# Writing Solaris Device Drivers in Java

**Hiroshi Yamauchi and Mario Wolczko**

# Writing Solaris Device Drivers in Java

**Hiroshi Yamauchi and Mario Wolczko**

SMLI TR-2006-156 April 2006

**Abstract:**

We present an experimental implementation of the Java Virtual Machine that runs inside the kernel of the Solaris operating system. The implementation was done by porting an existing small, portable JVM, Squawk, into the Solaris kernel. Our first application of this system is to allow device drivers to be written in Java. A simple device driver was ported from C to Java. Characteristics of the Java device driver and our device driver interface are described.

Sun Labs
16 Network Circle
Menlo Park, CA 94025

**email addresses:**
yamauchi@cs.purdue.edu
mario.wolczko@sun.com

© 2006 Sun Microsystems, Inc. All rights reserved. The SML Technical Report Series is published by Sun Microsystems Laboratories, of Sun Microsystems, Inc. Printed in U.S.A.

Unlimited copying without fee is permitted provided that the copies are not made nor distributed for direct commercial advantage, and credit to the source is given. Otherwise, no part of this work covered by copyright hereon may be reproduced in any form or by any means graphic, electronic, or mechanical, including photocopying, recording, taping, or storage in an information retrieval system, without the prior written permission of the copyright owner.

TRADEMARKS
Sun, Sun Microsystems, the Sun logo, Java, and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>.All technical reports are available online on our website, http://research.sun.com/techrep/.

# *Writing Solaris Device Drivers in Java*

Hiroshi Yamauchi
yamauchi@cs.purdue.edu

Mario Wolczko
Mario.Wolczko@Sun.com

Sun Microsystems Laboratories
16 Network Circle
Menlo Park, CA 94025

*April 2006*

## 1. Introduction

The C programming language enables the creation of flexible and efficient software. C is traditionally used to write system software, including operating system kernels. However, because of the lack of type and memory safety and the burden of manual memory management, it is error-prone, unsafe, and the resulting software is often vulnerable to security attacks, such as those exploiting buffer overruns.

On the other hand, the Java™ programming language has been popular in the last decade because of its type and memory safety, automatic memory management, rich standard libraries, object-orientation, portability, and other features. However, it has rarely been used to create system software; the Java Virtual Machine (JVM™) typically runs as a user application, interpreting or dynamically compiling an application compiled to Java bytecode.

In this report we describe the design and implementation of an experimental in-kernel JVM that allows Java bytecode to run inside the Solaris™ operating system kernel. Our system is based on a small and portable JVM, *Squawk* [SC05, SSB03]. We modified and extended Squawk to embed it within the Solaris kernel. To demonstrate the use of the JVM in this environment, we created a device driver framework in Java and rewrote a simple RAM disk device driver, originally in C, in Java.

In this report we describe Squawk, our modifications and extensions to it, our Java device driver interface, and our experience of porting the RAM disk device driver to Java. As a result of our experiences we believe that running Java code inside the kernel can provide the above-mentioned benefits of the Java programming language for kernel development and in particular for writing kernel extensions such as device drivers.

## *Extending, rather than rewriting, the operating system*

There have been several attempts to write complete operating systems in Java, such as JavaOS™ [Saulpaugh99], JX [GFWK02] and JNode [JNode05]. These have attempted to build, largely from the ground up, an OS-like environment around a JVM, without an underlying host operating system. They extend the JVM to access the hardware directly, and add the components necessary (e.g., a networking stack) to provide the functionalities required by Java applications. When multiple applications and users are to be supported, they also extend the computational model to support resource allocation, protection, etc.

In contrast, we have added a JVM to an existing operating system kernel. Our goal is not to have this JVM run user applications, but to serve as a vehicle for operating system extensions, such as device drivers. This more limited goal avoids the huge initial investment and risk incurred when building a new operating system from scratch. It also allows us to focus just on those aspects of the design that are relevant to the kernel environment, and omit the features specific to user applications. The Java Standard Edition libraries are substantial, and require a great deal of support from the JVM. The kernel environment is much more limited, and the facilities required by kernel extensions are more limited in scope, and different, from those of user applications. Finally, this approach clearly separates the design of the JVM for kernel extensions from the design of a JVM for user applications. State-of-the-art JVMs are large and complex beasts; the complexity is a result of maintaining the illusion of a rich virtual model of computation while providing performance comparable to native binary code. Earlier work (described in Section 10) persuaded us that we did not want to take on the task of porting such a complex system into the kernel environment. While in the longer term there may be value to fusing an in-kernel JVM to a user-level JVM, for this project we decided to keep them separate.

As an example of how an in-kernel JVM might be used, we chose to explore writing device drivers in Java. Solaris provides a device driver interface for drivers written in C [Sun05]. Inspired by this, we built a prototype object-oriented device driver interface for drivers to be written in Java, and ported a sample driver from C to Java. The resulting device performs reasonably well, given the limitations of the JVM we used.

