HABILITATIONSSCHRIFT

Time-predictable Computer Architecture

Vorgelegt zur Erlangung der Lehrbefugnis
für das Fach "Technische Informatik"

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Dr.techn. Martin Schöberl

Straußengasse 2-10/2/55
1050 Wien

Wien, September 2009

Contents

1 Introduction

Standard computer architecture is driven by the following paradigm: *Make the common case fast and the uncommon case correct*. This design approach leads to architectures where the *average-case* execution time is optimized at the expense of the *worst-case* execution time (WCET). Modeling the dynamic features of current processors for WCET analysis often results in computationally infeasible problems. The bounds calculated by the analysis are thus overly conservative.

We need a sea-change and we shall take the constructive approach by designing computer architectures where predictable timing is a first order design factor. For real-time systems we propose to design architectures with a new paradigm: *Make the worst case fast and the whole system easy to analyze*. Despite the advantages of an analyzable processor, only a few research projects exist in the field of WCET optimized hardware.

1.1 Time-predictable Processor Architecture

Today's general-purpose processors are optimized for maximum throughput [11]. Real-time systems need a processor with both a reasonable and a known worst-case execution time (WCET). Classic enhancements in computer architectures are: pipelining, instruction and data caching, dynamic branch prediction, out-of-order execution, speculative execution, and fine-grained chip multithreading. These features are increasingly harder to model for the low-level WCET analysis. Execution history is the key to performance enhancements, but also the main issue for WCET analysis. Thus we need techniques to manage the execution history.

It has to be noted that a processor designed for low WCET will never be as fast in the average case as a processor optimized for the average case. Those are two different design optimizations. Furthermore, WCET analysis can only provide WCET bounds that are higher than the real WCET. The difference between the actual WCET and the bound is caused by the pessimism of the analysis resulting from two factors: (a) certain information, e.g., infeasible execution paths, not being known statically and (b) the simplifications to make the analysis computationally practical. Here at this place we would like to present our definition of a time-predictable processor:

> Under the assumption that only feasible execution paths are analyzed, a time-predictable processor's WCET bound is equal or almost equal to the real WCET.

Pipelines shall be simple, with minimum dependencies between instructions. It is agreed that caches are mandatory to bridge the gap between processor speed and memory access time. Caches in general, and particularly data caches, are usually hard to analyze statically. Therefore, we are introducing caches that are organized to speed-up execution time and provide tight WCET bounds. We propose three different caches: (1) an instruction cache for full methods, (2) a stack cache and, (3) a small, fully associative buffer for heap access [30]. Furthermore, the integration of a program- or compiler-managed scratchpad memory can help to tighten bounds for hard to analyze memory access patterns [39].

Out-of-order execution and speculation result in processor models that are too complex for WCET analysis. We argue that the transistors are better used on chip-multiprocessors (CMP) with simple in-order pipelines. Real-time systems are naturally multithreaded and thus map well to the explicit parallelism of chip multiprocessors. We propose a multiprocessor model with one processor per thread [32]. Thread switching and schedulability analysis for each individual core disappears, but the access to the shared resource main memory still needs to be scheduled.

The following list points out the key arguments for a time-predictable computer architecture:

- There is a mismatch between performance oriented computer architectures and worst-case analyzability.
- Complex features result in increasingly complex models.
- Caches, a very important feature for high performance, need new organization.
- Thread level parallelism is natural in embedded systems. Exploration of this parallelism with simple chip-multiprocessors is a valuable option.
- One thread per processor obviates the classic schedulability analysis and introduces scheduling of memory access.

Catching up with WCET analysis of features that enhance the average case performance is not an option for future real-time systems. We have to take the constructive approach and design computer architectures with predictable timing.

We have implemented most of the proposed concepts for evaluation in the Java processor JOP, as presented in Chapter 2. JOP [29] is intended for real-time and safety critical applications written in a modern object oriented language. It has to be noted that all concepts can also be applied to a standard RISC processor. In Chapter 5 the WCET analysis issues of modern processor architectures are evaluated and architectural alternatives are proposed.

1.2 Real-Time Java

Java has its roots in the embedded domain. In the early '90s, Java, which was originally known as Oak [35], was created as a programming tool for a consumer device that we would today call a PDA. Over the years, Java technology has become a programming tool for desktop applications, web servers and server applications. Today Java again is used in embedded devices such as mobile phones.

Java is a strongly typed, object oriented language, with safe references, and array bounds checking. Java shares those features with Ada, the main language for safety-critical real-time systems. In contrast to Ada, Java has a large user and open-source code base. In [38] it is argued that Java will become the better technology for real-time systems. Furthermore, threads and synchronization, common idioms in real-time programming, are part of the language.

The object references replace error-prone C/C++ style pointers. Type checking is enforced by the compiler and performed at runtime. Those features greatly help to avoid program errors. Therefore Java is an attractive choice for safety-critical and real-time systems [12]. The Java specification request 302 [15], where the author is a member of the Expert Group, defines a standard for Java in safety-critical systems.

An early document published by the NIST [22] defines the requirements for real-time Java. Based on those requirements the Real-Time Specification for Java (RTSJ) [5] started as first Java specification request (JSR). In the expert group of the RTSJ garbage collection was considered as the main issue of Java in real-time systems. Therefore the RTSJ defines, besides other idioms, new memory areas (scoped memory) and a NoHeapRealtimeThread that can interrupt the garbage collector. However, real-time garbage collection is an active research area (e.g., [1]). In [28] it is shown that a correctly scheduled garbage collector can be used even in hard real-time systems. Hardware support for non-blocking, real-time garbage collection is presented in Chapter 3.

Java bytecode generation has to follow stringent rules [20] in order to pass the class file verification of the JVM. Those restrictions lead to an *analysis friendly* code, e.g. the stack size is known at each instruction. The control flow instructions are well defined. Branches are relative and the destination is within the same method. In Java class files there is more information available than in compiled C/C++ executables. All links are symbolic and it is possible to reconstruct the class hierarchy from the class files. Therefore, a WCET analysis tool can statically determine all possible targets for a virtual method invocation.

The known execution time of bytecodes on JOP (see Chapter 2) has enabled the creation of several WCET analysis tools, which target JOP: We built the first WCET analyzer for JOP in 2006 [31]. Trevor Harmon, from the University of California, Irvine, targets JOP with his WCET analysis tool developed during his PhD thesis [10]. Bogholm et al. developed an integrated WCET and scheduling analysis tool based on model checking [4]. Bendikt Huber has redesigned the WCET analysis tool for JOP during his Master's thesis [13].

Many embedded applications require very small platforms, therefore it is interesting to remove as much as possible of an underlying operating system, where a major part of code is dedicated to handling devices. As certification of safety-critical systems is very expensive, the usual approach is to minimize the code base and supporting tools. Using two languages (e.g., C for programming device handling and Java for the application) increases the complexity of certification. A Java only system reduces the complexity and therefore the certification effort. Even in less critical systems the same issues will show up as decreased productivity and dependability of the software. Thus it makes sense to investigate a general solution that interfaces Java to the hardware platform. This hardware abstraction layer (HAL) in Java is presented in Chapter 4.

1.3 Related Work

In this section we give a high-level overview of research groups working on time-predictable computer architecture and real-time Java. Detailed references can be found in the related work sections of the following chapters.

Although not too many research groups explicitly work on time-predictable computer architecture, the topic is getting some momentum in the last years: Thiele and Wilhelm argue that a new research discipline is needed for time-predictable embedded systems to "match implementation concepts with techniques to improve analyzability" [36]; Berg et al. identify design principles for a time-predictable processor [3]; Edwards and Lee argue: "It is time for a new era of processors whose temporal behavior is as easily controlled as their logical function" [9].

1.3.1 Real-Time Systems Group, University of York

Bate et al. [2] discuss the usage of modern processors in safety critical applications. They compare commercial off-the-shelf (COTS) processors with a customized processor developed specifically for the safety critical domain. While COTS processors benefit from a large user base and the resulting maturity of the design process, customized processors provide following advantages: (a) design in conjunction with the safety argument; (b) design for good worst-case performance; (c) using only features that can be easily analyzed, and (d) the processor can be treated as a *white box* during verification and testing. This argument is in line with our proposal of time-predictable computer architecture.

