JOURNAL OF OBJECT TECHNOLOGY
Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2005

Vol. 4, No. 1, January–February 2005

E-Bunny: A Dynamic Compiler for Embedded Java Virtual Machines

Mourad Debbabi[†] **Abdelouahed Gherbi**[†] **Lamia Ketari**[†]
Chamseddine Talhi[†] **Nadia Tawbi**[‡] **Hamdi Yahyaoui**[†]
Sami Zhioua[†]

[†]Concordia Institute for Information Systems Engineering,
Concordia University, Quebec, Canada.
[‡]Computer Science Department, Laval University, Quebec, Canada.

A new acceleration technology for Java embedded virtual machines is presented in this paper. Based on the selective dynamic compilation technique, this technology addresses the J2ME/CLDC (Java 2 Micro Edition for Connected Limited Device Configuration) platform. The primary objective of our work is to come up with an efficient, lightweight and low-footprint accelerated embedded Java Virtual Machine. This is achieved by the means of integrating a selective dynamic compiler that we called E-Bunny into the J2ME/CLDC virtual machine KVM. This paper presents the motivations, the architecture, the design and the implementation issues of E-Bunny and how we addressed them. Experimental results on the performance of our modified KVM demonstrate that we accomplished a speedup of 400% with respect to the Sun's latest version of KVM. This experimentation was carried on using standard J2ME benchmarks.

1 MOTIVATIONS AND BACKGROUND

With the advent and rising popularity of wireless systems, there is a proliferation of small internet-enabled devices (e.g. PDAs, cell phones, pagers, etc.). In this context, Java is emerging as a standard execution environment due to its security, portability, mobility and network support features. In particular, J2ME/CLDC (Java 2 Micro-Edition for Connected Limited Device Configuration) [13] is now recognized as the standard Java platform in the domain of mobile wireless devices such as pagers, handheld PDAs, TV set-top boxes, appliances, etc. It gained big momentum and is now standardized by the Java Community Process (JCP) and adopted by many standardization bodies such as 3GPP, MEXE and OMA. Another factor that has amplified the wide industrial adoption of J2ME/CLDC is the broad range of Java based solutions that are available in the market. All these factors make Java and J2ME/CLDC an ideal solution for software development in the arena of embedded systems.

The heart of J2ME/CLDC technology is Sun's Kilobyte Virtual Machine (KVM) [14]. The KVM is a Java virtual machine initially designed with the constraints of

Cite this article as follows: Mourad Debbabi et al: "E-Bunny: A Dynamic Compiler for Embedded Java Virtual Machines", in *Journal of Object Technology*, vol. 4, no. 1, January–February 2005, pages 81–106,
http://www.jot.fm/issues/issues_2005_01/article2

low-end mobile devices in mind. The performance and the security of the KVM will be two key factors in the successful deployment of Java technology in wireless and embedded systems.

The primary intent of our research initiative is to dramatically improve the performance of J2ME/CLDC virtual machines such as the KVM.

Lately, a surge of interest has been expressed in the acceleration of Java virtual machines for embedded systems. Two main approaches have been explored: hardware and software acceleration.

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

For software acceleration, a large spectrum of techniques has been advanced [1, 5, 6, 7, 12, 16]. These techniques could be classified into 4 categories: general optimizations, ahead-of-time optimizations, just-in-time 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 %) [15].

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.

This paper presents an extremely lightweight selective dynamic compiler for embedded Java virtual machine called E-Bunny. It is built on top of KVM. The contributions are threefold:

- Our system is the first academic work that targets CLDC-based embedded Java virtual machines optimization by dynamic compilation. The remaining systems are commercial products such as [15] and [17].
- Our solution, besides the compilation of all kind of bytecodes, covers the different issues of the integration of a dynamic compiler into a virtual machine such as multi-threading support, exception handling, garbage collection, switching mechanism between the compiler and the interpreter modes, etc.
- Our solution is efficient. It allows to accelerate the performance by a factor of 4 while the memory footprint overhead does not exceed 138 KB.

The remainder of this paper is organized as follows. Section 2 highlights related work relevant to the dynamic compilation in the embedded context. In section 3, we present the architecture as well as the key ideas of our system. Design issues are detailed in section 4. Section 5 presents the results of our implementation and finally section 6 is a conclusion.

2 RELATED WORK

Dynamic compilation became a popular approach to optimize Java performance. Almost all standard Java virtual machines [1, 2, 12, 20, 19] are endowed with a dynamic compiler.

Java HotSpot VM [12], which is the core component of Java 2 Standard Edition, is equipped with a selective dynamic compiler. Performance critical methods are detected by means of a counter-based profiler with additional heuristics. These heuristics investigate the caller methods in order to compile them with the triggering method. On-Stack-Replacement is also used to trigger compilation while a method is still running. The Java Hotspot VM dynamic compiler applies several classical

optimizations and is considered as one of the most efficient in the market. Another important reason of this efficiency is the aggressive method in-lining it applies. Indeed, methods that are detected frequent are not only compiled but also in-lined. The benefit of this optimization is that it produces larger blocks of code for the compiler to perform optimizations. Operating on large blocks of code increases the effectiveness of classical optimizations. However, using On-Stack-Replacement or applying expensive optimizations such as aggressive method in-lining is not adequate for embedded systems because they require important resources particularly memory space.

IBM Jalapeño [1] is a Java virtual machine equipped with a Just-In-Time compiler with different levels of optimizations. That is, according to the frequency of a method, this is compiled with the appropriate level of optimization. The profiler used by Jalapeño is very complex and is part of a bigger system called Adaptive Optimization System (AOS). As Java Hotspot VM, the features of IBM Jalapeño cannot be applied in an embedded context due to lack of resources. For instance, a Just-In-Time approach is not suitable because it requires that all methods are compiled, whereas, embedded systems lack the necessary memory to store all method generated code.

