Performance Optimization of Embedded Java Applications by a C/Java-hybrid Architecture

Wolfgang Klingauf[1], Lorenz Witte[2], Ulrich Golze[3]
Tech. Univ. Braunschweig (E.I.S.), Mühlenpfordtstr. 23, D-38106 Braunschweig, Germany

[1] ☎ +49 531 3913105
klingauf@eis.cs.tu-bs.de

[2] ☎ +49 511 3522572
lwitte@eis.cs.tu-bs.de

[3] ☎ +49 531 3912389
golze@eis.cs.tu-bs.de

Abstract. We present a software architecture for Java VMs and APIs that significantly increases the execution performance of Java applications on devices with considerable storage limitations. While present embedded Java devices usually implement standard API functions entirely in Java, we propose a shift of paradigm. By using a native programming language like C to implement the whole system library and by using a high-level C-to-Java interface, we obtain embedded Java applications running almost at the speed of fully native applications. Moreover, a drastic reduction of memory can be achieved. Thus, Java runs efficiently on low-cost devices with a memory size of less than 2 MB. Besides, our technique requires no changes of Java user applications.

1 Introduction

The use of Java on embedded systems has grown tremendously over the last years. While in 2002, 15 million units of mobile phones and other handheld devices where shipped with embedded Java, in 2003 it already were nearly 75 million units [12].

Java offers several serious advantages as compared with other programming languages for embedded systems. First, the portability of Java is attractive for reducing the cost of application development. Second, Java supports dynamic loading of applications. Together with other benefits of Java, these features can significantly contribute to forthcoming applications in the field of mobile and ubiquitous computing. For example, Java enables smart-phone users to download applications directly from the Internet, independently from the manufacturer.

However, due to limitations of processing power, energy consumption, size and cost, embedded devices cannot afford the low performance of purely interpreted Java.

As current performance optimization efforts focus on caching frequently used code fragments, so called hot spots, in a machine code buffer, this approach is not suitable for embedded devices with small memory resources such as mobile phones. As a reference, Sun's smallest commercial Java virtual machine with support for the hot spot technology, the CLDC HotSpot VM, requires at least 8 MB of ROM/Flash and 1 MB of free RAM memory [7]. This is quite expensive for just obtaining the capability of executing Java games and applications. Indeed, most of the currently used VMs in embedded devices are based on Sun's KVM and similar implementations [4, 5, 7, 11]. These VMs do not include any code caching capabilities [1, 2, 8].

In the present work, we introduce a software architecture for Java VMs and APIs that embeds native library functions into a highly portable three layer C/Java-hybrid architecture which requires no changes of Java user applications. We show that our approach not only significantly outperforms the KVM execution performance, but also surpasses the CLDC HotSpot VM.

This paper is organized as follows. We characterize the C/Java-hybrid architecture in the next section. Section 3 describes the experimental setup and presents detailed benchmark data. Section 4 compares our approach to other speed-optimization techniques. Finally, we summarize our conclusions and describe future work in section 5.

2 C/Java-hybrid Architecture

The virtual machine of the Java runtime environment for a specific device is usually implemented in a native language like C. In contrast, the various system and user libraries are largely implemented in pure Java [6, 8]. As Java applications typically spend 80 to 90 percent of time executing libraries [5] and as Java byte code interpretation is

considered as the performance bottleneck on embedded Java devices [6], our approach is to implement as many library functions as possible in C. In order to keep this code still portable and thus re-usable on other platforms, we propose a partitioning of these libraries into three layers. By applying component-based methodologies to this approach and by minimizing the relationship between the separate layers, a highly portable C/Java-hybrid architecture is obtained.

