JOP Reference Handbook

JOP Reference Handbook

Building Embedded Systems with a Java Processor

Martin Schoeberl

Version: September 18, 2008

Martin Schoeberl
Strausseng. 2-10/2/55
A-1050 Vienna, Austria

Email: martin@jopdesign.com
Visit the accompanying web site on http://www.jopdesign.com/ and
the JOP Wiki at http://www.jopwiki.com/

Published 2008 by CreateSpace,
http://www.createspace.com/

Library of Congress Cataloging-in-Publication Data

Schoeberl, Martin

JOP Reference Handbook: Building Embedded Systems
with a Java Processor / Martin Schoeberl
Includes bibliographical references and index.
ISBN 978-1438239699

Manufactured in the United States of America.
Typeset in 11pt Times by Martin Schoeberl

Foreword

This book is about JOP, the Java Optimized Processor. JOP began as a research project for a PhD thesis. JOP has been used in several industrial applications and, due to the fact that it is an open-source project, has a growing user base. This book is written for all of you who build this lively community.

For a long time the thesis, some research papers, and the web site have been the only available documentation for JOP. A thesis is quite different from a reference manual. Its focus is on research results and implementation details are usually omitted. This book complements the thesis and provides insight into the implementation of JOP and the accompanying Java virtual machine (JVM). It also gives you an idea how to build an embedded real-time system based on JOP.

Acknowledgements

Many users of JOP contributed to the design of JOP and to the tool chain. I also want to thank for the discussions with the students at the Vienna University of Technology during three years of the course "The JVM in Hardware" and one semester in Copenhagen at an embedded systems course in Java. Furthermore, the questions and discussions in the Java processor mailing list provided valuable input for the documentation now available in form of this book.

Ed Anuff wrote testmon.asm to perform a memory interface test and BlockGen.java to convert Altera .mif files to Xilinx memory blocks. BlockGen.java was the key tool to port JOP to Xilinx FPGAs in general and the Spartan-3 specifically. Flavius Gruian wrote the initial version of JOPizer to generate the .jop file from the application classes. JOPizer is based on the open source BCEL and is a substitute to the formerly used JavaCodeCompact from Sun.

Contents

Foreword v

Acknowledgements vii

1 Introduction 1
- 1.1 A Quick Tour on JOP . . . 1
  - 1.1.1 Building JOP and Running "Hello World" . . . 1
  - 1.1.2 The Design Structure . . . 3
- 1.2 A Short History . . . 3
- 1.3 JOP Features . . . 4
- 1.4 Is JOP the Solution for Your Problem? . . . 6
- 1.5 Outline of the Book . . . 6

2 The Design Flow 9
- 2.1 Introduction . . . 9
  - 2.1.1 Tools . . . 9
  - 2.1.2 Getting Started . . . 10
  - 2.1.3 Xilinx Spartan-3 Starter Kit . . . 11
- 2.2 Booting JOP — How Your Application Starts . . . 12
  - 2.2.1 FPGA Configuration . . . 12
  - 2.2.2 Java Download . . . 13
  - 2.2.3 Combinations . . . 13
- 2.3 The Design Flow . . . 14
  - 2.3.1 Tools . . . 14
  - 2.3.2 Targets . . . 15
- 2.4 Eclipse . . . 17
- 2.5 Simulation . . . 18
  - 2.5.1 JopSim Simulation . . . 18
  - 2.5.2 VHDL Simulation . . . 18
- 2.6 Files Types You Might Encounter . . . 20

2.7 Information on the Web . . . 22
2.8 Porting JOP . . . 22
2.8.1 Test Utilities . . . 22
2.9 Extending JOP . . . 23
2.9.1 Native Methods . . . 23
2.9.2 A new Peripheral Device . . . 24
2.9.3 A Customized Instruction . . . 25
2.9.4 Dependencies and Configurations . . . 25
2.10 Directory Structure . . . 26
2.10.1 The Java Sources for JOP . . . 27
2.11 The JOP Hello World Exercise . . . 28
2.11.1 Manual build . . . 28
2.11.2 Using make . . . 29
2.11.3 Change the Java Program . . . 29
2.11.4 Change the Microcode . . . 29
2.11.5 Use a Different Target Board . . . 30
2.11.6 Compile a Different Java Application . . . 30
2.11.7 Simulation . . . 31
2.11.8 WCET Analysis . . . 31

3 Java and the Java Virtual Machine 33
3.1 Java . . . 33
3.1.1 History . . . 35
3.1.2 The Java Programming Language . . . 36
3.2 The Java Virtual Machine . . . 37
3.2.1 Memory Areas . . . 38
3.2.2 JVM Instruction Set . . . 38
3.2.3 Methods . . . 40
3.2.4 Implementation of the JVM . . . 41
3.3 Embedded Java . . . 42
3.4 Summary . . . 44

4 JOP Hardware Architecture 45
4.1 Overview of JOP . . . 45
4.2 Microcode . . . 47
4.2.1 Translation of Bytecodes to Microcode . . . 47
4.2.2 Compact Microcode . . . 49

4.2.3 Instruction Set . . . 50
4.2.4 Bytecode Example . . . 51
4.2.5 Flexible Implementation of Bytecodes . . . 52
4.2.6 Summary . . . 52
4.3 The Processor Pipeline . . . 53
4.3.1 Java Bytecode Fetch . . . 54
4.3.2 JOP Instruction Fetch . . . 55
4.3.3 Decode and Address Generation . . . 56
4.3.4 Execute . . . 56
4.3.5 Interrupt Logic . . . 58
4.3.6 Summary . . . 59
4.4 An Efficient Stack Machine . . . 60
4.4.1 Java Computing Model . . . 60
4.4.2 Access Patterns on the Java Stack . . . 63
4.4.3 Common Realizations of a Stack Cache . . . 64
4.4.4 A Two-Level Stack Cache . . . 67
4.4.5 Resource Usage Compared . . . 73
4.4.6 Summary . . . 74
4.5 HW/SW Codesign . . . 76
4.6 Real-Time Predictability . . . 80
4.6.1 Interrupts . . . 80
4.6.2 Task Switch . . . 81
4.6.3 Architectural Design Decisions . . . 83
4.6.4 Summary . . . 85
4.7 A Time-Predictable Instruction Cache . . . 86
4.7.1 Cache Performance . . . 86
4.7.2 Proposed Cache Solution . . . 90
4.7.3 WCET Analysis . . . 94
4.7.4 Caches Compared . . . 96
4.7.5 Summary . . . 101

5 JOP Runtime System 103
5.1 A Real-Time Profile for Embedded Java . . . 103
5.1.1 Application Structure . . . 104
5.1.2 Threads . . . 104
5.1.3 Scheduling . . . 105
5.1.4 Memory . . . 107

5.1.5 Restrictions on Java . . . 107
5.1.6 Implementation Results . . . 108
5.2 User-Defined Scheduler . . . 108
5.2.1 Schedule Events . . . 111
5.2.2 Data Structures . . . 111
5.2.3 Services for the Scheduler . . . 112
5.2.4 Class Scheduler . . . 112
5.2.5 Class Task . . . 115
5.2.6 A Simple Example Scheduler . . . 116
5.2.7 Interaction of Task, Scheduler and the JVM . . . 118
5.2.8 Predictability . . . 119
5.2.9 Related Work . . . 123
5.2.10 Summary . . . 123
5.3 A Profile for Safety Critical Java . . . 124
5.4 Safety Critical Java . . . 124
5.4.1 SCJ Level 1 . . . 124
5.5 JVM Architecture . . . 128
5.5.1 Runtime Data Structures . . . 128
5.5.2 Booting the JVM . . . 135

6 Real-Time Garbage Collection 137
6.1 Introduction . . . 137
6.1.1 Incremental Collection . . . 139
6.1.2 Conservatism . . . 139
6.1.3 Safety Critical Java . . . 139
6.2 Scheduling of the Collector Thread . . . 140
6.2.1 An Example . . . 141
6.2.2 Minimum Heap Size . . . 143
6.2.3 Garbage Collection Period . . . 148
6.3 SCJ Simplifications . . . 154
6.3.1 Simple Root Scanning . . . 156
6.3.2 Static Memory . . . 157
6.4 Implementation . . . 159
6.4.1 Heap Layout . . . 159
6.4.2 The Collector . . . 160
6.4.3 The Mutator . . . 164
6.5 Evaluation . . . 168

6.5.1 Scheduling Experiments . . . 168
6.5.2 Measuring Release Jitter . . . 174
6.5.3 Measurements . . . 175
6.5.4 Discussion . . . 177
6.6 Analysis . . . 179
6.6.1 Worst Case Memory Consumption . . . 179
6.6.2 WCET of the Collector . . . 179
6.7 Summary . . . 180
6.8 Further Reading . . . 180
6.8.1 Related Work . . . 181

7 Worst-Case Execution Time 183
7.1 Microcode Path Analysis . . . 184
7.2 Microcode Low-level Analysis . . . 185
7.3 Bytecode Independency . . . 185
7.4 WCET of Bytecodes . . . 186
7.5 WCET Analysis of the Java Application . . . 188
7.5.1 An Example . . . 188
7.5.2 WCET Analyzer . . . 192
7.6 Discussion . . . 194

8 The SimpCon Interconnect 195
8.1 Introduction . . . 195
8.1.1 Features . . . 196
8.1.2 Basic Read Transaction . . . 196
8.1.3 Basic Write Transaction . . . 197
8.2 SimpCon Signals . . . 197
8.2.1 Master Signal Details . . . 197
8.2.2 Slave Signal Details . . . 199
8.3 Slave Acknowledge . . . 200
8.4 Pipelining . . . 202
8.4.1 Interconnect . . . 203
8.5 Examples . . . 203
8.5.1 IO Port . . . 203
8.5.2 SRAM interface . . . 204
8.5.3 Master Multiplexing . . . 206
8.6 Available VHDL Files . . . 206

8.6.1 Components . . . 206
8.6.2 Bridges . . . 207
8.7 Why a New Interconnection Standard? . . . 207
8.7.1 Common SoC Interconnections . . . 207
8.7.2 What's Wrong with the Classic Standards? . . . 209
8.7.3 Evaluation . . . 211
8.8 Summary . . . 212

9 Chip Multiprocessing 215
9.1 Booting a CMP System . . . 215
9.2 CMP Scheduling . . . 216
9.2.1 One Thread per Core . . . 217
9.2.2 Scheduling on the CMP System . . . 218

10 Results 221
10.1 Hardware Platforms . . . 222
10.2 Resource Usage . . . 223
10.3 Performance . . . 226
10.3.1 General Performance . . . 226
10.3.2 Discussion . . . 229
10.3.3 Execution Time Jitter . . . 233
10.4 Applications . . . 235
10.4.1 Motor Control . . . 235
10.4.2 Further Projects . . . 237
10.5 Summary . . . 239

11 Related Work 241
11.1 Hardware Translation and Coprocessors . . . 241
11.1.1 Hard-Int . . . 243
11.1.2 DELFT-JAVA Engine . . . 243
11.1.3 JIFFY . . . 243
11.1.4 Jazelle . . . 244
11.1.5 JSTAR, JA108 . . . 245
11.1.6 A Co-Designed Virtual Machine . . . 245
11.2 Java Processors . . . 246
11.2.1 picoJava . . . 246
11.2.2 aJile JEMCore . . . 249

11.2.3 Cjip . . . 250
11.2.4 Ignite, PSC1000 . . . 251
11.2.5 Moon . . . 251
11.2.6 Lightfoot . . . 251
11.2.7 LavaCORE . . . 252
11.2.8 Komodo . . . 252
11.2.9 FemtoJava . . . 253
11.2.10 jHISC . . . 253
11.3 Additional Comments . . . 253
11.4 Summary . . . 255

12 Summary **257**
12.1 A Real-Time Java Processor . . . 257
12.2 A Resource-Constrained Processor . . . 259
12.3 Future Research Directions . . . 260

A Publications **263**

B Acronyms **267**

C JOP Instruction Set **269**

D Bytecode Execution Time **297**

E Cyclone FPGA Board **309**

Bibliography **315**

Index **327**

F TODO **329**
F.1 Working Hours . . . 329
F.2 TODO Handbook . . . 329
F.2.1 Minor Change Ideas . . . 330
F.2.2 Editorial Stuff . . . 330
F.3 Additional Sections . . . 331
F.3.1 Notes . . . 332

G DONE **333**

H TODO for the 2nd Edition **335**

I Other Stuff **337**
I.1 Publishing the Book 337
I.2 A Possible Structure of the Book 338
I.3 JOPizer Description 340
I.4 YAFFS Notes 340
I.5 Debugger Description 340
I.6 TODO JOP 345

1 Introduction

This handbook introduces a Java processor for embedded real-time systems, in particular the design of a small processor for resource-constrained devices with time-predictable execution of Java programs. This Java processor is called JOP – which stands for Java Optimized Processor –, based on the assumption that a full native implementation of all Java bytecode instructions is not a useful approach.

1.1 A Quick Tour on JOP

In the following section we will give a quick overview on JOP and a short description how to get JOP running within an FPGA. A detailed description of the build process can be found in Chapter 2.

JOP is a soft-core written in VHDL plus tools in Java, a simplified Java library (JDK), and application examples. JOP is delivered in source only and the sources are hosted at Opencores.[1]

1.1.1 Building JOP and Running "Hello World"

To build JOP you first have to download the source tree from Opencores. A *Makefile* (or an Ant file) contains all necessary steps to build the tools, the processor, and the application. Configuration of the FPGA and downloading the Java application is also part of the Makefile.

In this description we assume the FPGA board Cycore (see Appendix E). This board is the default target for the Makefile. The board has to be connected to the power supply and to the PC via a ByteBlaster download cable and a serial cable.

The FPGA is configured via the ByteBlaster cable. The Java application is downloaded after the FPGA configuration via the serial cable. Besides the download the serial cable is also used as a communication link between JOP and the PC. `System.out` and `System.in` represent this serial link on JOP.

[1] `http://www.opencores.org/projects.cgi/web/jop`

In order to build the whole system you need a Java compiler[2] and an FPGA compiler. In our case we use the free web edition of Quartus from Altera.[3] As we use make and the preprocessor from the GNU compiler collection, Cygwin[4] should be installed under Windows.

When all tools are setup correctly[5] a simple make should build the tools, the processor, compile the "Hello World" example, configure the FPGA and download the application. The whole build process will take a few minutes. After typing

```
make
```

you see a lot of messages from the various tools. However, the last lines should be actual messages received from JOP. It should look similar to the following:

```
JOP start V 20080821
60 MHz, 1024 KB RAM, 1 CPUs
Hello World from JOP!

JVM exit!
```

Note that JOP prints some internal information, such as version and memory size, at startup. After that, the message "Hello World from JOP!" can be seen. Our first program runs on JOP!

As a next step, locate the Hello World example in the source tree[6] and change the output message. The tools and the processor have been built already. So we do not need to compile everything from scratch. Use the following make target to just compile the Java application and download the processor and the application:

```
make japp
```

The compile process should now be faster and the output similar to before.

The Hello World application is the default target in the Makefile. See Chapter 2 for a description how this target can be changed. In case you use a different FPGA board you can find information on how to change the build process also in Chapter 2.

[2] Download the Java SE Development Kit (JDK) from http://java.sun.com/javase/downloads/index.jsp.

[3] http://www.altera.com/

[4] http://www.cygwin.com/

[5] Check at the command prompt that javac is in the path.

[6] .../jop/java/target/src/test/test/HelloWorld.java

1.1.2 The Design Structure

Browsing the source tree of JOP can give the impression that the design is complex. However, the basic structure is not that complex. The design consists of three entities:

1. The processor JOP
2. Supporting tools
3. The Java library and applications

The different entities are also reflected during the configuration and download process. The download is a two step process:

1. Configuration of the FPGA: JOP is downloaded via a FPGA download cable (e.g., ByteBlaster on the PCs parallel port). After FPGA configuration the processor automatically starts and listens to the second channel (the serial line) for the software download.
2. Java application download: the compiled and linked application is downloaded usually via a serial line. JOP stores the application in the main memory and starts execution at `main()` after the download.

Further details of the source structure can be found in Section 2.10.

1.2 A Short History

The first version of JOP was created in 2000 based on the adaptation of earlier processor designs created between 1995 and 2000. The first version was written in Altera's proprietary AHDL language. The first *program* (3 bytecode instructions) ran on JOP on October 2, 2000. The first approach was a general purpose accumulator/register machine with 16-bit instructions, 32-bit registers, and a pipeline length of 3. It used the on-chip block memory to implement (somehow unusual) 1024 registers.

The JVM was implemented in the assembler of that machine. That concept was similar to the microcode in the current JOP version. The decoding of the bytecode was performed by a long jump table. In the best case (assuming a local, single cycle memory) a simple bytecode (e.g. `iadd`) took 12 cycles for fetch and decode and additional 11 cycles for execution.

A redesign followed in April 2001, now coded in VHDL. The second version of JOP introduced features to speed up the implementation of the JVM with specific instructions for

the stack access and a dedicated stack pointer. The register file was reduced to 16 entries and the instruction width reduced to 8 bits. The pipeline contained 5 stages and special support for decoding bytecode instructions was added – a first version of the dynamic bytecode to microcode address translation as it is used in the current version of JOP. The enhancements within JOP2 resulted in the reduction of the execution time for a simple bytecode to 3 cycles. A great enhancement compared to the 23 cycles in JOP1.

The next redesign (JOP3) followed in June 2001. The challenge was to execute simple bytecodes fully pipelined in a single cycle. The microcode instruction set was changed to implement a stack machine and the execution stage combined with the on-chip stack cache. Microcode instructions where coded in 16 bit and the pipeline was reduced to four stages. JOP3 is the basis of JOP as it is described in this handbook. The later changes have not been so radical to call them a redesign.

The first real-world application of JOP was in the project *Kippfahrleitung* (see Section 10.4.1). At the start of the project (October 2001) JOP could only execute a single static method stored in the on-chip memory. The project greatly pushed the development of JOP. After successful deployment of the JOP-based control system in the field, several projects followed (TeleAlarm, Lift, the railway control system). The source of the commercial applications is part of the JOP distribution. Some of these applications are now used as a test bench for embedded Java performance and to benchmark WCET analysis tools.

More details and the source code of JOP1[7], JOP2[8] and the first JOP3[9] version are available on the web site.

1.3 JOP Features

This book presents a hardware implementation of the Java virtual machine (JVM), targeting small embedded systems with real-time constraints. JOP is designed from the ground up with time-predictable execution of Java bytecode as a major design goal. All functional units, and especially the interactions between them, are carefully designed to avoid any time dependency between bytecodes.

JOP is a stack computer with its own instruction set, called microcode in this book. Java bytecodes are translated into microcode instructions or sequences of microcode. The difference between the JVM and JOP is best described as the following:

[7] http://www.jopdesign.com/jop1.jsp
[8] http://www.jopdesign.com/jop2.jsp
[9] http://www.jopdesign.com/jop3.jsp

The JVM is a CISC stack architecture, whereas JOP is a RISC stack architecture.

The architectural features and highlights of JOP are:

- Dynamic translation of the CISC Java bytecodes to a RISC, stack based instruction set (the microcode) that can be executed in a 3 stage pipeline.
- The translation takes exactly one cycle per bytecode and is therefore pipelined. Compared to other forms of dynamic code translation the proposed translation does not add any variable latency to the execution time and is therefore time predictable.
- Interrupts are inserted in the translation stage as special bytecodes and are transparent to the microcode pipeline.
- The short pipeline (4 stages) results in short conditional branch delays and a hard to analyze branch prediction logic or branch target buffer can be avoided.
- Simple execution stage with the two topmost stack elements as discrete registers. No write back stage or forwarding logic is needed.
- Constant execution time (one cycle) for all microcode instructions. The microcode pipeline never stalls. Loads and stores of object fields are handled explicitly.
- No time dependencies between bytecodes result in a simple processor model for the low-level WCET analysis.
- Time predictable instruction cache that caches whole methods. Only invoke and return instruction can result in a cache miss. All other instructions are guaranteed cache hits.
- Time predictable data cache for local variables and the operand stack. Access to local variables is a guaranteed hit and no pipeline stall can happen. Stack cache fill and spill is under microcode control and analyzable.
- No prefetch buffers or store buffers that can introduce unbound time dependencies of instructions. Even simple processors can contain an instruction prefetch buffer that prohibits exact WCET values. The design of the method cache and the translation unit avoids the variable latency of a prefetch buffer.
- Good average case performance compared to other non real-time Java processors.

- Avoidance of hard to analyze architectural features results in a very small design. Therefore an available real estate can be used for a chip multi-processor solution.
- JOP is the smallest hardware implementation of the JVM available to date. This fact enables usage of low-cost FPGAs in embedded systems. The resource usage of JOP can be configured to trade size against performance for different application domains.
- JOP is actually in use in several real-world applications showing that a Java based embedded system implemented in an FPGA is a viable option.

JOP is implemented as a soft-core in a field programmable gate array (FPGA) giving a lot of flexibility for the overall hardware design. The processor can easily be extended by peripheral components inside the same chip. Therefore, it is possible to customize the solution exactly to the needs of the system.

1.4 Is JOP the Solution for Your Problem?

I had a lot of fun, and still have, developing and using JOP. However, should you use JOP? JOP is a processor design intended as a time predictable solution for hard real-time systems. If your application or research focus is on those systems and you prefer Java as programming language, JOP is the right choice. If you are interested in larger, dynamic systems, JOP is the wrong choice. If average performance is important for you and you do not care about worst-case performance other solutions will probably do a better job.

1.5 Outline of the Book

Chapter 2 gives a detailed introduction into the design flow of JOP. It explains how the individual parts are compiled and which files have to be changed when you want to extend JOP or adapt it to a new hardware platform. The chapter is concluded by an exercise to explore the different steps in the design flow.

Chapter 3 provides background information on the Java programming language, the execution environment, and the Java virtual machine, for Java applications. If you are already familiar with Java and the JVM, feel free to skip this chapter.

Chapter 4 is the main chapter in which the architecture of JOP is described. The motivation behind different design decisions is given. A Java processor alone is not a complete JVM. Chapter 5 describes the runtime environment on top of JOP, including the definition of a real-time profile for Java and the description of the scheduler in Java.

Garbage collection (GC) is an important part of the Java technology. Even in real-time systems new real-time garbage collectors emerge. In Chapter 6 the formulas to calculate the correct scheduling of the GC thread are given and the implementation of the real-time GC for JOP is explained.

In Chapter 7 WCET analysis of the individual Java bytecodes is performed. It is shown how these bytecode instruction timings form the basis of WCET analysis of Java applications.

JOP uses a simple and efficient system-on-chip interconnection called SimpCon to connect the memory controller and peripheral devices to the processor pipeline. The definition of SimpCon and the rationale behind the SimpCon specification is given in Chapter 8.

In Chapter 10, JOP is evaluated with respect to size and performance. This is followed by a description of some commercial real-world applications of JOP.

Other hardware implementations of the JVM are presented in Chapter 11. Different hardware solutions from both academia and industry for accelerating Java in embedded systems are analyzed.

Finally, in Chapter 12, the work is summarized and the major contributions are presented. This chapter concludes with directions for future research using JOP and real-time Java.

A more theoretical treatment of the design of JOP can be found in the PhD thesis [107], which is also available as book [113].

2 The Design Flow

This section describes the design flow for JOP — how to build the Java processor and a Java application from scratch (the VHDL and Java sources) and download the processor to an FPGA and the Java application to the processor.

2.1 Introduction

JOP [107], the Java optimized processor, is an open-source development platform available for different targets (Altera and Xilinx FPGAs and various types of FPGA boards). To support several targets, the design-flow is a little bit complicated. There is a Makefile available and when everything is set up correctly, a simple

```
make
```

should build everything from the sources and download a *Hello World* example. However, to customize the Makefile for a different target it is necessary to understand the complete design flow. It should be noted that an Ant[1] based build process is also available.

2.1.1 Tools

All needed tools are freely available.

- Java SE Development Kit (JDK) Java compiler and runtime
- Cygwin GNU tools for Windows. Packages cvs, gcc and make are needed
- Quarts II Web Edition VHDL synthesis, place and route for Altera FPGAs

The PATH variable should contain entries to the executables of all packages (java and javac, Cygwin bin, and Quartus executables). Check the PATH at the command prompt with:

[1] http://ant.apache.org/

```
javac
gcc
make
cvs
quartus_map
```

All the executables should be found and usually report their usage.

2.1.2 Getting Started

This section shows a quick step-by-step build of JOP for the Cyclone target in the minimal configuration. All directory paths are given relative to the JOP root directory jop. The build process is explained in more detail in one of the following sections.

Download the Source

Create a working directory and download JOP from the www.opencores.org CVS server:

```
cvs -d :pserver:anonymous@cvs.opencores.org:/cvsroot/anonymous \
   -z9 co -P jop
```

All sources are downloaded to a directory jop. For the following command change to this directory. Create the needed directories with:

```
make directories
```

Tools

The tools contain Jopa, the microcode assembler, JopSim, a Java based simulation of JOP, and JOPizer, the application builder. The tools are built with following make command:

```
make tools
```

Assemble the Microcode JVM, Compile the Processor

The JVM configured to download the Java application from the serial interface is built with:

```
make jopser
```

This command also invokes Quartus to build the processer. If you want to build it within Quartus follow the following instructions:

Start Quartus II and open the project jop.qpf from directory quartus/cycmin in Quartus with *File – Open Project...*. Start the compiler and fitter with *Processing – Start Compilation*. After successful compilation the FPGA is configured with *Tools – Programmer* and *Start*.

Compiling and Downloading the Java Application

A simple *Hello World* application is the default application in the Makefile. It is built and downloaded to JOP with:

```
make japp
```

The "Hello World" message should be printed in the command window.

For a different application change the Makefile targets or override the make variables at the command line. The following example builds and runs some benchmarks on JOP:

```
make japp -e P1=bench P2=jbe P3=DoAll
```

The three variables P1, P2, and P3 are a shortcut to set the directory, the package name, and the main class of the application.

USB based Boards

Several Altera based boards use an FTDI FT2232 USB chip for the FPGA and Java program download. To change the download flow for those boards change the value of the following variable in the Makefile to true:

```
USB=true
```

The Java download channel is mapped to a virtual serial port on the PC. Check the port number in the system properties and set the variable COM_PORT accordingly.

2.1.3 Xilinx Spartan-3 Starter Kit

The Xilinx tool chain is still not well supported by the Makefile or the Ant design flow. Here is a short list on how to build JOP for a Xilinx board:

```
make tools
cd asm
jopser
cd ..
```

Now start the Xilinx IDE wirh the project file jop.npl. It will be converted to a new (binary) jop.ise project. The .npl project file is used as it is simple to edit (ASCII).

- Generate JOP by double clicking 'Generate PROM, ACE, or JTAG File'
- Configure the FPGA according to the board type

The above is a one step build for the processor. The Java application is built and downloaded by:

```
make java_app
make download
```

Now your first Java program runs on JOP/Spartan-3!

2.2 Booting JOP — How Your Application Starts

Basically this is a two step process: (a) configuration of the FPGA and (b) downloading the Java application. There are different possibilities to perform these steps.

2.2.1 FPGA Configuration

FPGAs are usually SRAM based and *lose* their configuration after power down. Therefore the configuration has to be loaded on power up. For development the FPGA can be configured via a download cable (with JTAG commands). This can be done within the IDEs from Altera and Xilinx or with command line tools such as quartus_pgm or jbi32.

For the device to boot automatically, the configuration has to be stored in non volatile memory such as Flash. Serial Flash is directly supported by an FPGA to boot on power up. Another method is to use a standard parallel Flash to store the configuration and additional data (e.g. the Java application). A small PLD reads the configuration data from the Flash and shifts it into the FPGA. This method is used on the Cyclone and ACEX boards.

2.2.2 Java Download

When the FPGA is configured the Java application has to be downloaded into the main memory. This download is performed in microcode as part of the JVM startup sequence. The application is a .jop file generated by JOPizer. At the moment there are three options:

Serial line JOP listens to the serial line and the data is written into the main memory. A simple echo protocol performs the flow control. The baud rate is usually 115 kBaud.

USB Similar to the serial line version, JOP listens to the parallel interface of the FTDI FT2232 USB chip. The FT2232 performs the flow control at the USB level and the echo protocol is omitted.

Flash For stand alone applications the Java program is copied from the Flash (relative Flash address 0, mapped Flash address is 0x80000[2]) to the main memory (usually a 32-bit SRAM).

The mode of downloading is defined in the JVM (jvm.asm). To select a new mode, the JVM has to be assembled and the complete processor has to be rebuilt – a full make run. The generation is performed by the C preprocessor (gcc) on jvm.asm. The serial version is generated by default; the USB or Flash version are generated by defining the preprocessor variables USB or FLASH.

VHDL Simulation To speed up the VHDL simulation in ModelSim there is a forth method where the Java application is loaded by the test bench instead of JOP. This version is generated by defining SIMULATION. The actual Java application is written by jop2dat into a plain text file (mem_main.dat) and read by the simulation test bench into the simulated main memory.

There are four small batch-files in directory asm that perform the JVM generation: jopser, jopusb, jopflash, and jopsim.

2.2.3 Combinations

Theoretically all variants to configure the FPGA can be combined with all variations to download the Java application. However, only two combinations are useful:

[2] All addresses in JOP are counted in 32-bit quantities. However, the Flash is connected only to the lower 8 bits of the data bus. Therefore a store of one word in the main memory needs four loads from the Flash.

1. For VHDL or Java development configure the FPGA via the download cable and download the Java application via the serial line or USB.

2. For a stand-alone application load the configuration and the Java program from the Flash.

2.3 The Design Flow

This section describes the design flow to build JOP in greater detail.

2.3.1 Tools

There are a few tools necessary to build and download JOP to the FPGA boards. Most of them are written in Java. Only the tools that access the serial line are written in C.[3]

Downloading

These little programs are already compiled and the binaries are checked in into the repository. The sources can be found in directory `c_src`.

`down.exe` The workhorse to download Java programs. The mandatory argument is the COM-port. Optional switch `-e` keeps the program running after the download and echoes the characters from the serial line (`System.out` in JOP) to stdout. Switch `-usb` disables the echo protocol to speed up the download over USB.

`e.exe` Echoes the characters from the serial line to stdout. Parameter is the COM-port.

`amd.exe` A utility to send data over the serial line to program the on-board Flash. The complementary Java program `util.Amd` must be running on JOP.

`USBRunner.exe` Download the FPGA configuration via USB with the FTDI2232C chip (dpsio board).

[3] The Java JDK still comes without the javax.comm package and getting this optional package correctly installed is not that easy.

Generation of Files

These tools are written in Java and are delivered in source form. The source can be found under java/tools/src and the class files are in jop-tools.jar in directory java/tools/dist/lib.

Jopa The JOP assembler. Assembles the microcoded JVM and produces on-chip memory initialization files and VHDL files.

BlockGen converts Altera memory initialization files to VHDL files for a Xilinx FPGA.

JOPizer links a Java application and converts the class information to the format that JOP expects (a .jop file). JOPizer uses the bytecode engineering library[4] (BCEL).

Simulation

JopSim reads a .jop file and executes it in a debug JVM written in Java. Command line option -Dlog="true" prints a log entry for each executed JVM bytecode.

pcsim simulates the BaseIO expansion board for Java debugging on a PC (using the JVM on the PC).

2.3.2 Targets

JOP has been successfully ported to several different FPGAs and boards. The main distribution contains the ports for the FPGAs:

- Altera Cyclone EP1C6 or EP1C12
- Xilinx Spartan-3
- Altera Cyclone-II (Altera DE2 board)
- Xilinx Virtex-4 (ML40x board)
- Xilinx Spartan-3E (Digilent Nexys 2 board)

For the current list of the supported FPGA boards see the list at the web site.[5] Besides the ports to different FPGAs there are ports to different boards.

[4] http://jakarta.apache.org/bcel/
[5] http://www.jopwiki.com/FPGA_boards

Cyclone EP1C6/12

This board is the workhorse for the JOP development and comes in two versions: with an Cyclone EP1C6 or EP1C12. The schematics can be found in Appendix E. The board contains:

- Altera Cyclone EP1C6Q240 or EP1C12Q240 FPGA
- 1 MB fast SRAM
- 512 KB Flash (for FPGA configuration and program code)
- 32 MB NAND Flash
- ByteBlasterMV port
- Watchdog with LED
- EPM7064 PLD to configure the FPGA from the Flash (on watchdog reset)
- Voltage regulator (1V5)
- Crystal clock (20 MHz) at the PLL input (up to 640 MHz internal)
- Serial interface (MAX3232)
- 56 general purpose I/O pins

The Cyclone specific files are jopcyc.vhd or jopcyc12 and mem32.vhd. This FPGA board is designed as a module to be integrated with an application specific I/O-board. There exist following I/O-boards:

simpexp A simple bread board with a voltage regulator and a SUBD connector for the serial line

baseio A board with Ethernet connection and EMC protected digital I/O and analog input

bg263 Interface to a GPS receiver, a GPRS modem, keyboard and a display for a railway application

lego Interface to the sensors and motors of the LEGO Mindstorms. This board is a substitute for the LEGO RCX.

I/O board	Quartus	I/O top level
simpexp, baseio	cycmin	scio_min.vhd
dspio	usbmin	scio_dspiomin.vhd
baseio	cycbaseio	scio_baseio.vhd
bg263	cybg	scio_bg.vhd
lego	cyclego	scio_lego.vhd
dspio	dspio	scio_dspio.vhd

Table 2.1: Quartus project directories and VHDL files for the different I/O boards

dspio Developed at the University of Technology Vienna, Austria for digital signal processing related work. All design files for this board are open-source.

Table 2.1 lists the related VHDL files and Quartus project directories for each I/O board.

Xilinx Spartan-3

The Spartan-3 specific files are jop_xs3.vhd and mem_xs3.vhd for the Xilinx Spartan-3 Starter Kit and jop_trenz.vhd and mem_trenz.vhd for the Trenz Retrocomputing board.

2.4 Eclipse

In folder eclipse there are four Eclipse projects that you can import into your Eclipse workspace. However, do not use *that* directory as your workspace directory. Choose a directory outside of the JOP source tree for the workspace.

All projects use the Eclipse path variable[6] JOP_HOME that has to point to the root directory (.../jop) of the JOP sources. Under *Window – Preferences...* select *General – Workspace – Linked Resources* and create the path variable JOP_HOME with *New...*.

Import the projects with *File – Import..* and *Existing Projects into Workspace*. It is suggested to an Eclipse workspace that is not part of the jop source tree. Select as root directory .../jop/eclipse, select the projects you want to import, select *Copy projects into workspace*, and press *Finish*. Table 2.2 shows all available projects.

Add the libraries from .../jop/java/lib (as external archives) to the build path (right click on the joptools project) of the project joptools.[7]

[6] Eclipse (path) variables are workspace specific.

[7] Eclipse can't use path variables for external .jar files.

Project	Content
jop	The target sources
joptools	Tools such as Jopa, JopSim, and JOPizer
pc	Some PC utilities (e.g. Flash programming via UDP/IP)
pcsim	Simulation of the basio hardware on the PC

Table 2.2: Eclipse projects

2.5 Simulation

This section contains the information you need to get a simulation of JOP running. There are two ways to simulate JOP:

- High-level JVM simulation with JopSim
- VHDL simulation (e.g. with ModelSim)

2.5.1 JopSim Simulation

The high level simulation with JopSim is a simple JVM written in Java that can execute the JOP specific application (the .jop file). It is started with:

```
make jsim
```

To output each executing bytecode during the simulation run change in the Makefile the logging parameter to -Dlog="true".

2.5.2 VHDL Simulation

This section is about running a VHDL simulation with ModelSim. All simulation files are vendor independent and should run on any versions of ModelSim or a different VHDL simulator. You can simulate JOP even with the free ModelSim XE II Starter Xilinx version, the ModelSim Altera version or the ModelSim Actel version.

To simulate JOP, or any other processor design, in a vendor neutral way, models of the internal memories (block RAM) and the external main memory are necessary. Beside this, only a simple clock driver is necessary. To speed-up the simulation a little bit, a simulation of the UART output, which is used for System.out.print(), is also part of the package.

VHDL file	Function	Initialization file	Generator
sim_jop_types_100.vhd	JOP constant definitions	-	-
sim_rom.vhd	JVM microcode ROM	mem_rom.dat	Jopa
sim_ram.vhd	Stack RAM	mem_ram.dat	Jopa
sim_jbc.vhd	Bytecode memory (cache)	-	-
sim_memory.vhd	Main memory	mem_main.dat	jop2dat
sim_pll.vhd	A dummy entity for the PLL	-	-
sim_uart.vhd	Print characters to stdio	-	-

Table 2.3: Simulation specific VHDL files

Table 2.3 lists the simulation files for JOP and the programs that generates the initialization data. The non-generated VHDL files can be found in directory vhdl/simulation. The needed VHDL files and the compile order can be found in sim.bat under modelsim.

The actual version of JOP contains all necessary files to run a simulation with ModelSim. In directory vhdl/simulation you will find:

- A test bench: tb_jop.vhd with a serial receiver to print out the messages from JOP during the simulation
- Simulation versions of all memory components (vendor neutral)
- Simulation of the main memory

Jopa generates various mem_xxx.dat files that are read by the simulation. The JVM that is generated with jopsim.bat assumes that the Java application is preloaded in the main memory. jop2dat generates a memory initialization file from the Java application file (MainClass.jop) that is read by the simulation of the main memory (sim_memory.vhd).

In directory modelsim you will find a small batch file (sim.bat) that compiles JOP and the test bench in the correct order and starts ModelSim. The whole simulation process (including generation of the correct microcode) is started with:

```
make sim
```

After a few seconds you should see the startup message from JOP printed in ModelSim's command window. The simulation can be continued with run -all and after around 6 ms *simulation time* the actual Java main() method is executed. During those 6 ms, which will probably be minutes of simulation, the memory is initialized for the garbage collector.

2.6 Files Types You Might Encounter

As there are various tools involved in the complete build process, you will find files with various extensions. The following list explains the file types you might encounter when changing and building JOP.

The following files are the *source* files:

.vhd VHDL files describe the hardware part and are compiled with either Quartus or Xilinx ISE. Simulation in ModelSim is also based on VHDL files.

.v Verilog HDL. Another hardware description language. Used more in the US.

.java Java — the language that runs native on JOP.

.c There are still some tools written in C.

.asm JOP microcode. The JVM is written in this stack oriented assembler. Files are assembled with Jopa. The result are VHDL files, .mif files, and .dat files for ModelSim.

.bat Usage of these DOS batch files still prohibit running the JOP build under Unix. However, these files get less used as the Makefile progresses.

.xml Project files for Ant. Ant is an attractive substitution to make. Future distributions on JOP will be ant based.

Quartus II and Xilinx ISE need configuration files that describe your project. All files are usually ASCII text files.

.qpf Quartus II Project File. Contains almost no information.

.qsf Quartus II Settings File defines the project. VHDL files that make up your project are listed. Constraints such as pin assignments and timing constraints are set here.

.cdf Chain Description File. This file stores device name, device order, and programming file name information for the programmer.

.tcl Tool Command Language. Can be used in Quartus to automate parts of the design flow (e.g. pin assignment).

.npl Xilinx ISE project. VHDL files that make up your project are listed. The actual version of Xilinx ISE converts this project file to a new format that is not in ASCII anymore.

.ucf Xilinx Foundation User Constraint File. Constraints such as pin assignments and timing constraints are set here.

The Java tools javac and jar produce following file types from the Java sources:

.class A class file contains the bytecodes, a symbol table and other ancillary information and is executed by the JVM.

.jar The Java Archive file format enables you to bundle multiple files into a single archive file. Typically a .jar file contains the class files and auxiliary resources. A .jar file is essentially a zip file that contains an optional META-INF directory.

The following files are generated by the various tools from the source files:

.jop This file makes up the linked Java application that runns on JOP. It is generated by JOPizer and can be either downloaded (serial line or USB) or stored in the Flash (or used by the simulation with JopSim or ModelSim)

.mif Memory Initialization File. Defines the initial content of on-chip block memories for Altera devices.

.dat memory initialization files for the simulation with ModelSim.

.sof SRAM Output File. Configuration file for Altera devices. Used by the Quartus programmer or by quartus_pgm. Can be converted to various (or too many) different format. Some are listed below.

.pof Programmer Object File. Configuration for Altera devices. Used for the Flash loader PLDs.

.jbc JamTM STAPL Byte Code 2.0. Configuration for Altera devices. Input file for jbi32.

.ttf Tabular Text File. Configuration for Altera devices. Used by flash programming utilities (amd and udp.Flash to store the FPGA configuration in the boards Flash.

.rbf Raw Binary File. Configuration for Altera devices. Used by the USB download utility (USBRunner) to configure the dspio board via the USB connection.

.bit Bitstream File. Configuration file for Xilinx devices.

2.7 Information on the Web

Further information on JOP and the build process can be found on the Internet at the following places:

- `http://www.jopdesign.com/` is the main web site for JOP
- `http://www.jopwiki.com/` is a Wiki that can be freely edited by JOP users.
- `http://tech.groups.yahoo.com/group/java-processor/` hosts a mailing list for discussions on Java processors in general and mostly on JOP related topics

2.8 Porting JOP

Porting JOP to a different FPGA platform or board usually consists of adapting pin definitions and selection of the correct memory interface. Memory interfaces for the SimpCon interconnect can be found in directory vhdl/memory.

2.8.1 Test Utilities

To verify that the port of JOP is successful there are some small test programs in asm/src. To run the JVM on JOP the microcode jvm.asm is assembled and will be stored in an on-chip ROM. The Java application will then be loaded by the first microcode instructions in jvm.asm into an external memory. However, to verify that JOP and the serial line are working correctly, it is possible to run small test programs directly in microcode.

One test program (blink.asm) does not need the main memory and is a first test step before testing the possibly changed memory interface. testmon.asm can be used to debug the main memory interface. Both test programs can be built with the make targets jop_blink_test and jop_testmon.

Blinking LED and UART output

The test is built with:

```
make jop_blink_test
```

After download, the watchdog LED should blink and the FPGA will print out 0 and 1 on the serial line. Use a terminal program or the utility e.exe to check the output from the serial line.

Test Monitor

Start a terminal program (e.g. HyperTerm) to communicate with the monitor program and build the test monitor with:

```
make jop_testmon
```

After download the program prints the content of the memory at address 0. The program understands following *commands*:

- A single CR reads the memory at the current addres and prints out the address and memory content
- addr=val; writes *val* into the memory location at address *addr*

One tip: Take care that your terminal program does not send an LF after the CR.

2.9 Extending JOP

JOP is a soft-core processor and customizing it for an application is an interesting opportunity.

2.9.1 Native Methods

The *native* language of JOP is microcode. A native method is implemented in JOP microcode. The interface to this native method is through a *special* bytecode. The mapping between native methods and the special bytecode is performed by JOPizer. When adding a new (*special*) bytecode to JOP, the following files have to be changed:

1. jvm.asm implementation
2. Native.java method signature
3. JopInstr.java mapping of the signature to the name
4. JopSim.java simulation of the bytecode
5. JVM.java (just rename the method name)
6. Startup.java (only when needed in a class initializer)

7. WCETInstruction.java timing information

First implement the native code in JopSim.java for easy debugging. The *real* microcode is added in jvm.asm with a label for the special byctecode. The naming convention is jopsys_name. In Native.java provide a method signature for the native method and enter the mapping between this signature and the name in jvm.asm and in JopInstr.java. Provide the execution time in WCETInstruction.java for the WCET analysis.

The native method is accessed by the method provided in Native.java. There is no calling overhead involved in the mechanism. The *native* method gets substituted by JOPizer with a *special* bytecode.

2.9.2 A new Peripheral Device

Creation of a new peripheral devices involves some VHDL coding. However, there are several examples in jop/vhdl/scio available.

All peripheral components in JOP are connected with the SimpCon [110] interface. For a device that implements the Wishbone [80] bus, a SimpCon-Wishbone bridge (sc2wb.vhd) is available (e.g., it is used to connect the AC97 interface in the dspio project).

For an easy start use an existing example and change it to your needs. Take a look into sc_test_slave.vhd. All peripheral components (SimpCon slaves) are connected in one module usually named scio_xxx.vhd. Browse the examples and copy one that best fits your needs. In this module the address of your peripheral device is defined (e.g. 0x10 for the primary UART). This I/O address is mapped to a negative memory address for JOP. That means 0xffffff80 is added as a base to the I/O address.

By convention this address mapping is defined in com.jopdesign.sys.Const. Here is the UART example:

```
// use negative base address for fast constant load
// with bipush
public static final int IO_BASE = 0xffffff80;
...
public static final int IO_STATUS = IO_BASE+0x10;
public static final int IO_UART = IO_BASE+0x10+1;
```

The I/O devices are accessed from Java by *native*[8] functions: Native.rdMem() and Native.wrMem() in pacakge com.jopdesign.sys. Again an example with the UART:

[8]These are not real functions and are substituted by special bytecodes on application building with JOPizer.

```
// busy wait on free tx buffer
// no wait on an open serial line, just wait
// on the baud rate
while ((Native.rdMem(Const.IO_STATUS)&1)==0) {
    ;
}
Native.wrMem(c, Const.IO_UART);
```

Best practise is to create a new I/O configuration scio_xxx.vhdl and a new Quartus project for this configuration. This avoids the mixup of the changes with a new version of JOP. For the new Quartus project only the three files jop.cdf, jop.qpf, and jop.qsf have to be copied in a new directory under quartus. This new directory is the project name that has to be set in the Makefile:

```
QPROJ=yourproject
```

The new VHDL module and the scio_xxx.vhdl are added in jop.qsf. This file is a plain ASCII file and can be edited with a standard editor or within Quartus.

2.9.3 A Customized Instruction

A customized instruction can be simply added by implementing it in microcode and mapping it to a native function as described before. If you want to include a hardware module that implements this instruction a new microinstruction has to be introduced. Besides mapping this instruction to a native method the instruction has also be added to the microcode assembler Jopa.

2.9.4 Dependencies and Configurations

As JOP and the JVM are a mix of VHDL and Java files, changes in the central data structures or some configurations needs an update in several files.

Stack Size

The on-chip stack size can be configured by changing following constants:

- ram_width in jop_config_xx.vhd
- STACK_SIZE in com.jopdesign.sys.Const
- RAM_LEN in com.jopdesign.sys.Jopa

Changing the Class Format

- JOPizer: CLS_HEAD, dump()
- GC.java uses CLASS_HEADR
- JMV.java uses CLASS_HEADR + offset (checkcast, instanceof)

2.10 Directory Structure

The top-level directories of the distribution are:

asm Microcode source files. The microcode part of the JVM and test files.

bat Old batch files – *not used – TODO: if they are no longer used, they should be moved out of the top-level directory or removed entirely.*

boards Pictures and text for the Eclipse plugin

c_src Some utilities in C (e.g. `down.exe` and `e.exe`).

doc LaTeXsources for this handbook and short notes.

eclipse Eclipse project files

ext External VHDL and Verilog sources

java All Java files

- **lib** External .jar files
- **pc** Tools on the PC
- **pcsim** High-level simulation on the PC
- **target** The Java sources for JOP
- **tools** All Java tools

jbc FPGA configuration files for `jbi32.exe` (generated)

jopc A C version of a JOP JVM simulation – *very* outdated

linux Scripts to start a network and SLIP

modelsim ModelSim simulation

pins Pin definitions for FPGA boards

quartus Quartus project files

rbf FPGA configuration files for USBRunner (generated)

sopc JOP as SoPC component and SRAM components

support Stand-alone Flash programming for the Cycore board

ttf FPGA configuration files for Flash programming (generated)

vhdl The processor sources

- **altera** Altera specific components (PLL, RAM)
- **config** Cycore PLD sources
- **core** The processor core
- **fpu** The floating-point unit
- **memory** Main memory connections via SimpCon
- **scio** IO components and configurations with SimpCon
- **simpcon** SimpCon bridges and arbiter
- **simulation** Memory and UART for ModelSim simulation
- **start** The VHDL version of *hello world* – a blinking LED
- **testbenches** no real content
- **top** Top-level and configuration (e.g. PLL setting) components
- **vga** A SimpCon VGA controller
- **xilinx** Xilinx specific components (RAM)

xilinx Xilinx project files

2.10.1 The Java Sources for JOP

The most important directory for all Java sources that run on JOP is in java/target.

dist Generated files

- **bin** The linked application (.jop)
- **classes** The class files
- **lib** The application class files in classes.zip – input for JOPizer

src The source

- **app** The applications
- **bench** The embedded benchmark suit
- **common** Utility classes
- **jdk_base** Base classes for the JDK
- **jdk11** JDK around version 1.1
- **jdk14** A test port of JDK 1.4 classes
- **rtapi** Experimental RT API dfinitions
- **test** Various test programs

wcet Output from the WCET analyzer (generated)

2.11 The JOP Hello World Exercise

This exercise gives an introduction into the design flow of JOP. JOP will be built from the sources and a simple *Hello World* program will run on it.

To understand the build process you have to run the build manually. This understanding will help you to find the correct files for changes in JOP and to adjust the Makefile for your needs.

2.11.1 Manual build

Manual build does not mean entering all commands, but calling the correct make target with the required arguments (if any) in the correct order. The idea of this exercise is to obtain knowledge of the directory structure and the dependencies of various design units.

Inspect the Makefile targets and the ones that are called from it before running them.

1. Create your working directory
2. Download the sources from the opencores CVS server
3. Connect the FPGA board to the PC (and the power supply)
4. Perform the build as described in Section 2.1.2.

As a result you should see a message at your command prompt.

2.11.2 Using make

In the root directory (jop) there is a Makefile. Open it with an editor and try to find the corresponding lines of code for the steps you did in the first exercise. Reset the FPGA by cycling the power and run the build with a simple

```
make
```

The whole process should run without errors and the result should be identical to the previous exercise.

2.11.3 Change the Java Program

The whole build process is not necessary when changing the Java application. Once the processor is built, a Java application can be built and downloaded with the following make target:

```
make japp
```

Change HelloWorld.java and run it on JOP. Now change the class name that contains the main() method from HelloWorld to Hello and rerun the Java application build. Now an embedded version of "Hello World" should run on JOP. Besides the usual greeting on the standard output, the LED on the FPGA board should blink at a frequency of 1 Hz. The first periodic task, an essential abstraction for real-time systems, is running on JOP!

2.11.4 Change the Microcode

The JVM is written in microcode and several .vhdl files are generated during assembly. For a test change only the version string[9] in jvm.asm to the actual date and run a full make.

```
version     = 20070831
```

change to:

```
version     = 20110101
```

The start message should reflect your change. As the microcode was changed a full make run is necessary. The microcode assembly generates VHDL files and the complete processor has to be rebuilt.

[9] The actual version date will probably be different from the actual sources.

	simpexp	dspio
FPGA	EP1C6	EP1C12
IO	UART	UART, USB, audio codec, sigma-delta codec
FPGA configuration	ByteBlaster	USBRunner
Java download	serial line	USB

Table 2.4: Differences between the two target boards

2.11.5 Use a Different Target Board

In this exercise, you will alter the Makefile for a different target board. Disconnect the first board and connect the board with an USB port (e.g. the dspio or Lego board).

Table 2.4 lists the differences between the first board (simpexp) and the new one (called dspio). The correct FPGA is already selected in the Quartus project files (jop.qpf). Alter the Makefile to set the variable USB to true. This will change:

1. The Quartus project from cycmin to usbmin
2. The Java download is now over USB instead of the serial line
3. The parameters for the download via down.exe are changed to use the virtual com-port of the USB driver (look into Windows hardware manager to get the correct number) and the switch -usb for the download is added

Now build the whole project with make. Change the Java program and perform only the necessary build step.

2.11.6 Compile a Different Java Application

The class that contains the main method is described by three arguments:

1. The first directory relative to java/target/src (e.g. app or test)
2. The package name (e.g. dsp)
3. The main class (e.g. HalloWorld)

These three values are used by the Makeile and are set in the variables P1, P2, and P3 in the Makefile.

Change the Makefile to compile the embedded Java benchmark jbe.DoAll. The parameters for the Java application can also be given to the make with following command line arguments:

```
make -e P1=bench P2=jbe P3=DoAll
```

The three variables P1, P2, and P3 are a shortcut to set the main class of the application. You can also directly set the variables TARGET_APP_PATH and MAIN_CLASS.

2.11.7 Simulation

This exercise will give you a full view of the possibilities to debug JOP system code or the processor itself. There are two ways to simulate JOP: A simple debugging JVM written in Java (JopSim as part of the tool package) that can execute *jopized* applications and a VHDL level simulation with ModelSim. The make targets are jsim and sim.

2.11.8 WCET Analysis

An important step in real-time system development is the analysis of the WCET of the individual tasks. Compile and run the WCET example Loop.java in package wcet. You can analyze the WCET of the method measure() with following make command:

```
make java_app wcet -e P1=test P2=wcet P3=Loop
```

Change the code in Loop.java to enable measurement of the execution time and compare it with the output of the static analysis. In this simple example the WCET can be measured. However, be aware that most non-trivial code needs static analysis for safe estimates of WCET values.

3 Java and the Java Virtual Machine

Java technology consists of the Java language definition, a definition of the standard library, and the definition of an intermediate instruction set with an accompanying execution environment. This combination helps to make *write once, run anywhere* possible.

The following chapter gives a short overview of the Java programming language. A more detailed description of the Java Virtual Machine (JVM) and the explanation of the JVM instruction set, the so-called bytecodes, follows. The exploration of dynamic instruction counts of typical Java programs can be found in Section **??**.

3.1 Java

Java is a relatively new and popular programming language. The main features that have helped Java achieve success are listed below:

Simple and object oriented: Java is a simple programming language that appears very similar to C. This 'look and feel' of C means that programmers who know C, can switch to Java without difficulty. Java provides a simplified object model with single inheritance[1].

Portability: To accommodate the diversity of operating environments, the Java compiler generates bytecodes – an architecture neutral intermediate format. To guarantee platform independence, Java specifies the sizes of its basic data types and the behavior of its arithmetic operators. A Java interpreter, the Java virtual machine, is available on various platforms to help make 'write once, run anywhere' possible.

Availability: Java is not only available for different operating systems, it is available at no cost. The runtime system and the compiler can be downloaded from Sun's website for Windows, Linux, and Solaris. Sophisticated development environments, such as Netbeans or Eclipse, are available under the GNU Public License.

[1] Java has *single inheritance* of *implementation* – only one class can be extended. However, a class can implement several interfaces, which means that Java has *multiple interface inheritance*.

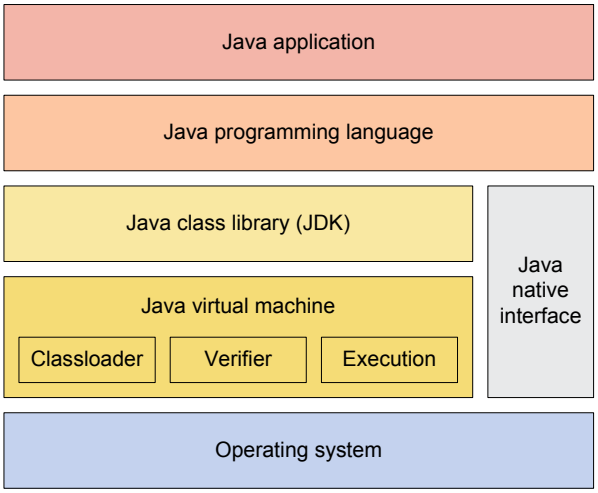

Figure 3.1: Java system overview

Library: The complete Java system includes a rich class library to increase programming productivity. Besides the functionality of a C standard library, it also contains other tools, such as collection classes and a GUI toolkit.

Built-in multithreading: Java supports multithreading at the language level: the library provides the `Thread` class, the language provides the keyword `synchronized` for critical sections and the runtime system provides monitor and condition lock primitives. The system libraries have been written to be thread-safe: the functionality provided by the libraries is available without conflicts due to multiple concurrent threads of execution.

Safety: Java provides extensive compile-time checking, followed by a second level of runtime checking. The memory management model is simple – objects are created with the `new` operator. There are no explicit pointer data types and no pointer arithmetic, but there is automatic garbage collection. This simple memory management model eliminates a large number of the programming errors found in C and C++ programs. A restricted runtime environment, the so-called *sandbox*, is available when executing small Java applications in Web browsers.

As can be seen in Figure 3.1, Java consists of three main components:

1. The Java programming language as defined in [42]

2. The class library, defined as part of the Java specification. All implementations of Java have to contain the library as defined by Sun

3. The Java virtual machine (defined in [69]) that loads, verifies and executes the binary representation (the *class file*) of a Java program

The Java native interface supports functions written in C or C++. This combination is sometimes called *Java technology* to emphasize the fact that Java is more than just another object-oriented language.

However, a number of issues have slowed down the broad acceptance of Java. The original presentation of Java as an Internet language led to the misconception that Java was not a general-purpose programming language. Another obstacle was the first implementation of the JVM as an interpreter. Execution of Java programs was *very* slow compared to compiled C/C++ programs. Although advances in its runtime technology, in particular the just-in-time compiler, have closed the performance gap, it is still a commonly held view that Java is slow.

3.1.1 History

The Java programming language originated as part of the Green project specifically for an embedded device, a handheld wireless PDA. In the early '90s, Java, which was originally known as Oak [122, 124], was created as the programming tool for this device. The device (known as *7) was a small SPARC-based hardware device with a tiny embedded OS. However, the *7 was never released as a product and Java was officially released in 1995 as the *new* language for the Internet. Over the years, Java technology has become a programming tool for desktop applications, web servers and server applications. These application domains resulted in the split of the Java platform into the Java standard edition (J2SE) and the enterprise edition (J2EE) in 1999. With every new release, the library (defined as part of the language) continued to grow. Java for embedded systems was clearly not an area Sun was interested in pursuing. However, with the arrival of mobile phones, Sun again became interested in this embedded market. Sun defined different subsets of Java, which have now been combined into the Java Micro Edition (J2ME).

In 1999, a document defining the requirements for real-time Java was published by the NIST [75]. Based on these requirements, two groups defined specifications for real-time Java: the Real-Time Core Extension [120] published under the J Consortium and the Real-Time Specification for Java (RTSJ) [21]. A comparison of these two specifications and a comparison with Ada 95's Real-Time Annex can be found in [22]. The RTSJ was the first Java Specification Request (JSR 1) under the Java Community Process (JCP) and started

Type	Description
boolean	either true or false
char	16-bit Unicode character (unsigned)
byte	8-bit integer (signed)
short	16-bit integer (signed)
int	32-bit integer (signed)
long	64-bit integer (signed)
float	32-bit floating-point (IEEE 754-1985)
double	64-bit floating-point (IEEE 754-1985)

Table 3.1: Java primitive data types

1999. The first release came out 2002 and further enhancement of the RTSJ (to version 1.1) are covered by the JSR 282 (started in 2005). Under JSR 302 (Safety Critical Java Technology) a subset of the RTSJ is currently defined for the safety critical domain (e.g., standard DO-178B/ED-12B [99]). A detailed description of the J2ME and specifications for real-time Java can be found in Chapter 4 of [107].

3.1.2 The Java Programming Language

The Java programming language is a general-purpose object-oriented language. Java is related to C and C++, but with a number of aspects omitted. Java is a strongly typed language, which means that type errors can be detected at compile time. Other errors, such as wrong indices in an array, are checked at runtime. The problematic[2] *pointer* in C and explicit deallocation of memory is completely avoided. The pointer is replaced by a *reference*, i.e., an abstract pointer to an object. Storage for an object is allocated from the heap during creation of the object with new. Memory is freed by automatic storage management, typically using a garbage collector. The garbage collector avoids memory leaks from a missing free() and the safety problems exposed by dangling pointers.

The types in Java are divided into two categories: primitive types and reference types. Table 3.1 lists the available primitive types. Method local variables, class fields and object fields contain either a primitive type value or a reference to an object.

Classes and class instances, the objects, are the fundamental data and code organization

[2] C pointers represent memory addresses as data. Pointer arithmetic and direct access to memory leads to common and hard-to-find program errors.

structures in Java. There are no global variables or functions as there are in C/C++. Each method belongs to a class. This 'everything belongs to a class or an object' combined with the class naming convention, as suggested by Sun, avoids name conflicts in even the largest applications.

New classes can extend exactly one superclass. Classes that do not explicitly extend a superclass become direct subclasses of Object, the root of the whole class tree. This single inheritance model is extended by *interfaces*. Interfaces are abstract classes that only define method signatures and provide no implementation. A concrete class can implement several interfaces. This model provides a simplified form of multiple inheritance.

Java supports multitasking through *threads*. Each thread is a separate flow of control, executing concurrently with all other threads. A thread contains the method stack as thread local data – all objects are shared between threads. Access conflicts to shared data are avoided by the proper use of synchronized methods or code blocks.

Java programs are compiled to a machine-independent bytecode representation as defined in [69]. Although this intermediate representation is defined for Java, other programming languages (e.g., ADA [26]) can also be compiled into Java bytecodes.

3.2 The Java Virtual Machine

The Java virtual machine (JVM) is a definition of an abstract computing machine that executes bytecode programs. The JVM specification [69] defines three elements:

- An instruction set and the meaning of those instructions – the *bytecodes*
- A binary format – the *class file* format. A class file contains the bytecodes, a symbol table and other ancillary information
- An algorithm to *verify* that a class file contains valid programs

In the solution presented in this book, the class files are verified, linked and transformed into an internal representation before being executed on JOP. This transformation is performed with JOPizer and is not executed on JOP. We will therefore omit the description of the class file and the verification process.

The instruction set of the JVM is stack-based. All operations take their arguments from the stack and put the result onto the stack. Values are transferred between the stack and various memory areas. We will discuss these memory areas first, followed by an explanation of the instruction set.

3.2.1 Memory Areas

The JVM contains various runtime data areas. Some of these areas are shared between threads, whereas other data areas exist separately for each thread.

Method area: The method area is shared among all threads. It contains static class information such as field and method data, the code for the methods and the constant pool. The constant pool is a per-class table, containing various kinds of constants such as numeric values or method and field references. The constant pool is similar to a symbol table.

