Embedded JIT Compilation with CACAO on YARI

Florian Brandner
Institute of Computer Languages
Vienna University of Technology, Austria
brandner@complang.tuwien.ac.at

Tommy Thorn
Unaffiliated Researcher
California, USA
tommy@thorn.ws

Martin Schoeberl
Institute of Computer Engineering
Vienna University of Technology, Austria
mschoebe@mail.tuwien.ac.at

Abstract

Java is one of the most popular programming languages for the development of portable workstation and server applications available today. Because of its clean design and typesafety, it is also becoming attractive in the domain of embedded systems. Unfortunately, the dynamic features of the language and its rich class library cause considerable overhead in terms of runtime and memory consumption. Efficient techniques to implement Java virtual machines that are suitable for use in resource constrained environments are thus needed. In this work we present a solution for very restricted environments based on CACAO. CACAO is a just-in-time compiling virtual machine implementation, combining high speed and small size. We have modified the original version of CACAO to run without an underlying operating system within only 1 MB of memory. In addition we present a new technique to selectively compile methods during the initialization phase of real-time Java applications to prevent unwanted interaction between dynamic compilation and critical tasks. Furthermore we present the YARI soft-core as the execution platform of CACAO within an field-programmable gate array. We compare our implementation with two well known Java processors, JOP and Sun's picoJava-II, on the same technology. Although JOP achieves a higher clock frequency and picoJava-II occupies nearly 4 times the resource of YARI, our solution is capable to outperform both of them by a factor of up to 2.8 *and* 2.2 *respectively.*

1 Introduction

In the last years Java became one of the most popular programming languages for application development on workstations and servers. This can be attributed to the simplicity, safety, and portability of the language. Because of these properties Java is also becoming more and more attractive to developers of embedded systems. Technologies utilized by Java virtual machines (JVM) on workstations are not practical for embedded systems, due to resource constraints. Just-in-time (JIT) code generation and adaptive optimizations lead to increased power and memory consumption, and may incur unacceptable runtime penalties.

The memory overhead of a fully compliant Java implementation can be overcome by offering only a subset of the rich Java library. The Java Platform Micro Edition (JavaME) [33] is a widely used variant of such a restricted environment intended for use in mobile and embedded devices. The Micro Edition consists of a minimal set of core classes required for a JVM to operate, and a set of optional extensions targeting specific domains, e.g., the Mobile Information Device Profile for mobile phones. Java Card offers an even smaller core library for more restricted environments, such as smart cards.

To further reduce the memory footprint the JVM itself has to be optimized for code and data size. Complex techniques such as JIT-compilation, runtime profiling, and adaptive optimizations can usually not be applied, instead, slow interpreters execute the Java bytecode.

Embedded systems very often have to fulfill timing constraints to operate correctly, e.g., to guarantee quality of service or to control external components in a timely fashion. Interpreters and JIT compilation do not guarantee tight real-time bounds. The slow interpretation techniques impose a natural limit for response time and throughput, similarly accounting for the expensive compilation step of a JIT system leads to overly conservative bounds of the programs overall execution time. Compiling the Java programs offline – *ahead-of-time* – allows to overcome these limitations by trading flexibility with small code size and fast execution within predictable time bounds. The use of dynamic features of the Java language is heavily restricted.

In this paper we present a just-in-time compiling JVM solution for small embedded systems. We have adapted the CACAO research JVM [15] to run without an underlying operating system in an environment offering only 1 MB of memory. Based on the execution model of safety critical

Java [14, 27], we propose a new technique called *mission-start-compilation*. Critical methods are pre-compiled upon transition to the mission phase, eliminating unwanted interference of the JIT compilation process with real-time tasks. Furthermore, we present the YARI soft-core as the execution platform within a field-programmable gate array (FPGA).

All software tools required for our solution are publicly available open source projects – including the build tools, the Newlib C library [21], the phoneME Java class library [30], the CACAO JVM [15], and the YARI soft-core [31]. We hope that this open source approach will facilitate the research on and development of JVMs in embedded systems. The major contributions presented in this work are as follows:

- An open source JVM implementation for embedded systems, including hard- and software components
- A Java JIT system running in a resource constrained environment offering only 1 MB of memory
- A new technique called *mission-start-compilation*, that allows the use of JIT compilation on systems with timing constraints

In the remainder of this paper we will present related work in Section 2. Section 3 introduces the YARI soft-core used as the execution environment for our JVM. In Section 4 we present the CACAO JVM, along with a detailed description of the required changes to run Java programs in a resource constrained environment. *Mission-start-compilation* is described in Section 5. We present results of the empirical evaluation in Section 6, comparing our solution with the JOP and picoJava-II Java processors. Finally we conclude in Section 7.

