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 Research
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 (JVM), 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 (JIT) compiling JVM implementation, combining high speed and small size. We have modified the original JVM to run without an underlaying operating system within only 1 MB of memory. In addition we present a new technique to selectively precompile methods during the initialization phase of real-time Java applications to prevent unwanted interaction between the JIT compilation and critical tasks. Furthermore we present the YARI softcore as the execution platform of CACAO within an FPGA. We compare our implementation with two well known Java processors, JOP and Sun's picoJava-II, on the same FPGA 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 2.2 and 1.7 respectively.

Categories and Subject Descriptors

D.3.4 [**Processors**]: [Run-time environments]; C.1.1 [**RISC/CISC, VLIW architectures**]: [RISC soft-core]; D.3.2 [**Programming Languages**]: [Java]

Keywords

Embedded Java, JIT compilation, Real-time system, Java processor

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 languages simplicity, safety, and portability. Because of these properties Java is also becoming more and more attractive to developers of embedded systems. Due to resource constraints, technologies utilized by Java Virtual Machines (JVM) on workstations are not practical for embedded systems. 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) is a widely used variant of such a restricted environment intended for use in mobile and embedded devices. JavaME 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. MIDP 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 and external components need to be controlled in a timely fashion. Interpreters and JIT compilation do not allow to 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.

A popular alternative is the use of a native Java processor to speed up the execution of Java bytecode. These processors implement a subset of the Java bytecode instruction set in hardware or microcode. More complex operations (such as object creation) are emulated using a software layer.

In this paper we present a just-in-time compiling JVM solution for small embedded systems. We have adapted the CACAO research JVM [10] to run without an underlaying operating system in an environment offering only 1 MB of memory. Based on the execution model of Safety Critical Java [9, 22], we propose a new technique called *mission-start-compilation*. Critical methods are precompiled upon transition to the mission phase, eliminating unwanted interference of the JIT compilation process with real-time tasks. Furthermore, we present YARI, a soft-core RISC processor

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 GNU build tools, the Newlib C library, the phoneMe Java class library, the CACAO JVM, and the YARI soft-core. 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 RISC 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 and discuss future work 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 sections gives a brief overview of the most relevant work in each area.

2.1 Java for Embedded Systems

A detailed performce comparison of embedded Java systems can be found in [19]. This paper also describes the benchmark we use in our evaluation in more detail.

SimpleRTJ [17] is a JVM intended for small embedded systems. SimpleRTJ is an interpreting JVM and requires about 18–24 KB of memory to run. In [2] a lightweight JIT compilation system, targeted for resource-constrained environments, is presented.

The Squawk VM [23] is an embedded JVM mostly written in Java that is now open source[1]. Squawk was originally developed for a wireless sensor platform based on ARM, the Sun SPOT[2]. It runs on the *bare metal* and provides functionality typical found in operating systems, e.g., device drivers, in Java.

Muvium [4] is an ahead-of-time compiling JVM solution for very resource constraint microcontrollers (Microchip PIC). Muvium introduces an Abstract Peripheral Toolkit (APT), that contains a large collection of classes to represent devices common in embedded systems.

2.2 Java Processors

Sun introduced the first version of picoJava [13] in 1997, although this processor was never released as a product by Sun. A redesign followed in 1999, known as picoJava-II that 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 stand alone processor [7]. It is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins.

The Cjip processor [6, 8] 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 [11] is a multithreaded 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 [25].

JOP [21] 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 [27] is a new Java processor based on the architecture of JOP and enhanced by a hardware garbage collector.

2.3 FPGA RISC Soft-cores

As FPGAs have grown in size and capabilities, it has increasingly proven beneficial to employ microprocessors as part of the design to handle less time-critical state machines. The major FPGA vendors offer various solutions for this, including embedding one or more microprocessors in the FPGAs, for example, the Virtex 4 FX which includes PowerPC^{TM}hard-cores.

Highly configurable and optimized soft-cores are also offered by FPGA vendors. Among these 32-bit soft-cores, probably the best known are MicroBlaze from Xilinx [26], Nios II from Altera [1], and Mico32 from Lattice [12]. Nios II and MicroBlaze are proprietary, whereas Mico32 is available under an open source licence. All of these are supported by complete development kits with compilers, libraries, and debuggers.

