A Selective Dynamic Compiler for Embedded Java Virtual Machines Targeting ARM Processors

Mourad Debbabi, Abdelouahed Gherbi, Azzam Mourad, and Hamdi Yahyaoui

Computer Security and Acceleration Research Group,
Concordia Institute for Information Systems Engineering,
Engineering and Computer Science Faculty,
Concordia University,
Montreal, Qc, Canada.

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 [17]. The primary objective of this work is to elaborate 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) [18]. In this paper we present the motivations, the architecture, the design and the implementation of Armed E-Bunny. The modified KVM is ported on a handheld PDA that is powered with embedded Linux and is tested using standard J2ME benchmarks. The experimental results demonstrate that a speedup of 360% over the last version of Sun's KVM is accomplished with a footprint that does not exceed 119 KB. An important result of this paper is also the proposition of an acceleration technique that leverages Armed E-Bunny by establishing a synergy between efficient interpretation and selective dynamic compilation. The main traits of this technique are: a one pass compilation by code reuse, an efficient threaded interpretation and a fast switching mechanism between the interpreted and compiled modes.

1. Motivations and Background

The use of wireless systems such as PDAs, cell phones, pagers, etc. becomes a need in our everyday life. Their popularity 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 types of wireless devices due to its security, portability, mobility and network support features. Another factor that has amplified the wide industrial adoption of J2ME/CLDC is the availability of its source code, which allows each company to undertake its modification according to their needs. All these factors make Java, J2ME/CLDC and its KVM, an ideal solution for software development in the arena of mobile embedded systems. At the same time, the ARM architecture [11,12] 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 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. This elucidates the wide adoption of the combination of the ARM architecture and J2ME/CLDC as a platform of choice in nowadays mobile and wireless devices. Accordingly, the fruitful research on this popular platform will have positive, substantial and widespread impact.

The primary intent of this work is to address the acceleration of the aforementioned platform, which is a key aspect in the successful deployment of the underlying mobile devices.

A great deal of interest has been expressed in the acceleration of the Java virtual machines and many techniques have been proposed. These techniques could be classified 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 resort to software acceleration of embedded Java virtual machine. 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 [4], Nazomi [20], etc. have proposed Java processors that execute in silicon Java bytecodes. For software acceleration, a large spectrum of techniques has been advanced [1,8,10,16]. These techniques could be classified into 4 categories: general optimizations, ahead-of-time (AOT) optimizations, just-in-time (JIT) compilation and selective dynamic compilation.

General optimizations consist in designing and implementing more efficient virtual machine components (better garbage collector, fast threading system, accelerated lookups, etc.). Ahead-of-time optimizations consist in using extensive static analysis (flow analysis, annotated type analysis, abstract interpretation, etc.) to optimize programs before execution. Just-in-time (JIT) compilation consists of the dynamic compilation of Java executables (bytecode). This dynamic compilation is achieved thanks to a compiler that is embedded in the Java virtual machine. The compiler is in charge of translating bytecode into the native code of the host platform on which the code is being executed. The selective dynamic compilation consists in compiling, on the fly, into native code only a selected set of methods that are performance-critical.

Experience demonstrated that general and ahead-of-time optimizations can lead to reasonable accelerations. However, they cannot compete with just-in-time and selective dynamic compilation in reaching big speedups (for instance an acceleration of more than 200 %) [19].

It is established that just-in-time compilers require a lot of memory to store the dynamic compiler and the binary code that it generates. The compilation process also implements sophisticated flow analysis and register allocation algorithms in order to generate optimized and high-quality native code. JIT compilers allow to reach high speedup but the static analysis that they use induces a significant overhead in terms of memory and time. This makes JIT compilers much more appropriate for J2SE (Java 2 Standard Edition) and J2EE (Java 2 Enterprise Edition) platforms.

Selective dynamic compilation deviates from the JIT compilation by selecting and compiling, on the fly, only those fragments of the class files that are frequently executed. These code fragments are generally referred to as hotspots. For instance, one can select only those methods that are frequently invoked and convert them to native code. By doing so, significant acceleration of the virtual machine could be reached since efficient optimizations are concentrated on performance critical fragments of the program. Another major advantage of this approach is to reduce memory overhead because only a part of a program is converted to native code. This makes selective dynamic compilation more adequate for embedded systems than JIT compilers.

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 mechanism between the compiler and the interpreter modes, etc. Strengthened by the results of the Armed E-Bunny project and the downstream insights, we started looking for more acceleration. This led us to the idea of establishing a synergy between efficient interpretation and selective dynamic compilation. Actually, efficient interpretation is achieved by a generated threaded interpreter that is made of a pool of codelets. The latter are native code units efficiently implementing the dynamic semantics of a given bytecode. Besides, each codelet carries out the dispatch to the next bytecode eliminating therefore the need for a costly centralized traditional dispatch mechanism. The acceleration technique that we propose here leverages Armed E-Bunny technology by taking advantage of the threaded interpreter and reusing most of the previously mentioned codelets. This tight collaboration between the interpreter and the dynamic compiler leads to a fast and lightweight (in terms of footprint) execution of Java class files.

Here is how the rest of the paper is organized. In Section 2, we present the state of the art. Sections 3 and 4 describe the architecture, design and implementation of Armed E-Bunny. In Section 5, we propose an acceleration technique that leverages Armed E-Bunny by establishing a synergy between dynamic compilation and code reuse. Section 6 describes our experimental results. Finally, Section 7 gives some concluding remarks on this work as well as a few statements on future research.

2. Related Work

The most appealing features of the Java language [9] are its portability and security. However, the revers of the medal is its poor performance when compared to other languages such as C and C++. Lately, a surge of interest has been expressed in the acceleration of Java virtual machines for embedded systems. Accelerating the interpretation

mechanism has been and is still a focus of interest for many researchers. Generally a pure bytecode interpreter is an infinite loop embedding a switch-case statement that dispatches to a sequence of bytecodes. Each switch case value implements one Java bytecode. This entails a significant overhead. To circumvent this drawback the use of direct threaded interpretation has been suggested. The latter is is an interpretation technique introduced in the Forth programming language [7]. Thanks to this technique, the central dispatch is eliminated. Each bytecode of the method being interpreted is replaced by an address of a corresponding implementation. In addition, such an implementation ends with the required dispatch to the next opcode.

The inline threading interpretation technique [22] improves upon the direct threading technique by eliminating the dispatch overhead within basic blocks. The former technique identifies bytecode sequences that form basic blocks. A new implementation is then dynamically created for such sequences by copying and catenating each bytecode's implementation in a new buffer. The dispatch code is then copied at the end.

More recently, Gagnon and Hendren [8] proposed a technique that copes with the difficulties that arise when adapting the inline threading technique to Java. In fact, Java's features such as lazy class initialization, lazy class loading and linking, and multi-threading support conflict with the implementation of inline-threaded interpretation technique. The proposed technique, called "preparation sequence", solves the problems caused by in-place code replacement within an inline-threaded interpreter of a Java virtual machine.

These acceleration techniques of the interpretation mechanism achieve a reasonable speed-up. However, they fail to compete with those approaches that introduce some sort of compilation (AOT, JIT, selective dynamic compilation, etc.) [1–3,10,16,25,26]. The drawbacks of these techniques are, however, the loss of portability and the increasing complexity.

