Java Dust: How Small Can Embedded Java Be?

Caska, James; Schoeberl, Martin

Published in:
Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2011)

Link to article, DOI:
10.1145/2043910.2043931

Publication date:
2011

Java Dust: How Small Can Embedded Java Be?

James Caska
Muvium
jamescaska@Muvium.com

Martin Schoeberl
Department of Informatics and Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

ABSTRACT

Java is slowly being accepted as a language and platform for embedded devices. However, the memory requirements of the Java library and runtime are still troublesome. A Java system is considered small when it requires less than 1 MB, and within the embedded domain small microcontollers with a few KB on-chip Flash memory and even less on-chip RAM are very common. For such small devices Java is a clearly challenging. In this paper we present the combination of the Java compiler Muvium for microcontrollers with the tiny soft-core Leros for an FPGA.

To the best of our knowledge, the presented embedded Java system is the smallest Java system available. The Leros processor consumes less than 5% of the logic cells of the smallest FPGA from Altera and the Muvium compiler produces a JVM, including the Java application, that can execute in a few KB ROM and less than 1 KB RAM. The Leros processor is available in open-source and the Leros port of Muvium is freely available.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.4 [**Programming Languages**]: Processors—*Compilers*

1. INTRODUCTION

There is a clear trend in embedded computing that divides computing devices into two basic categories, Linux enabled and sub-Linux systems. The range and cost point of Linux enabled devices is growing very quickly and for those devices Java J2SE running on Linux is a compelling option. For sub-Linux systems, application specific microcontroller systems, there are fewer choices for running Java and the requirements of real world applications change quite substantially. To compete effectively in this segment a Java compiler for embedded microcontrollers must handle extremely limited resources, work without an operating system, interface directly with the real-world, and compete effectively with the C language. Performance and code-density are perhaps the most critical factors for an embedded processor. Due to its interpretive nature the performance of Java has always been questioned. Mainstream JVMs employ just-in-time technologies to address this. For embedded systems, just-in-time compiling is hardly possible due to the memory consumption of the compiler. However, since the applications we target are dedicated static applications, ahead-of-time compilation is the approach to reduce resource consumption.

This ahead-of-time compilation approach is used by Muvium to target Java to highly resource constrained embedded devices. To enable Java for small microcontrollers we impose several restrictions to Java: only a small subset of the Java library (JDK) is available and in the default configuration of Muvium, Java integers are represented as 16-bit words.

In this paper we present a minimal embedded Java system consisting of the tiny microcontroller Leros [7] and the ahead-of-time compiler Muvium [2], which is optimized for processors with minimal memory resources. To the best of our knowledge, this is the smallest Java system available today — a first step toward bringing Java into very small systems and building up Java Dust.

2. RELATED WORK

The Squawk VM [8] is a platform for wireless sensors, targeting the embedded processor ARM9. Squawk is mostly written in Java and runs without an OS. Compared to Leros/Muvium, Squawk still needs a considerable amount of Flash and RAM. A stripped down Squawk runtime fits into 256 KB Flash.[1] Another approach for small embedded Java is the CACAO JVM [4], which has also been optimized for a small footprint. It is possible to run the CACAO JIT compiler for the MIPS target and a simple Java application within 1 MB memory [1]. This footprint figure for the well performing CACAO JIT is impressive, but targets a different class of embedded devices than our approach.

SimpleRTJ[2] is a small JVM interpreter, targeting embedded processors. SimpleRTJ requires 18-24 KB of ROM. However, it interprets Java bytecodes and it can easily take several thousand cycles to execute e.g. a field operation on a 16-bit controller.

KESO is another small JVM targeting microcontrollers [9]. KESO targets the automotive domain with a backend for OSEK/VDX. It is an ahead-of-time compiler that takes bytecodes as input, similar to Muvium. However, KESO generates C code instead of directly assembler code. This gives the full range of C based optimization, but the C compiler might miss some optimization potential from lost knowledge of the original bytecode. Muvium in contrast can optimize the stack layout, as it is given by the bytecode, to the available resources in the microcontroller.