Other Java virtual machines are endowed with dynamic compilers including IBM JDK [19], Intel ORP [1], IBM Mixed-Mode-Compiler [20], Latte [21] and OpenJIT [11].

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 [15]. 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 [15]. This leads to several cache misses that decreases the program performance.

Despite these difficulties, dynamic compilation is also used in CLDC-based em-

bedded virtual machines [15, 17, 18]. However, except one paper about KJIT [18], no detailed information about these systems is available in the literature due to commercial reasons.

KJIT [18] 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. This strategy seems to be very heavyweight and only feasible in server or desktop systems. The key idea to make this strategy adequate for embedded Java virtual machines is to compile only a subset of bytecodes. The remaining bytecodes continue to be handled by the interpreter. Indeed, whenever one of the interpreted bytecodes is encountered, execution switches back from the compiled mode to the interpreted mode. This requires an efficient handling of the switching mechanism since this operation is highly frequent. This is achieved in KJIT by pre-processing the bytecode 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 [15] is an embedded virtual machine introduced by Sun Microsystems. As its name indicates, it is strongly inspired by the standard Java Hotsopt VM. All features of Java Hotspot VM that can be adapted to resource-constrained environments are applied. Among these features, we find a selective dynamic compiler. Performance critical methods are detected by a single statistical profiler. The compilation is performed in one pass. Three basic optimizations are applied: constant folding, constant propagation and loop peeling. 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 ARCHITECTURE

E-Bunny is a selective dynamic compiler for embedded Java virtual machines that uses as its foundation the KVM. In this section, we present the key features that make E-Bunny an appropriate Java acceleration technology for embedded systems. The major features of E-Bunny are:

- **Reduced Memory Footprint**: The footprint resulting from the integration of the E-Bunny dynamic compiler does not exceed 150 KB. The key idea to reduce the code size of E-Bunny is to merge the compilation processing of some bytecodes. This is possible because several bytecodes have joint processing (e.g. *invokespecial*, *invokevirtual*). This strategy is applied mainly for some bytecodes that have fast versions [9](e.g. *getstatic*, *getstatic_fast*, *getstaticp_fast*, *getstatic2_fast*).

- **Selective Compilation**: Since selective dynamic compilation is the most

adequate compilation-based acceleration technique for embedded systems, it was adopted in E-Bunny. Only a subset of methods is compiled. The methods are selected according to their invocation frequencies. The unit of compilation is exclusively a method.

- **Efficient Stack-Based Code Generation**: For the compilation strategy, a trade-off has to be made between the compilation cost and the generated code quality. Although a register-based code is more efficient, we do not generate such code because it requires more passes over the bytecode. In E-Bunny, we generate a stack-based code because it requires only one pass over the bytecode. Thus, a one-pass code generation strategy is adopted, without using neither intermediate representations nor heavyweight optimizations. Only optimizations that might be applied in one pass are allowed.

- **Using two stacks**: Portability is an important issue in embedded systems. Virtual machines with dynamic compilers are strongly machine-dependent because they compile the bytecode into the target machine native code. Thus, it is unavoidable to become dependent on the target machine and consequently less portable. CLDC Hotspot [15] and IBM Mixed-Mode [20] virtual machines, which adopt a mixed-mode approach, use only the native stack for both interpretation and compiled methods execution. However, the native stack is part of the target machine architecture. This makes the virtual machine very dependent on the target machine. Our dynamic compilation approach advocates the use of two stacks (one for interpretation and one for compiled methods execution) to preserve portability to a high extent. Hence, interpretation remains totally machine-independent. Moreover, the platform-specific code is isolated in specific modules. Future porting of E-Bunny to another platform requires the modification or the re-creation of only these modules. However, the drawback of this way to preserve portability is the complexity introduced by the use of two stacks. Indeed, (1) method arguments must be transferred between the two stacks when necessary, (2) the garbage collector must consider the native stack when it scans the heap and (3) additional context information must be added to the thread data structure.

- **Multi-threading Support**: Another challenge introduced by dynamic compilation is the multi-threading support. Conventionally, each thread has its own execution stack. Since our approach uses two kind of stacks, each thread is assigned two stacks upon its creation: a Java stack to interpret methods and a native stack to run compiled ones. For the Java stack, we adopt the KVM way to manage the Java stacks. KVM allocates the Java stack from the heap and manipulates it at a software level. For the native stack, the approach adopted is to organize the native stack as a pool of segments and allocate a segment to each thread. Thus, the native stack will be shared by all living threads.

- **LRU Algorithm for Cache Management**: A limited memory space is

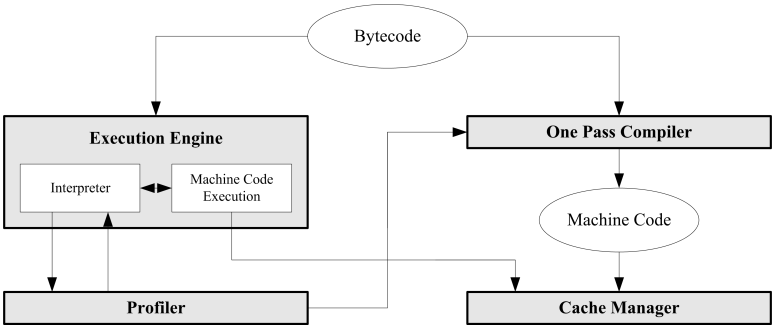


Figure 1: E-Bunny Architecture

allocated to the compiled code. When this space is full, a cache strategy based on a Least Recently Used (LRU) algorithm [10] is adopted to free the necessary space.

The E-Bunny architecture is depicted in Figure 1. It includes four major components: the execution engine, the profiler, the compiler and the cache manager. Initially, all invoked Java methods are interpreted. During interpretation, a counter-based profiler gathers profiling information. As the code is interpreted, the profiler identifies hotspot methods. Once a method is recognized as hotspot, its bytecodes are translated into native code by the compiler. The produced native code is stored in the dynamic compiler cache. On future references to the same method, the cached compiled method is executed instead of interpreting it. In the sequel, we highlight the main components of the proposed embedded dynamic compiler architecture.