Jack Whitham argues that the execution time of a basic block has to be independent of the execution history [40]. To reduce the WCET, Whitham proposes to implement the time critical functions in microcode on a reconfigurable function unit (RFU). The interesting approach in the MCGREP design is that the RFUs implement the same architecture and microcode as the main CPU. Therefore, mapping a sequence of RISC instructions to microcode for one or several RFUs is straightforward. With several RFUs, it is possible to explicitly extract instruction level parallelism (ILP) from the original RISC code in a similar way to VLIW architectures.

Whitham and Audsley extend the MCGREP architecture with a trace scratchpad [41]. The trace scratchpad caches microcode and is placed after the decode stage. The differences from a cache are that the execution from the trace scratchpad has to be explicitly started and the scratchpad has to be loaded under program control. Further work on a memory management unit for data scratchpad memory is presented in [42].

Andy Wellings is active in proposing Java and the RTSJ for future real-time systems [38, 37]. The seminal book on real-time programming [6] by Burns and Wellings features Java as a programming language.

1.3.2 PRET Group, University of California, Berkeley

Edward Lee argues that microprocessors for real-time systems need not only be time-predictable, but shall also allow repeatable execution timing [18]. Edwards and Lee present the concept of a precision timed (PRET) machine [9]. A first simulation of their PRET architecture is presented in [19]. PRET implements the SPARC V8 instruction set architecture (ISA) in a six-stage pipeline and performs chip level multithreading for six threads to eliminate data forwarding and branch prediction. The SPARC ISA is extended with a *deadline* instruction [14] that stalls the current thread until the deadline is reached. This instruction is used to perform time based, instead of lock based, synchronization for access to shared data. Scratchpad memories are used instead of instruction and data caches. The shared main memory is accessed via a TDMA scheme, called memory wheel, similar to the TDMA based arbiter used in the JOP CMP system [24]. A hierarchical memory architecture that uses pipelined access to different memory banks hides memory latencies [8].

1.3.3 Institute of Computer Engineering, Vienna University of Technology

Puschner and Burns argue for a single-path programming style [27] that results in a constant execution time. A set of code transformations [26] eliminates all input dependent control flow decisions. In that case, the WCET can easily be measured. A processor, called SPEAR [7], was especially designed to evaluate the single-path programming paradigm. A single predicate bit can be set with a compare instruction whereby several instructions (e.g., move, arithmetic operations) can be predicated. In [17] it is shown that processors with direct-mapped instruction caches, programmed in

single-path style, are time-predictable even with time-triggered task preemption. The combination of single-path programming and chip-multiprocessing is proposed to reconcile performance and predictability [33]. Furthermore, current research investigates possibilities to use super-scalar processors for real-time systems [16].

1.3.4 Compiler Design Lab, Saarland University

The compiler design lab at Saarland University is well know for research on timing analysis for real-time systems. Recent work also focuses on computer architecture to simplify WCET analysis [43]. Thiele and Wilhelm argue that a new research discipline is needed for time-predictable embedded systems [36]. Berg et al. identify the following design principles for a time-predictable processor: "... recoverability from information loss in the analysis, minimal variation of the instruction timing, non-interference between processor components, deterministic processor behavior, and comprehensive documentation" [3]. The authors propose a processor architecture that meets these design principles. The processor is a classic five-stage RISC pipeline with minimal changes in the instruction set

1.3.5 EC Funded Projects

The EC project Predator[1] is a three year research project funded by the European Commission. Predator aims to improve development methods and tools for safety-critical systems. Furthermore, architectural concepts to support WCET analysis will be developed. We agree on the following statement from the Predator web site:

> Embedded system design needs to go through a paradigm shift towards a reconciliation of both design goals, predictability and efficiency, taking into account the multi-objective nature of the underlying problem.

The research direction on cache aware, WCET optimized compilation [21] is most closely related to our research work.

The EC project Merasa[2] develops multi-core architectures for hard real-time systems that shall be time-predictable. The processor design will be co-developed with the WCET analysis to achieve the project goal. As an example, a multi-core memory arbiter is adapted to provide a mode, where the worst-case memory latency is enforced [23]. Only in this mode measurement-based WCET analysis can be used.

Real-time Java on chip-multiprocessors (CMP) is the topic of the EC funded project Jeopard [34]. Within Jeopard, the partners work on all layers involved in a CMP system, from the hardware architecture, via a CMP real-time OS, a CMP real-time JVM, Java APIs for CMP systems up to application examples and analysis tools. The CMP version of JOP [25] is used as the platform for the hardware architecture research.

1.4 Selected Papers

This thesis contains 4 papers published or accepted for publication in scientific Journals. The first paper introduces the time-predictable Java processor JOP, which has been used for research on several aspects of real-time Java systems (e.g., [25, 34]). Future real-time Java systems might use

[1] `http://www.predator-project.eu/`
[2] `http://www.merasa.org`

garbage collection (GC) to simplify memory management. If GC is used in real-time systems, the GC algorithm has to be analyzable [28] and all GC operations need to be non-blocking. The second paper proposes hardware and software solutions for a non-blocking garbage collector for mixed real-time systems. Another source of unpredictability is the operating system in a complex system. For safety-critical real-time systems the whole software stack, including the operating system, needs to be certified. In the third paper we propose a hardware abstraction layer in Java to avoid an underlying operating system and simplify the safety argument. The last paper analyzes the issues of current processor architectures with respect to WCET analysis. Then solutions for a time-predictable computer architecture are propose. This paper represents a generalization of the development within JOP to RISC processors and chip-multiprocessor systems.

A Java Processor Architecture for Embedded Real-Time Systems

Martin Schoeberl
Journal of Systems Architecture, Volume 54, Issue 1-2, pages 265–286, 2008, Elsevier

Architectural advancements in modern processor designs increase average performance with features such as pipelines, caches, branch prediction, and out-of-order execution. However, these features complicate worst-case execution time analysis and lead to very conservative estimates. JOP (Java Optimized Processor) tackles this problem from the architectural perspective – by introducing a processor architecture in which simpler and more accurate WCET analysis is more important than average case performance.

This paper presents a time-predictable processor for for real-time Java. JOP is the implementation of the Java virtual machine in hardware. JOP is intended for applications in embedded real-time systems and the primary implementation technology is in a field programmable gate array. It is shown that the architecture is WCET analysis friendly, as the execution time of bytecodes can be predicted cycle accurately. This paper demonstrates that a hardware implementation of the Java virtual machine results in a small design for resource-constrained devices.

Non-blocking Real-Time Garbage Collection

Martin Schoeberl and Wolfgang Puffitsch
Trans. on Embedded Computing Sys., 26 pages, accepted 2009, ACM

Garbage collection is an essential part of the Java runtime system. It enables automatic, dynamic memory management, which frees the programmer from complex and error prone explicit memory management. Garbage collection is now even considered for (soft) real-time systems. A real-time garbage collector has to fulfill two basic properties: ensure that programs with bounded allocation rates do not run out of memory [28] and provide short blocking times. Even for incremental garbage collectors, two major sources of blocking exist, namely root scanning and heap compaction. In this paper, we propose solutions to both issues.

Thread stacks are local to a thread, and root scanning therefore only needs to be atomic with respect to the thread whose stack is scanned. This fact can be utilized by either blocking only the thread whose stack is scanned, or by delegating the responsibility for root scanning to the application threads. The latter solution eliminates blocking due to root scanning completely.

During heap compaction, objects are copied. Copying is usually performed atomically to avoid interference with application threads. Copying of large objects introduces long blocking times that

are unacceptable for real-time systems. In this paper an interruptible copy unit is presented that implements non-blocking object copy. The unit can be interrupted after a single word move.

We evaluate a real-time garbage collector that uses the proposed techniques on a Java processor. With this garbage collector, it is possible to run high priority hard real-time tasks at 10 kHz parallel to the garbage collection task on a 100 MHz system.

