Journal of Circuits, Systems, and Computers
Vol. 18, No. 1 (2009) 103–120

World Scientific
www.worldscientific.com

STUDYING ENERGY-ORIENTED DYNAMIC OPTIMIZATIONS IN JAVA VIRTUAL MACHINES

YU SUN* and WEI ZHANG†

Department of Electrical and Computer Engineering,
Southern Illinois University Carbondale,
Carbondale, IL 62902, USA
**sunyu@engr.siu.edu*
†*zhang@engr.siu.edu*

Revised 7 July 2008

Virtual machines have been increasingly used in embedded and mobile devices. Limited processing power and energy resources become a major challenge for modern virtual machines. The traditional virtual machines running on server computers also face the similar requirement to reduce energy dissipation. To address these challenges, a number of research works have been done in the area of energy-oriented dynamic optimization in Java Virtual Machines (JVM). Some of them focus on virtual machine directed hardware optimizations, while others exploit pure software approaches on virtual machine or compiler to reduce energy consumption. Additionally, client/server framework among multiple processors is also used to offload power-consuming tasks from low-power devices. This paper surveys the current progress of energy-oriented dynamic optimizations in JVMs.

Keywords: Dynamic compiler; energy dissipation; Java Virtual Machine.

1. Introduction

Java technique[46] has been widely used in the past decades, from heavy workload servers to low-power mobile devices. Java Virtual Machine (JVM) is the most basic and crucial component of Java platform, which manages and executes Java programs. There are plenty of implementations of JVM, such as Sun JVM, Jikes RVM,[47] KVM,[48] etc. Besides providing basic compilation/interpretation from Java program to native machine code, modern JVMs also employ dynamic or adaptive optimizations to generate code that is able to achieve higher performance. Furthermore, dynamic optimizations like Refs. 1–4 are becoming the hotspot in JVM-related researches because JVM can collect accurate application characteristics during runtime.

In modern JVMs, performance is not the only important issue. Energy consumption has also become a critical design consideration for systems ranging from battery-operated embedded devices to high-performance servers. Total amount of

energy consumed by an application generally depends on two factors — average power consumption of hardware components per cycle (or time unit generally), which includes CPU core power, memory power and I/O power, and the total execution time of this application. The goal of energy-oriented optimization is to improve one or both of these two factors so that the total energy consumption can be reduced. While there have been many studies on software-based energy reduction, mainly directed by a static compiler, there are relatively fewer works in studying energy optimizations in the context of dynamic compilers or JVMs. Therefore, this paper attempts to survey the related and most recent work in energy-oriented dynamic optimizations in JVMs, which can help researchers to understand the state-of-the-art and to possibly advance this important research area.

The fundamental of energy-oriented optimizations in JVM is to model JVM's energy behavior accurately. Dynamic optimization requires runtime energy consumption measurement and estimation. References 5–8 introduced their work about JVM energy models and runtime energy measurement. With these works, dynamic optimization systems can estimate future energy consumption in different cases and dynamically make the best decision to achieve the least energy usage.

There are many approaches to implementing energy-oriented dynamic optimizations in JVM. Hardware methods are the mainstream methods used to reduce power usage. Some static hardware optimizations like Refs. 9–11 have also been studied. However, by combining these works with JVM technique, more improvement can be achieved. Pure software approaches also have the ability to reduce energy consumption. For example, compiler optimizations like Refs. 12, 13 are introduced to low-power devices. Moreover, distributed computing provide a new way for JVM to make remote/local energy tradeoffs, especially for low-power mobile devices working with a resource-rich server.

The survey is organized as follows: Sec. 2 describes the optimizations on single processor environments, including software directed hardware optimizations and pure software-based energy optimizations; Sec. 3 presents the optimizations based on client/server framework; Sec. 4 makes concluding remarks and discusses possible future work in this area.

2. Optimizations on Single Processor

Firstly, we explore the research works on traditional single processor environment. Both hardware and software approaches are presented in this field. These two kinds of optimizations are described in Secs. 2.1 and 2.2 respectively.

2.1. *Virtual machine directed hardware energy optimizations*

Any energy consumption of an application is directly generated by hardware circuit, although it can also be impacted by the software that runs on top of the hardware. Therefore, energy-efficient hardware optimizations are the most direct way to reduce

energy dissipation. Although JVM itself is software, the runtime information collected by JVM can be quite useful to intelligently guide energy-oriented hardware optimizations.

2.1.1. *Virtual machine driven dynamic voltage scaling*

In CMOS circuits, the dynamic energy consumption is proportional to where v is the supply voltage and f is the clock frequency. Dynamic Voltage Scaling (DVS) is a technique to reduce the energy consumption by scaling down the supply voltage. With a lower supply voltage, although the execution time of an application increases due to the slower frequency, the total energy consumption will decrease because of the quadratic effect of voltage scaling on the energy dissipation. DVS can be directly supported by many processors, such as Intel Speedstep[49] and Transmeta's Crusoe.[50] A number of research efforts on DVS at OS level[14–16] and static compiler level[13,17,18] have been conducted in the literature. By comparison, this paper will review related work in DVS driven by the dynamic compiler or JVM.

Wu *et al.*[19] presented a design framework of the runtime DVFS (Dynamic Voltage and Frequency Scaling) in a general dynamic compilation system. In this work, a new DVFS decision algorithm was proposed, which was based on an analytical DVFS decision model. Their evaluation, based on a real hardware platform and the measurement of CPU current and voltage, indicate that significant energy can be saved with little performance degradation.

