Armed E-Bunny: A Selective Dynamic Compiler for Embedded Java Virtual Machine Targeting ARM Processors

Mourad Debbabi
Computer Security Research Group
CIISE, Concordia University
Montreal(QC), Canada
debbabi@ciise.concordia.ca

Azzam Mourad
Computer Security Research Group
CIISE, Concordia University
Montreal(QC), Canada
mourad@ciise.concordia.ca

Nadia Tawbi
LSFM Research Group
Laval University
Quebec(QC), Canada
nadia.tawbi@ift.ulaval.ca

ABSTRACT

This paper presents a new selective dynamic compilation technique targeting ARM 16/32-bit embedded system processors. This compiler is built inside the J2ME/CLDC (Java 2 Micro Edition for Connected Limited Device Configuration) platform [8]. The primary objective of our work is to come up with an efficient, lightweight and low-footprint accelerated Java virtual machine ready to be executed on embedded machines. This is achieved by implementing a selective ARM dynamic compiler called Armed E-Bunny into Sun's Kilobyte Virtual Machine (KVM) [9]. In this paper, we present the motivations, the requirements, the architecture, the design, the implementation and debugging issues of Armed E-Bunny. The modified KVM is ported on an Embedded-Linux PDA and is tested using standard J2ME benchmarks. The experimental results on its performance demonstrate that a speedup of 360% over the last version of Sun's KVM is accomplished with a footprint overhead that does not exceed 119KB.

Keywords

ARM, Java,J2ME/CLDC, KVM, Selective Dynamic Compilation, Embedded Systems.

1. INTRODUCTION

Nowadays, we are, more and more, relying on the use of mobile and wireless systems such as PDAs, cell-phones, pagers, etc., for communication, work and entertainment. The popularity of these devices is increasing day after day. In this context, the Java platform and in particular J2ME/CLDC (Java 2 Micro-Edition for Connected Limited Devices Configuration) is now recognized as the standard execution environment for these wireless devices due to its security, portability, mobility and network support features. Another factor that has amplified the wide industrial adoption of the J2ME/CLDC is the availability of its source code, which allows each company to modify it with respect to their needs. All these factors make Java, J2ME/CLDC and its KVM, the Kilo Virtual Machine designed specially with the constraints of low-end mobile devices in mind, an ideal solution for software development in the domain of embedded systems.

At the same time, the ARM architecture [1, 2] is becoming the industry's leading 16/32 bit embedded system processor solution due to its performance and RISC (Reduced Instruction Set Computer) feature. ARM powered microprocessors are being routinely designed into a wider range of mobile products than any other 32-bit processor. This wide applicability is made possible by the ARM architecture, resulting in optimal system solutions at the crossroads of high performance, small memory size and low power consumption. All these factors make the combinations of the ARM architecture and J2ME/CLDC an interesting domain for research in terms of acceleration. The primary intent of our research initiative is to improve the performance of the J2ME/CLDC Kilo Virtual Machine and to be able to test and visualize our results directly on embedded devices.

Many people were interested in the acceleration of the Java virtual machine and many techniques have been proposed. These techniques are divided into two main approaches: hardware and software acceleration. Regarding hardware acceleration, a significant speedup in terms of virtual machine performance is achieved. However, the high power consumption and the cost of these acceleration technologies encourage researchers to recourse to software acceleration of embedded Java virtual machines. This energy issue is really damaging especially in the case of low-end mobile devices. As examples of these hardware acceleration techniques, many companies such as Zucotto Wireless, Nazomi, etc., have proposed Java processors that execute in silicon Java bytecodes.

General optimizations, ahead-of-time (AOT) optimizations, just-in-time (JIT) compilation and selective dynamic compilation are in general the four categories of software acceleration techniques. General optimizations consist in designing and implementing more efficient virtual machine components such as better garbage collector, fast threading system, accelerated lookups, etc.). Ahead-of-time optimiza-

SAC'05 March 13-17, 2005, Santa Fe, New Mexico, USA

tions consist in using extensive static analysis to optimize program before execution. Flow analysis, annotated type analysis, abstract interpretation, etc., are examples of static analysis. Some results demonstrate that general and ahead-of-time optimization can lead to a reasonable acceleration. However, these techniques cannot compete with other techniques such as JIT and selective dynamic compilation that can reach big speedups (more than 200%) [10]. JIT compilation is the technique used in most modern desktop JVM implementation. A JIT can dramatically increase the execution speed of Java programs. However, they are inappropriate in the context of embedded systems owing to their large code size. Selective dynamic compilation optimizes programs at runtime, based on information available only at runtime, thus offering the potential for greater performance and less memory use. It deviates from JIT compilation by selecting and compiling only those java code fragments that are frequently executed.