Part of this area, the code for the methods, is very frequently accessed (during instruction fetch) and therefore is a good candidate for caching.

Heap: The heap is the data area where all objects and arrays are allocated. The heap is shared among all threads. A garbage collector reclaims storage for objects.

JVM stack: Each thread has a private stack area that is created at the same time as the thread. The JVM stack is a logical stack that contains following elements:

1. A frame that contains return information for a method
2. A local variable area to hold local values inside a method
3. The operand stack, where all operations are performed

Although it is not strictly necessary to allocate all three elements to the same type of memory we will see in Section 4.4 that the argument-passing mechanism regulates the layout of the JVM stack.

Local variables and the operand stack are accessed as frequently as registers in a standard processor. A Java processor should provide some caching mechanism of this data area.

The memory areas are similar to the various segments in conventional processes (e.g. the method code is analogous to the 'text' segment). However, the operand stack replaces the registers in a conventional processor.

3.2.2 JVM Instruction Set

The instruction set of the JVM contains 201 different instructions [69]. This *bytecodes* can be grouped into the following categories:

Load and store: Load instructions push values from the local variables onto the operand stack. Store instructions transfer values from the stack back to local variables. 70 different instructions belong to this category. Short versions (single byte) exist to access the first four local variables. There are unique instructions for each basic type (int, long, float, double and reference). This differentiation is necessary for the bytecode verifier, but is not needed during execution. For example iload, fload and aload all transfer one 32-bit word from a local variable to the operand stack.

Arithmetic: The arithmetic instructions operate on the values found on the stack and push the result back onto the operand stack. There are arithmetic instructions for int, float and double. There is no direct support for byte, short or char types. These values are handled by int operations and have to be converted back before being stored in a local variable or an object field.

Type conversion: The type conversion instructions perform numerical conversions between all Java types: as implicit widening conversions (e.g., int to long, float or double) or explicit (by casting to a type) narrowing conversions.

Object creation and manipulation: Class instances and arrays (that are also objects) are created and manipulated with different instructions. Objects and class fields are accessed with type-less instructions.

Operand stack manipulation: All direct stack manipulation instructions are type-less and operate on 32-bit or 64-bit entities on the stack. Examples of these instructions are dup, to duplicate the top operand stack value, and pop, to remove the top operand stack value.

Control transfer: Conditional and unconditional branches cause the JVM to continue execution with an instruction other than the one immediately following. Branch target addresses are specified relative to the current address with a signed 16-bit offset. The JVM provides a complete set of branch conditions for int values and references. Floating-point values and type long are supported through compare instructions. These compare instructions result in an int value on the operand stack.

Method invocation and return: The different types of methods are supported by four instructions: invoke a class method, invoke an instance method, invoke a method that implements an interface and an invokespecial for an instance method that requires special handling, such as private methods or a superclass method.

A bytecode consists of one instruction byte followed by optional operand bytes. The length of the operand is one or two bytes, with the following exceptions: multianewarray contains 3 operand bytes; invokeinterface contains 4 operand bytes, where one is redundant and one is always zero; lookupswitch and tableswitch (used to implement the Java switch statement) are variable-length instructions; and goto_w and jsr_w are followed by a 4 byte branch offset, but neither is used in practice as other factors limit the method size to 65535 bytes.

3.2.3 Methods

A Java *method* is equivalent to a *function* or *procedure* in other languages. In object oriented terminology this *method* is *invoked* instead of *called*. We will use *method* and *invoke* in the remainder of this text. In Java and the JVM, there are five types of methods:

- Static or class methods
- Virtual methods
- Interface methods
- Class initialization
- Constructor of the parent class (super())

For these five types there are only four different bytecodes:

invokestatic: A class method (declared static) is invoked. As the target does not depend on an object, the method reference can be resolved at load/link time.

invokevirtual: An object reference is resolved and the corresponding method is invoked. The resolution is usually done with a dispatch table per class containing all implemented and inherited methods. With this dispatch table, the resolution can be performed in constant time.

invokeinterface: An interface allows Java to emulate multiple inheritance. A class can implement several interfaces, and different classes (that have no inheritance relation) can implement the same interface. This flexibility results in a more complex resolution process. One method of resolution is a search through the class hierarchy that results in a variable, and possibly lengthy, execution time. A constant time resolution is possible by assigning every interface method a unique number. Each class that implements an interface needs its own table with unique positions for each interface method of the *whole* application.

```
for (;;) {
    instr = bcode[pc++];
    switch (instr) {
        ...
        case IADD:
            tos = stack[sp]+stack[sp-1];
            --sp;
            stack[sp] = tos;
            break;
        ...
    }
}
```

Listing 3.1: Typical JVM interpreter loop

invokespecial**:** Invokes an instance method with special handling for superclass, private, and instance initialization. This bytecode catches many different cases. This results in expensive checks for common private instance methods.

3.2.4 Implementation of the JVM

There are several different ways to implement a virtual machine. The following list presents these possibilities and analyses how appropriate they are for embedded devices.

Interpreter: The simplest realization of the JVM is a program that interprets the bytecode instructions. The interpreter itself is usually written in C and is therefore easy to port to a new computer system. The interpreter is very compact, making this solution a primary choice for resource-constrained systems. The main disadvantage is the high execution overhead. From a code fragment of the typical interpreter loop, as shown in Listing 3.1, we can examine the overhead: The emulation of the stack in a high-level language results in three memory accesses for a simple iadd bytecode. The instruction is decoded through an indirect jump. Indirect jumps are still a burden for standard branch prediction logic.

Just-In-Time Compilation: Interpreting JVMs can be enhanced with just-in-time (JIT) compilers. A JIT compiler translates Java bytecodes to native instructions during runtime. The time spent on compilation is part of the application execution time. JIT

compilers are therefore restricted in their optimization capacity. To reduce the compilation overhead, current JVMs operate in mixed mode: Java methods are executed in interpreter mode and the call frequency is monitored. Often-called methods, the hot spots, are then compiled to native code.

JIT compilation has several disadvantages for embedded systems, notably that a compiler (with the intrinsic memory overhead) is necessary on the target system. Due to compilation during runtime, execution times are hardly predictable.[3]

Batch Compilation: Java can be compiled, in advance, to the native instruction set of the target. Precompiled libraries are linked with the application during runtime. This is quite similar to C/C++ applications with shared libraries. This solution undermines the flexibility of Java: dynamic class loading during runtime. However, this is not a major concern for embedded systems.

Hardware Implementation: A Java processor is the implementation of the JVM in hardware. The JVM bytecode is the native instruction set of such a processor. This solution can result in quite a small processor, as a stack architecture can be implemented very efficiently. A Java processor is memory-efficient as an interpreting JVM, but avoids the execution overhead. The main disadvantage of a Java processor is the lack of capability to execute C/C++ programs. This book describes JOP as an example of a JVM hardware implementation.

3.3 Embedded Java

In embedded systems the architecture of JVMs are more diverse than on desktop or server systems. Figure 3.2 shows variations of Java implementations in embedded systems and an example of the control flow for a web server application. The standard approach of a JVM running on top of an operating system (OS) is shown in sub-figure (a). A network connection bypasses the JVM via native functions and uses the TCP/IP stack implementation and the device drivers of the OS.

A JVM without an OS is shown in sub-figure (b). This solution is often called *running on the bare metal*. The JVM acts as the OS and provides the thread scheduling and the low-level access to the hardware. In that case the network stack can be written entirely in

[3] Even if the time for the compilation is known, the WCET for a method has to include the compile time! Furthermore, WCET analysis has to know in advance what code will be produced by JIT compilation.

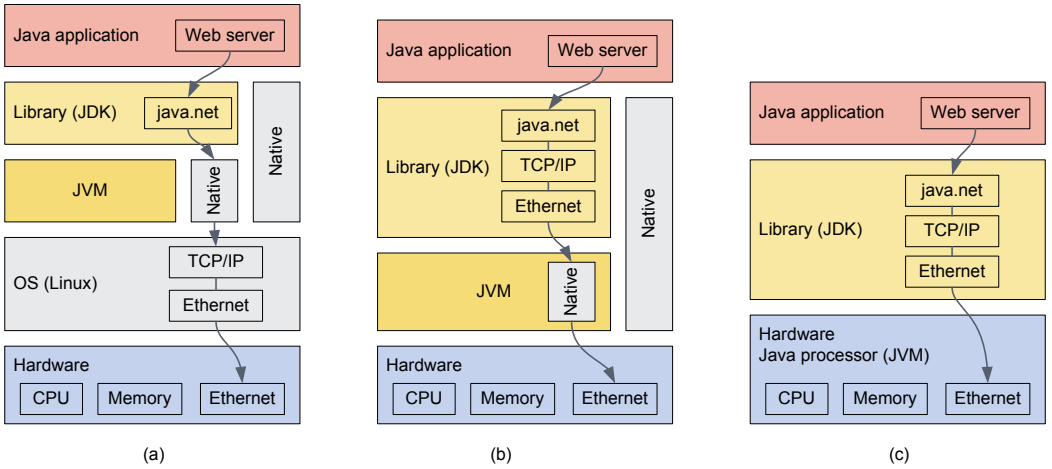

Figure 3.2: Implementation variations for an embedded JVM: (a) standard layers for Java with an operating system – equivalent to desktop configurations, (b) a JVM on the bare metal, and (c) a JVM as a Java processor.

Java. JNode[4] is an approach to implement the OS entirely in Java. This solution becomes popular even in server applications.[5]

Sub-figure (c) shows an embedded solution where the JVM is part of the hardware layer. That means it is implemented in a Java processor. With this solution the native layer can be completely avoided and all code (application and system code) is written entirely in Java.

Figure 3.2 shows how the flow from the application goes down to the hardware. The example consists of a web server and an Internet connection via Ethernet. In case (a) the application web server talks with `java.net` in the JDK. The flow goes through a native interface to the TCP/IP implementation and the Ethernet device driver within the OS (usually written in C). The device driver talks with the Ethernet chip. In (b) the OS layer is omitted: the TCP/IP layer and the Ethernet device driver are now part of the Java library. In (c) the JVM is part of the hardware layer and a direct access from the Ethernet driver to the Ethernet hardware is mandatory. Note how part of the network stack moves up from the OS layer to the Java library. Version (c) shows a pure Java implementation of the whole network stack.

[4] `http://www.jnode.org/`

[5] BEA System offers the JVM LiquidVM that includes basic OS functions and does not need a guest OS.

3.4 Summary

Java is a unique combination of the language definition, a rich class library and a runtime environment. A Java program is compiled to bytecodes that are executed by a Java virtual machine. Strong typing, runtime checks and avoidance of pointers make Java a *safe* language. The intermediate bytecode representation simplifies porting of Java to different computer systems. An interpreting JVM is easy to implement and needs few system resources. However, the execution speed suffers from interpreting. JVMs with a just-in-time compiler are state-of-the-art for desktop and server systems. These compilers require large amounts of memory and have to be ported for each processor architecture, which means they are not the best choice for embedded systems. A Java processor is the implementation of the JVM as a concrete machine. A Java processor avoids the slow execution model of an interpreting JVM and the memory requirements of a compiler, thus making it an interesting execution system for Java in embedded systems.

4 JOP Hardware Architecture

This chapter presents the architecture of JOP and the motivation behind the various different design decisions we faced. The first sections give an overview of JOP, describe the microcode and the pipeline.

Pipelined instruction processing calls for high memory bandwidth. Caches are needed in order to avoid bottlenecks resulting from the main memory bandwidth. As seen in Chapter 3, there are two memory areas that are frequently accessed by the JVM: the stack and the method area. In this chapter, time-predictable cache solutions for both areas that are implemented in JOP are presented.

4.1 Overview of JOP

This section gives an overview of JOP architecture. Figure 4.1 shows JOP's major functional units. A typical configuration of JOP contains the processor core, a memory interface and a number of IO devices. The module extension provides the link between the processor core, and the memory and IO modules.

The processor core contains the four pipeline stages *bytecode fetch*, *microcode fetch*, *decode* and *execute*. The ports to the other modules are the address and data bus for the bytecode instructions, the two top elements of the stack (A and B), input to the top-of-stack (Data) and a number of control signals. There is no direct connection between the processor core and the external world.

The memory interface provides a connection between the main memory and the processor core. It also contains the bytecode cache. The extension module controls data read and write. The *busy* signal is used by the microcode instruction `wait`[1] to synchronize the processor core with the memory unit. The core reads bytecode instructions through dedicated buses (BC address and BC data) from the memory subsystem. The request for a method to be placed in the cache is performed through the extension module, but the cache hit de-

[1] The busy signal can also be used to stall the whole processor pipeline. This was the change made to JOP by Flavius Gruian [44]. However, in this synchronization mode, the concurrency between the memory access module and the main pipeline is lost.

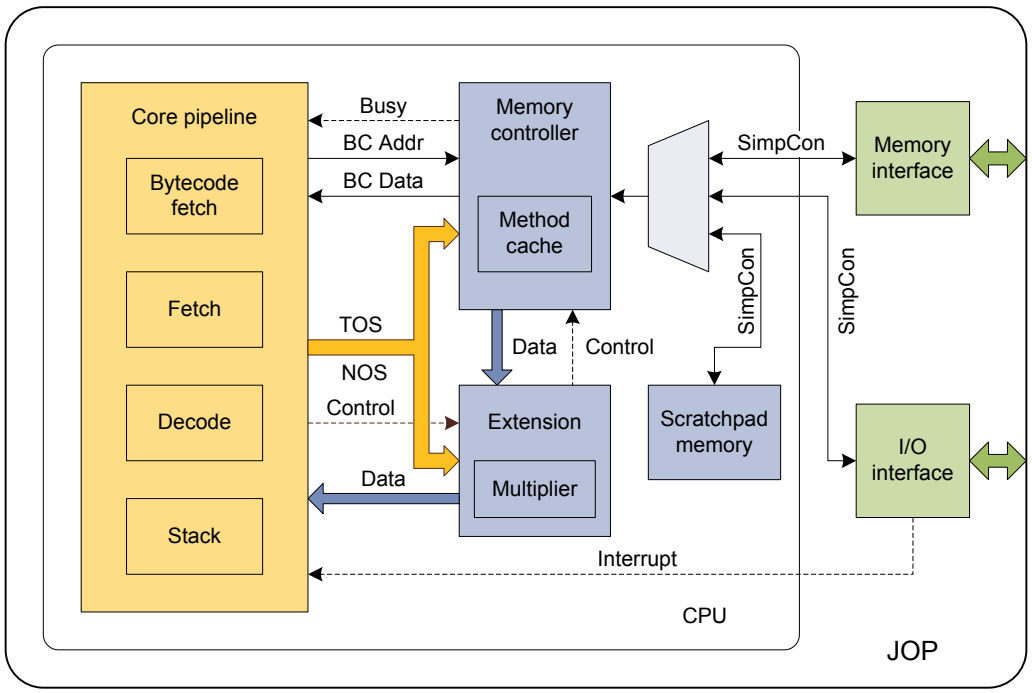

Figure 4.1: Block diagram of JOP

tection and load is performed by the memory interface independently of the processor core (and therefore concurrently).

The I/O interface contains peripheral devices, such as the system time and timer interrupt, a serial interface and application-specific devices. Read and write to and from this module are controlled by the extension module. All external devices[2] are connected to the I/O interface.

The extension module performs three functions: (a) it contains hardware accelerators (such as the multiplier unit in this example), (b) the control for the memory and the I/O module, and (c) the multiplexer for the read data that is loaded in the top-of-stack register. The write data from the top-of-stack (A) is connected directly to all modules.

The division of the processor into those four modules greatly simplifies the adaptation of JOP for different application domains or hardware platforms. Porting JOP to a new FPGA board usually results in changes in the memory module alone. Using the same board for different applications only involves making changes to the I/O module. JOP has been ported to several different FPGAs and prototyping boards and has been used in different applications (see Chapter 10), but it never proved necessary to change the processor core.

4.2 Microcode

The following discussion concerns two different instruction sets: *bytecode* and *microcode*. Bytecodes are the instructions that make up a compiled Java program. These instructions are executed by a Java virtual machine. The JVM does not assume any particular implementation technology. Microcode is the native instruction set for JOP. Bytecodes are translated, during their execution, into JOP microcode. Both instruction sets are designed for an extended[3] stack machine.

4.2.1 Translation of Bytecodes to Microcode

To date, no hardware implementation of the JVM exists that is capable of executing *all* bytecodes in hardware alone. This is due to the following: some bytecodes, such as `new`, which creates and initializes a new object, are too complex to implement in hardware. These bytecodes have to be emulated by software.

[2] The external device can be as simple as a line driver for the serial interface that forms part of the interface module, or a complete bus interface, such as the ISA bus used to connect e.g. an Ethernet chip.

[3] An extended stack machine is one in which there are instructions available to access elements deeper down in the stack.

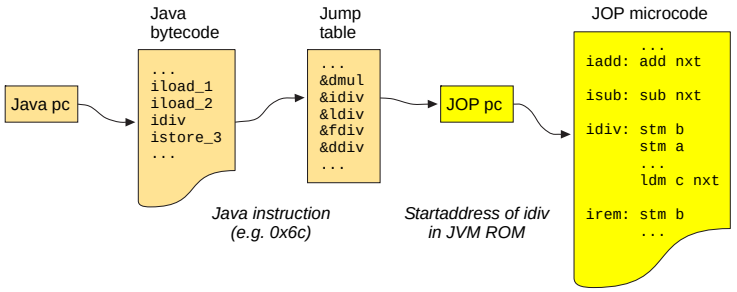

Figure 4.2: Data flow from the Java program counter to JOP microcode

To build a self contained JVM without an underlying operating system, direct access to the memory and I/O devices is necessary. There are no bytecodes defined for low-level access. These low-level services are usually implemented in *native* functions, which means that another language (C) is native to the processor. However, for a Java processor, bytecode is the *native* language.

One way to solve this problem is to implement simple bytecodes in hardware and to emulate the more complex and *native* functions in software with a different instruction set (sometimes called microcode). However, a processor with two different instruction sets results in a complex design.

Another common solution, used in Sun's picoJava [125], is to execute a subset of the bytecode native and to use a software trap to execute the remainder. This solution entails an overhead (a minimum of 16 cycles in picoJava, see 11.2.1) for the software trap.

In JOP, this problem is solved in a much simpler way. JOP has a single *native* instruction set, the so-called microcode. During execution, every Java bytecode is translated to either one, or a sequence of microcode instructions. This translation merely adds one pipeline stage to the core processor and results in no execution overheads (except for a bytecode branch that takes 4 instead of 3 cycles to execute). The area overhead of the translation stage is 290 LCs, or about 15% of the LCs of a typical JOP configuration. With this solution, we are free to define the JOP instruction set to map smoothly to the stack architecture of the JVM, and to find an instruction coding that can be implemented with minimal hardware.

Figure 4.2 gives an example of the data flow from the Java program counter to JOP microcode. The figure represents the two pipeline stages bytecode fetch/translate and microcode fetch. The fetched bytecode acts as an index for the jump table. The jump table contains the start addresses for the bytecode implementation in microcode. This address is loaded into the JOP program counter for every bytecode executed. JOP executes the

sequence of microcode until the last one. The last one is marked with *nxt* in microcode assembler. This *nxt* bit in the microcode ROM triggers a new translation i.e., a new address is loaded into the JOP program counter. In Figure 4.2 the implementation of bytecode idiv is an example of a longer sequence that ends with microcode instruction ldm c nxt.

The difference to other forms of instruction translation in hardware is that the proposed solution is time predictable. The translation takes one cycle (one pipeline stage) for each bytecode, independent from the execution history. Instruction folding, e.g., implemented in picoJava [77, 125], is also a form of instruction translation in hardware. Folding is used to translate several (stack oriented) bytecode instructions to a RISC type instruction. This translation needs an instruction buffer and the fill level of this instruction buffer depends on the execution history. The length of this history that has to be considered for analysis is not bounded. Therefore this form of instruction translation is not exactly time predictable.

4.2.2 Compact Microcode

For the JVM to be implemented efficiently, the microcode has to *fit* to the Java bytecode. Since the JVM is a stack machine, the microcode is also stack-oriented. However, the JVM is not a pure stack machine. Method parameters and local variables are defined as *locals*. These locals can reside in a stack frame of the method and are accessed with an offset relative to the start of this *locals* area.

Additional local variables (16) are available at the microcode level. These variables serve as scratch variables, like registers in a conventional CPU. However, arithmetic and logic operations are performed on the stack.

Some bytecodes, such as ALU operations and the short form access to *locals*, are directly implemented by an equivalent microcode instruction (with a different encoding). Additional instructions are available to access internal registers, main memory and I/O devices. A relative conditional branch (zero/non zero of TOS) performs control flow decisions at the microcode level. For optimum use of the available memory resources, all instructions are 8 bits long. There are no variable-length instructions and every instruction, with the exception of wait, is executed in a single cycle. To keep the instruction set this dense, the following concept is applied: immediate values and branch offsets are addressed through one indirection. The instruction just contains an index for the constants.

Two types of operands, immediate values and branch distances, normally force an instruction set to be longer than 8 bits. The instruction set is either expanded to 16 or 32 bits, as in typical RISC processors, or allowed to be of variable length at byte boundaries. A first implementation of the JVM with a 16-bit instruction set showed that only a small number of different constants are necessary for immediate values and relative branch distances.

In the current realization of JOP, the different immediate values are collected while the microcode is being assembled and are put into the initialization file for the on-chip memory. These constants are accessed indirectly in the same way as the local variables. They are similar to initialized variables, apart from the fact that there are no operations to change their value during runtime, which would serve no purpose and would waste instruction codes. The microcode local variables, the microcode constants and the stack share the same on-chip memory. Using a single memory block simplifies the multiplexer in the execution stage.

A similar solution is used for branch distances. The assembler generates a VHDL file with a table for all found branch constants. This table is indexed using instruction bits during runtime. These indirections during runtime make it possible to retain an 8-bit instruction set, and provide 16 different immediate values and 32 different branch constants. For a general purpose instruction set, these indirections would impose too many restrictions. As the microcode only implements the JVM, this solution is a viable option.

To simplify the logic for instruction decoding, the instruction coding is carefully chosen. For example, one bit in the instruction specifies whether the instruction will increment or decrement the stack pointer. The offset to access the *locals* is directly encoded in the instruction. This is not the case for the original encoding of the equivalent bytecodes (e.g. *iload_0* is 0x1a and *iload_1* is 0x1b). Whenever a multiplexer depends on an instruction, the selection is directly encoded in the instruction.

4.2.3 Instruction Set

JOP implements 45 different microcode instructions. These instructions are encoded in 8 bits. With the addition of the *nxt* and *opd* bits in every instruction, the effective instruction length is 10 bits.

Bytecode equivalent: These instructions are direct implementations of bytecodes and result in one cycle execution time for the bytecode (except st and ld): pop, and, or, xor, add, sub, st<n>, st, ushr, shl, shr, nop, ld<n>, ld, dup

Local memory access: The first 16 words in the internal stack memory are reserved for internal variables. The next 16 words contain constants. These memory locations are accessed using the following instructions: stm, stmi, ldm, ldmi, ldi

Register manipulation: The stack pointer, the variable pointer and the Java program counter are loaded or stored with: stvp, stjpc, stsp, ldvp, ldjpc, ldsp, star

Bytecode operand: The operand is loaded from the bytecode RAM, converted to a 32-bit word and pushed on the stack with: ld_opd_8s, ld_opd_8u, ld_opd_16s, ld_opd_16u

External memory access: The autonomous memory subsystem and the IO subsystem are accessed by using the following instructions: stmra, stmwa, stmwd, wait, ldmrd, stbcrd, ldbcstart, stald, stast, stgf, stpf, stcp

Multiplier: The multiplier is accessed with: stmul, ldmul

Microcode branches: Two conditional branches in microcode are available: bz, bnz

Bytecode branch: All 17 bytecode branch instructions are mapped to one instruction: jbr

A detailed description of the microcode instructions can be found in Appendix C.

4.2.4 Bytecode Example

The example in Figure 4.1 shows the implementation of a single cycle bytecode and an infrequent bytecode as a sequence of JOP instructions. The suffix nxt marks the last instruction of the microcode sequence. In this example, the dup bytecode is mapped to the equivalent dup microcode and executed in a single cycle, whereas dup_x1 takes five cycles to execute, and after the last instruction (ldm a nxt), the first instruction for the next bytecode is executed. The scratch variables, as shown in the second example, are stored in the on-chip memory that is shared with the stack cache.

```
dup:    dup nxt     // 1 to 1 mapping

//  a and b are scratch variables for the
//  JVM code.
dup_x1: stm a       // save TOS
        stm b       // and TOS-1
        ldm a       // duplicate former TOS
        ldm b       // restore TOS-1
        ldm a nxt   // restore TOS and fetch next bytecode
```

Listing 4.1: Implementation of dup and dup_x1

Some bytecodes are followed by operands of between one and three bytes in length (except lookupswitch and tableswitch). Due to pipelining, the first operand byte that follows the bytecode instruction is available when the first microcode instruction enters the execution stage. If this is a one-byte long operand, it is ready to be accessed. The increment of

the Java program counter after the read of an operand byte is coded in the JOP instruction (an *opd* bit similar to the *nxt* bit).

In Listing 4.2, the implementation of sipush is shown. The bytecode is followed by a two-byte operand. Since the access to bytecode memory is only one[4] byte per cycle, *opd* and *nxt* are not allowed at the same time. This implies a minimum execution time of $n+1$ cycles for a bytecode with n operand bytes.

```
sipush: nop opd          // fetch next byte
        nop opd          // and one more
        ld_opd_16s nxt   // load 16 bit operand
```

Listing 4.2: Bytecode operand load

4.2.5 Flexible Implementation of Bytecodes

As mentioned above, some Java bytecodes are very complex. One solution already described is to emulate them through a sequence of microcode instructions. However, some of the more complex bytecodes are very seldom used. To further reduce the resource implications for JOP, in this case local memory, bytecodes can even be implemented by *using* Java bytecodes. That means bytecodes (e.g., new or floating point operations) can be implemented in Java. This feature also allows for the easy configuration of resource usage versus performance.

During the assembly of the JVM, all labels that represent an entry point for the bytecode implementation are used to generate the translation table. For all bytecodes for which no such label is found, i.e. there is no implementation in microcode, a *not-implemented* address is generated. The instruction sequence at this address invokes a static method from a system class. This class contains 256 static methods, one for each possible bytecode, ordered by the bytecode value. The bytecode is used as the index in the method table of this system class. A single empty static method consumes three 32-bit words in memory. Therefore, the overhead of this special class is 3 KB, which is 9% of a minimal *hello world* program (34 KB memory footprint). As described in Section 4.5, this feature also allows for the easy configuration of resource usage versus performance.

4.2.6 Summary

In order to handle the great variation in the complexity of Java bytecodes we have proposed a translation to a different instruction set, the so-called microcode. This microcode is still an

[4]The decision is to avoid buffers that would introduce time dependencies over bytecode boundaries.

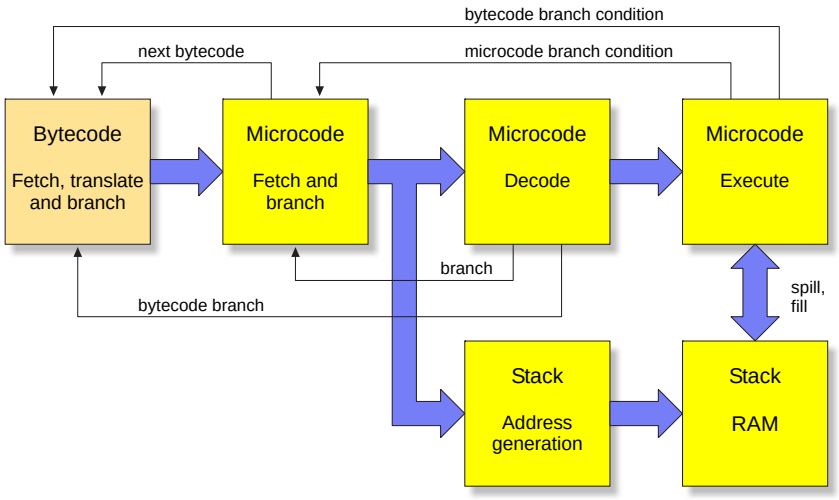

Figure 4.3: Datapath of JOP

instruction set for a stack machine, but more RISC-like than the CISC-like JVM bytecodes.

At the time of this writing 43 of the 201 different bytecodes are implemented by a single microcode instruction, 93 by a microcode sequence, and 40 bytecodes are implemented in Java. In the next section we will see how this translation is handled in JOP's pipeline and how it can simplify interrupt handling.

4.3 The Processor Pipeline

JOP is a fully pipelined architecture with single cycle execution of microcode instructions and a novel approach of translation from Java bytecode to these instructions. Figure 4.3 shows the datapath for JOP, representing the pipeline from left to right. Blocks arranged vertically belong to the same pipeline stage.

Three stages form the JOP core pipeline, executing microcode instructions. An additional stage in the front of the core pipeline fetches Java bytecodes – the instructions of the JVM – and translates these bytecodes into addresses in microcode. Bytecode branches are also decoded and executed in this stage. The second pipeline stage fetches JOP instructions from the internal microcode memory and executes microcode branches. Besides the usual decode function, the third pipeline stage also generates addresses for the stack RAM (the stack cache). As every stack machine microcode instruction (except nop, wait, and jbr) has

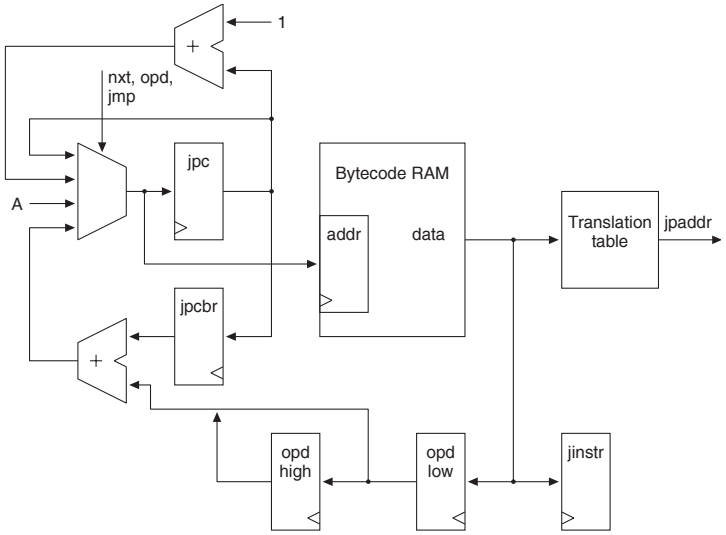

Figure 4.4: Java bytecode fetch

either *pop* or *push* characteristics, it is possible to generate fill or spill addresses for the *following* instruction at this stage. The last pipeline stage performs ALU operations, load, store and stack spill or fill. At the execution stage, operations are performed with the two topmost elements of the stack.

The stack architecture allows for a short pipeline, which results in short branch delays. Two branch delay slots are available after a conditional microcode branch. A stack machine with two explicit registers for the two topmost stack elements and automatic fill/spill to the stack cache needs neither an extra write-back stage nor any data forwarding. See Section 4.4 for a detailed description.

The method cache (*Bytecode Cache*), microcode ROM, and stack RAM are implemented with single cycle access in the FPGA's internal memories.

4.3.1 Java Bytecode Fetch

In the first pipeline stage, as shown in Figure 4.4, the Java bytecodes are fetched from the internal memory (*Bytecode RAM*). The bytecode is mapped through the translation table into the address (*jpaddr*) for the microcode ROM.

The fetched bytecode results in an absolute jump in the microcode (the second stage). If the bytecode is mapped one-to-one with a JOP instruction, the following fetched bytecode

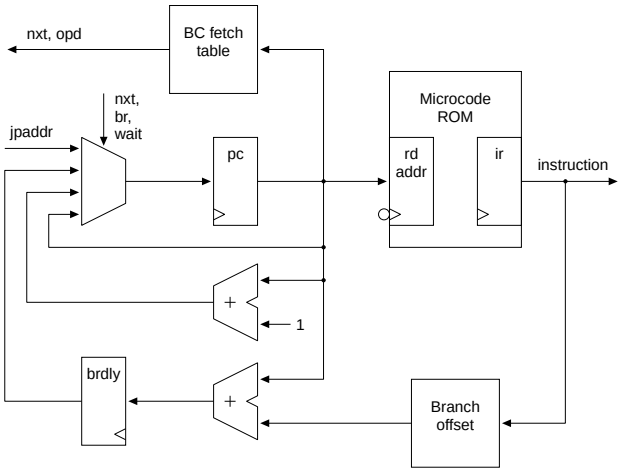

Figure 4.5: JOP instruction fetch

again results in a jump in the microcode in the following cycle. If the bytecode is a complex one, JOP continues to execute microcode. At the end of this instruction sequence, the next bytecode, and therefore the new jump address, is requested (signal *nxt*).

The bytecode RAM serves as the instruction cache and is filled on method invoke and return. Details about this time-predictable instruction cache can be found in Section 4.7.

The bytecode is also stored in a register for later use as an operand (requested by signal *opd*). Bytecode branches are also decoded and executed in this stage. Since *jpc* is also used to read the operands, the program counter is saved in *jpcbr* during an instruction fetch. *jinstr* is used to decode the branch type and *jpcbr* to calculate the branch target address.

4.3.2 JOP Instruction Fetch

The second pipeline stage, as shown in Figure 4.5, fetches JOP instructions from the internal microcode memory and executes microcode branches.

The JOP microcode, which implements the JVM, is stored in the microcode ROM. The program counter *pc* is incremented during normal execution. If the instruction is labeled with *nxt* a new bytecode is requested from the first stage and *pc* is loaded with *jpaddr*. *jpaddr* is the starting address for the implementation of that bytecode. The label *nxt* is the flag that marks the end of the microcode instruction stream for one bytecode. Another flag, *opd*, indicates that a bytecode operand needs to be fetched in the first pipeline stage. Both

flags are stored in a table that is indexed by the program counter.

brdly contains the target address for a conditional branch. The same offset is shared by a number of branch destinations. A table (*branch offset*) is used to store these relative offsets. This indirection means that only 5 bits need to be used in the instruction coding for branch targets and thereby allow greater offsets. The three tables *BC fetch table*, *branch offset* and *translation table* (from the bytecode fetch stage) are generated during the assembly of the JVM code. The outputs are plain VHDL files. For an implementation in an FPGA, recompiling the design after changing the JVM implementation is a straightforward operation. For an ASIC with a loadable JVM, it is necessary to implement a different solution.

FPGAs available to date do not allow asynchronous memory access. They therefore force us to use the registers in the memory blocks. However, the output of these registers is not accessible. To avoid having to create an additional pipeline stage just for a register-register move, the read address register of the microcode ROM is clocked on the negative edge.

An alternative solution for this problem would be to use the output of the multiplexer for the *pc* and the read address register of the memory. However, this solution results in a longer critical path, as the multiplexer can no longer be combined with the flip-flops that form the *pc* in the same LCs. This is an example of how implementation technology (the FPGA) can influence the architecture.

4.3.3 Decode and Address Generation

Besides the usual decode function, the third pipeline, as shown in Figure 4.6, also generates addresses for the stack RAM.

As we can see in Section 4.4 Table 4.5, read and write addresses are either relative to the stack pointer or to the variable pointer. The selection of the pre-calculated address can be performed in the decode stage. When an address relative to the stack pointer is used (either as read or as write address, never for both) the stack pointer is also decremented or incremented in the decode stage.

Stack machine instructions can be categorized from a stack manipulation perspective as either *pop* or *push*. This allows us to generate fill or spill TOS-1 addresses for the *following* instruction during the decode stage, thereby saving one extra pipeline stage.

4.3.4 Execute

At the execution stage, as shown in Figure 4.7, operations are performed using two discrete registers: TOS and TOS-1, labeled *A* and *B*.

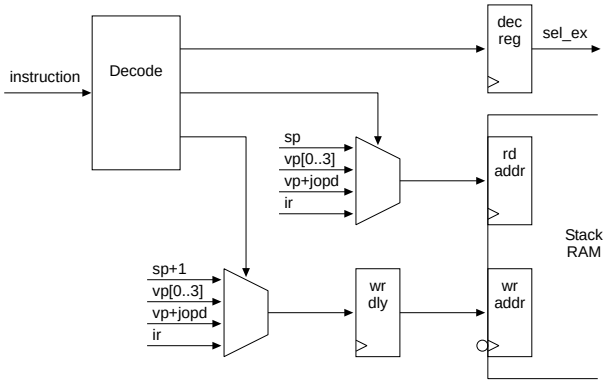

Figure 4.6: Decode and address generation

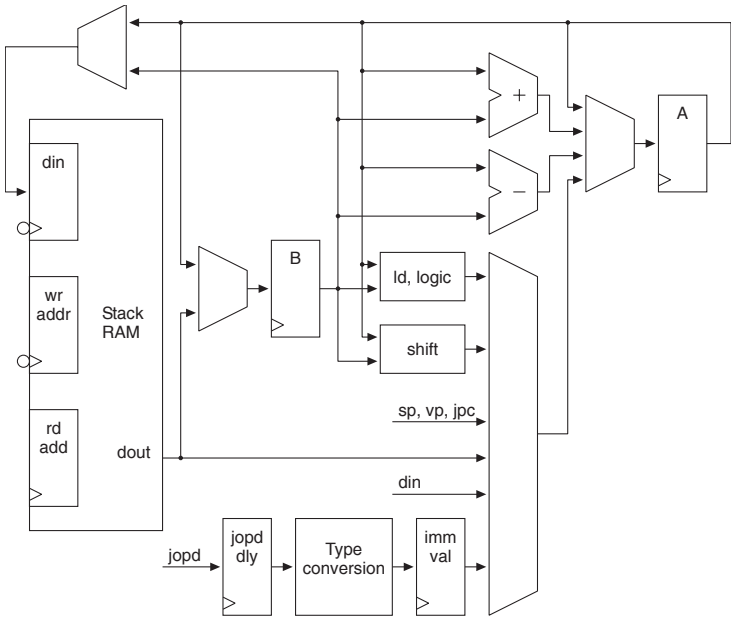

Figure 4.7: Execution stage

Each arithmetic/logical operation is performed with registers *A* and *B* as the source, and register *A* as the destination. All load operations (local variables, internal register, external memory and periphery) result in a value being loaded into register *A*. There is therefore no need for a write-back pipeline stage. Register *A* is also the source for the store operations. Register *B* is never accessed directly. It is read as an implicit operand or for stack spill on push instructions. It is written during the stack spill with the content of the stack RAM or the stack fill with the content of register *A*.

Beside the Java stack, the stack RAM also contains microcode variables and constants. This resource-sharing arrangement not only reduces the number of memory blocks needed for the processor, but also the number of data paths to and from the register *A*.

The inverted clock on data-in and on the write address register of the stack RAM is used, for the same reason, as on the read address register of the microcode ROM.

A stack machine with two explicit registers for the two topmost stack elements and automatic fill/spill needs neither an extra write-back stage nor any data forwarding. Details of this two-level stack architecture are described in Section 4.4.

4.3.5 Interrupt Logic

Interrupts and (precise) exceptions are considered hard to implement in a pipelined processor [49], meaning implementation tends to be complex (and therefore resource consuming). In JOP, the bytecode-microcode translation is used cleverly to avoid having to handle interrupts and exceptions (e.g., stack overflow) in the core pipeline.

Interrupts are implemented as special bytecodes. These bytecodes are inserted by the hardware in the Java instruction stream. When an interrupt is pending and the next fetched byte from the bytecode cache is an instruction (as indicated by the *nxt* bit in the microcode), the associated special bytecode is used instead of the instruction from the bytecode cache. The result is that interrupts are accepted at bytecode boundaries. The worst-case preemption delay is the execution time of the *slowest* bytecode that is implemented in microcode. Bytecodes that are implemented in Java (see Section 4.2.5) can be interrupted.

The implementation of interrupts at the bytecode-microcode mapping stage keeps interrupts transparent in the core pipeline and avoids complex logic. Interrupt handlers can be implemented in the same way as standard bytecodes are implemented i.e. in microcode or Java.

This special bytecode can result in a call of a JVM internal method in the context of the interrupted thread. This mechanism implicitly stores almost the complete context of the current active thread on the stack. This feature is used to implement the preemptive, fixed priority real-time scheduler in Java [103].

The main source for an interrupt is the μs accurate timer interrupt used by the real-time scheduler. Hardware generated exceptions, such as stack overflow or array bounds checks, generate a system interrupt. The exception reason can be found in a register.

4.3.6 Summary

In this section, we have analyzed JOP's pipeline. The core of the stack machine constitutes a three-stage pipeline. In the following section, we will see that this organization is an optimal solution for the stack access pattern of the JVM.

An additional pipeline stage in front of this core pipeline stage performs bytecode fetch and the translation to microcode. This organization has zero overheads for more complex bytecodes and results in the short pipeline that is necessary for any processor without branch prediction. This additional translation stage also presents an elegant way of incorporating interrupts virtually *for free*.

4.4 An Efficient Stack Machine

The concept of a stack has a long tradition, but stack machines no longer form part of mainstream computers. Although stacks are no longer used for expression evaluation, they are still used for the context save on a function call. A niche language, Forth [60], is stack-based and known as an efficient language for controller applications. Some hardware implementations of the Forth abstract machine do exist. These Forth processors are stack machines.

The Java programming language defines not only the language but also a binary representation of the program and an abstract machine, the JVM, to execute this binary. The JVM is similar to the Forth abstract machine in that it is also a stack machine. However, the usage of the stack differs from Forth in such a way that a Forth processor is not an ideal hardware platform to execute Java programs.

In this section, the stack usage in the JVM is analyzed. We will see that, besides the access to the top elements of the stack, an additional access path to an arbitrary element of the stack is necessary for an efficient implementation of the JVM. Two architectures will be presented for this mixed access mode of the stack. Both architectures are used in Java processors. However, we will also show that the JVM does not need a full three-port access to the stack as implemented in the two architectures. This allows for a simple and more elegant design of the stack for a Java processor. This proposed architecture will then be compared with the other two at the end of this section.

4.4.1 Java Computing Model

The JVM is not a pure stack machine in the sense of, for instance, the stack model in Forth. The JVM operates on a LIFO stack as its *operand stack*. The JVM supplies instructions to load values on the operand stack, and other instructions take their operands from the stack, operate on them and push the result back onto the stack. For example, the iadd instruction pops two values from the stack and pushes the result back onto the stack. These instructions are the stack machine's typical zero-address instructions. The maximum depth of this operand stack is known at compile time. In typical Java programs, the maximum depth is very small. To illustrate the operation notation of the JVM, Table 4.1 shows the evaluation of an expression for a stack machine notation and the JVM bytecodes. Instruction iload_n loads an integer value from a local variable at position *n* and pushes the value on TOS.

The JVM contains another memory area for method local data. This area is known as *local variables*. Primitive type values, such as integer and float, and references to objects are stored in these local variables. Arrays and objects cannot be allocated in a local variable, as

$A = B + C * D$	
Stack	JVM
push B	iload_1
push C	iload_2
push D	iload_3
*	imul
+	iadd
pop A	istore_0

Table 4.1: Standard stack notation and the corresponding JVM instructions

in C/C++. They have to be placed on the heap. Different instructions transfer data between the operand stack and the local variables. Access to the first four elements is optimized with dedicated single byte instructions, while up to 256 local variables are accessed with a two-byte instruction and, with the wide modifier, the area can contain up to 65536 values.

These local variables are very similar to registers and it appears that some of these locals can be mapped to the registers of a general purpose CPU or implemented as registers in a Java processor. On method invocation, local variables could be saved in a frame on a stack, different from the operand stack, together with the return address, in much the same way as in C on a typical processor. This would result in the following memory hierarchy:

- On-chip hardware stack for ALU operations
- A small register file for frequently-accessed variables
- A method stack in main memory containing the return address and additional local variables

However, the semantics of method invocation suggest a different model. The arguments of a method are pushed on the operand stack. In the invoked method, these arguments are not on the operand stack but are instead accessed as the first variables in the local variable area. The *real* method local variables are placed at higher indices. Listing 4.3 gives an example of the argument passing mechanism in the JVM. These arguments could be copied to the local variable area of the invoked method. To avoid this memory transfer, the entire variable area (the arguments *and* the variables of the method) is allocated on the operand stack. However, in the invoked method, the arguments are buried deep in the stack.

```
The Java source:

    int val = foo(1, 2);
    ...
    public int foo(int a, int b) {
        int c = 1;
        return a+b+c;
    }

Compiled bytecode instructions for the JVM:

The invocation sequence:
    aload_0             // Push the object reference
    iconst_1            // and the parameter onto the
    iconst_2            // operand stack.
    invokevirtual   #2  // Invoke method foo:(II)I.
    istore_1            // Store the result in val.

public int foo(int,int):
    iconst_1            // The constant is stored in a method
    istore_3            // local variable (at position 3).
    iload_1             // Arguments are accessed as locals
    iload_2             // and pushed onto the operand stack.
    iadd                // Operation on the operand stack.
    iload_3             // Push c onto the operand stack.
    iadd
    ireturn             // Return value is on top of stack.
```

Listing 4.3: Example of parameter passing and access

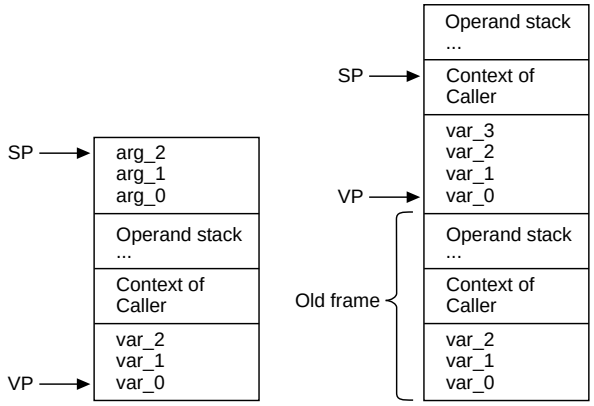

Figure 4.8: Stack change on method invocation

This asymmetry in the argument handling prohibits passing down parameters through multiple levels of subroutine calls, as in Forth. Therefore, an extra stack for return addresses is of no use for the JVM. This single stack now contains the following items in a frame per method:

- The local variable area
- Saved context of the caller
- The operand stack

A possible implementation of this layout is shown in Figure 4.8. A method with two arguments, `arg_1` and `arg_2` (`arg_0` is the *this* pointer), is invoked in this example. The invoked method *sees* the arguments as `var_1` and `var_2`. `var_3` is the only local variable of the method. SP is a pointer to the top of the stack and VP points to the start of the variable area.

4.4.2 Access Patterns on the Java Stack

The pipelined architecture of a Java processor executes basic instructions in a single cycle. A stack that contains the operand stack *and* the local variables results in the following access patterns:

Stack Operation: Read of the two top elements, operate on them and push back the result on the top of the stack. The pipeline stages for this operation are:

```
value1 ← stack[sp], value2 ← stack[sp-1]
result ← value1 op value2, sp ← sp-1
stack[sp] ← result
```

Variable Load: Read a data element deeper down in the stack, relative to a variable base address pointer (VP), and push this data on the top of the stack. This operation needs two pipeline stages:

```
value ← stack[vp+offset], sp ← sp+1
stack[sp] ← value
```

Variable Store: Pop the top element of the stack and write it in the variable relative to the variable base address:

```
value ← stack[sp]
stack[vp+offset] ← value, sp ← sp-1
```

For pipelined execution of these operations, a three-port memory or register file (two read ports and one write port) is necessary.

4.4.3 Common Realizations of a Stack Cache

As the stack is a heavily accessed memory region, the stack – or part of it – has to be placed in the upper level of the memory hierarchy. This part of the stack is referred to as a *stack cache*. As described in [49], a typical memory hierarchy contains the following elements, with increasing access time and size:

- CPU register
- On-chip cache memory
- Off-chip cache memory
- Main memory
- Magnetic disk for virtual memory

For a stack cache, a register file is the solution with the shortest access time. However, in order to store more than a few elements in the cache, an on-chip memory realization can provide a larger cache. Both variants have been used and are described below.

The Register File as a Stack Cache

An example of a Java processor that uses a register file is Sun's picoJava [125]. It contains 64 registers, organized as a circular buffer. To compensate for this *small* stack cache, an automatic spill and fill circuit needs another read/write port to the register file. aJile's JEM-Core [**?**] is a direct-execution Java processor core that contains 24 registers. Only six of them are used to cache the top elements of the stack. With this small register count, local variables are not part of the cache. Ignite [88] (formerly known as PSC1000) is a stack processor, originally designed as a Forth processor and now promoted as a Java processor, and has an operand stack that contains 18 registers with automatic spill and fill.

A basic pipeline for a stack processor with a register file contains the following stages:

1. IF – instruction fetch
2. ID – instruction decode
3. EX – read register file and execute
4. WB – write result back to register file

With this pipeline structure, a single data-forwarding path between WB and EX is necessary. The ALU with the register file (with a size of 16, a common size for RISC processors) and the bypass unit are shown in Figure 4.9. In Table 4.3 the hardware resources of this type of stack cache are approximated, using the values given in Table 4.2 (a MUX not found in this table is assumed to use combinations of the basic types; e.g. two 8:1 and one 2:1 for a 16:1). An experimental evaluation of this architecture in an FPGA is described in Section 4.4.5.

Basic function	Gate count
D-Flip-Flop	5
2:1 MUX	3
4:1 MUX	5
8:1 MUX	9
SRAM Bit	1.5

Table 4.2: Simplified gate count for basic functions

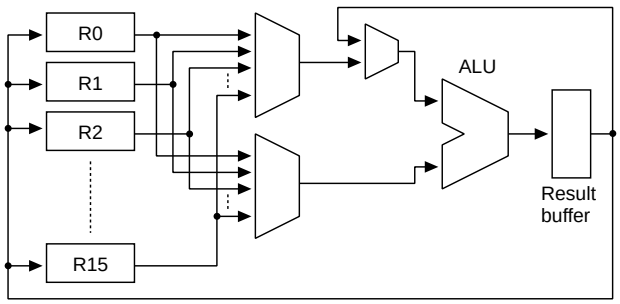

Figure 4.9: A stack cache with registers

Function block	Basic function	Gate count
Register File	512 D-Flip-Flops	2,560
Read MUX	2x32 16:1 MUX	1,344
Forward MUX	32 2:1 MUX	96
ALU buffer	32 D-Flip-Flops	160
Total		4,160

Table 4.3: Estimated gate count for a register stack cache

On-chip Memory as a Stack Cache

Using SRAM on the chip provides a *large* stack cache (e.g. 128 entries). However, as we have seen in Section 4.4.2, a three-port memory is necessary. An additional pipeline stage performs the cache memory read:

1. IF – instruction fetch
2. ID – instruction decode
3. RD – memory read
4. EX – execute
5. WB – write result back to memory

With this pipeline structure, two data forwarding paths are necessary. The resulting architecture is shown in Figure 4.10 and a gate count estimate is provided in Table 4.4. This version needs 70% more resources than the first one, but provides an eight times larger stack cache.

Example designs that use this kind of stack cache are (i) Komodo [137], a Java processor intended as a basis for research on multithreaded real-time scheduling, and (ii) FemtoJava [56], a research project to build an application specific Java processor.

A three-port memory is an expensive option for an ASIC and unusual in an FPGA. It can be emulated in an FPGA by two memories with a single read and write port. The write data is written in both memory blocks and each memory block provides a different read port. However, this solution also doubles the amount of memory.

Both designs (Komodo and FemtoJava) avoid the memory doubling by serializing the two reads. This serialization results in a minimum of two clock cycles execution time for basic instructions or halves the clock frequency of the whole pipeline.

4.4.4 A Two-Level Stack Cache

In this section, we will discuss access patterns of the JVM and their implication on the functional units of the pipeline. A faster and smaller architecture is proposed for the stack cache of a Java processor.

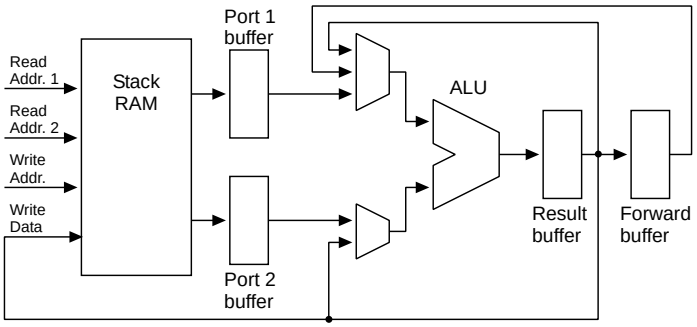

Figure 4.10: A stack cache with on-chip RAM

Function block	Basic function	Gate count
Stack RAM	e.g. 128x32 Bits	6,144
Port buffer	2x32 D-Flip-Flops	320
Forward MUX	32x 2:1 MUX, 3:1 MUX	288
ALU buffer	2x32 D-Flip-Flops	320
Total		7,072

Table 4.4: Estimated gate count for a stack cache with RAM

JVM Stack Access Revised

If we analyze the JVM's access patterns to the stack in more detail, we can see that a two-port read is only performed with the two top elements of the stack. All other operations with elements deeper in the stack, local variables load and store, only need one read port. If we only implement the two top elements of the stack in registers, we can use a standard on-chip RAM with one read and one write port.

We will show that all operations can be performed with this configuration. Let A be the top-of-stack, B the element below top-of-stack. The memory that serves as the second level cache is represented by the array *sm*. Two indices in this array are used: p points to the logical third element of the stack and changes as the stack grows or shrinks, v points to the base of the local variables area in the stack and n is the address offset of a variable. op is a two operand stack operation with a single result (i.e. a typical ALU operation).

Case 1: ALU operation

$A \leftarrow A\ op\ B$

$B \leftarrow sm[p]$

$p \leftarrow p - 1$

The two operands are provided by the two top level registers. A single read access from *sm* is necessary to fill B with a new value.

Case 2: Variable load (*Push*)

$sm[p+1] \leftarrow B$

$B \leftarrow A$

$A \leftarrow sm[v+n]$

$p \leftarrow p + 1$

One read access from *sm* is necessary for the variable read. The former TOS value moves down to B and the data previously in B is written to *sm*.

Case 3: Variable store (*Pop*)

$sm[v+n] \leftarrow A$

$A \leftarrow B$

$B \leftarrow sm[p]$

$p \leftarrow p - 1$

The TOS value is written to *sm*. A is filled with B and B is filled in an identical manner to Case 1, needing a single read access from *sm*.

We can see that all three basic operations can be performed with a stack memory with one read and one write port. Assuming a memory is used that can handle concurrent read and

write access, there is no structural access conflict between *A*, *B* and *sm*. That means that all operations can be performed concurrently in a single cycle.

As we can see in Figure 4.8 the operand stack and the local variables area are distinct regions of the stack. A concurrent read from and write to the stack is only performed on a variable load or store. When the read is from the local variables area the write goes to the operand area; a read from the operand area is concurrent with a write to the local variables area. Therefore there is no concurrent read and write to the same location in *sm*. There is no constraint on the read-during-write behavior of the memory (old data, undefined or new data), which simplifies the memory design. In a design where read and write-back are located in different pipeline stages, as in the architectures described above, either the memory must provide the new data on a read-during-write, or external forward logic is necessary.

From the three cases described, we can derive the memory addresses for the read and write port of the memory, as shown in Table 4.5.

Read address	Write address
p	p+1
v+n	v+n

Table 4.5: Stack memory addresses

The Datapath

The architecture of the two-level stack cache can be seen in Figure 4.11. Register *A* represents the top-of-stack and register *B* the data below the top-of-stack. ALU operations are performed with these two registers and the result is placed in *A*. During such an ALU operation, *B* is filled with new data from the stack RAM. A new value from the local variable area is loaded directly from the stack RAM into *A*. The data previously in *A* is moved to *B* and the data from *B* is spilled to the stack RAM. *A* is stored in the stack RAM on a store instruction to the local variable. The data from *B* is moved to *A* and *B* is filled with a new value from the stack RAM.

With this architecture, the pipeline can be reduced to three stages:

1. IF – instruction fetch
2. ID – instruction decode

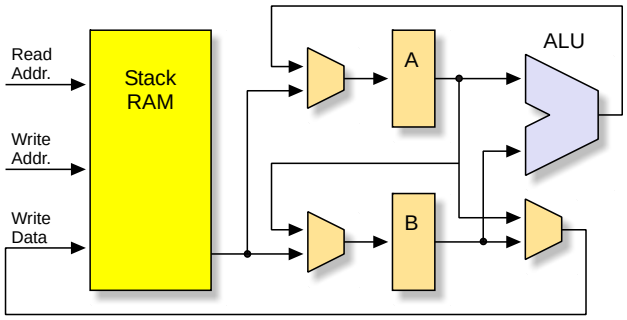

Figure 4.11: Two-level stack cache

Function block	Basic function	Gate count
Stack RAM	e. g. 128x32 Bits	6,144
TOS, TOS-1 buffer	2x32 D-Flip-Flops	320
Three MUX	3x32 2:1 MUX	288
Total		6,752

Table 4.6: Estimated gate count for a two-level stack cache

3. EX – execute, load or store

The estimated resource usage of this two-level stack cache architecture is given in Table 4.6. It can be seen that this architecture is roughly as complex as the solution given above (about 5% less gates). However, the reduced complexity with the two-port RAM instead of a three-port RAM is not included in the table. The critical path through the ALU contains only one 2:1 MUX to register *A* in this solution, rather than one 3:1 MUX in one ALU path and one 2:1 MUX in the other ALU path. As no data forwarding logic is necessary, the decoding logic is also simpler.

Data Forwarding – A Non-Issue

Data dependencies in the instruction stream result in the so-called *data hazards* [49] in the pipeline. Data forwarding is a technique that moves data from a later pipeline stage back to an earlier one to solve this problem. The term *forward* is correct in the temporal domain as data is transferred to an instruction in the future. However, it is misleading in the structural

domain as the forward direction is towards the *last* pipeline stage for an instruction.

As the probability of data dependency is very high in a stack-based architecture, one would expect several data forwarding paths to be necessary. However, in the two-level architecture proposed, with its resulting three-stage pipeline, no data hazards will occur and no data forwarding is therefore necessary. This simplifies the decoding stage and reduces the number of multiplexers in the execution path. We will show that none of the three data hazard types [49] is an issue in this architecture. With instructions i and j, where i is issued before j, the data hazard types are:

Read after write: j reads a source before i writes it. This is the most common type of hazard and, in the architectures described above, is solved by using the ALU buffers and the forwarding multiplexer in the ALU datapath. On a stack architecture, write takes three forms:

- Implicit write of TOS during an ALU operation
- Write to the TOS during a load instruction
- Write to an arbitrary entry of the stack with a store instruction

A read also occurs in three different forms:

- Read two top values from the stack for an ALU operation
- Read TOS for a store instruction
- Read an arbitrary entry of the stack with the load instruction

With the two top elements of the stack as discrete registers, these values are read, operated on and written back in the same cycle. No read that depends on TOS or TOS-1 suffers from a data hazard. Read and write access to a local variable is also performed in the same pipeline stage. Thus, the read after write order is not affected. However, there is also an additional hidden read and write: the fill and spill of register B:

- *B fill:* B is written during an ALU operation and on a variable store. During an ALU operation, the operands are the values from A and the old value from B. The new value for B is read from the stack memory and does not depend on the new value of A. During a variable store operation, A is written to the stack memory and does not depend on B. The new value for B is also read from the stack memory and it is not obvious that this value does not depend on the written value. However, the

variable area and the operand stack are distinct areas in the stack (this changes only on method invocation and return), guaranteeing that concurrent read/write access does not produce a data hazard.

- *B spill:* *B* is read on a load operation. The new value of *B* is the old value of *A* and does not therefore depend on the stack memory read. *B* is written to the stack. For the read value from the stack memory that goes to *A*, the argument concerning the distinct stack areas in the case of *B fill* described above still applies.

Write after read: *j* writes a destination before it is read by *i*. This cannot take place as all reads and writes are performed in the same pipeline stage keeping the instruction order.

Write after write: *j* writes an operand before it is written by *i*. This hazard is not present in this architecture as all writes are performed in the same pipeline stage.

4.4.5 Resource Usage Compared

The three architectures described above are implemented in Altera's EP1C6Q240C6 [3] FPGA. The three-port memory for the second solution is emulated with two embedded memory blocks. The ALU for this comparison is kept simple with the following functions: NOP, ADD, SUB, POP, AND, OR, XOR and load external data. The load of external data is necessary in order to prevent the synthesizer from optimizing away the whole design. A real implementation of an ALU for a Java processor, as described in Section 4.3, is a little bit more complex with a barrel shifter and additional load paths. In order to gain the maximum operating frequency for the design, the testbed for this architecture contains registers for the external data, the RAM address buses, and the control and select signals. Table 4.7 shows the resource usage and maximum operation frequency of the three different architectures.

LC stands for 'Logic Cell' and is the basic element in an FPGA: a 4-bit lookup table with a register. The LC count in the table includes the register count. The ALU alone without any stack cache needs 194 LCs. In the first line, the testbed is combined with the ALU without any stack caching, as a reference design. With this configuration, we can obtain the maximum possible speed of the registered ALU in this FPGA technology, in this case an operating frequency of 237 MHz or a 4.2 ns delay. This value is an upper bound of the system frequency. Every pipelined architecture needs one or more multiplexer in the ALU path, either for data forwarding or for operand selection, resulting in a longer delay. The fourth and fifth columns represent the resource usage of the cache logic without the testbed and ALU. The last column shows the effective cache size in data words.

Design	Total		Cache		Memory	fmax	Size
	LCs	Reg.	LCs	Reg.	(bit)	(MHz)	(word)
Testbed w. ALU	261	166	-	-	-	237	-
16 register cache	968	657	707	491	0	110	16
SRAM cache	372	185	111	19	8,192	153	128
Two-level cache	373	184	112	18	4,096	213	130

Table 4.7: Resource and performance compared

The version with the 16 registers was synthesized with two different synthesizer settings. In the first setting, the register file is implemented with discrete registers while, with a different setting, the register file is automatically implemented in two 32-bits embedded RAM blocks. Two different RAM blocks are necessary to provide two read ports and one write port. In both versions, the delay time to read the register file (delay through the 16:1 MUX of 4.9 ns or RAM access time of 4.6 ns) is in the same order as the delay time through the ALU, resulting in a system frequency of half the theoretical frequency of that with the ALU alone. As the structure of the version with the embedded RAM block is very similar with the SRAM cache, only the version with the discrete registers is shown in Table 4.7.