2.1 Structure

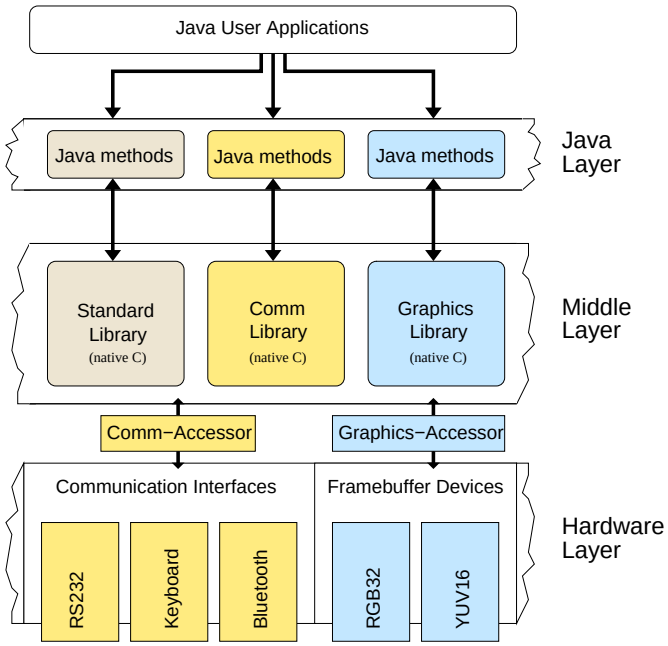

Figure 1. C/Java-hybrid architecture

Figure 1 shows the main components of the C/Java-hybrid architecture. It consists of three layers:

- The **Hardware Layer** represents the platform dependent part of the architecture. It is organized in groups of modules sharing the same functionality. For example, a group for external communication could contain modules for Bluetooth, RS232, and a keyboard. Each group is accompanied by an accessor helping upper layer functions to use the system hardware in an object-oriented manner. For this purpose, the accessors define a set of functions to be implemented by all modules of its group.
- The **Middle Layer** provides all native functions necessary to assemble the API functions in the Java Layer. It combines both simple and complex functions and algorithms and is fully platform independent. For example, a set of communication library functions would be implemented in this layer.
- The **Java Layer** represents the C-to-Java interface and thus the connection between the VM and the C/Java-hybrid architecture. It acts as a broker between the Java user applications and the Middle Layer and conceals the native implementation of its Java classes and methods from the application.

2.2 Library Function Execution

Java applications call native library functions through regular Java method calls. Utilizing the VM-dependent Java-native interface (e.g. Sun's K Native Interface [10]), our Java layer enables the library functions to access data fields of the application. The execution of a library function from the Java layer splits up into two phases:

1. Configuration phase - The library is supplied with the data enabling it to select appropriate hardware modules and configure them.
2. Execution phase - The actual execution of the library function is performed.

If the VM does not provide the capability to interrupt the execution of native code, this process runs atomically and blocks concurrent threads.

2.3 Module Selection

The selection of a hardware module is taken care of by the corresponding accessor. In the configuration phase, the accessor creates a temporary configuration structure. This structure is required by subsequent library calls as it contains information about the selected hardware module.

Following calls to the accessor functions are mapped to the corresponding functions of the selected hardware module. To handle this task efficiently, function lookup tables (FLUTs) are used. The modules each contain a FLUT holding the references to their own set of functions.

2.4 Example

To provide an example of the functionality of our architecture, we imagine a wireless MIDP device that is going to send audio data via Bluetooth to another device. For this purpose, the application creates an OutputStream object which is connected to the Bluetooth interface. The write method of OutputStream is located in the Middle Layer and implements the

Bluetooth audio profile. Hardware Connection Interface (HCI) functionality is provided by the Bluetooth module in the Hardware Layer.

Upon call, the Middle Layer write function obtains a configuration structure from the Comm-Accessor which allows it to access the Bluetooth hardware module. Then the audio stream is prepared for transmission, according to the Bluetooth audio profile specification. Finally, this data is passed to the Bluetooth hardware module through the Comm-Accessor.

3 Experimental Results

This section compares C and Java performance and reports the current performance of the C/Java-hybrid architecture. To this end, we have benchmarked various functions typically on embedded systems. These include simple algorithms as used when sorting address book entries on a mobile phone and more complex algorithms such as drawing a complete graphical user interface (GUI). In the endeavor to obtain highly comparable results, we have tried to produce corresponding code as similar as possible. For example, we have avoided the use of dynamic arrays, since the Java version would then suffer from dynamic heap allocation followed by garbage collection.

