Copyright Notice

The document is provided by the contributing author(s) as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. This is the author's version of the work. The final version can be found on the publisher's webpage.

This document is made available only for personal use and must abide to copyrights of the publisher. Permission to make digital or hard copies of part or all of these works for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage. This works may not be reposted without the explicit permission of the copyright holder.

Permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the corresponding copyright holders. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each copyright holder.

IEEE papers: © IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The final publication is available at http://ieeexplore.ieee.org

ACM papers: © ACM. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The final publication is available at http://dl.acm.org/

Springer papers: © Springer. Pre-prints are provided only for personal use. The final publication is available at link.springer.com

Energy Consumption Analysis for Two Embedded Java Virtual Machines

Sébastien Lafond[1] and Johan Lilius[2]

[1] Turku Centre for Computer Science, Embedded Systems Laboratory
Lemminkäisenkatu 14A, FIN-20520 Turku, Finland
sebastien.lafond@abo.fi
[2] Åbo Akademi University, Department of Computer Science
Lemminkäinengatan 14A, FIN-20520 Åbo, Finland
johan.lilius@abo.fi

Abstract. In this paper we present a general framework for estimating the energy consumption of an embedded Java virtual machine (JVM). We have designed a number of experiments to find the constant overhead and establish an energy consumption cost for individual Java Opcodes for two JVMs. The results show that there is a basic constant overhead for every Java program, and that a subset of Java opcodes have an almost constant energy cost. We also show that memory access is a crucial energy consumption component.

1 Introduction

Several techniques have been developed to optimize the energy consumption of embedded systems. Those techniques can be hardware based solutions, as well as software or co-design solutions [1]. Techniques for low power optimization of software have been mostly applied on processor instructions level [2, 3] by mainly using processor specific instructions [4, 5]. Techniques on memory management have also been widely applied for optimizing energy consumption [6, 7].

At the same time, the size and complexity of applications and development constraints like getting the product to market on time, make necessary the use of high-level languages. Due to the wide diversity of hardware and OS used in the world of handheld devices, portability across systems is not easy and needs efforts. Java language eases portability by allowing application developments with an abstraction of the target platform, making the concept "write once, run it anywhere" possible.

In this paper we present a general framework for estimating the energy consumption of an embedded Java virtual machine. We present a number of experiments to estimate the constant overhead of the JVMs energy consumption and establishe an energy consumption cost for individual Java Opcodes.

The major contributions of this paper are a better understanding of the energy consumption distribution of an embedded Java virtual machine (JVM) and the definition of the energy cost for the Java bytecodes for two different embedded JVMs.

The remainder of this paper is organized as follows. Section 2 presents the two JVMs used in this study, and proposes a methodology scheme used to characterize the energy consumption of an embedded JVM. Section 3 presents several experiments in order to define some constant overheads of the JVMs and comments the repartition of the JVMs energy consumption. Section 4 presents a measurement methodology used to define the energy cost of Java bytecode by cost comparison between two appropriate class files. Finally, section 5 concludes the paper and suggests future possible work. This paper extends [8] with a result comparision between two embedded JVMs.

2 An energy consumption model of Java applications

The Java Virtual machine is an abstract machine, making the interface between platform independent applications and the hardware, through a possible operating system. Thus the use of Java language can be seen as adding one more layer, the Java virtual machine, between the hardware and software layers. We want to study how well applying estimation techniques on the virtual machine opcodes level can be done, similarly to what has been done on processor instructions level. Figure 1 shows a simple view of a JVM life cycle. An efficient energy model should characterize each stage of the life cycle model, and thus shows in which stage(s) effort needs to be concentrated to achieve energy optimization. It seems obvious that such model needs to consider the system's hardware and software configuration and therefore is not directly portable. But the methodology used to build it can easily be applied on different configurations by changing the platform configuration parameters. As shown in [9] the memory consumption must also be included in the model, as the memory might represent the biggest source of energy consumption on a typical embedded system. This is even more important to take into account as the JVM is a stack machine and will therefore have a relatively high memory activity.