Interpreter

By interpreter, we mean the KVM's interpreter. It is the only way in KVM to execute Java programs. The interpreter is basically a loop that fetches, decodes and interprets bytecodes of a given method. E-Bunny adopts a mixed-mode[1] execution approach. Therefore, the interpreter cooperates with the machine code execution component to switch between the two modes: interpreted and native.

Machine Code Execution Component

The machine code execution component is responsible for invoking compiled methods. Basically, it looks for the corresponding machine code in the cache and then executes it. In addition, this component is responsible for transferring, if any, methods arguments from one stack to the other. More details about this mechanism are given in Section 4. When the machine code of a given method is executed, the

[1]Execution overlaps between interpretation mode and native code execution mode.

cache must be set up-to-date according to the cache algorithm. The machine code execution component triggers the cache update.

Profiler

A simple counter-based profiler is used in E-Bunny. A per-method counter is incremented each time the method is invoked. A method is considered as hotspot when its counter reaches an a-priori defined threshold. Consequently, a compilation request is triggered. Our profiling strategy assumes that every method called by a compiled method must be compiled. So, a compiled method cannot invoke an interpreted method. The motivation behind this strategy is to reduce the complexity of the switch mechanism between interpreted and native modes.

Compiler

The compiler is the core component of E-Bunny. It is triggered by the profiler. Basically, it compiles Java methods to machine code. The E-Bunny compiler is extremely lightweight. It goes through the bytecode in one pass and generates a stack-based code. Unlike traditional compilers, neither intermediate representations nor heavyweight optimizations are used. The compilation strategy is described in Section 4.

Cache Manager

Once a method is recognized as hotspot and compiled, the corresponding machine code needs to be stored in the heap. In E-Bunny, a fixed space, called the cache, is pre-allocated in the heap to store the generated code. The cache is pre-allocated in the permanent space of the heap to avoid any conflicts with the garbage collection mechanism. Conventionally, the garbage collector does not scan the permanent space. When the cache becomes full, the cache manager must select elements to remove in order to free space for newly compiled methods. For this purpose an LRU algorithm is used. This algorithm selects the method (or the methods) that has (have) not been invoked for the largest period of time and removes it (or them) from the cache. A queue is used to keep the chronological order of invoked methods. This queue is updated each time a compiled method is invoked. The LRU algorithm does not prevent methods from being recompiled several times, but our experiments show that using an LRU algorithm reduces efficiently the re-compilation rate.

4 DESIGN

In this section we discuss the design issues of E-Bunny. First, we detail the compilation strategy. Second, we focus on a delicate aspect of selective dynamic compilation which is the switch mechanism between interpreted and compiled modes. Then, we illustrate how E-Bunny supports multi-threading. Finally, we describe the interaction with the garbage collection mechanism.

Compilation Strategy

Our compilation strategy spans over a lightweight one pass compilation technique. This strategy avoids complex computations performed by common compilers and generates a code of reasonable quality. Indeed, the generated code is stack-based as Java bytecode but uses many information computed at the compilation step (field offsets, Constant Pool entry address etc.). These information are grafted in the generated code in order to avoid unnecessary further re-computation.

Compiling a method goes through three steps. First, generating context saving instructions (the prologue). Second, translating bytecodes into machine code instructions. Third, generating context restoration instructions (the epilogue). The second step, which is the core step of our compilation strategy, consists in translating each bytecode into a sequence of native instructions. We distinguish two categories of bytecodes. The first kind includes bytecodes that are completely translated into native instructions in a straight manner. They are called: simple bytecodes. The second kind of bytecodes are more complex to translate and are called complex bytecodes. E-Bunny targets Intel IA-32 architecture [8]. IA-32 is a 32 bit CISC machine which provides eight general purpose registers (EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP). EBP and ESP are dedicated to stack management. The remaining registers fulfill different other tasks. Hence, compiling a method consists in generating assembly instructions that reproduces the bytecode behavior on an IA-32 platform.

Context Saving and Context Restoration

Context saving and context restoration instructions are used to re-establish the calling method context after the execution exits from the callee method. Context saving instructions figure on top of the method generated code. They carry out three basic operations. First, save old EBP register value (of the calling method) on the native stack. Second, assign a new value to EBP register (of the callee method). Third, assign the new value of EBP to LastEBP field of the current thread structure (CurrentThread->LastEBP =EBP). Actually, LastEBP is a new field added to the thread data structure in E-Bunny. It gives for each thread the EBP value of the last called compiled method. This information is used to two purposes: garbage collection and exception handling. Context saving instructions

```
//saving old value of EBP on the native stack
push EBP
//setting the new value of EBP
mov EBP, ESP
//CurrentThread->LastEBP = EBP
mov EAX,&CurrentThread->LastEBP
mov [EAX], EBP
```

Table 1: Context Saving

```
//restore the old value of EBP register
mov ESP, EBP
pop EBP
//CurrentThread->LastEBP = EBP
mov EAX,&CurrentThread->LastEBP
mov [EAX], EBP
```

Table 2: Context Restoration

in E-Bunny are illustrated in Table 1.

Context restoration instructions figure on the end of method generated code. Since all Java methods finish by a return bytecode (ireturn, lreturn, areturn or return), context restoration instructions are generated when translating return bytecodes. Context restoration instructions carry out two operations. First, restore the old value of EBP register (saved on the native stack). Second, assign this value to LastEBP field of CurrentThread data structure. Consequently, when returning to the calling method, EBP register and CurrentThread->LastEBP are set to the appropriate values. Context restoration instructions in E-Bunny are illustrated in Table 2.