JTRES 2011 September 26-28, 2011, York, UK

[1] Private communication with Rasmus U. Pedersen
[2] `http://www.rtjcom.com/`

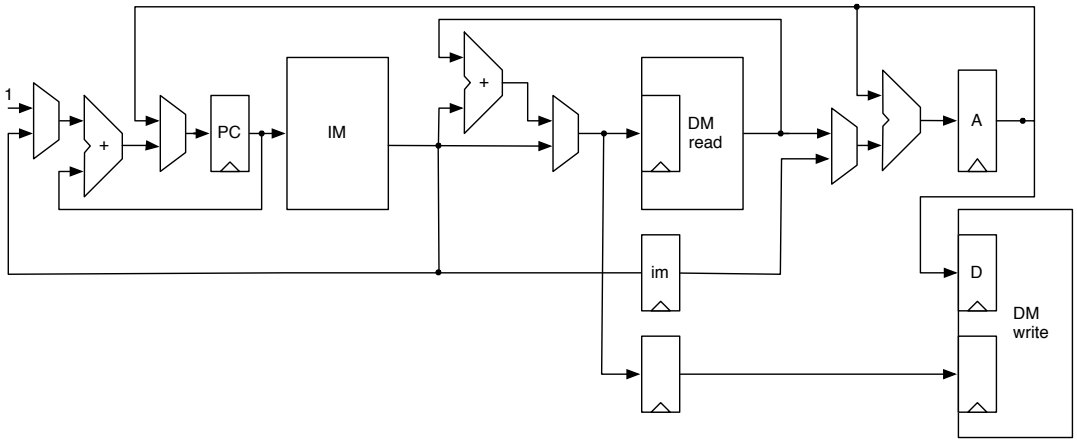

Figure 1: Two stage pipeline of Leros

Many small processors are available, which target FPGAs. We pick two examples and point out the differences to our Leros design. PicoBlaze is an 8-bit microcontroller optimized (and restricted) for Xilinx FPGAs [10]. This optimization results in restrictions such as maximal program size of 1024 instructions and 64 byte data memory. Compared to PicoBlaze, Leros has fewer restrictions on program and data size, as it is a 16-bit processor.

The SpartanMC is a small microcontroller optimized for FPGA technology [3]. The processor is a 16 register RISC architecture with two operand instructions and is implemented in a three-stage pipeline. Compared to the SpartanMC, Leros is further optimized for FPGAs using fewer resources. Leros simplifies the access to registers in on-chip memory by implementing an accumulator architecture instead of a register architecture.

3. THE LEROS PROCESSOR

The Leros processor [7] is a microcontroller optimized for low-cost field-programmable gate arrays (FPGAs). It is a 16-bit processor intended for utility functions in an FPGA based System-on-Chip (SoC) design. The design goals of Leros are a good balance between the number of logic cells and on-chip memories, reasonable performance, and a high maximum clock frequency.

The architecture, which follows from the design goals, is a pipelined 16-bit accumulator processor with additional directly addressable *registers* in an on-chip memory for local variables. Only a single dedicated register (the accumulator) is connected to the ALU output and provides one input to the ALU. To provide fast data locations, similar to a register file, the first 256 words in the on-chip data memory can be directly addressed for an ALU operation. The on-chip data memory is shared for those registers and general data. With an additional on-chip memory for the instructions only two memory blocks[3] are needed, and the pipeline can execute one instruction per clock cycle.

The basic building blocks in current FPGAs are logic cells (LC), on-chip memories, and DSP blocks. For a utility processor we are interested in the optimal relation between logic cell and on-chip memory consumption. On-chip memory in FPGAs is organized as fixed-sized blocks with a configurable data and address width. To optimize a tiny processor core we evaluated the relation of on-chip memories to logic resources on current low-cost FPGAs.

We have compared recent low-cost FPGA families from Altera and Xilinx [7]. For medium size and large FPGAs of the Altera Cyclone and Xilinx Spartan-6 series the relation between LCs and memory blocks stays in the range of 200 to 400 LCs per memory block independent of the device size. Therefore we conclude that the sweet spot for a Leros in current FPGAs is around 300 LCs per on-chip memory block.

Leros is named after the Greek island Leros,[4] where it was designed during an enjoyable vacation. Leros is available under open-source from `https://github.com/schoeberl/leros`.