Besides the soft-cores just mentioned, there are a large number of open sourced soft-cores. We cannot possibly cover all of them, but instead restrict ourselves to one of the better known: LEON. LEON [5] is an implementation of the SPARC V8 architecture. LEON, implemented on the same FPGA board we use for YARI, consumes about 8,000 LCs,[3] 11 KB on-chip memory and can be clocked at 35 MHz. Initially designed with for the purpose of radiation hardened implementations, LEON has been released under an open source licence.

3. YARI

YARI (Yet Another RISC Implementation), is an open source [24] 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^{TM}-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^{TM}architecture, designed for VLSI, may not be an optimal architecture for FPGAs.

[1] `https://squawk.dev.java.net/`

[2] `http://www.sunspotworld.com/`

[3] The basic resource in Altera FPGA is the Logic Cell (LC), which essentially is a four-bit lookup table and a registers.

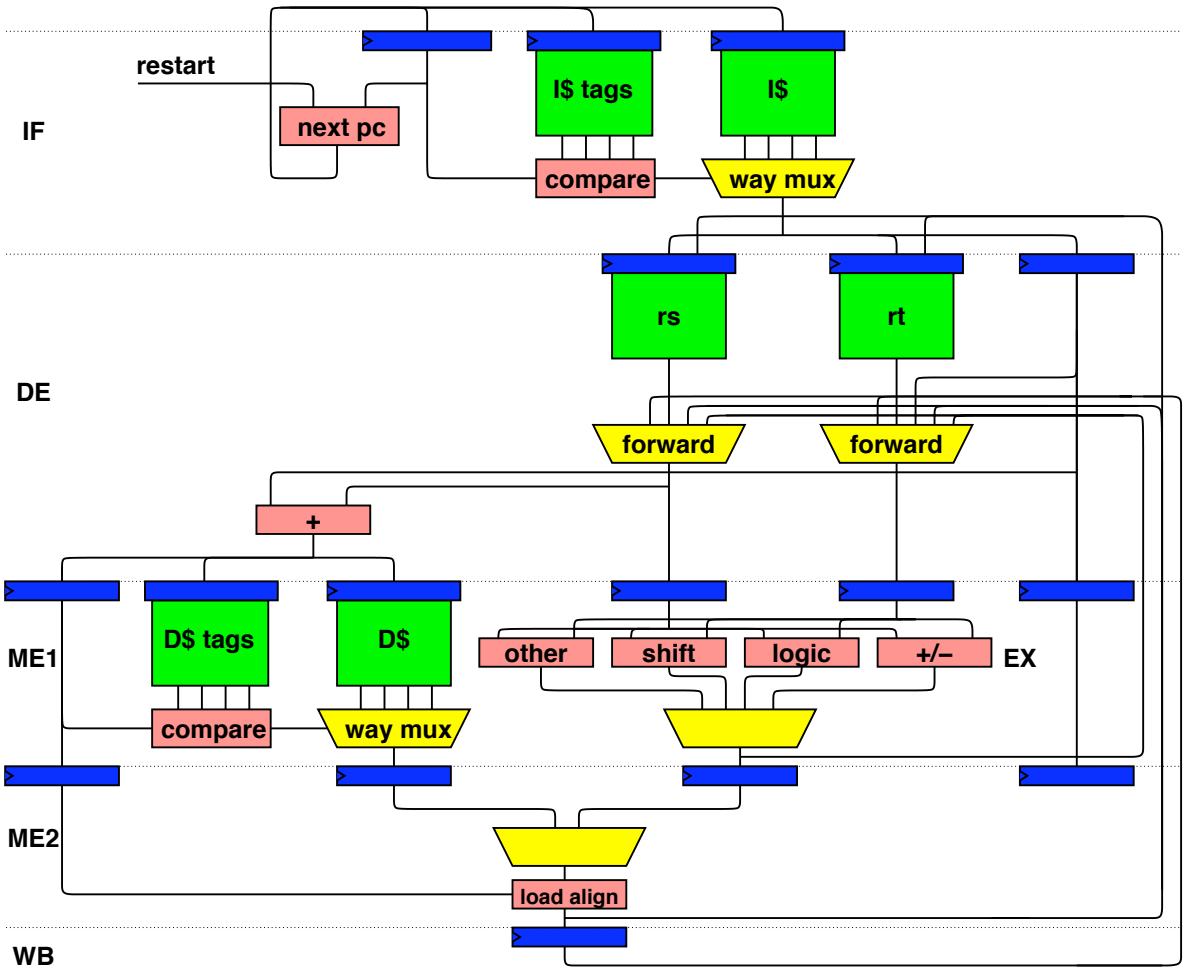

Figure 1: The YARI data path

From earlier experiments we have found that pipeline stalling can have surprisingly complicated interactions with branch delay slots and pipeline restarts, leading to hard to find bugs. 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 depending on the nature of the hazard. While this can result in more pipeline bubbles than stalling, the resulting simpler logic leads to a shorter cycle time, and thus, higher frequency.

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.

3.1 Simulation and Co-Simulation

The development of YARI has relied extensively on simulation and co-simulation. A simple architectural interpreter was developed and maintained as a "golden reference model" for the FPGA implementation. Any new feature was first implemented in the architectural simulator and the software was tested there.

This effort has payed back in countless ways. It is much easier and simpler to test and debug the software on the interpreter, which can be instrumented to catch events of particular interest.

For performance enhancements it has proven paramount to first extend the interpreter to collect statistics. For example, planned work on a better (but expensive) cache replacement policy was abandoned when simulation revealed an insignificant miss rate on a large number of benchmarks.