## *Overview of this report*

The rest of this report is organized as follows: Section 2 outlines the design considerations of running a JVM within the Solaris kernel. Section 3 describes our starting point, the Squawk JVM. Section 4 covers the differences between executing applications and device drivers from the perspective of Java. Section 5 discusses how to access existing kernel services from within Java. Section 6 describes the structure of our kernel modules to support Java device drivers, and Section 7 describes miscellaneous other modifications we made to

Squawk. Section 8 describes our framework for writing device drivers, and Section 9 presents our sample Java device driver (code is listed in the Appendix). We discuss related work in Section 10, possible future work in Section 11, and summarize in Section 12.

## 2. Considerations when embedding a JVM in the kernel

The main design issues we encountered in porting Squawk into the Solaris kernel were:

1. *Different execution model* The Java Virtual Machine was intended to run a single user application, potentially composed of many threads. The lifetime of a JVM instance is determined by the execution behavior of the application. The application is typically initiated by a user and begins by calling the *main* method. Normal termination occurs by either the main method returning, or the application calling *System.exit*. Abnormal termination can be due to an internal cause (e.g., an uncaught exception within the application) or an external cause (e.g., the JVM process being terminated in response to a signal).
   Little of this is appropriate within the kernel. We chose to restructure the execution model to make it appropriate for device drivers, described in Section 4.
2. *Access to C state and functions* The Java Native Interface (JNI) [Liang99] is usually used to access state and functions external to a Java application. However, Squawk does not implement the JNI; further, it is a heavyweight mechanism for access to kernel services, providing more generality than is required. We chose to devise and implement a simpler mechanism, described in Section 5.
3. *Division into kernel modules* Extensions to the kernel are structured as loadable kernel modules. We need to choose an appropriate partition of our system into modules (Section 6).
4. *Lack of library support* The kernel does not have the C libraries available to applications, including the standard C library (*libc*). Therefore, it is not possible to simply take an existing user-level JVM and run it, as is, within the kernel. If some feature from the C libraries is needed, it may have to be re-implemented in the kernel. We minimized this effort by starting with a JVM that has few library requirements.

In developing and debugging the virtual machine, one must also overcome the difficulties of developing within the kernel environment. First, there is no memory protection. Kernel address space is visible to all kernel components, and hence there is no protection from wild pointer bugs in one module corrupting other modules. A pointer bug in the virtual machine can (and does!) cause a kernel crash. We were fortunate in that we started with a virtual

machine that had already been debugged in user space. Second, there is much less support for debugging. Kernel-level debuggers are available, but they are not as sophisticated as the debuggers for user programs.

## 3. The *Squawk* Virtual Machine

Our system is based on the Squawk virtual machine [SC05, SSB03]. It has a Java Micro Edition heritage and features a small memory footprint. It was developed to be simple, with a minimum of external dependencies. Having few external dependencies was the key to embedding it quickly within the kernel (more on this in Section 10). Its simplicity made it easy to extend for our purposes.

The Squawk virtual machine core is mostly written in Java. It has an interpreter, and a just-in-time compiler is under development. The version we used had no compiler. The lower-level parts and the main loop of the interpreter are written in C, while the more complex instructions (such as those dealing with monitors) are implemented in Java. A ROM image, created ahead of time, stores the virtual machine code and its initialized state. Application code can be incorporated into the image when it is created, or loaded at runtime. At startup a tiny interpreter executable loads the image from the file system into memory and starts the virtual machine bootstrap.

The Squawk system also sports a graphical instruction tracing tool which allows postmortem debugging at the Java bytecode instruction level. This tool greatly helped us debug our changes.

The following sections describe our modifications to Squawk.

## 4. An execution model suited to device drivers

A Java application executes from the *main* method of the main class and terminates when either control reaches the end of the *main* method, the *System.exit* method is called, or there is an uncaught exception. In contrast, a typical device driver is passive, responding only when a kernel thread calls it. A device driver goes through a two-stage startup, first being loaded, and then having one or more devices attached. Thereafter, it responds to calls (typically, *open*, *read*, *write*, *ioctl* and *close*) and when called executes the appropriate function and returns. Devices may be detached (and re-attached, etc.); finally, the module may be unloaded. In the JVM we had to support an execution model suitable for such behaviors.

One way would be to structure the device driver like a server application. We would insert each call request into a call queue and have a thread process calls, first come, first served. When the call queue became empty the thread would wait for a call request to arrive. This approach does not require any change to the existing Java execution model, but has drawbacks. First, extra threads are required. We would have to choose whether each driver had its own thread (or even multiple threads per driver), and synchronize these

threads with the requesting threads. The call queue could become a performance bottleneck.