Simple Bytecodes Translation

The simple bytecode category includes loads (e.g. iload, iaload, ldc), stores (e.g. astore, lastore), stack manipulation (e.g. pop, dup), arithmetic, logic and shift (e.g. iadd, land, ishr, l2i) and branching (e.g. ifne, if_icmpeq, goto) bytecodes. These

ILOAD (0x15)

Description : Load integer from local variable

Format : ILOAD <index>

Corresponding Assembly Instructions :

```
push [EBP + 4 * ( <FrameSize> - <index> + 3)]
```

Table 3: iload Bytecode Translation

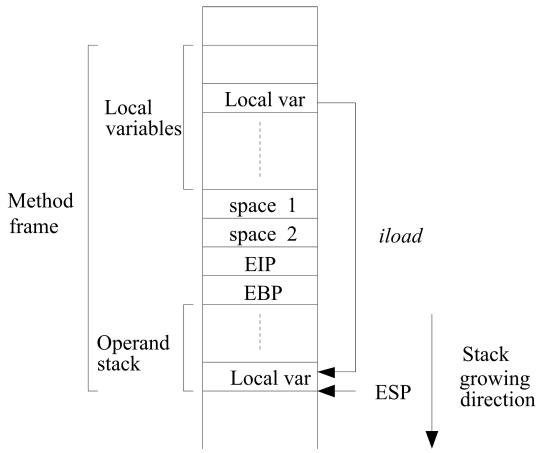


Figure 2: iload on the Native Stack

bytecodes are directly translated into stack-based machine code which reproduces the interpreter behavior on the native stack. As an example, we give the translation details of two bytecodes iload and ifeq. We have chosen iload bytecode to show how it behaves on the native stack and ifeq bytecode to illustrate how E-Bunny translates forward branching instructions in one pass.

The iload bytecode loads integer (32 bit value) from local variables to the top of the stack (stack operand). It is directly mapped to a push instruction as shown in Table 3. The symbols "[ ]" denote the content of the specified location. The FrameSize variable is an integer that gives the number of local variables of the current method. The displacement is multiplied by 4 because addresses in Intel [2] grow by 4. The value 3 in the offset of the push instruction denotes EIP register and two empty spaces (space 1 and space 2) as illustrated in Figure 2. Notice that space 1 and space 2 are left for the returned value of the method (see Section 4).

An important issue of the one pass translation strategy is how to deal with control flow bytecodes (or branching bytecodes). In the sequel, we highlight this issue and how it is processed in E-Bunny.

Control flow bytecodes are translated using a bytecode map table called *BCNativeMap*. It is an integer table that maps each bytecode index to its corresponding machine instruction index. For instance, $BCNativeMap[5] = 31$ means that the machine instructions corresponding to the fifth bytecode start at index 31 in the *NativeCode* table. A *BCNativeMap* table is associated with each compiled method. It is initialized to zero, and filled progressively when the translation of the method is going on. At a given moment of translation, the *BCNativeMap* table is filled for the scanned bytecodes and empty (zero values) for the rest.

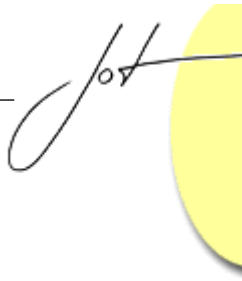
To deal with control flow bytecodes, there are two situations: forward branches

[2]The terms Intel stack and native stack are used interchangeably.

IFEQ (0x99)
Description : Branch if equal to zero
Format : IFEQ <offset>
PseudoCode :
If (*BCNativeMap*[currentBytecode + offset] == 0)
//forward branching//
Generate Jump instruction without operand
Leave space for the operand
Else *//backward branching//*
Generate jump instruction
Compute offset operand
Generate operand
Endif
Translate next bytecodes ...
When reaching the target bytecode:
Compute the current offset
Fill the space left before by the computed offset
Corresponding Assembly Instructions :
pop EAX
cmp EAX, 0
jz <computed machine code offset>

Table 4: ifeq Bytecode Translation

and backward branches. The translation of backward branches is straightforward. *BCNativeMap* table is used to get the target machine instruction index, and then to generate the corresponding jump native instruction. However, the translation of forward branches is more complex. Indeed, the corresponding *BCNativeMap* entry of a forward branch is not resolved yet (contains zero) as the target bytecode is not already compiled. The complete translation of the forward branching bytecode is postponed until the target bytecode is reached. Actually, a jump native instruction op-code (e.g. JL, JNE, JMP) is generated with an unresolved operand. Each next bytecode is checked whether it is a target of an unresolved branching. In such case, the corresponding offset is computed and then used to patch the unresolved operand of the jump native instruction. To handle multiple unresolved forward branching, a linked list is used. In Table 4, we give the translation algorithm of ifeq bytecode and the corresponding machine instructions.



PUTFIELD (0xb5)

Description : Set field value in object

Format : PUTFIELD <cp_index>

Function used: Putfield()

Pre-conditions:

The following items must be on the top of native stack:

- ip (bytecode ip)
- field value
- objectref.

Post-conditions:

Set EAX to the fileld size:

1 when the field needs one memory word

2 when the field type is two memory words

Corresponding Assembly Instructions:

```
//push the ip on the Intel stack
push ip
mov ECX, &Putfield
call ECX
    //popping the putfield arguments
    //Switch// ECX
    1 : add ESP, 12
    2 : add ESP, 16
```

Table 5: putfield Bytecode Translation

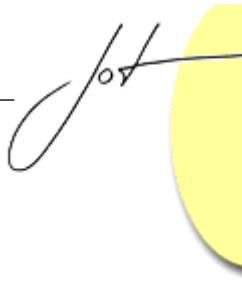
Complex Bytecodes Translation

In KVM, interpretation of some bytecodes involves the use of some virtual machine runtime services such as method lookup or field reference resolution. For example, getfield bytecode uses field reference resolution subroutine to resolve the reference of the appropriate field. In E-Bunny, bytecodes that require virtual machine services are called complex bytecodes. Translating this kind of bytecodes is more complex. A possible approach consists in generating the corresponding native code, instruction by instruction, including virtual machine services. This yields a complex and a very bulky code. The approach adopted in E-Bunny is to define for each bytecode a specific function that calls necessary virtual machine services. These functions are called from the native code. Hence, the generated machine code is compact and less complex. Complex bytecodes category includes field access, objects creation, array manipulation, method invocation, return, monitor, casting and exception bytecodes. In the sequel we illustrate the translation mechanism of field access bytecodes and exception bytecode (athrow).