2 Related Work

The presented project touches several areas in the embedded domain: embedded Java, Java processors, and RISC soft-cores for FPGAs. The following section gives a brief overview of the most relevant work in each area. A detailed performance comparison of embedded Java systems can be found in [25]. Java systems for real-time usually do not rely on JIT compilation; instead, the main vendors of real-time enabled JVMs perform some form of pre-compilation. For example, the JamaicaVM [28] and OVM [3] use ahead-of-time compilation, whereas Sun's RTS [6] and WebSphere RealTime [9] pre-compile classes at load time. Our proposed approach is different in the form that compilation of Java classes is deferred to the latest possible moment in the live time of a real-time application: the mission start phase. This feature allows more dynamics for soft real-time systems as classes can be loaded dynamically during the start-up phase of the application.

The Squawk VM [29] is an embedded JVM mostly written in Java, that is now open source. Squawk was originally developed for Sun SPOT, a wireless sensor platform. It runs on the *bare metal* and provides functionality typical found in operating systems in Java, e.g., device drivers. SimpleRTJ [23] is an interpreting JVM intended for small embedded systems, which requires only 18–24 KB of memory to run. In [4] a lightweight JIT compilation system, targeted for resource-constrained environments, is presented. Muvium [8] is an ahead-of-time compiling JVM solution for very resource constraint Microchip PIC microcontrollers.

An alternative to interpretation and compilation is direct execution of Java bytecode using a dedicated Java processor. Sun introduced the first version of picoJava [18] in 1997, although this processor was never released as a product by Sun. A redesign followed in 1999, known as picoJava-II, which is now freely available. It is the most complex Java processor available, and implements, among other optimization, a folding mechanism in hardware, that allows to execute short sequences of Java bytecodes as a single RISC-like instruction.

The JEMCore from aJile is a Java processor that is available as both an IP core and a standalone processor [12]. It is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. The Cjip processor [11, 13] supports multiple instruction sets, allowing Java, C, C++ and assembler to coexist. The JVM is implemented largely in microcode (about 88% of the Java bytecodes). Microcode instructions execute in two or three cycles. A JVM bytecode requires several microcode instructions. Komodo [16] is a multi-threaded Java processor for embedded real-time systems. The unique feature of Komodo is the concept of interrupt service threads. Komodo is now commercialized under the name jamuth [32].

JOP [26] is a Java processor designed especially for embedded real-time systems. The main design goal was a time predictable processor. All hard to analyze processor features, such as prefetching or automatic stack dribbling, as found in picoJava, have been avoided. To still provide acceptable performance a special stack cache and a WCET analyzable method cache have been developed. SHAP [35] is a new Java processor based on the architecture of JOP and enhanced by a hardware garbage collector.

Highly configurable and optimized RISC soft-cores are also offered by FPGA vendors. Among these 32-bit soft-cores, probably the best known are MicroBlaze from Xilinx [34], Nios II from Altera [2], and Mico32 from Lattice [17]. Nios II and MicroBlaze are proprietary, whereas Mico32 is available under an open source license. LEON [10] is an implementation of the SPARC V8 architecture. LEON, implemented on the same FPGA board we use for YARI, consumes about 8,000 logic cells,[1] 11 KB on-chip memory and can be clocked at 35 MHz. Initially designed

[1] The basic resource in FPGAs is the logic cell, which essentially is a four-bit lookup table and a register.

with for the purpose of radiation hardened implementations, LEON has been released under an open source license. All of these soft-cores are supported by complete development kits with compilers, libraries, and debuggers.

3 YARI

YARI (Yet Another RISC Implementation), is an open source [31] FPGA microprocessor implementation, created as a vehicle to investigate implementation ideas. To avoid the burden of having to provide a complete tool-chain the instruction set is designed to be mostly compatible with the MIPS-I™architecture, a seminal, thoroughly documented, and, for our purpose, sufficiently simple RISC architecture.

