The Spotless System: Implementing a Java™ System for the Palm Connected Organizer

Antero Taivalsaari, Bill Bush, and Doug Simon

The Spotless System: Implementing a Java™ System for the Palm Connected Organizer

Antero Taivalsaari, Bill Bush, and Doug Simon

SMLI TR-99-73 February 1999

Abstract:

The majority of recent Java implementations have been focused on speed. There are, however, a large number of consumer and industrial devices and embedded systems that would benefit from a small Java implementation supporting the full bytecode set and dynamic class loading. In this report we describe the design and implementation of the Spotless system, which is based on a new Java virtual machine developed at Sun Labs and targeted specifically at small devices such as personal organizers, cellular telephones, and pagers. We also discuss a set of basic class libraries we developed that supports small applications, and describe the version of the Spotless system that runs on the Palm Connected Organizer.

M/S MTV29-01
901 San Antonio Road
Palo Alto, CA 94303-4900

email address:
antero.taivalsaari@eng.sun.com
bill.bush@eng.sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@eng.sun.com>.

The Spotless System: Implementing a Java™ System for the Palm Connected Organizer

Antero Taivalsaari
Bill Bush
Doug Simon

Sun Microsystems Laboratories
901 San Antonio Road
Palo Alto, CA 94303

1. Introduction

The majority of recent Java™ implementations have been focused on speed. In contrast, relatively little effort has been spent making Java implementations small. Yet there is a large number of consumer and industrial devices and embedded systems that would benefit from a small Java implementation supporting the full Java programming language [JLS96]. In these domains small system size is usually crucial, whereas speed, while certainly important, is often a secondary consideration. Also, for many consumer device manufacturers there are other criteria, such as portability and a fast learning curve for the developers, that are often more valuable than raw speed.

The Spotless system is a new Java implementation developed at Sun Microsystems Laboratories. It is based on our experience building JavaInJava [Tai98], probably the first Java virtual machine written in the Java programming language. We had two goals. First, we wanted to build the smallest possible "complete" JVM that would support the full bytecode set, class loading, and standard non-graphical libraries. Rather than speed, the main design criteria for the JVM were small size, portability, and readability of the code. Second, we wanted to engineer a small Java implementation that included basic classfile support for small applications.

The current Spotless JVM takes only a few tens of kilobytes (on the order of 30 to 50 kilobytes of static memory on a PC, depending upon compilation and debugging options). It runs anywhere from 30% to 80% of the speed of JDK™ 1.1 without the Just-In-Time (JIT) compiler, which is noteworthy given that there is no machine code in the source code, and that only a few simple optimization techniques are used.

The Spotless JVM supports platform-independent multithreading, copying or non-moving garbage collection (two different garbage collectors have been implemented), and optional quick bytecodes. But, perhaps more importantly, the source consists of only 25 C/C++ files containing approximately 14,000 lines of thoroughly commented, highly portable code. Various debugging and tracing options are provided to help porting efforts. This base version runs on Windows 95, Windows NT, and Solaris™.

In order to test the Spotless JVM on a real-world embedded device, we ported it to the Palm Connected Organizer. We also developed a small set of classfiles that support Palm applications. In this report we summarize our experiences building the JVM, making it fit on the Palm computing platform, and implementing the small class library.

2. Issues in implementing a small Java virtual machine

2.1. Design criteria for the Spotless JVM

The Java programming language was originally designed for use in various consumer devices and appliances such as cable TV set-top boxes and personal digital assistants. The goal was not just to develop a better programming language, but to develop an entirely new, more dynamic, platform-independent way of creating applications for these devices. Rather than having to build a hard-wired, platform-dependent application for each particular device, the Java programming language would allow the applications for these devices and appliances to be enhanced and extended dynamically. Ideally, the same applications would run unchanged in any device supporting the Java virtual machine.

Achieving this goal would open up completely new dimensions for application development for embedded systems, allowing, for example, mobile phone or set-top box manufacturers to develop extensible devices whose customers could flexibly enhance their systems as their needs would grow and new services would become available. This flexibility contrasts with the majority of today's devices, which have a fixed feature set that cannot easily be changed after the device is manufactured. Ultimately, the Java programming language could serve as the basis for a whole new class of extensible devices and appliances, each using a standard Java virtual machine and a fully compatible set of libraries, regardless of the differences in the underlying hardware.

Unfortunately, many consumer devices such as cellular telephones, pagers, wristwatches, bicycle computers, and radios still have far more limited hardware resources than is required by a typical Java virtual machine. Today, the RAM in the majority of embedded devices is still measured in kilobytes, whereas most Java systems typically require at least a few megabytes of RAM to run. Even though the typical amount of memory in consumer devices is constantly increasing, it will still take a long time for memory prices to decline to the point that megabytes of memory will be available in the majority of consumer devices. Furthermore, by the time this happens, it is possible that, as computing becomes more ubiquitous, an entirely new class of low-end devices with much more limited resources will already have emerged. In general, it is likely that at least for the next 3 to 5 years, the amount of RAM in the majority of embedded systems will still be measured in tens or hundreds of kilobytes rather than in megabytes.

