Accelerating Embedded Java for Mobile Devices

Mourad Debbabi, Azzam Mourad, Chamseddine Talhi, and Hamdi Yahyaoui, Concordia University

ABSTRACT
With the proliferation of wireless devices, networks, and systems, the deployment of efficient embedded Java virtual machines is becoming a challenging and important research area. Accordingly, a plethora of acceleration techniques have been proposed. In this article we present a new acceleration technology that we developed for embedded Java virtual machines. Acceleration is achieved by the integration of a new selective dynamic compiler, which we called Armed E-Bunny, into the J2ME/CLDC (Java 2 Micro-Edition for Connected Limited Device Configuration) Kilobyte Virtual Machine (KVM). The modified KVM is ported on a handheld PDA that is powered with Embedded Linux. Experimental results demonstrate that we accomplished an important speedup (more than 360 percent) with respect to Sun’s latest version of KVM. This experimentation was carried out using standard J2ME benchmarks.

MOTIVATIONS AND BACKGROUND
With the advent and rising popularity of wireless systems, there is a proliferation of small Internet-enabled embedded devices (e.g., PDAs, cell phones, pagers, etc.). In this context, Java is emerging as a standard execution environment due to its security, portability, mobility, and network support features. In particular, J2ME/CLDC (Java 2 Micro-Edition for Connected Limited Device Configuration) is now recognized as the de facto execution engine in the domain of mobile wireless devices such as pagers, handhelds, TV set-top boxes, appliances, etc. J2ME/CLDC has gained significant momentum and is now standardized by the Java Community Process (JCP) and adopted by many standardization bodies such as 3GPP and OMA. Another factor that has amplified the wide industrial adoption of J2ME/CLDC is the broad range of Java-based solutions that are available in the market. All these factors make J2ME/CLDC an ideal solution for software development in the arena of embedded devices.

The heart of J2ME/CLDC technology is Sun’s Kilobyte Virtual Machine (KVM) [2]. The KVM is a Java virtual machine initially designed with the constraints of low-end mobile devices in mind. These limited-configuration devices do not have the same hardware capabilities as desktop and server systems. The latter have fast busses, hundred of megabytes of RAM, and fast microprocessors operating at over 2 GHz with on-chip 512 kb caches. Alternatively, limited configuration devices typically have memory less than 512 kb to handle the entire Java runtime environment, 300 kb of RAM, 1 Mb of flash and ROM, microprocessors operating at over 32 MHz and low battery capacity. These power and memory restrictions create challenging issues for the deployment of Java technology on limited-configuration devices. More precisely, the performance and security of the KVM are two key factors in the successful deployment of this technology. Lately, a surge of interest has been expressed in the acceleration of Java virtual machines for limited-configuration devices. Two main approaches have been explored: hardware and software acceleration.

For hardware acceleration, several companies (Zucotto Wireless [3], Nazomi [4], etc.) have proposed Java processors that execute Java bytecodes in silicon. Although these hardware accelerators achieve a significant speedup, their use comes with a high price in terms of power consumption. This energy issue is a serious drawback, especially in the case of limited-configuration devices. Moreover, the cost (royalties, licensing, etc.) of these hardware acceleration technologies is an additional obstacle to their adoption by the industry. These drawbacks created an interesting but challenging niche for software acceleration of embedded Java virtual machines.

For software acceleration, many techniques have been advanced [5–7]. They could be structured into three categories: general optimizations, ahead-of-time optimizations, and dynamic compilation. General optimizations consist of designing and implementing more efficient virtual machine components (better garbage collector, fast threading system, accelerated lookups, etc.). Ahead-of-time optimizations consist of using static analysis (flow analysis, annotated type systems, and abstract interpretation) to optimize programs before execution. Dynamic compilation consists of compiling, at runtime, fragments of Java executables. This dynamic
compilation is achieved by a compiler that is
generally embedded into the virtual machine.
The compiler is in charge of translating byte-
codes into the native code of the host platform.
Experience has demonstrated that general
and ahead-of-time optimizations can lead to reason-
able accelerations. However, they cannot com-
pete with dynamic compilation in reaching large
speedups (e.g., an acceleration of more than 200 percent) [8]. This makes dynamic compilation a
more appealing acceleration technique. In what
follows, we describe the principles underlying
the different forms of dynamic compilation, together
with their advantages and disadvantages.

The rest of the article is organized as follows.
A section is dedicated to dynamic compilation,
while another presents embedded Java virtual
machines. Finally, concluding remarks are given.