Among the acceleration techniques that rely on dynamic compilation, one can cite Jalapeño [1]. It consists in using multi-level optimizing compilers to generate an efficient machine code. All the methods are systematically compiled. This is not appropriate when it comes to embedded systems since the required memory space and the compilation overhead are important. Besides, the optimizations carried out by subsequent more optimizing compilers are based on the recompilation of performance-critical methods, which rises up the compilation overhead.

The Java HotSpot virtual machine [16] uses a selective approach to compile performance-critical methods. These are detected by means of a profiler while interpreting the remaining ones. The Java HotSpot compiler achieves improved performance by applying several optimizations [21]. These include class-hierarchy inlining, global value numbering, optimistic constant propagation, optimal instruction selection, graph coloring register allocation and peephole optimization. All these techniques are unaffordable in the setting of embedded systems.

Very little seems to have been published about the acceleration of embedded Java virtual machines. Sun introduced an embedded VM called CLDC HotSpot [19]. It is a lightweight accelerated VM for embedded systems. Its main features are: a compact object layout, a fast synchronization, a dynamic compilation of hotspot methods and a unified memory management. The CLDC-HotSpot dynamic compiler performs one pass over bytecodes to generate machine code. No more details about this compiler are provided.

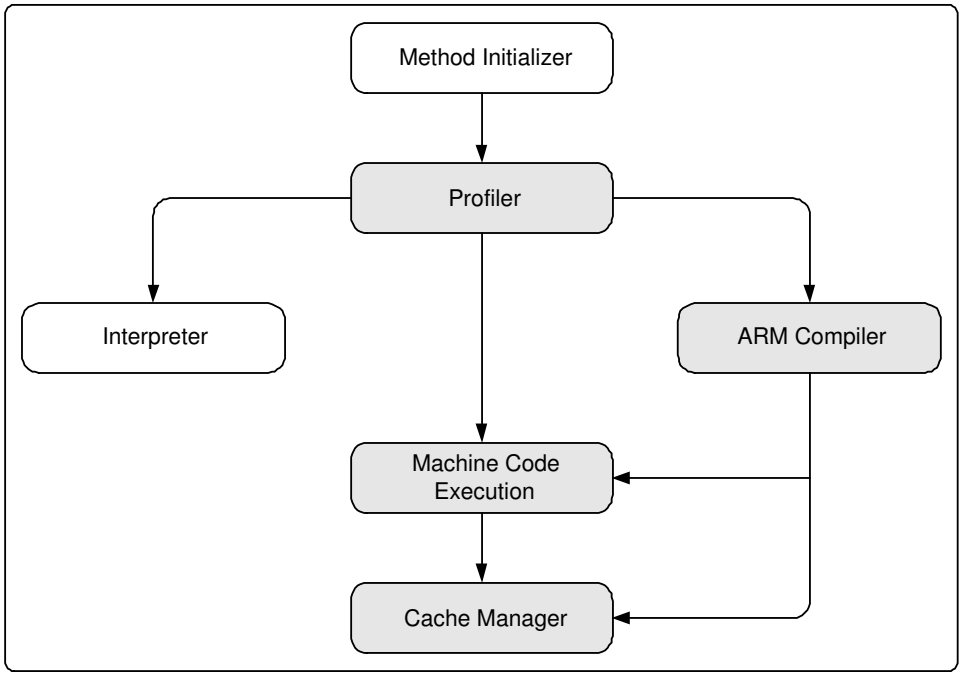

Figure 1. Armed E-Bunny Architecture

KJIT [24] uses a pre-compilation transformation of all method bytecodes. The aim of this transformation is to remove the creation of intermediate results at run-time by adding new local variables. Only a subset of bytecodes is translated into machine code. The proposed transformation allows a smooth register-based switch between interpreted and native modes. However, the proposed design introduces a code increase overhead equal to 30 %. Moreover, it does not take into consideration all the compilation issues such as thread switching and scheduling, exception handling, and the switching between the interpreted and compiled modes.

In the sequel, we present Armed E-Bunny architecture, design and implementation.

3. Armed E-Bunny Architecture

Armed E-Bunny contains six major components: the Method Initializer, the Profiler, the KVM 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. Notice that the Virtual Machine Execution Engine and the Interpreter already exist in the KVM.

Many features like reduced memory footprint and efficient use of different stacks make our Armed E-Bunny an appropriate Java acceleration technology for embedded systems. Armed E-Bunny is a selective dynamic compiler: only the frequently called methods are compiled and saved in the cache structure. This strategy led us to have reduced memory footprint that does not exceed 119 KB. Furthermore, the use of two stacks (one for interpretation and one for compiled method execution) is a real advantage to preserve the

portability to a high extent of the virtual machine. However, the drawback of this way is its complexity due to the following reasons. First, method's related data (e.g. arguments) should be transferred between the Java and native stacks when necessary. Second, since ARM architecture specifies a technique for a subroutine call that relies on the registers more than the stack, it is mandatory, each time a method is called, to transfer data needed (e.g. subroutine arguments) from the stack to registers and vice versa. Hereafter, we describe the mechanism of method interpretation and compilation.

Initially a method is supposed to be interpreted. Once the method is loaded by the virtual machine, the profiler checks its frequency by verifying if its counter has reached the threshold defined in our implementation. The result of this verification identifies if the method is hotspot [1]or not. If it is recognized as hotspot, a quick switch to the Machine Code Execution or to the ARM Compiler is performed by the profiler. Otherwise, the Interpreter takes its advantage over the other components and continues its work normally.

During the switch mechanism, two cases are possible. If the method is already compiled, the reference that points to its machine code in the cache is called. Otherwise, the ARM compiler translates the given method into ARM machine code and stores it in the cache. Once the translation is completed, the corresponding machine code is called and a reference to this code is saved in the structure of the method for future calls. Detailed explanation about all these mechanisms is described in section 4. Here we highlight the roles of the six main components of our proposed embedded ARM dynamic compiler's architecture and we describe the interaction between its components.

Method Initializer

This component already exists in the virtual machine. It is responsible for loading all the method's references and parameters needed for both interpretation and compilation.

Profiler

The profiler has many related roles and communicates with all the other components. By checking the method's counter, this component is able to identify if a method is hotspot or not and specify the mode in which the decoding should be performed. Once a switch to the compiled mode is done, additional role to the profiler allows it to choose either to execute the method's corresponding machine code found in the cache directly or to call the compiler and then run the relevant machine code. The ARM compiler is called only if the method is not already compiled.

Interpreter

The KVM interpreter decodes the bytecodes of a given method into executable machine code. In Armed E-Bunny, the interpreter communicates with the profiler before starting its decoding process. If the method is identified as hotspot, the interpreter stops its work and the profiler switches to other components.

Machine Code Executer

This component is responsible for invoking the machine code of a method. Once the stack and the registers are filled with the data needed for method execution, the reference of the machine code found in the cache is called. A possible switch back to the interpreter

[1]Frequently called method.

is triggered from this component.

ARM Compiler

The ARM compiler is called by the profiler. It is a one-pass compiler. Its role is to go through the bytecodes of a given method and generate the corresponding ARM executable machine code. Once the generation is done, a management of the cache is applied, if needed, and the code is saved inside. A reference to the generated code is saved in the method structure for future invocations. In some special cases, the compiler switches to the interpreter in order to gather some information or invoke some complex native methods and then returns back to continue its translation. Detailed explanation about its functionality is provided in Section 4.