In this paper, we describe a working implementation of Armed E-Bunny, a selective dynamic compiler for embedded Java virtual machines that targets ARM processors. Our dynamic compiler is very efficient while keeping the memory footprint overhead less than 119KB. Our results have been tested on an Embedded-Linux handheld. The benchmarks demonstrate that the modified virtual machine is 3.6 times faster than the last version of Sun's Java virtual machine. In addition, our system is the first academic work that targets the acceleration of J2ME/CLDC embedded Java virtual machines by dynamic compilation. It is also one of the very few commercial systems such as CLDC Hotspot and Jbed Micro-Edition that target ARM microprocessors. Our VM also addresses successfully the issues of integrating a dynamic compiler into a Java virtual machine such as exception handling, garbage collection, threads, switching mode mechanism, etc.

2. RELATED WORK

Dynamic compilation became a popular approach to optimize Java performance. Almost all standard Java virtual machines including Java HotSpot VM [7], IBM Jalapeño and OpenJIT are endowed with dynamic compilers. On the other hand, embedded systems lack hardware resources that are available in desktop systems such as hundreds megabytes of RAM or microprocessors operating at over 2 GHz. This sets several limitations on what dynamic compilation could accomplish in embedded systems. In the sequel, we outline these limitations.

In the context of embedded systems, dynamic compilation should cope with two major difficulties. First, the dynamic compiler should be maintained in memory while the application is executing. This is very challenging because of the stringent lack of memory resources. Second, heavyweight code optimizations are not affordable because of their overhead. However, without such optimizations, a dynamic compiler produces a code of low quality and large quantity. This code requires additional memory to be stored. It is worth mentioning that the produced native code could be 8 times the size of the original bytecode. Hence, embedded dynamic compilers are required to be extremely frugal with memory resources. Another consequence of the big size of native instructions compared to bytecode is the risk of instruction cache overflow. Indeed, among the hardware limitations of embedded systems is the reduced amount of on-chip processor instruction cache. The amount of this resource is suitable for bytecode interpretation. However, due to its big size, the machine code produced by the dynamic compiler can be several times larger than the size of the available instruction cache. This leads to several cache misses that decreases the program performance.

Despite these difficulties, dynamic compilation is also used in CLDC-based embedded virtual machines [10, 5, 6]. However, except one paper about KJIT [6], no detailed information about these systems is available in the literature due to commercial reasons.

KJIT [6] is a lightweight dynamic compiler that uses as its foundation the KVM. KJIT does not use any form of profiling for the simple reason that all methods are compiled. The key idea to make this strategy adequate for embedded Java virtual machines is to compile only a subset of bytecodes by pre-processing the bytecodes before their compilation. The cost of pre-processing, however, is an additional time required for pre-processing together with an additional space required to store the generated bytecode. The latter is 30% larger than the original bytecode.

CLDC Hotspot VM [10] is an embedded virtual machine introduced by Sun Microsystems that includes a selective dynamic compiler. Performance critical methods are detected by a single statistical profiler. The compilation is performed in one pass. The memory footprint required by CLDC Hotspot (including APIs) reaches 1 megabyte which is almost the double of the space required by KVM. No more details are provided about the CLDC Hotspot dynamic compiler.

3. ARMED E-BUNNY

In this section, we describe the elaboration of a dynamic compiler, called Armed E-Bunny, that targets the ARM platform. The proposed system uses, as starting virtual machine, the last version of Sun's KVM. The architecture of Armed E-Bunny is inspired by [3].