The core philosophy of the RISC methodology is to aim for the best balance between hardware and software, and thus also to create an architecture that is optimally suited to the underlying technology, e.g. VLSI. FPGAs differ from VLSI in the relative cost and speed of primitives: random logic, wires, and thus muxes, are relatively slow, whereas memory and adders are relatively fast, registers cheap, etc. As a consequence, the MIPS-I™architecture, designed for VLSI, may not be an optimal architecture for FPGAs.

YARI has a mostly classic five stage pipeline: instruction fetch, instruction decode/register files access, execute/memory, memory, and write back. Great emphasis has been placed on load/store performance. For this reason YARI is equipped with a four-way associative instruction cache, a four-way write-through data cache, and a store buffer.

Pipeline stalling can have surprisingly complicated interactions with branch delay slots and pipeline restarts. Furthermore, the control path for the stall logic is inherently timing critical, as the stall signal has to control every flip-flop in the stages it stalls. Given this, the design of YARI avoids pipeline stall completely. The only means to disrupt the pipeline is through a pipeline restart which flushes part of the pipeline. Pipeline registers are not cleared when the stage is flushed. Instead each state carries a "valid" bit which is only consulted at points where outputs of a pipeline stage changes architectural state. That includes restart signals, writes to the registers file, and stores. While this can result in more pipeline bubbles than stalling, the resulting simpler logic leads to a shorter cycle time, and thus, higher frequency.

In any cycle, one or more hazards can occur simultaneously, forcing a restart of the pipeline. The amount of pipeline stages flushed and the resulting latency in responding to the hazard is a major factor contributing to the cycles-per-instruction metric, thus inversely proportional to the observed performance. In the configuration used in this paper, YARI is configured with a radix-2 multiplier and divider, thus the result is only available roughly 33 cycles after issuing the operation. Any attempt at accessing the result earlier will cause a restart of that instruction.

A *load-use* hazard occurs when an instruction immediately following a load tries to use the load result. By construction, the result isn't ready to be forwarded, and we have to restart the load-use. This hazard is detected in the execution stage and has a two cycle penalty. Instruction schedulers of MIPS compilers typically know about this hazard and try to avoid code that violates it. Unfortunately, CACAO doesn't respect this load-use hazard resulting in the number one source of pipeline inefficiencies.

Since YARI uses a split data and instruction cache without coherency, it is necessary, that code writing data intended as instructions flush that region from the instruction cache using the `synci` instruction to force an update. This has to be taken into account by systems that dynamically generate code, e.g., JIT compilers such as CACAO.

The data cache is a classic implementation: the four tags are accessed in parallel for the four cache ways, followed by a late select based on which tag (if any) matched. As stores are destructive, we must know the destination way before we can execute it. Thus, the actual store to the cache way happens in the memory stage. The consequence is, that a load immediately following a store of the same address, will see stale data. This is known as *load-hit-store*. While we could add logic to forward the store data to the load, this case is so rare[2] that we instead trade off the occasional pipeline restart for a simpler data path.

4 CACAO

CACAO [15] is a research platform developed at the Vienna University of Technology. Over the years it was steadily improved and eventually grew into a stable and fast JVM for workstation and server applications. Because of its small size and fast JIT compilation it has become an attractive alternative for the development of Java enabled embedded systems. So far several projects successfully employed CACAO running on Embedded Linux for MIPS and ARM platforms. We were able to eliminate the need for an underlying operating system and enable CACAO to run in a minimal execution environment on top of the YARI soft-core.

In its default configuration CACAO has several prerequisites that have to be met in order to run Java programs. Most notably a full operating system, typically Linux, is required to manage I/O operations, memory and threads. Operating systems in turn demand more powerful hardware, offering virtual memory and protection mechanisms. For systems using Java as their sole execution platform the operating system can, and should be, avoided in order to reduce memory consumption and to lower the hardware requirements.

The Newlib [21] project by RedHat offers a minimal, but complete, C library implementation that allows to run programs without an underlying operating system on top of a

[2] Optimizing compilers generally avoid reloading a just stored value.

