Tero Säntti

A Co-Processor Approach for Efficient Java Execution in Embedded Systems

Turku Centre *for* Computer Science

TUCS Dissertations

No 108, September 2008

A Co-Processor Approach for Efficient Java Execution in Embedded Systems

Tero Säntti

To be presented, with the permission of the Faculty of Mathematics and Natural Sciences of the University of Turku, for public criticism in Auditorium Alpha on November 10, 2008, at 12 noon.

University of Turku
Department of Information Technology
University of Turku, FI-20014 Turku, Finland

2008

Supervisors

Adjunct professor, Ph.D. Juha Plosila
Department of Information Technology
University of Turku
FI-20014 Turku
Finland

D.Sc. (Tech) Seppo Virtanen
Department of Information Technology
University of Turku
FI-20014 Turku
Finland

Reviewers

Professor Dake Liu
Department of Electrical Engineering
University of Linköping
S-58183 Linköping
Sweden

Professor Timo D. Hämäläinen
Department of Computer Systems
Tampere University of Technology
FI-33101 Tampere
Finland

Opponent

Professor Jan Madsen
Department of Informatics and Mathematical Modeling
Technical University of Denmark
DK-2800 Lyngby
Denmark

ISBN 978-952-12-2155-2
ISSN 1239-1883

Abstract

This thesis deals with a hardware accelerated Java virtual machine, named REALJava. The REALJava virtual machine is targeted for resource constrained embedded systems. The goal is to attain increased computational performance with reduced power consumption. While these objectives are often seen as trade-offs, in this context both of them can be attained simultaneously by using dedicated hardware. The target level of the computational performance of the REALJava virtual machine is initially set to be as fast as the currently available full custom ASIC Java processors. As a secondary goal all of the components of the virtual machine are designed so that the resulting system can be scaled to support multiple co-processor cores.

The virtual machine is designed using the hardware/software co-design paradigm. The partitioning between the two domains is flexible, allowing customizations to the resulting system, for instance the floating point support can be omitted from the hardware in order to decrease the size of the co-processor core. The communication between the hardware and the software domains is encapsulated into modules. This allows the REALJava virtual machine to be easily integrated into any system, simply by redesigning the communication modules. Besides the virtual machine and the related co-processor architecture, several performance enhancing techniques are presented. These include techniques related to instruction folding, stack handling, method invocation, constant loading and control in time domain.

The REALJava virtual machine is prototyped using three different FPGA platforms. The original pipeline structure is modified to suit the FPGA environment. The performance of the resulting Java virtual machine is evaluated against existing Java solutions in the embedded systems field. The results show that the goals are attained, both in terms of computational performance and power consumption. Especially the computational performance is evaluated thoroughly, and the results show that the REALJava is more than twice as fast as the fastest full custom ASIC Java processor. In addition to standard Java virtual machine benchmarks, several new Java applications are designed to both verify the results and broaden the spectrum of the tests.

Acknowledgements

It gives me great pleasure to be able to express my gratitude to the individuals and institutions that have influenced and enabled the research in this thesis. First and foremost, I would like to thank my supervisor Ph.D. Juha Plosila for his continuous support, guidance and inspiration during the work leading up to this thesis. I am also very grateful to D.Sc. (Tech) Seppo Virtanen, who examined this thesis thoroughly and provided excellent comments.

I also wish to thank professors Dake Liu from University of Linköping and Timo D. Hämäläinen from Tampere University of Technology for their detailed reviews of this thesis and for providing insightful comments and suggestions for improvement. Their recommendations and suggestions raised the quality and the clarity of the final manuscript.

I gratefully acknowledge the funding from the late Department of Applied Physics and the Department of Information Technology during the first part of my postgraduate studies. The second part was funded by the "Teknologiateollisuuden 100-vuotissäätiö" -foundation of the Federation of Finnish Technology Industries, for which I am very grateful. The funding from the foundation allowed me to concentrate on the research and bring the work to a conclusion. I would also like to thank the Turku Centre for Computer Science (TUCS) for their financial support.

I wish to thank all the colleagues and co-workers in the Department. Especially Joonas Tyystjärvi is thanked for participating in the research and co-authoring several publications. I also wish to thank professors Jouni Isoaho, Ari Paasio and Johan Lilius (from Åbo Akademi) for all the discussions related to the research in this thesis and other topics as well. The legendary group of people from the room 013 should also be mentioned. It has been a pleasure and a privilege to share the journey from a bunch of wannabe scientists to something more substantial with each of you. The office staff and the technical personnel deserve a special acknowledgement for all the help and support in their respective fields throughout the years.

My friends have also been supporting and understanding during my doctoral studies. Especially all the wonderful people from Hayashi and Three Beers are thanked for listening to my ramblings about the research, and for providing occasional breaks from the work. The multidisciplinary discussions on all topics gave me much needed perspective at times.

I am grateful to my mother Merja Säntti and my brother Kari Säntti for their support and encouragement over the years. I also wish I could have shared these moments with my late father Jari Säntti, who always encouraged me to do my best at whatever I choose to start. Remembering his advice helped me in the darkest times during this work.

My final, most personal and sincerest thanks go to Johanna Tuominen. You have always kept the faith in me and my research. Your love and support in all areas of life have been the cornerstones of my life.

Turku, September 2008

Tero Säntti

Contents

List of Figures

List of Tables

List of Abbreviations

3D	three Dimensional
AAL5	ATM Adaptation Layer 5
ALU	Arithmetic Logic Unit
AMBA	Advanced Microcontroller Bus Architecture
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
ATM	Asynchronous Transfer Mode protocol
ATX	Advanced Technology Extended
AWT	Abstract Window Toolkit
CAM	Content Addressable Memory
CDC	Connected Device Configuration
CISC	Complex Instruction Set Computer
CLDC	Connected Limited Device Configuration
CO	Code Offset pointer
CPLD	Complex Programmable Logic Device
CPU	Central Processing Unit
DCR	Device Control Register
DDR	Double Data Rate
DSP	Digital Signal Processor
EDK	Embedded Development Kit
EDP	Energy-Delay Product
FAT	File Allocation Table
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GCC	GNU Compiler Collection
GPR	General Purpose Register
GUI	Graphical User Interface
GSM	Global System for Mobile communications
I/O	Input/Output
IDE	Integrated Drive Electronics
ILP	Instruction Level Parallelism
IP	Intellectual Property

IP	Internet Protocol
IRQ	Interrupt ReQuest
ISA	Instruction Set Architecture
J2EE	Java Enterprise edition
J2ME	Java Micro Edition
J2SE	Java Standard Edition
JAR	Java ARchive
JIT	Just In Time compilation
JNI	Java Native Interface
JPU	Java Processing Unit
JVM	Java Virtual Machine
LO	LOcal variable and parameter count register
LPT	Line Print Terminal
LUT	LookUp Table
LV	Local Variable pointer
MAC	Multiply-ACcumulate
MIDP	Mobile Information Device Profile
MOS	Metal Oxide Semiconductor
MP3	MPEG-1 audio layer 3
MSC	Message Sequence Chart
NoC	Network-On-Chip
NOP	No OPeration
OPB	On-chip Peripheral Bus
PC	Program Counter
PCI	Peripheral Component Interconnect
PDA	Personal Digital Assistant
PLB	Processor Local Bus
RAM	Random Access Memory
RISC	Reduced Instruction Set Computer
ROM	Read Only Memory
RTSJ	Real-Time Specification for Java
SDRAM	Synchronous Dynamic Random Access Memory
SOC	System-On-Chip
ST	Stack Top pointer
TCL	Tool Command Language
USB	Universal Serial Bus
VHDL	VHSIC Hardware Description Language
VHSIC	Very-High-Speed Integrated Circuit
VLIW	Very Long Instruction Word
VLSI	Very-Large-Scale Integration
XVM	unknown Virtual Machine

Chapter 1

Introduction

With the rising popularity of wireless systems, there is a proliferation of internet-enabled embedded devices, most notably PDAs and mobile phones. In this context Java is emerging as a standard execution environment due to its security, portability, mobility and network support. Java technology allows easy access to utilities like calendars, planners and email clients as well as entertainment like media players and games. The applications can be installed to the device over a wireless internet connection or with a data cable provided by the manufacturer. However the limitations in both storage space and computational power cause certain Java execution techniques, such as just in time compilation (JIT), to be unusable in this application domain. Yet the consumers demand faster systems with more capabilities and longer battery lifes. This conundrum has sparked several methodologies to reduce the overheads in Java execution.

The overheads are due to the fact that Java applications are not written, nor compiled, for any given hardware device. In fact they are written and compiled for a *virtual machine*, the Java virtual machine (JVM). The Java applications are executed by emulating the Java virtual machine on the host system. Clearly the extra emulation layer causes overheads, but it also provides opportunities for several improvements, like increased security and platform independent programming models. Also power management can be included at the virtual layer, by using slower but more energy efficient routines when the battery is running low.

Java provides a rich standard library, which includes methods for network access and wireless communication. The library is included in the virtual machine, so the actual user application does not have to copy the code imported from the library. This keeps the deployable size of a given Java application quite small, a feature well suited for portable devices with

limited storage capacity and communication bandwidth. The richness of the standard library also increases programmer productivity, by reducing the need to "reinvent the wheel".

Java is very popular and portable, as it is a write-once run-anywhere language. This enables coders to develop portable software for any platform. Java code is first compiled into bytecode, which is then run on a Java Virtual Machine. The JVM acts as an interpreter from bytecode to native microcode, or more recently uses JIT to affect the same result a bit faster at the cost of increased memory usage. This software only approach is quite inefficient in terms of power consumption and execution time. These problems rise from the fact that executing one Java bytecode instruction requires several native instructions. For instance, the research of Radhakrishnan et al. [44] showed that a single Java bytecode instruction translated to approximately 25 native instructions on the SPARC processor architecture used in that study. Also the fact that the Java virtual machine instruction set architecture (ISA) is stack based causes inefficiencies when executed on current processors, which typically have register based ISAs. Another source for inefficiency is the cache usage. As the JVM is the only part of software running natively, it occupies the instruction cache, whereas the Java bytecode is treated as data for the JVM, hence being located in the data cache. Also the actual data processed by the Java code is assigned to the data cache. This clearly causes more memory accesses missing the cache.

When the execution of the bytecode is performed on hardware many of the problems in the software approach are avoided and the overall amount of memory accesses is reduced. Some of the performance gain comes from the inherent parallelism available in the hardware domain. For instance, when using hardware, the Java program counter can be updated in parallel with the execution of an instruction, while in software these steps would have to be performed sequentially. The hardware can update most of the internal registers in similar fashion. Also the addition of a relatively small local memory to the hardware domain reduces the system wide physical memory accesses by removing unnecessary cache misses and keeping the temporary data locally cached. The cache performance is further improved by keeping the Java bytecode in the local memory, freeing the data cache for the actual data to be processed.

Since the Java virtual machine has the extra layer between software and hardware, it often is not suitable for low level hardware control, device drivers and hard realtime parts of a given embedded system. Using a full standalone Java processor in the system would be likely to require a general purpose processor for the low level accesses. This would make system in-

tegration both difficult and expensive. The solution proposed in this thesis is to take the best parts from both the full software and the full hardware approaches. This is done by using a co-processor for the execution of the Java applications. The co-processor approach provides easy integration with existing systems, with the execution speed of a standalone hardware Java virtual machine. The resulting system needs no special concerns related to accessing I/O devices and other services, since they are provided by the general purpose host processor. The integration is easier than in the standalone solutions since there is no need for sharing resources between two fully autonomous processors.

1.1 Objectives

The main objective in this thesis has been to design and prototype an efficient hardware accelerated Java virtual machine targeted for embedded systems. The secondary objective has been to develop understanding of using hardware in Java execution and behavior of Java virtual machines in embedded context. The efficiency of the resulting REALJava virtual machine is evaluated in terms of execution time and power consumption for a given application. Initially the computational performance is targeted to be at least as fast as the currently available full custom ASIC Java processors. No specific performance or timing requirements are set, since the REALJava is not designed for any predefined Java application, but rather for use as a general purpose Java virtual machine. The hardware acceleration is hidden inside the virtual machine so that both the Java programmers and the end users running the Java applications are left unaware of the acceleration, save for the increased performance. The research problems to be solved include:

1. Partitioning of the virtual machine. In order to use a hardware acceleration scheme the Java virtual machine is partitioned so that the platform specific parts are implemented in software while as much as possible of the actual Java bytecode execution is implemented in hardware. The partitioning has an impact on the number of communications between the two domains as well as on the overall efficiency of the resulting virtual machine.

2. Communication between the hardware and the software. The communication medium and protocol have a significant impact on the efficiency of the system. Thus the communication subsystem needs to be carefully tuned to avoid bottlenecks. The communication is to be separated from the rest of the virtual machine, both in hardware and software do-

mains, so that the virtual machine can be easily ported to new systems by redesigning only the communication modules.

3. Software partition. The virtual machine requires a software partition with support for the hardware acceleration. Besides the co-processor, this partition also controls the data structures to be kept in the memory region of the main CPU. Garbage collection and I/O also fall into the software domain, since they are highly platform[1] dependent.

4. Hardware partition. An efficient stack based hardware architecture is required in order to provide sufficient performance. The hardware partition is designed initially targeting asynchronous ASIC technologies, but it is prototyped using FPGA technology. The transition from one technology to the other requires several paradigm shifts, since the FPGAs have very different characteristics when compared to ASICs and especially asynchronous systems.

5. Performance issues. Once the virtual machine has been built from the previous parts, it needs to be analyzed and evaluated. The observations made here will be reflected back to the previous problems and they are fine tuned accordingly. The measurement data gathered will also be used to show the feasibility of the chosen strategies and technologies.

All of the above mentioned problems are to be solved using techniques that allow system level scaling in the future. This design choice is motivated by the current trend towards multicore processor systems and even more so by the inherent multithreading support in Java. Since the Java programming language supports multithreading readily at the language level, it is reasonable to integrate several co-processor cores into a single system. Each of these cores can then execute a Java thread of their own, in a truly parallel fashion.

The research presented in this thesis is a part of the REALJava project, which aims to design a Java co-processor that is easily integrated to various systems. Asynchronous techniques have been chosen for this project because that allows the virtual machine to achieve good performance with reasonable power consumption and very easy integration with existing systems, since no clock limitations need to be considered. Asynchronous self-timed circuit technology [50], where timing is based on local handshakes between circuit blocks instead of a global clock signal, provides a promising platform

[1]The platform refers to the underlying system, upon which the Java applications are to be executed. This includes the operating system and all of the hardware components in the system.

for obtaining a highly modular low-power and low-noise Java virtual machine implementation. The conceptual model of the REALJava is entirely based on the asynchronous design paradigm, both internally and externally. However, since prototyping is done using FPGAs, the prototypes use synchronous techniques. This is due to the inherent synchrony of the FPGA technology. Asynchronous systems prototyped with FPGAs would suffer significant performance penalty for artificial synchronization between modules.

1.2 List of Publications and Research Contributions

The research towards this thesis has been partly included in the following papers (in chronological order):

(1) T. Säntti and J. Plosila, **Communication Scheme for an Advanced Java Co-Processor**, *In Proc. Norchip 2004*, Oslo, Norway, November, 2004

(2) T. Säntti and J. Plosila, **Architecture for an Advanced Java Co-Processor**, *In Proc. ISSCS 2005*, Iasi, Romania, July, 2005

(3) T. Säntti and J. Plosila, **Instruction Folding for an Asynchronous Java Co-Processor**, *In Proc. 2005 International Symposium of System-on-Chip*, Tampere, Finland, November, 2005

(4) T. Säntti and J. Plosila, **Real Time Flow Control for an Advanced Java Co-Processor**, *In Proc. Norchip 2005*, Oulu, Finland, November, 2005

(5) T. Säntti, J. Tyystjärvi and J. Plosila, **Java Co-Processor for Embedded Systems**, In *Processor Design: System-on-Chip Computing for ASICs and FPGAs*, J. Nurmi, Ed. Kluwer Academic Publishers / Springer Publishers, ch. 13, pp. 287-308, 2007

(6) T. Säntti, J. Tyystjärvi and J. Plosila, **FPGA Prototype of the REALJava Co-Processor**, *In Proc. 2007 International Symposium of System-on-Chip*, Tampere, Finland, November, 2007

(7) T. Säntti, J. Tyystjärvi and J. Plosila, **A Novel Hardware Acceleration Scheme for Java Method Calls**, *In Proc. ISCAS*, Seattle, Washington, USA, May, 2008

The general principles of the REALJava virtual machine have been developed by the author. All of the hardware units presented in (1-7) have been designed by the author. The supporting software presented in (5-7) has been implemented by Joonas Tyystjärvi, save for the communication routines, which have been developed by the author. The research related to the software/hardware co-design issues has been carried out jointly by the author and Joonas Tyystjärvi. The same division of labor has also been used for the parts of the research that are not published in (1-7). All of the hardware and software components of the REALJava virtual machine have been designed completely from scratch, instead of modifying any previously existing systems. The only deviation from this is the Java standard library, for which the GNU Classpath [92] is used.

The research carried out for this thesis resulted in the following contributions (listed in the order they appear in this thesis):

- **Conceptual model of a hardware accelerated Java virtual machine.** The model outlines the basic mechanisms required for execution of Java applications using a co-processor to enhance the performance. The model is loosely based on the principles presented in [36], and it has been further improved by the author while knowledge of the behavior of the hardware assisted virtual machine has increased.

- **Asynchronous stack based co-processor architecture.** The architecture is designed for asynchronous environments, and it includes a modified pipeline structure tailored to suit the execution of Java bytecode. The architecture is based on the author's observations on the properties of the Java bytecode and general purpose processor architecture presented in several textbooks, including [23].

- **Asynchronous instruction folding unit.** The instruction folding as a concept is developed from the approach used in [19]. The structure of the folding unit and the instruction classification are results of the author's research.

- **Software partition of the co-designed virtual machine.** The software partition is designed form scratch, without any reference design. The architecture and functionality of the software is based on the Java virtual machine specification [38] and the additional commentary in [41]. Most of the work related to this partition has been done by Joonas Tyystjärvi, while the author's involvement has been mainly on the hardware/software co-design issues and the communication protocol and routines.

- **FPGA prototypes.** The prototypes are completely designed by the author, save for the board design. The prototypes are based on the architecture mentioned earlier, but they required modifications in order to make them suitable for the FPGA environment. All of the research and design here is the work of the author, save for the internal structures of the units implemented with the Xilinx CoreGenerator.

- **Java applications for identifying performance hindrances.** Several small Java applications were developed by the author with support from Joonas Tyystjärvi. Most of these applications simply measure the time taken for a given instruction or a class of instructions. The data gathered from the prototypes of the REALJava and other execution engines was then analyzed to find the instructions with relatively weak performance. Deeper analyses on the execution of those instruction resulted in discovery of most of the performance enhancements that follow. The analyses were performed by the author.

- **Stack caching and data transport architecture.** The stack in the prototypes was not cached using a ring buffer. Instead a new cache architecture was designed by the author. The cache is more efficient in terms of hardware resources on an FPGA and provides good performance, as shown in 5.2.

- **FPGA compatible pipeline structure.** The pipeline structure designed for asynchronous environments is not usable as such in the FPGA domain. The modifications to the architecture were designed by the author, and most of the reasons for the modifications were obtained from the Xilinx documentation and the general properties of FPGA devices.

- **Partial instruction folding.** Since the folding unit was initially designed for asynchronous environments, it was left out from the prototypes. The benefits of the data loading parts of the folding are obtained with the new stack cache mentioned previously, leaving the data saving parts so far unhandled. The partial folding addresses these, and provides the in effect the same results. This technique was discovered and designed by the author.

- **Method invocation acceleration.** The method invocation accelerator is entirely based on the research of the author. The additional benefits were discovered during the analyses of the invocations that are not cacheable.

- **Constant caches.** The constant caches are also based on the author's analyses on the execution of Java applications. Identification of the

instructions suitable for this and the structures needed for the caches are results of the author's research.

- **Time control mechanisms.** The basic ideas behind using hardware timers came from the author, while the application of timers in the thread time slicing was a result of hardware/software co-design by the author and Joonas Tyystjärvi. The timers were designed by the author, with influences taken from the timer structures found in Microchip's [91] PIC line of microcontrollers.

- **Control register bank with additional functionality.** The added functionalities in the control register bank are results of the author's research. The purpose for designing most of the additions was to reduce the communication between the host CPU and the co-processor, while others provide enhanced debugging capabilities.

- **Multicore Java virtual machine.** While the multicore support in the REALJava virtual machine is still rudimentary, the research has from the start been carried out with the requirements of adopting a multicore approach in mind. The software support required for the multicore version was developed by Joonas Tyystjärvi, while the hardware was designed by the author. The adoption of a multicore approach was motivated by the current trends of using several cores instead of one very powerful core. The execution model is based on the research of the author with the properties and limitations of the multithreading model used in Java.

- **Java applications for performance evaluation.** Several Java applications were designed jointly by the author and Joonas Tyystjärvi to validate the results of publicly available benchmarks and to expand the application spectrum used for the performance evaluations. The author also ported some of these to be suitable for execution on a mobile phone. This was done in order to get yet another reference system for the comparisons.

These contributions should be usable in the design of Java hardware assisted executions engines, regardless of whether they are co-processors or standalone Java processors. The suitability of a given technique is heavily dependent on the surrounding architecture, the provided services and the logical layout of the various data segments inside the virtual machine. The stack cache architecture can be used in any stack based architecture, possibly requiring modification to the depth of the cache depending on the instruction set of the target architecture. The Java applications used for tests and analyses are readily usable for similar testing of any Java virtual

machine, since the applications are standard Java applications without any customizations or modifications. The contributions are also expected to survive the increase in memory sizes of embedded devices caused by lowering memory prices. This can be justified by assuming that as memory sizes grow, the relationship between the memory sizes of desktop computers and embedded systems will remain. Also the Java applications will become larger as the amount of memory in desktop computers is increasing. These two assumptions together suggest that embedded systems will remain memory limited in the foreseeable future. Additionally the multicore approach will scale the performance to allow the REALJava and it's subsystems be viable far in to the future, even with heavier applications.

1.3 Overview of the Thesis

After having a look at the related work, the rest of the thesis is organized as follows. In Chapter 2 the structure of a generic Java virtual machine and the basics of the Java programming language are described. Chapter 3 describes the REALJava virtual machine at a conceptual level. Both the hardware based co-processor and the software partition of the virtual machine are presented. Chapter 4 presents the details of the FPGA prototypes of the REALJava virtual machine. Based on the analyses on the performance of the FPGA prototypes, modifications to the architecture are presented in Chapter 5. Measurement data of the performance of the REALJava virtual machine is presented in Chapter 6, with comparisons to existing Java systems. Finally, in Chapter 7 conclusions are drawn and future efforts related to the REALJava virtual machine are discussed.

1.4 Related Work

This section discusses various approaches that can be deemed relevant for and related to the research presented in this thesis. First a brief glance is directed at the software only solutions. Then some hardware based approaches are discussed. In the listings of both the software and hardware solutions only a representative selection is given, showing a wide spectrum of approaches. The representatives for the approaches are chosen by the author based on the commercial success or the number and quality of scientific data available, for commercial products and research projects respectively. Finally, a hardware/software co-designed Java virtual machine with a relatively similar approach to the REALJava virtual machine is discussed in detail.

1.4.1 Software Solutions

Java virtual machines are usually implemented in software only. This approach is currently dominant in the embedded application domain as well as in the desktop computer domain. In the embedded systems domain the Java virtual machine must cope with two major challenges: limited memory resources and power consumption. This renders the most advanced and aggressive techniques of dynamic compilation unsuitable in most cases. However there is still some room for optimization and light weight recompilation at the run time. The most notable implementation in this field is the J2ME from Sun, but other implementations for embedded environments do exist. As examples, a few software based Java systems are introduced next. The first complete Java virtual machine that used the GNU Classpath was Japhar [2] [93]. It has been built from the ground up without consulting Sun's sources, and thus qualifies for "cleanroom status". Open source solutions have also been provided by corporate entities, such as the Open Runtime Platform (ORP) [94] and the Jikes RVM [2] initially developed by Intel and IBM, respectively. Both of them are currently maintained as SourceForge [95] projects. Academic research has produced several Java virtual machines, with varying target focuses. One of the many interesting systems is the SableVM [16, 96], which is designed especially to be portable and easy to extend for research purposes.

Maybe the most well known and best documented free software Java Virtual Machine is the Kaffe [97]. It has been used as a reference design in several studies related to JVMs, including topics like JIT optimization and garbage collection. As examples Latte [63] and Jessica [39] can be mentioned. Also multitasking inside the Java virtual machine has been studied based on the Kaffe architecture, in the KaffeOS [4, 5] and Janos [60] projects.

1.4.2 Hardware Solutions

It is currently unrealistic to consider implementing an entire embedded software project in Java. For one thing, Java does not include a mechanism for directly accessing memory or hardware registers. This means that for instance the GSM communication of a mobile phone is hard (or even impossible in some cases) to implement using purely Java based software. Furthermore the execution speed and the unpredictability in time domain make Java quite unsuitable for hard realtime applications, such as handling the

[2] Also known as the Hungry Programers' Java VM.

data streams over GSM. So there will always be a need for device drivers and other pieces of supporting software written in C/C++ or assembly. This kind of low-level software might either be called from Java, in which case it is said to be a native method, or run as a separate thread of execution, in parallel with the Java runtime environment.

There are two main solution types for hardware acceleration in a Java virtual machine. The first one is designing the whole processor with Java bytecode instruction set, resulting in a system that needs no other processor. This solution is referred to as standalone solution and the PicoJava [19] developed by Sun Microsystems is a well known example of this approach. The other solution is to use a general purpose processor as the CPU and to add a co-processor to boost the performance when Java execution is required. The work in this thesis uses this approach, as does for instance the Jiffy [1]. Virtual machines based on both strategies have been researched to some extent, and full commercial solutions are also available. The research problems presented in Section 1.1 Are not directly to any of the solutions presented here, due to the differences in the approaches. One research project exists with closely matched approach, and it is given a deeper look to highlight the similarities and differences to REALJava. The analysis can be found in Section 1.4.3.

Standalone Java Processor Solutions

The processors in this group usually are limited in their instruction set, or support only a subset of the bytecodes. Also the classpath support may be limited. A list of the most notable representatives is presented in Table 1.1. The commercial solutions selected for this table are **Picojava**, **AJile**, **Cjip**, **Lightfoot** and **TINI**. In the research field the representatives are **FemtoJava** [28], focusing on minimizing the size of the Java processor, **JOP** [48], which focuses on realtime performance and **SHAP** [64], focusing on the security aspects of Java.

Co-Processor Approach

The co-processor approach can be further divided into two categories, namely the hardware interpreters and execution co-processors. The hardware interpreters take in Java bytecode stream and give a native instruction stream to the CPU. The executing co-processors actually perform the instructions in the Java bytecode stream. Some examples are listed in Table 1.2. The commercial representatives are **Jazelle** and **JSTAR**. In the research field

Processor	Implementation	Notes
PicoJava	FPGA	J2SE, never entered production
AJile	ASIC	J2ME
Cjip	ASIC	J2ME
Lightfoot	ASIC & FPGA	Available as IP core, J2ME
FemtoJava	FPGA	Subset with 16-bit ALU
TINI	ASIC	Enhanced 8051 running SW JVM
JOP	FPGA	Subset, requires precompilation
SHAP	FPGA	Realtime, subset, requires precompilation

Table 1.1: A short list of standalone type Java processors.

the selected projects are **REALJava**, the topic of this thesis focusing on performance and scalability, **Hard-Int**, which focuses on the on chip translation and **JIFFY**, focusing on JIT-like optimizations during the on chip translation.

Processor	Implementation	Notes
REALJava	FPGA	The topic of this thesis
Hard-Int	Simulation only	Translator
JIFFY	FPGA	Translator, JIT on FPGA
Jazelle	ASIC	Subset, ARM architecture only
JSTAR	ASIC	J2ME

Table 1.2: A short list of co-processor type Java processors.

1.4.3 Co-Designed Java Virtual Machine

A co-designed Java virtual machine has been proposed by Kent et al. in [32, 40]. The architecture they suggest is based on the co-processor approach, but it is targeted at desktop computers. The PCI bus has been adopted as the communication medium, limiting the usability of the virtual machine to systems with PCI support. The architecture is based on recognizing sequences of the Java application bytecode that are suitable for execution on the co-processor, and adding instructions to transfer the execution to the hardware. This is done during the class loading. At the same time instructions are added for returning the execution to the software, at the end of the hardware sequences. In their architecture the storage size of a Java method is further increased by alignment of the Java bytecodes. The

software partition of their virtual machine implements the complete instruction set of Java, and thus causes the executable file to be larger than in the REALJava virtual machine, which only implements a subset of instructions on software. The partitioning between hardware and software is also different. The most critical difference is that the local variables are handled in the software, whereas the REALJava system keeps them in the hardware. Since the transfers between the local variables and the stack top are very common in Java, locating them both to the hardware has a huge impact on the performance. Also the way methods are invoked and returned from is different. Their architecture executes all of the method changes in software while the REALJava performs most of these in hardware. As the good programming practices outline, most of Java applications are composed of a large number of relatively small methods. This causes the method invocation subsystem to have a major impact on the overall performance of the virtual machine. As a consequence of the method handling, their system has access to only one method's bytecode at a time. When the method is changed the hardware needs to retrieve the code segment again. In the REALJava the bytecode segments of the previously used methods are kept in the co-processors local memory, thus greatly reducing the amount of communication required for transferring data from the CPU to the co-processor. Finally the REALJava is both smaller in terms of LUTs used on the FPGA implementation and faster in terms of clock speed. In light of the publications related to their work with the Java virtual machine [20, 22, 29, 30, 31, 32, 40], it seems that the research has not led to an actual running virtual machine with hardware support. Rather than measuring actual performance, all of the data in the publications is obtained from simulations, and furthermore they often ignore the impact of communication between the co-processor and the host system. As a result no real performance metrics are available and this system is not used as a reference in the comparisons in Chapter 6.

1.4.4 History and Current Trends

This section reflects the authors personal views on the field of hardware accelerated Java execution. It is provided in order to shed light on the general history of the field, and also to try and predict the trends effecting the future of the field. In the past the efforts in the hardware acceleration of Java virtual machines have mainly been focusing on desktop computers and standalone Java chips or CPU specific co-processors on the embedded systems. The hardware acceleration in the desktop environment is rather challenging, mainly due to the fact that the raw number crunching performance of the CPUs is currently good enough to give very good results with software only solutions. Also the memory sizes in the desktop computers

are sufficient for JIT and other techniques to usable. Hardware accelerators running on FPGAs at mere hundreds of megahertzs are no mach for the CPUs running at several gigahertzs. In the embedded systems domain the Java chips are limited by their poor ability to be integrated into existing systems, causing the device manufacturers to prefer software solutions if a CPU specific co-processor is not available. The fact that the PicoJava Java processor never was produced commercially may also have dampened the research activities. If a major company in the field of Java virtual machines fails with their hardware implementation, how could smaller operators do better? The commercial success of the CPU specific co-processors on the other hand is very tightly coupled to the success of the CPU. Also being CPU specific naturally decreases usability across different platforms. The most successful CPU specific co-processors are designed by the CPU manufacturers, and they are closely integrated to the core of the CPU. The interest in the embedded system domain has not been a focus area in Java virtual machine development for more than about six to eight years, and of that time definitely most has been spend on the software only solutions and refining the J2ME specification.

The factors mentioned before have kept the academic interest towards the hardware acceleration of Java virtual machines relatively low, but now the efforts in the field are expected to rise. This is true especially in the embedded systems domain. This is partly motivated by the user device manufacturers, who are starting to move towards highly modular systems composed of several functional units and accelerators besides the main CPU. Like the desktop computers have done during the past few years, the embedded systems are also moving towards multicore platforms. The difference between the desktop computers and the embedded systems is that the multicore approach in desktop domain is currently homogeneous, using several identical CPUs, while the embedded systems domain is moving towards heterogeneous multicore systems, composed of several different processing elements. This trend was clearly presented in the NoTA conference [98], aiming to agree on a unified way to integrate, communicate and define module interfaces in embedded devices. The presentations, such as [27], were often motivated by statements that multicore technologies are now emerging in the embedded systems domain. Several of the case studies presented also contained highly specialized co-processors or functional units. This trend validates the field of the research presented in this thesis, and also shows that the industry is also starting to move towards that direction.

Chapter 2

Java Virtual Machine Technology

This chapter provides essential background information of the Java programming language and the Java virtual machine. The chapter also provides context for the work. Emphasis is given to the Java virtual machine specification, and some implementation techniques are also discussed. Finally the use of hardware to enhance the execution of Java applications is briefly touched.

2.1 Advantages of Virtual Machines

One of the most important reasons to use a virtual machine is that the code is write-once run-anywhere. This means that the code needs to be compiled only once, then it can be run on any platform, even over a network connection. Another important advantage is that new devices need only a new virtual machine implementation to run all existing software. This reduces the development cost of a new generation of devices, as well as time to market.

Even though the reasons listed above are very attractive, especially to industry, if not the consumer, they are not the only reasons to use virtual machines. Virtual machines provide improved security features: the additional layer between the code and the executing hardware can be used to increase security. While making it hard for you to shoot yourself in the foot, it also makes it harder for others to shoot you in the foot. Software downloaded from the internet can be verified to ensure it is original and not malevolent.

The security advantages are also present in some fully interpreted languages, such as TCL and JavaScript. The difference here is in the execution speed. Virtual machines get some kind of precompiled input files (Java bytecode for example), and thus they need less run-time interpreting and manipulation of the source code resulting in better performance. According to [41] TCL has been (informally) measured to be up to 200 times slower than fully compiled C++, where as precompiled languages fall in the range of 10 to 20 times slower than C++. With modern technologies, such as JIT (Just In Time compilation), the difference is dropping below "only" 5 times slower. The performance is still significantly below C++, due to the fact that a C++ compiler can optimize register allocation, whereas a JIT compiler has to work starting with Java bytecode operating on a stack.

2.2 Motivation for Choosing Java

Although platform independence has been hailed as Java's greatest strength, it is equally important to note that it is easier to produce bug-free software in Java than in C or C++. Java has been designed from the ground up to produce code that is simpler to write and easier to maintain. Even though the developers of Java based the language on the syntax of C, they eliminated many of that language's most troublesome features. These features sometimes make C hard to understand and maintain, and they frequently lead to undetected programming errors. Here are just a few of the improvements:

1. All of Java's primitive data types have a fixed size. For example, an integer is always 32 bits long in Java. Other languages for embedded systems have different assumptions based on the architecture.

2. Automatic bounds-checking prevents the programmer from writing or reading past the end of an array. Typically operating systems limit memory accesses to the threads allocated space, but they make no guarantees on the object accessed.

3. All test conditions must return either true or false. Common mistakes in languages with C/C++ like syntax, such as *if (x = 0)* $\{\ldots\}$ [1], are detected at compile-time, thus eliminating an entire set of bugs.

4. Built-in support for strings and string manipulation allows statements like "Hello, " + "world!".

[1] Using syntax like in C/C++, $x = 0$ is an assignment, while a condition testing should be $x == 0$.

In addition, Java is an object-oriented language similar to C++. This forces software developers to structure their data and functions into logical units called classes. Encapsulation, polymorphism, and inheritance are all available and are used extensively in the built-in class libraries. Java simplifies inheritance by eliminating multiple inheritance and replacing it with interfaces. It also adds many new features that are not available in C++, most notably:

1. Automatic garbage collection simplifies dynamic memory management and eliminates memory leaks.
2. Built-in thread library makes applications written in Java more portable by providing a consistent thread and synchronization interface across all operating systems and target devices.
3. An integrated exception mechanism organizes software exceptions into a logical class hierarchy and does not allow programmers to ignore them.

Finally, Java is extensively used in teaching of object oriented programming. This gives rise to a very large base of programmers who are familiar with the language. With so many coders well versed in the Java language, both industry and academia can start new software projects based on the Java technology and feel confident in the fact that a team with the necessary skill level can be hired easily.

2.3 Java Language

Java has become very popular lately. Some of the main features that have contributed to the success of Java are listed below.

- Portability: The application source code is compiled into bytecode format, which is architecture neutral. This is the "native" format of the Java Virtual Machine. Since the bytecode format is not platform dependent it makes write-once run-anywhere possible.
- Object oriented: Java is quite a simple language, with the look and feel of C/C++, resulting in easy transitioning for programmers familiar with C/C++. The object model of Java is simplified by allowing only single inheritance of implementation (ie. a subclass can be the extension of only one superclass). However a class can implement multiple interfaces, providing a mechanism for multiple inheritance.

- Standard library: A Java runtime system provides a rich standard library, called the Classpath. This helps to increase the productivity of programmers. It also keeps the code size of applications small since the routines from the classpath are not included in the executables, but are referenced from the classpath.

- Multithreading: Java supports multithreading at the language level. The system provides synchronization methods and the standard libraries are designed to be thread safe.

- Safety: Java systems perform safety and sanity checks on the classes while they are loaded. Memory leaks are largely avoided due to the garbage collection and the omission of pointer type protects the system from dangling pointers. The array indices are checked to be valid in runtime, making sure that no accesses outside of the actual array are allowed. The extra layer of software between user application and operating system also provides additional security.

- Availability: Java runtime environment is available for several target platforms at no cost. Also the compiler tools are free, and some more sophisticated development environments can also be found. There are several open-source projects focusing on Java virtual machine implementations, and they can form a good starting point if the intended target platform is not supported by the major commercial Java virtual machines.

2.4 Java Variants

Java systems are currently divided into three major categories, as depicted in Figure 2.1. The largest of these is the Java Enterprice Edition (J2EE) [104], which is targeted for large systems running server software in Java. The Java runtime system found in normal desktop computers is known as Java Standard Edition (J2SE) [105]. Te Java Micro Edition (J2ME) [106] is designed for embedded devices and other resource constrained environments. One more Java variant is the Java Card technology [107], which enables smart cards and other devices with very limited memory to run small Java applications.

The J2ME is further broken down by two criteria, based on the configuration and profile used for a given device. The configurations define the minimum subset of Java implemented by a device, while the profiles can add some functionality to a given configuration. A Java ME configuration only defines a minimum complement or the "lowest common denominator" of a given Java technology. All the features included in a configuration must

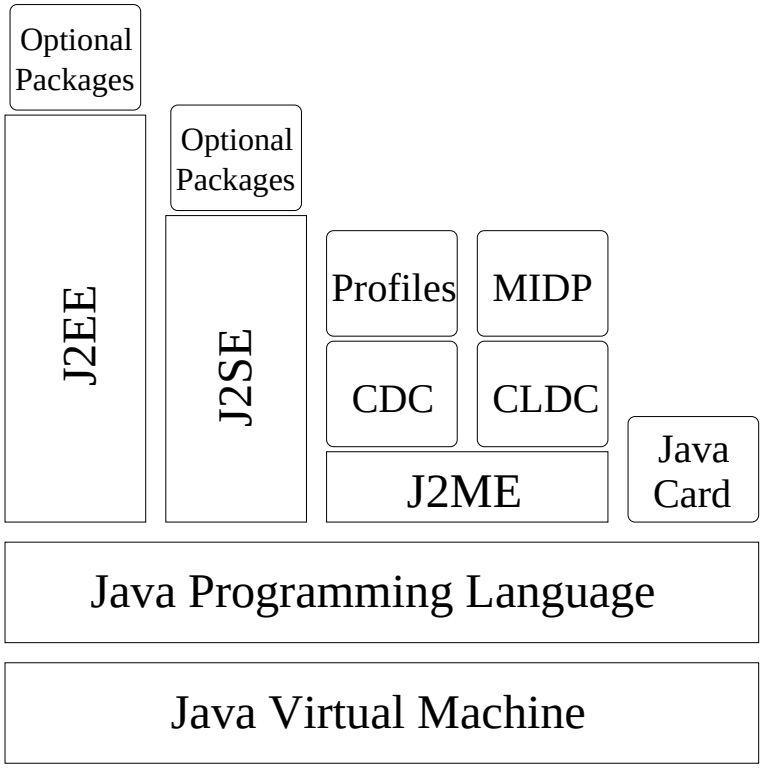


Figure 2.1: An overview of the variants for a Java virtual machine.

be generally applicable to a wide variety of devices. This means that the scope of a configuration is limited and often incomplete for any given real target devices. Additional features specific to a certain device, or category of devices, are defined in a profile specification.

2.4.1 Configurations and Profiles in J2ME

Currently there are two commonly used configurations. These are known as the Connected Device Configuration (CDC) and the Connected Limited Device Configuration (CLDC).

Connected Device Configuration is a framework for building a Java virtual machine suitable for 32-bit RISC/CISC/DSP microprocessors and a few megabytes of working memory. The applications on embedded devices range from pagers up to set-top boxes and high end PDAs. Three profiles are based on the CDC:

- Foundation Profile
  - Java SE-like application programming interface (API)
  - No graphical user interface (GUI)
- Personal Basis Profile
  - Extension to Foundation Profile
  - Lightweight GUI support
- Personal Profile
  - Extension to Personal Basis Profile
  - Full AWT (Abstract Window Toolkit) and applet support
  - Easy to port PersonalJava-based applications

This configuration is intended to be used by devices requiring a complete implementation of the Java virtual machine, and an API set that may, via the addition of profiles, include the entire J2SE API. Typical implementations use some subset of that API depending on the profiles supported. Two commonly used versions are 1.0b and 1.1.2. The CDC 1.0b API [67] is based primarily on the J2SE 1.3 API and it includes all of the Java language APIs defined in the Java ME CLDC 1.0 specification. The CDC 1.1.2 API [68] is based primarily on the J2SE 1.4.2 API and it includes all of the Java language APIs defined in the CLDC 1.1 specification.

The Connected Limited Device Configuration [69] defines a Java virtual machine suitable for 16-bit or 32-bit RISC/CISC microprocessors with a few hundred kilobytes of available memory. Typical applications include mobile phones and lower end PDAs. There are currently three versions available, numbered 1.0, 1.1 and 1.1.1. The first one is the most limited and it has no floating point support. The classes *float* and *double* are completely removed from the specification. This causes all of the bytecode instructions operating on those data types to be redundant, so they are also removed with supporting libraries. The CLDC version 1.1 adds the floating point data types and bytecode instructions as well as the supporting library methods in the classpath. The minimum memory budget is grown from 162kb to 190kb. This is mainly due to the added floating point support. Also some classes have been modified to be more compatible with J2SE, including Thread, Calendar, Date and TimeZone. The version 1.1.1 is only a maintenance release, and it adds only little to the 1.1 specification. The main difference is the addition of support for certain features of the Java technology security model, to provide a common base of Permission classes. Also some arithmetic functions have been added to java.lang.Math[2], such as sin, acos, atan and atan2.

[2]The capitalization is correct, however strange it may seem.

Only one profile is currently available for the CLDC. It is called "Mobile Information Device Profile" (MIDP). Currently, two versions of the MIDP are available, 1.0 and 2.0, with a third version in planning stage. The MIDP profile adds the following characteristics to the CLDC configuration: a display with minimum screen-size of 96x54 pixels (which should be approximately square), a keyboard or a touch screen, at least 256 kb of non-volatile memory for the MIDP implementation, 8 kb of non-volatile memory for application-created persistent data and 128 kilobytes of volatile memory for the Java runtime (e.g., the Java heap), networking with limited bandwidth and sound capabilities. This is the minimum set of requirements. Typically mobile phones with a Java virtual machine support this set and often even surpass the requirements.

2.4.2 History

The Java programming language was originally a part of a research project to develop software for network devices and embedded systems. In the beginning of the '90s, Javas predecessor, known as Oak, was created for a device called *7. The *7 was a small SPARC based device with a tiny operating system, much like an early version of a modern PDA. The *7 never made its way to a consumer device, but Oak, renamed to Java, was released in 1995. Java was then intended to be a new language for the Internet, and to be integrated into Netscape's browser. In 1997 Sun announced the PicoJava processor architecture. The PicoJava was supposed to be a standalone Java engine using dedicated hardware, but Sun never actually manufactured these processors. Since then Java has been expanding the application domain steadily, including areas like desktop applications, web servers and server applications. The diverse requirements and capabilities of these application domains caused the separation of the Java standard edition (J2SE) and the enterprise edition (J2EE) in 1999. Every new release brought new features to the standard library, thus increasing the size of the runtime system. The growth in the portable device market, ushered by the mobile phone industry, revived Sun's interest in the embedded systems. Sun defined different subsets of Java for embedded systems and collected them into Java micro edition (J2ME) in 2000. In order to broaden the application domain even further, the Real-Time Specification for Java (RTSJ) [6] was approved in 2002. Since the origins of Java lie in embedded systems and current trends have moved Java to that specific field, one might say that Java is coming home.

2.5 Generic Java Virtual Machine Architecture

The Java virtual machine is a definition of an abstract computing machine that executes bytecode programs. The JVM specification has three major components, namely:

1. Instruction set, called the bytecodes, and their functionality
2. Binary format, known as the class file, containing bytecode segments, symbol table and other ancillary data
3. Verification algorithm, used to verify that a class file contains valid programs

Several properties of the virtual machine are not specified, but rather suggested or left up to the implementors judgment. For instance the garbage collection is left to the implementor to choose. Similarly the execution engine is not specified, just the functionality of the instructions. All this freedom leaves the implementors with room for design decisions that suit the properties of the target device while still maintaining full compatibility with the JVM specification.