A Hardware Abstraction Layer in Java

Martin Schoeberl, Stephan Korsholm, Tomas Kalibera, and Anders P. Ravn
Trans. on Embedded Computing Sys., 42 pages, accepted 2009, ACM

Embedded systems use specialized hardware devices to interact with their environment, and since they have to be dependable, it is attractive to use a modern, type-safe programming language like Java to develop programs for them. Standard Java, as a platform independent language, delegates access to devices, direct memory access, and interrupt handling to some underlying operating system or kernel, but in the embedded systems domain resources are scarce and a Java virtual machine (JVM) without an underlying middleware is an attractive architecture.

Furthermore, Java is considered as the future language for safety critical systems [12], which need to be certified. However, the additional layers of a real-time operating system and the JVM increase the certification burden. In this paper, concepts for a hardware abstraction layer in Java towards an embedded real-time Java architecture without an OS layer are described.

The contribution of this paper is a proposal for Java packages with hardware objects and interrupt handlers that interface to such a JVM. We provide implementations of the proposal directly in hardware, as extensions of standard interpreters, and finally with an operating system middleware. The latter solution is mainly seen as a migration path allowing Java programs to coexist with legacy system components. An important aspect of the proposal is that it is compatible with the Real-Time Specification for Java (RTSJ).

Time-predictable Computer Architecture

Martin Schoeberl
EURASIP Journal on Embedded Systems, Volume 2009, Article ID 758480, 17 pages, 2009, Hindawi

Today's general-purpose processors are optimized for maximum throughput. Real-time systems need a processor with both a reasonable and a known worst-case execution time (WCET). Features such as pipelines with instruction dependencies, caches, branch prediction, and out-of-order execution complicate WCET analysis and lead to very conservative estimates. In this paper, we evaluate the issues of current architectures with respect to WCET analysis. Then we propose solutions for a time-predictable computer architecture. The proposed architecture is evaluated with implementation of some features in a Java processor. The resulting processor is a good target for WCET analysis and still performs well in the average case.

Bibliography

[1] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.

[2] Iain Bate, Philippa Conmy, Tim Kelly, and John A. McDermid. Use of modern processors in safety-critical applications. *The Computer Journal*, 44(6):531–543, 2001.

[3] Christoph Berg, Jakob Engblom, and Reinhard Wilhelm. Requirements for and design of a processor with predictable timing. In Lothar Thiele and Reinhard Wilhelm, editors, *Perspectives Workshop: Design of Systems with Predictable Behaviour*, number 03471 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2004. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[4] Thomas Bogholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-based schedulability analysis of safety critical hard real-time Java programs. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, pages 106–114, New York, NY, USA, 2008. ACM.

[5] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[6] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2001.

[7] Martin Delvai, Wolfgang Huber, Peter Puschner, and Andreas Steininger. Processor support for temporal predictability – the SPEAR design example. In *Proceedings of the 15th Euromicro International Conference on Real-Time Systems*, Jul. 2003.

[8] Stephen A. Edwards, Sungjun Kim, Edward A. Lee, Isaac Liu, Hiren D. Patel, and Martin Schoeberl. A disruptive computer design idea: Architectures with repeatable timing. In *Proceedings of IEEE International Conference on Computer Design (ICCD 2009)*, Lake Tahoe, CA, October 2009. IEEE.

[9] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 264–265, New York, NY, USA, 2007. ACM.

[10] Trevor Harmon. *Interactive Worst-case Execution Time Analysis of Hard Real-time Systems*. PhD thesis, University of California, Irvine, 2009.

[11] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[12] Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, Mar. 2009.

[13] Benedikt Huber. Worst-case execution time analysis for real-time Java. Master's thesis, Vienna University of Technology, Austria, 2009.

[14] Nicholas Jun Hao Ip and Stephen A. Edwards. A processor extension for cycle-accurate real-time software. In *IFIP International Conference on Embedded and Ubiquitous Computing (EUC)*, volume LNCS 4096, pages 449–458, Seoul, Korea, 2006. Springer.

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

[16] Albrecht Kadlec, Raimund Kirner, and Peter Puschner. Counter measures for timing anomalies using compile-time instruction scheduling. Technical report, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2009.

[17] Raimund Kirner and Peter Puschner. Time-predictable task preemption for real-time systems with direct-mapped instruction cache. In *Proc. 10th IEEE International Symposium on Object-oriented Real-time distributed Computing*, Santorini Island, Greece, May 2007.

[18] Edward A. Lee. Computing needs time. *Commun. ACM*, 52(5):70–79, 2009.

[19] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In Erik R. Altman, editor, *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008)*, pages 137–146, Atlanta, GA, USA, October 2008. ACM.

[20] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[21] Paul Lokuciejewski, Heiko Falk, and Peter Marwedel. WCET-driven cache-based procedure positioning optimizations. In *The 20th Euromicro Conference on Real-Time Systems (ECRTS 2008)*, pages 321–330. IEEE Computer Society, 2008.

[22] K. Nilsen, L. Carnahan, and M. Ruark. Requirements for real-time extensions for the Java platform. Available at http://www.nist.gov/rt-java/, September 1999.

[23] Marco Paolieri, Eduardo Qui nones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *The 36th International Symposium on Computer Architecture (ISCA 2009)*, pages 57–68, Austin, Texas, USA, 20-24, June 2009. ACM.

[24] Christof Pitter. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, 2008.

[25] Christof Pitter and Martin Schoeberl. A real-time Java chip-multiprocessor. *Trans. on Embedded Computing Sys. accepted for publication.*, 2009.

[26] Peter Puschner. Transforming execution-time boundable code into temporally predictable code. In Bernd Kleinjohann, K.H. (Kane) Kim, Lisa Kleinjohann, and Achim Rettberg, editors, *Design and Analysis of Distributed Embedded Systems*, pages 163–172. Kluwer Academic Publishers, 2002. IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES 2002).

[27] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 85–94, Washington, DC, USA, 2002. IEEE Computer Society.

[28] Martin Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006. IEEE.

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

[30] Martin Schoeberl. Time-predictable cache organization. In *Proceedings of the First International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.

[31] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[32] Martin Schoeberl and Peter Puschner. Is chip-multiprocessing the end of real-time scheduling? In *Proceedings of the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Dublin, Ireland, July 2009. OCG.

[33] Martin Schoeberl, Peter Puschner, and Raimund Kirner. A single-path chip-multiprocessor system. In *Proceedings of the Seventh IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS 2009)*. Springer, November 2009.

[34] Fridtjof Siebert. JEOPARD: Java environment for parallel real-time development. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, pages 87–93, New York, NY, USA, 2008. ACM.

[35] Sun. A brief history of the green project. Available at: http://today.java.net/jag/old/green/.

[36] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

[37] Andrew J. Wellings. *Concurrent and real-time programming in Java*. John Wiley and Sons, 2004.

[38] Andy Wellings. Is Java augmented with the RTSJ a better real-time systems implementation technology than Ada 95? *Ada Lett.*, XXIII(4):16–21, 2003.

[39] Andy Wellings and Martin Schoeberl. Thread-local scope caching for real-time Java. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.

[40] Jack Whitham. *Real-time Processor Architectures for Worst Case Execution Time Reduction*. PhD thesis, University of York, 2008.

[41] Jack Whitham and Neil Audsley. Using trace scratchpads to reduce execution times in predictable real-time architectures. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, pages 305–316, April 2008.

[42] Jack Whitham and Neil Audsley. Implementing time-predictable load and store operations. In *Proceedings of the International Conference on Embedded Software (EMSOFT 2009)*, 2009.

[43] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 28(7):966–978, 2009.

2 A Java Processor Architecture for Embedded Real-Time Systems

Journal of Systems Architecture, Volume 54, Issue 1-2, pages 265–286, 2008, Elsevier

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

Abstract

Architectural advancements in modern processor designs increase average performance with features such as pipelines, caches, branch prediction, and out-of-order execution. However, these features complicate worst-case execution time analysis and lead to very conservative estimates. JOP (Java Optimized Processor) tackles this problem from the architectural perspective – by introducing a processor architecture in which simpler and more accurate WCET analysis is more important than average case performance.