The stack cache with a RAM and registers on the RAM output (the additional pipeline stage) performs better than the first solution. However, the 3:1 MUX in the critical path still adds 2.3 ns to the delay time. Compared with the proposed solution (in the last line), we see that double the amount of RAM is needed for the two read ports.

The two-level stack cache solution performs at 213 MHz, i.e. almost the theoretical system frequency (in practice, about 10% slower). Only a 2:1 MUX is added to the critical path. The single read port memory needs half the number of memory bits of the other two solutions.

4.4.6 Summary

In this section, the stack architecture of the JVM was analyzed. We have seen that the JVM is different from the classical stack architecture. The JVM uses the stack both as an operand stack *and* as the storage place for local variables. Local variables are placed in the stack at a *deeper* position. To load and store these variables, an access path to an arbitrary position in the stack is necessary. As the stack is the most frequently accessed memory area in the JVM, caching of this memory is mandatory for a high-performing Java processor.

A common solution, found in a number of different Java processors, is to implement

this stack cache as a standard three-port register file with additional support to address this register file in a stack like manner. The architectures presented above differ in the realization of the register file: as a discrete register or in on-chip memory. Implementing the stack cache as discrete registers is very expensive. A three-port memory is also an expensive option for an ASIC and unusual in an FPGA. It can be emulated by two memories with a single read and write port. However, this solution also doubles the amount of memory.

Detailed analysis of the access patterns to the stack showed that only the two top elements of the stack are accessed in a single cycle. Given this fact, the proposed architecture uses registers to cache only the two top elements of the stack. The next level of the stack cache is provided by a simple on-chip memory. The memory automatically spills and fills the second register. Implementing the two top elements of the stack as fixed registers, instead of elements that are indexed by a stack pointer, also greatly simplifies the overall pipeline.

The proposed stack architecture has the following advantages: (i) Simpler cache memory results in having half the memory usage of the other solutions in an FPGA. (ii) Minimal impact on the raw speed of the ALU. Operates at almost the theoretical maximum system frequency of the ALU. (iii) Single read, execute and write-back pipeline stage results in an overall 3-stage pipeline processor design. (iv) No data forwarding is necessary, which simplifies instruction decode logic and reduces the multiplexer count in the critical path.

4.5 HW/SW Codesign

Using a hardware description language and loading the design in an FPGA the former strict border between hardware and software gets blurred. Is configuring an FPGA not more like loading a program for execution?

This looser distinction makes it possible to move functions easily between hardware and software resulting in a highly configurable design. If speed is an issue, more functions are realized in hardware. If cost is the primary concern these functions are moved to software and a smaller FPGA can be used. Let us examine these possibilities on a relatively expensive function: *multiplication*.

Bytecode imul performs a 32 bit signed multiplication with a 32 bit result. There are no exceptions on overflow. Since 32 bit single cycle multiplications are far beyond the possibilities of current, mainstream FPGAs the first solution is a sequential multiplier.

Sequential Booth Multiplier in VHDL Listing 4.4 shows the VHDL code of the multiplier. Two microcode instructions are used to access this function: stmul stores the two operands (from TOS and TOS-1) and starts the sequential multiplier. After 33 cycles, the result is loaded with ldmul. Listing 4.5 shows the microcode for imul.

Multiplication in Microcode If we run out of resources in the FPGA, we can move the function to microcode. The implementation of imul is almost identical to the Java code in Listing 4.6 and needs 73 microcode instructions.

Bytecode imul in Java Microcode is stored in an embedded memory block of the FPGA. This is also a resource of the FPGA. We can move the code to external memory by implementing imul in Java bytecode. Bytecodes not implemented in microcode result in a static Java method call from a special class (com.jopdesign.sys.JVM). This class has prototypes for each bytecode ordered by the bytecode value. This allows us to find the right method by indexing the method table with the value of the bytecode. Listing 4.6 shows the Java method for imul. The additional overhead for this implementation is a call and return with cache refills.

Implementations Compared Table 4.8 lists the resource usage and execution time for the three implementations. Execution time is measured with both operands negative, the worst-case execution time for the software implementations. The implementation in Java is

```
process(clk, wr_a, wr_b)

    variable count  : integer range 0 to width;
    variable pa     : signed(64) downto 0);
    variable a_1    : std_logic;
    alias p         : signed(32 downto 0)
                      is pa(64 downto 32);

begin
    if rising_edge(clk) then
        if wr_a='1' then
            p := (others => '0');
            pa(width-1 downto 0) := signed(din);

        elsif wr_b='1' then
            b <= din;
            a_1 := '0';
            count := width;
        else
            if count > 0 then
                case std_ulogic_vector'(pa(0), a_1) is
                    when "01" =>
                        p := p + signed(b);
                    when "10" =>
                        p := p - signed(b);
                    when others =>
                        null;
                end case;
                a_1 := pa(0);
                pa := shift_right(pa, 1);
                count := count - 1;
            end if;
        end if;
    end if;
    dout <= std_logic_vector(pa(31 downto 0));
end process;
```

Listing 4.4: Booth multiplier in VHDL

```
    imul:
            stmul       // store both operands and start
            pop         // pop second operand

            ldi 5       // 6*5+3 cycles wait
imul_loop:              // wait loop
            dup
            nop
            bnz imul_loop
            ldi -1      // decrement in branch slot
            add

            pop         // remove counter

            ldmul   nxt // load result
```

Listing 4.5: Microcode to access the Booth multiplier

slower than the microcode implementation as the Java method is loaded from main memory into the bytecode cache.

Only a few lines of code have to be changed to select one of the three implementations. This principle can also be applied to other expensive bytecodes: e.g. idiv, ishr, iushr and ishl. As a result, the resource usage of JOP is highly configurable and can be selected for each application according to the needs of the application. Treating VHDL as a software language allows easy movement of function blocks between hardware and software.

	Hardware [LC]	Microcode [Byte]	Time [Cycle]
VHDL	156	10	35
Microcode	0	73	750
Java	0	0	2,300

Table 4.8: Different implementations of imul compared

```
public static int imul(int a, int b) {

    int c, i;
    boolean neg = false;
    if (a<0) {
        neg = true;
        a = -a;
    }
    if (b<0) {
        neg = !neg;
        b = -b;
    }
    c = 0;
    for (i=0; i<32; ++i) {
        c <<= 1;
        if ((a & 0x80000000)!=0) c += b;
        a <<= 1;
    }
    if (neg) c = -c;
    return c;
}
```

Listing 4.6: Implementation of bytecode imul in Java

4.6 Real-Time Predictability

General-purpose processors are optimized for average throughput, and non real-time operating systems are responsible for fair and efficient scheduling of resources. Real-time systems need a processor with low and known WCET of instructions. Real-time operating systems have properties, such as fast interrupt response, rapid context switch, short blocking times and a scheduler that implements a simple, in most cases strictly priority driven, scheduling algorithm. This section describes design decisions for JOP to support such real-time systems.

4.6.1 Interrupts

Interrupts are usually associated with low-level programming of device drivers. The priorities of interrupts and their handler functions are above task priorities and yield to an immediate context switch. In this form, interrupts cannot be integrated in a schedule with *normal* tasks. The execution time of the interrupt handler has to be integrated in the schedulability analysis as additional blocking time. A better solution is to handle interrupts, which represent external events, as schedulable objects with priority levels in the range of real-time tasks, as suggested in the RTSJ.

The Timer Interrupt The timer or clock interrupt has a different semantic than other interrupts. The main purpose of the timer interrupt is representation of time and release of periodic or time triggered tasks. One common implementation is a clock tick. The interrupt occurs at a regular interval (e.g. 10 ms) and a decision has to be taken whether a task has to be released. This approach is simple to implement, but there are two major drawbacks: The resolution of timed events is bound by the resolution of the clock tick and clock ticks without a task switch are a waste of execution time.

A better approach, used in JOP, is to generate timer interrupts at the release times of the tasks. The scheduler is now responsible for reprogramming the timer after each occurrence of a timer interrupt. The list of sleeping threads has to be searched to find the nearest release time in the future of a higher priority thread than the one that will be released now. This time is used for the next timer interrupt.

External Events Hardware interrupts, other than the timer interrupt, are represented as asynchronous events with an associated thread. This means that the event is a *normal* schedulable object under the control of the scheduler. With a minimum interarrival time,

enforced by hardware, these events can be incorporated into the priority assignment and schedulability analysis in the same way as periodic tasks.

Software Interrupts The common software generated interrupts, such as illegal memory access or divide by zero, are represented by Java runtime exceptions and need no special handler. They can be detected with a try-catch block.

Asynchronous notification from the program is supported, in the same way as an external event, as a schedulable object with an associated thread. The event is triggered through the call of fire(). The thread with the handler is placed in the runnable state and scheduled according to priority.

Hardware Failures Serious hardware failures, such as illegal opcode or parity error from the memory systems, lead to a shutdown of the system. However, a *last try* to call a handler that changes the state of the system to a safe state and inform an upper level system, can improve the integrity of the overall system.

4.6.2 Task Switch

An important issue in real-time systems is the time for a task switch. A task switch consists of two actions:

- *Scheduling* is the selection of the task order and timing
- *Dispatching* is the term for the context switch between tasks

Scheduling Most real-time systems use a fixed-priority preemptive scheduler. Tasks with the same priority are usually scheduled in a FIFO order. Two common ways to assign priorities are rate monotonic or, in a more general form, deadline monotonic assignment. When two tasks get the same priority, we can choose one of them and assign a higher priority to that task and the task set is still schedulable. We get a strictly monotonic priority order and do not have to deal with FIFO order. This eliminates queues for each priority level and results in a single, priority ordered task list.

Strictly fixed priority schedulers suffer from a problem called *priority inversion* [**?**]. The problem where a low priority task blocks a high priority task on a shared resource is solved by raising the priority of the low priority task. Two standard priority inversion avoidance protocols are common:

Priority Inheritance Protocol: A lock assigns the priority of the highest-priority waiting task to the task holding the lock until that task releases the resource.

Priority Ceiling Emulation Protocol: A lock gets a priority assigned above the priority of the highest-priority task that will ever acquire the lock. Every task will be immediately assigned the priority of that lock when acquiring it.

The priority inheritance protocol is more complex to implement and the time when the priority of a task is raised is not so obvious. It is not raised because the task does anything, but because another task reaches some point in its execution path.

Using priority ceiling emulation with unique priorities, different from task priorities, the priority order is still strictly monotonic. The priority ordered task list is expanded with slots for each lock. If a task acquires a lock, it is placed in the corresponding slot. With this extension to the task list, scheduling is still simple and can be efficiently implemented.

Dispatching The time for a context switch depends on the size of the state of the tasks. For a stack machine it is not so obvious what belongs to the state of a task. If the stack resides in main memory, only a few registers (e.g. program counter and stack pointer) need to be saved and restored. However, the stack is a frequently accessed memory region of the JVM. The stack can be seen as a data cache and should be placed near the execution unit (in this case, *near* means on the chip and not in external memory). However, on-chip memory is usually too small to hold the stack content for all tasks. This means that the stack is part of the execution context and has to be saved and restored on a context switch.

In JOP, the stack is placed in local (on-chip) FPGA memory with single cycle access time. With this configuration, the next question is how much of the stack to place there. Either the complete stack of a thread or only the stack frame of the current method can reside locally. If the complete stack of a thread is stored in local memory, the invocation of methods and returns are fast, but the context is large. For fast context switches, it is preferable to have only a short stack in local memory. This results in less data being transferred to and from main memory, but more memory transfers on method invocation and return. The local stack can be further divided into small pieces, each holding only one stack frame of one thread. During the context switch, only the stack pointer needs to be saved and restored. The outcome of this is a very fast context switch, although the size of the local memory limits the maximum number of threads.

Since JOP is a soft-core processor, these different solutions can be configured for different application requirements. It is even possible to mix of these policies: some stack slots can be assigned to *important* threads, while the remaining threads share one slot. This stack

slot only needs to be exchanged with the main memory when switching *to* a less *important* thread.

4.6.3 Architectural Design Decisions

In hard real-time systems, meeting temporal requirements is of the same importance as functional correctness. This results in different architectural constraints than in a design for a non real-time system. A low upper bound of the execution time is of premium importance. Good average execution time is useless for a pure hard real-time system.

Common architectural components, found in general purpose processors to enhance average performance, are usually problematic for the WCET analysis. A pragmatic approach to this problem is to ignore these features for the analysis. With a processor designed for real-time applications, these features have to be substituted by predictable architecture enhancements.

Branch Prediction As the pipelines of current general-purpose processors get longer to support higher clock rates, the penalty of branches gets too high. This is compensated by branch prediction logic with branch target buffers. However, the upper bound of the branch execution time is the same as without this feature. In JOP, branch prediction is avoided. This results in pressure on the pipeline length. The core processor has a pipeline length of as little as three stages resulting in a branch execution time of three cycles in microcode. The two slots in the branch delay can be filled with instructions or *nop*. With the additional bytecode fetch and translation stage, the overall pipeline is four stages and results in a four cycle execution time for a bytecode branch.

Caches and Instruction Prefetch To reduce the growing gap between the clock frequency of the processor and memory access times multi-level cache architectures are commonly used. Since even a single level cache is problematic for WCET analysis, more levels in the memory architecture are almost not analyzable. The additional levels also increase the latency of memory access on a cache miss.

In a stack machine, the stack is a frequently accessed memory area. This makes the stack an ideal candidate to be placed near the execution unit in the memory hierarchy. In JOP the stack is implemented as internal memory with the two top elements as explicit registers. This single cycle memory can be seen as a data cache. However, unlike in picoJava, this limited memory is not automatically spilled and filled. Automatic spill and fill introduces unpredictable access to the main memory. Data exchange between internal stack and main

memory is under program control and can be done on method invocation/return or on a thread switch.

The next most accessed memory area is the code area. A simple prefetch queue, as it is found in older processors, could increase instruction throughput after executing a multi-cycle bytecode. For a stream of single cycle bytecodes, prefetching is useless and the frequent occurrence of branches and method invocations, about 12–23% (see Section **??**) in typical Java programs, reduces the performance gain. The prefetch queue also results in (probably unbounded) execution time dependencies over a stream of instructions, which complicates timing analysis.

JOP has a method cache with a novel replace policy. Since typical methods in Java programs are short and there are only relative branches in a method, a complete method is loaded in the cache on invocation and on return. This cache fill strategy lumps all cache misses together and is very simple to analyze. It also simplifies the hardware of the cache since no tag memory or address translation is necessary. The *romizer* tool JavaCodeCompact checks the maximum allowed method size. Section 4.7 describes the proposed cache solution in detail. Memory areas for the heap and class description with the constant pool are not cached in JOP.

Superscalar Processors A superscalar processor consists of several execution units and tries to extract instruction level parallelism (ILP) with out of order execution. Again, this is a nightmare for timing analysis. The code for a stack machine has less implicit parallelism than a register machine.

One form of enhancement, usually implemented in stack machines, is instruction folding. The instruction stream is scanned to find frequent patterns like load-load-add-store and substitutes these four instructions with one, RISC-like, operation. There are two issues with instruction folding in JOP: The combined instruction needs two read and one write access to the stack in a single cycle. This would result in doubling of the internal memory usage in the FPGA. It also needs, at minimum, four bytes read access to the method cache. To overcome word boundaries, prefetching has to be introduced after the method cache. This results in an additional pipeline stage, time dependency of instructions with a more complex analysis and more hardware resources for the multiplexers.

Programs for embedded and real-time systems are usually multi-threaded. In future work, it will be investigated if the additional hardware resources needed for ILP can be better used with additional processor cores utilizing this implicit thread-level parallelism.

Time-Predictable Instructions A good model of a processor with accurate timing information is essential for a tight WCET analysis. The architecture of JOP and the microcode are designed with this in mind. Execution time of bytecodes is known cycle accurately (see Chapter 7 and Appendix D). It is possible to analyze the WCET on the bytecode level [20] without the uncertainties of an interpreting JVM [18] or generated native code from ahead-of-time compilers for Java.

4.6.4 Summary

In this section, we argued that, while common techniques in processor architectures increase average throughput, they are not feasible for real-time systems. The influence of these architectural enhancements is at best hardly WCET-analyzable.

The proposed alternatives influence the processor architecture, as described in earlier sections, as well as the software architecture that will be described in Section 5.3.

However, the most important architectural enhancement for pipelined machines is caching, which is necessary even in embedded systems. We have shown in Section 4.4 how a time-predictable data cache for a stack machine can be implemented. In the following section, we will propose a time-predictable cache for instructions.

4.7 A Time-Predictable Instruction Cache

Worst-case execution time (WCET) analysis [91] of real-time programs is essential for any schedulability analysis. To provide a low WCET value, a good processor model is necessary. However, the architectural advancement in modern processor designs is dominated by the rule: '*Make the common case fast*'. This is the opposite of '*Reduce the worst case*' and complicates WCET analysis.

Cache memory for the instructions and data is a classic example of this paradigm. Avoiding or ignoring this feature in real-time systems, due to its unpredictable behavior, results in a very pessimistic WCET value. Plenty of effort has gone into research into integrating the instruction cache in the timing analysis of tasks [11, 47, 67] and the influence of the cache on task preemption [65, **?**]. The influence of different cache architectures on WCET analysis is described in [48].

We will tackle this problem from the architectural side – an instruction cache organization in which simpler and more accurate WCET analysis is more important than average case performance.

In this section, we will propose a method cache with a novel replacement policy. In Java bytecode only relative branches exist, and a method is therefore only left when a return instruction has been executed[5]. It has been observed that methods are typically short (see Section **??**) in Java applications. These properties are utilized by a cache architecture that stores complete methods. A complete method is loaded into the cache on both invocation and return. This cache fill strategy lumps all cache misses together and is very simple to analyze.

4.7.1 Cache Performance

In real-time systems we prefer time-predictable architectures over those with a high average performance. However, performance is still important. In this section, we will give a short overview of the formulas from [49] that are used to calculate the cache's influence on execution time. We will extend the single measurement *miss rate* to a two value set, memory read and transaction rate, that is architecture independent and better reflects the two properties (bandwidth and latency) of the main memory. To evaluate cache performance, MEM_{clk} memory stall cycles are added to the CPU execution time (t_{exe}) equation:

$$t_{exe} = (CPU_{clk} + MEM_{clk}) \times t_{clk}$$
$$MEM_{clk} = Misses \times MP_{clk}$$

[5] An uncaught exception also results in a method exit.

The miss penalty MP_{clk} is the cost per miss, measured in clock cycles. When the instruction count IC is given as the number of instructions executed, CPI the average clock cycles per instruction and the number of misses per instruction, we obtain the following result:

$$CPU_{clk} = IC \times CPI_{exe}$$
$$MEM_{clk} = IC \times \frac{Misses}{Instruction} \times MP_{clk}$$
$$t_{exe} = IC \times (CPI_{exe} + \frac{Misses}{Instruction} \times MP_{clk}) \times t_{clk}$$

As this section is only concerned with the instruction cache, we will split the memory stall cycles into misses caused by the instruction fetch and misses caused by data access.

$$CPI = CPI_{exe} + CPI_{IM} + CPI_{DM}$$

CPI_{exe} is the average number of clock cycles per instruction, given an ideal memory system without any stalls. CPI_{IM} are the additional clock cycles caused by instruction cache misses and CPI_{DM} the data miss portion of the CPI. This split between instruction and data portions of the CPI better reflects the split of the cache between instruction and data cache found in actual processors. The misses per instruction are often given as misses per 1000 instructions. However, there are several drawbacks to using a single number:

Architecture dependent: The average number of memory accesses per instruction differs greatly between a RISC processor and the Java Virtual Machine (JVM). A typical RISC processor needs one memory word (4 bytes) per instruction word, and about 40% of the instructions [49] are *load* or *store* instructions. Using the example of a 32-bit RISC processor, this results in 5.6 bytes memory access per instruction. The average length of a JVM bytecode instruction is 1.7 bytes and about 18% of the instructions access the memory for data load and store.

Block size dependent: Misses per instruction depends subtly on the block size. On a single cache miss, a whole block of the cache is filled. Therefore, the probability that a future instruction request is a hit is higher with a larger block size. However, a larger block size results in a higher miss penalty as more memory is transferred.

Main memory is usually composed of DRAMs. Access time to this memory is measured in terms of latency (the time taken to access the first word of a larger block) and bandwidth (the number of bytes read or written in a single request per time unit). These two values, along with the block size of a cache, are used to calculate the miss penalty:

$$MP_{clk} = Latency + \frac{Block\ size}{Bandwidth}$$

To better evaluate different cache organizations and different instruction sets (RISC versus JVM), we will introduce two performance measurements – memory bytes read per instruction byte and memory transactions per instruction byte:

$$MBIB = \frac{Memory\ bytes\ read}{Instruction\ bytes}$$
$$MTIB = \frac{Memory\ transactions}{Instruction\ bytes}$$

These two measures are closely related to memory bandwidth and latency. With these two values and the properties of the main memory, we can calculate the average memory cycles per instruction byte $MCIB$ and CPI_{IM}, i.e. the values we are concerned in this section.

$$MCIB = (\frac{MBIB}{Bandwith} + MTIB \times Latency)$$
$$CPI_{IM} = MCIB \times Instruction\ length$$

The misses per instruction can be converted to MBIB and MTIB when the following parameters are known: the average instruction length of the architecture, the block size of the cache and the miss penalty in latency and bandwidth. We will examine this further in the following example:

We use the following architecture to illustrate the conversion: a RISC architecture with a 4 bytes instruction length, an 8KB instruction cache with 64-byte blocks and a miss rate of 8.16 per 1000 instructions [49]. The miss penalty is 100 clock cycles. The memory system is assumed to deliver one word (4 bytes) per cycle.

Firstly, we need to calculate the latency of the memory system.

$$\begin{aligned} Latency &= MP_{clk} - \frac{Blocksize}{Bandwidth} \\ &= 100 - \frac{64}{4} = 84 \text{ clock cycles} \end{aligned}$$

With $Miss\ rate = \frac{Cache\ miss}{Cache\ access}$, we obtain MBIB.

$$\begin{aligned} MBIB &= \frac{Memory\ bytes\ read}{Instruction\ bytes} \\ &= \frac{Cache\ miss \times Block\ size}{Cache\ access \times Instruction\ length} \\ &= Miss\ rate \times \frac{Block\ size}{Instruction\ length} \\ &= 8.16 \times 10^{-3} \times \frac{65}{4} \\ &= 0.131 \end{aligned}$$

MTIB is calculated in a similar way:

$$\begin{aligned} MTIB &= \frac{Memory\ transactions}{Instruction\ bytes} \\ &= \frac{Cache\ miss}{Cache\ access \times Instruction\ length} \\ &= \frac{Miss\ rate}{Instruction\ length} \\ &= \frac{8.16 \times 10^{-3}}{4} \\ &= 2.04 \times 10^{-3} \end{aligned}$$

For a quick check, we can calculate CPI_{IM}:

$$\begin{aligned} MCIB &= \frac{MBIB}{Bandwith} + MTIB \times Latency \\ &= \frac{0.131}{4} + 2.04 \times 10^{-3} \times 84 \\ &= 0.204 \\ CPI_{IM} &= MCIB \times Instruction\ length \\ &= 0.204 \times 4 \\ &= 0.816 \end{aligned}$$

This is the same value as that which we get from using the miss rate with the miss penalty:

$$\begin{aligned} CPI_{IM} &= Miss\ rate \times Miss\ penalty \\ &= 8.16 \times 10^{-3} \times 100 \\ &= 0.816 \end{aligned}$$

However, MBIB and MTIB are architecture independent and better reflect the latency and bandwidth of the main memory.

4.7.2 Proposed Cache Solution

In this section, we will develop a solution for a predictable cache. Typical Java programs consist of short methods. There are no branches out of the method and all branches inside are relative. In the proposed architecture, the full code of a method is loaded into the cache before execution. The cache is filled on invocations and returns. This means that all cache fills are lumped together with a known execution time. The full loaded method and relative addressing inside a method also result in a simpler cache. Tag memory and address translation are not necessary.

However, we will first discuss an even simpler solution – no caching at all. Without an instruction cache, prefetching is mandatory, especially with a variable length instruction set. The issues surrounding prefetching are discussed in the next section.

Instruction Prefetching

A simple prefetch queue, as found in older processors, can increase instruction throughput after a multi-cycle bytecode is executed. However, for a stream of single-cycle bytecodes, prefetching is useless and the frequent occurrence of branches, method invocations, and method returns (see Section **??**) reduces the performance gain. Using a prefetch queue also results in execution time dependencies over a stream of instructions, which complicates timing analysis.

For a variable length instruction set, prefetching is also not a straightforward option. The prefetching unit needs to guarantee the availability of a complete instruction for the fetch unit. As the actual length of the instruction is not known at this stage, the prefetch unit must be a minimum of $maximum\ length - 1$ bytes ahead of the requested instruction. This can lead to unnecessary memory transfers. The return instruction is a typical example of this. It is 1 byte long and the additional prefetched instruction bytes are never used.

A memory interface with a bus width greater than one byte adds an artificial boundary to the instruction stream. For the purpose of this example, we are assuming a 4 byte memory interface. In this case we need an 8 byte prefetch buffer. On a branch to an address $address\ mod\ 4 > 4 - maximum\ instruction\ length$, two words need to be loaded from main memory before the processor can continue.

A memory technology, such as synchronous DRAM, has a large latency for the first accessed word and then a high bandwidth for the following words. Prefetching that only

loads small quantities (one or two words) from the memory is therefore impracticable with these memory technologies.

Single Method Cache

A single method cache, although less efficient than a conventional instruction cache, can be incorporated very easily into the WCET analysis. The time needed for the memory transfer must be added to the invoke and return instructions.

The method cache also simplifies the hardware of the cache, as it means that no tag memory or address translation is necessary. Other parts of the processor are also smaller. The program counter, the associated adders and multiplexer are shorter than in a standard cache solution. For example, for a 1KB cache, the size of these units is only 10 bits, instead of 32 bits.

The main disadvantage of this single method cache is the high overhead when a complete method is loaded into the cache and only a small fraction of the code is executed. This issue is similar to that encountered with unused data in a cache line. However, in extreme cases, this overhead can be very high. The second problem can be seen in the following example:

```
foo() {
    a();
    b();
}
```

This code sequence results in the following cache loads:

1. method foo is loaded on invocation of foo()
2. method a is loaded on invocation of a()
3. method foo is loaded on return from a()
4. method b is loaded on invocation of b()
5. method foo is loaded on return from b()

The main drawback of the single method cache is the multiple cache fill of foo() on return from methods a() and b(). In a conventional cache design, if these three methods fit in the cache memory at the same time and there is no placement conflict, each method is only loaded once. This issue can be overcome by caching more than one method. The simplest solution is a two-block cache.

Two-Block Cache

The two-block cache can hold up to two methods in the cache. This results in having to decide which block is replaced on a cache miss. With only two blocks, Least-Recently Used (LRU) is trivial to implement. The code sequence now results in the cache loads and hits as shown in Table 4.9. With the two-block cache, we have to double the cache memory

Instruction	Block 1	Block 2	Cache
foo()	foo	–	load
a()	foo	a	load
return	foo	a	hit
b()	foo	b	load
return	foo	b	hit

Table 4.9: Cache load and hit example with the two-block cache

or use both blocks for a single large method. The WCET analysis is slightly more complex than with a single block. A short history of the invocation sequence has to be used to find the cache fills and hits.

However, a cache that can only hold two methods is still very restrictive. The next code sequence shows the conflict. Table 4.10 shows the resulting cache loads.

```
foo() {
    a();
}
a() {
    b();
}
```

A memory (similar to the tag memory) with one word per block is used to store a reference to the cached method. However, this memory can be slower than the tag memory as it is only accessed on invocation or return, rather than on every cache access.

More Blocks

We can improve the hit rate by adding more blocks to the cache. If only one block per method is used, the cache size increases with the number of blocks. With more than two

Instruction	Block 1	Block 2	Cache
foo()	foo	–	load
a()	foo	a	load
b()	b	a	load
return	b	a	hit
return	foo	a	load

Table 4.10: Cache conflict example with the two-block cache

blocks, LRU replacement policy means that another word is needed for every block containing a use counter that is updated on every invoke and return. During replacement, this list is searched for the LRU block. Hit detection involves a search through the list of the method references of the blocks. If this search is done in microcode, it imposes a limit on the maximum number of blocks.

Variable Block Cache

Several cache blocks, all of the size as the largest method, are a waste of cache memory. Using smaller block sizes and allowing a method to span over several blocks, the blocks become very similar to cache lines. The main difference from a conventional cache is that the blocks for a method are all loaded at once and need to be consecutive.

Choosing the block size is now a major design decision. Smaller block sizes allow better memory usage, but the search time for a hit also increases.

With varying block numbers per method, an LRU replacement becomes impractical. When the method found to be LRU is smaller than the loaded method, this new method invalidates two cached methods.

For the replacement, we will use a pointer *next* that indicates the start of the blocks to be replaced on a cache miss. Two practical replace policies are:

Next block: At the very first beginning, *next* points to the first block. When a method of length l is loaded into the block n, *next* is updated to $(n+l)$ *mod block count*.

Stack oriented: *next* is updated in the same way as before on a method load. It is also updated on a method return – independent of a resulting hit or miss – to point to the first block of the leaving method.

We will show the operation of these different replacement policies in an example with three methods: a(), b() and c() of block sizes 2, 2 and 1. The cache consists of 4 blocks and is

```
a() {
    for (;;) {
        b();
        c();
    }
    ...
}
```

Listing 4.7: Code fragment for the replacement example

	a()	b()	ret	c()	ret	b()	ret	c()	ret	b()	ret
Block 1	A	→a	→a	C	c	B	b	b	→-	B	b
Block 2	A	a	a	→-	A	→a	→a	C	c	B	b
Block 3	→-	B	b	b	A	a	a	→-	A	→a	→a
Block 4	-	B	b	b	→-	B	b	b	A	a	a
Fill	2	4		5	7	9		11	13	15	

Table 4.11: Next block replacement policy

therefore too small to hold all the methods during the execution of the code fragment shown in Listing 4.7. Tables 4.11 and 4.12 show the cache content during program execution for both replacement policies. The content of the cache blocks is shown after the execution of the invoke or return instruction. An uppercase letter indicates that this block has been newly loaded. A right arrow depicts the block to be replaced on a cache miss (the *next* pointer). The last row shows the number of blocks that are filled during the execution of the program.

In this example, the stack oriented approach needs fewer fills, as only methods b() and c() are exchanged and method a() stays in the cache. However, if, for example, method b() is the size of one block, all methods can be held in the cache using the the *next block* policy, but b() and c() would be still exchanged using the *stack* policy. Therefore, the first approach is used in the proposed cache.

4.7.3 WCET Analysis

The proposed instruction cache is designed to simplify WCET analysis. Due to the fact that all cache misses are only included in two instructions (*invoke* and *return*), the instruction

	a()	b()	ret	c()	ret	b()	ret	c()	ret	b()	ret
Block 1	A	→a	a	a	a	→a	a	a	a	→a	a
Block 2	A	a	a	a	a	a	a	a	a	a	a
Block 3	→-	B	→b	C	→c	B	→b	C	→c	B	→b
Block 4	-	B	b	→-	-	B	b	→-	-	B	b
Fill	2	4		5		7		8		10	

Table 4.12: Stack oriented replacement policy

cache can be ignored on all other instructions. The time needed to load a complete method is calculated using the memory properties (latency and bandwidth) and the length of the method. On an invoke, the length of the invoked method is used, and on a return, the method length of the caller is used to calculate the load time.

With a single method cache this calculation can be further simplified. For every invoke there is a corresponding return. That means that the time needed for the cache load on return can be included in the time for the invoke instruction. This is simpler because both methods, the caller and the callee, are known at the occurrence of the invoke instruction. The information about which method was the caller need not be stored for the return instruction to be analyzed.

With more than one method in the cache, a cache hit detection has to be performed as part of the WCET analysis. If there are only two blocks, this is trivial, as (i) a hit on invoke is only possible if the method is the same as the last invoked (e.g. a single method in a loop) and (ii) a hit on return is only possible when the method is a leaf in the call tree. In the latter case, it is always a hit.

When the cache contains more blocks (i.e. more than two methods can be cached), a part of the call tree has to be taken into account for hit detection. The variable block cache further complicates the analysis, as the method length also determines the cache content. However, this analysis is still simpler than a cache modeling of a direct-mapped instruction cache, as cache block replacement depends on the call tree instead of instruction addresses.

In traditional caches, data access and instruction cache fill requests can compete for the main memory bus. For example, a load or store at the end of the processor pipeline competes with an instruction fetch that results in a cache miss. One of the two instructions is stalled for additional cycles by the other instruction. With a data cache, this situation can be even worse. The worst-case scenario for the memory stall time for an instruction fetch or a data load is two miss penalties when both cache reads are a miss. This unpredictable

behavior leads to very pessimistic WCET bounds.

A *method cache*, with cache fills only on invoke and return, does not interfere with data access to the main memory. Data in the main memory is accessed with *getfield* and *putfield*, instructions that never overlap with *invoke* and *return*. This property removes another uncertainty found in traditional cache designs.

4.7.4 Caches Compared

In this section, we will compare the different cache architectures in a quantitative way. Although our primary concern is predictability, performance remains important. We will therefore first present the results from a conventional direct-mapped instruction cache. These measurements will then provide a baseline for the evaluation of the proposed architecture.

Cache performance varies with different application domains. As the proposed system is intended for real-time applications, the benchmark for these tests should reflect this fact. However, there are no standard benchmarks available for embedded real-time systems. A real-time application was therefore adapted to create this benchmark. The application is from one node of a distributed motor control system [101] (see also Section 10.4.1). A simulation of the environment (sensors and actors) and the communication system (commands from the master station) forms part of the benchmark for simulating the real-world workload.

The data for all measurements was captured using a simulation of JOP and running the application for 500,000 clock cycles. During this time, the major loop of the application was executed several hundred times, effectively rendering any misses during the initialization code irrelevant to the measurements.

Direct-Mapped Cache

Table 4.13 gives the memory bytes and memory transactions per instruction byte for a standard direct-mapped cache. As we can see from the values for a cache size of 4KB, the kernel of the application is small enough to fit completely into the 4KB cache. The cache performs better (i.e. fewer bytes are transferred) with smaller block sizes. With smaller block sizes, the chance of unused data being read is reduced and the larger number of blocks reduces conflict misses. However, reducing the block size also increases memory transactions (MTIB), which directly relates to memory latency.

Which configuration performs best depends on the relationship between memory bandwidth and memory latency. Examples of average memory access times in cycles per instruc-

Cache size	Block size	MBIB	MTIB
1 KB	8	0.28	0.035
1 KB	16	0.38	0.024
1 KB	32	0.58	0.018
2 KB	8	0.17	0.022
2 KB	16	0.25	0.015
2 KB	32	0.41	0.013
4 KB	8	0.00	0.001
4 KB	16	0.01	0.000
4 KB	32	0.01	0.000

Table 4.13: Direct-mapped cache

Cache size	Block size	SRAM	SDRAM	DDR
1 KB	8	**0.18**	0.25	0.19
1 KB	16	0.22	**0.22**	0.16
1 KB	32	0.31	0.24	**0.15**
2 KB	8	**0.11**	0.15	0.12
2 KB	16	0.14	**0.14**	**0.10**
2 KB	32	0.22	0.17	0.11

Table 4.14: Direct-mapped cache, average memory access time

tion byte for different memory technologies are provided in Table 4.14. The third column shows the cache performance for a Static RAM (SRAM) that is very common in embedded systems. A latency of 1 clock cycle and an access time of 2 clock cycles per 32-bit word are assumed. For the synchronous DRAM (SDRAM) in the forth column, a latency of 5 cycles (3 cycles for the row address and 2 cycles for the CAS latency) is assumed. The memory delivers one word (4 bytes) per cycle. The Double Data Rate (DDR) SDRAM in the last column has an enhanced latency of 4.5 cycles and transfers data on both the rising and falling edge of the clock signal.

The data in bold give the best block size for different memory technologies. As expected, memories with a higher latency and bandwidth perform better with larger block sizes. For small block sizes, the latency clearly dominates the access time. Although the SRAM has half the bandwidth of the SDRAM and a quarter of the DDR, with a block size of 8 bytes,

Type	Cache size	MBIB	MTIB
Prefetch	8 B	1.37	0.342
Single method	1 KB	2.32	0.021
Two blocks	2 KB	1.21	0.013
Four blocks	4 KB	0.90	0.010

Table 4.15: Fixed block cache

it is faster than the DRAM memories. In most cases a block size of 16 bytes is the fastest solution and we will therefore use this configuration for comparison with the following cache solutions.

Fixed Block Cache

Cache performance for single method per block architectures is shown in Table 4.15. The measurements for a simple 8 byte prefetch queue are also given, for reference. With prefetching, we would expect to see an MBIB of about 1. The 37% overhead results from the fact that the prefetch queue fetches 4 bytes a time and has to buffer a minimum of 3 bytes for the instruction fetch stage. On a branch or return, the queue is flushed and these bytes are lost.

A single block that has to be filled on every invoke and return requires considerable overheads. More than twice the amount of data is read from the main memory than is consumed by the processor. However, the memory transaction count is 16 times lower than with simple prefetching, which can compensate for the large MBIB for main memories with high latency.

The solution with two blocks for two methods performs almost twice as well as the simple one method cache. This is due to the fact that, for all leaves in the call tree, the caller method can be found on return. If the block count is doubled again, the number of misses is reduced by a further 25%, but the cache size also doubles. For this measurement, an LRU replacement policy applies for the two and four block caches.

The same memory parameters as in the previous section are also used in Table 4.16. With the high latency of the DRAMs, even the simple one block cache is a faster (and more accurately predictable) solution than a prefetch queue. As MBIB and MTBI show the same trend as a function of the number of blocks, this is reflected in the access time in all three memory examples.

Type	Cache size	SRAM	SDRAM	DDR
Prefetch	8 B	1.02	2.05	1.71
Single Method	1 KB	1.18	0.69	0.39
Two blocks	2 KB	0.62	0.37	0.21
Four blocks	4 KB	0.46	0.27	0.16

Table 4.16: Fixed block cache, average memory access time

Cache size	Block count	MBIB	MTIB
1 KB	8	0.80	0.009
1 KB	16	0.71	0.008
1 KB	32	0.70	0.008
1 KB	64	0.70	0.008
2 KB	8	0.73	0.008
2 KB	16	0.37	0.004
2 KB	32	0.24	0.003
2 KB	64	0.12	0.001
4 KB	8	0.73	0.008
4 KB	16	0.25	0.003
4 KB	32	0.01	0.000
4 KB	64	0.00	0.000

Table 4.17: Variable block cache

Variable Block Cache

Table 4.17 shows the cache performance of the proposed solution, i.e. of a method cache with several blocks per method, for different cache sizes and number of blocks. For this measurement, a *next block* replacement policy applies.

In this scenario, as the MBIB is very high at a cache size of 1KB and almost independent of the block count, the cache capacity is seen to be clearly dominant. The most interesting cache size with this benchmark is 2KB. Here, we can see the influence of the number of blocks on both performance parameters. Both values benefit from more blocks. However, a higher block count requires more time or more hardware for the hit detection. With a cache size of 4KB and enough blocks, the kernel of the application completely fits into the

Cache size	Block count	SRAM	SDRAM	DDR
1 KB	8	0.41	0.24	0.14
1 KB	16	0.36	0.22	0.12
1 KB	32	0.36	0.21	0.12
1 KB	64	0.36	0.21	0.12
2 KB	8	0.37	0.22	0.13
2 KB	16	0.19	0.11	0.06
2 KB	32	0.12	0.08	0.04
2 KB	64	0.06	0.04	0.02

Table 4.18: Variable block cache, average memory access time

variable block cache, as we have seen with a 4KB traditional cache. From the gap between 16 and 32 blocks (within the 4KB cache), we can say that the application consists of fewer than 32 different methods.

It can be seen that even the smallest configuration with a cache size of 1KB and only 8 blocks outperforms fixed block caches with 2 or 4KB in both parameters (MBIB and MTIB). Compared with the fixed block solutions, MTIB is low in all configurations. This is due to the better hit rate, as indicated by the lower MBIB.

In most configurations, MBIB is higher than for the direct-mapped cache. It is very interesting to note that, in all configurations (even the small 1KB cache), MTIB is lower than in all 1KB and 2KB configurations of the direct-mapped cache. This is a result of the complete method transfers when a miss occurs and is clearly an advantage for main memory systems with high latency.

As in the previous examples, Table 4.18 shows the average memory access time per instruction byte for three different main memories.

In the DRAM configurations, the variable block cache directly benefits from the low MTBI. When comparing the values between SDRAM and DDR, we can see that the bandwidth affects the memory access time in a way that is approximately linear. The high latency of these memories is completely hidden. The configuration with 16 or more blocks and dynamic RAMs outperforms the direct-mapped cache of the same size. As expected, a memory with low latency (the SRAM in this example) depends on the MBIB values. The variable block cache is slower than the direct-mapped cache in the 1KB configuration because of the higher MBIB (0.7 compared to 0.3-0.6), and performs very similarly at a cache size of 2KB.

Cache type	MBIB	MTIB
Single method	2.32	0.021
Two blocks	1.21	0.013
Variable block (16)	0.37	0.004
Variable block (32)	0.24	0.003
Direct-mapped	0.25	0.015

Table 4.19: Caches compared

In Table 4.19, the different cache solutions with a size of 2KB are summarized. All full method caches with two or more blocks have a lower MTIB than a conventional cache solution. This becomes more significant with increasing latency in main memories. The MBIB value is only quite high for one or two methods in the cache. However, the most surprising result is that the variable block cache with 32 blocks outperforms a direct-mapped cache of the same size at both values.

We can see that predictability is indirectly related to performance – a trend we had anticipated. The most predictable solution with a single method cache performs very poorly compared to a conventional direct-mapped cache. If we accept a slightly more complex WCET analysis (taking a small part of the call tree into account), we can use the two-block cache that is about two times better.

With the variable block cache, it could be argued that the WCET analysis becomes too complex, but it is nevertheless simpler than that with the direct-mapped cache. However, every hit in the two-block cache will also be a hit in a variable block cache (of the same size). A tradeoff might be to analyze the program by assuming a two-block cache but using a version of the variable block cache. The additional performance gain can than be used by non- or soft real-time parts of an application.

4.7.5 Summary

In this section, we have extended the single cache performance measurement *miss rate* to a two value set, memory read and transaction rate, in order to perform a more detailed evaluation of different cache architectures. From the properties of the Java language – usually small methods and relative branches – we derived the novel idea of a *method cache*, i.e. a cache organization in which whole methods are loaded into the cache on method invocation and the return from a method. This cache organization is time-predictable, as all cache misses are lumped together in these two instructions. Using only one block for a

single method introduces considerable overheads in comparison with a conventional cache, but is very simple to analyze. We extended this cache to hold more methods, with one block per method and several smaller blocks per method.

Comparing these organizations quantitatively with a benchmark derived from a real-time application, we have seen that the variable block cache performs similarly to (and in one configuration even better than) a direct-mapped cache, in respect of the bytes that have to be filled on a cache miss. In all configurations and sizes of the variable block cache, the number of memory transactions, which relates to memory latency, is lower than in a traditional cache.

Only filling the cache on method invocation and return simplifies WCET analysis and removes another source of uncertainty, as there is no competition for the main memory access between instruction cache and data cache.

5 JOP Runtime System

A Java processor alone is not a complete JVM. This chapter describes the definition of a real-time profile for Java and a framework for a user-defined scheduler in Java. It concludes with the description of the JVM internal data structures to represent classes and objects.

5.1 A Real-Time Profile for Embedded Java

As standard Java is under-specified for real-time systems and the RTSJ does not fit for small embedded systems a new and simpler real-time profile is defined in this section and implemented on JOP. The guidelines of the specification are:

- High-integrity profile
- Easy syntax
- Easy to implement
- Low runtime overhead
- No syntactic extension of Java
- Minimum change of Java semantics
- Support for time measurement if a WCET analysis tool is not available
- Known overheads (documentation of runtime behavior and memory requirements of every JVM operation and all methods have to be provided)

The real-time profile under discussion is inspired by the restricted versions of the RTSJ described in [93] and [64] (see Section **??**). It is intended for high-integrity real-time applications and as a test case to evaluate the architecture of JOP as a Java processor for real-time systems.

The proposed definition is not compatible with the RTSJ. Since the application domain for the RTSJ is different from high-integrity systems, it makes sense for it *not* to be compatible with the RTSJ. Restrictions can be enforced by defining new classes (e.g. setting thread priority in the constructor of a real-time thread alone, enforcing minimum interarrival times for sporadic events).

All hardware interrupts are represented by threads under the control of the scheduler. With this solution, a priority is assigned to the device drivers and the execution time can be incorporated in the schedulability analysis with normal tasks. This solution also avoids problems with preemption latency provoked by device drivers. One example of this problem is the *caps-lock* issue in Linux [68]: A device driver performs a spinlock wait for keyboard acknowledgement and produces preemption latency up to 9166μs. With the proposed concept of hardware interrupts under scheduler control, a lower assigned priority to such a device driver avoids preemption delays of *more important* real-time threads and events.

To verify that this specification is expressive enough for high-integrity real-time applications, Ravenscar-Java (RJ) [64] (see Section **??**), with the additional necessary RTSJ classes, has been implemented on top of it. However, RJ inherits some of the complexity of the RTSJ. Therefore, the implementation of RJ has a larger memory and runtime overhead than this simple specification.

5.1.1 Application Structure

The application is divided in two different phases: *initialization* and *mission*. All non time-critical initialization, global object allocations, thread creation and startup are performed in the initialization phase. All classes need to be loaded and initialized in this phase. The mission phase starts after invocation of `startMission()`. The number of threads is fixed and the assigned priorities remain unchanged. The following restrictions apply to the application:

- Initialization and mission phase
- Fixed number of threads
- Threads are created at initialization phase
- All shared objects are allocated at initialization

5.1.2 Threads

Concurrency is expressed with two types of *schedulable objects*:

Periodic activities are represented by threads that execute in an infinite loop invoking waitForNextPeriod() to get rescheduled in predefined time intervals.

Asynchronous sporadic activities are represented by event handlers. Each event handler is in fact a thread, which is released by an hardware interrupt or a software generated event (invocation of fire()). Minimum interarrival time has to be specified on creation of the event handler.

The classes that implement the *schedulable objects* are:

RtThread represents a periodic task. As usual task work is coded in run(), which gets invoked on missionStart(). A scoped memory object can be attached to an RtThread at creation.

HwEvent represents an interrupt with a minimum interarrival time. If the hardware generates more interrupts, they get lost.

SwEvent represents a software-generated event. It is triggered by fire() and needs to override handle().

Listing 5.1 shows the definition of the basic classes.

Listing 5.2 shows the principle coding of a worker thread. An example for creation of two real-time threads and an event handler can be seen in Listing 5.3.

5.1.3 Scheduling

The scheduler is a preemptive, priority-based scheduler with unlimited priority levels and a unique priority value for each schedulable object. No real-time threads or events are scheduled during the initialization phase.

The design decision to use unique priority levels, instead of FIFO within priorities, is based on following facts: Two common ways to assign priorities are rate monotonic and, in a more general form, deadline monotonic assignment. When two tasks are given the same priority, we can choose one of them and assign a higher priority to that task and the task set will still be schedulable. This results in a strictly monotonic priority order and we do not need to deal with FIFO order. This eliminates queues for each priority level and results in a single, priority ordered task list with unlimited priority levels.

Synchronized blocks are executed with priority ceiling emulation protocol. An object, used for synchronization, for which the priority is not set, top priority is assumed. This avoids priority inversions on objects that are not accessible from the application (e.g. objects inside a library).

```
public class RtThread {

    public RtThread(int priority, int period)
    public RtThread(int priority, int period, int offset)
    public RtThread(int priority, int period, Memory mem)
    public RtThread(int priority, int period, int offset,
                    Memory mem)

    public void enterMemory()
    public void exitMemory()

    public void run()
    public boolean waitForNextPeriod()

    public static void startMission()
}

public class HwEvent extends RtThread {

    public HwEvent(int priority, int minTime, int number)
    public HwEvent(int priority, int minTime, Memory mem,
                   int number)

    public void handle()
}

public class SwEvent extends RtThread {

    public SwEvent(int priority, int minTime)
    public SwEvent(int priority, int minTime, Memory mem)

    public final void fire()
    public void handle()
}
```

Listing 5.1: Schedulable objects

In addition, the scheduler contains methods for worst-case time measurement for both the periodic work and handler methods. These measured execution times can be used during development when no WCET analysis tool is available.

5.1.4 Memory

The profile does not support a garbage collector. All memory should be allocated at the initialization phase. Without a garbage collector, the heap implicitly becomes immortal memory (as defined by the RTSJ). For objects created during the mission phase, a scoped memory is provided.[1] Each scoped memory area is assigned to one RtThread. A scoped memory area cannot be shared between threads. No references are allowed from the heap to scoped memory. Scoped memory is explicitly entered and left using invocations from the application logic. Memory areas are cleared both on creation and when leaving the scope (invocation of exitMemory()), leading to a memory area with constant allocation time, as opposed to memory with linear allocation time (as the memory type LTMemory in the RTSJ) [27].

5.1.5 Restrictions on Java

A list of some of the language features that should be avoided for WCET analyzable real-time threads and bound memory usage:

WCET: Only analyzable language constructs are allowed (see [91]).

Static class initialization: Since the definition when to call the static class initializer is problematic (see Section 5.5.1), they are disallowed. Move this code to a static method (e.g. init()) and invoke it explicitly in the initialization phase.

Inheritance: Reduce usage of interfaces and overridden methods.

String concatenation: In the immortal memory scope, only string concatenation with string literals is allowed.

Finalization: finalize() has a weak definition in Java. Because real-time systems run *forever*, objects in the heap, which is immortal in this specification, will never be finalized. Objects in scoped memory are released on exitMemory(). However, finalizations on these objects complicate WCET analysis of exitMemory().

[1] As we now consider real-time GC as the better solution, scopes are not supported in the current implementation of the profile.

Dynamic Class Loading: Due to the implementation and WCET analysis complexity dynamic class loading is avoided.

A program analysis tool can greatly help in enforcing these restrictions.

5.1.6 Implementation Results

The initial idea was to implement scheduling and dispatching in microcode. However, many Java bytecodes have a one to one mapping to a microcode instruction, resulting in a single cycle execution. The performance gain of an algorithm coded in microcode is therefore negligible. As a result, almost all of the scheduling is implemented in Java. Only a small part of the dispatcher, a memory copy, is implemented in microcode and exposed with a special bytecode.

Experimental results of basic scheduling benchmarks, such as periodic thread jitter, context switch time for threads and asynchronous events, can be found in [104].

To implement system functions, such as scheduling, in Java, access to JVM and processor internal data structures have to be available. However, Java does not allow memory access or access to hardware devices. In JOP, this access is provided by way of additional bytecodes. In the Java environment, these bytecodes are represented as static native methods. The compiled invoke instruction for these methods (invokestatic) is replaced by these additional bytecodes in the class file. This solution provides a very efficient way to incorporate low-level functions into a pure Java system. The translation can be performed during class loading to avoid non-standard class files.

A pure Java system, without an underlying RTOS, is an unusual system with some interesting new properties. Java is a safer execution environment than C (e.g. no pointers) and the boundary between *kernel* and *user space* can become quite loose. Scheduling, usually part of the operating system or the JVM, is implemented in Java and executed in the same context as the application. This property provides an easy path to a framework for user-defined scheduling.

5.2 User-Defined Scheduler

The novel approach to implement a real-time scheduler in Java opens up new possibilities. An obvious next step is to extend this system to provide a framework for user-defined scheduling in Java. New applications, such as multimedia streaming, result in *soft* real-time systems that need a more flexible scheduler than the traditional fixed priority based ones.

```
public class Worker extends RtThread {

    private SwEvent event;

    public Worker(int p, int t,
                  SwEvent ev) {

        super(p, t,
            // create a scoped memory area
            new Memory(10000)
        );
        event = ev;
        init();
    }

    private void init() {
        // all initialzation stuff
        // has to be placed here
    }

    public void run() {

        for (;;) {
            work();        // do some work
            event.fire(); // and fire an event

            // some work in scoped memory
            enterMemory();
            workWithMem();
            exitMemory();

            // wait for next period
            if (!waitForNextPeriod()) {
                missedDeadline();
            }
        }
        // should never reach this point
    }
}
```

Listing 5.2: A periodic real-time thread

```
// create an Event
Handler h = new Handler(3, 1000);

// create two worker threads with
// priorities according to their periods
FastWorker fw = new FastWorker(2, 2000);
Worker w = new Worker(1, 10000, h);

// change to mission phase for all
// periodic threads and event handler
RtThread.startMission();

// do some non real-time work
// and invoke sleep() or yield()
for (;;) {
    watchdogBlink();
    Thread.sleep(500);
}
```

Listing 5.3: Start of the application

This section provides a simple-to-use framework to evaluate new scheduling concepts for these applications in real-time Java.

The following section analyzes which events are exposed to the scheduler and which functions from the JVM need to be available in the user space. It is followed by the definition of the framework and examples of how to implement a scheduler using this framework.

5.2.1 Schedule Events

The most important element of the user-defined scheduler is to define which events result in the scheduling of a new task. When such an event occurs, the user-defined scheduler is invoked. It can update its task list and decide which task is dispatched.

Timer interrupt: For timed scheduling decisions, a programmable timer generates exact timed interrupts. The scheduler controls the time interval for the next interrupt.

HW interrupt: Each hardware-generated interrupt can be associated with an asynchronous event. This allows the execution of a device driver under the control of the scheduler. Latencies of the device driver can be controlled by assigning the right priority in a priority scheduler.

Monitor: To allow different implementations of priority inversion protocols, hooks for monitorenter and monitorexit are provided.

Thread block: Each thread can cease execution via a call of the scheduler. This function is used to implement methods such as waitForNextPeriod() or sleep(). The reason for blocking (e.g. end of periodic work) has to be communicated to the scheduler (e.g. next time to be unblocked for a periodic task).

SW event: Invoking fire() on an event provides support for signaling. wait(), notify() or notifyAll() are not necessary. However, this mechanism is not part of the scheduling framework. It can be implemented with the user-defined scheduler and an associated thread class.

5.2.2 Data Structures

To implement a scheduler in Java, some internal JVM data structures need to be accessible.

Object: In Java, any object (including an object from the class Class for static methods) can be used for synchronization. Different priority inversion protocols require different

data structures to be associated with an object. Each object provides a field, accessed through a Scheduler method, in which these data structures can be attached.

Thread: A list of all threads is provided to the scheduler. The scheduler is also notified when a new thread object is created or a thread terminates. The scheduler controls the start of threads.

5.2.3 Services for the Scheduler

The real-time JVM and the hardware platform have to provide some minimum services. These services are exposed through Scheduler:

Dispatch: The current active thread is interrupted and a new thread is placed in the run state.

Time: System time with high resolution (microseconds, if the hardware can provide it) is used for time derived scheduling decisions.

Timer: A programmable timer interrupt (not a timer tick) is necessary for accurate time triggered scheduling.

Interrupts: To protect the data structures of the scheduler all interrupts can be disabled and enabled.

5.2.4 Class Scheduler

The class Scheduler has to be extended to implement a user-defined scheduler. The class Task represents *schedulable objects*. For non-trivial scheduling algorithms, Task is also extended. The scheduler lives in normal thread space. There is no special context such as kernel space. The methods of Scheduler are categorized by the caller module and described in detail below.

Application To use a scheduler in an application, the application only has to create one instance of the scheduler class and has to decide when scheduling starts.

```
public Scheduler()
```

A single instance of the scheduler is created by the application.

```
public void start()
```

This method initiates the transition to the mission phase of the application. All created tasks are started and scheduled under the control of the user scheduler.

Task A user-defined scheduler usually needs an associated user-defined thread class (an extension of Task). This class interacts with the scheduler by invoking following methods from Scheduler:

```
void addTask(Task t)
```

The scheduler has access to the list of created tasks to use at the start of scheduling. For dynamic task creation after the start of the scheduler, this method is called by the constructor of Task, to notify the scheduler to update its list.

```
void isDead(Task t)
```

The scheduler is notified when a Task returns from the run() method. The scheduler removes this Task from the list of schedulable objects.

```
void block()
```

Every Task can cease execution via a call of the scheduler. This method is used to implement methods such as waitForNextPeriod() or sleep() in a user defined thread class.

Java Virtual Machine The methods listed below provide the essential points of communication between the JVM and the scheduler. As a response to an interrupt (hardware or timer), entrance or exit of a synchronized method/block the JVM invokes a method from the scheduler.

```
abstract void schedule()
```

This is the main entry point for the scheduler. This method has to be overridden to implement the scheduling algorithm. It is called from the JVM on a timed event or a software interrupt (see genInt()) is issued (e.g. when a Task gives up execution).

```
void interrupt(int nr)
```

The scheduler is notified on a hardware event. It can directly call an associated device driver or use this information to unblock a waiting task.

```
void monitorEnter(Object o)
void monitorExit(Object o)
```

These methods are invoked by the JVM on synchronized methods and blocks (JVM bytecodes monitorenter and monitorexit). They provide hooks for executing dynamic priority changes in the scheduler.

Scheduler Services of the JVM needed to implement a scheduler are provided through static methods.

```
static final void genInt()
```

This service from the JVM schedules a software interrupt. As a result, schedule() is called. This method is the standard way of switching control to the scheduler. It is e.g. invoked by block().

```
static final void enableInt()
static final void disableInt()
```

The scheduler cannot use monitors to protect its data structures as the scheduler itself is in charge of handling monitors. To protect the data structures of the scheduler, it can globally enable and disable interrupts.

```
static final void dispatch(Task nextTask, int nextTim)
```

This method dispatches a Task and schedules a timer interrupt at nextTim.

```
static final void attachData(Object obj, Object data)
static final Object getAttachedData(Object obj)
```

The behavior of the priority inversion avoidance protocol is defined by the user scheduler. The root of the Java class hierarchy (java.lang.Object) contains a JVM internal reference of generic type Object that can be used by the scheduler to attach data structures for monitors. The first argument of these methods is the object to be used as a monitor.

Scheduler or Task The following two methods are utility functions useful for the scheduler and the thread implementation.

```
static final int getNow()
```

To support time-triggered scheduling, the system provides access to a high-resolution time or counter. The returned value is the time since startup in microseconds. The exact resolution is implementation-dependent.

```
static final Task getRunningTask()
```

The current running Task (in which context the scheduler is called) is returned by this method.

```
public class Task {

    public Task()
    public Task(Memory mem)
    void start()

    public void enterMemory()
    public void exitMemory()

    public void run()

    static Task getFirstTask()
    static Task getNextTask()
}
```

Listing 5.4: A basic schedulable object

5.2.5 Class Task

A basic structure for schedulable objects is shown in Listing 5.4. This class is usually extended to provide a thread implementation that fits in the user-defined scheduler. The class Task is intended to be minimal. To avoid inheriting methods that do not fit for some applications, it does not extend java.lang.Thread. However, Task can be used to *implement* java.lang.Thread.

The methods enterMemory and exitMemory are used by the application to provide scoped memory for temporary allocated objects. Task provides a list of active tasks for the scheduler.

One issue, raised by the implementation of the framework, is the way in which access rights to methods need to be defined in Java. All methods, except start(), should be private or protected. However, some methods, such as schedule(), are invoked by a part of the JVM, which is also written in Java but resides in a different package. This results in defining the methods as public and *hoping* that they are not invoked by the application code. The C++ concept of friends would greatly help in sharing information over package boundaries without making this information public.

5.2.6 A Simple Example Scheduler

Listing 5.5 shows a full example of using this framework to implement a simple round robin scheduler.

The only method that needs to be supplied is `schedule()`. For a more advanced scheduler, it is necessary to provide a combination of a user defined thread class and a scheduler class. These two classes have to be tightly integrated, as the scheduler uses information provided by the thread objects for its scheduling decisions.

```
public class RoundRobin extends Scheduler {

    //
    //   test threads
    //
    static class Work extends Task {

        private int c;

        Work(int ch) {
            c = ch;
        }

        public void run() {

            for (;;) {
                Dbg.wr(c); // debug output

                // busy wait to simulate
                // 3 ms workload in Work.
                int ts = Scheduler.getNow();
                ts += 3000;
                while (ts-Scheduler.getNow()>0)
                    ;
            }
        }
    }

    //
    //    user scheduler starts here
    //

    public void addTask(Task t) {
        // we do not allow tasks to be
        // added after start().
    }
```

```
        //
        //    called by the JVM
        //
        public void schedule() {
            Task t = getRunningTask().getNextTask();
            if (t==null) t = Task.getFirstTask();
            dispatch(t, getNow()+10000);
        }


        public static void main(String[] args) {

            new Work('a');
            new Work('b');
            new Work('c');

            RoundRobin rr = new RoundRobin();

            rr.start();
        }
    }
```

Listing 5.5: A very simple scheduler

5.2.7 Interaction of Task, Scheduler and the JVM

The framework is used to re-implement the scheduler described in Section 5.3. In the original implementation, the interaction between scheduling and threads was simple, as the scheduling was part of the thread class. Using the framework, these functions have to be split to two classes, extending Task and Scheduler. Both classes are placed in the same package to provide simpler information sharing with some protection from the rest of the application. For performance reasons, data structures are directly exposed from one class to the other.

The resulting implementation is compatible with the first definition, with the exception that RtThread now extends Task. However, no changes in the application code are necessary.