Cache Manager

Compiled code is saved in a particular structure, that resides in the permanent space of the heap, called the cache. Since this structure has a limited size (no more than 64 KB), a management should be applied in order to find enough space for the generated code. This management process is invoked only if the cache is full. 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 use the LRU algorithm (Least Recently used) [14]. 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 in LRU algorithm is that some methods may be recompiled several times. However, our experiments show its efficiency.

4. Armed E-Bunny Design and Implementation

Our system covers, besides the compilation of all the 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. Profiling and mode switching, one-pass method compilation, garbage collection, exception handling and threads are the following discussed main points.

4.1. Profiling and Mode Switching

The profiler of Armed E-Bunny performs simple check over the frequency of a method in order to identify it as hotspot or not. If a method is recognized as hotspot, a switch from the interpreted to the compiled 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.

Once 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 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 code if it is already compiled. Or,
- Continues the interpretation of the corresponding method.

Table 1
Profiler Algorithm

```
// Profiler
if (methodCounter >= threshold)
begin //Hotspot Method
  if (currentMethodNotCompiled)
      compileCurrentMethod;
  if (currentMethodCompiled)
  begin //Machine code execution
  //The 3 following instructions prepare the native
  //stack before execution
      pushNativeStack methodArguments;
      leaveSpace; //for local variables
      leaveSpace; //for returned values
      call currentMethodMachineCode;
      popNativeStack returnedValue;
      pushJavaStack returnedValue;
  end //Machine Code execution
end //Hotspot Method
```

When one of the first two cases is chosen, all method parameters and needed information 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 generated code. Table 1 summarizes briefly the implementation of the profiler.

4.2. 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.

In addition to bytecode translation, and like all compilers, a list of ARM instructions is generated at the beginning of each method in order to save the values of some registers and variables. Also, another dual list 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 contexts. This section explains in details all the steps that our compiler passes through.

4.2.1. Prologue and Epilogue

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

Table 2
Machine Code Generation

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]!
```

GenAddRegisterToRegisterContent(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.....;
```

registers R10-R15 are restored and the value of the frame pointer saved in the thread data structure is updated. Indeed, prologue instructions figure on top of the method generated code and epilogue instructions figure in the generated code of the return bytecodes (*return, ireturn, areturn and lreturn*).

4.2.2. 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 to a list of C functions called "gen" functions (e.g *genPop*). 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 equivalent in ARM machine code hexadecimal form by our own ARM assembler that is implemented inside the virtual machine. Finally, each time an instruction is generated, there is a function responsible for saving it into a frame in the *MachineCode* table to be eventually executed. Table 2 illustrates our 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. A machine code instruction is the 32 bit binary number that is understood by the ARM microprocessor (e.g. *0xe3a01002* is the machine code representation of *mov R1, R2*). For

instance, loading an immediate value in a register requires the generation of a sequence of many instructions in addition to many bit-manipulation operations.

The above strategy of translation is applied on all the bytecodes, even though we differentiate them with respect to their implementation complexity and functionality. Some bytecodes such as loads (e.g. *iload*), stores (e.g. *astore*), stack manipulation (e.g. *push, pop*), arithmetic except division, logic and shift (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 throwing 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. Hence, the resulting generated machine code is compact and less complex. In the sequel, we present some categories of bytecodes and outline for some of them our translation strategy. Notice that the following ARM assembly instructions are represented in hexadecimal before saving them into the *MachineCode* table.

Branching Bytecodes This category includes the following bytecodes: *ifeq, ifne, iflt, ifge, ifgt, ifle, if_icompeq, if_icompne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, if_acmpeq, if_acmpne, goto, tableswitch, lookupswitch, goto_w*. Bytecodes of this category perform unconditional and conditional branching, which can serve for the if/else, switch and loop operations. Branching can be performed in two ways: forward and backward. Translating backward branching is reduced to the generation of a simple jump to the address of the already generated instruction. Such operation is not complex since the addresses of all the generated instructions are saved at the compilation time in a defined structure. On the other hand, for forward branching, the address of the instruction or bytecode to which the jump should be performed does not exist yet in the structure containing the addresses. For this reason, the complete translation of the forward branching is postponed until the target bytecode is reached and compiled, while the address of the uncomplete generated instruction is saved in a particular structure that is defined for this purpose. Whenever the compiler reaches the target instruction or bytecode, the address of the uncomplete generated instruction is loaded and its content is filled with the corresponding ARM machine code. Table 3 shows the translation algorithm of *goto* bytecode.

Allocation, Array Manipulation and Field Access Bytecodes This category includes the following bytecodes: *new, newarray, anewarray, arraylength, multianewarray , putfield, getfield, putstatic, getstatic*. Bytecodes of this category create and manipulate objects and arrays, and access to object fields using symbolic references in order to get or set their values. Indeed, such bytecodes need to call some KVM services in order to resolve field and class references, initialize classes and allocate memory space in the heap. To deal with this, we implement some functions in ARM assembly and C languages in order to call these KVM subroutines and return

Table 3
goto Translation Algorithm

```
// goto : Branch always {
  if (backward_branch)
    generate jump instruction;
  else //forward branch
    leaveSpace;
    continue bytecode generation Until reaching the target bytecode

  if (target_bytecode_reached){
    load the address of the corresponding free space leaved
    generate jump instruction at the loaded address; } }
```

Table 4
getfield Translation

```
// getfield : get field value in object {
   mov R0, ip; // ip is the address of the bytecode
   ldm sp!, {R1}; //pop the content of SP (top of the stack) to register R1
   mov LP, PC;
   mov PC, &getfield_function;
   //These two instructions are used to call the function getfield_function
   //The arguments of getfield_function ip and object_reference
   //are sent into R0 and R1. The function getfield_function
   //calls some KVM services to get the field value of the
   //addressed object and return the result into the register R0
   stm sp!, {R0}; //push the register R0 to the content of SP (top of the stack)
}
```

the resolved references and results into registers. Table 4 shows the translation of *getfield* bytecode.

Invoke Bytecodes This category includes the following bytecodes: *invokeVirtual, invokeSpecial, invokeInterface and invokeStatic*. Bytecodes of this category allow a method to call another one. In fact, such bytecodes need to call some KVM services such as method reference resolution and method lookup in order to load the method references, parameters and local variables and verify them. To deal with that, we apply the same strategy as with the allocation, array manipulation and field access bytecodes, i.e implementing some functions to call these KVM subroutines. Moreover, according to our strategy of compilation, compiled method can only call a compiled or native method, which means that a switch to interpreted mode is not allowed at this level of execution. Hence, there are three different cases to be treated. First, if the invoked method is native, a special technique is used in order to call the corresponding native function. This technique is detailed in Section 4.2.3. Second, if the invoked method is already compiled, a simple call to its executable machine code is performed. The third case is when the method is not yet compiled.