Field access bytecodes are putfield, getfield, putstatic, getstatic and their fast versions. These bytecodes access to object fields using symbolic references, to get or set their values. Such symbolic references have to be resolved so that the access could be possible. In E-Bunny, for each bytecode a specific function is defined. For getfield and getstatic, this function calls the field reference resolution subroutine and returns the value of the field in EAX register (EAX:ECX for long values). This value is then pushed on the native stack. For putfield and putstatic, this function calls the field reference resolution subroutine, sets this field and returns the field size in the EAX register. This value is then used to update the native stack. The translation of putfield is given in Table 5. The cp_index denotes a Constant Pool index. The "Switch" operator is used to clarify the illustration. Actually, it is implemented with jump instructions.

Method invocation bytecodes require a special care because they involve more complex processing than other bytecodes. Furthermore they are performance critical (on average 11% of total bytecodes are method invocation bytecodes [16]). The processing that method invocation bytecodes should carry out includes: performing a method reference resolution, checking the validity of the receiver object, performing a dynamic method lookup, creating a new frame for the invoked method, etc. Moreover, these bytecodes deal with the switch mechanism between interpreted and compiled modes (more details about the switch mechanism are given in next section).

In E-Bunny, to translate invoke bytecodes, specific functions have been defined (callVirtual for invokevirtual and invokespecial, callStatic for invokeStatic and callinterface for invokeinterface). These functions behave differently according to the kind of the called method (compiled or native). The conventions adopted to design these functions are the following: If the invoked method is native, the defined functions resolve the invoked method reference, check the validity of the receiving object, make the method lookup, transfer the arguments form the native stack to the Java stack (because the native functions consider the Java stack to get the arguments), invoke the native functions and finally transfer returned value to Java stack. In addition, these functions, set EAX register to 1 (to distinguish the native function call from the compiled method invocation) and EDX register to an integer value necessary to update the native stack.



INVOKEVIRTUAL (0Xb6)

Description : Invoke instance method

Format : INVOKEVIRTUAL <cp_Index>

Function used: callVirtual()

Pre-conditions:

The following items must be on the top of the Intel stack:

- ip (bytecode's ip)
- one empty word

Post-conditions:

IF (called method is a C (native) function)

- EAX = 1
- The returned value is on the native stack
- EDX = integer value to update the native stack.

ELSE *//called method is a compiled java method*

- EAX = 0
- [ESP] = local variables number
- EDX = machine code address.

ENDIF

Corresponding Assembly instructions :

```
sub ESP,4       //leaving the first space
push ip         //pushing ip (and the second space)
mov EAX, &callVirtual
call EAX
  //Switch// EAX
  1 : add ESP, EDX        //updating the native stack
  0 : mov EAX, [ESP]      //getting space for locals
      sub ESP, EAX        //leaving space for locals
      call EDX            //launching the native code
      add ESP, EAX        //updating the native stack
     POP_FRAME_FROM_NATIVE
```

Table 6: invokevirtual Bytecode Translation

ATHROW (0xbf)

Description : throws exception or error

Format : ATHROW

Function used: throwExc()

Pre-conditions:

The following item must be on top of the Intel stack:

- exception object reference

Post-conditions:

- jumps to the exception handler if any
- raise a virtual machine exception otherwise.

Corresponding Assembly Instructions :

```
mov EAX, &throwExc
call EAX
```

Table 7: athrow Bytecode Translation

If the invoked method is compiled, these defined functions resolve the invoked method reference, check the validity of the receiving object, make the method lookup, get the machine code from the cache, update the latter and exit. We note here that the defined function does not call the invoked method. Instead, the machine code address of the called method and the local variables number are returned respectively in EDX register and in the top of the stack. These values are used to prepare the invocation. After that, the method is invoked. When it exits, the method context information is discarded using the value computed by the translated return bytecode. As an illustration of this mechanism, the translation of the invokevirtual bytecode is given in Table 6.

Another important feature of the Java language is the exception handling mechanism. An exception is raised by the athrow bytecode. A major constraint that the dynamic compilation should respect is to preserve exception handling semantics. The dynamic compilation introduces new issues relevant to exception propagation. Indeed, a compiled method could propagate an exception to an interpreted method. E-Bunny handles the following situations:

- The method where the exception is thrown and the one where the exception is caught, are both interpreted: the original virtual machine exception handling mechanism is used.
- The method where the exception is thrown, is interpreted and the one where the exception is caught is a compiled method. This case is excluded since the adopted profiling strategy assumes that a compiled method cannot invoke an interpreted one.
- The method, where the exception is thrown, is a compiled method. There are

ALGORITHM 1 : Interpreter to native switch

Step1 : Transfer arguments

```
for (i <= NbrArguments)
    push arg_i
```

Step2 : Reserve space for local variables

Step3 : Reserve two words for the returned value

```
sub ESP, 8
```

Step4 : Call the compiled method generated code

```
call Method_Machine_code
```

Table 8: Interpreted to Native Switch Algorithm

two possible situations: the method catching the exception is either interpreted or compiled. In order to locate the method handling the exception, first, *BCNativeMap* is used to identify the bytecode corresponding to the native instruction throwing the exception. Then, an exception handler lookup is performed. If the method catching the exception (the method containing the handler) is a compiled one then its *BCNativeMap* is used to locate the native instruction corresponding to the bytecode handling the exception. A jump to this instruction is performed. Otherwise, we switch to the interpreted mode at the level of this bytecode.

