Mobile Devices-aware Refactorings for Scientific Computational Kernels

Ana Rodriguez[2], Cristian Mateos[1,2], and Alejandro Zunino[1,2]

[1] ISISTAN Research Institute. UNICEN University. Campus Universitario, Tandil (B7001BBO), Buenos Aires, Argentina. Tel.: +54 (249) 4439682.
[2] Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET)

Abstract. The increasing number of mobile devices with ever-growing processing capabilities, make them interesting for scientific applications development. However, we must take into account that mobile devices still have limited capabilities compared to fixed devices. Besides, mobile devices rely on battery as energy supply. For these reasons, this paper analyzes different micro-benchmarks battery consumption given by common operations in scientific computational kernels. Indirectly, we propose good programming practices or code refactorings in order to minimize mobile devices battery consumption.

Keywords: Smartphones, Android, Scientific Computing, Computational Kernels, Refactoring

1 Introduction

Mobile devices still have more limited resources than personal computers or servers. In particular, minimizing energy consumption is very important since mobile devices rely on battery as energy supply. However, devices such as smartphones, netbooks, notebooks and tablets represent an attractive market to develop applications, mostly because they are the commonest form of technological device in the world [22].

Firstly, these devices are no longer simple agendas or cellphones. Current mobile devices are "little computers", and their capabilities are steadily increasing. To appreciate the computational capabilities of modern mobile devices, it is necessary to look into the past. Back in 1986, the Cray X-MP, which had 16-128 megabytes of RAM and one, two or four processors running at 117 MHz each, was one of the most powerful supercomputers of its times. This computer needed a dedicated power line to satisfy its energy requirements [6]. On the other hand, current smartphones have a processor, in some cases with 2 to 4 cores running at up to 2 GHz, and up to 1 GB of RAM. Besides, while the supercomputer was interconnected with other computers by a wired network link, mobile devices can be connected through a range of wireless network technologies (such as 3G, WiFi or Bluetooth) with a speed higher than 1 Mbps [22]. Furthermore, mobile devices offer interesting features such as sensors, GPS, accelerometers, digital maps for navigation purposes, etc.

Additionally, it is important to mention the accelerated evolution of mobile devices. The firsts notebook, the Osborne 1, was released in 1981. Its power supply was through an electrical connection or an optional battery. The Osborne 1 had 60 KB of RAM and a Z80 processor that ran at 4 MHz. At present, any smartphone has more capacity than

this notebook.We can see the continued smartphones progress comparing the Samsung Galaxy S which was presented on September 9, 2010 and its successor Samsung Galaxy SII. The Samsung Galaxy S used the Samsung S5PC110 processor which combined a 45 nm 1 GHz ARM Cortex-A8 based CPU. This smartphone has 512 MB of RAM. Additionally, it has a 2 GB internal memory and 16 GB external memory capacity. With respect to battery, it has a talk time of up to 6 hours and standby time of up to 300 hours. The Samsung Galaxy S supports Bluetooth 3.0, Wi-Fi 802.11b/g/n and 3G data up to 7.2 Mbit/s. In contrast, the Galaxy S II has a 1.2 GHz dual core processor, 1 GB of RAM, 16 GB of internal mass storage, 64 GB microSD card slot. It has a battery standby time of up to 10.5 days and talk time of up to 8 hours.

These examples show that while computational capacity has increased rapidly, battery power has increased slowly [20]. Consequently, this paper main contributions are evaluating mobile device capabilities using different micro-benchmarks, and proposing various code refactorings to save battery in mobile devices. Currently, mobile devices are not only elements of connection and external computing resources, but also they are part of the computational infrastructure in scientific projects [16,22]. For instance, mobile devices new capabilities led to the scientific community interest in integrating them in common execution infrastructures used in simulations such as Grids and clusters [21,22]. Additionally, MIT researchers have recently proposed a hybrid computing model where mobile devices actually run semi-intensive sections of fluid simulation applications [13]. This paper analyzes the battery consumption of different versions of micro-benchmarks consisting of common programming operations found in scientific applications in this new hardware. The operations chosen are array copying, string handling, matrix traversal, arithmetic operations, exception handling, object field access and object creation.