The main objective of the Spotless project was to study implementation techniques that would allow the static and dynamic memory footprint of the Java programming language to be reduced substantially, preferably by at least an order of magnitude, without sacrificing the functionality of the Java programming language. In other words, we wanted to implement a small Java system that would need only a few tens or hundreds of kilobytes of memory to run, but that would nevertheless support the complete bytecode set, dynamic class loading, garbage collection, multithreading, and other essential features of the Java virtual machine. Other central goals were portability and ease of understanding.

These requirements contrast with some other embedded Java virtual machine projects that have tried to reduce the memory footprint by removing various language features, such as floating point arithmetic, multithreading or garbage collection (Java Card™), and by replacing dynamic class loading with precompiled, preloaded classes, possibly stored in ROM (Java Card and EmbeddedJava™). While such restrictions are reasonable for many platforms, we felt that there is a huge potential class of devices and applications in which it is crucial to support dynamic class loading and the complete bytecode set. After all, it is the dynamic nature, extensibility, and portability of the platform that is the main selling point of the Java programming language to many embedded systems developers and customers. Furthermore, dynamic class loading is highly desirable for devices implementing the Jini™ distributed architecture.

In general, unlike other embedded Java system development projects, we were unwilling to sacrifice the completeness of the virtual machine. In contrast, we were willing to simplify and reduce the libraries, provided that this could be done in a fashion that would preserve upward compatibility between applications for small systems and for large platforms.

2.2. Why are Java systems so large?

Current Java systems require on the order of a few megabytes to a few tens of megabytes of memory to run. In spite of this apparently large memory footprint, a Java virtual machine is really a relatively simple engine. The bytecode set of a Java virtual machine is fairly compact, and the runtime support needed in a basic virtual machine is relatively small. Nor is there inherently any need for special memory areas other than the standard memory heap(s). Certainly a Java virtual machine is much more complex and memory-consuming than, for example, a Forth programming environment [Bro84], but need not be much more complicated than virtual machines for more closely related programming languages, such as Smalltalk-80 [GoR83] or Self [UnS87]. Also, the general implementation techniques needed for building virtual machines have been studied for two decades and are quite well understood.

Then why do JVMs require so much memory? One explanation is the desire for speed. Modern Just-In-Time (JIT) compilers are pushing the limits of implementation technology, introducing a lot of additional complexity to the virtual machine and often requiring many additional specialized memory regions. Also, the use of native threading and pre-emptive scheduling introduces extra complexity due to mutual exclusion and synchronization. Furthermore, modern high-speed garbage collectors can be fairly complex, usually requiring several separate memory spaces in order to manage multiple generations of objects.

However, by far the most important reason for the large memory consumption of Java systems is the size of the standard class libraries. For instance, the extensive internationalization features that have been built in the standard I/O libraries do not come without a cost. In fact, the majority of the approximately 130,000 bytecodes that are executed during initialization of the JDK 1.1 JVM are spent on initializing the various internationalization and I/O facilities. Similarly, the various portability, security, reflection, and remote invocation features all add extra layers to the libraries, meaning that more and more classes need to be loaded even when invoking seemingly trivial operations. For instance, in JDK 1.1, printing out one string of text requires 20 to 30 classes to be loaded. Even though these advanced features are extremely important and valuable in many applications, their cost is prohibitive for most embedded systems. The presence of a large number of library classes also means that many native functions are needed to implement the platform-specific functionality needed by the library classes, even though a typical application uses only a small fraction of those functions.

Part of the high memory consumption of Java systems can also be explained by the way classfiles are stored. A classfile is a portable, device-independent representation of a class. As such, it is not efficient as a runtime structure. When classfiles are loaded into a Java virtual machine, it has to create more efficient internal representations (such as method tables and field tables to perform efficient runtime lookups) for every class. However, since bytecodes use classfile-specific constant pool indices, it also has to maintain the constant pool information that was originally loaded from each classfile, even though much of the same information is also stored in the method and field tables. This means that a straightforward JVM implementation may inherently have much internal redundancy.

2.3. Building a smaller Java system

In the previous section we argued that the Java virtual machine need not be very complicated or particularly large. If one follows good software engineering principles and practices, and adheres to basic implementa-

tion techniques, it appears actually rather easy to write a JVM that would be considerably smaller than a hundred kilobytes (excluding possible C/C++ runtime libraries). Such a straightforward bytecode interpreter implementation of the JVM might not be very fast, but there are ways to improve performance without requiring much additional memory.

Similarly, by carefully employing a few simple techniques, it is possible to greatly shrink the class libraries.