A simpler way is to mimic existing driver structure, by allowing the Java driver to act, in effect, as a modularized library of functions. To do this we altered Squawk's behavior so that control can transfer out of Squawk without the JVM state being lost. We encapsulate a single device driver within an instance of Squawk.

Our version of Squawk executes a device driver as follows. When the device driver is loaded into the kernel, the JVM starts up normally and the *main* method of the main class (a subclass of *DeviceDriver*) is executed. The *main* method creates and initializes an instance, and registers it in a static variable. The last thing the main method does is to call a Squawk native method, *VM.stopVM*, in such a way as to cause the JVM to exit while leaving the JVM state (heap, loaded classes, static fields and threads) intact. When a call is later made to the driver, a C trampoline routine transfers control back into the JVM, forcing the main thread to call the corresponding Java method in the registered instance. For example, a driver call to *read* executes the C trampoline *squawk_callback_read*, which calls *DeviceDriver.do_read*, which calls *read* in the registered driver. When *do_read* returns, it does so using *VM.stopVM* again, to preserve the JVM's state. When the device driver is to be unloaded from the Solaris kernel, the JVM is terminated and deallocates its global data structures (heap, etc.).

# 5. Accessing kernel services from Java

Device drivers may need to access kernel services. For that purpose, we must provide a way for a Java device driver to call kernel functions and access kernel data structures.

## *Calling kernel functions*

The original Squawk JVM has a simple native interface used within the virtual machine system code. It does not have a native interface (such as JNI) for application code. For system code, a list of native methods appearing in the virtual machine is constructed at the time a ROM image is created. Each method is assigned an identifying number. All calls to these native methods are replaced with the *invokenative* bytecode, with the identifier as an argument. Within the interpreter, the implementation of *invokenative* uses a switch statement to dispatch to the appropriate native method. We used this technique to call kernel functions, extending the switch statement as necessary.

## *Accessing kernel data structures*

For each type of kernel *struct* we wished to access we created a Java wrapper class; all such classes inherit from our class *AddressWrapper*. A device driver

accesses a *struct* via an instance of this class (a wrapper object). We were faced with a choice in the design of the wrapper objects. In one design, a wrapper would contain only a pointer to the underlying *struct*. Every attempt to access a field of a *struct* would result in a native method call to read or write the field. A drawback is the performance overhead of the calls. Alternatively, the wrapper object would duplicate the state of the *struct.* The object's fields could be accessed by Java code directly, but copying between the *struct* and the wrapper object would be required at the right times. We adopted the latter approach. The task of reconciling the contents of a wrapper object with the underlying *struct* was considerably simplified by only performing the reconciliation when a wrapper was passed to or from native code. We had to deal with the following three argument-passing conventions:

1. *Caller-allocated, callee only reads* The caller allocates and initializes the *struct* and passes a pointer to it to the callee. The callee only reads the contents of the *struct*.

2. *Caller-allocated, callee reads and writes* The caller allocates and initializes the *struct* and passes a pointer to it to the callee. The callee may read and modify the contents of the *struct*.

3. *Callee-allocated* The callee allocates and initializes the struct, and either returns a pointer to it or stores the pointer into a location specified as an argument.

Once it has been determined which of these patterns applies to a given call, it is straightforward to insert the appropriate copying behavior into the wrapper class.

*Reference arguments* The common C convention of passing a pointer to mimic call-by-reference is accommodated in two ways. When calling a Java method from C, we pass the pointer as a value of a special Squawk class, *Addr*. Although *Addr* is a class, its values are immediate addresses (C pointers), which the garbage collector ignores.[1] Native methods are provided to access the underlying location, convert between *Addr* and *long* and for *Addr* arithmetic. When calling C from Java, we pass a reference to a single-element Java array; conveniently, the layout of arrays in Squawk placed the first element at offset zero. When we need to pass a pointer to a field of a wrapper, we add a native method to the wrapper class which returns the address of the field as an *Addr*.

# 6. Kernel modules for Squawk and Java device drivers

We packaged our system into loadable Solaris kernel modules so that it could

1 The Squawk garbage collector is written in Java, and uses *Addr* to manipulate memory locations directly. Values of type *Addr* are also used within the Java heap to reference C entities outside the heap.

be used without the need for Solaris kernel source code. Our system is composed of three kernel modules (see Fig. 1):

> *kfileio* contains a kernel version of the file I/O operations (*open*, *close*, *read*, *write*, etc.), written in C. We derived the source from the Solaris implementation of the corresponding system calls. They differ in that they do not add entries to the per-process file descriptor table (so as to not pollute the file descriptor table of the invoking process). Also, because all data are consumed within the kernel address space, the mechanism for copying data in and out must be different. This module does not depend on any of our other modules. It is loaded using modload(1).
>
> *squawk* contains the in-kernel version of the Squawk virtual machine and the core libraries. This module depends on *kfileio.* It is loaded after *kfileio* using modload(1).
>
> *squawkddi* implements the C portion of our device driver interface. This module is the bridge between the Solaris Device Driver Interface and our Java device driver interface (see Section 8). It is loaded after *kfileio* and *squawk* using add_drv(1).