This paper presents a Java processor designed for time-predictable execution of real-time tasks. JOP is the implementation of the Java virtual machine in hardware. JOP is intended for applications in embedded real-time systems and the primary implementation technology is in a field programmable gate array. This paper demonstrates that a hardware implementation of the Java virtual machine results in a small design for resource-constrained devices.

2.1 Introduction

Compared to software development for desktop systems, current software design practice for embedded real-time systems is still archaic. C/C++ and even assembly language are used on top of a small real-time operating system. Many of the benefits of Java, such as safe object references, the notion of concurrency as a first-class language construct, and its portability, have the potential to make embedded systems much safer and simpler to program. However, Java technology is seldom used in embedded real-time systems, due to the lack of acceptable real-time performance.

Traditional implementations of the Java virtual machine (JVM) as interpreter or just-in-time compiler are not practical. An interpreting virtual machine is too slow and therefore waste of processor resources. Just-in-time compilation has several disadvantages for embedded systems, notably that a compiler (with the intrinsic memory overhead) is necessary on the target system. Due to compilation during runtime, execution times are practically not predictable[1].

[1] One could add the compilation time of a method to the WCET of that method. However, in that case we need a WCET analyzable compiler and the WCET gets impractical high.

This paper introduces the concept of a Java processor [51] for embedded real-time systems, in particular the design of a small processor for resource-constrained devices with time-predictable execution of Java programs. This Java processor is called JOP – which stands for Java Optimized Processor –, based on the assumption that a full native implementation of all Java bytecode instructions [30] is not a useful approach.

Worst-case execution time (WCET) estimates of tasks are essential for designing and verifying real-time systems. Static WCET analysis is necessary for hard real-time systems. In order to obtain a low WCET value, a good processor model is necessary. Traditionally, only simple processors can be analyzed using practical WCET boundaries. Architectural advancements in modern processor designs tend to abide by the rule: '*Make the average case as fast as possible*'. This is orthogonal to '*Minimize the worst case*' and has the effect of complicating WCET analysis. This paper tackles this problem from the architectural perspective – by introducing a processor architecture in which simpler and more accurate WCET analysis is more important than average case performance.

JOP is designed from ground up with time predictable execution of Java bytecode as major design goal. All function units, and especially the interaction between them, are carefully designed to avoid any time dependency between bytecodes. The architectural highlights are:

1. Dynamic translation of the CISC Java bytecodes to a RISC, stack based instruction set (the microcode) that can be executed in a 3 stage pipeline.

2. The translation takes exactly one cycle per bytecode and is therefore pipelined. Compared to other forms of dynamic code translation the proposed translation does not add any variable latency to the execution time and is therefore time predictable.

3. Interrupts are inserted in the translation stage as special bytecodes and are transparent to the microcode pipeline.

4. The short pipeline (4 stages) results in short conditional branch delays and a hard to analyze branch prediction logic or branch target buffer can be avoided.

5. Simple execution stage with the two topmost stack elements as discrete registers. No write back stage or forwarding logic is needed.

6. Constant execution time (one cycle) for all microcode instructions. No stalls in the microcode pipeline. Loads and stores of object fields are handled explicitly.

7. No time dependencies between bytecodes result in a simple processor model for the low-level WCET analysis.

8. Time predictable instruction cache that caches whole methods. Only invoke and return instruction can result in a cache miss. All other instructions are guaranteed cache hits.

9. Time predictable data cache for local variables and the operand stack. Access to local variables is a guaranteed hit and no pipeline stall can happen. Stack cache fill and spill is under microcode control and analyzable.

10. No prefetch buffers or store buffers that can introduce unbound time dependencies of instructions. Even simple processors can contain an instruction prefetch buffer that prohibits exact WCET values. The design of the method cache and the translation unit avoids the variable latency of a prefetch buffer.

11. Good average case performance compared to other non real-time Java processors.

12. Avoidance of hard to analyze architectural features results in a very small design. Therefore an available real estate can be used for a chip multi-processor solution.

In this paper, we will present the architecture of the real-time Java processor and the evaluation results for JOP, with respect to WCET, size and performance. We will show that the execution time of Java bytecodes can be exactly predicted in terms of the number of clock cycles. We will also evaluate the general performance of JOP in relation to other embedded Java systems. Although JOP is intended as a processor with a low WCET for all operations, its general performance is still important. We will see that a real-time processor architecture does not need to be slow.

In the following section, related work on real-time Java, Java processors, and issues with the low-level WCET analysis for standard processors is presented. In Section 2.3 a brief overview of the architecture of JOP is given, followed by a more detailed description of the microcode. In Section 2.4 it is shown that our objective of providing an easy target for WCET analysis has been achieved. Section 2.5 compares JOP's resource usage with other soft-core processors. In the Section 2.6, a number of different solutions for embedded Java are compared at the bytecode level and at the application level.

2.2 Related Work

In this section we present arguments for Java in real-time systems, various Java processors from industry and academia, and an overview of issues in the low-level WCET analysis that can be avoided by the proposed processor design.

2.2.1 Real-Time Java

Java is a strongly typed, object oriented language, with safe references, and array bounds checking. Java shares those features with Ada, the main language for safety critical real-time systems. It is even possible, and has been done [60, 8], to compile Ada 95 for the JVM. In contrast to Ada, Java has a large user and open-source code base. In [64] it is argued that Java will become the better technology for real-time systems.

The object references replace error-prone C/C++ style pointers. Type checking is enforced by the compiler and performed at runtime. Those features greatly help to avoid program errors. Therefore Java is an attractive choice for safety critical and real-time systems [56, 26]. Furthermore, threads and synchronization, common idioms in real-time programming, are part of the language.

An early document published by the NIST [33] defines the requirements for real-time Java. Based on those requirements the Real-Time Specification for Java (RTSJ) [7] started as first Java Specification Request (JSR). In the expert group of the RTSJ garbage collection was considered as the main issue of Java in real-time systems. Therefore the RTSJ defines, besides other idioms, new memory areas (scoped memory) and a NoHeapRealtimeThread that can interrupt the garbage collector. However, real-time garbage collection is an active research area (e.g., [5]). In [44] and [52] it is shown that a correctly scheduled garbage collector can be used even in hard real-time systems.

Discussion of the RTSJ, platforms for embedded Java and the definition and implementation of a real-time profile for embedded Java on JOP can be found in [48].

Java bytecode generation has to follow stringent rules [30] in order to pass the class file verification of the JVM. Those restrictions lead to an *analysis friendly* code, e.g. the stack size is known at each

	Target technology	Size Logic	Memory	Speed (MHz)
JOP	Altera, Xilinx FPGA	2050 LCs	3 KB	100
picoJava [58, 59]	No realization	128 Kgates	38 KB	
aJile [1, 19]	ASIC 0.25μ	25 Kgates	48 KB	100
Cjip [18, 25]	ASIC 0.35μ	70 Kgates	55 KB	80
Moon [62, 63]	Altera FPGA	3660 LCs	4 KB	
Lightfoot [9]	Xilinx FPGA	3400 LCs	4 KB	40
Komodo [27]	Xilinx FPGA	2600 LCs		33
FemtoJava [6]	Xilinx FPGA	2710 LCs	0.5 KB	56

Table 2.1: JOP and various Java processors

instruction. The control flow instructions are well defined. Branches are relative and the destination is within the same method. In Java class files there is more information available than in compiled C/C++ executables. All links are symbolic and it is possible to reconstruct the class hierarchy from the class files. Therefore, a WCET analysis tool can statically determine all possible targets for a virtual method invocation.

