# Java™ on the Bare Metal of Wireless Sensor Devices

## The Squawk Java Virtual Machine


Doug Simon
Sun Microsystems Laboratories
16 Network Drive
Menlo Park CA 94025, USA
doug.simon@sun.com

Cristina Cifuentes
Sun Microsystems Laboratories
Level 10, 80 Albert Street
Brisbane QLD 4000, Australia
cristina.cifuentes@sun.com

Dave Cleal
Syntropy Limited
2 Stambourne Way, West Wickham
Kent BR4 9NF, UK
dave@syntropy.co.uk

John Daniels
Syntropy Limited
2 Stambourne Way, West Wickham
Kent BR4 9NF, UK
jd@syntropy.co.uk

Derek White
Sun Microsystems Laboratories
One Network Drive
Burlington MA 01803, USA
derek.white@sun.com



## Abstract

The Squawk virtual machine is a small Java™ virtual machine (VM) written mostly in Java that runs without an operating system on a wireless sensor platform. Squawk translates standard class file into an internal pre-linked, position independent format that is compact and allows for efficient execution of bytecodes that have been placed into a read-only memory. In addition, Squawk implements an application isolation mechanism whereby applications are represented as object and are therefore treated as first class objects (i.e., they can be reified). Application isolation also enables Squawk to run multiple applications at once with all immutable state being shared between the applications. Mutable state is not shared. The combination of these features reduce the memory footprint of the VM, making it ideal for deployment on small devices.

Squawk provides a wireless API that allows developers to write applications for wireless sensor networks (WSNs), this API is an extension of the generic connection framework (GCF). Authentication of deployed files on the wireless device and migration of applications between devices is also performed by the VM.

This paper describes the design and implementation of the Squawk VM as applied to the Sun™ Small Programmable Object Technology (SPOT) wireless device; a device developed at Sun Microsystems Laboratories for experimentation with wireless sensor and actuator applications.



***Categories and Subject Descriptors*** D.3.4 [*Programming Languages*]: Processors—interpreters, run-time environments; D.3.3 [*Programming Languages*]: Language Constructs and Features—classes and objects; D.4.7 [*Operating Systems*]: Organization and Design—real-time systems and embedded systems; D.4.6 [*Operating Systems*]: Security and Protection—authentication; D.2.5 [*Software Engineering*]: Testing and Debugging—debugging aids

***General Terms*** Languages, Experimentation, Security

***Keywords*** Embedded systems, Java virtual machine, Sun SPOT, IEEE 802.15.4, Wireless sensor networks




*VEE'06* June 14–16, Ottawa, Ontario, Canada
ACM 1-59593-332-6/06/0006.


## 1. Introduction

The pervasive computing vision depicts a future in which computation is widely embedded in the everyday world, like "smart dust". One medium for enabling this vision is the tiny, wireless computer that connects to the world with sensors and actuators. Despite the large interest in the area, few significant applications have been written so far, we believe due to the lack of adequate tools and languages to aid in the prototyping of applications for that domain.

Processors associated with wireless sensor devices typically provide small amounts of memory, making it hard for managed runtime languages like Java to run on these devices, due to the static memory footprint of the virtual machine (VM) and the dynamic footprint of the runtime and the applications. Traditionally, wireless sensor applications use languages such as C and assembler to overcome the memory limitations, at the expense of longer application development time. However, it is widely accepted that development time using managed runtime languages is less than that of non-managed languages. Sensors and actuators are commonly used in robots, home appliances such as washing machines and microwaves, industrial appliances such as motors, set-top boxes, and many more.

At Sun Microsystems Laboratories, we have been investigating wireless sensor networks by creating a next generation device we are calling the Sun™ Small Programmable Object Technology, or Sun SPOT (see Figure 1). The Sun SPOT main board is based on an ARM-9 processor and has 512 KB of RAM and 4 MB of flash memory, plus a separate Chipcon 2420 IEEE 802.15.4 radio chip. Additional sensor boards can be attached: the "demo" sensor board includes a 3-axis accelerometer, a light sensor, a temperature sensor, an A/D converter, 8 tri-color LEDs, 5 general purpose I/O pins, and 4 hi current output pins.

We believe that running a managed runtime language like Java on a wireless sensor device will simplify application and device

**Figure 1. Sun SPOT wireless sensor/actuator device, with a demo sensor board on top, the main processor and radio board in the middle, and a battery board on the bottom.**

driver prototyping, thereby increasing the number of developers in this domain, as well as their productivity; resulting in more interesting applications sooner. Java brings with it garbage collection, pointer safety, exception handling, and a mature thread library with facilities for thread sleep, yield, and synchronization. Standard Java development and debugging tools can be used to write wireless sensor applications. Further, we provide tools for deploying and monitoring these devices in a graphical user interface called SpotWorld [SCS05]. This paper concentrates on the Java virtual machine that is available on the Sun SPOT platform: the Squawk virtual machine.

The Squawk virtual machine is a Java VM primarily written in Java and designed for resource constrained devices. Squawk is compliant with the Connected Limited Device Configuration (CLDC) 1.1 Java Micro Edition (Java ME) configuration [CLD] and runs without need for an underlying operating system; commonly referred to as running on the bare metal.

The initial port of Squawk to the ARM took one person 2 weeks, and a full Sun SPOT release, including hardware, networking, and demo sensor board libraries took two people 6 months.

By running on the bare metal, Squawk avoids the need for an operating system (OS) in the Sun SPOT, thereby freeing up memory that would otherwise be consumed by an OS. The OS functionality provided in the Squawk VM amounts to less memory than that required by embedded OSs such as embedded Linux. A lightweight configuration of embedded Linux requires 250 KB of ROM and 512 KB of RAM [All01]. The Squawk OS functionality includes the handling of interrupts, networking stack, and resource management. In Squawk, all device drivers and the 802.15.4 media access control (MAC) layer are written in Java.

A series of features made Squawk ideal for a wireless sensor platform. The Squawk JVM:

1. was designed for memory constrained devices,
2. runs on the bare metal on the ARM,
3. represents applications as objects (via the isolate mechanism),
4. runs multiple applications in the one VM,
5. migrates applications from one device to another, and
6. authenticates deployed applications on the device.

An earlier version of the Squawk VM was targetted at a next generation smart card [SSB03] which had 8 KB of RAM, 32 KB of non-volatile memory, and 160 KB of ROM. In common, both Squawk for smart card and Squawk for Sun SPOT were designed for memory constrained devices and therefore implement a split VM architecture that uses a more compact bytecode set and produces suite files to be loaded into the device. Squawk for Sun SPOT revised that design and extended it to provide support for items 2–6 in the above list.

This paper is organized in the following way. §2 reviews the literature in the area. The design of the Squawk JVM for Sun SPOT (§3) summarizes some of the architecture decisions made for Squawk for smart card, as well as expanding on the new parts of the design for the Sun SPOT, implementation aspects of the suite creator and on-device VM are described in §4 and §5, §6 describes Java programming for the Sun SPOT. Last, §7 provides some experimental results.

## 2. Related Work

We review Java VMs written in Java and other VMs written in the language they implement.

