Developing and Benchmarking Native Linux Applications on Android

Leonid Batyuk, Aubrey-Derrick Schmidt, Hans-Gunther Schmidt, Ahmet Camtepe, and Sahin Albayrak

Technische Universität Berlin, 10587 Berlin, Germany
{aubrey.schmidt,leonid.batyuk,hans-gunther.schmidt,ahmet.camtepe, sahin.albayrak}@dai-labor.de
http://www.dai-labor.de

Abstract. Smartphones get increasingly popular where more and more smartphone platforms emerge. Special attention was gained by the open source platform Android which was presented by the Open Handset Alliance (OHA) hosting members like Google, Motorola, and HTC. Android uses a Linux kernel and a stripped-down userland with a custom Java VM set on top. The resulting system joins the advantages of both environments, while third-parties are intended to develop only Java applications at the moment.

In this work, we present the benefit of using native applications in Android. Android includes a fully functional Linux, and using it for heavy computational tasks when developing applications can bring in substantional performance increase. We present how to develop native applications and software components, as well as how to let Linux applications and components communicate with Java programs. Additionally, we present performance measurements of native and Java applications executing identical tasks.

The results show that native `C` applications can be up to 30 times as fast as an identical algorithm running in Dalvik VM. Java applications can become a speed-up of up to 10 times if utilizing JNI.

Keywords: software, performance, smartphones, android, C, Java.

1 Introduction

With the growing customer interest in smartphones the number of available platforms increases steadily. Several open platforms emerged in the last years, including OpenMoko, LiMo, Mobilinux and, the most hyped one, Android. The latter has been presented by the Open Handset Alliance (OHA) where Google is one member beside others, including carriers like T-Mobile and Telefónica, and handset manufacturers like Motorola and HTC. Android's source code is freely available under terms of several open source licenses, which makes it an interesting open mobile platform. Android is modifiable and thus not limited to smartphones - it targets mobile internet devices and netbooks, too.

C. Giannelli (Ed.): Mobilware, LNICST 0007, pp. 381–392, 2009.

The key feature of Android is that is has a Java application framework on top of an Linux 2.6 Kernel - this construction unifies the mature security model of Linux with the convenient development in Java language. Most of the applications developed for Android are intentionally Java programs - but Linux-level `C/C++` programming is possible, too.

The question arises whether it makes always sense to stay on the Java layer, or cases can be identified in which native `C/C++` code is preferable. A very interesting aspect is the performance of computation - time-critical applications or software components which involve heavy computation might be moved to Linux layer for improving the execution speed. Since the graphical user interface is only accessible from Java framework (unless the user utilizes a terminal emulator), another resulting question is how to provide access to the information produced by native software components.

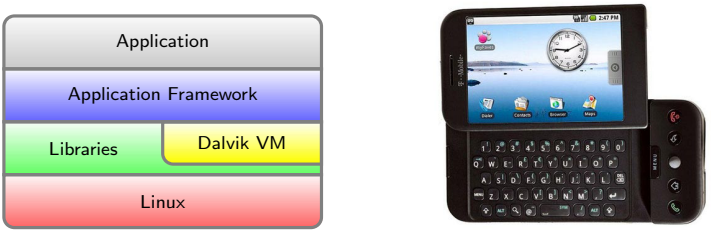
In this paper we will investigate the benefit of moving applications to the Linux layer of Android. We present mechanisms which enable communication between native and Java applications. This is of special interest since the OHA does not provide an officially supported way for doing so. Additionally, we present a performance comparison of Java and native Linux applications in Android for different kinds of tasks. This will help software designers to create efficient Android applications.

This work is structured as follows: in Section 2, we give an overview of the Android platform. In Section 3, we provide a summary on methods and tools which can be utilized in order to build native applications for the Linux layer in Android. In Section 4, we analyze the performance of the methods described in Section 3. In Section 5, we draw conclusions and describe viable future work.