Haldar *et al.*[20] described an approach, which used the fine-grained execution information about the actual workloads from virtual machine to make DVS decisions with higher precision. This VM-driven DVS[20] presented an algorithm based on runtime profiling of bytecodes executing in a virtual machine. The goal of this work was to reduce energy consumption by means of scaling down frequency, while at the same time minimizing performance loss. DVS decisions for methods were made by considering two factors. First, the new execution time of a method must stay within a threshold of the original runtime. Moreover, the voltage/frequency switch overhead must be significantly smaller than the method's runtime. The methods which meet these two conditions will be scaled down at next invocation. The on-line profiling and future estimation mechanism were the same as the heuristic in Jikes RVM.[47] Their work led up to a 13% saving in simulated processor power consumption with a performance loss of no more than 33%.[20]

Rauch and Gal[21] made some further research along this direction. They presented a dynamic adaptive power-management plug-in for the JVM that improved the precision of power management decisions by utilizing fine-grained high-level program state information. This virtual machine plug-in injected runtime profiling probes and power management triggers at the Java bytecode-level and was thereby independent of target architecture, JVM implementation, and host operating system. Java Virtual Machine Tool Interface (JVMTI)[51] was exploited to extract profiling information from runtime JVM. Up to 6% overhead was brought by

the profiling activities. Three heuristics were presented in this paper: (1) memory-allocation heuristic, scaling the CPU clock based on the memory throughput of the application so as to achieve the maximal throughput at the lowest possible clock frequency; (2) I/O heuristic, trying to run the program at the lowest clock frequency without compromising the I/O throughput; and (3) CPU heuristic, which reduced the CPU clock frequency only if the throughput was not affected. A power management system combining these three heuristics can achieve an average 19% energy saving with average 21% performance loss during the experiments on SpecJVM98 benchmarks.[52]

2.1.2. *Virtual machine based configurable unit managements*

Hu *et al.*[22,23] presented another software–hardware combined approach to reduce energy consumption of virtual machine-based application. The motivation of their work is that a group of configurable hardware units can be managed to fit the minimal requirement of different applications. Turning off unnecessary units can significantly reduce the energy consumption of execution. Previous research indicated that five components dominated the energy consumption of a microprocessor.[24] These units were issue queue, reorder buffer, L1 instruction and data caches, and L2 cache.

This research work focused on two major issues — when to adapt hardware resources and which configuration to adapt to. The first one was addressed by the program phase detection. It was implemented by Jikes RVM dynamic hot method detector, which can closely represent program phase behavior. The second issue also employed the Jikes RVM. More specifically, Hu *et al.*[22,23] proposed a scheme for efficient management of multiple CUs based on a generic dynamic optimization (DO) system. By exploiting the existing hot method detection mechanism and cost/benefit estimation model of the DO system, the proposed adaptive computing environment (ACE) framework adapted micro-architectural resources at hotspot boundaries. A hotspot tuning procedure was inserted after on-line optimization of Jikes RVM to select the best CU configuration for one hot method.

The experiments based on Dynamic Simple Scalar[53] (DSS), Jikes RVM and SpecJVM98 benchmark suite showed an energy reduction in L2 cache by as much as 52%, and the average energy reductions of all CUs are between 28% and 45%. On the other hand, this technique resulted in an average 5% performance degradation to all the benchmarks.

Furthermore, Hu and John[25] studied the impacts of JVM optimizations and garbage collection on energy consumption with adaptive hardware. Their research revealed that JIT optimizations and garbage collection interferred with hardware adaptation. Both JIT optimizations and garbage collection altered program behavior and runtime requirements. In adaptive micro architectures, such changes of runtime requirements can considerably affect the adaptation decisions of configurable hardware units, and eventually influence the overall energy consumption of

the underlying adaptive micro architecture. This research also studied the adaptation preferences of CUs on the JIT optimizer and the garbage collector. Owing to their distinct runtime characteristics, such as poor data cache performance for the both of them, the two dynamic optimizations had adaptation preferences that differed substantially from the applications. For instance, both the JIT compiler and garbage collector preferred a simple core for energy reduction. On the other hand, the JIT optimizer usually required larger data caches to sustain its performance, while the garbage collector chose smaller caches with minimal performance loss.

2.1.3. *Memory hierarchy energy reduction*

Memory hierarchy system, including cache and main memory, has great impact on the total energy consumption of entire system. Some important research works were presented in this particular field to improve memory energy efficiency by hardware approaches.

One of these studies was done by Chen *et al.*,[26] whose work centered on investigating data cache leakage energy for Java objects in virtual machine. Three techniques were explored to detect and save leakage energy in data cache lines: garbage collection, trace-based escape analysis and last-use analysis. Garbage collection (GC) was invoked when heap memory ran out. It traversed all Java objects and swept the objects that could not be reached from the root object. Turning off cache line containing "garbage objects" saved the data cache energy consumption by 9%. Escape analysis[27,28] identified local objects in methods and turned off corresponding cache lines when the methods returned. The combination of GC and this scheme achieved a 17% data cache energy reduction on average. Last-use analysis inserted "deactivate" instruction into method code to turn off some objects immediately after their last access instead of the return point of method. This more aggressive scheme exploited profiling traces and training sets to locate the position of deactivate instructions. On the average, they achieved 11% data cache energy savings through last-use analysis optimization and GC scheme. When it was combined with escape analysis, it could save 6% more energy on average. Additionally, access gap analysis was introduced to estimate the intervals between two consecutive accesses to objects and turn off cache lines when the intervals were long enough. A combination of GC, escape analysis, and access gap analysis can reduce the data cache energy by 21% on average.