(1) Start with nothing and add only what is absolutely necessary, rather than starting with the complete JDK and removing what was not needed. Much of the standard JDK is simply not too big to fit on a memory-limited platform.

(2) Remove dependencies between classes. The standard JDK has many dependencies, and, if one method is needed from a class, the entire class has to be loaded. Dependencies were removed by:

- folding classes together, such as System and Runtime;
- extending core classes, such as adding isInteger and toInteger to String, thereby avoiding the need for the Integer class;
- removing pass through functions, such as System::exit, which calls Runtime::exit; and
- inlining calls, such as calls to System::arrayCopy.

Also, starting with nothing makes it easier to discover dependencies in an incremental fashion and deal with them appropriately.

(3) Remove alternate forms of functions; for example, String(char[]) is simply a call to String(char[], 0, <length-of-char-array>). Note that, with smart inlining in a performance-oriented implementation, much of the overhead of these forms is eliminated.

(4) Eliminate classes that are seldom used or that can be added by the user if necessary, such as ThreadGroup.

(5) Avoid classes that create many short term objects, such as Point (by, for example, using x and y coordinates as parameters rather than Point objects).

(6) Reduce the number of distinct exceptions to the bare minimum, because each exception is its own class with all the associated overhead.

(7) Take advantage of native platform support for I/O and graphics. AWT in particular is inappropriate and too large for embedded platforms.

The ideas described above were used in the development of the Spotless system. We started by building a straightforward, small, consistently written, low-performance bytecode interpreter. When we ported it to the Palm Connected Organizer we initially allowed host system functions (PalmOS functions in our case) to be called directly from the virtual machine to speed up development and to study memory constraints and other platform-specific restrictions in more detail. After getting the basic system to run, we started studying space-efficient optimization techniques to improve the performance of the virtual machine, and, in parallel, started implementing a subset of the libraries that would be compatible with the standard libraries but with a substantially smaller memory footprint. The results of this work are described below.

3. Overall design of the Spotless JVM

The overall design of the Spotless JVM is similar to that of our JavaInJava virtual machine [Tai98]. Like JavaInJava, the Spotless JVM is a new implementation based on the Red Book specification [JVM96]. The virtual machine is implemented around a straightforward bytecode interpreter with some optimizations. The

representation of the internal structures is conventional in the sense that each class has a constant pool, method table, field table and interface table (see Figure 1). However, unlike JavaInJava, where every internal structure is implemented as a separate object, in the Spotless JVM most of the structures have been implemented as linear tables to provide faster access and to conserve memory space. Also, unlike JavaInJava, we have also taken advantage of various low-level features of the C programming language to achieve better performance.

Portability was one of the main goals in the Spotless JVM design. For this reason, non-portable code (such as functions to obtain information about the host computing system) has been minimized and limited to a single file. Even multithreading and the garbage collector have been implemented in a completely platform-independent fashion. Rather than relying on external interrupts, the multitasker performs thread switching on the basis of the number of bytecodes the current thread has executed, making thread switching more deterministic and easier to debug, thus facilitating porting efforts. This approach is practical in part because we are not targeting multiprocessor platforms.