2 Android

Following the descriptions of the Open Handset Alliance (OHA) [1], Android represents a software stack including an operating system, middleware, and applications. A significant aspect of this software stack is that Android uses Linux 2.6 as underlying operating system for core system services such as security, memory management, process management, network stack, and driver model. On top of Linux OHA placed a modified Java interpreter and runtime environment called Dalvik virtual machine (Dalvik VM). Intentionally, applications developed for Android will only use the Java Dalvik VM for execution. A key feature of this VM is the ability to run several instances of itself, each application in its own process. Executables for the Dalvik VM (.dex files) are optimized for minimal footprint where the Dalvik VM uses the threading and memory management functionality of the underlying Linux system.

For developing Android applications, the OHA provides an SDK with full access to the same framework APIs used by the core applications. The corresponding application architecture is intended to simplify the reuse of already developed components. In turn, this feature enables developers to replace



(a) Archtitecture [1] (b) T-Mobile G1 [14]

Fig. 1. Android

already existing components. Android applications are developed using the Java programming language. Therefore, Android includes a set of core libraries that mostly match the core libraries of the Java standard edition. Additional libraries were added to provide more convenient support for mobile needs, such as bluetooth and camera support.

The first phone running a port of Android is G1, which has been manufactured by HTC and introduced by T-Mobile in the USA. More devices running Android, including unofficial ports to existing hardware, are to be awaited since the source code of Android has been released under terms of free software licenses.

3 Software Engineering Aspects of Android

In this section we discuss the possible ways to develop software for the Android platform. In 3.1, we provide an overview on development of Java applications. Section 3.2 describes tools and techniques used to compile native applications. 3.3 contains additional information on Android specifics and possible workarounds.

3.1 Android Java Application Development

Different to most other mobile platforms, development of user applications for Android is pretty straight-forward. First of all, you need to download the SDK from Google page [1]. If you prefer to use an IDE for development, then you have to obtain Eclipse [6] or Netbeans [7]. For improving the development process, Eclipse users can download a plugin called Android Developer Tools (ADT) [1]. The SDK includes a built-in emulator that supports most of the important interfaces and can be used for application testing. If testing is successful, developers have to sign their application for making it available to devices. Android applications are packaged in APK files, which contain the executable bytecode, necessary resources (layout schemes, images, raw binary data), and application metadata. The metadata includes the name of the application, its version and the permissions which the program requires in order to run, such as access to contact database, using the internet connection or even making calls. Then, the

signed APK can be distributed to end devices through Android Market [3] - an internet platform which is open for developers and promises low rejection rates on new software. At this point it is important to mention that the signing of the application is, contrary to the Symbian platform, free of any cost and is primarily intended to make the distributor identifiable, and not to prevent software from unsubscribed developers from running on end devices.

3.2 Android Linux Application Development

Android provides a complete operating system running currently on the ARM-Architecture. Compiling software for ARM requires a specific environment. In following sections, we will describe two different ways of compiling software successfully within an ARM-compatible environment. Additionally, a list of working open source security tools running on Android can be found in our previous work [15].

Base environment. Ubuntu i686 GNU/Linux [10], a Linux-distribution provided and supported by Canonical, provides the basis for all further steps. Based on the Intel-architecture, a vast amount of tools, especially for creating and compiling software, can be obtained through Ubuntu's package repositories. Additional, non-standard, package repositories can be easily integrated.

GNU toolchain for ARM processors. CodeSourcery [8] offers the cross-compile toolchain *G++ Lite* that can be used to cross-compile source code for ARM on various architectures other than ARM. Consisting of C/C++-compiler, linker, libraries, several tools for debugging and more, it offers everything required for compiling tools that can be executed in an Android environment. Providing the required information during configuration run (passing parameters to use a different compiler, compile for a different architecture, additional usual compile parameters), there is little to no difference between compiling source code for ARM or for Intel architectures. Once compiled, the software needs to be transferred to the Android environment in order to test its functionality. This might be, at some times, a tiring task, and may be even restricted on the end device. Therefore, a second way to compile source code for ARM is presented in the next section.

