Accelerating Embedded Java for Mobile Devices

Mourad Debbabi, Azzam Mourad, Chamseddine Talhi, Hamdi Yahyaoui
Computer Security Laboratory,
Concordia Institute for Information Systems Engineering,
Concordia University,
Montreal, Quebec, Canada.

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

***Index Terms*— Java, J2ME, CLDC, KVM, Selective Dynamic Compilation, Embedded Devices.**

I. Motivations and Background

With the advent and rising popularity of wireless systems, there is a proliferation of small Internet-enabled embedded devices (e.g. PDAs, cell phones, pagers, etc.). In this context, Java is emerging as a standard execution environment due to its security, portability, mobility and network support features. In particular, J2ME/CLDC (Java 2 Micro-Edition for Connected Limited Device Configuration) [9] is now recognized as the de facto execution engine in the domain of mobile wireless devices such as pagers, handhelds, TV set-top boxes, appliances, etc. J2ME/CLDC gained a big momentum and is now standardized by the Java Community Process (JCP) and adopted by many standardization bodies such as 3GPP and OMA. Another factor that has amplified the wide industrial adoption of J2ME/CLDC is the broad range of Java based solutions that are available in the market. All these factors make J2ME/CLDC an ideal solution for software development in the arena of embedded devices.

The heart of J2ME/CLDC technology is Sun's Kilobyte Virtual Machine (KVM) [10]. The KVM is a Java virtual machine initially designed with the constraints of low-end mobile devices in mind. These limited configuration devices do not have the same hardware capabilities as desktop and server systems. The latter have fast busses, hundred of megabytes of RAM and fast microprocessors operating at over 2 GHz with on-chip 512 kilobytes caches. Alternatively, limited configuration devices typically have memory less than 512 kilobytes to handle the entire Java runtime environment, 300 kilobytes of RAM, 1 megabytes of flash and ROM, microprocessors operating at over 32 MHz and low battery capacity. These power and memory restrictions create challenging issues for the deployment of Java technology on limited configuration devices. More precisely, the performance and the security of the KVM are two key factors in the successful deployment of this technology. Lately, a surge of interest has been expressed in the acceleration of Java virtual machines for limited configuration devices. Two main approaches have been explored: hardware and software acceleration.

For hardware acceleration, several companies (Zucotto Wireless [2], Nazomi [12], etc.), have proposed Java processors that execute Java bytecodes in silicon. Although these hardware accelerators achieve a significant speedup, their use comes with a high price in terms of power consumption. This energy issue is a serious drawback especially in the case of limited configuration devices. Moreover, the cost (royalties, licensing, etc.) of these hardware acceleration technologies is an additional obstacle to their adoption by the industry. These drawbacks created an interesting but challenging niche for software acceleration of embedded Java virtual machines.

For software acceleration, many techniques have been advanced [1], [6], [8]. They could be structured into 3 categories: General optimizations, ahead-of-time optimizations and dynamic compilation. General optimizations consist of designing and implementing more efficient virtual machine components (better garbage collector, fast threading system, accelerated lookups, etc.). Ahead-of-time optimizations consist of using static analysis (flow analysis, annotated type systems, and abstract interpretation) to optimize programs before execution. Dynamic compilation consists of compiling, at runtime, fragments of Java executables. This dynamic compilation is achieved by a compiler that is generally embedded into the virtual machine. The compiler is in charge of translating bytecodes into the native code of the host platform. Experience has demonstrated that general and ahead-of-time optimizations can lead to reasonable accelerations. However, they cannot compete with dynamic compilation in reaching large speedups (e.g. an acceleration of more than 200 %) [11]. This makes dynamic compilation a more appealing acceleration technique. In what follows, we detail the principles underlying the different forms of dynamic compilation together with their advantages and disadvantages.

The rest of the paper is organized as follows. Section II is dedicated to dynamic compilation. Section III presents embedded Java virtual machines. Finally, concluding remarks are given in Section IV.

II. Dynamic Compilation Techniques

Different forms of dynamic compilation could be distinguished according to the following three questions:

- What to compile?
- When to compile?
- How to compile?

The first question deals with the unit of the compilation process (a program, a class, a method, a code fragment, etc.). The second question addresses the issue of when the compilation process is triggered (when a class is loaded, when a frequently called fragment is detected, when a threshold is exceeded, etc.). The third question deals with the inner workings and nature of the compilation process (one-pass, multi-passes, optimizing compilation, extensive static analysis, register allocation, etc.). Generally, three types of dynamic compilation are distinguished: Just-In-Time (JIT) dynamic compilation, Dynamic Adaptive Compilation (DAC), and Selective Dynamic Compilation (SDC). We detail hereafter each of these approaches.