DYNAMIC COMPILATION
TECHNIQUES

Different forms of dynamic compilation could be
distinguished according to the following three questions: What to compile? When to compile? How to compile?
The first question deals with the unit of the
compilation process (a program, a class, a
method, a code fragment, etc.). The second ques-
tion addresses the issue of when the compilation
process is triggered (when a class is loaded, when
a frequently called fragment is detected, when
a threshold is exceeded, etc.). The third question
deals with the inner workings and nature of the
compilation process (one-pass, multi-passes, opti-
mizing compilation, extensive static analysis, regis-
ter allocation, etc.). Generally, three types of
dynamic compilation are distinguished: Just-In-
Time (JIT) dynamic compilation, Dynamic Adap-
tive Compilation (DAC), and Selective Dynamic
Compilation (SDC). We now discuss details of
each of these approaches.

JUST-IN-TIME COMPILATION

Just-In-Time (JIT) compilation is the first pro-
posed approach to dynamic compilation. JIT con-
­sists of compiling a method when it is invoked for
the first time and saving the generated machine
code for future calls. By doing so, the virtual
machine interpreter is no longer used since all the
bytecodes are compiled to native code and exe-
cuted (natively) on the host machine. This modus
operandi is appealing from the performance
standpoint since interpretation is replaced by
native execution. However, it presents some
drawbacks that become major in the context of
limited-configuration devices. First, a JIT compil-
er translates all the invoked methods, even those
that are called infrequently. This systematic trans-
lcation is costly in terms of time. Second, the mem-
ory requirements of JITs are very high. They are
generally fully-fledged compilers with significant
footprints. In addition, they demand a large mem-
ory space to store the generated native code.
Third, the quality of the generated code is the
same for all the methods regardless of their invo-
cation frequencies. All these disadvantages make
the JIT approach inappropriate for limited-con-
figuration and low-end devices.

DYNAMIC ADAPTIVE COMPILATION

Dynamic Adaptive Compilation (DAC) has been
proposed as an improvement of the JIT
approach. The essence of DAC is to produce
different qualities of the generated code,
depending on the invocation frequency of each
method. It consists of using different compilers
that produce different levels of code quality. A
fast compiler that produces low quality code is
used for infrequently called methods, while a
slow optimizing compiler that produces a high
quality (optimized) code is used to translate fre-
quently called methods. Generally, all the meth-
ods are first compiled with the fast compiler.
Whenever a method is identified as a hotspot (a
frequently called method) it is then recompiled
by an optimizing compiler that generates high
quality code. Spending more time to generate
high quality code for frequently called methods,
can enhance the execution performance consid-
erably. Once the machine code of a method is
generated by the slow compiler, the code already
generated by the fast compiler is totally discard-
ed and freed from the cache memory. Again,
this technique is not suitable for embedded Java
virtual machines because of its high memory
requirements (the footprint resulting from host-

ing many compilers within the virtual machine).

SELECTIVE DYNAMIC COMPILATION

Selective Dynamic Compilation (SDC) deviates
from the other techniques by selecting and
compiling, on the fly, only those fragments of
the class files that are frequently executed
(hotspots). For instance, a method whose num-
ber of invocations exceeds a certain threshold
will be declared as a hotspot. This is generally
done by a profiler. The hotspot method is then
compiled to native code, which is going to be
executed upon future invocations of the method
in question. By doing so, significant accelera-
tion of the virtual machine could be reached
since efficient optimizations are concentrated
on performance-critical fragments of the pro-
gram. Another major advantage of this approach
is the reduction in memory overhead since only a part of a program is converted to
native code. This makes selective dynamic com-
putation more adequate for limited configura-
tion devices. However, special care should be
given to obtaining a lightweight implementation
(low footprint) of the SDC components (profiler,
compiler, etc.).

DYNAMIC COMPILATION IN
EMBEDDED VIRTUAL MACHINES

In the setting of embedded systems, dynamic
compilation should cope with two major chal-
genese: memory and power consumption. The
stringent limitation of memory resources in lim-
ited configuration devices makes heavy weight
code optimizations unaffordable. In addition,
such optimizations are generally achieved by
slow compilers that are costly in power con-
sumption. Besides, the compiler should have a
small footprint in order to fit in the memory
budget of limited-configuration devices.
Despite these difficulties, dynamic compilation is being extensively used in several CLDC-based embedded virtual machines [8–10]. Apart from one article about KJIT [14], no detailed information on these systems is available in the literature. In the following sections we present the best known embedded Java virtual machines that are endowed with dynamic compilers.