Figure 5.1 is an interaction example of this scheduler within the framework. The inter-

```
for (;;) {
    doPeriodicWork();
    waitForNextPeriod();
}
```

Listing 5.6: Code fragment oft the application

action diagram shows the message sequences between two application tasks, the scheduler, the JVM and the hardware. The hardware represents interrupt and timer logic. The corresponding code fragments of the application, RtThread and PriorityScheduler are shown in Listing 5.6, 5.7 and 5.8. Task 2 is a periodic task with a higher priority than Task 1.

The first event is a timer event to unblock Task 2 for a new period. The generated timer event results in a call of the user defined scheduler. The scheduler performs its scheduling decision and issues a context switch to Task 2. With every context switch the timer is reprogrammed to generate an interrupt at the next time triggered event for a higher priority task. Task 2 performs the periodic work and ceases execution by invocation of waitForNextPeriod(). The scheduler is called and requests an interrupt from the hardware resulting in the same call sequence as with a timer or other hardware interrupt. The software generated interrupt imposes negligible overhead and results in a single entry point for the scheduler. Task 1 is the only ready task in this example and is resumed by the scheduler.

Using a general scheduling framework for a real-time scheduler is not without its costs. Additional methods are invoked from a scheduling event until the actual dispatch takes place. The context switch is about 20% slower than in the original implementation. It is the opinion of the author that the additional cost is outweighed by the flexibility of the framework.

5.2.8 Predictability

The architecture of JOP is designed to simplify WCET analysis. Every JVM bytecode maps to one ore more microcode instructions. Every microcode instruction takes exactly one cycle to execute. Thus, the execution time at the bytecode level is known cycle accurately. The microcode contains no data dependent or unbound loops that would compromise the WCET analysis (see Chapter 7).

The worst-case time for dispatching is known cycle accurately on this architecture. Only the time behavior of the user scheduler needs to be analyzed. With the known WCET of every bytecode, as listed in Appendix D, the WCET of the scheduler can be obtained by

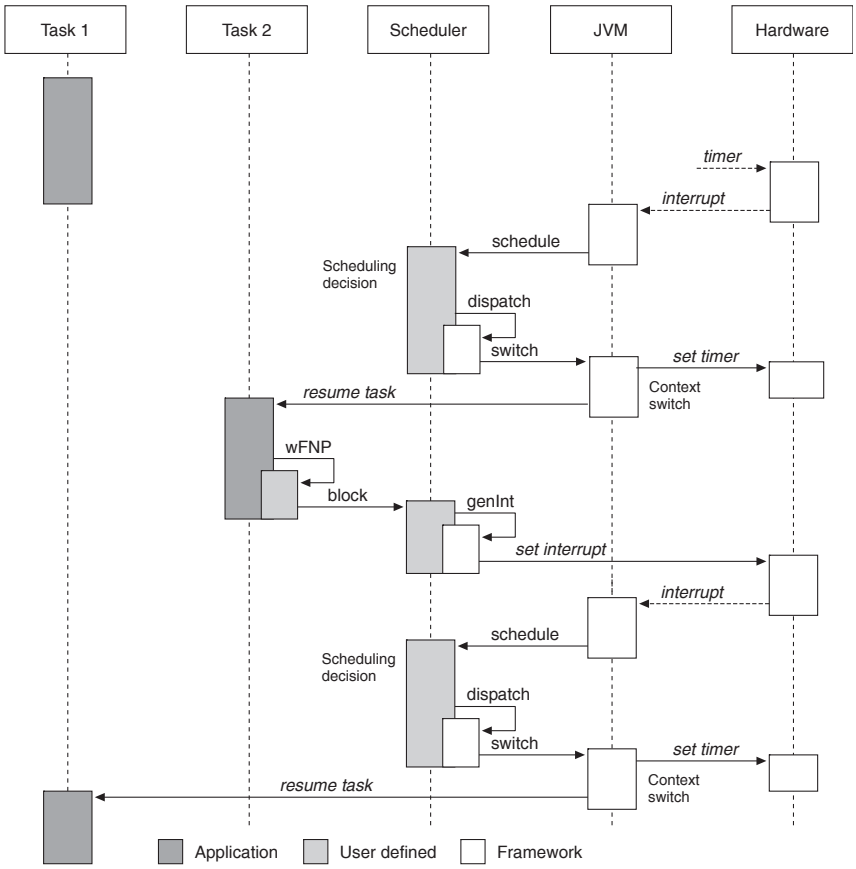

Figure 5.1: Interaction and message exchange between the application, the scheduler, the JVM and the hardware

```
public boolean waitForNextPeriod() {

    synchronized(monitor) {

        // ps is the instance of
        // the PriorityScheduler
        int nxt = ps.next[nr] + period;

        int now = Scheduler.getNow()
        if (nxt-now < 0) {
            // missed deadline
            doMissAction();
            return false;
        } else {
            // time for the next unblock
            ps.next[nr] = nxt;
        }
        // just schedule an interrupt
        // schedule() gets called.
        ps.block();
    }
    return true;
}
```

Listing 5.7: Implementation in RtThread

```
public void schedule() {

    // Find the ready thread with
    // the highest priority.
    int nr = getReady();

    // Search the list of sleeping threads
    // to find the nearest release time
    // in the future of a higher priority
    // thread than the one that will be
    // released now.
    int time = getNextTimer(nr);

    // This time is used for the next
    // timer interrupt.
    // Perform the context switch.
    dispatch(task[nr], time);
    // No access to locals after this point.
    // We are running in the NEW context!
}
```

Listing 5.8: Implementation of the PriorityScheduler

examining it at the bytecode level. This can be done manually or with a WCET analysis tool.

5.2.9 Related Work

Several implementations of user-level schedulers in standard operating systems have been proposed. In [68], the Linux scheduling mechanism is enhanced. It is divided into a dispatcher and an allocator. The dispatcher remains in kernel space; while the allocator is implemented as a user space function. The allocator transforms four basic scheduling parameters (priority, start time, finish time and budget) into scheduling attributes to be used by the dispatcher. Many existing schedulers can be supported with this parameter set, but others that are based on different parameters cannot be implemented. This solution does not address the implementation of protocols for shared resources.

A different approach defines a new API to enable applications to use application-defined scheduling in a way compatible with the scheduling model defined in POSIX [96]. It is implemented in the MaRTE OS, a minimal real-time kernel that provides the C and Ada language POSIX interface. This interface has been submitted to the Real-Time POSIX Working Group for consideration.

One approach to user-level scheduling in Java can be found in [37]. A thread *multiplexor*, as part of the FLEX ahead-of-time compiler system for Java, is used for utility accrual scheduling. However, the underlying operating system – in this case Linux – can still be seen through the framework and there is no support for Java synchronization.

5.2.10 Summary

This section and Section 5.3 consider the implementation of real-time scheduling on a Java processor. The novelty of the described approach is in implementing functions usually associated with an RTOS in Java. That means that real-time Java is not based on an RTOS, and therefore not restricted to the functionality provided by the RTOS. With JOP, a self-contained real-time system in pure Java becomes possible. This system is augmented with a framework to provide scheduling functions at the application level. The implementation of the specification, described in Section 5.3, is successfully used as the basis for a commercial real-time application in the railway industry. Future work will extend this framework to support multiple schedulers. A useful combination of schedulers would be: one for standard `java.lang.Thread` (optimized for throughput), one for soft real-time tasks and one for hard real-time tasks.

5.3 A Profile for Safety Critical Java

The proposed profile is an refinement of the profile described in Section . Further development of applications on JOP shall be based on this profile and the current applications (e.g. ÖBB bg and Lift) should be migrated to this profile.

TODO: update with our ISROC 2007 paper and remove Jan's text.

5.4 Safety Critical Java

Puschner and Wellings were the first to consider the concerns of safety- and mission-critical systems in the context of the RTSJ for Java [93]. Their proposal adopts the approach pioneered by the Ravenscar tasking profile for Ada [23] which defined a strict subset of the Ada language for high-integrity systems. This work was later refined by Schoeberl et al. [115].

In this paper we focus on the Safety Critical Java (SCJ) specification, a new standard for safety critical applications which is being drafted by the JSR 302 expert group.

We should note that JSR 302 has not been finalized, thus our presentation gives an overview of work in progress. Furthemore, our proposal of real-time garbage collection to SCJ is an extension of the proposed standard.

This draft JSR 302 standard, like previous work, defines a strict subset of the RTSJ which is intended to provide a programming model suited to a large class of safety critical applications. Restricting the features of the RTSJ is intended to make programs more amenable to worst case analysis and manual or automatic validation. The SCJ is structured in three increasingly expressive levels: Level 0 restricts applications to a single threaded cyclic executive, level 1 assumes a single "mission" with a static thread assignment, and level 2 is a multi-mission model with dynamic thread creation. This paper focuses on level 1 which is expected to cover a large number of existing SC applications. It should be noted that while all levels are designed to run on a vanilla RTSJ VM, it is expected that vendors will provided implementations that are optimized for the particular features of each level.

5.4.1 SCJ Level 1

Level 1 of the SCJ requires that all threads be defined during an initial *initialization* phase. This phase is run only once at virtual machine startup. The second phase, called the *mission* phase, begins only when all threads have been started. This phase runs until virtual machine shutdown. Level 1 supports only two kinds of schedulable objects: periodic threads and sporadic events. The latter can be generated by either hardware or software. This re-

```
package javax.safetycritical;

public abstract class RealtimeThread {

    protected RealtimeThread(RelativeTime period,
        RelativeTime deadline,
        RelativeTime offset, int memSize)

    protected RealtimeThread(String event,
        RelativeTime minInterval,
        RelativeTime deadline, int memSize)

    abstract protected boolean run();

    protected boolean cleanup() {
        return true;
    }
}

public abstract class PeriodicThread
        extends RealtimeThread {

    public PeriodicThread(RelativeTime period,
        RelativeTime deadline,
        RelativeTime offset, int memSize)

    public PeriodicThread(RelativeTime period)
}
```

Figure 5.2: Periodic thread definition for SCJ

strictions keeps the schedulability analysis simple. In SCJ priority ceiling emulation is the default monitor control policy. The default ceiling is top priority.

The Java wait and notify primitives are not allowed in SCJ level 0 and 1. This further simplifies analysis. The consequence is that a thread context switch can only occur if a higher priority thread is released or if the current running thread yields (in the case of SCJ by returning from the run() method).

In the RTSJ, periodic tasks are expressed by unbounded loops with, at some point, a call to the waitForNextPeriod() (or wFNP() for short) method of class RealtimeThread.

```
new PeriodicThread(
    new RelativeTime(...)) {

        protected boolean run() {
            doPeriodicWork();
            return true;
        }
};
```

Figure 5.3: A periodic application thread in SCJ

This has the effect of yielding control to the scheduler which will only wake the thread when its next period starts or shortly thereafter. In SCJ, as a simplification, periodic logic is encapsulated in a `run()` method which is invoked at the start of every period of a given schedulable object. When the thread returns from `run()` it is blocked until the next period.

Figure 5.2 shows part of the definition of the SCJ thread classes from [115][2]. Figure 5.3 shows the code for a periodic thread. This class has only one `run()` method which performs a periodic computation.

The loop construct with `wFNP()` is not used. The main intention to avoid the loop construct, with the possibility to split application logic into *mini* phases, is simplification of the WCET analysis. Only a single method has to be analyzed per thread instead of all possible control flow path between `wFNP()` invocations.

We contrast the SCJ threading with Figure 5.4 where a periodic RTSJ thread is shown. Suspension of the thread to wait for the next period is performed by an explicit invocation of `wFNP()`. The coding style in this example makes analysis of the code more difficult than necessary. First the initialization logic is mixed with the code of the mission phase, this means that a static analysis may be required to discover the boundary between the startup code and the periodic behavior. The code also performs mode switches with calls to `wFNP()` embedded in the logic. This makes the worst case analysis more complex as calls to `wFNP()` may occur anywhere and require deep understanding of feasible control flow paths. Another issue, which does not affect correctness, is the fact that object references can be preserved in local variables across calls to `wFNP()`. As we will see later this has implications for the GC.

[2]These are similar to the draft JSR 302 class definitions, but as the specification is still in the process of being finalized we choose to use the classes available in the infrastructure we use for our implementation.

```
public void run() {

    State local = new State();
    doSomeInit();
    local.setA();
    waitForNextPeriod();

    for (;;) {
        while (!switchToB()) {
            doModeAwork();
            waitForNextPeriod();
        }
        local.setB();
        while (!switchToA()) {
            doModeBWork();
            waitForNextPeriod();
        }
        local.setA();
    }
}
```

Figure 5.4: Possible logic for a periodic thread in the RTSJ

5.5 JVM Architecture

This section presents the details of the implementation of the JVM on JOP. The representation of objects and the stack frame is chosen to support JOP as a processor for real-time systems. However, since the data structures are realized through microcode they can be easily changed for a system with different needs. For example: to simplify a compacting GC a handle to an object can be implemented by changing the microcode of getfield, putfield and new.

5.5.1 Runtime Data Structures

Memory is addressed as 32-bit data, which means that memory pointers are incremented for every four bytes. No single byte or 16-bit access is necessary. The abstract type reference is a pointer to memory that represents the object or an array. The reference is pushed on the stack before an instruction can operate on it. A null reference is represented by the value 0.

Stack Frame

On invocation of a method, the invoker's context is saved in a newly allocated frame on the stack. It is restored when the method returns. The saved context consists of the following registers:

SP: Immediately before invocation, the stack pointer points to the last argument for the called function. This value is reduced by the argument count (i.e. the arguments are consumed) and saved in the new stack frame.

PC: The pointer to the next bytecode instruction after the invoke instruction.

VP: The pointer to the memory area on the stack that contains the locals.

CP: The pointer to the constant pool of the class from the invoking method.

MP: The pointer to the method structure of the invoking method.

SP, PC and VP are registers in JOP while CP and MP are local variables of the JVM. Figure 5.5 provides an example of the stack before and after invoking a method. In this example, the called method has two arguments and contains two local variables. If the method is a virtual one, the first argument is the reference to the object (the *this*-pointer). The arguments implicitly become locals in the called method and are accessed in the same

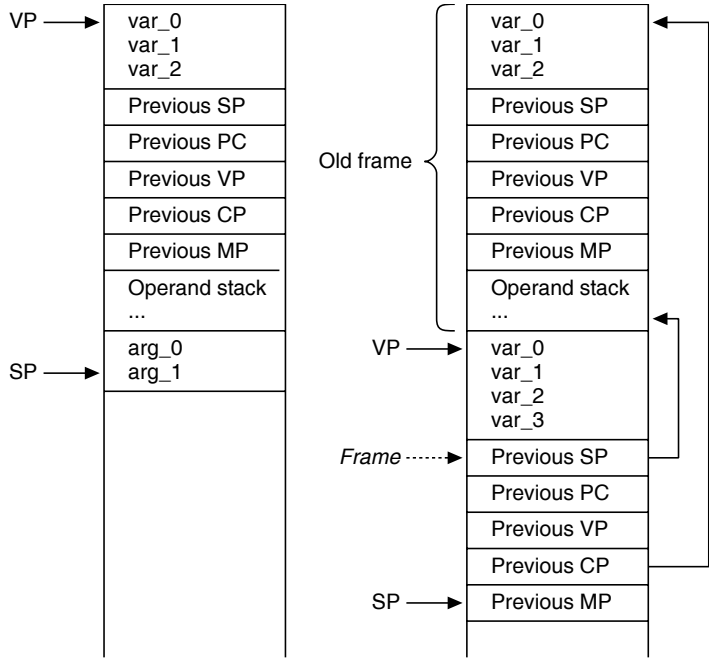

Figure 5.5: Stack change on method invocation

way as local variables. The start of the stack frame (*Frame* in the figure) needs not to be saved. It is not needed during execution of the method or on return. To access the starting address of the frame (e.g. for an exception) it can be calculated with information from the method structure:

$$Frame = VP + arg\_cnt + locals\_cnt$$

Object Layout

Figure 5.6 shows the representation of an object in memory. The object reference points to the first instance variable of the object. At the offset -1, a pointer is located to access class information. To speed-up method invocation, it points directly to the method table of the objects class instead of the beginning of the class data. With the GC, the object is accessed via one indirection, the handle (see Chapter 6). The pointer to the method vector base is part of the handle.

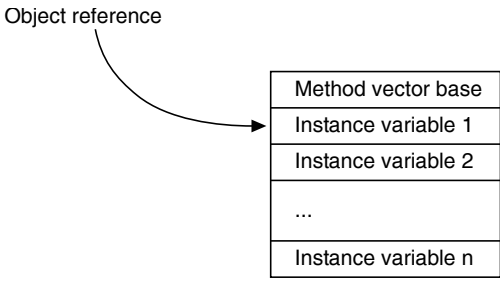

Figure 5.6: Object format

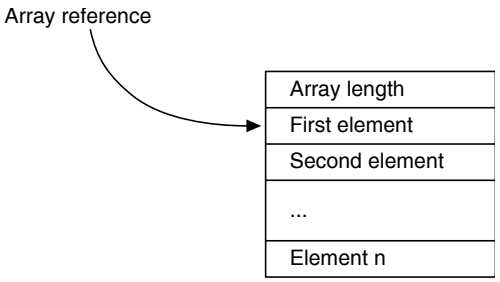

Figure 5.7: Array format

Array Layout

Figure 5.7 shows the representation of an array in memory. The object reference points to the first element of the array. At the offset -1, the length of the array can be found. With the GC the array is accessed via one indirection, the handle (see Chapter 6). The size fields is part of the handle.

Class Structure

Runtime class information, as shown in Figure 5.8, consists of the class variables, the dispatch table for the methods, the constant pool and an optional interface table.

The class reference is obtained from the constant pool when a new object is created. The method vector base pointer is a reference from an object to its class (see Figure 5.6). It is used on invokevirtual with an index retrieved from the constant pool. A pointer to the method structure of the current method is saved in the JVM variable MP. The method structure, as shown in Figure 5.9, contains the starting address and length of the method (in

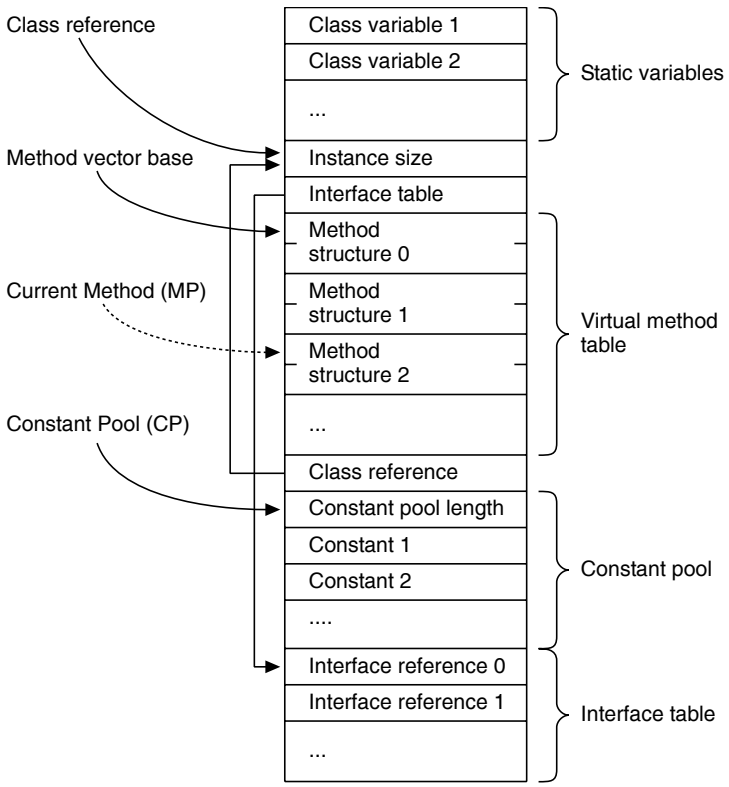

Figure 5.8: Runtime class structure

Start address	Method length	
Constant pool	Local count	Arg. count

Figure 5.9: Method structure

32-bit words), argument and local variable count and a pointer to the constant pool of the class. Since the constant pool is an often accessed memory area, a pointer to it is kept in the JVM variable CP.

The interface table contains references to the method structures of the implementation. Only classes that implement an interface contain this table. To avoid searching the class hierarchy on invokeinterface, each interface method is assigned a unique index. This provides constant execution time, but can lead to large interface tables.

The constant pool contains various constants of a class. The entry at index 0 is the length of the pool. All constants, which are symbolic in the class files, are resolved on class loading or during pre-linking. The different constant types and their values after resolving are listed in Table 5.1. The names for the types are the same as in the JVM specification [69].

Class Initialization Issues

According to [69] the static initializers of a class C are executed immediately before one of the following occurs: (i) an instance of C is created; (ii) a static method of C is invoked or (iii) a static field of C is used or assigned. The issue with this definition is that it is not allowed to invoke the static initializers at JVM startup and it is not so obvious when it gets invoked.

It follows that the bytecodes getstatic, putstatic, invokestatic and new can lead to class initialization and the possibility of high WCET values. In the JVM, it is necessary to check every execution of these bytecodes if the class is already initialized. This leads to a loss of performance and is violated in some existing implementations of the JVM. For example, the first version of CACAO [62] invokes the static initializer of a class at compilation time. Listing 5.9 shows an example of this problem.

JOPizer tries to find a correct order of the class initializers and puts this list into the application file. If a circular dependency is detected the application will not be built. The class initializers are invoked at JVM startup.

Constant type	Description
Class	A pointer to a class (class reference)
Fieldref	For static fields: a direct pointer to the field For object fields: the position relative to the object reference
Methodref	For static methods: a direct pointer to the method structure For virtual methods: the offset in the method table (= index*2) and the number of arguments
InterfaceMethodref	A system wide unique index into the interface table
String	A pointer to the string object that represents the string constant
Integer	The constant value
Float	The constant value
Long	This constant value spans two entries in the constant pool
Double	Same as for long constants
NameAndType	Not used
Utf8	Not used

Table 5.1: Constant pool entries

```
public class Problem {

    private static Abc a;
    public static int cnt; // implicitly set to 0

    static {
        // do some class initializaion
        a = new Abc();  //even this is ok.
    }

    public Problem() {
        ++cnt;
    }
}

// anywhere in some other class, in situation,
// when no instance of Problem has been created
// the following code can lead to
// the execution of the initializer
int nrOfProblems = Problem.cnt;
```

Listing 5.9: Class initialization can occur very late

Synchronization Issue

Synchronization is possible with methods and on code blocks. Each object has a monitor associated with it and there are two different ways to gain and release ownership of a monitor. Bytecodes monitorenter and monitorexit explicitly handle synchronization. In other cases, synchronized methods are marked in the class file with the access flags. This means that all bytecodes for method invocation and return must check this access flag. This results in an unnecessary overhead on methods without synchronization. It would be preferable to encapsulate the bytecode of synchronized methods with bytecodes monitorenter and monitorexit. This solution is used in Suns picoJava-II [126]. The code is manipulated in the class loader. Two different ways of coding synchronization, in the bytecode stream and as access flags, are inconsistent. With JOPizer the same manipulation is of the methods is performed to wrap the method code in a synchronized block when the method is defined synchronized.

5.5.2 Booting the JVM

One interesting issue for an embedded system is the how the startup or boot-up is performed. On power-up, the FPGA starts the configuration state machine to read the FPGA configuration data either from a Flash or via a download cable (for development). When the configuration has finished, an internal reset is generated. After that reset, microcode instructions are executed starting from address 0. At this stage, we have not yet loaded any application program (Java bytecode). The first sequence in microcode performs this task. The Java application can be loaded from an external Flash or via a serial line (or USB port) from a PC. The microcode assembly configured the mode. Consequently, the Java application is loaded into the main memory. To simplify the startup code we perform the rest of the startup in Java itself, even when some parts of the JVM are not yet setup.

In the next step, a minimal stack frame is generated and the special method Startup.boot() is invoked. From now on JOP runs in Java mode. The method boot() performs the following steps:

1. Send a greeting message to *stdout*
2. Detect the size of the main memory
3. Initialize the data structures for the garbage collector
4. Initialize java.lang.System
5. Print out JOP's version number, detected clock speed, and memory size
6. Invoke the static class initializers in a predefined order
7. Invoke the main method of the application class

6 Real-Time Garbage Collection

Automatic memory management or garbage collection greatly simplifies development of large systems. However, garbage collection is usually not used in real-time systems due to the unpredictable temporal behavior of current implementations of a garbage collector. In this chapter we describe a concurrent collector that is scheduled periodically in the same way as ordinary application threads. We provide an upper bound for the collector period so that the application threads will never run out of memory. This chapter is based on the published papers [108] and [116].

6.1 Introduction

Garbage Collection (GC) is an essential part of the Java runtime system. GC enables automatic dynamic memory management which is essential to build large applications. Automatic memory management frees the programmer from complex and error prone explicit memory management (malloc and free).

However, garbage collection is considered unsuitable for real-time systems due to the unpredictable blocking times introduced by the GC work. One solution, used in the Real-Time Specification for Java (RTSJ) [21], introduces new thread types with program-managed, scoped memory for dynamic memory requirements. This scoped memory (and static memory called *immortal* memory) is not managed by the GC, and strict assignment rules between different memory areas have to be checked at runtime. This programming model differs largely from standard Java and is difficult to use [74, 86].

We believe that for the acceptance of Java for real-time systems, the restrictions imposed by the RTSJ are too strong. To simplify creation of possible large real-time applications, most of the code should be able to use the GC managed heap. For a collector to be used in real-time systems two points are essential:

- The GC has to be incremental with a short maximum blocking time that has to be known
- The GC has to keep up with the garbage generated by the application threads to avoid out-of-memory stalls

The first point is necessary to limit interference between the GC thread and high-priority threads. It is also essential to minimize the overhead introduced by read- and write-barriers, which are necessary for synchronization between the GC thread and the application threads. The design of a GC within these constraints is the topic of this chapter.

The second issue that has to be considered is scheduling the GC so that the GC collects enough garbage. The memory demands (static and dynamic) by the application threads have to be analyzed. These requirements, together with the properties of the GC, result in scheduling parameters for the GC thread. We will provide a solution to calculate the maximum period of the GC thread that will collect enough memory in each collector cycle so we will never run out of memory. The collector cycle depends on the heap size and the allocation rate of the application threads.

To distinguish between other garbage collectors and a collector for (hard) real-time systems we define a real-time collector as follows:

> A real-time garbage collector provides time predictable automatic memory management for tasks with a bounded memory allocation rate with minimal temporal interference to tasks that use only static memory.

The collector presented in this chapter is based on the work by Steele [121], Dijkstra [32] and Baker [16]. However, the copying collector is changed to perform the copy of an object concurrently by the collector and not as part of the mutator work. Therefore we name it *concurrent-copy* collector.

We will use the terms first introduced by Dijkstra with his *On-the-Fly* concurrent collector. The application is called the *mutator* to reinforce that the application changes (mutates) the object graph while the GC does the collection work. The GC process is simply called *collector*. In the following discussion we will use the color scheme of white, gray, and black objects as introduced by Dijkstra with his *On-the-Fly* concurrent collector.

Black indicates that the object and all immediate descendants have been visited by the collector.

Grey objects have been visited, but the descendants may not have been visited by the collector, or the mutator has changed the object.

White objects are unvisited. At the beginning of a GC cycle all objects are white. At the end of the tracing, all white objects are garbage.

At the end of a collection cycle all black objects are live (or floating garbage) and all white objects are garbage.

6.1.1 Incremental Collection

An incremental collector can be realized in two ways: either by doing part of the work on each allocation of a new object or by running the collector as an independent process. For a single-threaded application, the first method is simpler as less synchronization between the application and the collector is necessary. For a multi-threaded environment there is no advantage by interleaving collector work with object allocation. In this case we need synchronization between the collector work done by one thread and the manipulation of the object graph performed by the other mutator thread. Therefore we will consider a concurrent solution where the collector runs in its own thread or processor. It is even possible to realize the collector as dedicated hardware [45].

6.1.2 Conservatism

Incremental collector algorithms are conservative, meaning that objects becoming unreachable during collection are not collected by the collector — they are floating garbage. Many approaches exist to reduce this conservatism in the general case. However, algorithms that completely avoid floating garbage are impractical. For different conservative collectors the worst-case bounds are all the same (i.e., all objects that become unreachable remain floating garbage). Therefore the level of conservatism is not an issue for real-time collectors.

6.1.3 Safety Critical Java

In Section 5.3 a profile for safety critical Java (SCJ) is defined. SCJ has two interesting properties that may simplify the implementation of a real-time collector. Firstly, the split between initialization and mission phase, and secondly the simplified threading model (which also mandates that self-blocking operations are illegal in mission). During initialization of the application a SCJ virtual machine does not have to meet any real-time constraints (other than possibly a worst case bound on the entire initialization phase). It is perfectly acceptable to use a non-real-time GC implementation during this phase – even a stop-the-world GC. As the change from initialization to mission phase is explicit, it is clear when the virtual machine must initiate real-time collection and which code runs during the mission phase.

Simplifying the threading model has the following advantage, if the collector thread runs at a lower priority than all other threads in the system, it is the case that when it runs *all* other threads have returned from their calls to `run()`. This is trivially true due to the priority

preemptive scheduling discipline[1]. Any thread that has not returned from its `run()` method will preempt the GC until it returns. This has the side effect that the GC will never see a root in the call stack of another thread. Therefore, the usually atomic operation of scanning call stacks can be omitted in the mission phase. We will elaborate on this in Section 6.3.

6.2 Scheduling of the Collector Thread

The collector work can be scheduled either *work* based or *time* based. On a work based scheduling, as performed in [117], an incremental part of the collector work is performed at object allocation. This approach sounds quite natural as threads that allocate more objects have to pay for the collector work. Furthermore, no additional collector thread is necessary. The main issue with this approach is to determine how much work has to be done on each allocation – a non trivial question as collection work consists of different phases. A more subtle question is: Why should a high frequency (and high priority) thread increase its WCET by performing collector work that does not have to be done at that period? Leaving the collector work to a thread with a longer period will allow higher utilization of the system.

On a time based scheduling of the collector work, the collector runs in its own thread. Scheduling this thread as a *normal* real-time thread is quite natural for a hard real-time system. The question is: which priority to assign to the collector thread? The Metronome collector [15] uses the highest priority for the collector. Robertz and Henriksson [97] and Schoeberl [108] argue for the lowest priority. When building hard real-time systems the answer must take scheduling theory into consideration: the priority is assigned according to the period, either rate monotonic [70] or more general deadline monotonic [12]. Assuming that the period of the collector is the longest in the system and the deadline equals the period the collector gets the lowest priority.

In this section we provide an upper bound for the collector period so that the application threads will never run out of memory. The collector period, besides the WCET of the collector, is the single parameter of the collector that can be incorporated in standard schedulability analysis.

The following symbols are used in this section: heap size for a mark-compact collector (H_{MC}) and for a concurrent-copying collector (H_{CC}) containing both semi-spaces, period of the GC thread (T_{GC}), period of a single mutator thread (T_M), period of mutator thread i (T_i) from a set of threads, and memory amount allocated by a single mutator (a) or by mutator i (a_i) from a set of threads.

[1] If we would allow blocking in the application threads, we would also need to block the GC thread.

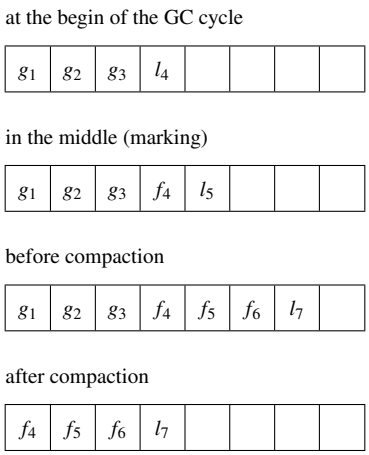

Figure 6.1: Heap usage during a mark-compact collection cycle

We assume that the mutator allocates all memory at the start of the period and the memory becomes garbage at the end. In other words the memory is live for one period. This is the worst-case[2], but very common as we can see in the following code fragment.

```
for (;;) {
    Node n = new Node();
    work(n);
    waitForNextPeriod();
}
```

The object Node is allocated at the start of the period and n will reference it until the next period when a new Node is created and assigned to n. In this example we assume that no reference to Node is stored (inside work) to an object with longer lifetime.

6.2.1 An Example

We start our discussion with a simple example[3] where the collector period is 3 times the mutator period ($T_{GC} = 3T_M$) and a heap size of 8 objects ($8a$). We show the heap during

[2] See Section 6.2.3 for an example where the worst-case lifetime is two periods.

[3] The relation between the heap size and the mutator/collector proportion is an arbitrary value in this example. We will provide the exact values in the next sections.

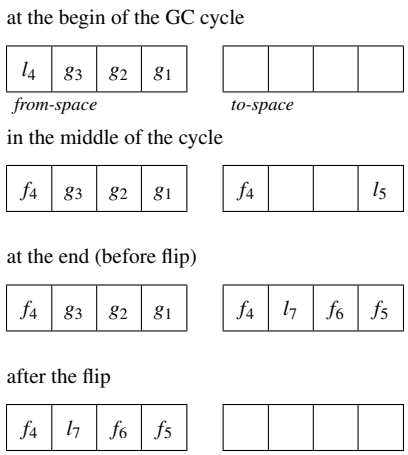

Figure 6.2: Heap usage during a concurrent-copy collection cycle

one GC cycle for a mark-compact and a concurrent-copy collector. The following letters are used to show the status of a memory cell (that contains one object from the mutator in this example) in the heap: g_i is garbage from mutator cycle i, l is the live memory, and f is floating garbage. We assume that all objects that become unreachable during the collection remain floating garbage.

Figure 6.1 shows the changes in the heap during one collection cycle. At the start there are three objects (g_1, g_2, and g_3) left over from the last cycle (floating garbage) which are collected by the current cycle and one live object l_4. During the collection the live objects become unreachable and are now floating garbage (e.g. f_4 in the second sub-figure). At the end of the cycle, just before compacting, we have three garbage cells (g_1-g_3), three floating garbage cells (f_4-f_6) and one live cell l_7. Compaction moves the floating garbage and the live cell to the start of the heap and we end up with four free cells. The floating garbage will become garbage in the next collection cycle and we start over with the first sub-figure with three garbage cells and one live cell.

Figure 6.2 shows one collection cycle of the concurrent-copy collector. We have two memory spaces: the *from-space* and the *to-space*. Again we start the collection cycle with one live cell and three garbage cells left over from the last cycle. Note that the order of the cells is different from the previous example. New cells are allocated in the to-space from the top of the heap, whereas moved cells are allocated from the bottom of the heap. The second sub-figure shows a snapshot of the heap during the collection: formerly live object

l_4 is already floating garbage f_4 and copied into to-space. A new cell l_5 is allocated in the to-space. Before the flip of the two semi-spaces the from-space contains the three garbage cells (g_1-g_3) and the to-space the three floating garbage cells (f_4-f_6) and one live cell l_7. The last sub-figure shows the heap after the flip: The from-space contains the three floating cells which will be garbage cells in the next cycle and the one live cell. The to-space is now empty.

From this example we see that the necessary heap size for a mark-compact collector is similar to the heap size for a copying collector. We also see that the compacting collector has to move more cells (all floating garbage cells and the live cell) than the copying collector (just the one cell that is live at the beginning of the collection).

6.2.2 Minimum Heap Size

In this section we show the memory bounds for a mark-compact collector with a single heap memory and a concurrent-copying collector with the two spaces *from-space* and *to-space*.

The following symbols are used for the rest of the paper: heap size for a mark-compact collector (H_{MC}) and for a concurrent-copying collector (H_{CC}) containing both semi-spaces, period of the GC thread (T_{GC}), period of a single mutator thread (T_M), period of mutator thread i (T_i) from a set of threads, and memory amount allocated by a single mutator (a) or by mutator i (a_i) from a set of threads.

Mark-Compact

For the mark-compact collector, the heap H_{MC} can be divided into allocated memory M and free memory F

$$H_{MC} = M + F = G + \overline{G} + L + F \tag{6.1}$$

where G is garbage at the start of the collector cycle that will be reclaimed by the collector. Objects that become unreachable during the collection cycle and will not be reclaimed are floating garbage $\overline{G}$. These objects will be detected in the next collection cycle. We assume the worst case that all objects that die during the collection cycle will not be detected and therefore are floating garbage. L denotes all live,i.e. reachable, objects. F is the remaining free space.

We have to show that we will never run out of memory during a collection cycle ($F \geq 0$). The amount of allocated memory M has to be less than or equal to the heap size H_{MC}

$$H_{MC} \geq M = G + \overline{G} + L \tag{6.2}$$

In the following proof the superscript n denotes the collection cycle. The subscript letters S and E denote the value at the start and the end of the cycle, respectively.

Lemma 1. *For a collection cycle the amount of allocated memory M is bounded by the maximum live data L_{max} at the start of the collection cycle and two times A_{max}, the maximum data allocated by the mutator during the collection cycle.*

$$M \leq L_{max} + 2A_{max} \tag{6.3}$$

Proof. During a collection cycle G remains constant. All live data that becomes unreachable will be floating garbage. Floating garbage $\overline{G}_E$ at the end of cycle n will be detected (as garbage G) in cycle $n+1$.

$$G^{n+1} = \overline{G}_E^n \tag{6.4}$$

The mutator allocates A memory and transforms part of this memory and part of the live data at the start L_S to floating garbage $\overline{G}_E$ at the end of the cycle. L_E is the data that is still reachable at the end of the cycle.

$$L_S + A = L_E + \overline{G}_E \tag{6.5}$$

with $A \leq A_{max}$ and $L_S \leq L_{max}$. A new collection-cycle start immediately follows the end of the former cycle. Therefore the live data remains unchanged.

$$L_S^{n+1} = L_E^n \tag{6.6}$$

We will show that (6.3) is true for cycle 1. At the start of the first cycle we have no garbage ($G = 0$) and no live data ($L_S = 0$). The heap contains only free memory.

$$M_S^1 = 0 \tag{6.7}$$

During the collection cycle the mutator allocates A^1 memory. Part of this memory will be live at the end and the remaining will be floating garbage.

$$A^1 = L_E^1 + \overline{G}_E^1 \tag{6.8}$$

Therefore at the end of the first cycle

$$\begin{aligned} M_E^1 &= L_E^1 + \overline{G}_E^1 \\ M^1 &= A^1 \end{aligned} \tag{6.9}$$

As $A^1 \leq A_{max}$ (6.3) is fulfilled for cycle 1.

Under the assumption that (6.3) is true for cycle n, we have to show that (6.3) holds for cycle $n+1$.

$$M^{n+1} \leq L_{max} + 2A_{max} \tag{6.10}$$

$$\begin{aligned} M^n &= G^n + \overline{G}_E^n + L_E^n && (6.11)\\ M^{n+1} &= G^{n+1} + \overline{G}_E^{n+1} + L_E^{n+1} && (6.12)\\ &= \overline{G}_E^n + L_S^{n+1} + A^{n+1} && \text{apply (6.4) and (6.5)}\\ &= \overline{G}_E^n + L_E^n + A^{n+1} && \text{apply (6.6)}\\ &= L_S^n + A^n + A^{n+1} && \text{apply (6.5)} \quad (6.13) \end{aligned}$$

As $L_S \leq L_{max}$, $A^n \leq A_{max}$ and $A^{n+1} \leq A_{max}$

$$M^{n+1} \leq L_{max} + 2A_{max} \tag{6.14}$$

□

Concurrent-Copy

In the following section we will show the memory bounds for a concurrent-copying collector with the two spaces *from-space* and *to-space*. We will use the same symbols as in Section 6.2.2 and denote the maximum allocated memory in the from-space as M_{From} and the maximum allocated memory in the to-space as M_{To}.

For a copying-collector the heap H_{CC} is divided in two equal sized spaces H_{From} and H_{To}. The amount of allocated memory M in each semi-space has to be less than or equal to $\frac{H_{CC}}{2}$

$$H_{CC} = H_{From} + H_{To} \geq 2M \tag{6.15}$$

Lemma 2. *For a collection cycle, the amount of allocated memory M in each semi-space is bounded by the maximum live data L_{max} at the start of the collection cycle and A_{max}, the maximum data allocated by the mutator during the collection cycle.*

$$M \leq L_{max} + A_{max} \tag{6.16}$$

Proof. Floating garbage at the end of cycle n will be detectable garbage in cycle $n+1$

$$G^{n+1} = \overline{G}_E^n \tag{6.17}$$

Live data at the end of cycle n will be the live data at the start of cycle $n+1$

$$L_S^{n+1} = L_E^n \tag{6.18}$$

The allocated memory M_{From} in the from-space contains garbage G and the live data at the start L_s.

$$M_{From} = G + L_S \tag{6.19}$$

All new objects are allocated in the to-space. Therefore the memory requirement for the from-space does not change during the collection cycle. All garbage G remains in the from-space and the to-space only contains floating garbage and live data.

$$M_{To} = \overline{G} + L \tag{6.20}$$

At the start of the collection cycle, the to-space is completely empty.

$$M_{To\_S} = 0 \tag{6.21}$$

During the collection cycle all live data is copied into the to-space and new objects are allocated in the to-space.

$$M_{To\_E} = L_S + A \tag{6.22}$$

At the end of the collector cycle, the live data from the start L_S and new allocated data A stays either live at the end L_E or becomes floating garbage $\overline{G}_E$.

$$L_S + A = L_E + \overline{G}_E \tag{6.23}$$

For the first collection cycle there is no garbage ($G = 0$) and no live data at the start ($L_S = 0$), i.e. the from-space is empty ($M_{From}^1 = 0$). The to-space will only contain all allocated data A^1, with $A^1 \leq A_{max}$, and therefore (6.16) is true for cycle 1.

Under the assumption that (6.16) is true for cycle n, we have to show that (6.16) holds for cycle $n+1$.

$$\begin{aligned} M_{From}^{n+1} &\leq L_{max} + A_{max} \\ M_{To}^{n+1} &\leq L_{max} + A_{max} \end{aligned} \tag{6.24}$$

At the start of a collection cycle, the spaces are flipped and the new to-space is cleared.

$$\begin{aligned} H_{From}^{n+1} &\Leftarrow H_{To}^{n} \\ H_{To}^{n+1} &\Leftarrow \emptyset \end{aligned} \tag{6.25}$$

The from-space:

$$M_{From}^{n} = G^{n} + L_{S}^{n} \tag{6.26}$$

$$\begin{aligned} M_{From}^{n+1} &= G^{n+1} + L_{S}^{n+1} && (6.27)\\ &= \overline{G}_{E}^{n} + L_{E}^{n} \\ &= L_{S}^{n} + A^{n} && (6.28) \end{aligned}$$

As $L_S \leq L_{max}$ and $A^n \leq A_{max}$

$$M_{From}^{n+1} \leq L_{max} + A_{max} \tag{6.29}$$

The to-space:

$$M_{To}^{n} = \overline{G}_{E}^{n} + L_{E}^{n} \tag{6.30}$$

$$\begin{aligned} M_{To}^{n+1} &= \overline{G}_{E}^{n+1} + L_{E}^{n+1} && (6.31)\\ &= L_{S}^{n+1} + A^{n+1} \\ &= L_{E}^{n} + A^{n+1} && (6.32) \end{aligned}$$

As $L_E \leq L_{max}$ and $A^{n+1} \leq A_{max}$

$$M_{To}^{n+1} \leq L_{max} + A_{max} \tag{6.33}$$

□

From this result we can see that the dynamic memory consumption for a mark-compact collector is similar to a concurrent-copy collector. This is contrary to the common belief that a copy collector needs the double amount of memory.

We have seen that the double-memory argument against a copying collector does not hold for an incremental real-time collector. We need double the memory of the allocated data during a collection cycle in either case. The advantage of the copying collector over a compacting one is that newly allocated data are placed in the to-space and do not need to be copied. The compacting collector moves all newly created data (that is mostly floating garbage) at the compaction phase.

6.2.3 Garbage Collection Period

GC work is inherently periodic. After finishing one round of collection the GC starts over. The important question is which is the *maximum* period the GC can be run so that the application will never run out of memory. Scheduling the GC at a shorter period does not hurt but decreases utilization.

In the following, we derive the maximum collector period that guarantees that we will not run out of memory. The maximum period T_{GC} of the collector depends on L_{max} and A_{max} for which safe estimates are needed.

We assume that the mutator allocates all memory at the start of the period and the memory becomes garbage at the end. In other words the memory is live for one period. This is the worst case, but very common.

Single Mutator Thread

First we give an upper bound for the collector cycle time for a single mutator thread.

Lemma 3. *For a single mutator thread with period T_M that allocates memory "a" each period, the maximum collector period T_{GC} that guarantees that we will not run out of memory is*

$$T_{GC} \leq T_M \left\lfloor \frac{H_{MC} - a}{2a} \right\rfloor \tag{6.34}$$

$$T_{GC} \leq T_M \left\lfloor \frac{H_{CC} - 2a}{2a} \right\rfloor \tag{6.35}$$

Proof. The maximum live data referenced by a single mutator is the maximum data allocated by the mutator in a single cycle.

$$L_{max} = a \tag{6.36}$$

A single mutator allocates a memory during the period T_M. Therefore the maximum allocation during the collector period T_{GC} is

$$A_{max} = a \left\lceil \frac{T_{GC}}{T_M} \right\rceil \tag{6.37}$$

Using equations (6.2) and (6.3) we get the minimum heap size H_{MC} for a mark-compact collector

$$\begin{aligned} H_{MC} &\geq L_{max} + 2A_{max} \\ H_{MC} &\geq a\left(1 + 2\left\lceil \frac{T_{GC}}{T_M} \right\rceil\right) \end{aligned} \tag{6.38}$$

Equations (6.15) and (6.16) result in the minimum heap size H_{CC}, containing both semi-spaces, for the concurrent-copy collector

$$\begin{aligned} H_{CC} &\geq 2(L_{max} + A_{max}) \\ H_{CC} &\geq 2a\left(1 + \left\lceil \frac{T_{GC}}{T_M} \right\rceil\right) \end{aligned} \tag{6.39}$$

The ceiling function covers the worst-case schedule between the collector thread and the mutator thread. We are interested in the maximum collector period T_{GC} with a given heap size H_{MC} or H_{CC}

$$\left\lceil \frac{T_{GC}}{T_M} \right\rceil \leq \frac{H_{MC} - a}{2a} \tag{6.40}$$

$$\left\lceil \frac{T_{GC}}{T_M} \right\rceil \leq \frac{H_{CC} - 2a}{2a} \tag{6.41}$$

The maximum quotient ($\frac{T_{GC}}{T_M}$) that fulfills (6.40) or (6.41) is an integer n. n is the largest integer that is less than or equal the right side of (6.40) or (6.41). Therefore we get for the mark-compact collector

$$\frac{T_{GC}}{T_M} \leq \left\lfloor \frac{H_{MC} - a}{2a} \right\rfloor \tag{6.42}$$

$$\Rightarrow T_{GC} \leq T_M \left\lfloor \frac{H_{MC} - a}{2a} \right\rfloor \tag{6.43}$$

and for the concurrent-copy collector

$$\frac{T_{GC}}{T_M} \leq \left\lfloor \frac{H_{CC} - 2a}{2a} \right\rfloor \tag{6.44}$$

$$\Rightarrow T_{GC} \leq T_M \left\lfloor \frac{H_{CC} - 2a}{2a} \right\rfloor \tag{6.45}$$

□

Several Mutator Threads

In this section the upper bound of the period for the collector thread is given for n independent mutator threads.

Theorem 1. *For "n" mutator threads with period T_i where each thread allocates a_i memory each period, the maximum collector period T_{GC} that guarantees that we will not run out of memory is*

$$T_{GC} \leq \frac{H_{MC} - 3\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}} \tag{6.46}$$

$$T_{GC} \leq \frac{H_{CC} - 4\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}} \tag{6.47}$$

Proof. For n mutator threads with periods T_i and allocations a_i during each period the values for L_{max} and A_{max} are

$$L_{max} = \sum_{i=1}^{n} a_i \tag{6.48}$$

$$A_{max} = \sum_{i=1}^{n} \left\lceil \frac{T_{GC}}{T_i} \right\rceil a_i \tag{6.49}$$

The ceiling function for A_{max} covers the individual worst cases for the thread schedule and cannot be solved analytically. Therefore we use a conservative estimation A'_{max} instead of A_{max}.

$$A'_{max} = \sum_{i=1}^{n} \left(\frac{T_{GC}}{T_i} + 1 \right) a_i \geq \sum_{i=1}^{n} \left\lceil \frac{T_{GC}}{T_i} \right\rceil a_i \tag{6.50}$$

From (6.2) and (6.3) we get the minimum heap size for a mark-compact collector

$$\begin{aligned} H_{MC} &\geq L_{max} + 2A_{max} \\ &\geq \sum_{i=1}^{n} a_i + 2\sum_{i=1}^{n} \left\lceil \frac{T_{GC}}{T_i} \right\rceil a_i \end{aligned} \tag{6.51}$$

For a given heap size H_{MC} we get the conservative upper bound of the maximum collector

period T_{GC} [4]

$$2A'_{max} \leq H_{MC} - L_{max}$$

$$2\sum_{i=1}^{n}\left(\frac{T_{GC}}{T_i}+1\right)a_i \leq H_{MC} - L_{max} \tag{6.52}$$

$$T_{GC} \leq \frac{H_{MC} - L_{max} - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n}\frac{a_i}{T_i}} \tag{6.53}$$

$$\Rightarrow T_{GC} \leq \frac{H_{MC} - 3\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n}\frac{a_i}{Ti}} \tag{6.54}$$

Equations (6.15) and (6.16) result in the minimum heap size H_{CC}, containing both semi-spaces, for the concurrent-copy collector

$$\begin{aligned} H_{CC} &\geq 2L_{max} + 2A_{max} \\ &\geq 2\sum_{i=1}^{n} a_i + 2\sum_{i=1}^{n}\left\lceil\frac{T_{GC}}{T_i}\right\rceil a_i \end{aligned} \tag{6.55}$$

For a given heap size H_{CC} we get the conservative upper bound of the maximum collector period T_{GC}

$$2A'_{max} \leq H_{CC} - 2L_{max}$$

$$2\sum_{i=1}^{n}\left(\frac{T_{GC}}{T_i}+1\right)a_i \leq H_{CC} - 2L_{max} \tag{6.56}$$

$$T_{GC} \leq \frac{H_{CC} - 2L_{max} - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n}\frac{a_i}{T_i}} \tag{6.57}$$

$$\Rightarrow T_{GC} \leq \frac{H_{CC} - 4\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n}\frac{a_i}{Ti}} \tag{6.58}$$

□

[4] It has to be noted that this is a conservative value for the maximum collector period T_{GC}. The maximum value $T_{GC_{max}}$ that fulfills (6.51) is in the interval

$$\left(\frac{H_{MC} - 3\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n}\frac{a_i}{T_i}}, \frac{H_{MC} - \sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n}\frac{a_i}{T_i}}\right)$$

and can be found by an iterative search.

Producer/Consumer Threads

So far we have only considered threads that do not share objects for communication. This execution model is even more restrictive than the RTSJ scoped memories that can be shared between threads. In this section we discuss how our GC scheduling can be extended to account for threads that share objects.

Object sharing is usually done by a producer and a consumer thread. I.e., one thread allocates the objects and stores references to those objects in a way that they can be accessed by the other thread. This other thread, the consumer, is in charge to *free* those objects after use.

An example of this sharing is a device driver thread that periodically collects data and puts them into a list for further processing. The consumer thread, with a longer period, takes all available data from the list at the start of the period, processes the data, and removes them from the list. During the data processing, new data can be added by the producer. Note that in this case the list will probably never be completely empty. This typical case cannot be implemented by an RTSJ shared scoped memory. There would be no point in the execution where the shared memory will be empty and can get recycled.

The question now is how much data will be alive in the worst case. We denote T_p as the period of the producer thread τ_p and T_c as the period of the consumer thread τ_c. τ_p allocates a_p memory each period. During one period of the consumer τ_c the producer τ_p allocates

$$\left\lceil \frac{T_C}{T_P} \right\rceil a_p$$

memory. The worst case is that τ_c takes over all objects at the start of the period and frees them at the end. Therefore the maximum amount of live data for this producer/consumer combination is

$$2 \left\lceil \frac{T_C}{T_P} \right\rceil a_p$$

To incorporate this extended lifetime of objects we introduce a lifetime factor l_i which is

$$l_i = \begin{cases} 1 & : \text{for normal threads} \\ 2 \left\lceil \frac{T_c}{T_i} \right\rceil & : \text{for producer } \tau_i \text{ and associated consumer } \tau_c \end{cases} \tag{6.59}$$

and extend L_{max} from (6.48) to

$$L_{max} = \sum_{i=1}^{n} a_i l_i \tag{6.60}$$

The maximum amount of memory A_{max} that is allocated during one collection cycle is not changed due to the *freeing* in a different thread and therefore remains unchanged.

The resulting equations for the maximum collector period are

$$T_{GC} \leq \frac{H_{MC} - \sum_{i=1}^{n} a_i l_i - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}} \tag{6.61}$$

and

$$T_{GC} \leq \frac{H_{CC} - 2\sum_{i=1}^{n} a_i l_i - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}} \tag{6.62}$$

Static Objects

The discussion about the collector cycle time assumes that all live data is produced by the periodic application threads and the maximum lifetime is one period. However, in the general case we have also live data that is allocated in the initialization phase of the real-time application and stays alive until the application ends. We incorporate this value by including this static live memory L_s in L_{max}

$$L_{max} = L_s + \sum_{i=1}^{n} a_i l_i \tag{6.63}$$

A mark-compact collector will moves all static data to the bottom of the heap in the first and second[5] collection cycle after the allocation. It does not have to compact these data during the following collection cycles in the mission phase. The concurrent-copy collector would move these static data in each collection cycle. Furthermore, the memory demand for the concurrent copy is increased by the double amount of the static data (compared to the single amount in the mark-compact collector)[6].

As these static objects live *forever*, we propose a similar solution to the immortal memory of the RTSJ. We divide our application into an initialization and a mission phase [93]. All static data is allocated during the initialization phase (where no application threads are scheduled). As part of the transition to the mission phase we perform a *special* collection cycle in a stop-the-world fashion. Live data that exists after this cycle are assumed to be *immortal* data and make up the *immortal* memory area. The remaining memory is used for the garbage collected heap.

[5] A second cycle is necessary as this static data can get intermixed by floating garbage from the first collector cycle.

[6] Or the collector period gets shortened.

This static live data will still be scanned by the collector to find references into the heap but it is not collected. The main differences between our immortal memory and the memory areas of the RTSJ are:

- We do not have to state explicitly which data belongs to the application life-time data. This information is implicitly gathered by the start-mission transition.
- References from the static memory to the garbage collected heap are allowed contrary to the fact in the RTSJ that references to scoped memories, that have to be used for dynamic memory management without a GC, are not allowed from immortal memory.

The second fact greatly simplifies communication between threads. For a typical producer/consumer configuration the container for the shared data is allocated in immortal memory and the actual data in the garbage collected heap.

With this *immortal* memory solution the actual L_{max} only contains allocated memory from the periodic threads.

Object Lifetime

Listing 6.1 shows our Java example of a periodic thread that allocates an object in the main loop and the resulting bytecodes.

There is a time between allocation of `Node` and the assignment to `n` where a reference to the former `Node` (from the former cycle) and the new `Node` (on the operand stack) is live. To handle this issue we can either change the values of L_{max} and A_{max} to accommodate this additional object or change the top-level code of the periodic work to explicitly assign a null-pointer to the local variable `n` as it can be seen in Listing 6.2 from the evaluation section. Programming against the SCJ profile avoids this issues (see Section 6.3).

However, this null pointer assignment is only necessary at the top-level method that invokes `waitForNextPeriod` and is therefore not as complex as explicit freeing of objects. Objects that are created inside `work` in our example do not need to be *freed* in this way as the reference to the object gets *lost* on return from the method.

6.3 SCJ Simplifications

The restrictions of the computational model for safety critical Java allow for optimizations of the GC. We can avoid atomic stack scanning for roots and do not have to deal with exact pointer finding. Static objects, which would belong into immortal memory in the RTSJ, can

```
    public void run() {

        for (;;) {
            Node n = new Node();
            work(n);
            waitForNextPeriod();
        }
    }

public void run();
  Code:
   0:   new #20; //class Node
   3:   dup
   4:   invokespecial   #22; //"<init>":()V
   7:   astore_1
   8:   aload_1
   9:   invokestatic    #26; //work:(Node)V
   12:  aload_0
   13:  invokevirtual   #30; //wFNP:()Z
   16:  pop
   17:  goto    0
```

Listing 6.1: Example periodic thread and the corresponding Java bytecodes

be detected by a special GC cycle at transition to the mission phase. We can treat those objects specially and do not need to collect them during the mission phase. This static memory area is automatically sized.

It has to be noted that our proposal is extending JSR 302. Clearly, adding RTGC to SCJ reduces the importance of scopes and would likely relegate them to the small subset of applications where fast deallocation is crucial. Discussing the interaction between scoped memory and RTGC is beyond the scope of this chapter.

6.3.1 Simple Root Scanning

Thread stack scanning is usually performed atomically. Scanning of the thread stacks with a snapshot-at-beginning write barrier [136] allows optimization of the write barriers to consider only field access (putfield and putstatic) and array access. Reference manipulation in locals and on the operand stack can be ignored for a write barrier. However, this optimization comes at the cost of a possible large blocking time due to the atomicity of stack scanning.

A subtle difference between the RTSJ and the SCJ definition is the possibility to use local variables within run() (see example in Figure 5.4). Although handy for the programmer to preserve state information in locals,[7] GC implementation can greatly benefit from *not* having reference values on the thread stack when the thread suspenses execution.

If the GC thread has the lowest priority and there is no blocking library function that can suspend a real-time thread, then the GC thread will only run when all real-time threads are waiting for their next period – and this waiting is performed after the return from the run() method. In that case the other thread stacks are completely *empty*. We do not need to scan them for roots as the only roots are the references in static (class) variables.

For a real-time GC root scanning has to be exact. With conservative stack scanning, where a primitive value is treated as a pointer, possible large data structures can be kept alive artificially. To implement exact stack scanning we need the information of the stack layout for each possible GC preemption point. For a high-priority GC this point can be at each bytecode (or at each machine instruction for compiling Java). The auxiliary data structure to capture the stack layout (and information which machine register will hold a reference for compiled Java) can get quite large [79] or require additional effort to compute.

With a low-priority GC and the RTSJ model of periodic thread coding with wFNP() the number of GC preemption points is decreased dramatically. When the GC runs all threads will be in wFNP(). Only the stack information for those places in the code have to be

[7] Using multiple wFNP() invocations for local mode changes can also come handy. The author has used this fact heavily in the implementation of a modem/PPP protocol stack.

available. It is also assumed that wFNP() is not invoked very deep in the call hierarchy. Therefore, the stack high will be low and the resulting blocking time short.

As mentioned before, the SCJ style periodic thread model results in an empty stack at GC runtime. As a consequence we do not have to deal with exact stack scanning and need no additional information about the stack layout.

6.3.2 Static Memory

A SCJ copying collector will perform best when all live data is produced by periodic threads and the maximum lifetime of a newly allocated object is one period. However, some data structures allocated in the initialization phase stay alive for the whole application lifetime. In an RTSJ application this data would be allocated in immortal memory. With a real-time GC there is no notion of immortal memory, instead we will use the term *static* memory.[8] Without special treatment, a copying collector will move this data at each GC cycle. Furthermore, the memory demand for the collector increases by the amount of the static data.

As those static objects (mostly) live forever, we propose a solution similar to the immortal memory of the RTSJ. All data allocated during the initialization phase (where no application threads are scheduled) is considered potentially static. As part of the transition to the mission phase we perform a *special* collection cycle in a stop-the-world fashion. Objects that are still alive after this cycle are assumed to live forever and make up the *static* memory area. The remaining memory is used for the garbage collected heap.

The initialization phase and the transition to the mission phase are usually not time critical. However, there are classes of applications for which startup is critical, for example in avionics systems it is essential for the system to come up promptly after a momentary power failure. There are two potential solutions, one could trade initialization time GC against more copy work during the mission phase, or, as an alternative, one could push most of the initialization time work to virtual machine build-time as is done in Ovm [10].

This static memory will still be scanned by the collector to find references into the heap but it is not collected. The main differences between our static memory and the immortal memory of the RTSJ are: Firstly, that the choice of allocation context is implicit. There is no need to specify where an object must be allocated. And secondly, that references from the static memory to the garbage collected heap are allowed. This greatly simplifies communication between threads. For a typical producer/consumer configuration the container for the shared data is allocated in static memory and the actual data in the garbage collected heap.

[8] This is a slight misnomer – as object allocated in static memory are mutable and can die. In the context of the SCJ the latter is expected to be the exception.

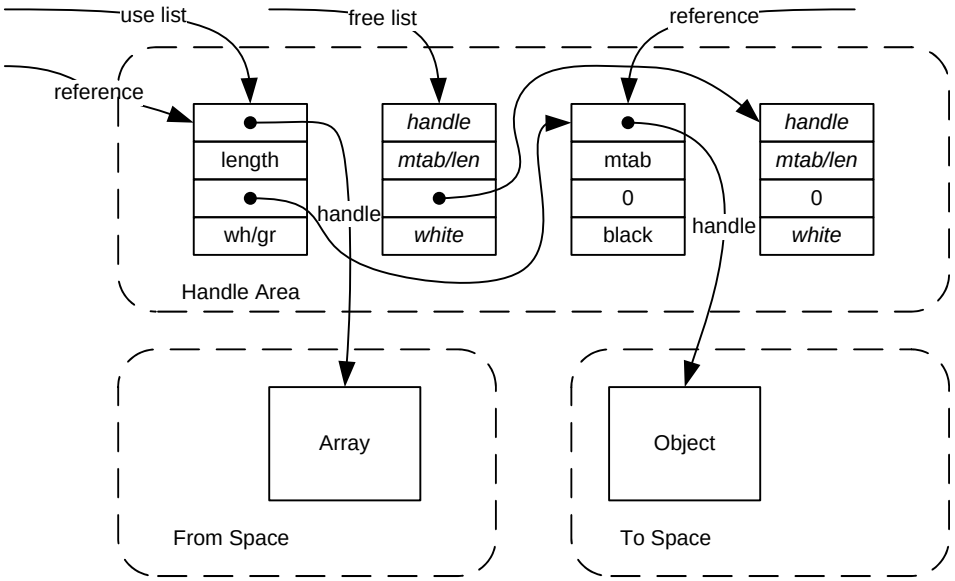

Figure 6.3: Heap layout with the handle area

6.4 Implementation

Our collector is an incremental collector with a snapshot-at-the-beginning write barrier [136]. The GC is based on the copy collector by Cheney [25] and the incremental version by Baker [16]. To avoid the expensive read barrier in Baker's collector we perform all object copies concurrently by the collector. Therefore we call it the *concurrent-copy* collector. We have implemented the concurrent-copy GC on the Java processor JOP [107, 112]. The whole collector, the `new` operation, and the write barriers are implemented in Java (with the help of two native functions for direct memory access). Only the copy operation is optimized by a faster microcode implementation. Although we show the implementation on a Java processor, the GC is not JOP specific and can also be implemented on a conventional processor.

