Energy vs. Memory Trade-Offs on Energy-and-Memory Constraint Embedded JVM

Jie Tang  
Runtime Technology Lab  
Intel  
Beijing, China  
tangjie.bit@gmail.com

Chen Liu  
Department of Electrical and Computer Engineering  
Clarkson University  
Potsdam, NY, USA  
cliuclearson.edu

Abstract—Due to the thriving of the mobile devices market, there has been a great momentum in adopting Java technology in embedded mobile devices where optimizing energy consumption is extremely critical. However, since Java Virtual Machines (JVMs) are conventionally used for server side clusters, the energy and memory tradeoff on embedded devices are not well understood. In this work, we attempt to find out how the memory heap size on embedded devices (running JVMs) could impact energy consumption. To approach this problem, we use an in-house-developed production-level micro-JVM as a case study. This micro-JVM runs on top of a low-power co-processor with tight memory and battery constraints, and is designed for hosting security workloads. Focusing on security-related workloads, we first study the energy consumption distribution among JVM components. Second, we vary the size of the available physical memory, and find out how this would impact energy consumption. Third, we propose strategies to optimize memory consumption. Finally, we explore hardware strategies to further optimize energy consumption. The results show that garbage collector (GC) overhead is a function of memory heap size, the bigger heap the smaller overhead incurred. However, the increase of memory heap will not always translate into the energy reduction due to the offset from memory side energy cost.

Keywords—Java Virtual Machine, Garbage Collector, Hardware Accelerator, Energy-aware Computing

1. INTRODUCTION

Java is becoming increasingly popular in embedded environments in recent years [1]. Java enables the applications on top of it with the “written once and running everywhere” feature, leading to less development efforts and shorter development cycle. Java Virtual Machine (JVM) is natured with security due to its sand-boxing isolation and equipped with the automatic memory management which frees the developers from the prone memory leakage. These characteristics make Java-enabled embedded devices such as smart-phones, tablets and ebook-readers have grown incredibly fast.

Running Java in an embedded/portable environment, however, is not without its problems. First, most embedded devices are memory-constrained. Consequently, JVM as well as the on-top applications should be executed with a small footprint. Second, since embedded devices are battery driven, energy consumption arises as a new optimization parameter, which is as important as performance and hardware resource.

As a result, embedded JVM may need re-evaluation, and possibly redesign within these restrictions. Generally, its overall design principle is to minimize the usage of system resources, such as memory and energy. Performance turns to be a secondary goal, and a significant amount of optimizations have been purposely removed. For instance, a straightforward, small, and low-performance byte code interpreter will be the candidate instead of the JIT compiler. Also, it may remove a number of class libraries and keep only those that are absolutely necessary. For example, Sun’s KVM [3] eliminates a lot of essential JVM components, resulting in a static memory footprint of 50 kB with 14,000 lines of code (LOC).

Under this background, we perform a study on a micro Java Virtual Machine (micro-JVM) which is designed for secured embedded system. We use this JVM for running security workloads because it provides good isolation, and thus protecting the workload from the rest of the computing environment. Compared with conventional JVMs, it runs on top of a co-processor within the system, thus has more restrictions on memory, performance and power.

We study how micro-JVM’s garbage collector (GC) with different heap size deviating both the performance and energy consumption efficiency. The results show that with a very small heap/memory, GC would occur often, leading to massive CPU activities and energy inefficiency. On the other hand, with a large heap/memory, although GC frequency would be greatly reduced, the memory leakage power may overwhelm. To alleviate this problem, we also show that GC hardware accelerators can efficiently reduce GC energy overheads, and thus enable embedded JVMs to run on extremely memory-constrained devices.

The rest of the paper is organized as follows. We review the background research in Section II. We introduce the experimental methodology in Section III. We present the performance evaluation results and energy evaluation results in Sections IV and V, separately. The case study on the GC hardware accelerator is presented in Section VI. Finally, we conclude our findings in Section VII.
II. BACKGROUND

Performance optimization of JVM has been studied extensively. As Java gains more popularity in embedded systems, the optimizations dedicated to power/energy consumption emerge and keep rising [4]. The garbage collector in JVM is the pure performance consumer. For example, GC has been found to take over 10% of the average execution time [5] and in some JVM implementations this number will rise to as much as 43% [10]. Hence, it also gains great attention on how to get more energy efficiency if finely tuned.

For simplicity and limitation of the small memory footprint, in embedded JVM it is recommended to use Mark-Sweep Garbage Collector (MS GC) algorithm [6, 7]. When the heap has no more free space, Mark-Sweep GC is triggered and the collectors start to trace the heap. If an object can be reached, then it is a live object and the status is marked in the object header or some other metadata area. After tracing, all unmarked objects are swept and their occupied spaces are recycled. The free space in the heap is managed with a linked list. It will yield high GC efficiency, but suffer from heavy fragmentation. This fragmentation will produce poor data locality. In the worst case, JVM even cannot find a proper space for the newly allocated object.

