Accepted in Proceedings of the 3rd International Symposium on Principles and Practice of Programming in Java, 2004
URL: http://dl.acm.org/citation.cfm?id=1071565&CFID=720746752&CFTOKEN=71946228

A Synergy Between Efficient Interpretation and Fast Selective Dynamic Compilation for the Acceleration of Embedded Java Virtual Machines

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

[†]Concordia Institute for Information Systems Engineering,
Concordia University, Quebec, Canada.
debbabi@ciise.concordia.ca,
gherbi@ece.concordia.ca, zhioua@ece.concordia.ca

[‡]Computer Science Department, Laval University, Quebec, Canada.
{lamia,talhi,hamdi}@ift.ulaval.ca

Abstract

We propose, in this paper, a technique for the acceleration of embedded Java virtual machines. The technique 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.

Keywords: Java, Virtual Machine, Acceleration, Performance, J2ME/CLDC, KVM, Threaded Interpretation, Selective Dynamic Compilation, Embedded Systems, Code Reuse.

1. MOTIVATIONS AND BACKGROUND

The ultimate objective of the Java Technology is to allow the development of efficient and secure cross-platform software. The scope of the underlying platforms covered by Java goes from powerful systems such as servers to resources-limited devices such that PDAs, cell phones, pagers and house appliances. In order to cope with the different requirements of this large range of platforms, Sun Microsystems offers three adequate platforms: J2EE (Java 2 Enterprise Edition) for servers [Shannon 2001], J2SE (Java 2 Standard Edition) for desktop workstations [Sun 2003a] and J2ME (Java 2 Micro-Edition) for small devices [Sun 2000].

The Java virtual machine (JVM) [Lindholm and Yellin 1996] is the cornerstone of the Java Technology. The JVM is traditionally interpreter-based. It uses an interpreter that emulates the execution of Java bytecodes on a specific platform. While the main advantages of the interpretation mechanism are simplicity and portability, its severe drawback remains definitely its poor performance.

Enhancing the Java virtual machine performance is a very active research area. Different approaches were and are being considered. Some of the proposed techniques put the emphasis on the acceleration of the interpretation mechanism while other ones introduce a certain form of compilation (ahead of time (AOT), just-in-time (JIT), dynamic adaptive compilation (DAC)) to improve the overall performance of a Java virtual machine.

Embedding a Java virtual machine into resource-constrained devices or systems poses very challenging but interesting problems in terms of footprint, computation and energy consumption. These three factors stand in the way of the acceleration techniques requiring huge data structures and complex computations, and hence increasing energy consumption.

We successfully conducted an active research initiative on the acceleration of the J2ME/CLDC [Sun 2000]

platform. In particular, we elaborated, designed and implemented a selective dynamic compiler, called E-Bunny, on top of the Kilobyte Virtual Machine (KVM) [Sun 2003b]. E-Bunny is a lightweight, small-footprint selective dynamic compiler that yields an acceleration ratio of 400% over KVM version 1.04 with respect to the CaffeineMark benchmark [Pendragon 1996]. To leverage the obtained results,we explore 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 described in this paper advocates the use of a fast one-pass 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.
The remainder of this paper is organized as follows. Section 2 is devoted to a presentation of the related work. Section 3 is dedicated to the presentation of the main features of the technique described in this paper. Section 4 presents the main features of a generated threaded native interpreter as required by our acceleration technique. Section 5 focuses on the the reuse, by a one-pass selective dynamic compiler, of the interpreter codelets. Section 6 presents a smooth switching mechanism between the interpreter and the compiler modes. Section 7 shows a scenario that illustrates the technique. Some concluding remarks together with a discussion of future work are ultimately sketched as a conclusion in Section 8.