6.4.1 Heap Layout

Figure 6.3 shows a symbolic representation of the heap layout with the handle area and two semi-spaces, *fromspace* and *tospace*. Not shown in this figure is the memory region for runtime constants, such as class information or string constants. This memory region, although logically part of the heap, is neither scanned, nor copied by the GC. This constant area contains its own handles and all references into this area are ignored by the GC.

To simplify object move by the collector, all objects are accessed with one indirection, called the handle. The handle also contains auxiliary object data structures, such as a pointer to the method table or the array length. Instead of Baker's read barrier we have an additional mark stack which is a threaded list within the handle structure. An additional field (as shown in Figure 6.3) in the handle structure is used for a free list and a use list of handles.

The indirection through a handle, although a very light-weight read barrier, is usually still considered as a high overhead. Metronome [15] uses a forwarding pointer as part of the object and performs forwarding *eagerly*. Once the pointer is forwarded, subsequent uses of the reference can be performed on the direct pointer until a GC preemption point. This optimization is performed by the compiler.

We use a hardware based optimization for this indirection [109]. The indirection is unconditionally performed in the memory access unit. Furthermore, null pointer checks and array bounds checks are done in parallel to this indirection.

There are two additional benefits from an explicit handle area instead of a forwarding pointer: (a) access to the method table or array size needs no indirection, and (b) the forwarding pointer and the auxiliary data structures do not need to be copied by the GC.

The fixed handle area is not subject to fragmentation as all handles have the same size

and are recycled at a sweep phase with a simple free list. However, the reserved space has to be sized (or the GC period adapted) for the maximum number of objects that are live or are floating garbage.

6.4.2 The Collector

The collector is scheduled periodically at the lowest priority and within each period it performs the following steps:

Flip An atomic flip exchanges the roles of tospace and fromspace.

Mark roots All static references are pushed onto the mark stack. Only a single push operation needs to be atomic. As the thread stacks are empty we do not need an atomic scan of thread stacks.

Mark and copy An object is popped from the mark stack, all referenced objects, which are still white, are pushed on the mark stack, the object is copied to tospace and the handle pointer is updated.

Sweep handles All handles in the use list are checked if they still point into tospace (black objects) or can be added to the handle free list.

Clear fromspace At the end of the collector work the fromspace that contains only white objects is initialized with zero. Objects allocated in that space (after the next flip) are already initialized and allocation can be performed in constant time.

The longest atomic operation is the copy of an object or array. To reduce blocking time, we plan to implement an array[9] copy and access hardware module within JOP. The hardware can perform copies in an interruptible fashion, and records the copy position on an interrupt. On an array access the hardware knows whether the access should go to the already copied part in the tospace or in the not yet copied part in the fromspace. It has to be noted that splitting larger arrays into smaller chunks, as done in Metronome [15] and in the GC for the JamaicaVM [117], is a software option to reduce the blocking time.

The collector has two modes of operation: one for the initialization phase and one for the mission phase. At the initialization phase it operates in a stop-the-world fashion and gets invoked when a memory request cannot be satisfied. In this mode the collector scans the stack of the single thread conservatively. It has to be noted that each reference points into the handle area and not to an arbitrary position in the heap. This information is considered

[9] Since objects are typically small, this optimization is likely to pay off only for arrays.

by the GC to distinguish pointers from primitives. Therefore the chance to keep an object artificially alive is low.

As part of the mission start one stop-the-world cycle is performed to clean up the heap from garbage generated at initialization. From that point on the GC runs in concurrent mode in its own thread and omits scanning of the thread stacks.

Implementation Code Snippets

This sections shows the important code fragments of the implementation. As can be seen, the implementation is quite short.

Flip involves manipulation of a few pointers and changes the meaning of black (toSpace) and white.

```
synchronized (mutex) {
    useA = !useA;
    if (useA) {
        copyPtr = heapStartA;
        fromSpace = heapStartB;
        toSpace = heapStartA;
    } else {
        copyPtr = heapStartB;
        fromSpace = heapStartA;
        toSpace = heapStartB;
    }
    allocPtr = copyPtr+semi_size;
}
```

Root Marking When the GC runs in concurrent mode only the static reference fields form the root set and are scanned. The stop-the-world mode of the GC also scans all stacks from all threads.

```
int addr = Native.rdMem(addrStaticRefs);
int cnt = Native.rdMem(addrStaticRefs+1);
for (i=0; i<cnt; ++i) {
    push(Native.rdMem(addr+i));
}
```

Push All gray objects are pushed on a gray stack. The gray stack is a list threaded within the handle structure.

```
if (Native.rdMem(ref+OFF_GREY)!=0) {
    return;
}
if (Native.rdMem(ref+OFF_GREY)==0) {
    // pointer to former gray list head
    Native.wrMem(grayList, ref+OFF_GREY);
    grayList = ref;
}
```

Mark and Copy The following code snippet shows the central GC loop.

```
for (;;) {

    // pop one object from the gray list
    synchronized (mutex) {
        ref = grayList;
        if (ref==GREY_END) {
            break;
        }
        grayList = Native.rdMem(ref+OFF_GREY);
        // mark as not in list
        Native.wrMem(0, ref+OFF_GREY);
    }

    // push all childs
    // get pointer to object
    int addr = Native.rdMem(ref);
    int flags = Native.rdMem(ref+OFF_TYPE);
    if (flags==IS_REFARR) {
        // is an array of references
        int size = Native.rdMem(ref+OFF_MTAB_ALEN);
        for (i=0; i<size; ++i) {
            push(Native.rdMem(addr+i));
        }
```

```
        } else if (flags==IS_OBJ){
            // its a plain object
            // get pointer to method table
            flags = Native.rdMem(ref+OFF_MTAB_ALEN);
            // get real flags
            flags = Native.rdMem(flags+MTAB2GC_INFO);
            for (i=0; flags!=0; ++i) {
                if ((flags&1)!=0) {
                    push(Native.rdMem(addr+i));
                }
                flags >>>= 1;
            }
        }

        // now copy it - color it BLACK
        int size = Native.rdMem(ref+OFF_SIZE);
        synchronized (mutex) {
            // update object pointer to the new location
            Native.wrMem(copyPtr, ref+OFF_PTR);
            // set it BLACK
            Native.wrMem(toSpace, ref+OFF_SPACE);
            // copy it
            for (i=0; i<size; ++i) {
                Native.wrMem(Native.rdMem(addr+i), copyPtr+i);
            }
            copyPtr += size;
        }
    }
```

Sweep Handles At the end of the mark and copy phase the handle area is swept to find all unused handles (the one that still point into fromSpace) and add them to the free list.

```
    synchronized (mutex) {
        ref = useList;      // get start of the list
        useList = 0;        // new uselist starts empty
    }
```

```
while (ref!=0) {

    int next = Native.rdMem(ref+OFF_NEXT);
    // a BLACK one
    if (Native.rdMem(ref+OFF_SPACE)==toSpace) {
        // add to used list
        synchronized (mutex) {
            Native.wrMem(useList, ref+OFF_NEXT);
            useList = ref;
        }
    // a WHITE one
    } else {
        // add to free list
        synchronized (mutex) {
            Native.wrMem(freeList, ref+OFF_NEXT);
            freeList = ref;
            Native.wrMem(0, ref+OFF_PTR);
        }
    }
    ref = next;
}
```

Clear Fromspace The last step of the GC clears the fromspace to provide a constant time allocation after the next flip.

```
    for (int i=fromSpace; i<fromSpace+semi_size; ++i) {
        Native.wrMem(0, i);
    }
```

6.4.3 The Mutator

The coordination between the mutator and the collector is performed within the new and newarray bytecodes and within write barriers for JVM bytecodes putfield and putstatic for reference fields, and bytecode aastore.

Allocation

Objects are allocated black (in tospace). In non real-time collectors it is more common to allocate objects white. It is argued [32] that objects die young and the chances are high that the GC never needs to touch them. However, in the worst case no object that is created and becomes garbage during the GC cycle can be reclaimed. Those floating garbage will be reclaimed in the next GC cycle. Therefore, we do not benefit from the white allocation optimization in a real-time GC. Allocating a new object black has the benefit that those objects do not need to be copied. The same argument applies to the chosen write barrier. The following code shows our simple implementation of bytecode new:

```
synchronized (mutex) {
    // we allocate from the upper part
    allocPtr -= size;
    ref = getHandle(size);
    // mark as object
    Native.wrMem(IS_OBJ, ref+OFF_TYPE);
    // pointer to method table in the handle
    Native.wrMem(cons+CLASS_HEADR, ref+OFF_MTAB_ALEN);
}
```

As the old fromspace is cleared by the GC we do not need to initialize the new object and perform new in constant time. The methods Native.rdMem() and Native.wrMem() provide direct access to the main memory. Only those two native methods are necessary for an implementation of a GC in pure Java.

Write Barriers

For a concurrent (incremental) GC some coordination between the collector and the mutator are necessary. The usual solution is a write barrier in the mutator to not foil the collector. According to [132] GC concurrent algorithms can be categorized into:

Snapshot-at-beginning Keep the object graph as it was at the the GC start

- Save to-be-overwritten pointer
- More conservative – not an issue for RTs as worst case counts
- Allocate black
- New objects (e.g. new stack frames) do not need a write barrier

- Optimization: with atomic root scan of the thread stacks no write barrier is necessary for locals and the JVM stack

Incremental update *Help* the GC by doing some collection work in the mutator

- Preserve strong tri-color invariant (no pointer from black to white objects)
- On black to white shade the white object (shade the black is unusual)
- Allocate black (in contrast to [32])
- Needs write barriers for locals and manipulation on the stack
- Less conservative than snapshot-at-beginning

The usual choice is snapshot-at-beginning with atomic root scan of all thread stacks to avoid write barriers on locals. Assume the following assignment of a reference:

```
o.r = ref;
```

There are three references involved that can be manipulated:

- The old value of o.r
- The new value ref
- The object o

The three possible write barriers are:

1. Snapshot-at-beginning/weak tri-color invariant:

   ```
   if (white(o.r)) markGrey(o.r);
   o.r = ref;
   ```

2. Incremental/strong tri-color invariant with push forward

   ```
   if (black(o) && white(ref)) markGrey(ref);
   o.r = ref;
   ```

   This barrier can be optimized to only check if ref is white.

3. Incremental/strong tri-color invariant with push back

   ```
   if (black(o) && white(ref)) markGrey(o);
   o.r = ref;
   ```

We have no stack roots when the collector runs. Therefore we could use the incremental write barrier for object fields only. However, for the worst case all floating garbage will not be found by the GC in the current cycle. Therefore, we use the snapshot-at-begin write barrier in our implementation.

A snapshot-at-begin write barrier synchronizes the mutator with the collector on a reference store into a static field, an object field, or an array. The *to be overwritten* field is pushed on the mark stack when it points to a white object. The following code shows the implementation of putfield for reference fields:

```
private static void f_putfield_ref(int ref, int val,
                                   int index) {

    if (ref==0) {
        throw new NullPointerException();
    }
    synchronized (GC.mutex) {
        // handle indirection
        ref = Native.rdMem(ref);
        // snapshot-at-beginning barrier
        int oldVal = Native.rdMem(ref+index);
        if (oldVal!=0 &&
            Native.rdMem(oldVal+GC.OFF_SPACE)!=GC.toSpace) {

            GC.push(oldVal);
        }

        Native.wrMem(val, ref+index);
    }
}
```

The shown code is part of a special class (com.jopdesign.sys.JVM) where Java bytecodes that are not directly implemented by JOP can be implemented in Java [107]. All putfield bytecodes are replaced by quick variants on class linking. During this step also putfield instructions for references and double-word length fields (double and long) are replaced by special bytecodes. Therefore, the code shows the special bytecode putfield_ref.

6.5 Evaluation

To evaluate our proposed real-time GC we execute a simple test application on JOP and measure heap usage and the release time jitter of high priority threads. The test setup consists of JOP implemented in an Altera Cyclone FPGA clocked at 100 MHz. The main memory is a 1 MB SRAM with an access time of two clock cycles. JOP is configured with a 4 KB method cache (a special form of instruction cache) and a 128 entry stack cache. No additional data cache is used.

6.5.1 Scheduling Experiments

In this section we test an implementation of the concurrent-copy garbage collector on JOP. The tests are intended to get some confidence that the formulas for the collector periods are correct. Furthermore we visualize the actual heap usage of a running system.

The examples are synthetic benchmarks that emulate worst-case execution time (WCET) by executing a busy loop after allocation of the data. The WCET of the collector was measured to be 10.4 ms when executing it with scheduling disabled during one collection cycle for example 1 and 11.2 ms for example 2. We use 11 ms and 12 ms respectively as the WCET of the collector for the following examples[10].

Listing 6.2 shows our worker thread with the busy loop. The data is allocated at the start of the period and freed after the simulated execution. waitForNextPeriod blocks until the next release time for the periodic thread.

For the busy loop to simulate *real* execution time, and not elapsed time, the constant MIN_US has to be less than the time for two context switches, but larger than the execution time of one iteration of the busy loop. In this case only cycles executed by the busy loop are counted for the execution time and interruption due to a higher priority thread is not part of the execution time measurement.

In our example we use a concurrent-copy collector with a heap size (for both semi-spaces) of 100 KB. At startup the JVM allocates about 3.5 KB data. We incorporate[11] these 3.5 KB as static live data L_s.

[10] It has to be noted that measuring execution time is not a safe method to estimate WCET values.

[11] We have not yet implemented the suggested handling of static data to be moved to *immortal* memory at mission start in our prototype.

```
public void run() {

    for (;;) {
        int[] n = new int[cnt];
        // simulate work load
        busy(wcet);
        n = null;
        waitForNextPeriod();
    }
}

final static int MIN_US = 10;

static void busy(int us) {

    int t1, t2, t3;
    int cnt;

    cnt = 0;
    // get the current time in us
    t1 = Native.rd(Const.IO_US_CNT);

    for (;;) {
        t2 = Native.rd(Const.IO_US_CNT);
        t3 = t2-t1;
        t1 = t2;
        if (t3<MIN_US) {
            cnt += t3;
        }
        if (cnt>=us) {
            return;
        }
    }
}
```

Listing 6.2: Example periodic thread with a busy loop

	T_i	C_i	a_i
τ_1	5 ms	1 ms	1 KB
τ_2	10 ms	3 ms	3 KB
τ_{GC}	77 ms	11 ms	

Table 6.1: Thread properties for experiment 1

Independent Threads

The first example consists of two threads with the properties listed in Table 6.1. T_i is the period, C_i the WCET, and a_i the maximum amount of memory allocated each period. Note that the period for the collector thread is also listed in the table although it is a result of the worker thread properties and the heap size.

With the periods T_i and the memory consumption a_i for the two worker threads we calculate the maximum period T_{GC} for the collector thread τ_{GC} by using Theorem 1

$$\begin{aligned} T_{GC} &\leq \frac{H_{CC} - 2\left(L_s + \sum_{i=1}^{n} a_i\right) - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}} \\ &\leq \frac{100 - 2(3.5+4) - 2 \cdot 4}{2\left(\frac{1}{5} + \frac{3}{10}\right)} \text{ms} \\ &\leq 77\text{ms} \end{aligned}$$

The priorities are assigned rate-monotonic [70] and we perform a quick schedulability check with the periods T_i and the WCETs C_i by calculation of the processor utilization U for all three threads

$$\begin{aligned} U &= \sum_{i=1}^{3} \left(\frac{C_i}{T_i}\right) \\ &= \frac{1}{5} + \frac{3}{10} + \frac{11}{77} \\ &= 0.643 \end{aligned}$$

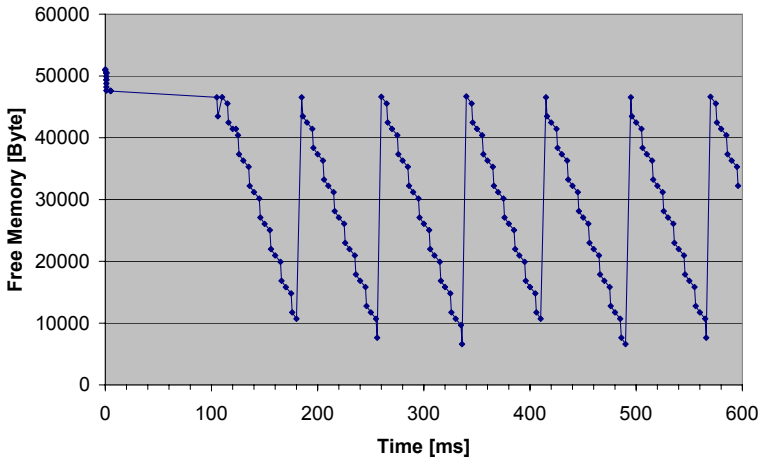

Figure 6.4: Free memory in experiment 1

which is less than the maximum utilization for three tasks

$$\begin{aligned} U_{max} &= m * (2^{\frac{1}{m}} - 1) \\ &= 3 * (2^{\frac{1}{3}} - 1) \\ &\approx 0.78 \end{aligned}$$

In Figure 6.4 the memory trace for this system is shown. The graph shows the free memory in one semi-space (the to-space, which is 50 KB) during the execution of the application. The individual points are recorded with time-stamps at the end of each allocation request.

In the first milliseconds we see allocation requests that are part of the JVM startup (most of it is static data). The change to the mission phase is delayed 100 ms and the first allocation from a periodic thread is at 105 ms. The collector thread also starts at the same time and the first semi-space flip can be seen at 110 ms (after one allocation from each worker thread). We see the 77 ms period of the collector in the jumps in the free memory graph after the flip. The different memory requests of two times 1 KB from thread τ_1 and one time 3 KB from thread τ_2 can be seen every 10 ms.

In this example the heap is used until it is almost full, but we run never out of memory and no thread misses a deadline. From the regular allocation pattern we also see that this

	T_i	C_i	a_i
τ_1	5 ms	0.5 ms	1 KB
τ_2	10 ms	3 ms	3 KB
τ_3	30 ms	2 ms	
τ_{GC}	55 ms	12 ms	

Table 6.2: Thread properties for experiment 2

collector runs concurrently. With a stop-the-world collector we would notice gaps of 10 ms (the measured execution time of the collector) in the graph.

Producer/Consumer Threads

For the second experiment we split our thread τ_1 to a producer thread τ_1 and a consumer thread τ_3 with a period of 30 ms. We assume after the split that the producer's WCET is halved to 500 us. The consumer thread is assumed to be more efficient when working on lager blocks of data than in the former example (C_3=2 ms instead of 6*500 μs). The rest of the setting remains the same (the worker thread τ_2). Table 6.2 shows the thread properties for the second experiment.

As explained in Section 6.2.3 we calculate the lifetime factor l_1 for memory allocated by the producer τ_1 with the corresponding consumer τ_3 with period T_3.

$$l_1 = 2\left\lceil \frac{T_3}{T_1} \right\rceil = 2\left\lceil \frac{30}{5} \right\rceil = 12$$

The maximum collector period T_{GC} is

$$\begin{aligned} T_{GC} &\leq \frac{H_{CC} - 2\left(L_s + \sum_{i=1}^{n} a_i l_i\right) - 2\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{Ti}} \\ &\leq \frac{100 - 2(3.5 + 1 \cdot 12 + 3 + 0) - 2 \cdot 4}{2\left(\frac{1}{5} + \frac{3}{10} + \frac{0}{30}\right)} \text{ms} \\ &\leq 55\text{ms} \end{aligned}$$

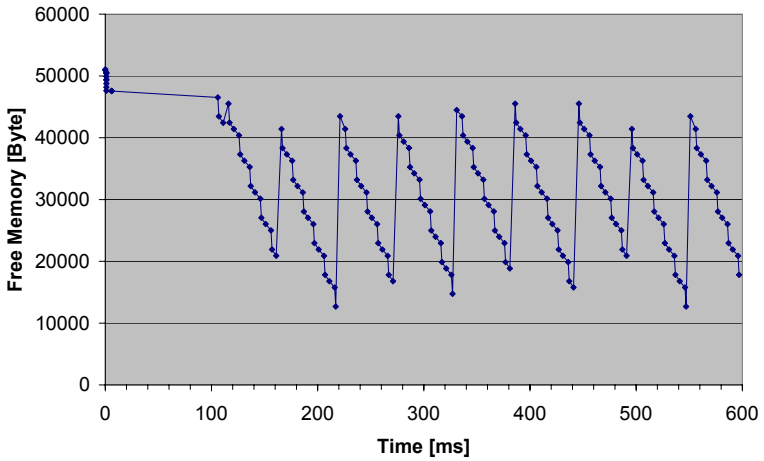

Figure 6.5: Free memory in experiment 2

We check the maximum processor utilization:

$$
\begin{aligned}
U &= \sum_{i=1}^{4} \left(\frac{C_i}{T_i} \right) \\
&= \frac{0.5}{5} + \frac{3}{10} + \frac{2}{30} + \frac{12}{55} \\
&= 0.685 \leq 4 * (2^{\frac{1}{4}} - 1) \approx 0.76
\end{aligned}
$$

In Figure 6.5 the memory trace for the system with one producer, one consumer, and one independent thread is shown. Again, we see the 100 ms delayed mission start after the startup and initialization phase, in this example at about 106 ms. Similar to the former example the first collector cycle performs the flip a few milliseconds after the mission start. We see the shorter collection period of 55 ms. The allocation pattern (two times 1 KB and one time 3 KB per 10 ms) is the same as in the former example as the threads that allocate the memory are still the same.

We have also run this experiment for a longer time than shown in Figure 6.5 to see if we find a point in the execution trace where the remaining free memory is less than the value at 217 ms. The pattern repeats and the observed value at 217 ms is the minimum.

```
public boolean run() {

    int t = Native.rdMem(Const.IO_US_CNT);
    if (!notFirst) {
        expected = t+period;
        notFirst = true;
    } else {
        int diff = t-expected;
        if (diff>max) max = diff;
        if (diff<min) min = diff;
        expected += period;
    }
    work();

    return true;
}
```

Listing 6.3: Measuring release time jitter

6.5.2 Measuring Release Jitter

Our main concern on garbage collection in real-time systems is the blocking time introduced by the GC due to atomic code sections. This blocking time will be seen as release time jitter on the real-time threads. Therefore we want to measure this jitter.

Listing 6.3 shows how we measure the jitter. Method `run()` is the main method of the real-time thread and executed on each periodic release. Within the real-time thread we have no notion about the start time of the thread. As a solution we measure the actual time on the first iteration and use this time as first release time. In each iteration the expected time, stored in the variable `expected`, is incremented by the `period`. In each iteration (except the first one) the actual time is compared with the expected time and the maximum value of the difference is recorded.

As noted before, we have no notion about the *correct* release times. We measure only relative to the first release. When the first release is delayed (due to some startup code or interference with a higher priority thread) we have a positive offset in `expected`. On an exact release in a later iteration the time difference will be negative (in `diff`). Therefore, we

Period	Jitter
200 μs	0 μs
100 μs	0 μs
50 μs	17 μs

Table 6.3: Release jitter for a single thread

also record the minimum value for the difference between the actual time and the expected time. The maximum measured release jitter is the difference between `max` and `min`.

To provide a baseline we measure the release time jitter of a single real-time thread (plus an endless loop in the `main` method as an idle non-real-time background thread). No GC thread is scheduled. The code is similar to the code in in Figure 6.3. A stop condition is inserted that prints out the minimum and maximum time differences measured after 1 million iterations.

Table 6.3 shows the measured jitter for different thread periods. We observed no jitter for periods of 100 μs and longer. At a period of 50 μs the scheduler introduces a considerable amount of jitter. From this measurement we conclude that 100 μs is the practical shortest period we can handle with our system. We will use this period for the high-priority real-time thread in the following measurement with an enabled GC.

6.5.3 Measurements

The test application consisting of three real-time threads (τ_{hf}, τ_p, and τ_c), one logging thread τ_{log}, and the GC thread τ_{gc}. All three real-time threads measure the difference between the expected release time and the actual release time (as shown in Figure 6.3). The minimum and maximum values are recorded and regularly printed to the console by the logging thread τ_{log}. Table 6.4 shows the release parameters for the five threads. Priority is assigned deadline monotonic. Note that the GC thread has a shorter period than the logger thread, but a longer deadline. For our approach to work correctly the GC thread *must* have the lowest priority. Therefore all other threads with a longer period than the GC thread must be assigned a shorter deadline.

Thread τ_{hf} represents a high-frequency thread without dynamic memory allocation. This thread should observe minimal disturbance by the GC thread.

The threads τ_p and τ_c represent a producer/consumer pair that uses dynamically allocated memory for communication. The producer appends the data at a frequency of 1 kHz to a simple list. The consumer thread runs at 100 Hz and processes all currently available data

Thread	Period	Deadline	Priority
τ_{hf}	100 μs	100 μs	5
τ_p	1 ms	1 ms	4
τ_c	10 ms	10 ms	3
τ_{log}	1000 ms	100 ms	2
τ_{gc}	200 ms	200 ms	1

Table 6.4: Thread properties of the test program

Threads	Jitter
τ_{hf}	0 μs
τ_{hf}, τ_{log}	7 μs
τ_{hf}, τ_{log},τ_p,τ_c	14 μs
τ_{hf}, τ_{log},τ_p,τ_c,τ_{gc}	54 μs

Table 6.5: Jitter measured on a 100 MHz processor for the high priority thread in different configurations

in the list and removes them from the list. The consumer will process between 9 and 11 elements (depending on the execution time of the consumer and the thread phasing).

It has to be noted that this simple and common communication pattern cannot be implemented with the scoped memory model of the RTSJ. First, to use a scope for communication, we have to keep the scope alive with a *wedge* thread [86] when data is added by the producer. We would need to notify this wedge thread by the consumer when all data is consumed. However, there is no single instant available where we can *guarantee* that the list is empty. A possible solution for this problem is described in [86] as *handoff* pattern. The pattern is similar to double buffering, but with an explicit copy of the data. The elegance of a simple list as buffer queue between the producer and the consumer is lost.

Thread τ_{log} is not part of the real-time systems simulated application code. Its purpose is to print the minimum and maximum differences between the measured and expected release times (see former section) of threads τ_{hf} and τ_p to the console periodically.

Thread τ_{gc} is a standard periodic real-time thread executing the GC logic. The GC thread period was chosen quite short in that example. A period in the range of seconds would be enough for the memory allocation by τ_p. However, to stress the interference between the GC thread and the application threads we artificially shortened the period.

As a first experiment we run only τ_{hf} and the logging thread τ_{log} to measure jitter introduced by the scheduler. The maximum jitter observed for τ_{hf} is 7 μs – the blocking time of the scheduler.

In the second experiment we run all threads except the GC thread. For the first 4 seconds we measure a maximum jitter of 14 μs for thread τ_{hf}. After those 4 seconds the heap is full and GC is necessary. In that case the GC behaves in a stop-the-world fashion. When a new object request cannot be fulfilled the GC logic is executed in the context of the allocating thread. As the bytecode `new` is itself in an atomic region the application is blocked until the GC finishes. Furthermore, the GC performs a conservative scan of all thread stacks. We measure a release delay of 63 ms for all threads due to the blocking during the full collection cycle. From that measurement we can conclude for the sample application and the available main memory: (a) the measured maximum period of the GC thread is in the range of 4 seconds; (b) the estimated execution time for one GC cycle is 63 ms. It has to be noted that measurement is not a substitution for static timing analysis. Providing WCET estimates for a GC cycle is a challenge for future work.

In our final experiment we enabled all threads. The GC is scheduled periodically at 200 ms as the lowest priority thread – the scenario we argue for. The GC logic is set into the concurrent mode on mission start. In this mode the thread stacks are not scanned for roots. Furthermore when an allocation request cannot be fulfilled the application is stopped. This radical stop is intended for testing. In a more tolerant implementation either an out-of-memory exception can be thrown or the requesting thread has to be blocked, its thread stack scanned and released when the GC has finished its cycle.

We ran the experiment for several hours and recorded the maximum release jitter of the real-time threads. For this test we used slightly different periods (prime numbers) to avoid the regular phasing of the threads. The harmonic relation of the original periods can lead to too optimistic measurements. The applications never ran out of memory. The maximum jitter observed for the high priority task τ_{hf} was 54 μs. The maximum jitter for task τ_p was 108 μs. This higher value on τ_p is expected as the execution interferes with the execution of the higher priority task τ_{hf}.

6.5.4 Discussion

With our measurements we have shown that quite short blocking times are achievable. Scheduling introduces a blocking time of about 7–14 μs and the GC adds another 40 μs resulting in a maximum jitter of the highest priority thread of 54 μs. In our first implementation we performed the object copy in pure Java, resulting in blocking times around 200 μs. To speedup the copy we moved this function to microcode. However, the microcoded *mem-*

cpy still needs 18 cycles per 32-bit word copy. Direct support in hardware can lead to a copy time of 4–5 clock cycles per word.

The maximum blocking time of 54 μs on a 100 MHz processor is less than blocking times reported for other solutions.

Blocking time for Metronome (called pause times in the papers) is reported to be 6 ms [14] on a 500 MHz PowerPC at 50% CPU utilization. Those large blocking times are due to the scheduling of the GC at the highest priority with a polling based yield within the GC thread. A fairer comparison is against the *jitter* of the pause time. In [13] the variation of the pause time is given between 500 μs and 2.4 ms on a 1 Ghz machine. It should be noted that Metronome is a GC intended for mixed real-time systems whereas we aim only for hard real-time systems.

Robertz performed a similar measurement as we did for his thesis [98] with a time-triggered GC on a 350 MHz PowerPC. He measured a maximum jitter of 20 μs ($\pm$10 μs) for a high priority task with a period of 500 μs.

It has to be noted that our experiment is a small one and we need more advanced real-time applications for the evaluation of real-time GC. The problem is that it is hard to find even static based real-time application benchmarks (at least applications written for safety critical Java). Running standard benchmarks that measure average case performance (e.g., SPEC jvm98) is not an option to evaluate a real-time collector.

Although we measured a low blocking time in our experiment we think there is room for improvements. As a first enhancement we will implement a hardware *memcpy* in the memory unit of JOP to reduce the blocking time. However, for very large arrays the resulting blocking time may still be too large. A common solution is to break up arrays into smaller chunks sometimes called Arraylets [15]. However, this comes at a more complex array access with a higher cost.

As we are running our GC on a soft-core Java processor our design space is larger and we can consider implementing a function unit that supports incremental copy. This copy unit will be integrated with the array (field) access unit. On a timer interrupt (for a scheduling decision) the memory copy will also be interrupted and the application thread can run. The copy/access unit remembers the copy position and will redirect the array/field access either to fromspace or to tospace.

Another option is a full hardware implementation of the GC. The proposed algorithm is not very complex and should result in a not too complex hardware. However, this design direction should be carefully evaluated against a way simpler parallel solution: running the GC on one CPU of a chip multiprocessor version of JOP.

6.6 Analysis

To integrate GC into the WCET and scheduling analysis we need to know the worst-case memory consumption including the maximum lifetime of objects and the WCET of the collector itself.

6.6.1 Worst Case Memory Consumption

Similar to the necessary WCET analysis of the tasks that make up the real-time system, a worst case memory allocation analysis of the tasks is necessary. For objects that are not shared between tasks this analysis can be performed by the same methods known from the WCET analysis. We have to find the worst-case program path that allocates the maximum amount of memory.

The analysis of memory consumption by objects that are shared between tasks for communication is more complex as an inter-task analysis is necessary.

6.6.2 WCET of the Collector

For the schedulability analysis the WCET of the collector has to be known. The collector performs following steps[12]:

1. Traverse the live object graph
2. Copy the live data
3. Initialize the free memory

The execution time of the first step depends on the maximum amount of live data and the number of references in each object. The second step depends on the size of the live objects. The last step depends on the size of the memory that gets freed during the collection cycle. For a concurrent-copy collector this time is constant as a complete semi-space gets initialized to zero. It has to be noted that this initialization could also be done at the allocation of the objects (as the LTMemory from the RTSJ implies). However, initialization in the collector is more efficient and the necessary time is easier to predict.

The maximum allocated memory and the type of the allocated objects determine the control flow (the flow facts) of the collector. Therefore, this information has to be to incorporate into WCET analysis of the collector thread.

[12] These steps can be distinct steps as in the mark-compact collector or interleaved as in the concurrent-copy collector.

6.7 Summary

In this chapter we have presented a real-time garbage collector in order to benefit from a more dynamic programming model for real-time applications. Our collector is incremental and scheduled as a normal real-time thread and, according to its deadline, assigned the lowest priority in the system. The restrictions from the SCJ programming model and the low priority result in two advantages: (a) avoidance of stack root scanning and (b) short blocking time of high priority threads. We have implemented the proposed GC on the Java processor JOP. At 100 MHz we measured 40 μs maximum blocking time introduced by the GC thread.

To guarantee that the applications will not run out of memory, the period of the collector thread has to be short enough. We provided the maximum collector periods for a mark-compact collector type and a concurrent-copy collector. We have also shown how a longer lifetime due to object sharing between threads can be incorporated into the collector period analysis.

A critical operation for a concurrent, compacting GC is the atomic copy of large arrays. We consider extending JOP with a copy unit that can be interrupted. This unit is integrated with the array access unit and will redirect the access to either fromspace or tospace depending on the array index and the value of the copy pointer.

6.8 Further Reading

Garbage collection was first introduced for list processing systems (LISP) in the 1960s. The first collectors were *stop-the-world* collectors that are called when a request for a new element can not be fulfilled. The collector, starting from pointers known as the root set, scans the whole graph of reachable objects and marks these objects live. In a second phase the collector *sweeps* the heap and adds unmarked objects to the free list. On the marked objects, which are live, the mark is reset in preparation for the next cycle.

However, this simple sweep results in a fragmented heap which is an issue for objects of different sizes. An extension, called *mark-compact*, moves the objects to compact the heap instead of the sweep. During this compaction all references to the moved objects are updated to point to the new location.

Both collectors need a stack during the marking phase that can grow in the worst-case up to the number of live objects. Cheney [25] presents an elegant way how this mark stack can be avoided. His GC is called *copying-collector* and divides the heap into two spaces: the *to-space* and the *from-space*. Objects are moved from one space to the other as part of the

scan of the object graph.

However, all the described collectors are still stop-the-world collectors. The pause time of up to seconds in large interactive LISP applications triggered the research on incremental collectors that distribute collection work more evenly [121, 32, 16]. These collectors were sometimes called *real-time* although they do not fulfill hard real-time properties that we need today. A good overview of GC techniques can be found in [57] and in the GC survey by Wilson [132].

6.8.1 Related Work

Baker [16] extends Cheneys [25] copying collector for incremental GC. However, it uses an expensive read barrier that moves the object to the to-space as part of the mutator work. Baker proposes the *Treadmill* [17] to avoid copying. However, this collector works only with objects of equal size and still needs an expensive read barrier.

In [100] a garbage-collected memory module is suggested to provide a real-time collector. A worst-case delay time of 1μs is claimed without giving the processor speed.

Despite the title [15] is still not a real-time collector. Non real-time applications are used (SPECjvm98) in the experiments. They propose a collector with constant utilization to meet real-time requirements. However, utilization is *not* a real-time measure per se; it should be schedulability or response time instead. Pause times are in the range of 12ms.

In [82] two collectors based on [32] and [17] are implemented on a multithreaded microcontroller. Higuera suggests in [50] the use of hardware features from picoJava to speed up RTSJ memory region protection and garbage collection.

In [97] the authors provide an upper bound in the GC cycle time as[13]

$$T_{GC} \leq \frac{\frac{H-L_{max}}{2} - \sum_{i=1}^{n} a_i}{\sum_{i=1}^{n} \frac{a_i}{T_i}}$$

Although stated that this bound "is thus not dependent of any particular GC algorithm", the result applies only for single heap GC algorithms (e.g. mark-compact) and not for a copying collector. A value for L_{max} is not given in the paper. If we use our value of $L_{max} = \sum_{i=1}^{n} a_i$ the result is

$$T_{GC} \leq \frac{H - 3\sum_{i=1}^{n} a_i}{2\sum_{i=1}^{n} \frac{a_i}{T_i}}$$

This result is the same as in our finding (see Theorem 1) for the mark-compact collector. No analysis is given how objects with longer lifetime and static objects can be incorporated.

[13] We use our symbols in the equation for easier comparison to our finding.

7 Worst-Case Execution Time

Worst-case execution time (WCET) estimates of tasks are essential for designing and verifying real-time systems. WCET estimates can be obtained either by measurement or static analysis. The problem with using measurements is that the execution times of tasks tend to be sensitive to their inputs. As a rule, measurement does not guarantee safe WCET estimates. Instead, static analysis is necessary for hard real-time systems. Static analysis is usually divided into a number of different phases:

Path analysis generates the control flow graph (a directed graph of basic blocks) of the program and annotates (manual or automatic) loops with bounds.

Low-level analysis determines the execution time of basic blocks obtained by the path analysis. A model of the processor and the pipeline provides the execution time for the instruction sequence.

Global low-level analysis determines the influence of hardware features such as caches on program execution time. This analysis can use information from the path analysis to provide less pessimistic values.

WCET Calculation collapses the control flow graph to provide the final WCET estimate. Alternative paths in the graph are collapsed to a single value (the largest of the alternatives) and loops are collapsed once the loop bound is known.

For the low-level analysis, a good timing model of the processor is needed. The main problem for the low-level analysis is the execution time dependency of instructions in modern processors that are not designed for real-time systems. JOP is designed to be an easy target for WCET analysis. The WCET of each bytecode can be predicted in terms of the number of cycles it requires. There are no dependencies between bytecodes.

Each bytecode is implemented by microcode. We can obtain the WCET of a single bytecode by performing WCET analysis at the microcode level. To prove that there are no time dependencies between bytecodes, we have to show that no processor states are *shared* between different bytecodes.

WCET analysis has to be done at two levels: at the microcode level and at the bytecode level. The microcode WCET analysis is performed only once for a processor configuration and described in the next sections. The result from this microcode analysis is the timing model of the processor. The timing model is the input for the WCET analysis at the bytecode level (i.e. the Java application) as shown in the example in Section 7.5.1 and in the WCET tool description in Section 7.5.2.

It has to be noted that we cannot provide WCET values for the other Java systems from Section 10.3, e.g. the aJile Java processor, as there is no information available on their instruction timing.

7.1 Microcode Path Analysis

To obtain the WCET values for the individual bytecodes we perform the path analysis at the microcode level. First, we have to ensure that a number of restrictions (from [91]) of the code are fulfilled:

- Programs must not contain unbounded recursion. This property is satisfied by the fact that there exists no call instruction in microcode.
- Function pointers and computed gotos complicate the path analysis and should therefore be avoided. Only simple conditional branches are available at the microcode level.
- The upper bound of each loop has to be known. This is the only point that has to be verified by inspection of the microcode.

To detect loops in the microcode we have to find all backward branches (e.g. with a negative branch offset)[1]. The branch offsets can be found in a VHDL file (offtbl.vhd) that is generated during microcode assembly. In the current implementation of the JVM there are ten different negative offsets. However, not each offset represents a loop. Most of these branches are used to share common code. Three branches are found in the initialization code of the JVM. They are not part of a bytecode implementation and can be ignored. The only loop that is found in a regular bytecode is in the implementation of imul to perform a fixed delay. The iteration count for this loop is constant.

[1] The loop branch can be a forward branch. However, the basic blocks of the loop contain at least one backward branch. Therefore we can identify all loops by searching for backward branches only.

A few bytecodes are implemented in Java[2] and can be analyzed in the same way as application code. The bytecodes idiv and irem contain a constant loop. The bytecodes new and anewarray contain loops to initialize (with zero values) new objects or arrays. The loop is bound by the size of the object or array. The bytecode lookupswitch[3] performs a linear search through a table of branch offsets. The WCET depends on the table size that can be found as part of the instruction.

As the microcode sequences are very short, the calculation of the control flow graph for each bytecode is done manually.

7.2 Microcode Low-level Analysis

To calculate the execution time of basic blocks in the microcode, we need to establish the timing of microcode instructions on JOP. All microcode instructions except wait execute in a single cycle, reducing the low-level analysis to a case of merely counting the instructions.

The wait instruction is used to stall the processor and wait for the memory subsystem to finish a memory transaction. The execution time of the wait instruction depends on the memory system and, if the memory system is predictable, has a known WCET. A main memory consisting of SRAM chips can provide this predictability and this solution is therefore advised. The predictable handling of DMA, which is used for the instruction cache fill, is explained in [105]. The wait instruction is the only way to stall the processor. Hardware events, such as interrupts (see [102]), do not stall the processor.

Microcode is stored in on-chip memory with single cycle access. Each microcode instruction is a single word long and there is no need for either caching or prefetching at this stage. We can therefore omit performing a low-level analysis. No pipeline analysis [36], with its possible unbound timing effects, is necessary.

7.3 Bytecode Independency

We have seen that all microcode instructions except wait take one cycle to execute and are therefore independent of other instructions. This property directly translates to independency of bytecode instructions.

The wait microcode instruction provides a convenient way to hide memory access time. A memory read or write can be triggered in microcode and the processor can continue

[2] The implementation can be found in the class com.jopdesign.sys.JVM.

[3] lookupswitch is one way of implementing the Java switch statement. The other bytecode, tableswitch, uses an index in the table of branch offsets and has therefore a constant execution time.

with microcode instructions. When the data from a memory read is needed, the processor explicitly waits, with the wait instruction, until it becomes available.

For a memory store, this wait could be deferred until the memory system is used next (similar to a write buffer). It is possible to initiate the store in a bytecode such as putfield and continue with the execution of the next bytecode, even when the store has not been completed. In this case, we introduce a dependency over bytecode boundaries, as the state of the memory system is *shared*. To avoid these dependencies that are difficult to analyze, each bytecode that accesses memory waits (preferably at the end of the microcode sequence) for the completion of the memory request.

Furthermore, if we would not wait at the end of the store operation we would have to insert an additional wait at the start of every read operation. Since read operations are more frequent than write operations (15% vs. 2.5%, see [107]), the performance gain from the hidden memory store is lost.

7.4 WCET of Bytecodes

The control flow of the individual bytecodes together with the basic block length (that directly corresponds with the execution time) and the time for memory access result in the WCET (and BCET) values of the bytecodes. These values can be found in Appendix D.

Simple bytecode instructions are executed by either one microinstruction or a short sequence of microinstructions. The execution time in cycles equals the number of microinstructions executed. As the stack is on-chip it can be accessed in a single cycle. We do not need to incorporate the main memory timing into the instruction timing. Table 7.1 shows examples of the execution time of such bytecodes.

Object oriented instructions, array access, and invoke instructions access the main memory. Therefore we have to model the memory access time. We assume a simple SRAM with a constant access time. Access time that exceeds a single cycle includes additional wait states (r_{ws} for a memory read and w_{ws} for a memory write). The following example gives the execution time for getfield, the read access of an object field:

$$t_{getfield} = 10 + 2r_{ws}$$

However, the memory subsystem performs read and write parallel to the execution of microcode. Therefore, some access cycles can be hidden. The following example gives the exact execution time of bytecode ldc2_w in clock cycles:

$$t_{ldc2\_w} = 17 + \begin{cases} r_{ws} - 2 & : \quad r_{ws} > 2 \\ 0 & : \quad r_{ws} \leq 2 \end{cases} + \begin{cases} r_{ws} - 1 & : \quad r_{ws} > 1 \\ 0 & : \quad r_{ws} \leq 1 \end{cases}$$

Opcode	Instruction	Cycles	Funtion
3	iconst_0	1	load constant 0 on TOS
4	iconst_1	1	load constant 1 on TOS
16	bipush	2	load a byte constant on TOS
17	sipush	3	load a short constant on TOS
21	iload	2	load a local on TOS
26	iload_0	1	load local 0 on TOS
27	iload_1	1	load local 1 on TOS
54	istore	2	store the TOS in a local
59	istore_0	1	store the TOS in local 0
60	istore_1	1	store the TOS in local 1
89	dup	1	duplicate TOS
90	dup_x1	5	complex stack manipulation
96	iadd	1	integer addition
153	ifeq	4	conditional branch

Table 7.1: Execution time of simple bytecodes in cycles

Thus, for a memory with two-cycle access time ($r_{ws} = 1$), as used for a 100 MHz version of JOP with a 15 ns SRAM, the wait state is completely hidden by microcode instructions for this bytecode.

Memory access time also determines the cache load time on a miss. For the current implementation, the cache load time is calculated as follows: the wait state c_{ws} for a single word cache load is:

$$c_{ws} = \begin{cases} r_{ws} - 1 & : \quad r_{ws} > 1 \\ 0 & : \quad r_{ws} \leq 1 \end{cases}$$

On a method invoke or return the bytecode has to be loaded into the cache on a cache miss. The load time l is:

$$l = \begin{cases} 6 + (n+1)(2 + c_{ws}) & : \quad \text{cache miss} \\ 4 & : \quad \text{cach hit} \end{cases}$$

where n is the length of the method in number of 32-bit words. For short methods the load time of the method on a cache miss, or part of it, is hidden by microcode execution.

As an example the exact execution time for the bytecode invokestatic is:

$$
\begin{aligned}
t &= 74 + r + \begin{cases} r_{ws} - 3 & : \quad r_{ws} > 3 \\ 0 & : \quad r_{ws} \leq 3 \end{cases} + \begin{cases} r_{ws} - 2 & : \quad r_{ws} > 2 \\ 4 & : \quad r_{ws} \leq 2 \end{cases} \\
&+ \begin{cases} l - 37 & : \quad l > 37 \\ 0 & : \quad l \leq 37 \end{cases}
\end{aligned}
$$

For invokestatic a cache load time l of up to 37 cycles is completely hidden. For the example SRAM timing the cache load of methods up to 36 bytes long is hidden. The WCET analysis tool, as described in the next section, knows the length of the invoked method and can therefore calculate the time for the invoke instruction cycle accurate.

7.5 WCET Analysis of the Java Application

We conclude this section with a worst-case analysis (now at the bytecode level) of Java applications. First we provide manual analysis on a simple example and then a brief description of the automation through a WCET analyzer tool.

7.5.1 An Example

In this section we perform manually a worst and best case analysis of a classic example: the Bubble Sort algorithm. The values calculated are compared with the measurements of the execution time on JOP on all permutations of the input data. Listing 7.1 shows the test program in Java. The algorithm contains two nested loops and one condition. We use an array of five elements to perform the measurements for all permutations (i.e. $5! = 120$) of the input data. The number of iterations of the outer loop is one less than the array size: $c_1 = N - 1$, in this case four. The inner loop is executed $c_2 = \sum_{i=1}^{c_1} i = c_1(c_1+1)/2$ times, i.e. ten times in our example.

The annotated control flow graph (CFG) of the example is shown in Figure 7.1. The edges contain labels showing how often the path between two nodes is taken. We can identify the outer loop, containing the blocks B2, B3, B4 and B8. The inner loop consists of blocks B4, B5, B6 and B7. Block B6 is executed when the condition of the if statement is true. The path from B5 to B7 is the only path that depends on the input data.

In Table 7.2 the basic blocks with the start address (Addr.) and their execution time (Cycles) in clock cycles and the worst and best case execution frequency (Count) is given. The values in the forth and sixth columns (Count) of Table 7.2 are derived from the CFG and show how often the basic blocks are executed in the worst and best cases. The WCET and

```
final static int N = 5;

static void sort(int[] a) {

    int i, j, v1, v2;
    // loop count = N-1
    for (i=N-1; i>0; --i) {
        // loop count = (N-1)*N/2
        for (j=1; j<=i; ++j) {
            v1 = a[j-1];
            v2 = a[j];
            if (v1 > v2) {
                a[j] = v1;
                a[j-1] = v2;
            }
        }
    }
}
```

Listing 7.1: Bubble Sort test program for the WCET analysis

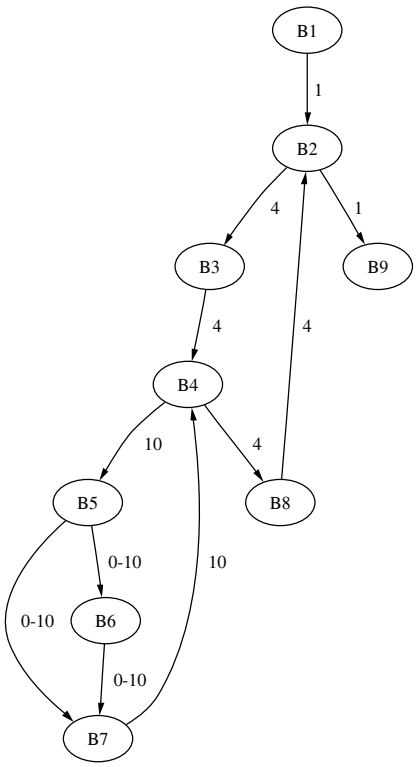

Figure 7.1: The control flow graph of the Bubble Sort example

			WCET		BCET	
Block	Addr.	Cycles	Count	Total	Count	Total
B1	0:	2	1	2	1	2
B2	2:	5	5	25	5	25
B3	6:	2	4	8	4	8
B4	8:	6	14	84	14	84
B5	13:	74	10	740	10	740
B6	30:	73	10	730	0	0
B7	41:	15	10	150	10	150
B8	47:	15	4	60	4	60
B9	53:		1		1	
Execution time calculated				1799		1069
Execution time measured				1799		1069

Table 7.2: WCET and BCET in clock cycles of the basic blocks

BCET value for each block is calculated by multiplying the clock cycles by the execution frequency. The overall WCET and BCET values are calculated by summing the values of the individual blocks B1 to B8. The last block (B9) is omitted, as the measurement does not contain the return statement.

The execution time of the program is measured using the cycle counter in JOP. The current time is taken at both the entry of the method and at the end, resulting in a measurement spanning from block B1 to the beginning of block B9. The last statement, the `return`, is not part of the measurement. The difference between these two values (less the additional 8 cycles introduced by the measurement itself) is given as the execution time in clock cycles (the last row in Table 7.2). The measured WCET and BCET values are exactly the same as the calculated values.

In Figure 7.2, the measured execution times for all 120 permutations of the input data are shown. The vertical axis shows the execution time in clock cycles and the horizontal axis the number of the test run. The first input sample is an already sorted array and results in the lowest execution time. The last sample is the worst-case value resulting from the reversely ordered input data. We can also see the 11 different execution times that result from executing basic block B6 (which performs the element exchange and takes 73 clock cycles) between 0 and 10 times.

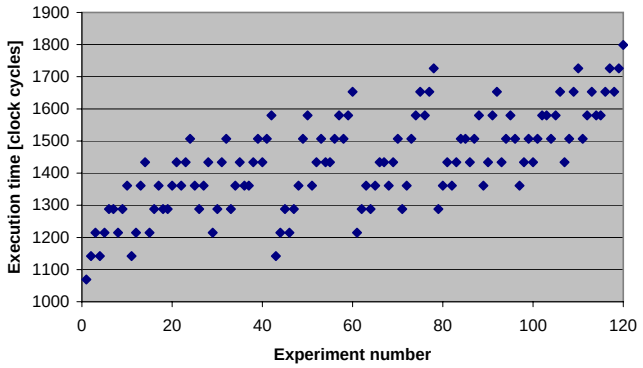

Figure 7.2: Execution time in clock cycles of the Bubble Sort program for all 120 permutations of the input data

7.5.2 WCET Analyzer

In [114] we have presented a static WCET analysis tool for Java. During the high-level analysis the relevant information is extracted from the class files. The control flow graph (CFG) of the basic blocks[4] is extracted from the bytecodes. Annotations for the loop counts are extracted from comments in the source. Furthermore, the class hierarchy is examined to find all possible targets for a method invoke.

The tool performs the low-level analysis at the bytecode level. The behavior of the method cache is integrated for a simpler form (a two block cache). The well known execution times of the different bytecodes (see Section 7.4) simplify this part of the WCET analysis, which is usually the most complex one, to a great extent. As there are no pipeline dependencies, the calculation of the execution time for a basic block is merely just adding the individual cycles for each instruction.

The actual calculation of the WCET is transformed to an integer linear programming problem, a well known technique for WCET analysis [92, 66]. We performed the WCET analysis on several benchmarks (see Table 7.3). We also *measured* the WCET values for the benchmarks. It has to be noted that we actually cannot measure the real WCET. If we could measure it, we would not need to perform the WCET analysis at all. The measurement gives us an idea of the pessimism of the analyzed WCET. The benchmarks Lift and Kfl are real-world examples that are in industrial use. Kfl and Udplp are also part of an embedded

[4] A basic block is a sequence of instructions without any jumps or jump targets within this sequence.

Program	Description	LOC
crc	CRC calculation for short messages	8
robot	A simple line follower robot	111
Lift	A lift controler	635
Kfl	*Kippfahrleitung* application	1366
Udplp	UDP/IP benchmark	1297

Table 7.3: WCET benchmark examples

Program	Measured (cycle)	Estimated (cycle)	Pessimism (ratio)
crc	1552	1620	1.04
robot	736	775	1.05
Lift	7214	11249	1.56
Kfl	13334	28763	2.16
Udplp	11823	219569	18.57

Table 7.4: Measured and estimated WCETs with results in clock cycles

Java benchmark suite that is used in Section 10.3.

Table 7.4 shows the measured execution time and the analyzed WCET. The last column gives an idea of the pessimism of the WCET analysis. For very simple programs, such as crc and robot, the pessimism is quite low. For the Lift example it is in an acceptable range. The difference between the measurement and the analysis in the Kfl example results from the fact that our measurement does not cover the WCET path. The large conservatism in Udplp results from the loop bound in the IP and UDP checksum calculation. It is set for a 1500 byte packet buffer, but the payload in the benchmark is only 8 bytes. The last two examples also show the issue when a real-time application is developed without a WCET analysis tool available.

The WCET analysis tool, with the help of loop annotations, provides WCET values for the schedulability analysis. Besides the calculation of the WCET, the tool provides user feedback by generating bytecode listings with timing information and a graphical representation of the CFG with timing and frequency information. This representation of the WCET path through the code can guide the developer to write WCET aware real-time code.

7.6 Discussion

The Bubble Sort example and experiments with the WCET analyzer tool have demonstrated that we have achieved our goal: JOP is a simple target for the WCET analysis. Most bytecodes have a single execution time (WCET = BCET), and the WCET of a task (the analysis at the bytecode level) depends only on the control flow. No pipeline or data dependencies complicate the low-level part of the WCET analysis.

The same analysis is not possible for other Java processors. Either the information on the bytecode execution time is missing[5] or some processor features (e.g., the high variability of the latency for a trap in picoJava) would result in very conservative WCET estimates. Another example that prohibits exact analysis is the mechanism to automatically fill and spill the stack cache in picoJava. The time when the memory (cache) is occupied by this spill/fill action depends on a long instruction history. Also the fill level of the 16-byte-deep prefetch buffer, which is needed for instruction folding, depends on the execution history. All these automatic buffering features have to be modeled quite conservatively. A pragmatic solution is to assume empty buffers at the start of a basic block. As basic blocks are quite short, most of the buffering/prefetching does not help to lower the WCET.

Only for the Cjip processor is the execution time well documented [54]. However, as seen in Section 10.3.2, the *measured* execution time of some bytecodes is *higher* than the documented values. Therefore the documentation is not complete to provide a safe processor model of the Cjip for the WCET analysis.

[5] We tried hard to get this information for the aJile processor.

8 The SimpCon Interconnect

SimpCon [110] is the main interconnection interface used for JOP. The IO modules and the main memory are connected via this standard. In the following chapter an introduction to SimpCon is presented.

The VHDL files in vhdl/scio are SimpCon IO components (e.g. sc_uart.vhd is a simple UART) and SimpCon IO configurations. The IO configurations define the IO devices and the address mapping for a JOP system. All those configurations start with scio_. The IO components start with sc_. Configuration scio_min contains the minimal IO components for a JOP system: the system module sc_sys.vhd and a UART sc_uart.vhd for program download and basic communication (System.in and System.out).

The system module sc_sys.vhd contains the clock counter, the μs counter, timer interrupt, the SW interrupt, exception interrupts, the watchdog port, and the connection to the multiprocessor synchronization unit (cmpsync.vhd).

In directory vhdl/memory the memory controller mem_sc.vhd is a SimpCon master that can be connected to various SRAM memory controllers sc_sram*.vhd. Other memory controller (e.g. the free Altera SDRAM interface) can be connected via SimpCon bridges to Avalon, Wishbone, and AHB slave (available in vhdl/simpcon).

8.1 Introduction

The intention of the following SoC interconnect standard is to be simple and efficient with respect to implementation resources and transaction latency.

SimpCon is a fully synchronous standard for on-chip interconnections. It is a point-to-point connection between a master and a slave. The master starts either a read or write transaction. Master commands are single cycle to free the master to continue on internal operations during an outstanding transaction. The slave has to register the address when needed for more than one cycle. The slave also registers the data on a read and provides it to the master for more than a single cycle. This property allows the master to delay the actual read if it is busy with internal operations.

The slave signals the end of the transaction through a novel *ready counter* to provide

an early notification. This early notification simplifies the integration of peripherals into pipelined masters.

Slaves can also provide several levels of pipelining. This feature is announced by two static output ports (one for read and one write pipeline levels).

Off-chip connections (e.g. main memory) are device specific and need a slave to perform the translation. Peripheral interrupts are not covered by this specification.

8.1.1 Features

- Master/slave point-to-point connection
- Synchronous operation
- Read and write transactions
- Early pipeline release for the master
- Pipelined transactions
- Open-source specification
- Low implementation overheads

8.1.2 Basic Read Transaction

Figure 8.1 shows a basic read transaction for a slave with one cycle latency. The acknowledge signals are omitted from the figure. In the first cycle, the address phase, the `rd` signals the slave to start the read transaction. The address is registered by the slave. During the following cycle, the read phase[1], the slave performs the read and registers the data. Due to the register in the slave, the data is available in the third cycle, the result phase. To simplify the master, `rd_data` stays valid until the next read request response. It is therefore possible for a master to issue a pre-fetch command early. When the pre-fetched data arrives too early it is still valid when the master actually wants to read it.

[1] It has to be noted that the read phase can be longer for devices with a high latency. For simple on-chip IO devices the read phase can be omitted completely (0 cycles). In that case `rdy_cnt` will be zero in the cycle following the address phase.

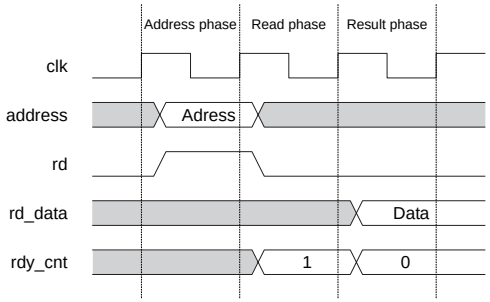

Figure 8.1: Basic read transaction

8.1.3 Basic Write Transaction

A write transaction consists of a single cycle address/command phase started by assertion of `wr` where the address and the write data are valid. `address` and `wr_data` are usually registered by the slave. The end of the write cycle is signalled to the master by the slave with `rdy_cnt`. See Section 8.3 and an example in Figure 8.3.

8.2 SimpCon Signals

This sections defines the signals used by the SimpCon connection. Some of the signals are optional and may not be present on a peripheral device.

All signals are a single direction point-to-point connection between a master and a slave. The signal details are described by the device that drives the signal. Table 8.1 lists the signals that define the SimpCon interface. The column Direction indicates whether the signal is driven by the master or the slave.

8.2.1 Master Signal Details

This section describes the signals that are driven by the master to initiate a transaction.

address

Master addresses represent word addresses as offsets in the slave's address range. `address` is valid a single cycle either with `rd` for a read transaction or with `wr` and `wr_data` for a write transaction.

Signal	Width	Direction	Required	Description
address	1–32	Master	No	Address lines from the master to the slave port
wr_data	32	Master	No	Data lines from the master to the slave port
rd	1	Master	No	Start of a read transaction
wr	1	Master	No	Start of a write transaction
rd_data	32	Slave	No	Data lines from the slave to the master port
rdy_cnt	2	Slave	Yes	Transaction end signalling
rd_pipeline_level	2	Slave	No	Maximum pipeline level for read transactions
wr_pipeline_level	2	Slave	No	Maximum pipeline level for write transactions

Table 8.1: SimpCon port signals

The number of bits for address depends on the slave's address range. For a single port slave, address can be omitted.

wr_data

The wr_data signals carry the data for a write transaction. It is valid for a single cycle together with address and wr. The signal is typically 32 bits wide. Slaves can ignore upper bits when the slave port is less than 32 bits.

rd

The rd signal is asserted for a single clock cycle to start a read transaction. address has to be valid in the same cycle.

wr

The wr signal is asserted for a single clock cycle to start a write transaction. address and wr_data have to be valid in the same cycle.

sel_byte

The sel_byte signal is reserved for future versions of the SimpCon specification to add individual byte enables.