Scratchbox cross-compilation toolkit. Scratchbox [9] not only provides the necessary compilers and linkers, it also provides a complete environment simulating an ARM platform-based operating system. All tools compiled within this environment can be tested immediately giving a very fast feedback to the developer. Once, the Ubuntu package repository has been extended by the official Scratchbox repository, all necessary files for Scratchbox can be easily installed via Ubuntu's package management tools. Scratchbox offers a wide variety of possible compilers, in different versions and characteristics. After installation, a user account has to be created for use within the Scratchbox environment. Shortly

after logging into the new environment, preliminary steps are required: select the desired compiler and add additional tools if wanted (`strace`, `gdb`). From this point, source code can be compiled as usual, no specific parameters have to be provided. The host and build type are distinguished automatically, standard locations for installing binaries, libraries, etc. are provided. As long as the given source code is ARM-compatible, it will most likely compile within Scratchbox without any significant problems. Having successfully compiled all files, these can be packed into an archive for being transferred to the Android environment for deployment.

3.3 Important Facts for Native Development

Filesystem specifics. Google provides an ARM Linux with a filesystem layout which greatly differs from usual Linux filesystem layouts:

- System relevant files are found in the System image, mounted to `/system` (binaries are, for the most part, found in `/system/bin`, libraries reside in `/system/lib`, configuration files in `/system/etc`, etc.)
- User data relevant files reside within the user data image, mounted to `/data`.

Handling these changes does not require much adaptation.

All-in-one binary toolbox. Furthermore, Google provides standard Linux tools with the help of the all-in-one binary `toolbox`. It only offers a very restricted set of tools making it at certain times hard to accomplish standard procedures. Special care has to be taken here when including shell scripts that rely upon various Linux system tools, since, if at all available, their behaviour would probably differ from what one would expect.

Installing Busybox [11], also a all-in-one static binary offering numerous standard Linux system tools, helps greatly, but is restricted on the G1.

Location-awareness of tools. Certain tools within Android are *location-aware*. A specific action, e.g. changing file permissions or ownership, will execute successfully without any further notice in `/system`. The same action, executed for files in your SD-Card-image will simply fail. This implies that tools can only be executed from within `/system` or `/data`. Adding and executing tools via a freely resizeable SD-Card-image will not be possible.

Disk space limitations. `/data` and `/system` offer only very limited flexibility as they are both limited to a maximum filesystem size of 65Mbytes. While in a standard, untouched, Android Linux, there is about 40MB of space left within `/data`, the System image, at the same time, offers only approximately 20MB for additional tools. This fact requires appropriate counter-measures when configuring given source code for compilation (e.g. ClamAV database needs to be placed in a different location as it exceeds the given 20MB on `/system`).

Page alignment causes changes in linking. Of very high impact on the success of compiling software for Android is the fact that Google forces compatible binaries to not be page aligned for the text and data section. This requires changes in the way of linking object files. For self-written software, one can take precautions and react on this fact with compiling all shared libraries accordingly. For already existing source code, changing the linker's behavior can present a very tiring and, often, an even impossible task.

Static linking. Due to the different approach of linking, the only way to run open source software on Android without altering the source code is to compile the source code statically. The output binary will have only small dependencies to existing libraries making it relatively autonomous. For a fair amount of available open source software, this method has been executed successfully. Still though, tools like "iptables" or "Snort" will not accept this method and fail compiling.

3.4 Bridging between Java and Linux

As already discussed in Section 1, it could be an interesting option to delegate certain computational tasks from Java to native binaries written in, e.g., the `C` programming language. We cover two possible solutions - Java Native Interface (JNI), a commonly accepted standard in the Java development community, and named pipes which are commonly used on unixoid systems for simple interprocess communication.

Java Native Interface (JNI). JNI is used to call native functions of the underlying operating system. Using this interface, the developer risks losing the platform independence of Java unless the native call exists on all intended platforms. At the moment, JNI is not supported on Android although it is used across the system. Following Romain Guy, an Android developer at Google, Android currently uses JNI only for the framework and not for the applications [4]. Nonetheless, Google seems to be working on a *native* SDK officially providing JNI calls.