2. RELATED WORK

The most appealing features of the Java language [Gosling et al. 1996] are its portability and security. However, the revers of the medal is its poor performance. Lately, a surge of interest has been expressed in the acceleration of JVMs for embedded systems. A wide spectrum of software acceleration techniques has been advanced [Alpern et al. 2000; Sun 1999]. These techniques could be classified into 4 categories: general optimizations [Debbabi et al. 2003], ahead-of-time (AOT) optimizations [Proebsting et al. 1997], just-in-time compilation and selective dynamic compilation [Arnold et al. 2000; Debbabi et al. 2004; Shaylor 2002]. General and AOT optimizations are limited. JIT compilers are much more appropriate for J2SE and J2EE platforms because they require a lot of the memory to store the dynamic compiler and the generated code and use sophisticated flow analysis and allocation algorithms in order to generate high-quality native code. Selective dynamic compilation deviates from the JIT compilation by selecting and compiling, on the fly, just the frequently executed fragments of the class files (hotspots). This allows to reach significant acceleration of the virtual machine.

Accelerating the interpretation mechanism has been and is still a focus of interest for many researchers. Generally a pure bytecode interpreter entails a significant overhead. To circumvent this drawback, the use of direct threaded interpretation has been suggested where the central dispatch is eliminated. Each bytecode is replaced by an address of a corresponding implementation. The latter ends with the required dispatch to the next opcode. The inline threading interpretation technique [Piumarta and Riccardi 1998] improves upon the direct threading technique by eliminating the dispatch overhead within basic blocks. A new implementation is then dynamically created for such blocks 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 2003] proposed a technique, called "preparation sequence". It copes with the difficulties that arise when adapting the inline threading technique to Java and 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.) [Alpern et al. 2000; Azevedo et al. 1999; Cierniak et al. 2000; Hsieh et al. 1996; Sun 1999; Suganuma et al. 2000; Yang et al. 1999]. The drawbacks of these techniques are the loss of portability and an increasing complexity. Jalapeno [Alpern et al. 2000], HotSpot [Sun 1999] are the main acceleration techniques that rely on dynamic compilation. They are, however, unaffordable in the setting of embedded systems. Very little seems to have been published about the acceleration of embedded Java virtual machines. These are mainly CLDC HotSpot [Sun 2004] a lightweight accelerated VM for embedded systems introduced by Sun

Microsystems and KJIT [Shaylor 2002].

The closest work to the research reported in this paper is [Manjunath and Krishnan 2000]. The presented technique 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 technique reported in this paper leverages also the interpreter code but it deviates from [Manjunath and Krishnan 2000] 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.

3. APPROACH

The threaded interpretation achieves a relative speed-up over the switch based interpretation[Gagnon and Hendren 2003; Piumarta and Riccardi 1998]. In order to reach significant performance, it is necessary to introduce dynamic compilation. Embedding a dynamic compiler into a Java virtual machine targeting small devices limits severely the application of traditional optimizations (flow-based analysis). In this context, selective dynamic compilation is the most adequate approach. Performance-critical methods, also called hotspots, should be dynamically identified and compiled at run-time. The compilation cost in time and footprint should be lightweight.

It is crucial to design a system allowing a cooperation between an efficient interpretation mechanism together with a lightweight selective dynamic compilation. The technique we propose in this paper, 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. 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.

Generating a native interpreter allows us to implement the method frames in the native stack. This induces a fast switching between the interpreter and the compiler modes 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 interpreter mode to the native mode and vice versa. The technique we present in this paper is sustained by a smooth and uniform switching mechanism.

4. GENERATED NATIVE THREADED INTERPRETER

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 (translatable to a native code that is independent of dynamic information such as `aload_0`) and context-dependent bytecodes (requires some dynamic information to be translated to native code such as `iload`).