8.2.2 Slave Signal Details

This section describes the signals that are driven by the slave as a response to transactions initiated by the master.

rd_data

The rd_data signals carry the result of a read transaction. The data is valid when rdy_cnt reaches 0 and stays valid until a new read result is available. The signal is typically 32 bits wide. Slaves that provide less than 32 bits should pad the upper bits with 0.

rdy_cnt

The rdy_cnt signal provides the number of cycles until the pending transaction will finish. A 0 means that either read data is available or a write transaction has been finished. Values of 1 and 2 mean the transaction will finish in at least 1 or 2 cycles. The maximum value is 3 and means the transaction will finish in 3 or *more* cycles. Note that not all values have to be used in a transaction. Each monotonic sequence of rdy_cnt values is legal.

rd_pipeline_level

The static rd_pipeline_level provides the master with the read pipeline level of the slave. The signal has to be constant to enable the synthesizer to optimize the pipeline level dependent state machine in the master.

wr_pipeline_level

The static wr_pipeline_level provides the master with the write pipeline level of the slave. The signal has to be constant to enable the synthesizer to optimize the pipeline level dependent state machine in the master.

8.3 Slave Acknowledge

Flow control between the slave and the master is usually done by a single signal in the form of *wait* or *acknowledge*. The `ack` signal, e.g. in the Wishbone specification, is set when the data is available or the write operation has finished. However, for a pipelined master it can be of interest to know it *earlier* when a transaction will finish.

For many slaves, e.g. an SRAM interface with fixed wait states, this information is available inside the slave. In the SimpCon interface, this information is communicated to the master through the two bit ready counter (`rdy_cnt`). `rdy_cnt` signals the number of cycles until the read data will be available or the write transaction will be finished. Value 0 is equivalent to an *ack* signal and 1, 2, and 3 are equivalent to a wait request with the distinction that the master knows how long the wait request will last.

To avoid too many signals at the interconnect, `rdy_cnt` has a width of two bits. Therefore, the maximum value of 3 has the special meaning that the transaction will finish in 3 or *more* cycles. As a result the master can only use the values 0, 1, and 2 to release actions in its pipeline. If necessary, an extension for a longer pipeline is straightforward with a larger `rdy_cnt`[2].

Idle slaves will keep the former value of 0 for `rdy_cnt`. Slaves that do not know in advance how many wait states are needed for the transaction can produce sequences that omit any of the numbers 3, 2, and 1. A simple slave can hold `rdy_cnt` on 3 until the data is available and set it than directly to 0. The master has to handle those situations. Practically, this reduces the possibilities of pipelining and therefore the performance of the interconnect. The master will read the data later, which is not an issue as the data stays valid.

Figure 8.2 shows an example of a slave that needs three cycles for the read to be processed. In cycle 1, the read command and the address are set by the master. The slave registers the address and sets `rdy_cnt` to 3 in cycle 2. The read takes three cycles (2–4) during which `rdy_cnt` gets decremented. In cycle 4 the data is available inside the slave and gets registered. It is available in cycle 5 for the master and `rdy_cnt` is finally 0. Both, the `rd_data` and `rdy_cnt` will keep their value until a new transaction is requested.

Figure 8.3 shows an example of a slave that needs three cycles for the write to be processed. The address, the data to be written, and the write command are valid during cycle 1. The slave registers the address and write data during cycle 1 and performs the write operation during cycles 2–4. The `rdy_cnt` is decremented and a non-pipelined slave can accept a new command after cycle 4.

[2]The maximum value of the ready counter is relevant for the early restart of a waiting master. A longer latency from the slave e.g., for DDR SDRAM, will map to the maximum value of the counter for the first cycles.

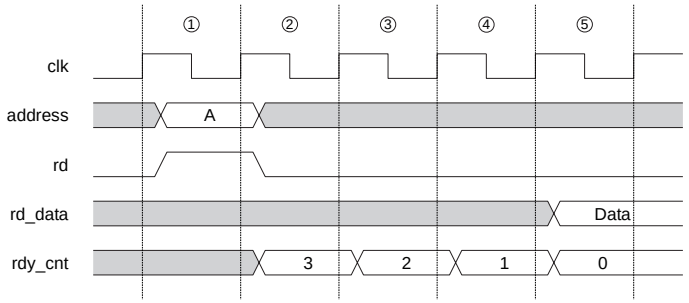

Figure 8.2: Read transaction with wait states

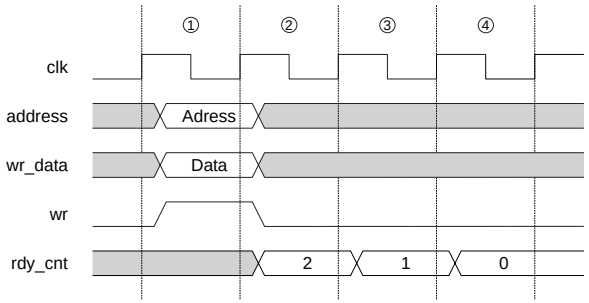

Figure 8.3: Write transaction with wait states

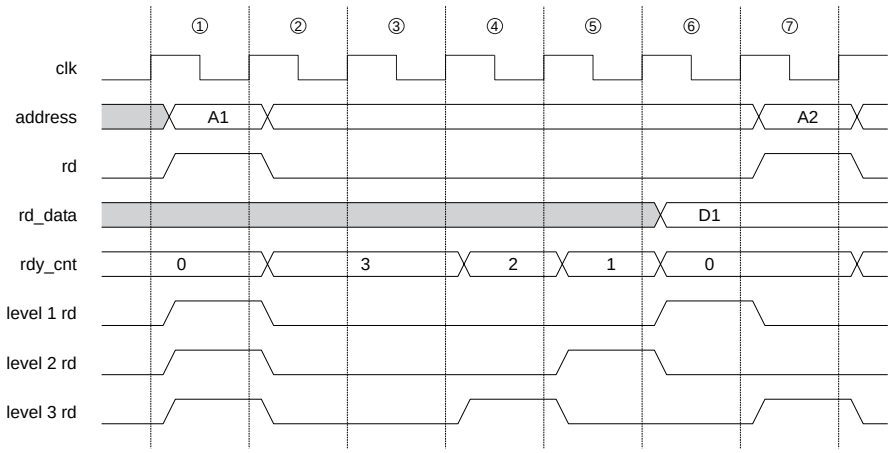

Figure 8.4: Different pipeline levels for a read transaction

8.4 Pipelining

Figure 8.4 shows a read transaction for a slave with four clock cycles latency. Without any pipelining, the next read transaction will start in cycle 7 after the data from the former read transaction is read by the master. The three bottom lines show when new read transactions (only the `rd` signal is shown, address lines are omitted from the figure) can be started for different pipeline levels. With pipeline level 1, a new transaction can start in the same cycle when the former read data is available (in this example in cycle 6). At pipeline level 2, a new transaction (either read or write) can start when `rdy_cnt` is 1, for pipeline level 3 the next transaction can start at a `rdy_cnt` of 2.

The implementation of level 1 in the slave is trivial (just two more transitions in the state machine). It is recommended to provide at least level 1 for read transactions. Level 2 is a little bit more complex but usually no additional address or data registers are necessary.

To implement level 3 pipelining in the slave, at least one additional address register is needed. However, to use level 3 the master has to issue the request in the same cycle as `rdy_cnt` goes to 2. That means this transition is combinatorial. We see in Figure 8.4 that `rdy_cnt` value of 3 means three or more cycles until the data is available and can therefore not be used to trigger a new transaction. Extension to an even deeper pipeline needs a wider `rdy_cnt`.

8.4.1 Interconnect

Although the definition of SimpCon is from a single master/slave point-to-point viewpoint, all variations of multiple slave and multiple master devices are possible.

Slave Multiplexing

To add several slaves to a single master, `rd_data` and `rdy_cnt` have to be multiplexed. Due to the fact that all `rd_data` signals are already registered by the slaves, a single pipeline stage will be enough for a large multiplexer. The selection of the multiplexer is also known at the transaction start but at least needed one cycle later. Therefore it can be registered to further speed up the multiplexer.

Master Multiplexing

SimpCon defines no signals for the communication between a master and an arbiter. However, it is possible to build a multi-master system with SimpCon. The SimpCon interface can be used as an interconnect between the masters and the arbiter and the arbiter and the slaves. In this case the arbiter acts as a slave for the master and as a master for the peripheral devices. An example of an arbiter for SimpCon, where JOP and a VGA controller are two masters for a shared main memory, can be found in [83]. The same arbiter is also used to build a chip-multiprocessor version of JOP.

The missing arbitration protocol in SimpCon results in the need to queue $n-1$ requests in an arbiter for n masters. However, this additional hardware results in a zero cycle bus grant. The master, which gets the bus granted, starts the slave transaction in the same cycle as the original read/write request.

8.5 Examples

This section provides some examples for the application of the SimpCon definition.

8.5.1 IO Port

TODO: Show how simple an IO port can be with SimpCon. We need no addresses and can tie `bsy_cnt` to 0. We only need the `rd` or `wr` signal to enable the port.

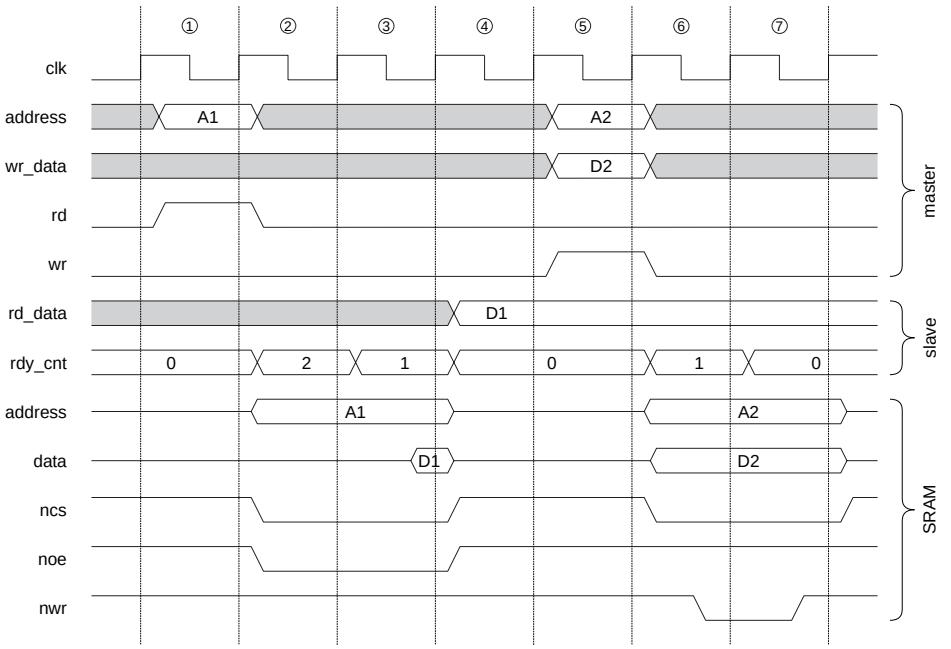

Figure 8.5: Static RAM interface without pipelining

8.5.2 SRAM interface

The following example is taken from an implementation of SimpCon for a Java processor. The processor is clocked at 100 MHz and the main memory consists of 15 ns static RAMs. Therefore the minimum access time for the RAM is two cycles. The slack time of 5ns forces us to use output registers for the RAM address and write data and input registers for the read data in the IO cells of the FPGA. These registers fit nicely with the intention of SimpCon to use registers inside the slave.

Figure 8.5 shows the memory interface for a non-pipelined read access followed by a write access. Four signals are driven by the master and two signals by the slave. The lower half of the figure shows the signals at the FPGA pins where the RAM is connected.

In cycle 1 the read transaction is started by the master and the slave registers the address. The slave also sets the registered control signals `ncs` and `noe` during cycle 1. Due to the placement of the registers in the IO cells, the address and control signals are valid at the FPGA pins very early in cycle 2. At the end of cycle 3 (15 ns after `address`, `ncs` and `noe`

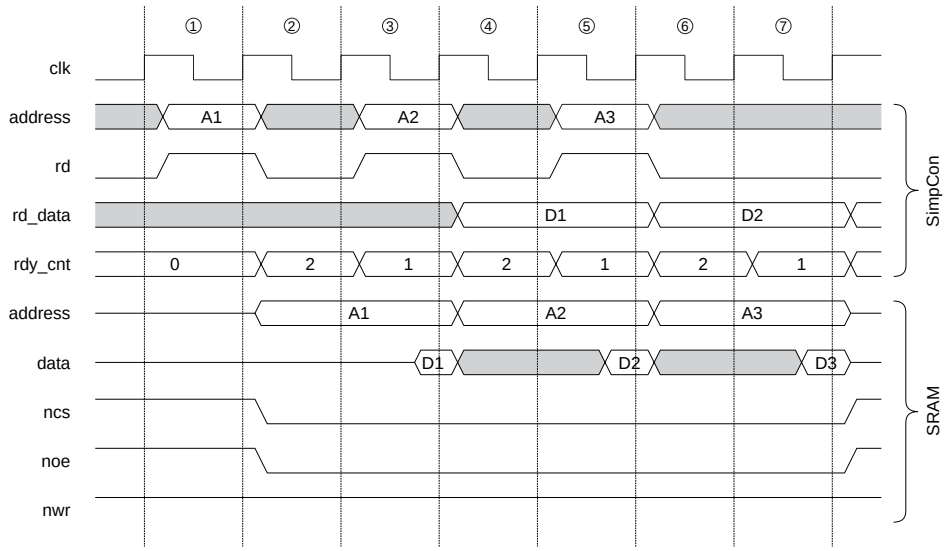

Figure 8.6: Pipelined read from a static RAM

are stable) the data from the RAM is available and can be sampled with the rising edge for cycle 4. The setup time for the read register is short, as the register can be placed in the IO cell. The master reads the data in cycle 4 and starts a write transaction in cycle 5. Address and data are again registered by the slave and are available for the RAM at the beginning of cycle 6. To perform a write in two cycles the `nwr` signal is registered by a negative triggered flip-flop.

In Figure 8.6 we see a pipelined read from the RAM with pipeline level 2. With this pipeline level and the two cycles read access time of the RAM we achieve the maximum possible bandwidth.

We can see the start of the second read transaction in cycle 3 during the read of the first data from the RAM. The new address is registered in the same cycle and available for the RAM in the following cycle 4. Although we have a pipeline level of 2 we need no additional address or data register. The read data is available for two cycles (`rdy_cnt` 2 or 1 for the next read) and the master is free to select one of the two cycles to read the data.

It has to be noted that pipelining with one read per cycle is possible with SimpCon. We just showed a 2 cycle slave in this example. For a SDRAM memory interface the ready counter will stay either at 2 or 1 during the single cycle reads (depending on the slave

pipeline level). It will go down to 0 only for the last data word to read.

8.5.3 Master Multiplexing

To add several slaves to a single master the `rd_data` and `bsy_cnt` have to be multiplexed. Due to the fact that all `rd_data` signals are registered by the slaves a single pipeline stage will be enough for a large multiplexer. The selection of the multiplexer is also known at the transaction start but needed at most in the next cycle. Therefore it can be registered to further speed up the multiplexer.

TODO: add a schematic for the master `rd_data` multiplexer.

8.6 Available VHDL Files

Besides the SimpCon documentation, some example VHDL files for slave devices and bridges are available from `http://www.opencores.org/projects.cgi/web/simpcon/overview`. All components are also part of the standard JOP distribution.

8.6.1 Components

- sc_pack.vhd defines VHDL records and some constants.
- sc_test_slave is a very simple SimpCon device. A counter to be read out and a register that can be written and read. There is no connection to the outer world. This example can be used as basis for a new SimpCon device.
- sc_sram16.vhd is a memory controller for 16-bit SRAM.
- sc_sram32.vhd is a memory controller for 32-bit SRAM.
- sc_sram32_flash.vhd is a memory controller for 32-bit SRAM, a NOR Flash, and a NAND Flash as used in the Cycore FPGA board for JOP.
- sc_uart.vhd is a simple UART with configurable baud rate and FIFO width.
- sc_usb.vhd is an interface to the parallel port of the FTDI 2232 USB chip. The register definition is identical to the UART and the USB connection can be used as a drop in replacement for a UART.

- sc_isa.vhd interfaces the old ISA bus. It can be used for the popular CS8900 Ethernet chip.
- sc_sigdel.vhd is a configurable sigma-delta converter for an FPGA that needs at minimum just two external components: a capacitor and a resistor.
- sc_fpu.vhd provides an interface to the 32-bit FPU available at `www.opencores.org`.
- sc_arbiter.vhd is a zero cycle latency, priority-based SimpCon arbiter written by Christof Pitter [84].

8.6.2 Bridges

- sc2wb.vhd is a SimpCon/Wishbone [80] bridge.
- sc2avalon.vhd is a SimpCon/Avalon [5] bridge to integrate a SimpCon based design with Altera's SOPC Builder [6].
- sc2ahbsl.vhd provides an interface to AHB slaves as defined in Gaisler's GRLIB [39]. Many of the available GPL AHB modules from the GRLIB can be used in a SimpCon based design.

8.7 Why a New Interconnection Standard?

There are many interconnection standards available for SoC designs. The natural question is: Why propose a new one? The answer is given in the following section. In summary, the available standards are still in the tradition of backplane busses and do not fit very well for pipelined on-chip interconnections.

8.7.1 Common SoC Interconnections

Several point-to-point and bus standards have been proposed. The following section gives a brief overview of common SoC interconnection standards.

The Advanced Microcontroller Bus Architecture (AMBA) [7] is the interconnection definition from ARM. The specification defines three different busses: Advanced High-performance Bus (AHB), Advanced System Bus (ASB), and Advanced Peripheral Bus (APB). The AHB is used to connect on-chip memory, cache, and external memory to the processor. Peripheral devices are connected to the APB. A bridge connects the AHB to the lower bandwidth APB. An AHB bus transfer can be one cycle with burst operation. With

the APB a bus transfer requires two cycles and no burst mode is available. Peripheral bus cycles with wait states are added in the version 3 of the APB specification. ASB is the predecessor of AHB and is not recommended for new designs (ASB uses both clock phases for the bus signals – very uncommon for today's synchronous designs). The AMBA 3 AXI (Advanced eXtensible Interface) [8] is the latest extension to AMBA. AXI introduces out-of-order transaction completion with the help of a 4 bit transaction ID tag. A ready signal acknowledges the transaction start. The master has to hold the transaction information (e.g. address) until the interconnect signals ready. This enhancement ruins the elegant single cycle address phase from the original AHB specification.

Wishbone [80] is a public domain standard used by several open-source IP cores. The Wishbone interface specification is still in the tradition of microcomputer or backplane busses. However, for a SoC interconnect, which is usually point-to-point[3], this is not the best approach. The master is requested to hold the address and data valid through the whole read or write cycle. This complicates the connection to a master that has the data valid only for one cycle. In this case the address and data have to be registered *before* the Wishbone connect or an expensive (time and resources) multiplexer has to be used. A register results in one additional cycle latency. A better approach would be to register the address and data in the slave. In that case the address decoding in the slave can be performed in the same cycle as the address is registered. A similar issue, with respect to the master, exists for the output data from the slave: As it is only valid for a single cycle, the data has to be registered by the master when the master is not reading it immediately. Therefore, the slave should keep the last valid data at its output even when the Wishbone strobe signal (*wb.stb*) is not assigned anymore. Holding the data in the slave is usually *for free* from the hardware complexity – it is *just* a specification issue. In the Wishbone specification there is no way to perform pipelined read or write. However, for blocked memory transfers (e.g. cache load) this is the usual way to achieve good performance.

The Avalon [5] interface specification is provided by Altera for a system-on-a-programmable-chip (SOPC) interconnection. Avalon defines a great range of interconnection devices ranging from a simple asynchronous interface intended for direct static RAM connection up to sophisticated pipeline transfers with variable latencies. This great flexibility provides an easy path to connect a peripheral device to Avalon. How is this flexibility possible? The *Avalon Switch Fabric* translates between all those different interconnection types. The switch fabric is generated by Altera's SOPC Builder tool. However, it seems that this switch fabric is Altera proprietary, thus tying this specification to Altera FPGAs.

[3] Multiplexers are used instead of busses to connect several slaves and masters.

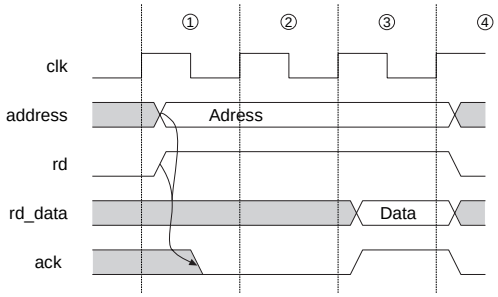

Figure 8.7: Classic basic read transaction

The On-Chip Peripheral Bus (OPB) [52] is an open standard provided by IBM and used by Xilinx. The OPB specifies a bus for multiple masters and slaves. The implementation of the bus is not directly defined in the specification. A distributed ring, a centralized multiplexer, or a centralized AND/OR network are suggested. Xilinx uses the AND/OR approach and all masters and slaves must drive the data busses to zero when inactive.

Sonics Inc. defined the Open Core Protocol (OCP) [78] as an open, freely available standard. The standard is now handled by the OCP International Partnership[4].

8.7.2 What's Wrong with the Classic Standards?

All SoC interconnection standards, which are widely in use, are still in the tradition of a backplane bus. They force the master to hold the address and control signals until the slave provides the data or acknowledges the write request. However, this is not necessary in a clocked, synchronous system. Why should we force the master to hold the signals? Let the master move on after submitting the request in a single cycle. Forcing the address and control valid for the complete request disables any form of pipelined requests.

Figure 8.7 shows a read transaction with wait states as defined in Wishbone [80], Avalon [5], OPB [52], and OCP [78].[5] The master issues the read request and the address in cycle 1. The slave has to reset the `ack` in the same cycle. When the slave data is available, the acknowledge signal is set (`ack` in cycle 3). The master has to read the data and register them within the same clock cycle. The master has to hold the address, write data, and control signal active until the acknowledgement from the slave arrives. For pipelined reads,

[4] `www.ocpip.org`

[5] The signal names are different, but the principle is the same for all mentioned busses.

the ack signal can be split into two signals (available in Avalon and OCP): one to accept the request and a second one to signal the available data.

The master is blind about the status of the outstanding transaction until it is finished. It could be possible that the slave informs the master in how many cycles the result will be available. This information can help in building deeply pipelined masters.

Only the AMBA AHB [7] defines a different protocol. A single cycle address phase followed by a variable length data phase. The slave acknowledgement (HREADY) is only necessary in the data phase avoiding the combinatorial path from address/command to the acknowledgement. Overlapping address and data phase is allowed and recommended for high performance. Compared to SimpCon, AMBA AHB allows for single stage pipelining, whereas SimpCon makes multi-stage pipelining possible using the ready counter (rdy_cnt). The rdy_cnt signal defines the delay between the address and the data on a read, signalled by the slave. Therefore, the pipeline depth of the bus and the slaves is only limited by the bit width of rdy_cnt.

Another issue with all interconnection standards is the single cycle availability of read data at the slaves. Why not keep the read data valid as long as there is no new read data available? This feature would allow the master to be more flexible when to read the data. It would allow issuing a read command and then continuing with other instructions – a feature known as data pre-fetching to hide long latencies.

The last argument sounds contradictory to the first argument: provide the transaction data at the master just for a single cycle, but request the slave to hold the data for several cycles. However, it is motivated by the need to free up the master, keep it *moving*, and move the data hold (register) burden into the slave. As data processing bottlenecks are usually found in the master devices, it sounds natural to move as much work as possible to the slave devices to free up the master.

Avalon, Wishbone, and OPB provide a single cycle latency access to slaves due to the possibility of acknowledging a request in the same cycle. However, this feature is a scaling issue for larger systems. There is a combinatorial path from master address/command to address decoding, slave decision on ack, slave ack multiplexing back to the master and the master decision to hold address/command or read the data and continue. Also, the slave output data multiplexer is on a combinatorial path from the master address.

AMBA, AHB, and SimpCon avoid this scaling issue by requesting the acknowledge in the cycle following the command. In SimpCon and AMBA, the select for the read data multiplexer can be registered as the read address is known at least one cycle before the data is available. The later acknowledgement results in a minor drawback on SimpCon and AMBA (nothing is for free): It is not possible to perform a single cycle read or write without pipelining. A single, non pipelined transaction takes two cycles without a wait

Performance	Memory	Interconnect
16,633	32 bit SRAM	SimpCon
14,259	32 bit SRAM	AMBA
14,015	32 bit SRAM	Avalon/PTF
13,920	32 bit SRAM	Avalon/VHDL
15,762	32 bit on-chip	Avalon
14,760	16 bit SRAM	SimpCon
11,322	16 bit SRAM	Avalon
7,288	16 bit SDRAM	Avalon

Table 8.2: JOP performance with different interconnection types

state. However, a single cycle read transaction is only possible for very simple slaves. Most non-trivial slaves (e.g. memory interfaces) will not allow a single cycle access anyway.

8.7.3 Evaluation

We compare the SimpCon interface with the AMBA and the Avalon interface as two examples of common interconnection standards. As an evaluation example, we interface an external asynchronous SRAM with a tight timing. The system is clocked at 100 MHz and the access time for the SRAM is 15 ns. Therefore, there are 5 ns available for on-chip register to SRAM input and SRAM output to on-chip register delays. As an SoC, we use an actual low-cost FPGA (Cyclone EP1C6 [3] and a Cyclone II).

The master is a Java processor (JOP [107, 112]). The processor is configured with a 4 KB instruction cache and a 512 byte on-chip stack cache. We run a complete application benchmark on the different systems. The embedded benchmark (*Kfl* as described in [106]) is an industrial control application already in production.

Table 8.2 shows the performance numbers of this JOP/SRAM interface on the embedded benchmark. It measures iterations per second and therefore higher numbers are better. One iteration is the execution of the main control loop of the *Kfl* application. For a 32 bit SRAM interface, we compare SimpCon against AMBA and Avalon. SimpCon outperforms AMBA by 17% and Avalon by 19%[6] on a 32 bit SRAM.

The AMBA experiment uses the SRAM controller provided as part of GRLIB [39] by

[6]The performance is the measurement of the execution time of the whole application, not only the difference between the bus transactions.

Gaisler Research. We avoided writing our own AMBA slave to verify that the AMBA implementation on JOP is correct. To provide a fair comparison between the single master solutions with SimpCon and Avalon, the AMBA bus was configured without an arbiter. JOP is connected directly to the AMBA memory slave. The difference between the SimpCon and the AMBA performance can be explained by two facts: (1) as with the Avalon interconnect, the master has the information when the slave request is ready at the same cycle when the data is available (compared to the `rdy_cnt` feature); (2) the SRAM controller is conservative as it asserts `HREADY` one cycle later than the data is available in the read register (`HRDATA`). The second issue can be overcome by a better implementation of the SRAM AMBA slave.

The Avalon experiment considers two versions: an SOPC Builder generated interface (PTF) to the memory and a memory interface written in VHDL. The SOPC Builder interface performs slightly better than the VHDL version that generates the Avalon `waitrequest` signal. It is assumed that the SOPC Builder version uses fixed wait states within the switch fabric.

We also implemented an Avalon interface to the single-cycle on-chip memory. SimpCon is even faster with the 32 bit off-chip SRAM than with the on-chip memory connected via Avalon. Furthermore, we also performed experiments with a 16 bit memory interface to the same SRAM. With this smaller data width the pressure on the interconnection and memory interface is higher. As a result the difference between SimpCon and Avalon gets larger (30%) on the 16 bit SRAM interface. To complete the picture we also measured the performance with an SDRAM memory connected to the Avalon bus. We see that the large latency of an SDRAM is a big performance issue for the Java processor.

8.8 Summary

This document describes a simple (with respect to the definition and implementation) and efficient SoC interconnect. The novel signal `rdy_cnt` allows an early signalling to the master when read data will be valid. This feature allows the master to restart a stalled pipeline earlier to react for arriving data. Furthermore, this feature also enables pipelined bus transactions with a minimal effort on the master and the slave side.

We have compared SimpCon quantitative with AMBA and Avalon, two common interconnection definitions. The application benchmark shows a performance advantage of SimpCon by 17% over AMBA and 19% over Avalon interfaces to an SRAM.

SimpCon is used as the main interconnect for the Java processor JOP in a single master, multiple salves configuration. SimpCon is also used to implement a shared memory chip-multiprocessor version of JOP. Furthermore, in a research project on time-triggered

network-on-chip [111] SimpCon is used as the *socket* to this NoC.

The author thanks Kevin Jennings and Tommy Thorn for the interesting discussions about SimpCon, Avalon, and on-chip interconnection in general at the Usenet newsgroup `comp.arch.fpga`.

9 Chip Multiprocessing

This chapter describes the configuration of a chip multiprocessor (CMP) version of JOP. The various SimpCon based arbiters have been developed by Christof Pitter and are described in [83, 84, 85].

The project file to start with is cyccmp, a configuration for three processors with a TDMA based arbiter in the Cycore board with the EP1C12.

9.1 Booting a CMP System

One interesting issue for a CMP system is the question how the startup or boot-up is performed. Before we explain the CMP solution, we need an understanding of the boot-up sequence of JOP in an FPGA. On power-up, the FPGA starts the configuration state machine to read the FPGA configuration data either from a Flash or via a download cable (for development). When the configuration has finished an internal reset is generated. After that reset, microcode instructions are executed starting from address 0. At this stage, we have not yet loaded any application program (Java bytecode). The first sequence in microcode performs this task. The Java application can be loaded from an external Flash or via a serial line (or USB port) from a PC. The microcode assembly configured the mode. Consequently, the Java application is loaded into the main memory. To simplify the startup code we perform the rest of the startup in Java itself, even when some parts of the JVM are not yet setup.

In the next step, a minimal stack frame is generated and the special method Startup.boot() is invoked. From now on JOP runs in Java mode. The method boot() performs the following steps:

- Send a greeting message to *stdout*
- Detect the size of the main memory
- Initialize the data structures for the garbage collector
- Initialize java.lang.System
- Print out JOP's version number, detected clock speed, and memory size
- Invoke the static class initializers in a predefined order
- Invoke the main method of the application class

The boot-up process is the same for all processors until the generation of the internal reset and the execution of the first microcode instruction. From that point on, we have to take care that *only one* processor performs the initialization steps.

All processors in the CMP are functionally identical. Only one processor is designated to boot-up and initialize the whole system. Therefore, it is necessary to distinguish between the different CPUs. We assign a unique CPU identity number (CPU ID) to each processor. Only processor CPU0 is designated to do all the boot-up and initialization work. The other CPUs have to wait until CPU0 completes the boot-up and initialization sequence. At the beginning of the booting sequence, CPU0 loads the Java application. Meanwhile, all other processors are waiting for an *initialization finished* signal of CPU0. This busy wait is performed in microcode. When the other CPUs are enabled, they will run the same sequence as CPU0. Therefore, the initialization steps are guarded by a condition on the CPU ID.

In our current prototype, we let all additional CPUs also invoke the main method of the application. This is a shortcut for a simple evaluation of the system[1]. In a future version, the additional CPUs will invoke a system method to be integrated into the normal scheduling system.

9.2 CMP Scheduling

There are two possibilities to run multiple threads on the CMP system:

[1] In the main method we execute different applications based on the CPU ID.

1. A single thread per processor
2. Several threads on each processor

For the configuration of one thread per processor the scheduler does not need to be started. Running several threads on each core is managed via the JOP real-time threads RtThread. On each core a local preemptive, priority based scheduler runs. Threads cannot migrate from one core to another one.

9.2.1 One Thread per Core

The first processor executes, as usual, main(). To execute code on the other cores a Runnable has to be registered for each core. After registering those Runnables the other cores need to be started. The following code shows an example that can be found in test/cmp/HelloCMP.java.

```
public class HelloCMP implements Runnable {

    int id;

    static Vector msg;

    public HelloCMP(int i) {
        id = i;
    }

    /**
     * @param args
     */
    public static void main(String[] args) {

        msg = new Vector();

        System.out.println("Hello World from CPU 0");

        SysDevice sys = IOFactory.getFactory().getSysDevice();
        for (int i=0; i<sys.nrCpu-1; ++i) {
            Runnable r = new HelloCMP(i+1);
```

```
            Startup.setRunnable(r, i);
        }

        // start the other CPUs
        sys.signal = 1;

        // print their messages
        for (;;) {
            int size = msg.size();
            if (size!=0) {
                StringBuffer sb = (StringBuffer) msg.remove(0);
                System.out.println(sb);
            }
        }
    }

    public void run() {

        StringBuffer sb = new StringBuffer();
        sb.append("Hello World from CPU ");
        sb.append(id);
        msg.addElement(sb);
    }

}
```

9.2.2 Scheduling on the CMP System

Running several threads on each core is possible with RtThread and setting the core for each thread with RtThread.setProcessor(nr). The following example (test/cmp/RtHelloCMP.java) shows registering 50 threads on three cores. On missionStart() the threads are distributed to the cores, a scheduler for each core registered as timer interrupt handler, and the other cores started.

```
public class RtHelloCMP extends RtThread {

    public RtHelloCMP(int prio, int us) {
```

```
        super(prio, us);
    }

    int id;

    public static Vector msg;

    final static int NR_THREADS = 50;

    /**
     * @param args
     */
    public static void main(String[] args) {

        msg = new Vector();

        System.out.println("Hello World from CPU 0");

        SysDevice sys = IOFactory.getFactory().getSysDevice();
        for (int i=0; i<NR_THREADS; ++i) {
            RtHelloCMP th = new RtHelloCMP(1, 1000*1000);
            th.id = i;
            th.setProcessor(i%sys.nrCpu);
        }

        System.out.println("Start mission");
        // start mission and other CPUs
        RtThread.startMission();
        System.out.println("Mission started");

        // print their messages
        for (;;) {
            RtThread.sleepMs(5);
            int size = msg.size();
            if (size!=0) {
                StringBuffer sb = (StringBuffer) msg.remove(0);
                // System.out.print(sb);
```

```
                // converts StringBuffer to a String and creates garbage
                for (int i=0; i<sb.length(); ++i) {
                    System.out.print(sb.charAt(i));
                }
            }
        }
    }

    public void run() {

        StringBuffer sb = new StringBuffer();
        StringBuffer ping = new StringBuffer();

        sb.append("Thread ");
        sb.append((char) ('A'+id));
        sb.append(" start on CPU ");
        sb.append(IOFactory.getFactory().getSysDevice().cpuId);
        sb.append("\r\n");
        msg.addElement(sb);
        waitForNextPeriod();

        for (;;) {
            ping.setLength(0);
            ping.append((char) ('A'+id));
            msg.addElement(ping);
            waitForNextPeriod();
        }
    }

}
```

10 Results

In this chapter, we will present the evaluation results for JOP, with respect to size, performance, and WCET. Table 10.1 compares JOP with other Java hardware solutions (see also Chapter 11). The column year indicates the first date at which the processor became available or the first publication about the processor. The research project Komodo has now ceased, while FemtoJava is still being used as a basis for active research.

We can see that JOP is the smallest realization in an FPGA and also has the highest clock frequency of all soft-core implementations. JOP also has a minimum CPI of 1 while Komodo and FemtoJava have minimum CPIs of four and three respectively.

TODO: CPI of four is not true anymore (for jamuth).

	Target technology	Size	Speed (MHz)	Java standard	Min. CPI	Year
JOP	Altera, Xilinx FPGA	1830 LCs, 3 KB RAM	100	J2ME CLDC	1	2001
picoJava	No realization	128K gates + memory		Full	1	1999
aJile	ASIC 0.25μ	25K gates + ROM	100	J2ME CLDC		2000
Moon	Altera FPGA	3660 LCs, 4 KB RAM				2000
Lightfoot	Xilinx FPGA	3400 LCs	40			2001
Komodo	Xilinx FPGA	2600 LCs	33		4	2000
FemtoJava	Altera Flex 10K	2000 LCs	4	Subset: 69 bytecodes, 16-bit ALU	3	2001

Table 10.1: Comparison of Java hardware with JOP

In the following section, the hardware platform that is used for benchmarking is described. This is followed by a comparison of JOP's resource usage with other soft-core

processors. In the 'General Performance' section, a number of different solutions for embedded Java are compared at the bytecode level and at the application level. The basic properties of the real-time scheduler are evaluated using the Reference Implementation (RI) of the RTSJ on a Linux system and the real-time profile from Section 5.3 on top of JOP. It is also shown that our objective of providing an easy target for WCET analysis has been achieved. This chapter concludes with a short description of real-world applications that use JOP.

10.1 Hardware Platforms

During the development of JOP and its predecessors, several different FPGA boards were developed. The first experiments involved using Altera FPGAs EPF8282, EPF8452, EPF10K10 and ACEX 1K30 on boards that were connected to the printer port of a PC for configuration, download and communication. The next step was the development of a stand-alone board with FLASH memory and static RAM. This board was developed in two variants, one with an ACEX 1K50 and the other with a Cyclone EP1C6 or EP1C12. Both boards are pin-compatible and are used in commercial applications of JOP. The Cyclone board is the hardware that is used for the following evaluations.

This board is an ideal development system for JOP. Static RAM and FLASH are connected via independent buses to the FPGA. All unused FPGA pins and the serial line are available via four connectors. The FLASH can be used to store configuration data for the FPGA and application program/data. The FPGA can be configured with a ByteBlasterMV download cable or loaded from the flash (with a small CPLD on board). As the FLASH is also connected to the FPGA, it can be programmed from the FPGA. This allows for upgrades of the Java program and even the processor core itself in the field. The board is slightly different from other FPGA prototyping boards, in that its connectors are on the bottom side. Therefore, it can be used as a module (60 mm x 48 mm), i.e. as part of a larger board that contains the periphery. The Cyclone board contains:

- Altera Cyclone EP1C6Q240 or EP1C12Q240
- Step Down voltage regulator (1V5)
- Crystal clock (20MHz) at the PLL input (up to 640MHz internal)
- 512KB FLASH (for FPGA configuration and program code)
- 1MB fast asynchronous RAM (15 ns)

- Up to 128MB NAND FLASH
- ByteBlasterMV port
- Watchdog with LED
- EPM7064 PLD to configure the FPGA from the FLASH on watchdog reset
- Serial interface driver (MAX3232)
- 56 general-purpose IO pins

The RAM consists of two independent 16-bit banks (with their own address and control lines). Both RAM chips are on the bottom side of the PCB, directly under the FPGA pins. As the traces are very short (under 10 mm), it is possible to use the RAMs at full speed without reflection problems. The two banks can be combined to form 32-bit RAM or support two independent CPU cores. Pictures and the schematic of the board can be found in Appendix E.

An expansion board hosts the CPU module and provides a complete Java processor system with Internet connection. A step down switching regulator with a large AC/DC input range supplies the core board. All input and output pins are EMC/ESD-protected and routed to large connectors (5.08 mm Phoenix). Analog comparators can be used to build sigma-delta ADCs. For FPGA projects with a network connection, a CS8900 Ethernet controller with an RJ45 connector is included on the expansion board.

10.2 Resource Usage

Cost is an important issue for embedded systems. The cost of a chip is directly related to the die size (the cost per die is roughly proportional to the square of the die area [49]). Processors for embedded systems are therefore optimized for minimum chip size. In this section, we will compare JOP with different processors in terms of size.

One major design objective in the development of JOP was to create a small system that can be implemented in a low-cost FPGA. Figure 10.1 and Table 10.2 show the resource usage for different configurations of JOP and different soft-core processors implemented in an Altera EP1C6 FPGA [3]. Estimating equivalent gate counts for designs in an FPGA is problematic. It is therefore better to compare the two basic structures, Logic Cells (LC) and embedded memory blocks. The maximum frequency for all soft-core processors is in the same technology or normalized (SPEAR) to the technology.

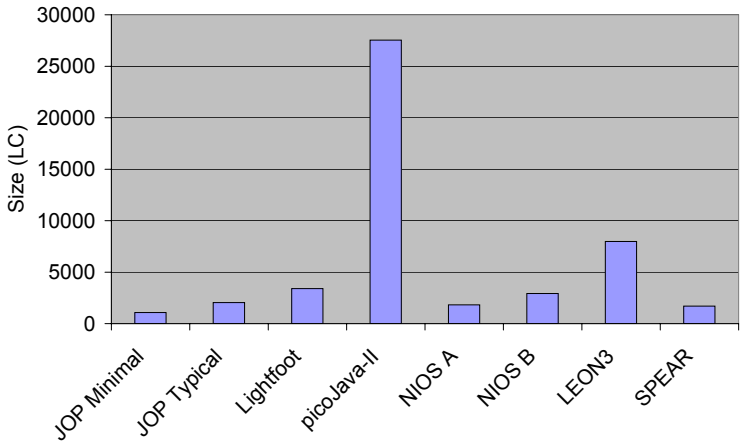

Figure 10.1: Size in logic cells (LC) of different soft-core processors

All configurations of JOP contain the on-chip microcode memory, the 1 KB stack cache, a 1 KB method cache, a memory interface to a 32-bit static RAM, and an 8-bit FLASH interface for the Java program and the FPGA configuration data. The minimum configuration implements multiplication and the shift operations in microcode. In the typical configuration, these operations are implemented as a sequential Booth multiplier and a single-cycle barrel shifter. The typical configuration also contains some useful I/O devices such as an UART and a timer with interrupt logic for multi-threading. The typical configuration of JOP consumes about 30% of the LCs in a Cyclone EP1C6, thus leaving enough resources free for application-specific logic.

As a reference, NIOS [4], Altera's popular RISC soft-core, is also included in Table 10.2. NIOS has a 16-bit instruction set, a 5-stage pipeline and can be configured with a 16 or 32-bit datapath. Version A is the minimum configuration of NIOS. Version B adds an external memory interface, multiplication support, and a timer. Version A is comparable with the minimal configuration of JOP, and Version B with its typical configuration.

LEON3 [38], the open-source implementation of the SPARC V8 architecture, has been ported to the exact same hardware that was used for the JOP numbers. LEON3 is representative for a RISC processor that is used in embedded real-time systems (e.g., by ESA for space missions).

Processor	*Resources* (*LC*)	Memory (KB)	fmax (MHz)
JOP Minimal	1077	3.25	98
JOP Typical	2049	3.25	100
Lightfoot[1] [28]	3400	4	40
picoJava-II [89]	27543	47.6	43
NIOS A [4]	1828	6.2	120
NIOS B [4]	2923	5.5	119
LEON3 [38]	7978	10.9	35
SPEAR[2] [29]	1700	8	80

Table 10.2: Size and maximum frequency of FPGA soft-core processors

SPEAR [29] (Scalable Processor for Embedded Applications in Real-time Environments) is a 16-bit processor with deterministic execution times. SPEAR contains predicated instructions to support single-path programming [90]. SPEAR is included in the list as it is also a processor designed for real-time systems.

To prove that the VHDL code for JOP is as portable as possible, JOP was also implemented in a Xilinx Spartan-3 FPGA [134]. Only the instantiation and initialization code for the on-chip memories is vendor-specific, whilst the rest of the VHDL code can be shared for the different targets. JOP consumes about the same LC count (1844 LCs) in the Spartan device, but has a slower clock frequency (83 MHz).

From this comparison we can see that we have achieved our objective of designing a small processor. The commercial Java processor, Lightfoot, consumes 1.7 times the logic resources of JOP in the typical configuration (with a lower clock frequency). A typical 32-bit RISC processor (NIOS) consumes about 1.5 times (LEON about four times) the resources of JOP. However, the NIOS processor can be clocked 20% faster than JOP in the same technology. The vendor independent and open-source RISC processor LEON can be clocked only with 35% of JOP's frequency. The only processor that is similar in size is

[1] The data for the Lightfoot processor is taken from the data sheet [28]. The frequency used is that in a Virtex-II device from Xilinx. JOP can be clocked at 100 MHz in the Virtex-II device, making this comparison valid.

[2] As SPEAR uses internal memory blocks in asynchronous mode it is not possible to synthesize it without modification for the Cyclone FPGA. The clock frequency of SPEAR in an Altera Cyclone is an estimate based on following facts: SPEAR can be clocked at 40 MHz in an APEX device and JOP can be clocked at 50 MHz in the same device.

Processor	Core [gate]	Memory [gate]	Sum. [gate]
JOP	12K	43K	55K
picoJava	128K	314K	442K
aJile	25K	912K	937K
Pentium MMX			1125K

Table 10.3: Gate count estimates for various processors

SPEAR. However, while SPEAR is a 16-bit processor, JOP contains a 32-bit datapath.

Table 10.3 provides gate count estimates for JOP, picoJava, the aJile processor, and the Intel Pentium MMX processor that is used in the benchmarks in the next section. Equivalent gate count for an LC[3] varies between 5.5 and 7.4 – we chose a factor of 6 gates per LC and 1.5 gates per memory bit for the estimated gate count for JOP in the table. JOP is listed in the typical configuration that consumes 2050 LCs. The Pentium MMX contains 4.5M transistors [34] that are equivalent to 1125K gates.

We can see from the table that the on-chip memory dominates the overall gate count of JOP, and to an even greater extent, of the aJile processor. The aJile processor is roughly the same size as the Pentium MMX, and both are about 20 times larger than JOP.

10.3 Performance

In this section, we will evaluate the performance of JOP in relation to other Java systems. Although JOP is intended as a processor with a low WCET for all operations, its general performance is still important. In the first section, we will evaluate JOP's average performance.

The comparison of the implementation of the simple real-time profile, as described in Section 5.3, on JOP with the RI of the RTSJ on top of Linux can be found in [104].

10.3.1 General Performance

Running benchmarks is problematic, both generally and especially in the case of embedded systems. The best benchmark would be the application that is intended to run on the

[3]The factors are derived from the data provided for various processors in Chapter 11 and from the resource estimates in Section 4.4.

system being tested. To get comparable results, SPEC provides benchmarks for various systems. However, the one for Java, the SPECjvm98 [119], needs more functionality than what is usually available in a CLDC compliant device (e.g., a filesystem and `java.net`). Some benchmarks from the SPECjvm98 suites also need several megabytes of heap.

Due to the absence of a *standard* Java benchmark for embedded systems, a small benchmark suite that should run on even the smallest device is provided here. It contains several micro-benchmarks for evaluating the number of clock cycles for single bytecodes or short sequences of bytecodes, and two application benchmarks.

To provide a realistic workload for embedded systems, a real-time application was adapted to create the first application benchmark (Kfl). The application is taken from one of the nodes of a distributed motor control system [101] (the first industrial application of JOP). The application is written as a cyclic executive. A simulation of both the environment (sensors and actors) and the communication system (commands from the master station) forms part of the benchmark, so as to simulate the real-world workload. The second application benchmark is an adaptation of a tiny TCP/IP stack for embedded Java. This benchmark contains two UDP server/clients, exchanging messages via a loopback device. The Kfl benchmark consists of 511 methods and 14 KB code, the UDP/IP benchmark of 508 methods and 13 KB code (including the supporting library).

As we will see, there is a great variation in processing power across different embedded systems. To cater for this variation, all benchmarks are 'self adjusting'. Each benchmark consists of an aspect that is benchmarked in a loop and an 'overhead' loop that contains any overheads from the benchmark that should be subtracted from the result (this feature is designed for the micro-benchmarks). The loop count adapts itself until the benchmark runs for more than a second. The number of iterations per second is then calculated, which means that higher values indicate better performance.

All the benchmarks measure how often a function is executed per second. In the Kfl benchmark, this function contains the main loop of the application that is executed in a periodic cycle in the original application. In the benchmark, the wait for the next period is omitted, so that the time measured solely represents execution time. The UDP benchmark contains the generation of a request, transmitting it through the UDP/IP stack, generating the answer and transmitting it back as a benchmark function. The iteration count is the number of received answers per second.

The accuracy of the measurement depends on the resolution of the system time. For the measurements under Linux, the system time has a resolution of 10ms, resulting in an inaccuracy of 1%. The accuracy of the system time on leJOS, TINI and the aJile is not known, but is considered to be in the same range. For JOP, a μs counter is used for time measurement.

The following list gives a brief description of the Java systems that were benchmarked:

JOP is implemented in a Cyclone FPGA [3], running at 100 MHz. The main memory is a 32-bit SRAM (15ns) with an access time of 2 clock cycles. The benchmarked configuration of JOP contains a 4 KB method cache organized in 16 blocks.

leJOS As an example for a low-end embedded device, we use the RCX robot controller from the LEGO MindStorms series. It contains a 16-bit Hitachi H8300 microcontroller [51], running at 16 MHz. leJOS [118] is a tiny interpreting JVM for the RCX.

KVM is a port of Sun's KVM, is part of the Connected Limited Device Configuration (CLDC) [123], to Altera's NIOS II processor on MicroC Linux. NIOS is implemented on a Cyclone FPGA and clocked at 50 MHz. Besides the different clock frequency, this is a good comparison of an interpreting JVM running in the same FPGA as JOP.

TINI is an enhanced 8051 clone running a software JVM. The results were taken from a custom board with a 20 MHz crystal, and the chip's PLL is set to a factor of 2.

Cjip The measured system [53] is a replacement of the TINI board and contains a Cjip [55] clocked with 80 MHz and 8 MB DRAM.

Komodo Komodo [63] is a Java processor as a basis for research on real-time scheduling on a multithreaded microcontroller (see Section 11.2.8). The benchmark results were obtained by Matthias Pfeffer [81] on a cycle-accurate simulation of Komodo. The values are obtained without garbage collection. According to Pfeffer, Komodo can be clocked with 33MHz in a Xlinix XCV800.

aJile aJile's JEMCore is a direct-execution Java processor that is available in two different versions: the **aJ80** and the **aJ100** [2]. The aJ100 provides a generic 8-bit, 16-bit, or 32-bit external bus interface, while the aJ80 only provides an 8-bit interface.

EJC The Embedded Java Controller (EJC) platform [35] is a typical example of a JIT system on a RISC processor. The system is based on a 32-bit ARM720T processor running at 74 MHz. It contains up to 64 MB SDRAM and up to 16 MB of NOR flash.

gcj is the GNU compiler for Java. This configuration represents the batch compiler solution, running on a 266 MHz Pentium MMX under Linux.

MB is the realization of Java on a RISC processor for an FPGA (Xilinx MicroBlaze [133]). Java is compiled to C with a Java compiler for real-time systems [76] and the C program is compiled with the standard GNU toolchain.

It would be interesting to include the other soft-core Java processors (Moon, Lightfoot, and FemtoJava) in this comparison. However, it was not possible to obtain the benchmark data. The company that produced Moon seems to have disappeared and FemtoJava could not run all benchmarks.

In Figure 10.2, the geometric mean of the two application benchmarks is shown. The unit used for the result is iterations per second. Note that the vertical axis is logarithmic, in order to obtain useful figures to show the great variation in performance. The top diagram shows absolute performance, while the bottom diagram shows the same results scaled to a 1 MHz clock frequency. The results of the application benchmarks and the geometric mean are shown in Table 10.4.

It should be noted that scaling to a single clock frequency could prove problematic. The relation between processor clock frequency and memory access time cannot always be maintained. To give an example, if we were to increase the results of the 100 MHz JOP to 1 GHz, this would also involve reducing the memory access time from 15 ns to 1.5 ns. Processors with 1 GHz clock frequency are already available, but the fastest asynchronous SRAM to date has an access time of 10 ns.

10.3.2 Discussion

When comparing JOP and the aJile processor against leJOS, KVM, and TINI, we can see that a Java processor is up to 500 times faster than an interpreting JVM on a standard processor for an embedded system. The average performance of JOP is even better than a JIT-compiler solution on an embedded system, as represented by the EJC system.

Even when scaled to the same clock frequency, the compiling JVM on a PC (gcj) is much faster than either embedded solution. However, the kernel of the application is smaller than 4 KB [105]. It therefore fits in the level one cache of the Pentium MMX. For a comparison with a Pentium class processor we would need a larger application.

JOP is about 7 times faster than the aJ80 Java processor on the popular JStamp board. However, the aJ80 processor only contains an 8-bit memory interface, and suffers from this bottleneck. The SaJe system contains the aJ100 with 32-bit, 10 ns SRAMs. JOP with its 15 ns SRAMs is about 12% faster than the aJ100 processor.

The MicroBlaze system is a representation of a Java batch-compilation system for a

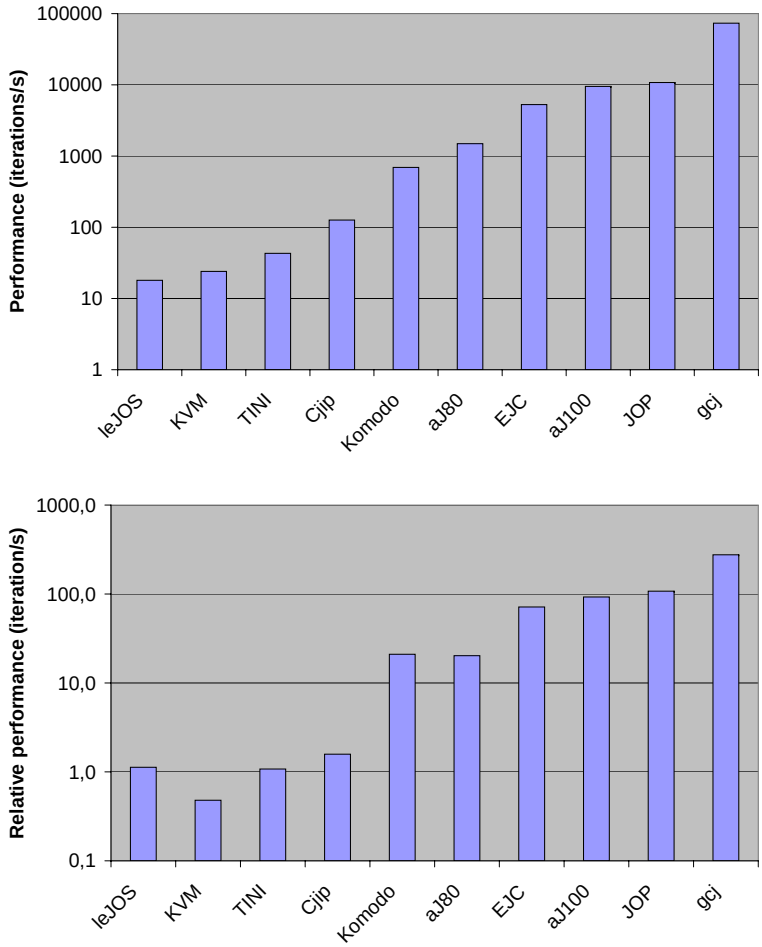

Figure 10.2: Performance comparison of different Java systems with application benchmarks. The diagrams show the geometric mean of the two benchmarks in iterations per second – a higher value means higher performance. The top diagram shows absolute performance, while the bottom diagram shows the result scaled to 1 MHz clock frequency.

	Frequency (MHz)	Kfl	UDP/IP	Geom. Mean (Iterations/s)	Scaled
JOP	100	17111	6781	10772	108
leJOS	16	25	13	18	1.1
TINI	40	64	29	43	1.1
KVM	50	36	16	24	0.5
Cjip	80	176	91	127	1.6
Komodo	33	924	520	693	21
aJ80	74	2221	1004	1493	20
aJ100	103	14148	6415	9527	92
EJC	74	9893	2822	5284	71
gcj	266	139884	38460	73348	276
MB	100	3792			

Table 10.4: Application benchmarks on different Java systems. The table shows the benchmark results in iterations per second – a higher value means higher performance.

RISC processor. MicroBlaze is configured with the same cache[4] as JOP and clocked at the same frequency. JOP is about four times faster than this solution, thus showing that native execution of Java bytecodes is faster than batch-compiled Java on a similar system. However, the results of the MicroBlaze solution are at a preliminary stage[5], as the Java2C compiler [76] is still under development.

The micro-benchmarks are intended to give insight into the implementation of the JVM. In Table 10.5, we can see the execution time in clock cycles of various bytecodes. As almost all bytecodes manipulate the stack, it is not possible to measure the execution time for a single bytecode in the benchmark loop. The single bytecode would trash the stack. As a minimum requirement, a second instruction is necessary in the loop to reverse the stack operation.

For JOP we can deduce that the WCET for simple bytecodes is also the average execution time. We can see that the combination of `iload` and `iadd` executes in two cycles, which means

[4] The MicroBlaze with an 8 KB data and 8 KB instruction cache is about 1.5 times faster than JOP. However, a 16 KB memory is not available in low-cost FPGAs and is an unbalanced system with respect to the LC/memory relation.

[5] As not all language constructs can be compiled, only the Kfl benchmark was measured. Therefore, the bars for MicroBlaze are missing in Figure 10.2.

	JOP	leJOS	TINI	Cjip	Komodo	aJ80	aJ100
iload iadd	2	836	789	55	8	38	8
iinc	8	422	388	46	4	41	11
ldc	9	1340	1128	670	40	67	9
if_icmplt taken	6	1609	1265	157	24	42	18
if_icmplt n/taken	6	1520	1211	132	24	40	14
getfield	22	1879	2398	320	48	142	23
getstatic	15	1676	4463	3911	80	102	15
iaload	36	1082	1543	139	28	74	13
invoke	128	4759	6495	5772	384	349	112
invoke static	100	3875	5869	5479	680	271	92
invoke interface	144	5094	6797	5908	1617	531	148

Table 10.5: Execution time in clock cycles for various JVM bytecodes

that each of these two operations is executed in a single cycle. The iinc bytecode is one of the few instructions that does not manipulate the stack and can be measured alone. As iinc is not implemented in hardware, we have a total of 8 cycles that are executed in microcode. It is fair to assume that this comprises too great an overhead for an instruction that is found in every iterative loop with an integer index. However, the decision to implement this instruction in microcode was derived from the observation that the dynamic instruction count for iinc is only 2% (see Section **??**).

The sequence for the branch benchmark (if_icmplt) contains the two load instructions that push the arguments onto the stack. The arguments are then consumed by the branch instruction. This benchmark verifies that a branch requires a constant four cycles on JOP, whether it is taken or not.

The Cjip implements the JVM with a stack oriented instruction set. It is the only example (except JOP) where the instruction set is documented *including* the execution time [54]. We will therefore check some of the results with the numbers provided in the documentation. The execution time is given in nanoseconds, assuming a 66 MHz clock. The execution time for the basic integer add operation is given as 180 ns resulting in 12 cycles. The load of a local variable (when it is one of the first four) takes 35 cycles. In the micro-benchmark we measure 55 cycles instead of the theoretical 47 (iadd + iload_n). We assume that we have to add some cycles for the fetch of the bytecodes from memory.

For compiling versions of the JVM, these micro-benchmarks do not produce useful re-

sults. The compiler performs optimizations that make it impossible to measure execution times at this fine a granularity.

During the evaluation of the aJile system, unexpected behavior was observed. The aJ80 on the JStamp board is clocked at 7.3728 MHz, and the internal frequency can be set with a PLL. The aJ80 is rated for 80MHz and the maximum PLL factor that can be used is therefore ten. Running the benchmarks with different PLL settings gave some strange results. For example, with a PLL multiplier setting of ten, the aJ80 was about 12.8 times faster! Other PLL factors also resulted in a greater than linear speedup. The only explanation we could find was that the internal time, System.currentTimeMillis(), used for the benchmarks depends on the PLL setting. A comparison with the wall clock time showed that the internal time of the aJ80 is 23% faster with a PLL factor of 1 and 2.4% faster with a factor of ten – a property we would not expect on a processor that is marketed for real-time systems.

The SaJe board is also clocked with 7.3728 MHz and the PLL factor is set to 14. This gives a 103.2192 MHz internal clock frequency. However, it is not known how accurate the internal time is in this setting. The results for the SaJe board can also suffer from the problem described above.

10.3.3 Execution Time Jitter

For real-time systems, the worst-case of the execution time is of primary importance. We have measured the execution times of several iterations of the main function from the Kfl benchmark. Figure 10.3 shows the measurements, scaled to the minimum execution time.

A period of four iterations can be seen. This period results from simulating the commands from the base station that are executed every fourth iteration. At iteration 10, a command to start the motor is issued. We see the resulting rise in execution time at iteration 12 to process this command. At iteration 54, the simulation triggers the end sensor and the motor is stopped.

The different execution times in the different modes of the application are inherent in the design of the simulation. However, the ratio between the longest and the shortest period is five for the JStamp, four for the gcj system, and only three for JOP. Therefore, a system with an aJile processor needs to be 1.7 times faster than JOP in order to provide the same WCET for this measurement. At iteration 33, we can see a higher execution time for the JStamp system that is not seen on JOP. This variation at iteration 33 is not caused by the benchmark.

The execution time under gcj on the Linux system showed some very high peaks (up to ten times the minimum, not shown in the figures). This observation was to be expected, as the gcj/Linux system is not a real-time solution. The Sun JIT-solution is omitted from the

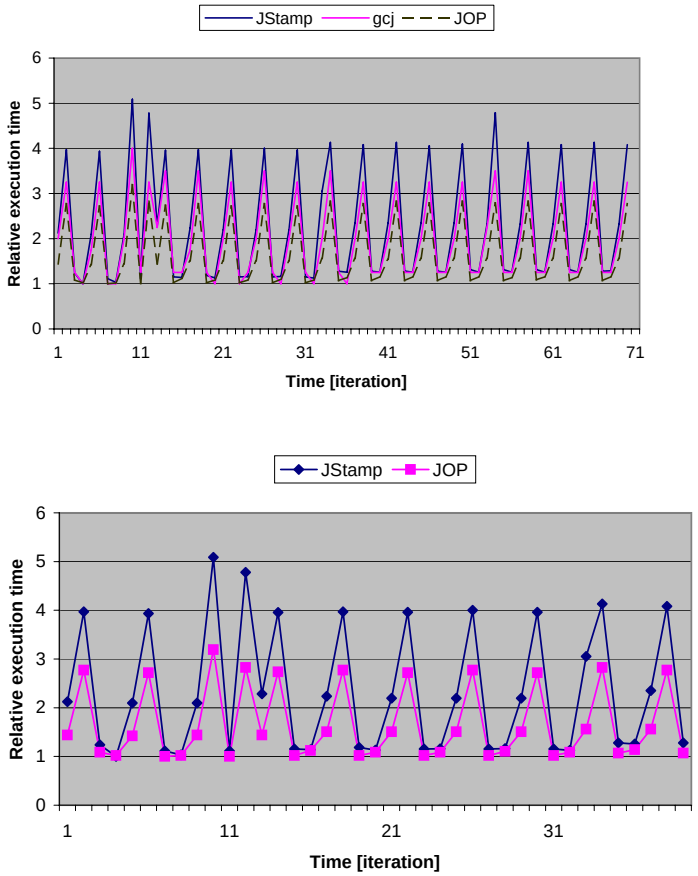

Figure 10.3: Execution time of the main function for the Kfl benchmark. The values are scaled to the minimum execution time. The bottom figure shows a detail of the top figure.

figure. As a result of the invocation of the compiler at some point during the simulation, the worst-case ratio between the maximum and minimum execution time was 1313 – showing that a JIT-compiler is impractical for real-time applications.