A. Just-In-Time Compilation

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

B. Dynamic Adaptive Compilation

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

C. Selective Dynamic Compilation

Selective Dynamic Compilation (SDC) deviates from the other techniques by selecting and compiling, on the fly, only those fragments of the class files that are frequently executed (hotspots). For instance, a method whose number of invocations exceeds a certain threshold will be declared as a hotspot. This is generally done by a profiler. The hotspot method is then compiled to native code, which is going to be executed upon future invocations of the method in question. By doing so, significant 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 the reduction in memory overhead since only a part of a program is converted to native code. This makes selective dynamic compilation more adequate for limited configuration devices. However, special care should be given to obtaining a lightweight implementation (low footprint) of the SDC components (profiler, compiler, etc.).

III. Dynamic Compilation in Embedded Virtual Machines

In the setting of embedded systems, dynamic compilation should cope with two major challenges: memory and power consumption. The stringent limitation of memory resources in limited configuration devices makes heavyweight code optimizations unaffordable. In addition, such optimizations are generally achieved by slow compilers that are costly in power consumption. Besides, the compiler should have a small footprint in order to fit in the memory budget of limited configuration devices.

Despite these difficulties, dynamic compilation is being extensively used in several CLDC-based embedded virtual machines [11], [13], [14]. Apart from one paper about KJIT [14], no detailed information on these systems is available in the literature. Hereafter, we present the most known embedded Java virtual machines that are endowed with dynamic compilers.

A. CLDC Hotspot

CLDC Hotspot [11] is an embedded Java virtual machine introduced by Sun Microsystems. As the name indicates, it is strongly inspired by the standard Java Hotspot VM [8]. All the features of Java Hotspot VM that can be adapted to an embedded environment are deployed. More precisely, CLDC Hotspot is endowed with a selective dynamic compiler. Frequently called methods are detected by a single statistical

profiler and their compilation is done in one-pass. Moreover, the compiler performs three basic optimizations: Constant folding, constant propagation, and loop peeling. The memory space required by CLDC Hotspot is almost double of the space required by the KVM. In fact, it reaches 1 megabyte. Besides, CLDC Hotspot is said to be 10 times faster than the KVM. No more technical details are provided about the CLDC Hotspot dynamic compiler.

B. Jbed Micro Edition CLDC

Jbed is a Java virtual machine for limited configuration devices [13]. Jbed is proposed by the company Esmertec and is intended to IntelXscale and Strong ARM architectures. The dynamic compiler of Jbed is a small one-pass compiler. It compiles all the classes at the loading time instead of waiting for their first execution. Then, it links the compiled classes into the application. Consequently, the execution delay is reduced. Experimental results show that Jbed virtual machine is 4 times faster than KVM.

C. KJIT

KJIT is a lightweight dynamic compiler that has been integrated into KVM [14]. KJIT does not use any form of profiling for the simple reason that all the loaded methods are compiled. This strategy seems to be heavyweight and only feasible in server or desktop systems. The key idea to make this strategy adequate for embedded Java virtual machines is to compile only a subset of method bytecodes while the remaining bytecodes are handled by the interpreter. This requires a switching mechanism between the compiler and the interpreter. In fact, whenever one of the interpreted bytecodes is encountered, a switch back from the native to the interpreted mode is needed. KJIT performs such a switching by pre-processing the bytecodes before their compilation. However, there is a significant overhead, in terms of time and memory space, to perform this pre-processing. For instance, the pre-processed code is 30% larger than the original one. KJIT implementation speeds up the execution by a factor ranging between 5.7 and 10.7.

D. JBlend

Aplix Corporation proposed the JBlend platform [3], which includes a high-performance, small-footprint Java Virtual Machine for CLDC configuration. Actually, JBlend virtual machine is based on CLDC Hotspot acceleration technology. In terms of deployment, JBlend is a leading Java platform in the market of handsets with more than 50 millions (2004 statistics) virtual machines running on mobile devices. JBlend offers both interpreted (CLDC based) and more recently compiled (CLDC Hotspot based) solutions. The interpreted version achieves, according to a private communication with Aplix engineers, a speedup that is between 2 and 4. The CLDC Hotspot solution achieves a speedup that is up to 10 times. However, it is worth to mention that there is no technical/academic publication that provides the inner workings of these industrial acceleration technologies.

