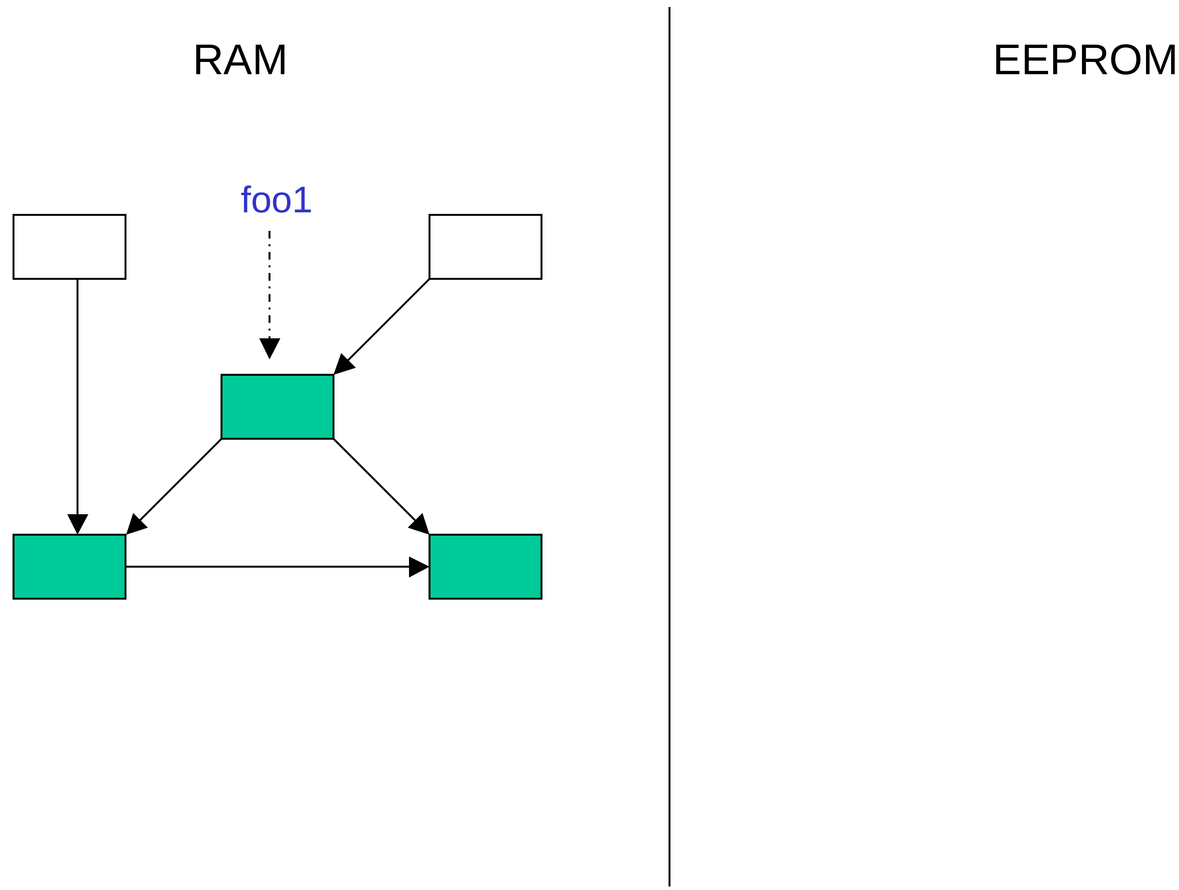
Copying to EEPROM

Foo foo2 = (Foo)PersistentMemory.makePersistentCopy(foo1);