Concretely, we have focused on Android-powered devices because the Android platform is present in millions of smartphones, tablets, and other devices and it is an open-source project led by Google and based on Linux. It includes extensible and tailorable middleware, key applications and software stack.

The rest of this paper is organized as follows. First, Section 2 describes in detail each micro-benchmark chosen. After that, Section 3 shows the decisions made and the problems encountered in developing the experiments. The Section also analyzes the obtained results in detail. Finally, Section 4 presents conclusions results and discusses about future research directions.

2 Micro-benchmarks

Some of the most important problems to develop applications for mobile devices are their limited capabilities in terms of CPU processing capacity and battery time. To minimize mobile devices battery consumption, and indirectly using less CPU time, this work aims to find the relation between different ways of writing the same micro-benchmark and their corresponding battery consumption. As a corollary, we are looking for guidelines for programming common operations in mobile devices so that applications in general and scientific codes in particular are more efficient in terms of battery usage and execution time.

For this, we chose seven groups of micro-benchmarks. The election was based on the recurrent use of these structures in scientific applications. These groups are array copying, matrix traversal, string handling, arithmetic operations, exception handling, object field access and object creation. The next paragraph explains the reasons why these groups were selected.

Over the years several different libraries for scientific use were developed in diverse languages such as C++ and Java [12,19]. In consequence, we decided to determine battery consumption improvement using these libraries to copy an array over implementing manually the same functionality. Additionally, matrixes are a common structure in scientific software. Matrixes have important mathematical and computational uses. For instance matrix-based 3D transformations [1] and Matrix-Matrix Multiplication are important kernels in linear algebra algorithms [17]. Concerning to programs manipulating strings, concatenation is the most important string operation [5]. Our interest in strings is based on the above statement and the uses in the scientific community as those shown in the following example. In [11] authors analyze the problem of covering a set of strings S with a set C of substrings in S (C covers S if every string in S can be written as a concatenation of the substrings in C). Regarding the fourth group, several studies [25,2,9] focus their efforts on arithmetic operations or involve large numbers of them. Moreover, exceptions represent a mechanism for elegant error handling and are commonly used in languages that support them. However, exceptions produce performance loss. In consequence, works such as [14] analyze mechanisms for removing overheads imposed by the existence of exception handlers and motivate us to investigate its impact on battery consumption. The next group, method invocation, was chosen for similar reasons. In object-oriented (OO) programming, a method is a subroutine associated with a class. As a consequence, calling a method is a common operation in OO applications. This motivates us to analyze the battery consumption that this operation implies. Finally, we chose object creation not only because objects are the focus of OO programing and object creation itself reduces the performance of the application, but also application performance indirectly declines because of garbage collectors, which clean memory from unused objects. Then, an excessive number of objects can seriously decrease application performance.

Next subsections detail each micro-benchmark and mention some relevant Java characteristics since it is the high level programming language the Android platform relies on. Additionally, Java provides portability, security, robustness and simplicity[15]. In fact, Java provides and supports pure OO programming, multi-threading and distributed computing implemented at language level, platform independence, automatic memory management and exception handling which make it interesting for general purpose applications [1]. To execute these applications, Android use an special optimized virtual machine called Dalvik.

2.1 Array copying

Most current languages provide libraries whose main goals are to modularize and reuse common functionality easily. Examples are libraries for searching and sorting data structures, or random number generation algorithms. Using these libraries has significant advantages over using an ad-hoc implementation.

Firstly, by using a standard library, developers take advantage of the knowledge of the experts who wrote it and the experience of those who used it before them. A second advantage of using the libraries is that developers do not have to waste time writing common functionality. As a result developers can place their code in the mainstream. Additionally, standard libraries performance tends to be improved over time. In particular, many of the Java platform libraries have been rewritten over the years resulting in dramatic performance improvements [3].

In this line, this paper compares the use of the System.arraycopy library with a manual solution for the same functionality. Array structures are one of the most important

structures in scientific codes. Programmers use arrays instead of multiple variable declarations. For instance, in mathematics, arrays are used for polynomials representation or combinatorial analysis.

2.2 Matrix traversal

Matrixes are also mathematical structures that have many applications such as writing problems conveniently and compactly, helping to solve problems with linear and differential equations and coordinating change in some kind of integrals. Additionally, in graph theory an adjacency matrix can be associated to each graph where the position [i,j] indicates if the vertex i is connected with the vertex j. With this matrix for example the grade of a particular vertex can be calculated.