2.2.2 Java Processors

Table 2.1 lists the relevant Java processors available to date. Sun introduced the first version of picoJava [36] in 1997. Sun's picoJava is the Java processor most often cited in research papers. It is used as a reference for new Java processors and as the basis for research into improving various aspects of a Java processor. Ironically, this processor was never released as a product by Sun. A redesign followed in 1999, known as picoJava-II that is now freely available with a rich set of documentation [58, 59]. The architecture of picoJava is a stack-based CISC processor implementing 341 different instructions [36] and is the most complex Java processor available. The processor can be implemented in about 440K gates [11]. Simple Java bytecodes are directly implemented in hardware, most of them execute in one to three cycles. Other performance critical instructions, for instance invoking a method, are implemented in microcode. picoJava traps on the remaining complex instructions, such as creation of an object, and emulates this instruction. A trap is rather expensive and has a minimum overhead of 16 clock cycles. This minimum value can only be achieved if the trap table entry is in the data cache and the first instruction of the trap routine is in the instruction cache. The worst-case trap latency is 926 clock cycles [59]. This great variation in execution times for a trap hampers tight WCET estimates.

aJile's JEMCore is a direct-execution Java processor that is available as both an IP core and a stand alone processor [1, 19]. It is based on the 32-bit JEM2 Java chip developed by Rockwell-Collins. Two silicon versions of JEM exist today: the aJ-80 and the aJ-100. Both versions comprise a JEM2 core, 48 KB zero wait state RAM and peripheral components. 16 KB of the RAM is used for the writable control store. The remaining 32 KB is used for storage of the processor stack. The aJile processor is intended for real-time systems with an on-chip real-time thread manager. aJile Systems was part of the expert group for the RTSJ [7]. However, no information is available about bytecode execution times.

The Cjip processor [18, 25] supports multiple instruction sets, allowing Java, C, C++ and assembler to coexist. Internally, the Cjip uses 72 bit wide microcode instructions, to support the different

instruction sets. At its core, Cjip is a 16-bit CISC architecture with on-chip 36 KB ROM and 18 KB RAM for fixed and loadable microcode. Another 1 KB RAM is used for eight independent register banks, string buffer and two stack caches. Cjip is implemented in 0.35-micron technology and can be clocked up to 80 MHz. The JVM is implemented largely in microcode (about 88% of the Java bytecodes). Java thread scheduling and garbage collection are implemented as processes in microcode. Microcode instructions execute in two or three cycles. A JVM bytecode requires several microcode instructions. The Cjip Java instruction set and the extensions are described in detail in [24]. For example: a bytecode nop executes in 6 cycles while an iadd takes 12 cycles. Conditional bytecode branches are executed in 33 to 36 cycles. Object oriented instructions such getfield, putfield or invokevirtual are not part of the instruction set.

Vulcan ASIC's Moon processor is an implementation of the JVM to run in an FPGA. The execution model is the often-used mix of direct, microcode and trapped execution. As described in [62], a simple stack folding is implemented in order to reduce five memory cycles to three for instruction sequences like *push-push-add*. The Moon2 processor [63] is available as an encrypted HDL source for Altera FPGAs or as VHDL or Verilog source code.

The Lightfoot 32-bit core [9] is a hybrid 8/32-bit processor based on the Harvard architecture. Program memory is 8 bits wide and data memory is 32 bits wide. The core contains a 3-stage pipeline with an integer ALU, a barrel shifter and a 2-bit multiply step unit. According to DCT, the performance is typically 8 times better than RISC interpreters running at the same clock speed. The core is provided as an EDIF netlist for dedicated Xilinx devices.

Komodo [27] is a multithreaded Java processor with a four-stage pipeline. It is intended as a basis for research on real-time scheduling on a multithreaded microcontroller. The unique feature of Komodo is the instruction fetch unit with four independent program counters and status flags for four threads. A priority manager is responsible for hardware real-time scheduling and can select a new thread after each bytecode instruction. Komodos multi-threading is similar to hyper-threading in modern processors that are trying to hide latencies in instruction fetching. However, this feature leads to very pessimistic WCET values (in effect rendering this performance gain useless in hard real-time systems). The fact that the pipeline clock is only a quarter of the system clock also wastes a considerable amount of potential performance.

FemtoJava [6] is a research project to build an application specific Java processor. The bytecode usage of the embedded application is analyzed and a customized version of FemtoJava is generated in order to minimize the resource usage. The resource usage is very high, compared to the minimal Java subset implemented and the low performance of the processor.

Besides the *real* Java processors a FORTH chip (PSC1000 [38]) is marketed as Java processors. Java coprocessors (e.g. JSTAR [32]) provide Java execution speedup for general-purpose processors. Jazelle [4] is an extension of the ARM 32-bit RISC processor. It introduces a third instruction set (bytecode), besides the Thumb instruction set (a 16-bit mode for reduced memory consumption), to the processor. The Jazelle coprocessor is integrated into the same chip as the ARM processor.

So far, all processors described (except Cjip) perform weakly in the area of time-predictable execution of Java bytecodes. However, a low-level analysis of execution times is of primary importance for WCET analysis. Therefore, the main objective is to define and implement a processor architecture that is as predictable as possible. However, it is equally important that this does not result in a low performance solution. Performance shall not suffer as a result of the time-predictable architecture. In Section 2.6, the overall performance of various Java systems, including the aJile processor, Komodo, and Cjip, is compared with JOP.

2.2.3 WCET Analysis

WCET Analysis can be divided in high-level and low-level analysis (see also Section 2.4). The high-level analysis is a mature research topic [29, 43, 40]. The main issues to be solved are in the low-level analysis. The processors that can be analyzed are usually several generations behind actual architectures [14, 34, 20]. An example: Thesing models in his 2004 PhD thesis [61] the PowerPC 750 (the MPC755 variant). The PowerPC 750 was introduced 1997 and the MPC755 is now (2006) *not recommended for new designs*.

The main issues in low-level analysis are many features of modern processors that increase average performance. All those features, such as multi-level caches, branch target buffer, out-of-order (OOO) execution, and speculation, include a lot of state that depends on a large execution history. Modeling this history for the WCET analysis leads to a state explosion for the final WCET calculation. Therefore low-level WCET analysis usually performs simplifications and uses conservative estimates. One example of this conservative estimate is to classify a cache access, if the outcome of the cache access is unknown, as a miss to be on the safe side. In [31] it is shown that this intuitive assumption can be wrong on dynamically scheduled microprocessors. An example is provided where a cache hit can cause a longer execution than a cache miss. In [28] a hypothetical OOO microprocessor is modeled for the analysis. However, verification of the proposed approach on a real processor is missing. Another issue is the missing or sometimes wrong documentation of the processor internals [13]. From a survey of the literature we found that modeling a new version of a microprocessor and finding all undocumented details is usually worth a full PhD thesis.

We argue that trying to catch up on the analysis side with the growing complexity of modern computer architectures is not feasible. A paradigm shift is necessary, either on the hardware level or on the application level. Puschner argues for a single-path programming style [41] that results in a constant execution time. In that case execution time can be simply measured. However, this programming paradigm is quite unusual and restrictive. We argue in this paper that the computer architecture has to be redefined or adapted for real-time systems. Predictable and *analyzable* execution time is of primary importance for this computer architecture.

2.3 JOP Architecture

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

The name JOP stands for *Java Optimized Processor* to enforce that the microcode instructions are optimized for Java bytecode. A direct implementation of all bytecodes [30] in hardware is not a useful approach. Some bytecodes are very complex (e.g., new has to interact with the garbage collector) and the dynamic instruction frequency is low [36, 16]. All available Java processors implement only a subset of the instructions in hardware.

Figure 2.1 shows JOP's major function units. A typical configuration of JOP contains the processor core, a memory interface and a number of IO devices. The module extension provides the link between the processor core, and the memory and IO modules.