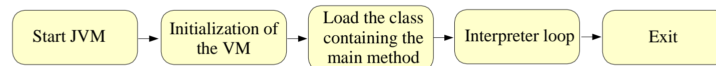

Fig. 1. Simple view of the JVM life cycle

2.1 Measurements methodology

We chose to use the Sun Microsystems K Virtual Machine (KVM), CLDC v1.0.3, and the simple Real-Time-Java (simpleRTJ) vitual machine. KVM is a small virtual machine containing about 50-80 Kb of object code in its standard configuration and has a total memory footprint in the range of 128-256 Kb. KVM

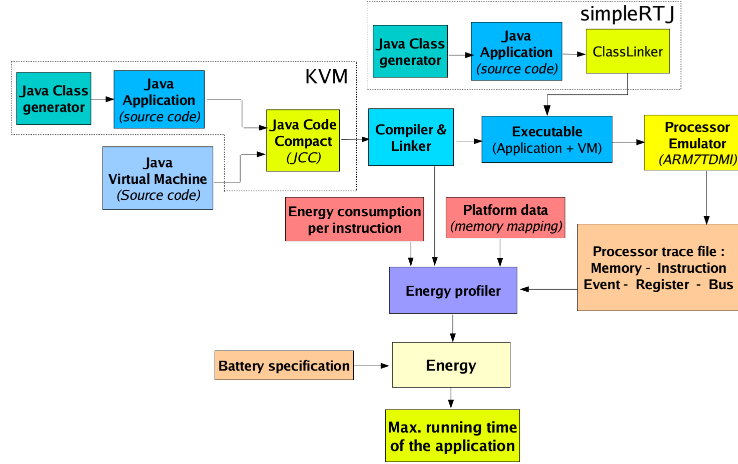

Fig. 2. Measurements methodology scheme

can run on a 16-bit or 32-bit RISC/CISC processor clocked from 25MHz. The simpleRTJ is a tiny JVM targeting 8/16/32 bit embedded systems and requiring on average about 18-24KB of code memory to run.

To build an energy model of the JVMs we adapted the energy profiler *enprofiler* [10] developed by the Embedded Systems Groups at Dortmund University. The adaptation was done in order to integrate the Java environment in the results provided by the energy profiler. With the adaptation, when summing up the energy cost for each instruction execution or memory access the *enprofiler* checks in which JVM stage the event occurred and increments the corresponding costs variable. Enprofiler is a processor instructions level energy profiler for ARM7TDMI processor cores operating in Thumb mode [11] and integrating the consumption of memory accesses. It has been built from physical measurements done on an Atmel AT91EB01 evaluation board consisting of a AT91M40400 processor clocked at 33MHz and an external 512K bytes SRAM. A detailed description of the energy model used by *enprofiler* is given in [12]. According to [12] *enprofiler* shows a precision of 1.7% for the cost measurement of 12 instructions in an endless loop.

Figure 2 shows the measurements methodology scheme used to characterize each stage of the JVM life cycle. The Java class generator generates, from template classes, Java applications with the possibility to modify parameters inside the class source code. With the Java Code Compact (JCC) we compile and link together the KVM source code and the generated Java application. For simpleRTJ the java application is pre-linked with all needed classes into a single binary image. The executable code is run on the ARM7TDMI emulator ARMulator, which traces instructions, memory accesses and events that occur during the application execution. From this trace, we extract all information

concerning the memory access addresses, size and type (read, write, sequential, non-sequential), the instructions addresses and their corresponding processor opcodes. The energy profiler *enprofiler* reads the emulator trace and accesses databases providing processor instruction costs and the cost of a memory access depending of its address, size and type. The energy profiler estimates the energy consumed by the application and provides information on how the energy is distributed between the processor and memories for each JVM stage.

3 Experiments

We have run the measurement process over several representative benchmarks to characterize each stage of the JVMs life cycle and determine if some stages are dominant. We used as reference an empty application in order to reflect the JVMs basic costs. Dedicated intensive allocation applications was also used in order to study the behavior of the JVMs stage costs.

3.1 Benchmarks

Empty application: We run the empty application through the measurement process in order to find out if overhead constants in the JVMs energy consumption can be determined. Its source code is the following:

```
public class HelloWord {
 public static void main(String arg[])
  {
  //nothing to do
  }
}
```