3.1 Architecture

Armed E-Bunny contains six major components: the method initializer, the profiler, the interpreter, the machine code execution engine, the ARM compiler and the cache manager. Figure 1 depicts the architecture of Armed E-Bunny and shows the relationship between its components. Many features, such as the reduced memory footprint and the efficient use of different stacks, make our Armed E-Bunny an appropriate Java acceleration technology for embedded virtual machines. The use of different stacks for interpretation and compilation is a real advantage to preserve the portability to a high extent of the virtual machine, even though it is complex. Armed E-Bunny is an embedded selective dynamic compiler. Only the frequently called methods are compiled to native code and saved in the cache structure. This strategy led us to a reduced memory footprint that does not exceed 119KB.

Once the method is initialized by the method initializer of the KVM, the profiler is able to identify if a method is a hotspot or not and specify the mode in which the decoding should be performed. In Armed E-Bunny, the interpreter communicates with the profiler before starting its decoding process. If the method is identified as a hotspot, the interpreter stops its work and the profiler switches either to the machine code executer or the ARM compiler. If a

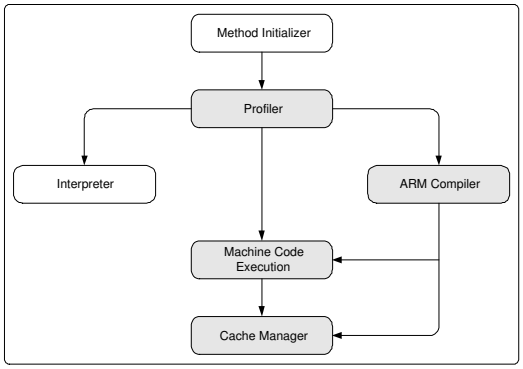

Figure 1: Armed E-Bunny Architecture

switchThe ARM compiler is a one pass compiler. Its role is to go through the bytecodes of a given method, generate the corresponding ARM executable machine code and save the machine code in the cache. At this level, a cache management process is invoked only if the cache is full. This process is performed by the cache manager and its role is to find enough space for the generated code. Actually, 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. We used the LRU (Least Recently used) algorithm. The machine code executer is the component responsible of invoking the cached machine code of a method already or recently compiled.

3.2 Design

Our system covers besides the compilation of all kind of bytecodes, the different issues of the integration of a dynamic compiler into a virtual machine such as garbage collection, exception handling, etc. In this section, we discuss in detail the design of Armed E-Bunny.

3.2.1 Profiling and Mode Switching

The profiler of Armed E-Bunny performs a check over the frequency of a method call in order to declare it as a hotspot. If a method is recognized as a hotspot, a switch from interpretation to compilation mode is applied. 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. Once the virtual machine finishes loading the parameters of a method, the profiler compares the value of its counter to the threshold specified in the implementation. Depending on the result, the profiler either: Compiles the method and then executes its corresponding generated code if its counter reaches the threshold and it is not already compiled. Or, executes the method's generated codes if its counter reaches the threshold and it is already compiled. Or, continues the interpretation of the corresponding method. When one of the first two cases is chosen, all the method parameters and information needed are transferred from the Java stack to the native stack before execution. Then,the results are pushed back into the Java stack after finishing the execution of the code generated.

3.2.2 One Pass Method Compilation

Our compilation spans over a lightweight one pass compilation technique that generates a code of reasonably good quality. The generated code is stack-based as Java bytecode, but uses many information that are computed at the compilation level. The main role of the compiler is to pass through the method bytecodes and translate them into executable machine code (binaries) for ARM machines. The generated code is saved in the permanent memory and a reference to it is saved in the structure of the method for future calls. This section explains in details all the steps that our compiler passes through.

3.2.2.1 Machine Code Prologue and Epilogue.

A list of ARM instructions are generated at the beginning and the end of each method execution in order to save the values of some registers and variables. Re-establishing the calling method context after the execution of the called method, handling native garbage collection and manipulating threads are the main reasons for generating the prologue and epilogue. During prologue, the values of the registers *R10-R15* are pushed into the native stack, the value of frame pointer is saved in the thread data structure, the reference of the generated code is saved in the method data structure and the current method counter is incremented by 1. During epilogue, the values of the registers *R10-R15* are restored and the value of the frame pointer saved in the thread data structure is updated.

