

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.
debbabi@ciise.concordia.ca,
gherbi@ece.concordia.ca, zhioua@ece.concordia.ca

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

Abstract

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 latest version of KVM. This experimentation was carried on using standard J2ME benchmarks.

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

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) [Sun 2000] 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) [Sun 2003]. The KVM is a Java virtual machine initially designed with the constraints of 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 [Comeau 2002], Nazomi [Communications 2002], etc.), had proposed Java processors that execute in silicon Java bytecodes. Although these hardware accelerators achieve a significant speedup in terms of

*The initial stage of this work has been funded by Panasonic Information and Networking Technologies Laboratory, Princeton, New Jersey, USA.

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 [Alpern et al. 2000; Cramer et al. 1997; Gagnon and Hendren 2003; Hsieh et al. 1996; Sun 1999; Radhakrishnan et al. 2001]. 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 %) [Sun 2004]. 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 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 [Sun 2004] and [Schmid 2002].
- 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 [Alpern et al. 2000; Cierniak et al. 2000; Sun 1999; Suganuma et al. 2000; Suganuma et al. 2001] are endowed with a dynamic compiler. The features of these VMs cannot be applied in an embedded context due to lack of resources. 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 [Bytecodes 2003]. 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 [Bytecodes 2003]. This leads to several cache misses that decreases the program performance.

Despite these difficulties, dynamic compilation is also used in CLDC-based embedded virtual machines [Sun 2004; Schmid 2002; Shaylor 2002]. However, except one paper about KJIT [Shaylor 2002], no detailed information about these systems is available in the literature due to commercial reasons.

KJIT [Shaylor 2002] 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 [Sun 2004] is an embedded virtual machine introduced by Sun Microsystems. As its name indicates, it is strongly inspired by the standard Java Hotspot 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 138 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 [Lindholm and Yellin 1996] (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

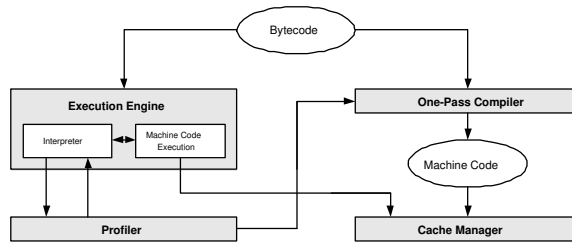


Fig. 1. E-Bunny Architecture

optimizations. Only optimizations that might be applied in one-pass are allowed.

- 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 allocated to the compiled code. When this space is full, a cache strategy based on a Least Recently Used (LRU) algorithm [Majercik and Littman 1998] 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 one-pass 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.

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.

4.1 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 offset, 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 (e.g. loads (`iload`, `iaload`, `ldc`), stores (`astore`, `lastore`) and stack manipulation bytecodes (`pop`, `dup`)). They are called: simple bytecodes. The second kind of bytecodes are more complex to translate and are called complex bytecodes (e.g. field manipulation (`putfield`), method call (`invokevirtual`) and exception propagation bytecodes (`athrow`)). The first version of E-Bunny targets Intel IA-32 architecture [Intel 2000]. Another version for ARM is under development.

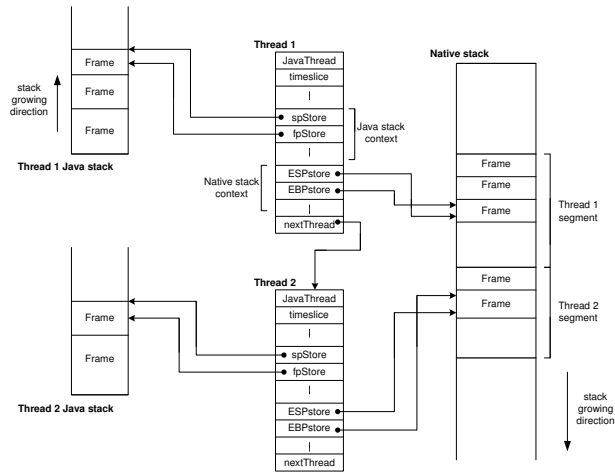


Fig. 2. Multi-Threading in E-Bunny

4.2 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.

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.

4.3 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. 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 2).

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.