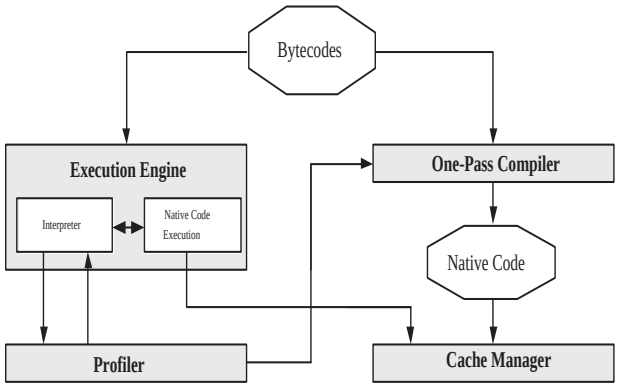

Fig. 1. Armed E-Bunny Architecture

E. Armed E-Bunny

In this section, we present a new acceleration technology that we developed for embedded Java virtual machines [4], [5]. Acceleration is achieved by the integration of a new selective dynamic compiler that we called Armed E-Bunny into the KVM. The modified KVM is ported on a handheld PDA that is powered with Embedded Linux.

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

Many features like reduced memory footprint and efficient use of both native and Java stacks make Armed E-Bunny an appropriate Java acceleration technology for limited configuration devices. The footprint of the compiler does not exceed 119 KB. Furthermore, the use of two stacks (one for interpretation and one for dynamic compilation) allows to preserve the portability of the virtual machine. However, the main drawback of this strategy is its complexity. In fact, a method's related data (e.g. arguments) should be transferred between the Java and the native stacks when necessary. Since ARM architecture specifies a technique for subroutine calls that rely on the registers more than the stack, it is mandatory, each time a method is called, to transfer the needed data from the stack to the registers and vice versa.

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

Profiling: The profiler of Armed E-Bunny performs a simple check over the frequency of calls to a method in order to identify it as a hotspot or not. If a method is recognized as a

hotspot, a switching from the interpreted to the native mode is applied. Otherwise, the interpreter continues its execution. In order to perform this check, a counter is added to the structure of the method and is updated each time the method is called. 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:

- Invokes the compiler to translate the method, then executes its corresponding generated code. This is if its counter has reached the threshold and it has not been already compiled, or,
- Executes the method's generated code if it is already compiled, or,
- Continues the interpretation of the corresponding method.

When one of the first two cases is chosen, all the method parameters together with the needed information are transferred from the Java to the native stack before execution. The results are then pushed back into the Java stack after finishing the execution of the generated code.

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

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

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

Threads Management: The technique that is used in handling threads is inspired by a previous E-Bunny prototype for Intel Architecture [4]. During interpretation, KVM runs its original threading switch services, while during compilation, additional code is generated for bytecodes to cause control transfer between threads. In the interpreted mode, methods are executed using the Java stack, while during native mode, the generated code is executed using the native stack. In this context, the data structure representing a thread in the virtual machine must hold information about both Java and native stacks in order to handle switching issues. These structures are updated each time a switching between threads occurs.

Exception Handling: Exception handling is an important aspect of the Java language, which has specific semantics to be respected [7]. Our dynamic compiler takes this aspect into consideration by generating efficient code for the bytecode *athrow*, which is responsible for raising an exception. Indeed, additional ARM assembly code is added to the virtual machine functions that are called by *athrow*. The main intent of such code is to handle the issue of exception propagation in the presence of two execution modes. In fact, during the native mode, the method that throws the exception is compiled, while the method that catches the exception can be either interpreted or compiled. This requires careful handling of these two situations. The added code solves this issue as follows: If the method catching the exception is compiled, the additional added code is used to locate the native instruction corresponding to the bytecode handling the exception. Once the handled native code is located, the native mode continues and a jump to the native instruction is executed. Otherwise, a switching to the interpreter is performed in order to apply the KVM built-in exception handling process.