However, by far the most important benefit of the interpreter has been its use in co-simulation. By simulating the YARI FPGA implementation in parallel with running the same workload on the interpreter and checking that the two agree on committing operations we can pin-point bugs in the FPGA implementation. The vast majority of bugs have occurred within a handful of cycles of where the divergence was detected.

The use of co-simulation has enabled us to locate bugs very quickly and effortlessly and has eliminated the need for a battery of directed tests, something known to be very time consuming (and tedious) to develop and maintain.

For the committed state we only look at the writes to user registers. As all control decisions in MIPS are register based, state changes outside the registers and memory tends to propagate to registers fairly quickly. Once a divergence is found, the last part of the FPGA simulation output is emitted together with the expected result.

Workload	Instructions	Cycles	CPI	Branches (%I/%C)	Load-use hazards (%I/%C)	I$ miss	D$ miss
Sieve	192,004,736	315,009,920	1.64	29,412,650 (15.3%/ 9.3%)	29,516,077 (15.4%/28.1%)	274,959	80,351
Kfl	192,733,376	301,150,976	1.56	37,074,095 (19.2%/12.3%)	21,914,081 (11.4%/21.8%)	651,340	117,646
UdpIp	204,952,816	332,401,024	1.62	33,646,768 (16.4%/10.1%)	26,763,089 (13.1%/24.2%)	1,155,853	115,345
Lift	186,618,112	304,367,840	1.63	31,802,817 (17.0%/10.4%)	27,395,432 (14.7%/27.0%)	316,685	95,253
Micro	2,765,275,264	3,955,059,056	1.43	477,021,107 (17.3%/12.1%)	243,378,471 (8.8%/18.5%)	409,956	113,305

Table 1: Source of pipeline inefficiencies on five CACAO benchmarks.

3.2 The Pipeline

YARI has a mostly classic five stage pipeline: instruction fetch (IF), instruct decoding/register files access (DE), execution / memory (EX/ME1), memory (ME2), and write back (WB). Unlike in VLSI technology, adders in FPGAs are fast and muxes slow, thus it proved advantageous to perform the address calculation in the decoding stage, enabling the slow sub-word extraction and optional sign-extension for the different MIPS load instructions to occupy a full cycle. Figure 1 shows the essential data path in YARI pipeline.

Unlike VLSI, where adding one input to a gate generally carries a very small fixed incremental cost, logic structures in FPGAs are built from four-input lookup tables (LUT4). Thus, adding an input could (ignoring routing resources) range from being free to adding an additional level of lookup tables. Thus, as a rule of thumb we try to minimize the number of inputs to logical expressions. As a consequence, 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.