The processor core contains the three microcode pipeline stages *microcode fetch*, *decode* and *execute* and an additional translation stage *bytecode fetch*. The ports to the other modules are the two top elements of the stack (A and B), input to the top-of-stack (Data), bytecode cache address and

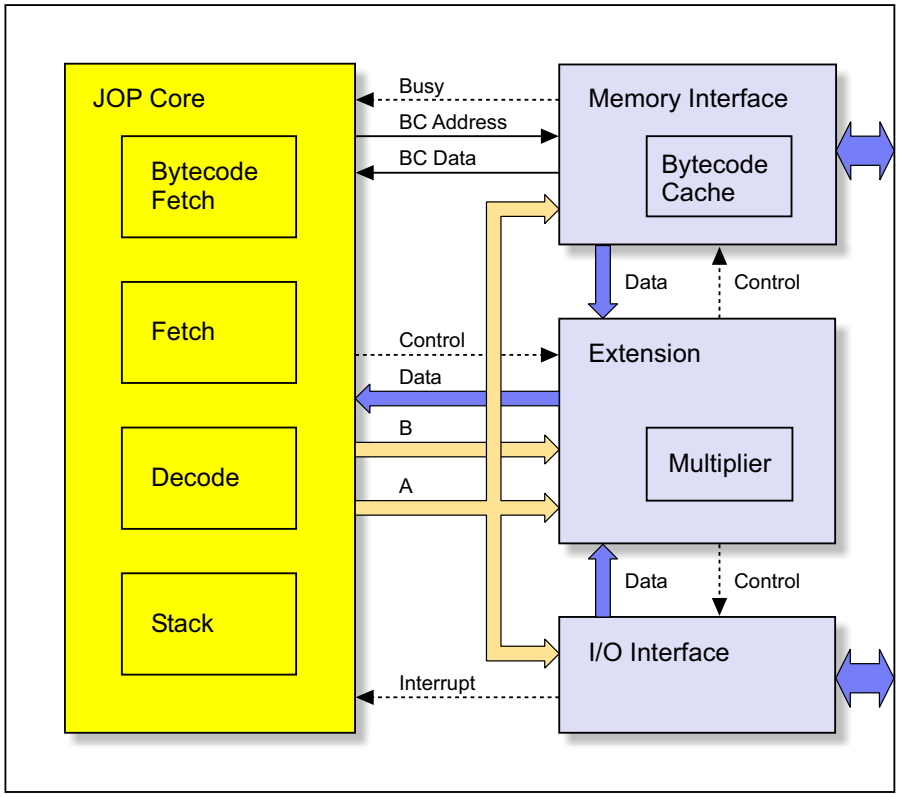


Figure 2.1: Block diagram of JOP

data, and a number of control signals. There is no direct connection between the processor core and the external world.

The memory interface provides a connection between the main memory and the processor core. It also contains the bytecode cache. The extension module controls data read and write. The *busy* signal is used by the microcode instruction wait[2] to synchronize the processor core with the memory unit. The core reads bytecode instructions through dedicated buses (BC address and BC data) from the memory subsystem. The request for a method to be placed in the cache is performed through the extension module, but the cache hit detection and load is performed by the memory interface independently of the processor core (and therefore concurrently).

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

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

[2]The busy signal can also be used to stall the whole processor pipeline. This was the change made to JOP by Flavius Gruian [17]. However, in this synchronization mode, the concurrency between the memory access module and the main pipeline is lost.

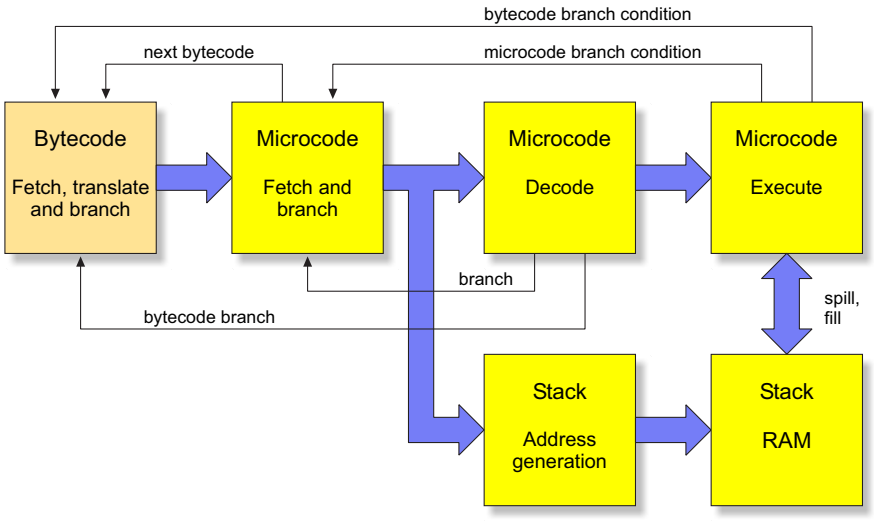


Figure 2.2: Datapath of JOP

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

2.3.1 The Processor Pipeline

JOP is a fully pipelined architecture with single cycle execution of microcode instructions and a novel approach of translation from Java bytecode to these instructions. Figure 2.2 shows the datapath for JOP, representing the pipeline from left to right. Blocks arranged vertically belong to the same pipeline stage.

Three stages form the JOP core pipeline, executing microcode instructions. An additional stage in the front of the core pipeline fetches Java bytecodes – the instructions of the JVM – and translates these bytecodes into addresses in microcode. Bytecode branches are also decoded and executed in this stage. The second pipeline stage fetches JOP instructions from the internal microcode memory and executes microcode branches. Besides the usual decode function, the third pipeline stage also generates addresses for the stack RAM (the stack cache). As every stack machine microcode instruction (except nop, wait, and jbr) has either *pop* or *push* characteristics, it is possible to generate fill or spill addresses for the *following* instruction at this stage. The last pipeline stage performs ALU operations, load, store and stack spill or fill. At the execution stage, operations are performed with the two topmost elements of the stack.

A stack machine with two explicit registers for the two topmost stack elements and automatic fill/spill to the stack cache needs neither an extra write-back stage nor any data forwarding. Figure 2.3 shows the architecture of the execution stage with the two-level stack cache. The operands for the ALU operation reside in the two registers. The result is written in the same cycle into register *A* again. That means execute and write back is performed in a single pipeline stage.

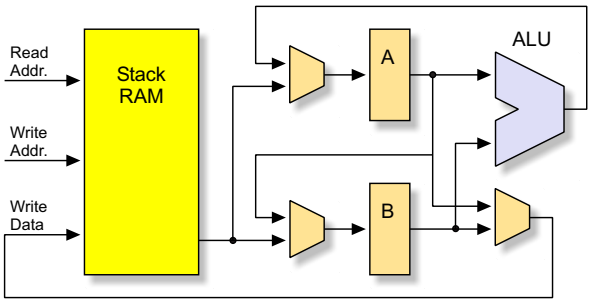


Figure 2.3: The execution stage with the two-level stack cache

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

Case 1: ALU operation
$A \leftarrow A\ op\ B$
$B \leftarrow sm[p]$
$p \leftarrow p - 1$
The two operands are provided by the two top level registers. A single read access from *sm* is necessary to fill B with a new value.

Case 2: Variable load (*Push*)
$A \leftarrow sm[v+n]$
$B \leftarrow A$
$sm[p+1] \leftarrow B$
$p \leftarrow p + 1$
One read access from *sm* is necessary for the variable read. The former TOS value moves down to B and the data previously in B is written to *sm*.

Case 3: Variable store (*Pop*)
$sm[v+n] \leftarrow A$
$A \leftarrow B$
$B \leftarrow sm[p]$
$p \leftarrow p - 1$
The TOS value is written to *sm*. A is filled with B and B is filled in an identical manner to Case 1, needing a single read access from *sm*.

We can see that all three basic operations can be performed with a stack memory with one read and one write port. Assuming a memory is used that can handle concurrent read and write access, there is no structural access conflict between A, B and *sm*. That means that all operations can be performed concurrently in a single cycle. Further details of this two-level stack architecture, and that there are no RAW conflicts, are described in [50].

The short pipeline results in a short delay for a conditional branch. Therefore, a hard to analyze (with respect to WCET) branch prediction logic can be avoided. One question remains: Is the

pipeline well balanced? Compared to other FPGA designs (see Section 2.5) the maximum frequency is quite high. To evaluate if we could do better we performed some experiments by adding pipeline stages in the critical path. In the 4-stage pipeline the critical path is in the first stage, the bytecode fetch and translation stage (100 MHz). Pipelining this unit increased the maximum frequency to 106 MHz and moved the critical path to the execution stage (the barrel shifter). Pipelining this barrel shifter resulted in 111 MHz and the critical path moved to the feedback of the branch condition (located in the microcode fetch stage). Pipelining this path moved the critical path to the microcode decode stage. That means that not a single stage dominates the critical path. From these experiments we conclude that the design with four pipeline stages result in a well balanced design.