Tomar and Kim[29] presented another approach to improving energy efficiency of Java applications, by applying local memory in on-chip memory instead of using cache-only architecture. Local memory was used in their work to store frequently used Java objects, avoiding them to be replaced from cache due to conflicts. In presence of the LM, the main memory energy decreased by 11.2% on average across different benchmarks. The reason was fewer accesses to the main memory, which was a result of higher number of cache hits and more references being captured in the LM. Local memory also consumed less energy than the cache due to the absence

of tag access. This improvement was made in large size cache and local memory. Small size local memory like 1 KB or 2 KB could not make any profit because small local memory did not contain as many heap references as to cause a drop in the miss rate, and subsequently, energy. Annotation-based implementation was used to decide which Java objects were to be placed into local memory. According to their experiment results, this implementation can achieve 10.37% energy reduction on average with 4 KB cache and 4 KB local memory, compared to 8 KB cache only.

Code cache, which is a small on-chip memory storing dynamically compiled native code of Java methods, was used in Ref. 30 to make improvement in energy aspect. In their system, a strategy that combined recency of use with frequency of use was adopted to decide which methods had the right to put their compiled native code into code cache. With a proper configuration of the code cache management strategy, an average 40% energy reduction across 6 SpecJVM98 benchmarks could be achieved compared to all-compile strategy, and 50% energy reduction compared to all-interpretation approach.

Guha *et al.*[74] presented several techniques to improve code cache performance by reducing exit stub spaces. Both code traces and exit stubs were stored in a code cache. Exit stubs kept track of the branches of a trace and consumed up to 66.7% space of the code cache. They first described two schemes that used a deletion approach to reduce the space occupied by stubs in an application-independent and partially JVM-independent manner. Furthermore, Guha *et al.*[74] proposed two schemes, which identified characteristics of stubs that could be further optimized to minimize their space requirements. The four schemes are Deletion of Stubs, Avoiding Stub Compilation, Exit Stub Size Reduction and Target Address Specific Stubs. These techniques reduced memory consumption of the code cache by 43.5% while improving performance by 1.5%.

On-chip scratch-pad memory (SPM) and pretenuring technique, which segregated Java objects into separated memory regions, were introduced in Ref. 31 to reduce energy consumption and data accessing cycles for Java execution. In their work, Java objects were divided into several categories: (1) young objects, which lived a quite short period of time; (2) hot-immortal objects, which were frequently referenced objects living the whole life time of application; (3) hot-mature objects, where were frequently referenced but with short life time; (4) cool-immortal and (5) cool-mature objects, which were not frequently referenced during application running. The scratch-pad memory bypassed caches and was employed to store young, hot-immortal and hot-mature objects. This policy decreased excessive copies on dead young objects, as well as conflicts of hot-immortal and hot-mature objects in cache. As a result, the energy efficiency of memory hierarchy could be improved. Besides, an effective pretenuring mechanism, which made use of object lifetime and reference density, was presented in their paper. This mechanism can significantly reduce object tenuring costs. However, no experimental results about energy consumption were presented.

Chen *et al.*[32] also used on-chip scratch-pad memory in their approach, where compression technique was combined with SPM and a mechanism that turns off power supply to the unused portions of the memory to control leakage. In this approach, the code of the embedded JVM system and the associated library classes were stored in a compressed form in the memory. Whenever the compressed code or classes were required by the processor core, a mapping structure stored in a reserved part of the SPM was used to locate the required block of data in the compressed store. Then, the block of data after decompression was brought into the SPM. The use of scratch-pad memory in conjunction with a compressed memory store led to the reduction of both dynamic and leakage energy dissipation. KVM, Shade[54] and the CACTI tool Version 2.0[55] were used in their simulation experiments. The results showed an average 20% energy reduction in read-only portion of the main memory and the SPM and 7% in all the main memory and the SPM.

2.2. *Pure software-based energy optimizations*

In addition to the aforementioned hardware-based energy optimizations, there are also some pure software energy optimizations in the literature. A major advantage of pure software-based approach is that no additional hardware modification is needed. Thus the software optimizations can be directly applied on existing systems to gain the benefits of energy reduction.

2.2.1. *Energy-oriented compiler optimizations*

Parikh *et al.*[33] presented and evaluated several instruction scheduling algorithms that reorder a given sequence of instructions, taking into account the energy considerations. Their work compared a performance-oriented scheduling technique with three energy-oriented instruction scheduling algorithms: Top-Down, Bottom-Up and Look-Ahead approaches. They also proposed three scheduling algorithms by considering energy and performance at the same time. Their experiment results showed that the best scheduling from the performance perspective was not necessarily the best scheduling from the energy perspective. Furthermore, scheduling techniques that considered both energy and performance simultaneously were quite successful in reducing energy consumption and their performance was comparable to that of a pure performance-oriented scheduling. Another conclusion from the results was that the energy-oriented scheduling reduced energy consumption by up to 30% compared to the performance-oriented scheduling.

In the work of Kadayif *et al.*,[34] a novel Energy-Aware Compilation (EAC) framework that can estimate and optimize energy consumption of a given code was presented. It provided a low-cost high-level energy estimation model that could be incorporated into an optimizing compiler, and a validation of the compiler-directed energy estimation using a cycle-accurate architectural-level energy simulator for a

simple architectural model. An energy-constrained version of iteration space tiling[35] was also presented in their paper as an example.

Some other software-based energy optimizations were introduced in Refs. 36 and 37. However, these energy-oriented compiler optimizations are not specially designed for Java language or JVM. There is still more research to be performed so that these techniques can be adopted by JVM.

2.2.2. *Optimizations on garbage collector in JVMs*

Garbage collector (GC) is a very important feature of Java technique. GC is performed in almost any Java application dealing with Java objects stored in memory.