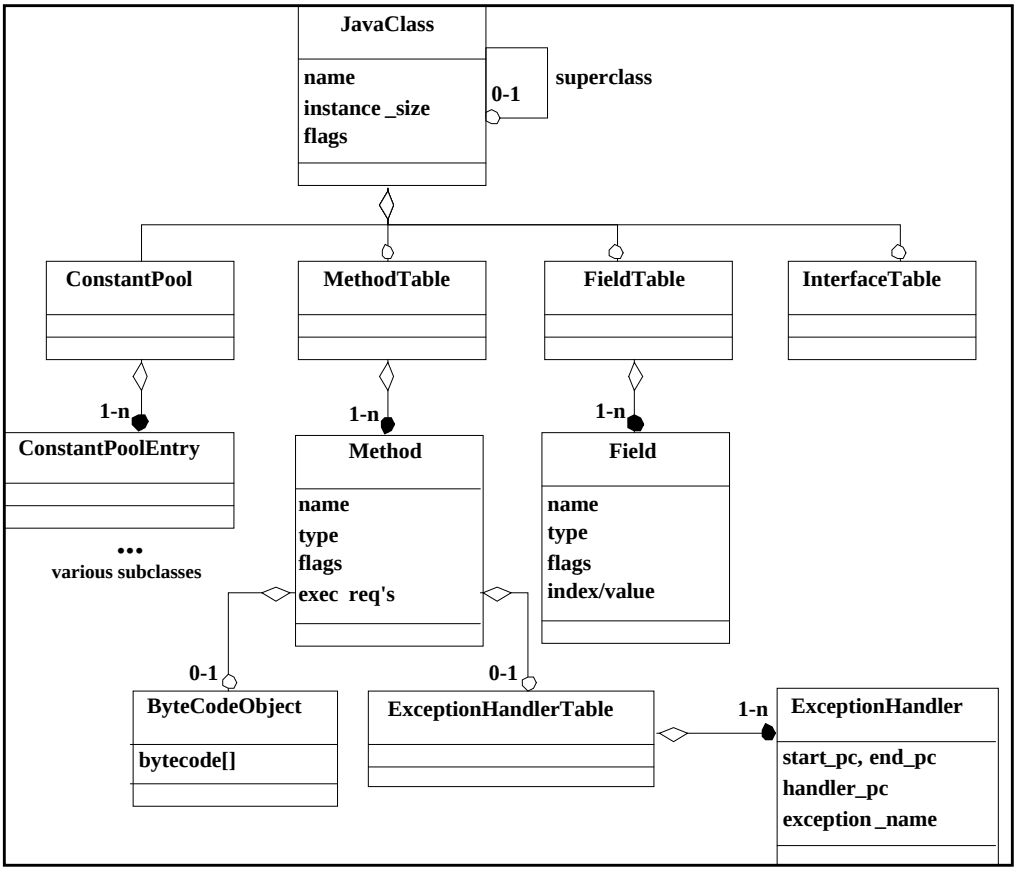

Figure 1: Internal class representation.

Garbage collector(s). Two different garbage collectors have been implemented. The original collector used a copying garbage collection algorithm with multiple memory spaces. However, this approach turned out to consume far too much memory for devices that have only a few tens of kilobytes of memory available. In general, modern generational garbage collection algorithms tend to be unsuitable for applications with extremely limited memory. For this reason, a simpler, non-moving, single-space mark-and-sweep garbage collector was written. The new collector operates well with heap sizes of just a few tens of kilobytes. Both collectors are handle-free, that is, object references are always direct rather than indirect.

Native function support. To minimize memory usage, the native functions and wrappers used to call host system functions have been implemented by making them a part of the virtual machine, rather than by using the Java Native Interface. Currently, the extent of native function support varies by platform. On Windows and Solaris it is extensive enough to cover most non-graphical libraries; due to size constraints it is very limited on the Palm organizer. There is no support for AWT, Swing, or any other graphics libraries, so graphics are done by calling platform-specific graphics functions.

C/C++ runtime libraries. The Spotless JVM has been implemented primarily in C, with small pieces in C++. The system is thus somewhat dependent on the C runtime libraries. This could be a problem on small platforms that do not have support for C libraries. However, the number of C runtime functions that the virtual machine calls has been minimized, to ease implementation or replacement of those functions. If any of the debugging modes in the source code are enabled, then the target platform must support the fprintf function in the standard C I/O library (stdio), or a function with the same interface as fprintf.

The Spotless JVM source is usually compiled using a target-system-specific C cross-compiler rather than a regular compiler. Development for the Palm organizer was done on PCs, for instance. It should be remembered that many embedded systems platforms do not have support for all the features of the C language (for instance, floating point support is not available on many small platforms), and hence the cross-compiler might not be able to generate support for certain bytecodes.

Compilation options; optimizations. The Spotless JVM source code includes many compile-time options for tuning virtual machine parameters (such as the size of the internal stacks), for debugging (to help porting efforts), and for choosing various optimizations (size versus speed). Most of the optimizations in the Spotless JVM are focused on minimizing space. In general, we would like to minimize the amount of memory required without slowing down the virtual machine, but in some situations the virtual machine developer will have to choose between size and speed.

4. The Spotless system on the Palm Connected Organizer

Because of its small size, our original target device candidate for the Spotless system was the PC-card-sized Rolodex REX personal organizer sold by Franklin Electronic Publishers (http://www.franklin.com/rex/). However, development tools for that device were not readily available, so we chose the Palm Connected Organizer by 3Com, because it is very popular, is well supported in terms of cross-platform development, and is a typical embedded device with limited dynamic memory.

The Spotless system for the Palm organizer is composed of five major components:

- the virtual machine itself,
- a small set of class libraries,
- a database and user interface for storing and managing classfiles,
- utilities for moving classfiles onto the organizer from a desktop machine, and
- demo applications that run on the Palm organizer.

Spotless JVM on the Palm organizer. Porting the Spotless JVM to the Palm computing platform was relatively straightforward using version 4 of the Metrowerks Palm development environment. Some data alignment problems were encountered. The JVM was originally designed to run on devices with four-byte (32-bit) data alignment, whereas the Palm organizer uses two-byte (16-bit) data alignment by default. Because of this, many four-byte read and write operations failed or caused unexpected results.