3.3 Pipeline Control

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 (CPI) metric, thus inversely proportional to the observed performance.

We can distinguish between *intentional* hazards, e.g., branches, and the remaining *unintentional* hazards. Restarts for intentional hazards are issued from the EX stage. Thanks to the branch delay slot, only the IF stage needs to be flushed, in other words, YARI handles all branches with one cycle penalty.

There are eight unintentional hazards, four of which are directly related to memory and are generally unpredictable: *instruction cache miss*, *data cache miss*, *store buffer full*, and *branch delay slot cache miss*. The latency of the cache fill depends on memory contention and memory speed. For the platform used for this paper, we can fill the 16-byte cache lines in about 10 cycles when the full memory bandwidth is available. Restarting a load can further add up to three cycles of additional latency.

The *branch delay slot cache miss* hazard can occur when the instruction in the delay slot of a taken branch misses in the instruction cache. If left as is, the branch would be taken, the pipeline flushed and, once the cached had serviced the miss, restarted from the target address without the delay slot ever being executed. If we could stall the pipeline until the cache is ready, then this would not be an issue. Instead we simply restart the branch every time this hazard is encountered. The penalty for this is at most three cycles, but it is a relatively rare hazard.

The remaining four unintentional hazards are *multiplier/divider structural hazard*, *synci*, *load-hit-store*, and *load-use*.

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.

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. Handling this flush has a five cycle penalty.

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 ME2 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[4] that we instead trade off the occasional pipeline restart for a simpler data path.

Finally, the *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 EX and has a two cycle penalty. GCC's instruction scheduler knows about this hazard and never generates code that violates it. Unfortunately, CACAO doesn't respect this load-use hazard resulting in the number one source of pipeline inefficiencies.

Table 1 shows the behavior on five workloads, including the number of branches observed, load-use hazards, instruction cache misses, and data cache misses. For branches and load-use we also show these numbers in terms of fraction of overall instructions and cycles (remembering the one-cycle penalty for branches and the three-cycle penalty for a load-use). For example, the Sieve benchmark spends 28.1% of all cycles dealing with restarts due to load-use hazards. We will return to the pipeline efficiency in the conclusion.

4. CACAO

CACAO [10] 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, such as the Eclipse platform and the Tomcat application server. 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 un-

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

derlaying operating system and enable CACAO to run in a minimal execution environment on top of the YARI soft-core.

4.1 Just-in-Time Compilation

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 an intermediate representation (IR), that is better suited for further processing than the stack-oriented Java bytecode. This internal representation is register-oriented similar to a simple RISC machine. In the next step stack slots containing intermediate results are converted to virtual registers. Mapping stack slots to registers is straightforward for basic blocks. However, on control flow splits and joins copy operations need to be inserted, to correctly reflect the state of the stack at basic block boundaries. The register allocation phase maps the virtual registers to machine specific registers. During a linear traversal of the IR, first virtual registers corresponding to intermediate results on the stack are assigned, then spare registers left over are assigned to local variables. 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 IR to instructions of the target machine. It is important to note that all these phases are at most linear in runtime. More information on the internals of CACAO's JIT compiler can be found in [10].

Because JIT compilation is relatively expensive, methods are compiled on demand, i.e., only when a method is to be executed the first time. Similarly, 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 startup phase of a program.

In fact it is hard to predict beforehand when the JIT compiler is actually invoked, making timing analysis for JIT based systems impossible. To overcome this limitation we propose a new technique called *mission-start-compilation*, that allows to precompile critical parts of a program, in order to prevent JIT compilation to interfere with real-time tasks. Details on this approach are presented in Section 5.

4.2 A Minimal Execution Environment

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 safe memory resource and lower the hardware requirements.

The Newlib project by RedHat[5] offers a minimal, but complete, C library implementation that allows to run programs without an underlaying operating system on top of a bare processor. It specifically targets small embedded devices and thus has minimal prerequisites. Porting Newlib to YARI required only marginal modifications of the well supported MIPS, more specifically *mips-elf*, port.