### 2.1 Java VMs Written in Java

We review Java VMs that are written in Java as well as JVMs that provide some OS support to run on the bare metal.

IBM's Jikes Research Virtual Machine (RVM), formerly known as the Jalapeño virtual machine [AAB+99, AAB+00], and the OVM project [PBF+03, FHV03], are Java VMs written in Java. They both make use of the GNU classpath to support desktop and server-level Java libraries. The GNU classpath is a series of J2SE and J2EE Java libraries that are written in the C language. Both Jikes and OVM run desktop and server applications, and require an OS to run on. OVM implements the real-time specification for Java (RTSJ) and has been used by Boeing to test run an unmanned plane.

JX [GFWK02] and Jnode [Loh05] are Java operating systems that implement a Java VM as well as an OS. Security in the OS is provided through the typesafety of the Java bytecodes. JX runs on the bare metal on x86 and PowerPC, and Jnode on the x86. They both access IDE disks, video cards and network interface cards. JX runs on cell phones and desktops. Jnode runs desktop applications. The core of JX is written in the C language. Neither VM makes assurances on the latency to service an interrupt, and both allow for GC to happen while servicing interrupts.

### 2.2 Other Language VMs Written in Their Own Language

Smalltalk was the first object oriented language in which everything is built from objects. Smalltalk was inspired by the Simula, Sketchpad, and Lisp languages. Squeak [IKM+97] was the first usable implementation of Smalltalk written in Smalltalk itself. The Squeak VM is written in a subset of Smalltalk called Slang that can be translated to C. This allows the VM to be written and debugged in Smalltalk, yet the translated VM performs well and is easy to port. The VM can be extended with plugins written in either Slang or C code.

Klein [USA05] is a VM written in the Self language that implements Self. Klein's architecture was driven by the insight that most VMs have three different compilation systems, making the VM complex and hard to maintain. In the Klein architecture there is only need for one compiler, which can be used statically as an ahead-of-time compiler for the system classes of the VM itself, and dynamically as a JIT compiler. The Klein VM assumes memory resources typical of that on desktop machines.

## 3. The Squawk Virtual Machine

The Squawk JVM is the result of an effort to write a J2ME CLDC compliant JVM in Java that provides OS level mechanisms for small devices, easing porting and debugging of the VM. The observation was that most JVMs are written in the C and C++ languages, even though complex processes performed by the JVM can be bet-

ter expressed in the Java language, which offers features such as type safety, garbage collection, and exception handling.

Squawk came out of earlier similar efforts at Sun Labs on systems such as the KVM [TBS99]. A large part of its design was driven by the insight that performing up front transformations on Java bytecode into a more friendly execution format can greatly simplify other parts of the VM. Squawk, as its name suggests, was also inspired by the Squeak Smalltalk VM [IKM+97].

### 3.1 Split VM Architecture

Resource constrained devices do not normally have enough memory to implement class file loading on-device. A common design for these devices is what is known as a split VM architecture, namely, class file loading is performed on a desktop machine, the intermediate representation of the file is then deployed onto the device, and that representation is then run on-device.

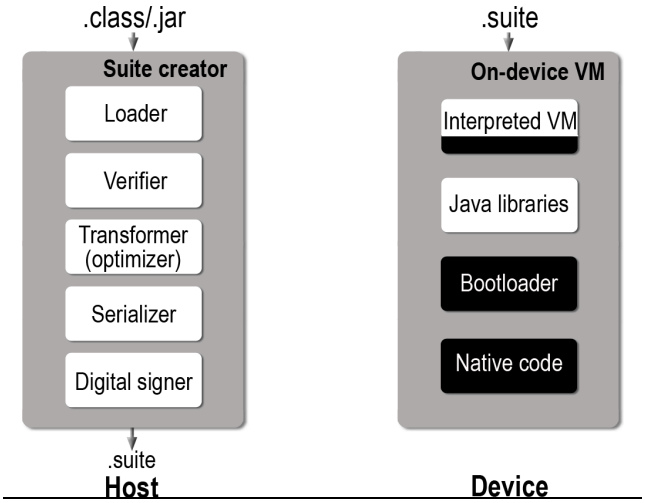


**Figure 2. The Squawk Split VM Architecture. To the left is the Suite Creator and to the right is the On-device VM. The Suite Creator runs on the desktop and the On-device VM runs on the device. White boxes represent Java code, black boxes represent C code.**

Figure 2 shows Squawk's split VM architecture, with the class file preprocessor, called the *suite creator*, to the left, and the *on-device VM* to the right. The Squawk interpreter is written in C. In future, the interpreter will be rewritten in Java and converted to C in much the same way that the garbage collector is currently converted to C. All other parts of the VM are written in Java.

The suite creator transforms the Java bytecodes into a more compact internal representation known as the *Squawk bytecodes*. These bytecodes can be optionally optimized for code-size reduction. The internal object memory representation of an application can be serialized and saved into a file, called a *suite* file.

The on-device VM interprets the suite files on-device, while servicing interrupts from the device itself.

To support the split VM configuration requires a means for building and deploying the VM's bootstrap suite onto the Sun SPOT device. The suite creator is run in a hosted desktop VM such as HotSpot, and all Java classes of the on-device VM, and possibly the debug agent, are fed to the suite creator to produce the bootstrap suite.

Note that components of the on-device VM that are either written (interpreter) or translated (garbage collector) to C are not processed in this form. A separate bootloader binary (also written in C) runs on the Sun SPOT and is responsible for receiving both these components and the bootstrap suite and flashing them into well-known locations on the Sun SPOT. The bootloader can then launch the on-device VM.

### 3.2 Squawk Bytecodes and Suite File Format

The Squawk bytecodes are a compact version of Java bytecodes and were optimized for space, in-place execution, and to simplify garbage collection as follows:

- Space optimization: commonly used Java bytecodes are two bytes instead of three bytes. For example, Squawk bytecodes contain 2-byte branches, more 1-byte load/store/const instructions, 2-byte field access, and 2-byte invoke. There is an escape mechanism in place for float and double instructions, as well as widened operands.
- In-place execution optimization: symbolic references to other classes, fields, and methods are resolved into (direct) pointers, object offsets, and method table offsets, respectively, eliminating the constant pool and dereferencing into it.
- Simplification of garbage collection (GC) optimization: local variables are re-allocated such that slots are partitioned to hold only pointer or non-pointer values, allowing for one pointer map per method. Further, the operand stack is guaranteed to contain only the operands for certain instructions whose execution may result in a memory allocation; i.e., empty operand stack at GC points. This is achieved by means of inserting spills and fills.

  Both these optimizations obviate the need for stack maps and analysis during the collection, simplifying GC, at the expense of requiring more slots in the activation records. Each method only requires a single pointer map and there is no need to scan the operand stack at GC points.

Suite files were designed as objects that contain a collection of Squawk-internal class data structures, including Squawk bytecodes. Suite files are pre-processed sets of class file designed to be executed in-place (i.e., they contain position-independent bytecode). The serialized version of an application's object memory (a graph of objects) is stored in the suite file; all pointers in the serialized object graph are relocated to canonical addresses. Classes in a suite file can refer to classes in the same suite or a parent(s) suite. For example, an application suite depends on a sensor library suite, which in turn depends on the VM's bootstrap suite. This chain of suites forms a transitive class closure.

