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

An Energy Consumption Model for an Embedded Java Virtual Machine

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 establish 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 of the Virtual Machine and establish an energy consumption cost for individual Java Opcodes. 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

In recent years we have seen an explosion of markets for portable electronic devices such as PDAs, personal communicators and mobile phones. These battery-operated devices provide more and more functionalities and as a consequence become more and more complex. They have in common strong constraints on energy consumption, and thus maximizing battery life for such devices is crucial.

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 establish 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 JVM 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.

The remainder of this paper is organized as follows. Section 2 proposes a methodology scheme used to characterize the energy consumption of an embedded Java Virtual Machine. Section 3 presents several experiments in order to define some constant overheads of the JVM and comments the repartition of the JVM 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 is presenting the main results of [8] where more example graphs and results can be found.

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 the 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.

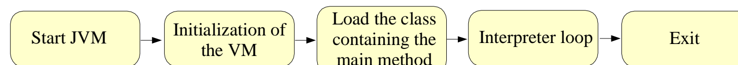

Fig. 1. Simple view of the JVM life cycle

2.1 Measurements methodology

We chose to use the Sun Microsystems K Virtual Machine (KVM), CLDC v1.0.3, as it has been developed for a resource-constrained platform and has its source code freely available. 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 can run on a 16-bit or 32-bit RISC/CISC processor clocked from 25MHz.

To build an energy model of the KVM 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 KVM 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 KVM 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 JVM source code and the generated Java application. 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 KVM stage.

2.2 Energy profiler

The energy profiler provides the number of instructions, memory accesses and garbage collections that occur during each KVM stage. It needs as input information on the JVM stage addresses inside the emulator trace. These addresses are provided by the linker from which eight useful address symbols are collected :

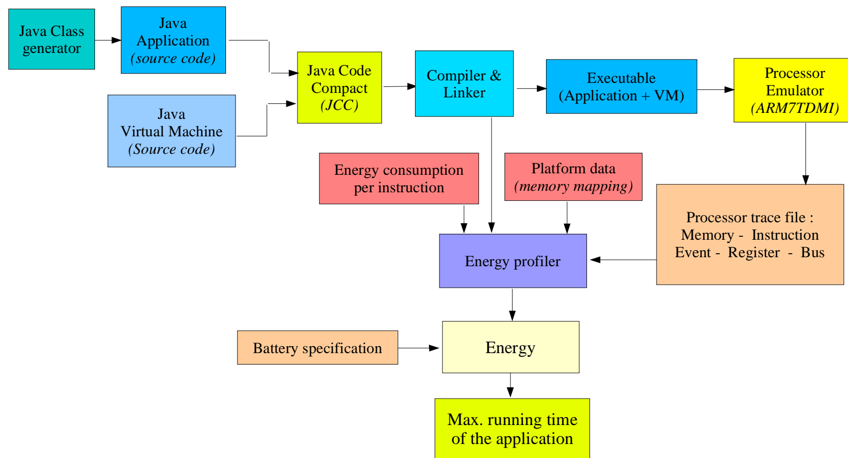

Fig. 2. Measurements methodology scheme

– main: this symbol represents the *main()* function of KVM, and is used by the energy profiler to detect the start of the KVM execution.
– StartJVM: represents the *StartJVM(argc, argv)* function (in StartJVM.c source file). This function only checks if the user gave a class name as argument, and then calls the *KVM_Start()* function.
– KVM_Start: represents the *KVM_Start()* function (in StartJVM.c source file). This function initializes the VM, the global variables, the profiling variables, the memory system, the hashtable, the class loading interface, the Java system classes, the class file verifier and the event handling system. It also initializes the multithreading system after loading the main application class.
– garbageCollect: represents the *garbageCollect()* function (in garbage.c source file) that performs a mark-and-sweep garbage collection.
– ExitGarbage : the ExitGarbage symbol was added into the KVM source code in order to detect the end of the garbage collector.
– Interpret : represents the *Interpret()* function (in execute.c source file) that runs the interpreter loop.
– KVM_Cleanup : KVM_Cleanup represents the *KVM_Cleanup()* function (in StartJVM.c source file). It runs several finalization functions when the VM is shut down.
– ExitVM : This symbol is used to detect the end of the KVM execution.

3 Experiments

We have run the measurement process over several representative benchmarks to characterize each stage of the KVM life cycle and determine if some stages are dominant. The benchmarks used are: a) the dhrystone benchmark, b) parts of The Java Grande Forum Benchmark Suite and the DHPC Java Grande Benchmarks. In addition to these established benchmarks we also used as reference an

empty application in order to reflect the KVM basic costs. Dedicated intensive allocation applications was also used in order to study the behavior of the KVM stage costs. All benchmarks can be retrieve from [13]. For all measurements, if not explicitly expressed the KVM was compiled with an heap size of 256 Kb.