In our current implementation, *squawk* has hard-coded in the path to the ROM image containing the bytecodes of the device driver to be executed, and *squawkddi* has hard-coded in the name of the main device driver class.

# 7. Other modifications to Squawk

We further modified Squawk in the following ways:

1. *64-bit object references* Since the target version of the Solaris kernel, Solaris 10 for SPARC®, is a 64-bit kernel we had to make a 64-bit version of the original 32-bit Squawk VM. The major modifications included extending the size of object references (pointers) to 64 bits within a ROM image and extending the local variable and evaluation stack slots to 64 bits.

2. *Termination* The original Squawk terminates using *exit*. We added an exit flag, which when set would cause the interpreter loop to terminate, and replaced all calls to *exit* with code that sets the flag.

3. *Multiple JVM instances* To allow each driver to reside within a separate instance of the JVM, we made it possible for multiple instances to co-exist. To do this we moved Squawk's global state into a per-VM-instance structure, initialized at JVM startup. The Squawk JVM is not re-entrant, however, and we did not undertake to rectify this. This limits its utility; more on this in Section 11.

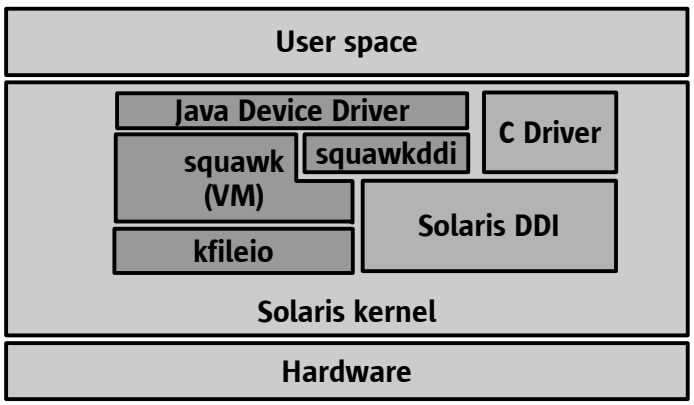

*Figure 1. Kernel module arrangement*

# 8. Writing device drivers in Java

This section describes our Java device driver interface, and the existing Solaris device driver interface on which it is based [Sun05].

## *The Solaris device driver interface*

A device driver will typically be a loadable kernel module containing implementations of the required functions. In response to system calls, the kernel locates and calls the appropriate device driver function. For example, when a driver module is first loaded, the driver's *_init* function is called so that the device driver can perform initializations, such as allocating kernel resources. Conversely, *_fini* is called before the module is unloaded. This is a partial list of required device driver functions:

*probe* checks that the device is ready

*attach* is called when a device is attached

*detach* is called when a device is detached

*getinfo* gets configuration information from the driver

*open* opens the device

*close* closes the device

*read* reads from the device

*write* writes to the device

A device driver developer has to implement the device-specific version of each function. Pointers to the functions are passed, encapsulated in a *struct dev_ops*, to the *modinstall* function, which installs the driver and is typically called from within the driver's *_init* function.

### *The Java device driver interface*

Our device driver interface uses a single object to contain the methods that would correspond to the functions and the pointers to them within a *struct dev_ops*. A driver is written by subclassing the abstract device driver class, *DeviceDriver*. *DeviceDriver* provides access to various kernel functions and data structures in the form of Java native methods and classes that wrap pointers to *structs*, as well as abstract method declarations for the driver functions.

### *The device driver interface kernel module (squawkddi)*

We built our Java device driver interface on top of the C-based Solaris device driver interface using a C-based device driver module called *squawkddi* which bridges the two interfaces. The base module is a normal device driver from the viewpoint of the Solaris kernel and is written so that each driver call is redirected to the Java device driver in the *squawk* kernel module.

## 9. An example driver: A RAM disk

Using our Java device driver interface, we implemented a RAM disk device based on a sample driver, written in C, available from Sun's web site [Sun]. We chose this driver because we wanted to experiment with a relatively small pseudo-device driver that did not deal with a real I/O device. We had to fix the sample driver code before porting it to Java since it was outdated and did not work on Solaris 10. The fixed C driver had 578 lines of code including occasional comments. The Java version has 422 lines of code, and is a little simpler than the C version. The Java source is listed in the Appendix (with some debugging lines elided).

*RamDiskDeviceDriver's* main method is called when the virtual machine starts up. It creates an instance of the driver class and registers it for calls. The other methods in the class are semantically equivalent versions of the C functions defined in the C version.

We performed the following simple performance measurements to compare the Java version of the driver to the C version. The measurements were performed on a Sun E420R system with four 450MHz UltraSPARC II CPUs and 1GB of RAM, using Solaris 10 (Build 76).