3.1 The Pipeline

Leros is implemented in a two-stage pipeline with following visible architectural state: the program counter (PC), the accumulator register (A), the instruction memory (IM), and the data memory (DM). Figure 1 shows the pipeline of Leros (slightly simplified). The DM is shown twice as it is read in one pipeline stage and written in a different one. Register A is the accumulator and PC the program counter.

In the first pipeline stage, instructions are fetched from the on-chip IM and decoded. Decoding the few instructions is simple, so an additional decode stage is not needed. In the second stage, operands are read from the DM and the ALU operation is performed. The result is placed in the accumulator. Similar to the first stage, the 16-bit ALU operation is fast enough to perform it in the same stage as reading from DM.

The read and write address of the DM is either a constant from the instruction (for the on-chip registers) or an indirection via the DM plus an offset (for loads and stores). The write data for the DM is either A for store instructions or the PC for a jump-and-link instruction that saves the PC in a register.

As the data memory is shared for registers and general data, load and stores are implemented by two instructions. With the first instruction the address for the register is sent to the DM. The following instruction uses the value of the DM (the register content) and adds an offset, which is part of that instruction, to form the effective address. The data word to be written is provided by A; the result of a load is stored into A.

For on-chip memories with independent read and write ports the question arises what happens on a concurrent write to and read from the same address in the same cycle. There are three options: (1) read the newly written value, (2) read the old value, or (3) undefined. For option (2) and (3) a read following a write to the same

[3] Very small programs can even be implemented using logic for the instruction memory.

[4] `http://www.lerosisland.com/`

register, as shown in the following code, will not deliver the *expected* result.

```
store r1
load r1 // old value of r1 or undefined
```

However, for an accumulator machine this behavior is not a big issue. The last value written to the DM is still in register A and can be reused when needed by the next instruction.

3.2 I/O Interface

The interface to I/O components is via an 8-bit I/O address, 16-bit input and output data, a read, and a write signal. This interface allows easy attachment of I/O components. Simple I/O ports, such as switches and LEDS, can be attached with a few lines of VHDL code. More complex components, such as serial interfaces, are available in open source (e.g., at `http://opencores.org/`). For `System.out` and `System.in` we have adapted the UART from the JOP project [6] for the simplified I/O interface.

3.3 Instruction Set

Leros is a 16-bit architecture with 16-bit data and instructions. In an accumulator design the addresses of one source operand and the destination are implicit. Therefore, only one operand (address) needs to be encoded in an instruction. Furthermore, this relaxed instruction encoding allows for 8-bit immediate values in the instruction. Register naming convention is `rn`, where `n` is between 0 and 255.

The common ALU operations, such as addition and logic operations are supported with one operand from the register area (256 words) in the data memory or with an 8-bit immediate value. The following code snippet shows adding the content of register `r1` to the accumulator and masking bit zero with an immediate `and` operation.

```
add r1
and 1
```

The data memory can be accessed via indirect loads and stores. As described before, the sharing of the on-chip memory for the data memory and the registers results in a 2 cycle operation for a load or store. In the following example the content of register `r3` is loaded into the temporary address register (`ar`), and in the following load instruction the memory at offset 1 from the address register is loaded into to accumulator.

```
loadaddr r3
load (ar+1)
```

Conditional and unconditional branches use 8 bit relative offsets. Longer jump destinations, function calls, function returns, and computed jumps are supported by a jump-and-link (`jal`) instruction. The `jal` instruction jumps to the address contained in the accumulator and stores the PC into one of the 256 register.

4. COMPILING FOR LEROS

Muvium is a compiler that generates natively executable binaries directly from Java class files. Muvium is also a linker, it generates code from the source class files and links them with the Muvium libraries. Executable binaries are minimized by only linking in libraries that are referenced and only those parts of the libraries that are potentially used.

4.1 The Compilation Flow

Figure 2 shows the full compilation flow from the Java source down to the configuration of the FPGA. The compilation flow is a little bit more complex, as the Java application is finally encoded in VHDL files. That means that an application specific processor is generated in order to consume just the needed logic resources.