The generated code for the athrow bytecode calls a specific function called throwExc. This function uses the virtual machine exception handling functionality to locate the exception handler. Then, it resumes the execution at the appropriate location. The translation of athrow is illustrated in Table 7.

Switch Mechanism

In E-Bunny, compiled methods are executed in the native stack while interpreted methods are interpreted in the Java stack located in the heap. Hence, execution of Java programs overlaps between native stack and Java stack. The switch between the two modes implies context transferring between the two stacks. We distinguish two situations where the switch occurs between the two modes: interpreted to native and native to interpreted.

The interpreted to native switch occurs when interpreting an invoke bytecode and the invoked method happens to be compiled. Coming from the interpretation mode, the called method arguments are on the top of the Java stack. The switch to the native mode requires their transfer to the native stack. This is carried out according to the algorithm in Table 8. The algorithm is based on the method frame layout in the native stack, depicted in Figure 3.

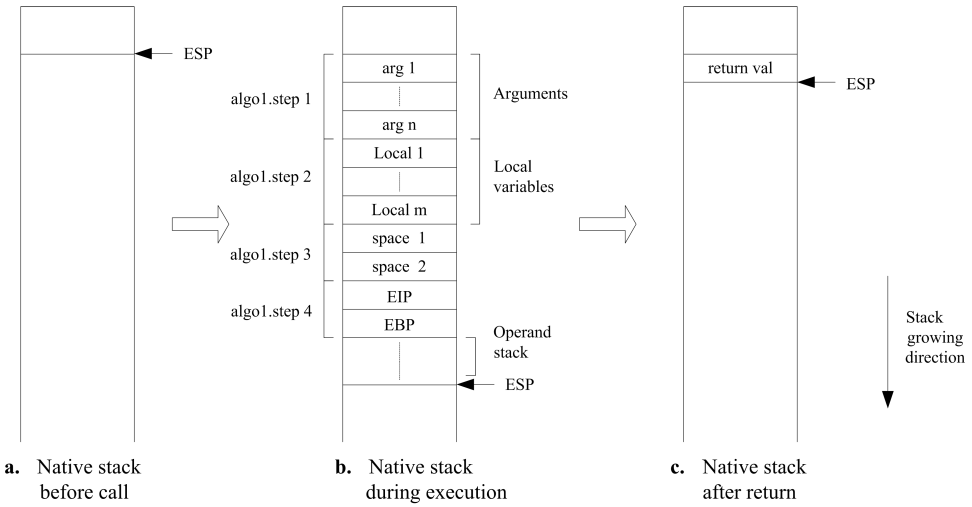


Figure 3: Switch Mechanism to the Native Mode

The switch from the native mode to the interpreted mode occurs in two situations. First, when a compiled method calls an interpreted method. Second, when a compiled method exits and returns back to its interpreted caller method. The profiling strategy we adopt assumes that every method called by a compiled method should be compiled. The switch is then reduced only to the second situation (return case). Handling this switch consists in transferring the returned value, if any, from the native stack to the Java stack. The implementation of this action depends on the returned value type as illustrated in Table 9.

Figure 3 shows the native stack in different phases of the switch. First, before method call (a). Second, after algorithm 1 (b). Third, when the translated called method exits (c). In (c), we assume that the called method returns a one word size value (e.g. integer, short, reference, etc.).

Threads Management

A Java virtual machine provides a framework to run properly different threads. Each thread has its own stack. In the interpreted mode, these stacks are created and managed at a software level. Basically, thread stacks are allocated in the heap. However, in E-Bunny, since two stacks are used, compiled methods have to be run on a native stack. Consequently, threads should use the native stack.

In the current implementation, the native stack is organized as a pool of segments. A segment is assigned to a thread when the latter is created. The segment pool management is based on a bit map. An entry of this map is a bit indicating whether the corresponding segment is used or free. Therefore, each thread executes its compiled methods in its own segment. A consequence of managing several threads with two stacks is that each thread has two forms of context information.

ALGORITHM 2 : Native to Interpreter return
Switch (returned value type size) :
0 (void)
Nothing
1 (integer, reference, etc.)
pop *value*
pushStack(*value*)
2 (long)
pop *lowHalf*
pop *highHalf*
pushStack(*highHalf*)
pushStack(*lowHalf*)

Table 9: Native to Interpreted Switch Algorithm

The first is relevant to Java stack (e.g. sp: stack pointer, fp: frame pointer, lp: locals pointer) and the second is relevant to the native stack (ESP, EBP and EIP registers). The data structure representing the thread in the virtual machine holds information representing both contexts (Figure 4).

Thread scheduling in the KVM is based on a round robin scheduling model. Each thread keeps control during a time-slice. This is decremented after each bytecode that may cause a control transfer (e.g. branching and invoke bytecodes). When the time-slice becomes zero, the virtual machine stops the current thread and resumes the next one in the running threads queue. Method compilation requires the support of thread scheduling. To achieve this purpose, additional code is generated for bytecodes causing transfer control. This code, mainly, decrements the ESI register, dedicated to hold time-slice value, and triggers a thread switch when ESI reaches the NULL value. In addition, a special care to save both contexts relevant to Java and native stacks, is taken.

Garbage Collection Issues

KVM garbage collection is based on a mark-sweep with compaction algorithm. With the selective approach of E-Bunny, compiled methods are executed on the native stack. Consequently, the native stack may contain some object references. Since the current garbage collection algorithm scans only the heap, the native stack will not be considered, and then, object references on it neither will be marked nor updated. Therefore, the current garbage collection algorithm is inefficient with a selective approach.