Table 5
invokeVirutal Translation

```
// invokeVirtual : invoke virtual method {
  mov R0, ip; // ip is the address of the bytecode
  sub SP, SP, #8;
  //This instruction is used to leave 2 empty spaces in the native stack
  mov LP, PC;
  mov PC, &invokevirtual_function;
  //These two instructions are used to call the function
  //invokevirtual_function. The arguments of invokevirtual_function ip
  //is sent into R0. The function invokevirtual_function
  //returns its results into R0, R1 and the top of the native stack
  //R0 contains 0 or 1, and R1 contains the address of the compiled method

  Switch R0
  1: add SP, SP, R1;
      //If the value of R0 is 1, this means that the called method is
      //Java native and was treated in the function invokevirtual_function
      //This instruction is used to update the native stack
  0: ldr R0, [SP];
     //If the value of R0 is 0, this means that the address of the
     //method to be executed is sent in the register R1 and the
     //number of local variables is pushed at the top of the native stack
     sub SP, SP, R0;
     //This instruction is used to leave empty space for local variables
     mov LP, PC;
     mov PC, R1;
     //These two instructions are used to call the machine code
     //of the method
     add SP, SP, R0;
     //This instruction is used to update the native stack
     //after the method call
     UPDATE_Interpreter_Global_Variables;}
```

In such case, the method is first compiled and then its generated machine code is executed. Table 5 shows the translation of *invokeVirtual* bytecode.

Return Bytecodes This category includes the following bytecodes: *return, ireturn, areturn and lreturn.* Bytecodes of this category are always placed at the end of a method in order to return to the calling method. In addition to the return operation, such bytecodes restore the calling method context (by epilogue instructions) by pushing the returned values in the stack, restoring the old values of status registers and deleting the method frame from the stack. Table 6 shows the translation of *return* bytecode.

4.2.3. 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

Table 6
return Translation

```
// return : return from method {
  ldm R13!, {R11, R14, R10, R12};
  //This instruction is used to pop the top four values
  //in the stack into R11, R14, R10 and R12 respectively
  mov R13, R12;
  //Restore the stack frame pointer of the caller method
  mov CurrentThread->LastFramePointer, R10;
  mov R0, (fsize*4)+8;
  //fsize is the number of local variables of the current method
  //(fsize*4)+8) is the total frame size reserved in the stack
  //for a given method
  //fsize is multiplied by 4 because the addresses in the stack grow by 4
  mov PC, LP; //This instruction is used to return to the caller }
```

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 uses 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.

4.3. Garbage Collection

Our system allows native method calls and memory allocation during compiled 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 does not take into account the object 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 objects inside it. Object addresses are then changed and therefore, references to them are updated. Hence, the main issue here is to enhance the marking algorithm to scan both the Java and the native stacks and mark their live referenced objects.

This is done in our compiler by adding some features to the KVM garbage collection. Using C and ARM assembly language code, information about a thread native stack are gathered and passed to the marking process, which passes over the given native stack, scans it and marks all its live objects in the heap. Then, a switch to the sweep and compact functions of the KVM is performed. At the end of this mechanism, if necessary, native stack references are updated through a code added to the KVM functions responsible for reference updating.

4.4. Exception Handling

Exception handling is an important feature of the Java language, which has specific semantics to be respected [13]. Our dynamic compiler handles it by generating efficient code for the bytecode *athrow*, which is responsible for raising an exception. Indeed, additional ARM assembly code is also added to the functions that are called by *athrow* and new issues relevant to exception propagation are introduced. During compiled mode, the method that throws the exception is always compiled. However, the method that catches the exception can be either interpreted or compiled. In these 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 is used to locate the native instruction corresponding to the bytecode handling the exception. Once the handled native code is located, the compiled 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.

4.5. Threads

The technique that has been used in handling threads is inspired by a previous prototype of this compiler for Intel Architecture [6] and is still under enhancement. During interpretation, the KVM runs its original threads switch services, while during compilation, additional code is generated for bytecodes causing transfer control and contexts saving. During interpreted mode, the methods are executed on the Java stack, while during compiled mode, the generated code is executed on the native stack. In this context, the data structure representing the 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 switch between threads occurred. Figure 2 shows the new thread structure. Notice that R13Store and R11Store respectively carry the values of the top and the frame pointer of the native stack. A dedicated register is used to hold the time-slice value. Whenever this value reaches zero, a thread switch is triggered.

Our experience with the first prototype of Armed E-Bunny revealed that the designed switch (between interpreted and compiled modes) mechanism could be significantly enhanced. In the sequel, we present a technique that leverages Armed E-Bunny by establishing a synergy between dynamic compilation and efficient interpretation in order to come up with a fast and lightweight switch mechanism.

5. Synergy between Dynamic Compilation and Efficient Interpretation

It is crucial to design a system allowing a cooperation between an efficient interpretation mechanism together with a lightweight selective dynamic compilation. The proposed technique fits in this framework. The main idea is to generate a native threaded interpreter. This is a pool of code units, called codelets. Each codelet is a native implementation of a Java bytecode. This interpreter is generated at the start-up of the Java virtual machine. When a method is identified as a hotspot, it is dynamically compiled by a one-pass compiler like Armed E-Bunny. The compiler makes use of the interpreter codelets during the code generation process. This leads to a lightweight compilation mechanism that is appropriate in an embedded context and allows to improve the bidirectional switch mechanism designed in Armed E-Bunny.

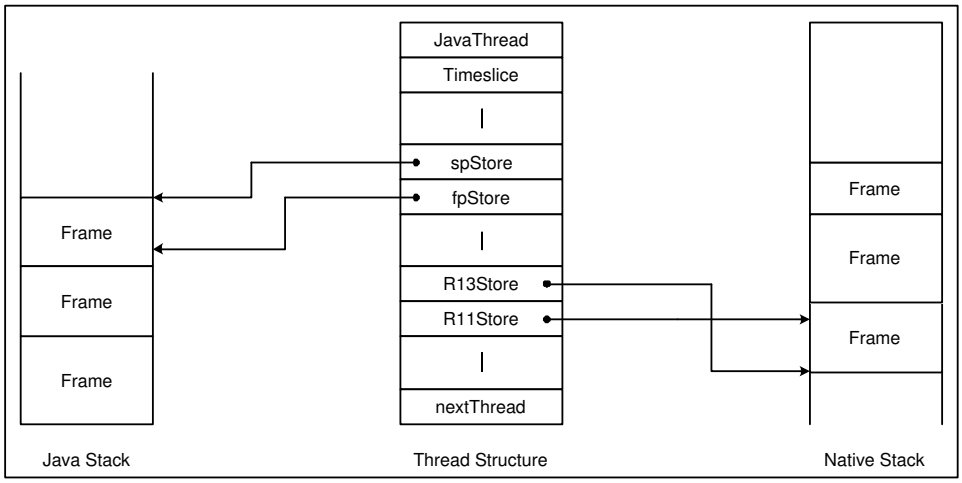

Figure 2. Thread Structure

Generating a native interpreter allows us to implement the Java stack on the native stack. This induces a fast switching between the interpreted and the compiled modes of execution since the frames of an interpreted method and a compiled one are on the same stack. The transfer of parameters between the Java and the native stacks is no more needed when switching from the interpreted mode to the native mode and vice versa. The technique we present in this paper is sustained by a smooth and uniform switching mechanism.

The closest work to the proposed technique is [15]. The authors presented a technique that combines interpretation with compilation to get a sort of hybrid interpretation strategy. This technique targets embedded systems and presents a code generation technique that leverages the interpreter self-code. The interpreter is written in the C language to achieve portability. This limits however the interpreter code reuse. Moreover, the compilation targets method fragments. The rationale underlying this choice is to reduce the size of the generated code. However, by doing so, the switch frequency between the interpreted and the compiled modes is increased and therefore an additional complexity and overhead are introduced. The method advanced by the authors requires: (1) The use of some kind of bytecode analysis to detect basic blocks; (2) The use of a peephole optimizer to improve the quality of the produced code. These two analyses lead to an additional overhead.