2.5.1 Lifespan of the Java Virtual Machine

The Java virtual machine is started when the user requests a Java application to be started. At this time the virtual machine is created, and it performs a boot sequence, much like typical desktop computers. During the boot sequence the virtual machine initializes it's data areas and sets system values, such as the current time and the time zone. After all these preparations are completed, the virtual machine loads and verifies the main class of the application requested by the user. Then the execution of the application starts. While executing, user interaction may be required. The execution of a Java application usually involves invoking more than just the main method. When a new method is invoked, it is loaded and verified before the execution resumes in the new method. Finally, when the application is completed, the virtual machine performs shutdown. This last phase releases all of the memory areas reserved during the lifespan of the virtual machine. Also open files are closed and, if the virtual machine uses them, locks on resources are released. Then the virtual machine sends an exit code to the underlying system, and is destroyed. A Message Sequence Chart (MSC) of the sequence is shown in Figure 2.2.

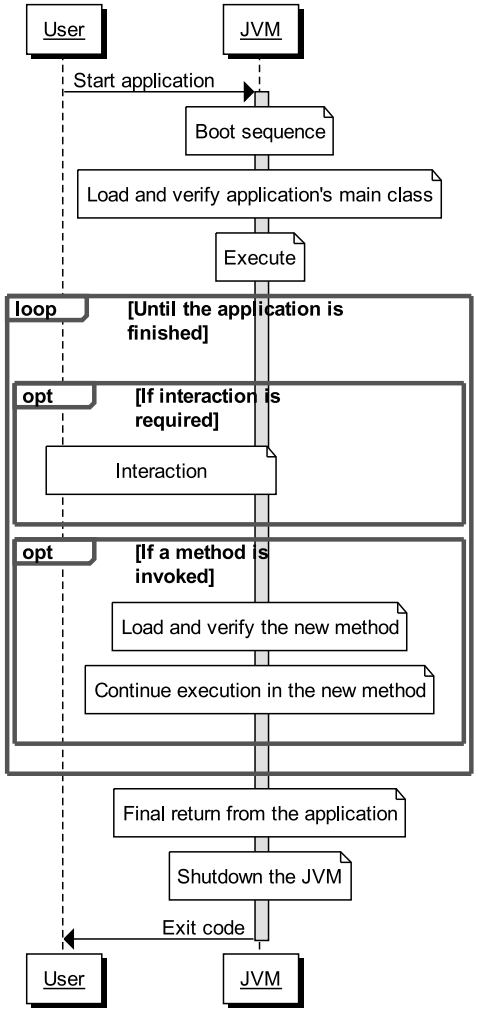


Figure 2.2: A MSC of the lifespan of a Java virtual machine.

2.5.2 Data Placement Inside a JVM

The data inside a JVM is typically placed in different areas depending on the type of the data. This is done in one part to make the structure clearer and in another part to ease the garbage collection process. When different areas are used for the various data types present, only the heap area needs to be collected. The following classification is a typical implementation, several aspects can be varied at the implementors discretion. Figure 2.3 shows the data areas and the software components of a typical implementation of the Java virtual machine.

Class data is an area dedicated for storing class specific information, such as the constant pool data. It also contains the field data for the class. This area is shared between all the threads running in the JVM.

Method area is also shared among the threads. It contains the bytecode implementations of the methods loaded to the JVM. This area provides the instruction streams during the execution of a Java application.

Native method area is similar to the method area, but instead of storing bytecode segments it stores platform dependent native code segments. Like the method area, this area is also shared between the threads.

Java heap is the last shared memory area, and it is used for storing the objects created during the runtime of the Java application. This includes all of the arrays and the other objects. The garbage collector clears any unnecessary objects from this area.

Java stack is private to each thread running on the JVM. In essence this means that every thread has a separate stack instance. A Java stack contains stack frame information, actual operand stack and the local variable area of the currently executing method. These items can be placed in any order. The specification does not dictate anything regarding the layout of the data. The operand stack is accessed very frequently during execution, so it should be fast.

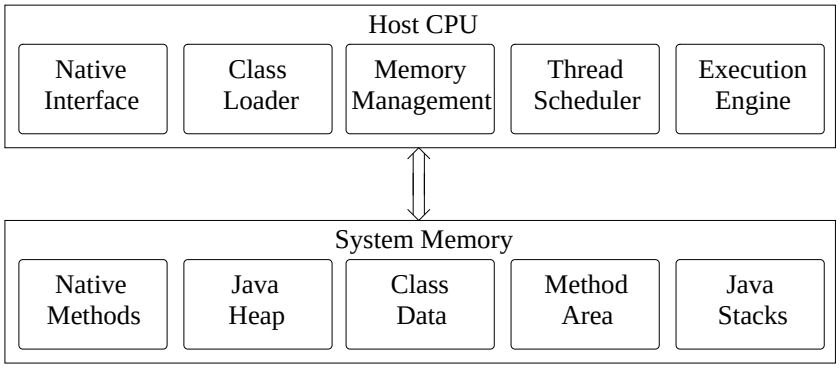


Figure 2.3: An overview of the data areas and the software components in a Java virtual machine.

2.5.3 Bytecode Instruction Set

The bytecode instruction set contains 201 instructions. All of the Java variants use the same instruction set, save for the smallest ones, which omit the floating point instructions. The bytecode instructions can be grouped into eight categories. Most virtual machine implementations also provide fast versions of some of the instructions. These are typically instructions that might cause a class to be loaded. When the class in question has already been loaded, the instruction will be replaced with the fast version, which does not check whether the class needs to be loaded. These are not part of the standard Java bytecode instruction set. Typically the fast versions are assigned bytecodes above 202. The bytecode instruction set is listed in the Appendix A, followed by the extended instructions used in the REALJava virtual machine.

The load and store category contains instructions that transfer data from local variables to the stack (load) or in the other direction (store). This category contains all together 70 instruction, but many of them have the same implementation. This is caused by the fact that the instructions contain type information of the data being transferred. For instance the *iload* and the *fload* both transfer a 32-bit data item, and the type of the data item has no significance other than making verification possible. The large number of instructions in this group is an indication of their frequency in Java applications.

The arithmetic-logical category contains data manipulation instructions. These operate on the data found in the Java stack. Usual cycle is to pop the top two elements of the stack, perform an operation on them and then store the result back to the top of the stack. These instructions are also typed, and here the type information is actually meaningful. For instance the addition of two integers, *iadd*, and the addition of two floats, *fadd*, are completely different in implementation point of view. Even though the JVM supports types shorter than 32 bits (byte, short and char), there are no arithmetic-logical instructions for them. Instead they are calculated using operations for integers and then the result must be explicitly converted to the correct type. The Java compiler includes these conversions automatically, and the programmers do not need to consider it [3]. Oddly, the comparisons that take floating point data types or long integers as inputs are part of this category. Those instructions pop the values to be compared, and then push an integer whose value tells the result of the comparison.

[3] Of course the additional conversion has an effect on the performance, but the functionality will be correct.

The type conversion category provides means for converting data items to different types. When converting integers to shorter types, such as bytes, the conversion is done simply by dropping the high-order bits and then sign extending the result back to the 32 bit data type, which is used internally. This can cause the sign of the result to be incorrect. Converting longs to integers is done in the same fashion. Also the conversions to and from floating point types exist.

The object and array category includes instructions used for creating and accessing objects and arrays. The data access instructions for objects are not typed, but the array access instructions contain the type information. The instructions used for finding or checking the class of an object belong to category also.

The stack manipulation category contains instructions that remove, copy or move data items in the top of the stack. These instructions are not type specific, but separate versions for 32-bit and 64-bit data items are included.

The branch category has the instructions for both conditional and unconditional branches. The branching always happens inside the current method and remains in the current stack frame. The conditional branches compare either the top element of the stack to zero or the two top elements to each other. The conditional branches use integer data types in the comparisons, for comparisons with floating point types or long integers the testing needs to be done separately. Object references can also be tested to find out whether the reference is *null* or not.

The method category provides instructions used for invoking methods and returning from methods. There are four different invocation instruction, which differ mainly in the way the actual target method is found. The return instructions can pass a return value to the calling method, and the data can be 32-bit or 64-bit. If a single return value is not sufficient for a given method, the return data must be placed in an object, and the reference to the object can be returned. The instructions in this category change the stack frame, either to a new frame in case of an invocation or to a previously used frame in case of a return.

Finally, **the miscellaneous category** is used for the rest of the instructions. These include instructions operating on monitors, throwing an exception and index widening instructions. The monitors are used for synchronization between threads.

In order to analyze the instruction set, the Kaffe [97] virtual machine was modified[4]. The modified version produced an output file of the executed instructions. This file was then analyzed to find the dynamic frequencies of the bytecode instructions during the execution of selected Java applications. The analysis showed that the load and store category is the most common in real applications, amounting around 50% of all instructions on average. The arithmetic and object groups are quite neck to neck for the second place with around 16% each. The branch and method categories take roughly 8% each, while the rest total up to 2% when counting all together. Similar results have been measured by others, including [51].

The code segment of a method consists of the instruction bytecodes, and some instructions have parameter data attached. This parameter data is placed directly after the related instruction, effectively becoming part of the instruction stream. As an example, the *iload* instruction is followed by one byte index to the local variable array of the current method. All instructions do not have parameter data attached, and the length of the parameter data varies. The amount of parameter data for each instruction can be found in Appendix A.

2.5.4 Java Methods

The methods in Java are roughly equivalent to functions or procedures in other languages. Terminologically methods are invoked whereas functions and procedures are called. There is no difference between invoking and calling, save for the naming. In the Java bytecode the methods are invoked using one of the following instructions:

- *Invokestatic*, a class method [5] is invoked. The method being invoked does not depend on the type of the object. Finding the actual method to be invoked is a simple constant pool lookup.

- *Invokevirtual*, invokes a virtual method, based on the type of the object on the top of the stack. This requires virtual method resolution in order to find the correct method for a given object.

- *Invokeinterface*, an interface method is invoked. The interfaces provide means for multiple inheritance in Java. Finding the actual method can

[4]The Kaffe was used, because at this time the REALJava virtual machine was still in the planning stage. Later the facilities required for similar analysis were included to the REALJava.

[5]A method that is declared *static* in the source code

be a complex procedure, if several interfaces are implemented in the application.

- *Invokespecial*, is used on special cases. These include methods in the superclass of `this`[6], private methods of `this` and the instance initialization method, `<init>`.

Java applications tend to perform a lot of method invocations. This is because of the structure of the classpath, at least the GNU implementation [92] of it, and the fact that many programming courses teaching Java programming promote the use of small methods. The small methods are considered to be good programming practice and also help to make the code more reusable. Object oriented programming style often involves several small routines used to access data in the object fields. These facts make it very important to have an efficient method invocation subsystem.

2.5.5 Classes, Class Files and Class Loading

An application program written in Java language is first compiled into the Java bytecode. During compilation all the methods used in the program are compiled separately, and stored in class files. This does not include the methods imported from the standard library. They are just referenced, and the virtual machine links them in during the execution. A single class can contain, and they usually do, several methods. The class files contain also other data, most importantly the constant pool. The constant pool is used for storing constant values and strings as well as for storing method names.

During the initialization a virtual machine first sets some internal properties, usually related to the underlying operating system, current time and so on. After this the virtual machine loads the starting method, usually the *main()*, of the user application.

The Class File Format

The classes that make up an application are delivered in the class file format, described in the Java Virtual Machine Specification [38] and shown in Figure 2.4. The class files are similar to object files generated by normal compilers (for example GCC). A class file contains several critical pieces of

[6] Java uses "this" to refer to the object being operated on.

data, such as the bytecode segments that make up the methods in the class, a reference to the superclass of this class, a list of the fields defined by the class, the constant pool of this class and other data required by the virtual machine.

All valid class files start with a magic number 0xCAFEBABE. This is followed by version information of the class. Then the tables containing the constant pool, methods and other data items are defined. The last table is for the attributes of the class, and it can contain arbitrary definitions. If the virtual machine running the application does not understand some of the defined attributes, they are simply ignored.

The constant pool of a class is used to store a mixture of data items. These include names of the methods and the fields, the numerical constant values and the string constants belonging to the class. The method invocations in Java use the class and method names stored in the constant pool to find the required method. Since the constant pool is used frequently, it is important that the virtual machine can access it efficiently.

The class files are often packaged into "Java archives" (JAR). As a deployment form, the JAR file format provides many benefits over distributing the class files separately. Most notable benefits include:

- **Security:** The contents of a JAR file can be digitally signed. If a virtual machine recognizes the signature, it can then optionally grant the application security privileges it would not otherwise have.
- **Compression:** The JAR format allows the compression of the class files for efficient storage.
- **Decreased download time:** If an application is bundled in a JAR file, the class files and associated resources can be downloaded in a single transaction without the need for opening a new connection for each file. Naturally, the compression also reduces the download time.
- **Package sealing:** Packages stored in JAR files can be optionally sealed so that the package can enforce version consistency. Sealing a package within a JAR file means that all classes defined in that package must be found in the same JAR file.
- **Package versioning:** A JAR file can hold data about the files it contains, such as vendor and version information.

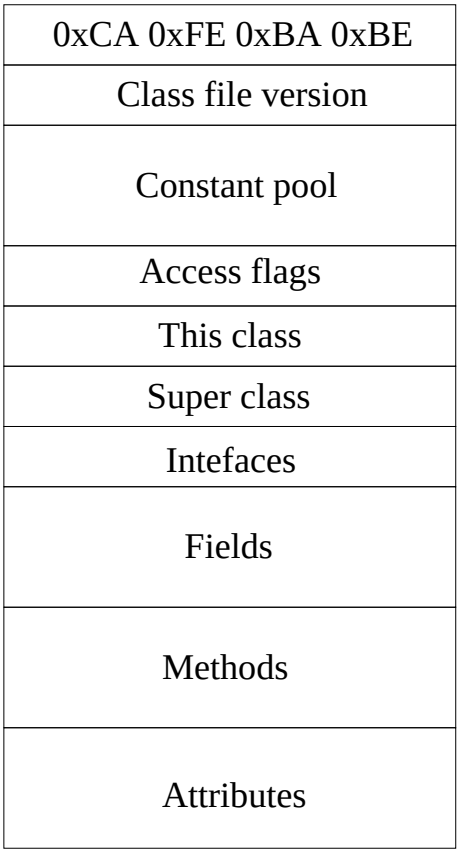


Figure 2.4: The structure of a Java class file.

Besides the actual class files, the JAR file contains metadata defining the main method of the application. The JAR file may also contain other pieces of data, such as an icon for the application, audio and images. The support for the zip packing is optional in the specification, but most virtual machine implementations include it.

Class Loading (and Unloading)

When a virtual machine starts to execute an application it first loads the main class of the application. Loading is the process of retrieving the data that defines a class. Classes can be loaded from any source, including (but not limited to) ROMs, disks and network connections. All further classes are loaded when they are needed.

The specification defines two mechanisms for class loading. The more important one is system class loader, which is a built-in class loader used for the standard classes in the classpath and also for the classes of the user application. The other mechanism uses a user defined instance of the `ClassLoader`. The latter is sometimes disabled for security reasons, since it would enable malevolent programmers to load and execute arbitrary Java code via web browsers etc.

After a class has been loaded it is not ready to be executed. It requires further processing steps. The next step is linking, which may cause new classes to be loaded recursively in order to find all the necessary superclasses. This step also performs verification of the previously loaded class. Some virtual machines omit the verification completely while others verify only some of the classes. A virtual machine may well believe that its own standard classes are safe and thus they are not necessarily verified.

After the verification the class is prepared. During preparation the static fields are given their initial values. This is followed by initialization, when the virtual machine first checks that all of the required superclasses have been initialized. Then the static initialization methods defined in the class are invoked. At this point the methods contained in the class are ready to be used.

The specification allows also the removal of previously loaded classes. This is done if the virtual machine wants to perform garbage collection on the loaded classes. The functionality is not defined in the specification, but if a finalizer method exists in the class, then it will be invoked before unloading of the class. This functionality is not present in all of the Java virtual machine implementations.

2.5.6 Exception Handling

In Java exceptions are used to signal errors that occur during the execution of an application. Like C++, Java uses the try/catch model for exceptions. If an exception is thrown within a "try-catch" block, execution branches abruptly from the current instruction to the corresponding catch clause. The catch clause is used to create an exception handler. If there is no matching handler, the current method terminates abruptly, and the exception is thrown in the calling method. Similarly, if the calling method does not define a matching handler, the exception propagates up the call stack until either a matching handler is found or there are no more stack frames. If no handler is found, the exception is considered to be unhandled. Unhan-

dled exceptions terminate the current thread. An exception can be raised directly by executing a throw statement or indirectly as a side effect of executing a bytecode instruction or runtime operation. For instance, any array load instruction can throw an exception if the array index is out of bounds. Similarly, attempts to dereference a null pointer result in a null pointer exception. It is worth reminding, that Java does not provide arithmetic exceptions for overflows or underflows. Arithmetic exceptions are available only for division by zero, and even those are limited to integer types. There are no arithmetic exceptions for floating point data types.

2.6 Java Virtual Machine Implementations

A typical Java runtime environment for embedded systems contains the following components:

1. A Java Virtual Machine to translate Java's platform-independent bytecodes into the native machine code of the target processor and to perform dynamic class loading. This usually takes the form of either a simple interpreter or a just-in-time compiler (JIT). The execution strategy is left entirely up to the implementor.

2. A standard set of Java class libraries, in the bytecode form. If the used applications do not reference any of these classes, then they are not strictly required. However, most Java runtime environments are designed to conform to one of Sun's standard API's. Several non-commercial Java virtual machine projects use the GNU Classpath as the standard library because it is open source. The GNU Classpath is not quite complete, but most of the methods required in the embedded systems domain are present.

3. Any native methods required by the class libraries or virtual machine. These are functions that are written in some other language, precompiled, and linked with the Java virtual machine. They are primarily required to perform functions that are either processor-specific or unable to be implemented directly in Java.

4. A multithreading support system to provide the internal implementation of Java's threading and thread synchronization mechanisms. Often the operating system's native threading mechanisms are used, but if the underlying operating system does not support threads, then the virtual machine must implement a thread handler of its own.

5. Garbage collection. The garbage collector runs periodically, or whenever the pool of dynamic memory is unable to satisfy an allocation request, to reclaim memory that has been allocated but is no longer being used by the application. The collection can also be started by an user application, when the application invokes the System.gc method.

The components of such a system are shown in Figure 2.5. The Figure shows the logical hierarchy of a Java runtime system, from the Java application all the way down to the operating system level.

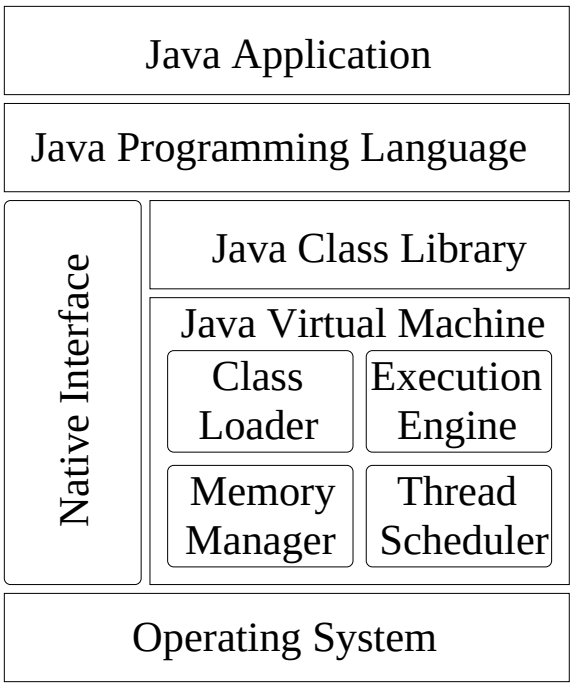


Figure 2.5: An overview of a Java runtime system.

The traditional way to implement the execution engine of a Java Virtual Machine is to use a software *interpreter*. This approach takes the precompiled Java bytecode methods and executes them by interpreting one instruction at a time. The execution speed of this way is rather poor, partly due to stack emulation and partly because each bytecode instruction is processed using an indirect jump from the main loop to the implementation of a given instruction with the current instruction as a key. The most positive feature of an interpreter is the portability. An interpreter doe not require any assembler optimization, and it can be implemented using only portable structures.

Direct threading is an improvement over the basic interpreter. It was first used for accelerating Forth execution [14], but has also been applied to Java execution. For instance the SableVM [96] uses this as one of the optional execution engines. Direct threading reduces the speed penalty caused by the instruction dispatch by preprocessing the bytecode during class loading. The preprocessing replaces the instruction opcodes with the addresses of the interpreter's case statements implementing the instructions. The execution is then transformed to simple jumps directly to the corresponding native implementations. The preprocessing is not very complex, so direct threading can be useful on fairly simple devices. The most notable drawback is that since all jumps in Java are relative, all of the offsets for the jumps must be recalculated. Direct threading is also fairly portable, as it requires no assembler optimization. However the addresses are retrieved one at a time, just like the opcodes in an interpreter, causing the fetching to use a lot of time.

In order to reduce time used for instruction fetching, it is necessary to execute multiple instructions with a single fetch. Certain segments of bytecode instructions can be replaced by joining the blocks of native code that implement the instructions in the segment. This replacement process is called *inlining* [43]. Like direct threading, inline threading can be implemented mostly using portable structures. However, during the implementation of the inline threaded execution engine for the SableVM, it has been found that the only practical way to find out which instructions are inlinable is to test each instruction carefully on the exact target architecture, severely cramping portability. Although the basic idea of instruction inlining is simple, implementing it in practice is much more complicated than implementing direct threading. For instance besides recalculating the jump offsets, the inlining engine must ensure that none of the original jump targets are inside a segment to be inlined. Replacing instructions with the fast versions mentioned earlier after class initialization also becomes more difficult with inline threading, because many sequences of instructions can be inlined only after the instruction replacement.

Recent developments in the area of execution engines have led to use of *just in time compilation* (JIT) [11], which means that the software executing bytecode takes pieces of the code and compiles them to the underlying hardware's native instruction set. The most advanced systems using this approach do not recompile or optimize everything, but focus more attention on code segments that are used often (loops etc.). These segments are often called hotspots. Virtual machines using JIT are not easily portable to new systems, since they typically require heavy assembler optimization. In [49] Smith et al. stated that retargeting a hotspot aware JIT based Java virtual

machine to Irix/MIPS platform took about 5 man-years of engineering effort. Comparing that to the mere minutes taken for an interpreter written in standard C/C++, it is clear that the performance obtained from JIT comes with added problems in the portability. JIT is not suitable for resource limited environments, as the recompiled code segments require extra memory space at run time, and the dynamic compiler required for the JIT takes both memory space and permanent storage space. Another downside of JIT is the unpredictability of execution time, one (usually) cannot know in advance if a given code segment is already compiled or not. Also resource limited systems might purge old precompiled segments out of the memory to make room for new segments. In [42] Nicolaescu and Veidenbaum studied the behavior of JIT compilers and interpreters running on modern high-end superscalar CPU with Out-Of-Order execution. They found out that both the data and instruction caches generate more misses when JIT is used. Also the branch prediction misses more often, when compared to interpreting. They concluded the study with an interesting remark: "One interesting implication of these results is that improvements from JIT compilation are likely to be noticeably lower in embedded systems..." Their reasoning is based on the poor cache coherence and branch prediction, and also on the fact that they discovered that a large share of the performance gained with JIT is due to the processors ability to increase instruction level parallelism (ILP). The features needed for ILP are not currently available in the embedded domain. Similar results have been reported by Radhakrishnan et al. in [45, 46]. Figure 2.6 shows possible paths for developing systems based on the Java language and also using other input languages with the systems. Both interpreting and JIT compiling run on the highest path in the figure. The other paths will be discussed next.

Java bytecode can also be transformed to some other virtual machine architecture (XVM) using a transcoder. This transcoding can be done at execution time or during software download. The execution time transcoding moves along the highest path in Figure 2.6 breaking down at the last fork, whereas the download time option breaks down at the previous fork. This approach also allows other languages to be compiled for the same XVM, even all the way from the source code. However there are drawbacks, such as increased storage space requirements to store the original code and the XVM code, longer download delays for non-local software that needs to be transcoded and successive optimizations performed by compilers with different target architectures producing inefficient code. This inefficiency is clear in case when the first optimization has been performed targeting a stack based architecture, such as Java, and the second round targets register based architecture, which would include practically all major processor implementations currently available. Another drawback, when storing the

applications in transcoded form, is that the application cannot be directly transferred to a different device.

The REALJava virtual machine discussed in this thesis is of the first type. It is 100% Java bytecode compatible, but it transfers the execution away from the CPU to a co-processor designed to execute simple Java bytecode instructions. The CPU still maintains control of all system specific operations (such as filesystem, network, I/O), complex instruction (class loading and verifying) and memory management (especially garbage collection). This partitioning provides easy integration to multiple systems, since the underlying host architecture is irrelevant to the co-processor. The user executing a Java application does not need know that the execution is performed on a co-processor, since that happens inside the virtual machine. The use of a co-processor can be seen as implementing the highlighted part of JVM in Figure 2.6 using dedicated hardware.

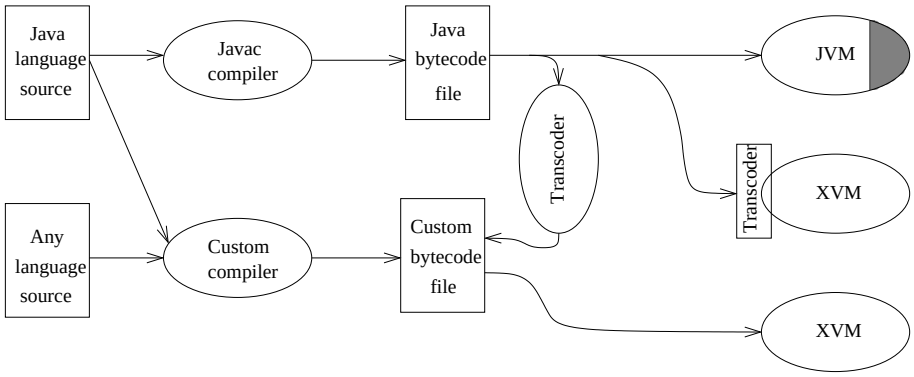


Figure 2.6: Possibilities to develop the Java virtual machine.

2.6.1 Using Hardware Systems in Virtual Machine Implementations

A stand-alone solution implements the whole virtual machine in hardware, and thus needs no CPU. As mentioned in Section 1.4, examples of this approach include Sun's PicoJava, JOP and aJile. With this strategy the complex instructions (for instance instructions requiring class loading) and garbage collection are hard to implement, and the resulting virtual machine is not easily integrated to an existing system. Out of these three example systems only PicoJava implements the full J2SE (standard edition of Java) [100]. The other two implement J2ME (micro edition for small systems)

[101] and for instance the garbage collection is completely left out from JOP, due to real time performance goals set for the system.

Using a co-processor for the execution of Java bytecode provides easier integration to existing systems, as platform dependent features, such as the GSM communication for mobile phones and I/O devices, can be coded in easily portable software for the CPU. Also the physical size of the processor core is reduced, as the complex functionality is performed in the CPU, leaving the co-processor to deal with the instructions that are suitable for direct hardware implementation. This approach has been used with the inSilicon JVXtreme, the CCL Java co-processor and the REALJava virtual machine. The Jazelle [103] co-processor by ARM is also an example of this approach, even though it is more tightly coupled to the CPU than most Java execution co-processors.

Co-processors typically use either autonomous or parallel execution model. The parallel model is similar to Intel x87 floating point unit architecture, where each instruction is routed to both the CPU and the co-processor and the unit whose instruction set a given instruction falls into executes it. On the other hand in autonomous execution model the CPU just configures the co-processor and lets it handle execution on its own. During this time the CPU is free for other tasks. The REALJava co-processor uses a semi-autonomous model. This model is similar to the autonomous model. The difference is that in the semiautonomous model the co-processor can ask for CPU support on the instructions it can not handle. It should be noted, that the semiautonomous model allows using several co-processors in parallel, unlike the other hardware schemes presented here. Using several co-processors in a JVM is well justified, as Java supports multithreading at language level.

Solutions using a hardware interpreter have also been used. These are exclusively targeted to a certain CPU, as the interpreter translates the bytecode instructions to the CPU's native instruction set. The Jiffy [1] mentioned earlier in Chapter 1 is a very interesting example of this approach. The Jiffy performs JIT on the FPGA to optimize the resulting native code.

Some studies, like [9, 10, 35], have also focused on mitigating the inefficiencies in Java execution by moving part of the workload to a remote server. In this approach the user device is expected to have a permanent connection to the server, usually a wireless connections are assumed. For instance, the research by Chen et al. [9] focused on selectively transferring some of the JIT compilation and method execution to a remote server. The requirement of a continuous connection to the server naturally increases the power consumption by forcing the radio subsystem to remain active all of

the time. The time used for the transfers was not considered in that study, only the impact on power consumption.

2.7 Chapter Summary

An overview of the Java technology was given in order to provide required background information and context for the design of the REALJava virtual machine. The topics covered in this chapter included the Java programming language, the Java virtual machine specification and generic Java virtual machine architecture. The variants of Java were described, with a glance at the history of Java. Implementations were shortly discussed, with focus on the hardware assisted systems currently available.

Chapter 3

Conceptual Model of the REALJava Virtual Machine

This chapter details the conceptual model of the REALJava virtual machine. The chapter begins with a description of the execution model and moves on to explain the partitioning between the hardware and the software. Then the structure of the co-processor is detailed. The software partition is also shortly discussed in order to give context for the hardware and provide a fully functional virtual machine implementation. Finally the logical data structures inside the virtual machine are presented.

3.1 Execution Model

The execution model used for the REALJava virtual machine is "semiautonomous". This means that the co-processor has full control over the execution once the control is transferred there, but the co-processor can not initiate processing by itself. In the REALJava execution model the Java bytecode instruction stream is first loaded to the co-processor, which is subsequently commanded to start execution of the stream. At this point the CPU can do something else, if required, or simply go to idle mode until the co-processor sends an interrupt request (IRQ) to notify the CPU that some assistance is required. This execution model frees the CPU to other tasks in the system and also makes it possible to use several co-processors, since the instruction streams in each co-processor do not interact in any way. A MSC of the model is shown in Figure 3.1.

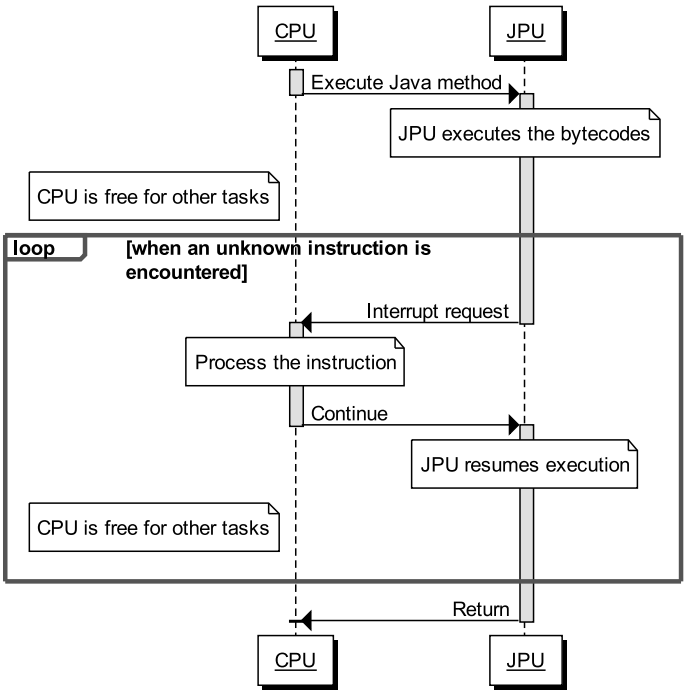


Figure 3.1: A MSC of the execution model for the REALJava. The highlighting in the vertical bar shows the unit responsible for the execution at a given time.

3.1.1 Other Execution Models for Co-Processors

The execution model is different than the models used for floating point unit in the Intel x86 architecture and the model used for decoding media streams. The original Intel x86 floating point unit strategy is quite simple, both the CPU and the co-processor receive all instructions. If the instruction falls in the co-processors instruction set the CPU performs NOPs while the co-processor executes the instruction. If the instruction falls in the CPUs instruction set the co-processor executes NOPs until the next floating point instruction arrives. The other often seen execution model is used for streaming media and other applications with similar dataflow. In this model the co-processor does not receive instruction per se, but rather it is first configured to suitable settings and the CPU sends a data stream to the co-processor. The co-processor performs some function on the stream, for instance decoding a mp3 music stream, and possibly sends the resulting stream straight to the output device. Also the 3D engines in modern graphics cards use this approach, even though they also receive individual commands to be executed. An example of an individual command could be:

draw a line from one point to another. Still, most of the 3D rendering is performed independently.

3.2 Partitioning

The tasks of a virtual machine are divided between the hardware and software based on their complexity and need for access to the various resources of the system. As presented in [56], all of the most complex tasks are assigned to the software. The selection is partly motivated by the fact that the software is easier to modify to fit to the specific details of the underlying host platform. These complex tasks include memory management (not the stack) , I/O access, class loading and of course native method interface. The hardware is assigned the execution engine and stack management. Some of the instructions are not executed in the co-processor, since they require access to the heap memory or class loading. Also instructions operating on data belonging to the double or long types are handled by the software. The 64-bit data types are omitted from the co-processor as the intended application domain of embedded systems is unlikely to make extensive use of them. The support for them is provided via software, so as to stay compatible with the Java specification. Large amounts of arithmetics using the 64-bit data types slows the resulting system down, but the Java applications run correctly and the full J2SE specification is met.

Some of the instructions can be dynamically reprogrammed. This approach is often used in software implementations of the Java virtual machine. For instance the class loading instructions can be reprogrammed to fast versions after the required class is loaded. The subsequent executions of the reprogrammed instruction do not need to load the class again. Besides speeding up the execution of the reprogrammed instructions, this strategy makes it possible to move some functionality to the hardware domain after the prohibitive parts have already been performed. This effectively moves the original instruction from the software partition to the hardware.

The power consumption of a Java virtual machine has been studied from several perspectives. In [33, 34] Lafond and Lilius showed that during the execution of a Java application the memory accesses dominate the power consumption. The power consumption has also been studied using pocket computer. The findings in [15] also indicate that the memory subsystem is responsible for a large share of the energy consumed during the execution of a given Java application. The REALJava approach eliminates a large share of the memory accesses by placing the stack locally inside the co-processor.

The amount of accesses is also reduced because the execution engines running on software need to maintain the internal registers as variables residing in the memory. At minimum any virtual machine needs registers for storing the current program counter and the stack top. In the REALJava virtual machine these are maintained in the hardware, thus eliminating unnecessary memory accesses as well as the extra clock cycles needed to update the registers.

3.3 Structure of the Co-Processor

The architecture of the REALJava is shown in Figure 3.2. The figure shows also the two domains, the host system and the co-processor. The local memory, which is used for the stack and the method area, is marked inside the co-processor with a dotted line because it does not have to reside physically inside the co-processor. External memory can be used for the local memory, but accessing it would naturally be slower and consume more energy. The structure of the execution pipeline of the Java co-processor differs from the structure normally used for general purpose processors, as presented in [53]. This is due to the fact, that normally the instruction set of a processor is engineered with hardware implementation in mind, but this is not the case for Java. The Java bytecode has been designed to be executed in software, resulting in several significant differences. Additionally the bytecode instructions are based on a stack, instead of the normal RISC approach of using several general purpose data registers. This calls for optimizations not seen in modern general purpose processor design.

3.3.1 General Purpose Processor Pipeline Architecture

The text book strategy for pipelining a general purpose processor involves 5 stages, namely:

1. instruction fetch
2. instruction decode / register access
3. execute / ALU
4. memory access
5. write back

This approach has been used in several processors and is also presented in several text books, such as the DLX processor presented in [23]. This

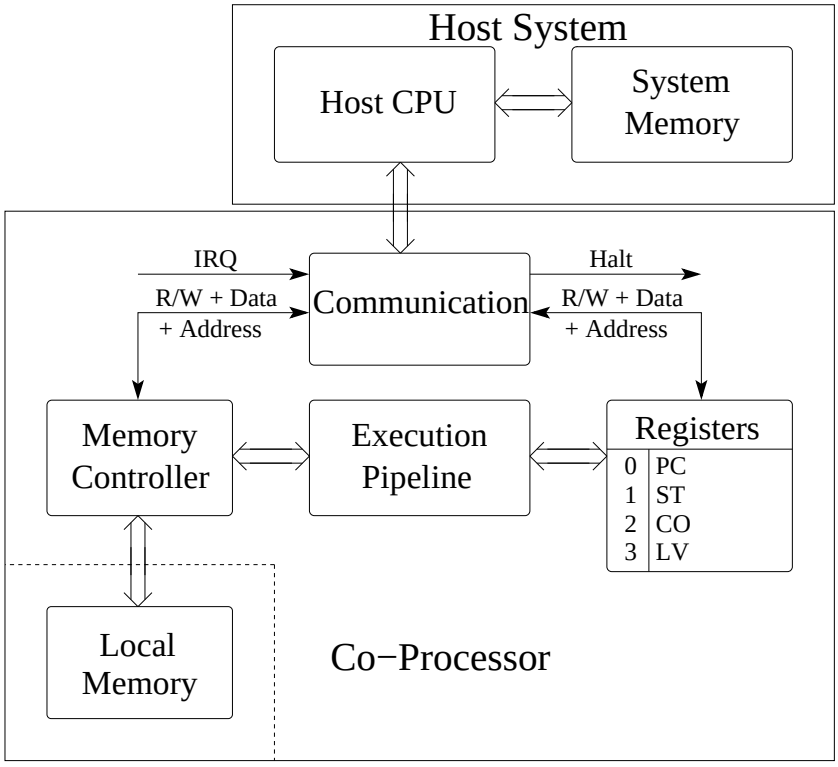


Figure 3.2: The architecture of the REALJava virtual machine.

strategy is based on the assumption that the processor has internal registers for temporary or working data storage. Usually these registers can be accessed in parallel, and there are several registers available. As an example the DLX processor has 32 32-bit general-purpose registers. Some processors also include separate registers for storing floating point numbers. The DLX processor provides 32 32-bit floating point registers, which can be used as even-odd pairs to hold 16 64-bit double-precision values. Several register access optimization strategies have been developed, including operand forwarding and splitting register accesses to writes in the first half of the clock cycle and reads in the second half. These strategies are not directly applicable to a Java virtual machine, since there are no internal data registers.

3.3.2 The Modified Architecture for the Java Co-Processor

The Java Virtual Machine Specification [38] states that the JVM has no internal data registers, instead the temporary and working data is stored in a stack. Normally the software coder can improve performance by re-

ordering the register accesses to keep the pipeline flowing, but in Java this is not possible, since all instructions manipulate data which is located at the top of the stack. This situation is somewhat comparable with a normal processor architecture with only one register available to the programmer, or the old accumulator architecture. This would keep the pipeline stalled for a large portion of the time, because of unavoidable data dependency issues. To keep the pipeline in effective use, the normal pipelining strategy has been modified to better suit the stack based operation.

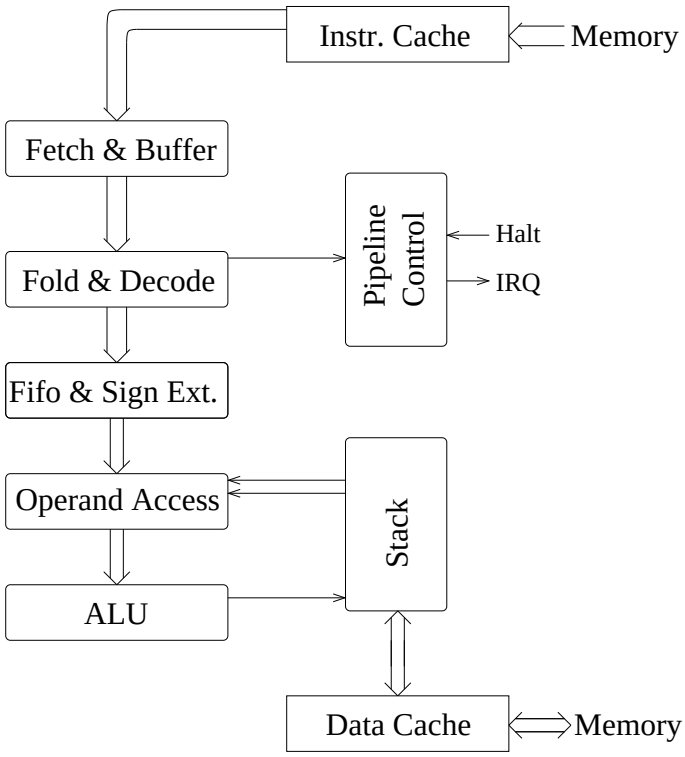


Figure 3.3: A structural view of the pipeline. The handshake signals are omitted for clarity.

As shown in Figure 3.3, the modified architecture begins with instruction fetching. A fifo is placed inside this unit to provide the folding unit with fast access to the instruction stream. The instruction decoder is the next unit. A technique called instruction folding, which will be explained in more detail in Section 3.3.7, is used to reduce unnecessary stack accesses. The folding is included in the decoder stage. After that is an intermediate buffer level to store the folded instructions before execution. This buffer also performs minor operations, such as extending data items to 32 bits.

The next stage performs operand fetching, if necessary. Then comes the ALU, which contains the write back stage. The write back stage is included into the ALU because the bytecode instructions are based on the stack. One might wonder what this has to do with selecting the pipeline stages, but the answer is rather simple. In Java bytecode the instructions take the operands from the stack and write the result back to the stack. This would cause the "traditional" pipeline structure to generate excessive stalls to move the data to and from the stack. Thus the execution in the ALU would be often halted while the data is moved back and forth. Actually, two other methods to alleviate this problem will be presented later in Sections 3.3.5 and 3.3.7.

3.3.3 Shared Resources

Several pipeline stages need to access shared resources. These include the stack, the control registers and the program counter. Access to these resources is controlled by similar handshakes as the data flow through the pipeline. The main difference is, that since several units need to access these resources, they must provide mechanisms to prevent simultaneous accesses and to guarantee the correct ordering of events.

The pipeline control unit can also be seen as a shared resource, as it is connected to the pipeline stages. The pipeline control unit sends a halt command to all pipeline stages upon receiving an external halt command or a halt request from the fold and decode unit. The fold and decode unit is required to have halt access to facilitate pipeline halting when a software handled instruction is encountered. After the whole pipeline is idle, the pipeline control sends an IRQ to the host processor. Note, that these two methods of halting differ in their reaction speed. If the halt is requested by the CPU, it is performed as soon as possible. When the halt is caused by a trapped instruction, the halt is performed when all previous instructions have been fully processed.

The last shared resource is the local memory. The local memory is used to house the stack and local variables in the data side and bytecode segments of the methods to executed on the instruction side. This local memory is local logically, which means that it can be implemented as an external memory region assigned to the Java processing unit (JPU) or as a real, physically local memory placed inside the JPU module. In case of a physically local memory the caches can be small or even omitted. Our tests have shown that relatively small local memory space is required. According to [48] 98.75% of static methods in the runtime library are under 512 bytes in length, and

our own studies have shown that the stack frame for one method rarely exceeds 10 words (40 bytes), which totals to about 1 kilobyte, including the local variables. Naturally larger is better, as invoking a method or returning from a method back to the calling one is much faster if the memory is large enough to contain the stack frames and code segments for several methods at the same time.

3.3.4 Instruction Preprocessing

This block starts after the instruction cache. The cache handles all communication with the local memory, regardless of whether the memory is physically local or external. This partitioning of responsibilities allows using different memory technologies without modifications to the instruction fetching unit. The physical addresses to the local memory are generated at the instruction fetching buffer, using the program counter (PC) and code offset (CO) registers. The code offset register holds the starting address of the current code segment in the local memory. The actual address of the instruction to be executed is obtained by simply adding the code offset and the program counter together. After the fetching, the instructions are checked for folding in the fold and decode stage. The details of the instruction folding process are described later in Section 3.3.7.

As shown in Figure 3.4, the pipeline control unit is connected to the folding and decoding unit with two way communication. The folding unit needs to request a halt when it encounters an instruction to be handled in software. Of course the pipeline control unit must be able to stop the processing in this segment, so there needs to be bidirectional channel. The control unit also connects to all other pipeline stages, with a halt signal. The CPU can also request a halt, for thread switching or setting new values to internal registers.

The folding and decode unit has two communication channels to the instruction buffer, one for actual instructions and one for literal data. After an instruction has been decoded and folded, the VLIW (very long instruction word) is sent to the fifo in the main pipeline. The instruction folding unit is covered in more detail in Section 3.3.7. The fifo is only a few levels long and provides timing margin for folding and performs sign extension. The sign extension is used for extending data items that are shorter than 32 bits. These include constants loaded with *bipush* and *sipush* instructions and indexes used in local variable manipulation instructions like *iload*.