Despite the official Google statement that JNI is currently not supported for user applications and won't work, we have successfully compiled a Java application which uses a custom JNI shared library. It was possible to install and run the application on the G1 without any further modification. The native component has been packaged into an APK as a raw binary resource and unpacked upon first execution of the Java program. After doing so, it is possible to load the shared object as a JNI library via invoking `java.lang.System.load(String filename)`.

From our point of view, JNI is rather hard to implement, since compiling a shared library for Android is a challenging task because of the unusual page alignment. But, it is most probably going to become the only official way to include native code in Android applications, and also has shown good performance.

Pipes. Using pipes is a commonly used technique in Linux and Unix systems to allow communication between separated processes. Two main types of pipes are known: unnamed pipes [12] and named pipes (also called FIFOs) [13].

Unnamed pipes are well known to most Linux and Unix users: e.g. have a look at the command `cat help.txt | grep a`. The first command displays the every line of the text file `help.txt` while the second command displays only the lines which contain the letter `a`. The horizontal bar indicates the usage of unnamed pipes where the results of the first command are stored in the kernel-side-located pipe and then are used by the second command. Writing and reading pipes works line-by-line following the First-in-First-out paradigm.

Named pipes are called FIFOs and are similar to unnamed pipes in their functionality. The main difference is that they are created explicitly and unrelated pipes are able to use them if no appropriete access control specifications are made. For creating named pipes the command `mkfifo` can be used. Additional information on pipes can be found in the corresponding man pages.

Using pipes on Android for communication between Java and native executables is rather straight-forward, since Java provides a lot of convenient writer, reader, and stream classes. But, when it comes to deploying and executing the binary in the restricted environment where an application can only write to its own folder, this approach requires decent programming skills. We have packaged the binary as a "raw resource", then the application itself unpacked it to the writieable directory. Then, the binary has to be made executable in order to be started, which is impossible from within Java. This is where the sources of the Android OS are needed - using the Android Build System instead of Ant, it is possible to utilize the class `android.os.Exec`, which is not included in the SDK classpath. It allows execution of native binaries as a subprocess of the Java application. Using `Exec.createSubprocess("/system/bin/sh", ...)`, it is possible to start the Linux shell and utilize `chmod` to make the binary executable. Afterwards, we can launch the native program and communicate with it through a named pipe.

We make use of an undocumented class, `android.os.Exec`. Still, the fact that it is present on the G1, in the emulator and in the source code of the platform, it is rather improbable that it will ever be removed, especially since some of the Google and third-party software on the Android Market already utilizes it.

The pipe technique is much more complex in its deployment than JNI, but it gives the developer a possibility to start a persistent daemon on a Linux layer which would collect information while being independent from the Java application lifecycle. The performance evaluations in Section 4 show that this approach is rather unsuitable for data-intensive tasks. But it is still a good option to consider if the application's logic forces the developer to work around the application lifecycle of Dalvik VM.

4 Software Performance on Android

Only few up-to-date references can be found pointing out comparisons between different programming languages and the corresponding compilers. The reason