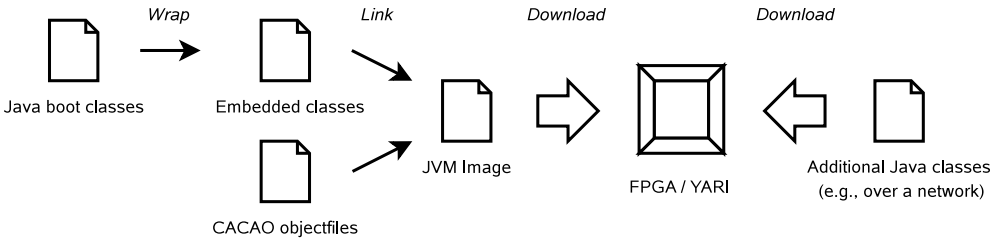

Figure 1. Downloading a Java application to YARI.

bare processor. It specifically targets small embedded devices and thus has minimal prerequisites. The Newlib project does not aim to offer a complete replacement for operating systems, and thus lacks some functionality. Most notably Newlib does not offer any kind of process or thread management. Similarly, signal handling is not provided, on which CACAO relies on, to invoke the code generator, the garbage collector, and to handle traps and Java exceptions.

In order to run in a restricted environment minor extensions and modifications to the core of CACAO are required. Most of these modifications disable features of the standard Java implementation. For example networking support, all encryption and security related features, file compression, and similar components of a workstation Java implementation are disabled. In addition the size of many internal buffers has been adapted for use in an embedded system. Most notably the size of the Java heap, which usually occupies 128 MB of memory, is reduced to only a few KB.

Resolving native methods at runtime using a dynamic loader is not possible without an operating system. As a consequence native methods need to be statically linked into the executable binary. Similarly, classes required during the system start-up phase need to be embedded into the executable binary. A minimal set of these bootstrap classes is determined by static analysis, wrapped into regular object files, and finally linked with the CACAO executable file. Figure 1 depicts the necessary steps to embed a Java application into CACAO and download it to YARI.

CACAO is already designed to support different Java class libraries. The phoneME package by Sun lends itself for embedded systems, because of its small size. More specifically the CLDC core class library is used as the basis for this project. The use of phoneME has, besides its small memory footprint, the advantage to be compatible with other JavaME based technologies, such as the Real-Time Specification for Java [7] and JSR 302 on Safety Critical Java [14, 27]. Although we strive for compatibility some features are not fully compliant with the JavaME platform. As noted, Newlib does not provide any process management functionality, multithreading in Java programs is thus disabled. *Green threads*, i.e., threads managed by CACAO itself, could be used to circumvent this deviancy. In addition garbage collection is disabled, because of the memory requirements of the currently used Boehm collector [5]. Development of a replacement, which is expected to be considerable smaller, is in progress.

5 JIT-Compilation in Real-time Systems

In contrast to most high performance JVM implementations CACAO adopts a compile only approach, i.e., all Java bytecode is compiled to machine code of the target machine before its execution. This approach greatly simplifies the internal organization, but also entails some drawbacks. Infrequently executed code, e.g., static class initializers and other initialization code, causes considerable overhead in terms of compilation time and memory consumption. To reduce the compilation overhead CACAO offers a highly tuned JIT compiler.

Code generation is divided into four major steps, namely parsing, stack analysis, register allocation, and machine code emission. First the Java bytecode is translated into a register-oriented intermediate representation, which is better suited for further processing than the stack-oriented Java bytecode. In the next step, stack slots containing intermediate results are converted to virtual registers. The register allocation phase maps the virtual registers to machine specific registers. The last step of code generation is the emission of the final machine code. This is done using a simple macro expansion of operations in the intermediate representation to instructions of the target machine. It is important to note, that all these phases are at most linear in runtime, and thus very fast. More information on the internals of CACAO's JIT compiler can be found in [15].

Because JIT compilation is relatively expensive, methods are compiled on demand, i.e., only when a method is to be executed the first time. If the target method of a call is not yet compiled, the code generator emits a call stub instead of a regular method call. The stub triggers the compilation of the method if required, and is replaced by a regular call using a code patching mechanism afterwards. This lazy approach may cause considerable delay during the start-up phase of a program.

JIT compilation is usually avoided for real-time systems due to this unpredictability. An exact analysis of bounds for the execution time of the original Java program is impossible in general, because it is uncertain when the JIT compila-

Soft-Core	Logic Cells	Memory	Frequency
JOP	3,300	7.6 KB	100 MHz
YARI	6,668	18.9 KB	80 MHz
pico-Java-II	27,560	47.6 KB	40 MHz

Table 1. FPGA synthesis results of the JOP, YARI, and picoJava-II soft-cores.