2.3.2 Interrupt Logic

Interrupts and (precise) exceptions are considered hard to implement in a pipelined processor [21], meaning implementation tends to be complex (and therefore resource consuming). In JOP, the bytecode-microcode translation is used cleverly to avoid having to handle interrupts and exceptions (e.g., stack overflow) in the core pipeline. Interrupts are implemented as special bytecodes. These bytecodes are inserted by the hardware in the Java instruction stream. When an interrupt is pending and the next fetched byte from the bytecode cache is an instruction, the associated special bytecode is used instead of the instruction from the bytecode cache. The result is that interrupts are accepted at bytecode boundaries. The worst-case preemption delay is the execution time of the *slowest* bytecode that is implemented in microcode. Bytecodes that are implemented in Java (see Section 2.3.4) can be interrupted.

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

This special bytecode can result in a call of a JVM internal method in the context of the interrupted thread. This mechanism implicitly stores almost the complete context of the current active thread on the stack. This feature is used to implement the preemptive, fixed priority real-time scheduler in Java [47].

2.3.3 Cache

A pipelined processor architecture calls for higher memory bandwidth. A standard technique to avoid processing bottlenecks due to the lower available memory bandwidth is caching. However, standard cache organizations improve the average execution time but are difficult to predict for WCET analysis [20]. Two time-predictable caches are proposed for JOP: a *stack cache* as a substitution for the data cache and a *method cache* to cache the instructions.

As the stack is a heavily accessed memory region, the stack – or part of it – is placed in on-chip memory. This part of the stack is referred to as the *stack cache* and described in [50]. The *stack cache* is organized in two levels: the two top elements are implemented as registers, the lower level as a large on-chip memory. Fill and spill between these two levels is done in hardware. Fill and spill between the on-chip memory and the main memory is subjected to microcode control and therefore time-predictable. The exchange of the on-chip stack cache with the main memory can be either done on method invocation and return or on a thread switch.

In [49], a novel way to organize an instruction cache, as *method cache*, is given. The idea is to cache complete methods. A cache fill from main memory is only performed on a miss on method

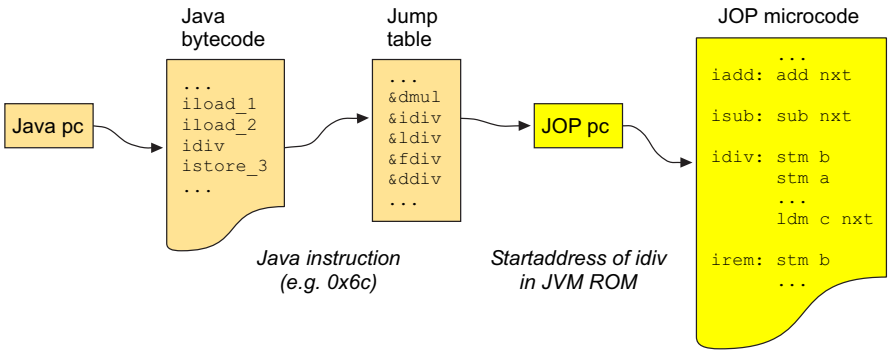


Figure 2.4: Data flow from the Java program counter to JOP microcode

invocation or return. Therefore, all other bytecodes have a guaranteed cache hit. That means no instruction can stall the pipeline.

The cache is organized in blocks, similar to cache lines. However, the cached method has to span continuous[3] blocks. The *method cache* can hold more than one method. Cache block replacement depends on the call tree, instead of instruction addresses. This *method cache* is easy to analyze with respect to worst-case behavior and still provides substantial performance gain when compared against a solution without an instruction cache. The average case performance of this *method cache* is similar to a direct mapped cache [49]. The maximum method size is restricted by the size of the method cache. The pre-link tool verifies that the size restriction is fulfilled by the application.

2.3.4 Microcode

The following discussion concerns two different instruction sets: *bytecode* and *microcode*. Bytecodes are the instructions that make up a compiled Java program. These instructions are executed by a Java virtual machine. The JVM does not assume any particular implementation technology. Microcode is the native instruction set for JOP. Bytecodes are translated, during their execution, into JOP microcode. Both instruction sets are designed for an extended[4] stack machine.

Translation of Bytecodes to Microcode

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

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

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

[3] The cache addresses wrap around at the end of the on-chip memory. Therefore, a method is also considered continuous when it spans from the last to the first block.

[4] An extended stack machine contains instructions that make it possible to access elements deeper down in the stack.

Another common solution, used in Sun's picoJava [58], is to execute a subset of the bytecode native and to use a software trap to execute the remainder. This solution entails an overhead (a minimum of 16 cycles in picoJava) for the software trap.

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

Figure 2.4 gives an example of the data flow from the Java program counter to JOP microcode. The figure represents the two pipeline stages bytecode fetch/translate and microcode fetch. The fetched bytecode acts as an index for the jump table. The jump table contains the start addresses for the bytecode implementation in microcode. This address is loaded into the JOP program counter for every bytecode executed. JOP executes the sequence of microcode until the last one. The last one is marked with *nxt* in microcode assembler. This *nxt* bit in the microcode ROM triggers a new translation i.e., a new address is loaded into the JOP program counter. In Figure 2.4 the implementation of bytecode idiv is an example of a longer sequence that ends with microcode instruction ldm c nxt.

Some bytecodes, such as ALU operations and the short form access to *locals*, are directly implemented by an equivalent microcode instruction. Additional instructions are available to access internal registers, main memory and I/O devices. A relative conditional branch (zero/non zero of the top-of-stack) performs control flow decisions at the microcode level. A detailed description of the microcode instructions can be found in [51].

The difference to other forms of instruction translation in hardware is that the proposed solution is time predictable. The translation takes one cycle (one pipeline stage) for each bytecode, independent from the execution history. Instruction folding, e.g., implemented in picoJava [36, 58], is also a form of instruction translation in hardware. Folding is used to translate several (stack oriented) bytecode instructions to a RISC type instruction. This translation needs an instruction buffer and the fill level of this instruction buffer depends on the execution history. The length of this history that has to be considered for analysis is not bounded. Therefore this form of instruction translation is not exactly time predictable.

Bytecode Example

The example in Figure 2.5 shows the implementation of a single cycle bytecode and an infrequent bytecode as a sequence of JOP instructions. The suffix nxt marks the last instruction of the microcode sequence. In this example, the dup bytecode is mapped to the equivalent dup microcode and executed in a single cycle, whereas dup_x1 takes five cycles to execute, and after the last instruction (ldm a nxt), the first instruction for the next bytecode is executed. The scratch variables, as shown in the second example, are stored in the on-chip memory that is shared with the stack cache.

Some bytecodes are followed by operands of between one and three bytes in length (except lookupswitch and tableswitch). Due to pipelining, the first operand byte that follows the bytecode instruction is available when the first microcode instruction enters the execution stage. If this is a one-byte long operand, it is ready to be accessed. The increment of the Java program counter after the read of an operand byte is coded in the JOP instruction (an *opd* bit similar to the *nxt* bit).