A fully compliant implementation of all standard I/O operations (e.g., open, close, read, write, printf, etc.) is available. By default most of these functions do not contain a useful implementation, and merely report well defined I/O errors. In order to perform basic I/O, a minimal implementation to redirect the standard file streams *stdin*, *stdout*, and *stderr* to the serial line has been implemented. Other file operations will still fail, they are currently not required anyway, because CACAO has been extended to load the Java class files from internal buffers, as we explain later. To safeguard against heap overflow the default implementation of the *sbrk* system function has been replaced. Time measurement is available through the standard C routine *times*, which reads the cycle counter of YARI and reports the elapsed time since the applications startup in milliseconds.

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. Nevertheless all critical functions are reentrant, which allows Newlib to be used in a multithreaded environment. Similarly, signal handling is not provided. Among other tasks CACAO relies on signals to invoke the code generator, the garbage collector, and to handle traps and Java exceptions.

In addition to standard C library functions CACAO extensively utilizes additional library, operation system, and architecture specific features. For example, CACAO heavily relies on memory protection features available on standard operating systems. Unfortunately no such functionality is available with Newlib, memory protection is not supported by YARI either. Thus, checks for null pointers, array overflow, and similar faults need to be implemented using a software emulation. The dynamic generation of code in combination with separate data and instruction caches necessitates a minimal set of operations to invalidate contents of the instruction cache that became stale. These functions are highly architecture and operating system specific. YARI specific support functions have been added to Newlib accordingly.

4.3 Extensions to CACAO

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 JavaSE 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. Resolving these functions by name is performed via a lookup table that is statically determined during the build process from symbol information of the intermediate objects files. Similarly, classes required during the system startup 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 2 depicts the necessary steps to embed a Java application into CACAO and download it to YARI.

[5] http://sourceware.org/newlib/

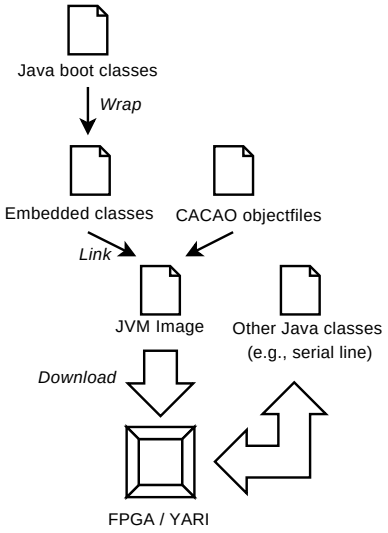

Figure 2: Downloading a Java application to YARI.

4.4 Java Library

The rich Java class library is, in combination with the language features, a corner stone of Java's success story. In contrast to other object oriented languages (e.g., C++), Sun early established a rich set of functions, classes, and infrastructure. Today the term Java is often used interchangeably to denote the Java language and the Java class library. In the domain of embedded systems the development of a common platform did not receive broader attention up until the recent past. Even Sun's effort, JavaME, is a mere conglomerate of different technologies (CDC, CLDC, MIPD, etc.) that still lack homogeneity.

CACAO is already designed to support different Java class libraries. The popular GNU Classpath project is enabled by default, but also the open source JavaSE and JavaME implementations, OpenJDK and phoneME by Sun, are supported out of the box. Although theoretically all three class libraries are ready to run with CACAO, the much smaller size of the phoneME package lends itself for embedded systems. 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 Specificaton for Java (RTSJ) [3] and JSR 302 on Safety Critical Java [9, 22]. Although we strive for compatibility some features are not fully compliant with the JavaME platform:

Java Native Interface Although the JavaME platform explicitly excludes all JNI functionality, our system allows to retain all the JNI functionality available with CACAO on workstation systems.

Garbage Collection The widely adopted Böhm garbage collector is used to reclaim memory. Because of its large size, it is not feasible to make use of Böhm in a resource constrained system, thus the garbage collector is currently disabled. Development of a replacement for the current garbage collector, which is expected to be considerable smaller, is still in progress.