Our technique leverages also the interpreter code but it deviates from [15] in two directions:

- First, the interpreter code is efficiently generated and is not the result of a high-level compiler. This makes it suitable for reuse by a dynamic compiler to generate code by copy and concatenation.

- Second, the compilation unit in our technique is a method. By doing so, we reduce significantly the technical complexity of the switching mechanism and the underlying

overhead and frequency.

5.1. Native Threaded Interpreter Generation

Since the dynamic compiler reuses the codelets during the code generation process, these codelets have to be implemented in a way that facilitates code reuse. We distinguish two categories of bytecodes, context-free bytecodes and context-dependent bytecodes:

- A context-free bytecode can be translated to a native code that is independent of dynamic information. For instance, the bytecode `aload_0` is always translated to a `push` instruction of the 0^{th} local variable on the stack.
- A context-dependent bytecode requires some dynamic information (e.g ip: instruction pointer) to be translated to native code. For instance, the bytecode `iload` requires computing the index of a local variable, which is extracted from the bytecode stream using the instruction pointer.

A scan of the instruction set (without floats) of the Kilo Virtual Machine (KVM) proves that 95 bytecodes are context-free while the number of context-dependent bytecodes is 66. Hence, context-free bytecodes represent more than 61% of the standard bytecodes handled by the KVM.

The main motivation underlying our bytecode taxonomy is to confine the codelet reuse to context-free bytecodes. Actually, these bytecodes are very frequently used in Java applications as exemplified by [23]. According to the results of [23], it is easy to see that our context-free bytecodes correspond to the Loads, Stores and ALU categories, which are very frequently used. For instance, with respect to the SPECjvm98 benchmark [5], these categories represent respectively 35.54%, 6.65% and 9.94% of the bytecodes. Hence, the compilation process of performance-critical methods by copying the interpreter codelets instead of regenerating them is improved. This relies on the relatively high frequency of context-free bytecodes. In the sequel, we highlight the interpreter structure and components.

The present technique is based on a generated native threaded interpreter. The generation of the interpreter is a one-time virtual machine operation performed at the start-up. Basically, the interpreter may be considered as a pool of code units called codelets. Each codelet is an efficient native implementation of a bytecode. The implementation of this interpreter uses a data structure composed of a jump table and a codelet table. Each entry in the codelet table contains the generated native implementation of the corresponding bytecode.

The interpretation main loop is reduced to a jump into the codelet table via the jump table according to the current bytecode in the method under interpretation. In addition, each codelet ends up by dispatching to the next bytecode. This eliminates the traditional centralized bytecode dispatch overhead. Besides, the codelets are generated in a way that allows to reuse them in the code generation phase during a method compilation. Figure 3.a outlines the interpreter architecture.

The template of a codelet for a bytecode is as follows:

- Native code;

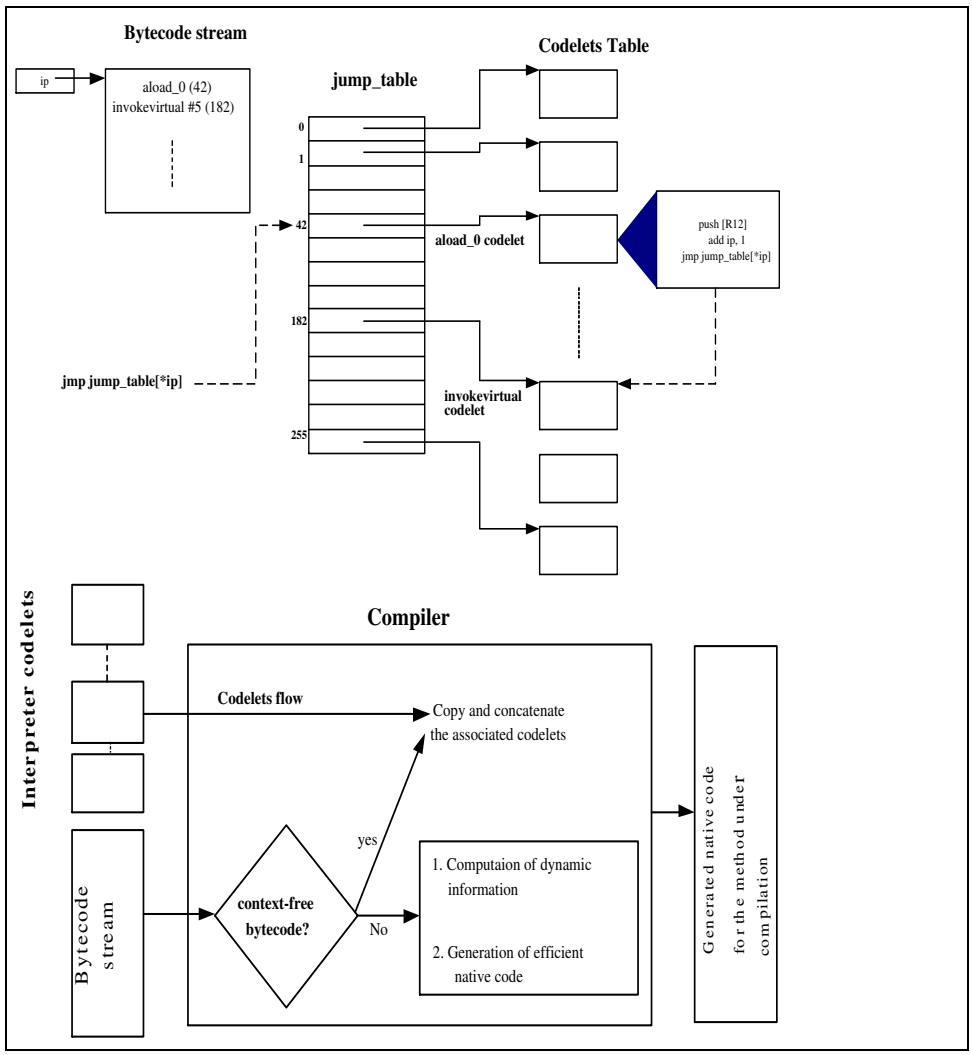

Figure 3. (a) Native Threaded Interpreter Generation (b) Interpreter Codelet Reuse

- Increment of the instruction pointer (ip);
- Jump to the entry corresponding the bytecode at the current ip.

Table 7 illustrates the codelet, in assembly-like pseudo-code, for the `aload_3` bytecode.

The algorithm of Table 8 depicts the interpreter codelets generation step. The start-up of the interpretation of a method is reduced to a jump to the codelet associated to the opcode of the first bytecode of this method. The codelets are generated and their respective addresses are stored in a jump table. The latter maps each opcode with the address of the corresponding codelet.