Suite files are deserialized by the VM on-device and relocated by a single pass over the object memory using a pointer map that was stored with the suite file. This mechanism greatly improves VM startup time and provides a faster alternative to standard class file loading.

Results presented in [SSB03] when comparing the sizes of class files as oppossed to suite files on a set of desktop benchmarks show that, on average, suite files are 38% the size of class files; these results are corroborated with the benchmarks used in this paper (see §7.4).

Note that the suite files are not compressed, this was a design decision made to avoid uncompressing of files on-device and to allow in-place execution of the application without any extra overhead.

## 4. Suite Creator Implementation

### 4.1 Data Structures

Squawk optimizes a number of its core data structures to save space. In this section we describe the choices made for object layout, method objects, and a class' symbolic information.

**Object layout**

Non-array objects have a single 32-bit word header (the class pointer) and array objects have a two word header (the class pointer and the array's length).

All Java objects must support hashcode and monitor operations. In order to save space, the data associated with these relatively infrequent operations is stored separately from the objects themselves. For in-RAM objects, this data is bundled into an *ObjectAssociation* object that is interposed between the original object and the object's class. For objects stored in ROM, the hashcode is a function of an object's address, and the monitor (if needed) is stored in a per-isolate hashtable.

**Method objects**

Methods are encoded as modified byte arrays with a variable length header containing information needed to execute the method (e.g, exception tables, pointer to defining class, number of parameters and local variables, etc.). The array contains the bytecode. An encoding of 4 words is used for the common case of methods with no handlers, less than 32 parameters, less than 32 locals, and less than 32 stack locations.

**String objects**

Strings containing only ASCII characters are encoded as a special type of byte array and all other strings are encoded with a modified type of char array. This is contrasted with the standard implementation of a string as two objects (a `String` instance and a char array). As an example, on a 32-bit platform, the string "squawk" occupies 16 bytes in Squawk (8 byte header, 8 byte body) as opposed to 48 in HotSpot (8 byte header and 16 byte body for the `String` object, and 12 byte header and 12 byte body for the char array in the `String` object).

**Symbols and metadata**

Symbols for a class (names and signatures of fields and methods, and access flags) are encoded in a byte array. The method body metadata (line number table and local variable tables) are stored separately.

The symbolic information for a set of classes can be stripped to a varying degrees by the translator, or discarded entirely if they are never to be linked against by classes in a subsequent translation.

### 4.2 Bytecode Optimizations

Simple bytecode optimizations were thought useful due to the nature of the VM code: Squawk is written using an object-oriented approach, using setter and getter accessor methods wherever possible. Any such accessors that are determined to be non-virtual by static analysis are inlined. Other small static or final methods are also inlined.

Inlining exposes more optimization opportunities, therefore, the translator also performs simple optimizations that reduce the size of the bytecodes: constant folding, and constant and copy propagation. While all optimizations are done at a method level, the current implementation requires that the intermediate representation for all methods be available in order to do the inlining transformation.

The complete size-reducing benefit of these optimizations requires dead code elimination of dead stores, dead loads, and unreachable code. This would remove the methods that are always inlined and cannot be linked against by further translations (i.e. they are private or their symbols are stripped). This analysis is not yet implemented in Squawk.

**Verifier**

Clearly, any set of transformations can potentially introduce errors into the translated code. To catch some of these errors, a Squawk bytecode verifier was designed, to verify the correctness of the generated Squawk bytecode. The Squawk bytecode verifier was designed to be similar to the Java verifier. For example, it needs to check that stack sizes are consistent across different execution paths, it needs to determine the level of consistency of stack, local, and parameter types across different execution paths, it needs to check that no pops are done on an empty stack, and it needs to check type safety.

However, the Squawk bytecode verifier has to differ from a Java verifier due to the nature of some of the Squawk bytecodes: some Squawk bytecodes place further constraints on the operand stack (e.g., `invokevirtual` expects only the operands of the invocation to be on the stack), and other bytecodes require the verifier to keep track of constant integer values (e.g., `invokeslot` and `findslot` which replace the standard `lookupswitch` Java bytecode). As no stack maps[1] are available, an iterative data flow analysis for locals is used.

## 5. On-device VM Implementation

### 5.1 Garbage Collection

Squawk implements a mark and compact generational garbage collector, Lisp 2 [JL96]. This collector is non-preemptible, which has implications for handling interrupts in a device driver written in Java.

The Lisp 2 collector uses two generations and performs three passes over the heap during the compaction phase. This algorithm preserves the order of objects and is suitable for nodes of varying sizes. A sliding window is used for the young generation. Slices are used to obviate the need for an extra pointer-sized field in the header of each object to store a forwarding address. In Squawk, the same algorithm is used for marking and compaction regardless of whether a full or partial collection is being performed. This is a preference for simplicity over performance. A bit vector is used for the entire heap: mark bits for the collection space and write-barrier bits for the old generation. This bit vector uses 3% of the memory available for the object heap. The collector has been extended slightly to support deep-copying of an object graph.

The garbage collector is written in a subset of the Java language and converted to C by means of a limited Java to C convertor based on the upcoming Java 1.6 [Mus05], which has an API for accessing the abstract syntax trees of the Java source compiler. Annotations of the Java source code were used to simplify the translation process.

Executing the collector as compiled C code linked with the interpreter as opposed to Java bytecode being interpreted by the interpreter improved the performance of the collector by a factor of 10.47x on the benchmarks reported in this paper.

### 5.2 Thread Scheduler

Being a bare metal VM, Squawk implements green threads. Green threads emulate multi-threaded environments without relying on any native operating system capabilities. They run code in user space that manages and schedules threads. Green threads switch control when control is explicitly given up by a thread (e.g., `Thread.yield()`), or when a thread performs a blocking operation (e.g., `read()`).

In Squawk, the thread scheduler supports blocking in native methods: a thread is blocked on an event queue polled by the scheduler, and events correspond to an interrupt.

[1] A stack map is a data structure created by the CLDC preverifier that is stored in CLDC class files. Stack maps specify the types in the local variables and operand stack slots at various points within the code. They're used to optimize verification in a JVM by effectively making verification a one pass operation for each method.

Thread rescheduling is done at backward branches in application code. Squawk's system code is non-preemptible, which greatly simplifies the VM design, and assumes that most time will be spent executing application code.

### 5.3 Interrupt Handling and Device Driver Support

The Squawk VM handles interrupts coming from the Sun SPOT device. The device driver enables the device's interrupt. The device driver thread blocks waiting for the VM to signal an event. When an interrupt occurs, an assembler interrupt handler sets a bit in an interrupt status word (ISW) and disables the interrupt to avoid repeats. At each VM reschedule point, the ISW is checked, the event signalled, the scheduler resumes the device driver thread, which in turns handles the interrupt and (typically) re-enables the interrupt. The VM reschedules every 1,000 backward branches and when it wakes up, either due to an event that has happened (typically an interrupt on the SPOT) or a timed wait that has completed (that is, one or more threads that were sleeping are now ready to go). Device drivers are written in Java making use of a peek and poke interface to the device's memory.