The garbage collection algorithm has to be extended to deal with the native stack. Precisely, translated method frames in native stack have to be scanned in order to

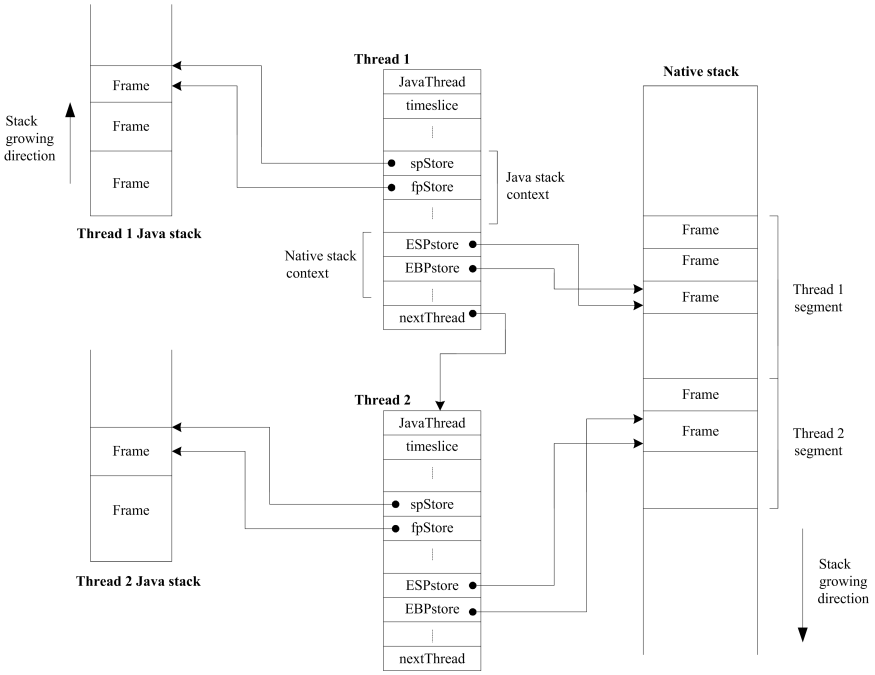


Figure 4: Multi-threading in E-Bunny

mark and update object references. In E-Bunny, the garbage collection algorithm is enhanced to address this issue. Mainly, the garbage collection functionalities are modified to take into account the object references in the native stack. Indeed, for both marking and updating loops, we check if the frame corresponds to a compiled method or not. If the method is compiled we consider the native stack, otherwise we consider the Java stack.

5 IMPLEMENTATION AND RESULTS

E-Bunny is implemented using the C programming language. In our experiments, we used the GNU compiler to build the latest version of KVM (KVM 1.0.4) with E-Bunny. Table 10 shows the executable size of KVM with and without E-Bunny. The first column gives the total executable footprint of KVM without E-Bunny. The second column gives the total executable footprint of KVM equipped with E-Bunny dynamic compiler. Finally, column 3 of the table shows that using GCC to build KVM with E-Bunny produces a footprint overhead of 64 KB. To summarize, E-Bunny requires 64 KB for executable footprint overhead, 64 KB for storing translated methods and 10 KB for a map between bytecodes and native instructions which is used for compiling control flow instructions and for exception handling mechanism. Hence, the total memory resources required by E-Bunny dynamic compiler is 138 KB.

To evaluate the performance of E-Bunny, we have run CaffeineMark benchmark

KVM	E-Bunny	Footprint Overhead
144K	208 K	64 K

Table 10: Executable File Footprint overhead

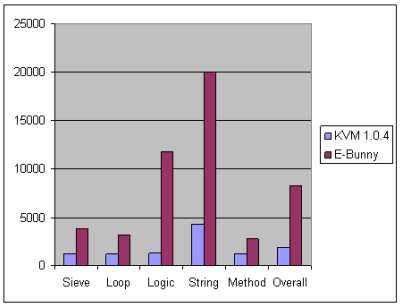


Figure 5: CaffeineMark Scores of KVM 1.0.4 and E-Bunny with GCC

(without the float test) on the original version of KVM with and without E-Bunny.

E-Bunny produces an overall speedup of 4 over original KVM 1.0.4 (Overall score in Figure 5). Moreover, we built the MIDP 2.0 profile, intended to CLDC devices, using E-Bunny and we ran successfully several midlets. Figure 6 shows a snapshot of MIDP emulator illustrating CaffeineMark midlet results.

6 CONCLUSION AND FUTURE WORK

We reported, a new 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 E-Bunny, into the J2ME/CLDC virtual machine KVM. We presented the motivations, the architecture, the design as well as the technical issues of E-Bunny and how we addressed them. Experimental results demonstrated that we accomplished a speedup of 400% with respect to the Sun's latest version of KVM.

Currently, many enhancements of E-Bunny are in progress. The major one concerns the bidirectional smooth switching between the interpreted and compiled modes. In fact, the profiling strategy adopted in the current version of E-Bunny, which consists in compiling each method called from the compiled one, is less complex to implement. However, it presents a drawback since it leads to compile non-performance-critical methods. On the other hand, a more efficient centralized thread scheduling is being implemented. This is expected to reduce the generated native

Figure 6: CaffeineMark midlet ran by KVM 1.0.4 and by E-Bunny

code size.

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 Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, February 2000.

[2] 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.

[3] G. Comeau. Java Companion Processors versus Accelerators. http://www.zucotto.com, 2002.

[4] Nazomi Communications. Boosting the performance of Java Software on Smart Handheld Devices and Internet Appliance. http://www.nazomi.com, April 2002.

[5] T. Cramer, R. Friedman, T. Miller, D. Seherger, R. Wilson, and M. Wolczko. Compiling Java Just in Time. *IEEE Micro*, 17(3):36–43, 1997.

[6] 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 Euro-*

pean 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.

[7] C.-H. A. Hsieh, J. C. Gyllenhaal, and W. W. Hwu. Java Bytecode to Native Code Translation: the Caffeine Prototype and Preliminary Results. In IEEE, editor, *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture, December 2-4, 1996, Paris, France*. IEEE Computer Society Press, 1996.