At present, programmers cannot be oblivious to these structures. Matrixes are used to store any data type and are a common structure in any 3D application, where they are often used to apply transformations to 3D images. Therefore, we tested micro-benchmarks where NxM matrixes are traversed by rows and columns. Although they are not yet generalizable, the results of [18] shows that traversing matrixes by rows in Android smartphones is faster than traversing by columns. Then, we measure the impact of this in battery consumption.

2.3 String handling

For programmers, strings are broadly the main way to represent and handle text in programs. Applications often use the data type String either to save or read data or, display a message to the user, among other uses. Usually, concatenating these Strings is necessary to process data.

Then, we work with concatenation using the "+" operator and using the specific Java class StringBuilder. Considering Strings in Java are inmutables, i.e. their values cannot be changed after they are created, the use of the "+" operator is not efficient for a large number of concatenations. According to [3], the string concatenation operator is a convenient way to combine a few strings into one, for instance, for generating a single line of output or for constructing the string representation of a small object. However, this does not scale. To use the string concatenation operator repeatedly to concatenate n strings requires time quadratic in n. In consequence, we expect an improvement using the StringBuilder class.

2.4 Arithmetic operations

Arithmetic operations are one of the most usual operations in applications. This is illustrated by management applications, accounting applications, data compression applications and mathematical applications. Particularly, scientific applications with mathematic operations are very common and these often need several calculations to achieve their goal. As a result, the more efficient the arithmetic operations are, the lower the battery consumption a device experiences.

This paper measures the addition battery consumption using primitive data types. These types are *int*, *long*, *float* and *double*. We expect the operations with *int* (integer) data type to be more efficient than floating point operations. This is due to the greater complexity that floating point operations present compared to integer operations.

2.5 Exception handling

Java provides an exception handling mechanism for elegant error handling. Using exceptions is the ordinary way to manage any unexpected event like division by zero. Basically, when an object is in a condition it can not handle, the object creates and raises an exception that has to be captured by someone else high in the call stack.

The main advantage of this mechanism is avoiding uncomfortable if-then-else error handling blocks. Additionally, Java applications have other several advantages since this error-handling mechanism. First, error handling is cleanly performed in a separate code area. In addition, we can propagate the error to the first method that called the various method up to the error. Finally, errors can be grouped and differentiated.

However, the exception mechanism has a negative effect on application performance. First, there is little interest among JVMs in optimizing exception handling because this mechanism was designed for exceptional situations. Additionally, writing code inside a try-catch block prevents the application of some optimizations. Finally, throwing an exception involves object creation. Therefore this paper analyzes two methods which are functionally equivalent, one using exceptions to respond to valid situations and one without these. In other words, we show the advantage of using exceptions only in situations that can not be handled by the object in any way. For instance, detect a division by zero is a simple task for programmers and, in many applications, they know how the application must respond to this situation. As a result, developers could avoid using exceptions under this situation.

2.6 Object field access

OO languages call stacks rely on the use of methods. Objects have a set of properties or attributes, and a set of methods that implement their behavior. One of the practices recommended by the OO programming paradigm is hiding information. Each class provides public methods to which other classes can refer in order to access these fields or attributes. While this practice results in more flexible designs, easy to understand and maintain, the continual invocation of getter methods also has a negative impact on application performance. For our purposes we measured the battery consumption to obtaining an attribute value, which in one case is performed through a method call and in the other is performed directly, without having the functionality encapsulated in a method.

2.7 Object creation

OO programming is based on sorting data in modular sets of information items. Additionally, in object-oriented programming, object creation is inherent because different entities with different states are required. Furthermore, the creation of objects in memory and its maintenance always involves some computational cost. Sometimes developers can avoid creating new objects by reusing objects no longer used after resetting their attributes. In consequence we analyze the impact that object creation has on battery consumption over reusing them.

As in the libraries case, we chose an structure commonly used in programming such as lists. For instance, lists are used as modules for many other data structures such as stacks, queues and their variations. In this paper, we compare the battery consumption levels when creating a new list object against reusing an existing list. To reuse an existing list we use a method to restore the list to its initial (empty) state.