Chen *et al.*'s research[38] showed that GC is not only important for limited-memory systems but also for energy-constrained architectures. They presented a GC-controlled leakage energy optimization technique that shut off memory banks that did not hold live data. Two Mark-and-Sweep (M&S) garbage collectors of KVM, which were commonly employed in current embedded JVM environments, were tuned to test the impact of GC's configurations. Assuming that GC worked with a banked memory system in which memory banks could be turned off to save energy, the energy simulation performed by Shade showed that this leakage control policy could save 31% heap energy consumption on the average. The results also showed that more frequent GC was beneficial for energy-constraint systems. With an Active-Bank-First object allocation strategy which tried current active memory banks first, more than 30% heap energy could be saved. Their work also studied the impact of object compaction and cache memory.

A new energy-efficient garbage collector was proposed by Griffin *et al.*[39] It was a hybrid scheme falling between the standard mark-sweep-compact collector available in Sun's KVM and a limited-field reference counter. This scheme could reclaim memory space, resulting in less garbage collection invocations. Lower energy consumption could be achieved by this GC scheme because memory access accounted for a large amount of power dissipation. The experimental results showed that the proposed scheme could reduce the energy consumption in two out of six applications. For the remaining four applications, however, there were almost no differences between the default and the proposed algorithms.

Velasco *et al.*[40] performed a complete study from an energy viewpoint of the different state-of-the-art garbage collectors mechanisms (e.g., mark-and-sweep, generational garbage collectors) for embedded systems. Their work was based on Jikes RVM and DSS environment. Five GC policies were explored: Mark-and-Sweep, SemiSpace, Generational Copy (GenCopy), Generational Mark-and-Sweep (GenMS) and Copying collect with Mark-and-Sweep. Their results showed that all GCs based on Generational collectors (i.e., GenMS and GenCopy) achieved the best energy results compared to more typical GCs implemented in the JVM of real-life Java-based embedded devices.

2.3. *Embedded virtual machines and compilers*

Besides the abovementioned approaches applied, this section will introduce energy-oriented optimizations specifically targeting JVM-enabled embedded devices. These systems typically share some common features such as lightweight, low memory usage and energy-efficiency.

E-Bunny selective dynamic compiler[56,57] was applied on J2ME/CLDC[58] virtual machine KVM, which targeted low-end mobile devices. This compiler selected methods to be compiled based on their invocation frequency. The low frequency methods stayed at interpreted mode. It reduced the energy dissipation based on two factors. First, it had a small memory footprint as low as 138 KB. Second, it implemented an efficient one-pass stack-based machine code generation. Also, E-Bunny accomplished a speedup of 400% with respect to the original version of Sun's KVM.

Scylla[59] is a specially designed Java virtual machine for embedded devices with limited energy resources. Energy estimation was performed whenever a method was to be compiled in Scylla virtual machine. The core energy estimation was based on a look-up table whose entries corresponded to energy estimates obtained for the native architecture via instruction level power analysis similar to that in Ref. 60. Another part of energy estimation came from communication energy calculation based on the payload size of transmission. The virtual machine would terminate applications whose estimated energy cost exceeded some limit, and attempt to execute the application's fault handler. Scylla can monitor and manage the energy consumption of Java methods during runtime. Besides, Scylla's instruction set was carefully designed to get close to popular processors used in embedded devices. Consequently, the compilation complexity was quite small and the compilation energy consumption was reduced.

Koshy and Pandey[61] introduced VM* — a software framework for synthesizing runtime environments for wireless sensor networks (WSN). The approach was VM-based; programmers wrote applications over a common abstract interface. Device-specific features were accessed through a lightweight native interface. Porting the VM across various node architectures allowed applications to be deployed uniformly in heterogeneous environments. VM* was based on the key insight that the VM running on a specific device did not need to reflect the full VM specification. It only needed to provide services that were needed by the application running on the device. Furthermore, different services could be implemented differently on devices based on resource availability. VM contained a general description of a VM, which was instantiated and specialized for each application and device. It also implemented a continuous updated model, in which WSN nodes could be updated incrementally when changes occurred in applications and the VM. By tracking program changes at the VM abstraction level, the cost of distributing and applying application updates was significantly reduced.

Mate[62] was a stack-oriented VM implemented using TinyOS.[63] It was built on top of several system components providing access to sensors, transceivers, and external storage. Mate instructions hid the asynchronous nature of the TinyOS event model to provide a simpler synchronous programming interface. Also, it implemented a fixed thread-pool of contexts that could react to hardware events and commands from the application. Each context had its own operand stack for passing data between operations. Contexts could share variables, and ran concurrently at instruction granularity. Each instruction was executed as a TinyOS task. The instruction set was designed for compactness. Users could also define custom instructions for a domain specific instruction set. A key goal of Mate was the reprogramming capability. VM applications were injected as code capsules into deployments of nodes programmed with Mate. A viral code distribution scheme infected nodes in the network with the application capsules. More recent work on Mate generalized the framework for building application-specific VMs.

Zhang and Krintz[64] studied the design, implementation, and empirical evaluation of a compiled-code management system that could be integrated into any compilation-based JVM. The system unloaded compiled code to reduce the memory usage of the VM. It did so by dynamically identifying and unloading dead or infrequently used code. If the code was later reused, it was then recompiled by the system. Therefore, the system adaptively traded off memory footprint and its associated memory management costs, with recompilation overhead. Their empirical evaluation showed that the code management system significantly reduced the memory requirements of a compile-only JVM, while maintaining the performance benefits enabled by compilation. Moreover, Zhang and Krintz investigated a number of implementation alternatives that used dynamic program behavior and system resource availability to determine when to unload and what code to unload. Their system reduced code size by 36–62%, on average, which translated into significant reduction of the execution time and energy dissipation.