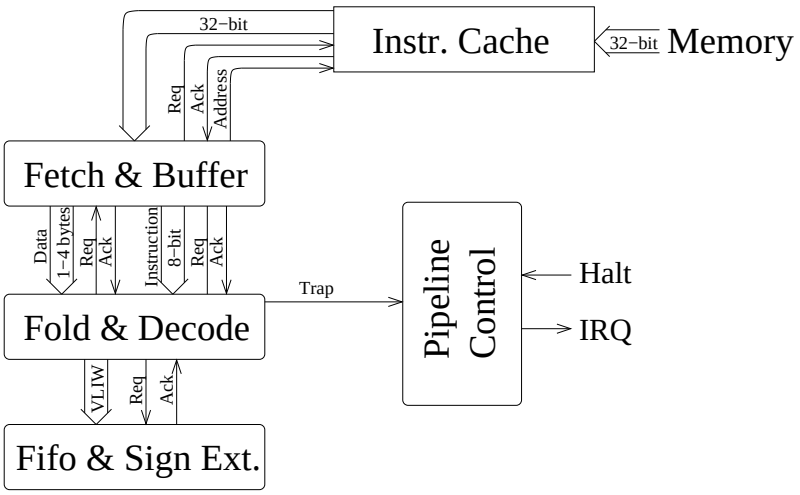


Figure 3.4: The instruction preprocessing pipeline.

3.3.5 Operand Access, ALU and Result Storing

The operand access unit takes care of providing the ALU with the actual operands, which may come from the local variable area, the stack or as literal data from the bytecode stream. The operand access unit has two read channels to the top of the stack, one read channel to the local variable area and one bypass channel to the end of the ALU. This bypass channel reduces unnecessary traffic to and from the stack. This can be demonstrated with an example of an addition followed by a multiplication. In the straightforward method the operations would be carried out as follows. First the addition is performed and the result stored to the stack, then the stack is read out to perform the multiplication. The result of the addition is consumed by the multiplication and does not remain in use. The improved method eliminates the consecutive write and read functions and replaces them with a straight connection from the result of the ALU to the operand access unit. This solution provides better performance in terms of execution time and power consumption. Note that the bypass method provides similar benefits as instruction folding and they address the same shortcoming of Java bytecode. The difference between these two methods is that folding can be done in advance and is performed on load-calculate-store sequences, whereas bypassing takes place after the first instruction is completed and is performed on consecutive calculate type operations. It is also worth noticing that the bypass method is quite similar to operand forwarding in the register banks of general purpose processors.

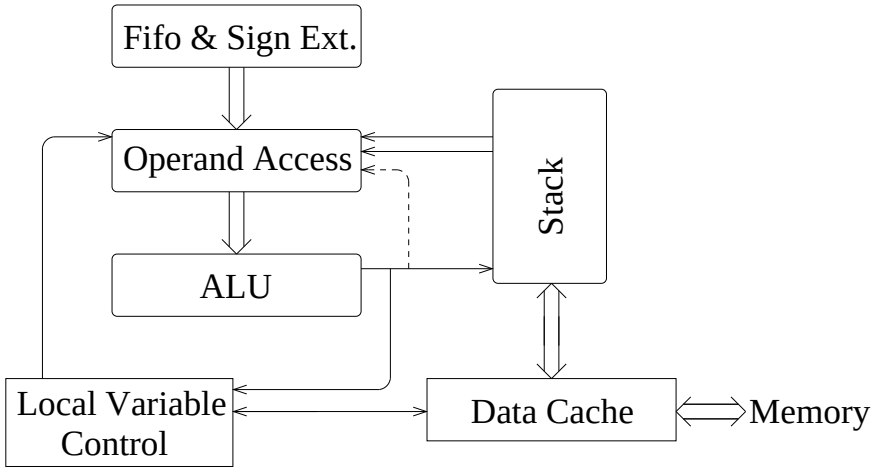


Figure 3.5: The execution pipeline and data transportation. The bypass channel is highlighted with a dashed line.

Figure 3.5 shows the data connections in the execution part of the pipeline. The request and acknowledge signals are not shown, in order to keep the figure readable. The result of the ALU usually goes to the top of the stack. In some cases the result is directed to a local variable. The third possibility is to intercept the result and direct it straight to the operand access unit. This happens when the current instruction in the ALU would push its result to the top of the stack and the next instruction would pop it away. The state of the stack remains as if the first result had never been pushed. The interception thus saves power and time, at the cost of slightly more complex logic.

3.3.6 Caches, Stack and Registers

The JPU contains two caches, namely the data cache and the instruction cache. The instruction cache is (quite naturally) read-only, whereas the data cache can be written and read. The instruction cache is less complex also because it is connected to only one unit, namely the instruction buffer. The data cache, on the other hand, is connected to the stack and to the local variable control. The writing to the data cache is implemented using the write-through strategy, in order to keep the memory consistency during traps and context switches easier to manage. Both caches also give state information to the pipeline control unit, to notify the controller when the current operations are finished.

The stack is implemented as a ring buffer with memory roll-back. The buffer holds the top of the stack. When the buffer is close to full, the bottom of the ring is rolled to the memory via data cache. Naturally if the buffer is close to being empty more data is retrieved from the memory. The stack performs these transactions automatically, and no direct commands are required during normal execution. However a command for flushing the stack to the memory is required, since jumping to a method causes a new stack-frame to be initialized, with its own local variables and return information.

The internal registers of the JPU are all addressable from the CPU. This is required in order to be able to configure the JPU in the beginning of the execution as well as during thread switching. There are four internal registers, which are shown in Table 3.1. The registers are not general purpose data registers used for temporary data storage, but control registers used for controlling the execution. In addition to the PC and the CO presented earlier, the register bank contains a pointer for both the top of the stack (ST) and the beginning of the local variable area (LV). The pointers as well as the CO register are direct addresses in the local memory region of the co-processor. The internal register bank also contains configuration data from the JPU to the CPU. The most important piece of information delivered here is the size of the local memory. The REALJava virtual machine also supports multiple instruction sets so the instruction set of the co-processor needs to be delivered to the software. Currently two instruction sets have been defined, one with floating point instructions in hardware and one with software emulation. Other subsets of the Java bytecode can be defined, in order to suit the needs of a specific target device and application.

Address	**Name**	**Function**
0	PC	Program counter
1	ST	Pointer to the top of the stack
2	CO	Code offset
3	LV	Pointer to the local variable area

Table 3.1: The internal control registers.

3.3.7 Details of Instruction Folding

Instruction folding [54] is performed in order to remove unnecessary cycles in ALU and also to minimize redundant stack accesses. These performance hindrances are caused by bytecode instructions first pushing a value to the

stack and immediately popping it out for processing. The folding procedure removes these two (or more) instructions, and replaces them with one instruction carrying the value and the processing instruction to the ALU in one cycle. This eliminates some of the completely unnecessary memory accesses, thus reducing power consumption and improving performance in time domain. Memory accesses dominate the power consumption of a JVM, according to [33, 34] around 70% of the energy is consumed in memory accesses. The results are gathered using a software based Java virtual machine running on ARMulator, an emulator for the ARM7TDMI processor. It is reasonable to assume, that the power consumption of the REALJava virtual machine will follow same trends.

LV	A local variable load or a constant load
OP1	An operation that uses the topmost element of the stack and pushes a one word result to the top of the stack
OP2	An operation that pops the top two entries of the stack and pushes a one word result to the top of the stack
OP1_B	An operation that uses the topmost element of the stack and breaks the group
OP2_B	An operation that uses the top two entries of the stack and breaks the group
MEM	A local variable store
NF	Non-foldable instructions
TRAP	An instruction which is trapped by the hardware and is executed in software instead

Table 3.2: Instruction folding classes

With the classes presented in Table 3.2, folding of the instructions can be performed in the patterns shown in Table 3.3. These patterns all produce a very long instruction word (VLIW) with up to two literal data elements, an opcode and a destination identifier. It can be noticed that the maximum length of folding is four instructions. This, however, does not mean "only" four bytes in the original bytecode stream. The original stream may have had some literal data included, and these are also placed in the VLIW, as shown later in Figure 3.6. The folding classes and the amount of parameter data for the bytecode instructions can be found in Appendix A. After identifying the possible folding patterns, a preliminary analysis was performed. According to the analysis the reduction in the number of executed instructions was between 26.8% and 33.3%. In stack accesses the reduction varied between 39.3% and 51.2%. This analysis was run on a modified version of

SableVM [96], which was equipped with a dummy folding unit that produced the foldings according to the rules but still executed the instructions as they were in the original stream.

Pattern	Instructions
LV LV OP2 MEM	4
LV LV OP2	3
LV LV OP2_B	3
LV OP2 MEM	3
LV OP1 MEM	3
LV OP1	2
LV OP1_B	2
LV OP2	2
LV OP2_B	2
LV MEM	2
OP1 MEM	2
OP2 MEM	2

Table 3.3: Possible folding patterns

The fact that the whole co-processor is asynchronous helps the folding process. In asynchronous circuits the blocks can run at independent speeds. This means that the folding unit can perform for instance a maximum of n foldings per second, whereas the ALU may be significantly slower, say $n/2$ operations per second. The negative effects of independent speed, such as waiting for one long operation halting all other pipeline segments, can be reduced using an intermediate fifo. The timing marginal for folding is increased because with asynchronous techniques all units exhibit average case performance. This means that the ALU may complete some instructions (bit-wise OR, etc.) in very short time, whereas some instructions (32-bit multiplication) take a lot more time. Since folding may produce new VLIW instructions at the rate of 1/1 to 1/4 in comparison to the original bytecode stream, the fifo balances the effects of both folding and the average case performance of the ALU. In this architecture the fifo also performs minor tasks, such as sign extension and address calculation for local variable accesses. These tasks would otherwise have to be preformed in an additional pipeline stage, so the buffer serves a computational purpose besides the balancing of the execution rates.

The folding unit receives data from the instruction buffer. The instruction cache handles the actual memory accessing, so the instruction buffer

needs only to access the cache. The physical address in the local memory is generated at the instruction buffer. The folding and decode unit has two communication channels to the instruction buffer. This is required because instructions may be followed by data, such as a literal operand or an address. The amount of data can be found out only by decoding the instruction first. After the decoding is completed, the correct amount of data bytes is read in parallel. The amount of data is between 0 and 4 bytes. If it is 0 bytes, no request is sent to the data read port. Since the data items are read in parallel to the fetch data module shown in Figure 3.6, the instruction buffer can move the next instruction to the output stage of the buffer without unnecessary delays.

After the instruction has been decoded and the data related to that instruction is read in, the next instruction is checked to see if it can be folded with the previous one. If it can be, then the procedure is repeated to see if the third instruction can be folded. If at any point the instructions cannot be folded together, the previously folded instructions are sent out, and the procedure starts over with the current instruction as a tentative base for a new folding.

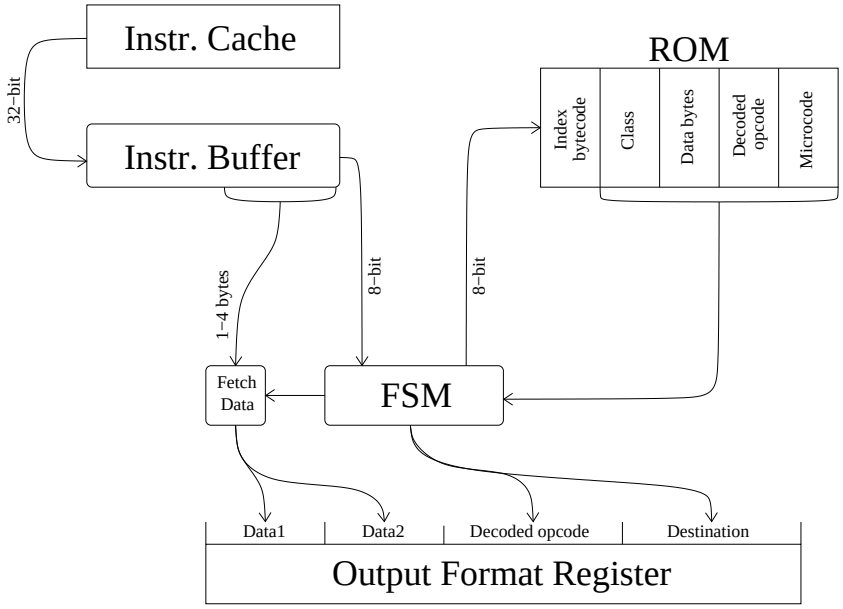


Figure 3.6: The internal structure of the folding unit.

Figure 3.6 shows the internal structure of the folding unit. The FSM stands for Finite State Machine, which controls the operation of the unit.

The ROM table approach is chosen, because Java bytecode is not optimized for hardware decoding. The instructions of Java bytecode are just listed in seemingly random order and given the order number as an opcode. This would lead to a very complicated decoder, if implemented directly using standard logic elements. The ROM approach is also further validated by the fact that microcode and other metadata can be easily stored in the same table, as well as the instruction classes and the number of literal data bytes related to a given instruction. This keeps our FSM simple and fast. All the entries in the ROM table are coded with one-hot scheme and the table is implemented as a precharged MOS NOR ROM matrix. The precharging is done when request is low, so the response time is minimal. The response time of the ROM is critical for the performance of the whole folding unit, since every instruction needs to be decoded and classified by the data from the ROM.

The output format register stores partial foldings until they are completed. If a folding pattern is not terminated with a valid instruction for that pattern, the partial folding is executed one by one, and folding of the next instruction will be attempted. The register keeps record of which fields in it are valid at any given time. When a pattern is completed, the register pushes its contents to the fifo in the main pipeline, and prepares for a new folding autonomously.

3.4 Software Partition

Because the co-processor executes only a subset of the instructions in the Java virtual machine instruction set, executing actual Java programs with it requires supporting software written for a general-purpose CPU. This supporting software needs to do most of the things that a generic virtual machine does, like class loading and garbage collection, but it also needs to control the bytecode execution and the local memory of the co-processor. In this Section, the most important functions of the software partition of the REALJava virtual machine are described. The software components required to support a co-processor are discussed in more detail than the parts found in typical software Java virtual machines. The software coding techniques are left out as they fall outside the scope of this thesis, and only the high level functionality is presented. More details of the software partition can be found in [62], which deals with the initial version of the software partition.

3.4.1 Virtual Machine Software

A typical Java virtual machine implemented purely in software loads Java classes, manages resources such as memory and threads, provides an interface to the virtual machine for native code, and most importantly, controls the bytecode execution. The part of a virtual machine that executes bytecode is called its execution engine. This is the part that typically uses the largest amount of CPU time during the execution of a Java application.

The simplest software implementation of a bytecode execution engine is a bytecode interpreter. In an interpreter, the software fetches one instruction at a time, then branches according to the opcode and finally executes the native instructions implementing the Java bytecode instruction in question. This loop is continued until the interpreter encounters an instruction that requires special processing. For example, a method invocation instruction may require calling a native function or loading a new class to the virtual machine.

Although an interpreter is simple to implement, it is not very efficient. Since the Java virtual machine is entirely stack-based, interpreting bytecode requires a large amount of memory accesses even for relatively simple operations. For example, the following sequence of instructions, which multiplies a local variable by 10, requires six stack accesses:

```
iload_0    ; push 1
bipush 10  ; push 1
imul       ; pop 2, push 1
istore_0   ; pop 1
```

The source code for the example could be `x = x * 10;`. Besides the stack accesses, the code segment also accesses the local variable area two times, once for the first load and another time for the last store. This makes the total number of data side accesses to the memory eight. For the instruction side, the number of accesses in this example is five. That number is composed of the four opcodes and the parameter data for the *bipush* instruction. Since the data and the instruction stream are likely to be placed in the same physical memory, the total number of accesses is 13, and this still does not include the house keeping, such as updating the PC and the top of the stack.

There is also a per-instruction overhead in an interpreter caused by the pointer access used to fetch the instruction and by the instruction dispatch itself. Many optimizations have been developed to reduce this overhead.

Direct-threading and inline-threading [16] seek to reduce the instruction fetch and instruction dispatch time by converting opcodes into the corresponding native instructions before execution. Just-in-time-compilation [11] further optimizes the generated code and reduces stack accesses as well by using the host CPUs registers to store intermediate results. Because the Java instructions are replaced with native instructions, these optimization strategies also remove the bytecode program counter.

The structure of the execution engine needs to be changed to support a bytecode co-processor. Rather than executing the bytecode instructions in software, the virtual machine loads the required method's code segment to the local memory of the co-processor and sets the internal registers to appropriate values. Then the execution is continued on the co-processor until it encounters an instruction that it cannot execute or the software commands it to halt. The execution in the co-processor can also be suspended because the current thread has used its time slice. The thread scheduling algorithm is discussed later in Section 3.4.3.

Since most of the processing is done on the co-processor, many improvements to the software part of the execution engine become unnecessary and impractical to implement. Because the virtual machine needs to be able to update the stack and the internal registers of the co-processor when the execution is transferred from one domain to the other, optimizations that reduce stack accesses or replace the program counter become unusable as such. However, they also become largely unnecessary, because the co-processor takes care of most of the menial stack manipulation and the program counter and stack pointer are updated in parallel to the actual execution in the co-processor, utilizing the inherently parallel nature of hardware.

The software partition of the REALJava virtual machine is implemented in C++. The virtual machine supports JNI [99] and the standard edition of the Java 2 platform [100]. Currently the REALJava virtual machine runs in Windows and Linux on x86 computers or in Linux on PowerPC based systems. Since the software is coded in C++ with no assembler optimizations, porting the software to new architectures and operating systems should be relatively easy. The virtual machine also contains a simple emulator of the hardware's capabilities, and can be used for testing new functionality on software.

The structure of the REALJava virtual machine is shown in Figure 3.7. Like a generic Java virtual machine, it contains a native interface, a heap memory manager and a class loader. The execution engine of the REALJava virtual machine, however, is split between the software and the co-processor.

The software partition of the virtual machine also manages the local memory of the co-processor for java stack and method code segments and implements a simple thread scheduler for allocating threads to the co-processor.

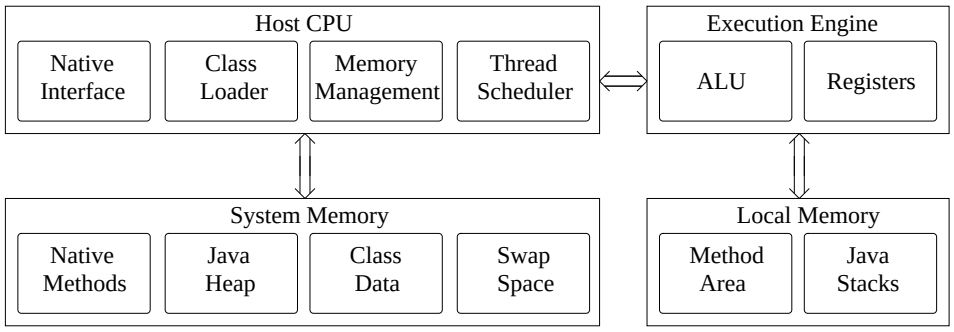


Figure 3.7: Logical layout of the REALJava Virtual Machine. The left side of the figure is the host CPU's domain, while the right side is the co-processor's domain. On both sides the upper part represents the computational loads and the lower part shows the memory regions.

3.4.2 Bytecode Execution and Method Invocation and Return

The virtual machine needs to do some preparation before it starts executing a bytecode segment on the co-processor. First, it has to acquire the lock on the co-processor. Co-processor locking is discussed in section 3.4.3. Second, it has to check that the current thread's stack frame and the current method are loaded in the co-processor's local memory. Memory management is discussed further in section 3.4.4. Finally, it has to update the internal registers of the co-processor. While the co-processor is executing bytecode, the software is free to do other tasks. For example, it could optimize methods by converting instructions to "fast" versions in advance or load and verify classes that will be needed soon.

When the virtual machine calls a new method on the co-processor, it has to do some extra work on the stack. First, it checks if there is enough space to initialize a new stack frame in the currently allocated stack page. If there is not, or a stack page is not currently allocated for the thread, it allocates a new stack page on the co-processor. Once there is enough space for the stack frame, the method parameters are popped from the original stack and passed to the new stack frame. This involves no actual data transferring, the data is moved logically, not physically. Then the current

registers are pushed to the new top of the stack. These values are used when returning from the invoked method. The details of stack behavior during method invocation are further discussed in Chapter 5. If the previous stack frame was swapped out by the allocation or there is none, a magic number (a bit pattern of all ones is used) is pushed instead of the current program counter. After the registers have been pushed, the local variable pointer is set to the previous method's stack top pointer (after logically removing the parameters), the new stack top pointer is incremented by the amount of local variables and the return information size, and the program counter is initialized to zero. Once these operations have been completed, the new frame is initialized and the execution can proceed on co-processor. When a method has already been successfully invoked, the required data is already available to the co-processor. This can be used to accelerate future invocations of the same method, as discussed later in Chapter 5. A simplified view of the procedure is shown in Figure 3.8.

Sometimes, a return from a function must be executed in software when a method returns in the bytecode. For example, when calling a Java method from native code, the software needs to be able to return to the proper native function after the call. For this reason, the software must be able to force a trap in the return instructions. The most significant bit of the program counter register is used for this purpose. A limitation in Java's exception handler implementation practically limits the Java program counter to 16 bits [38], so the upper 16 bits can be used to store data required by the virtual machine. If a software return is required, the bit is set to 1 when storing the registers of the previous stack frame. When a return instruction traps, the software handles it like the co-processor would, except it also clears the most significant bit of the program counter and also performs one return in the software thread.

The virtual machine also needs to be able to handle traps from the co-processor. Most of the time this means simply executing code for the instruction that caused the trap. As an example, when an *iaload* instruction causes a trap, the CPU reads the required parameters from the co-processor, fetches the data from the heap memory, pushes the data to the co-processors stack and signals the co-processor to continue execution. A simple interpreter is used to execute the bytecode instructions in software. The interpreter is based on a "switch" statement, with a "default" branch that transfers the execution back to the co-processor. This way, the instructions that can be handled on the co-processor are never executed in software. This simplifies the switch and also reduces the size of the software.

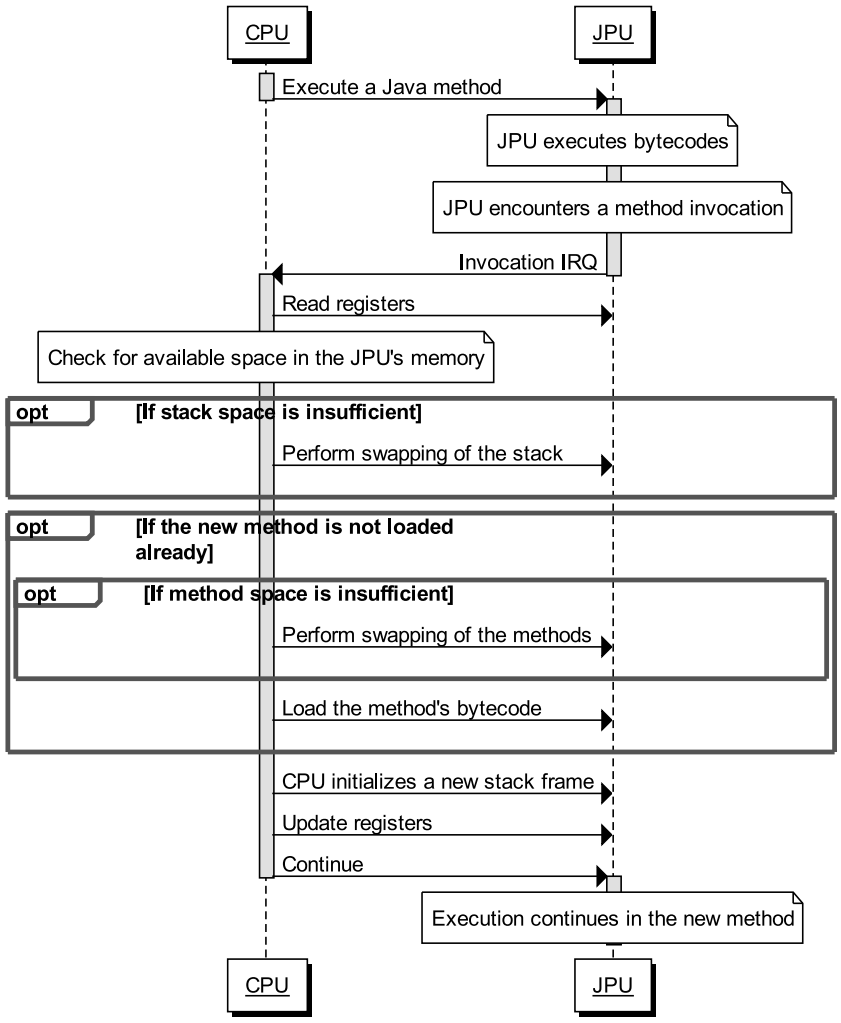


Figure 3.8: A MSC of a method invocation.

3.4.3 Co-Processor Allocation with Multithreading

In a single-threaded environment, controlling the co-processor usage is simple. Since there is only one thread using the co-processor, no locking or pre-empting is necessary. When the thread needs to execute code on the co-processor, it always knows that the co-processor remains in the state that it was when it made the last interrupt request. Therefore, it can send "continue" commands to the co-processor any time without additional checks.

In a multithreaded environment, the software partition of the virtual machine must control co-processor usage for two reasons. First, to prevent

invalid behavior, multiple threads must be prevented from accessing a single co-processor simultaneously. For this reason, each co-processor is required to have a lock that the threads acquire before using it. Second, a single thread must be prevented from holding the lock on a co-processor indefinitely, because it could lead to starvation and possibly deadlocks in the other threads.

A simple time slice-based pre-empting system is implemented in the software. To implement pre-empting, each thread's "resume" routine occasionally polls the virtual machine's access control system to see if the thread has exceeded its time slice. If it has, a "halt" command is send to the co-processor. This stops the execution and after that the thread releases the lock on the co-processor. Once the thread can acquire the lock again, it can resume execution. The access control system is also polled whenever the co-processor sends an interrupt request. Another way to implement pre-empting, which requires hardware support, is to have the hardware trap after a predefined number of clock cycles has passed since the last thread switch. This removes the need for active polling. The hardware assisted thread scheduling strategy is further discussed in the Chapter 5.

A thread must also release its lock on a co-processor if it does something that could potentially take a long time without needing the co-processor. Otherwise, other threads might starve or even become deadlocked. The most common possibly lengthy operations during normal execution are monitor acquisitions and native method invocations. For monitor acquisitions, it is usually possible to atomically test whether a monitor can be acquired and acquire it if possible. In this case releasing the lock of the co-processor is only necessary if the monitor is held by another thread, causing the current thread to wait for the monitor.

Native methods, on the other hand, are essentially "black boxes" for the virtual machine, since it cannot know what a dynamically loaded chunk of native code will do or how long it will take to execute the code. A native function might, for example, initiate a native I/O operation and block until the input arrives. Therefore, for most of the native method calls, the lock on the co-processor must be released. As an exception, certain known "safe" functions in the standard library can be assumed to execute quickly. The method Math.sin, for example, takes a constant, short amount of time to execute regardless of input, and therefore should not require releasing the lock. A simple list of such methods is implemented in the software partition of the REALJava virtual machine to avoid unnecessary release-acquire cycles.

3.4.4 Co-Processor Memory Management

In the REALJava virtual machine the local memory on the co-processor is used for two purposes: the thread stacks and the method bytecode segments. Memory on both is allocated in a similar way.

The local memory of the co-processor is logically split to a stack region and a method region when the virtual machine starts. The available memory is initially split simply in half, but the division point may move during execution. In practice, the amount of memory required for stacks is not very large. Programs that do not use heavy recursion usually do not require stacks larger than 10 kilobytes [16].

Memory is allocated in fixed-size pages. This makes reclaiming and swapping out memory easier. The software partition of the virtual machine must swap out the memory contents from the co-processor if it runs out of memory allocated for stack pages. Methods are never swapped out, because they are also stored in the memory region of the host CPU. If the virtual machine has to allocate a new method page, it simply overwrites the least recently used page.

The software stores some information for each page. First, it uses a bit vector in which a one indicates a page that is currently in use. Second, it uses an array to store the time that the page was last used and certain information used when swapping out pages. The time is updated every time the page is used by the software.

In order to allocate a page, or multiple pages, the virtual machine first checks if there is an unused page in the region required. If there is, this page is returned. Otherwise, the virtual machine finds the least recently used page, swaps it out to the host CPUs memory if it is a stack page, and returns that. A simple "sliding window" algorithm is used to allocate multiple pages.

If a stack page refers to another page that gets swapped out, the referring page must be updated to prevent return statements from returning to a page used by another thread. Therefore, if there is a stack frame with a pointer to the swapped out page, the previous frame's program counter in that frame is set to the magic number mentioned in Section 3.4.2. When this return is executed, an interrupt is generated by the co-processor and the software partition swaps the page back into the local memory. Swapping out method pages is easier, because the only thing that needs to be done is to remove the mapping from the swapped out methods to the pages.

3.4.5 Garbage Collection

Garbage collection means finding the set of objects that are reachable and reclaiming the memory used by unreachable objects. Reachable objects are ones that are referred to from the thread stacks, the local variables, the static member variables or the other reachable objects. Most Java virtual machines implement garbage collection to ensure that the virtual machine will not run out of memory.

Although the garbage collector can be modified to run in parallel with normal execution, the rest of the system is usually stopped for garbage collection. This is known as *stop-the-world* collection. Garbage collection is started when the virtual machine runs out of heap memory or by a request from the user application. Since the virtual machine is often running multiple threads, the garbage collector has to wait until all of the threads have stopped running. The individual threads therefore need to poll the garbage collector at certain points during execution to check if garbage collection is starting. These points typically include method invocations and backwards jumps. Polling at backwards jumps is important because without it, an infinite or very long loop could prevent garbage collection and stall the whole virtual machine.

When the bytecode execution is performed on a co-processor, polling for garbage collection at backwards jumps becomes impractical and time-consuming. However, since the co-processor thread scheduler, discussed in more detail in Chapter 5, is guaranteed to halt the execution at some point after the thread's time slice runs out, this is not a problem. Polling can be done every time when a thread releases the lock on the co-processor allocated to the thread. This way, every thread is guaranteed to stop at some point when garbage collection has to be started. If so desired, the software can also set the thread slice timers to a very small value, for instance 10 clock cycles, in order to cause threads stop faster.

The virtual machine also needs to be able to locate the current stack frames for garbage collection. A simple array is used to store information for each stack page. This array contains data on whether the page is loaded in the local memory of the co-processor, what its hardware address is, and if it is swapped out, where the contents are stored. All stack frames are found by traversing backwards from the top stack frame until the bottom of the stack is reached.

The actual garbage collection is implemented using a tracing garbage collection algorithm. This is implemented in three phases. In the first phase the root references are collected. The root references are object references stored in the stack and in the static variables of loaded classes. The second phase traces all the reference paths from the root set, marking all reached objects as live. This marking phase uses Dijkstra's [13] three color model. Finally, in the third phase, the live objects are compressed, in order to defragment the memory region.

The third phase is implemented in two variants. The first variant is used when the system memory has limited free space available. This variant uses simple mark-sweep-compact algorithm to compress the live objects. This means that all of the live objects are moved to a new location inside the already reserved memory space, one after another so that no empty space is left between objects. The other variant is used when the free space in the system memory is sufficient. This variant reserves a new, larger, memory space, copies all of the live objects to the new space and finally releases the original memory space. The latter is faster, but since the REALJava virtual machine is primarily targeted at embedded devices, it is reasonable to assume that the memory available to the virtual machine will be limited, thus forcing the system to use occasionally the slower but more memory efficient algorithm.

3.4.6 Porting the Software

The software partition of the REALJava virtual machine was originally implemented using Microsoft VisualStudio. This was done under the Windows XP operating system with an x86 compatible processor. At this time the REALJava virtual machine supported two modes, pure software execution and co-processor execution using the XESS[1] board connected via parallel port. Later the whole REALJava was ported to the Linux environment with GNU compiler tools. Also the Linux system was based on an x86 compatible processor architecture. The virtual machine has the same two modes also in this environment. The transition from Windows to Linux was relatively simple, since most of the code for the software partition is platform independent. Most of the differences are in the communication module. The hardware is exactly the same for these two versions. The last environment was the PowerPC 405 based Linux system. This was implemented using the ML310 and ML410 demonstration boards from Xilinx. The PowerPC version has two modes, one for software only execution and one for accelerated execution with a co-processor and communication via PLB bus. Details of

[1]For descriptions of the XESS, ML310 and ML410 boards, see Chapter 4.

the systems are presented later in Chapter 4. Finally a rudimentary multicore support is added to the PowerPC version. This is briefly discussed in Chapter 6. The multicore version is not implemented on the x86 architecture, since the co-processor connects to the host system via parallel port, and computer systems usually have only one parallel port available.

3.5 Data Structures

The REALJava virtual machine has three main data structures, namely the bytecode, the stack and the heap. All of these have a unique structure, and the heap is also subject to the garbage collection. Each of the data structures is placed in a location of their own, both logically and physically. The bytecodes are stored to the method area, located in the upper half of the co-processor's local memory, with the stack in the lower half. The heap is stored in the system memory.

3.5.1 Bytecode Storage Model

The bytecode segment for the method to be executed is stored to the local memory of the co-processor. Besides the actual code, two custom fields are stored before the code. These items are the method id, used by the software partition to identify the currently running method, and the constant pool pointer, used to locate the constant pool of the current method. The latter provides significantly better performance for accessing the constant pool since no symbolic resolution is required. The constant pool address is a direct pointer to the host CPU's memory, and accessing a specific constant simply requires adding the offset of the constant to the pointer. Without this custom field the software partition would have to perform a lookup to find out the current methods descriptor, then find the constant pool address in the descriptor and finally continue as in the strategy described before.

The code segments are allocated starting from the top of the co-processor memory space. New methods are always added below the previous ones, growing the method area of the co-processor from the top of the memory towards the bottom. The amount of methods that can be stored in the co-processors memory is not limited to any fixed number, but rather depends on the amount of memory available and the amount of stack space required by the application. If required, the software partition is responsible for removing unused methods from the memory or performing page swap on the method area.

3.5.2 Stack Layout

The layout of the data in stack is not defined by the Java specification. The specification does, however, define the way parameters to a method invocation are arranged. The calling method loads the parameters to the top of its own stack, and once the method is being called, the parameters become the local variables of the new method. At the same time the parameters are removed from the stack of the calling method. This clearly suggests that placing the local variables at the bottom of the stack frame and the operand stack at top of the frame avoids unnecessary copying of the parameter data. Since the stack frame also contains return information, it is natural to place that information in between the other two components as shown in Figure 3.9. With this arrangement only one dynamically changing pointer is needed, pointing to the stack top (ST). The local variables are accessed using offsets from the local variable pointer (LV). The return information is accessed by adding the total number of the local variables in the current method to the local variable pointer.

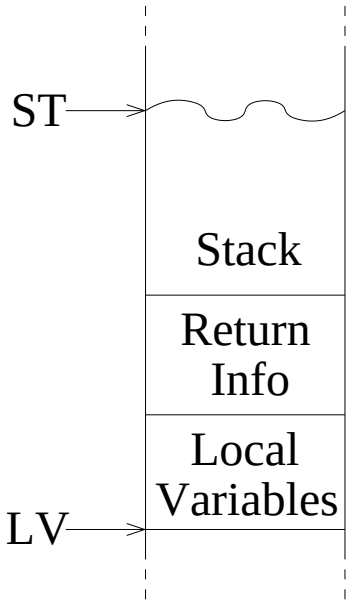


Figure 3.9: The layout of data items in the stack frame.

When the REALJava virtual machine is started the stack pointer is initialized to the bottom of the co-processor's local memory. The stack then naturally grows from the bottom towards the top of the memory space. The maximum amount of stack space is not limited. When the stack space of the co-processor is becoming full, an interrupt is generated and the software

partition performs swapping of the stack pages, as mentioned in Section 3.4.4. The swapping is expected to take a relatively long time, depending on the bus connecting the CPU and the co-processor and of course on the speed of the CPU. Overall, the swapping mechanism is not intended for heavy use, as it would slow the system down seriously, much like swapping the main memory to a hard drive in typical desktop computers. The mechanism is provided for completeness and also in order to facilitate different local memory sizes in the future. With the stack growing in this direction and the method area growing in the opposite direction the memory can be used efficiently for cases when either the stack or the method area grows faster than the other.

3.5.3 Heap Management

Objects and arrays, which are handled as ordinary objects in heap manager, are allocated in the heap memory. The software partition controls the heap memory, taking care of allocation, data access and freeing unused memory via garbage collection. Thus, the heap memory region is placed in the memory of the CPU. The REALJava virtual machine starts by allocating a relatively small amount of memory for the heap, and growing it if necessary.

The garbage collection has some practical requirements for the heap manager. Since the heap memory is subjected to the collection, and this means that the objects in the heap may be relocated during the garbage collection, it is necessary to provide a mechanism that allows referring to objects without using their actual physical memory addresses. If physical addresses were used as references in the virtual machine, the references would have to be updated every time the garbage collection algorithm moves the object. This would be very hard to do, since the references may reside in other objects or anywhere in the stack. To avoid this problem, a two-tiered referencing system is adopted. All the references stored in the objects or the stack are actually indices to a global reference table. This table in turn contains the addresses of the actual objects in the physical memory space of the CPU. When an object is moved, only the entry in the reference table is updated, and all of the references stored elsewhere remain as they were. The referencing is shown in Figure 3.10.

When an object is deemed to be garbage and subsequently removed, the address in the reference table is marked as free. The next `new` instruction that allocates a new object receives the first free reference location from the reference table. The address of the newly created object will be placed into that location, and the reference returned to the Java application is the index

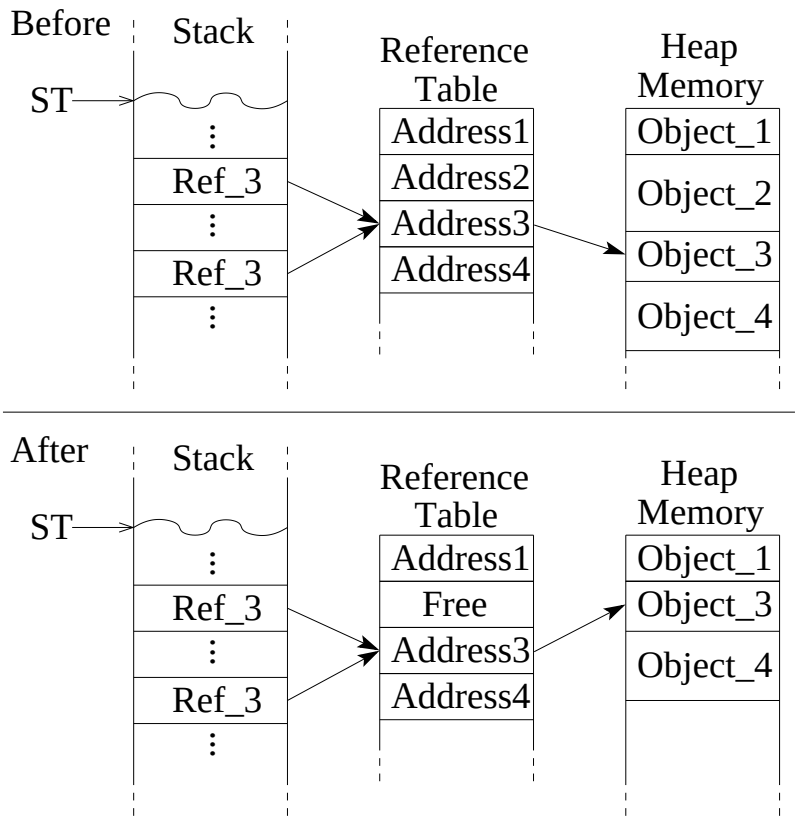


Figure 3.10: An example of the reference table during garbage collection. Object 2 is found to be garbage, thus it is removed. The other objects are compacted, but the references to them remain as before in the stack, only the pointer in the reference table is modified.

to the reference table. As a result the order of the objects in the memory is not related to the index number of the reference. The allocation strategy chosen for the new references being inserted to the table also renders the index number in the reference table meaningless as an indicator of the objects lifetime.

3.5.4 Other Data Structures

Besides the three main data structures, the REALJava has six additional data structures. All of these are stored in the system memory, and are thus controlled by the software partition. They will be described next. (1) The **reference table** mentioned above, used for referencing to the Java objects

in the heap. (2) The **static member area**, used for storing the static members of classes. The data items here are initialized when a new class is loaded. (3) The **class area**, which is used for storing the methods and the constant pools of loaded classes. (4) The **string area**. It is used for storing unique strings, such as "Hello world!" and so on. (5) The **code segment area**. This area contains the executable native code of the software partition and also the required libraries for I/O and native interfaces. (6) The **swap area**, which will be created if the co-processors local memory is full, and needs to be swapped to the system memory. This area is initialized only if it is required. The run time sizes of the areas are highly application dependent. Since all of the areas, save for the code segment area, can grow when needed, the sizes can be set to any values that are seen as suitable for a given system. There are no actual limits posed by the REALJava, except that the pointers to the regions are 32 bits long, causing the maximum total amount for all areas to be 4 Gb. This should be more than sufficient for embedded systems.

3.6 Chapter Summary

In this chapter the REALJava virtual machine was described at a conceptual level. The execution model used in the research for this thesis was shown. The hardware software partitioning was defined, followed by the structure of the hardware partition. The software partition was also discussed shortly, with the emphasis being on the components related to the co-processor. Finally, the details of the internal data structures were specified.

Chapter 4

Prototyping the REALJava Virtual Machine

The Java virtual machine presented in Chapter 2 was partitioned in Chapter 3. In this chapter FPGA technology is used in the prototyping of the resulting REALJava virtual machine. The prototypes have also been presented in [57]. The assumptions made during the partitioning and structural specification are replaced by ones fitting to the target technology. This chapter begins with details of the major changes required to meet the FPGA specific constraints. The FPGA platforms used for the prototypes are shortly described, along with the properties of the PowerPC 405 embedded processor integrated in the larger FPGA platforms. The FPGA specific tools and techniques are outlined next. The physical size of the co-processor core is also evaluated with comparison systems spanning from Java co-processors to full general purpose processors. Finally, the communication subsystems for all of the platforms are presented.

4.1 Major Changes from the Conceptual Model to the FPGA Implementation

The first and possibly the most drastic change is the instruction folding unit, which was dropped from the design. The folding was not included because the FPGAs in use do not have fast ROMs, which were specified in Section 3.3.7 to be a prerequisite. Also the fact that the FSM inside the folding unit would have to be synchronous, and thus run at the same speed as the rest of the system, is clearly prohibitive. Since the folding was left out some other measures had to be taken in order to keep the execution rate high enough. To this end the data flow between the stack and the ALU was modified. All the data in the stack goes through a cache and is automatically forwarded

to the ALU. The ALU maps the data items as operands using predefined rules obtained from the Java bytecode instruction set. The rules state for instance that in integer subtraction the topmost entry of the stack is subtracted from the second entry. Since the Java compilers produce code that minimizes the number of stack locations used, most of the code produced follows the lines of load, load, compute, store. This means that the top two locations of the stack are generally loaded from the local variable area just before an arithmetic operation, causing the data items to be in the cache and thus readily presented to the ALU. A partial version of folding is performed at the output of the ALU. If the result is going to be moved into a local variable, then it is written directly there. In the straight forward implementation the result would be first written to the top of the stack, and immediately moved to the local variable. This form of instruction folding needs only to check whether the next instruction is a local variable store. This check is easy to implement, since the instruction fetch unit provides the instruction stream parameters to the ALU. The parameters are located after the related instruction, so reading the parameter data during the execution of an instruction that does not require parameters actually provides the next instruction.

The last changes were made to the control register bank. The method invocation module described in Section 5.5 required two additional registers to be implemented into the co-processor. These registers provide the ALU with the number of local variables and the number of parameters required by the Java method to be invoked. Since both of these numbers are 16 bits long, they were combined into a single 32-bit register location, which is named LO. Also a set of other addresses were added, in order to minimize unnecessary communications between the CPU and the co-processor. These additions will be presented later in Chapter 5.

4.1.1 Changes in the SW Partition

The software partition required only minimal changes from the conceptual view, and most of the changes were brought on by new acceleration strategies applied to the co-processor. Since the execution model was not changed, only the instruction set supported by the co-processor caused major modifications. Even those were limited to selection of which instruction was to be implemented in which partition. The changes in the communication protocol were not really changes in the software partition, since the communication scheme for each platform is encapsulated in its own module. This allows the same routines to be used throughout the software regardless of the platform used for the implementation.

4.2 Description of the FPGA Platforms

Xilinx FPGAs have been chosen for the prototypes, since they provide CPU(s) embedded into the same chip with the FPGA logic. Xilinx uses PowerPC 405 processors as the embedded CPU(s). Two such CPUs are integrated on the Virtex II Pro chip found on the ML310 demonstration board and also on the Virtex4FX on the ML410. The XESS 3S1000 does not provide an embedded CPU, and the duties of the CPU have to be performed off-board.