3 Experiments

Evaluating a system performance is not a trivial task [10]. Unlike traditional compiled languages, Java compiled code (bytecode) is platform-independent. Java Virtual Machines (JVMs) provide an environment in which Java bytecode can be executed. The traditional approach is based on the interpretation of the bytecodes. A second solution includes the just in time (JIT) compilation in which bytecode-to-assembly translation occurs continuously to minimize performance degradation. JVMs have an interpreter (or a JIT compiler) integrated with an optimizing dynamic compiler which performs an optimized compilation only of the parts of the program that are most frequently used [1]. In addition, the Garbage Collector, which is the automatic dynamic memory manager, has always been a key factor in terms of Java Virtual Machine performance. In the next paragraphs, we present the constraints and the solutions to run micro-benchmarks in Java:

- Different Java compilers make different optimizations. Some compilers might detect dead code (code not affecting the output) and delete it. Consequently, micro-benchmarks run faster than the same code compiled with dead code. To prevent dead code problem we can add extra code but this code can change the results.
- Secondly, Java developments have different optimization levels. First iterations of the benchmarked code include a large amount of dynamic compilation. Later iterations are usually faster because they include less compilation and the executed code is optimized [7]. This problem is linked with the Just In Time compiler. Measuring when the executions converge can reduce the impact.
- Thirdly, performance evaluation in virtual machines is difficult [24] because they provide an extra abstraction layer which could change results in different runs.
- Next, the Garbage Collector can activate asynchronously while the micro-benchmark is executing.
- Finally, the load of classes at the beginning might cause the first executions to be slower and consume more resources.

Based on these problems, we decided to use a framework called Caliper [8]. Caliper is Google's open-source framework for writing, running and viewing the results of Java micro-benchmarks dealing with the problems mentioned.

The idea was to measure how many operations could be completed in a battery cycle. Each micro-benchmark was executed at least 10 times before reaching conclusions. Each test was ran after a device restart and executing only the essential operating system applications. Besides, the device was fully charged and unplugged before starting each test. The device used for testing is a Samsung I5500 with the following characteristics: 600 MHz CPU (model MSM7227-1 ARM11), 256 MB RAM, 170 MB up to 16 GB of storage and battery Lithium Ion 100 mAh.

The test results are shown in Table 1. The standard deviation was below 3% w.r.t to the average. This deviation is a result of battery consumption generated by operating system-essential applications. As we will discuss in the following subsections, within each test, the more the achieved executions, the more the battery efficiency.

3.1 Array copying

In this paper, System.arraycopy Java library, which provides methods for copying arrays, was chosen because this functionality is commonplace in the scientific community. To evaluate and compare the efficiency of this library we used a manual implementation of the same functionality. The results are shown in Table 1 and graphically in Fig. 1a.

Benchmark	Number of executions(Average)	Standard deviation (%)
Use of Language Built-in Libraries		
Manual Array Copy	11,103,650	0.64
System Array Copy	13,428,990	1.37
Matrix Traversal		
By-column Matrix Iteration	149,443	2.29
By-row Matrix Iteration	320,785	1.84
String Handling		
String Concatenation (+)	1,791	2.58
String Builder	1,399,352	2.81
Arithmetic Operations		
Add Constant to *double*	412,940,900,000	2.28
Add Constant to *float*	435,343,470,000	2.56
Add Constant to *long*	788,482,635,000	2.60
Add Constant to *int*	1,183,897,395,000	2.01
Exception Handling		
Use Exception	2,612,158,650	2.93
No Exception	250,029,732,150	2.99
Object Field Access		
Getter-based Access	115,459,467,000	2.97
Direct Access	919,194,540,000	2.13
Object Creation		
On-demand Object Creation	1,691,926,500	2.70
Object Reuse	15,433,805,000	2.81

Table 1: Micro-benchmark results

Quantities of executions obtained show that using the library improves by a 20.94% battery usage. As this paper is focused on one library, we can not extend the results to other libraries, but using libraries for the case of array copying is a good practice.