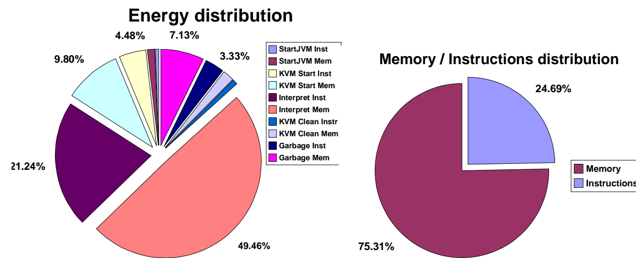

Fig. 3. Empty Application - KVM Energy consumption distributions

Intensive allocation applications: Two intensive allocation applications were used in order to study a possible application related evolutions in the JVMs costs. The first application, called alloc1, instantiates inside a loop one object of class MyClass. This class doesn't contain any field and has just one *main* method. Each new class MyClass created by main is stored in the heap, and will contain only a reference to the class definitions area. Each instantiation will create a new stack frame and call the MyClass constructor which by default will only call java/lang/Object constructor method. The stack frame created by the main method contains two operand stacks and three local variables to store the object reference, the length and the loop index. This application is used to observe the evolution of different KVM stage costs with the length of the loop. The source code for alloc1 is the following:

```
public class MyClass {
  public static void main(String arg[])
  {
   int length = X;
   for(int i=0; i<=length ; i++) {
   new Myclass();
   }
  }
}
```

The second intensive allocation application, called alloc2, is similar to the precedent one with the difference that MyClass contain one field define by an integer array of size 500. Alloc2 is used to observe the weight that can take the garbage collector in comparison to the other JVMs stages in extreme allocation rate. As each new instance takes approximatively 2Kb, with an heap size of 128Kb the garbage collector needs to be triggered every 64th objects created in the loop to reclaim the heap space occupied by the unreferenced objects. The source code for alloc2 is the following:

```
public class MyClass {
int[] tab = new int[500];
  public static void main(String arg[])
  {
   int length = X ;
   for(int i=0; i<=length ; i++) {
   new Myclass();
   }
  }
}
```

3.2 Results

This section presents the results obtained by the introduced applications through the measurement process.

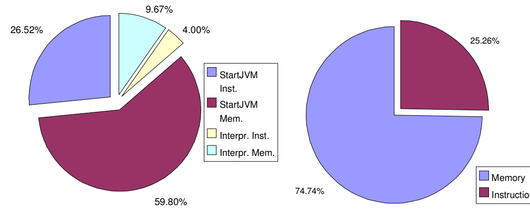

Fig. 4. Empty Application - simpleRTJ Energy consumption distributions

Empty application: The empty application has been used in order to find out if overhead constants in the JVMs energy consumption can be determined.

Tables 1 and 2 show the obtained results for respectively KVM and simpleRTJ. Figures 3 and 4 present the energy consumption distribution among repectively all KVM and simpleRTJ stages. The distribution between the energy consumed by memory accesses and processor instruction execution is also presented on these figures.

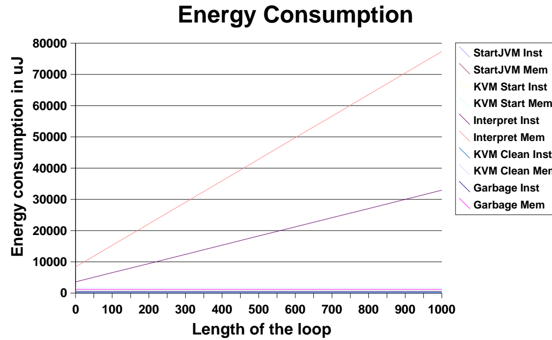

Fig. 5. Alloc1 - KVM's stages energy consumption depending of the loop length

Table 1. Empty application - Energy consumption of KVM's stages in μJ

StartJVM Inst.	StartJVM Mem.	KVMStart Inst.	KVMStart Mem.	Interpr. Inst.	Interpr. Mem.
9,42	20,08	748,81	1639,18	3552,28	8273,34
		KVM Clean Inst.	KVM Clean Mem.		
		144,92	326,38		

For the simpleRTJ virtual machine we defined only one initialization stage and any cleanup or post interpreter stage. This is because (a)from its code imple-

Table 2. Empty application - Energy consumption of simpleRTJ stages in μJ

StartJVM Inst.	StartJVM Mem.	Interpr. Inst.	Interpr. Mem.
10601,52	23905,32	1599,04	3866,66

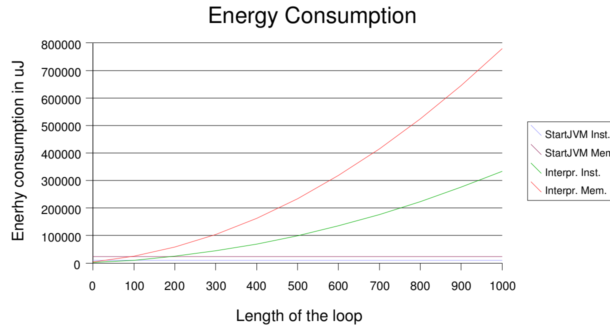

Fig. 6. Alloc1 - simpleRTJ's stages energy consumption depending of the loop length

mentation it is logical to keep its initialization stage StartJVM as a single block, and (b)the simpleRTJ exit almost immediately after last opcode is executed.