1. **Raw system call overhead** We measured the time to call the *close* system call on the RAM disk device. Because the *close* function of the RAM disk driver is empty, the measurement yields the total overhead from the system call down to the device driver routine and back. We repeated the measurement ten times and computed the average time. The Java version took 4.48 microseconds whereas the C version took 3.84 microseconds, an overhead of 16.7%. The virtual machine was given 512 KB of heap.

2. **Throughput** We measured the time to copy a 1MB file within the RAM disk. This measurement should indicate the performance of the block I/O of the driver. We repeated the measurement ten times and computed the average time. The Java version took 178 microseconds and the C version took 63 microseconds. When a GC occurred during a copy, the Java version took 230 microseconds. The Java version took approximately 2.8 times and 3.8 times longer than the C version without and with a GC, respectively. The virtual machine was given 512 KB of heap. We believe the overhead in the Java version is mainly due to bytecode interpretation. The results are encouraging because the Squawk virtual machine we used was an early unoptimized version, with a just-in-time compiler in development.

As a demonstration of the safety properties provided by Java, we introduced a null pointer dereferencing bug into both the C and Java versions of the RAM disk device. The effect in the Java version was that a Java exception was thrown, resulting in an error code being returned from the kernel to the calling application. The C version caused a panic crash of the Solaris kernel. (This was not unexpected, but it does make for a dramatic conclusion to a presentation!)

# 10. Related work

*Operating systems written in pointer-safe and type-safe languages*
Many operating systems have been implemented entirely or primarily in pointer- and type-safe programming languages. SPIN was implemented in Modula-3 [BSP*95]. Pilot was implemented in Mesa [RDH*79]. Native Oberon was based on Oberon [WG92]. JavaOS, JX and JNode are operating systems primarily written in Java [Saulpaugh99, GFWK02, JNode05]. Singularity is a research operating system to be written in C# [HL04]. These systems use to their advantage the robustness and safety inherent in high-level programming languages. Our system exploits Java in the same way, but differs in that it extends an existing commercial operating system, Solaris, written in C, rather than building an operating system from scratch.

*OS-like functionalities for Java applications.* MVM is a multitasking virtual machine implementation [CD01]. It modifies a JVM to enable multiple Java applications to run within a single address space for better startup performance and scalability. Security and isolation between multiple applications are ensured by careful isolation of application states. It also provides resource control for Java applications. KaffeOS is a Java runtime system that provides application isolation and resource management [BHL00]. JRes provided a resource accounting interface for Java applications [CE98]. J-Kernel provides multiple protection domains for Java applications [HCC*98]. These systems provide some of the functions traditionally associated with an operating system within a user-space JVM, rather than in the kernel space.

An earlier Sun Labs project, undertaken by Greg Czajkowski, Glenn Skinner

and Ben Titzer, attempted to port Sun's Java HotSpot™ virtual machine [Sun99] into the Solaris kernel. The Java HotSpot VM is orders of magnitude more complicated than Squawk, and the project ultimately foundered due to this complexity. It has many dependencies on external libraries. Problems included bridging the gap between user and kernel space threading models, dynamic linking, limitations on kernel module size, and signals. They ported a substantial part of *libc* into the kernel. One of the reasons for choosing Squawk for our project was that we observed first hand the difficulties presented by a complex virtual machine with so many external dependencies.

# 11. Future work

Our experience in porting the sample driver falls far short of the complexity in creating drivers for real devices. There is likely much to learn in implementing non-trivial drivers in Java.

At present, interfacing to Solaris kernel services requires the tedious creation of wrapper classes. A tool to automate much of this work, which could parse C header files and generate appropriate wrappers, would be useful. Perhaps SWIG (www.swig.org) could be used for this.

There are various enhancements to Squawk which would make the system more useful:

- The current implementation of Squawk schedules threads cooperatively, like the “green threads” of early JVMs intended for desktop applications. This is a reasonable design for a JVM intended for embedded applications. However, the result is that the JVM is not reentrant and only one kernel thread can be executing within a single instance. To enable multiple threads to execute within a driver simultaneously, Squawk would need to be made reentrant.
- Incorporating performance enhancements to Squawk, especially in the area of compilation, should dramatically improve the performance and utility of our system.
- Our current design isolates drivers one from another by embedding them within different JVM instances. Another direction would be to use isolates within a single JVM instance [JCP05]. There is work in progress adding isolate support to Squawk. To make this complete, one would also have to modify the kernel modules to accept configuration information for different drivers (currently, each module is hard-coded for a particular driver, as described in Section 6).

Our native method interface does not support passing a reference to a Java method to a C function expecting a function pointer, but this has proved unnecessary so far.

