The Squawk Virtual Machine: Java(TM) on the Bare Metal

[Extended Abstract]

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

Cristina Cifuentes
Sun Microsystems Laboratories
16 Network Drive
Menlo Park, CA 94025
cristina.cifuentes@sun.com

ABSTRACT

The Squawk virtual machine is a small Java(TM) VM written in Java that runs without an OS on small devices. Squawk implements an isolate mechanism allowing applications to be reified. Multiple isolates can run in the one VM, and isolates can be migrated between different instances of the VM.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features—*classes and objects*; D.3.4 [**Programming Languages**]: Processors—*interpreters, run-time environments*

General Terms

Languages

1. INTRODUCTION

Java virtual machines are typically written in C/C++. However, a number of complex processes performed by a JVM can be better expressed in Java, which offers features such as type safety, garbage collection and exception handling. This technique has been adopted by a few other JVMs, as well as VMs for other languages such as Smalltalk[3]. Implementing most of the VM in Java also eases the porting and debugging of the VM. For small devices, Java bytecodes typically offer a space saving over natively compiled code.

On small devices, providing OS functionality in the JVM allows for a simpler, more compact VM/OS that supports handling interrupts and writing device drivers in Java. Other OS functionality such as networking stack and resource management need also be provided by the VM. This extended abstract describes our approach at providing a solution to this problem space.

2. THE SQUAWK VIRTUAL MACHINE

The Squawk JVM is the result of an effort to write a CLDC compliant JVM in Java which also provides OS level mechanisms for small devices. Squawk came out of earlier similar efforts at Sun Labs on systems such as the KVM [6]. A lot 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.

The Squawk architecture can be thought of as a split VM with the classfile preprocessor, called the *translator*, on one end and the execution engine on the other. The translator produces a more compact version of the input Java bytecodes that have the following properties: 1- symbolic references to other classes, fields and methods have been resolved; 2- local variables are re-allocated such that slots are partitioned to hold only pointer or non-pointer values; and 3- the operand stack is guaranteed to be empty for certain instructions whose execution may result in a memory allocation.

The last two transformations greatly simplify garbage collectors, as each method only requires a single pointer map and there is no need to scan the operand stack.

3. OBJECT MEMORY SERIALIZATION AND SUITE FILES

The Squawk JVM includes a mechanism for serializing a graph of objects. It is very similar to the mark phase of a garbage collector and is actually implemented on top of the collector. All the pointers in the serialized object graph are relocated to canonical addresses. The serialized form can be deserialized back into a live object graph at a later time.

The output of the translator is a collection of internal class data structures encapsulated in an object called a suite. The translator loads a transitive closure of classes. Suites can form a chain where each subsequent suite only refers to previous suites.

By combining the chained nature of suites with the object serialization mechanism, Squawk can save a set of loaded and translated classes to a file. These suite files can be loaded into a subsequent execution of Squawk, providing a faster alternative to standard classfile loading, greatly improving VM startup time.

The ability to save suites to files also enables Squawk to operate as a split VM for deployment of classfiles to devices that do not have the resources required for classfile loading. On average, suite files are 35% the size of classfiles.

4. THE ISOLATION MECHANISM

Application isolation is implemented by placing application specific state such as class initialization state and class variables inside an *isolate*. 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.

Isolate Migration

The object memory serialization mechanism used to save suites to a file can also be used to externalize the running state of an application. This capability allows checkpointing of applications. It has also been used to achieve migration of a running isolate from one VM to another over a network connection. The serialization/deserialization process also works when migrating between machines with different endianness.

However, true migration of an application's state is in general a very hard, if not impossible problem, given that a substantial amount of state may not be under complete control of the VM (e.g., open socket connections). The Squawk JVM sidesteps these issues by simply throwing a (catchable) exception if an application has open connections when an attempt is made to migrate it.

5. INTERRUPTS

Squawk supports interrupts written in Java. When an interrupt occurs, a low level assembler routine disables the source of an interrupt and sets a bit in an interrupt status word (ISW). An interrupt handler thread is blocked on an event correlated with the bit in the ISW. At each reschedule point, the scheduler resumes the interrupt handler thread which handles the interrupt, and reenables it.

This design had an unpredictable latency due to the non-preemptible garbage collector in Squawk. To address this, the VM was extended to support two separate contexts, each with its own heap and processor state registers. Interrupts now cause an immediate switch to the single-threaded kernel context, which sequentially executes all the relevant blocked interrupt handlers. *This mechanism is work in progress.*

6. CASE STUDY: SQUAWK ON THE SUN SPOT PLATFORM

The Sun SPOT platform is an experimental wireless transducer platform under development at Sun Labs. Transducers; sensors combined with actuator mechanisms; are commonly used in sensors, robots, home appliances, motors, and other devices. The Sun SPOT platform includes an ARM-7 with 256 Kb of RAM and 2 Mb of flash, an 802.15.4 radio, and a general purpose sensor application board (with a 3D accelerometer, a temperature sensor, a light sensor, two LEDs and two switches). The device can be powered from a variety of power sources, including 1.5V batteries.

The interpreted version of Squawk is deployed on the Sun SPOT platform. The core VM is 80 Kb of RAM and the libraries (CLDC 1.0, 802.15.4, and hardware and sensor integration/control) are 270 Kb of flash. All device drivers and the 802.15.4 MAC layer are written in Java.

The performance of Squawk on the Sun SPOT platform is comparable to that of the KVM; an interpreted JVM written in C that runs on top of an OS. Further, deployed Java applications in the form of suite files are about one third the size of their classfile counterpart.

7. COMPARISON TO OTHER WORK

IBM's Jikes Research Virtual Machine(RVM), formerly known as the Jalapeño virtual machine [1], and the Ovm project [5], are Java VMs written in Java. They both run desktop and server applications, and require an OS to run. Ovm implements the real-time specification for Java (RTSJ) and has been used by Boeing to test run an unmanned plane.

JX [2] and Jnode [4] 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 PPC, and Jnode on the x86. They both access IDE disks, video cards and NICs. JX runs on cell phones and desktops. Jnode runs desktop applications.

8. CONCLUSIONS

Squawk is a Java VM written in itself, designed for small devices, and servicing hardware interrupts. Squawk is currently deployed on the Sun SPOT platform, an ARM-7 based wireless transducer device where all device drivers are written in Java.

Squawk implements an isolation mechanism, can run multiple applications in the one VM, and can migrate applications to another machine that runs the same VM.

Acknowledgments

The original design and implementation of Squawk was due to Nik Shaylor. For more information on Squawk refer to `http://research.sun.com/projects/squawk`

9. REFERENCES

[1] B. Alpern and D. A. et al. Implementing Jalapeño in Java. In *Proceedings ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Denver, Colorado, Nov. 1999. ACM Press.

[2] 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.

[3] 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, Oct. 1997.

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

[5] K. Palacz, J. Baker, C. Flack, C. Grothorff, H. Yamauchi, and J. Vitek. Engineering a customizable intermediate representation. In *Proceedings of IVME*, 2003.

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