**CLDC HOTSPOT**

CLDC Hotspot [8] is an embedded Java virtual machine introduced by Sun Microsystems. As the name indicates, it is strongly inspired by the standard Java Hotspot VM [7]. All the features of Java Hotspot VM that there is no technical/academic publication.

**JBED MICRO EDITION CLDC**

Jbed is a Java virtual machine for limited-configuration devices [9]. Jbed is proposed by the company Esmartec and is intended for Intel Xscale and Strong ARM architectures. The dynamic compiler of Jbed is a small one-pass compiler. It compiles all the classes at the loading time instead of waiting for their first execution. Then, it links the compiled classes into the application. Consequently, the execution delay is reduced. Experimental results show that Jbed virtual machine is four times faster than KVM.

**KJIT**

KJIT is a lightweight dynamic compiler that has been integrated into KVM [10]. KJIT does not use any form of profiling for the simple reason that all the loaded methods are compiled. This strategy seems to be heavyweight and only feasible in server or desktop systems. The key idea to make this strategy adequate for embedded Java virtual machines is to compile only a subset of method bytecodes while the remaining bytecodes are handled by the interpreter. This requires a switching mechanism between the compiler and the interpreter. In fact, whenever one of the interpreted bytecodes is encountered, a switch back from the native to the interpreted mode is needed. KJIT performs such a switching by pre-processing the bytecodes before their compilation. However, there is a significant overhead, in terms of time and memory space, to perform this pre-processing. For instance, the preprocessed code is 30 percent larger than the original one. KJIT implementation speeds up the execution by a factor ranging between 5.7 and 10.7.

**JBLEND**

Aplix Corporation proposed the JBlend platform [11], which includes a high-performance, small-footprint Java Virtual Machine for CLDC configuration. Actually, JBlend virtual machine is based on CLDC Hotspot acceleration technology. In terms of deployment, JBlend is a leading Java platform in the market of handsets with more than 50 million (2004 statistics) virtual machines running on mobile devices. JBlend offers both interpreted (CLDC-based) and more recently compiled (CLDC Hotspot-based) solutions. The interpreted version achieves, according to a private communication with Aplix engineers, a speedup that is between 2 and 4. The CLDC Hotspot solution achieves a speedup that is up to 10 times. However, it is worth mentioning that there is no technical/academic publication that provides the inner workings of these industrial acceleration technologies.

**ARMED E-BUNNY**

In this section we present a new acceleration technology that we developed for embedded Java virtual machines [12, 13]. Acceleration is achieved by the integration of a new selective dynamic compiler, which we called Armed E-Bunny, into the KVM. The modified KVM is ported on a handheld PDA that is powered with Embedded Linux.

The Armed E-Bunny architecture is depicted in Fig. 1. It includes four major components: the execution engine, the profiler, the one-pass compiler, and the cache manager. Initially all the invoked Java methods are interpreted. During interpretation, a counter-based profiler gathers profiling information. As the code is interpreted, the profiler identifies hotspot methods. Once a method is recognized as hotspot, its bytecodes are translated into native code by the compiler. The produced native code is stored in the cache. On future references to the same method, the cached native code is executed instead of resorting to interpretation.

Many features, such as reduced memory footprint and efficient use of both native and Java stacks, make Armed E-Bunny an appropriate Java acceleration technology for limited-configuration devices. The footprint of the compiler does not exceed 199 kbytes. Furthermore, the use of two stacks (one for interpretation and one for dynamic compilation) makes it possible to preserve the portability of the virtual machine. However, the main drawback of this strategy is...
its complexity. In fact, a method’s related data (e.g., arguments) should be transferred between the Java and the native stacks when necessary. Since ARM architecture specifies a technique for subroutine calls that rely on the registers more than the stack, it is mandatory, each time a method is called, to transfer the needed data from the stack to the registers and vice versa.

**Armed E-Bunny Design and Implementation** — Besides the compilation of all kinds of bytecode, Armed E-Bunny covers the different issues of the integration of a dynamic compiler into a virtual machine such as garbage collection, exception handling, etc. We now discuss the design and implementation of Armed E-Bunny.

**Profiling:** The profiler of Armed E-Bunny performs a simple check over the frequency of calls to a method in order to identify it as a hotspot or not. If a method is recognized as a hotspot, a switching from the interpreted to the native mode is applied. Otherwise, the interpreter continues its execution. In order to perform this check, a counter is added to the structure of the method and is updated each time the method is called. After the virtual machine finishes loading the method parameters, the profiler compares the value of its counter to the threshold specified in the implementation. Depending on the result, the profiler performs one of the following three actions:

- **Invokes the compiler to translate the method, then executes its corresponding generated code.** This occurs if its counter has reached the threshold and it has not already been compiled.
- **Executes the method’s generated code if it is already compiled.**
- **Continues the interpretation of the corresponding method.** When one of the first two actions is chosen, all the method parameters together with the needed information are transferred from the Java to the native stack before execution. The results are then pushed back into the Java stack after finishing the execution of the generated code.