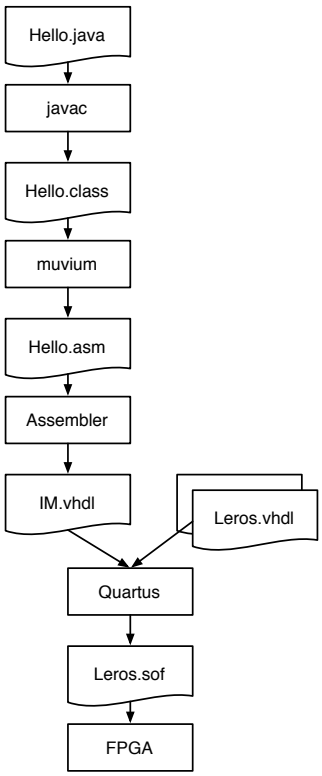

Figure 2: Compilation flow for Java on Leros

The application in Java is compiled via a standard Java compiler to Java class files. The class files are the input to the Muvium compiler, which generates a single assembler file. The Leros assembler generates a memory definition (the instruction memory) in the form of a VHDL file. The Leros VHDL sources and the instruction memory are compiled by a FPGA synthesis tool (Altera Quartus or Xilinx ISE) and generate the bitstream file that is used to configure the FPGA.

4.2 Compilation Techniques

To produce a tight executable, several bytecode specific optimizations are applied in the Muvium compiler. Muvium currently targets two microcontrollers: the Microchip PIC and Leros.

Muvium employs static frames, that is to say frames reside directly in the processor registers. To achieve this the stack operations are unwound into a series of register operations by predicting which slots are involved in any operation. For example with an add operation, normally a JVM will perform the operation `push(pop() + pop())`, Muvium will track the current stack depth as N and perform the operation `RegisterN-1 = RegisterN + RegisterN-1`. To achieve this, devices with large register sets are desirable. In Leros the first 256 words of memory can be addressed as registers. Therefore, the whole frame, including local variables, is managed statically.

For architectures where fewer registers are available, only the first few slots, which are the most frequently used, are placed in the register based stack frame. The other elements are placed into the heap allocated stack frame and have a higher cost for read and write operations.

The static frame mechanism, although very efficient, also implies some strict limitations on the number of local variables and the size of the stack that can be manipulated. As Leros supports 256 words of memory accessible as registers, the maximum stack frame per method is 256 words. We consider this large enough for microcontroller applications.

Further optimization is applied by the construction of virtual bytecodes from patterns detected in the bytecode stream. For example

```
localVariableA = LocalVariableA + Constant
```

would produce the pattern

```
push(Constant), push (LocalVariableA),
add, pop and store LocalVariableA.
```

The compiler tracks the local variables statically in the frame and would detect this pattern to produce

```
RegisterLocA = RegisterLocA + Constant
```

which can be 1 or 2 instructions, depending on the constant value and the instruction set. Pattern matching is applied to a database to replace sequences of bytecode with optimized 'virtual' bytecodes. As new patterns are added, pre-post conditions can be applied during testing to verify the equivalence of the virtual bytecode operator. This procedure is similar to peephole optimization.

4.3 Garbage Collector

Garbage collection is always a major challenge in embedded systems. Many do not support garbage collection at all. Muvium employs a reference counting mechanism with direct compiler support. The compiler attempts to minimize the number of times the reference counters are incremented and decremented. While it is well understood that reference counting does not solve the problem in general, for example self references and cycles are not handled, the benefit is that the major cost of computing the references during a stop-the-world hierarchical mark-sweep is avoided. When the GC is invoked the destructors of objects with zero reference count are called, this can be recursive or early exit if sufficient memory has been released. A compactor is then called to compact the heap.

The GC is interruptible and can be run in the background although presently it is used when a `new` fails to allocate sufficient memory or when `System.gc()` is called directly. Due to the efficiency and the small amount of memory typical of a Muvium system the GC invocation is usually not noticeable at all. Muvium libraries specifically avoid the use of self referencing code to take advantage of the performance of the integrated reference counter. To handle the case of self references a special-case mark-sweep type algorithm is used. In the case when all threads are in the root entry frame and the `System.gc()` is manually invoked, a *special case* mark sweep is applied to rebuild the reference counts. This works by first resetting all reference counters to zero and then calling the destructors of each of the thread objects which increment rather than decrements reference counts as the destructor normally does. This is then applied recursively to implement a mark-sweep algorithm rebuilding reference counters. Then the normal `gc()` is called. This is avoided wherever possible as it has the familiar performance problems of a stop-the world mark-sweep.