Benchmark	JOP	picoJava-II	CACAO
Sieve	7386	7721	12109
Kfl	19907	23813	51200
UdpIp	8837	11950	23223
Lift	18930	25444	52261

Table 2. Number of iterations per second for the JBE application benchmarks. A higher number means faster.

tion will actually take place. Modern systems often choose to interpret most methods several times, until a threshold is reached indicating that a given piece of code is worth the effort of the expensive JIT compilation. It is thus hard for an offline timing analysis to predict the code that actually will be executed.

CACAO follows a compile-only approach, eliminating some of the problems beforehand. The code generation scheme is largely based on simple macro expansion. It is thus relatively easy to predict basic properties of the code that actually will be executed. Therefore, if we can tolerate the overhead induced by JIT compilation during the warm-up phase, CACAO is an option for soft real-time systems. In general, however, this overhead is not acceptable for real-time systems. The uncertainty at what time compilation will be necessary still impedes the calculation of meaningful bounds for the programs execution time. To avoid compilation during the critical phase of a real-time task, we propose compilation during the non-critical initialization phase of the application. We adopt the programming model for safety critical Java [20, 27] that defines three major phases: the *initialization*, the *mission start*, and finally the *mission*, that runs forever. It has to be noted, that we do not target safety-critical applications, at least not the most rigid levels of DO-178B [22], with our system. We just *borrow* the concept for less demanding real-time applications.

We propose a compile at mission start model. During the initialization phase all classes are loaded and data structures allocated. On the transition to the mission phase – the mission start – we analyze the application on the target and build a list of all methods that are possibly invoked during the mission phase. The listed methods are then pre-compiled using the regular compiler. JIT overhead during the mission phase is completely eliminated, allowing a more accurate timing analysis.

Pre-compiling Java programs offline is well known, and usually referred to as ahead-of-time compilation. Systems relying on ahead-of-time compilation usually do not allow for dynamic features of the Java language, e.g., class loading. The main benefit of our new mission-start-compilation is that dynamic class loading can be done during the initialization phase without any limitation. For example, an application can, at each reboot, check for updates of individual classes, and even download and make use of these classes

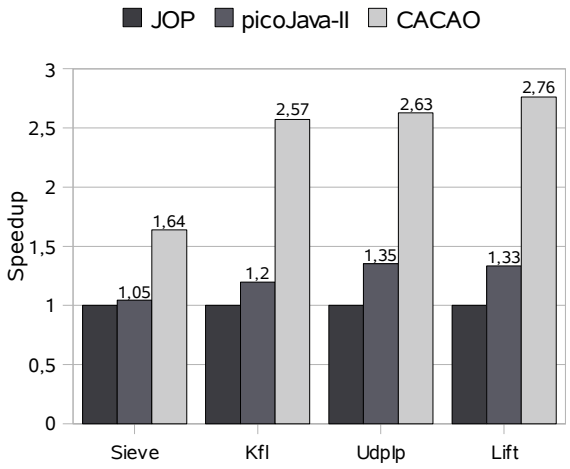

Figure 2. Performance of picoJava-II and CACAO on YARI for some embedded benchmarks, normalized to JOP.

using dynamic class loading over a network. Especially in the case of expensive communication, e.g., because of low-bandwidth and high power consumption of radio elements, this approach is beneficial, as the amount of data that needs to be transferred is heavily reduced. In the case of ahead-of-time compilation, selectively downloading individual classes is not possible. Instead the complete application binary needs to be transmitted, stored into the systems flash memory, and an additional reboot performed in order to acquire updates.

6 Evaluation

In this section we provide an evaluation of the combination of CACAO and YARI within an FPGA. We show execution performance on a set of embedded Java benchmarks and micro-benchmarks from the JavaBenchEmbedded (JBE) suite [1]. A detailed description of the benchmarks can be found in [25]. We compare the obtained results to two Java processors, namely JOP [24] and an FPGA implementation of Sun's picoJava-II [19]. All three soft-cores are synthe-

Micro-Benchmark	JOP	picoJava-II	CACAO
iload3 iadd	2	2	1
iinc	4	3	2
ldc	9	3	3
if_icmplt_taken	6	6	4
if_icmplt_not_taken	6	-	3
getfield	16	3	3
getstatic	9	5	5
iaload	11	3	9
invoke	128	24	13
invokestatic	100	24	12
invokeinterface	144	196	15

Table 3. Cycles required to execute specific Java bytecodes for JOP, picoJava-II, and CACAO on YARI.

sized using the free Altera design software Quartus 7.1 for an Altera FPGA; YARI and JOP for the Cyclone EP1C12C6 FPGA and picoJava-II for a larger Cyclone II FPGA. Table 1 lists the basic properties of the synthesized soft-cores. The FPGA is integrated on an evaluation board offering 1 MB RAM. All benchmarks were executed on the same platform, with all required Java classes readily available in the systems memory. In the case of CACAO all required classes were embedded into the executable binary.