In [8], authors presented a GC-controlled leakage energy optimization technique that shuts off memory banks that do not hold live data. A variety of parameters, such as memory bank size, the garbage collection frequency, object allocation style, compaction style, and compaction frequency, are tuned for energy saving. As the continuing research, they also made a study on energy behavior of Java application from the memory perspective; however, they only focused on dynamic energy consumption [9]. Paul Griffin et al. presented an energy consumption comparison between two well-known garbage collection algorithms - mark-sweep compact and reference counting (RC). They found that an efficient RC implementation can facilitate a dramatic reduction in GC-invocations with a relatively simple allocator [10]. With the fabrication technology transiting from 120nm towards 32nm, Esmaeilzadeh et al. evaluated the performance and power of Java applications on IA-32 microprocessors with this technology progress [11]. Seo et al. defined a framework to estimate the energy consumption of pervasive Java-based systems [12]. They first broke the system into components and energy cost of each component was further divided into computational and communication parts.

Our work differentiates from previous related work in the aspect that we study the energy behavior of a micro-JVM embedded in a co-processor, which has very strict constrains in memory (in the scale of 50~100 kB) and energy consumption (in the scale of 50~100 mW). In the rest of paper, the GC is referred as the one implementing the Mark-Sweep GC.

III. METHODOLOGY

The design for the micro-JVM on top of the co-processor is included in Fig. 1. It consists of four common components:

- The class loader, which loads the binary representation of a class into the JVM;
- The execution engine, which generates the execution flow of the Java program;
- The memory management unit, which handles object allocation and garbage collection;
- The native method unit, which coordinates the execution of native methods with Java methods.

Besides, to communicate with the main processor, the micro-JVM has the additional component name communicator to get the commands and data from the host. To adapt to the execution in embedded environment, the micro-JVM is also equipped with the format converter, which can turn the class library and loaded application into more compact format.

Due to the design constrains, the conventional class loading mechanism is not fully developed, and only one single bootstrap class loader is used to load the applications. The garbage collector uses the simple non-compact Mark-Sweep to reduce the overheads of GC operation. In execution, interpreter is chosen for its straightforward execution style. Especially, it is very memory-constrained. Currently the memory usage of VM should fulfill the following requirements: the runtime footprint of JVM plus class library should be lower than 200kB; and the runtime footprint of Java applications should be lower than 100kB.

To study the sensitivity of performance and energy behavior to different heap sizes, we run the emulator for the co-processor in a 3.5GHz Intel® Core™ i7-3770K CPU. To capture the system-level metrics and periodical energy consumption, we take utilization of Intel® VTune™ Amplifier [13], a performance profiler and analyzer. Since the co-processor is dedicated for security execution environment, we take the java applications in Table I as the workloads for experiments.

<table>
<thead>
<tr>
<th>Name</th>
<th>Descriptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>PEAT</td>
<td>Application for private key based authentication and digital signature</td>
</tr>
<tr>
<td>MD5</td>
<td>Java implementation of MD5 encryption/decryption</td>
</tr>
<tr>
<td>DES</td>
<td>Java implementation of DES encryption/decryption</td>
</tr>
<tr>
<td>SHA1</td>
<td>Java implementation of SHA1 encoding/decoding</td>
</tr>
<tr>
<td>HASH</td>
<td>Java implementation of HASH encoding/decoding</td>
</tr>
</tbody>
</table>

Fig. 1. The structure of the micro-JVM
IV. PERFORMANCE EVALUATION

In this section, we study the GC execution behavior as a function of heap size, varied from 30kB, 50kB, to 70kB. We used the Intel VTune Amplifier to capture the fraction of the overall execution time incurred by GC operations in different heap size configurations. The results are shown in Fig. 2. The x-axis shows the selected workloads and the y-axis shows the percentage of the overall execution time that is incurred by GC with different heap size configurations. What we can see is that GC is really the time-consuming operation in JVM, even for computation-intensive encryption applications. On average, when running the five workloads, the GC overhead has reached a percentage of 31%, 18% and 13%, individually. This is because these applications always use complex data structure and make the heap consumed fast.

When the heap size increased, we can see that the overhead of the GC is reduced. In the first 20kB size increase, the overhead goes down sharply. For example, in workload PEAT it reached a 50% overhead reduction. However, the limited memory capacity made a boundary for the increase of the heap size. Hence, GC operation will be invoked frequently and produce considerable pure overhead to the performance. It can be concluded in embedded system, the constraints on memory capacity leave little room for heap and make the GC operations as the major contributor to the execution overhead.

It is the indication in performance and in the following section we will see how this performance overhead translates into the overhead of energy consumption.