The limited memory model of the Palm computing platform was also problematic. Even though the Palm devices can have megabytes of RAM, the amount of real, directly addressable, dynamic RAM is small. The Palm computing platform supports a maximum of 96 kilobytes of dynamic RAM. A PalmPilot Professional device has only 64 kilobytes of dynamic RAM; older PalmPilot devices have only 32 kilobytes. The rest of the memory, which is static RAM, behaves like a RAM disk and has to be accessed using special PalmOS database API calls. We originally planned to use the static RAM as virtual memory for the Spotless JVM, but unfortunately the database API calls turned out to be so slow that this was not practical. Since the PalmOS system (the operating system of the Palm computing platform) itself normally uses 10-20 kilobytes of dynamic RAM, the current Palm Spotless JVM has less than 40 kilobytes for the heap on the PalmPilot Professional and less than 64 kilobytes on the Palm III. This imposes serious restrictions on the kinds of applications that can be run.

The PalmOS system has no direct support for standard C or C++ libraries. Many of the individual library functions are supported, but the names of the functions differ from those in the standard libraries (for example, strcpy is StrCopy, and sprintf is StrPrintF). Also, the PalmOS does not have support for C or C++ style input/output (such as the stdio library). For this reason, the PalmOS port of the VM includes its own support for fprintf (including the standard output streams stdout and stderr), which made debugging much easier. Debugging was also assisted by developing on the Palm OS Emulator (freely available from 3Com at http://www.palm.com/devzone/pose/pose.html), which is integrated nicely with Release 4 of Metrowerks Codewarrior for PalmOS.

The current Palm Spotless JVM supports the complete bytecode set, full class loading, garbage collection, and various Palm native functions. The total size of the executable (PRC file) containing the virtual machine, the user interface (discussed below), and native function support for the small class libraries is currently 40 kilobytes.

Some memory reducing techniques were added to the Palm Spotless JVM. Lazy class loading is performed, in which a class is loaded only during execution when one of its methods is needed; only Object and Class are loaded at initialization. Minimal perfect hashing is used for the native function table, so that the table has no empty or chained entries. Duplicate strings in loaded classes are eliminated by using a canonical string pool, which on average saves roughly 50% of the memory used for string storage.

A small set of class libraries. A minimized set of class libraries supporting the Palm platform was developed, including a very small subset of java.lang and a set of new libraries specific to the Palm computing platform (but of more general utility for personal organizers and perhaps small devices generally), called spotless.

java.lang.Object. Object is required as the root of the object hierarchy. A number of its methods have been retained, as they are useful even on a small platform. The getClass, toString, and hashcode methods can be used for debugging at the source level using printf style debugging. Also, hashcode and equals provide a good base for a simple hash table implementation. The remaining methods all involve thread synchronization. Combined with the Thread class, these methods provide the threading capabilities that are integral to the Java environment.

java.lang.Class. Class is fundamental and has to be supported, although it is unclear how often its reflective capabilities are used. Given that the majority of its methods are native, it is easier to include it in the core set of classes, since adding native methods later requires access to the VM source code.

java.lang.String and *java.lang.StringBuffer*. Both classes are used by the javac compiler to implement string constants, and are present in code that uses string constant expressions even if they are not explicitly used

in the source. In the Spotless system, String does all the conversion of primitive types to and from Strings that is normally done by the primitive wrapper classes (such as Integer and Boolean). For example, new methods in String provide Integer conversion functionality (specifically isInteger and toInteger). StringBuffer uses these conversion facilities in its versions of append. The append methods are the primary reason StringBuffer has to be provided in addition to String. These methods are generated by javac to implement expressions such as *str = "Age: " + age;*.

java.lang.Runtime. This class is critical in a memory-limited embedded environment. It is a combination of the standard Runtime and System classes. It provides methods for examining the runtime environment. It is particularly useful in the absence of the OutOfMemoryError class. The currentTime method differs slightly from the currentTimeMillis found in java.lang.System in that it returns the time elapsed since VM initialization instead of some well known epoch. This is because the fine grain timer of the underlying OS is reset upon each OS reset, rather than set to some absolute time. It also stops while the device is in sleep mode. The Runtime class also provides a method to return an optionally seeded Random number.

java.lang.Thread. This class provides basic threading functionality. Threads can be started, put to sleep, and yielded. There are also static methods that return the number of threads in the system and return a handle on the current thread. There are no methods to stop a thread explicitly, as these have unclear semantics and have been deprecated as of JDK 1.2. In keeping with our general minimalist philosophy, no threads are created implicitly, and there is no support for thread groups, which are too heavyweight and can be created for specific applications if needed. Threading is nonetheless of great value because it greatly simplifies event driven applications.

java.lang.Throwable, java.lang.Exception, java.lang.RuntimeException, java.lang.NullPointerException, java.lang.IndexOutOfBoundsException, and *java.lang.Error*. IndexOutOfBoundsException and NullPointerException are the core exceptions. The other classes are required by javac when code uses exception handling. Applications can also define their own exceptions, but should do so only when necessary, because exception handlers consume relatively large amounts of memory. In all other error cases, an error message is printed, indicating the class, method, and bytecode offset where the error occurred. Future plans to reorganize the memory management of classes will most likely result in more of the standard exceptions being supported.

java.lang.Runnable. This interface allows objects to run in a thread without their class having to extend the Thread class.