3. Client/Server Framework Optimizations

As more and more Java applications are now running on low-power mobile devices with wireless network connection, it is a good idea to move part of compilation or execution tasks from energy constraint clients to a connected resource-rich server. The transmission of source code, native code, execution parameters and return values brings some overhead. However, an energy reduction can be made by using proper offloading policy that controls which tasks should be performed remotely.

3.1. *Remote compilation*

Palm and Lee[41] described a remote compilation/optimization service based on Jikes RVM. A tethered compilation/optimization server provided its service via wireless network to a number of handheld computers with different hardware and operating systems. Using the SpecJVM98 benchmark suite, they evaluated the energy

consumption of four configurations: local baseline, local optimized, server baseline and server optimized. The mechanism of server compilation/optimization was simply to move these tasks to the server and client devices staying idle before it could download the compiled/optimized machine code from the server. The experiment results showed that downloading both Java bytecode and machine code did not have much impact on the total energy consumption, while the idle energy spent on waiting server to do optimizations did. Smart compilation/optimization framework could significantly improve the energy efficiency of Java programs running on mobile devices. However, good selective optimization mechanism and precompilation should be used to achieve the best improvement.

JCod,[65] the Java compiled-on-demand, was proposed to avoid the memory overheads of a JIT compiler on resource-limited devices. Whenever JCod determined that a method should be compiled, it sent this method to a compilation server on the local network. The compilation server replied by sending the native code back to JCod, which installed it within the VM. From that time on, the native code was used, resulting in performance and energy improvement.

Teodorescu and Pandey[66] investigated JUCE (Java for Ubiquitous Computing Environments). The JUCE restructured the traditional JVM by introducing two new concepts: Remote Just-In-Time compiler (R-JIT) and Configurable Runtime System (C-RTS). R-JIT exploited the fact that much of the overhead associated with Java bytecode processing (for instance, code verification or just-in-time compiling) could be relocated to a remote host from the device executing the code. Thus, the code targeting an embedded device was compiled just-in-time on a remote host and migrated to the device in the native code format. This meant that a resource-limited device could efficiently execute a Java application while a more powerful node used to tolerate the actual Java-specific overhead. The C-RTS had a modular structure that allowed JUCE to adapt its configuration according to the resources availability and application requirements. Thus, in JUCE, the different RTS modules were loaded as required by the application. In the traditional approach the Java VM was entirely loaded before starting the execution of the application. In this way, a device running a Java application only needed to deal with the overhead produced by the RTS services currently required by the application.

Newsome and Watson[67] explored MoJo, a new native compiler which allowed embedded clients to connect to servers and delegate compilation of Java class packages to native code libraries. MoJo implemented a custom-designed, client/server protocol for proxy compilation sessions. It assumed TCP packet transport and consequently did not concern itself with packet loss handling. The protocol allowed clients to specify target code requirements to servers and to receive native binaries back from the server. The MoJo proxy compilation scheme gave a 94% speed increase over the fastest surveyed interpreter system and a 20% speed increase over the fastest surveyed JIT system. The MoJo-generated binaries for the application also proved to be 45 times smaller than those required by its nearest iPAQ JRE competitor and 275 times smaller than the Sun JRE v1.3.1 for iPAQ.

3.2. *Remote execution*

Tallam and Gupta[42] presented a fast profile-guided partitioning algorithm that could choose the program parts to be executed remotely so as to save energy on the embedded device. The partitioning was performed at Java object level. Profiled information included: (1) the number of objects of each java class created; (2) the amount of communication, in bytes, between each object pair; and (3) the processor time spent in executing in the context of every object. Energy estimation based on the profile was modeled in the form of a graph. Then in the partitioning step, either a MIN-CUT algorithm[43] or a "fast greedy" approach would be selected. According to the experiment results, the fast algorithm could achieve energy savings from 29% to 43%, while the complicated optimal algorithm reduced the energy consumption by 44% on average.

Based on their VM* virtual machine, Wirjawan *et al.*[68] described a remote Just-In-Time (JIT) compilation service that was effective in combining interpretation with native execution to arrive at an efficient hybrid execution configuration. The VM* JIT compilation framework implemented a distributed infrastructure for identifying a small set of performance critical Java methods. Nodes collected profiling data about performance-critical methods, and sent the data over the radio to the base station. The VM* JIT compilation framework used Soot[69] for compiling performance critical methods. Then it used a remote linker to integrate generated binary on sensor nodes. Their test applications yielded speedups of up to 77% in application configurations suitable for JIT compilation, with program footprints increased by a factor of 5.58 for Mica2[70] and 3.74 for Telos.[71] Overall stack requirements, however, depended on application call patterns. The experimental results demonstrated that through selective compilation of performance-critical methods, the efficiency of VM-based execution environments can be significantly improved, making them viable as a basis for software development, deployment and execution in WSNs.

Based on profiling information on computation time and data sharing at the level of procedure calls, Li *et al.*[72] studied a scheme to offload computation from energy-constraint devices. In their scheme, offloading units were defined at the level of procedure calls. Each unique call site to a function (or a procedure) was identified as a task. They first collected profiling information on computation time and data sharing at the level of procedure calls. Based on that, a cost graph was constructed for the given program so that they could apply a task mapping algorithm to statically divide the program into server tasks and client tasks to minimize energy dissipation. Experiments were performed on a suite of multimedia benchmarks, which indicated considerable energy saving for several programs through offloading.