## 12. Summary

We have described our port of the Squawk JVM into the Solaris kernel environment and its initial use for writing Solaris device drivers. We believe that this experience demonstrates that it is feasible to port a Java virtual machine into the Solaris kernel, thereby deriving the benefits of Java for kernel extension, and provides a basis for further experimentation.

## Acknowledgments

We are indebted to Glenn Skinner and Greg Czajkowski for sharing their experiences attempting to embed HotSpot into Solaris, and their assistance with Solaris nuances. Nik Shaylor and Doug Simon helped us understand the inner workings of Squawk. Glenn, Greg and Doug also provided invaluable comments and suggestions on the report.

## References


[BHL00] Godmar Back, Wilson C. Hsieh and Jay Lepreau,
*Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java,*
In Proceedings of the 4th Symposium on Operating Systems Design and Implementation, October 2000.

[BSP*95] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc Fiuczynski, David Becker, Susan Eggers and Craig Chambers,
*Extensibility, Safety, and Performance in the Spin Operating System,*
In Proceedings of 5th Symposium on Operating Systems Principles, 1995.

[CD01] Grzegorz Czajkowski and Laurent Daynès,
*Multitasking without Compromise: a Virtual Machine Evolution,*
In Proceedings of the 16th ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications, 2001.

[CE98] Grzegorz Czajkowski and Thorsten von Eicken,
*JRes: A Resource Accounting Interface for Java,*
In Proceedings of the ACM conference on Object-Oriented Programming, Systems, Languages, and Applications, 1998.

[GFWK02] Michael Golm, Meik Felser, Christian Wawersich and Juergen Kleinoeder,
*The JX Operating System,*
In Proceedings of the 2002 USENIX Annual Technical Conference, 2002.

[HCC*98] Chris Hawblitzel, Chi-Chao Chang, Grzegorz Czajkowski, Deyu Hu and Thorsten von Eicken,
*Implementing Multiple Protection Domains in Java,*
In Proceedings of the 1998 USENIX Annual Technical Conference, 1998.

[HL04] Galen C. Hunt and James R. Larus,
*Singularity Design Motivation* (Singularity Technical Report 1),
Microsoft Research, MSR-TR-2004-105, 2004.

[JCP05] Java Community Process,
*Application Isolation API Specification*,
`http://jcp.org/jsr/detail/121.jsp`, 2002.

[JNode05] *Java New Operating System Design Effort,*
`http://www.jnode.org/`, 2005.

[Liang99] Sheng Liang,
*The Java Native Interface: Programmer's Guide and Specification,*
Addison-Wesley, 1999.

[RDH*79] David D. Redell, Yogen K. Dalal, Thomas R. Horsley, Hugh C. Lauer, William C. Lynch, Paul R. McJones, Hal G. Murray and Stephen C. Purcell,
*Pilot: An Operating System for a Personal Computer,*
In Proceedings of the 7th ACM Symposium on Operating Systems Principles, 1979.

[Saulpaugh99] Tom Saulpaugh and Charles A. Mirho,
*Inside the JavaOS Operating System,*
Addison-Wesley, 1999.

[SSB03] Nik Shaylor, Doug Simon and Bill Bush,
*A Java Virtual Machine Architecture for Very Small Devices,*
Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems, ACM Press, June 2003.

[SC05] Doug Simon and Cristina Cifuentes,
*The Squawk Virtual Machine: Java on the Bare Metal,*
To appear in the Companion to the Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, 2005.

[Sun99] Sun Microsystems, Inc.,
*The Java Hotspot Performance Engine Architecture,*
`http://java.sun.com/products/hotspot/whitepaper.html`, April 1999.

[Sun] Sun Microsystems, Inc.,
*Sample Driver Source for Solaris 8,*
`http://developers.sun.com/solaris/developer/support/driver/src/SUNWdrvs-8.html`.

[Sun05] Sun Microsystems, Inc.,
*Writing Device Drivers,*
`http://docs.sun.com/app/docs/doc/816-4854?a=load`, 2005.

[WG92] Niklaus Wirth and Jürg Gutknecht,
*Project Oberon: The Design of an Operating System and Compiler,*
Addison-Wesley, 1992.