3.1 Benchmarks

Empty application: We run the empty application through the measurement process in order to find out if overhead constants in the KVM energy consumption can be determined. We can predict that one or several stage(s), like StartJVM, will have a constant energy consumption, as they have an application independent behavior. Its source code is the following:

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

Intensive allocation applications: Two intensive allocation applications were used in order to study a possible application related evolutions in the KVM 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 KVM 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();
   }
  }
}
```

Dhrystone: Dhrystone tests the system's integer performance. It is a well established benchmark for performance measurement of general purpose system. We conducted the measurement process with two test executions of 50 and 250 benchmark runs.

Table 1. Benchmarks used from Java Grande Forum Benchmark suite

Low level operation benchmarks	
Name	Short description
Arith	Execution of arithmetic operations
Assign	Variable assignment
Create	Creating objects and arrays
Exception	Exception handling
Loop	Loop overheads
Math	Execution of maths library operations
Method	Method invocation
Generic	Local and Static variable handling

Java Grande Benchmarks: We used the sequential benchmarks which are the one suitable for single processor execution. Several low level operation benchmarks was used from the Java Grande Forum Benchmark Suite and the DHPC Java Grande Benchmarks. Table 1 summarize all benchmarks used for our study.

3.2 Results

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

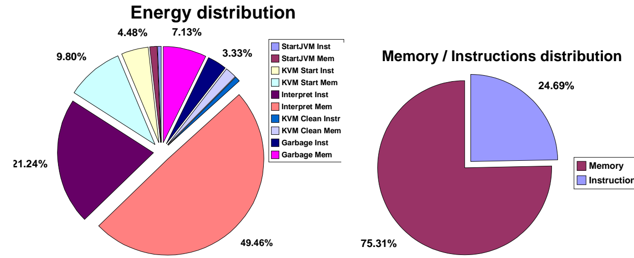

Fig. 3. Empty Application - Energy consumption distributions

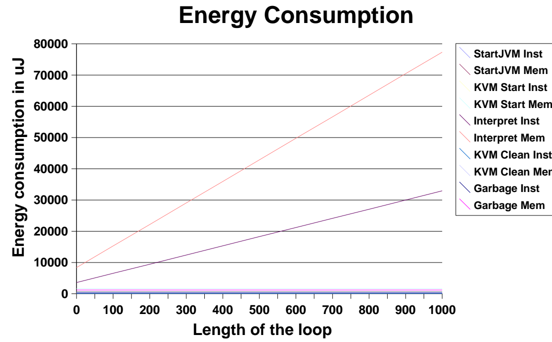

Fig. 4. Alloc1 - KVM's stages energy consumption depending of the loop length

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

Table 2 shows the obtained results and figure 3 presents the energy consumption distribution among all KVM stages and also the distribution between the energy consumed by memory accesses and processor instruction execution.

Table 2. 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		

We can make some remarks from figure 3. Even if this application does absolutely nothing, it has to be noticed that the interpreter stage represents about 70 % of the consumed energy from all stages, and memory accesses represent 75% of the total consumed energy. As the application was $'empty'$ the values in table 2 represent the KVM basic costs or the minimal overhead energy cost that any application will have to dissipate.

Intensive allocation applications: From the alloc1 results in figure 4 we note that only the energy consumed by the interpreter is dependent on the loop length value. All other stages of the KVM 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 also important to notice that the energy consumed by the interpreter stage is linear and proportional to the loop length. This can be explain by the fact that the interpreter is looping over a number of constant Java opcodes. These opcodes are:

```
4 goto 18
7 new\#2  -> create a new 'MyClass' object in the heap
10 dup    -> duplicate new object reference in the operand stack
11 invokespecial \#3 -> call the constructor
14 pop -> remove the top of the operand stack
17 iinc 2 1 -> increment the second local variable by 1
18 iload\_2 -> load 2nd local variable in operand stack (i)
19 iload\_1 -> load 1st local variable in operand stack (length)
20 if\_icmple 65543
```

As the energy profiler evaluates the cost of a memory access according to the memory technology, i.e. have for each memory type (RAM, ROM, Flash, etc.) an average cost for each access type regardless of its address, and as the *new* opcode allocates the same amount of memory for all created (and already resolved) objects, it will have an identical cost for each execution.

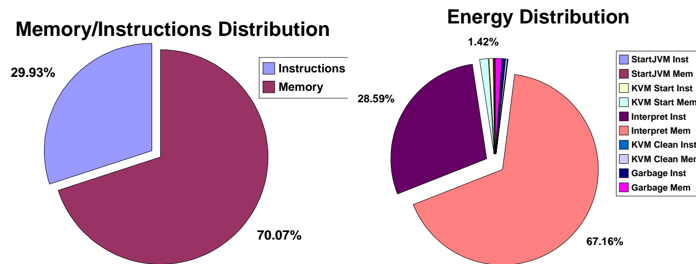

Fig. 5. Alloc1 - Energy distribution for loop length equal to 1000

The energy distribution for a loop length of 1000 presented in figure 5, is 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 KVM 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, and heap fragmentation. However, as shown on figure 6, the garbage collection stage will hardly exceed more than 15% of the total energy consumed even for application with intensive allocation rate. Table

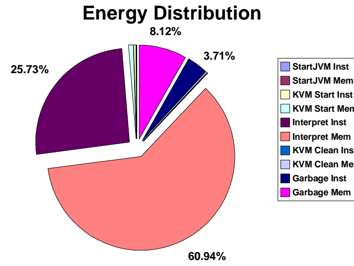

Fig. 6. Garbage collection weight

3 shows the energy values consumed by the interpreter and garbage collector for alloc2 application with a loop length of 1000 where the garbage collection represent 13,65% of the interpreter stage energy consumption.

Benchmarks: Table 4 and 5 gather the results for all benchmarks. Table 4 shows for the used benchmarks the energy values in μJ for StartJVM, KVMStart and KVMClean stages. We can notice that the obtained values for each stage are very similar for all benchmarks, and there values and variations extremely small compare to the interpreter stage values show in table 5 (in mJ). We can say that with an average of 98% of the total energy consumption the interpreter stage is fare ahead the stage where the energy consumption is dissipated inside the KVM, and that StartJVM, KVMStart and KVMClean have an almost constant and insignificant energy consumption. All measurements were done on an opteron 244 1.8GHz machine with 4Gb of RAM, and for the slowest benchmark JGFMathBench the measurement process took about 36 hours.

Table 3. Energy consumption values for a loop length of 1000 in μJ

Interpreter Inst.	Interpreter Mem.	Garb. Collect. Inst.	Garb. Collect. Mem.
54 035	127 949	7 789	17 057

From all experiments done it is clear that the interpreter stage is far ahead the main source of energy consumption and a better comprehension of it is needed if someone wants to achieve energy optimization on the KVM. 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.

Table 4. Stable energy costs for StartJVM ,KVMStart and KVMClean stages in μJ

Benchmark	StartJvm Instuction	StartJvm Memory	KVMStart Instuction	KVMStart Memory	KVMClean Instruction	KVMClean Memory
Dhrystone250	9,42	20,08	857,74	1868,40	155,41	350,31
Dhrystone50	9,42	20,08	849,82	1851,51	154,74	348,82
Arith	9,42	20,08	815,78	1776,04	145,67	328,40
Assign	9,42	20,08	823,32	1791,93	145,94	329
Create	9,42	20,08	807,81	1833,57	147,48	335,21
Exception	9,42	20,08	814,08	1772,99	145,94	329
Loop	9,42	20,08	810,01	1764,06	145,67	328,40
Method	9,42	20,08	823,89	1793,72	146,75	330,93
Generic	9,42	20,08	838,76	1828,55	152,78	344,39
Math	9,42	20,08	823,89	1793,72	146,75	330,93

Table 5. Interpreter stage energy cost and weight in mJ

	Dhrystone250	Dhrystone50	Arith	Assign	Create	Exception
Inst.	97-29.65%	88-29.30%	877-29.21%	2380-29,87%	1053-26,38%	2250-29,82%
Mem.	850-69.90%	207-68.92%	2121-70.62%	5584-70,05%	2779-69,61%	5475-70,04%

	Loop	Math	Method	Generic
Inst.	533-29,32%	2718-29,77%	533-29,86%	611-29,65%
Mem.	228-69,03%	6408-70,17%	1246-69,84%	1445-70,09%

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 KVM 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.

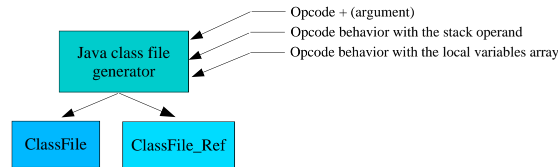

Fig. 7. Bytecode executable class file generator

4.1 Measurements methodology

Figure 7 shows an abstract view of the class files generator used to create two class files, named ClassFile and ClassFile_Ref. The opcode behavior towards

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. Figure 8 shows an example of generated bytecode classes for the Java opcode *NOP (0x00)*. In this example ClassFile method 1, the *main* method, executes 256 *NOP* opcodes when the ClassFile_Ref method 1 executes only the compulsory *return* opcode in order to return *void* from the main method. By comparing the interpreter energy consumption for both class files we can get the energy consumption estimation for 256 *NOP* executions and thus the energy cost of one *NOP* opcode.

ClassFile