3.3. *Combination of remote compilation and execution*

Chen *et al.*[44] proposed both adaptive compilation and execution strategies on a client/server model. When a method was to be compiled, the client computed its

energy cost to offload the compilation to server and compared it to the version of local compilation to select the better one. Similar procedure was performed to select the best execution strategy. The energy cost was computed based on hardware constants, method size, transmitted data size and current wireless network condition. A series of proxies that could be added by the programmer or automatically created using profile data at server, were designed to do this job. The proxies evaluated and compared local compilation and offloading cost when an invocation occurred. The proxies were also in charge of transmitting method name and parameters when remote execution was performed, as well as receiving the results back. Additionally, reconfigurable data path[45] and data compression at transmission were also used to further reduce the energy consumption.

The Proxy Virtual Machine[73] was introduced to reduce the energy consumption of mobile devices by combining application partitioning with dynamic optimization. The framework positioned a powerful server infrastructure, the proxy, between mobile devices and the Internet. The proxy included a just-in-time compiler and bytecode translator. A high bandwidth, low-latency secure wireless connection mediated communication between the proxy and mobile devices in the vicinity. Users can request Internet applications as they normally would through a Web browser. The proxy intercepted these requests and reissues them to a remote Internet server. The server sent the proxy the application in mobile code format. The proxy verified the application, compiled it to native code, and sent part of the generated code to the target. The remainder of the code executed on the proxy itself to significantly reduce energy consumption on the mobile device.

4. Concluding Remarks

We have reviewed the current state and progress of energy-efficient compiler techniques in Java Virtual Machines. This review is expected to help researchers understand the state-of-the-art and conduct further studies to enhance the energy efficiency for Java programs, which is especially important for battery-operated embedded Java devices.

We have generally classified the energy-oriented dynamic optimizations into hardware-based, software-based and hardware/software hybrid approaches. We find that hardware-based approaches are clearly the most direct way to reduce energy consumption, while software-based approaches can save energy directly for existing systems without modifying hardware. The software–hardware-combined techniques, however, seem to be able to achieve the largest energy reduction. There are relative few research efforts in the direction of hardware–software cooperative energy optimizations, which, we think should be emphasized. For example, the DVS on virtual machine introduced above was based on an interpreter. It is possible to adopt the work to a JIT dynamic compilation-based virtual machine such as Jikes RVM. The preciseness of configurable unit management can also be improved by applying better dynamic profiling mechanisms available in JVM.

Also, there have been plenty of studies on energy-oriented compiler optimizations for statically-compiled programs; however, few of them have been currently applied or extended to modern JVMs. Therefore, in our future work, we would like to evaluate the feasibility and potential of extending those optimizations to an open-source Java virtual machine such as Jikes RVM to significantly improve the energy efficiency of dynamic compilation and applications running on Java Virtual Machines. Moreover, we intend to study how to exploit the runtime information that is available to the dynamic optimizers to further enhance the energy efficiency of Java applications.

Acknowledgments

This work was funded in part by the NSF grant CNS 0613633. We would like to thank the anonymous referees for the detailed comments that helped us improve the paper.

References

1. M. G. Burke, J. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan and J. Whaley, The Jalapeño dynamic optimizing compiler for Java, *Proc. ACM'99: Conf. Java Grande*, June 1999.
2. M. Arnold, M. Hind and B. G. Ryder, Online feedback-directed optimization of Java, *SIGPLAN Not.* **37** (November 2002).
3. T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu and T. Nakatani, A dynamic optimization framework for a Java just-in-time compiler, *SIGPLAN Not.* **36** (November 2001).
4. M. Arnold, S. J. Fink, D. Grove, M. Hind and P. F. Sweeney, A survey of adaptive optimization in virtual machines, *Proc. IEEE* (2005).
5. N. Vijaykrishnan, M. Kandemir, S. Kim, S. Tomar, A. Sivasubramaniam and M. J. Irwin, Energy behavior of Java applications from the memory perspective, 1st *USENIX Java Virtual Machine Research and Technology Symp.*, April 2001.
6. S. Lafond and J. Lilius, An energy consumption model for an embedded Java virtual machine, *Int. Conf. Architecture Computing Systems* (2006).
7. K. I. Farkas, J. Flinn, G. Back, D. Grunwald and J. M. Anderson, Quantifying the energy consumption of a pocket computer and a Java virtual machine, *Proc. 2000 ACM SIGMETRICS Int. Conf. Measurement and Modeling of Computer Systems*, June 2000.
8. C. Seo, S. Malek and N. Medvidovic, An energy consumption framework for distributed Java-based software systems, *ACM SIGSOFT* (2006).
9. J. J. Sharkey, D. V. Ponomarev, K. Ghose and O. Ergin, Instruction packing: Reducing power and delay of the dynamic scheduling logic, *Proc. 2005 Int. Symp. Low Power Electron. Des.* (*ISLPED*), August 2005.
10. M. Valluri, L. John and H. Hanson, Exploiting compiler-generated schedules for energy savings in high-performance processors, *Proc. 2003 Int. Symp. Low Power Electronics and Design* (2003).
11. M. Valluri and L. John, Hybrid-scheduling: A compile-time approach for energy-efficient superscalar processors, *IBM Austin Conference on Energy-Efficient Design*, March 2004.