# Appendix: source code

Below, we present selected parts of the Java class DeviceDriver, other auxiliary classes, and the RAM disk driver subclass, RamDiskDeviceDriver.

```
public abstract class DeviceDriver {

    private static DeviceDriver dd; //  The registered device driver

    public static void register(DeviceDriver d);

    ...

    // device driver call back methods – all follow the same pattern

    public static void do__init() { VM.returnFromCallbackInt(dd == null  ? 0 : dd._init()); }

    public static void do__fini() { VM.returnFromCallbackInt(dd == null  ? 0 : dd._fini()); }

    public static void do__info(Addr modinfop) {
        VM.returnFromCallbackInt(dd == null  ? 0 : dd_info(new Modinfo(modinfop))); }

    public static void do_probe(Addr dip) { ... }

    public static void do_attach(Addr dip,  int cmd) { ... }

    public static void do_detach(Addr dip, int cmd) { ... }

    public static void do_getinfo(Addr dip, int cmd, Addr arg,  Addr result) { ... }

    public static void do_open(Addr devp, int flag, int otype, Addr cred) {
        VM.returnFromCallbackInt(dd == null  ? 31 : dd.open(new Ptr(devp), flag, otype, new Cred(cred))); }

    public static void do_close(Addr dev, int flag, int otype, Addr cred) {
        VM.returnFromCallbackInt(dd == null  ? 0 : dd.close(new Dev(dev), flag, otype, new Cred(cred))); }

    public static void do_write(Addr dev, Addr uiop, Addr cred) { ... }

    public static void do_read(Addr dev, Addr uiop, Addr cred) { ... }

    public static void do_print(Addr dev, byte[] str) { ... }

    public static void do_strategy(Addr buf) { ... }

    public static void do_ioctl(Addr dev, int cmd, Addr arg, int mode, Addr cred, Addr rvalp) { ... }


    /* methods to be implemented by each device driver */

    public abstract int _init();

    public abstract int _info(Modinfo modinfop);

    public abstract int _fini();

    public abstract int probe(DevInfo dip);

    public abstract int attach(DevInfo dip, int cmd);

    public abstract int detach(DevInfo dip, int cmd);

    public abstract int getinfo(DevInfo dip, int cmd, Ptr arg, Ptr result);

    public abstract int open(Ptr devp, int flag, int otype, Cred cred);

    public abstract int close(Dev dev, int flag, int otype, Cred cred);

    public abstract int read(Dev dev, Uio uiop, Cred cred);

    public abstract int write(Dev dev, Uio uiop, Cred cred);
```

```
public abstract int strategy(Buf buf);
public abstract int ioctl(Dev dev, int cmd, Ptr arg, int mode, Cred cred, Ptr rvalp);
public abstract int print(Dev dev, String str);
...
/*
 * Constants from the Solaris kernel header files
 */
public static final int DDI_SUCCESS = 0;
public static final int DDI_FAILURE = -1;
...
public static class AddressWrapper { ... }
public static class Ptr extends AddressWrapper { ... }
public static class KMem extends AddressWrapper {
    public static KMem alloc(long size, int flag) { ... }
    public void free() {  ...  }
    public long size() { ... }
    public Addr seek(long offset) { ... }
}
// Wrappers for C kernel structs.  Class X wraps struct x.
public static class DkCinfo extends AddressWrapper { ... } // struct dk_cinfo
public static class DkGeom extends AddressWrapper { ... } // struct dk_geom
public static class Vtoc32 extends AddressWrapper { ... }
public static class Vtoc extends AddressWrapper { ... }
public static class Uio extends AddressWrapper { ... }
public static class Buf extends AddressWrapper { ... }
public static class Dev extends AddressWrapper { ... } // dev_t
public static class DevOps extends AddressWrapper { ... }
public static class DevInfo extends AddressWrapper { ... } // dev_info_t
public static interface PhysioInterface {
    public int strategy(Buf buf);
    public void mincnt(Buf buf);
    public int getinfo(DevInfo dip, int cmd, Ptr arg, Ptr result);
}
public static class Iovec extends AddressWrapper { ... }
public static class As extends AddressWrapper { ... }
public static class Proc extends AddressWrapper { ... }
public static class Cred extends AddressWrapper { ... } // cred_t
public static class Modinfo extends AddressWrapper { ... }
// support routines
```

```
    public static int physio(PhysioInterface physioi, Buf bp, Dev dev, int rw, Uio uio) { ... }
    public static int ddi_copyout(AddressWrapper from, AddressWrapper to, long size, int mode) { ... }
    public static int ddi_writeout(int val, AddressWrapper to, int mode) { ... }
    public static void bcopy(Addr src, Addr dst, long size) { ...  }
    public static  Addr kmem_alloc(long size, int flag) { return VM.kmem_alloc(size, flag); }
    public static void kmem_free(Addr p, long size) { VM.kmem_free(p, size); }
    public static void console_printf(String msg) { ... }
    public static void console_printf_long(long v) { ... }
}

public class RamDiskDeviceDriver extends DeviceDriver implements DeviceDriver.PhysioInterface {

    public static void main(String[] args) {
        DeviceDriver dd = new RamDiskDeviceDriver();
        DeviceDriver.register(dd);