From figures 3 and 4 we can already notice that the energy distribution between memory access and instruction execution is similar between the two JVM. The second observation is the difference of weight the interpreter takes for executing the empty application. As we will see later, this difference can be explained by the fact that the simpleRTJ implementation implies more expensive heap allocations than the KVM implementation. This reduce the simpleRTJ interpreter weight compare to the StartJVM stage weight.

As the application was *'empty'* the values in table 1 represent the virtual machines basic costs or the minimal overhead energy cost that any application will have to dissipate.

Intensive allocation applications: From the alloc1 results in figures 5 and 6 we note that only the energy consumed by the interpreter is dependent on the loop length value. All other stages of the JVMs consume a constant energy including the garbage collector, as the maximum number of created object was not enough to fill up the Java heap and trigger off a garbage collection. It is important to notice that the evolution of the interpreter stage energy consumption with the loop length is different between KVM and the simpleRTJ. For KVM the interpreter stage cost is linear and proportional to the loop length, whereas simpleRTJ interpreter stage cost is exponential to the length of the loop. This difference can be explain by looking at each JVM implementation for allocating new object into the heap. KVM uses a list of free memory chunk available in the heap. For each new allocation it traverses the list until it finds a free chunk enough big to hold the new object. In our case it will always find the first chunk

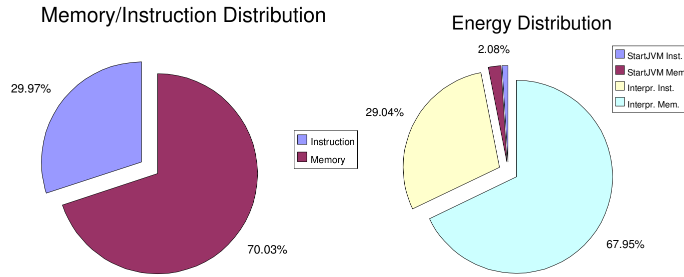

Fig. 7. Alloc1 - RTJ Energy distribution for loop length equal to 1000

available to store the new *MyClass* object, thus executing each time the same successive instructions. As a consequence the KVM interpreter stage cost will be linear and proportional to the loop length. SimpleRTJ uses only a list of object allocated in the heap. Each object contains a flag telling if the object is actually free space or not. For each new allocation simpleRTJ is traversing the list to find possible object having his flag set and enough big to hold the new object. If none is found simpleRTJ will allocate a new memory bloc. In our case for each new *MyClass* object allocation simpleRTJ will first probe all already allocated *MyClass* objects before allocating a new memory bloc, thus executing each time a exponential number of instructions. As a consequence simpleRTJ interpreter stage cost will be exponential to the length of the loop.

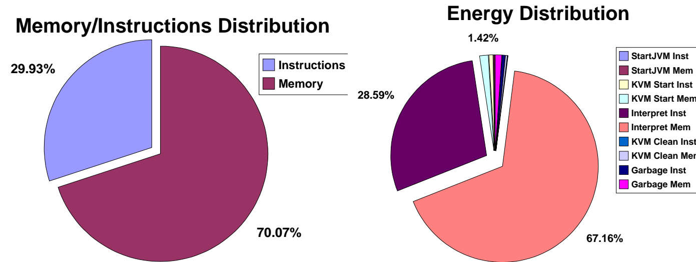

Fig. 8. Alloc1 - KVM Energy distribution for loop length equal to 1000

The energy distributions for a loop length of 1000 presented in figures 8 and 7, are similar to the first experiment with an interpreter stage even more dominant, representing over 95% of the total energy consumed.