As mentioned above, in addition to the java.lang classes, there are classes supporting a Spotless application framework. These classes provide a simple model for event handling and simple graphics capability.

spotless.Spotlet. The Spotlet class implements pen-based event handling. It consists of handler methods that are invoked when associated events occurs. The handler methods are penDown, penUp, penMove, and keyDown. Applications using this class are called *spotlets*. Spotlets must register with the Spotlet class to get and relinquish the event focus, using the register and unregister methods. Event handlers run in a dedicated VM thread. The VM remains alive if a spotlet is registered, even if no events are being processed. The VM will conserver power by putting the Palm device to sleep after an interval with no activity.

spotless.Graphics. The Graphics class provides straightforward access to the underlying functionality of the Palm platform. Methods exist to draw shapes (drawLine, drawRectangle, and drawBorder), display strings (drawString), get display-specific properties (getHeight and getWidth), set a clipping rectangle (setDrawRegion and resetDrawRegion), move a region (copyRegion), and draw bitmaps (drawBitmap). Numerous drawing modes are available; plain, grayscale, inverted, and erase.

spotless.Bitmap. The Bitmap class is used to represent simple black and white bitmaps. Having a bitmap object simplifies the use of bitmaps, since the bitmap data (pixels) need only be specified once (during construction of a Bitmap object) as opposed to every time a bitmap is drawn.

Database and user interface for storing and managing classfiles. One of our main goals in developing the Spotless JVM was to implement a small virtual machine that would support dynamic loading of classfiles. However, the Palm computing platform does not have a file system. For this reason, we had to implement our own database for storing and managing classfiles. This was accomplished by using the database and heap APIs of the PalmOS libraries. Rather than storing classfiles in a normal file system, the classfiles are stored in the RAM database located in the static RAM of the Palm device. In order to make it possible to manage the Palm classfile database, we also had to implement a simple user interface, known as the *class manager*, for displaying, invoking, and deleting classes and packages. Screen snapshots illustrating the user interface are shown in Section 5.

Classfile download utilities. Ideally, we would like to have a complete, integrated Java programming environment for the Palm computing platform, allowing classes to be written, compiled, debugged, and executed on the Palm organizer itself. Unfortunately, like many other small devices, the Palm organizer does not have enough memory to support a complete programming environment. Furthermore, writing source code with a pen instead of a keyboard would be tedious. The small screen size would also make the editing of source code rather difficult. In order to avoid these problems, software development takes place on a desktop machine such as a PC or a UNIX workstation. Regular Java development tools can be used for writing source code and compiling the source files into classfiles. Classfiles are then transferred to the Palm device using one of two mechanisms.

HotSync classfile data conduit. The HotSync classfile data conduit we implemented is an extension of the Palm computing platform's standard HotSync software used for synchronizing an organizer with a desktop computer. The PC version of the data conduit is implemented as a dynamically linked library (DLL) invoked automatically during a HotSync operation. When invoked, the conduit transfers new or updated classfiles from the Palm software directory on the PC to the classfile database on the Palm organizer.

Database packager. The database packager is simpler than the data conduit. The packager, called *mkpdb* and written in Java, packages classfiles into a pdb (Palm data base) file that can be installed with the Palm Install tool, just like a prc (application binary) file.

At some point either utility may be extended to perform standard bytecode verification and annotate classes for the class manager (indicating whether a class is executable—has a main entry point, or is a system class).

Demo applications. Various demo programs have been implemented, illustrating the capabilities of the Palm Spotless JVM. These programs range from simple graphical and numerical tests to fractals and three-dimensional graphics.