Table 7
Codelet for `aload_3`

```
push the 3rd local variable
add ip, 1
jmp jump_table[*ip]
```

Table 8
Interpreter Codelet Generation

```
generateInterpreter() {
  generate a jump instruction to jump_table[*ip]
  for each bytecode
    generate the corresponding native code
    insert the codelet address into the next entry of jump_table }
```

5.2. Reusing Codelets for Dynamic Compilation

Dynamic compilation occurs at run-time. The compilation overhead is then a critical issue particulary in the embedded context. Therefore, it is important to minimize the compilation time in order to reduce the overall execution time. The classical compilation techniques (flow analysis, aggressive optimizations etc.) produce a high code quality. They require however, huge data structures and consume time, which makes them unaffordable when targeting virtual machines that are meant to be embedded in resources-constrained devices.

The technique described within this paper aims at improving Armed E-Bunny. In fact, we avoid the systematic regeneration of native code each time a performance-critical method is detected. We achieve this goal by the reuse of already-generated code for the interpreter at the virtual machine start-up.

Context-free bytecodes are translated by a simple copy of the already generated codelets. Thus, we save the time spent in regenerating it. However, to avoid re-computing dynamic information such as instruction pointer values, constant values in the constant pool, as it is required by the interpretation process, the dynamic compiler computes them efficiently, once for all, for the method under compilation. Hence, context-dependent bytecodes are translated to native code using these dynamic information. Figure 3.b depicts the native code generation scheme used in this technique.

The algorithm outlined in Table 9 is executed for each bytecode in the method under compilation. When the opcode of the bytecode under compilation is context-free the interpreter codelet associated to this opcode is just copied and concatenated to the already generated code in a buffer. A special care is taken to discard the original dispatch code in each codelet. In the case of a context-dependent bytecode, the compiler generates an efficient code from scratch. The compiler uses the dynamic information available at run-time, such as the value of ip, to generate efficient native code. Section 5.4 gives more details on this issue.

Table 9
Compilation of a Bytecode

```
compile(bytecode) {
  if is_context_free(bytecode)
    copy jump_table[bytecode]
  else {
    compute dynamic information
    generate native code for the bytecode }}
```

5.3. Smooth Switching Mechanism

The technique, we propose, is supported by a smooth and uniform switching mechanism between the interpreted mode and compiled mode. In order to achieve this goal, we use a unique native stack per thread of control. In the sequel, we illustrate the stack layout that supports the switch mechanism and describe the switch implementation.

A straightforward translation approach would maintain a run-time stack and manipulates it the same way the interpreter does with the Java stack [10]. Consequently, the bytecode execution is based on two stacks (Java stack and native stack). The switch between the interpreter and native modes is much expensive. This is due to the unnecessary memory traffic between the two stacks.

We propose a design where the frames of an interpreted method and a compiled method for a thread are represented in the same native stack. This leads to a smooth and fast switch from the interpreted to the compiled mode and vice versa. Indeed, using a unique execution stack (native stack) saves the overhead induced by transfers of call parameters from the Java stack to the native stack. Besides, some registers are dedicated to hold some information. For instance, R12 register contains the parameters address whereas R14 register contains the instruction pointer (ip). Figure 4.a depicts the stack layout.

The use of a unique stack speeds up the switch between interpreted and compiled methods. However, it introduces new issues. In fact, we need to know the kind of calling method (interpreted or compiled) to restore the calling context.

Actually, returning to an interpreted method resumes the execution at the next opcode following the invocation bytecode whereas returning to a compiled method resumes the execution at the next native instruction following the call. Hence, the return address has different semantics in our design depending on the calling method type.

We propose a uniform mechanism allowing to restore smoothly the calling method context. This is based on the use of an artificial opcode and a specific codelet associated with it. When interpreting a method invocation bytecode (*invokeVirtual*, *invokeSpecial*, *invokeInterface*, *invokeStatic* etc.), the opcode of the following bytecode is pushed on the stack. In the case of a compiled method, the native return address and the artificial opcode are pushed on the stack. This artificial opcode allows to restore the context of the compiled calling method transparently without any explicit test.

Actually, the codelet associated with this artificial opcode is responsible for jumping

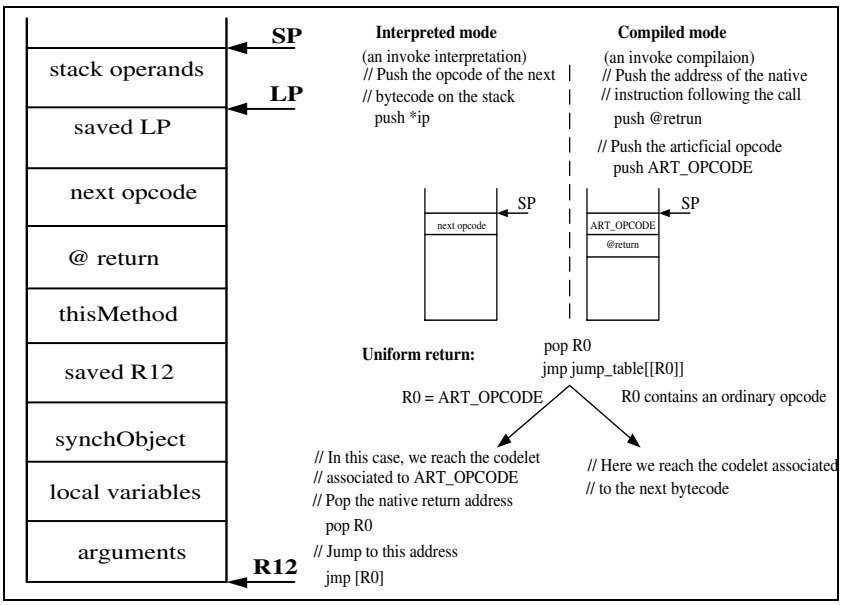

Figure 4. (a) The Stack Layout (b) A Lightweight Switch Mechanism

to the native address previously pushed on the stack as well. When returning to an interpreted method, a jump to the codelet associated with the already pushed opcode is performed. Figure 4.b outlines the implementation of this lightweight uniform switching mechanism.

The switch is implemented by means of two stubs. A prologue stub ensures saving the context of a calling method and setting the context of the called one. Reciprocally, an epilogue stub is responsible for restoring the context of the calling method. Tables 10 and 11 outline these stubs.

5.4. Scenario

We illustrate, in the sequel, the code generation technique based on interpreter codelets reuse to compile Java bytecodes. Table 12 illustrates the bytecode sequence under compilation.

Figure 5.a illustrates a snapshot of the code generation technique in action. The right-hand part of Figure 5.a depicts a snippet of the interpreter codelets generated at the virtual machine start-up whereas the left-hand part depicts how the different bytecodes are handled. The bytecodes `aload_1 (43)` and `iconst_3 (6)` belong to the context-free category mentioned above. The native code generation for these bytecodes is then reduced to a copy of the corresponding interpreter codelets as depicted in Figure 5.a. The bytecode `bipush 5`, is however a context-dependent bytecode. Indeed, an efficient compilation of this bytecode requires the actual current value of ip in order to access the index associated with the opcode `bipush` in the bytecode stream (method), which is 5 in the present example. This information is not available at the generation of the interpreter.