6.1 Performance

Our new solution, based on CACAO running on top of YARI, offers the best performance compared to the two other options. Table 2 shows the raw performance numbers and Figure 2 the speedup of picoJava-II and CACAO on YARI relative to JOP. Although JOP achieves the highest frequency in our setup, the results show the least performance for the four benchmarks. For all our tests the CACAO JVM is faster by a factor of 1.64 to 2.8 in comparison to JOP. CACAO is also able to outperform picoJava-II, resulting in a speedup of a factor between 1.57 and 2.15.

In addition to an overall comparison based on these larger benchmark programs, we also conducted experiments to evaluate the efficiency of each approach for individual Java bytecodes. The JBE suite contains several micro-benchmarks for this purpose. Each micro-benchmark tests only one or two specific Java bytecodes and reports the number of cycles required for its execution. The results of this experiment are summarized in Table 3. For simple bytecodes, e.g., *iadd*, JOP and picoJava-II basically require the same amount of cycles. CACAO on YARI, in comparison, executes most of these opcodes in about half the cycles. More complex opcodes, e.g., *invokeinterface*, are considerably more expensive on all three platforms. Nevertheless CACAO executes these opcodes by far more efficiently, with speedups of up to a factor of 10. Because of inaccuracy in determining the overhead of surrounding code of the micro-benchmarks it is not always possible to derive meaningful cycle counts. The value of *if_icmplt_not_taken* for picoJava-II is omitted because of that inaccuracy.

Benchmark	Classes	Data	Heap	Total
Sieve	56,861	81,223	230,533	715,201
Kfl	72,526	80,990	267,074	767,174
UdpIp	69,059	81,465	267,796	764,824
Lift	62,167	81,117	251,159	741,027

Table 4. Memory consumption in bytes for class files, static and heap data, and the total memory consumption.

Bench.	Total	Compiled	Bc-instr.	Mips-instr.
Sieve	405	78	3,419	22,436
Kfl	465	117	6,577	33,368
UdpIp	455	107	5,443	33,308
Lift	422	89	4,522	27,572

Table 5. Statistics on JIT compilation, showing the total number of methods, the number of compiled methods, and the size in bytes of the translated bytecode and machine instructions.

6.2 Memory Consumption

For resource constrained embedded systems memory consumption is of utmost importance. Memory typically contributes a fair amount to the overall costs of such a system, and is thus minimized as much as possible. All benchmarks presented in the last section were run within 1 MB of memory. In the case of CACAO the whole JVM, the original Java classes, the dynamically generated code, and all data of the Java programs need to fit into this small amount of RAM. Table 4 summarizes the amount of memory required to hold the Java classes of the benchmark, data of the CACAO JVM, and finally the peak amount of heap memory allocated at runtime. In addition Table 5 presents details on the JIT compilation performed at runtime. In general only a small amount, between 19% and 25%, of the available methods are actually compiled. The fraction of dynamically generated code is thus relatively small and never exceeds 4.3% of the overall consumed memory.

In JOP the main part of the JVM is implemented in hardware and the Java library is very restricted. The memory requirements are thus by far less demanding. For example, the memory consumption of the linked class files for a *Hello World* program is about 36 KB.