12. N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim and W. Ye, Energy-driven integrated hardware-software optimizations using SimplePower, *Proc. 27th Int. Symp. Computer Architecture* (2000).
13. C. H. Hsu and U. Kremer, The design, implementation and evaluation of a compiler algorithm for CPU energy reduction, *Proc. ACM SIGPLAN Conf. Programming Languages, Design, and Implementation* (2003).
14. A. Dudani, F. Mueller and Y. Zhu, Energy-conserving feedback EDF scheduling for embedded systems with real-time constraints, *Proc. Conf. Languages, Compilers and Tools for Embedded Systems* (2002).
15. Y. Zhu and F. Mueller, Preemption handling and scalability of feedback DVS-EDF, *Proc. Workshop Compilers and Operating Systems for Low Power* (2002).
16. F. Gruian, Hard real-time scheduling for low-energy using stochastic data and DVS processors, *Proc. 2001 Int. Symp. Low Power Electronics and Design* (2001).
17. D. Shin, J. Kim and S. Lee, Low-energy intra-task voltage scheduling using static timing analysis, *Proc. 38th Conf. Design Automation* (2001).
18. S. Lee and T. Sakurai, Run-time voltage hopping for low-power real-time systems, *Proc. 37th Conf. Design Automation* (2000).
19. Q. Wu, M. Martonosi, D. W. Clark, V. J. Reddi, D. Connors, Y. Wu, J. Lee and D. Brooks, A dynamic compilation framework for controlling microprocessor energy and performance, *Proc. 38th IEEE/ACM Int. Symp. Microarchitecture* (2005).
20. V. Haldar, C. W. Probst, V. Venkatachalam and M. Franz, Virtual-machine driven dynamic voltage scaling, Technical Report CS–03–21, University of California, Irvine, CA, October 2003.
21. M. Rauch, A. Gal and M. Franz, Dynamic adaptive power management for and by a Java Virtual Machine, Technical Report No. 06–19, Donald Bren School of Information and Computer Science, University of California, Irvine, November, 2006.
22. S. Hu, M. Valluri and L. K. John, Effective adaptive computing environment management via dynamic optimization, *Code Generation and Optimization* (2005).
23. S. Hu, M. Valluri and L. K. John, Effective management of multiple configurable units using dynamic optimization, *ACM Trans. Architecture and Code Optimization* **3** (December 2006).
24. D. Folegnani and A. Gonzalez, Energy-effective issue logic, *Proc. 28th Int. Symp. Computer Architecture* (2001).
25. S. Hu and L. K. John, Impact of virtual execution environments on processor energy consumption and hardware adaptation, *Proc. 2nd Int. Conf. Virtual Execution Environments*, June 2006.
26. G. Chen, N. Vijaykrishnan, M. Kandemir, M. J. Irwin and M. Wolczko, Tracking object life cycle for leakage energy optimization, *1st IEEE/ACM/IFIP Int. Conf. Hardware/Software Codesign and System Synthesis* (2003).
27. J. Whaley and M. Rinard, Compositional pointer and escape analysis for Java programs, *ACM SIGPLAN Conf. Object-Oriented Programming Systems, Languages, and Applications*, November 1999.
28. J. D. Choi, M. Gupta, M. J. Serrano, V. C. Sreedhar and S. P. Midkiff, Escape analysis for java, *Conf. Object-Oriented Programming Systems, Languages and Applications* (1999).
29. S. Tomar, S. Kim, N. Vijaykrishnan, M. Kandemir and M. J. Irwin, Use of local memory for efficient Java execution, *Proc. 2001 Int. Conf. Computer Design* (2001).
30. G. Chen, M. Kandemir, N. Vijaykrishnan and M. J. Irwin, Energy-aware code cache management for memory-constrained Java devices, *Proc. IEEE Int. SOC Conf.* (2003).

31. K. F. Chong, C. Y. Ho and A. S. Fong, Pretenuring in Java by object lifetime and reference density using scratch-pad memory, *15th EUROMICRO Int. Conf. Parallel, Distributed and Network-Based Processing* (2007).
32. G. Chen, M. Kandemir, N. Vijaykrishnan, M. J. Irwin and W. Wolf, Energy savings through compression in embedded java environments, *Proc. 10th Int. Symp. Hardware/Software Codesign* (2002).
33. A. Parikh, S. Kim, M. Kandemir, N. Vijaykrishnan and M. J. Irwin, Instruction scheduling for low power, *J. VLSI Signal Process. Syst.* **37** (May 2004).
34. I. Kadayif, M. Kandemir, G. Chen, N. Vijaykrishnan, M. J. Irwin and A. Sivasubramaniam, Compiler-directed high-level energy estimation and optimization, *Trans. Embedded Comput. Syst.* **4** (November 2005).
35. M. D. M. Wolf and D. Chen, Combining loop transformations considering caches and scheduling, *Proc. Int. Symp. Microarchitecture* (1996).
36. V. Tang, J. Siu, A. Vasilevskiy and M. Mitran, A framework for reducing instruction scheduling overhead in dynamic compilers, *Proc. 2006 Conf. of the Center for Advanced Studies on Collaborative Research* (2006).
37. J. Cavazosand and J. Moss, Inducing heuristics to decide whether to schedule, *Proc. Conf. Programming Language Design and Implementation*, June 2004.
38. G. Chen, R. Shetty, M. Kandemir, N. Vijaykrishnan, M. J. Irwin and M. Wolczko, Tuning garbage collection in an embedded Java environment, *Proc. 8th Int. Symp. High-Performance Computer Architecture*, February 2002.
39. P. Griffin, W. Srisa-an and J. M. Chang, An energy efficient garbage collector for Java embedded devices, *Proc. 2005 ACM SIGPLAN/SIGBED Conf. Languages, Compilers, and Tools for Embedded Systems*, June 2005.
40. M. Velasco, D. Atienza, L. Pinuel and F. Catthoor, Energy-aware modelling of garbage collectors for new dynamic embedded systems, *Proc. 1st Int. Workshop Power-Aware Real-Time Computing*, September 2004.
41. J. Palm, H. Lee, A. Diwan and J. Moss, When to use a compilation service? *Proc. Joint Conf. Languages, Compilers and Tools for Embedded Systems* (2002).
42. S. Tallam and R. Gupta, Profile-guided Java program partitioning for power-aware computing, *Proc. 18th Int. Parallel and Distributed Processing Symp.*, April 2004.
43. Z. Li, C. Wang and R. Xu, Task allocation for distributed multimedia processing on wirelessly networked handheld devices, *Int. Parallel and Distributed Processing Symp.*, April 2002.
44. G. Chen, B. T. Kang, M. Kandemir, N. Vijaykrishnan, M. J. Irwin and R. Chandramouli, Studying energy trade offs in offloading computation/compilation in Java-enabled mobile devices, *IEEE Trans. Parallel and Distributed Syst.* **15** (2004) 705–809.
45. Z. Huang and S. Malik, Exploiting operation level parallelism through dynamically reconfigurable datapaths, *Proc. 39th Design Automation Conf.* (2002).
46. Sun Microsystems, Java Technology, http://java.sun.com/
47. Jikes RVM, http://jikesrvm.org/
48. Sun Microsystems, KVM — Kilobyte Virtual Machine White Paper, http://java.sun.com/products/cldc/wp/
49. Mobile Intel Pentium III Processors, http://www.intel.com/support/processors/mobile/pentiumiii/ss.htm
50. Transmeta Corporation, LongRun power management: Dynamic power management for Crusoe processors, White Paper (2001).
51. JVM Tool Interface, http://java.sun.com/j2se/1.5.0/docs/guide/jvmti.2006
52. Standard Performance Evaluation Corporation, SPEC JVM98 Benchmarks, http://www.spec.org/osg/jvm98/