[8] Intel. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*. Intel Corporation, California, USA, order number 245470 edition, 2000.

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

[10] S. M. Majercik and M. L. 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.

[11] F. Maruyama. OpenJIT 2: The Design and Implementation of Application Framework for JIT Compilers. In USENIX, editor, *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM'01): April 23–24, 2001, Monterey, California, USA. Berkeley, CA*, Berkeley, CA, USA, 2001. USENIX.

[12] Sun MicroSystems. The Java HotSpot Performance Engine Architecture. Technical report, Sun MicroSystems, California, USA, April 1999.

[13] Sun MicroSystems. Connected, Limited Device Configuration. Specification Version 1.0, Java 2 Platform Micro Edition. Technical report, Sun MicroSystems, California, USA, May 2000.

[14] Sun MicroSystems. KVM Porting Guide. Technical report, Sun MicroSystems, California, USA, September 2001.

[15] Sun MicroSystems. CLDC HotSpot Implementation Virtual Machine. Technical report, Sun MicroSystems, California, USA, 2002.

[16] R. Radhakrishnan, N. Vijaykrishnan, L. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan. Java Runtime Systems: Characterization and Architectural Implications. *IEEE Transactions on Computers*, 50(2):131–146, 2001.

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

[18] 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.

[19] 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, 2000.

[20] T. Suganuma, T. Yasue, M. Kawahito, H. Komatsu, and T. Nakatani. A Dynamic Optimization Framework for a Java Just-in-Time Compiler. *ACM SIGPLAN Notices*, 36(11):180–195, November 2001.

[21] B. Yang, S.M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y.C. Chung, S. Kim, K. Ebcioğlu, and E. Altman. LaTTe: A Java VM just-in-time compiler with fast and efficient register allocation. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99)*, pages 128–138, Newport Beach, California, October 12–16, 1999. IEEE Computer Society Press.

ABOUT THE AUTHORS

Mourad Debbabi holds Ph.D. and M.Sc. degrees in computer science from Paris-XI Orsay, University, France. He published more than 60 research papers in international journals and conferences on computer security, formal semantics, mobile and embedded platforms, Java technology security and acceleration, cryptographic protocol specification, design and analysis, malicious code detection, programming languages, type theory and specification and verification of safety-critical systems. He is a Full Professor and the Associate Director of the Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada. He is the Specification Lead of four Standard JAIN (Java Intelligent Networks) Java Specification Requests (JSRs) dedicated to the elaboration of standard specifications for presence and instant messaging through the Java Community Process (JCP) program. In the past, he served as Senior Scientist at the Panasonic Information and Network Technologies Laboratory, Princeton, New Jersey, USA; Associate Professor at the Computer Science Department of Laval University, Quebec, Canada; Senior Scientist at General Electric Research Center, New York, USA; Research Associate at the Computer Science Department of Stanford University, California, USA; and Permanent Researcher at the Bull Corporate Research Center, Paris, France. He can be reached at debbabi@ciise.concordia.ca. See also http://www.ciise.concordia.ca/~debbabi.

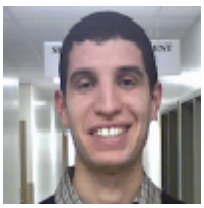
Abdelouahed Gherbi received his Engineering degree in Computer Engineering in 1992 and his Master degree in 1997, from the University of Constantine, Algeria. He is a Ph.D. student in the Electrical and Computer Engineering Department at Concordia University in Montreal. The main topic of his research activities is the acceleration of embedded Java virtual machines. He can be reached at gherbi@ciise.concordia.ca.

Lamia Ketari is a researcher at CSA (Computer Security and Acceleration) research group at Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada. She holds an MBA in Management Information Systems form Laval University. Currently, she is pursuing a Ph.D. thesis on the acceleration of Java for wireless devices and semantic-based optimization correctness. In the past, she worked on malicious code detection in COTS (Commercial Off-The-Shelf) applications. More specifically, she worked, with the collaboration of other research group members, on the implementation of a tool for monitoring critical system resources for malicious code detection, called: DaMon (Dynamic Analysis Monitoring). She can be reached at ketari@ciise.concordia.ca.

Chamseddine Talhi is a researcher at CSA (Computer Security and Acceleration) research group at Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada. He holds an M.Sc. degree from Constantine University, Algeria. He is pursuing a Ph.D. thesis on the security of embedded Java platforms. He is interested in characterizing enforceable security policies in resource-constrained platforms. In the past years, he worked on formal specification and verification of telecommunication services. He participated in the design and the implementation of a Java optimizing compiler for J2ME/CLDC. He can be reached at talhi@ciise.concordia.ca.

Nadia Tawbi holds a Ph.D and M.Sc. degrees from Pierre et Marie Curie University, Paris France. She was affiliated to Bull SA research center from 1989 to 1996. Since this year she joined the computer science department of Laval University where she is now an associate professor. Her research interests include static analysis of programs, optimization, object oriented languages, formal verification and security. She can be reached at Nadia.Tawbi@ift.ulaval.ca.

Hamdi Yahyaoui is a researcher at CSA (Computer Security and Acceleration) research group at Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada. He holds an M.Sc. degree from Laval University. He is pursuing a Ph.D. thesis on the acceleration and semantic foundations of embedded Java virtual machines. In the past years, he worked on formal dynamic semantics and control flow analysis of Java. He participated in the design and implementation of a dynamic optimizing compiler for J2ME/CLDC. He can be reached at hamdi@ciise.concordia.ca.



Sami Zhioua is a Ph.D. student at Concordia Institute for Information Systems Engineering, Concordia University, Montreal, Quebec, Canada. The main topic of his research activities is the acceleration of Java in the context of embedded systems. The goal of his research is to design and implement new techniques in order to improve the performance of embedded Java virtual machines. He participated in the design and implementation of a dynamic optimizing compiler for J2ME/CLDC. He can be reached at zhioua@ciise.concordia.ca.