4.2.1 XESS XSA-3S1000

The smaller prototype is implemented on a XESS XSA-3S1000 board [75]. This board provides a Xilinx Spartan3 1000 [77, 78] FPGA chip and a 32 MB memory chip. The FPGA runs at 100 MHz and the memory is an SDRAM with 70 ns access time. The board also has a Xilinx XC9572XL CPLD device, which is used as a bootloader. The CPLD is programmed with a communication system that accepts the bit stream for FPGA configuration from the parallel port. After the FPGA has been correctly programmed, the CPLD gives the FPGA chip full control over the parallel port connection. A simple seven segment display shows the current state of the parallel port connection when the FPGA device is being loaded. Further details of the board can be found in the datasheet [75].

Since the Spartan3 series does not have an embedded PowerPC processor, a standard desktop computer is used as the CPU for the virtual machine. The communication between the co-processor and the CPU is implemented using a parallel port (in standard LPT mode). This solution allows great platform independency, but provides only a slow link between the co-processor and the CPU. The communication subsystem was measured to have a bandwidth of about 80 kilobytes per second when running Windows XP, and about 200 kilobytes per second when running Linux (Kernel 2.6.18). This difference is most likely due to the fact that in Windows the communication uses a separate software device driver, while in Linux the communication is implemented as direct memory mapped I/O. The main reason for designing and maintaining this prototype is the time it takes to synthesize and implement a design. The whole cycle takes less than 15 minutes on an Intel Core2Duo running at 2 GHz, while the larger system takes about 2 hours. This allows fast paced iterative co-design of the software and hardware partitions. The XESS board also eases debugging, as the board

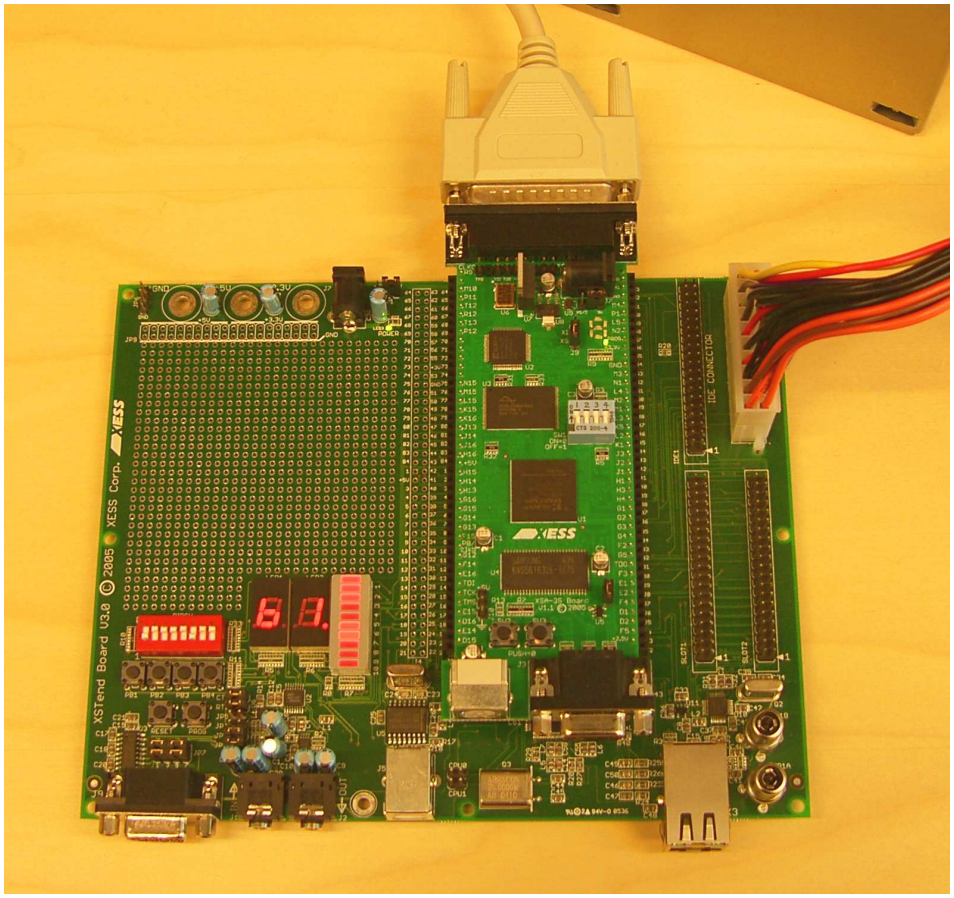


Figure 4.1: The XESS XSA-3S1000 board with the XStend 3.0 add-on board.

provides general purpose leds, which are used to show state information from within the co-processor.

Figure 4.1 [1] shows both of the XESS boars connected together. The XSA 3S1000 board is placed on top of the XStend 3.0 board. The FPGA chip is located at the center of the XSA board, just above the "XESS" logo. Below that is the SDRAM chip. Above the FPGA chip are the flash RAM and the CPLD chip, in that order. The power to the system is supplied from a standard ATX power supply, the cabling can be seen on the top right hand corner. The parallel port cable used for configuring the FPGA and for the communication between the co-processor and the CPU can be seen in the top of the figure. When the photograph was taken, the system was config-

[1] For those reading this thesis as an electric copy, please zoom in to the photograph.

ured to show the currently executed instruction on the double seven segment displays. In the figure they show a hexadecimal value of 0xB1, which stands for *return*. The instruction in question is always the last instruction of any Java application that terminates normally.

The XESS 3S1000 is extended using XStend 3.0 add-on board [76]. This board provides a wide spectrum of enhanced capabilities to the system. The added functionality includes a network controller, an USB controller (1.1, slave only), buttons and switches, seven segment displays and various connectors. The USB was considered as the communication medium, but since USB 1.1 is not considerably faster that the parallel port, and the drivers for accessing the USB port on the host side would be much more complicated, it was not adopted. Also the network controller was rejected, due to the packet based communication model used in the ethernet. The communication in the REALJava virtual machine consists of scattered small transmissions rather than few larger packets. The switches are used to select the source of debug data to be shown on the seven segment displays. These allow debugging of each of the subsystems in the co-processor without resynthesis.

The first versions of the REALJava virtual machine were implemented only on this platform. The co-processor originally used the 32 megabytes external memory chip as the local memory of the co-processor. It soon became clear that the size of the local memory is not as important as the speed, and the external memory was replaced with the internal BRAMs of the Spartan3. The memory size was reduced to 49152 bytes, but that has been found to be sufficient to run all of the applications used for the tests and analyses without the need for swapping the stack pages or the method pages to the memory space of the CPU. As a curiosity it is worth mentioning that the first version of the REALJava virtual machine with hardware support and a minimal co-processor implementation on the XESS board took about six hours to run the *Fibonacci* test, which is described in Section 6.2. The final system Running on the ML410 takes only 433 milliseconds for the same task.

4.2.2 Xilinx ML310 and ML410

The larger prototype was in the beginning based on a Xilinx ML310 demonstration board [79]. This board provides all the features and services one might expect of a desktop computer, such as a network controller, an IDE hard drive controller, PCI busses and so on. The FPGA chip is a Virtex2Pro30 [80, 81], which includes two PowerPC CPUs. The co-processor is connected to the CPU via the Processor Local Bus (PLB) as shown in Figure 4.2. This setup improved the communication speed to about 12

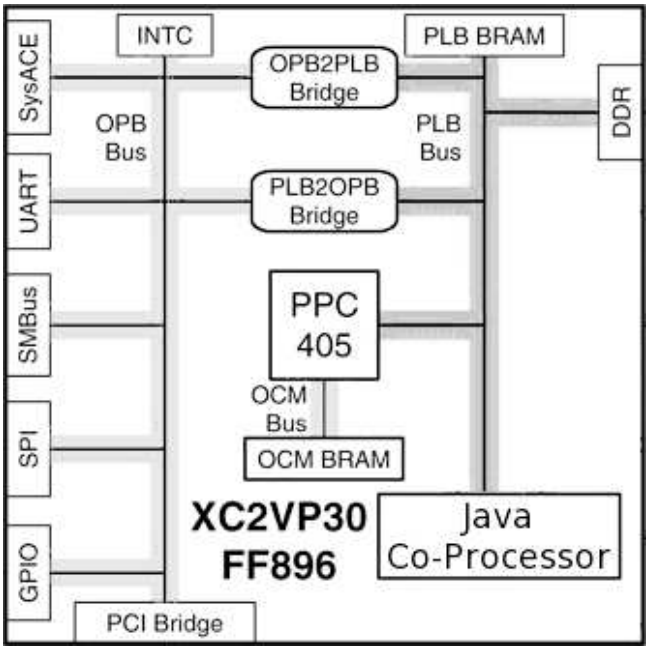


Figure 4.2: Bus structure and co-processor placement on the Virtex 2Pro. The structure is similar on the Virtex4, with the addition of the network controller on the PLB bus.

megabytes per second, including the time to fetch or store the data at the CPU. The maximum peak transfer rate without the fetching and storing was measured to be around 30 megabytes per second. The Virtex2Pro offers two other busses, and they were considered also. The On-Chip Peripheral Bus (OPB) was discarded as it is slower than the PLB (OPB runs at 33 MHZ, compared to 100 MHz for the PLB). The OPB is subordinate to the PLB, making it obvious that the transfer rates must be inferior. The Device Control Register (DCR) was a bit harder to ignore, as it provided a maximum peak transfer rate of 170 megabytes per second. This speed was attained by placing the measurement loop inside the kernel driver required to access the DCR. When the measuring loop was placed in a user mode code section, the transfer rate dropped radically. This drop was introduced because the operating system had to switch to the kernel mode before every transfer, and back to the user mode right after completion. The visit to the kernel mode was measured to cost about 700 clock cycles. As the communication pattern between the co-processor and the CPU was already characterized using the smaller board, it was easy to see that the relatively small bursts of data would not warrant the use of higher bandwidth at the cost of the added latency. Similar analysis is required, when selecting a communication medium for a completely new system. Especially if the new system is going

to use one of the currently popular network-on-chip (NoC) approaches, the characteristics of the data traffic should be carefully considered.

In later versions, starting from version 2.00, the ML310 board was replaced by a newer version, namely the ML410 [82]. This board provides basically the same resources as the ML310, but the FPGA chip is a Virtex4FX60 [83, 84]. The newer FPGA chip provides twice the amount of lookup tables (LUT), and the board has a faster external memory module. The ML310 has a DDR type memory module while the ML410 has DDR2 type memory. The increased memory bandwidth can clearly be seen in the results for the REALJava versions 1.01 and 2.00, which have no other differences save for the upgraded platform. The results for Kaffe also show the impact of the faster memory. Naturally the impact is slightly smaller in the hardware assisted execution engine, since the amount of memory accesses targeting the physical system memory is reduced when the co-processor is used.

The actual core of the co-processor is almost exactly the same as in the smaller prototype. The only difference is in the integer multiplication, which takes one clock cycle for the Virtex FPGAs and requires two cycles when implemented on the Spartan3. In the versions of the REALJava virtual machine starting from 1.01 the co-processor cores are identical. This was achieved by reducing the clock rate for the XESS board. Both of the Virtex based systems run the core at 100 MHz, while the Spartan3 was configured to run the core at 66MHz. The PowerPC CPUs in the ML310 and the ML410 run at 300 MHz. One of the CPUs runs Linux 2.4.20 as the operating system providing services (network, filesystem, etc.) to the virtual machine and also the software partition of the REALJava virtual machine. The other PowerPC CPU core is not connected to the rest of the system, and remains idle.

The ML310 and ML410 boards contain a Xilinx ACE-controller, which is used to select the device configuration when the board is powered up. This controller communicates with the outside world via a serial port, and upon receiving a valid configuration number it loads the specified ace-file. An ace file typically contains a bit-file used to configure the FPGA device on the board and executable software which is stored to the main memory. The ace-files are stored to a CompactFlash memory card connected to the ACE-controller. The CompactFlash card is partitioned to have a FAT type partition as the first partition. This partition contains a directory structure which is used to select the ace-file for each configuration. The ace-files in that partition can be updated using any normal CompactFlash card reader or by running a configuration with Linux and support for the ACE-controller

and then updating the data from the Linux running on the same board. The latter approach is used, as it allows new configurations to be entered without removing the CompactFlash card.

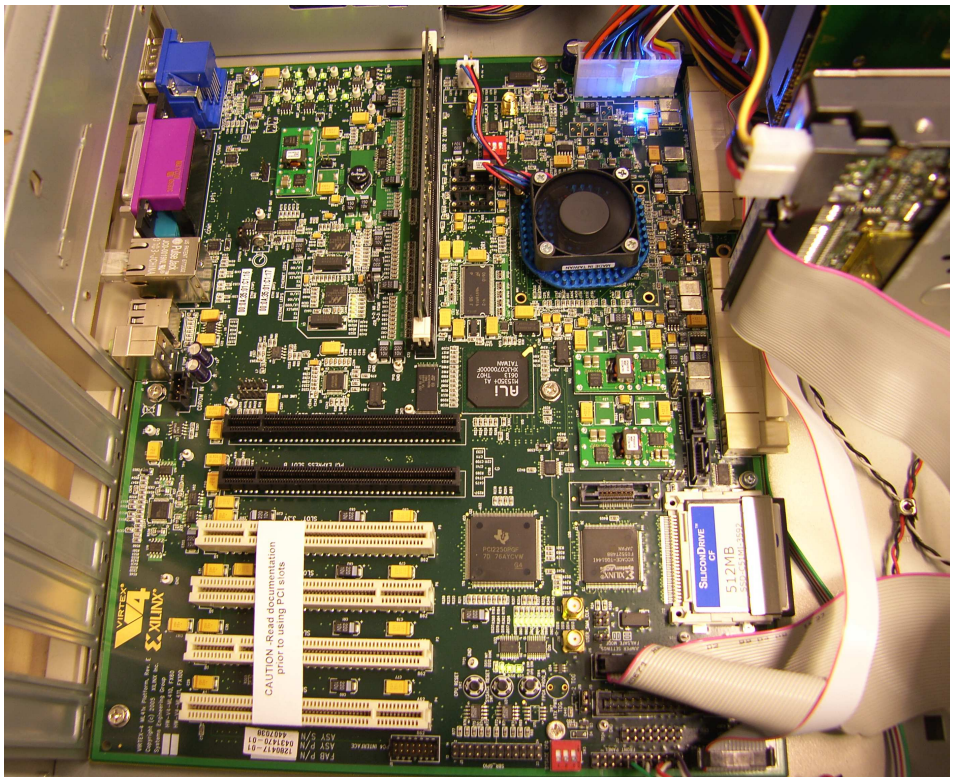


Figure 4.3: The ML410 board.

Figure 4.3 shows the ML410 board placed in a standard ATX casing. The Virtex4 FPGA is located at the top of the photograph, covered by a heat sink and a fan. Below that, and slightly to the left, is a chip with the "ALi" logo. That is the ALi M1535D+ south bridge [88]. It provides all the services found in the south bridge of a typical desktop computer, including an IDE controller. In the lower right hand corner is the SiliconDrive compact flash card containing the configurations and the root file system for the Linux. The ACE controller is located to the left of the memory card. The hard drive is located at the right of the photograph, and it is used for user file system. This data area could have been placed on the compact flash card, but to facilitate several parallel versions of the software and to provide several hundreds of megabytes of storage space for traces and log files, the physical hard drive was chosen. Finally, the DDR2 memory module is located above the PCI Express connectors, left of the FPGA chip. The

memory is 256 megabytes in size. When the ML410 is configured with the system containing the REALJava virtual machine and the software components required for the Linux operating system, it is a fully autonomous embedded system. The communication with the system is established via ethernet, using simple telnet connection over ethernet. The user logs in to the system, just like to any desktop computer running Linux, and is the able to start the REALJava virtual machine from the command line. Both the hardware accelerated and the software only versions are available. Also the Kaffe virtual machine can be used to execute Java applications. This setup makes running comparisons very easy and provides meaningful benchmarks, since the underlying system for all three virtual machines is exactly the same.

The PowerPC 405 CPU Core

As mentioned before, the CPU integrated on the Xilinx devices is a PowerPC 405 [70]. The CPU provides a well rounded set of features one might expect to find in the CPU of a modern embedded system. These features include caches for both the data and the instructions, memory management unit capable of providing virtual addresses for the software and moderate length pipeline with five stages. While running at 300 MHz, the performance of the CPU is roughly of the same level as the CPUs found in embedded systems currently.

The PowerPC 405 CPU can be implemented in various configurations. The parameters listed here are the ones used in the Xilinx implementation. Since the CPU is implemented as a dedicated hardware block, most of these parameters cannot be changed, only some of the functionality can be disabled via internal control registers in the CPU core. The caches, for both the data and the instructions, are 16 kilobytes in size with two way set associative structure. The cachelines are eight words long (word being 32 bits long). The caches are non-blocking during line fills (both) and flushes (data cache only). Static branch prediction is included in the CPU. The pipeline is five stages long, with single cycle execution of most instructions, including loads and stores. The pipeline stages are fetch, decode, execute, write-back, and load write-back. The CPU also provides hardware multiply and divide for faster integer arithmetics (4 cycles for a multiplication, 35 cycles for a division), but it does not provide floating point arithmetics in hardware. The lack of floating point arithmetics applies to both single and double precision instructions. All of the floating point arithmetic instructions are preformed via software emulation. Two addressing modes are supported, real and virtual. The virtual mode allows addressing up to 4 Gb (32-bit address space limit). The CPU has a single-issue execution unit containing

the general-purpose register file (GPR), arithmetic-logic unit (ALU), and the multiply-accumulate unit (MAC). The GPR consist of thirty-two 32-bit registers that are accessed by the execute unit using three read ports and two write ports. During the decode stage, data is read out of the GPR for use by the execute unit. During the write-back stage, results are written to the GPR. The CPU also offers a variety of tracing and debugging facilities, but since they have not been used during the implementation of the REAL-Java virtual machine, they are not discussed in detail.

4.3 FPGA Techniques and Tools Used

The co-processor core has been designed using Xilinx ISE tool set [85, 86]. Versions of the ISE have ranged from 7.1i to 9.1i during the design of the system. The ISE is also used for the full implementation of the XESS based system. The ML310 and ML410 boards require more attention in order to get all the features and services properly configured and the Linux kernel running on the PowerPC CPU core. The flow starts with generating a netlist of the co-processor core in the ISE. Then this netlist is imported to the EDK tool [87], with all the other required IP-blocks, such as the PLB bus and the Xilinx ethernet controller for the ML410. All of the blocks are connected and their address ranges specified inside the EDK. Also the PowerPC processors receive their startup programs in this phase. Both of the PowerPC cores are assigned a bootloop, which basically keeps the processor in a tight loop in order to stop the processors from executing invalid code while the memory is being initialized. The EDK tool invokes ISE for the synthesis and for the place and route to generate a netlist and placement information for all of the logic in the system. Then a bit-file is produced for configuring the FPGA device. This is updated to contain the instruction streams of the bootloops for both PowerPC cores. Finally the file is merged with the executable Linux kernel to produce the ace-file which is transferred to the board. As mentioned earlier, the Xilinx ACE-controller is used for actually loading the required data to the FPGA and to the memories.

The co-processor is designed using "good coding style", as presented in [65, 66]. This means that there are no latches in the design and all of the signals that are delivered to several subsystems are registered. FPGA technology is in general suited for synchronous design only, and latches, on the other hand, are more commonly found in asynchronous designs. Also no VHDL variables are used as they tend to cause problems in synthesis. The documentation of Xilinx tools and several Xilinx white papers, such as

[8, 18] [2], also provided coding style tips. These coding techniques provide a more stable and predictable implementation on an FPGA.

Some of the subsystems were generated automatically using the Xilinx CoreGenerator. This method was applied to the memories, which were implemented using BlockRAMs, and to a few arithmetic units. All of the implemented floating point operations were included as generated cores. Also the integer division was implemented using the CoreGenerator. The amount of local memory in the Spartan3 based version is limited to 12288 32-bit words. This uses 22 of the 24 BlockRAMs available. In the versions with Virtex FPGAs the amount of memory is increased to 32768 words, which uses 58 out of the 232 [3] available BlockRAMs.

The integer multiplication was implemented using the dedicated multipliers in Xilinx FPGAs. In the Spartan3 and the Virtex2Pro the multiplier module is called MULT18X18, which performs 18 bits wide multiplication, while in the Virtex4 the dedicated multiplication module is called DSP48 and it includes support for multiply-accumulate structures, with otherwise the same functionality. The DSP48 units are used in compatibility mode, ignoring the added functionality. The *imul* instruction requires 3 dedicated multipliers to produce a 32-bit result from two 32-bit inputs. It is worth noting that Java specifies the instruction to produce 32-bit result, so that the lowest 32 bits are returned and the rest are ignored without a warning or any kind of overflow signal or exception. Four more of the multiplier units are used for the floating point multiplication (*fmul*). All together seven multiplication units are required.

4.3.1 Size of the Co-Processor Core

This section provides details on the physical size of the REALJava co-processor core. The statistics are split into subsections to provide insight into the distribution of logic resources in the various subsystems. The size is compared with several other execution engines, including full processors and co-processors, some targeting Java and others being general purpose processors.

[2] These two white papers contributed to the design by suggesting that as few as possible of the registers are connected to the global reset and that the global reset should be synchronous. All the other white papers that provided insight to the Xilinx specific coding styles are not referenced, they can be found at the Xilinx website [108].

[3] There are 232 BRAMs in the Virtex4FX60 and 136 in the Virtex2Pro30.

Logic and Resource Utilization

All together 8357 LUTs are used for the REALJava co-processor when implemented on the Spartan3 1000 FPGA chip found on the XESS board. Besides the co-processor core, this includes the communication module, reset and clock generation and supporting logic used for resetting the auxiliary chips on the boards. The debugging registers and counters were temporarily removed to evaluate size of the core. The debugging facilities include the drivers for the seven segment displays and several counters and timers. These counters keep track of the stack usage, the method space size, live time of the co-processor, the number of traps and four previously executed instructions. The reduction in the amount of LUTs achieved by the removal of the debug logic was relatively small, just under 300 LUTs.

Besides the LUTs, the system also requires other resources, which are listed in Table 4.1. The BRAMs row refers to the BlockRAMs mentioned earlier, while the MULT18X18s row reports the number of the multiplication modules. The amount of multiplication units used is the same for all three FPGAs used, even though the Virtex4 series uses the newer architecture for the units. The GCLKs row lists the number of the global clock buffers. Finally the DCMs row tells the number of the digital clock managers used by the design. The DCM is used to generate clock signals, while the GCLKs are used to deliver the signals to the logic cells that require it. In the XESS board version the DCM is used to divide the incoming 100 MHz clock signal by three and then multiply it by two, resulting in the 66 MHz clock used internally.

Resource	Used	Utilization
Flip Flops	4782	31%
4 input LUTs	8357	54%
as logic	8095	
as route-thru	172	
as shift registers	90	
BRAMs	22	91%
MULT18X18s	7	29%
GCLKs	2	25%
DCMs	1	25%

Table 4.1: FPGA resource usage of the REALJava co-processor. The utilization shows the percentage of the resources available on the Spartan3 1000.

The size is reasonable, in comparison with other co-processors and processors implemented with FPGAs. The co-designed Java virtual machine from Kent et al. [22] is the first reference system. The authors have not named their system, so it will be referred to as the Kent system. The co-processor requires a minimum of 26898 logic elements when implemented on an Altera Stratix FPGA. The co-processor achieves around 25 MHz clock frequency with small variance due to the configuration of the co-processor. The Altera tools report a logic element count, which is similar to the LUT count reported by Xilinx tools. Altera logic element contains one LUT and associated register and support circuitry. The jHISC [59] is a standalone processor optimized for object oriented operations. The processor core uses 15573 LUTs and runs at 30 MHz. The Milk [7] is a floating point co-processor which is reported to require 20000 logic elements when implemented on an Altera Stratix FPGA. In that configuration the co-processor runs at 67MHz. The Tensilica Xtensa customizable processor [71] is reported to use between 6166 and 14811 slices when implemented on a Virtex-II series device. Here, the comparison to the REALJava is hard to make, because even though it is known that one slice contains two LUTs, things are not quite so straight forward as multiplying the number of slices by two. This is because the Xilinx tools do not use both LUTs in a slice unless the two LUTs are deemed to be related logic. A LUT without a related pair occupies a slice all by itself, unless the device is too full. Only in that case the tools start packing unrelated logic to the slices. It is safe to say that the Xtensa takes at least 6166 LUTs (minimum configuration and no related logic at all) and at maximum 29622 LUTs (Maximum configuration and all slices contain related logic). The PowerPC 405 can be implemented as a soft core, and in that case it uses 33840 LUTs. The performance is very limited, since the soft core version can only run at 25 MHz. Xilinx also has a smaller soft core processor, the MicroBlaze, which takes only 2120 LUTs on a Virtex-II device, while running at 100 MHz. LEON2 [73] is a 32-bit processor core with SPARCV8 architecture, using from 5000 up to 15000 LUTs. The LUT counts and clock frequencies for these systems and the REALJava co-processor are shown in Table 4.2. The data for the Xtensa variants, the PowerPC and the MicroBlaze are from [72]. The data for the LEON variants is from [3] and [17]. In general the comparison of sizes based on LUTs, logic elements and slices is far from accurate, as the metrics are not uniform and the scores depend highly on the implementation options and also on the routing. Thus the numerical values are to be taken as rough estimates rather than strict facts.

(Co-)Processor	LUTs	MHz
REALJava	8389	100
Kent system (small)	26898	25.18
Kent system (large)	34471	23.78
jHisc	15573	30
Milk	20000	67
Xtensa (small)	12332	33
Xtensa (large)	29622	29
PowerPC	33840	25
MicroBlaze (+FPU)	2120	100
LEON2 (small)	5000	60-80
LEON2 (large)	15000	60-80

Table 4.2: FPGA logic usage of various processors and co-processors.

Logic Distribution Among Subsystems

The various subsystems in the REALJava co-processor were declared as "partitions" in the Xilinx ISE in order to find out their individual logic usages. The results are listed in Table 4.3. The total number of LUTs clearly exceeds the number presented earlier. This is caused by the partition declarations, which rule out optimization across partition barriers. The partitioning was done along design units, namely the VHDL files that provide the code for the various blocks. This means that the partitions are not pipeline stages and partitions can contain code for non-consecutive pipeline stages. As an example, the memory controller unit mentioned here contains the logic needed for stack handling, instruction stream fetching and local variable access. The ALU contains also the decode stage of the pipeline and the partial folding mentioned earlier. The registers, method invoker and constant caches are discussed in detail in Chapter 5.

4.4 Communication

This section describes the communication schemes used in the FPGA platforms. The communication modules in hardware partition of the virtual machine have exactly the same internal interface for each of the platforms. This allows porting of the virtual machine to new environments and bus structures so that only the communication module needs to be redesigned. A communication module and protocol [52] were designed for the pipelined bus [37] by Liljeberg et al. also, but since that platform is not used with the prototype, it is not discussed further. Technically, a communication module

Unit	LUTs	% of the total
Memory controller	1523	16
ALU	2897	30
+ Integer division	500	5
+ Shifters	459	5
+ Floating point unit	1303	13
Registers	1111	11
Method invoker	494	5
Constant caches	612	6
Communication	527	5
Clock, reset and I/O	358	4
Total	9784	100

Table 4.3: FPGA logic usage by subsystems.

can be easily designed for any bus or network-on-chip structure. The physical communication subsystem is required to satisfy two conditions, namely: 1) the datagrams must arrive in their destination in the same order that they were sent, and 2) the datagrams arriving from two different sources to a same destination must be identifiable. The first property can be achieved with a lower level network protocol, like the ATM adaptation layer 5 (AAL5) [26] used for internet protocol (IP) communication, or by the physical structure of bus. If the communication protocol is responsible for the order of the datagrams, the actual data can arrive to the module in any order, and the data items are reordered before passing them to the co-processor. The second property is quite natural, and should be present in all communication solutions, either in the structure or as a part of the protocol.

4.4.1 XESS Board with Parallel Port Communication

The XESS board uses standard parallel port as the communication medium. The data channel from the CPU to the co-processor is 8 bits wide while the channel from the co-processor to the CPU is 4 bits wide. Both of these channels are synchronized to a bus clock signal produced by the host computer. This is not the fastest possible way to communicate over the parallel port, but this provides good enough performance for the testing purposes stated previously. The protocol is quite simple, with three data types defined. The types are command, address and data. The commands are 8 bits long, and the meaning of each bit is listed in Table 4.4. Communication starts with the CPU sending a command, and if the command is a data transfer (read or write) then the address. If the command is a write, then the data is trans-

ferred to the device, and in case of a read, the data is transferred from the device. If the burst mode bit is set, then the address in the communication module is automatically incremented, and the next data item is transferred without sending the new address. Since the addresses used here are defined to be 24 bits long, the burst mode saves three bytes for each transfer operation. Considering that the data items to be transferred are always 32 bits long, the 8 bits long commands and the communication channel widths, the total number of communication cycles per transfer is eight for a write and 12 for a read without the burst mode and five and nine with the burst mode, in the same order.

Bit	Command	Description
0	Burst	Enables burst mode
1	SingleStep	Execute only one command at a time
2	IrqReply	Signals the JPU that the IRQ has been handled
3	JPU/#MEM	Target address in JPU register space
4	Write	Write data
5	Read	Read data
6	Halt	Place the JPU in halt mode
7	Continue	Enable execution

Table 4.4: Communication command bits.

Naturally the communication scheme for the XESS board is the same whether Windows or Linux is used as the host operating system. The only differences are in the software communication module, which calls different functions to physically access the parallel port. The protocol is exactly the same, and the co-processor does not even need to know which operating system is used in the host computer.

4.4.2 ML310 and ML410 with PLB Communication

The ML310 and ML410 boards use the processor local bus (PLB) as the communication medium. The PLB is part of the CoreConnect [114] standard developed by IBM. The CoreConnect is a microprocessor bus architecture for SoC designs. It is designed to ease the integration and reuse of processor, system and peripheral cores within standard and custom SoC designs. Elements of this architecture include the processor local bus (PLB), the on-chip peripheral bus (OPB), bus bridges, and the device control register (DCR) bus. High-performance peripherals connect to the high-bandwidth, low-latency PLB. Slower peripheral cores can be connected to the OPB,

in order to reduce the traffic and capacitive load on the PLB. The CoreConnect architecture also supports other bus standards, like the Advanced Microcontroller Bus Architecture (AMBA) [115], by providing bridges that allow data to pass from one bus type to the other.

The CoreConnect is available as a no-fee, no-royalty architecture to tool vendors, IP core companies and chip development companies. According to IBM [89], it is licensed by over 1500 electronics companies, including Cadence, Ericsson, Lucent, Nokia, Siemens and Synopsys. Adopting such a widely used bus architecture helps in integrating the REALJava virtual machine to other systems in the future.

The CoreConnect is an integral part of IBM's Power Architecture offering and is used extensively in their PowerPC 4xx based designs. Xilinx uses CoreConnect as the infrastructure for all of their embedded processor reference designs including MicroBlaze based systems.

The protocol used by the REALJava virtual machine for the PLB communication is a simplified version of the protocol for the parallel port. Since the PLB provides read and write request signals, no commands are needed for them. Also the burst mode is not meaningful, since the PLB always provides the address for every transaction. The JPU/#MEM signal is replaced by dividing the address space of the co-processor in two parts, namely one for the local memory and the other for the internal registers. The remaining command bits (SingleStep, IrqReply, Halt and Continue) remain in their original locations.

4.4.3 Internal Interface

Both of the communication systems have identical internal interface between the co-processor core and the communication module. If other busses should be required, they too would replicate the internal interface. The purpose is to keep the co-processor core separated from the communication module in order to keep the core easily portable to other systems.

The connections are shown in Figure 4.4. The communication module is connected to the internal registers and to the memory controller. Also system wide IRQ and Halt signals are connected to the communication module. The connection to the register bank allows accessing the internal registers, but also a set of helper addresses. These are not ordinary registers, but rather data request forwarders. As examples the addresses 64, 66, and 255 operate on the top of the stack, the instruction stream parameter and the

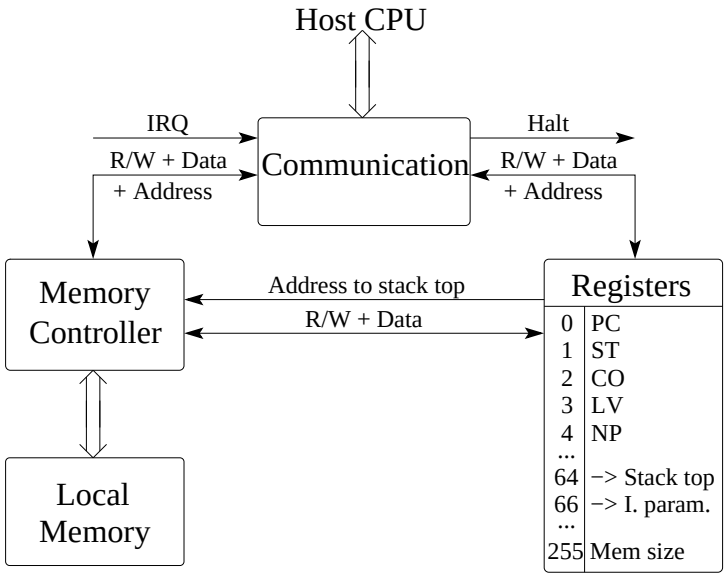


Figure 4.4: The internal interface to the communication module.

size of the local memory, in that order. The functionality and other details of these and the other implemented helper addresses will be discussed in more detail in Chapter 5.

4.5 Memory Areas of the Prototypes

Table 4.5 shows the various memory areas in the prototypes. The areas were described in Section 3.5. The table shows the memory domain of each area and also the initial sizes of the areas. The initial sizes for the areas in the local memory are one half of the total size of the local memory for each. This translates to 24576 bytes in the XESS version and to 65536 bytes in the larger boards. The software only version uses the same sizes as the larger boards. The areas which are collected for garbage are identified in the GC column. The grows column shows whether the area can grow if required. Finally the last column describes the data items placed in the areas. No addresses are shown, since the system memory is provided by the operating system, and thus uses virtual addresses. In the local memory the method area starts from the highest address and the stack starts from the lowest address. The initial sizes for the areas stored in the system memory are variables in the software partition, and they can be changed to suit the demands of a given application and the limitations of a given system. The sizes reported in the table are the values used in the prototypes of the RE-

ALJava. The initial size of the heap is chosen to be 16 Mb since the Kaffe used in the comparisons in Chapter 6 uses 16 Mb as the default size of the heap. Using the same initial size keeps the comparisons on a level playing field, since garbage collection is forced at the same times. The sizes are also reasonable when comparing them to the 64 Mb available to the operating system on the larger boards. For simplicity the sizes are the same on the desktop computer versions of the REALJava. The code segment size is measured on the ML410 board, and it includes the executable software partition, which is 330 kb in size, and the libraries, which take 170 kb. The size of this area is platform dependent, since the software is compiled for a given host CPU, and the size is effected by the instruction set of the CPU as well as by the compilers abilities to perform optimization for the architecture in question. The swap area is not shown in the table, since it is only created if needed. The initial size of the swap area would depend on the size of the region to be swapped.

Area	Memory	Size	GC	Grows	Data items
Heap	System	16 Mb	Yes	Yes	Java objects
Reference table	System	1 Mb	No	Yes	References to the Java objects
Static members	System	0	No	Yes	Static class members
Class area	System	4 kb	No	Yes	Methods and constant pools
String area	System	0	No	Yes	String data
Code segment	System	500 kb	No	No	Executable code segment of the software partition and the required libraries
Method area	Local	1/2	No	Yes	Method code segments
Stack	Local	1/2	No	Yes	Stack frames

Table 4.5: Memory areas of the prototypes. The sizes of the areas in the local memory are initially set to one half of the total amount of the local memory.

4.6 Chapter Summary

Based on the conceptual model shown in Chapter 3, and the generic principles of the Java technology outlined in the Chapter 2, the REALJava virtual machine was prototyped using FPGA platforms. The structure of the coprocessor was modified to suit the properties of the FPGA technology. The physical resources needed for the prototype were listed and analyzed with several other systems as reference points. The FPGA platforms were described, and the communication schemes used for each platform were also detailed.

Chapter 5

Performance Increasing Techniques

This chapter details the techniques developed and applied to increase the performance of the REALJava virtual machine. Before going into the actual performance boosting techniques, the strategies employed to identify the performance hindrances are explained. Each of the improvements presented is motivated by an inefficiency found using these strategies. First a simple but effective stack cache architecture is presented. It is followed by details of the pipeline structure tailored to the modified stack subsystem. Then the Java method invocation is enhanced by employing caching of the required data with the stack frame initialization performed in hardware. As some of the values retrieved from the CPU are constants, caching systems for these are shown next. The method and constant caches are deployed in order to increase data locality and to reduce the number of communications by reducing the number of traps generated by a given application. The possibilities for hardware level control in the time domain are outlined, with focus on the thread scheduling system, which uses hardware timers for time slice preempting. Finally an extended register map is presented, with emphasis on the additional functionality provided via the register address space. The multicore approach is very briefly mentioned. The performance increasing techniques are evaluated for their effectiveness by showing measurement results before and after a given technique was applied. More measurements are available in the Appendix B, and the corresponding version history is given in the Appendix C.

5.1 Identifying the Performance Hindrances

In order to find out the bottlenecks in the REALJava virtual machine several techniques were used. Most of the bottlenecks were identified by running several Java applications on the REALJava and on Kaffe. The results from the two virtual machines were then compared to find applications that show relatively poor performance [1] for the REALJava. These applications were then further analyzed to find out the reasons for the poor performance.

In the first level of analysis the applications were simply disassembled using a Java disassembler called Jad [116]. The disassembled listings provided some insight to the operation of an application, but the contribution of the methods imported from the standard library was not shown in the listings. Also the number of times a given instruction was executed had to be deduced. For these reasons more accurate analysis techniques were required.

The most detailed analyses were performed by creating full instruction traces of the execution of the application in question. These traces were created with the software version of the REALJava virtual machine. The software version was used because the disk accesses are very much faster on the desktop computer than on the ML310 or the ML410. The software virtual machine's execution engine was configured to write every instruction to a file, while it was being executed. It should be said, that the execution in REALJava progresses along exactly the same path, regardless of whether the execution is performed on the co-processor or on the software. Thus the trace files were valid for both of the execution engines. This technique provided the dynamic instruction counts, representing the actual number of executions for each instruction, including the effect of the standard library.

The amount and type of trapped instructions was also analyzed. To this end a counter register was integrated to the co-processor core. Besides just counting the amount of the traps, the instructions causing them were traced. This provided profiles of the trap behavior for the applications. From the data gathered with this technique some of the bottlenecks were identified.

Besides the instruction traces, also other data items were traced to a file in similar fashion. These includes the names of the methods to be invoked and the values of the constants to be loaded. Since the first traces indicated that the method invocations were a possible cause for relatively low performance in some of the applications, some profiling of the invocations was required in order to develop new techniques to improve the performance.

[1] The "relatively poor performance" means that a given application was not as many times faster as the other applications.

The applications were also analyzed based on the percentage of time the co-processor was executing the application. This was measured by measuring the total time for a given application and the actual "live time" in the co-processor. The values showed, that some applications achieved more than 90% of hardware execution, while others were initially below 10%. This analysis provided insight to the distribution of the instructions in the time domain, instead of just the number of executions.

Several Java applications were created for the analyses presented above. Some of them are described later with the performance evaluations of the final version of the REALJava virtual machine. The others were mainly very short applications aimed to pinpoint certain inefficiencies. These applications are not described in detail, since they were used for very strictly specified purposes. As an example one such application is described. To see the effect of the instruction stream parameter loading, an application with three tests was designed. The tests comprised the same loop, with only one difference between them. The first loop used a local variable for the arithmetics, the second loop used a constant in the 8-bit range and finally the third used a constant in the 16-bit range. The instructions fetch zero, one or two bytes of parameters, respectively. By measuring the execution times for each of the loops, it was possible to calculate the effect of fetching the parameters from the instruction stream. Similar constructs were used for other cases, like conditional jumps taken or not taken, different forms of method invocations, temporary data saved to the local variable or to the stack, variations in the order of instructions causing traps and many others. Practically the whole instruction set was tested and compared to the other execution engines available.

Also the software partition of the REALJava virtual machine contributed to the identification of the performance hindrances. The software was analyzed to find out possibilities for moving some of the functionality related to a given instruction to the hardware. This technique was used for instructions that have some components that need to be performed on the software. With this technique the computational load experienced by the CPU of the related instructions was minimized. This was achieved by moving as much as possible of the basic functionality to the hardware domain. As an example, the push and pop registers were designed to remove unnecessary software computation of the new stack top pointer.

Finally some of the performance increasing techniques were discovered during the prototyping of other techniques. This means that while designing the required facilities for a given acceleration technique, the possibility of

using the same facilities for other cases also was discovered. Additionally some modifications changed the structure so as to allow deeper or more aggressive pipelining.

In order to compare and analyze the performance of the REALJava virtual machine in respect to other virtual machines, a database was created. This database stores the results for all of the tests available. The database is accessible via a net browser, for both updating new results and analyzing the existing results [111]. The user interface allows selecting the systems to be compared and the analysis to be performed. The system can show the results for up to nine virtual machines at once. In the basic view the best and the worst scores are highlighted for clarity. The results can be analyzed to see the relative performance, instead of plain test scores. Additionally the results can be shown in graphical from, in order to visualize the results. The database also served as a form of research journal. Detailed version information has been logged to the database, including the version number, the release date, the description of the major changes and the full set of the results from the tests.

5.2 Stack Handling

The top of the stack is cached, but instead of the large ring buffer suggested in Chapter 3, a four places long linear cache was implemented. This number was chosen based on studies on the code generated by Java compilers. The Sun Java compiler performs practically no optimization on the user code. The only optimization goal is to keep the number of stack locations required by an application at a minimum. This results in code that pumps the stack fast and with small amplitude, first loading data to stack, operating on the data and finally saving the result to local variable. Making the stack cache larger than four places would provide only marginal improvement. Since the maximum number of stack data popped by any given instruction is four, it is also sufficient for the most data hungry instructions.

The linear cache refers to an architecture where the contents of the individual registers are moved up or down along the cache as the cached area moves up or down. This architecture is shown in Figure 5.1. The ring buffer on the other hand refers to an architecture where the data items in the cache remain in place, but the access pointers move up or down. The ring buffer, shown in Figure 5.2, is quite efficient in ASIC designs, but since the architecture incurs a lot of multiplexers, which are expensive in FPGAs, it is not well suited for FPGA based designs. Using the linear approach the

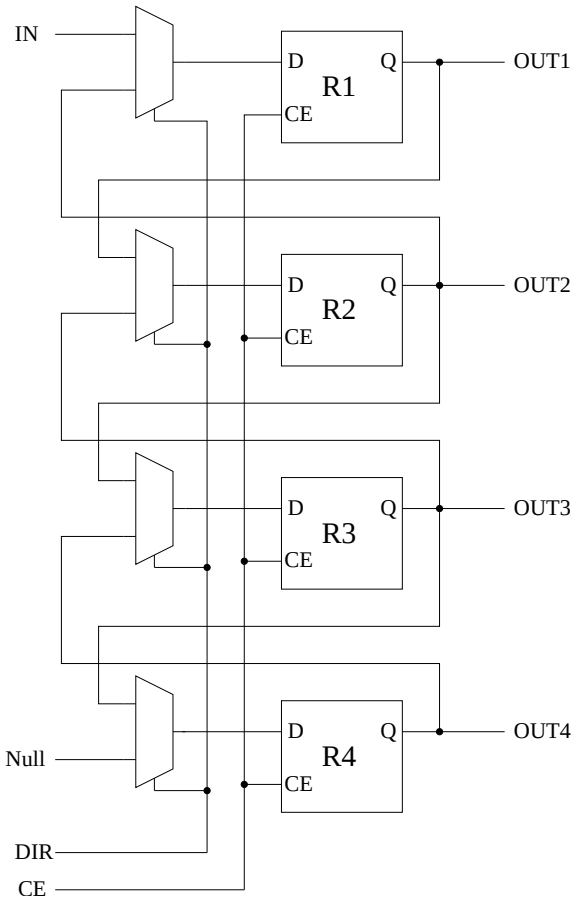


Figure 5.1: Stack cache with linear architecture.

number of multiplexers is one per cache location, regardless of the number of output signals. The direction of the movement is assigned to all of the multiplexers as the selection signal. Also all of the registers share a common clock enable line. In the ring buffer approach the number of multiplexers is highly dependent on the number of outputs, since each output signal requires a multiplexer tree of its own. Naturally the outputs require control signals for selecting the appropriate register, thus increasing the number of registers. At the input side the incoming data can be directly mapped to all registers, but the clock enable signal needs to be decoded from the pointer showing the location of the next free element. In the case of four places the number of multiplexers is three per output port, totaling 12 for four outputs. The linear approach also provides outputs faster after a clock edge. This is because the multiplexing is done before the registers, not after it. The faster outputs provide more time for the ALU to process the data, thus decreasing the minimum clock period.