53. X. Huang, J. Moss and K. McKinley, Dynamic SimpleScalar: Simulating Java virtual machines, *1st Workshop on Managed Run Time Environment Workloads*, March 2003.
54. B. Cmelik and D. Keppel, Shade: A fast instruction-set simulator for execution profiling, *Proc. ACM SIGMETRICS Conf. Measurement and Modeling of Computer Systems*, May 1994.
55. G. Reinman and N. Jouppi, *An Integrated Cache Timing and Power Model* (COMPAQ Wester Research Lab, 1999).
56. M. Debbabi, A. Gherbi, L. Ketari, C. Talhi and N. Tawbi, E-Bunny: A dynamic compiler for embedded Java virtual machines, *J. Object Technol.* **4** (January–February 2005).
57. M. Debbabi, A. Mourad and N. Tawbi, Armed E-Bunny: A selective dynamic compiler for embedded Java virtual machine targeting ARM processors, *Proc. 2005 ACM Symp. Applied Computing* (2005).
58. Connected, Limited Device Configuration, Specification Version 1.0, Java 2 Platform Micro Edition, White Paper.
59. P. Stanley-Marbell and L. Iftode, Scylla: A smart virtual machine for mobile embedded systems, *Proc. 3rd IEEE Workshop Mobile Computing Systems and Applications* (2000).
60. V. Tiwari, S. Malik and A. Wolfe, Power analysis of embedded software: A first step towards software power estimation, *IEEE/ACM Int. Conf. Computer-Aided Design*, August 1994, pp. 384–390.
61. J. Koshy and R. Pandey, VM*: Synthesizing scalable runtime environments for sensor networks, *Proc. 3rd Int. Conf. Embedded Networked Sensor Systems*, November 2005.
62. P. Levis and D. Culler, Mate: A tiny virtual machine for sensor networks, *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems* (2002).
63. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler and K. Pister, System architecture directions for networked sensors, *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Cambridge (2000), pp. 93–104.
64. L. Zhang and C. Krintz, The design, implementation, and evaluation of adaptive code unloading for resource-constrained devices, *ACM Trans. Archit. Code Optim.*, June 2005.
65. B. Delsart, V. Joloboff and E. Paire, JCOD: A lightweight modular compilation technology for embedded Java, *Proc. 2nd Int. Conf. Embedded Software*, October 2002.
66. R. Teodorescu and R. Pandey, Using JIT compilation and configurable runtime systems for deployment of Java programs on ubiquitous devices, *Proc. Ubiquitous Computing 2001*, September 2001.
67. M. Newsome and D. Watson, Proxy compilation of dynamically loaded Java classes with MoJo, *Proc. Joint Conf. Languages, Compilers and Tools for Embedded Systems: Software and Compilers for Embedded Systems*, June 2002.
68. I. Wirjawan, J. Koshy, R. Pandey and Y. Ramin, Balancing computation and code distribution costs: The case for hybrid execution in sensor networks, *Tech. Rep. TR-CSE-2006-35*, University of California, Davis, 2006.
69. R. Vallee-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon and P. C. Soot, A Java optimization framework, *Proc. Centers for Advanced Studies Conference* (1999).
70. J. Hill and D. Culler, Mica: A wireless platform for deeply embedded networks, *IEEE Micro* **22** (2002) 12–24.
71. J. Polastre, R. Szewczyk and D. Culler, Telos: Enabling ultra-low power wireless research, *Proc. 4th Int. Conf. Information Processing in Sensor Networks: Special Track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)* (2005).

72. Z. Li, C. Wang and R. Xu, Computation offloading to save energy on handheld devices: A partition scheme, *Proc. 2001 Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, November 2001.
73. V. Venkatachalam, L. Wang, A. Gal, C. Probst and M. Franz, ProxyVM: A network-based compilation infrastructure for resource-constrained devices, Technical Report 03–13, University of California, Irvine, March 2003.
74. A. Guha, K. Hazelwood and M. L. Soffa, Reducing exit stub memory consumption in code caches, *Proc. High Performance Embedded Architectures and Compilers* (2007).