Cache Management: The compiled code is saved in a particular cache structure that resides in the permanent space

TABLE I
COMPARISON OF KVM AND ARMED E-BUNNY PERFORMANCE

Tests	KVM 1.0.4	Armed E-Bunny	Speedup
Sieve Score	40	83	2.07
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

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

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

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

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

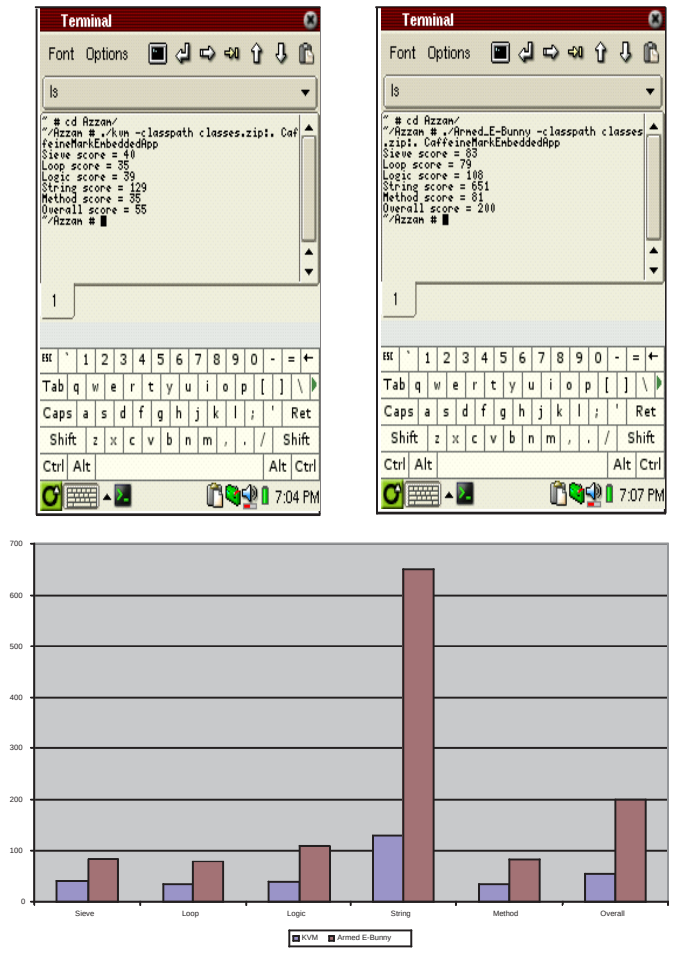

Fig. 2. Caffeine Scores of the KVM and Armed E-Bunny

and a comparison chart of our tests on an IPAQ handheld that is powered with Embedded Linux.

IV. CONCLUSION

In this paper, we presented different acceleration techniques for embedded Java virtual machines. These techniques belong to two main categories: Hardware and software accelerations. Hardware acceleration techniques allow to achieve a significant enhancement of virtual machine performance. However, the high power consumption and the cost of the underlying hardware processors compel researchers to resort to software acceleration of embedded Java virtual machines. General optimizations, static compilation and dynamic compilation are in general the three categories of software acceleration techniques. The most prominent software acceleration technique is dynamic compilation. This technique achieves high speedup rates of Java virtual machines for desktop and server systems. However, many issues should be solved when it comes to limited configuration devices. In fact, due to the limitations in terms of power and memory in limited configuration devices, a dynamic compilation technique should establish a tradeoff between code quality and compilation time. In this context, we succeeded to design and implement an efficient, lightweight and low-footprint selective dynamic compiler for J2ME/CLDC

targeting ARM architectures. The experimental results demonstrate that an important speedup (more than 360% over the last version of Sun's KVM) is accomplished with a footprint that does not exceed 119 KB.

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] G. Comeau. Java Companion Processors versus Accelerators. `http://www.zucotto.com`, 2003.

[3] Aplix Corporation. The JBlend Java Platform. `http://www.aplixcorp.com/products/jblend.html`.

[4] 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.

[5] M. Debbabi, A. Mourad, and N. Tawbi. Armed E-Bunny: A Selective Dynamic Compiler for Embedded Java Virtual Machine Targeting ARM Processors. In *Proceedings of the 2005 ACM Symposium on Applied Computing (SAC)*, Santa Fe, USA, March 2005. ACM Press.

[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 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.

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

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

[9] Sun MicroSystems. Java 2 Platform, Micro Edition, Version 1.0 Connected, Limited Device Configuration Specification, May 2000. White Paper.

[10] Sun MicroSystems. KVM Porting Guide. "`http://java.sun.com/products/cldc/wp/KVMwp.pdf`", September 2001. White Paper.

[11] Sun MicroSystems. CLDC HotSpot Implementation Virtual Machine, 2002. White Paper.

[12] Nazomi. Bootsing the Performance of Java Software on Smart Handheld Devices and Internet Appliance. `http://www.nazomi.com`, 2003.

[13] K. Schmid. Esmertec's Jbed Micro Edition CLDC and Jbed Profile for MID, Spring 2002.

[14] 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.

[15] Pendragon Software. The CaffeineMark Benchmark Version 3.0. `http://www.pendragon-software.com`, 1997.