```
Method 1:
0000d8 0009       access flags = 9
0000da 0008       name = #8<main>
0000dc 0009       descriptor = #9<([Ljava/lang/String;)V>
0000de 0001       1 field/method attributes:
                  field/method attribute 0
0000e0 0006        name = #6<Code>
0000e2 00000119    length = 281
0000e6 0000        max stack: 0
0000e8 0001        max locals: 1
0000ea 00000101    code length: 257
0000ee 00          0 nop
0000ef 00          1 nop
0000f0 00          2 nop
0000f1 00          3 nop
..............
0001ed 00          255 nop
0001ee b1          256 return
0001ef 0000        0 exception table entries:
```

ClassFile_Ref

```
Method 1:
0000d8 0009       access flags = 9
0000da 0008       name = #8<main>
0000dc 0009       descriptor = #9<([Ljava/lang/String;)V>
0000de 0001       1 field/method attributes:
                  field/method attribute 0
0000e0 0006        name = #6<Code>
0000e2 00000019    length = 25
0000e6 0000        max stack: 0
0000e8 0001        max locals: 1
0000ea 00000001    code length: 1
0000ee b1          0 return
0000ef 0000        0 exception table entries:
```

Fig. 8. Example of generated byte-code class files

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 KVM stages. All measurements were done on a Linux 700Mhz Pentium III machine with 256MB of RAM, and on average the estimation of a Java opcode cost took 3 minutes.

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. 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.

Due to space requirement all obtained values for each studied opcode 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 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. 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 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 KVM *lookupswitch* implementation uses the binary search algorithm to retrieve the branch offsets associated with the case values of the switch statement. In consequence, the *lookupswitch* cost depends on the number of needed iterations through the binary tree which is determined by the position of the researched case value in the tree. As on average for a binary tree of size n it takes $(\log_2 n - 1)$ iterations to found the researched value, it is possible to determine an *lookupswitch* average cost

depending on the number of case values included in the switch statement. 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 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. Moreover, in order to access to one multidimensional array value the JVM has to retrieve from the first dimension the second dimension address and so one until it reaches the last dimension.

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.

The obtained values allow us to get an estimation of how long the KVM will run for a given battery. If we suppose that on average the execution of one Java opcode consumes a total of 3.372μJ and is executed in 200 cycles, the average power dissipated by the processor (clocked at 33MHz) to execute Java opcodes is 0.556 Watt. Thus for the processor supply voltage sets at 3.3 Volts, an ideal 3.3 Volts 500 mAh battery will allow the KVM to run for 200 minutes.

4.3 Opcode costs verification

In order to verify the obtained opcode costs we calculated for each benchmark execution the value $\sum(Opcodecost * OpcodeOccurrence)$. The computed value was then compared with the cost given by the energy profiler for the interpreter stage. The occurrence for each opcode was calculated thanks to the KVM tracing ability. 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 6. Verification results

Dhrystone50	Arith	Assign	Loop	Create	Method	Math	Generic
103,99	105,31	95,55	100,30	97,95	102,51	96,74	109,43

Table 6 presents the normalized verification results where the value 100 represent for each benchmark the energy cost given by the energy profiler for the interpreter 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 lost in precision has to be balance with the time needed to compute it. It takes only few seconds to calculate the occurrence for each opcode and compute the value $\sum(Opcodecost * OpcodeOccurrence)$, compare to several hours needed by the energy profiler.

5 Conclusion

Several observations have been done in this paper concerning the energy consumption of the KVM. For the hardware configuration fixed by the energy profiler, the distribution between the processor and memories is constant over the KVM execution with 70% of the energy consumed by memory accesses. This shows the major importance of the memories for embedded system runtime performance.

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 JVM 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 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.C.T.H.V.I.J.P.L.: 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. (1994)
3. Anil Seth, Ravindra B Keskar, R.: 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, S.: Power analysis and low-power scheduling. In: International Symposium on System Synthesis. (1995)
6. Catherine H. Gebotys: Low Energy Memory and Register Allocation Using Network Flow. In: Design Automation Conference. (1997) 435–440
7. Fan, X., Ellis, C., Lebeck, A.: Memory controller policies for DRAM power management. International Symposium on Low Power Electronics and Design (ISLPED) (2001)
8. Lafond, S., Lilius, J.: An energy consumption model for java vitual machine. Technical Report 597, Turku Centre for Computer Science (2004)
9. Kaushik Roy, M.C.J.: Software design for low power. In: Low Power Design in Deep Submicron Electronics. (1997) 433–460
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, L.W.P.M.: An accurate fine grain instruction-level energy model supporting software optimization. In: PATMOS 01. (2001)
13. (http://www.abo.fi/˜slafond/javacosts)