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 [Radhakrishnan et al. 2001]. According to the results of [Radhakrishnan et al. 2001], 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 [Spec 1998], these categories represent respectively 35.54%, 6.65% and 9.94% of the bytecodes. Hence, the compilation process of performance-critical methods by copying 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

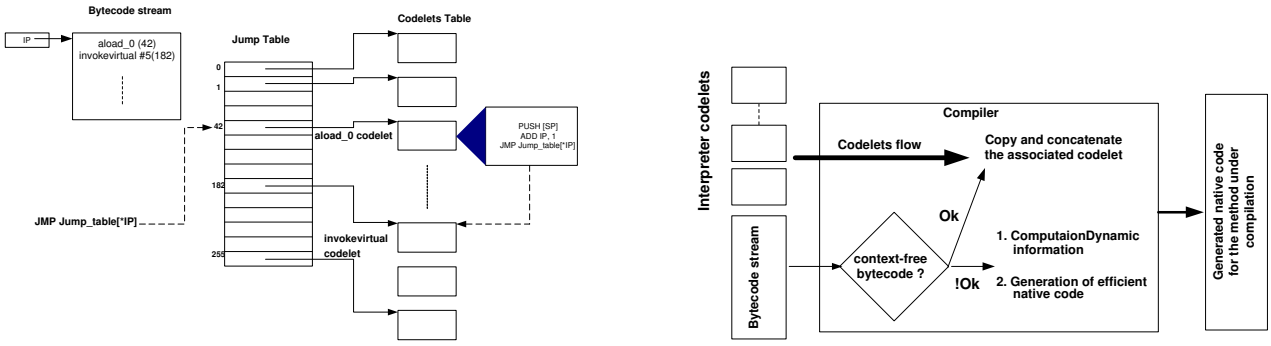

Fig. 1. (a)Generated Native Threaded Interpreter Structure (b)Interpreter Codelet Reuse Code Generation.

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 1 (a) outlines the interpreter structure.

The following C-like pseudo-code algorithm, `Generate_Interpreter()`, 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.

```
Generate_Interpreter( ) {
  Generate a jump instruction to Jump_table[*ip]
  For each bytecode
    generate corresponding native code
    Insert the address of the codelet
    into the next entry of the Jump_table
}
```

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

5. REUSING CODELETS FOR DYNAMIC COMPILATION

Dynamic compilation occurs at run-time. The compilation overhead is then a critical issue mainly 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 resource-constrained devices.

We have experimented the implementation of a lightweight one-pass dynamic compilation of java bytecodes in E-Bunny. The generated native code is stack-based. It is a fast and lightweight compilation technique. The results were very encouraging and promising. The technique described within this paper is a natural continuation of the work done on E-Bunny. 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 constant pool, etc, 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 1 (b) depicts the native code generation scheme used in this technique.

The `Compile(bytecode)` algorithm, in the C-like pseudo-code outlined above, is executed for each bytecode in the method under compilation. For context-free bytecode, the interpreter codelet associated with this opcode is just copied and concatenated to the already generated code in the 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,