It should be noted that execution time measurement is not a safe method for obtaining WCET estimates. However, in situations where no WCET analysis tool is available, it can give some insight into the WCET behavior of different systems.

10.4 Applications

During the research for the PhD thesis, the first working version of JOP was used in a real-world application. Using an architecture under development in a commercial project entails risks. Nevertheless, this was deemed to be the best way to prove the feasibility of the processor. In this section, the experiences of the first project involving JOP are summarized.

10.4.1 Motor Control

In rail cargo, a large amount of time is spent on loading and unloading of goods wagons. The contact wire above the wagons is the main obstacle. Balfour Beatty Austria developed and patented a technical solution, the so-called *Kippfahrleitung*, to tilt up the contact wire. This is done on a line up to one kilometer. An asynchronous motor on each mast is used for this tilting. However, it has to be done synchronously on the whole line.

Each motor is controlled by an embedded system. This system also measures the position and communicates with a base station. Figure 10.4 shows the mast with the motor and the control system in the 'down' and 'up' positions. The base station has to control the deviation of individual positions during the tilt. It also includes the user interface for the operator. In technical terms, this is a distributed, embedded real-time control system, communicating over an RS485 network.

Real Hardware

Although this system is not mass-produced, there were nevertheless cost constraints. Even a small FPGA is more expensive than a general purpose CPU. To compensate for this, additional chips for the memory and the FPGA configuration were optimized for cost. One standard 128KB Flash was used to hold FPGA configuration data, the Java program and a logbook. External main memory was reduced to 128KB with an 8-bit data bus.

To reduce external components, the boot process is a little complicated. A watchdog circuit delivers a reset signal to a 32 macro-cell PLD. This PLD loads the configuration

Figure 10.4: Picture of a *Kippfahrleitung* mast in down and up position

data into the FPGA. When the FPGA starts, it disables the PLD and loads the Java program from the Flash into the external RAM. After the JVM is initialized, the program starts at main().

The motor is controlled by silicon switches connected to the FPGA with opto couplers. The position is measured with two end sensors and a revolving sensor. The processor supervises the voltage and current of the motor supply. A display and keyboard are attached to the base station for the user interface. The communication bus (up to one kilometer) is attached via an isolated RS485 data interface.

Synthesized Hardware

The following I/O modules were added to the JOP core in the FPGA:

- Timer
- UART for debugging
- UART with FIFO for the RS485 line

- Four sigma delta ADCs
- I/O ports

Five switches in the power line needed to be controlled by the program. A wrong setting of the switches due to a software error could result in a short circuit. Ensuring that this could not happen was a straightforward task at the VHDL level. The sigma-delta ADCs are used to measure the temperature of the silicon switches and the current through the motor.

Software Architecture

The main task of the program was to measure the position using the revolving sensor and communicate with the base station. This has to be done under real-time constraints. This is not a very complicated task. However, at the time of development, many features from a full-blown JVM implementation, such as threads or objects, were missing in JOP. The resulting Java was more like a *tiny Java*. It had to be kept in mind which Java constructs were supported by JOP. Because there was no multi-threading capability, and in the interests of simplicity, a simple infinite loop with constant time intervals was used. Listing 10.1 shows the simplified program structure. After initialization and memory allocation, this loop enters and never exits.

Communication

Communication is based on a client-server structure. Only the base station is allowed to send a request to a single mast station. The station is then required to reply. The maximum reply time is bounded by two time intervals. The base station handles timeout and retry. If an irrecoverable error occurs, the base station switches off the power to the mast stations, including the power supply to the motor. This is the safe state of the whole system.

From the mast station perspective, every mast station supervises the base station. The base station is required to send requests on a regular basis. If this requirement is violated, the mast station switches off its motor. The data is exchanged in small packets of four bytes, including a one-byte CRC. To simplify the development, commands to program the Flash in the mast stations and force a reset were included. It is therefore possible to update the program, or even change the FPGA configuration, over the network.

10.4.2 Further Projects

TAL, short for TeleAlarm, is a remote tele-control and data logging system. TAL communicates via a modem or an Ethernet bus with a SCADA system or via SMS with a mobile

```
public static void main(String[] args) {

    init();
    Timer.start();
    forever();
    // this point is NEVER reached
}

private static void forever() {

    for (;;) {
        Msg.loop();
        Triac.loop();
        if (Msg.available) {
            handleMsg();
        } else {
            chkMsgTimeout();
        }
        handleWatchDog();
        Timer.waitForNextInterval();
    }
}
```

Listing 10.1: Simplified program structure

phone. For this application, a minimal TCP/IP stack needed to be implemented. This stack was the reason for implementing threads and a simple real-time system in JOP.

Another application of JOP is in a communication device with soft real-time properties – Austrian Railways' (ÖBB) new security system for single-track lines. Each locomotive is equipped with a GPS receiver and a communication device. The position of the train, differential correction data for GPS, and commands are exchanged with a server at the central station over a GPRS virtual private network. JOP is the heart of the communication device in the locomotive. The flexibility of the FPGA and an Internet connection to the embedded system make it possible to upgrade the software and even the processor in the field.

10.5 Summary

In this chapter, we presented an evaluation of JOP. We have seen that JOP is the smallest hardware realization of the JVM available to date. Due to the efficient implementation of the stack architecture, JOP is also smaller than a *comparable* RISC processor in an FPGA. Implemented in an FPGA, JOP has the highest clock frequency of all known Java processors.

We compared JOP against several embedded Java systems and, as a reference, with Java on a standard PC. A Java processor is up to 500 times faster than an interpreting JVM on a standard processor for an embedded system. JOP is about six times faster than the aJ80 Java processor and as fast as the aJ100[6]. Preliminary results using compiled Java for a RISC processor in an FPGA, with a similar resource usage and maximum clock frequency to JOP, showed that native execution of Java bytecodes is faster than compiled Java.

We compared the basic properties of the real-time scheduler on JOP against the RTSJ implementation on Linux. The integration of the scheduler in the JVM, and the timer interrupt under scheduler control, results in an efficient platform for Java in embedded real-time systems. JOP performs better and more predictably than the reference implementation of the RTSJ under Linux.

We also performed WCET analysis of the implemented JVM at the microcode level. This analysis provides the WCET and BCET values for the individual bytecodes. We have also shown that there are no dependencies between individual bytecodes. This feature, in combination with the method cache (see Section 4.7), makes JOP an easy target for low-level WCET analysis of Java applications.

[6]The measured aJ100 system contained faster SRAMs than the FPGA board for JOP.

Usage of JOP in three real-world applications showed that the processor is mature enough to be used in commercial projects.

11 Related Work

Two different approaches can be found to improve Java bytecode execution by hardware. The first type operates as a Java coprocessor in conjunction with a general-purpose microprocessor. This coprocessor is placed in the instruction fetch path of the main processor and translates Java bytecodes to sequences of instructions for the host CPU or directly executes basic Java bytecodes. The complex instructions are emulated by the main processor. Java chips in the second category replace the general-purpose CPU. All applications therefore have to be written in Java. While the first type enables systems with mixed code capabilities, the additional component significantly raises costs. Table 11.1 provides an overview of the described Java hardware.

Blank fields in the table indicate that the information is not available or not applicable (e.g. for simulation-only projects). Minimum CPI is the number of clock cycles for a simple instruction such as `nop`. One entry, the TINI system, is not a real Java hardware, but is included in the table since it is often incorrectly[1] cited as an embedded Java processor.

11.1 Hardware Translation and Coprocessors

The simplest enhancement for Java is a translation unit, which substitutes the switch statement of an interpreter JVM (bytecode decoding) through hardware and/or translates simple bytecodes to a sequence of RISC instructions on the fly.

A standard JVM interpreter contains a loop with a large switch statement that decodes the bytecode (see Listing 3.1). This switch statement is compiled to an indirect branch. The destinations of these indirect branches change frequently and do not benefit from branch-prediction logic. This is the main overhead for simple bytecodes on modern processors. The following approaches enhance the execution of Java programs on a standard processor through the substitution of the memory read and switch statement with bytecode fetch and decode through hardware.

[1] TINI is a standard interpreting JVM running on an enhanced 8051 processor.

[2] J2ME CLDC stands for Java2 Micro Edition, Connected Limited Device Configuration, which is described in Section **??**.

	Type	Target technology	Size	Speed [MHz]	Java standard	Min. CPI
Hard-Int	Translation	Simulation only				
DELFT	Translation	Simulation only				
JIFFY	Translation	Xilinx FPGA	3800 LCs, 1KB RAM			
Jazelle	Co-processor	ASIC 0.18μ	12K gates	200		
JSTAR	Co-processor	ASIC 0.18μ Softcore	30K gates + 7KB	104	J2ME CLDC[2]	
TINI	Software JVM	Enhanced 8051 clone			Java 1.1 subset	
picoJava	Processor	No realization	128K gates + memory		Full	1
aJile	Processor	ASIC 0.25μ	25K gates + ROM	100	J2ME CLDC[2]	
Cjip	Processor	ASIC 0.35μ	70K gates + ROM, RAM	67	J2ME CLDC[2]	6
Ignite	Stack processor	Xilinx FPGA	9700 LCs			
Moon	Processor	Altera FPGA	3660 LCs, 4KB RAM			
Lightfoot	Processor	Xilinx FPGA	3400 LCs	40		
LavaCORE	Processor	Xilinx FPGA	3800 LCs 30K gates	20		
Komodo	Processor	Xilinx FPGA	2600 LCs	20	Subset: 50 bytecodes	4
FemtoJava	Processor	Altera Flex 10K	2000 LCs	4	Subset: 69 bytecodes, 16-bit ALU	3
JSM [24]	Processor	Xilinx FPGA		3.5	Java Card	

Table 11.1: Java hardware

11.1.1 Hard-Int

Radhakrichnan [94] proposes an additional architecture for a standard RISC processor to speed up a JVM interpreter. The architecture, called Hard-Int, is placed between the cache and instruction fetch of the RISC processor. Simple Java bytecodes are translated to a sequence of RISC instructions. For native RISC code, the unit is bypassed. This architecture implements the expensive switch statement of a typical interpreter in hardware. A simulation of a SPARC processor with four execution units shows a speedup by a factor of 2.6 over JDK 1.2 JIT with SPECjvm98. Since the architecture is only evaluated in a software simulation, the impact of the inserted hardware on the clock frequency of the RISC processor is unknown. No estimation of the additional hardware cost for the translation unit is given.

11.1.2 DELFT-JAVA Engine

In his thesis [41], Glossner describes a processor for multimedia applications in Java. A RISC processor is extended with DSP capabilities and Java specific instructions. This combination results in a very complex processor. Simple JVM instructions are dynamically translated to the DELFT instruction set. However, no explanation is given as to how this is done. A new register-addressing mode, indirect register addressing with auto increment or decrement, provides support for stack caching in the register file. The translation of JVM bytecode to the DELFT instruction set maps stack-based dependencies into pipeline dependencies. The author expects that these dependencies can be resolved with standard techniques such as register renaming and out-of-order execution. To accelerate dynamic linking, a link translation buffer cache resolves entries from the constant pool.

The processor is validated through a C++ model. An experiment with a synthetic benchmark (vector multiplication) compared a stack machine with an ideal register machine. The ideal register machine performs register renaming and out-of-order execution on multiple execution units. The achieved speedup in this experiment was 2.7. The high-level simulation model is more a proof of concept and no estimation is given for the resources needed to implement this complex processor. Since only a restricted subset of the JVM was simulated, no Java applications could be used to estimate the expected speedup.

11.1.3 JIFFY

An interesting approach to enhance Java execution in embedded systems is presented in Acher's thesis [1]. He states that JIT-compilation in software is not possible on most em-

bedded devices because of resource constraints. JIFFY, a JIT in an FPGA, is proposed as a solution to this problem. The compilation is done in the following steps:

The Java bytecode is translated into an intermediate language with three registers and a stack. The reduction to three registers is due to the fact that bytecodes are using a maximum of three stack operands, and it simplifies translation to CISC-architectures with a low register count. In the next step, this instruction sequence, which is still stack-based, is optimized. The main effect of this optimization is to transform stack-based operations into register-based operations. These optimized instructions in the intermediate language are translated to native instructions of the target architecture in the last step.

The quality of the generated code was tested with software versions of JIFFY for a CISC (80586) and a RISC (Alpha 21164) architecture. The resulting code is about 1.1 to 7.5 times faster than interpreting Java bytecode on the x86 architecture. The speedup is similar to Sun's first JIT compiler (sunwjit in JDK 1.1). The compilation time is estimated to be 50 to 70 clock cycles for one bytecode. This is 10 times faster than the efficient CACAO JIT [61]. A first prototype implementation in an FPGA used 3800 LCs and 8KBits RAM (80 % of a Xilinx XC2S200).

11.1.4 Jazelle

Jazelle [9] is an extension of the ARM 32-bit RISC processor, similar to the Thumb state (a 16-bit mode for reduced memory consumption). The Jazelle coprocessor is integrated into the same chip as the ARM processor. The hardware bytecode decoder logic is implemented in less than 12K gates. It accelerates, according to ARM, some 95% of the executed bytecodes. 140 bytecodes are executed directly in hardware, while the remaining 94 are emulated by sequences of ARM instructions. This solution also uses code modification with *quick* instructions to substitute certain object-related instructions after link resolution. All Java bytecodes, including the emulated sequences, are re-startable to enable a fast interrupt response time.

A new ARM instruction puts the processor into the Java state. Bytecodes are fetched and decoded in two stages, compared to a single stage in ARM state. Four registers of the ARM core are used to cache the top stack elements. Stack spill and fill is handled automatically by the hardware. Additional registers are reused for the Java stack pointer, the variable pointer, the constant pool pointer and locale variable 0 (the *this* pointer in methods). Keeping the complete state of the Java mode in ARM registers simplifies its integration into existing operating systems.

11.1.5 JSTAR, JA108

Nozomi's JA108 [73], previously known as JSTAR, is a Java coprocessor that sits between the native processor and the memory subsystem. JA108 fetches Java bytecodes from memory and translates them into native microprocessor instructions. JA108 acts as a pass-through when the core processor's native instructions are being executed. The JA108 is targeted for use in mobile phones to increase performance of Java multimedia applications. The coprocessor is available as standalone package or with included memory and can be operated up to 104 MHz. The resource usage for the JSTAR is known to be about 30K gates plus 45 Kbits for the microcode.

11.1.6 A Co-Designed Virtual Machine

In his thesis [58], Kent proposes an interesting new form of Java coprocessor. He investigates hardware/software co-design for a JVM within the context of a desktop workstation. The execution of the JVM is partitioned between an FPGA and the host processor. An FPGA board with local memory is connected via the PCI bus to the host. This solution provides an add-on accelerator without changing the system. Moreover, as the FPGA can be configured for a different task, the add-on hardware can be used for non-Java applications.

The critical issue in this approach is the partitioning of the JVM and the memory regions between hardware and software. Not all Java bytecodes can be executed in hardware. All object-oriented bytecodes are performed in software. However, once these bytecodes are replaced by their *quick* variants, some of them can then be executed in hardware. The most accessed data structures, i.e. the method's bytecode, execution stack, and local variables, are placed in the FPGA board memory. The constant pool and the heap reside in the PC's main memory. The software part of the JVM decides during runtime which instruction sequences can be executed by the hardware. Due to the high cost of a context switch, this is a critical decision. Kent explored various algorithms with different block sizes to find the optimum partitioning of the instructions between the host processor and the FPGA. Tests with small benchmarks on a simulation showed performance gains by a factor of 6 to 11, when compared with an interpreting JVM. Kent is now working on the concurrent use of the FPGA and the host system to execute Java applications. Additional performance increases are expected for multi-threaded applications.

In our view, there are two potential problems with this approach. Firstly, the execution context for the hardware is too small. As invokevirtual and the quick version are implemented in the software partition, the maximum context is one method body. As shown in Section **??**, Java methods are usually small (about 30% are less than 9 bytes long), resulting

in many context switches. The second issue is the raw speedup, without communication overhead, of the FPGA solution. This speedup is stated to be around of 10 times greater, with the same clock frequency. However, FPGA clock rate will never reach the clock rate of a general-purpose processor. With a meaningful design, such as a CPU, the clock rate of an FPGA is about 20 to 50 times lower. However, everyone who uses an FPGA as target technology for a processor design faces this problem. It is better not to try to compete against mainstream PC technology.

11.2 Java Processors

Java Processors are primarily used in an embedded system. In such a system, Java is the native programming language and all operating system related code, such as device drivers, are implemented in Java. Java processors are simple or extended stack architectures with an instruction set that resembles more or less the bytecodes from the JVM.

11.2.1 picoJava

Sun's picoJava is the Java processor most often cited in research papers. It is used as a reference for new Java processors and as the basis for research into improving various aspects of a Java processor. Ironically, this processor was never released as a product by Sun. After Sun decided to not produce picoJava in silicon, Sun licensed picoJava to Fujitsu, IBM, LG Semicon and NEC. However, these companies also did not produce a chip and Sun finally provided the full Verilog code under an open-source license.

Sun introduced the first version of picoJava [77] in 1997. The processor was targeted at the embedded systems market as a pure Java processor with restricted support of C. picoJava-I contains four pipeline stages. A redesign followed in 1999, known as picoJava-II. This is the version described below. picoJava-II is now freely available with a rich set of documentation [125, 126].

Simple Java bytecodes are directly implemented in hardware; most of them execute in one to three cycles. Other performance critical instructions, for instance invoking a method, are implemented in microcode. picoJava traps on the remaining complex instructions, such as creation of an object, and emulates the instruction. To access memory, internal registers, and for cache management, picoJava implements 115 extended instructions with 2-byte opcodes. These instructions are necessary to write system-level code to support the JVM.

Traps are generated on interrupts, exceptions, and for instruction emulation. A trap is rather expensive and has a minimum overhead of 16 clock cycles:

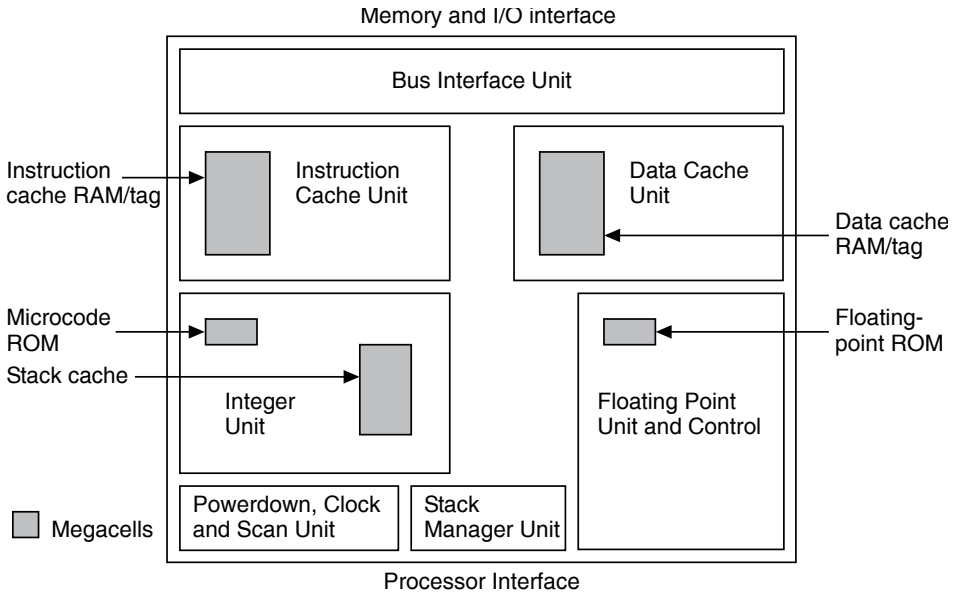

Figure 11.1: Block diagram of picoJava-II (from [125])

```
6 clocks trap execution
n clocks trap code
2 clocks set VARS register
8 clocks return from trap
```

This minimum value can only be achieved if the trap table entry is in the data cache and the first instruction of the trap routine is in the instruction cache. The worst-case interrupt latency is 926 clock cycles [126].

Figure 11.1 shows the major function units of picoJava. The integer unit decodes and executes picoJava instructions. The instruction cache is direct-mapped, while the data cache is two-way set-associative, both with a line size of 16 bytes. The caches can be configured between 0 and 16 Kbytes. An instruction buffer decouples the instruction cache from the decode unit. The FPU is organized as a microcode engine with a 32-bit datapath supporting single- and double-precision operations. Most single-precision operations require four cycles. Double-precision operations require four times the number of cycles as single-precision operations. For low-cost designs, the FPU can be removed and the core traps on floating-point instructions to a software routine to emulate these instructions. picoJava

provides a 64-entry stack cache as a register file. The core manages this register file as a circular buffer, with a pointer to the top of stack. The stack management unit automatically performs spill to and fill from the data cache to avoid overflow and underflow of the stack buffer. To provide this functionality the register file contains five memory ports. Computation needs two read ports and one write port, the concurrent spill and fill operations the two additional read and write ports. The processor core consists of following six pipeline stages:

Fetch: Fetch 8 bytes from the instruction cache or 4 bytes from the bus interface to the 16-byte-deep prefetch buffer.

Decode: Group and precode instructions (up to 7 bytes) from the prefetch buffer. Instruction folding is performed on up to four bytecodes.

Register: Read up to two operands from the register file (stack cache).

Execute: Execute simple instructions in one cycle or microcode for multi-cycle instructions.

Cache: Access the data cache.

Writeback: Write the result back into the register file.

The integer unit together with the stack unit provides a mechanism, called instruction folding, to speed up common code patterns found in stack architectures, as shown in Figure 11.2. When all entries are contained in the stack cache, the picoJava core can fold these four instructions into one RISC-style single cycle operation.

picoJava contains a simple mechanism to speed-up the common case for monitor enter and exit. The two low order bits of an object reference are used to indicate the lock holding or a request to a lock held by another thread. These bits are examined by monitorenter and monitorexit. For all other operations on the reference, these two bits are masked out by the hardware. Hardware registers cache up to two locks held by a single thread.

To efficiently implement a generational or an incremental garbage collector, picoJava offers hardware support for write barriers through memory segments. The hardware checks all stores of an object reference if this reference points to a different segment (compared to the store address). In this case, a trap is generated and the garbage collector can take the appropriate action. Additional two reserved bits in the object reference can be used for a write barrier trap.

The architecture of picoJava is a stack-based CISC processor implementing 341 different instructions [77] and is the most complex Java processor available. The processor can

A Java instruction

```
c = a + b;
```

translates to the following bytecodes:

```
iload_1
iload_2
iadd
istore_3
```

Figure 11.2: A common folding pattern that is executed in a single cycle

be implemented [31] in about 440K gates (128K for the logic and 314K for the memory components: 284x80 bits microcode ROM, 2x192x64 bits FPU ROM and 2x16 KB caches). We have implemented picoJava-II in a Cyclone-II FPGA [89] and the design consumed 27500 LCs and 48 KB on-chip memory.

11.2.2 aJile JEMCore

aJile's JEMCore is a direct-execution Java processor that is available as both an IP core and a stand alone processor [2, **?**]. It is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. JEM2 is an enhanced version of JEM1, created in 1997 by the Rockwell-Collins Advanced Architecture Microprocessor group. Rockwell-Collins originally developed JEM for avionics applications by adapting an existing design for a stack-based embedded processor. Rockwell-Collins decided not to sell the chip on the open market. Instead, it licensed the design exclusively to aJile Systems Inc., which was founded in 1999 by engineers from Rockwell-Collins, Centaur Technologies, Sun Microsystems, and IDT.

The core contains 24 32-bit wide registers. Six of them are used to cache the top elements of the stack. The datapath consists of a 32-bit ALU, a 32-bit barrel shifter and the support for floating point operations (disassembly/assembly, overflow and NaN detection). The control store is a 4K by 56 ROM to hold the microcode that implements the Java bytecode. An additional RAM control store can be used for custom instructions. This feature is used to implement the basic synchronization and thread scheduling routines in microcode. It results in low execution overhead with a thread-to-thread yield in less than one μs (at

100 MHz). An optional Multiple JVM Manager (MJM) supports two independent, memory protected JVMs. The two JVMs execute time-sliced on the processor. According to aJile, the processor can be implemented in 25K gates (without the microcode ROM). The MJM needs additional 10K gates.

Two silicon versions of JEM exist today: the aJ-80 and the aJ-100. Both versions comprise a JEM2 core, the MJM, 48KB zero wait state RAM and peripheral components, such as timer and UART. 16KB of the RAM is used for the writable control store. The remaining 32KB is used for storage of the processor stack. The aJ-100 provides a generic 8-bit, 16-bit or 32-bit external bus interface, while the aJ-80 only provides an 8-bit interface. The aJ-100 can be clocked up to 100MHz and the aJ-80 up to 66MHz. The power consumption is about 1mW per MHz.

Since aJile was a member of the Real-Time for Java Expert Group, the complete RTSJ will be available in the near future. One nice feature of this processor is its availability. A relatively cheap development system, the JStamp [127], was used to compare this processor with JOP.

11.2.3 Cjip

The Cjip processor [46, 55] supports multiple instruction sets, allowing Java, C, C++, and assembler to coexist. Internally, the Cjip uses 72 bit wide microcode instructions, to support the different instruction sets. At its core, Cjip is a 16-bit CISC architecture with on-chip 36KB ROM and 18KB RAM for fixed and loadable microcode. Another 1KB RAM is used for eight independent register banks, string buffer and two stack caches. Cjip is implemented in 0.35-micron technology and can be clocked up to 66MHz. The logic core consumes about 20% of the 1.4-million-transistor chip. The Cjip has 40 program controlled I/O pins, a high-speed 8 bit I/O bus with hardware DMA and an 8/16 bit DRAM interface.

The JVM is implemented largely in microcode (about 88% of the Java bytecodes). Java thread scheduling and garbage collection are implemented as processes in microcode. Microcode is also used to implement virtual peripherals such as watchdog timers, display and keyboard interfaces, sound generators, and multimedia codecs.

Microcode instructions execute in two or three cycles. A JVM bytecode requires several microcode instructions. The Cjip Java instruction set and the extensions are described in detail in [54]. For example: a bytecode nop executes in 6 cycles while an iadd takes 12 cycles. Conditional bytecode branches are executed in 33 to 36 cycles. Object oriented instructions, such getfield, putfield, or invokevirtual are not part of the instruction set.

11.2.4 Ignite, PSC1000

The PSC1000 [88] is a stack processor, based on ShBoom (originally designed by Chuck Moore [71]), designed for high speed Forth applications. The PSC1000 was later renamed to Ignite and promoted as a Java processor, though it has roots in Forth. The instruction set, called ROSC (Removed Operand Set Computer), is different from Java bytecodes. A small JVM driver converts Java bytecode into the stack instruction set of the processor.

The processor contains two on-chip stacks, as usual in Forth processors [60], and additional 16 global registers. The first elements of the stacks are directly accessible. The bottleneck of instruction fetching without a cache is avoided by fetching up to four 8-bit instructions from a 32-bit memory. To simplify instruction decoding, immediate values and branch offsets are placed right-aligned in such an instruction group. The PSC1000 is available as an ASIC at 80 MHz and as a soft-core for Xilinx FPGAs (9700 LCs).

11.2.5 Moon

Vulcan ASIC's Moon processor is an implementation of the JVM to run in an FPGA. The execution model is the often-used mix of direct, microcode and trapped execution. As described in [129], a simple stack folding is implemented in order to reduce five memory cycles to three for instruction sequences like *push-push-add*. The first version of Moon uses 3.840 LCs and 10 embedded memory blocks in an Altera FPGA. The Moon2 processor [130] is available as an encrypted HDL source for Altera FPGAs (22% of an APEX 20K400E equates to 3660 LCs) or as VHDL or Verilog source code. The minimum silicon cost is given as 27K gates plus 3KB ROM and 1KB single port RAM. The single port RAM is used to implement 256 entries of the stack.

11.2.6 Lightfoot

The Lightfoot 32-bit core [28] is a hybrid 8/32-bit processor based on the Harvard architecture. Program memory is 8 bits wide and data memory is 32 bits wide. The core contains a 3-stage pipeline with an integer ALU, a barrel shifter, and a 2-bit multiply step unit. There are two different stacks with the top elements implemented as registers and memory extension. The data stack is used to hold temporary data – it is not used to implement the JVM stack frame. As the name implies, the return stack holds return addresses for subroutines and it can be used as an auxiliary stack. The TOS element is also used to access memory. The processor architecture specifies three different instruction formats: soft bytecodes, non-returnable instructions, and single-byte instructions that can be folded with a return instruction. Soft bytecode instructions cause the processor to branch to one of 128 locations

in low program memory, where the implementation of the soft bytecodes resides. This operation has a single cycle overhead and the address of the following instruction is pushed onto the return stack. The instruction set implies that it is optimized to write an efficient interpreted JVM.

The core is available in VHDL and can be implemented in less than 30K gates. According to DCT, the performance is typically 8 times better than RISC interpreters running at the same clock speed. The core is also provided as an EDIF netlist for dedicated Xilinx devices. It needs 1710 CLBs (= 3400 LCs) and 2 Block RAMs. In a Vertex-II (2V1000-5), it can be clocked up to 40MHz.

11.2.7 LavaCORE

LavaCORE [30] is another Java processor targeted at Xilinx FPGA architectures. It implements a set of instructions in hardware and firmware. Floating-point operations are not implemented. A 32x32-bit dual-ported RAM implements a register-file. For specialized embedded applications, a tool is provided to analyze which subset of the JVM instructions is used. The unused instructions can be omitted from the design. The core can be implemented in 1926 CLBs (= 3800 LCs) in a Virtex-II (2V1000-5) and runs at 20MHz.

11.2.8 Komodo

Komodo [137] is a multithreaded Java processor with a four-stage pipeline. It is intended as a basis for research on real-time scheduling on a multithreaded microcontroller [63]. Simple bytecodes are directly implemented, while more complex bytecodes, such as `iaload`, are implemented as a microcode sequence. The unique feature of Komodo is the instruction fetch unit with four independent program counters and status flags for four threads. A priority manager is responsible for hardware real-time scheduling and can select a new thread after each bytecode instruction.

The first version of Komodo in an FPGA implements a very restricted subset of the JVM (only 50 bytecodes). The design can be clocked at 20MHz. However, the pipeline runs at 5MHz for single cycle external memory access and three-port access of stack memory in one pipeline stage. The resource usage is 1300 CLBs (= 2600 LCs) in a Xilinix XC 4036 XL.

11.2.9 FemtoJava

FemtoJava [56] is a research project to build an application specific Java processor. The bytecode usage of the embedded application is analyzed and a customized version of FemtoJava is generated. FemtoJava implements up to 69 bytecode instructions for an 8 or 16 bit datapath. These instructions take 3, 4, 7 or 14 cycles to execute. Analysis of small applications (50 to 280 byte code) showed that between 22 and 69 distinct bytecodes are used. The resulting resource usage of the FPGA varies between 1000 and 2000 LCs. With the reduction of the datapath to 16 bits the processor is not Java conformant.

11.2.10 jHISC

The jHISC project [128] proposes a high-level instruction set architecture for Java. This project is closely related to picoJava. The processor consumes 15500 LCs in an FPGA and the maximum frequency in a Xilinx Virtex FPGA is 30 MHz. According to [128] the prototype can only run simple programs and the performance is estimated with a simulation. In [135] the clocks per instruction (CPI) values for jHISC are compared against picoJava and JOP. However, it is not explained with which application the CPI values are collected. We assume that the CPI values for picoJava and JOP are derived from the manual and do not include any effects of pipeline stalls or cache misses.

11.3 Additional Comments

The two classes of hardware accelerators for Java can be further subdivided as shown in Figure 11.3. Many of the Java processors are stack machines that have been derived from Forth processors. Two different stacks in these so-called Java processors (Cjip, Ignite and Lightfoot) do not fit very well for the JVM. Although stack based, Forth is different from Java bytecode. Instruction mix in Forth shows about 25% call and returns [60], so Forth processors are optimized for fast call and return. In Java, the percentage of call/return is only about 6% (see Section **??**). With subroutine exits so common, it is no wonder that most of the Forth stack machines have a mechanism for combining subroutine exits with other instructions and provide two stacks to avoid the mixture of parameters and return addresses. However, a JVM stack frame is more complex than in Forth (see Section 4.4) and there is no use for such a mechanism. An additional return stack provides no advantage for the JVM.

In Forth only the top elements can be accessed, which results in a simple stack design with only one access port. In the JVM, parameters for a method are explicitly pushed onto the stack before invocation. These parameters are then accessed in the method relative to

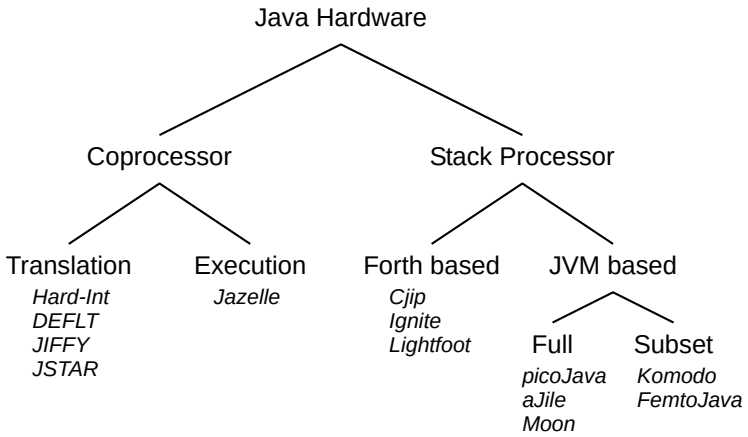

Figure 11.3: Java hardware

a variable pointer. This mechanism needs a dual ported memory with simultaneous read and write access. These basic differences between Forth and the JVM lead to a sub-optimal implementation of the JVM on a Forth based processor.

There are problems in getting information about commercial products. When new companies started developing Java processors, a lot of information was available. This information was usually more of a presentation of the concept; nevertheless it gave some insights into how they approached the different design problems. However, at the point at which the projects reached production quality, this information quietly disappeared from their websites. It was replaced with colorful marketing prospectuses about the wonderful world of the new Java-enabled mobile phones. Only one company, aJile Ltd., presented information about their product in a refereed conference paper.

Many research projects for a Java processor in an FPGA exists. Examples can be found in [56], [59] and [72]. These projects have much in common – the basic implementation of a stack machine with integer instructions is easy. However, the realization of the complete JVM is the hard part and therefore beyond the scope of these projects.

Other than the aJile processor and the Komodo project, no solution addresses the problem of real-time predictability. For this reason, as well as its availability, the aJile processor is used for comparison with JOP.

11.4 Summary

In Table 11.2, features of selected Java processors are compared. The category 'Predictability' means how well the processor is time-predictable. In the category 'Size', the chip size is estimated, and the category 'Performance' means average performance. The category 'JVM conformance' lists how complete the implementation of the JVM specification [69] is. The 'Flexibility' parameter indicates how well the processor can be adapted to different application domains.

The assessment of the various parameters is, however, somewhat subjective as the information is mainly derived from written documentation. In Section 10.3, the overall performance of various Java systems, including the aJile processor, is compared with JOP.

The last column of the table shows the features required for JOP. This is, therefore, our research objective in a nutshell.

	picoJava	aJile	Komodo	FemtoJava	JOP
Predictability	−−	·	−	·	++
Size	−−	−	+	−	++
Performance	++	+	−	−−	+
JVM conformance	++	+	−	−−	·
Flexibility	−−	−−	+	++	++

Table 11.2: Feature comparison of selected Java processors

Due to the great variation in execution times for a trap, picoJava is given a double minus in the 'Predictability' category. picoJava is also the largest processor in the list. However, its performance and JVM compatibility are expected to be superior to those of other processors.

The aJile processor is intended as a solution for real-time systems. However, no information is available about bytecode execution times. As this processor is a commercial product and has been on the market for some time, it is expected that its JVM implementation would conform to Java standards, as defined by Sun.

Komodo's multithreading is similar to hyper-threading in modern processors that are trying to hide latencies in instruction fetching. However, this feature leads to very pessimistic WCET values (in effect rendering the performance gain useless). The fact that the pipeline clock is only a quarter of the system clock also wastes a considerable amount of potential performance.

FemtoJava is given a double plus for flexibility, due to the application-dependent generation of the processor. However, FemtoJava is only a 16-bit processor and therefore not JVM compliant. The resource usage is also very high, compared to the minimal Java subset implemented and the low performance of the processor.

So far, all processors in the list perform weakly in the area of time-predictable execution of Java bytecodes. However, a low-level analysis of execution times is of primary importance for WCET analysis. Therefore, the main objective of JOP is to define and implement a processor architecture that is as predictable as possible. However, it is equally important that this does not result in a low performance solution. Performance shall not suffer as a result of the time-predictable architecture.

The second main aim of this work is to design a small processor. Size and the resulting energy consumption are a main concern in embedded systems. The proposed Java processor needs to be small enough to be implemented in a low-cost FPGA device. With this constraint, an implementation in an ASIC will also result in a very small core that can be part of a larger system-on-a-chip.

The embedded market is diverse and one size does not fit all. A configurable processor in which we can trade size for performance provides the flexibility for a variety of application domains. The aim of the architecture of JOP is to support this flexibility.

12 Summary

In this chapter we will undertake a short review of the project and summarize the contributions. Java for real-time systems is a very new and active research area. This chapter offers suggestions for future research, based on the proposed Java processor.

The research contributions made by this work are related to two areas: real-time Java and resource-constrained embedded systems.

12.1 A Real-Time Java Processor

The goal of time-predictable execution of Java programs was a first-class guiding principle throughout the development of JOP:

- The execution time for Java bytecodes can be exactly predicted in terms of the number of clock cycles. JOP is therefore a straightforward target for low-level WCET analysis. There is no mutual dependency between consecutive bytecodes that could result in unbounded timing effects.

- In order to provide time-predictable execution of Java bytecodes, the processor pipeline is designed without any prefetching or queuing. This fact avoids hard-to-analyze and possibly unbounded pipeline dependencies. There are no pipeline stalls, caused by interrupts or the memory subsystem, to complicate the WCET analysis.

- A pipelined processor architecture calls for higher memory bandwidth. A standard technique to avoid processing bottlenecks due to the higher memory bandwidth is caching. However, standard cache organizations improve the average execution time but are difficult to predict for WCET analysis. Two time-predictable caches are proposed for JOP: a *stack cache* as a substitution for the data cache and a *method cache* to cache the instructions.

  As the stack is a heavily accessed memory region, the stack – or part of it – is placed in local memory. This part of the stack is referred to as the *stack cache* and described

in Section 4.4. Fill and spill of the stack cache is subjected to microcode control and therefore time-predictable.

In Section 4.7, a novel way to organize an instruction cache, as *method cache*, is given. The cache stores complete methods, and cache misses only occur on method invocation and return. Cache block replacement depends on the call tree, instead of instruction addresses. This *method cache* is easy to analyze with respect to worst-case behavior and still provides substantial performance gain when compared to a solution without an instruction cache.

- The time-predictable processor described above provides the basis for real-time Java. The issues with standard Java and the Real-Time Specification for Java were analyzed in Chapter **??**. To enable real-time Java to operate on resource-constrained devices, a simple real-time profile was defined in Section 5.3 and implemented in Java on JOP. The beauty of this approach is in implementing functions usually associated with an RTOS in Java. This means that real-time Java is not based on an RTOS, and therefore not restricted to the functionality provided by the RTOS. With JOP, a self-contained real-time system in pure Java becomes possible.

  The tight integration of the scheduler and the hardware that generates schedule events results in low latency and low jitter of the task dispatch.

- The defined real-time profile suggests a new way to handle hardware interrupts to avoid interference between blocking device drivers and application tasks. Hardware interrupts other than the timer interrupt are represented as asynchronous events with an associated thread. These events are *normal* schedulable objects and subject to the control of the scheduler. With a minimum interarrival time, these events, and the associated device drivers, can be incorporated into the priority assignment and schedulability analysis in the same way as normal application tasks.

The contributions described above result in a time-predictable execution environment for real-time applications written in Java, without the resource implications and unpredictability of a JIT-compiler. The proposed processor architecture is a straightforward target for low-level WCET analysis.

Implementing a real-time scheduler in Java opens up new possibilities. The scheduler is extended to provide a framework for user-defined scheduling in Java. In Section 5.2, we analyzed which events are exposed to the scheduler and which functions from the JVM need to be available in the user space. A simple-to-use framework to evaluate new scheduling concepts is given.

12.2 A Resource-Constrained Processor

Embedded systems are usually very resource-constrained. Using a low-cost FPGA as the main target technology forced the design to be small. The following architectural features address this issue:

- The architecture of JOP is best described as:

  > The JVM is a CISC stack architecture, whereas JOP is a RISC stack architecture.

  JOP contains its own instruction set, called microcode in this handbook, with a novel way of mapping bytecodes to microcode addresses. This mapping has zero overheads as described in Section 4.2. Basic bytecode instructions have a one-to-one mapping to microcode instructions and therefore execute in a single cycle. The stack architecture allows compact encoding of microinstructions in 8 bits to save internal memory.

  This approach allows flexible implementation of Java bytecodes in hardware, as a microcode sequence, or even in Java itself.

- The analysis of the JVM stack usage pattern in Section 4.4 led to the design of a resource-efficient two-level stack cache. This two-level stack cache fits to the embedded memory technologies of current FPGAs and ASICs and ensures fast execution of basic instructions.

  Part of the stack cache, which is implemented in an on-chip memory, is also used for microcode variables and constants. This resource sharing not only reduces the number of memory blocks needed for the processor, but also the number of data paths to and from the execution unit.

- Interrupts are considered hard to handle in a pipelined processor, resulting in a complex (and therefore resource consuming) implementation. In JOP, the above mentioned bytecode-microcode mapping is used in a clever way to avoid interrupt handling in the core pipeline. Interrupts generate special bytecodes that are inserted in a transparent way in the bytecode stream. Interrupt handlers can be implemented in the same way as bytecodes are implemented: in microcode or in Java.

The above design decisions where chosen to keep the size of the processor small without sacrificing performance. JOP is the smallest Java processor available to date that provides the basis for an implementation of the CLDC specification (see Section **??**). JOP is a fast execution environment for Java, without the resource implications and unpredictability of

a JIT-compiler. The average performance of JOP is similar to that of mainstream, non real-time Java systems.

JOP is a flexible architecture that allows different configurations for different application domains. Therefore, size can be traded against performance. As an example, resource intensive instructions, such as floating point operations, can be implemented in Java. The flexibility of an FPGA implementation also allows adding application-specific hardware accelerators to JOP.

The small size of the processor allows the use of low-cost FPGAs in embedded systems that can compete against standard microcontroller. JOP has been implemented in several different FPGA families and is used in different real-world applications.

Programs for embedded and real-time systems are usually multi-threaded and a small design provides a path to a multi-processor system in a mid-sized FPGA or in an ASIC.

A tiny architecture also opens new application fields when implemented in an ASIC. Smart sensors and actuators, for example, are very sensitive to cost, which is proportional to the die area.

12.3 Future Research Directions

JOP provides a basis for various directions for future research. Some suggestions are given below:

Real-time garbage collector: In Section 6, a real-time garbage collector was presented. Hardware support of a real-time GC would be an interesting topic for further research.

Another question that remains with a real-time GC is the analysis of the worst-case memory consumptions of tasks (similar to the WCET values), and scheduling the GC so that it can keep up with the allocation rate.

Hardware accelerator: The flexibility of an FPGA implementation of a processor opens up new possibilities for hardware accelerators. We have shown in Section 4.5 how the implementation of a bytecode can be moved between hardware and software. A further step would be to generate an application specific-system in which part of the application code is moved to hardware. Ideally, the hardware description should be extracted automatically from the Java source. Preliminary work in this area, using JOP as its basis, can be found in [44].

Hardware scheduler: In JOP, scheduling and dispatch is done in Java (with some microcode support). For tasks with very short periods, the scheduling overheads can

prove to be too high. A scheduler implemented in hardware can shorten this time, due to the parallel nature of the algorithm.

Multiprocessor JVM: In order to generate a small and predictable processor, several advanced and resource-consuming features (such as instruction folding or branch prediction) were omitted from the design. The resulting low resource usage of JOP makes it possible to integrate more than one processor in an FPGA. Since embedded applications are naturally multi-threaded systems, the performance can easily be enhanced using a multi-processor solution. A multi-processor JVM with shared memory offers research possibilities such as: scheduling of Java threads and synchronization between the processors or WCET analysis for the shared memory access.

Initial results on a JOP CMP are described in [83, 84].

Instruction cache: The cache solution proposed in Section 4.7 provides predictable instruction cache behavior while, in the average case, still performing in a similar way to a direct-mapped cache. However, an analysis tool for the worst-case behavior is still needed. With this tool, and a more complex analysis tool for traditional instruction caches, we also need to verify that the worst-case miss penalty is lower than with a traditional instruction cache.

A second interesting aspect of the proposed method cache is the fact that the replacement decision on a cache miss only occurs on method invoke and return. The infrequency of this decision means that more time is available for more advanced replacement algorithms.

Real-time Java: Although there is already a definition for real-time Java, i.e. the RTSJ [21], this definition is not necessarily adequate. There is ongoing research on how memory should be managed for real-time Java applications: scoped memory, as suggested by the RTSJ, usage of a real-time GC, or application managed memory through memory pools. However, almost no research has been done into how the Java library, which is major part of Java's success, can be used in real-time systems or how it can be adapted to do so. The question of what the best memory management is for the Java standard library remains unanswered.

Java computer: How would a processor architecture and operating system architecture look in a 'Java only' system? Here, we need to rethink our approach to processes, protection, kernel- and user-space, and virtual memory. The standard approach of using memory protection between different processes is necessary for applications

that are programmed in languages that use memory addresses as data, i.e. pointer usage and pointer manipulation. In Java, no memory addresses are visible and pointer manipulation is not possible. This very important feature of Java makes it a *safe* language. Therefore, an error-free JVM means we do not need memory protection between processes and we do not need to make a distinction between kernel and user space (with all the overhead) in a Java system. Another reason for using virtual addresses is link addresses. However, in Java this issue does not exist, as all classes are linked dynamically and the code itself (i.e. the bytecodes) only uses relative addressing.

Another issue here is the paging mechanism in a virtual memory system, which has to be redesigned for a Java computer. For this, we need to merge the virtual memory management with the GC. It does not make sense to have a virtual memory manager that works with plain (e.g. 4 KB) memory pages without knowledge about object lifetime. We therefore need to incorporate the virtual memory paging with a generational GC. The GC knows which objects have not been accessed for a long time and can be swapped out to the hard disk. Handling paging as part of the GC process also avoids page fault exceptions and thereby simplifies the processor architecture.

Another question is whether we can substitute the process notation with threads, or whether we need several JVMs on a Java only system. It depends. If we can live with the concept of shared static class members, we can substitute heavyweight processes with lightweight threads. It is also possible that we would have to define some further thread local data structures in the system.

It is the opinion of the author that Java is a promising language for future real-time systems. However, a number of issues remain to be solved. JOP, with its time-predictable execution of Java bytecodes, is an important part of a real-time Java system.

A Publications

- Martin Schoeberl. Using a Java Optimized Processor in a Real World Application. In *Proceedings of the First Workshop on Intelligent Solutions in Embedded Systems (WISES 2003)*, pages 165–176, Austria, Vienna, June 2003.
- Martin Schoeberl. Design Decisions for a Java Processor. In *Tagungsband Austrochip 2003*, pages 115–118, Linz, Austria, October 2003.
- Martin Schoeberl. JOP: A Java Optimized Processor. In R. Meersman, Z. Tari, and D. Schmidt, editors, *On the Move to Meaningful Internet Systems 2003: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2003)*, volume 2889 of *Lecture Notes in Computer Science*, pages 346–359, Catania, Italy, November 2003. Springer.
- Martin Schoeberl. Restrictions of Java for Embedded Real-Time Systems. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 93–100, Vienna, Austria, May 2004.
- Martin Schoeberl. Design Rationale of a Processor Architecture for Predictable Real-Time Execution of Java Programs. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.
- Martin Schoeberl. Real-Time Scheduling on a Java Processor. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.
- Martin 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.
- Martin Schoeberl. A Time Predictable Instruction Cache for a Java Processor. In Robert Meersman, Zahir Tari, and Angelo Corsario, editors, *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and*

Embedded Systems (JTRES 2004), volume 3292 of *Lecture Notes in Computer Science*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

- Flavius Gruian, Per Andersson, Krzysztof Kuchcinski, and Martin Schoeberl. Automatic generation of application-specific systems based on a micro-programmed java core. In *Proceedings of the 20th ACM Symposium on Applied Computing, Embedded Systems track*, Santa Fee, New Mexico, March 2005.

- Martin Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

- Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

- Martin Schoeberl. Evaluation of a Java processor. In *Tagungsband Austrochip 2005*, pages 127–134, Vienna, Austria, October 2005.

- Martin 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.

- Martin Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006.

- Martin Schoeberl. Instruction cache fr echtzeitsysteme, April 2006. Austrian patent AT 500.858.

- Rasmus Pedersen and Martin Schoeberl. An embedded support vector machine. In *Proceedings of the Fourth Workshop on Intelligent Solutions in Embedded Systems (WISES 2006)*, pages 79–89, Jun. 2006.

- Rasmus Pedersen and Martin Schoeberl. Exact roots for a real-time garbage collector. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2006)*, Paris, France, October 2006.

- Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2006)*, Paris, France, October 2006.

- Martin Schoeberl, Hans Sondergaard, Bent Thomsen, and Anders 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.

- Martin Schoeberl. Mission modes for safety critical java. In *5th IFIP Workshop on Software Technologies for Future Embedded & Ubiquitous Systems*, May 2007.

- Raimund Kirner and Martin Schoeberl. Modeling the function cache for worst-case execution time analysis. In *Proceedings of the 44rd Design Automation Conference, DAC 2007*, San Diego, CA, USA, June 2007. ACM.

- Martin Schoeberl. A time-triggered network-on-chip. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, Amsterdam, Netherlands, August 2007.

- Christof Pitter and Martin Schoeberl. Time predictable CPU and DMA shared memory access. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, Amsterdam, Netherlands, August 2007.

- Martin Schoeberl. A Java processor architecture for embedded real-time systems. *Journal of Systems Architecture*, doi:10.1016/j.sysarc.2007.06.001, 2007.

- Wolfgang Puffitsch and Martin Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, Vienna, Austria, September 2007. ACM Press.

- Martin Schoeberl. Architecture for object oriented programming languages. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, Vienna, Austria, September 2007. ACM Press.

- Christof Pitter and Martin Schoeberl. Towards a Java multiprocessor. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, Vienna, Austria, September 2007. ACM Press.

- Martin Schoeberl and Jan Vitek. Garbage collection for safety critical Java. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, Vienna, Austria, September 2007. ACM Press.

B Acronyms

ADC	Analog to Digital Converter
ALU	Arithmetic and Logic Unit
ASIC	Application-Specific Integrated Circuit
BCET	Best Case Execution Time
CFG	Control Flow Graph
CISC	Complex Instruction Set Computer
CLDC	Connected Limited Device Configuration
CPI	average Clock cycles Per Instruction
CRC	Cyclic Redundancy Check
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
EDF	Earliest Deadline First
EMC	Electromagnetic Compatibility
ESD	Electrostatic Discharge
FIFO	Fist In, First Out
FPGA	Field Programmable Gate Array
GC	Garbage Collect(ion/or)
IC	Instruction Count
ILP	Instruction Level Parallelism
JOP	Java Optimized Processor
J2ME	Java2 Micro Edition
J2SE	Java2 Standard Edition
JDK	Java Development Kit
JIT	Just-In-Time
JVM	Java Virtual Machine
LC	Logic Cell
LRU	Least-Recently Used
MBIB	Memory Bytes read per Instruction Byte
MCIB	Memory Cycles per Instruction Byte

MP	Miss Penalty
MTIB	Memory Transactions per Instruction Byte
MUX	Multiplexer
OO	Object Oriented
OS	Operating System
RISC	Reduced Instruction Set Computer
RT	Real-Time
RTOS	Real-Time Operating System
RTSJ	Real-Time Specification for Java
SCADA	Supervisory Control And Data Acquisition
SDRAM	Synchronous DRAM
SRAM	Static Random Access Memory
TOS	Top Of Stack
UART	Universal Asynchronous Receiver/Transmitter
VHDL	Very High Speed Integrated Circuit (VHSIC) Hardware Description Language
WCET	Worst-Case Execution Time

C JOP Instruction Set

The instruction set of JOP, the so-called microcode, is described in this appendix. Each instruction consists of a single instruction word (8 bits) without extra operands and executes in a single cycle[1]. Table C.1 lists the registers and internal memory areas that are used in the dataflow description.

[1] The only multicycle instruction is wait and depends on the access time of the external memory

Name	Description
A	Top of the stack
B	The element one below the top of stack
stack[]	The stack buffer for the rest of the stack
sp	The stack pointer for the stack buffer
vp	The variable pointer. Points to the first local in the stack buffer
ar	Address register for indirect stack access
pc	Microcode program counter
offtbl	Table for branch offsets
jpc	Program counter for the Java bytecode
opd	8 bit operand from the bytecode fetch unit
opd_{16}	16 bit operand from the bytecode fetch unit
ioar	Address register of the IO subsystem
memrda	Read address register of the memory subsystem
memwra	Write address register of the memory subsystem
memrdd	Read data register of the memory subsystem
memwrd	Write data register of the memory subsystem
mula, mulb	Operands of the hardware multiplier
mulr	Result register of the hardware multiplier
membcr	Bytecode address and length register of the memory subsystem
bcstart	Method start address register in the method cache

Table C.1: JOP hardware registers and memory areas

pop

Operation Pop the top operand stack value

Opcode `00000000`

Dataflow $B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent `pop`

Description Pop the top value from the operand stack.

and

Operation Boolean AND `int`

Opcode `00000001`

Dataflow $A \wedge B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent `iand`

Description Build the bitwise AND (conjunction) of the two top elements of the stack and push back the result onto the operand stack.

or

Operation Boolean OR int

Opcode 00000010

Dataflow $A \vee B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent ior

Description Build the bitwise inclusive OR (disjunction) of the two top elements of the stack and push back the result onto the operand stack.

xor

Operation Boolean XOR int

Opcode 00000011

Dataflow $A \not\equiv B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent ixor

Description Build the bitwise exclusive OR (negation of equivalence) of the two top elements of the stack and push back the result onto the operand stack.

add

Operation Add int

Opcode 00000100

Dataflow $A + B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent iadd

Description Add the two top elements from the stack and push back the result onto the operand stack.

sub

Operation Subtract int

Opcode 00000101

Dataflow $A - B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent isub

Description Subtract the two top elements from the stack and push back the result onto the operand stack.

stmra

Operation Store memory read address

Opcode `00001000`

Dataflow
$A \rightarrow memrda$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description The top value from the stack is stored as read address in the memory subsystem. This operation starts the concurrent memory read. The processor can continue with other operations. When the datum is needed a `wait` instruction stalls the processor till the read access is finished. The value is read with `ldmrd`.

stmwa

Operation Store memory write address

Opcode `00001001`

Dataflow
$A \rightarrow memwra$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description The top value from the stack is stored as write address in the memory subsystem for a following `stmwd`.

stmwd

Operation Store memory write data

Opcode `00001010`

Dataflow
$A \rightarrow memwrd$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description The top value from the stack is stored as write data in the memory subsystem. This operation starts the concurrent memory write The processor can continue with other operations. The wait instruction stalls the processor till the write access is finished.

stald

Operation Start array load

Opcode `00001011`

Dataflow
$A \rightarrow memidx$
$B \rightarrow A$
$B \rightarrow memptr$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent xaload

Description The top value from the stack is stored as array index, the next as reference in the memory subsystem. This operation starts the concurrent array load. The processor can continue with other operations. The wait instruction stalls the processor till the read access is finished. A null pointer or out of bounds exception is generated by the memory subsystem and thrown at the next bytecode fetch.

stast

Operation Start array store

Opcode `00001100`

Dataflow
$A \rightarrow memval$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$
$nextcycle$
$A \rightarrow memidx$
$B \rightarrow A$
$B \rightarrow memptr$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent xastore

Description In the first cycle the top value from the stack is stored as value into the memory subsystem. A microcode pop hast to follow. In the second cycle the top value from the stack is stored as array index, the next as reference in the memory subsystem. This operation starts the concurrent array store. The processor can continue with other operations. The wait instruction stalls the processor till the write access is finished. A null pointer or out of bounds exception is generated by the memory subsystem and thrown at the next bytecode fetch.

stmul

Operation Multiply int

Opcode 00001101

Dataflow
$A \rightarrow mula$
$B \rightarrow mulb$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description The top value from the stack is stored as first operand for the multiplier. The value one below the top of stack is stored as second operand for the multiplier. This operation starts the multiplier. The result is read with the ldmul instruction.

stbcrd

Operation Start bytecode read

Opcode `00001111`

Dataflow $A \rightarrow membcr$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp-1 \rightarrow sp$

JVM equivalent –

Description The top value from the stack is stored as address and length of a method in the memory subsystem. This operation starts the memory transfer from the main memory to the bytecode cache (DMA). The processor can continue with other operations. The wait instruction stalls the processor till the transfer has finished. No other memory accesses are allowed during the bytecode read.

st<n>

Operation Store 32-bit word into local variable

Opcode `000100nn`

Dataflow $A \rightarrow stack[vp+n]$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp-1 \rightarrow sp$

JVM equivalent astore_<n>, istore_<n>, fstore_<n>

Description The value on the top of the operand stack is popped and stored in the local variable at position n.

st

Operation Store 32-bit word into local variable

Opcode `00010100`

Dataflow $A \rightarrow stack[vp + opd]$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent astore, istore, fstore

Description The value on the top of the operand stack is popped and stored in the local variable at position opd. opd is taken from the bytecode instruction stream.

stmi

Operation Store in local memory indirect

Opcode `00010101`

Dataflow $A \rightarrow stack[ar]$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description The top value from the operand stack is stored in the local memory (stack) at position ar.

stvp

Operation Store variable pointer

Opcode `00011000`

Dataflow
$A \rightarrow vp$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description The value on the top of the operand stack is popped and stored in the variable pointer (`vp`).

stjpc

Operation Store Java program counter

Opcode `00011001`

Dataflow
$A \rightarrow jpc$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description The value on the top of the operand stack is popped and stored in the Java program counter (`jpc`).

star

Operation Store adress register

Opcode `00011010`

Dataflow
$A \rightarrow ar$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description The value on the top of the operand stack is popped and stored in the address register (ar). Due to a pipeline delay the register is valid on cycle later for usage by ldmi and stmi.

stsp

Operation Store stack pointer

Opcode `00011011`

Dataflow
$A \rightarrow sp$
$B \rightarrow A$
$stack[sp] \rightarrow B$

JVM equivalent –

Description The value on the top of the operand stack is popped and stored in the stack pointer (sp).

ushr

Operation Logical shift rigth int

Opcode 00011100

Dataflow $B >>> A \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent iushr

Description The values are popped from the operand stack. An int result is calculated by shifting the TOS-1 value rigth by s position, with zero extension, where s is the value of the low 5 bits of the TOS. The result is pushed onto the operand stack.

shl

Operation Shift left int

Opcode 00011101

Dataflow $B << A \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent ishl

Description The values are popped from the operand stack. An int result is calculated by shifting the TOS-1 value left by s position, where s is the value of the low 5 bits of the TOS. The result is pushed onto the operand stack.

shr

Operation Arithmetic shift rigth int

Opcode 00011110

Dataflow $B >> A \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent ishr

Description The values are popped from the operand stack. An int result is calculated by shifting the TOS-1 value rigth by s position, with sign extension, where s is the value of the low 5 bits of the TOS. The result is pushed onto the operand stack.

stm

Operation Store in local memory

Opcode 001nnnnn

Dataflow $A \rightarrow stack[n]$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description The top value from the operand stack is stored in the local memory (stack) at position n. These 32 memory destinations represent microcode local variables.

bz

Operation Branch if value is zero

Opcode `010nnnnn`

Dataflow if $A = 0$ then $pc + offtbl[n] + 2 \rightarrow pc$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description If the top value from the operand stack is zero a microcode branch is taken. The value is popped from the operand stack. Due to a pipeline delay, the zero flag is delayed one cycle, i.e. the value from the last but one instruction is taken. The branch is followed by two branch delay slots. The branch offset is taken from the table $offtbl$ indexed by n.

bnz

Operation Branch if value is not zero

Opcode `011nnnnn`

Dataflow if $A \neq 0$ then $pc + offtbl[n] + 2 \rightarrow pc$
$B \rightarrow A$
$stack[sp] \rightarrow B$
$sp - 1 \rightarrow sp$

JVM equivalent –

Description If the top value from the operand stack is not zero a microcode branch is taken. The value is popped from the operand stack. Due to a pipeline delay, the zero flag is delayed one cycle, i.e. the value from the last but one instruction is taken. The branch is followed by two branch delay slots. The branch offset is taken from the table $offtbl$ indexed by n.

nop

Operation	Do nothing
Opcode	`10000000`
Dataflow	–
JVM equivalent	`nop`
Description	The famous no operation instruction.

wait

Operation	Wait for memory completion
Opcode	`10000001`
Dataflow	–
JVM equivalent	–
Description	This instruction stalls the processor until a pending memory instruction (`stmra`, `stmwd` or `stbcrd`) has completed. Two consecutive `wait` instructions are necessary for a correct stall of the decode and execute stage.

jbr

Operation Conditional bytecode branch and goto

Opcode `10000010`

Dataflow –

JVM equivalent ifnull, ifnonnull, ifeq, ifne, iflt, ifge, ifgt, ifle, if_acmpeq, if_acmpne, if_icmpeq, if_icmpne, if_icmplt, if_icmpge, if_icmpgt, if_icmple, goto

Description Execute a bytecode branch or goto. The branch condition and offset are calculated in the bytecode fetch unit. Arguments must be removed with pop instructions in the following microcode instructions.

ldm

Operation Load from local memory

Opcode `101nnnnn`

Dataflow $stack[n] \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description The value from the local memory (stack) at position n is pushed onto the operand stack. These 32 memory destinations represent microcode local variables.

ldi

Operation Load from local memory

Opcode `110nnnnn`

Dataflow
$stack[n+32] \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description The value from the local memory (stack) at position $n+32$ is pushed onto the operand stack. These 32 memory destinations represent microcode constants.

ldmrd

Operation Load memory read data

Opcode `11100010`

Dataflow
$memrdd \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description The value from the memory system after a memory read is pushed onto the operand stack. This operation is usually preceded by two `wait` instructions.

ldmul

Operation Load multiplier result

Opcode `11100101`

Dataflow
$mulr \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent (imul)

Description The result of the multiplier is pushed onto the operand stack.

ldbcstart

Operation Load method start

Opcode `11100111`

Dataflow
$bcstart \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description The method start address in the method cache is pushed onto the operand stack.

ld<n>

Operation Load 32-bit word from local variable

Opcode `111010nn`

Dataflow
$stack[vp+n] \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent aload_<n>, iload_<n>, fload_<n>

Description The local variable at position *n* is pushed onto the operand stack.

ld

Operation Load 32-bit word from local variable

Opcode `11101100`

Dataflow
$stack[vp+opd] \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent aload, iload, fload

Description The local variable at position *opd* is pushed onto the operand stack. *opd* is taken from the bytecode instruction stream.

ldmi

Operation Load from local memory indirect

Opcode `11101101`

Dataflow
$stack[ar] \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description The value from the local memory (stack) at position ar is pushed onto the operand stack.

ldsp

Operation Load stack pointer

Opcode `11110000`

Dataflow
$sp \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description The stack pointer is pushed onto the operand stack.

ldvp

Operation Load variable pointer

Opcode `11110001`

Dataflow
$vp \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description The variable pointer is pushed onto the operand stack.

ldjpc

Operation Load Java program counter

Opcode `11110010`

Dataflow
$jpc \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description The Java program counter is pushed onto the operand stack.

ld_opd_8u

Operation Load 8-bit bytecode operand unsigned

Opcode `11110100`

Dataflow
$opd \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description A single byte from the bytecode stream is pushed as `int` onto the operand stack.

ld_opd_8s

Operation Load 8-bit bytecode operand signed

Opcode `11110101`

Dataflow
$opd \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent (`bipush`)

Description A single byte from the bytecode stream is sign-extended to an `int` and pushed onto the operand stack.

ld_opd_16u

Operation Load 16-bit bytecode operand unsigned

Opcode `11110110`

Dataflow
$opd\_16 \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent –

Description A 16-bit word from the bytecode stream is pushed as `int` onto the operand stack.

ld_opd_16s

Operation Load 16-bit bytecode operand signed

Opcode `11110111`

Dataflow
$opd\_16 \rightarrow A$
$A \rightarrow B$
$B \rightarrow stack[sp+1]$
$sp+1 \rightarrow sp$

JVM equivalent (`sipush`)

Description A 16-bit word from the bytecode stream is sign-extended to an `int` and pushed onto the operand stack.

dup

Operation	Duplicate the top operand stack value
Opcode	`11111000`
Dataflow	$A \rightarrow B$ $B \rightarrow stack[sp+1]$ $sp+1 \rightarrow sp$
JVM equivalent	`dup`
Description	Duplicate the top value on the operand stack and push it onto the operand stack.