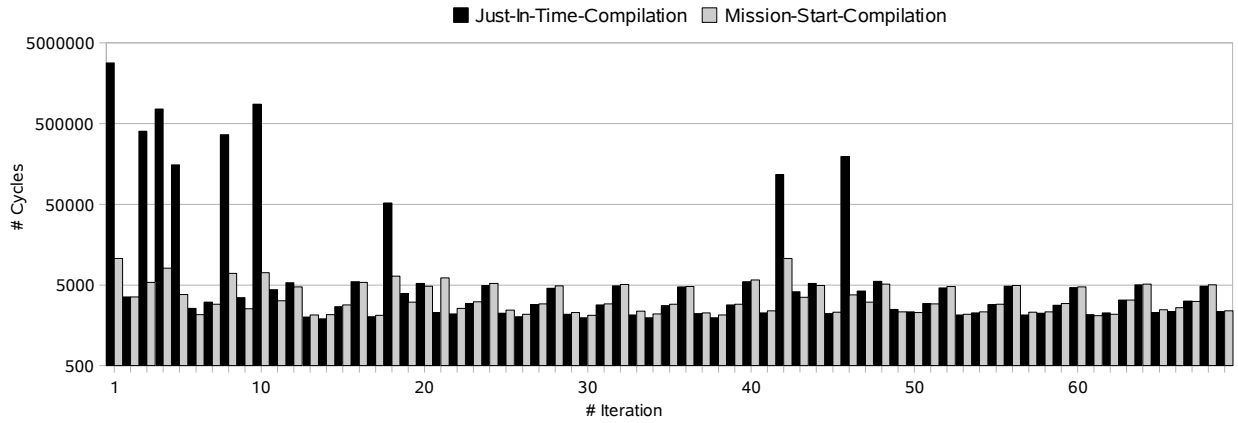

Figure 3. Execution time per iteration for just-in-time-compilation vs. mission-start-compilation over time.

6.3 Mission Start Compilation

We have evaluated our compile at mission start approach using a small Java real-time application, *Kfl* from the JBE suite. The program is a simple control algorithm that monitors a set of sensors and controls a set of actuators. The environment for the sensors and actuators is simulated in Java. The test executes the control function repetitively in a loop. However, at different iterations of the loop different methods get invoked, depending on the internal state of the controller and its virtual environment. As a consequence JIT compilation inevitably interrupts the regular execution of the program to translate new methods that are executed for the first time. In normal operation, i.e. no exceptional event occurred, the last method is compiled in iteration 46.

With our mission-start-compilation strategy we are able to completely eliminate unwanted compilation during the critical phase of a real-time task. Figure 3 shows a comparison of the execution time of the first few iterations of the control loop. The bars in black show the execution time of each iteration for the regular JIT approach, while the gray bars represent our new approach – note the logarithmic scale. As can be seen the overhead of dynamic compilation is completely eliminated. Even the first few iterations closely resemble the behavior of the application in its steady state. Because of initialization overhead contained in the Java program itself, the first iterations still show a slightly increased execution time. This behavior is inherent to the program and can not be eliminated.

7 Conclusion

From our evaluation we see that the combination of a well designed RISC core with JIT compilation of Java performs better than a Java processor directly executing Java bytecode. Especially when compared with the highly optimized, but resource hungry picoJava. In terms of FPGA resources, picoJava is about three times as big as YARI, but our RISC approach outperforms picoJava by 90%.

When comparing CACAO on YARI against JOP the speed advantage is even bigger. However, JOP was designed for time predictability and therefore omitted all architectural features (e.g., a general purpose data cache) that improve average case throughput only. When we relate performance to size, JOP is half the size of YARI and the memory consumption is lower than with CACAO.

Although JIT compilation is usually avoided in real-time Java systems we have found a way to reduce the influence of the compiler on real-time performance. JIT at mission start reintroduces some dynamics, e.g., class loading during the initialization phase, into real-time Java without compromising real-time constraints.

Availability

Our modified version of CACAO, as well as the YARI soft-core are open source projects that are publicly available. To our knowledge our system is the only available open source Java solution that includes a soft-core RISC and the software stack. We hope that the system is adopted in research and industry projects, it can be retrieved using the software code management tool *git* from: `http://repo.or.cz/w/yari.git`

Acknowledgment

We would like to thank Christian “Twisti” Thalinger, the main developer and maintainer of CACAO, for his support and insights on CACAO.

References

[1] JavaBenchEmbedded. Available at http://www.jopwiki.com/JavaBenchEmbedded.

[2] Altera Corporation. Nios II processor reference handbook, 2008. Version 8.0.0.

[3] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A real-time Java virtual machine with applications in avionics. *Trans. on Embedded Computing Sys.*, 7(1):1–49, 2007.

[4] C. Badea, A. Nicolau, and A. V. Veidenbaum. A simplified Java bytecode compilation system for resource-constrained embedded processors. In *CASES '07: Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 218–228, New York, NY, USA, 2007. ACM.