**One-Pass Method Compilation:** Armed E-Bunny uses a lightweight one-pass compilation technique that generates native code of reasonably good quality. Like Java bytecode, the generated code is stack-based, but uses information that is computed at the compilation level. The main role of the compiler is to pass through method bytecodes and translate them into ARM machine code. The generated code is saved into the permanent memory and a reference to it is saved in the structure of the method for future calls. In addition to bytecode translation, a sequence of ARM instructions is generated at the beginning of each method in order to save the values of some registers and variables. Moreover, another sequence is generated at the end of the same method to restore the saved values and switch back to the interpreter. These two processes are called, respectively, prologue and epilogue and are used to save and restore method calling contexts.

**Prologue and Epilogue:** Reestablishing a method-calling context after the execution of a called method, handling native garbage collection, and manipulating threads are the main reasons for generating the prologue and epilogue. During the prologue, the values of the registers R10-R15 are restored, and the value of the frame pointer is saved in the thread data structure, and the current method counter is incremented by 1. These operations are performed by prologue instructions, which figure on top of the method-generated code. During the epilogue, the values of R10-R15 are restored, and the value of the frame pointer in the thread data structure is updated. These operations are carried out by epilogue instructions, which figure in the generated code of the return bytecodes (return, ireturn, arreturn, and ireturn).

**Switching Mechanism:** In Armed E-Bunny the compiled methods are executed using the native stack while interpreted methods are executed using the Java stack, which is stored in the heap. Hence, the execution of Java programs alternates between the native and Java stacks. The switching between the interpreted and native modes implies context transferring between the two stacks. We distinguish between two situations where the switching occurs: interpreted native and native to interpreted. The switching from the interpreted native mode occurs when an invoke bytecode (e.g., invokevirtual, invokespecial, invokestatic) is executed and the invoked method is already compiled. Coming from the interpreted mode, the called method arguments are at the top of the Java stack. The switching to the native mode requires their transfer to the native stack. The switching from the native mode to the interpreted mode occurs in two situations: first, when a compiled method calls an interpreted method; and second, when a compiled method exits and returns back to its interpreted caller method. The adopted profiling strategy assumes that every method called by a compiled method should be compiled. The switching is then reduced only to the second situation (return case). Handling this switching consists of transferring the returned value, if any, from the native stack to the Java stack.

**Threads Management:** The technique that is used in handling threads is inspired by a previous E-Bunny prototype for Intel Architecture [12]. During interpretation, KVM runs its original threading switch services, while during compilation, additional code is generated for bytecodes to cause control transfer between threads. In the interpreted mode methods are executed using the Java stack, while during native mode the generated code is executed using the native stack. In this context, the data structure representing a thread in the virtual machine must hold information about both Java and native stacks in order to handle switching issues. These structures are updated each time a switching between threads occurs.
Exception Handling: Exception handling is an important aspect of the Java language, which has specific semantics to be respected [14]. Our dynamic compiler takes this aspect into consideration by generating efficient code for the bytecode `throw`, which is responsible for raising an exception. Indeed, additional ARM assembly code is added to the virtual machine functions that are called by `throw`. The main intent of such code is to handle the issue of exception propagation in the presence of two execution modes. In fact, during the native mode the method that throws the exception is compiled, while the method that catches the exception can be either interpreted or compiled. This requires careful handling of these two situations. The added code solves this issue as follows. If the method catching the exception is compiled, the additional added code is used to locate the native instruction corresponding to the bytecode handling the exception. After the handled native code is located, the native mode continues and a jump to the native instruction is executed. Otherwise, a switching to the interpreter is performed in order to apply the KVM built-in exception handling process.

Cache Management: The compiled code is saved in a particular cache structure that resides in the permanent space of the heap. Since this structure has a limited size (no more than 64 kbytes), a suitable management should be applied in order to provide enough space for the generated code. This management process is invoked only if the cache is full and is based on the LRU algorithm (Least Recently Used). More precisely, this process passes through all the methods generated in the cache, selects the ones that have not been called for the largest period of time, and removes them. A queue is used to keep the chronological order of invoked methods. This queue is updated each time a compiled method is invoked. The only disadvantage of the LRU algorithm is that some methods may be recompiled several times. However, our experiments show that it is convenient for embedded Java applications.