The interrupt latency is thus dependent on the time from the global interrupt handler running until the next VM schedule. This is optimal if the VM is idle. If the VM is executing bytecodes in another thread, the penalty is quite small as the VM reschedules after a certain number of backward branches. There is some unpredictability in this case according to how close to the next reschedule the interrupt occurs. However, if the VM is executing a garbage collection, the reschedule is delayed until after the GC completes, hence deteriorating the latency to service the interrupt.

In the case where a simple application has no active threads other than the one waiting for an interrupt, we see an average latency of 0.1 milliseconds. Where other threads are executing and creating garbage, we've recorded worst case latencies of the order of 13 milliseconds, although the mode is still around 0.15 milliseconds for an 100 KB heap size. Note that no real-time claims are made about this interrupt handling mechanism.

## 6. Java Programming for the Sun SPOT

The Sun SPOT devices combine an interesting and customizable set of hardware features with the simplicity of Java application development. This section provides more details on the features of a Sun SPOT device, and by examples, the flavor of application development.

### 6.1 Application Isolation

In Squawk, each application is represented by a Java object. This object is an instance of the class `Isolate`, and can be used to query the status of the associated application, and even directly affect that application through methods such as `start()`, `pause()`, `resume()`, and `exit()`, thereby allowing for the reification of applications.

The Squawk Isolate class is an implementation of an isolation mechanism similar to that of Java Specification Request (JSR) 121: Application Isolation API Specification [JSR05, Cza00]. Isolates are analogous to processes in an operating system: each isolate has resources that are shared amongst the threads of that isolate. In Squawk the immutable state of an isolate is shared, e.g., bytecode, string constants, and parts of classes. Non-shared class state includes static fields, class initialization state, and class monitors.

Isolates have a simple API: an isolate object is instantiated by providing the application's name and its arguments. For example, the following sample code instantiates an isolate for the `com.sun.spots.SelfHibernator` application and passes the url argument to the application. It then starts the isolate and sends an output stream to the isolate.

```
Isolate isolate =
    new Isolate ("com.sun.spots.SelfHibernator",
                 url());
isolate.start();
send (isolate, outStream);
...
```

Application isolation is implemented by placing application specific state such as class initialization state and class variables inside the isolate object. The VM is always executing in the context of a single *current* isolate and access to this state is indirected to the relevant data in the isolate object. This indirection prevents two applications from interfering with each other via access to class variables or even by synchronizing on shared immutable data structures (such as instances of `java.lang.Class`).

#### Isolate Migration

In Squawk, applications can be checkpointed and stored to a file by using the serialization mechanism used with suite creation as well as the deep-copy support in the GC. During checkpointing, the status of each thread, including all temporary variables, can be serialized to a stream for storage. Because each application can be serialized to a stream, that stream can be read into another Squawk VM to reconstitute an isolate on the destination device, skipping the step of storing it onto disk. This effectively migrates the isolate between VMs.

We have started simple experiments with isolate migration, and can currently move running applications from one Sun SPOT to another, or from one desktop or server to another. When moving to an architecture with a different endianness, the appropriate translation is performed.

Isolate migration and checkpointing may be reminiscent of similar facilities available in, say, Smalltalk snapshots and the automatic persistent memory management extension for the Spotless VM [SMES01]. In Smalltalk snapshots, the snapshot may wake up on a different machine, with different Ethernet address, different display size, and several other different hardware capabilities. In the extensions to the Spotless VM, the automatic memory management provided orthogonal persistence including thread state. Java programs could be suspended and at a later time resumed on the same device or a different device, as suspended programs could be beamed between Palm organizers.

In Squawk however, an individual application is the granularity of serialization. Before moving, the isolate must close all open connections to the external world and record relevant information, so that upon waking it can restore the connections. The waking isolate must sense the new environment, and reconnect accordingly. In principle, it may not be possible to successfully connect to the new environment. Thus we expect isolate migration to be utilized by developers in specific situations where such problems are known to be manageable.

We have added a `moveTo(IPAddress ip)` method for isolates, to facilitate our early experiments. With this method, an application can itself decide to move from one device to another. We expect this facility could be used for load balancing, or for scripting a single client server application that moves rather than writing two applications that connect. Isolate migration could be especially useful to effect an in-the-field replacement of one device by another (e.g., with fresh batteries) by letting the user simply pull the software from the old device onto the new. A summer intern wrote an application that migrated itself home upon encountering an exception, so that it could be debugged on the programmer's workstation before being sent back into the field.

### 6.2 Accessing Sensors in the Demo Sensor Board

The demo sensor board library is written in Java and relies on peripheral classes that have been packaged in the `com.sun.squawk.peripheral` domain. The library is 400 lines of commented Java code.

An application can access the accelerometer and get its X, Y and Z coordinates as follows:

```
Accelerometer3D acc =
                DemoSensorBoard.getAccelerometer();
RangeInput x = acc.getX();
RangeInput y = acc.getY();
RangeInput z = acc.getZ();
```

A particular LED (numbered 1 to N) can be accessed, turned on, and set to a particular red, green and blue color combination in the following way:

```
SensorBoardColouredLED led =
                 SensorBoardColouredLED.getLed1();
led.setOn();
led.setRGB (50,60,10);
```

To endlessly changes the LED color by displaying either red, green or blue based on the direction of the accelerometer's motion, the following code can be written:

```
int lastX = 0,lastY = 0,lastZ = 0;
while(true) {
   int xValue = x.getValue();
   int yValue = y.getValue();
   int zValue = z.getValue();

   int r = Math.abs(xValue-lastX) > 35 ? 255:0;
   int g = Math.abs(yValue-lastY) > 35 ? 255:0;
   int b = Math.abs(zValue-lastZ) > 35 ? 255:0;

   led.setRGB(r,g,b);

   lastX = xValue;
   lastY = yValue;
   lastZ = zValue;
}
```

### 6.3 Accessing the Radio Through the Wireless API

The Sun SPOT has a multilayer communications stack as shown in Figure 3.

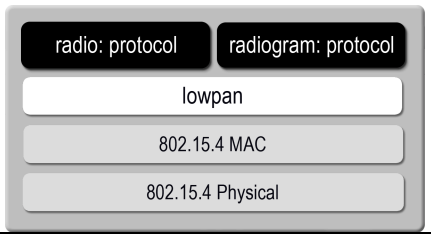


**Figure 3. The Sun SPOT Radio Stack**

The two lowest levels of this stack partially implement the 802.15.4 standard. This standard is targetted at devices with low data rates (of 250 kbps, 40 kbps, and 20 kbps), with multi-month to multi-year battery life and very low complexity. It operates on an unlicensed, international frequency band. A radio that supports this standard can be accessed by means of the devices unique IEEE address and a channel. This implementation provides robust single-hop communication between Sun SPOTs with clear channel checking, packet acknowledgement and retries.

The lowpan multiplexes traffic for up to 255 protocols over the radio connection between two Sun SPOTs or between a host application and a Sun SPOT. The intention of this design is that researchers can build their own protocol handlers above that level. However, to facilitate simple applications, two example protocols have been implemented using the GCF framework.