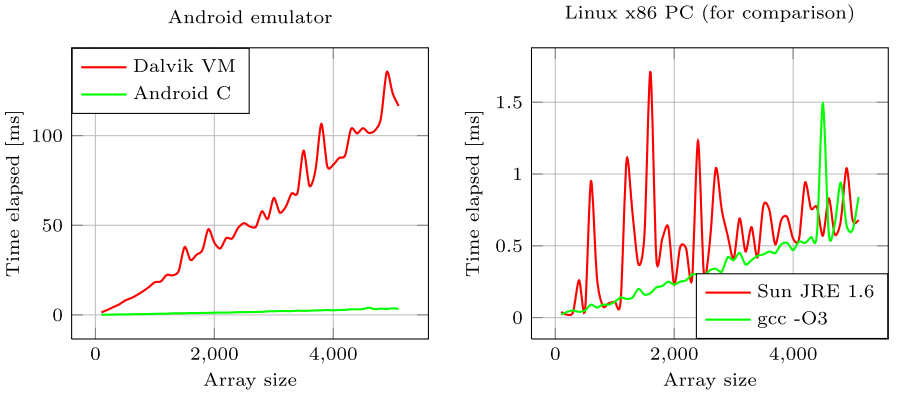


Fig. 2. Execution time of quick sort run in a Java VM and as a native binary, comparing the relative performance of Java to C in Dalvik and in Sun's JRE. While the execution time of both techniques on a Linux PC grows similarly and no significant difference occurs, the Dalvik VM shows a very bad performance and takes up to 30 times as long as a native Android executable.

for this might be the fast progress in the technology and computer sector that renders such kinds of results useless only few years after gathering them. A *newer* publication was made by McConnel [16] in the year 2004. In this book he describes in "Chapter 25: Code-Tuning Strategies" how to improve software code and presents performance comparisons on different types of programming styles as well as of different programming languages and compilers. Additionally, he states that Java programs have 50% higher *relative execution time* than, e.g. `C++` or Visual Basic. If these results were applicable to Android, this would mean that a performance increase would be achieved by transferring data sets from Java to a native executable, and retrieving back the results after computation. Of course, this is only true if we assume that the speed of the data transfer between the layers is fast enough for a trade-off to be found.

We would benefit from this increase until the prediction of Reinholtz [5] turns true. He stated that, in future, Java will be faster than `C++` since dynamic compilation gives the Java compiler access to runtime information not available to a `C++` compiler. But regarding the current Android platform, our current findings show that this state is far away from being reached in most cases.

4.1 Performance Evaluation

We have performed a series of microbenchmarks to compare the performance of identical sorting algorithms on various platforms. In the first experiment, we have compared the performance of various sorting algorithms implemented as standalone Java and `C` executables. Second, we have evaluated the performance of possible techniques which allow delegation of heavy computational tasks to

the native layer or to built-in Java facilities, which we assumed to be faster than our own implementations.

Raw performance of sorting algorithms. In [20], Okumura et al. propose various benchmarking techniques to evaluate the performance of Java VMs. One of those is sorting objects and primitives, which we see as the most fitted for our purpose, since we concentrate on data-intensive tasks.

First, we have implemented three common sorting algorithms in both Java and `C`: bubble sort [17], quicksort [18], and heapsort [19]. Arrays of random integers have been generated and sorted with the corresponding algorithms in standalone executables. The time used for each of the algorithms has been recorded while the size of the array steadily increased. This first simple measurement was performed on the Android emulator, and on a Linux PC for a reference.

Analysis of bridging and built-in techniques. After receiving the first benchmark results it becomes obvious that delegating computational tasks