4.4 Java Restrictions

For such a small system, as presented here, there are several restrictions on Java. First, with those tight memory constraints, a full JDK is practically unusable. Muvium contains a small subset of the JDK, similar to the CLDC version of J2ME. To support deeply embedded controller applications Muvium offers an additional library to interface I/O components.

Second, integer values are currently represented as 16-bit values. This is a departure from the standard JVM, which is similar to the original Java Card JVM. We plan to implement a compiler switch to decide on the representation of integers as fast 16-bit types or as JVM compliant 32-bit types.

These restrictions might be seen as a large departure from standard Java, and one could argue that C would be a better solution. However, Java, even when restricted, is a safer language than C. Furthermore, the Leros/Muvium combination is intended to build virtual devices that can be attached to a main processor. That main processor can be a Java processor, such as JOP [5]. In that case Java can be used for the high-level and low-level programming of an embedded system.

Table 1: Comparison of Leros with PicoBlaze and SpartanMC

Processor	Logic (LC)	Memory (blocks)	Fmax (MHz)
Leros	188	1	115
PicoBlaze	177	1	117
SpartanMC	1271	3	50

5. EVALUATION

We evaluated Muvium and Leros with small programs that are typical for tiny microcontrolles and report the resource consumption. We also compare the performance of Muvium for Leros and for a PIC microcontroller.

5.1 Hardware Resources

We have implemented Leros on several different FPGA boards with Altera and Xilinx FPGAs. The VHDL code of Leros is highly portable. The only changes needed for a port are the pin definitions for the board and a device specific PLL component.

Table 1 shows the resource utilization and maximum clock frequency of Leros and two other small microcontrollers for a Spartan-3E device. Leros is comparable in resource consumption and maximum clock frequency with PicoBlaze. However, Leros implements a 16-bit data path and can execute an instruction in a single cycle, whereas PicoBlaze is an 8-bit processor and needs 2 clock cycles for each instruction. In contrast to Leros, SpartanMC implements a register machine and needs therefore more resources. The lower clock frequency of SpartanMC is due to the use of two phase-shifted clocks for sub-dividing of two pipeline stages into two phases.

For the evaluation with Java we have added a serial port (UART) to Leros, which itself consumes 96 LCs. Leros with a simple "Hello World" program in Java, which prints "Leros" and then echoes characters received via the serial line, consumes 170 bytes of instruction memory. The resulting hardware is 435 logic cells and one memory block for the RAM. Out of the 435 LCs, 104 are consumed by the instruction ROM and 96 by the UART.

5.2 Example Application

The example application is based on the vision of executing the Android Midget graphical framework on Leros. The Android Midget graphical framework is a clean-room, light-weight Java implementation of the widget, event based Android programming model, including buttons, menus, text boxes, spinners, images, checkboxes, and custom views. For the evaluation we have im-

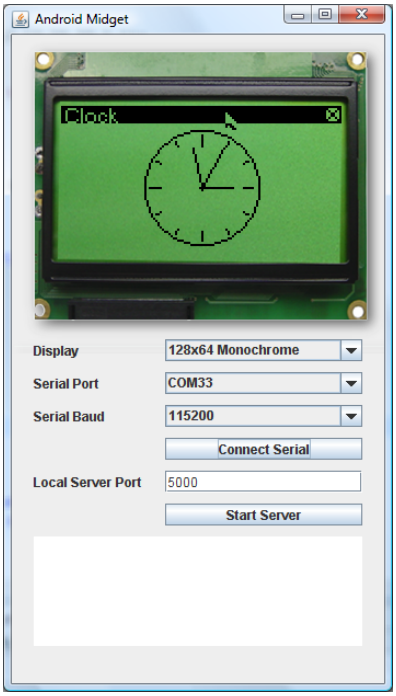

Figure 3: The Midget simulation on the PC

plemented line drawing for using Leros as an embedded graphics engine. As Leros is so tiny we intend to use a many-core version for the inherently parallel application.