Multithreading As noted in the previous section, Newlib does not provide any process management functionality, multithreading in Java programs is thus disabled. Prior versions of CACAO implemented *green threads*, i.e., threads are managed by a software layer within CACAO, to work around this issue. Unfortunately in current versions of CACAO green threads are not functional anymore. Because of this deficiency, multithreading support is currently not available.

5. JIT FOR REAL-TIME SYSTEMS

JIT compilation is usually avoided for real-time systems due to its unpredictability. An exact analysis of bounds for the execution time of the original Java program is in general impossible, because it is uncertain when the JIT compilation 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. Some systems even contain several optimization levels for the compilation, i.e., methods that are already compiled, but still dominate the execution time, may be selected for *recompilation* to further improve performance. Especially in the case of mixed interpretation and compilation, and multiple optimization levels it is hard for an offline timing analysis to predict the code that actually will be executed, which further complicates timing analysis.

As shortly described in Section 4.1, CACAO follows a compile-only approach, eliminating some of the problems beforehand. The code generation scheme is largely based on simple macro expansion, in addition CACAO does not offer multiple optimization levels. Recompilation may still occur in rare cases, if inlining decisions become invalidated through dynamic class loading. It is thus relatively easy to predict basic properties of the code that will actually be executed. Therefore, if we can tolerate the overhead induced by JIT compilation during the warmup 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 (level 1) [15, 22] 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 [16], with our CACAO/YARI system. We just *borrow* the concept for less demanding real-time applications.

We propose and have implemented 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 start mission – 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 precompiled using the regular compiler. JIT overhead during the mission phase is completely eliminated, allowing a more accurate timing analysis.

Precompiling Java programs offline is well known, and usually referred to as ahead-of-time (AOT) compilation. Systems relying on AOT 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 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 AOT compilation, selectively downloading individual classes is not possible. Instead the complete application binary needs to be downloaded, stored it into the systems flash

Soft-Core	Logic Cells	Memory	Frequency
JOP	3,300	7.6 KB	100 MHz
YARI	7,008	18.9 KB	75 MHz
pico-Java-II	27,560	47.6 KB	40 MHz

Table 2: FPGA synthesis results of the JOP, YARI, and picoJava-II soft-cores.

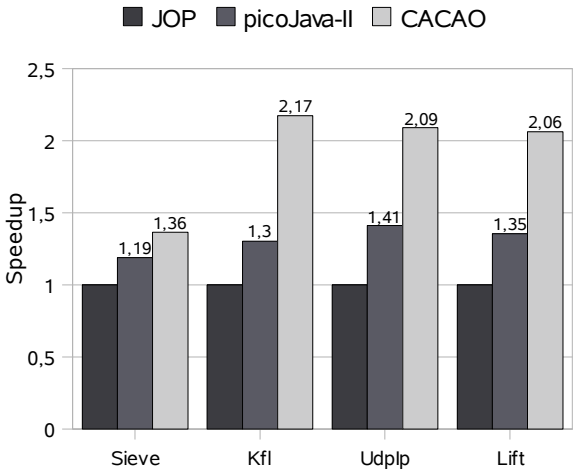

Figure 3: Performance of picoJava-II and CACAO/YARI for some embedded benchmarks, normalized to JOP.

memory, and an additional reboot performed in order to acquire updates.

6. EVALUATION

In this section we provide a first evaluation of the combination of CACAO and YARI within an FPGA. We show execution performance on a set of embedded Java benchmarks (JBE)[6] and some micro-benchmarks. We compare the obtained results to two Java processors, namely JOP [18] and an FPGA implementation of Sun's picoJava-II [13, 14]. All three soft-cores are synthesized 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 2 lists the basic properties of the synthesized soft-cores. The FPGA is integrated on an evaluation board offering 1 MB of external 32-bit asynchronous SRAM with 15 ns access time. The board also uses a serial line for program download and used by *stdout* and *stdin* during program execution.

All benchmarks where executed on the same platform, with all required Java classes readily available in the systems memory. In the case of CACAO all required classes where embedded into the executable binary. This allows for an interesting comparison between the two Java processors and a compiling Java solution running on a RISC.