D Bytecode Execution Time

Table D.1 lists the bytecodes of the JVM with their opcode, mnemonics, the implementation type and the execution time on JOP. In the implementation column *hw* means that this bytecode has a microcode equivalent, *mc* means that a microcode sequence implements the bytecode, *Java* means the bytecode is implemented in Java, and a '-' indicates that this bytecode is not yet implemented. For bytecodes with a variable execution time the minimum and maximum values are given.

Opcode	Instruction	Implementation	Cycles
0	nop	hw	1
1	aconst_null	hw	1
2	iconst_m1	hw	1
3	iconst_0	hw	1
4	iconst_1	hw	1
5	iconst_2	hw	1
6	iconst_3	hw	1
7	iconst_4	hw	1
8	iconst_5	hw	1
9	lconst_0	mc	2
10	lconst_1	mc	2
11	fconst_0	Java	
12	fconst_1	Java	
13	fconst_2	Java	
14	dconst_0	-	
15	dconst_1	-	
16	bipush	mc	2
17	sipush	mc	3
18	ldc	mc	7+r
19	ldc_w	mc	8+r

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
20	ldc2_w[20]	mc	17+2*r
21	iload	mc	2
22	lload	mc	11
23	fload	mc	2
24	dload	mc	11
25	aload	mc	2
26	iload_0	hw	1
27	iload_1	hw	1
28	iload_2	hw	1
29	iload_3	hw	1
30	lload_0	mc	2
31	lload_1	mc	2
32	lload_2	mc	2
33	lload_3	mc	11
34	fload_0	hw	1
35	fload_1	hw	1
36	fload_2	hw	1
37	fload_3	hw	1
38	dload_0	mc	2
39	dload_1	mc	2
40	dload_2	mc	2
41	dload_3	mc	11
42	aload_0	hw	1
43	aload_1	hw	1
44	aload_2	hw	1
45	aload_3	hw	1
46	iaload[46]	hw	7+3*r
47	laload	mc	43+4*r
48	faload[46]	hw	7+3*r
49	daload	-	
50	aaload[46]	hw	7+3*r
51	baload[46]	hw	7+3*r
52	caload[46]	hw	7+3*r

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
53	saload[46]	hw	7+3*r
54	istore	mc	2
55	lstore	mc	11
56	fstore	mc	2
57	dstore	mc	11
58	astore	mc	2
59	istore_0	hw	1
60	istore_1	hw	1
61	istore_2	hw	1
62	istore_3	hw	1
63	lstore_0	mc	2
64	lstore_1	mc	2
65	lstore_2	mc	2
66	lstore_3	mc	11
67	fstore_0	hw	1
68	fstore_1	hw	1
69	fstore_2	hw	1
70	fstore_3	hw	1
71	dstore_0	mc	2
72	dstore_1	mc	2
73	dstore_2	mc	2
74	dstore_3	mc	11
75	astore_0	hw	1
76	astore_1	hw	1
77	astore_2	hw	1
78	astore_3	hw	1
79	iastore[79]	hw	10+2*r+w
80	lastore[1]	mc	48+2*r+2*w
81	fastore[79]	hw	10+2*r+w
82	dastore	-	
83	aastore	Java	
84	bastore[79]	hw	10+2*r+w
85	castore[79]	hw	10+2*r+w

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
86	sastore[79]	hw	10+2*r+w
87	pop	hw	1
88	pop2	mc	2
89	dup	hw	1
90	dup_x1	mc	5
91	dup_x2	mc	7
92	dup2	mc	6
93	dup2_x1	mc	8
94	dup2_x2	mc	10
95	swap[2]	mc	4
96	iadd	hw	1
97	ladd	Java	
98	fadd	Java	
99	dadd	-	
100	isub	hw	1
101	lsub	Java	
102	fsub	Java	
103	dsub	-	
104	imul	mc	35
105	lmul	Java	
106	fmul	Java	
107	dmul	-	
108	idiv	Java	
109	ldiv	Java	
110	fdiv	Java	
111	ddiv	-	
112	irem	Java	
113	lrem	Java	
114	frem	Java	
115	drem	-	
116	ineg	mc	4
117	lneg	Java	
118	fneg	Java	

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
119	dneg	-	
120	ishl	hw	1
121	lshl	Java	
122	ishr	hw	1
123	lshr	Java	
124	iushr	hw	1
125	lushr	Java	
126	iand	hw	1
127	land	Java	
128	ior	hw	1
129	lor	Java	
130	ixor	hw	1
131	lxor	Java	
132	iinc	mc	8
133	i2l	Java	
134	i2f	Java	
135	i2d	-	
136	l2i	mc	3
137	l2f	-	
138	l2d	-	
139	f2i	Java	
140	f2l	-	
141	f2d	-	
142	d2i	-	
143	d2l	-	
144	d2f	-	
145	i2b	Java	
146	i2c	mc	2
147	i2s	Java	
148	lcmp	Java	
149	fcmpl	Java	
150	fcmpg	Java	
151	dcmpl	-	

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
152	dcmpg	-	
153	ifeq	mc	4
154	ifne	mc	4
155	iflt	mc	4
156	ifge	mc	4
157	ifgt	mc	4
158	ifle	mc	4
159	if_icmpeq	mc	4
160	if_icmpne	mc	4
161	if_icmplt	mc	4
162	if_icmpge	mc	4
163	if_icmpgt	mc	4
164	if_icmple	mc	4
165	if_acmpeq	mc	4
166	if_acmpne	mc	4
167	goto	mc	4
168	jsr	*not used*	
169	ret	*not used*	
170	tableswitch[170]	Java	
171	lookupswitch[171]	Java	
172	ireturn[172]	mc	23+r+l
173	lreturn[173]	mc	25+r+l
174	freturn[172]	mc	23+r+l
175	dreturn[173]	mc	25+r+l
176	areturn[172]	mc	23+r+l
177	return[177]	mc	21+r+l
178	getstatic	mc	12+2*r
179	putstatic	mc	13+r+w
180	getfield	hw	11+2*r
181	putfield	hw	13+r+w
182	invokevirtual[182]	mc	98+4r+l
183	invokespecial[183]	mc	74+3*r+l
184	invokestatic[183]	mc	74+3*r+l

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
185	invokeinterface[185]	mc	112+6r+l
186	unused_ba	-	
187	new[187]	Java	
188	newarray[188]	Java	
189	anewarray	Java	
190	arraylength	mc	6+r
191	athrow[3]	Java	
192	checkcast	Java	
193	instanceof	Java	
194	monitorenter	mc	19
195	monitorexit	mc	22
196	wide	*not used*	
197	multianewarray[4]	Java	
198	ifnull	mc	4
199	ifnonnull	mc	4
200	goto_w	*not used*	
201	jsr_w	*not used*	
202	breakpoint	-	
203	reserved	-	
204	reserved	-	
205	reserved	-	
206	reserved	-	
207	reserved	-	
208	reserved	-	
209	jopsys_rd[209]	mc	4+r
210	jopsys_wr	mc	5+w
211	jopsys_rdmem	mc	4+r
212	jopsys_wrmem	mc	5+w
213	jopsys_rdint	mc	3
214	jopsys_wrint	mc	3
215	jopsys_getsp	mc	3
216	jopsys_setsp	mc	4
217	jopsys_getvp	hw	1

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
218	jopsys_setvp	mc	2
219	jopsys_int2ext[219]	mc	14+r+n*(23+w)
220	jopsys_ext2int[220]	mc	14+r+n*(23+r)
221	jopsys_nop	mc	1
222	jopsys_invoke	mc	
223	jopsys_cond_move	mc	5
224	getstatic_ref	mc	12+2*r
225	putstatic_ref	Java	
226	getfield_ref	mc	11+2*r
227	putfield_ref	Java	
228	getstatic_long	mc	
229	putstatic_long	mc	
230	getfield_long	mc	
231	putfield_long	mc	
232	jopsys_memcpy	mc	
233	reserved	-	
234	reserved	-	
235	reserved	-	
236	invokesuper	mc	?
237	reserved	-	
238	reserved	-	
239	reserved	-	
240	sys_int[240]	Java	
241	sys_exc[240]	Java	
242	reserved	-	
243	reserved	-	
244	reserved	-	
245	reserved	-	
246	reserved	-	
247	reserved	-	
248	reserved	-	
249	reserved	-	
250	reserved	-	

Table D.1: Implemented bytecodes and execution time in cycles

Opcode	Instruction	Implementation	Cycles
251	reserved	-	
252	reserved	-	
253	reserved	-	
254	sys_noimp	Java	
255	sys_init	*not used*	

Table D.1: Implemented bytecodes and execution time in cycles

Memory Timing

The external memory timing is defined in the top level VHDL file (e.g. jopcyc.vhd) with ram_cnt for the number of cycles for a read and write access. At the moment there is no difference for a read and write access. For the 100MHz JOP with 15ns SRAMs this access

[1] The exact value is given below.

[2] Not tested as javac does not emit the swap bytecode.

[3] A simple version that stops the JVM. No catch support.

[4] Only dimension 2 supported.

[20] The exact value is $17+\begin{cases} r-2 & : & r>2 \\ 0 & : & r\leq 2 \end{cases}+\begin{cases} r-1 & : & r>1 \\ 0 & : & r\leq 1 \end{cases}$

[46] The exact value is *no hidden wait states at the moment.*

[79] The exact value is *no hidden wait states at the moment.*

[170] tableswitch execution time depends to a great extent on the caching of the corresponding Java method or the memory transfer time for the method.

[171] lookupswitch execution time depends to a great extent on the caching of the corresponding Java method or the memory transfer time for the method. lookupswitch also depends on the argument as it performs a linear search in the jump table.

[172] The exact value is: $23+\begin{cases} r-3 & : & r>3 \\ 0 & : & r\leq 3 \end{cases}+\begin{cases} l-10 & : & l>10 \\ 0 & : & l\leq 10 \end{cases}$

[173] The exact value is: $25+\begin{cases} r-3 & : & r>3 \\ 0 & : & r\leq 3 \end{cases}+\begin{cases} l-11 & : & l>11 \\ 0 & : & l\leq 11 \end{cases}$

[177] The exact value is: $21+\begin{cases} r-3 & : & r>3 \\ 0 & : & r\leq 3 \end{cases}+\begin{cases} l-9 & : & l>9 \\ 0 & : & l\leq 9 \end{cases}$

[182] The exact value is: $100+2r+\begin{cases} r-3 & : & r>3 \\ 0 & : & r\leq 3 \end{cases}+\begin{cases} r-2 & : & r>2 \\ 0 & : & r\leq 2 \end{cases}+\begin{cases} l-37 & : & l>37 \\ 0 & : & l\leq 37 \end{cases}$

[183] The exact value is: $74+r+\begin{cases} r-3 & : & r>3 \\ 0 & : & r\leq 3 \end{cases}+\begin{cases} r-2 & : & r>2 \\ 0 & : & r\leq 2 \end{cases}+\begin{cases} l-37 & : & l>37 \\ 0 & : & l\leq 37 \end{cases}$

[185] The exact value is: $114+4r+\begin{cases} r-3 & : & r>3 \\ 0 & : & r\leq 3 \end{cases}+\begin{cases} r-2 & : & r>2 \\ 0 & : & r\leq 2 \end{cases}+\begin{cases} l-37 & : & l>37 \\ 0 & : & l\leq 37 \end{cases}$

[187] new execution time depends to a great extent on the caching of the corresponding Java method or the memory transfer time for the method. new also depends on the size of the created object as the memory for the object is filled with zeros – This will change with the GC

[188] newarray execution time depends to a great extent on the caching of the corresponding Java method or the memory transfer time for the method. newarray also depends on the size of the array as the memory for the object is filled with zeros – This will change with the GC

[209] The native instructions jopsys_rd and jopsys_wr are alias to the jopsys_rdmem and jopsys_wrmem instructions just for compatibility to existing Java code. IO devices are now memory mapped. In the case for simple IO devices there are no wait states and the exact values are 4 and 5 cycles respective.

[219] The exact value is $14+r+n(23+\begin{cases} w-8 & : & w>8 \\ 0 & : & w\leq 8 \end{cases})$. n is the number of words transferred.

[220] The exact value is $14+r+n(23+\begin{cases} r-10 & : & r>10 \\ 0 & : & r\leq 10 \end{cases})$. n is the number of words transferred.

[240] *Is the interrupt and the exception still a bytecode or is it now inserted just as microcode address?*

time is two cycles (ram_cnt=2, 20ns). Therefore the wait state n_{ws} is 1 (ram_cnt-1). A basic memory read in microcode is as follows:

```
stmra    // start read with address store
wait     // fill the pipeline with two
wait     // wait instructions
ldmrd    // push read result on TOS
```

In this sequence the *last* wait executes for $1+n_{ws}$ cycles. Therefore the whole read sequence takes $4+n_{ws}$ cycles. For the example with ram_cnt=2 this basic memory read takes 5 cycles.

A memory write in microcode is as follows:

```
stmwa    // store address
stmwd    // store data and start the write
wait     // fill the pipeline with wait
wait     // wait for the memory ready
```

The last wait again executes for $1+n_{ws}$ cycles and the basic write takes $4+n_{ws}$ cycles. For the native bytecode jopsys_wrmem an additional nop instruction for the nxt flag is necessary.

The read and write wait states r_{ws} and w_{ws} are:

$$r_{ws} = w_{ws} = \begin{cases} ram\_cnt - 1 & : \quad ram\_cnt > 1 \\ 0 & : \quad ram\_cnt \leq 1 \end{cases}$$

In the instruction timing we use r and w instead of r_{ws} and w_{ws}. The wait states can be hidden by other microcode instructions between stmra/wait and stmwd/wait. The exact value is given in the footnote.

Instruction Timing

The bytecodes that access memory are indicated by an r for a memory read and an w for a memory write at the cycles column (r and w are the additional wait states). The wait cycles for the memory access have to be added to the execution time. These two values are implementation dependent (clock frequency versus RAM access time, data bus width); for the Cyclone EP1C6 board with 15ns SRAMs and 100MHz clock frequency these values are both 1 cycle (ram_cnt-1).

For some bytecodes, part of the memory latency can be hidden by executing microcode during the memory access. However, these cycles can only be subtracted when the wait states (r or w) are larger then 0 cycles. The exact execution time with the subtraction of the saved cycles is given in the footnote.

Cache Load

For the method cache load the cache wait state c_{ws} is:

$$c_{ws} = \begin{cases} r_{ws} - 1 & : \quad r_{ws} > 1 \\ 0 & : \quad r_{ws} \leq 1 \end{cases}$$

On a method invoke or return the bytecode has to be loaded into the cache on a cache miss. The load time l is:

$$l = \begin{cases} 6 + (n+1)(2 + c_{ws}) & : \quad \text{cache miss} \\ 4 & : \quad \text{cach hit} \end{cases}$$

with n as the length of the method in number of 32-bit words. For short methods the load time of the method on a cache miss, or part of it, is hidden by microcode execution. The exact value is given in the footnote.

lastore

$$t_{lastore} = 48 + 2r_{ws} + w_{ws} + \begin{cases} w_{ws} - 3 & : \quad w_{ws} > 3 \\ 0 & : \quad w_{ws} \leq 3 \end{cases}$$

get/putfield/ref/long

TODO: add different values for 32-bit, 64-bit and reference type.

TODO: add invokesuper - a special version of invokespecial

E Cyclone FPGA Board

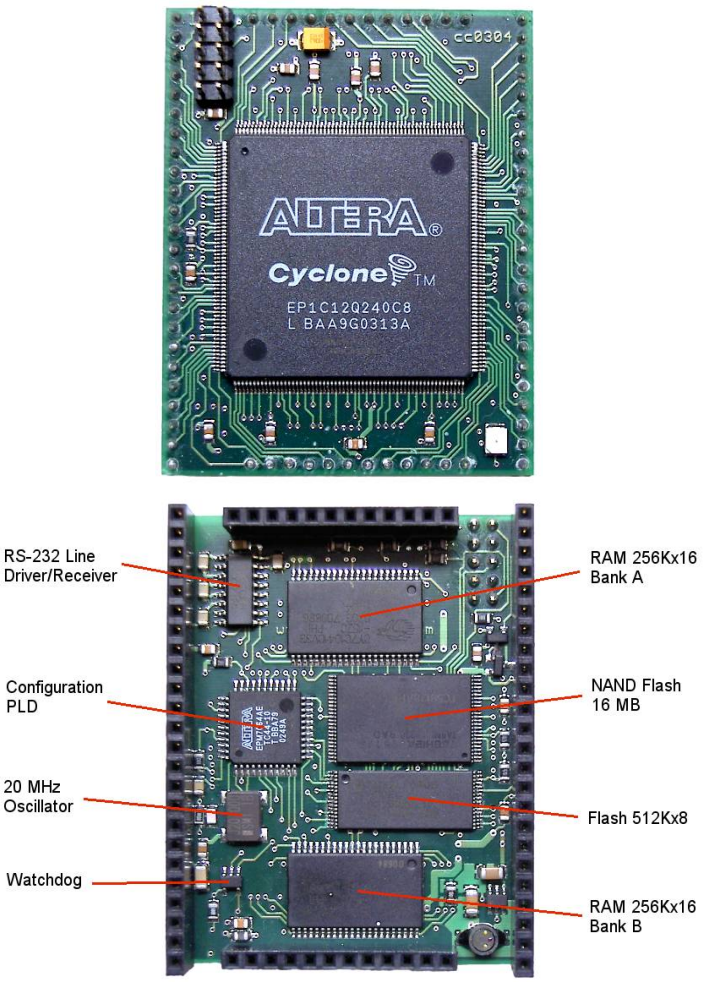

Figure E.1: Top and bottom side of the Cyclone FPGA board

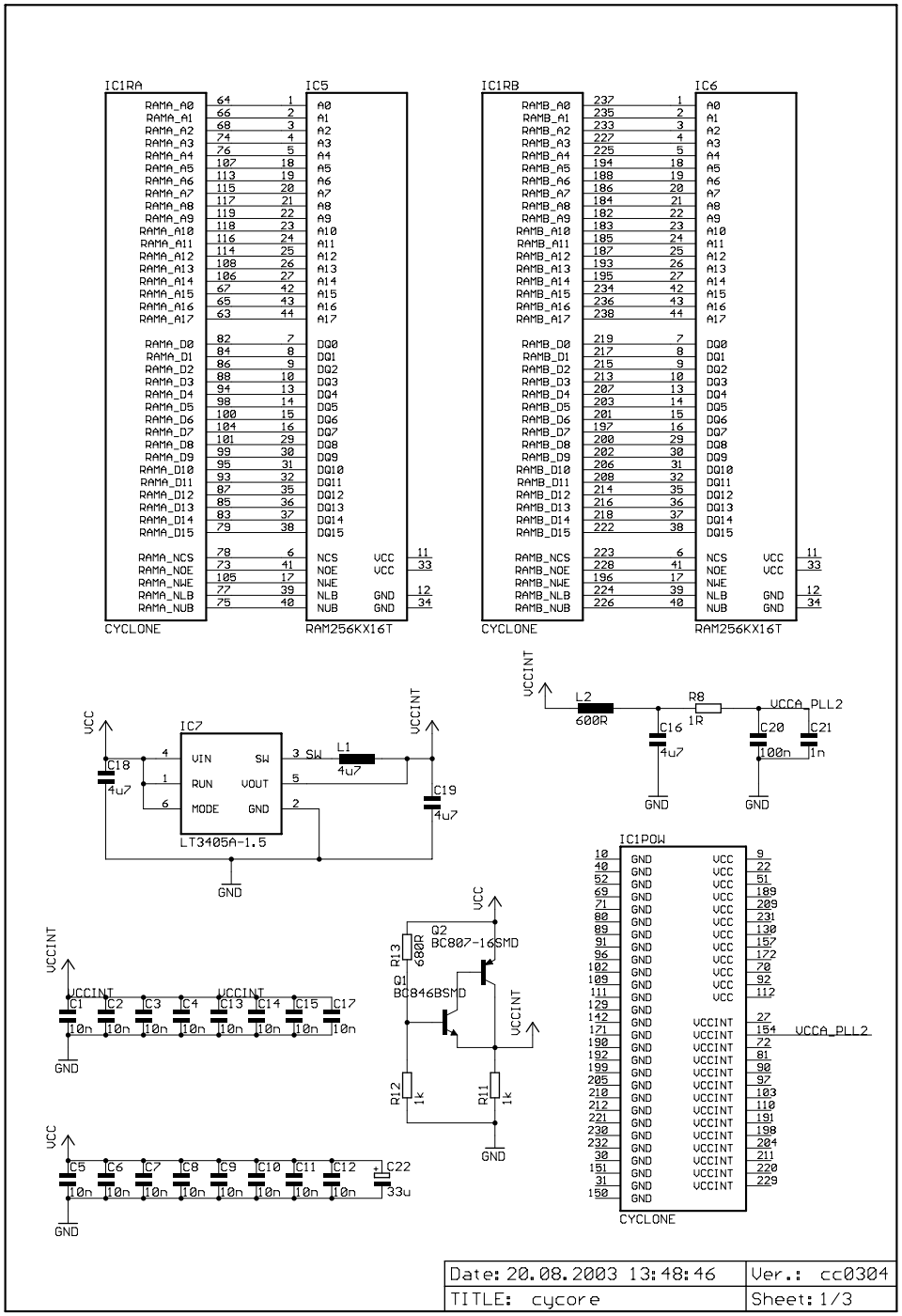

Figure E.2: Schematic of the Cyclone FPGA board, page 1

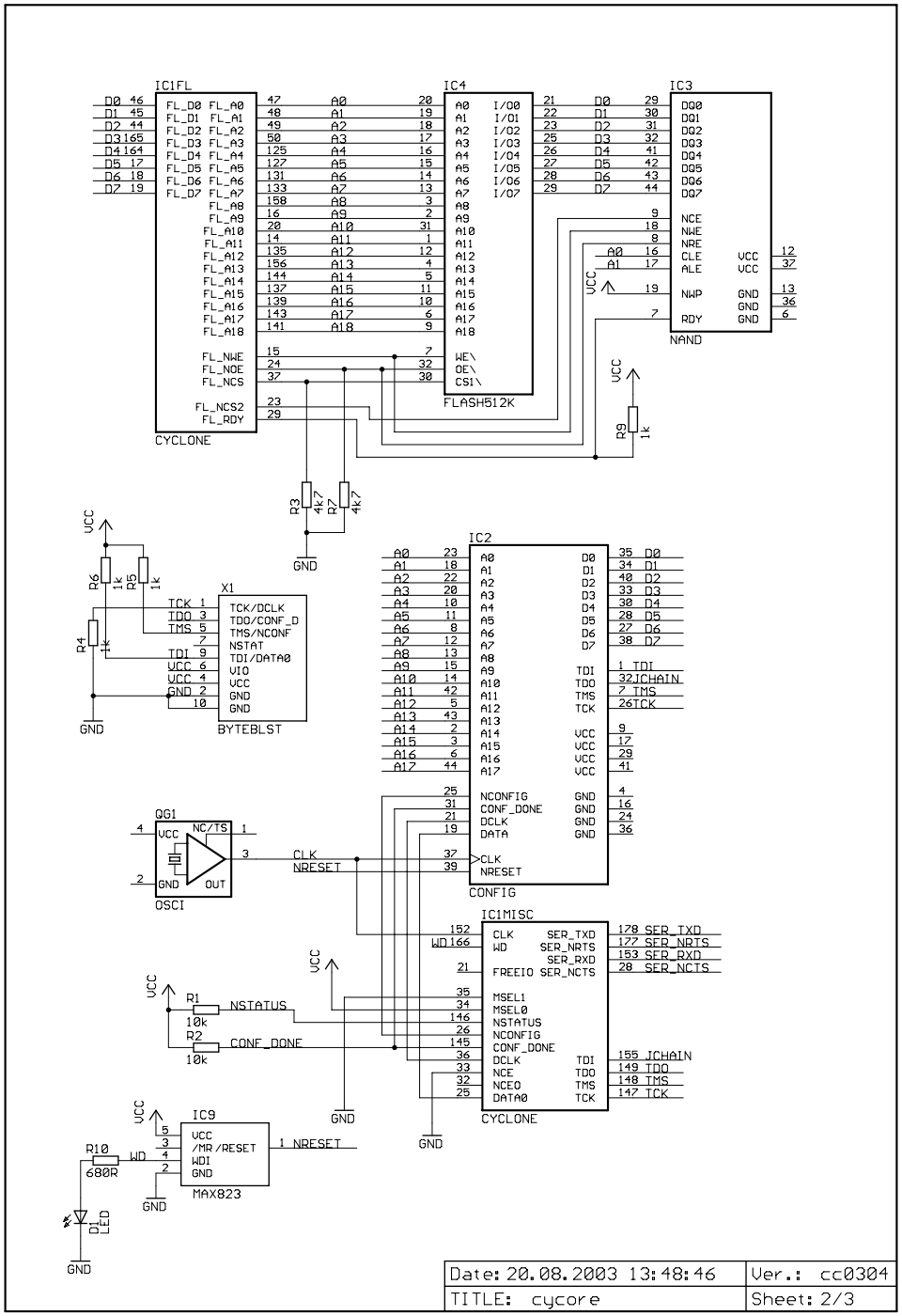

Figure E.3: Schematic of the Cyclone FPGA board, page 2

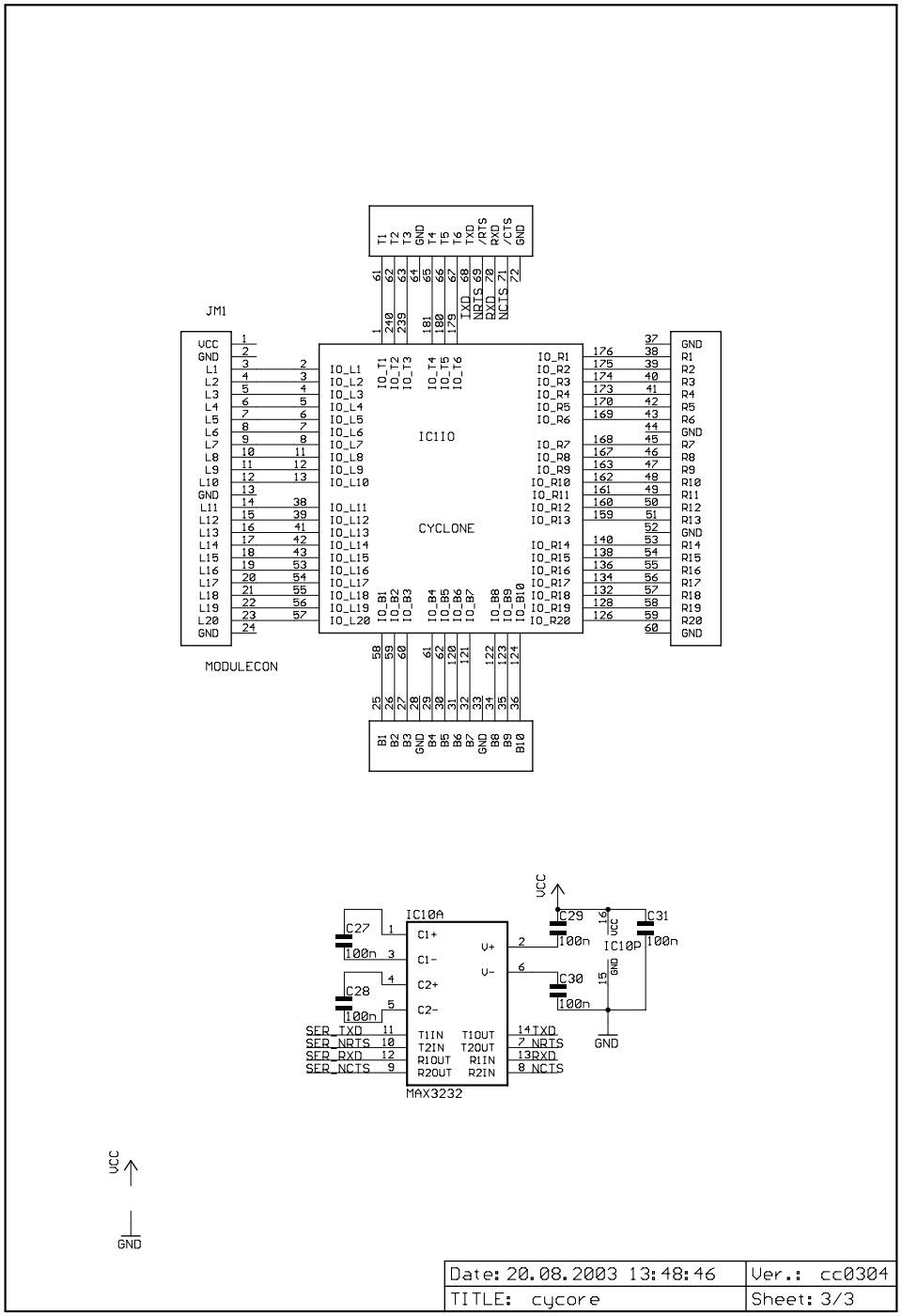

Figure E.4: Schematic of the Cyclone FPGA board, page 3

Bibliography

[1] Georg Acher. *JIFFY — Ein FPGA-basierter Java Just-in-Time Compiler für eingebettete Anwendungen*. PhD thesis, Technische Universität München, 2003.

[2] aJile. aj-100 real-time low power Java processor. preliminary data sheet, 2000.

[3] Altera. Cyclone FPGA Family Data Sheet, ver. 1.2, April 2003.

[4] Altera. Nios 3.0 CPU. data sheet, version 2.2, October 2004.

[5] Altera. Avalon interface specification, April 2005.

[6] Altera. Quartus ii version 7.1 handbook, May 2007.

[7] ARM. AMBA specification (rev 2.0), May 1999.

[8] ARM. AMBA AXI protocol v1.0 specification, March 2004.

[9] ARM. Jazelle technology: ARM acceleration technology for the Java platform. white paper, 2004.

[10] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A real-time java virtual machine with applications in avionics. *Trans. on Embedded Computing Sys.*, 7(1):1–49, 2007.

[11] R. Arnold, F. Mueller, D. Whalley, and M. Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, 1994.

[12] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atalanta, 1991.

[13] David F. Bacon, Perry Cheng, David Grove, Michael Hind, V. T. Rajan, Eran Yahav, Matthias Hauswirth, Christoph M. Kirsch, Daniel Spoonhower, and Martin T.

Vechev. High-level real-time programming in java. In *EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software*, pages 68–78, New York, NY, USA, 2005. ACM Press.

[14] David F. Bacon, Perry Cheng, and V. T. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In Robert Meersman and Zahir Tari, editors, *OTM Workshops*, volume 2889 of *Lecture Notes in Computer Science*, pages 466–478. Springer, 2003.

[15] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.

[16] Henry G. Baker. List processing in real time on a serial computer. *Commun. ACM*, 21(4):280–294, 1978.

[17] Henry G. Baker. The treadmill: real-time garbage collection without motion sickness. *SIGPLAN Not.*, 27(3):66–70, 1992.

[18] Iain Bate, Guillem Bernat, Greg Murphy, and Peter Puschner. Low-level analysis of a portable Java byte code WCET analysis framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications*, pages 39–48, Dec. 2000.

[19] Elliot Berk. Jlex: A lexical analyzer generator for java. Available at http://www.cs.princeton.edu/ appel/modern/java/JLex/.

[20] Guillem Bernat, Alan Burns, and Andy Wellings. Portable worst-case execution time analysis using java byte code. In *Proc. 12th EUROMICRO Conference on Real-time Systems*, Jun 2000.

[21] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[22] Ben Brosgol and Brian Dobbing. Real-time convergence of Ada and Java. In *Proceedings of the 2001 annual ACM SIGAda international conference on Ada*, pages 11–26. ACM Press, 2001.

[23] Alan Burns, Brian Dobbing, and G. Romanski. The ravenscar tasking profile for high integrity real-time programs. In *Proceedings of the 1998 Ada-Europe International Conference on Reliable Software Technologies*, pages 263–275. Springer-Verlag, 1998.

[24] Clemens Cap, Dirk Timmermann, Frank Golatowski, Hagen Ploog, Stephan Preuss, and Thomas Geithner. Integration of Java processor core JSM into smartdev(ices). In *Proceedings of the 8th IEEE International Conference on Emerging Technologies and Factory Automation*, Oktober 2001.

[25] C. J. Cheney. A nonrecursive list compacting algorithm. *Commun. ACM*, 13(11):677–678, 1970.

[26] Cyrille Comar, Gary Dismukes, and Franco Gasperoni. Targeting gnat to the java virtual machine. In *Proceedings of the conference on TRI-Ada '97*, pages 149–161. ACM Press, 1997.

[27] Angelo Corsaro and Douglas C. Schmidt. The design and performance of real-time Java middleware. *IEEE Transactions on Parallel and Distributed Systems*, 14(11):1155–1167, November 2003.

[28] DCT. Lightfoot 32-bit Java processor core. data sheet, September 2001.

[29] Martin Delvai, Wolfgang Huber, Peter Puschner, and Andreas Steininger. Processor support for temporal predictability – the spear design example. In *Proceedings of the 15th Euromicro International Conference on Real-Time Systems*, Jul. 2003.

[30] Derivation. Lavacore configurable Java processor core. data sheet, April 2001.

[31] S. Dey, P. Sanchez, D. Panigrahi, L. Chen, C. Taylor, and K. Sekar. Using a soft core in a SOC design: Experiences with picoJava. *IEEE Design and Test of Computers*, 17(3):60–71, July 2000.

[32] Edsger W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[33] Tom Dowling, James Power, and John Waldron. Relating static and dynamic measurements for the java virtual machine instruction set. In N.E. Mastorakis, editor, *Recent Advances in Simulation, Computational Methods and Soft Computing*. WSEAS Press, 2002.

[34] M. Eden and M. Kagan. The Pentium processor with MMX technology. In *Proceedings of Compcon '97*, pages 260–262. IEEE Computer Society, 1997.

[35] EJC. The ejc (embedded java controller) platform. Available at http://www.embedded-web.com/index.html.

[36] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

[37] S. Feizabadi, W. Beebee, B. Ravindran, P. Li, and M. Rinard. Utility accrual scheduling with real-time Java. *Lecture Notes in Computer Science*, 2889:550–563, 2003.

[38] Jiri 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.

[39] Jiri Gaisler, Edvin Catovic, Marko Isomäki, Kristoffer Carlsson, and Sandi Habinc. GRLIB IP core user's manual, version 1.0.14. Available at http://www.gaisler.com/, February 2007.

[40] Vincent Gay-Para. Kjc kopi java compiler. Available at http://www.dms.at/.

[41] C. J. Glossner. *The DEFLT-JAVA Engine*. PhD thesis, Delft University of Technology, 2001.

[42] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification Second Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2000.

[43] David Gregg, James Power, and John Waldron. Benchmarking the java virtual architecture - the specjvm98 benchmark suite. In N. Vijaykrishnan and M. Wolczko, editors, *Java Microarchitectures*, pages 1–18. Kluwer Academic, 2002.

[44] Flavius Gruian, Per Andersson, Krzysztof Kuchcinski, and Martin Schoeberl. Automatic generation of application-specific systems based on a micro-programmed java core. In *Proceedings of the 20th ACM Symposium on Applied Computing, Embedded Systems track*, Santa Fee, New Mexico, March 2005.

[45] Flavius Gruian and Zoran Salcic. Designing a concurrent hardware garbage collector for small embedded systems. In *Proceedings of Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC 2005*, pages 281–294. Springer-Verlag GmbH, October 2005.

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

[47] C.A. Healy, D.B. Whalley, and M.G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, 1995.

[48] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.

[49] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 3rd ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303, 2002.

[50] Teresa Higuera, Valerie Issarny, Michel Banatre, Gilbert Cabillic, Jean-Philippe Lesot, and Frederic Parain. Memory management for real-time Java: an efficient solution using hardware support. *Real-Time Systems Journal*, 2002.

[51] Hitachi. Hitachi single-chip microcomputer h8/3297 series. Hardware Manual.

[52] IBM. On-chip peripheral bus architecture specifications v2.1, April 2001.

[53] Imsys. Snap, simple network application platform. Available at http://www.imsys.se/.

[54] Imsys. ISAJ reference 2.0, January 2001.

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

[56] S.A. Ito, L. Carro, and R.P. Jacobi. Making Java work for microcontroller applications. *IEEE Design & Test of Computers*, 18(5):100–110, 2001.

[57] Richard E. Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[58] K. B. Kent. *The Co-Disgn of Virtual Machines Using Reconfigurable Hardware*. PhD thesis, University of Victoria, 2003.

[59] A. Kim and J. M. Chang. Designing a java microprocessor core using fpga technology. *IEE Computing & Control Engineering Journal*, 11(3):135–141, June 2000.

[60] Phillip Koopman. *Stack Computers: The New Wave*. Ellis Horwood, 1989. Out of print, now available over the internet.

[61] Andreas 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, Paris, October 12–18, 1998. IEEE Computer Society Press.

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

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

[64] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java: A high integrity profile for real-time Java. In *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*, pages 131–140. ACM Press, 2002.

[65] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, 1998.

[66] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.

[67] Yau-Tsun Steven Li, Sharad Malik, and Andrew Wolfe. Performance estimation of embedded software with instruction cache modeling. In *Proceedings of the 1995 IEEE/ACM international conference on Computer-aided design*, pages 380–387. IEEE Computer Society, 1995.

[68] Kwei-Jay Lin and Yu-Chung Wang. The design and implementation of real-time schedulers in red-linux. *Proceedings of the IEEE*, 91(7):1114–1130, July 2003.

[69] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[70] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[71] Chuck Moore. ShBoom on ShBoom: A microcosm of software and hardware tools. In *Proceedings 1990 Rochester Forth Conference*, pages 21–27, New York, June 1990.

[72] M. Mrva, K. Buchenrieder, and R. Kress. A scalable architecture for multi-threaded java applications. In *Proceedings of the conference on Design, automation and test in Europe*, pages 868–874. IEEE Computer Society, 1998.

[73] Nazomi. JA 108 product brief. Available at http://www.nazomi.com.

[74] Albert F. Niessner and Edward G. Benowitz. RTSJ memory areas and their affects on the performance of a flight-like attitude control system. In *Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES), LNCS*, 2003.

[75] K. Nilsen, L. Carnahan, and M. Ruark. Requirements for real-time extensions for the Java platform. Available at http://www.nist.gov/rt-java/, September 1999.

[76] Anders Nilsson. Compiling java for real-time systems. Licentiate thesis, Dept. of Computer Science, Lund University, May 2004.

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

[78] OCP-IP Association. Open core protocol specification 2.1. http://www.ocpip.org/, 2005.

[79] Rasmus Pedersen and Martin Schoeberl. Exact roots for a real-time garbage collector. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 77–84, New York, NY, USA, 2006. ACM Press.

[80] Wade D. Peterson. WISHBONE system-on-chip (SoC) interconnection architecture for portable IP cores, revision: B.3. Available at http://www.opencores.org, September 2002.

[81] Matthias Pfeffer. *Ein echtzeitfähiges Java-System für einen mehrfädigen Java-Mikrocontroller*. PhD thesis, University of Augsburg, 2000.

[82] Matthias Pfeffer, Theo Ungerer, Stephan Fuhrmann, Jochen Kreuzinger, and Uwe Brinkschulte. Real-time garbage collection for a multithreaded java microcontroller. *Real-Time Systems*, 26(1):89–106, 2004.

[83] Christof Pitter and Martin Schoeberl. Time predictable CPU and DMA shared memory access. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, pages 317 – 322, Amsterdam, Netherlands, August 2007.

[84] Christof Pitter and Martin Schoeberl. Towards a Java multiprocessor. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 144–151, Vienna, Austria, September 2007. ACM Press.

[85] Christof Pitter and Martin Schoeberl. Performance evaluation of a Java chip-multiprocessor. In *Proceedings of the 3rd IEEE Symposium on Industrial Embedded Systems (SIES 2008)*, Jun. 2008.

[86] Filip Pizlo, J. M. Fox, David Holmes, and Jan Vitek. Real-time java scoped memory: Design patterns and semantics. In *Proceedings of the 7th IEEE International Symposium on, Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 101–110, 2004.

[87] James Power and John Waldron. A method-level analysis of object-oriented techniques in java. Technical Report NUIM-CS-TR-2002-07, Department of Computer Science, NUI Maynooth, Ireland, 2002.

[88] PTSC. Ignite processor brochure, rev 1.0. Available at http://www.ptsc.com.

[89] Wolfgang Puffitsch and Martin Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 213–221, Vienna, Austria, September 2007. ACM Press.

[90] Peter Puschner. Experiments with WCET-oriented programming and the single-path architecture. In *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Feb. 2005.

[91] Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[92] Peter Puschner and Anton Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, Jul. 1997.

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

[94] R. Radhakrishnan. *Microarchitectural Techniques to Enable Efficient Java Execution*. PhD thesis, University of Texas at Austin, 2000.

[95] Ramesh Radhakrishnan, N. Vijaykrishnan, Lizy Kurian John, Anand Sivasubramaniam, Juan Rubio, and Jyotsna Sabarinathan. Java runtime systems: Characterization and architectural implications. *IEEE Trans. Comput.*, 50(2):131–146, 2001.

[96] Mario Aldea Rivas and Michael Gonzlez Harbour. Posix-compatible application-defined scheduling in marte os. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, page 67. IEEE Computer Society, 2002.

[97] Sven Gestegard Robertz and Roger Henriksson. Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 93–102, New York, NY, USA, 2003. ACM Press.

[98] Sven Gestegøard Robertz. *Automatic memory management for flexible real-time systems*. PhD thesis, Department of Computer Science Lund University, 2006.

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

[100] William J. Schmidt and Kelvin D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 76–85, New York, NY, USA, 1994. ACM Press.

[101] Martin Schoeberl. Using a Java optimized processor in a real world application. In *Proceedings of the First Workshop on Intelligent Solutions in Embedded Systems (WISES 2003)*, pages 165–176, Austria, Vienna, June 2003.

[102] Martin Schoeberl. Design rationale of a processor architecture for predictable real-time execution of Java programs. In *Proceedings of the 10th International Con-*

ference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004), Gothenburg, Sweden, August 2004.

[103] Martin Schoeberl. Real-time scheduling on a Java processor. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.

[104] Martin Schoeberl. Restrictions of Java for embedded real-time systems. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 93–100, Vienna, Austria, May 2004.

[105] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

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

[107] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[108] Martin Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006.

[109] Martin Schoeberl. Architecture for object oriented programming languages. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 57–62, Vienna, Austria, September 2007. ACM Press.

[110] Martin Schoeberl. SimpCon - a simple and efficient SoC interconnect. In *Proceedings of the 15th Austrian Workhop on Microelectronics, Austrochip 2007*, Graz, Austria, October 2007.

[111] Martin Schoeberl. A time-triggered network-on-chip. In *International Conference on Field-Programmable Logic and its Applications (FPL 2007)*, pages 377 – 382, Amsterdam, Netherlands, August 2007.

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

[113] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. Number ISBN 978-3-8364-8086-4. VDM Verlag Dr. Müller, July 2008.

[114] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[115] Martin Schoeberl, Hans Sondergaard, Bent Thomsen, and Anders 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.

[116] Martin Schoeberl and Jan Vitek. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 85–93, Vienna, Austria, September 2007. ACM Press.

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

[118] Jose Solorzano. leJOS: Java based os for lego RCX. Available at: http://lejos.sourceforge.net/.

[119] SPEC. The spec jvm98 benchmark suite. Available at http://www.spec.org/, August 1998.

[120] International J Consortium Specification. Real-time core extensions, draft 1.0.14. Available at http://www.j-consortium.org/, September 2000.

[121] Guy L. Steele. Multiprocessing compactifying garbage collection. *Commun. ACM*, 18(9):495–508, 1975.

[122] Sun. A brief history of the green project. Available at: http://today.java.net/jag/old/green/.

[123] Sun. Java 2 platform, micro edition (J2ME). Available at: http://java.sun.com/j2me/docs/.

[124] Sun. Java technology: The early years. Available at: http://java.sun.com/features/1998/05/birthday.html.

[125] Sun. *picoJava-II Microarchitecture Guide*. Sun Microsystems, March 1999.

[126] Sun. *picoJava-II Programmer's Reference Manual*. Sun Microsystems, March 1999.

[127] Systronix. Jstamp real-time native Java module. data sheet.

[128] Y.Y. Tan, C.H. Yau, K.M. Lo, W.S. Yu, P.L. Mok, and A.S. Fong. Design and implementation of a java processor. *Computers and Digital Techniques, IEE Proceedings-*, 153:20–30, 2006.

[129] Vulcan. Moon v1.0. data sheet, January 2000.

[130] Vulcan. Moon2 - 32 bit native Java technology-based processor. product folder, 2003.

[131] Tim Wilkinson. Kaffe – a virtual machine to run java code. Available at http://www.kaffe.org, 1996.

[132] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[133] Xilinx. Microblaze processor reference guide, edk v6.2 edition. data sheet, December 2003.

[134] Xilinx. Spartan-3 FPGA family: Complete data sheet, ver. 1.2, January 2005.

[135] Tan Yiyu, Richard Li Lo Wan Yiu, Yau Chi Hang, and Anthony S. Fong. A java processor with hardware-support object-oriented instructions. *Microprocessors and Microsystems*, 30(8):469–479, 2006.

[136] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.

[137] R. Zulauf. Entwurf eines Java-Mikrocontrollers und prototypische Implementierung auf einem FPGA. Master's thesis, University of Karlsruhe, 2000.

Index

application
  download, 13

CMP, 245
  booting, 245
  scheduling, 246

extension
  .asm, 20
  .bat, 20
  .bit, 21
  .cdf, 20
  .class, 21
  .c, 20
  .dat, 21
  .jar, 21
  .java, 20
  .jbc, 21
  .jop, 21
  .mif, 21
  .npl, 20
  .pof, 21
  .qpf, 20
  .qsf, 20
  .rbf, 21
  .sof, 21
  .tcl, 20
  .ttf, 21
  .ucf, 21
  .vhd, 20
  .v, 20
  .xml, 20
  BlockGen, 15
  JOPizer, 15
  JopSim, 15
  Jopa, 15
  USBRunner.exe, 14
  amd.exe, 14
  down.exe, 14
  e.exe, 14
  pcsim, 15

JOPizer, 372

native method, 23

simulation
  JopSim, 15, 18
  VHDL, 13, 18

Xilinx, 11, 17

F TODO

F.1 Working Hours

Start 130h counting down from 14.9.2008
Count: 120 18.9.

F.2 TODO Handbook

- Green book corrections of
  - Chapter 4
  - Chapter 5
  - Chapter 6
  - Chapter 7
  - Chapter 8
  - Chapter 9
  - Chapter 10
  - Chapter 11
  - Chapter 12
- Acknowledgements
- Update Results section (and decide on a different name)
- Rewrite user scheduler to a scheduler description
- Derived work: Flavius BlueJEP, Christof, Dresden!
- Figure from Slides on configuration (FPGA + serial/USB channel)
- Index
- Trevor's changes in the Wiki design flow page

- Cover and back cover including text on Martin Schoeberl
- Redraw block diagram
  - I/O after memory controller, memory interface
  - jopcpu
  - B goes to memory contrller
  - Method instead of Bytecode cache
- instruction frequency of Kfl, Lift and UDP/IP into Java benchmark section

F.2.1 Minor Change Ideas

- Perhaps the formulaes from Chapter 6 in an appendix (including the changes in the Outline)
- Add references to the work that is only described in the thesis.
- list Makefile targets and variables
- JOP configurations should be in its own section instead of the design flow.

F.2.2 Editorial Stuff

- check java* fonts – has been 9pt for the green book
- check figure sizes for the orange book size
- unbounded vs. unbound in the whole test
- removed 'proposed'
- the WCET analysis – WCET analysis (for all analysis)
- missing commas
- adjective 'ly'
- articles (the, a)
- compatible *with*
- IO to I/O

F.3 Additional Sections

- VHDL Hello world with cycore
- RTS Introduction (based on thesis intro)
- SCJ definition (+ implementation)
- (Mission modes + implementation)
- Teaching (aids)
- Examples section
  - RtThread examples
  - ejip examples
  - low-level access
- About boards (Baseio, dspio,...)
- OOHW copy
- HWO copy or low level access w Native.rdMem()
- restructure with background info and related work in each chapter
- Document design alternatives you didn't take and why you didn't take them
- Javadoc of some classes
- Embedded Java section
  - Add Sun/IBM solution to the class initialization issue in Chapter **??** and a detailed description how JOPizer handles it. Give a simple example of a cyclic dependency.
  - Check if the Micro Edition layers are still used by Sun
  - check the Sun statement on profiles and provide a reference
  - add PhoneME
  - Still only one profile (MIDP) defined for the CLDC?
  - Check if CDC definition is still true
  - RTSJ implementations: add Angelo and York to jRate and additional RTSJ implementations (aicas, IBM, appogee, Sun)

  - add SCJ in Subsets of the RTSJ
  - check status of JSR-50

- JOP Runtime System
  - SCJ to the real-time profile (plus implementation) – the JTRES text (Jan!) is commented out in the handbook
  - Update object, array, and class layout!

- Describe ejip (it's now commented out in handbook in intro)
- Update performance section with actual numbers
- Add interrupt controller description
  - use text from oohw paper for the explanation of the benchmark
  - own jbe section?
  - picoJava, jamuth
- Update WCET chapter with JTRES paper and references to Elena and Trevors work + further reading section
- IFAC paper into application section
- Related Work
  - Cut it down
  - Add jHISC, jamuth
  - Add derived work (again)
- SC Java

F.3.1 Notes

Based in thesis:

- too much scientific stuff

+- Change related work to comparison

+ HOTOs

+ Internal docu

Copy stuff from JSA paper. However, only paragraphs to avoid *copyright issues*.

G DONE

- Green book corrections of
  - Chapter 1
  - Chapter 2
  - Chapter 3
  - Chapter **??** (removed)
  - Some text from Chapter **??** into the embedded section of Chapter 3, Intro from it as part of the history section

H TODO for the 2nd Edition

I Other Stuff

I.1 Publishing the Book

Target is real book with ISBN (= publication).

- http://www.createspace.com/ no setup, no copyright transfer
- http://www.virtualbookworm.com/ $ 360 – $ 440, incl. Amazon only, 4 computer books, form mail sent, e-mail sent
- http://www.authorhouse.com/ $ 700.- incl. Amazon, no control on layout, form mail sent
- http://www.wingspanpress.com/ $ 500 incl. Amazon, no computer books, e-mail possible
- http://www.pagefreepublishing.com/ form mail sent, e-mail sent
- http://www.aventinepress.com/ e-mail sent
- http://www.iuniverse.com/ $ 399,- ($ 599,- incl. Amazon), Author discount, not many computer related books (old topics) – e-mail only possible with a Friends name and address!
- http://trafford.com/, EUR 750,-($ 1600,- incl. Amazon), EUR 11,- per book, some computer related books
- http://www.booksurgepages.com/amzn/ondemand/
- http://www.booksurge.com/ e-mail sent $ 300.-

Wishlist:

- ISBN

- A publisher who has serious technical books - no pornography!
- Cheap upfront
- Order possible from Amazon
- Cheap order for myself
- Right to sell it myself

Send an e-mail and ask following questions:

- Can I provide my own layout (submitting a PDF)?
- Can I still provide a free PDF version of my book on my web site?
- Can I update the content of the book (new edition)?
- Can I sell the book myself?
- Add a link to the manuscript.

No go's:

- `http://www.lulu.com/` cheap $ 99, but not clearely listed, freedom (e.g. GNU licence), no Amazon
- `http://www2.xlibris.com/` fixed layout
- http://superiorbooks.com/
- http://www.ubooks.de/

I.2 A Possible Structure of the Book

- Introduction
- A Quick Tour of JOP
  - Intro what JOP is
  - A Quick Start (Hello World) or Getting Started
  - A Short History

  - About this handbook (based on, organization)
  - VHDL Hello World
- Architecture (HW)
- Architecture (SW)
- Build.pdf
- Source organization
- Background information
  - JVM
  - GC
  - Related Work
  - Real-time systems
- WCET
- Real-time threads
- Tools
  - JOPizer
  - WCET Analyzer
- JDK + support
- Library: util, ejip, YAFFS,...
- Board descriptions
- Appendix
  - Microcode
  - Instruction timing
  - Javadoc for classes
- Further reading (related work) at the end of each chapter
- Browse other books for structural ideas

I.3 JOPizer Description

JOPizer is the tool to link all Java classes and translate the application to a executable image for JOP. The output is a .jop file (in ASCII with comments to debug the class structure). The executable image is either downloaded via serial or USB interface of programmed into a Flash. The image is located at address 0.

JOPizer performs following steps:

1. Find the transitive hull of all needed classes starting from the class that contains the main method.
2. ...
3. Find a static class initializer order (ClinitOrder). Throws an error on a cyclic dependency.
4. Dump the application to the image file

I.4 YAFFS Notes

- new in yaffs2.port.yaffs_guts_C.yaffs_GetObjectName(...)

I.5 Debugger Description

```
------------------------------------------------------------
Attachments: - Modified_jop_files.zip
  Some modified files needed to run the debugger

  Changed files:
  Makefile
  java/tools/source/com/jopdesign/build/MethodInfo.java
  java/tools/source/com/jopdesign/build/JOPizer.java
  java/tools/source/com/jopdesign/build/ClassInfo.java
  (there's no need to change RtThreadImpl anymore)

  Added (to the development tree):
```

```
  java/target/source/test/JopDebugKernel/
  java/tools/dist/lib/JopDebugger.jar   (after "make tools" run)

- JopDebugger.zip
  Latest development version of the debugger module.

- JopDebugKernel.zip
  This will run inside JOP.


  The changes in the makefile are to allow it to debug JOPizer and JopSim
  through the network and to create the symbol file.
  The last change modify one step in the build process,
  to call JOPizer from another class which will create the symbol file
  right after JOPizer runs.

------------------------------------------------------------



Instructions to run the example:

- Open two shells (one for Jop, other for the JopDebugger) - Shell
1: Download a clean development tree from Jop CVS.

- Copy modified Jop classes (from Modified_jop_files.zip)
  in the correct folders.
  Those small changes I did in the tool set were just to allow
  serialization and access to some fields.
  Copy also the "Makefile" included over your Makefile.

- Build the modified tools: make tools

- Shell 2:
  Decompress the attached JopDebugger.zip in a sibling folder.

- Build the project lib: cd JopDebugger; ant lib
```

```
- Copy the file JopDebugger.jar into java\tools\dist\lib
  Remeber that if you run "make tools" it will erase the
  lib folder, which will remove JopDebugger.jar.

- Unpack "JopDebugKernel.zip" and move it into
  jop\java\target\src\test\JopDebugKernel
  (but don't put it into JopDebugKernel\JopDebugKernel,
  this is a common mistake).

- Run the line below.
  This test will run a stand-alone test under JopSim. If it runs and print some
  simple messages, then it's possible to test the methods which access JOP's
  internal structures.

  make jsim -e P1=test/JopDebugKernel/source P2=debug P3=TestJopDebugKernel

This will build all the system as usual: compile all, run JOPizer,
bild the symbols file and launch JopSim.

However, there are a few differences in this makefile. 1) It allows
anyone to debug JopSim remotely (manually changing the makefile) 2)
It allows anyone to debug JOPizer remotely (manually changing the
makefile) 3) The modified makefile also change the build process to
store a symbol table by calling JOPizer from another tool (symbol
storage and loading is working) 4) It can connect JopSim to the
network by calling JopSim from another tool (a JopServer)


This test will run for a few seconds
 while and then will stop to wait for input, showing the text below.

 ----------------------------------------
 test4_read_write()
 Debug server. Current stack depth: 2
 ----------------------------------------
```

```
When this happens, force it to stop (ctrl-C). The remaining of the
test needs commands from a server (through the network), explained
next.


To connect JopSim to the network using JopServer:
-------------------------------------------------

- Open the .jop file created in the previous run.

- Locate two method pointers:
  search for: "TestJopDebugKernel.printValue(I)V" and also
  for "TestJopDebugKernel.printLine()V".

- Update the file com.jopdesign.debug.jdwp.test.TestJopServer
  (lines 152 and 162) with those addresses.

- Run the line below:

  make jsim_server -e P1=test/JopDebugKernel/source P2=debug P3=TestJopDebugKernel

- Wait until the server is running.
  It will print a message: "Server launched at port: 8004"

- After the Jop server start running, go back to the other shell and
  launch the ant target "TestJopServer". It should recompile the modified
  java sources and launch the test.

  ant TestJopServer

If the methods addresses are not corrected, there is a big chance
that the test will break with an error message like this: "Calling
method now: make: *** [jsim] Error 25584194"

Everytime JOPizer run, new method addresses are calculated when
classes change. This error is just due to a couple wrong method
pointers. So, if you want to change anything in the code which will
```

run inside Jop, it's necessary to fix this test (or comment the method invocation calls). Otherwise it will break.

To fix method pointers: - Open the .jop file - Search for the two method signatures printed by the TestJopServer class - Extract manually the two method addresses (hey, this is not finished yet;) it's just a test, you know). - Open class com.jopdesign.debug.jdwp.test.TestJopServer. - Change the code by pasting the method addresses

Launch again the JopServer using make Run again TestJopServer.

You may see that it's working by choosing other static void methods to call. This works for static methods with one or zero parameters only. Support for more is very easy to implement but was not needed now, so I didn't (yet). Returned values are currently ignored.

Well, that's it. There are a lot of details and other small tools, but I hope now you have a much better idea about how my development effort is going until here.

Those changes were tested on JOP from Sep. 11, so unless there was some big change in the last days, they should work ok on the latest version.

The first test I mentioned shows you evidence of methods that access JOP structures.

The test with JopServer shows that it's possible to connect JopSim to the network and send/receive commands which access and change JOP internal structures. Since this was accomplished without changing JopSim, it should be possible (at least in thery) to switch it with an FPGA running JOP and get the same results.

This shows it's possible to control JOP from the network and perform some tasks related to debugging (such as get/set values).

It's possible also to run another test which launch a real debugger which connects to a server, handshake and receive JDWP commands. But this test still does not do much, so I don't know if you want to see it.

There's also a test to check the symbol file content.

About the JOP side, it's still necessary to write code to set/reset breakpoints and handle interruptions properly.

It's also necessary to glue together the symbol file, the Jop debug server and the JDWP module to control debugging. In short, it's a lot of work.

If you need help to run or test it or if you have questions, please let me know.

Regards, Paulo

I.6 TODO JOP

- Javadoc for the handbook