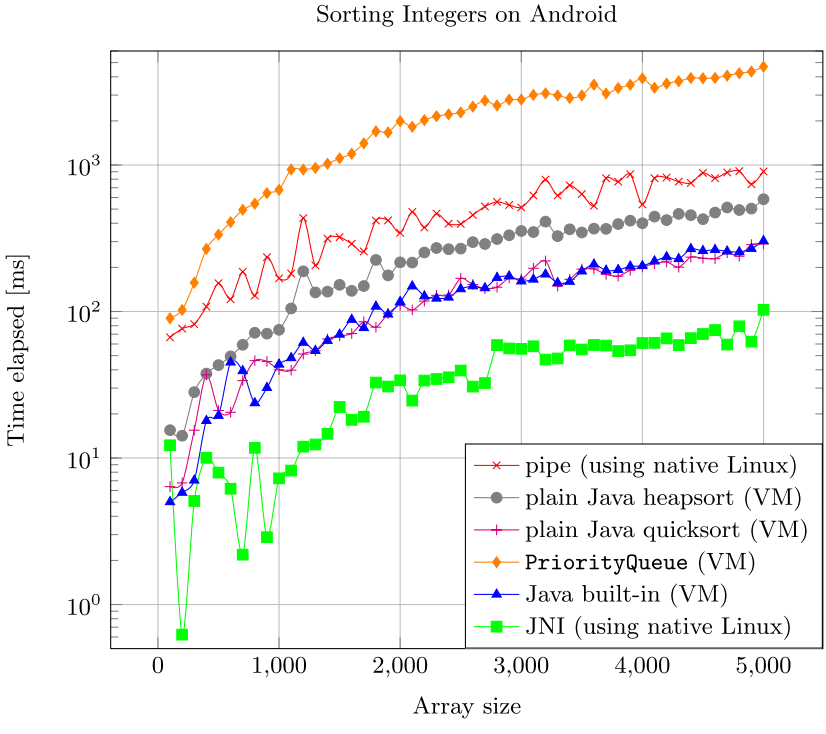


Fig. 3. Performance of sorting algorithms using various techniques on Android. All test runs involve random data generation on Java, handing over the unsorted array to the sorting function, and then retrieving back the results. Using a priority queue for sorting numbers has proved to be the slowest approach, followed by the pipe. All pure-Java sorting algorithms show similar performance, and only JNI shows a significant increase in performance.

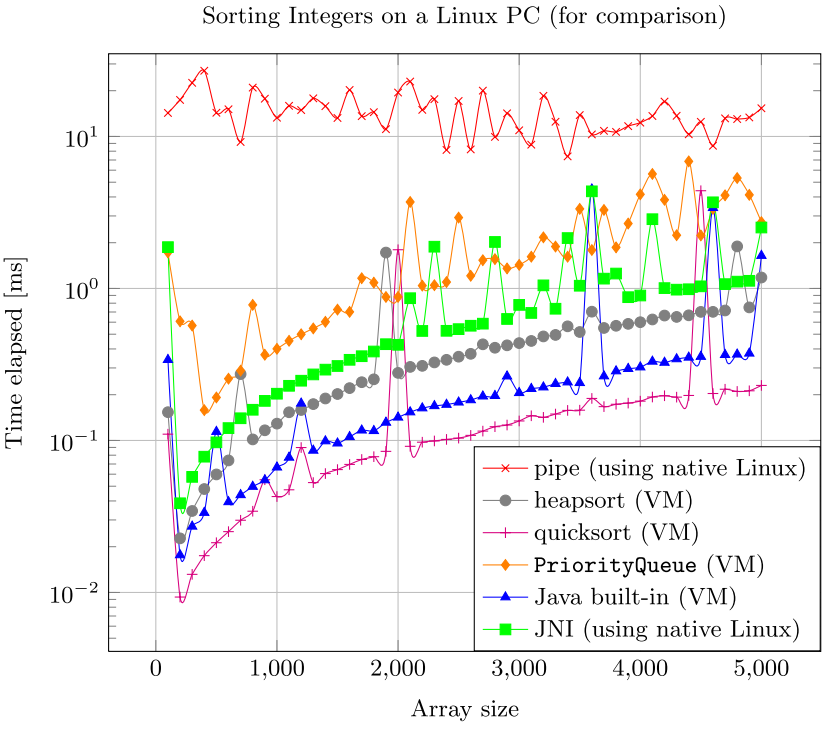


Fig. 4. Performance of sorting algorithms using various techniques on a Linux based OS using Sun JRE 1.6. The values on this environment show substantially different results. Using a pipe is the slowest approach, and JNI is also relatively slow. This proves the efficiency of Sun's Java platform and shows that the Dalvik VM has room for optimization.

to the native layer could drastically improve the speed of Java applications. We have analyzed the performance of the two bridging techniques introduced in section 3.4: the more common JNI, and a more straight-forward and platform-independent direct file communication through named pipes. Additionally, built-in Java techniques are being evaluated: object-oriented sorting using a `java.util.PriorityQueue` as a heap and a built-in sort with `java.util.Arrays.sort(int[])`. For comparison, heapsort, and quicksort implemented in plain Java are also presented. Figures 3 and 4 show the results for the Android emulator and a Linux PC, respectively.