For the evaluation of our system we use an FPGA board with a single Leros processor, connected via a serial line to a PC. The PC simulates the Midget and the graphics display. The drawing coordinates are sent via the UART from the PC to the FPGA board and the resulting pixel coordinates are sent back to the PC. The display simulation is shown in Figure 3.

The `drawLine()` method is used to illustrate the concept. This routine is an essential part of the Android Midget framework and 16-bit integers are sufficient for this application. Since these conditions are met by the Muvium Leros compiler, we are able to execute this Java code in a Leros Java thread. This application needs 385 instructions (770 bytes) on Leros. For reference, the source of the line drawing routine is available in the repository of Leros.

5.3 Performance

For performance evaluation we have hardcoded the drawing coordinates in the function and executed it on Leros and on a cycle-accurate simulation of a Microchip PIC.[5] This PIC microcontroller is an 8-bit processor and therefore will be less efficient than Leros.

With an interpreting JVM we measured that the `drawLine()` method executes 728 bytecodes. Table 2 shows the size of the instruction memory, the number of cycles to execute the line drawing method, and the resulting clock cycles per bytecode. However, it has to be noted that this program contains mostly simple bytecodes and no method invocations. The 16-bit architecture of Leros leads to less code and shorter execution time than the 8-bit PIC microcontroller. The instruction memory number is smaller in this table as given before, as the input data is hard coded instead of being read from the UART.

[5]e.g., a PIC18F67K22, see `http://ww1.microchip.com/downloads/en/DeviceDoc/39960d.pdf`

Table 2: Comparison of Muvium for Leros and PIC

Processor	Instructions (byte)	Execution time (clocks)	Cycles/bytecode (clocks)
Leros	574	1577	2.2
Microchip PIC	1326	5196	7.1

6. CONCLUSION

In this paper we presented the probably smallest embedded Java system currently available. It consists of a tiny soft-core implemented in an FPGA and an ahead-of-time compiler that is optimized for generating a small footprint JVM and Java application.

With this system we open the path to use Java in very small microcontroller applications. With Leros a few lines of Java code can implement intelligent peripherals in software, for example a fully software implemented serial interface. As the Leros processor is available in open-source and the Leros port of Muvium is freely available, we hope to see many micro-Java projects building up the Java Dust.

As future work we will explore a many-core version of Leros and mapping of channel-based communication between the cores into embedded Java. Furthermore, we will evaluate how additional instructions in Leros might help the efficiency of the JVM without increasing the hardware resources too much.

7. REFERENCES

[1] F. Brandner, T. Thorn, and M. Schoeberl. Embedded JIT compilation with CACAO on YARI. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, pages 63–70, March 2009. IEEE Computer Society.

[2] J. P. Caska. Architecture for static frames in a stack machine for an embedded device. US Patent 2005/0076172 A1, 2004.

[3] G. Hempel and C. Hochberger. A resource optimized processor core for FPGA based SoCs. In H. Kubatova, editor, *Proceedings of the 10th Euromicro Conference on Digital System Design (DSD 2007)*, pages 51–58. IEEE, 2007.

[4] A. Krall. Efficient JavaVM just-in-time compilation. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT '98)*, pages 205–212. IEEE Computer Society Press, 1998.

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

[6] M. Schoeberl. *JOP Reference Handbook: Building Embedded Systems with a Java Processor*. Number ISBN 978-1438239699. CreateSpace, August 2009.

[7] M. Schoeberl. Leros: A tiny microcontroller for FPGAs. In *Proceedings of the 21st International Conference on Field Programmable Logic and Applications (FPL 2011)*, September 2011. IEEE Computer Society.

[8] 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.

[9] I. Thomm, M. Stilkerich, C. Wawersich, and W. Schröder-Preikschat. KESO: an open-source multi-JVM for deeply embedded systems. In *JTRES'10*, pages 109–119. ACM, 2010.

[10] Xilinx. PicoBlaze 8-bit embedded microcontroller user guide, 2010. Xilinx Inc.