3.1 Experimental Setup

All tests were run under Linux with kernel 2.4.26 on a 1.4 Ghz Athlon computer running in single-user mode. Each test program was implemented both in C and Java and took the following examination: The C version was compiled with gcc 3.3.1 using the optimization flags -O2 -funroll-loops and run several times, taking the minimum running time. Then we examined the performance of the Java version running on the Sun embedded Java interpreter KVM 1.1 [8] with 1 MB heap size, which represents a typical embedded environment. In addition, we took the execution time on Sun's highly optimized J2SE HotSpot VM 1.4.2_04 [9]. As the latter is designed for desktop computers with huge storage capabilities, its results cannot be compared directly to the KVM outcomes. However, according to [7], the derived values provide an indication of the prospective performance of Sun's CLDC HotSpot VM, which only is available for sale. For the sake of completeness, we also measured the performance of the J2SE VM with its hot spot features disabled (-Xint).

3.2 Algorithmic Benchmarks

Our algorithmic benchmarks implement the following standard algorithms:

1. Integer → string → integer conversion
2. Calculate hash value from a string
3. Compare two strings for equality
4. Compare two strings case-insensitive
5. Haystack/needle search
6. Bubble sort
7. Quick sort

For all operations, we used fixed-length arrays. Table 1 provides the results of the algorithmic benchmarks. Figure 2 shows the Java execution speed against those of the C implementation, expressed as a percentage.

Table 1. Algorithmic benchmarks in ms

benchmark	GCC 3.3.1	J2SE 1.4.2_0 4 HotSpot	J2SE 1.4.2_04 -Xint	J2ME KVM 1.1
Int->String->Int	83	359	4064	9724
Hash calculation	39	267	3785	14429
String compare	43	123	2601	6588
String compare 2	143	341	8199	18183
Haystack/needle	640	880	14519	34119
Bubble sort	1018	2243	56428	137477
Quick sort	21	68	1313	3322

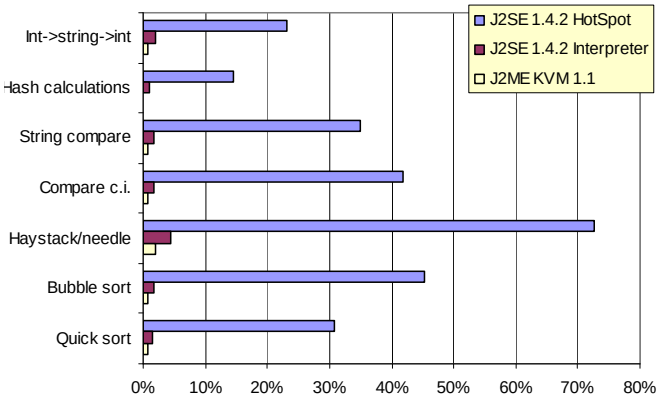

Figure 2. Algorithmic benchmarks in percent against C

In Figure 2, the performance gap between interpreted Java and C is tremendous. The results of the KVM range from 1% to 3% of the C performance and even the HotSpot VM drops down to 15% when calculating hash values from strings. On the other hand, the HotSpot VM comes close to the speed of native code when running the haystack/needle search. Both algorithms were taken from the Sun standard API implementation.

3.3 Graphics Benchmarks

The results in Table 2 and Figure 3 quantify the performance of simple 2-D drawing and image conversion algorithms, which are basically required for the creation of graphical user interfaces.