The generic connection framework (GCF) is part of J2ME, and defines a hierarchy of interfaces and classes that create connections (such as HTTP, datagram, or streams), and perform I/O. The GCF provides a generic approach to connectivity and is defined in the `javax.microedition.io` package. The GCF is based on standard uniform resource locators (URLs) to indicate the connection types to create (e.g., `http`, `file`, `sms`, `socket`, etc.).

The two example protocols provided are a streaming connection and a datagram-style connection. The URL for a streaming radio connection type is as follows, where `address` is the unique IEEE (MAC) address of the Sun SPOT device to be communicating to, and `port` is the channel to be used:

```
radio://{address}:{port}
```

An application can open a stream over the radio and then operate on that connection. For example, the following code will open a radio connection to Sun SPOT device 1020 on channel 42, and will output the number 5 onto that radio stream:

```
StreamConnection conn = (StreamConnection)
                Connector.open("radio://1020:42");
DataOutputStream output =
                        conn.openDataOutputStream();
output.writeInt(5);
output.flush();
```

The form of a datagram-style URL is as follows:

```
radiogram://{address}|broadcast:{port}
```

The radiogram protocol allows for broadcasting to multiple listeners in addition to normal point-to-point communications. The following code shows a radiogram being broadcast on channel 10:

```
DatagramConnection sendConn = (DatagramConnection)
       Connector.open("radiogram://broadcast:10");
dg.writeUTF("My message");
sendConn.send(dg);
```

and the following code receives a radiogram on channel 10:

```
DatagramConnection recvConn = (DatagramConnection)
                Connector.open("radiogram://:10");
recvConn.receive(dg);
String answer = dg.readUTF();
```

An application can also send a radiogram to a specific Sun SPOT device. In the following example, the message "Hello world" is sent to the remote Sun SPOT at address 1020 on channel 42. The connection established between both Sun SPOTs can then wait for receiving a datagram from the remote Sun SPOT:

```
StreamConnection conn = (StreamConnection)
              Connector.open("radiogram://1020:42");
Datagram dg =
          conn.newDatagram(conn.getMaximumLength());
dg.writeUTF("Hello world");
conn.send(dg);       // send the datagram
conn.receive(dg);    // reuse the datagram to receive
                     // from remote SPOT
```

The Sun SPOT radio range is 90 meters. Other facilities exist to reduce transmission power so that Sun SPOTs only communicate with other Sun SPOTs in close range.

### 6.4 Debugging Support

The Squawk JVM allows applications on SPOT devices to be debugged using Java debugging environments that support the Java Debug Wire Protocol (JDWP) [JDW], such as NetBeans and JDB. Due to strict memory constraints on the SPOT device, Squawk does not implement JDWP fully on the device, but splits the work between three components, as shown in Figure 4. There is a *debug proxy* that runs on the developer's workstation, a *debug agent* that is used to control the application being debugged and communicate with the debug proxy, and a small *debug agent support* in the VM itself. The debug isolate and debug proxy communicate using a subset of the JDWP known as the Squawk Debug Wire Protocol (SDWP).

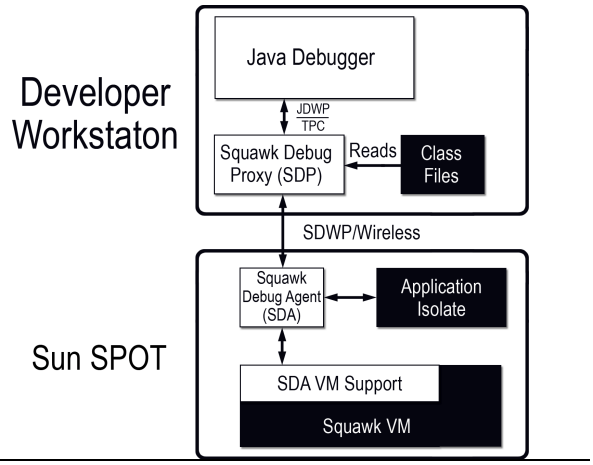


**Figure 4. The Squawk Debug Architecture**

This split architecture allows many of the memory-consuming components of a Java Platform Debugger Architecture (JPDA)-compliant debugging environment to be located on the development workstation instead of in Squawk, reducing memory overhead. In particular, the debug proxy on the workstation has access to the original class files that went into the suites on the device, so it has access to line number tables and method, field, and local variable names that may have been stripped from the suites.

By default, Squawk is build with SDWP support, as well as a second low-level debugger to aid in debugging the hardware. Each of these debuggers adds a 10% overhead on the interpreter loop, by checking whether a breakpoint has been set or not.

### 6.5 Authentication of Deployed Applications

In a split VM implementation, assurances need to be given as to the authenticity of the file to be run on-device, given that someone may tamper with the file to be deployed from the translator (the desktop part of the VM) and such file may bring down the VM on the device (something not permitted in Java VMs).

At suite creation time, a digital signature is applied to the suite using a private key (stored in the desktop). The on-device VM checks the validity of the suite at deploy time, by authenticating the suite's signature using its public key (stored on-device). If the signature is authenticated, the suite is installed on the device. The public and private key pairs are generated at SDK installation time.

## 7. Experimental Results

We measured a variety of data on the Sun SPOT platform running medium-sized Java applications that are used in the object oriented community; Richards and Delta Blue; the Game of Life; a Math application that measures integer and long computation performance; and the Java ME GrinderBench benchmarks used in cell phones; Chess, Crypto, KXML, Parallel, and PNG.

The Richards benchmark, a medium-sized language benchmark (400-500 lines) simulates the task dispatcher in the kernel of an operating system. The original program by Richards was written in BCPL. Richards (gibbons) is Gibbons' translation into Java of a version of the benchmark in C; the code is not very object-oriented. Richards (deutsch_no_acc) is Deutsch's object-oriented implementation of this benchmark, where some of the task state is moved into separate objects. There are several variations to each: Richards (gibbons_final) defines classes and methods to be final where possible, Richards (gibbons_no_switch) replaces an 8-valued integer with three booleans and the switch in the scheduler is replaced by tests of these variables, Richards (deutsch_acc_virtual) encapsulates all object state so that it is accessed via methods, Richards (deutsch_acc_final) virtual calls are made non-virtual by making classes and methods final where possible, and Richards (deutsch_acc_interface) has all classes inheriting from an interface class, simulating the most object-oriented framework-like behavior.

DeltaBlue is a constraint solver benchmark of about 1000 lines of code. The Game of Life simulates a cellular automaton which in turn simulates life of cells in a grid. The Math benchmark tests the performance of integer and long operations.

The benchmarks in the GrinderBench suite are as follows: Chess is a chess playing engine that is used to determine a set of chess moves, Crypto is a suite of algorithms including DES, DESede, IDEA, Blowfish, and Twofish, measuring the performance of Java implementations in cryptographic transactions, KXML measures XML parsing and/or DOM tree manipulation, Parallel exercises a Java implementation's ability to perform its user interface while interacting with the Internet, having multiple threads with some communication threads running on the background, and PNG shows how fast a Java implementation can decode a PNG photo image of a typical size used on a mobile phone. The GrinderMark™ is a single number score that the Embedded Microprocessor Benchmark Consortium (EEMBC) provides, in addition to scores based on individual benchmark applications within the GrinderBench suite.