3.2.2.2 Bytecodes Translation.

Unlike common compilers, Armed E-Bunny is based on a bytecode translation technique which avoids complex computations. The translator passes through the method's bytecodes using a while loop, identifies the bytecode and then generates its corresponding ARM machine code. The machine code generation is implemented by following a top-down strategy. First, each bytecode is translated into a list of C functions called "Gen" functions (e.g *GenMOV*). Each one of them is able to generate a list of one or more ARM assembly language instructions. Second, inside the "Gen" functions, each instruction is transformed to its equivalence in ARM machine code hexadecimal form by our own ARM assembler implemented inside the virtual machine. Finally, each time an instruction is generated, there is a function responsible of saving it into a frame in the *MachineCode* table to be eventually executed. Table 3 shows an example of such a translation technique.

Actually, our ARM assembler that is implemented inside the virtual machine does not perform the work of a real assembler. However, its role is to transform directly part of the assembler instructions into ARM machine code ready to be executed. We refer in our implementation to the documentation of ARM assembly language and we follow the same architecture used in transforming assembly instruction to executable machine code. Since all the instructions size is fixed to 32 bits, in some cases we were obliged to represent an assembly instruction by a set of many machine code instructions. A machine code instruction is the 32 bits binary number that is understood by the ARM microprocessor (e.g *0xe3a01002* is the machine code representation of *mov r1, r2*).

The above strategy of translation is applied on all the bytecode, even though we differentiate them with respect to their implementation complexity. Some bytecodes such as loads (e.g. *iload*), stores (e.g. *astore*), stack manipulation (e.g. *push, pop*), arithmetic except division, logic and shift

IADD bytecode Implementation

```
GenPopToRegister(R0);
GenAddRegisterToRegisterContent(R13,R0);
```

GenPopToRegister(R0) Implementation

```
//This function generates the hexadecimal values
//of the assembler instruction
SetMachineCode(0x........); //ldr r0, [r13]!
```

GenAddRegToRegContent(R13,R0) Implementation

```
//This function generates the hexadecimal values
//of the assembler instruction
SetMachineCode(0x........); //ldr r1, [r13]
SetMachineCode(0x........); //add r1, r0
SetMachineCode(0x........); //str [r13], r1
```

SetMachineCode(0x.....) Implementation

```
*(MachineCodeTable) = 0x........;
```

Table 1: Code Generation

(e.g. *iadd, iand*, ishr), and branching (e.g. *ifne, ificmpeq*)are directly translated into machine code, which reproduces the interpreter behavior on the native stack. Other bytecodes such as field access, object creation, array manipulation, method invocation, return, monitor, casting and exception require some virtual machine services (e.g. method lookup, field reference resolution) at runtime in order to be translated. Generating the corresponding native code instruction by instruction, including virtual machine services, yields a complex and very bulky code. For this reason, we adopted in Armed E-Bunny a different approach, which allows to call these services from the native code. In addition to calling virtual machine services, we implemented some C functions to generate some complicated operations of some bytecodes. These C functions use the same stack we use for native code, so we do not need to transfer the method parameters to the Java stack each time we need to switch to C mode. Hence, the resulting generated machine code is compact and less complex.

3.2.2.3 Fast ByteCodes Translation.

In KVM, some method's bytecodes such as *getfield, putstatic, invokevirtual*, etc. are replaced by fast bytecodes the first time they are executed in this method [4]. Such bytecodes need virtual machine services such as *resolveMethodReference*, which calls a set of functions in order to load the method references and parameters. To avoid that each time a method is called, KVM uses a mechanism that replaces some bytecodes by their corresponding fast bytecodes (e.g. *invokevirtual* by *fastinvokevirtual*) and saves all the values needed in the cache. The next time the same method is executed, all the references and parameters are loaded from the cache instead of calling the virtual machine functions. Armed E-Bunny compiles also all the fast bytecodes. It uses the same KVM's mechanism to replace the bytecodes by their corresponding fast bytecodes. At this level of compilation of this kind of bytecodes, the system does not need to switch the execution mode in order to call the virtual machine services that are needed.