Alloc2 application was used to observe the garbage collector weight in comparison to other JVMs stages. Several factors can influence the garbage collection behavior and thus its energy consumption: the size of the heap, the sizes and numbers of live or dead objects,heap fragmentation and naturally the technique used to implement it. Both JVMs use a mark and sweep garbage collection al-

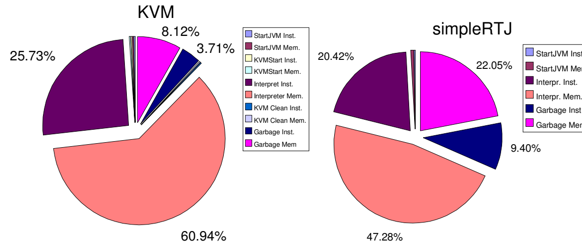

Fig. 9. Alloc2 - Garbage collection weight for KVM and SimpleRTJ

gorithm, with the difference that KVM implementation tries to do all its work in a single pass without any recursive calls.

Figure 9 presents the garbage collector weight for a loop length of 1000. We can observe that for an very intensive allocation rate of *dead objects* the KVM GC energy consumption represents only 10% of the total KVM energy consomption. On the other hand with the same application and parameter the simpleRTJ GC will represent almost one thrid of the total simpleRTJ energy consomption. This major difference is comming from the implementation variance between the JVM garbage collections.

We also run the measurement process with simpleRTJ over the all representative benchmarks presented in [8], and have the same following observation than in [8]: from all experiments done it is clear that the interpreter stage is far ahead the main source of energy consumption. Thus a better comprehension of it is needed if someone wants to achieve energy optimization on the JVMs.

As the interpreter reads and executes the Java bytecode, having a closer view on the interpreter implies increasing the granularity of its energy consumption model by looking at the cost of each Java opcode interpreted.

4 Java opcode energy cost

In order to get a better understanding of the interpreter energy consumption, an evaluation of each Java opcode energy cost is needed. As a strict class file structure needs to be respected, it is not possible to only execute one Java opcode. Thus a cost comparison between two class files is needed to estimate the cost difference between them. The general measurements methodology scheme used to characterize each JVMs stage life cycle can be re-used with different inputs. Instead of using Java source code files we will use as input appropriate byte-code executable class files.

4.1 Measurements methodology

Figure 10 shows an abstract view of the class files generator used to create two class files, named ClassFile and ClassFile_Ref. The opcode behavior towards

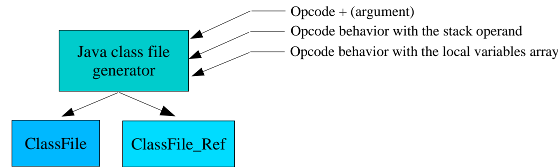

Fig. 10. Bytecode executable class file generator

the Java operand stack and the local variables array has to be defined for each studied Java opcode, i.e. provide the operand stack state needed before and resulting after the studied opcode execution as well as the number of local variables needed.

To ensure the estimation quality for each opcode we generate several pairs of class files executing the studied opcode and also monitor the possible energy consumption differences between all other JVMs stages.

4.2 Results

From all Java opcodes we will not study the 51 opcodes which handle floating point values as floating point is not supported by the CLDC specification. In addition as the simpleRTJ VM does not support long type, all opcodes manipulating longs are only analysed for the KVM. The opcode *athrow* was also omitted from this study, it is not possible to directly estimate its energy cost using this comparison method as its cost can not be extracted from the context cost. All the same, in table 5 in [13] we can see from the opcode *checkcast* the cost of throwing an *ClassCastExeption* exception and exiting the KVM.

As a general observation we can say that for most opcodes simpleRTJ gives a more expensive implementation in terms of energy and number of cycles than KVM. However the cost differences between opcode functional groups within each virtual machine are similar. Due to space requirement all obtained values for each studied opcode and each JVM are published in [13], where the opcodes are divided in six functional groups:

Stack and local variable operations opcodes : Tables 2 and 3 in [13] show the results concerning opcodes that operate on the operand stack and local variable. We can notice that loading a value from the local variables array to the operand stack is lightly more expensive than storing the same value back to the local variable. It is also interesting to note that for KVM the opcode *bipush* consumes about 9% less energy than *iload* and 5% less than *ilaod_x*. Thus it is more energy efficient to load an constant integer lower than 256 into the operand stack using *bipush* than initializing the local variable array with the constant and use *iload* or *ilaod_x*. The same is true if a constant integer lower than 65536 has to be loaded into the operand stack, it will be more efficient to use the opcode *bipush* instead of *iload*. But in case the integer constant can be stored in the first 4 local variables then *iload_x* becomes the most efficient opcode.