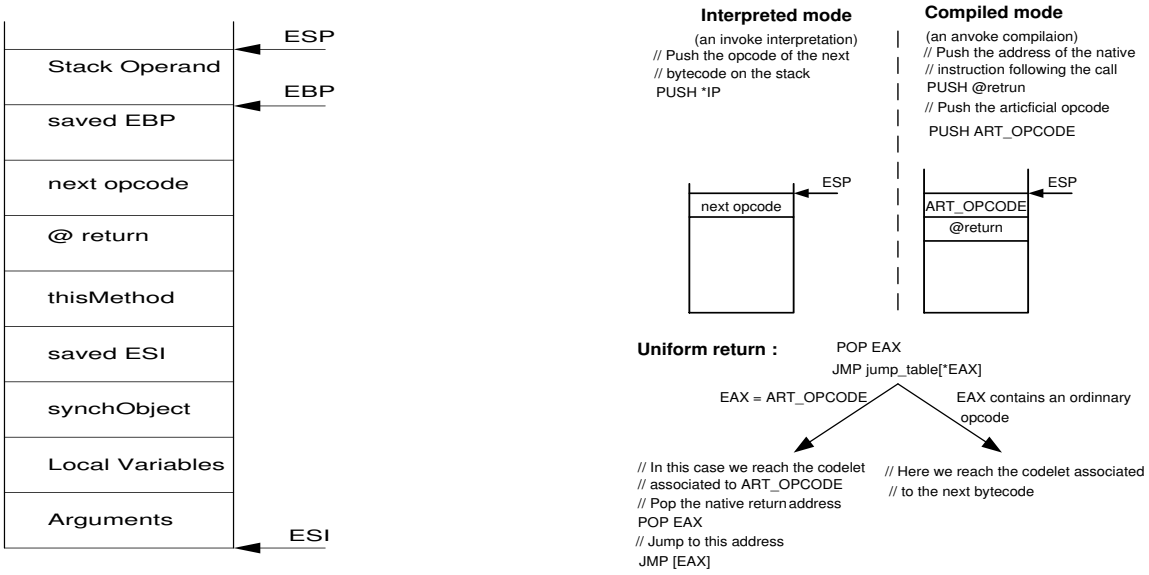

Fig. 2. (a) The stack Layout. (b) A Lightweight Interpreter/Compiler Mode Switch Mechanism.

such as the value of IP, to generate efficient native code.

6. SMOOTH SWITCHING MECHANISM

The technique we propose is supported by a smooth and uniform switching mechanism between the interpreter 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 native stack and manipulates it the same way the interpreter does with the Java stack [Hsieh et al. 1996]. The switch between the interpreter and native modes is expensive because of 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. Figure 2 (a) depicts the stack layout. Besides, some registers are dedicated to hold some information. For instance, the ESI register contains the parameter address whereas the EBX register contains the instruction pointer (IP).

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 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 2.(b) outlines the implementation of this lightweight uniform switching mechanism.

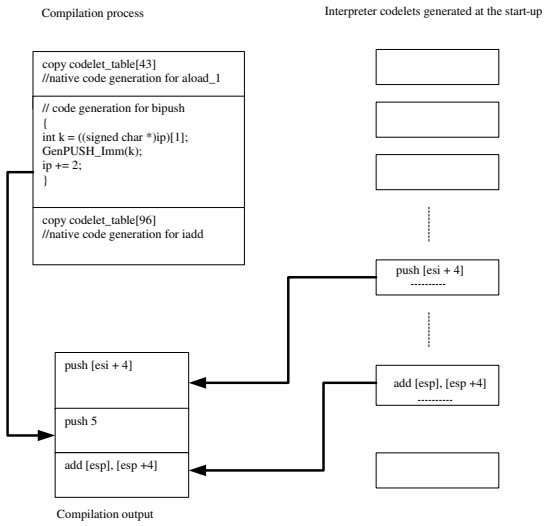

Fig. 3. A Scenario of the Proposed Technique

7. SCENARIO

As an illustration for the technique we propose, we consider the following bytecode sequence: `aload_1`, `bipush 5`, `iadd`. Figure 3 illustrates a snapshot of the code generation technique in action. The righthand part of Figure 3 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 `iadd (96)` are context-free. The native code generation for these bytecodes is then reduced to a copy of the corresponding interpreter codelets as depicted in Figure 3. 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. 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.

8. CONCLUSION

We reported, in this paper, a technique for the acceleration of embedded Java virtual machines. The technique 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.

REFERENCES

Alpern, B., Attanasio, C., Barton, J., Burke, M., Cheng, P., Choi, J., Cocchi, A., Fink, S., Grove, D., M. Hind, S. H., Lieber, D., Litvinov, V., Mergen, M., Ngo, T., Russell, J., Sarkar, V., Serrano, M., Shepherd, J., Smith, S., Sreedhar, V., Srinivasan, H., and Whaley, J. 2000. The Jalapeno Virtual Machine. *IBM Systems Journal 39,* 1, 211–238.

Arnold, M., Fink, S., Grove, D., Hind, M., and Sweeney, P. 2000. Adaptive optimization in the Jalapeño JVM. *ACM SIGPLAN Notices 35,* 10, 47–65.