3.2.2.4 Java Native Methods.

The Java virtual machine provides a list of Java native methods that are neither interpreted nor compiled. These methods are implemented directly in the C language. They are based on the Java stack. In Armed E-bunny, the profiler deals with this kind of methods and calls their corresponding native functions before switching to the compilation mode. However, these subroutines may be called also during compilation by the invoke bytecodes (i.e. *invokevirtual, invokeinterface and invokestatic*). For this reason, a process of three steps is performed inside the implementation of these bytecodes once a native method is detected. First, the compiler use the Java stack to push the method's arguments. Second, the *invokenativefunction* of KVM is called in order to invoke the method. Finally, when the execution is accomplished, the results and the arguments are popped from the Java stack and only the results are pushed back into the native stack. Indeed, this mechanism allows us to switch successfully between the two stacks without the need to return back to the interpreter.

3.2.3 Garbage Collection

Our system allows native method calls and memory allocation during compilation mode. This means that the garbage collection of KVM may be called and some references saved in the heap may be lost because the current algorithm of KVM garbage collection doesn't take into account the objet allocated in the native stack. The current KVM garbage collection goes through three steps: mark, sweep and compact. Based on the result of marking, the garbage sweeps the free chunks to constitute consistent blocks and compact the heap, leading to a move of the object inside it. This is done in our compiler by adding some features to the KVM garbage collection. Using C and ARM assembly language code, information about the native stack are gathered and passed to the marking process which passes over the native stack, scans it and marks all its live objects in the heap. By doing that, the garbage collection will treat, whenever is called, all the native marked object as if they are Java stack object references.

3.2.4 Exception Handling

Exception handling is an important feature of the Java language which has specific semantics to be respected [4]. Our dynamic compiler handles it by generating efficient code for the bytecode *athrow* which is responsible of raising an exception. Additional ARM assembly code is also added to the functions that are called by *athrow* (i.e *throwException*) and new issues relevant to exception propagation are introduced. During compilation mode, the method that throws the exception is always compiled. However, the method that catches the exception can be either interpreted or compiled. In the two situations, a call to virtual machine functions is applied to throw the exception. If the method catching the exception is compiled, the additional code added to the virtual machine *throwexception* function is used to locate the native instruction corresponding to the bytecode handling the exception. Once the handled native code is located, the compilation mode continues and a jump to the native instruction is executed. Otherwise, a switch to the interpreter is applied to continue its normal exception handling process.

3.2.5 Threads

The technique that have been used in handling threads is inspired by [3]. During interpretation, the KVM runs its original threads switch services, while during compilation,