Type conversion opcodes : Table 1 in [13] shows the results for opcodes that convert value from one primitive type to another. The costs are in the same range as the stack and local variable operations opcodes as the conversion opcodes pop a value from the stack, perform a right shift or truncate the popped value and push back the result.

Arithmetic opcodes : Table 4 in [13] shows the costs for arithmetic opcodes. As it was easy to predict, the cost of an arithmetic operation is dependent on the type of the operands and the operation. For the KVM operations on long types are about 50% more expensive than on integers, except for the division of types long which is about two times more expensive than to divide integers.

Logic opcodes : As for the arithmetic opcodes, the cost of logic opcodes is also depending of the type of the operand and for the KVM operations on longs are from 23% to 37% more expensive than operation on integers. Table 9 in [13] shows the costs for logic opcodes.

Control flow opcodes : The control flow opcodes are the opcodes that implement the following Java language statements: *do-while*, *while*, *if*, *if-else*, *for* and *switch*. Table 8 in [13] shows the cost for the 25 control flow opcodes. For all conditional *if* opcodes (i.e. opcodes from 0x99 to 0xa6 and *ifnull*, *ifnonnull*) the energy cost depends on a two values comparison success. If the comparison success the VM jumps to a target defined by the opcode operands, in the other case the VM continues by executing the following opcodes.

The *tableswitch* opcode performs the same task as *lookupswitch*, with the difference that it requires a consecutive list of case values contained between one low and high endpoint. Thus the VM knows in advance the position of all case values so that the retrieving cost is the same for all cases. Compared with *lookupswitch*, *tableswitch* has a lower energy cost but generates all the more bigger class file size as the gape between the case values is great.

Objects and arrays opcodes : Tables 5 and 6 in [13] show the cost of opcodes that create and manipulate arrays and objects. The creation cost, with *newarray*, of a single dimension array of primitive type integer, long, short, byte, char or boolean is not directly dependent on array type and size, but more on the memory size that needs to be allocated for its creation. That means that the creation cost is identical for an integers array of size 8, a shorts array of size 16, or for the KVM a longs array of size 4. The creation cost, with *multiarray*, of multidimensional arrays is dependent on the array dimensions and dimensions indexes values. Each dimension adds a basic cost to the array creation cost, thus creating a 2*2*2 integers array will be 70% more expensive than creating a 2*4 integers array, and especially 18 times more expensive than creating a single dimension integers array of size 8.

The objects creation cost depends on the objects themselves, i.e on the type and size of their constant pool, interfaces, fields,methods and their super-classes, and also on their resolution flags inside each class constant pool. A new object is

resolved only once within a same class, and its address is stored in the constant pool structure of the class. Table 5 in [13] shows as an example the creation cost of an object of type *java.lang.Object* and *java.lang.String*. In addition, table 5 in [13] refers to two objects called *Class* and *subClass* which is a empty (none interface,field nor method) sub class of *nonResolvedClass* itself empty sub class of *java.lang.Object*.

Method invocation and return opcodes Because invoking a method implies returning from it at some point, table 7 in [13] shows the costs of different invoke/return pairs. They all invoke an empty 'already resolved' method within the same class or instance. We can notice from this table that calling a static, public or private method costs almost the same, and that the type of the returned value has not a great influence on the overall cost.

It is also important to compare all obtained values with the *NOP* energy consumption. As the opcode *NOP* is the first case statement in the interpreter switch and doesn't execute any instruction, its energy consumption represents the minimum overhead cost due to the interpreter mechanism. For the most of the stack and local variable operation opcodes the interpreter mechanism overhead represents about 70% of their energy consumption.

4.3 Opcode costs verification

In order to verify the obtained opcode costs we calculated for each benchmark execution used for the first experiments the value $\sum(Opcodecost*OpcodeOccurrence)$. The computed value was then compared with the cost given by the energy profiler for the interpreter stage. KVM has a build-in implementation to trace all executed opcodes. We also added such feature to the simpleRTJ VM in order to calculate the occurrence of each opcode. For control flow opcodes we checked if the branch was taken or not to attribute the correct opcode cost, but to keep the verification simple we didn't looked at the type of variable handled by $putfield$, $getfield$, *putstatic* and *getstatic*. There respective cost for handling integer was used for all occurrences. In addition for all other none static opcode costs only the respective basic cost was used. The benchmark *Exception* from the Java Grande Forum Benchmark Suite was not used as we didn't studied the cost for the opcode *athrow*.