3.2 Matrix traversal

As mentioned in Section 2, one of the most common operations used in matrix structures is exploring its elements to do some processing later. This paper uses NxM matrix structures and compares the traverse by row with the traverse by columns. Specifically, a matrix of $1024x1024$ is used to run tests. One of the key advantages of these micro-benchmarks is the triviality of changing the traverse mode. The results show an improvement of 114.65% using the traverse by-row operation. These conclusions are supported by numerical results presented in the Table 1 and Fig. 1b.

3.3 String handling

The results of the two alternatives to this micro-benchmark are in Table 1 were obtained. As expected, using the specific class StringBuilder generates an improvement

of 78,023.73% in Strings with 1,000 concatenations. Due to the large difference in the results, these are presented in an special graphic with logarithmic scale in Fig. 1c.

The main reason for this variation is that String literals in Java programs, such as "abc", are implemented as instances of this class. Strings are constants, their values cannot be changed after they are created. Consequently, using the "+" operator involves the creation of a temporal object which contains the final string. Instead of the "+" operator, the StringBuilder class maintains a mutable string of characters and provides methods to modify it without creating new objects.

3.4 Arithmetic operations

The micro-benchmark involved adding a constant value to a numerical variable declared several times by varing its data type. Table 1 and Fig. 1d show the obtained results. Operations with float numbers have more cost than operations with integer numbers. Thus, the latter group uses less battery than the former group. To specify, using *int* data type has an improvement of 186.7%, 171.95% and 50.15% over relying on the *double*, *float* and *long* data types, respectively. As a result, using the most specific data type is a good practice to save battery in mobile devices.

Additionally, the *double* and *long* data types consume more battery than the *float* and *int* data types, respectively, because the first data types provide greater accuracy than the second ones. Consequently, greater precision implies faster battery depletion.

3.5 Exception handling

The results shown in Table 1 and Fig. 1e support the first hypothesis presented in Section 2.5. As expected, programmers can generate significant battery saving not using exceptions in his applications. According to this, Fig. 1e shows an improvement of 9,471%. Reasons shown in Section 2.5, such as the creation of objects and the limited optimizations made by the JVM, produce higher battery consumption. To ensure minimum battery consumption, we can conclude that the use of exceptions must be reserved only for error situations that the object can not deal with itself.

3.6 Object field access

Obtaining the variable value directly and not through a getter represents an improvement of 696.12% of battery consumption. Table 1 and Fig. 1f show that direct access to variables which are frequently used, results in significant battery savings. However, the programmer must not declare as public all variables of a class, or combine several methods of a class that are unrelated to avoid repetitive invocations. Loss of legibility and the high coupling of the resulting applications would outweigh the benefit. In consequence, developers must determine to what extent it is valuable to set aside design concerns in order to favor performance. However, there are classic cases such as accessing to the class variables by the same class methods which can directly access the variable.

3.7 Object creation

As discussed in Section 2.7, the hypothesis that object creation produces more battery consumption than reusing objects is reflected in the results presented. A 812.20% improvement is shown in Table 1 and Fig. 1g. Developers must be careful not to create

objects that are not needed in the application. Thus, they can achieve significant savings in battery consumption. Programmers must also avoid reusing objects which are being used or may be used in future by the application for correctness reasons. Additionally, we must consider that improvements may vary depending on the object to be created; however in this paper we tested with lists, which represent an object type very popular to implement other data structures.

4 Conclusions

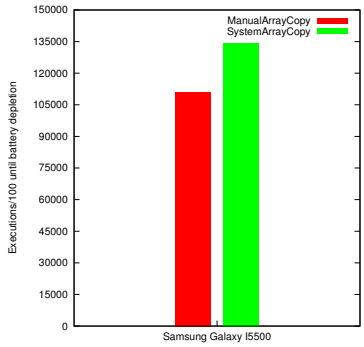
In this paper, different common micro-benchmarks were presented for analyzing their associated battery consumption in a state-of-the-art and very popular mobile device. In all cases we confirmed the expected results: micro-benchmarks that have worse performance in terms of time [18] consume more battery as well. In particular, the results show that it is recommendable to use libraries instead of using a manual implementation of the functionalities. Either, something trivial as how a matrix is iterated can improve an application. Additionally, this paper reflects the importance of avoiding operations with immutable data types as Strings. As regards to numerics data type, developers must analyze the application and use the most specific data type to save battery. Other method to save battery significantly is to avoid using exceptions on the application when possible. Many times, objects can handle errors proposing mechanisms to overcome undesirable situations in the application, without throwing an exception. Finally, referring to basic and common operations in OO programming such as object field access and object creation, we showed that when these operations are not necessary they should avoid it as much as possible in devices which rely on battery as energy supply.