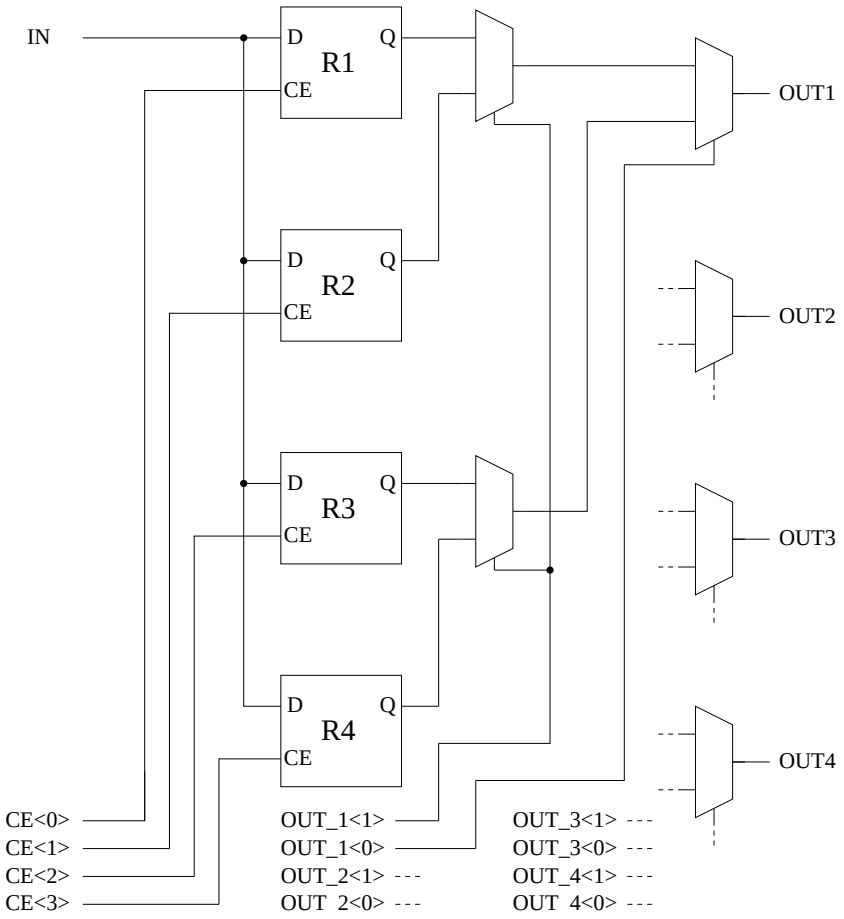


Figure 5.2: Stack cache with ring buffer architecture. Only the output multiplexers connected to OUT1 are shown, similar structures would be needed for each output, with their own control signals.

The data in the stack cache is hardwired to the ALU. This means that the instructions, like *iadd*, which perform an operation on the top stack elements, do not need to actively fetch the data. Rather the data is provided directly to the inputs of the arithmetic unit. In case the stack cache is not valid, a validate request is send to the stack unit. When the stack unit receives the validate request, it validates as many stack top locations as specified by the request. If some of the locations are already valid, they are naturally skipped. After the validation process is completed, the ALU receives an acknowledgment signal from the stack module and is free to continue processing the data. All writes to the stack go through the cache and all the reads come from the cache.

The fact that the topmost stack elements are cached and provided to the ALU gives the REALJava co-processor a clear performance edge over the naive architecture in which every operand has to be retrieved from the memory before an operation can be performed on them. The effect of the caching cannot be demonstrated by the measured results, since the cache was already present in the first version of the prototype. However, the effect of the direct connection to the ALU can be seen by comparing the results of versions 0.06 and 0.08. The connection was introduced in version 0.07 and fine tuned in version 0.08. The performance increase was measured to be roughly 25% in the byte arithmetic tests and just under 20% in the integer tests [2]. The performance increase is shown in Figure 5.3. The aforementioned tests contain one arithmetic instruction inside the test loop with two load instruction before it and one store instruction after it. Since the direct connection to the ALU has no effect on the loads and stores, the performance increase for the arithmetics alone is considerably higher.

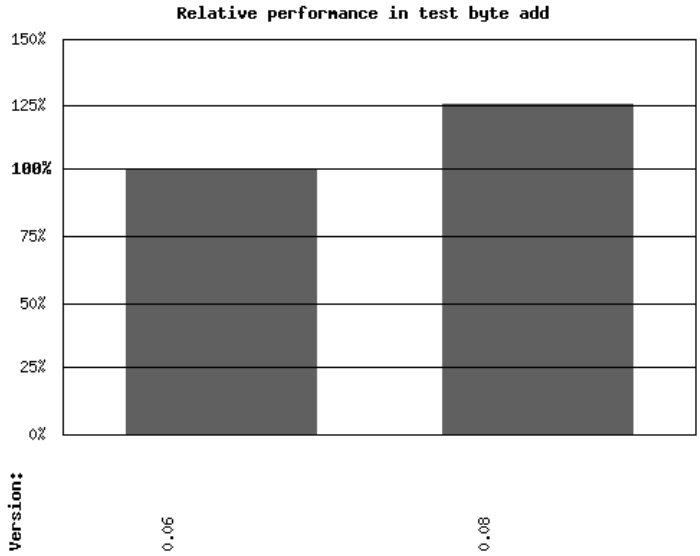


Figure 5.3: The effect of the direct connection from the stack cache to the ALU on the byte arithmetics.

[2]Excluding the divisions, which are multicycle instructions.

5.3 Pipeline Structure

The pipeline structure of the FPGA prototype is relatively similar to the structure presented in Chapter 3. The main differences come from the fact that the instruction folding unit has been omitted. This caused the buffering in the next stage ("fifo and sign extend") to become unnecessary. The sign extending is performed inside the ALU. Also the "operand access" stage becomes obsolete as the folding is not performed for the load type instructions. The duties of the operand access stage are handled by the direct data mapping from the top elements of the stack to the ALU, as discussed in the previous section.

The instruction fetching is changed slightly, since no asynchronous requests for the parameter data in the instruction stream are available. The parameters are instead offered automatically, with valid signals telling the ALU when the parameters are correct for the current instruction. If the parameter data is not already loaded, the instruction fetching unit autonomously fetches it. The fetching is done using the same mechanism that performs instruction prefetching and thus it consumes no additional resources and also it incurs no time penalty. The ALU then notifies the internal register bank to increase the program counter (PC) by the amount of parameter data consumed by the instruction.

The ALU handles also the multicycle instructions, such as integer division. When an instruction requires multiple clock cycles to be completed, the ALU signals the rest of the pipeline stages to stall, if they already have been filled. Empty stages still continue until the pipeline is full.

Since the "operand access" stage was removed and the operands are always presented straight from the top of the stack to the ALU, most of the stack manipulation instructions do not need the ALU to perform their function. These instructions include the local variable loads and stores as well as the simple *pop* and *dup* instructions. These instructions skip the ALU completely, and they are handled directly by the memory controller. In the original architecture a local variable load would have first loaded the data item into the ALU and then saved it to the top of the stack. The modified architecture removes the unnecessary cycle in the ALU and thus speeds up the operation, since the memory controller can perform the data transfer internally. The *dup* type instructions that relocate or duplicate more than one data item need to use the ALU in order to provide temporary storage for the data elements being relocated. The effect of this enhancement is shown in Figure 5.4.

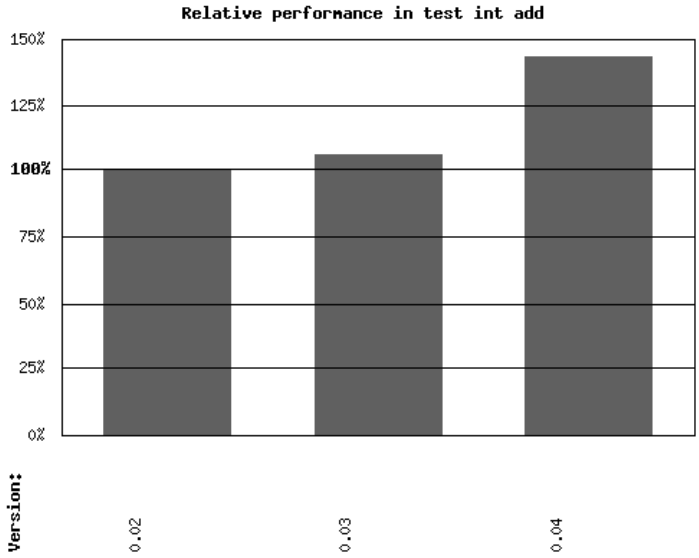


Figure 5.4: The effect of the enhanced stack manipulation on the integer addition test. The enhancement was applied in the version 0.03 and it was fine tuned in the version 0.04.

The resulting pipeline structure is shown in Figure 5.5. The figure also includes the partial folding, which will be discussed in the next section. The figure shows the direct connection between the ALU and the stack, four words from the top of the stack provided automatically. The write connection from the ALU to the stack goes to the top of the stack. The instruction parameter data connection from the instruction fetch stage to the ALU is also shown. The instructions to be handled in the software partition are detected in the decoder, which sends a trap request to the pipeline control unit. The partial folding stage is placed as separate stage, since it redirects the result of the current computation so, that the next computation can proceed with the stack in coherent state. This path is used only if the current computation is followed by a local variable store instruction. Since the FPGAs used for the prototypes have very fast internal memory [3], which is used as the local memory containing the stack and the method area, the cache sizes are set to zero. The caches would be used, if an external or otherwise slower memory would be used. The signals that tell the ALU whether the stack top elements and the instruction parameters are valid are left out of the figure in order to keep it readable.

[3]Single cycle access time.

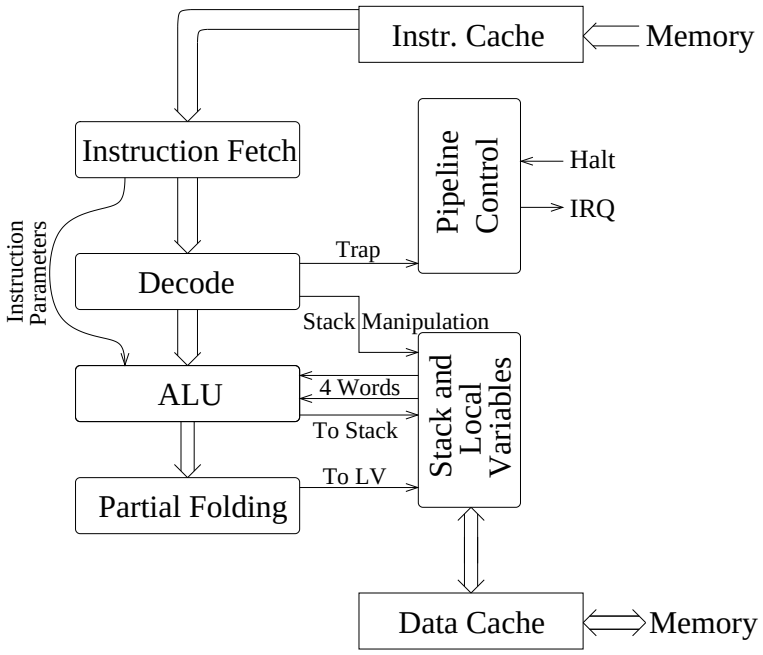


Figure 5.5: The pipeline structure used in the FPGA prototypes, including the partial instruction folding.

5.4 Partial Instruction Folding

The partial folding is performed at the output of the ALU. The partial folding works by identifying instruction sequences that move the result of the current computation into a local variable. In these cases the result is written directly to the local variable in question, without putting it to the top of the stack. In the straight forward implementation the result would be first written to the top of the stack, and then moved to the local variable. Using the classes presented earlier in Chapter 3 in Table 3.2, the partial folding handles sequences "OP1 MEM" and "OP2 MEM". Since the Java compilers typically generate code with small stack space requirements, a lot of the intermediate results in computation are stored to local variables. This causes a significant portion of the arithmetic operations to be followed by a local variable store.

This technique was applied in the version 2.08. The pipelining of the folded local variable stores was included in the version 2.09. The results show that these improved the performance of the integer tests by more than a factor of two. Figure 5.6 shows the results for the integer addition test for the versions in question. In the real life applications the performance increased by more than 22 %, on average.

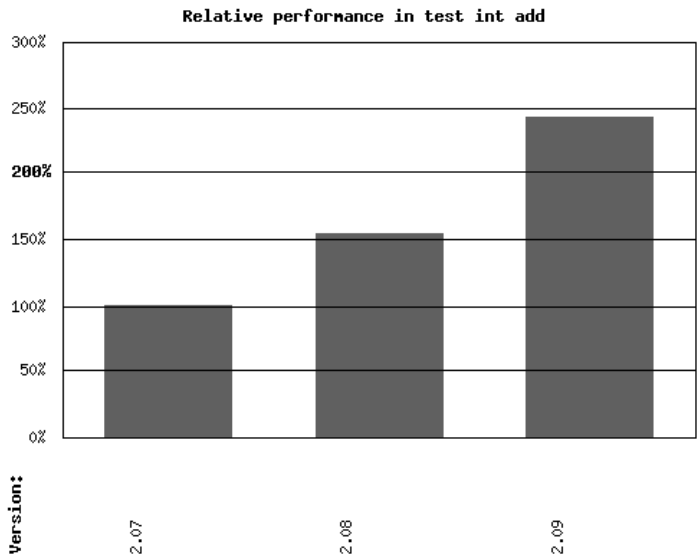


Figure 5.6: The effect of the partial folding on the integer addition test. The folding was applied in the version 2.08 and the pipelining was fine tuned in the version 2.09.

5.5 Method Invocation

Because the method invocations were noticed to be a performance bottleneck, a hardware acceleration scheme [58] for method invocation was designed. The invoker speeds up the invocation of methods that are already loaded to the local memory of the co-processor. The strategy to identify invocations that have happened earlier relies on the reprogramming of the invocation instructions. When a given invoke instruction is first encountered, it performs a lot of mandatory checks. These include finding out if the method in question is already loaded, and if not, then loading the required class. The class loading can, in turn, cause new classes to be loaded. This phase is only necessary when the invocation is executed for the first time. After that the system knows, that the method has been loaded, and no further checks are required. In cases like this it is common to reprogram the instruction to a fast version, that does not perform the checks. When this is done, the hardware can recognize the fast version, and execute the instruction without the checks. The method invocation process is rather complex, and requires several data items from a variety of sources. It also includes the initialization of a new stack frame. A simple MSC of the invo-

cation procedure, as performed in the straight forward technique, is shown in Figure 5.7. Statistics about the stack frame sizes and the lengths of the methods are shown in Table 5.1 along with the amount of invocations performed in the tests [4].

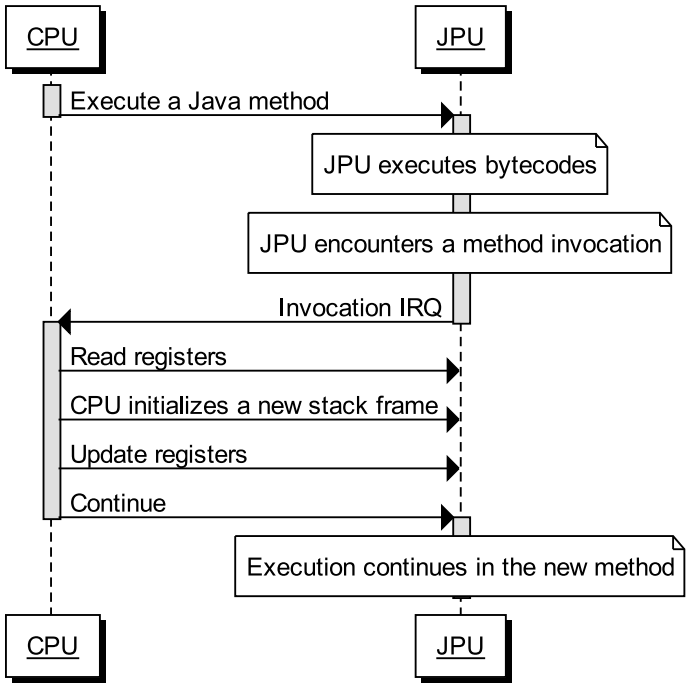


Figure 5.7: A MSC of the straightforward invocation sequence.

	Neural Net	**Salesman**	**Sort**
Stack frame size	11.25	16.95	10.64
Method length	14.29	37.79	7.00
Total invocations	341821	993604	18424832

Table 5.1: Statistics from method invocations in selected benchmarks. The first two rows are averages and they are measured in 32-bit words.

The connections of the invoker module are shown in Figure 5.8. When a fast invocation command is encountered in the ALU, it sends the constant pool index of the method to the invoker module and sets query high. The pool index is used as a method id. At this time the invoker performs a look up in the content addressable memory (CAM) using the method id and the

[4]Descriptions of the tests are given in the Chapter 6.

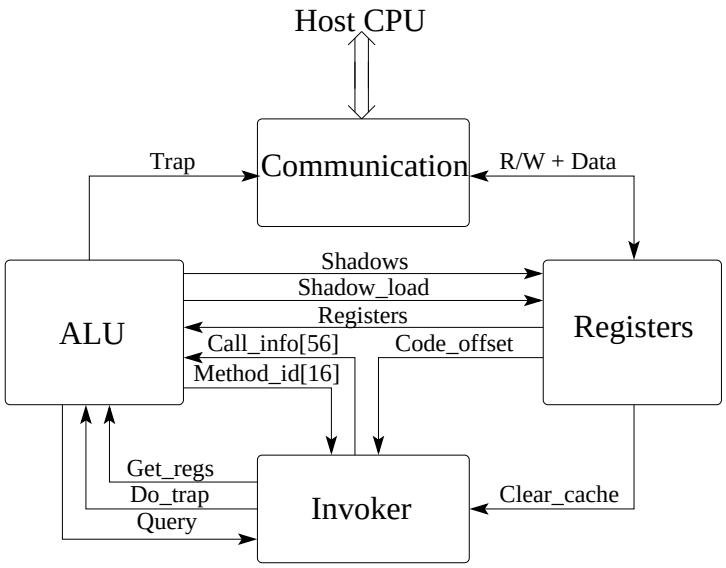


Figure 5.8: The invoker connected to the ALU and the registers.

code offset as the key. The key for the CAM is 40 bits wide, as the code offset is 24 bits and the method index is 16 bits. The code offset is not in byte address space but rather in 32-bit word address space in order to align the data in the memory correctly and save the lower two bits. The offset has been limited to 24 bits, since it is not reasonable to assume that an embedded co-processor core would have more than 64 megabytes of memory locally for each core. As mentioned earlier, the prototypes of the REALJava co-processor core have at maximum 128 kilobytes of local memory, and in all of the tests the system has not been forced to swap out anything.

After the key has been found in the CAM, the match address is sent to normal RAM, as shown in Figure 5.9. The RAM stores the information needed to perform the method call. This RAM is 56 bits wide, and consists

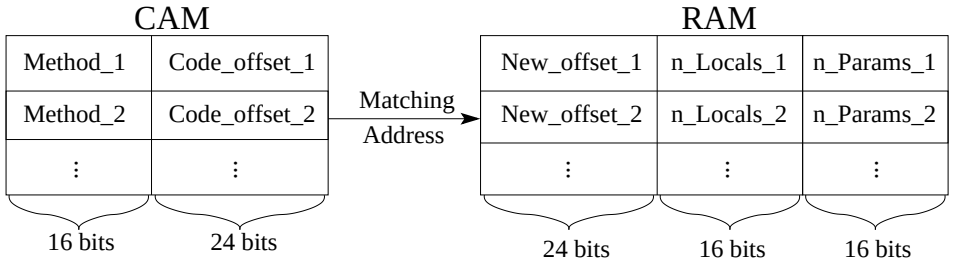


Figure 5.9: The invoker CAM and RAM structure.

of 24 bits for the code offset of the new method, 16 bits for the number of local variables and finally 16 bits for the number of parameters taken by the new method. These are sent to the ALU with get_regs high to indicate a valid match. The ALU then moves old register values to the local memory and calculates new register values using the following rules. The PC_{new} is always 0, since the execution of a new method starts at the beginning. The ST_{new} is counted as ST_{old}-NP_{new}+NL_{new}+5. The 5 is added to make room for the return information on the stack frame. The CO_{new} and LO_{new} come directly from the invoker module. The LV_{new} is counted as ST_{old}-NP_{new}. Then the computation resumes from the beginning of the new method. The behavior of the stack during method invocation is shown in Figure 5.10.

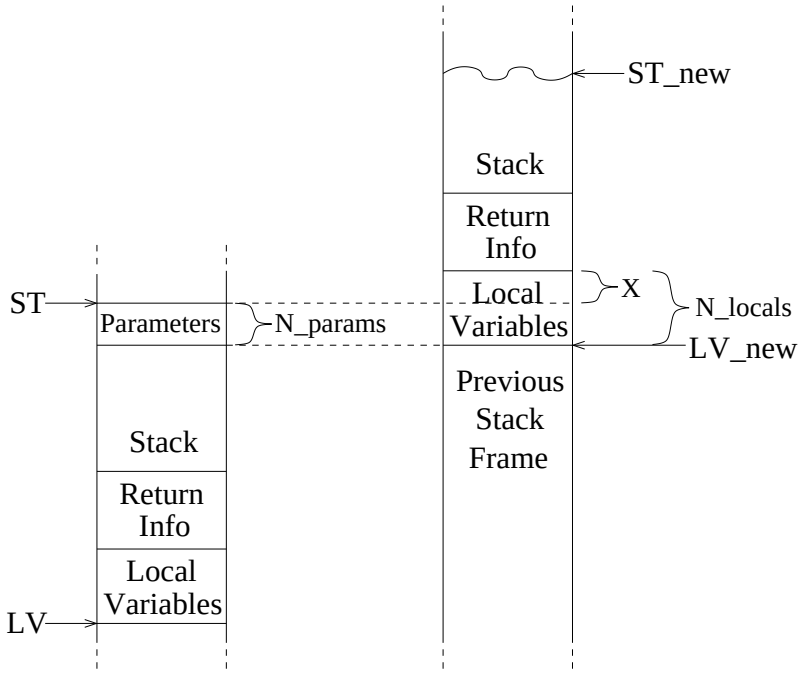


Figure 5.10: Stack behavior during method invocation.

In case a match is not found in the CAM, a trap is produced. To indicate this condition to the ALU the do_trap signal is set high. Upon receiving this signal the ALU sets the trap signal high to communication module, and finally the host CPU performs the needed actions to start execution of the new method. At the same time the invoker module saves the key to the CAM, unless it has received a signal from the CPU disabling the caching for this invocation. When the execution resumes after the trap, the invoker module captures the required register values and saves them to the RAM. Now the invoker is ready to speed up the execution in case the method is

called again. When the invoker module saves a new key to the CAM it uses circular oldest algorithm to choose which entry to replace. This scheme provides reasonably close approximation of the least recently used algorithm with very low complexity. The resulting cached invocation procedure is shown in Figure 5.11. The figure shows both cache hit and miss, as well as the procedure for methods that are not cacheable.

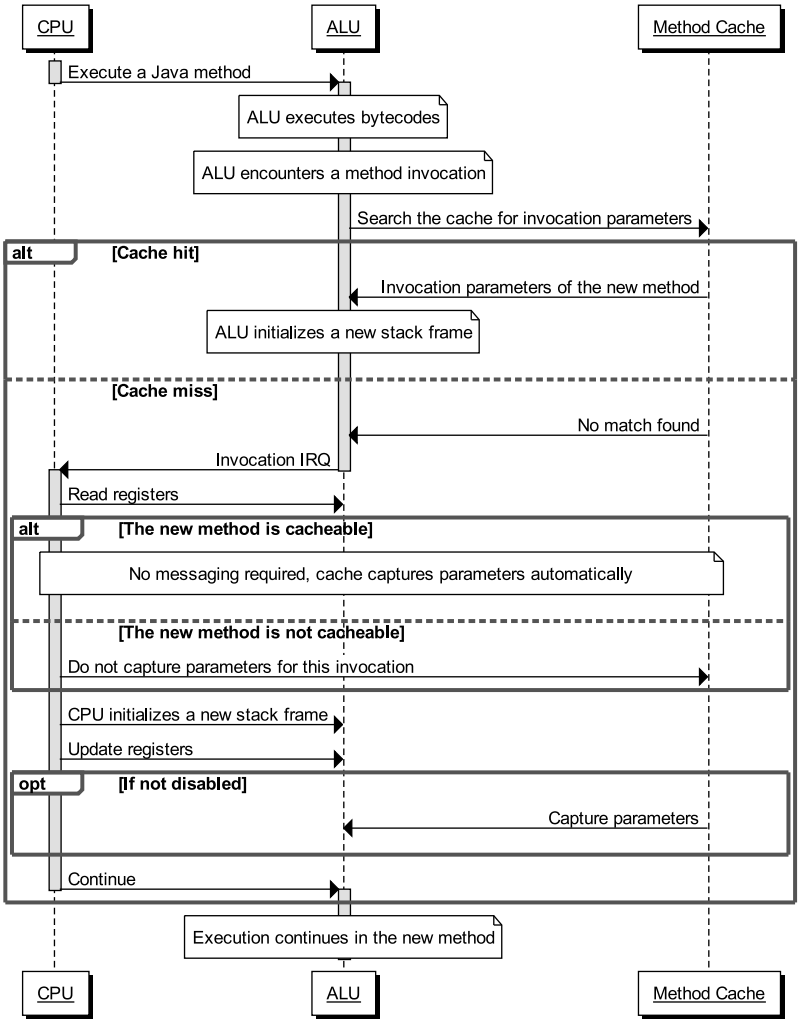


Figure 5.11: A MSC of the cached invocation procedure.

The invoker module can also clear its contents. This is required for situations where a virtual method has been cached to the module, and a new overloading virtual method needs to be loaded. Overloading of methods causes them to fall out of the cache because selecting the implementation

for a specific call requires access to heap data. The host CPU is better suited for this kind of task, so it is assigned to there.

5.5.1 Results

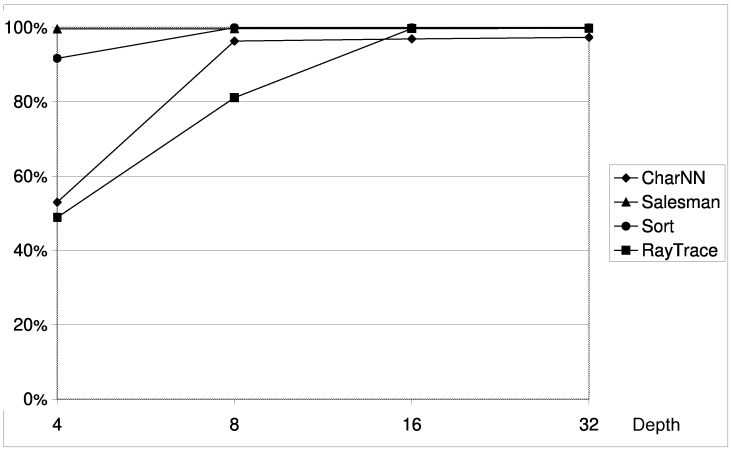


Figure 5.12: The effect of the method cache size on the cache hit rate.

The module was integrated into the REALJava co-processor prototype as eight places deep. This depth was chosen as the statistics in Figure 5.12 show that the size in question provides the highest impact on performance with the least resources. The effect of the invoker module can be seen in Figures 5.13 and 5.14. The first figure shows the performance in the method invocation test of the Embedded Java BenchMark. This test performs 200000 simple method invocations and measures the execution time in order to calculate the number of invocations per second. The second figure shows the effect of this acceleration on the vector sorting benchmark. Details of the benchmarks are given in Section 6.2. In both of the figures the method invocation module was activated in the version 1.00, and the results clearly show the impact to be significant.

The results in Table 5.2 show that the invoker module has significant impact on execution times of the benchmarks. In the table REALJava (ON) stands for a configuration with the invoker enabled, REALJava (OFF) stands for a configuration with the invoker disabled and Kaffe is the Kaffe Virtual Machine running on the same PowerPC processor. REALJava, *even though running at lower clock speed*, clearly outperforms the Kaffe. The gain is the percentage of improvement caused by activating the invoker module.

The first set of benchmarks are a collection of method call tests. They measure only the method call performance, and do not include (significant

amounts of) arithmetics. The first one simply calls an empty method and then returns. The results from the best reference systems are shown for comparison in Figure 5.15. The next four are taken from the Java Grande Suite [110] to show the performance gains for various method types. It is worth noticing that the synchronized calls are not accelerated at all, as stated previously.

The next set of benchmarks are a collection of tests that have been written to evaluate true life performance. The benchmark applications do not contain any special optimizations for our hardware. Short descriptions of the benchmarks follow. *Life* plays Game of Life for a while, *Text* just exercises the text output functions, *Salesman* solves the traveling salesman problem using a naive try all combinations method, *Sort* tests array performance by creating arrays of random numbers and then sorting them, *Neural Net* trains a backpropagational neural network with bitmaps of letters and then recognizes them and finally *Raytrace* renders a 3D sphere above a plane. As the benchmarks emphasize different aspects of the system, together they should give a rather good estimation of different practical applications that might be found on an embedded Java system. The results show 5 to 52 percent improvement in the execution speed when the invocation module is activated. The relatively small improvement in the *Neural Net* test is due to

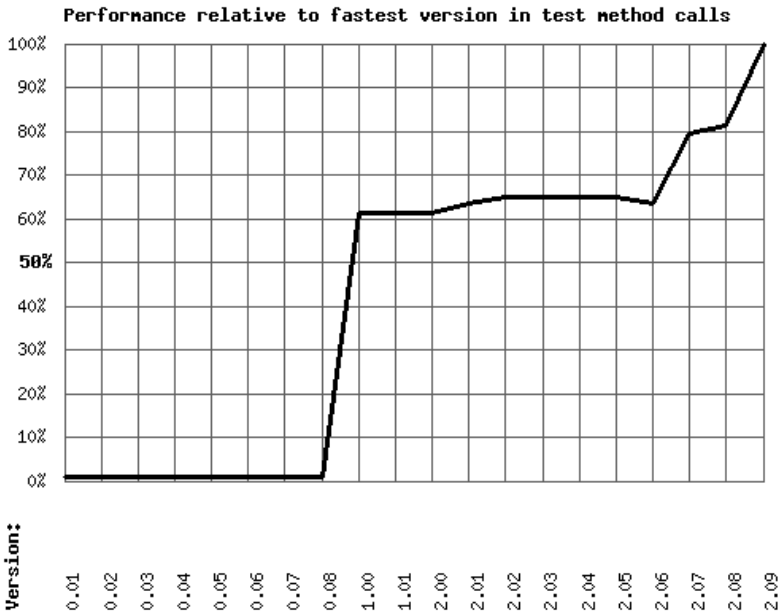


Figure 5.13: The effect of the invoker acceleration. The invoker module was activated in version 1.00.

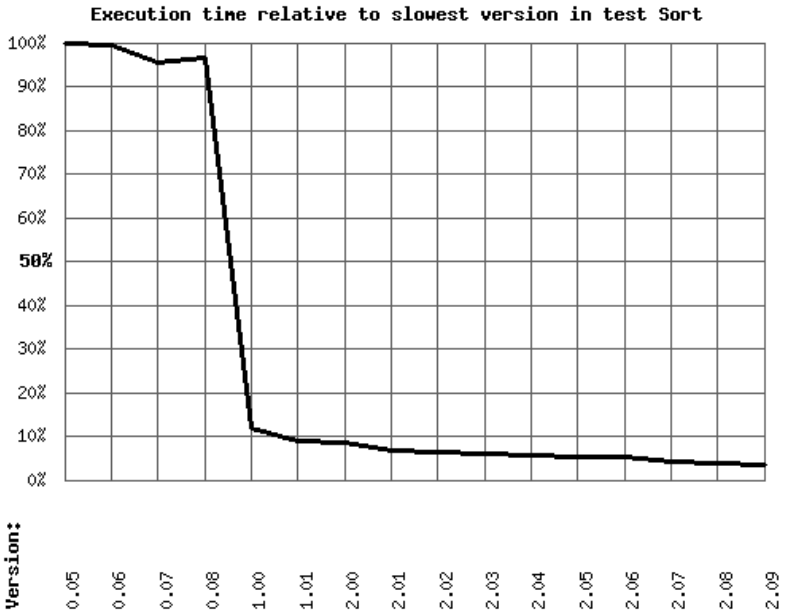


Figure 5.14: The effect of the invoker acceleration on a real test (sort). The invoker module was activated in version 1.00.

Processor	**REALJava ON**	**REALJava OFF**	**Kaffe**	**Units**	**Gain %**
Engine speed	100	100	300	MHz	N/A
Simple call	1754385	265604	32690	1/s	660.5
Instance call	355630	151032	19460	1/s	135.5
Synch. call	33551	33366	15567	1/s	0.6
Final call	334881	158220	18090	1/s	111.7
Class call	341220	164408	18847	1/s	107.5
Fibonacci	1377	1499	5522	ms	8.9
Life	2022	2237	9705	ms	10.6
Text	1709	1931	9455	ms	13.0
Salesman	37496	41744	136079	ms	11.3
Sort	11153	16903	142110	ms	51.6
Neural Net	180441	188705	748649	ms	4.6
Raytrace	19689	25633	223918	ms	30.2

Table 5.2: Amount of method calls per second and execution times for various benchmarks. The execution times include the startup time for the virtual machine. The results are obtained using core version 1.00.

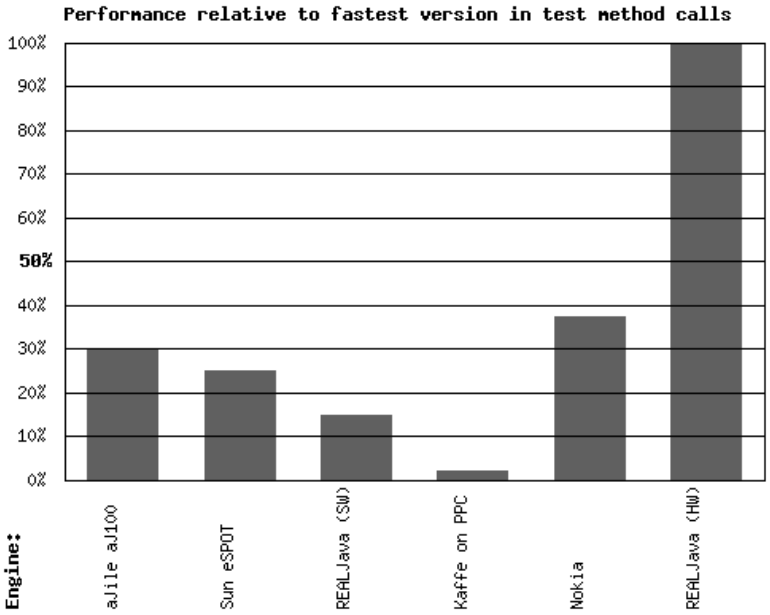


Figure 5.15: The relative performance of the fastest reference systems in the simple method invocations. Also two software virtual machines are shown. The reference systems are described in Chapter 6.

the fact that the number of method invocations in that benchmark is very small in relation to the total execution time. More results can be found at the REALJava results site [111] and in the Appendix B.

5.5.2 Additional Benefits

In addition to the clear performance gain shown previously, the method invocation module produced another opportunity for increasing the performance of the REALJava system. The idea is quite straight forward. Since the invocation module can perform a method invocation with just the 56 bits from the cache and no CPU interaction, the same mechanism can be used for the method calls that cannot be cached. This would include overloaded virtual method invocations and synchronized method invocations. In the first case the CPU simply performs virtual method lookup and once the correct method has been identified the CPU passes the required 56 bits to the method invocation module. The module then performs all necessary steps, as if the information required for the invocation had been found in the cache. The synchronized method invocations follow the same lines, but instead of the virtual method lookup phase the CPU performs tasks related

to the synchronization. The difference can be seen by comparing the previously shown Figure 5.11 and Figure 5.16.

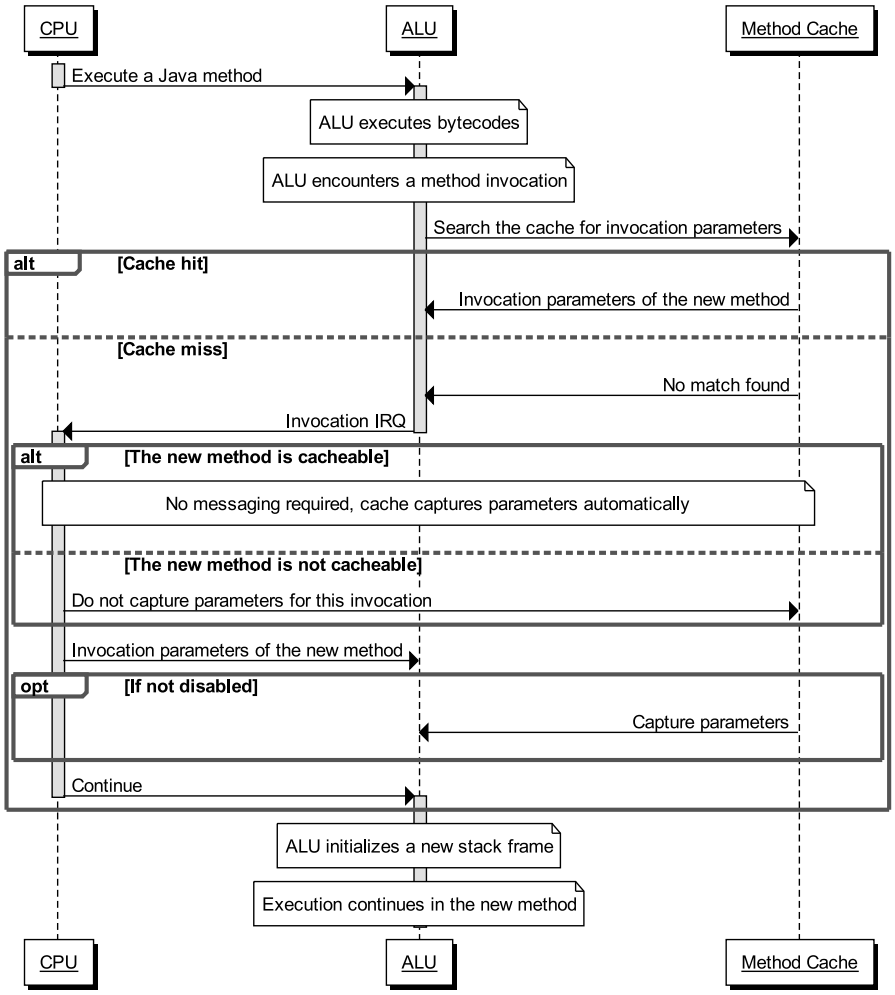


Figure 5.16: A MSC of the final invocation sequence.

This form of acceleration was applied to the system between versions 2.02 and 2.03. The impact of this secondary method invocation acceleration can be seen most clearly in the *RayTrace* test. The results are shown in Figure 5.17. The execution time in the *RayTrace* test was reduced by more than 45% in comparison to the previous version. This surprisingly large gain is caused by the massive amount of virtual method calls used for the file I/O stream writing. The methods used in file I/O are overloaded virtual functions in the classpath, and thus fall out of the hardware based method invocation caching scheme. None of the other tests perform massive file I/O.

For comparison the *RayTrace2* test, which performs the same calculations with out the file I/O only improved by about 11%.

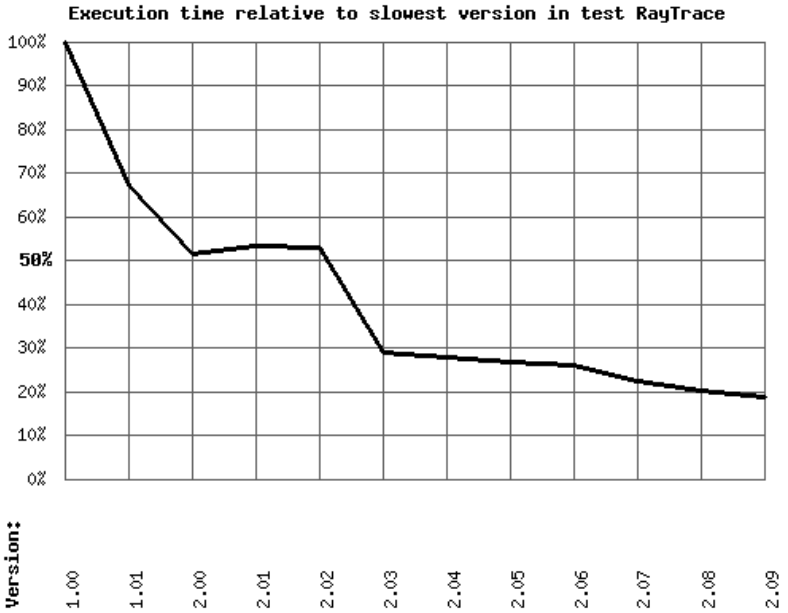


Figure 5.17: The effect of the improved software method invocation on a real test (RayTrace). The invoker module used for SW invocations starting from version 2.03.

5.6 Constant Caches

Java bytecode instruction set can only load signed 8-bit or 16-bit integers to the stack with the data value in the instruction stream as literal data. Also a few constant values can be loaded without any parameter data, these are integers of values -1, 0, 1, 2, 3, 4 and 5, single precision floating point numbers of values 0, 1 and 2 and both long integers and double precision floating point numbers of values 0 and 1. All other values for integers and floating point numbers must be loaded from the constant pool. This would cause an IRQ for each such load, but since the data is constant, as suggested by the name constant pool, it can be cached inside the co-processor. The structure of the constant value cache is quite similar to the structure of the method invocation cache. The key is exactly the same, as both of these caches seek the data values based on the current method and the index in the current methods constant pool. The data value is untyped 32-bit data word, which

can be either a byte, a short, a char, an integer or a single precision floating point number. Since the whole co-processor is designed without hardware support for 64-bit data types the cache also omits those types.

The constant pool load is performed by the *ldc* instruction. This instruction may load other types than supported, so all of the instances of this instruction that load supported data types are reprogrammed to *ldc_fast*. Whenever the co-processor encounters an *ldc_fast* instruction it first performs a look-up in the constant cache. If a match was found it is pushed to the top of the stack. In case of a miss, an IRQ is produced and the CPU pushes the correct value to the stack. At this time the constant cache unit copies the value to its own memory, so that it will be found in the cache if it is needed again. The cache is updated in the same fashion as the method invocation cache. The constant cache is implemented as eight places deep in the co-processor.

Careful investigation of the bytecode instruction set revealed also one more instruction with constant return values. This is the *arraylength* instruction, which pops first a reference to an array from the stack and then pushes the length of that array back to the stack. Since the arrays as well as other objects are referenced using the global reference table, the array reference remains constant regardless of garbage collection. This means that the reference value can be used as a key to a similar cache as presented for the constants. This time the key is the 32-bit reference popped from the stack and the data value is a 32-bit integer whose value is the length of the array in question. This cache is updated in the same way as the constant cache. The rationale for caching the arraylengths comes from the fact that some Java applications perform loops with an arraylength in the loop end condition. This would cause an IRQ for each loop iteration, when the end condition is tested. Since these cases use only one value per loop, which of course can be nested, it makes sense to use only a few locations for the cache. The depth was chosen to be four in the implementation.

5.7 Time Control

The co-processors time control capabilities have not been fully utilized at this point. The time control is used for time slicing the execution between threads. This reduces the overhead at the software partition when more than one threads are running. Without the hardware timers the software partition would have to check the time slice information during every trap produced by the co-processor. With the hardware timer based slicing co-

processor notifies the CPU that it is time to perform a context switch. When the time slicing is done in the hardware the accuracy of the slicing improves as well. This method of slicing will become more and more useful as the number of co-processor cores and simultaneously running threads increase.

The slicing is performed by a counter counting down, and causing an interrupt once the timer reaches zero. The counting progresses one step for each clock cycle. When the counter generates the interrupt, it also resets it self to a specified value. This value is stored in another register, and can be changed by software, if the length of the time slice is to be changed. Setting the counter to value of 0xFFFFFFFF stops the timer. This mode can be used for single thread operation, which requires no slicing, and also for threads with realtime priority.

Hardware timers can also be used to implement high precision timers [55]. The impact of these timers would be clear in a system used for measuring or controlling some real time events than in a system with no real time inputs. No realistic test cases for the time control were designed so it is left for future work.

Finally a third type of time control is available, namely a system level timer used for non-critical events in Java. This includes the `wait()` method of a thread. The `wait()` method is used to push the thread out of execution for a given time. This time is used as the minimum time the threads is not executed, and it can be exceeded by arbitrary amount of time. Since there may be several threads waiting for their timeouts to elapse it gets rather hard for the software to check every one of the timeouts. In theory that would require comparing each system level timer value with the operating systems timer every time a trap is being handled. Instead a system level timer is provided in hardware, signaling the CPU when the first timeout occurs. Then the CPU finds out the thread to reactivated and restarts the system timer with the next timeout as the timers value. Using this scheme frees the CPU from querying the operating systems timer, improves accuracy and speeds up trap handling.

5.8 Additional Registers and Helper Addresses

The internal register bank houses the five control registers mentioned earlier, the four mentioned already in Chapter 3 and the LO register added in Chapter 4. It should be noted that these are not general purpose data registers used for storing of intermediate values during computation. Besides

these it also contains some additional debug registers and helper addresses. The full address space mapping is shown in Table 5.3. For instance when the CPU needs to access the stack top, in the straightforward way the procedure would be for the CPU to first read the ST register, then read the local memory for the data and finally update the ST register. Using the helper address 64 the CPU just reads that register to receive the data. During the read operation the address of the stack top is given to the memory controller by the register bank, which also simultaneously updates the ST register accordingly. The sequence is the same for writing, only the direction of the data is reversed. This way reduces the number of required communications from three to just one per stack item. Also the amount of arithmetics performed by the CPU is reduced, since the new ST register value does not need to be calculated by the CPU. Additionally this improves the cache coherence on the host CPU, since the CPU needs to operate on fewer variables. Both push and pop operations go through the stack cache, ensuring that the cache coherence is maintained and that the cache validity is maximized regardless of the trapped instruction. The impact of the push and pop registers, as shown in Figure 5.18, is about 25% in the integer arithmetics.