5. Using the Spotless JVM on the Palm Connected Organizer

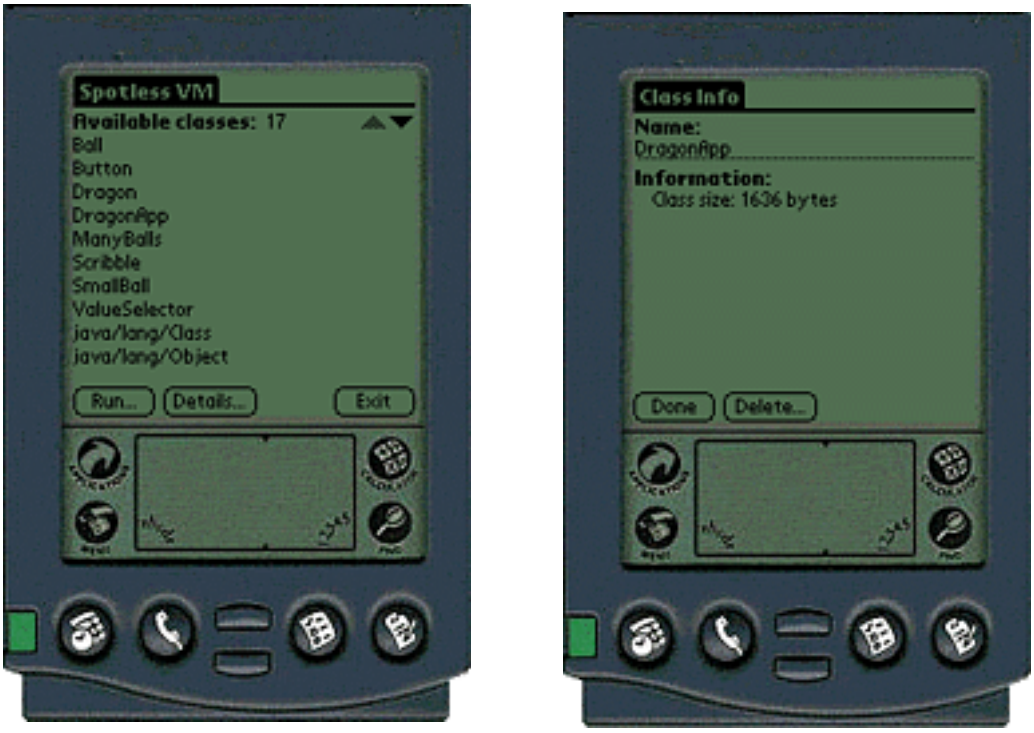

Figure 2: Spotless class manager display and class details dialog.

As discussed above, the Palm version of the Spotless JVM is built around an application known as the *class manager*. The class manager serves as the primary interface to classfiles on the Palm organizer, allowing the user to see which classes and packages are available, launch Java applications, see details of classes, and delete them as necessary.

The primary class manager display, entitled “Spotless VM”, appears on the left in Figure 2, listing the currently loaded classfiles. The “Class Info” dialog appears on the right, showing the name and the size of a selected classfile, and providing the option to delete it.

To run a classfile, the user selects it from the list of available ones and presses the “Run” button. Another display, entitled “Run Class”, then appears, and is pictured in Figure 3. This display allows the user to supply options and command line parameters to the virtual machine, and, if the “Verbose” option is turned on, follow the initialization progress of the virtual machine. The “Scroll Delay” list, shown in expanded form on the left in Figure 3, is used to control scrolling of the standard output stream during execution. The “ViewOutput” button opens another display, shown in Figure 4, that allows the user to view the contents of the two output streams, stdout and stderr, after execution. These buffers are cleared when the “Cancel” button is pressed and the user returns to the primary class manager display.

Figure 3: Class invocation dialog before execution (left) and after execution (right).

Figure 4: Output stream display after execution.

Figure 5: Java applications running on the Palm organizer.

Figure 5 shows two Java programs (DragonApp and Scribble) running on the Palm organizer. These programs use the spotless application framework presented earlier. Each program is a single spotlet; the user can switch between the two by pressing the arrow in the lower right hand corner.

6. Current status and experiences

The design and implementation of the Spotless JVM was started in February 1998. The first version of it ran in April, and a suite of small demonstration programs were running in May. Garbage collection was implemented in June. The minimized class libraries were working in November.

One of the main things we learned during the implementation of the Spotless system is that by restricting the use of the standard libraries, a Java system can easily fit in a few tens of kilobytes.

Since performance was never a major consideration for us, we do not have detailed benchmarks. Being a pure bytecode interpreter, the Spotless JVM cannot compete with virtual machines with a JIT compiler. The Windows 95/NT version runs between 30% and 80% of the speed of Sun's standard JDK 1.1 on Windows 95/NT without JIT, usually performing better on large applications and worse on those that involve a lot of numerical computation. This is reasonable, given that there is no machine code in the Spotless JVM source code, and that internal virtual machine registers—such as the instruction pointer and the stack pointers—are stored in regular C variables rather than in machine registers. By adding a few dozen lines of machine code to the most critical locations in the source code, performance could probably be substantially improved.

Measuring and comparing the performance of the Palm Spotless JVM is even more problematic, because there are no real points of reference. Since the PalmPilot Professional model we used for measurement only has a 16 MHz Motorola Dragonball (68328) processor with a 16-bit external data bus, the performance of the virtual machine on it is nowhere near its performance on a modern PC or UNIX workstation. However, the performance of the simple graphics programs seems surprisingly snappy. An empty loop (100,000 iterations) takes about 15 seconds to execute on the PalmPilot Professional—about five times slower than in Ruka, a simple Forth-like threaded code interpreter that we wrote earlier to understand the memory and performance limitations of the Palm device. This difference appears to be due to the fact that Java compilers (javac in particular) generate four bytecodes (IINC, ILOAD, LDC, IF_ICMPLT) for a simple loop control structure, whereas a Forth-like interpreter typically uses a single instruction for performing the same operation.