Table 10
Prologue

```
prologue() {
  sub SP, n*4; //leave space for n local variables
  push syncObject;
  //push the call receiver object on the stack
  push R12
  push thisMethod //push the method pointer
  push returnAddress; //push return address (ip or pc)
  // artificial opcode in case of compiled
  // or opcode of the next instruction
  push opcode
  push R11
  mov R11, SP }
```

Table 11
Epilogue

```
epilogue() {
  pop R11
  //artificial opcode or opcode of the next instruction
  pop opcode //return address (ip or pc)
  pop R14 //remove the method pointer from the stack
  add SP, 4
  pop R12 //remove the call receiver from the stack
  add SP, 4
  mov SP, R12
  jmp jump_table[opcode] }
```

Table 12
Bytecode Sequence under Compilation

```
aload_1
bipush 5
iconst_3
```

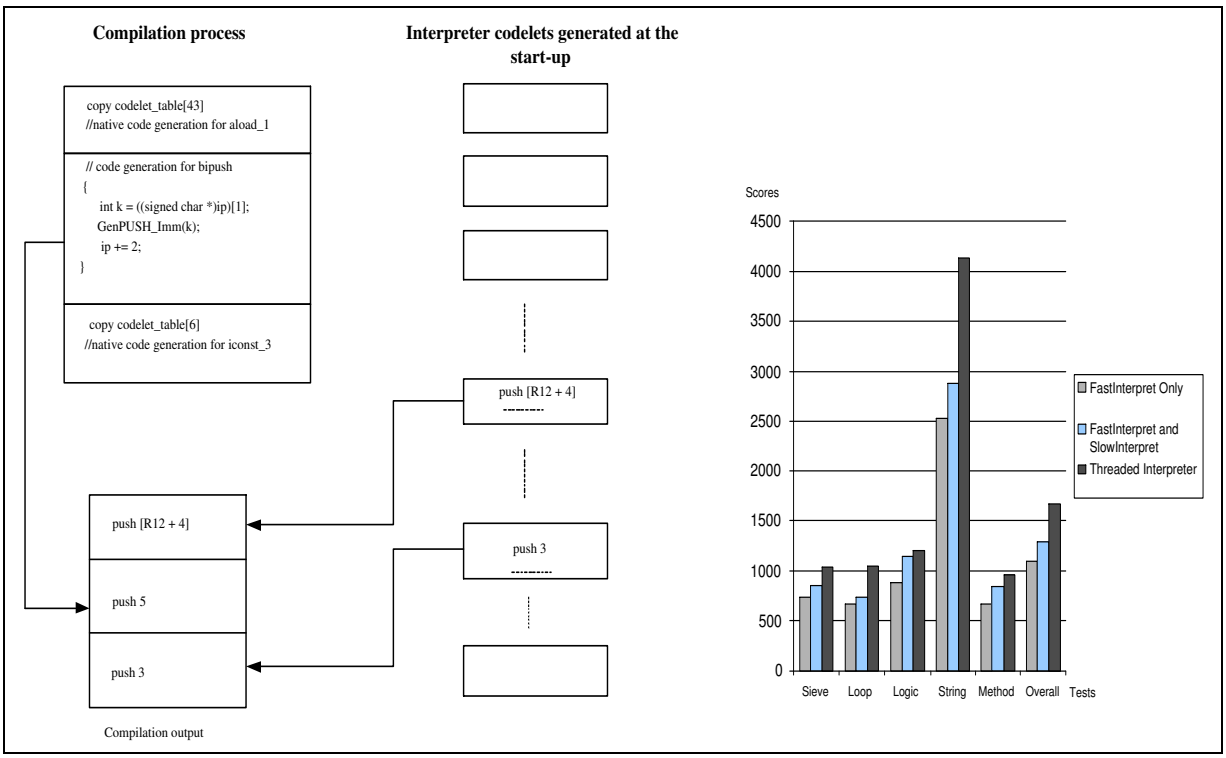

Figure 5. (a) A Scenario for compilation by reuse (b) Threaded Interpreter Performance

We recall that the interpreter generation is a one-time virtual machine operation, which occurs prior to the execution of any Java method. As a result, the codelet associated with `bipush` could not be reused.

The dynamic compiler, relying on the dynamic information available at run-time (value of ip for instance) generates an efficient code for context-dependent bytecodes from scratch. The value of the index (5) is extracted from the bytecode stream. The generated code is just a push of the latter value onto the stack: `push 5`. The interpreter codelet is inefficient because it contains the extra instructions required to access the value of the index then afterwards it has to push this value on the stack.

6. 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 for execution. Our results demonstrates that Armed E-Bunny requires additional memory space that does not exceed 119 KB, 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 with and without Armed E-Bunny. The results illustrated in Table 13, demonstrate that Armed E-Bunny produces