These results represent a set of guidelines or best practices that developers can apply in order to get the most benefit of mobile devices for applications heavily relying on such operations, particularly scientific codes. The guidelines are at the same time refactorings programmers should apply when porting standard Java applications to run on Android-powered devices.

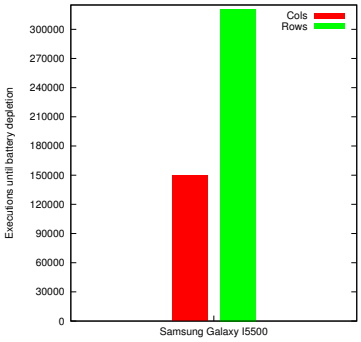
This work can be extended in several directions. Firstly, we will analyze the possibility of using other micro-benchmarks to assess mobile devices capabilities. These micro-benchmarks will be related with handling linear data structures (Vector, LinkedList) in memory, using of primitive data types over using of object data types, etc. Secondly, we will run the tests in other devices to prove whether the obtained results are device-dependent or not. In addition, we will study how to perform automatic code refactoring, and analyze real applications. Finally, we will test the impact of these refactorings using native Android applications, which is also a hot topic in the area [4,23].

References

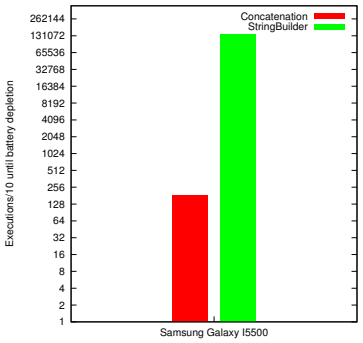
1. Barisone, A., Bellotti, F., Berta, R., De Gloria, A.: Jsbricks: a suite of microbenchmarks for the evaluation of java as a scientific execution environment. Future Generation Computer Systems 18, 293–306 (2001)
2. Baron, R., Lioubashevski, O., Katz, E., Niazov, T., Willner, I.: Elementary arithmetic operations by enzymes: A model for metabolic pathway based computing. Angewandte Chemie International Edition 45, 1572–1576 (2006)
3. Bloch, J.: Effective Java programming language guide. Sun Microsystems, Inc., Mountain View, CA, USA (2001)