        DeviceDriver.waitForCallbacks();
    }

    private static class Devstate {
        KMem        ram;
        long        rd_size;
        int         maxphys;
        DevInfo     dip;
        Vtoc        rd_vtoc;
        DkGeom      rd_dkg;
        DkCinfo     rd_ci;
    }

    private Devstate[] rsp = new Devstate[16];

    public int _init()                  { return 0; }
    public int _info(Modinfo modinfop)              { return 0; }
    public int _fini()                  { return 0; }

    public int probe(DevInfo dip)    { return nulldev(); }
    public int attach(DevInfo dip, int cmd) { ... }

    public int detach(DevInfo dip, int cmd) { ... }

    public int getinfo(DevInfo dip, int cmd, Ptr arg, Ptr result) {
        switch(cmd) {
        case DDI_INFO_DEVT2DEVINFO:
            result.store(rsp[new Dev(arg).getMinor()].dip);
            return DDI_SUCCESS;
        case DDI_INFO_DEVT2INSTANCE:
            result.store((long)(new Dev(arg).getMinor()));
            return DDI_SUCCESS;
        default
            return DDI_FAILURE;
        }
    }

    public int open(Ptr devp, int flag, int otype, Cred cred) {
        if (otype != OTYP_BLK && otype != OTYP_CHR)
            return EINVAL;
        return 0;
```

```
}

public int close(Dev dev, int flag, int otype, Cred cred) { ... }

...
public int read(Dev dev, Uio uiop, Cred cred) {
    int instance = dev.getMinor();
    if (uiop.getOffset() >= rsp[instance].rd_size)
        return EINVAL;
    return physio(this, null, dev, Buf.B_READ, uiop);
}
public int write(Dev dev, Uio uiop, Cred cred) {
    int instance = dev.getMinor();
    if (uiop.getOffset() >= rsp[instance].rd_size)
        return EINVAL;
    return physio(this, null, dev, Buf.B_WRITE, uiop);
}
public int ioctl(Dev dev, int cmd, Ptr arg, int mode, Cred cred, Ptr rvalp) {
    int minor;
    int error;
    int dkstate;

    minor = dev.getMinor();
    switch (cmd) {
    case VOLIOCINFO:
        /* pcfs does this to see if it needs to set PCFS_NOCHK */
        /* 0 means it should set it */
        return 0;
    case DKIOCGVTOC:
        switch (modelConvertFrom(mode & FMODELS)) {
        case DDI_MODEL_ILP32: {
            Vtoc32 vtoc32 = new Vtoc32(rsp[minor].rd_vtoc);
            vtoc32.allocInternal();
            vtoc32.sync();
            if (ddi_copyout(vtoc32, arg, Vtoc32.sizeofCStruct(), mode) != 0) {
                vtoc32.freeInternal();
                return EFAULT;
            }
            vtoc32.freeInternal();
        }
        break;

        case DDI_MODEL_NONE: {
            Vtoc vtoc = rsp[minor].rd_vtoc;
            vtoc.allocInternal();
            vtoc.sync();
            if (ddi_copyout(vtoc, arg, Vtoc.sizeofCStruct(), mode) != 0) {
                vtoc.freeInternal();
                return EFAULT;
            }
            vtoc.freeInternal();
            break;
        }
        } // end switch
        return 0;
    case DKIOCINFO: {
        DkCinfo ci = rsp[minor].rd_ci;
        ci.allocInternal();
        ci.sync();
        error = ddi_copyout(ci, arg, DkCinfo.sizeofCStruct(), mode);
        if (error != 0) {
            ci.freeInternal();
```

```
                return EFAULT;
            }
            ci.freeInternal();
            return 0;
        }
        case DKIOCG_VIRTGEOM:
        case DKIOCG_PHYGEOM:
        case DKIOCGGEOM: {
            DkGeom dkg = rsp[minor].rd_dkg;
            dkg.allocInternal();
            dkg.sync();
            error = ddi_copyout(dkg, arg, DkGeom.sizeofCStruct(), mode);
            if (error != 0) {
                dkg.freeInternal();
                return EFAULT;
            }
            dkg.freeInternal();
            return 0;
        }
        case DKIOCSTATE:
            /* the file is always there */
            dkstate = DKIO_INSERTED;
            error = ddi_writeout(dkstate, arg, mode); /* write an int to memory */
            if (error != 0)
                return EFAULT;
            return 0;
        default
            return ENOTTY;
        }
    }
    ...
}
```

## About the authors

Hiroshi Yamauchi is a graduate student at Purdue University. This work was performed during an internship at Sun Microsystems Laboratories. His research interests include virtual machines and compilers. He has B.Sc. and M.Sc. degrees in Computer Science from the University of Tokyo.

Mario Wolczko is a Distinguished Engineer at Sun Microsystems Laboratories. His current projects include computer architectures for object-based systems and performance instrumentation hardware design and usage. He joined Sun Labs in 1993, and has worked on the Self system, various research and production JVMs, and two SPARC microprocessors. His research interests include object-oriented programming language design and usage, the implementation of virtual machines (in both hardware and software), memory system design and management (including garbage collection) and the co-design of virtual machines and hardware architectures. He holds B.Sc., M.Sc., and Ph.D. degrees in Computer Science from the University of Manchester.