### 7.1 Static Footprint: Interpreted JVMs on the ARM

We provide static footprint measurements for two different versions of Squawk: Squawk 1.0 (Squawk with CLDC 1.0 libraries, no debugging support, and no authentication of suite support), and Squawk 1.1 (Squawk with CLDC 1.1 (floating point) libraries, SDWP debugging support, authentication of suite files, a partial implementation of the Information Module Profile (IMP), and a second low-level debugger used to debug the hardware). Squawk 1.0 measurements were collected on an ARM7-based Sun SPOT with 256 KB of RAM and 2 MB of flash, and Squawk 1.1 measurements were collected on an ARM9-based Sun SPOT with 512 KB of RAM and 4 MB of flash.

The Squawk 1.0 interpreter executable is 80 KB and ran out of RAM. The rest of the VM and its libraries (CLDC 1.0, networking (IEEE 802.15.4 media access control (MAC) layer), radio library to drive the Chipcon radio, and hardware and sensor integration/control libraries) occupy 270 KB and ran out of flash. The demo sensor board library is 20 KB.

The Squawk 1.1 interpreter executable is 149 KB, the rest of the VM is 363 KB and the libraries (CLDC 1.1, networking (IEEE 802.15.4 media access control (MAC) layer), radio library to drive

| VM | Target | Debugging Support | VM | CLDC libraries | Sun SPOT libraries |
|---|---|---|---|---|---|
| Squawk 1.0 | ARM7tdmi | No | 80 KB | 270 KB | |
| Squawk 1.1 | ARM920T | Yes | 149 KB | 363 KB | 156 KB |
| KVM 1.1 | ARMv4l | No | 131 KB | 504 KB | n/a |
| KVM_d 1.1 | ARMv4l | Yes | 198 KB | 504 KB | n/a |

**Table 1. Static Footprint of Interpreted JVMs Running on an ARM**

the Chipcon radio, and hardware and sensor integration/control libraries) occupy 156 KB. The demo sensor board library is 20 KB.

The size difference between Squawk 1.0 and Squawk 1.1 are due to the new functionality added to the VM in a short amount of time in order to obtain Java ME 1.1 compliance. The codebase has not been cleaned up at this point in time.

Table 1 shows comparisons of the Squawk footprint against the KVM CLDC 1.1 as compiled on an ARMv4l Linux machine. Two versions of the KVM were compiled: KVM 1.1, the production build, excluding the ROMizing library, and KVM_d 1.1, the same build including KDWP support.

As seen in the table, Squawk 1.1's size is comparable to that of KVM 1.1 and KVM_d 1.1. Both Squawk 1.1 and KVM 1.1 provide CLDC 1.1 libraries, and both Squawk 1.1 and KVM_d 1.1 provide debugging support that allows developers to debug Java programs with any JDWP-compatible debugger.

### 7.2 Sun SPOT Memory Map

Squawk runs out of flash memory on the (ARM9-based) Sun SPOT. The Sun SPOT flash is very low power with 1 million cycles/sector endurance. Out of the 4 MB of flash, one third is reserved for system code, not all of which is in use, and two thirds are reserved for applications and data. Figure 5 shows the distribution of memory for the different components of the Squawk VM and associated libraries. The system memory is configured as follows: 256 KB are reserved for the VM binary; 149 KB are in use at present, 512 KB are reserved for the VM suite; 363 KB are used, 64 KB of which are for the debugger (the debug agent support in the VM), 448 KB are reserved for the library suite; 156 KB are in user, and 64 KB are reserved and used by the bootloader. The user memory has two application slots, each of 384 KB, and 2,040 KB of data space available to applications.

The Sun SPOT has 512 KB of SRAM. Less than 20% of SRAM is reserved for system memory, the rest is available for application objects. Figure 6 shows the distribution of RAM memory. The system memory is configured as follows: 16 KB are used by the page tables, 8 KB are used by the C stack, 8 KB are used by the GC stack, 16 KB are used by the C heap, 5 KB are used by C data, and 14 KB are used at startup. The Java heap has 459 KB reserved for it.

### 7.3 Performance: Interpreted JVMs on the ARM

Table 2 shows results obtained from running Squawk 1.1 on a 180 MHz 32-bit ARM920T Sun SPOT and the KVM 1.1 on a Sharp Zaurus 200 MHz 32-bit ARMv4l Linux machine. The Richards results for Squawk show that, as expected, performance degrades when virtual accessors and interfaces are introduced, and that the use of finals slightly improves the performance. The KVM shows a similar trend, though it performs much worse when virtual accessors and interfaces are introduced.

Table 3 shows preliminary data for the performance of Squawk CLDC 1.1, compared to the KVM CLDC 1.0, when using the GrinderBench benchmarks. Squawk was run on a Sun SPOT ARM920T configured with 460 KB of heap. The KVM data was provided by Sun's Java ME team and was run on an ARM926EJ-S 60 MHz with 1 MB of Java heap.

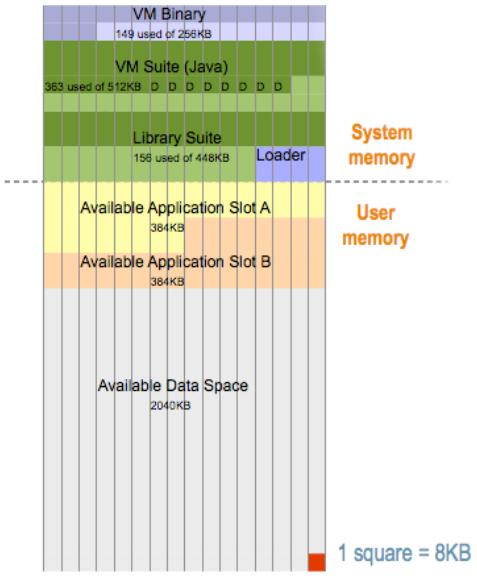


**Figure 5. Sun SPOT Flash Memory**

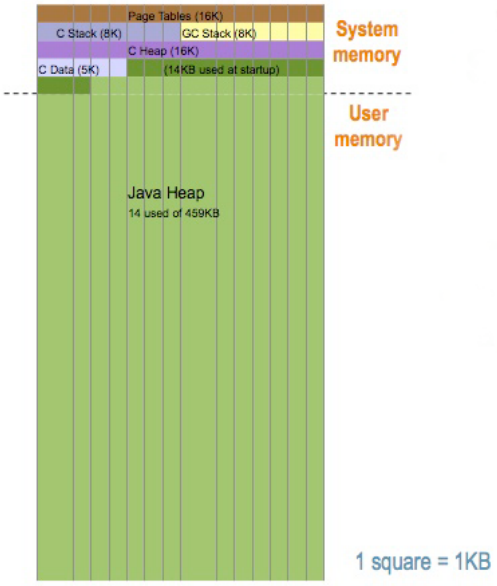


**Figure 6. Sun SPOT RAM Memory**

| Benchmark | Squawk (ARM920T 180 MHz) ms | KVM (ARMv4l 200 MHz) ms |
|---|---|---|
| Richards (gibbons) | 1,296 | 980 |
| Richards (gibbons_final) | 1,287 | 948 |
| Richards (gibbons_no_switch) | 1,412 | 1,262 |
| Richards (deutsch_no_acc) | 1,895 | 2,118 |
| Richards (deutsch_acc_virtual) | 3,314 | 6,002 |
| Richards (deutsch_acc_final) | 3,303 | 3,119 |
| Richards (deutsch_acc_interface) | 3,664 | 4,555 |
| DeltaBlue | 792 | 470 |
| Game of Life | 6,699 | 5,848 |
| Math int | 6,764 | 4,077 |
| Math long | 27,282 | 12,813 |