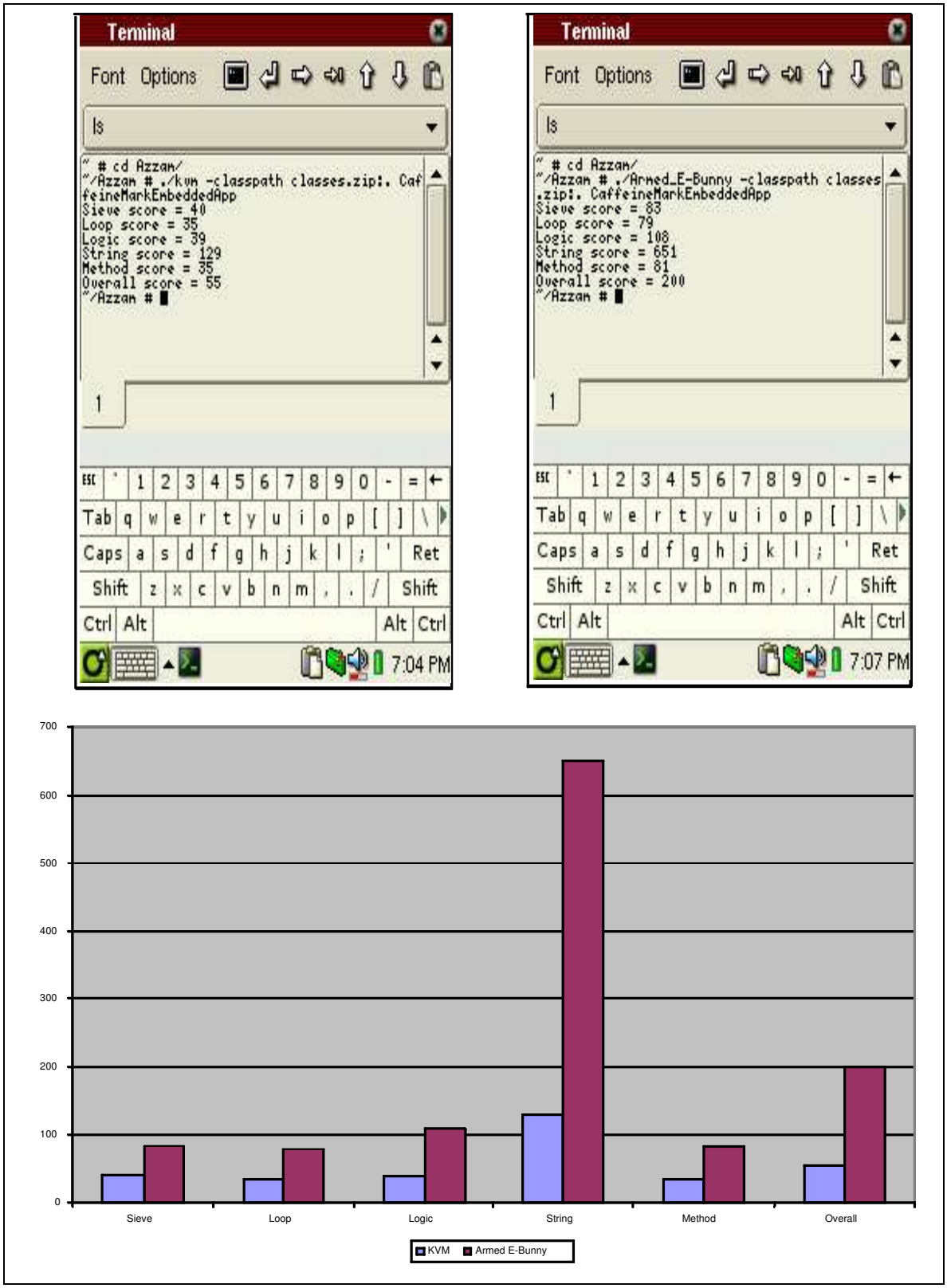

Figure 6. Caffeine Scores of the KVM and Armed E-Bunny

Table 13
Comparison of KVM and Armed E-Bunny Performance

	KVM 1.0.4	Armed E-Bunny	Speedup
Sieve Score	40	83	2.075
Loop Score	35	79	2.26
Logic Score	39	108	2.77
String Score	129	651	5.05
Method Score	35	81	2.32
Overall Score	55	200	3.64

an overall speedup of 360 % over the original KVM. Figure 6 shows a snapshot and a comparison chart of our tests on an IPAQ handheld that is powered with Embedded-Linux.

Moreover, we have implemented and integrated a threaded interpreter in the KVM. The experimental results obtained using the CaffeineMark benchmark, and highlighted in Figure 5.b, show an execution enhancement up to 53% with respect to the non-optimized KVM main-loop (*FastInterpret*) and 30% with respect to the optimized KVM main-loop (*FastInterpret* with *SlowInterpret*) thanks to the threaded interpreter technique.

7. Conclusion

We reported, an acceleration technology for Java embedded virtual machines that is based on selective dynamic compilation. This technology targets the J2ME/CLDC (Java 2 Micro Edition for Connected Limited Device Configuration) platform. We designed and implemented an efficient, lightweight and low-footprint accelerated embedded Java Virtual Machine. This has been achieved by the means of integrating a selective dynamic compiler, called Armed E-Bunny, into the J2ME/CLDC (Java 2 Micro Edition for Connected Limited Device Configuration) virtual machine KVM. We presented the architecture, the design as well as the technical issues of Armed E-Bunny and how we addressed them. Experimental results, on an IPAQ handheld powered with Embedded-Linux, demonstrated that we accomplished a speedup of 360% with respect to the Sun's latest version of KVM.

Moreover, we proposed in this paper, an acceleration technique that relies on an established synergy between efficient interpretation and selective dynamic compilation. Actually, efficient interpretation is achieved by a generated threaded interpreter that is made of a pool of codelets. The latter are native code units efficiently implementing the dynamic semantics of a given bytecode. Besides, each codelet carries out the dispatch to the next bytecode eliminating therefore the need for a costly centralized traditional dispatch mechanism. The acceleration technique described in this paper advocates the use of a selective dynamic compiler to translate performance-critical methods to native code. The translation process takes advantage of the threaded interpreter by reusing most of the previously mentioned codelets. This tight collaboration between the interpreter and the dynamic compiler leads to a fast and lightweight (in terms of footprint) execution of Java

class files. Currently, we are carrying out a research on other aspects of Java VM acceleration such as object layout, garbage collection and annotation-based acceleration. The anticipated speedup will be important in our opinion and the footprint will be reasonable.

REFERENCES

1. B. Alpern, C. Attanasio, J. Barton, M. Burke, P. Cheng, J. Choi, A. Cocchi, S. Fink, D. Grove, S. Hummel M. Hind, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. Russell, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.
2. A. Azevedo, A. Nicoleau, and J. Hummel. Java Annotation-aware Just-in-Time (AJIT) Compilation System. In *ACM Java Grande 99*, pages 142–151, San Francisco, California, USA, June 1999.
3. M. Cierniak, G. Lueh, and J. Stichnoth. Practicing JUDO: Java under Dynamic Optimizations. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*, pages 13–26, Vancouver, Canada, June 2000.
4. G. Comeau. Java Companion Processors versus Accelerators. `http://www.zucotto.com`, 2003.
5. Standard Performance Evaluation Corporation. Spec jvm98 benchmarks. http://www.specbench.org/osg/jvm98/.
6. 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, June 2004. ACM Press.
7. M. Ertl. A Portable Forth Engine. In *EuroFORTH '93 conference proceedings*, 1993.
8. E. Gagnon and L. Hendren. Effective Inline-Threaded Interpretation of Java Bytecode Using Preparation Sequences. In *Proceedings of Compiler Construction, 12th International Conference, CC 2003, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003*, volume 2622 of *Lecture Notes in Computer Science*, pages 170–184. Springer-Verlag, 2003.
9. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.
10. C. Hsieh, J. Gyllenhaal, and W. Hwu. Java Bytecode to Native Code Translation: The Caffeine Prototype and Preliminary Results. In *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture*, pages 90–99, Paris, France, December 1996.
11. ARM Limited. ARM7TDMI Data Sheet. Technical report, ARM Limited, 2001.
12. ARM Limited. ARM Developer Suite Assembler Guide. Technical report, ARM Limited, 2001.
13. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, 1996.
14. S. Majercik and M. Littman. Using Caching to Solve Larger Probabilistic Planning Problems. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98) and of the 10th Conference on Innovative Applications of Artificial Intelligence (IAAI-98)*, pages 954–960, Menlo Park, July 26–30 1998. AAAI Press.
15. G. Manjunath and V. Krishnan. A Small Hybrid JIT for Embedded Systems. *SIGPLAN Notices*, 35(4):44–50, April 2000.

16. Sun MicroSystems. The Java HotSpot Performance Engine Architecture. Technical report, Sun MicroSystems, California, USA, April 1999.
17. Sun MicroSystems. Java 2 Platform, Micro Edition, Version 1.0 Connected, Limited Device Configuration. Specification. Technical report, Sun Microsystems, California, USA, May 2000.
18. Sun MicroSystems. KVM Porting Guide. Technical report, Sun MicroSystems, California, USA, September 2001.
19. Sun MicroSystems. CLDC HotSpot Implementation Virtual Machine. Technical report, Sun MicroSystems, California, USA, 2002.
20. Nazomi. Bootsing the Performance of Java Software on Smart Handheld Devices and Internet Appliance. `http://www.nazomi.com`, 2003.
21. M. Paleczny, C. Vick, and C. Click. The Java HotSpot Server Compiler. In *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01)*, pages 1–12, Monterey, California, April 2001.
22. I. Piumarta and F. Riccardi. Optimizing Direct-threaded Code by Selective Inlining. In *In Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, 1998.
23. R. Radhakrishnan, N. Vijaykrishnan, L. Kurian John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java Runtime Systems: Characterization and Architectural Implications. *IEEE Transactions on Computers*, 50(2):131–146, 2001.
24. 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, August 2002.
25. T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal*, 39(1):175–193, February 2000.
26. B. Yang, S. Moon, S. Park, J. Lee, S. Lee, J. Park, Y. Chung, S. Kim, K. Ebcioglu, and E. Altman. LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation. In *IEEE PACT*, pages 128–138, California, USA, October 1999.