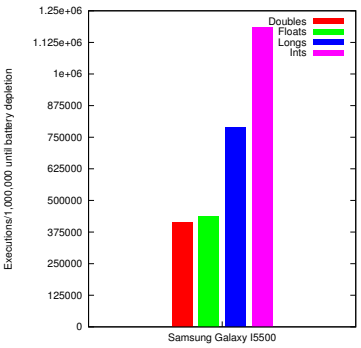
(a) Array copying



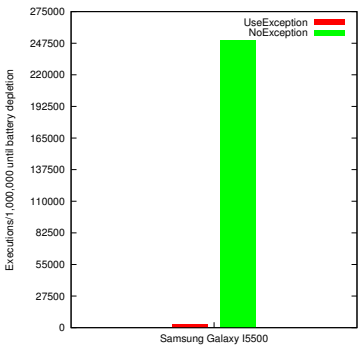
(b) Matrix traversal



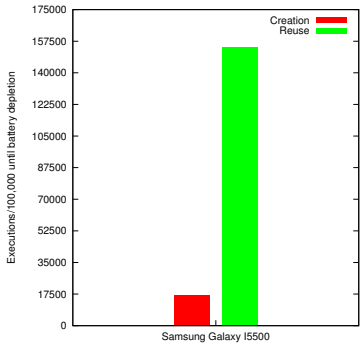
(c) String handling



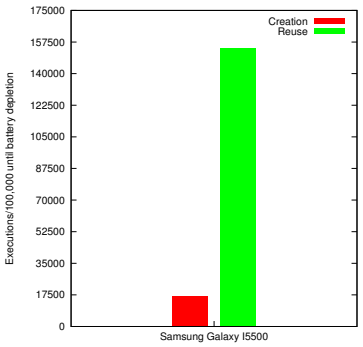
(d) Arithmetic operations



(e) Exception handling



(f) Object field access



(g) Object creation

Fig. 1: Micro-benchmarks results

4. Cheng-Min, L., Jyh-Horng, L., Chyi-Ren, D., Chang-Ming, W.: Benchmark dalvik and native code for android system. In: Second International Conference on Innovations in Bio-inspired Computing and Applications (2011)
5. Christensen, A.S., Moller, A., Schwartzbach, M.I.: Precise analysis of string expressions. Lecture Notes in Computer Science 2694 (2003)
6. Foster, I., Kesselman, C.: The Grid: Blueprint for a New Computing Infrastructure (2nd Edition). Morgan Kaufmann (2003)
7. Georges, A., Buytaert, D., Eeckhout, L.: Statistically rigorous java performance evaluation. SIGPLAN Notices 42, 57–76 (Oct 2007)
8. Google: Caliper. http://code.google.com/p/caliper/
9. Gurram, S.R., Agarwal, S.: Image compression using simple arithmetic operations. In: International Conference on Computational Intelligence and Multimedia Applications (2007)
10. Hazelhurst, S.: Truth in advertising : reporting performance of computer programs, algorithms and the impact of architecture and systems environment. South African Computer Journal 46, 24,37 (2010)
11. Hermelin, D., Rawitz, D., Rizzi, R., Vialette, S.: The minimum substring cover problem. Information and Computation/information and Control - IANDC 206, 1303–1312 (2008)
12. Hofschuster, W., Krämer, W.: C-xsc 2.0 - a c++ library for extended scientific computing. Lecture notes in computer science 2991, 259–276 (2004)
13. Huynh, D., Knezevic, D., Peterson, J., Patera, A.: High-fidelity real-time simulation on deployed platforms. Computers & Fluids 43, 74–81 (2011)
14. Lee, S., Yang, B.S., Kim, S., Park, S., Moon, S.M., Olu, K.E., Altman, E.: Efficient java exception handling in just-in-time compilation. In: ACM Java Grande 2000 Conference (2000)
15. Mateos, C., Zunino, A., Hirsch, M., Fernández, M.: Enhancing the BYG gridification tool with state-of-the-art Grid scheduling mechanisms and explicit tuning support. Advances in Engineering Software 43(1), 27–43 (2012)
16. Murray, D., Yoneki, E., Crowcroft, J., Hand, S.: The case for crowd computing. In: In 2nd. ACM SIGCOMM Workshop on Networking, Systems, and Applications on Mobile Hand-helds (2010)
17. Nicolaos, A., Vasileios, K., George, A., Harris, M., Angeliki, K., Costas, G.: A data locality methodology for matrix-matrix multiplication algorithm. Journal of Supercomputing 59, 830–851 (2012)
18. Ordiales Coscia, J.L.: Evaluación de las capacidades computacionales de dispositivos móviles. http://www.exa.unicen.edu.ar/ cmateos/files/OrdialesCoscia-2011.pdf (2011)
19. Papadimitriou, S., Terzidis, K., Mavroudi, S., Likothanassis, S.: Exploiting java scientific libraries with the scala language within the scalalab environment. IET Software 5, 543–551 (2011)
20. Paradiso, J.A., Starner, T.: Energy scavenging for mobile and wireless electronics. IEEE Persvasive Computing 4, 18–27 (2005)
21. Rodriguez, J.M., Zunino, A., Campo, M.: Mobile grid seas: Simple energy-aware scheduler. In: 39 JAIIO (2010)
22. Rodriguez, J.M., Zunino, A., Campo, M.: Introducing mobile devices into grid systems: a survey. International Journal of Web and Grid Services 7, 1–40 (2011)
23. Sangchul, L., Wook, J.J.: Evaluating performance of android platform using native c for embedded systems. In: International Conference on Control, Automation and Systems (2010)

24. Sinschek, J., Sewe, A., Mezini, M.: Vm performance evaluation with functional models: an optimist's outlook. In: Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages. ACM, New York, NY, USA (2009)
25. Vaidya, P.M.: An algorithm for linear programming which requires o(((m+n)n2+(m+n)1.5n)l) arithmetic operations. Mathematical Programming 47, 175–201 (2006)