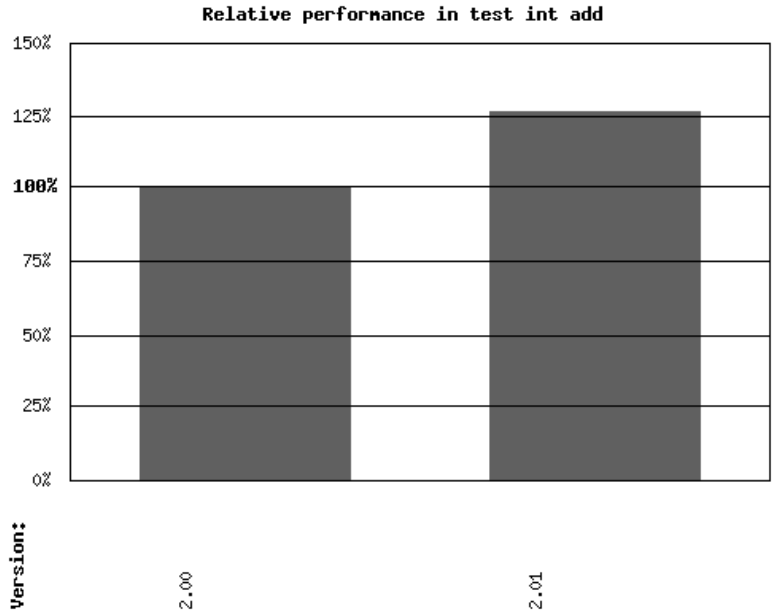


Figure 5.18: The effect of the push and pop registers on the integer arithmetics.

A secondary address is given for the write direction. This is used in case the CPU knows that the write operation is the last action required for

handling the current trap. Writing to this address moves the data to the top of the stack and updates the ST register just as in the normal version. The difference is that after the data is moved to its correct location, the execution on the co-processor is automatically resumed. This removes the need for sending the execute command separately. The same method is not implemented for reads, since the Java bytecode instruction set has no suitable candidates for that kind of optimization. For instance the *iastore* instruction, which stores an integer to an array, pops three values from the stack. The first one is the value to be stored, then is the index to the array and finally the third item is the array reference identifying the array in which the data is to be stored. The execution cannot be resumed directly after any of these reads, since the virtual machine must perform checks on the array index and the reference. Only after they both are found to be valid, can the execution be allowed to continue on the co-processor. If either one of them is invalid, an exception is thrown and the program counter value of the offending instruction is used to find a handler for the exception.

A similar helper address is provided for accessing the parameter data from the instruction stream. Reading this address returns the parameter data for the current instruction. Using the straightforward procedure, the CPU would have to first read the code offset and the PC in order to calculate the actual address of the current instruction, the read the data from the memory and finally update the PC accordingly to reflect the fact that the parameter data was read. Again, the helper address reduces the communications to just one data item per parameter, as opposed to the four communications required before the helper was introduced.

As mentioned in Chapter 3, while discussing the bytecode storage model, the code segment also contains information other than the actual instruction stream. These items are the method id of the method in question and the constant pool address of the class it belongs to. These are accessible through the normal connection from the CPU to the local memory, but accessing them that way requires the CPU to first get the CO register and then access the data. Using the two provided helpers the CPU can omit one read cycle and also saves the arithmetics required to calculate the offset of the data items in the local memory.

Two helper addresses are also provided for the method invocations that cannot be cached inside the co-processor. As mentioned earlier, the method invocation module is used to accelerate these by providing the values of the CO and the LO registers for the new method. Here the later one of these two registers also activates the execution on the co-processor, in similar fashion as in the writes to the top of the stack. The method invocation cache also

needs to be reset when certain events happen, as stated earlier in this Chapter. To provide a channel for that information a control register is placed in this address space.

The registers used for the time slicer are also in this address space. The first one provides the software partition with possibility to control the length of the current slice by setting the counter register to a suitable value. The second one on the other hand allows setting the next slices length in advance. The latter is not reset when it is copied to the actual timer, so the setting remains until it is changed by the software.

After all of these helper addresses were included to the control register bank, the execution sequence was significantly simplified. Figure 5.19 shows the original approach, which required the registers to be read and written for every trap. The improved approach is shown in Figure 5.20. Besides the number of communications, also the amount of arithmetics in the CPU is reduced. The fact that the CPU does not have to calculate new values for the registers additionally improves the cache performance of the host CPU. This is due to the reduction in the amount of variables the CPU has to operate on during a trap.

One control register is implemented for testing purposes. This register has been used to turn modules and features on and off during the testing phases of the system. The functionality of a module or a feature is easier to verify when it can be switched off when needed. Also the performance impact of some modification can be more accurately seen, when tests can be run with and without it. For instance the performance impact of the method invocation module was analyzed using this approach.

The debugging registers and counter are also located in the register address space. These provide debugging and performance evaluation information. The data obtained from these was used in debugging both the software and the hardware during the design phase. They also provided information on the bottlenecks of the system and helped in locating them. All of the debug registers are reset by a reset signal to the co-processor, so they retain their contents after execution. The debug registers can be removed from the system with no effect, but they are kept in order to help debugging the future versions of the REALJava virtual machine as well as new Java applications. These registers provide data of the four previous instructions executed in the co-processor, the amount of time the co-processors has been executing code, the number of traps generated since the last reset, the maximum value of the stack pointer and the amount of space taken by the code segments. Additionally one address is reserved for debugging internal sig-

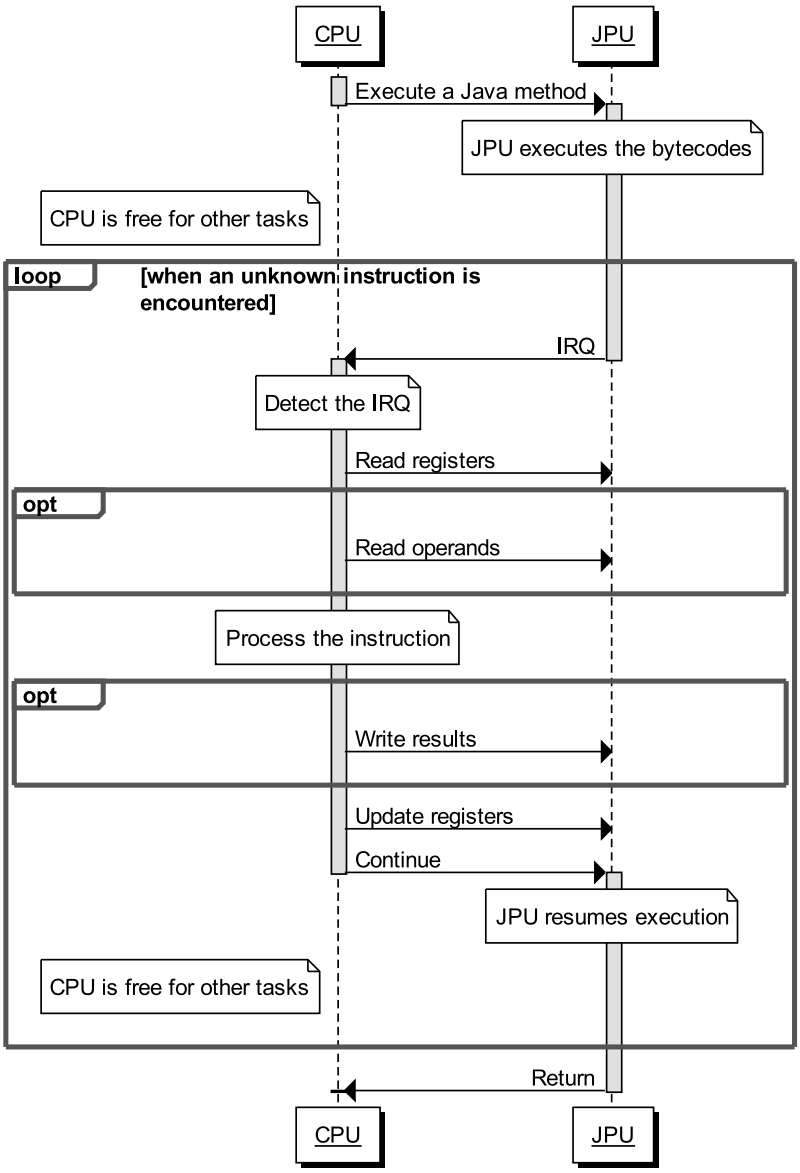


Figure 5.19: A MSC of the original execution sequence.

nals from the co-processor. In case of dubious behavior, the suspect module would be connected to this register in order to see the internal behavior of the module in question.

The last address is used to convey information about the size of the local memory. This is required since the Spartan3 has less BRAMs available. Since accessing the top of the stack from the CPU is handled by the regis-

Address	Name	Access	Function
0	PC	R/W	Program counter
1	ST	R/W	Pointer to the top of the stack
2	CO	R/W	Code offset
3	LV	R/W	Pointer to the local variable area
4	LO	R/W	Number of local variables and parameters
16	Slicer1	R/W	The current value of the time slicer
17	Slicer2	R/W	The resetting value for the slicer
64	− >ST	R/W	Forwarding to/from the top of the stack
65	− >ST_exec	W	Forwarding to the top of the stack, with automatic execute
66	− >Param	R	Access to the parameter data associated to the current instruction
68	New CO	W	Code offset for the new method
69	New LO	W	Number of local variables and parameters for the new method, activates the method invocation automatically
70	MI	R	Retrieve the ID of the current method
71	CP	R	Retrieve the constant pool address of the current method
128	MC	R/W	Method cache control
129	Control	R/W	Control for optional features
249	I_trace	R	Show the last four instructions
250	Live	R	Live timer, does not count the time spend in traps
251	Traps	R	Amount of traps generated
252	Max_ST	R	Maximum value of ST
253	Max_codes	R	Maximum amount of code segments in the local memory
254	Check	R	Temporary debug variable
255	Mem_size	R	Memory size of the co-processor

Table 5.3: Contents of the register address space.

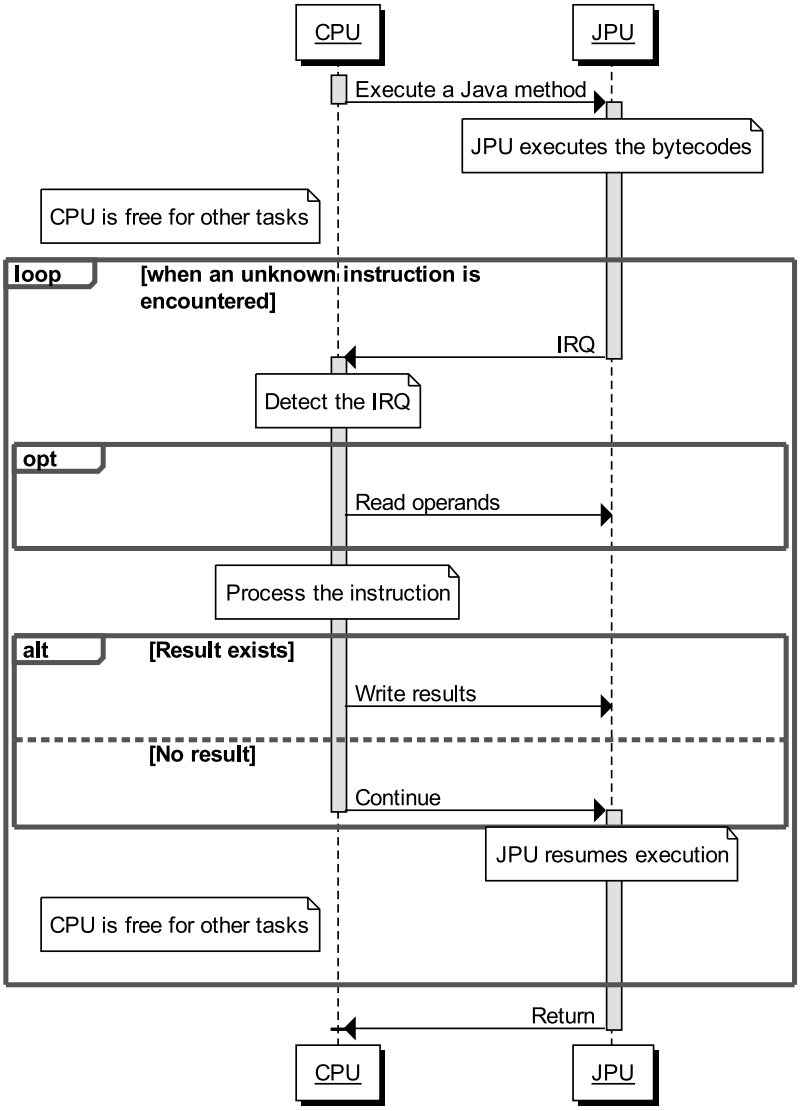


Figure 5.20: A MSC of the improved execution sequence.

ter bank, the link to the local memory is used mainly for storing the Java methods to be executed to the method area of the local memory and during garbage collection to seek the references from the stack.

5.9 Impact of the Techniques

This section shows the overall impact of the performance increasing techniques applied to the REALJava virtual machine. Several figures are used to show the performance gains attained during the research. The version numbers and their relation to the individual techniques can be found in Appendix C. Again, the figures are drawn on relative scales, to make them more readable. Full numerical results of these, and all the other tests, are provided in Appendix B.

First, the arithmetic computations are discussed. To show the improvements in that field, the results from two tests are shown. The first one, shown in Figure 5.21, shows how the integer additions have improved. Since integer, byte and float arithmetics are all performed on the co-processors, they follow roughly the same lines, as well as the other operations on the mentioned types. The second one shows double precision floating point arithmetics, which were left out of the co-processor, causing them to be trapped to the CPU. In order to show the impact of the applied techniques on arithmetics that require CPU intervention, Figure 5.22 shows performance increases in the double precision floating point addition test. As a reference, the aJile aJ100 [5] scores translate to 20% in the integer additions and 300% in the double precision floating point arithmetics. The results show, that in the integer addition test the REALJava surpassed the performance of the aJile between version 0.04 and 0.05. The large difference inthe double arithmetics is due to the lack of floating point unit in the PowerPC CPU and the fact that the REALJava does not perform double precision floating point arithmetic on the hardware.

Second, the impact on the method invocation has already been shown in Figure 5.13. Similar results can been seen in the Figure 5.23, which shows the effect of the performance increasing techniques on the string comparison test. The shape of the curve follows rather closely the shape of before mentioned figure, because the string comparison test uses methods from the standard library. The required method Invocations have a strong impact on the results of this test. The score of the aJile aJ100 in the string comparison test is 69% of the REALJava version 2.09.

Third, the performance in real life applications is evaluated. The results for the *Sort* and the *RayTrace* have already been shown in Figures 5.14 and 5.17, respectively. The Figure 5.24 shows the relative execution times of the *Mandel* test. This application is heavily arithmetics oriented, so also

[5] The aJile aJ100 is the fastest of the full custom solutions used for performance evaluations, and it is described in more detail in the next chapter.

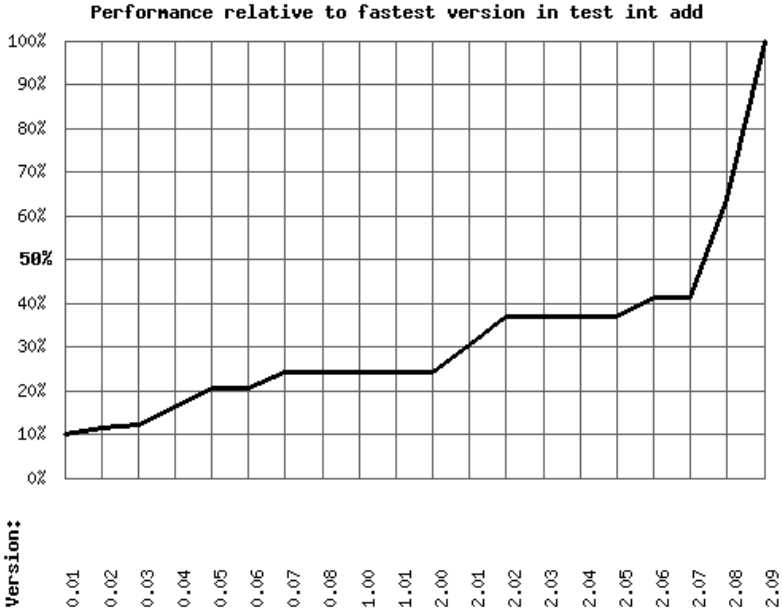


Figure 5.21: Integer performance through the versions of the REALJava.

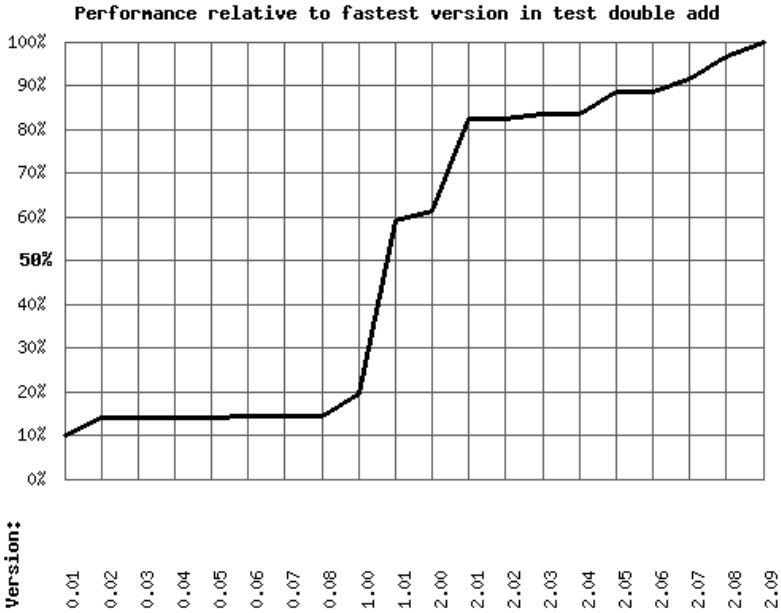


Figure 5.22: Double precision floating point performance through the versions of the REALJava.

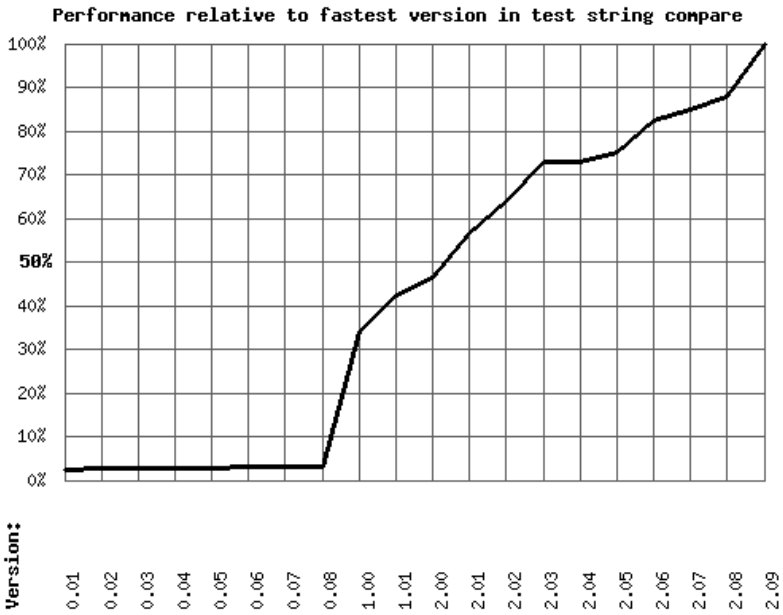


Figure 5.23: Performance of the string comparisons through the versions of the REALJava.

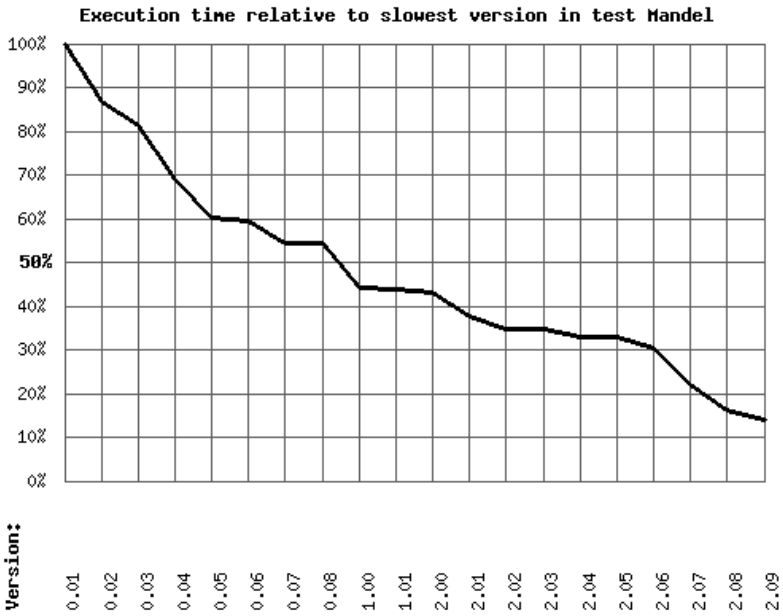


Figure 5.24: Relative execution times in the Mandel test through the versions of the REALJava.

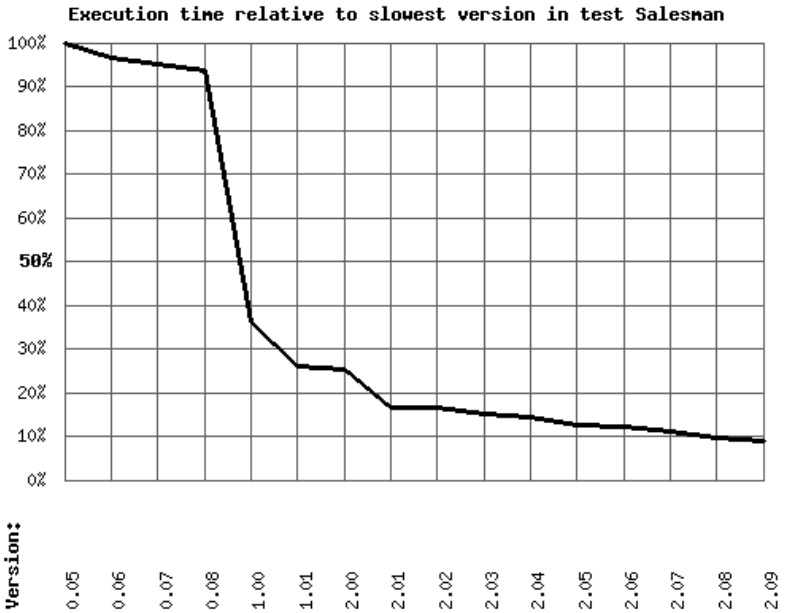


Figure 5.25: Relative execution times in the Salesman test through the versions of the REALJava.

another test is shown, exhibiting a different behavior through the versions. This other test is the *Salesman*, whose relative execution times are shown in Figure 5.25.

Overall, all of the figures show improvement throughout the whole version history, with larger changes occurring at different times due to the applications using different instruction mixes. From the figures it can be estimated that the performance has been increased by a factor of 10, translating to a decrease in the execution times by the same factor.

5.10 Multicore Approach

Even though the multicore support of the REALJava virtual machine falls outside the scope of this thesis, it is worth while to mention that the system is currently capable of running multithreaded Java applications using two parallel co-processor cores. The multicore system provides reasonable performance improvement as it is, but only if the application has been programmed with multiple threads. The efficiency of the multicore architecture can be increased by implementing some parts of the standard library so that the library methods perform their function in multithreaded style.

This would allow the second core to be used in the library routines even if the original user application has only one thread. This and other techniques relating to the multicore execution model, and also to the programming of such a virtual machine, are left for future work.

The REALJava virtual machine would support a higher number of cores, even though the current multicore version uses only two of them. The dual-core approach is used because it physically fits in the currently used FPGA chip without any modification required. The fact that the REALJava is able to run two cores in parallel shows that the multicore capability, rudimentary as it might be, is achieved. The scalability of the virtual machine is not very good at the moment due to the lack of direct communication between the cores. This would lead to the CPU being overwhelmed if the number of co-processor cores would be high. A revised communication module with better multicore support is left for future work.

5.11 Chapter Summary

Several techniques were used to increase the performance of the REALJava virtual machine. The caches for method invocations, constant values and the top four places of the stack were presented. Also the functionality of the method invocation module was detailed. The time domain control features of the co-processor were discussed, focusing on the thread time slicing mechanism. An extended register map was given, highlighting the additional functionality. The helper addresses specified here greatly reduce the number of communication cycles during the execution of a trapped instruction. Finally the multicore approach was briefly discussed.

Chapter 6

Performance Evaluation

This chapter evaluates the performance of the REALJava virtual machine. The reference systems are first described shortly, to provide context for the results. Then the results from various benchmarks are shown and analyzed. The benchmarks include some commonly used ones as well as few Java applications designed to see the real life performance. The real life tests attempt to cover as wide a spectrum of applications as possible. This chapter only shows the best performing reference systems for some of the benchmarks, the full result tables are presented in the Appendix B. The amount of memory accesses during the execution of Java applications is also measured. The performance of the REALJava is broken down to reflect the relative contributions of the CPU, the communication bus and the co-processor. Finally some preliminary results for the multicore version of the REALJava virtual machine are presented.

6.1 Overview of the Reference Systems

The benchmark results are reported for a variety of reference systems. Some of the systems are definitely low-end products, at least in terms of performance, but others demonstrate the pinnacle of the current embedded Java processors.

In the **Sun eSPOT**[1] [90] the main processor is an Atmel AT91RM9200 system-on-chip (SOC) integrated circuit. This unit incorporates the ARM920T ARM Thumb processor, based on the v4T ARM architecture ARM9TDMI. There is no operating system used. According to Sun, the Sun SPOT "runs a Java VM on the bare metal".

[1]SPOT stands for Small Programmable Object Technology.

JStik and **JStamp** are both high speed native-execution modules based on the aJile 32-bit core. JStik has a 32-bit wide data path to external memory, so it can fetch opcodes and data in one cycle. On the other hand the JStamp has an 8-bit data path. This makes JStik 4-5 times faster than JStamp at the same clock rate. Both of these systems use aJile processors, the JStamp uses the aJ80 and the JStik uses the aJ100. Both of the processors execute Java bytecodes natively, but also support extended instructions. In order to use the extended instruction set, a tool called JEMBuilder must be used. This means that the extensions are not available on dynamically loaded Java classes. More details on the aJ100 can be found in [21].

TStik is based on a much lower cost 8-bit 8051 "super core", the Dallas/Maxim DS80C400. TStik executes an interpreted JVM, so it is significantly slower than native execution systems, but it also costs less. The **DSTINI** is based on the DS80C390, which is an older sister model of the DS80C400.

SNAP fits between TStik and JStik in terms of cost and performance. It uses the Cjip [74] processor, which 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, the 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, a string buffer and two stack caches.

Nokia 6170 mobile phone was included as a reference system in order to compare the performance of the FPGA implementation to the performance of a highly optimized embedded software virtual machine. The 6170 uses a translating (no JIT[2]) virtual machine for Java execution. The phone in question was purchased roughly in the December of 2005, when it belonged to the higher middle class of phones available. Unfortunately technical details of the phone, such as the processor type and speed, were not to be found.

The last reference system is **Kaffe** [97] running on the same processor as the software partition of the REALJava. Kaffe is a well known open source JVM, and is has been used as a reference in several studies relating to the development of JVMs. Since the Kaffe is running on the same hardware (PowerPC 405 integrated to the Virtex4FX device) and uses the same external memories etc. as the REALJava, it provides a meaningful reference.

[2]This assumption is based on the fact that the empty measurement loops were not optimized away. All JIT compilers tested did remove at least the empty loops.

The performance of Kaffe and the software only version of the REALJava virtual machine were also evaluated on an x86 based computer, and the results can be found in the Appendix D.

The **REALJava** virtual machine is benchmarked with the system running Linux as operating system, actually the setup is exactly the same as for the Kaffe benchmarks. The older results are collected from Xilinx ML310 board housing a Virtex2Pro FPGA as the main processing element while the newer results are from a ML410 with a Virtex4FX FPGA. The transition from ML310 to ML410 was done between versions 1.01 and 2.00. The board is actually the only difference between the two versions. The cores and the software are generated from the same source codes. The differences in the results can be explained by the fact that the ML310 used DDR type memory while the ML410 moved to a newer generation DDR2 memory. This boosted the performance in all areas that required high bandwidth to the memory. Also the network controller is different on the ML410. In the older system the network controller is a separate chip on the mainboard, and it is accessed via the OPB bus. The Virtex4 device supports built-in tri-mode network controller, which is used in the ML410 board. This built-in controller connects to the CPU directly via the PLB bus.

Also software-only versions of the REALJava virtual machine were tested. The versions with no co-processor are marked "REALJava (SW)" while the versions with a co-processor are named "REALJava (HW)".

Finally a few of the benchmarks were converted to multithreaded form, in order to test the preliminary multicore version of the REALJava virtual machine. The system was composed of two unmodified co-processor cores and rudimentary multicore support on the software partition. Even though the multicore version of the virtual machine falls outside the scope of this thesis, it is worthwhile to show that the whole virtual machine has been designed to support the multicore approach. This applies to both the software and the hardware.

6.2 Descriptions of the Benchmarks

The first benchmark collection is taken from "Practical Embedded Java" [112], under the benchmarking section [3]. This benchmark set is designed especially for embedded environments. The results for the REALJava, the Kaffe and the Nokia 6170 are actual measurements, and the results for the

[3]The version of the benchmark used is 1.1a.

other systems have been taken from the web site. The benchmark calculates the scores for the various sub-tests by first measuring the time to perform an empty loop. The number of rounds in the loop is the same as for the test run. Then the instructions to be tested are inserted to the loop, and the time is measured again. Finally the first loops execution time is subtracted from the test loops time. The purpose of this is to eliminate the effect of the loop from the results. The tested instructions in the testing loops are accompanied by data transfers. According to the authors of the benchmark application, this is done in order to provide more realistic results. This claim is validated by the fact that also in real applications the operands need to be loaded to the stack before the operation and the results are typically saved to the local variables. For the "integer add" test the sequence is:

```
ldc <Int 0x44332211>
iload 8
iadd
istore 10
```

This sequence first loads a constant value, then the contents of the local variable. These values are then added and the result is saved to another local variable. The benchmark measures the time taken to execute these four instructions 200000 times. This is used to count the number of sequences per second, which is the score reported by the benchmark. All of the sub-tests operate in similar fashion. The "Total Loop Executions" (T.L.E.) score is a composite score, which is calculated from the total execution time of the benchmark application. It does not reflect a number of sequences per second, like the individual tests do. Rather it shows a weighted average of all of the sequences tested. The weight factors of the individual tests have been chosen by the authors of the benchmark.

The next set of benchmarks is a collection of tests that have been written to check the correct functionality of the prototype as well as to evaluate the performance. The benchmark programs do not contain any special optimizations for our hardware. Short descriptions of the benchmarks follow. *Mandel* counts the Mandelbrot set, using fixed point arithmetics, *Fibonacci* counts a sequence of the Fibonacci numbers, *Life* plays Game of Life for a while, *Text* just exercises the text output functions, *Salesman* solves the traveling salesman problem using a naive try all combinations method, *Sort* tests array performance by creating arrays of random numbers and then sorting them, *Neural Net* trains a backpropagational neural network with bitmaps of letters and then recognizes them, finally *RayTrace* performs raytracing to create a 3D image of a ball on top of a plane. Since the *RayTrace*

writes the resulting image to the hard drive, which very slow on the ML310 and ML410, another version of the test was designed. It is called *RayTrace2*, and it is otherwise the same, but it does not write the image to the hard drive. As the benchmarks emphasize different aspects of the system, together they should give a rather good estimation of different practical applications that might be found on an embedded Java system. The execution times reported for these benchmarks include the startup and shutdown time for the virtual machines. This is because the times are measured using the **time** Unix command. No web applications are tested, because the majority of the Java web benchmarks available are targeted for servers rather than embedded systems. Also the rather poor network performance of the ML310 and ML410 would cause any web benchmarks to measure mainly the performance of the network connection. The computation between the data transfers would behave similarly to the applications used here.

Some of the benchmarks in the second set were also converted to be J2ME compliant, and they can be identified by the "*_ME*" appended to the name. Specifically the CLDC 1.1 configuration and MIDP 2.0 profile were used, because that combination is supported by the Nokia 6170 mobile phone. The time measurement in these benchmarks had to be moved inside the benchmark programs because the phone's operating system does not provide means of measuring the execution time of an application. The impact of the virtual machine startup time is thus not measured in these benchmarks.

The amount of local memory required for each of the tests is listed in Table 6.1. The table shows the maximum sizes of both the Method area and the stack, as well as the total amount of local memory required and the total size of the class files. The table also shows the memory sizes for the CaffeineMark, which will be discussed later. Besides the actual operand stack, the stack sizes include the local variables and the stack frame data. This is due to the stack frame structure. Similarly the method space sizes include the additional data placed before the actual methods. The total amount of memory for each application, calculated as the sum of the two parts, is the minimum size of the local memory that does not require swapping of the methods or the stack. As mentioned in Chapter 5, the size of the local memory in the smallest prototype, based on the XESS board, is 49152 bytes. That size is clearly sufficient for all of the test applications, and the larger systems have even more headroom. The small memory requirements for the applications was mentioned as one of the attractive points for using Java in embedded systems, and the statistics support that. Especially the class files are very small, thus being very well suited for deployment over networks and also for permanent storage on embedded systems with limited

storage capacity. The deployable size can be further decreased by packaging the calls files into JAR files, as mentioned in Section 2.5.5.

Test	Method space	Stack space	Total	Class files
BenchMark	20572	4140	26116	6365
Mandel	18376	912	19288	1109
Fibonacci	18312	904	19216	1021
Life	18536	904	19440	1256
Text	17988	880	18868	504
Salesman	18492	892	19384	1630
Sort	18648	1968	20616	637
Neural Net	20004	1036	21040	5517
RayTrace	9708	1032	10740	5432
RayTrace2	19036	952	19988	3037
CaffeineMark	21976	4140	26116	14031

Table 6.1: Size statistics of the tests. Besides the requirements for the local memory, also the total sizes of the deployable class files are presented. The sizes are measured in bytes. The test "BenchMark" refers to the benchmark collection from "Practical Embedded Java".

The method space statistics show a very small value for the *RayTrace* test. This was analyzed, to verify the correctness of the value. The test in question is the only one that does not produce any textual output, so the text output was examined by two Java applications. The first one does absolutely nothing [4], and the second one writes one word to the terminal. The method space requirements for the two applications were 2164 bytes and 17924 bytes, respectively. The difference is caused by initializing the text handling system, including character converters, string manipulation tools and so on. Clearly a large share of the method space is used by the text output methods. Since these come from the standard library, they are not modified in this thesis, but as a future improvement they should be looked at.

6.3 Results

Table 6.2 show the results of the benchmark set from "Practical Embedded Java" [112]. Only the best performing hardware virtual machines are

[4] Actually the only method in that application does contain one return instruction, used to return from the user code.

included. The scores for the software virtual machines used for the performance evaluations are shown in Table 6.3, the rest can be found at [112], at the REALJava results site [111] or in the Appendix B. The REALJava site offers also other benchmarks, such as those presented in Table 6.6, and also the measurement data for the different versions of the REALJava core [5]. The results show that REALJava (HW), *even though running at the lowest clock speed*, outperforms the fastest competitors in all of the areas where the hardware acceleration has been applied. The benchmarks that indicate lower performance (array accesses and double precision arithmetics) are all handled by the software partition. The scores are surprisingly good, considering that the aJile is designed using ASIC technology and the REALJava virtual machine is using an FPGA based co-processor. Especially the integer arithmetics are significantly faster in the REALJava. This suggests that when the complex instructions are left out of the hardware, the remaining instructions can be optimized further. Also the Total Loop Executions scores show that the overall performance of the REALJava virtual machine is more than twice as fast as the aJile.

Two software virtual machines are also used as reference systems. Both of these run on the PowerPC of the larger Xilinx boards. The results are shown in Table 6.3. The results show that the hardware accelerated REALJava is clearly the fastest in all of the sub-tests. The T.L.E. scores indicate that the Kaffe is roughly 20 times slower and that the REALJava running in software only is about four times slower. The most surprising observation is that the Kaffe is almost five times slower than the software version of the REALJava. This would suggest that the software architecture of the Kaffe is somehow unsuitable for the PowerPC 405. Both of these software virtual machines have been compiled using the same tools and the same options. The performance of the software only version of the REALJava virtual machine is actually higher that the Sun eSPOT's by a factor of two. Even the normalized T.L.E. / MHz score is higher.

The T.L.E. scores for all of the applicable reference systems are tabulated in Table 6.4. The results are normalized by the clock frequency of the execution engine in each system. Two of the systems clearly stand out in the T.L.E. / MHz column, namely the aJile and the REALJava (HW). For clarity, the scores are also shown in Figure 6.1.

Since the double precision floating point arithmetics are not implemented in hardware in the REALJava virtual machine, a modified version of the

[5]The site defaults to the newest version available, the version used to report the results in this thesis is 2.09.

Processor	**aJile aJ100**	**Sun eSPOT**	**REALJava (HW)**
Engine speed (MHz)	103	180	100
byte array access	**879677**	479239	791975
byte array copy	32768000	6241523	**52428800**
int array access	**1008246**	432580	789590
int array copy	7281777	1927529	**14563555**
byte add	2702702	1438848	**10000000**
byte sub	2777777	1449275	**9090909**
byte mul	1369863	1428571	**9090909**
byte div	1360544	1190476	**1960784**
int add	2898550	1724137	**14285714**
int sub	2898550	1724137	**12500000**
int mul	1587301	1652892	**14285714**
int div	1351351	1369863	**2083333**
float add	2985074	250000	**4545454**
float sub	2666666	246002	**4545454**
float mul	1408450	199401	**6666666**
float div	1379310	100452	**2702702**
double add	**1754385**	226500	578034
double sub	**1612903**	215517	540540
double mul	**581395**	193236	460829
double div	**574712**	70348	163800
string concat	3711	533	**13726**
string compare	270270	67340	**392156**
method calls	847457	706713	**2857142**
object creations	36101	86206	**645161**
Total Loop Executions	146536	34553	**302941**
T.L.E./MHz	1423	192	**3029**

Table 6.2: Execution rates for various loops (cycles per second). The best scores are highlighted.

Processor	**Kaffe on PPC**	**REALJava (SW)**	**REALJava (HW)**
Engine speed (MHz)	300	300	100
byte array access	98773	405168	**791975**
byte array copy	26214400	18724571	**52428800**
int array access	88622	468114	**789590**
int array copy	9362285	10082461	**14563555**
byte add	224215	1098901	**10000000**
byte sub	225733	1104972	**9090909**
byte mul	174064	1005025	**9090909**
byte div	219298	995024	**1960784**
int add	355239	1388888	**14285714**
int sub	326797	1388888	**12500000**
int mul	371057	1351351	**14285714**
int div	351493	1183431	**2083333**
float add	303030	675675	**4545454**
float sub	296735	655737	**4545454**
float mul	312012	696864	**6666666**
float div	251889	500000	**2702702**
double add	270635	277777	**578034**
double sub	257731	274348	**540540**
double mul	183654	252206	**460829**
double div	124610	126182	**163800**
string concat	265	1320	**13726**
string compare	15372	142857	**392156**
method calls	59488	425531	**2857142**
object creations	18761	86206	**645161**
Total Loop Executions	15221	74057	**302941**
T.L.E./MHz	51	247	**3029**

Table 6.3: Execution rates for various loops (cycles per second). The best scores are highlighted.

	Engine Speed	T.L.E.	T.L.E. / MHz
DS80C390	36.864	555	15
DS80C4000	29.491	492	17
Imsys Cjip	80	11557	144
aJile aJ80	73	41106	563
aJile aJ100	103	146536	1423
Sun eSPOT	180	34553	192
Kaffe on PPC	300	15221	51
REALJava (SW)	300	74057	247
REALJava (HW)	100	302941	3029

Table 6.4: Total loop executions for various systems.

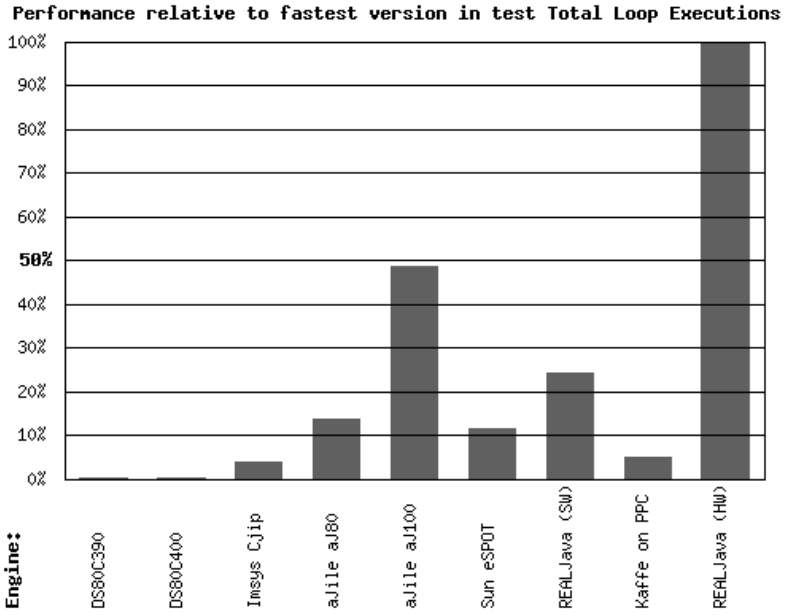


Figure 6.1: Total loop executions for all available systems.

benchmark was created. This modified version does not include the double type tests. Since the source code for the benchmark is available at [112], it was simple to just leave the double arithmetics out from the T.L.E. calculations. Knowing the way the benchmark calculates the T.L.E. score, an estimator was programmed to find out the T.L.E. scores for the other systems, assuming that the double precision floating point arithmetics are ignored. The accuracy of the estimator is reasonable, as demonstrated by the estimates for the Kaffe and both of the REALJava versions. The estima-

tor predicted T.L.E. scores 14460, 77700 and 464886 for Kaffe, REALJava (SW) and REALJava (HW), while the measured scores were 14805, 78074 and 465759 respectively. The error of the estimate, based on the data points available, is within 3 %. The estimates and measurement results are presented in Table 6.5. The results show that, if the double arithmetics are left out, the REALJava (HW) is more than three times as fast as the best reference system.

	Engine Speed	T.L.E.	T.L.E. / MHz
DS80C390	36.864	731	20
DS80C4000	29.491	668	23
Imsys Cjip	80	11418	143
aJile aJ80	73	39832	546
aJile aJ100	103	147989	1437
Sun eSPOT	180	34727	193
Kaffe on PPC	300	14805	49
REALJava (SW)	300	78074	260
REALJava (HW)	100	465759	4657

Table 6.5: Total loop executions for various systems when double arithmetics has been ignored. The results for Kaffe and both versions of REALJava are measurements, the others are estimates.

The next set of benchmarks is executed only on the Kaffe and on both of the versions of the REALJava. Table 6.6 shows the execution times for the benchmarks. The times are measured using the **time** Unix command. The results show that the REALJava (HW) is between 9 and 25 times faster than the Kaffe. This supports nicely the results of the previous benchmark. The hardware accelerated version of the REALJava is between 2 and 7 times faster than the software version, also supporting the results from the previous benchmark.

In Comparison to a Mobile Phone

The performance of the REALJava virtual machine was also compared to the performance of a mobile phone. The phone in question is Nokia 6170, a roughly two and a half years old phone, which belonged to the higher middle class of that time. The results for the first benchmark set are shown in the first section of Table 6.7. Note that the scores for the object creation test and the T.L.E. are not reported. This is because the Java virtual machine of the phone did not allow creating new instances of the benchmark object.

Processor	REALJava (HW)	REALJava (SW)	Kaffe on PPC
Mandel	**5325**	36784	134878
Fibonacci	**433**	1127	4545
Life	**571**	1601	7804
Text	**518**	1429	7489
Salesman	**12625**	29437	115769
Sort	**4396**	13961	106285
Neural Net	**64794**	127553	590552
RayTrace	**12168**	58507	229174
RayTrace2	**8349**	29862	177037

Table 6.6: Execution times for various benchmarks (times are in milliseconds). The best scores are highlighted.

This violated the security rules of the virtual machine, stating that no new object belonging to the `midlet` class are allowed. The T.L.E. score thus becomes incompatible with the other systems. The REALJava again dominates in the sub-tests that have been accelerated in the co-processor, and falls behind the phone in the array accesses and in the double arithmetics. A surprising results can be seen in the string manipulation tests. While the REALJava (HW) was 3.70 and 1.45 times as fast as the aJile in the concatenations and the compares respectively, the mobile phone outperforms the REALJava by 13 and 31 percent in the same sub-tests. Even though the difference is relatively small, it suggests that the string handling routines in the phones classpath are very highly optimized. The REALJava virtual machine uses the GNU Classpath without any modifications or optimizations.