[6]JavaBenchEmbedded V 1.1 available at `http://www.jopwiki.com/JavaBenchEmbedded`

Bench.	JOP	picoJava-II	CACAO
Sieve	6496	7721	8861
Kfl	18275	23813	39742
UdpIp	8467	11950	17702
Lift	18649	25444	38437

Table 3: Number of iterations per second for the JBE application benchmarks. A higher number means faster.

Micro-Benchmark	JOP	picoJava-II	CACAO
iload3 iadd	2	2	1
iinc	4	3	2
ldc	9	3	5
if icmplt taken	6	6	4
if icmplt not taken	6	-	3
getfield	16	3	5
getstatic	15	5	9
iaload	11	3	11
invoke	128	24	15
invokestatic	100	24	14
invokeinterface	144	196	19

Table 4: Cycles required to execute specific Java bytcodes for JOP, picoJava-II, and CACAO/YARI.

6.1 Performance

Our new solution, based on CACAO running on top of YARI, offers the best performance compared to the two other options. Figure 3 shows the speedup of picoJava-II and CACAO/YARI relative to JOP. Although JOP achieves the highest frequency in our setup, the results show the least performance for the four benchmarks of the JBE suite. For all our tests the CACAO JVM is faster by a factor of 1.36 to 2.17 in comparison to JOP. CACAO is also able to outperform the picoJava-II, resulting in a speedup of a factor between 1.15 and 1.67. Absolute iteration counts are provided for reference in Table 3.

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 4. 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 efficient, 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 such an inaccuracy.

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

Benchmark	Classes	Code	Data	Heap	Total
Sieve	56,861	346,584	81,223	230,533	715,201
Kfl	72,526	346,584	80,990	267,074	767,174
UdpIp	69,059	346,584	81,465	267,796	764,824
Lift	62,167	346,584	81,117	251,159	741,027

Table 5: Memory consumption in bytes for class files, static code, data, and heap.

Bench.	JIT	Methods	Bc-instr. (size)	Mips-instr. (size)
Sieve	94	78/405	1,421 (3,419 B)	5,609 (22,436 B)
Kfl	153	117/465	2,913 (6,577 B)	8,342 (33,368 B)
UdpIp	136	107/455	2,404 (5,443 B)	8,327 (33,308 B)
Lift	111	89/422	2,005 (4,522 B)	6,893 (27,572 B)

Table 6: Statistics on JIT compilation, showing the number of JIT invocations, methods compiled, bytecode instructions translated, and the number of resulting MIPS machine instructions.

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 5 summarizes the amount of memory required to hold the Java classes of the benchmark, static code and data of the CACAO JVM, and finally the peak amount of heap memory allocated at runtime. In addition Table 6 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, and thus, the fraction of dynamically generated code is 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.

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 and controls a set of sensors and actuators of the environment, that is also simulated in Java. The test executes a 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.

For soft real-time systems the JIT overhead may be tolerated during a warmup phase, even if some deadlines are missed – our example application is a control loop that tolerates and regulates disturbance from the environment anyway. However, in general this can not be accepted. Even after the warmup phase compilation may be necessary, e.g., in the case of exceptional events and failure situations. JIT compilation of error handlers can not be tolerated in critical systems.

With our mission-start-compilation strategy we are able to completely eliminate unwanted compilation during the critical phase of a real-time task. Figure 4 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 – please 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 AND FUTURE WORK

From our evaluation we see that the combination of a well designed RISC core with JIT compilation of Java performs better than a processor designed for Java. 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 30%.

When comparing CACAO/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 only average case throughput. Execution time of most bytecodes can be estimated cycle accurate [20] for JOP. When we relate performance to size, JOP is half the size of YARI and the memory consumption is lower than with CACAO. However, even a dual core version of JOP will not be as fast as CACAO/YARI as speed scales usually logarithmic with hardware resources.

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.

While the performance of CACAO/YARI is already very good, there are clear avenues for improvements. Work is already progressing on the next version of YARI with a focus on cycle time improvements and lower penalty for branch and load-use hazards. The expected performance improvement for the Java based benchmarks is 44%.