**Table 2. Runtime Performance of Interpreted JVMs Running on an ARM**

| VM | MHz | Heap | Chess | Crypto | KXML | Parallel | PNG | GrinderMark |
|---|---|---|---|---|---|---|---|---|
| Squawk 1.1 | 180 | 460 KB | 264 | 550 | 452 | 593 | 563 | 456.73 |
| KVM 1.0 | 60 | 1 MB | 288 | 269 | 399 | 272 | 244 | 290.01 |

**Table 3. GrinderBench Results**

The data shows that the Squawk JVM performs in the general ballpark of KVM; i.e., that Squawk is comparable in performance to other interpreted JVMs despite the fact the JVM itself is mainly written in Java.

### 7.4 Suite File vs Class file Sizes

We compare the sizes of Java .class files, Java compressed .class file (.jar file), and the corresponding Squawk .suite file. The GrinderBench benchmarks require input data, such data is stored in resource files and can account for up to 20% of the size of the combined .class and resource files. Table 4 shows the results of measuring the size of the Java class files, compressed JAR file equivalent, and the corresponding Squawk suite files on a SPOT.

As seen in Table 4, the compounded size of the suite files is 37% the size of standard class and resources files, and 56% and 90% the size of compressed JAR files. This latter figure is large when resource files are included in the JAR file: JAR files compress both code and data, whereas the Squawk suite files only compress code by using a different representation of the Java bytecodes, and it does not compress data in any way. The first set of data is more representative of how Java applications get deployed and used on a Sun SPOT.

### 7.5 Bytecode Optimization

We present measurements of the bytecode optimizer as running the Richards benchmarks on a desktop machine. Future work will integrate the bytecode optimizer into the Squawk for Sun SPOT release, without affecting debuggability of the optimized class files.

Figure 7 shows three pieces of data: applications running without any optimizations (the base line at 100%), applications optimized with constant folding and constant and copy propagation, and applications optimized with inlining and the previous optimizations. Note that the benefits of these optimizations will be better seen once dead code elimination is implemented in Squawk.

As seen in Figure 7, inlining of methods greatly improves the performance of the more object-oriented versions of the Richards benchmark, reducing execution time to almost half when all object state is encapsulated and accessed via methods.

### 7.6 Radio Performance

The radio communicates over-the-air at 250 kbps, and has an onboard ring buffer large enough for at least one radio packet. If a second packet is received before the first is read from the buffer, and both packets are reasonably large, then the second packet will be lost.

The radio stack is designed so that data does not get copied through the layers and consequently garbage is kept to a minimum. As a result, applications that do little processing and generate minimal garbage can deal with data at a continuous rate faster than 250 kbps, making it possible to receive packets near continuously. This is especially so for smaller packets where the buffer may hold more than one.

Also, for most practical applications, packets do not arrive continuously and the presence of the buffer means the actual processing rate is not as high. Nevertheless, for applications that do significant processing and receive lots of packets in very quick succession, packets may be lost.

If the packets being lost are point-to-point from another Sun SPOT, that Sun SPOT will not get acknowledgements and will therefore retry, making things slower. If the pakcets are broadcast packets, then neither sender nor receiver will know that the packets were missed. This is the nature of the IEEE 802.15.4 protocol. If an application needs to guarantee delivery of broadcast packets, an acknowledgement scheme at the application level is needed.

It's worth pointing out that the same considerations apply if a packet is lost due to radio interference. Hence, a design to solve the interference problem will also solve the overflow issues.

## 8. Conclusions

The Squawk Java VM is a small, mostly written in Java JVM that can easily be ported to run on other platforms.

Squawk was designed for small, resource constrained devices, and can run without need for an underlying operating system on the Sun SPOT device. Squawk's architecture is that of a split VM architecture, where class loading is done on the desktop, and execution is done on-device. A file format known as suites is used to transfer applications from the desktop to the device.

Facilities easily implemented in Squawk, such as the isolation mechanism and isolate migration, are of much interest and use in writing wireless sensor network applications, as isolates can be reified, and applications can be migrated from one device to another.

| Benchmark | resources files | class files | JAR file | suite file | suite/(class+resource) | suite/jar |
|---|---|---|---|---|---|---|
| Richards (gibbons) | 0 | 10,975 | 7,968 | 4,072 | 0.37 | 0.51 |
| Richards (gibbons_final) | 0 | 10,981 | 7,973 | 4,080 | 0.37 | 0.51 |
| Richards (gibbons_no_switch) | 0 | 10,865 | 7,972 | 4,156 | 0.38 | 0.52 |
| Richards (deutsch_no_acc) | 0 | 16,560 | 11,637 | 6,044 | 0.36 | 0.52 |
| Richards (deutsch_acc_virtual) | 0 | 21,442 | 13,180 | 8,040 | 0.37 | 0.61 |
| Richards (deutsch_acc_final) | 0 | 21,440 | 13,191 | 8,040 | 0.37 | 0.61 |
| Richards (deutsch_acc_interface) | 0 | 22,632 | 14,131 | 8,040 | 0.39 | 0.63 |
| DeltaBlue | 0 | 27,584 | 16,478 | 9,212 | 0.33 | 0.56 |
| Game of Life | 0 | 8,467 | 5,444 | 3,472 | 0.41 | 0.64 |
| Math | 0 | 2,224 | 2,122 | 1,264 | 0.57 | 0.60 |
| Subtotal | 0 | 153,170 | 100,096 | 56,420 | 0.37 | 0.56 |
| Chess | 58,878 | 133,435 | 33,780 | 33,780 | 0.25 | 0.57 |
| Crypto | 9,954 | 89,954 | 60,690 | 55,232 | 0.55 | 0.91 |
| KXML | 19,109 | 111,346 | 66,318 | 57,732 | 0.44 | 0.87 |
| Parallel | 38,731 | 99,747 | 49,848 | 49,848 | 0.50 | 1.29 |
| PNG | 15,472 | 49,401 | 46,025 | 33,404 | 0.51 | 0.73 |
| Subtotal | 142,144 | 483,883 | 256,661 | 229,996 | 0.37 | 0.90 |