Table 6.7 also shows the execution times for some of the benchmarks from the second set. These are not on the same scale as the previously presented results, since the benchmarks had to be slightly modified to be suitable for execution on the phone. The *Sort* test was performed ten times inside the time measurement loop, since the original version was too fast. This caused the timer on the phone to report values with about 10 % spread. The spread was reduced to less than 1 % after the loop was run ten times. The results show clear performance lead for the REALJava in all of the tests except the *Salesman*. This is due to the fact that the *Salesman* contains relatively little arithmetics mixed with a lot of array accesses. As shown in the first section of Table 6.7, the Nokia is faster in array accesses. The Salesman runs a naive algorithm to find the solution to the traveling salesman's problem. It is implemented by storing the distances between the cities in a

	REALJava	**Nokia 6170**
byte array access	791975	**962879**
byte array copy	**52428800**	22795130
int array access	789590	**916587**
int array copy	**14563555**	6096372
byte add	**10000000**	2724795
byte sub	**9090909**	2840909
byte mul	**9090909**	2695417
byte div	**1960784**	1694915
int add	**14285714**	3361344
int sub	**12500000**	3316749
int mul	**14285714**	3210272
int div	**2083333**	1917545
float add	**4545454**	1733102
float sub	**4545454**	1572327
float mul	**6666666**	1633986
float div	**2702702**	973709
double add	578034	**1207729**
double sub	540540	**1240694**
double mul	460829	**788643**
double div	163800	**310173**
string concat	13726	**15467**
string compare	392156	**512820**
method calls	**2857142**	1069518
Mandel ME	**4832**	14098
Salesman ME	11597	**10546**
Sort ME	**40342**	55777
RayTrace ME	**7891**	9036

Table 6.7: Results of the benchmarks modified for mobile phone. The top section gives the scores of the first benchmark set while the lower section reports the execution times of the second set. The best scores are highlighted.

matrix. The arithmetics required are simply adding the distances together and comparing the result to the best solution found so far.

CaffeineMark

Several websites and research papers dedicated to Java execution have used the CaffeineMark [109] as a performance measurement. The CaffeineMark is also available as an embedded version, which omits the graphical tests from the desktop version. The test scores are calibrated so that a score of 100 would equal the performance of a desktop computer with 133 MHz Intel Pentium class processor. The individual tests cover a broad spectrum of applications. The descriptions of the tests follow. The *Sieve* is the classic sieve of Eratosthenes, and it finds prime numbers. The *Loop* test uses sorting and sequence generation to measure the execution rate of loops. The *Logic* tests the speed with which the virtual machine executes decision-making instructions. The *String* test operates on string data, performing concatenations and seeks on the strings. The *Float* simulates a 3D rotation of objects around a point. Finally, the *Method* test executes recursive function calls to see how well the virtual machine handles method calls. The Overall score is a composite of the individual tests, calculated as the geometric mean of the individual scores.

Some of the results gathered from various sources can be seen in Table 6.8. The results for IPAQ are from [12], which does not specify the type of the device, but states that it is a handheld device running Linux as operating system. The virtual machine running on that system is the Sun KVM. The Wikipedia lists various models of the IPAQ [113], and the clock frequencies reported there range from 200 MHz to 624 MHz. The scores for the SHAP [64] are also presented. The SHAP is a standalone Java processor focusing on the security and real-time aspects of the virtual machine. The results indicate that the REALJava is twice as fast as the second best reference system, when considering the overall score without the floating point arithmetic tests. The results for the floating point tests are not available for all of the systems. For instance the SHAP does not support any floating point arithmetics. Even though slower in the overall score, the SHAP performs very well in the string test, and actually also the IPAQ does relative well in that test, when compared to the overall score. Since the REALJava and the Kaffe both use the GNU classpath and the IPAQ, as a commercial product, probably uses an optimized classpath, it seems reasonable to assume that the SHAP also has some kind of optimized classpath implementation. The scores of the string test here and the string tests run on the mobile phone suggest that the GNU classpath is somehow inefficient in the string

operations. Finding the cause for this is left for future work, since this is not really a problem in the REALJava virtual machine, but rather in the supporting library.

	REALJava	**SHAP**	**IPAQ**	**Kaffe**
Sieve	**140**	15	40	9
Loop	**121**	114	35	6
Logic	**466**	321	39	21
String	95	**200**	129	14
Float	**61**	N/A	N/A	8
Method	**285**	64	35	9
Overall	**153**	N/A	N/A	10
(no float)	**184**	93	55	11

Table 6.8: Results of the CaffeineMark test suite.

6.4 Memory Accesses

In order to evaluate the impact of REALJava virtual machine on the amount of physical memory accesses, an analyzer was attached to the memory bus of the CPU. The analyzer simply counts the number of accesses to the address space of the memory controller. Both the reads and the writes are counted in the same counter. The measurement results are presented in Table 6.9.

As predicted earlier, the hardware acceleration significantly reduced the number of accesses to the physical memory. The results are well in line with the study [25] of Hsieh et al., who analyzed the cache and branch prediction behavior of Java execution. Seshadri et al. [47] also studied the same aspects with different workloads, and they also concluded that Java execution is not favorable to caches and branch predictors.

Using the power consumption profiles from [33, 34], which state that around 70% of the energy consumed during the execution of a Java application is used on memory accesses, the total energy consumption can be divided between the CPU and memory. It can be estimated that the portion of the power consumption caused by the memory accesses decreases proportionally to the amount of the memory accesses. Using E_{CPU} for the energy used in the CPU, E_{MEM} for the memory and E_{JPU} for the co-processor, and estimating [6] that the co-processor uses as much energy *per*

[6]This estimate is very conservative, since the PowerPC is four times larger than the

	REALJava (HW)	REALJava (SW)		Kaffe	
Start	3337362	11475149	(3.44)	26141221	(7.83)
Fibonacci	5828262	13599265	(2.33)	51458556	(8.83)
Life	6492661	15399044	(2.37)	88610969	(13.65)
Mandel	6640936	16718282	(2.52)	105763181	(15.93)
Salesman	7048099	29258496	(4.15)	660082767	(93.65)
Sort	6662480	14561877	(2.19)	791217728	(118.76)
Neural	18438158	85038430	(4.61)	3134828762	(170.02)
Total	60082862	199446750	(3.32)	4898708471	(81.53)
Average			(3.09)		(61.24)

Table 6.9: The number of accesses to the physical memory during the execution of the benchmarks. The numbers in the parenthesis show the ratio of the accesses in comparison to the hardware accelerated REALJava.

clock cycle as the PowerPC CPU, the following estimates can be formulated:

$$E_{SW} = E_{CPU} + E_{MEM}$$

$$E_{MEM} = 0.7 * E_{SW}$$

$$E_{CPU} = 0.3 * E_{SW}$$

$$E_{HW} = E_{CPU} + E_{JPU} + E_{MEMHW}$$

$$E_{JPU} = 1/3 * E_{CPU}$$

Assuming that the amount of energy consumed by the processing elements is distributed evenly over time, and using C_{reduct} for the factor of reduction in the memory accesses, T_{EXEC} for the total execution time and P_{CPU} for the power consumption, the following expressions can be obtained:

$$E_{CPU} = T_{EXEC} * P_{CPU}$$

$$0.3 * E_{SW} = T_{EXEC} * P_{CPU}$$

$$P_{CPU} = 0.3 * E_{SW}/T_{EXEC}$$

REALJava co-processor core.

$$E_{HW} = E_{CPU} + 1/3 * E_{CPU} + 1/C_{reduct} * E_{MEM}$$

$$E_{HW} = 4/3 * T_{EXEC} * P_{CPU} + 1/C_{reduct} * 0.7 * E_{SW}$$

$$E_{HW} = (0.4 * T_{HW}/T_{SW} + 0.7/C_{reduct}) * E_{SW}$$

The total execution times for all the test in Table 6.9 are 88358 ms, 210463 ms and 959833 ms for REALJava (HW), REALJava (SW) and Kaffe, respectively. Thus, substituting the appropriate values, the relative energy consumption values can be estimated:

$$E_{REALJava(HW)} = 0.0454 * E_{Kaffe}$$

and

$$E_{REALJava(HW)} = 0.3788 * E_{REALJava(SW)}$$

Using the values obtained above, the energy consumptions and energy-delay products (EDP) for the systems are presented in Table 6.10. The EDP values show that the hardware accelerated version of the REALJava virtual machine is by far the most efficient of the systems.

	Time	Energy	EDP
REALJava (HW)	88358	1.00	1.00
REALJava (SW)	210463	2.64	6.29
Kaffe	959833	22.03	239.31

Table 6.10: Execution times, relative energy consumption estimates and energy-delay products (EDP).

6.5 Breakdown of the Performance Factors

In order to estimate the performance of the REALJava virtual machine on other host systems with different communication channels and speeds for the CPU and the co-processor the system is evaluated by breaking down of the performance figures to evaluate the effect of each parameter. The T.L.E. score of the first benchmark set is used as the performance metric. Since the double precision floating point arithmetics are supposedly rare, and thus not implemented in the co-processor, the T.L.E. score is also evaluated without the doubles. The equations obtained here are of course only

rough estimates of the resulting performance, but they can be used as estimation tools, when selecting the individual components of a new system. For more accurate predictions the Java applications intended to be used on the new platform should be known in advance so that the profiling could be performed with them. The actual profiling would then proceed as here, but with the correct applications.

First the impact of the communication channel is estimated. The estimation is obtained by artificially slowing down the communication in the REALJava. This is done by inserting dummy communications to the communication routines in the software partition. In the Table 6.11 the execution times and the resulting T.L.E. scores are presented. The slowdown factor indicates the resulting communication speed so that factor of one represents the original speed, two is twice as slow (one dummy and the actual communication), three is three times as slow (two dummies and the actual communication) and so on up to ten (nine dummies and the actual communication). The slowdown factor of zero is an estimate based on the linear regression equation, and it shows the performance level that would be attained with instantaneous communication. It can be noticed, that the results for slowdown factor of one differ marginally from the ones presented earlier in Table 6.4 and in Table 6.5. This is due to the insertion of the code that facilitates slowing down the communication. The data is also shown in Figure 6.2, along with the linear regression curves and the derived equations.

Slowdown Factor	**Time**	**Time** no doubles	**T.L.E.**	**T.L.E.** no doubles
0	*4757*	*2673*	*383946*	*608467*
1	6076	3521	300597	461923
2	7290	4235	250539	384045
3	8507	5096	214697	319158
4	9835	5922	185707	274642
5	10798	6686	169145	243259
6	12112	7554	150795	215307
7	13224	8155	138114	199439
8	14646	9103	124705	178669
9	15969	9957	114373	163345
10	17404	10756	104943	151211

Table 6.11: The execution times (in milliseconds) and resulting T.L.E. scores with slowed down communication. The values in italics are estimates based on linear regression of the execution times.

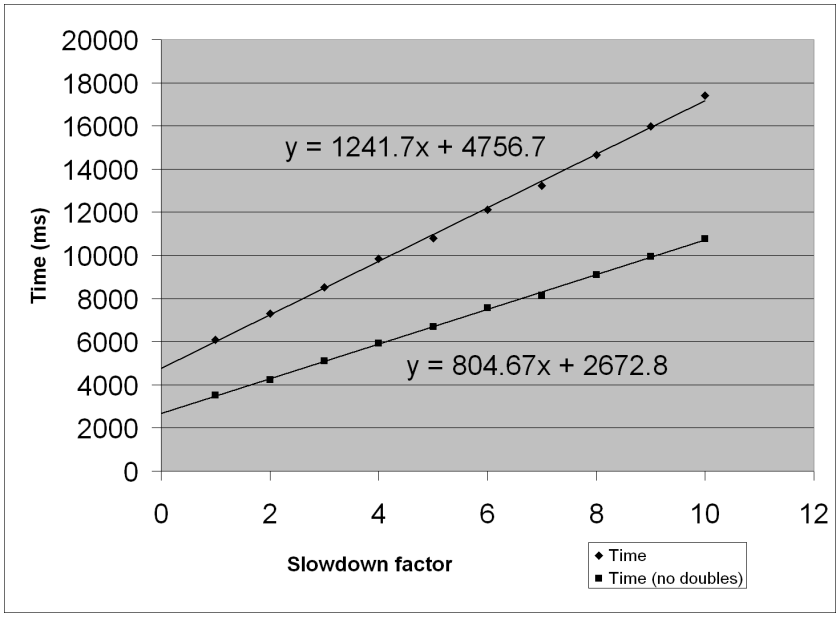


Figure 6.2: The execution times of the first benchmark set with slowed down communication. Regression curves are shown with the corresponding equations.

Based on the regression equations the T.L.E. scores can be expressed as a function of the relative communication speed, expressed as $R_{Comm} = \frac{old\ transfer\ rate}{new\ transfer\ rate}$. Using $T.L.E.$ for the T.L.E. score and $T.L.E._{ND}$ for the T.L.E. score without double precision floating point arithmetics, the equations are:

$$T.L.E. = \frac{1826432}{1.2417 * R_{Comm} + 4.7567}$$

$$T.L.E._{ND} = \frac{1626432}{0.80467 * R_{Comm} + 2.6728}$$

The equations were checked by calculating the T.L.E. scores for each of the slowdown factors, and the results were within ± 1 of the measured value. This shows that the equations are quite accurate. The relative communication speed needs to be transformed to a more understandable form in order to make the equations usable in the evaluation of a possible new system. The communication speed was measured to be 49875311 bytes/second for the writes and 43859649 bytes/second for the read. Since the PLB bus operates at 100 MHz and the data is transmitted in 32-bit words, this translates to 8 clock cycles for the writes and 9 clock cycles for the reads. The difference is due to the fact that the co-processor needs one cycle to retrieve the data requested by the CPU, while the writes are pipelined, effectively hiding

the one cycle needed to save the data presented by the CPU. It is also worth noting, that the communication speeds were measured as single word transactions. Since the communication profile of the REALJava mainly consists of very small transmissions, fast burst mode would not be effective. Also packet based communication channels may have good transmission speeds when single communication units are large, but still the performance of the REALJava would not reflect the high speed since the packets would not be filled with useful data.

Since the equations obtained above can be used to calculate the time used for the communication and the co-processor contains a counter measuring the time spend on the hardware accelerator, the time spend on the CPU can also be calculated. The calculation is based on the assumption that the total time consumed can be expressed in three parts, namely the CPU, the communication and the co-processor. These are denoted by t_{CPU}, t_{COMM} and t_{JPU}, respectively. The T.L.E. scores can then be calculated from:

$$T.L.E. = \frac{1826432}{t_{CPU}+t_{COMM}+t_{JPU}}$$

$$T.L.E._{ND} = \frac{1626432}{t_{CPU}+t_{COMM}+t_{JPU}}$$

The regression equations reveal the time spend on the CPU and JPU to be 4756.7 and 2672.8 milliseconds with and without the doubles, and the live time counter shows that the JPU has been active for 2051.0 and 1725.5 milliseconds. Using these yields the CPU time to be 2705.7 and 947.3, again in the same order. Using $R_{CPU} = \frac{old\ clock\ frequency}{new\ clock\ frequency}$ for the relative speed of the CPU and $R_{JPU} = \frac{old\ clock\ frequency}{new\ clock\ frequency}$ for the relative speed of the JPU, the following equations can be obtained:

$$T.L.E. = \frac{1826432}{2.7057*R_{CPU}+1.2417*R_{Comm}+2.0510*R_{JPU}}$$

$$T.L.E._{ND} = \frac{1626432}{0.9473*R_{CPU}+0.80467*R_{Comm}+1.7255*R_{JPU}}$$

Normalizing the coefficients of the denominator to the smallest, namely the communication, results in the values shown in Table 6.12. From the values in the table it can be seen, that the relative effect of the CPU is higher in the T.L.E. score calculated with the double precision floating point operations. This is to be expected, since they are handled by the software partition. The fact that the doubles are handled on the software also causes the communication channel to have a slightly more higher factor, thus reducing the JPU's share.

	Coefficient	**Coefficient** no doubles
CPU	2.179	1.1772
Communication	1	1
JPU	1.652	2.144

Table 6.12: Coefficients of the performance. The higher the coefficient, the more significant effect the unit has on the overall performance.

The equations were used to estimate how much the CPU of the ML410 could be slowed down to get the same T.L.E. scores as the best performing reference system, the aJile aJ100. In case of the normal T.L.E. score, the CPU could be slowed down by a factor of 3.4, which would result in a clock frequency of 88 MHz. Without the doubles, the factor comes to 8.9, equaling a clock frequency of 34 MHz. This kind of system would, however, be unrealistic, as the CPU typically runs at the same or higher frequency than the bus. Scaling all components, that is the CPU, the bus and the co-processor, to the same frequency, the T.L.E. score of the aJ100 is attained at about 92 MHz with doubles and at about 49 MHz without. Table 6.13 shows the system clock frequencies required to match the performance of the reference systems.

	Reference MHz	**T.L.E.**	**Clock** MHz	**T.L.E.** no doubles	**Clock** MHz
DS80C390	36.864	555	0.35	731	0.24
DS80C4000	29.491	492	0.31	668	0.22
Imsys Cjip	80	11557	7.22	11418	3.77
aJile aJ80	73	41106	25.7	39832	13.2
aJile aJ100	103	146536	91.5	147989	48.9
Sun eSPOT	180	34553	21.6	34727	11.5
Kaffe	300	15221	9.51	14805	4.89
SW 2.00	300	74057	46.3	78074	25.8

Table 6.13: Clock frequencies that give the REALJava the same performance as the reference systems have. The CPU, the bus and the co-processor are all scaled to the same clock frequency.

The equations do not address the effects of various CPU architectures, pipeline lengths, instruction sets or memory subsystems for the CPU. All of

these are considered to be lumped into the relative CPU speed. The relative speed of the JPU is also considered to affect the speed of the local memory and all other subsystems in the co-processor. The speed of the co-processor should be the same, if implemented in a different FPGA, or even in ASIC technology, as long as the clock frequency remains the same. When the clock frequency is scaled, the performance of the co-processor should scale directly in proportion, if the clock frequency scaling is applied to all of the components and subsystems of the co-processor.

Additionally the time required for transferring the execution from the co-processor to the CPU and back was measured. This was done using a specially generated Java application which had 20000 non-existent instructions in a row, enclosed in a loop. A non-existent instruction was chosen because the software partition could then be configured to return control directly to the co-processor, without performing any additional processing. Executing the sequence produced altogether 10000000 trapped instructions. The time required for the sequence was measured, and it was 1713141 μs. Assuming that 13141 μs of the total time was the overhead caused by the loop and the time measurement system, the time translates into 17 clock cycles of 100 MHz per trapped instruction. The time required for control transfer in one direction could not be measured, since the time measurement has to be located in either the software partition or in the co-processor. Trying to communicate the reception of the control from one domain to another would cause too much overhead. Measuring the time for the entire cycle, moving control in one direction and back again, however can be done in either of the domains, since they are informed of when they have the control over the execution. As a reference, in [62], the time required for the control transfer to the software and back was also measured from version 0.05 of the REALJava. The time was then 1.781 μs per trap, over ten times as long as the time measured here. Most of the improvement comes from the use of the helper addresses mentioned in Section 5.8.

6.6 Preliminary Results for the Multicore REALJava

A simple multicore version of the REALJava virtual machine was designed in order to prove that the system is scalable. This version contains only rudimentary support for multiple co-processor cores in the software, and the co-processor core is exactly the same as in the unicore virtual machine. The results are thus only preliminary and several performance enhancing techniques will be developed in future research.

The benchmarks used in this section are slightly modified versions of the second set of benchmarks. Only the longer benchmarks were modified since the short ones do not contain enough processing for the multicore approach to show significant benefits. This is because the virtual machine startup sequence is executed as a single thread and only the user application is multithreaded. The startup time dominates in the shortest test, thus masking the effect of the multicore setup. The modifications are quite simple, just splitting the processing into two or more threads. For instance, in the *Mandel* benchmark the processing is divided by assigning even pixels in the image to one thread and odd pixels to another one. A third thread is used for control and output of the resulting image. Similar techniques were used for all the benchmarks presented here. Table 6.14 shows the execution times and the efficiency of the dualcore version in comparison to the unicore version.

	REALJava (dualcore)	**REALJava (unicore)**	**Efficiency**
Mandel	2690	5099	1.90
Salesman	6825	8878	1.30
Neural	171112	246990	1.44
RayTrace	5421	8228	1.52
Average			1.54

Table 6.14: Execution times (in milliseconds) for various benchmarks using multicore virtual machine.

The reported efficiency of the multicore REALJava virtual machine could also be increased by designing the benchmarks from the scratch to support multithreading. The current benchmarks are not well suited for the multithreaded execution model, especially if the number of threads is large. In the future work also new programming models suitable for multicore Java virtual machines will be developed.

6.7 Chapter Summary

The performance of the REALJava virtual machine was shown to be superior to all of the reference systems in most of the benchmarks. The results are summarized in Table 6.15, which shows the best reference system for each individual test, along with the scores of the test and the relative performance

of the REALJava virtual machine in that test. In the table SW stands for the software only version of the REALJava, followed by the version number. This shorthand notation was adopted in order to fit the table within the page borders. In the few benchmarks that the REALJava did not dominate, the difference was very small. Especially the Kaffe, a software only solution for the PowerPC, was clearly outperformed, even by the software only version of the REALJava virtual machine. The results support the initial assumptions of increased computational performance with reduced power consumption. The accesses to the physical memory were measured from all virtual machines running on the ML410 board. These results show a significant decrease when the co-processor is used. The preliminary results for the multicore version of the REALJava show that the secondary goal of avoiding decisions that would lead to problems in multicore environments was also attained.

	Best Reference	Reference Score	REALJava 2.09 (HW)	Relative Performance
byte array access	Nokia	962879	791975	82.25%
byte array copy	aJile aJ100	32768000	52428800	160.00%
int array access	aJile aJ100	1008246	789590	78.31%
int array copy	Kaffe	9362285	14563555	155.56%
byte add	Nokia	2724795	10000000	367.00%
byte sub	Nokia	2840909	9090909	320.00%
byte mul	Nokia	2695417	9090909	337.27%
byte div	Nokia	1694915	1960784	115.69%
int add	Nokia	3361344	14285714	425.00%
int sub	Nokia	3316749	12500000	376.88%
int mul	Nokia	3210272	14285714	445.00%
int div	Nokia	1917545	2083333	108.65%
float add	aJile aJ100	2985074	4545454	152.27%
float sub	aJile aJ100	2666666	4545454	170.45%
float mul	Nokia	1633986	6666666	408.00%
float div	aJile aJ100	1379310	2702702	195.95%
double add	aJile aJ100	1754385	578034	32.95%
double sub	aJile aJ100	1612903	540540	33.51%
double mul	Nokia	788643	460829	58.43%
double div	aJile aJ100	574712	163800	28.50%
string concat	Nokia	15467	13726	88.74%
string compare	Nokia	512820	392156	76.47%
method calls	Nokia	1069518	2857142	267.14%
object creations	Sun eSPOT	86206	645161	748.39%
T.L.E.	aJile aJ100	146536	302941	206.73%
Mandel	SW 1.00	35127	5325	659.66%
Fibonacci	SW 2.00	1127	433	260.28%
Life	SW 2.00	1601	571	280.39%
Text	SW 2.00	1429	518	275.87%
Salesman	SW 2.00	29437	12625	233.16%
Sort	SW 2.00	13961	4396	317.58%
Neural Net	SW 2.00	127553	64794	196.86%
RayTrace	SW 2.00	58507	12168	480.83%
RayTrace2	SW 2.00	29862	8349	357.67%
Mandel ME	Nokia	14098	4832	291.76%
Salesman ME	Nokia	10546	11597	90.94%
Sort ME	Nokia	55777	40342	138.26%
RayTrace ME	Nokia	9036	7891	114.51%

Table 6.15: The scores of the best reference systems in individual tests.

Chapter 7

Conclusions

This thesis proposed a co-processor approach for Java execution in embedded systems. The proposed approach was used to design a hardware accelerated Java virtual machine, called REALJava. The REALJava virtual machine was partitioned so that the platform specific parts were implemented in software while as much as possible of the actual Java bytecode processing was implemented in hardware. This partitioning scheme allows the co-processor to be used in a variety of environments, since the platform dependent features are assigned to easily portable software. Including the REALJava virtual machine to new systems is further facilitated by the separation of the communication from the rest of the virtual machine, both in hardware and software domains. Since all of the modifications to the Java virtual machine specification [38] are hidden inside the virtual machine, the Java programmers do not need to employ any special coding techniques. Also all existing Java applications are readily usable with the REALJava, without any modifications. From the viewpoint of the end users running the Java applications the REALJava virtual machine behaves as any other Java virtual machine. All of the variants of Java are currently based on the same instruction set, and it is likely that future variants will maintain this trend. New variants have so far consisted mainly of modifications to the standard library and the minimum requirements for the underlying system. If these assumptions hold the test of time, the REALJava virtual machine will follow suit.

The software partition was designed and developed in portable C++ code, without any assembler level optimizations that might hinder the adoption of new system architectures. An efficient stack based hardware architecture was designed to be used as the main execution engine in the co-processor. Initially the co-processor was designed targeting asynchronous ASIC technologies, but later the system was prototyped using FPGA tech-

nology. The transition from one technology to the other required several architectural modifications. The FPGA technology allowed fast paced, iterative design and evaluation of the REALJava virtual machine. The virtual machine was analyzed and evaluated in detail, using actual hardware instead of simulations. All of the performance metrics reported in this thesis have been obtained from actually running the REALJava virtual machine on the platforms specified in Chapter 4. The observations made here were reflected back to the specifications and design of the individual units. The measurement data gathered was also used to show the feasibility of the chosen strategies and technologies.

The co-processor was prototyped using Xilinx FPGAs and Xilinx tools. The co-processor approach was shown to be valid and the actual performance increased significantly, even though the co-processor was running at mere 100 MHz while as the software only virtual machines were running on the PowerPC CPU with 300 MHz clock rate. Also the number of memory accesses was greatly reduced when the co-processor executed the Java applications. Since a large portion of the power consumed while executing a Java program is used in the memory accesses the co-processor accelerated virtual machine clearly improved the energy efficiency as well as computational performance.

A number of performance increasing techniques were developed during the prototyping phase. Most notably the method invocation mechanism was under scrutiny. Also strategies to increase and exploit data locality were studied. Efficient caching of constant data in the co-processor core helped in reducing the amount of traps generated by a given Java application. The control of multiple threads in the time domain was assigned to the hardware, freeing the CPU of this time consuming task. The communication overhead was further reduced by including data forwarding services to the internal control register bank. These provide access to various data items so, that the CPU does not need to know the physical addresses of the data items. All of the presented techniques can be applied to other hardware based Java execution engines, provided that the surrounding architecture, internal services and the assignment of memory areas are similar to the REALJava. Some of the techniques that are not specifically related to Java and it's properties, like the stack caching, can be used in a wider spectrum of systems.

The performance of the REALJava virtual machine was compared to commercial hardware accelerated Java virtual machines and to software virtual machines running on embedded devices. A mobile phone was also used as a reference. The REALJava virtual machine outperformed all these ref-

erence systems. The best performing hardware reference system was aJile aJ100, a dedicated Java processor on full custom hardware. The aJ100 reference system runs at 103 MHz. Despite the slightly slower clock rate and the inherent inefficiency of FPGA technology when compared to a full custom VLSI chip, the REALJava was roughly five times faster in integer arithmetics. The REALJava virtual machine was a bit behind in the array accesses and more so in the double precision floating point operations. The overall score however shows that the aJ100 has less than half of the speed of the REALJava virtual machine.

A multicore version of the REALJava was briefly discussed, and some preliminary results were shown. The multicore version had two parallel co-processor cores and achieved roughly 50% performance gain on average. This result is actually better than expected, since the multicore support is still rudimentary. Also the benchmarks used here were simply divided into parallel threads. This is most likely not the optimal test to show the efficiency of a multicore system. However the results clearly show that the concept is valid, and definitely worth further study.

7.1 Current and Future Work

The next step in the evolution of the REALJava system is improving the multicore support. This will greatly speed up execution of multithreaded java applications as several threads can be executed in parallel, each using their own co-processor. The thread scheduling and co-processor assignment for threads will be re-evaluated. Also the communication between the co-processor cores will be investigated. Possibilities for hardware based thread synchronization will be studied. Finally programming models for the new truly parallel multicore hardware accelerated virtual machine will be developed. This will include analysis and modification of the standard library as well.

This system can be further improved by including multitasking inside the REALJava virtual machine. In the current solution each Java program requires a separate instance of the virtual machine software, thus using excessive resources. It is worth noticing that for instance the Java virtual machine implemented by Sun works the same way, starting a new fully separate instance of the virtual machine for every Java application. The multitasking approach is expected to obtain better results in allocation of the co-processors to the threads running in the systems. This is due to the fact that parallel virtual machine instances cannot evaluate each others load

characteristics. When the applications run inside the same virtual machine, the load evaluation is quite straightforward, leading to improved load balancing and thread allocation.

As a further improvement a three tier memory model will be considered. This model would have all the same data in the local memory of the co-processor, but some of the heap memory space of the CPU would be moved to the third memory space. This space would allow direct access from the co-processors, thus requiring no CPU intervention for object and array accesses. The impact of the third memory region will be more prominent as the number of cores is increased. With the current architecture the CPU would easily be overwhelmed by the object access requests. The three tier model can also provide new methods for synchronization of the object accesses, since it might be possible to move at least parts of the locking mechanisms to the new memory controller.

After the REALJava virtual machine has been fine tuned for the multitasking, an ASIC will be designed. The purpose of this ASIC is to see the performance impact of moving from the synchronous design space to the asynchronous. Also the effects of using FPGAs versus ASIC technology can be evaluated. The case study will include several co-processors, connected to each other and to the CPU by some network structure. The type of the network is open at this time, but as stated earlier, the REALJava virtual machine does not care about that anyway. The design will be implemented using the Haste [102] asynchronous synthesis tool set, which has been found [61] to provide good synthesis results.

One more interesting direction in the future of the REALJava virtual machine is the use of dynamic reconfiguration facilities in modern FPGAs to provide *hardware-on-demand*. The tool support for designing such systems is improving, for instance the PARBIT developed by Horta et al. [24] could be used. Xilinx FPGAs can be only partially reconfigured, allowing the operating system to request a new co-processor when needed. Of course the device must have enough space available at the time to be able to respond to the request. A system with several co-processor specifications stored to the memory could then instantiate a set of co-processors and replace some of them with others when needed. The result would be a power and cost efficient system with very high peak performance.

Finally, since the software only version of the REALJava will evolve together with the hardware accelerated version, it will be interesting to see, whether the software only approach can be scaled up to support multicore processors. Currently desktop computers have two or four CPU cores, and

the number is very likely to grow. Assigning the software partition of the hardware accelerated REALJava to one core and starting separate threads for execution engines in the other ones would utilize the resources more effectively than current single threaded Java virtual machines. The execution engine threads would perform the same functions as the co-processors in the hardware accelerated REALJava. Using the experiences gained with the multicore version of the REALJava will provide additional insight to the possible problems and also provide ways to deal with them.

Bibliography

[1] G. Acher, "JIFFY - Ein FPGA-basierter Java Just-in-Time Compiler für eingebettete Anwendungen", (JIFFY - A FPGA-Based Java Just-in-Time Compiler for Embedded Applications) Ph.D. thesis, Technische Universität München, Deutschland, 2003

[2] B. Alpern, S. Augart, S.M. Blackburn, M. Butrico, A. Cocchi, P Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, K.S. McKinley, M. Mergen, J.E.B. Moss, T. Ngo, V. Sarkar and M. Trapp, "The Jikes Research Virtual Machine project: Building an open-source research community", *In IBM Systems Journal*, Volume 44, Number 2, 2005

[3] L. Bacciarelli, G. Lucia, S. Saponara, L. Fanucci and M. Forliti, "Design, testing and prototyping of a software programmable I2C/SPI IP on AMBA bus", *In Proc. Ph.D. Research in Microelectronics and Electronics 2006*, June, 2006

[4] G. Back, "Isolation, Resource Management and Sharing in the KaffeOS Java Runtime System", Doctoral Dissertation, University of Utah, 2002

[5] G. Back, W.C. Hsieh and J. Lepreau, "Processes in KaffeOS: Isolation, Resource Management, and Sharing in Java", *In Proc. Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, U.S.A, October, 2000

[6] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin and M. Turnbull, "The Real-Time Specification for Java", Addison-Wesley, ISBN 0-201-70323-8, 2000

[7] C. Brunelli and J. Nurmi, "Coprocessor Approach to Accelerating Multimedia Applications", In *Processor Design: System-on-Chip Computing for ASICs and FPGAs*, J. Nurmi, Ed. Kluwer Academic Publishers / Springer Publishers, 2007, ch. 10, pp. 209-228, 2007

[8] Ken Chapman, "Get your Priorities Right - Make your Design Up to 50% Smaller", Xilinx white paper, WP275 (v1.0.1), October, 2007

[9] G. Chen, B. Kang, M. Kandemir, N. Vijaykrishnan, M.J. Irwin and R. Chandramouli, "Energy-aware compilation and execution in Java-enabled mobile devices", *In Proc. Parallel and Distributed Processing Symposium*, April, 2003

[10] G. Chen, B. Kang, M. Kandemir, N. Vijaykrishnan, M.J. Irwin and R. Chandramouli, "Studying energy trade offs in offloading computation/compilation in Java-enabled mobile devices", *In IEEE Transactions on Parallel and Distributed Systems*, Volume 15, Issue 9, September, 2004

[11] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson and M. Wolczko, "Compiling Java Just in Time", *In IEEE Micro*, Volume 17, Issue 3, May, 1997

[12] M. Debbabi, A. Mourad, C. Talhi and H. Yahyaoui, "Accelerating embedded Java for mobile devices", *In Communications Magazine*, Vol. 43, Issue 9, September 2005

[13] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten and E.F.M. Steffens, "On-the-fly garbage collection: an exercise in cooperation", *In Communications of the ACM*, Volume 21, Issue 11, November, 1978

[14] M. A. Ertl, "A Portable Forth Engine", *In Proc. EuroFORTH '93*, Marienbad, Czech Republic, 1993

[15] K.I. Farkas, J. Flinn, G. Back, D. Grunwald and J.M. Anderson, "Quantifying the Energy Consumption of a Pocket Computer and a Java Virtual Machine", *In ACM SIGMETRICS*, Volume 28, Issue 1, June, 2000

[16] Gagnon E, "A portable research framework for the execution of Java bytecode", PHD Thesis, McGill University, Montreal, December, 2002

[17] J Gaisler, "LEON Open-Source Processor", press release, Gaisler Research

[18] P. Garrault and B. Philofsky, "HDL Coding Practices to Accelerate Design Performance", Xilinx white paper, WP231 (1.1), January, 2006

[19] H. McGhan and M. OConnor, "PicoJava: a direct execution engine for Java bytecode", *In Computer*, Volume 31, Issue 10, October, 1998

[20] T.S. Hall and K.B. Kent, "Thread-level parallel execution in co-designed virtual machines", *In Proc. Rapid System Prototyping*, June, 2005

[21] D. Hardin, "Crafting a Java virtual machine in silicon", *In IEEE Instrumentation & Measurement Magazine*, Volume 4 , Issue 1, March, 2001

[22] M. Hejun, K. Kent and D. Luke, "An implementation of the hardware partition in a software/hardware co-designed Java virtual machine", *In Proc. Canadian Conference on Electrical and Computer Engineering*, Canada, May, 2004

[23] J. Hennessy and D. Patterson, "Computer Architecture: a Quantitative Approach", Second Edition, Morgan Kaufmann Publishers, Inc., 1996

[24] E. Horta, J. Lockwood and D. Parlour, "Dynamic Hardware Plugins in an FPGA with Partial Run-time Reconfiguration", *In Proc. DAC 2002*, New Orleans, Louisiana, USA, June, 2002

[25] C.-H.A. Hsieh, M.T. Conte, T.L. Johnson, J.C. Gyllenhaal and W.-M.W. Hwu, "A study of the cache and branch performance issues with running Java on current hardware platforms", *In Proc. Compcon '97*, San Jose, CA, USA, February, 1997

[26] International Telecommunication Union, Telecommunication Standardization Sector, "ITU-T Recommendation I.363.5: B-ISDN ATM Adaptation Layer specification: Type 5 AAL", August, 1996

[27] N. Irie, "NoTA on Multicore Platform", emphIn 1st International NoTA Conference, Helsinki, Finland, June, 2008

[28] S.A. Ito, L. Carro and R.P. Jacobi, "Making Java work for microcontroller applications", *In Design & Test of Computers*, Volume 18, Issue 5, 2001

[29] K.B. Kent and M. Serra, "Hardware/software co-design of a Java virtual machine", *In Proc. Rapid System Prototyping*, June, 2000

[30] K.B. Kent and M. Serra, "Hardware architecture for Java in a hardware/software co-design of the virtual machine", *In Proc. Digital System Design*, September, 2002

[31] K.B. Kent, M. Hejun and M. Serra, "Rapid prototyping of a co-designed Java virtual machine", *In Proc. Rapid System Prototyping*, June, 2004

[32] K.B. Kent, M. Serra and N. Horspool, "Hardware/software co-design for virtual machines", *In Proc. Computers and Digital Techniques*, September, 2005

[33] S. Lafond and J. Lilius, "An Energy Consumption Model for an Embedded Java Virtual Machine", *In Proc. Architecture of Computing Systems - ARCS 2006*, Frankfurt/Main, Germany, March, 2006

[34] S. Lafond and J. Lilius, "An Energy Consumption Model for Java Virtual Machine", TUCS Technical Report, no. 597, 2004

[35] Z. Li and R. Xu, "Energy impact of secure computation on a handheld device", *In Proc. IEEE International Workshop on Workload Characterization*, November, 2002

[36] Z. Liang, J. Plosila, and K. Sere, "Asynchronous Java Accelerator for Embedded Java Virtual Machine", *In Proc. of IEEE CAS Symposium on Emerging Technologies, Frontiers of Mobile and Wireless Communication*, Shanghai, China, June 2004

[37] P. Liljeberg, J. Plosila, and J. Isoaho, "Self-Timed Communication Platform for Implementing High-Performance Systems-on-Chip", *the VLSI Integration* Journal 38, Elsevier, 2004

[38] T. Lindholm and F. Yellin, "The Java Virtual Machine Specification", Second Edition, Addison-Wesley, 1997

[39] M.J.M. Ma, C.L. Wang, F.C.M. Lau and Z. Xu, "JESSICA: Java-Enabled Single-System-Image Computing Architecture", *In Proc. International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, Nevada, U.S.A., July, 1999

[40] G. Mathias and K.B. Kent, "An Embedded Java Virtual Machine Using Network-on-Chip Design", *In Proc. Rapid System Prototyping*, Canada, June, 2006

[41] J. Meyer and T. Downing, "Java Virtual Machine", O'Reilly & Associates, Inc., 1997

[42] D. Nicolaescu and A. Veidenbaum, "Understanding and comparing the performance of optimized JVMs", *In Innovative Architecture for Future Generation High-Performance Processors and Systems*, January, 2005

[43] I. Piumarta and F. Riccardi, "Optimizing Direct Threaded Code by Selective Inlining", *In Proc. PLDI '98*, New York, NY, U.S.A., 1998

[44] R. Radhakrishnan, J. Rubio and L.K. John, "Characterization of Java applications at bytecode and ultra-SPARC machine code levels", *In Proc. International Conference on Computer Design*, Austin, TX, U.S.A., October, 1999

[45] R. Radhakrishnan, N. Vijaykrishnan, L.K. John and A. Sivasubramaniam, "Architectural issues in Java runtime system", *In Proc. International Symposium on High-Performance Computer Architecture*, Toulouse, France, January, 2000

[46] R. Radhakrishnan, N. Vijaykrishnan, L.K. John, A. Sivasubramaniam, J. Rubio, and J. Sabarinathan, "Java runtime systems: characterization and architectural implications", *In IEEE Transactions on Computers*, Volume 50, Issue 2, February, 2001

[47] P. Seshadri and A. Mericas, "Workload characterization of multithreaded Java servers on two PowerPC processors", *In IEEE International Workshop on Workload Characterization*, December, 2001

[48] M. Schöberl, "JOP: A Java Optimized Processor for Embedded Real-Time Systems", Ph.D. thesis, Vienna University of Technology, Austria, 2005

[49] T. Smith, S. Srinivas, P. Tomsich and J. Park, "Experiences with retargeting the Java HotSpot virtual machine", *In Proc. Parallel and Distributed Processing Symposium*, 2002

[50] J. Sparso and S. Furber, "Principles of Asynchronous Circuit Design - A System Perspective", Kluwer Academic Publishers, 2001

[51] O. Strom, A. Klauseie and E.J. Aas, "A study of dynamic instruction frequencies in byte compiled Java programs", *In Proc. EUROMICRO Conference*, Milan, Italy, 1999

[52] T. Säntti and J. Plosila, "Communication Scheme for an Advanced Java Co-Processor", *In Proc. Norchip 2004*, Oslo, Norway, November, 2004

[53] T. Säntti and J. Plosila, "Architecture for an Advanced Java Co-Processor", *In Proc. ISSCS 2005*, Iasi, Romania, July, 2005

[54] T. Säntti and J. Plosila, "Instruction Folding for an Asynchronous Java Co-Processor", *In Proc. 2005 International Symposium of System-on-Chip*, Tampere, Finland, November, 2005

[55] T. Säntti and J. Plosila, "Real Time Flow Control for an Advanced Java Co-Processor", *In Proc. Norchip 2005*, Oulu, Finland, November, 2005

[56] T. Säntti, J. Tyystjärvi and J. Plosila, "Java Co-Processor for Embedded Systems", In *Processor Design: System-on-Chip Computing for ASICs and FPGAs*, J. Nurmi, Ed. Kluwer Academic Publishers / Springer Publishers, ch. 13, pp. 287-308, 2007

[57] T. Säntti, J. Tyystjärvi and J. Plosila, "FPGA Prototype of the REALJava Co-Processor", *In Proc. 2007 International Symposium of System-on-Chip*, Tampere, Finland, November, 2007

[58] T. Säntti, J. Tyystjärvi and J. Plosila, "A Novel Hardware Acceleration Scheme for Java Method Calls", *In Proc. ISCAS*, Seattle, Washington, USA, May, 2008

[59] 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" *In Proc. Computers and Digital Techniques*, January, 2006

[60] P. Tullmann, M. Hibler and J. Lepreau, "Janos: A Java-oriented OS for Active Networks", *In IEEE Journal on Selected Areas of Communication*,Volume 19, Number 3, March, 2001

[61] J. Tuominen, T. Säntti and J. Plosila, "Comparative Study of Synthesis for Asynchronous and Synchronous Cache Controllers", *In Proc. Norchip 2006*, Linkping, Sweden, November, 2006

[62] J. Tyystjärvi, "A Virtual Machine for Embedded Systems with a Co-Processor", M.Sc. Thesis, University of Turku, 2007

[63] B.S. Yang, S.M. Moon, S. Park, J. Lee, S. Lee, J. Park, Y.C. Chung, S. Kim, K. Ebcioglu and E. Altman, "LaTTe: A Java VM Just-in-Time Compiler with Fast and Efficient Register Allocation", *In Proc. Parallel Architectures and Compilation Techniques*, Newport Beach, CA, U.S.A., October, 1999

[64] M. Zabel, T. Preuber, P. Reichel and R. Spallek, "Secure, Real-Time and Multi-Threaded General-Purpose Embedded Java Microarchitecture", *In Proc. Digital System Design Architectures, Methods and Tools*, August, 2007

[65] J. Tuominen, Course Material for "SoC Design" , University of Turku, Department of Information Technology, 2008

[66] T. Säntti, Course Material for "Embedded Virtual Machines on FPGAs", University of Turku, Department of Information Technology, 2007

[67] "Sun Microsystems, Connected Device Configuration", Specification Version 1.0b, December 2005

[68] "Sun Microsystems, Connected Limited Device Configuration", Specification Version 1.1.2, August 2006

[69] "Sun Microsystems, Connected Limited Device Configuration", Specification Version 1.1.1, November 2007

[70] "PowerPC 405 Processor Block Reference Guide", UG018 (v2.2), Xilinx documentation, June, 2007

[71] "Xtensa-V Configurable Processor", Alliance Core Product Specification, October, 2002

[72] "MicroBlaze, An Enhanced 32-Bit Processor Core for FPGA Integration", Xilinx documentation

[73] "LEON2 XST User's Manual", Version 1.0.22, Gaisler Research, May, 2004

[74] "Imsys IM1000 - a Multimedia Platform for Java Applications", Product Brief, Imsys Technologies

[75] "XSA-3S1000 Board V1.1 User Manual", X Engineering Software Systems Corporation product manual, September, 2007

[76] "XStend Board V3.0 Manual", X Engineering Software Systems Corporation product manual, March, 2005

[77] "Spartan-3 Generation FPGA User Guide", Xilinx documentation, UG331 (v1.3), February, 2008

[78] "Spartan-3 FPGA Family: Complete Data Sheet", DS099 (v2.3), November, 2007

[79] "ML310 User Guide Virtex-II Pro Embedded Development Platform", Xilinx documentation, UG068 (v1.1.5), February, 2007

[80] "Virtex-II Pro and Virtex-II Pro X FPGA User Guide", Xilinx documentation, UG012 (v4.2), November, 2007

[81] "Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet", Xilinx documentation, DS083 (v4.7), November, 2007

[82] "ML410 Embedded Development Platform User Guide", Xilinx documentation, UG085 (v1.7), September, 2007

[83] “Virtex-4 FPGA User Guide”, Xilinx documentation, UG070 (v2.4), April, 2008

[84] “Virtex-4 Family Overview”, Xilinx documentation, DS112 (v3.0), September, 2007

[85] “ISE 9.1i Release Notes and Installation Guide”, Xilinx documentation

[86] “XST User Guide 9.1i”, Xilinx documentation

[87] “Embedded System Tools Reference Manual - Embedded Development Kit (EDK 9.1i)”, Xilinx documentation, UG111 (v7.0), January, 2007

[88] “M1535D+ South Bridge For Multimedia Desktop PCs”, Product Brief, Document Number: 1535PBdp1.doc, Acer Labs

[89] “IBM CoreConnect bus cores”, IBM documentation, 2006

[90] “Sun Small Programmable Object Technology (Sun SPOT) Theory of Operation”, Sun Labs, Part No. 820-1248-10, Revision 1.0.8, May, 2007

[91] “Microchip Technology Inc.”, consulted 15 July 2008, URL: *http://www.microchip.com/*

[92] “GNU Classpath - GNU Project - Free Software Foundation (FSF)”, consulted 17 June 2008, URL: *http://www.gnu.org/software/classpath/*

[93] “Japhar - The Hungry Java Runtime”, consulted 02 June 2008, URL: *http://www.hungry.com/old-hungry/products/japhar/*

[94] “Open Runtime Platform”, consulted 02 June 2008, URL: *http://orp.sourceforge.net/*

[95] “Welcome to SourceForge.net”, consulted 04 June 2008, URL: *http://sourceforge.net/*

[96] “SableVM Project”, consulted 02 June 2008, URL: *http://www.sablevm.org/*

[97] “Kaffe.org”, consulted 04 June, 2008, URL: *http://www.kaffe.org/*

[98] “NoTA”, consulted 23 August, 2008, URL: *http://www.notaworld.org/*

[99] "Java Native Interface Specification", consulted 12 June 2008, URL: *http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html*

[100] "Java 2 Platform Specification, version 1.4.2", consulted 12 June 2008, URL: *http://java.sun.com/j2se/1.4.2/docs/api/*

[101] "Java ME APIs & Docs", consulted consulted 12 June 2008, URL: *http://java.sun.com/javame/reference/apis.jsp*

[102] "Handshake Solutions", consulted 12 June 2008, URL: *http://www.handshakesolutions.com/*

[103] "ARM Jazelle Technology", consulted 12 June 2008, URL: *http://www.arm.com/products/esd/jazelle_home.html*

[104] "Java EE at a Glance", consulted 05 June 2008, URL: *http://java.sun.com/javaee/*

[105] "Java SE - Overview - at a Glance", consulted 05 June 2008, URL: *http://java.sun.com/javase/*

[106] "The Java ME Platform - the Most Ubiquitous Application Platform for Mobile Devices", consulted 05 June 2008, URL: *http://java.sun.com/javame/*

[107] "Java Card Technology", consulted 05 June 2008, URL: *http://java.sun.com/javacard/index.jsp*

[108] "FPGA and CPLD Solutions from Xilinx, Inc.", consulted 10 June 2008, URL: *http://www.xilinx.com/*

[109] "CaffeineMark 3.0 Benchmark Information", consulted 12 June 2008, URL: *http://www.benchmarkhq.ru/cm30/info.html*

[110] "JavaG Benchmarking", consulted 12 June 2008, URL: *http://www2.epcc.ed.ac.uk/computing/research_activities/java_grande/*

[111] "BenchMark Results" consulted 12 June 2008, URL: *http://vco.ett.utu.fi/~teansa/REALResults.*

[112] "Embedded Java Book Index", consulted 12 June 2008, URL: *http://www.practicalembeddedjava.com/.*

[113] "iPAQ - Wikipedia, the free encyclopedia" consulted 15 April 2008, URL: *http://en.wikipedia.org/wiki/IPAQ*

[114] "IBM Semiconductor solutions | Power Architecture | Licensing | Coreconnect", consulted 13 May 2008,
URL: *http://www-03.ibm.com/technology/ges/semiconductor/power/licensing/coreconnect/index.html*

[115] "AMBA Overview", consulted 13 May 2008,
URL: *http://www.arm.com/products/solutions/AMBAHomePage.html*

[116] "Home Page of Jad - the fast Java decompiler", consulted 27 May 2008, URL: *http://www.kpdus.com/jad.html*

Appendix A

Java Instruction Set

This appendix presents the Java bytecode instruction set. Besides the opcodes and their meaning, also some ancillary data is give here. This includes the grouping of the instructions, which is done according to the classes presented in Chapter 3 while discussing the instruction folding. Additional groupings are added to show which proprietary opcodes are executed in software (SW) or both software and hardware (SW/HW). Also a sub group denoted by C is added to show the instructions that are cacheable.