7. Future directions

The Spotless system is a fairly complete JVM implementation with a high quality source code base, and has enough library support to enable the writing of small Java applications. Nonetheless, it is still an experimental system missing various JDK features. As such, the system may be used as the basis for more research and experimentation.

On the research side we are interested in carefully adding functionality to both the java.lang subset libraries and to the spotless library. We are also interested in designing and implementing more interactive development environments for small devices, to avoid the lengthy edit-compile-download-run cycle. There are various other relevant research areas, such as the real-time aspects of an embedded JVM, that should be studied in detail.

On the more practical side, a Palm database access framework and explicit class unloading would be very useful for writing more practical applications. Better performance could be obtained by rewriting the interpreter loop in assembler. It would also be interesting to rewrite some of the standard Palm applications (most of which are small) in the Java programming language.

We are also interested in porting the Spotless JVM to other devices such as the Rolodex REX organizer or palm-sized PCs. In general, there are many potential devices, including cell phones, pagers, credit card readers, and personal organizers, that could be target devices for porting efforts.

8. Conclusions

In this paper we have summarized our experiences in building the Spotless system, a Java system for the Palm Connected Organizer. We described the general design constraints for a small Java system, presented the overall design of the Spotless JVM and class libraries, and then discussed our experiences building the Spotless system for the Palm organizer. Screen snapshots illustrating the user interface built for the virtual machine were also included.

The Spotless project shows that a Java system does not have to be particularly large or complicated. In general, the results of the project have been encouraging, since they show that it is relatively easy to build a JVM that fits in a small device with limited memory, such as the Palm organizer. The Spotless JVM is based on a conceptually clean and simple overall design and has a portable, well commented source code base that contains various options and hooks to help porting efforts. The project has also shown that the Java libraries can be both reduced and augmented to support programming on embedded devices.

9. References

Bro84 Brodie, L., *Thinking Forth: A Language and Philosophy for Solving Problems*. Prentice-Hall, 1984 (2nd edition 1994).

GoR83 Goldberg, A., Robson, D., *Smalltalk-80: the language and its implementation*. Addison-Wesley, 1983.

JLS96 Gosling, J., Joy, B., Steele, G., *The Java Language Specification*. Addison-Wesley, 1996.

JVM96 Lindholm, T., Yellin, F., *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

Tai98 Taivalsaari, A., *Implementing a Java Virtual Machine in the Java Programming Language*, Technical Report SMLI-98-64, Sun Microsystems Laboratories, March 1998 (23 pages).

UnS87 Ungar, D., Smith, R.B., "Self: the power of simplicity". In Meyrowitz, N. (ed): *OOPSLA'87 Conference Proceedings* (Orlando, Florida, October 4-8), ACM SIGPLAN Notices vol 22, nr 12 (Dec) 1987. pp.227-241.

About the Authors

Antero Taivalsaari is a staff engineer at Sun Microsystems Laboratories in Mountain View, California, where he is studying new implementation techniques for object-oriented programming languages. His research interests include object-oriented programming and design, implementation of interactive programming languages, mobile and wireless systems, and collaborative software engineering environments. He has published a number of research papers in international journals and conferences, given invited lectures on various topics, and organized several international workshops in the field of object-oriented programming.

Before joining Sun Microsystems Laboratories in August 1997, Antero worked for four years at Nokia Research Center in Helsinki, Finland, where he lead a research group of ten people implementing an advanced collaborative software design environment, and managed some of Nokia's international research projects. Prior to this, he worked for several years in the academic world, and completed a doctoral degree in computer science at the University of Jyväskylä, Finland, in 1993, after spending one and a half years as a guest researcher at Concordia University and University of Victoria in Canada. His doctoral thesis, entitled "A Critical View of Inheritance and Reusability in Object-Oriented Programming," was judged the best doctoral dissertation in computer science in Finland in 1994.

Bill Bush is a staff engineer at Sun Microsystems Laboratories, where he has worked on various implementations of the Java programming language. His research interests include programming language design and implementation, program analysis, computer-aided design, computer architecture, and software engineering, in which areas he has published papers.

Before joining Sun, Bill was a founder and principal scientist of Intrinsa Corporation, where he co-invented a new analysis technique for detecting programming errors in source code. Before that he worked on research projects at Harvard and U.C. Berkeley, from which he received a Ph.D. in computer science.

Doug Simon is an intern with the Kanban group in Sun Labs. During his internship he researched and implemented a path profiling algorithm, and used it to collect profiles of Java methods at the bytecode level, before working on the Spotless system.

Before coming to Sun, Doug completed a Bachelor's degree in Information Technology at the University of Queensland, graduating with first class honors. As part of this degree he completed a thesis entitled "Structuring assembly programs" which presented techniques for translating assembly code into a higher level representation closely resembling C code. He plans to start work on a Ph.D. in 1999.