Table 2. Graphics benchmarks in ms

benchmark	GCC 3.3.1	J2SE 1.4.2_0 4 HotSpot	J2SE 1.4.2_04 -Xint	J2ME KVM 1.1
Lines	2720	5637	113390	256746
Ovals	2754	5920	96057	243256
Polygons	2866	5101	63543	160367
RGB -> gray	23	36	404	840
Averaging	11	40	582	1099
Bilinear	249	552	13373	25810

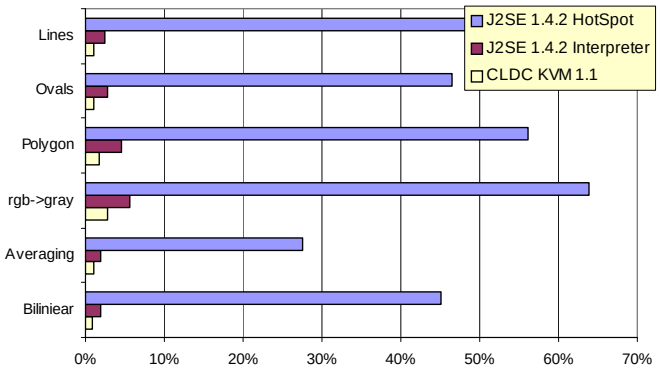

Figure 3. Graphics benchmarks in percent against C

To this end, the following algorithms were implemented:

1. Draw lines (Bresenham's algorithm)
2. Fill ovals (Kappel based scan line algorithm)
3. Fill polygons (scan line algorithm)
4. RGB → Gray image conversion
5. Image resampling 320x240 → 128x96 by averaging pixel values
6. Image resampling 320x240 → 960x720 by bilinear interpolation

These results show similar tendencies as the algorithmic benchmarks. It should be mentioned though that drawing lines and ovals are already found as native functions in the recent J2ME implementations. However, the resampling of images is not.

3.4 GUI Application Benchmark

In contrast to the cleanroom character of the results presented so far, we now evaluate a more complex Java application. Therefore, by applying the methodologies as characterized in section 2, we implemented a graphical menu prompt using visual features of recent mobile phones.

Figure 4. GUI Application

Figure 4 shows a screen-shot of the application. In front of a background image, a list of menu items is displayed. Navigation is done by pressing the up and down keys, moving a half-transparent navigation bar.

The GUI application runs on KVM and uses a Framebuffer object to render the user interface. We implemented the Framebuffer in two ways. The first utilizes native code for drawing images as well as lines. Clipping and color gradient functions are implemented in Java. This is equivalent to the Sun MIDP. The second Framebuffer realization was entirely implemented in C.

Table 3. GUI Application benchmarks in µs

benchmark	Version 1	Version 2
Switch selection	1153	495
Repaint UI	3185	2381

Table 3 lists the results of the GUI application benchmarks. The table shows that moving the selection bar in the C/Java-hybrid implementation takes less than half of the time as compared to the MIDP equivalent version. This improvement originates from the native implementation of the color gradient method, although the Java implementation already relied on a native line drawing function.

In this example, only a few lines of Java code were transferred into their native implementation. This gives an impression of possible improvements to application reactivity without a big effort necessary.

4 Conclusion

4.1 Performance

Our experimental results reveal the outstanding performance advantages of C as compared to Java. While the KVM lags alarmingly behind, even the J2SE HotSpot VM only achieves 30% to 80% of C performance.

The GUI application experiment shows that our C/Java-hybrid architecture is very flexible and that there is neither a notable performance loss due to the C-to-Java interface nor the accessor framework. However, one should not expect the full performance gains as illustrated by our algorithmic benchmarks. Depending on the VM used, additional time is spent on accessing Java objects and passing parameters.

Even though just-in-time compilation is a forward-looking approach and subject of intensive research, it is only rendered possible by the allocation of extensive additional memory resources [7, 9]. This makes an embedded Java system more expensive and less cost-effective.

In contrast to ahead-of-time compilation, our C/Java-hybrid architecture retains all important Java features such as runtime class loading, class security and platform independency.

4.2 Portability

Although our three layer approach might seem tedious, the independency achieved between generic libraries and hardware modules has crucial benefits:

- Isolation of hardware dependent code simplifies the porting to other platforms.
- Hardware modules are “pluggable“ on compile time, as only modules for actually present hardware need to be compiled.
- A decrease of code footprint is achieved, since the same library code can work on different hardware modules, and different Java classes can share the same libraries.