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

Table I. Executable File Footprint overhead

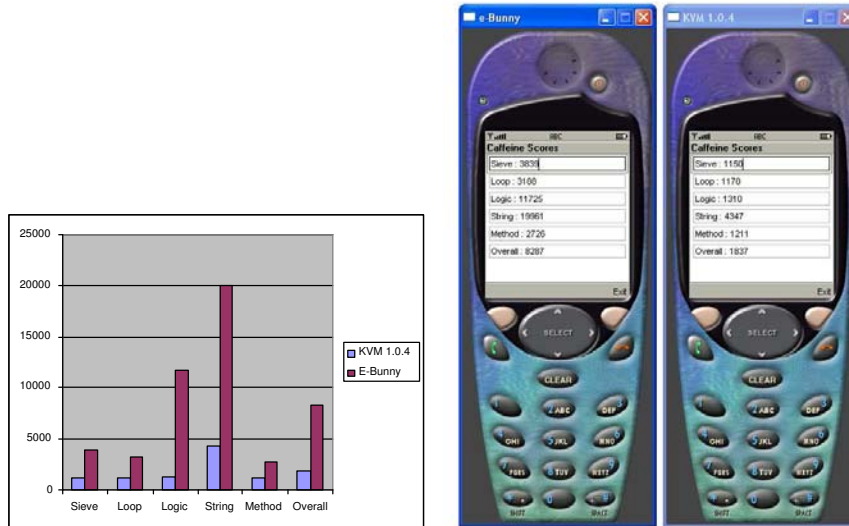


Fig. 3. CaffeineMark Scores of KVM and E-Bunny with GCC

4.4 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 inaccurate 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 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 I 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. 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 the E-Bunny dynamic compiler is 138 KB. To evaluate the performance of E-Bunny, we have run CaffeineMark benchmark (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. Moreover, we built the MIDP 2.0 profile, intended to CLDC devices, using E-Bunny and we ran successfully several midlets. Figure 3 shows a snapshot of the MIDP emulator illustrating CaffeineMark midlet results and an histogram presenting obtained individual

scores. The overall scores of E-Bunny and KVM show the overall speedup.

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 code size.

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 Jalapeño Virtual Machine. *IBM Systems Journal* 39, 1, 211–238.
- BYTECODES. 2003. Just In Time Compilers. <http://www.bytecodes.com/techJITC2.html>.
- CIERNIAK, M., LUEH, G., AND STICHOOTH, 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.
- COMEAU, G. 2002. Java Companion Processors versus Accelerators. <http://www.zucotto.com>.
- COMMUNICATIONS, N. 2002. Boosting the performance of Java Software on Smart Handheld Devices and Internet Appliance. <http://www.nazomi.com>.
- CRAMER, T., FRIEDMAN, R., MILLER, T., SEHERGER, D., WILSON, R., AND WOLCZKO, M. 1997. Compiling Java Just in Time. *IEEE Micro* 17, 2, 36–43.
- 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.
- 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.
- INTEL. 2000. *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, Order Number 245470 ed. Intel Corporation, California, USA.
- LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification*. Addison Wesley, CA, USA.
- MAJERCIK, S. AND LITTMAN, M. 1998. 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)*. AAAI Press, Menlo Park, 954–959.
- RADHAKRISHNAN, R., VIJAYKRISHNAN, N., JOHN, L., SIVASUBRAMANIAM, A., RUBIO, J., AND SABARINATHAN, J. 2001. Java Runtime Systems: Characterization and Architectural Implications. *IEEE Transactions on Computers* 50, 2, 131–146.
- SCHMID, K. 2002. Esmertec's Jbed Micro Edition CLDC and Jbed Profile for MID. Tech. rep., Esmertec AG, Dubendorf, Switzerland. Spring.
- 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.
- 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.
- SUGANUMA, T., YASUE, T., KAWAHITO, M., KOMATSU, H., AND NAKATANI, T. 2001. A Dynamic Optimization Framework for a Java Just-in-Time Compiler. *ACM SIGPLAN Notices* 36, 11, 180–194.
- 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. 2003. KVM Porting Guide. White Paper.
- SUN. 2004. CLDC HotSpot Implementation Virtual Machine. White Paper.