**Table 4. Class File, JAR, and Suite File Size Comparison in Bytes**

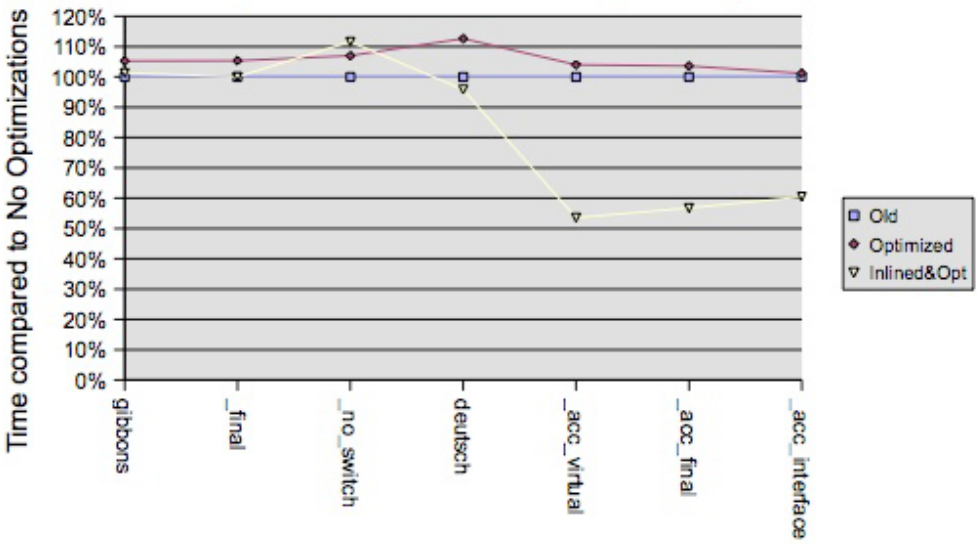


**Figure 7. Results of Performing Bytecode Optimizations on the Richards Benchmarks**

Squawk provides a wireless API for the IEEE 802.15.4 protocol, which extends on the generic connection framework (GCF) and provides for radio and radiogram connection types. The radiogram connection allows for normal point-to-point communication, as well as broadcasting to multiple listeners.

Results show that, even without performance tuning, the Squawk JVM performs reasonably well when compared to other interpreted JVMs, even though Squawk is mainly written in Java. Squawk's size is small despite implementing OS-level functionality to run on the bare metal, and the suite files it generates are about one third of the size of standard Java class files.

## Acknowledgments

The original design and implementation of Squawk was due to Nik Shaylor.

The initial drive for the Squawk on Sun SPOTs was due to John Nolan. John also implemented the demo sensor board library and some of the initial applications on the Sun SPOTs.

We would like to thank Mario Wolczko, Greg Wright, Mikel Lujan, Mike Van Emmerik, and Randy Smith for comments and suggestions on ways of improving the presentation of this paper.

Thanks also go to Bill Pittori for providing access to a Linux/X86 and ARMv4l machines to compile and collect KVM data, Simon Long for collecting some of the data for this paper, Eric Arseneau and Martin Morissette for contributing to the CLDC 1.1 conformance, Eric, Vipul Gupta and Christian Puhringer for the design and implementation of the suite signing architecture, and Nancy Snyder for the diagrams in this paper.

For more information on Squawk refer to `http://research.sun.com/projects/squawk` and for more information on the Sun SPOT project refer to `http://www.sunspotworld.com`

## References


[AAB+99] B. Alpern, D. Attanasio, J. Barton, A. Cocchi, S.F. Hummel, D. Lieber, M. Mergen, T. Ngo, J. Shepherd, and S. Smith. Implementing Jalapeño in Java. In *Proceedings ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Denver, Colorado, November 1999. ACM Press.

[AAB+00] B. Alpern, C.R. Attanasio, J.J. Barton, M.G. Burke, P. Cheng, J.D. Choi, A. Cocchi, S.J. Fink, D. Grove, M. Hind, S.F. Hummel, D. Lieber, V. Litnivoc, M.F. Mergen, T. Ngo, J.R. Russell, V. Sarkar, M.J. Serrano, J.C. Shepherd, S. Smith, V.C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM System Journal*, 39(1), February 2000.

[All01] D. Allison. Embedded Linux applications: An overview. `http://www-128.ibm.com/developerworks/linux/library/l-embl.html`, 2001.

[CLD] JSR 139 - CLDC 1.1. `http://jcp.org/aboutJava/communityprocess/final/jsr139/index.html`.

[Cza00] G. Czajkowski. Application isolation in the Java™ virtual machine. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications OOPSLA*, pages 354–366, Minneapolis, Minnesota, October 15–19 2000.

[FHV03] C. Flack, T. Hosking, and J. Vitek. Idioms in OVM. Technical Report CSD-TR-03-017, Purdue University, Department of Computer Science, 2003.

[GFWK02] M. Golm, M. Felser, C. Wawersich, and J. Kleinoeder. The JX operating system. In *Proceedings of the USENIX Annual Technical Conference*, pages 45–58, Monterey, CA, June 2002.

[IKM+97] D. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the future: The story of Squeak, a practical Smalltalk written in itself. In *Proceedings ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM Press, October 1997.

[JDW] Java Platform Debugger Architecture - Java Debug Wire Protocol. `http://java.sun.com/products/jpda/doc/jdwp-spec.html`.

[JL96] R. Jones and R. Lins. *Garbage Collection–Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, Chichester, England, 1996.

[JSR05] JSR 121 - Application isolation API specification. `http://jcp.org/aboutJava/communityprocess/pfd/jsr121/index.html`, 2005.

[Loh05] S. Lohmeier. Jini on the Jnode Java OS. Online article at `http://monochromata.de/jnodejni.html`, June 2005.

[Mus05] Java 1.6 Mustang. `https://mustang.dev.java.net/`, 2005.

[PBF+03] K. Palacz, J. Baker, C. Flack, C. Grothorff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *Proceedings of the Workshop on Interpreters, Virtual Machines and Emulators (IVME)*, pages 67–76. ACM Press, June 2003.

[SCS05] R. Smith, C. Cifuentes, and D. Simon. Enabling Java$^{TM}$ for small wireless devices with Squawk and Spotworld. OOPSLA Workshop Bringing Software to Pervasive Computing, Oct 16 2005.

[SMES01] D. Schneider, B. Mathiske, M. Ernst, and M. Seidl. Automatic persistent memory management for the Spotless Java™ virtual machine on the Palm connected organizer. In *Proceedings of the Java$^{TM}$ Virtual Machine Research and Technology Symposium (JVM'01)*. USENIX, April 2001.

[SSB03] N. Shaylor, D. Simon, and B. Bush. A Java virtual machine architecture for very small devices. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pages 34–41. ACM Press, June 2003.

[TBS99] A. Taivalsaari, B. Bill, and D. Simon. The Spotless system: Implementing a Java™ system for the Palm connected organizer. Technical Report SMLI TR-99-73, Sun Microsystems Research Laboratories, Mountain View, California, February 1999.

[USA05] D. Ungar, A. Spitz, and A. Ausch. Constructing a metacircular virtual machine in an exploratory programming environment. In *Companion Proceedings to the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications OOPSLA*, pages 11–20. ACM Press, October 2005.