Since in most cases, the VM for a particular system is implemented in C, we can reason that an adequate C compiler is available (GCC, on which KVM is built, currently exists for more than 50 platforms [3]). Hence we think that the C/Java-hybrid architecture presented is a promising solution for the Java performance issue. As shown in section 3, it can improve the execution performance of embedded Java applications by a factor of 80 and more. Nevertheless, it is highly portable, and all important features of Java are retained.

Should just-in-time compilers for embedded systems come into vogue, portability might emerge as their most important issue, because porting the integrated compiler probably is much more complicated than porting the rest of the VM.

5 Summary and Future Work

In the present work we introduced a straightforward approach to improve the performance of Java on embedded devices with considerable storage limitations. Our proposal is to implement the majority of standard API functions in C, as this can easily be done by utilizing existing Java environments.

We elaborated the performance of Java as compared to C. While purely interpreting VMs such as the KVM reach only about 3% to 5%, the results show hot spot VMs to almost reach the performance of native code. As hot spot techniques require a lot of additional memory and make VMs less portable, they are not applicable in many cases.

In order to achieve a high level of portability and to provide object-oriented hardware access schemes, we characterized a software architecture for Java VMs and APIs. By partitioning the code into three layers, most parts of this architecture are platform independent. Moreover, no modifications at all have to be made to user applications.

Until now, we have applied our programming technique to the embedded graphics library gfxlib, which is part of a research project at the Technical University of Braunschweig and was entirely implemented in Java before. Besides, we applied it to the GUI application example from section 3.4 for benchmark purposes.

The next step will be the adaptation of the J2ME standard libraries for our architecture, so that the execution of usual J2ME applications becomes feasible. Further research should be done concerning dynamic linking of the native libraries into the VM. As this is done statically by now, the VM is inflated by each native library function.

Finally, the development of a simple methodology for merging the Hardware Layer of our architecture with the Sun K Native Interface would be of great benefit, for it then would instantly run on all platforms supported by the KVM.

6 References

[1] M. Arnold, S. Fink, D. Grove, M. Hind, and P. Sweeney. A Survey of Adaptive Optimization in Virtual Machines, Research Report, Thomas J. Watson Research Center, IBM, Yorktown Heights (NY), 2003.

[2] K. Burgaard and J. Erichsen. Virtual Machines for Limited Devices, Research Report, Dept. of Computer Science, University of Aarhus, Denmark, 2000.

[3] Gnu Compiler Collection. Host/Target specific installation notes for GCC. http://gcc.gnu.org/install/specific.html, 2004.

[4] IBM. Websphere Micro Environment. http://www-306.ibm.com/software/wireless/wme, 2004.

[5] G. Lawton. Moving Java into Mobile Phones. *IEEE Computer*, June 2002.

[6] Sun Microsystems. Connected Limited Device Configuration: Specification. http://jcp.org/about Java/communityprocess/final/jsr139/index.html, 2003.

[7] Sun Microsystems. The CLDC HotSpot Implementation Virtual Machine. http://java.sun.com/products/cldc/wp/CLDC_HI_WhitePaper.pdf, 2002.

[8] Sun Microsystems. J2ME Building Blocks for Mobile Devices. http://java.sun.com/products/kvm/wp/KVMwp.pdf, 2000.

[9] Sun Microsystems. The Java Hotspot Performance Engine Architecture. http://java.sun.com/products/hotspot/whitepaper.html, 2002.

[10] Sun Microsystems. K Native Interface. Specification, Version 1.0. Sun, 2002.

[11] SuperWaba. Product Specification. http://www.superwaba.com.br/en/swxj2me.asp, 2004.

[12] Venture Development Corporation. The Embedded Software Strategic Market Intelligence Program 2002/2003 - Volume IV: Java in Embedded Systems. http://www.vdc-corp.com/embedded/white/03/03esdtvol4.pdf, 2004.