![Fig. 2. GC overhead in a function of heap size](image)

V. ENERGY EVALUATION

In this section, we estimate the energy consumption of the execution with different heap size to observe GC’s overhead in terms of energy. While there is no direct way to measure this data, we utilized the counter-based power consumption approximation approach used in [14]. The counter-based energy consumption model we used is presented in Equation 1, which approximates the total system energy consumption $E$ as the summation of energy consumed by hardware components, including the processors, the system bus, and the memory modules. The energy consumed by component $i$ can be approximated by the product of the number of times the component is triggered $C_i$ and the energy consumed by the component each time $E_i$. To approximate the energy consumption of the processor, we utilized the hardware performance counter to obtain the number of retired instructions. For the system bus energy consumption, we obtained the number of bus transactions during the JVM execution. For the memory subsystem energy consumption, we monitored the number of memory accesses during the JVM execution.

$$E = \sum_{i=0}^{n} C_i E_i$$ (1)

Next, by using Intel VTune Amplifier, we captured the number of retired instructions, the number of bus transactions, and the number of memory accesses incurred by each workload execution. By taking the counter information into Equation 1 above and assuming that each activity consumes the same amount of energy (e.g., $E_i$ is the same for memory access, bus transaction, and instruction execution), we were able to approximate the energy consumption during execution; and in a similar way, we can approximate the energy consumed by the garbage collector.

In Figure 3, we present the results that total energy consumption of the JVM with different heap size configurations. The x-axis is the selected workload and the y-axis is the normalized energy consumption with the 50kB heap size as the baseline. In this way we can show how energy consumption in other configurations biases the baseline. It can be seen that in small heap sized configuration the energy is more than that in larger heap configuration. This is mainly due to the more frequently GC operation invocations. GC itself consumes sizable portion of overall energy during execution, it also influences the energy consumed in memory during the execution. When the heap size increases, the GC operation is less invoked, leading to less energy cost. However, the applications’ data locality and the number of memory banks available in the underlying architecture will have an impact on the energy efficiency of the GC operation. Especially in Mark-Sweep algorithm, its produced memory fragmentation will lead to poor data locality. Hence, in workload MD5, we can see that its energy cost increased rather than drop down.

![Fig. 3. The energy consumption of the micro-JVM](image)
In Figure 4, we present the energy breakdown of the hardware components, including memory, CPU and system bus. The x-axis is the observed components and the y-axis is the average percentage that each component takes up in total energy consumption. We get the following observations: First, even though Mark-Sweep algorithm is performance efficient, it produces a fragmented heap space which means more active banks (at a given time frame), and consequently, more energy consumption in memory. For example, in 70kB heap size configuration, memory costs as much as 38% total energy consumption; Second, embedded Java execution is going to stress the system bus more compared to conventional JVMs. Byte-codes are treated as data and need to be fetched from memory for interpretation and installed; Third, CPU is less stressed when GC is less invoked. It can be seen that in 70kB configuration, the energy cost of CPU has drop down to 30% of the total one. This is because in Mark-Sweep algorithm, it needs to scan each object the entire heap from the root set and makes mark correspondingly.

From these results, we learn the following:
- If the heap is too small, GC will get triggered more frequently, leading to extra energy consumption, mainly on the CPU.
- If the heap is too large, extra energy consumption (mainly from leakage power) would incur in the memory subsystem.

VI. CASE STUDY: HARDWARE ACCELERATION

In this section we explore how GC hardware acceleration impacts the energy efficiency of embedded JVMs. The GC accelerators that we previously developed [5, 15] apply an instruction collapse technique to execute a whole “hotspot” basic block with one or few special instructions.

Referring the data in Fig. 2, on average, the GC overhead is 31%, 18% and 13% individually for 30kB, 50kB and 70kB memory heap size setting. After we apply the hardware acceleration into the micro-JVM, the GC overhead can be reduced to 12.4%, 7.2% and 5.2% on average. In the scenario with smallest heap size configuration (30kB heap), in the worst case where GC can bring up as much as 37.5% overhead for PEAT, with hardware acceleration it can be reduced to 15%. This indicates that for memory constrained JVM, it would be a good choice to use hardware acceleration to alleviate the stress from the high overhead GC. For our micro-JVM, we can make further customization by extending extra Mark-Sweep related instructions to improve performance more.

Since energy cost is more sensitive for embedded mobile device, we also investigated how hardware accelerator will impact the energy behavior. We summarized the energy cost after applying the hardware acceleration in Figure 5. Here we use the energy consumption without acceleration as the baseline. It can be seen that hardware acceleration can provide at least 20% total energy reduction. For smallest heap size configuration (30kB heap), its benefit can reach up to 25%. This is because the smaller the heap size is, GC would occur more frequently, and the hardware acceleration can provide more benefits in terms of energy efficiency. It can be further concluded that from the energy perspective, hardware acceleration is more needed to be applied into micro-JVM for its first concern on limited energy budget.

**VII. CONCLUSIONS**

Java technology has been popular in embedded mobile devices where strict energy budget is extremely critical. In this work, we looked into how memory heap size of JVM could impact energy consumption of embedded device, by taking an in-house-developed production-level micro-JVM as a case study. Our experimental results show that garbage collector (GC) overhead is a function of memory heap size, and the bigger heap the smaller overhead incurred. However, this function does not continue in energy part, because memory energy cost sometime offsets the benefit from the less GC invocations. To alleviate this problem, we have found that hardware GC acceleration is an efficient technique, which enables energy efficiency even on devices with small heaps. Our results show that hardware GC acceleration is able to reduce energy consumption on memory-constrained embedded devices by as much as 20%.

**REFERENCES**