The Table A.1 also contains information about the data traffic associated with the instructions. The column titled I contains the number of parameter bytes read from the instruction stream. The push and pop columns show how many stack operations are performed by the instruction. The local column lists accesses to the local variable area of the current method. The const column shows the instruction that perform reads from the constant pool. Finally the heap column lists accesses to the data contained in the heap memory.

Opcode	Mnemonic	I	Description	Pop	Push	Local	Const	Heap	Group
0 (00)	nop		Do not operate						NF
1 (01)	aconst_null		Push null object		1				LV
2 (02)	iconst_m1		Push integer constant -1		1				LV
3 (03)	iconst_0		Push integer constant 0		1				LV
4 (04)	iconst_1		Push integer constant 1		1				LV
5 (05)	iconst_2		Push integer constant 2		1				LV
6 (06)	iconst_3		Push integer constant 3		1				LV
7 (07)	iconst_4		Push integer constant 4		1				LV
8 (08)	iconst_5		Push integer constant 5		1				LV
9 (09)	lconst_0		Push long integer constant 0		2				Trap
10 (0a)	lconst_1		Push long integer constant 1		2				Trap
11 (0b)	fconst_0		Push float constant 0.0		1				LV
12 (0c)	fconst_1		Push float constant 1.0		1				LV
13 (0d)	fconst_2		Push float constant 2.0		1				LV
14 (0e)	dconst_0		Push double float 0.0		2				Trap
15 (0f)	dconst_1		Push double float 1.0		2				Trap
16 (10)	bipush	1	Push 1-byte integer		1				LV
17 (11)	sipush	2	Push 2-byte integer		1				LV
18 (12)	ldc	1	Load constant from the constant pool		1		1		Trap
19 (13)	ldc_w	2	Load constant from constant pool using a wider offset (16-bit index)		1		1		Trap
20 (14)	ldc2_w	2	Load long or double from constant pool		2		2		Trap
21 (15)	iload	1	Load local integer variable		1	1			LV
22 (16)	lload	1	Load local long variable		2	2			NF
23 (17)	fload	1	Load local float variable		1	1			LV
24 (18)	dload	1	Load local double float variable		2	2			NF
25 (19)	aload	1	Load local object variable		1	1			LV
26 (1a)	iload_0		Load local variable 0		1	1			LV

Opcode	Mnemonic	I	Description	Pop	Push	Local	Const	Heap	Group
27 (1b)	iload_1		Load local variable 1		1	1			LV
28 (1c)	iload_2		Load local variable 2		1	1			LV
29 (1d)	iload_3		Load local variable 3		1	1			LV
30 (1e)	lload_0		Load local long variable 0		2	2			NF
31 (1f)	lload_1		Load local long variable 1		2	2			NF
32 (20)	lload_2		Load local long variable 2		2	2			NF
33 (21)	lload_3		Long local long variable 3		2	2			NF
34 (22)	fload_0		Load local float variable 0		1	1			LV
35 (23)	fload_1		Load local float variable 1		1	1			LV
36 (24)	fload_2		Load local float variable 2		1	1			LV
37 (25)	fload_3		Load local float variable 3		1	1			LV
38 (26)	dload_0		Load local double variable 0		2	2			NF
39 (27)	dload_1		Load local double variable 1		2	2			NF
40 (28)	dload_2		Load local double variable 2		2	2			NF
41 (29)	dload_3		Load local double variable 3		2	2			NF
42 (2a)	aload_0		Load local object variable 0		1	1			LV
43 (2b)	aload_1		Load local object variable 1		1	1			LV
44 (2c)	aload_2		Load local object variable 2		1	1			LV
45 (2d)	aload_3		Load local object variable 3		1	1			LV
46 (2e)	iaload		Load integer from array	2	1			1	Trap
47 (2f)	laload		Load long from array	2	2			2	Trap
48 (30)	faload		Load float from array	2	1			1	Trap
49 (31)	daload		Load double from array	2	2			2	Trap
50 (32)	aaload		Load object from array	2	1			1	Trap
51 (33)	baload		Load signed byte from array	2	1			1	Trap
52 (34)	caload		Load character from array	2	1			1	Trap
53 (35)	saload		Load short from array	2	1			1	Trap
54 (36)	istore	1	Store integer into local variable	1		1			MEM

Opcode	Mnemonic	I	Description	Pop	Push	Local	Const	Heap	Group
55 (37)	lstore	1	Store long into local variable	2		2			NF
56 (38)	fstore	1	Store float into local variable	1		1			MEM
57 (39)	dstore	1	Store double into local variable	2		2			NF
58 (3a)	astore	1	Store object into local variable	1		1			MEM
59 (3b)	istore_0		Store integer into local variable 0	1		1			MEM
60 (3c)	istore_1		Store integer into local variable 1	1		1			MEM
61 (3d)	istore_2		Store integer into local variable 2	1		1			MEM
62 (3e)	istore_3		Store integer into local variable 3	1		1			MEM
63 (3f)	lstore_0		Store long into local variable 0	2		2			NF
64 (40)	lstore_1		Store long into local variable 1	2		2			NF
65 (41)	lstore_2		Store long into local variable 2	2		2			NF
66 (42)	lstore_3		Store long into local variable 3	2		2			NF
67 (43)	fstore_0		Store float into local variable 0	1		1			MEM
68 (44)	fstore_1		Store float into local variable 1	1		1			MEM
69 (45)	fstore_2		Store float into local variable 2	1		1			MEM
70 (46)	fstore_3		Store float into local variable 3	1		1			MEM
71 (47)	dstore_0		Store double into local variable 0	2		2			NF
72 (48)	dstore_1		Store double into local variable 1	2		2			NF
73 (49)	dstore_2		Store double into local variable 2	2		2			NF
74 (4a)	dstore_3		Store double into local variable 3	2		2			NF
75 (4b)	astore_0		Store object into local variable 0	1		1			MEM
76 (4c)	astore_1		Store object into local variable 1	1		1			MEM
77 (4d)	astore_2		Store object into local variable 2	1		1			MEM
78 (4e)	astore_3		Store object into local variable 3	1		1			MEM
79 (4f)	iastore		Store integer into array	3				1	Trap
80 (50)	lastore		Store long into array	4				2	Trap
81 (51)	fastore		Store float into array	3				1	Trap
82 (52)	dastore		Store double into array	4				2	Trap

Opcode	Mnemonic	I	Description	Pop	Push	Local	Const	Heap	Group
83 (53)	aastore		Store object into array	3				1	Trap
84 (54)	bastore		Store signed byte into array	3				1	Trap
85 (55)	castore		Store character into array	3				1	Trap
86 (56)	sastore		Store short into array	3				1	Trap
87 (57)	pop		Pop top entry in stack	1					NF
88 (58)	pop2		Pop top two entries in stack	2					NF
90 (5a)	dup_x1		Duplicate top word, put two down	2	3				NF
91 (5b)	dup_x2		Duplicate top word, put three down	3	4				NF
92 (5c)	dup2		Duplicate top two words	2	4				NF
93 (5d)	dup2_x1		Duplicate top two words, put three down	3	5				NF
94 (5e)	dup2_x2		Duplicate top two words, put four down	4	6				NF
95 (5f)	swap		Swap top two stack words	2	2				NF
96 (60)	iadd		Add integer	2	1				OP2
97 (61)	ladd		Add long	4	2				Trap
98 (62)	fadd		Add float	2	1				OP2
99 (63)	dadd		Add double	4	2				Trap
100 (64)	isub		Subtract integer	2	1				OP2
101 (65)	lsub		Subtract long	4	2				Trap
102 (66)	fsub		Subtract float	2	1				OP2
103 (67)	dsub		Subtract double	4	2				Trap
104 (68)	imul		Multiply integer	2	1				OP2
105 (69)	lmul		Multiply long	4	2				Trap
106 (6a)	fmul		Multiply float	2	1				OP2
107 (6b)	dmul		Multiply double	4	2				Trap
108 (6c)	idiv		Divide integer	2	1				OP2
109 (6d)	ldiv		Divide long	4	2				Trap
110 (6e)	fdiv		Divide float	2	1				OP2

Opcode	Mnemonic	I	Description	Pop	Push	Local	Const	Heap	Group
111 (6f)	ddiv		Divide double	4	2				Trap
112 (70)	irem		Compute integer remainder	2	1				OP2
113 (71)	lrem		Compute long remainder	4	2				Trap
114 (72)	frem		Compute float remainder	2	1				Trap
115 (73)	drem		Compute double remainder	4	2				Trap
116 (74)	ineg		Negate integer	1	1				OP1
117 (75)	lneg		Negate long	2	2				Trap
118 (76)	fneg		Negate float	1	1				OP1
119 (77)	dneg		Negate double	2	2				Trap
120 (78)	ishl		Shift left integer	2	1				OP2
121 (79)	lshl		Shift left long	3	2				Trap
122 (7a)	ishr		Arithmetic shift right integer	2	1				OP2
123 (7b)	lshr		Arithmetic shift right long	3	2				Trap
124 (7c)	iushr		Logical shift right integer	2	1				OP2
125 (7d)	lushr		Logical shift-right of a long	3	2				Trap
126 (7e)	iand		Compute bitwise AND	2	1				OP2
127 (7f)	land		Compute long bitwise AND	4	2				Trap
128 (80)	ior		Compute integer bitwise OR	2	1				OP2
129 (81)	lor		Compute long bitwise OR	4	2				Trap
130 (82)	ixor		Compute integer bitwise XOR	2	1				OP2
131 (83)	lxor		Compute long bitwise XOR	4	2				Trap
132 (84)	iinc	2	Increment local variable by constant			2			NF
133 (85)	i2l		Convert integer to long	1	2				Trap
134 (86)	i2f		Convert integer to float	1	1				Trap
135 (87)	i2d		Convert integer to double	1	2				Trap
136 (88)	l2i		Convert long to integer	2	1				Trap
137 (89)	l2f		Convert long to float	2	1				Trap
138 (8a)	l2d		Convert long to double	2	2				Trap

Opcode	Mnemonic	I	Description	Pop	Push	Local	Const	Heap	Group
139 (8b)	f2i		Convert float to integer	1	1				Trap
140 (8c)	f2l		Convert float to long	1	2				Trap
141 (8d)	f2d		Convert float to double	1	2				Trap
142 (8e)	d2i		Convert double to integer	2	1				Trap
143 (8f)	d2l		Convert double to long	2	2				Trap
144 (90)	d2f		Convert double to float	2	1				Trap
145 (91)	i2b		Convert integer to byte	1	1				OP1
146 (92)	i2c		Convert integer to character	1	1				OP1
147 (93)	i2s		Convert integer to short	1	1				OP1
148 (94)	lcmp		Compare long	4	1				Trap
149 (95)	fcmpl		Compare two floats (-1 if NaN)	2	1				OP2
150 (96)	fcmpg		Compare two floats (1 if NaN)	2	1				OP2
151 (97)	dcmpl		Compare two doubles (-1 if NaN)	4	1				Trap
152 (98)	dcmpg		Compare two doubles (1 if NaN)	4	1				Trap
153 (99)	ifeq	2	Branch if equal to 0	1					OP1_B
154 (9a)	ifne	2	Branch if not equal to 0	1					OP1_B
155 (9b)	iflt	2	Branch if less than 0	1					OP1_B
156 (9c)	ifge	2	Branch if greater than or equal 0	1					OP1_B
157 (9d)	ifgt	2	Branch if greater than 0	1					OP1_B
158 (9e)	ifle	2	Branch if less than or equal 0	1					OP1_B
159 (9f)	if_icmpeq	2	Compare top two stack items, branch if =	2					OP2_B
160 (a0)	if_icmpne	2	Compare top two stack items, branch if ! =	2					OP2_B
161 (a1)	if_icmplt	2	Compare top two stack items, branch if <	2					OP2_B
162 (a2)	if_icmpge	2	Compare top two stack items, branch if >=	2					OP2_B

Opcode	Mnemonic	I	Description	Pop	Push	Local	Const	Heap	Group
163 (a3)	if_icmpgt	2	Compare top two stack items, branch if >	2					OP2_B
164 (a4)	if_icmple	2	Compare top two stack items, branch if <=	2					OP2_B
165 (a5)	if_acmpeq	2	Compare top two stack objects, branch if =	2					OP2_B
166 (a6)	if_acmpne	2	Compare top two stack objects, branch if ! =	2					OP2_B
167 (a7)	goto	2	Branch always						NF
168 (a8)	jsr	2	Jump to subroutine		1				Trap
169 (a9)	ret	1	Return control to the PC stored in local variable index			1			Trap
170 (aa)	tableswitch	...	Access jump table by index & jump	1					Trap
171 (ab)	lookupswitch	...	Access jump table by match & jump	1					Trap
172 (ac)	ireturn		Return integer from procedure		1[1]				OP1_B
173 (ad)	lreturn		Return long from procedure		2[1]				NF
174 (ae)	freturn		Return float from procedure		1[1]				OP1_B
175 (af)	dreturn		Return double from procedure		2[1]				NF
176 (b0)	areturn		Return object from procedure		1[1]				OP1_B
177 (b1)	return		Return void from procedure						NF
178 (b2)	getstatic	2	Get static field value		1/2		1	1/2	Trap
179 (b3)	putstatic	2	Set static field in class	1/2			1	1/2	Trap
180 (b4)	getfield	2	Get field value	1	1/2		1	1/2	Trap
181 (b5)	putfield	2	Set field in class	2/3			1	1/2	Trap
182 (b6)	invokevirtual	2	Call method based on object	...[2]					Trap
183 (b7)	invokespecial	2	Call method not based on object	...[2]					Trap

[1]Pushed to return frame

[2]Not necessarily popped; stack pointer moved for return

Opcode	Mnemonic	I	Description	Pop	Push	Local	Const	Heap	Group
184 (b8)	invokestatic	2	Call a static method	...[2]					Trap
185 (b9)	invokeinterface	4	Call an interface method	...[2]					Trap
186 (ba)	Unused								
187 (bb)	new	2	Create new object		1		1	...	Trap
188 (bc)	newarray	1	Allocate an array	1	1			...	Trap
189 (bd)	anewarray	2	Allocate an array of objects	1	1		1	...	Trap
190 (be)	arraylength		Get length of array	1	1			1	Trap/C
191 (bf)	athrow		Throw an exception	1	1[3]				Trap
192 (c0)	checkcast	2	Ensure if object is of given type	1	1		1	...	Trap
193 (c1)	instanceof	2	Test if object is of given type	1	1		1	...	Trap
194 (c2)	monitorenter		Enter a monitored region of code	1					Trap
195 (c3)	monitorexit		Exit a monitored region of code	1					Trap
196 (c4)	wide	3/5	Extend local variable index	...	...	...			Trap
197 (c5)	multianewarray	3	Allocate a multidimensional array	...	1		1	...	Trap
198 (c6)	ifnull	2	Test if null	1					OP1_B
199 (c7)	ifnonnull	2	Test if not null	1					OP1_B
200 (c8)	goto_w	4	Branch always (wide index)						Trap
201 (c9)	jsr_w	4	Jump subroutine (4-byte offset)			1			Trap
202 (ca)	breakpoint		Reserved for debuging						Trap
203 (cb)	getfield_fast	2	Get field using offset	1	1			1	SW
204 (cc)	getfield_wide_fast	2	Get 2-word field using offset	1	2			2	SW
206 (ce)	putfield_fast	2	Put field using offset	2				1	SW
207 (cf)	putfield_wide_fast	2	Put 2-word field using offset	3				2	SW
208 (d0)	invokevirtual_fast	2	Invoke method using vtable lookup					1	SW
209 (d1)	getstatic_fast	2	Get 1-word static field using class and field index		1			1	SW

[3]Stack is cleared; possibly pushed in another frame

Opcode	Mnemonic	I	Description	Pop	Push	Local	Const	Heap	Group
210 (d2)	getstatic_wide_fast	2	Get 2-word static field using class and field index		2			2	SW
211 (d3)	putstatic_fast	2	Put 1-word static field using class and field index	1				1	SW
212 (d4)	putstatic_wide_fast	2	Put 2-word static field using class and field index	2				2	SW
213 (d5)	invokespecial_sw	2	Invoke native/synchronized method using pointer from constant pool				1		SW
214 (d6)	invokestatic_sw	2	Invoke native/synchronized static method using pointer from constant pool				1		SW
215 (d7)	new_fast	2	Allocate object using class pointer from constant pool		1		1	...	SW
216 (d8)	anewarray_fast	2	Allocate object array using class pointer from constant pool	1	1		1	...	SW
217 (d9)	multianewarray_fast	3	Allocate object multidim. array using class pointer from constant pool	...	1		1	...	SW
218 (da)	checkcast_fast	2	Verify instance using class pointer from constant pool	1	1		1	...	SW
219 (db)	instanceof_fast	2	Check instance using class pointer from constant pool	1	1		1	...	SW
220 (dc)	invokevirtual_nosub	2	Invoke non-overridden virtual method				1		HW/SW
221 (dd)	invokespecial_fast	2	Invoke normal bytecode method using pointer from constant pool				1		HW/SW
222 (de)	invokestatic_fast	2	Invoke normal bytecode static method using pointer from constant pool				1		HW/SW
223 (df)	invokevirtual_slow	2	Invoke virtual method using method lookup				1	...	SW

Opcode	Mnemonic	I	Description	Pop	Push	Local	Const	Heap	Group
224 (e0)	ldc_fast	1	Load constant that has been loaded before		1		1		SW/C
225 (e1)	ldc_w_fast	2	Load constant that has been loaded before with 16-bit constant pool index		1		1		SW
255 (ff)	slicer_trap		Trap on thread slicer timeout						Trap

Appendix B

Measurement Data

This Appendix tabulates all of the measurement data available, save for the results of the multicore version of the REALJava virtual machine and the results of the CaffeineMark benchmark suite. All of the Tables here have the same formatting. The first part of any given Table gives the results of the benchmark collection from "Practical Embedded Java" [112], using loops per second as the unit. The results are to be interpreted so that the higher the score, the higher the performance. The second part of a given table gives the execution times of the second benchmark set, as described in the Chapter 6. Since these values are the execution times (given in milliseconds), a lower the number means higher performance.

The Tables B.1 and B.2 show the results of the hardware reference systems, while the Table B.3 shows the results of the software reference systems. All of the software only versions of the REALJava virtual machine are shown in the Table B.4. The Tables B.5 though to B.9 give the results for the older versions [1] of the hardware accelerated REALJava virtual machine. Finally, the Table B.10 shows the measurement data for the mobile phone.

[1] The major changes between the versions are described in the Appendix C.

	DS80C390 1.11	DS80C400 1.11	Imsys Cjip 0.7.1	REALJava 2.09 (HW)
byte array access	3644	4068	106519	791975
byte array copy	655360	1448309	23831272	52428800
int array access	3031	3891	105109	789590
int array copy	165913	360087	6241523	14563555
byte add	11142	13542	263504	10000000
byte sub	11123	13514	262812	9090909
byte mul	10395	12189	241545	9090909
byte div	6015	5859	105042	1960784
int add	11337	13793	90867	14285714
int sub	11318	13765	90579	12500000
int mul	7877	8228	88183	14285714
int div	3225	2827	60864	2083333
float add	2525	2149	143781	4545454
float sub	2220	1871	137268	4545454
float mul	1836	1528	125234	6666666
float div	254	201	76804	2702702
double add	3254	3104	96993	578034
double sub	2859	2657	93896	540540
double mul	1989	1755	64040	460829
double div	259	207	34506	163800
string concat	45	46	247	13726
string compare	1320	1756	5390	392156
method calls	5566	5556	14729	2857142
object creations	558	496	3635	645161
T.L.E.	555	492	11557	302941
Mandel				5325
Fibonacci				433
Life				571
Text				518
Salesman				12625
Sort				4396
Neural Net				64794
RayTrace				12168
RayTrace2				8349
Mandel ME				4832
Salesman ME				11597
Sort ME				40342
RayTrace ME				7891

Table B.1: Measurement data for the hardware reference systems, part I.

	aJile aJ80 3.16.07	aJile aJ100 3.16.07	Sun eSPOT 2006Aug25	REALJava 2.09 (HW)
byte array access	206575	879677	479239	791975
byte array copy	4443118	32768000	6241523	52428800
int array access	233224	1008246	432580	789590
int array copy	1092266	7281777	1927529	14563555
byte add	793650	2702702	1438848	10000000
byte sub	803212	2777777	1449275	9090909
byte mul	561797	1369863	1428571	9090909
byte div	561797	1360544	1190476	1960784
int add	641025	2898550	1724137	14285714
int sub	641025	2898550	1724137	12500000
int mul	478468	1587301	1652892	14285714
int div	621118	1351351	1369863	2083333
float add	735294	2985074	250000	4545454
float sub	847457	2666666	246002	4545454
float mul	826446	1408450	199401	6666666
float div	561797	1379310	100452	2702702
double add	595238	1754385	226500	578034
double sub	813008	1612903	215517	540540
double mul	377358	581395	193236	460829
double div	311526	574712	70348	163800
string concat	879	3711	533	13726
string compare	64935	270270	67340	392156
method calls	271739	847457	706713	2857142
object creations	17331	36101	86206	645161
T.L.E.	41106	146536	34553	302941
Mandel				5325
Fibonacci				433
Life				571
Text				518
Salesman				12625
Sort				4396
Neural Net				64794
RayTrace				12168
RayTrace2				8349
Mandel ME				4832
Salesman ME				11597
Sort ME				40342
RayTrace ME				7891

Table B.2: Measurement data for the hardware reference systems, part II.

	Kaffe ML310	Kaffe ML410	REALJava 2.00 (SW)	REALJava 2.09 (HW)
byte array access	92597	98773	405168	791975
byte array copy	26214400	26214400	18724571	52428800
int array access	84617	88622	468114	789590
int array copy	8738133	9362285	10082461	14563555
byte add	214362	224215	1098901	10000000
byte sub	215285	225733	1104972	9090909
byte mul	171379	174064	1005025	9090909
byte div	209424	219298	995024	1960784
int add	345423	355239	1388888	14285714
int sub	345423	326797	1388888	12500000
int mul	344234	371057	1351351	14285714
int div	326264	351493	1183431	2083333
float add	271002	303030	675675	4545454
float sub	265604	296735	655737	4545454
float mul	277777	312012	696864	6666666
float div	236966	251889	500000	2702702
double add	255102	270635	277777	578034
double sub	246305	257731	274348	540540
double mul	178890	183654	252206	460829
double div	121065	124610	126182	163800
string concat	220	265	1320	13726
string compare	10893	15372	142857	392156
method calls	32690	59488	425531	2857142
object creations	11013	18761	86206	645161
T.L.E.	12794	15221	73057	302941
Mandel	144829	134878	36784	5325
Fibonacci	5522	4545	1127	433
Life	9705	7804	1601	571
Text	9455	7489	1429	518
Salesman	136079	115769	29437	12625
Sort	142110	106285	13961	4396
Neural Net	748649	590552	127553	64794
RayTrace	268908	229174	58507	12168
RayTrace2	223918	177037	29862	8349
Mandel ME		126686	23867	4832
Salesman ME		116063	20148	11597
Sort ME		893590	79478	40342
RayTrace ME		168745	17223	7891

Table B.3: Measurement data for the software reference systems.

	REALJava 0.01 (SW)	REALJava 1.00 (SW)	REALJava 2.00 (SW)	REALJava 2.09 (HW)
byte array access	32768	297890	405168	791975
byte array copy	38347922	20164923	18724571	52428800
int array access	32768	292571	468114	789590
int array copy	11671106	13107200	10082461	14563555
byte add	83333	1123595	1098901	10000000
byte sub	71428	1123595	1104972	9090909
byte mul	83333	1117318	1005025	9090909
byte div	71428	995024	995024	1960784
int add	100000	1369863	1388888	14285714
int sub	100000	1369863	1388888	12500000
int mul	100000	1351351	1351351	14285714
int div	100000	1190476	1183431	2083333
float add	83333	344827	675675	4545454
float sub	100000	334448	655737	4545454
float mul	100000	451467	696864	6666666
float div	83333	347826	500000	2702702
double add	55555	143575	277777	578034
double sub	55555	141442	274348	540540
double mul	62500	150943	252206	460829
double div	50000	94607	126182	163800
string concat	333	1128	1320	13726
string compare	7142	44444	142857	392156
method calls	18181	265251	425531	2857142
object creations	10000	32626	86206	645161
T.L.E.			73057	302941
Mandel	486676	35127	36784	5325
Fibonacci	3858	1496	1127	433
Life	7931	2312	1601	571
Text	7107	2049	1429	518
Salesman	325644	34142	29437	12625
Sort	237617	21934	13961	4396
Neural Net	1049729		127553	64794
RayTrace		101891	58507	12168
RayTrace2		34836	29862	8349
Mandel ME			23867	4832
Salesman ME			20148	11597
Sort ME			79478	40342
RayTrace ME			17223	7891

Table B.4: Measurement data for the software only versions of REALJava.

	REALJava 0.01 (HW)	REALJava 0.02 (HW)	REALJava 0.03 (HW)	REALJava 0.04 (HW)
byte array access	262144	146777	149369	157633
byte array copy	44739242	43690666	52428800	52428800
int array access	123361	146612	149114	157349
int array copy	12201611	11915636	11915636	11915636
byte add	1200000	1351351	1459854	1923076
byte sub	1200000	1388888	1515151	2000000
byte mul	1200000	1351351	1470588	1923076
byte div	162162	877192	925925	1086956
int add	1428571	1626016	1724137	2325581
int sub	1428571	1680672	1785714	2439024
int mul	1428571	1626016	1724137	2325581
int div	188679	1005025	1036269	1234567
float add	142857	1324503	1388888	1754385
float sub	136363	1360544	1428571	1818181
float mul	142857	1449275	1538461	2000000
float div	125000	1129943	1176470	1428571
double add	57692	80289	80160	80645
double sub	51724	79428	79239	79617
double mul	63829	73072	73019	73340
double div	45454	52548	52465	52590
string concat	500	479	485	483
string compare	9090	11173	10869	10989
method calls	24793	23906	24366	23696
object creations	6666	7082	7275	7267
T.L.E.				
Mandel	38431	33287	31166	26414
Fibonacci	3384	3306	3289	3297
Life	6372	6329	6290	6289
Text	5643	5628	5587	5573
Salesman				
Sort				
Neural Net				
RayTrace				
RayTrace2				
Mandel ME				
Salesman ME				
Sort ME				
RayTrace ME				

Table B.5: Measurement data for the older versions of REALJava, part I.

	REALJava 0.05 (HW)	REALJava 0.06 (HW)	REALJava 0.07 (HW)	REALJava 0.08 (HW)
byte array access	162418	171112	174413	179305
byte array copy	52428800	52428800	43690666	52428800
int array access	162017	171112	174297	179060
int array copy	13107200	13107200	13107200	13107200
byte add	2500000	2500000	3125000	3125000
byte sub	2500000	2500000	3125000	3125000
byte mul	2325581	2325581	2857142	2857142
byte div	1204819	1204819	1333333	1333333
int add	2941176	2941176	3448275	3448275
int sub	3125000	3125000	3703703	3703703
int mul	2941176	2941176	3448275	3448275
int div	1369863	1369863	1470588	1470588
float add	2083333	2083333	2272727	2272727
float sub	2173913	2173913	2380952	2380952
float mul	2409638	2409638	2702702	2702702
float div	1639344	1639344	1754385	1754385
double add	80677	83125	82918	82918
double sub	79808	82000	81799	81799
double mul	73152	80032	79808	79808
double div	52603	55370	56545	56609
string concat	489	504	501	501
string compare	11299	11627	11976	11976
method calls	23920	23688	24035	24035
object creations	7217	7283	7315	7473
T.L.E.				
Mandel	23104	22896	20874	20866
Fibonacci	3282	3278	3276	3258
Life	6271	6258	6196	6185
Text	5505	5494	5489	5481
Salesman	143841	138655	136697	134747
Sort	127445	126948	121693	123239
Neural Net	631251	592376	591292	591225
RayTrace				
RayTrace2				
Mandel ME				
Salesman ME				
Sort ME				
RayTrace ME				

Table B.6: Measurement data for the older versions of REALJava, part II.

	REALJava 1.00 (HW)	REALJava 1.01 (HW)	REALJava 2.00 (HW)	REALJava 2.01 (HW)
byte array access	238529	325240	331827	510007
byte array copy	20164923	20164923	18724571	17476266
int array access	234475	320469	326862	506069
int array copy	13107200	13107200	10922666	10922666
byte add	3125000	3125000	3125000	4000000
byte sub	3125000	3125000	3125000	3703703
byte mul	2857142	2857142	2857142	3448275
byte div	1333333	1333333	1333333	1449275
int add	3448275	3448275	3448275	4347826
int sub	3703703	3703703	3703703	4545454
int mul	3448275	3448275	3448275	4347826
int div	1470588	1470588	1470588	1587301
float add	2272727	2272727	2272727	2631578
float sub	2380952	2380952	2380952	2702702
float mul	2702702	2702702	2702702	3225806
float div	1754385	1754385	1754385	1923076
double add	111919	342465	353982	475059
double sub	93896	329489	338409	450450
double mul	106553	298062	305343	378071
double div	75272	136986	136798	154202
string concat	1407	1616	1930	2994
string compare	133333	166666	181818	222222
method calls	1754385	1754385	1754385	1818181
object creations	54347	62305	136054	350877
T.L.E.				
Mandel	16973	16817	16525	14444
Fibonacci	1466	1377	1143	1641
Life	2251	2022	1601	1984
Text	1953	1709	1392	1875
Salesman	52052	37496	36236	23952
Sort	14991	11153	10886	8380
Neural Net		180441	173345	121350
RayTrace	65164	43831	33454	34701
RayTrace2		19689	19342	14697
Mandel ME				
Salesman ME				
Sort ME				
RayTrace ME				

Table B.7: Measurement data for the older versions of REALJava, part III.

	REALJava 2.02 (HW)	REALJava 2.03 (HW)	REALJava 2.04 (HW)	REALJava 2.05 (HW)
byte array access	510007	562540	562540	562540
byte array copy	17476266	17476266	17476266	17476266
int array access	510007	557753	557753	557753
int array copy	10922666	10922666	10922666	10922666
byte add	4761904	4761904	4761904	4761904
byte sub	4347826	4347826	4347826	4347826
byte mul	3571428	3571428	3571428	3571428
byte div	1470588	1470588	1470588	1470588
int add	5263157	5263157	5263157	5263157
int sub	5263157	5263157	5263157	5263157
int mul	5263157	5263157	5263157	5263157
int div	1652892	1666666	1666666	1666666
float add	2941176	2941176	2941176	2941176
float sub	2941176	2941176	2941176	2941176
float mul	3703703	3703703	3703703	3703703
float div	2040816	2040816	2040816	2040816
double add	475059	481927	481927	511508
double sub	450450	454545	454545	480769
double mul	392927	396039	396825	417536
double div	154202	154918	146412	146520
string concat	3021	3759	3937	3960
string compare	250000	285714	285714	294117
method calls	1851851	1851851	1851851	1851851
object creations	357142	384615	465116	487804
T.L.E.				
Mandel	13354	13330	12635	12588
Fibonacci	1638	1633	970	964
Life	1962	1932	1186	1157
Text	1803	1781	1136	1093
Salesman	23927	21488	20777	18157
Sort	8170	7864	7167	6944
Neural Net	119717	100557	99970	86701
RayTrace	34546	18846	18189	17333
RayTrace2	14614	13021	12081	11549
Mandel ME				
Salesman ME				
Sort ME				
RayTrace ME				

Table B.8: Measurement data for the older versions of REALJava, part IV.

	REALJava 2.06 (HW)	REALJava 2.07 (HW)	REALJava 2.08 (HW)	REALJava 2.09 (HW)
byte array access	566185	585142	732245	791975
byte array copy	17476266	43690666	52428800	52428800
int array access	560136	579964	724154	789590
int array copy	10922666	14563555	14563555	14563555
byte add	4545454	5000000	8333333	10000000
byte sub	4545454	5000000	7692307	9090909
byte mul	4347826	4761904	7692307	9090909
byte div	1587301	1639344	1886792	1960784
int add	5882352	5882352	9090909	14285714
int sub	5882352	6250000	10000000	12500000
int mul	5882352	5882352	9090909	14285714
int div	1754385	1785714	1960784	2083333
float add	3030303	3125000	4000000	4545454
float sub	3030303	3225806	4166666	4545454
float mul	3846153	4000000	5555555	6666666
float div	2083333	2173913	2564102	2702702
double add	512820	527704	558659	578034
double sub	481927	497512	523560	540540
double mul	418410	428265	448430	460829
double div	147928	159489	162206	163800
string concat	4043	4511	13046	13726
string compare	322580	333333	344827	392156
method calls	1818181	2272727	2325581	2857142
object creations	487804	540540	555555	645161
T.L.E.		185537	282598	302941
Mandel	11682	8464	6225	5325
Fibonacci	949	457	438	433
Life	1155	636	587	571
Text	1072	575	540	518
Salesman	17312	15817	13829	12625
Sort	6652	5510	4974	4396
Neural Net	83470	78775	70443	64794
RayTrace	16838	14584	13187	12168
RayTrace2	10839	10001	9299	8349
Mandel ME			5474	4832
Salesman ME			13030	11597
Sort ME			45501	40342
RayTrace ME			8666	7891

Table B.9: Measurement data for the older versions of REALJava, part V.

	Nokia 6170	REALJava 2.09 (HW)
byte array access	962879	791975
byte array copy	22795130	52428800
int array access	916587	789590
int array copy	6096372	14563555
byte add	2724795	10000000
byte sub	2840909	9090909
byte mul	2695417	9090909
byte div	1694915	1960784
int add	3361344	14285714
int sub	3316749	12500000
int mul	3210272	14285714
int div	1917545	2083333
float add	1733102	4545454
float sub	1572327	4545454
float mul	1633986	6666666
float div	973709	2702702
double add	1207729	578034
double sub	1240694	540540
double mul	788643	460829
double div	310173	163800
string concat	15467	13726
string compare	512820	392156
method calls	1069518	2857142
object creations		645161
T.L.E.		302941
Mandel		5325
Fibonacci		433
Life		571
Text		518
Salesman		12625
Sort		4396
Neural Net		64794
RayTrace		12168
RayTrace2		8349
Mandel ME	14098	4832
Salesman ME	10546	11597
Sort ME	55777	40342
RayTrace ME	9036	7891

Table B.10: Measurement data for the Nokia 6170.

Appendix C

Version History

This Appendix shows the version history of the REALJava virtual machine. The history is recorded starting from the first version that was prototyped using the ML310 demonstration board. The older versions used only the XESS board, and during those early days the system was very slow and contained several bugs, both in the software partition and in the hardware as well. As a result, new versions were introduced daily, sometimes even several version in a single day. The major changes introduced by every new version are also listed in the Table C.1. Naturally all of the changes are not listed here, only those that required significant redesign of the co-processor and/or produced a measurable performance boosts.

The dates in the Table show when the version in question was prototyped using the larger boards. Most of the techniques were developed over time, with several *pre*prototype versions designed for the XESS board. The techniques were tested for both correctness and compatibility with each other. Some of the modifications needed some tweaking in order to facilitate some other changes. Thus the dates of the versions seem to be packed towards the end of the time period in question.

The Table C.2 shows the performance increasing techniques presented in the Chapter 5 with the versions in which they were applied. If a given technique was modified in more than one version, all are listed. The section describing a given technique is shown in parenthesis for reference.

Name	Version	Date	Comment
REALJava	0.01	19.02.07	The first tests run on Xilinx ML310 with PowerPC 405 @ 300 MHz and REALJava co-processor @ 100 MHz
REALJava	0.02	13.03.07	Added IDIV,IREM and single precision floats, streamlined internal memory management
REALJava	0.03	16.03.07	Optimized stack <=> memory
REALJava	0.04	05.04.07	More optimization on memory controller and fixed int to 0 comprisons
REALJava	0.05	25.04.07	Instruction prefetch and early stack idle detection
REALJava	0.06	03.05.07	Method changed testing enabled
REALJava	0.07	29.05.07	Partial stack visibility to alu
REALJava	0.08	08.06.07	Improved visibility
REALJava	1.00	10.09.07	New SW and method invoker
REALJava	1.01	03.10.07	FCMP{L|G} + improved I/O (CPU<=>JPU)
REALJava	2.00	04.01.08	Moved to ML410, Virtex4, othervice same as 1.01
REALJava	2.01	19.02.08	Push and pop registers
REALJava	2.02	19.02.08	Fast instruction data (8-bit)
REALJava	2.03	26.02.08	Push execute and improved method invocation on cache miss
REALJava	2.04	29.02.08	Constant cache
REALJava	2.05	04.03.08	Arraylenght cache
REALJava	2.06	12.03.08	Fast instruction data (all widths)
REALJava	2.07	19.03.08	Cleanups, on HW and SW
REALJava	2.08	25.03.08	Partial folding on arithmetic results
REALJava	2.09	08.04.08	Added pipeline stage for partial foldings

Table C.1: Version history of the REALJava virtual machine.

Technique	Introduced in
Direct connection between the stack and the ALU (5.2)	0.07, 0.08
Stack manipulations in the memory controller (5.3)	0.03, 0.04
Partial instruction folding (5.4)	2.08, 2.09
Method invoker module (5.5)	1.00
Using invoker with SW invocations (5.5.2)	2.03
Constant cache (5.6)	2.04
Arraylength cache (5.6)	2.05
Push and pop registers (5.8)	2.01

Table C.2: The performance increasing techniques and the corresponding versions. The sections describing the techniques are given in parenthesis.

Appendix D

Comparing Kaffe and REALJava on an x86 processor

This appendix provides estimation on the Kaffe virtual machine's performance in comparison with the software only version of the REALJava virtual machine. Since both systems can be run on the PowerPC based demonstration boards as well as on a standard laptop with an x86 architecture based processor, the results can be used to draw conclusions about the possible optimizations Kaffe might have. This possibility is considered because the software only version of the REALJava clearly outperformed Kaffe on the PowerPC architecture. The software only version of the REALJava is exactly the same as used for the PowerPC, except that it has been recompiled for the x86 architecture. The same is true for the Kaffe, the version is the same as used on the PowerPC, as are the configuration options. The results are shown in the Table D.1, and they show that the Kaffe is slower (based on the T.L.E. score) by a factor of 7.8. On the PowerPC the factor is 4.8, suggesting that Kaffe contains no optimizations for the x86 architecture. Rather it seems that the Kaffe performs relatively better on the PowerPC system. The discrepancy in the factors is most likely due to a combination of the amount of memory, the cache system, the processors architecture and other properties of the systems. Also the compilers can optimize code differently for different target architectures. The evaluation is not extended beyond the tests shown here, since the target domain of the REALJava virtual machine is in the embedded systems.

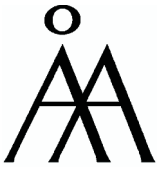
	REALJava 2.00 (SW)	Kaffe
byte array access	6241523	1110779
byte array copy	262144000	262144000
int array access	5957818	1159929
int array copy	131072000	131072000
byte add	25000000	3448275
byte sub	28571428	3448275
byte mul	28571428	3508771
byte div	25000000	3389830
int add	16666666	4347826
int sub	20000000	4444444
int mul	22222222	4444444
int div	22222222	4255319
float add	33333333	4255319
float sub	28571428	4255319
float mul	28571428	4166666
float div	22222222	4081632
double add	14285714	4255319
double sub	14285714	4081632
double mul	10000000	4081632
double div	10526315	4081632
string concat	136054	8305
string compare	2857142	425531
method calls	18181818	1626016
object creations	5000000	273972
T.L.E.	2643172	336855

Table D.1: Performance evaluation between the software only version of the REALJava virtual machine and the Kaffe virtual machine on an x86 based system.

Turku
Centre *for*
Computer
Science

Joukahaisenkatu 3-5 B, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics



Åbo Akademi University
- Department of Information Technologies

Turku School of Economics
- Institute of Information Systems Sciences

ISBN 978-952-12-2155-2
ISSN 1239-1883