Benchmark results. Our measurements show clearly that there is a great difference between the Dalvik VM and the JRE from Sun. Sun's optimization of byte code is very effective, which results in performance which is comparable to native binaries. Since JNI involves significant overhead when a function is being called, it is not the fastest technique on a regular PC.

Contrary to this, the fastest computational technique for Android devices is JNI. It beats even the optimized built-in algorithm from `java.util.Arrays`. Unfortunately, the simple approach of data delegation through a pipe has proven to be relatively slow, which can be explained by the slow IO performance of the Dalvik VM. The most disappointing results have been delivered by the object-oriented method using Java's `PriorityQueue` as a min-heap.

5 Conclusion and Future Work

Our results show that Google still has much room for optimization. On one hand, just-in-time compilation should be considered, and on the other hand, native implementations of computationally complex classpath methods should be introduced. Since Google is supposedly planning to introduce JNI capabilities to the Android SDK, we recommend developers to embrace its potential and shift heavy computation to the native layer.

In our future work, we are planning to port an existing benchmarking suite to Android, most probably LINPACK [21]. We will port the benchmark to both Java and C for Android and test the performance of real hardware, including the G1 and other handsets which will emerge this year. Benchmarking different mobile operating systems on the same hardware would also deliver valuable results. Comparing the performance of OpenMoko and Android on the Neo FreeRunner handset is currently the only option available, but we are looking forward to new flashable handsets in 2009.

References

1. Android - An Open Handset Alliance Project (2008/12/05), http://code.google.com/android/
2. Android Open Source Project (2008/12/05), http://source.android.com/
3. Android Market (2008/12/05), http://www.android.com/market/
4. Android Developer Mailing List Post (2008/12/05), http://groups.google.com/group/android-developers/browse_thread/thread/f87e6fce2b26db36
5. Reinholtz, K.: Java will be faster than C++. ACM SIGPLAN Not. 35, 25–28 (2000)
6. Eclipse Intergrated Development Environment (2008/12/05), http://www.eclipse.org/
7. Netbeans Intergrated Development Environment (2008/12/05), http://www.netbeans.org/
8. Codesourcery (2008/12/05), http://www.codesourcery.com/
9. Scratchbox (2008/12/05), http://www.scratchbox.org/
10. Ubuntu Home Page (2008/12/05), http://www.ubuntu.com/
11. Busybox (2008/12/05), http://www.busybox.net/
12. Unnamed Pipes (2008/12/05), http://docs.sun.com/app/docs/doc/816-1042/6m7g4ma79
13. Named Pipes (FIFOs) (2008/12/05), http://docs.sun.com/app/docs/doc/816-1042/6m7g4ma7a
14. HTC T-Mobile G1 (2008/12/05), http://www.htc.com/www/product/g1/overview.html

15. Schmidt, A.-D., Schmidt, H.-G., Clausen, J., Yüksel, K.A., Kiraz, O., Camtepe, A., Albayrak, S.: Enhancing Security of Linux-based Android Devices. In: Proceedings of 15th International Linux Kongress. Lehman Verlag, Hamburg (2008)
16. McConnel, S.: Code Complete, 2nd edn., pp. 180–197. Microsoft Press, Redmond (2004)
17. Knuth, D.E.: The Art of Computer Programming, 2nd edn. Sorting and Searching, vol. 3, pp. 180–197. Addison-Wesley, Reading (1997)
18. Hoare, C.A.R.: Quicksort. Computer Journal 5(1), 10–15 (1962)
19. Williams, J.W.J.: Algorithm 232 - Heapsort. Communications of the ACM 7(6), 347–348 (1964)
20. Okumura, T., Childers, B., Mossé, B.: Running a Java VM Inside an Operating System Kernel. In: VEE 2008: Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments, Seattle, WA, USA (2008)
21. Dongarra, J.J.: The LINPACK Benchmark: An explanation. In: Houstis, E.N., Polychronopoulos, C.D., Papatheodorou, T.S. (eds.) ICS 1987. LNCS, vol. 297. Springer, Heidelberg (1988)