The open source stack from the processor up to the JVM provides opportunities for future work. For example, Java specific extensions of the instruction set of YARI would allow major improvements. The original RISC design was optimized for procedural languages such as C and not for object oriented and managed languages such as Java or C#. From the micro-benchmarks we notice quite an overhead for field and array access. Both operations have to perform a null pointer check, the array access in addition also needs to check for array bounds. Dedicated, trapping instructions would clearly allow to improve the overall performance.

CACAO already produces simple sequences of RISC instructions for fast compilation. Therefore, with the detailed knowledge of the YARI pipeline, it should be possible to analyze those sequences for worst-case execution time (WCET). It should even be possible to estimate execution times for the instruction sequences that represent bytecode instructions, which would eventually allow to perform WCET analysis at bytecode level.

In its current state some major features, such as multithreading and garbage collection, are disabled. Thus future work on CACAO for embedded systems will also concentrate on providing a more compliant JavaME implementation.

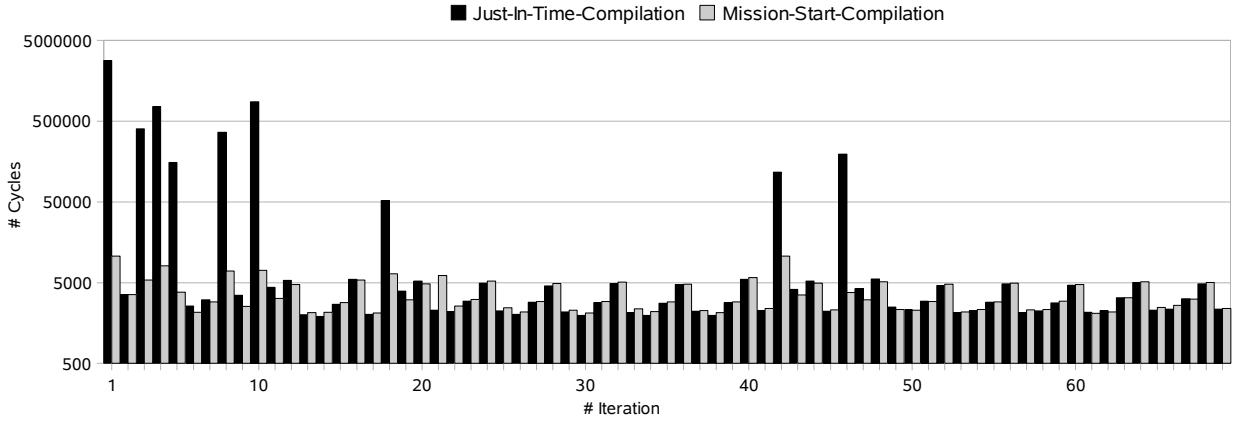

Figure 4: Execution time per iteration for just-in-time-compilation vs. mission-start-compilation over time

The modified version of CACAO as well as the YARI soft-core are available 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.

8. REFERENCES

[1] Altera Corporation. Nios II. `http://www.altera.com/products/ip/processors/nios2/ni2-index.html`, 2008.

[2] 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.

[3] 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.

[4] J. Caska. micro [μ] virtual-machine. http://muvium.com/.

[5] 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.

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

[7] 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.

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

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

[10] 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.

[11] 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.

[12] Lattice Semiconductor Corporation. Mico32. `http://www.latticesemi.com/products/intellectualproperty/ipcores/mico32/index.cfm`, 2007.

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

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

[15] 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.

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

[17] RTJ Computing. simpleRTJ a small footprint Java VM for embedded and consumer devices. online at http://www.rtjcom.com/.

[18] 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.

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

[20] M. Schoeberl. A time predictable Java processor. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE 2006)*, pages 800–805, Munich, Germany, March 2006.

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

[22] 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.

[23] 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.

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

[25] S. Uhrig and J. Wiese. jamuth: an ip processor core for embedded java real-time systems. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 230–237, New York, NY, USA, 2007. ACM Press.

[26] Xilinx Corporation. MicroBlaze. `http://www.xilinx.com/products/design_resources/proc_central/microblaze.htm`, 2008.

[27] M. Zabel, T. B. Preuber, 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.