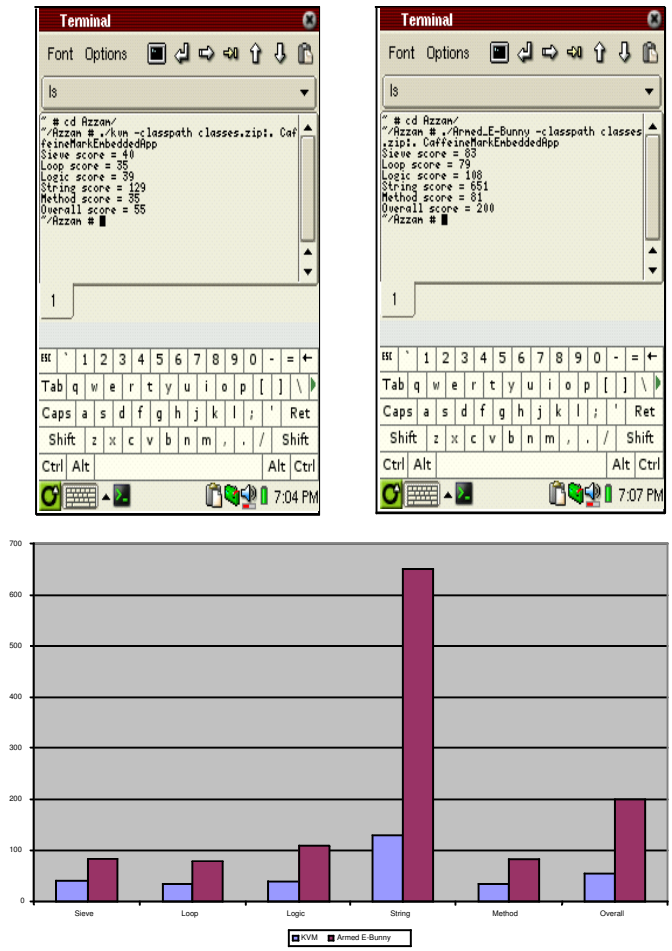

Figure 2: Caffeine Scores of the KVM and Armed E-Bunny

additional code is generated for bytecodes causing transfer control and contexts saving. Moreover, the data structure representing the thread in the virtual machine holds information about both Java and native stack contexts for switching issues. Whenever the register holding the time-slice value reaches the zero value, a thread switch is triggered. Actually, the application of this switch mechanism prevents a compiled thread from going into an infinite loop and the virtual machine from hanging indefinitely.

3.3 Debugging

Armed E-Bunny is implemented in the C programming language and the ARM assembly language. Our dynamic compiler is cross-compiled on an Intel workstation using the GNU arm-linux-gcc and then is ported on an Embedded-Linux Handheld for execution. For the debugging issues, we used the GNU arm-linux-gdb built on an Intel workstation and the server of this debugger installed on the Handheld. A connection between the debugger and its server permitted us to trace the execution of the virtual machine. Debugging using these tools was difficult due to their restricted options and the delay exhibited during program tracing. However, the execution speedup we reached made all this worthwhile.

4. EXPERIMENTAL RESULTS

To test the results of Armed E-Bunny in the virtual machine, we ported its ARM executable to a Handheld and we executed it. Our results shows that Armed E-Bunny requires additional memory space that does not exceed 119KB, including the executable footprint overhead and the translated code storage. The performance of Armed E-Bunny selective dynamic compiler is evaluated by running the CaffeineMark benchmark on the original version of KVM 1.0.4 with and without Armed E-Bunny. The results demonstrated that Armed E-Bunny produces an overall speedup of 360 % over the original KVM 1.0.4. Figure 3 shows a snapshot and a comparison chart of our tests on an Ipaq H3600 under Embedded-Linux.

5. CONCLUSION AND FUTURE WORK

This paper describes a new acceleration technology for Java embedded virtual machines that targets the ARM 16/32-bit embedded system processors. This technology is based on selective dynamic compilation. The compiler is built inside The J2ME/CLDC (Java 2 Micro Edition for Connected Limited Device Configuration). Our results shows that our work comes up with an efficient, lightweight and low-footprint accelerated Java virtual machine ready to be executed on ARM embedded machines. Our experimental results proves that a speedup of 360% over the last version of Sun's KVM is accomplished by Armed E-Bunny with a footprint overhead that does not exceed 119KB.

Regarding our future work, we are still working on enhancing the quality of our ARM dynamic compiler in order to accomplish better speedups. At the same time, we are trying to optimize our cache management, garbage collection and threading mechanisms.

6. REFERENCES

[1] ARM. ARM7TDMI Data Sheet. White Paper, 2001.

[2] ARM. ARM Developer Suite Assembler Guide. White Paper, 2001.

[3] M. Debbabi, A. Gherbi, L. Ketari, C. Talhi, N. Tawbi, H. Yahyaoui, and S. Zhioua. E-Bunny: A Dynamic Compiler for Embedded Java Virtual Machines. In *Proceedings of the Third International Conference on the Principles and Practice of Programming in Java*, pages 100–107, Las Vegas, USA, 2004. ACM Press.

[4] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.

[5] K. Schmid. Esmertec's Jbed Micro Edition CLDC and Jbed Profile for MID. Technical report, Esmertec AG, Dubendorf, Switzerland, Spring 2002.

[6] N. Shaylor. A Just-in-Time Compiler for Memory-Constrained Low-Power Devices. In *Proceedings of the 2nd Java Virtual Machine Research and Technology Symposium*, pages 119–126, San Francisco, CA, USA, Aug. 2002.

[7] Sun. The Java HotSpot Performance Engine Architecture. White Paper, April 1999.

[8] Sun. CLDC Specification v1.0, Java 2 Platform Micro Edition. White Paper, May 2000.

[9] Sun. KVM Porting Guide. White Paper, March 2003.

[10] Sun. CLDC HotSpot Implementation Virtual Machine. White Paper, March 2004.