Table 3. Verification results

	Dhrystone50	Arith	Assign	Loop	Create	Method	Math	Generic
KVM	103,99	105,31	95,55	100,30	97,95	102,51	96,74	109,43
sompleRTJ	102,35	101,56	98,75	102,28	100,15	103,12	98,95	103,34

Table 3 presents the normalized verification results where the value 100 represent for each benchmark the energy cost given by the energy profiler for the inter-

preter stage. For each benchmark the accuracy obtained by calculating the value $\sum(Opcodecost * OpcodeOccurrence)$ is staying between -5 and +10% of the cost given by the energy profiler. But this loss in precision has to be balanced with the time needed to compute it. It takes only few seconds to calculate the occurrence for each opcode and compute the value $\sum(Opcodecost * OpcodeOccurrence)$, compared to several hours needed by the energy profiler.

5 Conclusion

Several observations have been done in this paper concerning the energy consumption of the JVMs. For the hardware configuration fixed by the energy profiler, the distribution between the processor and memories is constant over the JVMs execution with 70% of the energy consumed by memory accesses. This shows the major importance of the memories for embedded system runtime performance. We also showed that implementation differences between two embedded JVMs can imply great divergence concerning the JVM energy consumption.

This paper can also guide developers to produce energy-aware java application by limiting the use of long data type, avoiding multidimentional array and trying to use consecutive case values inside a switch statement. Furthermore, the opcodes energy cost can be helpful for developing a energy-aware Java compiler as well as optimizing the JVMs by pointing out the expensive opcodes. This paper shows the first steps toward an energy aware performance analysis tool for Java application, as a such tool would ask for a more detailed model for a subset of opcodes.

Also as the interpreter mechanism overhead cost is a predominant factor in opcode execution cost, it will be interesting in the future to look at the cost differences between the two possible Java execution modes: interpreted or JIT compilation. JIT compilation increases significantly the execution speed, but in the same time increases memory footprint. A trade-off between execution time and memory footprint size will certainly have to be found to reach the optimum optimization point for energy consumption.

References

1. F. Parain, M. Banatre, G. Cabillic, T. Higuera, V. Issarny, and J-P Lesot. Techniques de reduction de la consommation dans les systemes embarques temps-reel. Technical report, INRIA Rennes, 2000.
2. Vivek Tiwari and Sharad Malik and Andrew Wolfe. Power Analysis of Embedded Software. In *International Conference on Computer-Aided Design, San Jose CA.*, nov 1994.
3. Anil Seth, Ravindra B Keskar, and R. Venugopal. Algorithms for energy optimization using processor instructions. In *International conference on Compilers, architecture, and synthesis for embedded systems- Atlanta, Georgia, USA*, 2001.
4. Wen-Tsong Shiue. Retargetable Compilation for Low Power. Technical report, Silicon Metrics Corporation.

5. Mike Tien-Chien Lee, Vivek Tiwari, and Sharad Malik. Power analysis and low-power scheduling. In *International Symposium on System Synthesis*, sep 1995.
6. Catherine H. Gebotys. Low Energy Memory and Register Allocation Using Network Flow. In *Design Automation Conference*, pages 435–440, jun 1997.
7. X. Fan, C. Ellis, and A. Lebeck. Memory controller policies for DRAM power management. International Symposium on Low Power Electronics and Design (ISLPED), aug 2001.
8. Sébastien Lafond and Johan Lilius. An energy consumption model for an embedded java virtual machine. In *ARCS*, pages 311–325, 2006.
9. Kaushik Roy and Mark C. Johnson. Software design for low power. In *Low Power Design in Deep Submicron Electronics*, pages 433–460, 1997.
10. Enprofiler. http://ls12-www.cs.uni-dortmund.de/research/encc/.
11. An introduction to thumb. Technical report, Advenced RISC Machines Ltd, 1995.
12. Stefan Steinke, Markus Knauer, Lars Wehmeyer, and Peter Marwedel. An accurate fine grain instruction-level energy model supporting software optimization. In *PATMOS 01*, 2001.
13. http://www.abo.fi/˜slafond/javacosts.