Azevedo, A., Nicoleau, A., and Hummel, J. 1999. Java Annotation-aware Just-in-Time (AJIT) Compilation System. In *Proceedings of ACM Java Grande 99*. ACM Press, San Francisco, California, USA, 142–151.

Cierniak, M., Lueh, G., and Stichnoth, J. 2000. Practicing JUDO: Java under Dynamic Optimizations. In *Proceedings of SIGPLAN 2000 Conference on Programming Languages Design and Implementation*. ACM Press, Vancouver, Canada, 13–26.

Debbabi, M., Erhioui, M., Ketari, L., Tawbi, N., Yahyaoui, H., and Zhioua, S. 2003. Method Call Acceleration in Embedded Java Virtual Machines. In *Proceedings of International Conference on Computational Science (ICCS'03)*, P. Sloot, D. Abramson, A. Bogdanov, J. Dongara, A. Zomaya, and Y. Gorbachev, Eds. Lecture Notes in Computer Science, LNCS 2659. Springer-Verlag, Melbourne, Australia, 750–759.

Debbabi, M., Gherbi, A., Ketari, L., Talhi, C., Tawbi, N., Yahyaoui, H., and Zhioua, S. 2004. E-Bunny: A Dynamic Compiler for Embedded Java Virtual Machines. In *Proceedings of ACM PPPJ'04*. ACM Press, Las Vegas, USA.

Gagnon, E. and Hendren, L. 2003. 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)*, G. Hedin, Ed. Lecture Notes in Computer Science, vol. 2622. Springer Verlag, Warsaw, Poland, 170–184.

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification*. Addison Wesley, CA, USA.

Hsieh, C., Gyllenhaal, J., and Hwu, W. 1996. Java bytecode to Native Code Translation: the Caffeine Prototype and Preliminary Results. In *Proceedings of the 29th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, Paris, France, 90–99.

Lindholm, T. and Yellin, F. 1996. *The Java Virtual Machine Specification*. Addison Wesley, CA, USA.

Manjunath, G. and Krishnan, V. 2000. A Small Hybrid JIT for Embedded Systems. *SIGPLAN Notices 35,* 4, 44–50.

Pendragon. 1996. CaffeineMark. http://www.benchmarkhq.ru/cm30/.

Piumarta, I. and Riccardi, F. 1998. Optimizing Direct-threaded Code by Selective Inlining. In *Proceedings of SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, Montreal, Canada, 291–300.

Proebsting, T., Townsend, G., Bridges, P., Hartman, J., Newsham, T., and Watterson, S. 1997. Toba: Java for Applications: A Way Ahead of Time (WAT) Compiler. In *Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems*. Usenix Association, Berkeley, 41–54.

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

Shannon, B. 2001. *Java 2 Platform Enterprise Edition, v 1.3*, Version 1.3 ed. Sun Microsystems Inc.

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

Spec. 1998. SPEC JVM98 Benchmarks. http://www.specbench.org/osg/jvm98/.

Suganuma, T., Ogasawara, T., Takeuchi, M., Yasue, T., Kawahito, M., Ishizaki, K., Komatsu, H., and Nakatani, T. 2000. Overview of the IBM Java Just-in-Time Compiler. *IBM Systems Journal 39,* 1, 175–193.

Sun. 1999. The Java HotSpot Performance Engine Architecture. White Paper.

Sun. 2000. Connected, Limited Device Configuration. Specification Version 1.0, Java 2 Platform Micro Edition. White Paper.

Sun. 2003a. *Java 2 Platform, Standard Edition, v 1.4.2 API Specification*, Version 1.4.2 ed. Sun Microsystems Inc.

Sun. 2003b. KVM Porting Guide. White Paper.

Sun. 2004. CLDC HotSpot Implementation Virtual Machine. White Paper.

Yang, B., Moon, S., Park, S., Lee, J., Lee, S., Park, J., Chung, Y., Kim, S., Ebcioğlu, K., and Altman, E. 1999. 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)*. IEEE Computer Society Press, Newport Beach, California, 128–138.