[5] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 157–164, New York, NY, USA, 1991. ACM.

[6] G. Bollella, B. Delsart, R. Guider, C. Lizzi, and F. Parain. Mackinac: Making HotSpot^{TM} real-time. In *ISORC*, pages 45–54. IEEE Computer Society, 2005.

[7] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[8] J. Caska. micro [μ] virtual-machine. Available at http://muvium.com/.

[9] M. Fulton and M. Stoodley. Compilation techniques for real-time Java programs. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, pages 221–231, Washington, DC, USA, 2007. IEEE Computer Society.

[10] J. Gaisler. A portable and fault-tolerant microprocessor based on the SPARC v8 architecture. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, page 409, Washington, DC, USA, 2002. IEEE Computer Society.

[11] T. R. Halfhill. Imsys hedges bets on Java. *Microprocessor Report*, August 2000.

[12] D. S. Hardin. Real-time objects on the bare metal: An efficient hardware realization of the Java virtual machine. In *Proceedings of the Fourth International Symposium on Object-Oriented Real-Time Distributed Computing*, page 53. IEEE Computer Society, 2001.

[13] Imsys. Im1101c (the Cjip) technical reference manual / v0.25, 2004.

[14] Java Expert Group. Java specification request JSR 302: Safety critical java technology. Available at http://jcp.org/en/jsr/detail?id=302.

[15] A. Krall and R. Grafl. CACAO – A 64 bit JavaVM just-in-time compiler. In G. C. Fox and W. Li, editors, *PPoPP'97 Workshop on Java for Science and Engineering Computation*, Las Vegas, June 1997. ACM.

[16] J. Kreuzinger, U. Brinkschulte, M. Pfeffer, S. Uhrig, and T. Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.

[17] Lattice Semiconductor Corporation. LatticeMico32 processor reference manual, 2007. Version 6.1 SP1.

[18] J. M. O'Connor and M. Tremblay. picoJava-I: The Java virtual machine in hardware. *IEEE Micro*, 17(2):45–53, 1997.

[19] W. Puffitsch. picoJava-II in an FPGA. Master's thesis, Vienna University of Technology, 2007.

[20] P. Puschner and A. Wellings. A profile for high integrity real-time Java programs. In *4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001.

[21] Red Hat, Inc. Newlib. Available at http://sourceware.org/newlib/.

[22] RTCA/DO-178B. Software considerations in airborne systems and equipment certification. December 1992.

[23] RTJ Computing. simpleRTJ a small footprint Java VM for embedded and consumer devices. Available at http://www.rtjcom.com/, 2000.

[24] M. Schoeberl. Java technology in an FPGA. In *Proceedings of the International Conference on Field-Programmable Logic and its Applications (FPL 2004)*, Antwerp, Belgium, August 2004.

[25] M. Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

[26] M. Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, 54/1–2:265–286, 2008.

[27] M. Schoeberl, H. Sondergaard, B. Thomsen, and A. P. Ravn. A profile for safety critical Java. In *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'07)*, pages 94–101, Santorini Island, Greece, May 2007. IEEE Computer Society.

[28] F. Siebert. *Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages*. Number ISBN: 3-8311-3893-1. aicas Books, 2002.

[29] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java on the bare metal of wireless sensor devices: the squawk Java virtual machine. In *Proceedings of the 2nd international conference on Virtual execution environments (VEE 2006)*, pages 78–88, New York, NY, USA, 2006. ACM Press.

[30] Sun Microsystems, Inc. phoneME. Available at https://phoneme.dev.java.net/.

[31] T. Thorn. Yet another RISC implementation. http://thorn.ws/yari, 2008.

[32] S. Uhrig and J. Wiese. jamuth: an IP processor core for embedded Java real-time systems. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 230–237, New York, NY, USA, 2007. ACM Press.

[33] J. White. An introduction to java 2 micro edition (j2me); java in small things. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 724–725, Washington, DC, USA, 2001. IEEE Computer Society.

[34] Xilinx Inc. MicroBlaze processor reference guide, 2008. Version 9.0.

[35] M. Zabel, T. B. Preusser, P. Reichel, and R. G. Spallek. Secure, real-time and multi-threaded general-purpose embedded Java microarchitecture. In *Prceedings of the 10th Euromicro Conference on Digital System Design Architectures, Methods and Tools (DSD 2007)*, pages 59–62, Aug. 2007.