Garbage Collection Issues: KVM garbage collection is based on a mark-sweep with compaction algorithm. With the selective approach of Armed E-Bunny, compiled methods are executed in the native stack. Consequently, the native stack may contain some object references. Since the current garbage collection algorithm scans only the heap, the native stack will not be considered, and then object references on it will be neither marked nor updated. Therefore, the current KVM garbage collection algorithm is inaccurate with a selective approach; it must be extended to deal with the native stack. More precisely, the translated method frames in the native stack must be scanned in order to mark and update object references. In Armed E-Bunny the garbage collection algorithm is enhanced to address this issue. Mainly, garbage collection functionalities are modified to take into account object references in the native stack.

Experimental Results — To test the results of Armed E-Bunny in the virtual machine, we cross-compiled and ported its ARM executable to a handheld device that is powered with Embedded Linux. Our results demonstrate that Armed E-Bunny requires additional memory space that does not exceed 119 kbytes, including the executable footprint overhead and the translated code storage.

The performance of the Armed E-Bunny selective dynamic compiler is evaluated by running CaffeineMark [15], which is the standard J2ME benchmark, on the original version of KVM with and without Armed E-Bunny. The results illustrated in Table 1 demonstrate that Armed E-Bunny produces an overall speedup of 360 percent over the original KVM. Note that a higher score in this table means better performance. Particularly, the String test is drastically improved thanks to our dynamic compiler since this test contains a loop in which a method that appends strings is frequently called. Figure 2 shows a snapshot and a comparison chart of our tests on an IPAQ handheld that is powered with Embedded Linux.

CONCLUSION

In this article, we presented different acceleration techniques for embedded Java virtual machines. These techniques belong to two main categories: hardware and software accelerations. Hardware acceleration techniques make it possible to achieve a significant enhancement of virtual machine performance. However, the high power consumption and the cost of the underlying hardware processors compel researchers to resort to software acceleration of embedded Java virtual machines. General optimizations, static compilation, and dynamic compilation are in general the three categories of software acceleration techniques. The most prominent software acceleration technique is dynamic compilation. This technique achieves high speedup rates of Java virtual machines for desktop and server systems. However, many issues should be solved when it comes to limited-configuration devices. In fact, due to the limitations in terms of power and memory in limited-configuration devices, a dynamic compilation technique should establish a tradeoff between code quality and compilation time. In this context, we succeeded in designing and implementing an
efficient, lightweight, and low-footprint selective dynamic compiler for J2ME/CLDC targeting ARM architectures. The experimental results demonstrate that an important speedup (more than 360 percent over the last version of Sun’s KVM) is accomplished with a footprint that does not exceed 119 KB.

REFERENCES


BIographies

MOUAD DEBBABI (debbabi@ciise.concordia.ca) holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay, University, France. He has published more than 70 research papers in international journals and conferences on computer security, formal semantics, mobile and embedded platforms, Java technology security and acceleration, cryptographic protocol specification, design and analysis, malicious code detection, programming languages, type theory, and specification and verification of safety-critical systems. He is a full professor and the associate director of the Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada. He holds an M.Sc. degree from Laval University. The main topic of his research activities is the acceleration of Java for ARM Processors. More recently, he started a project on security design patterns.

CHAMSEDDINE TALHI (talhi@ciise.concordia.ca) is a researcher at the Computer Security Laboratory (CSL) at Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada. He holds an M.Sc. degree from Constantine University, Algeria. He is pursuing a Ph.D. thesis on the security of embedded Java platforms. He is interested in characterizing enforceable security policies in resource-constrained platforms. He has worked on the formal specification and verification of telecommunication services. He participated in the design and implementation of a dynamic compiler for J2ME/CLDC.

HAMDY YAHYAOU (hamdy@ciise.concordia.ca) is a researcher at the Computer Security Laboratory (CSL) at Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada. He holds an M.Sc. degree from Laval University. He is pursuing a Ph.D. thesis on the acceleration and semantic foundations of embedded Java virtual machines. He has worked on formal dynamic semantics and control flow analysis of Java. He participated in the design and implementation of a dynamic optimizing compiler for J2ME/CLDC.

IEEE Communications Magazine • September 2005
85

Authorized licensed use limited to: CONCORDIA UNIVERSITY LIBRARIES. Downloaded on September 30, 2009 at 17:10 from IEEE Xplore. Restrictions apply.