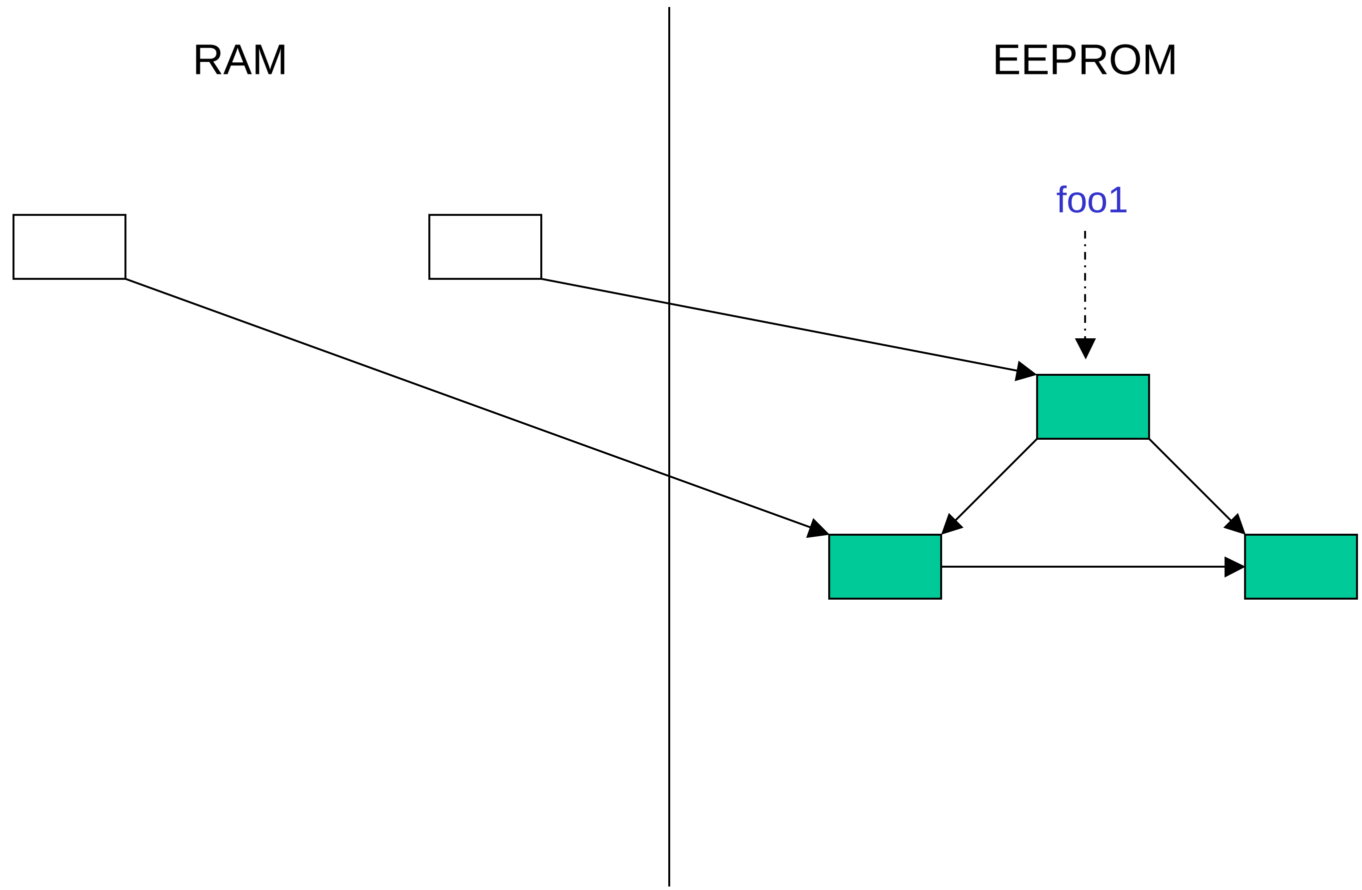
Migrating to EEPROM

PersistentMemory.makePersistent(foo1);



Migrating to EEPROM

PersistentMemory.makePersistent(foo1);



Interpreter core engineering

Java to C conversion

All of the Squawk project is written in Java. The core interpreter is converted into C.

Interpreter.java → squawk.c

```
$ls -l *.java

    17519 Dec 20 13:28 Interpreter.java
   114021 Dec 18 17:30 Interpret.java
    14871 Dec 18 17:30 Memory.java
    99364 Dec 20 10:59 ObjectMemory.java
    47358 Dec 23 14:12 PlatformAbstraction.java
```

```
$ls -l *.c

    16972 Dec 23 14:39 squawk.c
   113779 Dec 23 14:39 interp.c
    14341 Dec 23 14:39 memory.c
    93520 Dec 23 14:39 object.c
    46381 Dec 23 14:39 platform.c
```

Example code

```
case OPC_IADD:  { int r = pop() ; int l = pop() ; push(l + r); continue; }
case OPC_ISUB:  { int r = pop() ; int l = pop() ; push(l - r); continue; }
case OPC_IAND:  { int r = pop() ; int l = pop() ; push(l & r); continue; }
case OPC_IOR:   { int r = pop() ; int l = pop() ; push(l | r); continue; }
case OPC_IXOR:  { int r = pop() ; int l = pop() ; push(l ^ r); continue; }
etc…
```

Most of the interpreter is written in a subset or Java and C.

Language differences

Original Java

```
    void copyBytes(int src, int dst, int num) {
        if (num < 0) {
            fatalVMError("Negative range");
        }
/*IFJ*/ System.arraycopy(memory, src, memory, dst, num);
//IFC// memmove(memory+dst, memory+src, num);
    }
```

Derived C

```
    void copyBytes(int src, int dst, int num) {
        if (num < 0) {
            fatalVMError("Negative range");
        }
/**** Line deleted by Squawk builder ****/
        memmove(memory+dst, memory+src, num);
    }
```

Language differences are solved using a special form of conditional compilation

Feature elimination

```
/*if[FLOATS]*/

    public final void writeFloat(float v) throws IOException {
        writeInt(Float.floatToIntBits(v));
    }

/*end[FLOATS]*/
```

Features can be excluded from the interpreter core and the Java runtime libraries.