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

//  a and b are scratch variables at
//  the microcode level.

dup_x1: stm a      // save TOS
          stm b      // and TOS-1
          ldm a      // duplicate former TOS
          ldm b      // restore TOS-1
          ldm a nxt  // restore TOS and fetch
                     // the next bytecode
```

Figure 2.5: Implementation of dup and dup_x1

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

Figure 2.6: Bytecode operand load

In Listing 2.6, the implementation of sipush is shown. The bytecode is followed by a two-byte operand. Since the access to bytecode memory is only one[5] byte per cycle, *opd* and *nxt* are not allowed at the same time. This implies a minimum execution time of $n+1$ cycles for a bytecode with n operand bytes.

Flexible Implementation of Bytecodes

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

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

2.3.5 Architecture Summary

In this section, we have introduced JOP's architecture. In order to handle the great variation in the complexity of Java bytecodes we have proposed a translation to a different instruction set, the so-

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

called microcode. This microcode is still an instruction set for a stack machine, but more RISC-like than the CISC-like JVM bytecodes. The core of the stack machine constitutes a three-stage pipeline. An additional pipeline stage in front of this core pipeline stage performs bytecode fetch and the translation to microcode. This organization has no execution time overheads for more complex bytecodes and results in the short pipeline that is necessary for any processor without branch prediction. The additional translation stage also presents an elegant way of incorporating interrupts virtually *for free*. Only a multiplexor is needed in the path from the translation stage to the microcode decode stage. The microcode scratch variables are only valid during a microcode sequence for a bytecode and need not be saved on an interrupt.

At the time of this writing 43 of the 201 different bytecodes are implemented by a single microcode instruction, 93 by a microcode sequence, and 40 bytecodes are implemented in Java.

2.4 Worst-Case Execution Time

Worst-case execution time (WCET) estimates of tasks are essential for designing and verifying real-time systems. WCET estimates can be obtained either by measurement or static analysis. The problem with using measurements is that the execution times of tasks tend to be sensitive to their inputs. As a rule, measurement does not guarantee safe WCET estimates. Instead, static analysis is necessary for hard real-time systems. Static analysis is usually divided into a number of different phases:

Path analysis generates the control flow graph (a directed graph of basic blocks) of the program and annotates (manual or automatic) loops with bounds.

Low-level analysis determines the execution time of basic blocks obtained by the path analysis. A model of the processor and the pipeline provides the execution time for the instruction sequence.

Global low-level analysis determines the influence of hardware features such as caches on program execution time. This analysis can use information from the path analysis to provide less pessimistic values.

WCET Calculation collapses the control flow graph to provide the final WCET estimate. Alternative paths in the graph are collapsed to a single value (the largest of the alternatives) and loops are collapsed once the loop bound is known.

For the low-level analysis, a good timing model of the processor is needed. The main problem for the low-level analysis is the execution time dependency of instructions in modern processors that are not designed for real-time systems. JOP is designed to be an easy target for WCET analysis. The WCET of each bytecode can be predicted in terms of number of cycles it requires. There are no dependencies between bytecodes.

Each bytecode is implemented by microcode. We can obtain the WCET of a single bytecode by performing WCET analysis at the microcode level. To prove that there are no time dependencies between bytecodes, we have to show that no processor states are *shared* between different bytecodes.

WCET analysis has to be done at two levels: at the microcode level and at the bytecode level. The microcode WCET analysis is performed only once for a processor configuration and described in the next sections. The result from this microcode analysis is the timing model of the processor. The

timing model is the input for the WCET analysis at the bytecode level (i.e. the Java application) as shown in the example in Section 2.4.5 and in the WCET tool description in Section 2.4.5.

It has to be noted that we cannot provide WCET values for the other Java systems from Section 2.6, e.g. the aJile Java processor, as there is no information on the instruction timing available.

2.4.1 Microcode Path Analysis

To obtain the WCET values for the individual bytecodes we perform the path analysis at the microcode level. First, we have to ensure that a number of restrictions (from [42]) of the code are fulfilled:

- Programs must not contain unbounded recursion. This property is satisfied by the fact that there exists no call instruction in microcode.
- Function pointers and computed gotos complicate the path analysis and should therefore be avoided. Only simple conditional branches are available at the microcode level.
- The upper bound of each loop has to be known. This is the only point that has to be verified by inspection of the microcode.

To detect loops in the microcode we have to find all backward branches (e.g. with a negative branch offset)[6]. The branch offsets can be found in a VHDL file (offtbl.vhd) that is generated during microcode assembly. In the current implementation of the JVM there are ten different negative offsets. However, not each offset represents a loop. Most of these branches are used to share common code. Three branches are found in the initialization code of the JVM. They are not part of a bytecode implementation and can be ignored. The only loop that is found in a regular bytecode is in the implementation of imul to perform a fixed delay. The iteration count for this loop is constant.

A few bytecodes are implemented in Java[7] and can be analyzed in the same way as application code. The bytecodes idiv and irem contain a constant loop. The bytecodes new and anewarray contain loops to initialize (with zero values) new objects or arrays. The loop is bound by the size of the object or array. The bytecode lookupswitch[8] performs a linear search through a table of branch offsets. The WCET depends on the table size that can be found as part of the instruction.

As the microcode sequences are very short, the calculation of the control flow graph for each bytecode is done manually.

2.4.2 Microcode Low-level Analysis

To calculate the execution time of basic blocks in the microcode, we need to establish the timing of microcode instructions on JOP. All microcode instructions except wait execute in a single cycle, reducing the low-level analysis to a case of merely counting the instructions.

The wait instruction is used to stall the processor and wait for the memory subsystem to finish a memory transaction. The execution time of the wait instruction depends on the memory system and, if the memory system is predictable, has a known WCET. A main memory consisting of SRAM chips can provide this predictability and this solution is therefore advised. The predictable handling

[6] The loop branch can be a forward branch. However, the basic blocks of the loop contain at least one backward branch. Therefore we can identify all loops by searching for backward branches only.

[7] The implementation can be found in the class com.jopdesign.sys.JVM.

[8] lookupswitch is one way of implementing the Java switch statement. The other bytecode, tableswitch, uses an index in the table of branch offsets and has therefore a constant execution time.

of DMA, which is used for the instruction cache fill, is explained in [49]. The wait instruction is the only way to stall the processor. Hardware events, such as interrupts (see [46]), do not stall the processor.

Microcode is stored in on-chip memory with single cycle access. Each microcode instruction is a single word long and there is no need for either caching or prefetching at this stage. We can therefore omit performing a low-level analysis. No pipeline analysis [13], with its possible unbound timing effects, is necessary.

2.4.3 Bytecode Independency

We have seen that all microcode instructions except wait take one cycle to execute and are therefore independent of other instructions. This property directly translates to independency of bytecode instructions.

The wait microcode instruction provides a convenient way to hide memory access time. A memory read or write can be triggered in microcode and the processor can continue with microcode instructions. When the data from a memory read is needed, the processor explicitly waits, with the wait instruction, until it becomes available.

For a memory store, this wait could be deferred until the memory system is used next (similar to a write buffer). It is possible to initiate the store in a bytecode such as putfield and continue with the execution of the next bytecode, even when the store has not been completed. In this case, we introduce a dependency over bytecode boundaries, as the state of the memory system is *shared*. To avoid these dependencies that are difficult to analyze, each bytecode that accesses memory waits (preferably at the end of the microcode sequence) for the completion of the memory request.

Furthermore, if we would not wait at the end of the store operation we would have to insert an additional wait at the start of every read operation. Since read operations are more frequent than write operations (15% vs. 2.5%, see [51]), the performance gain from the hidden memory store is lost.

2.4.4 WCET of Bytecodes

The control flow of the individual bytecodes together with the basic block length (that directly corresponds with the execution time) and the time for memory access result in the WCET (and BCET) values of the bytecodes. These exact values for each bytecode can be found in [51].

Simple bytecode instructions are executed by either one microinstruction or a short sequence of microinstructions. The execution time in cycles equals the number of microinstructions executed. As the stack is on-chip it can be accessed in a single cycle. We do not need to incorporate the main memory timing into the instruction timing. Table 2.2 shows examples of the execution time of such bytecodes.

Object oriented instructions, array access, and invoke instructions access the main memory. Therefore we have to model the memory access time. We assume a simple SRAM with a constant access time. Access time that exceeds a single cycle includes additional wait states (r_{ws} for a memory read and w_{ws} for a memory write). The following example gives the execution time for getfield, the read access of an object field:

$$t_{getfield} = 10 + 2r_{ws}$$

However, the memory subsystem performs read and write parallel to the execution of microcode. Therefore, some access cycles can be hidden. The following example gives the exact execution time

Opcode	Instruction	Cycles	Funtion
3	iconst_0	1	load constant 0 on TOS
4	iconst_1	1	load constant 1 on TOS
16	bipush	2	load a byte constant on TOS
17	sipush	3	load a short constant on TOS
21	iload	2	load a local on TOS
26	iload_0	1	load local 0 on TOS
27	iload_1	1	load local 1 on TOS
54	istore	2	store the TOS in a local
59	istore_0	1	store the TOS in local 0
60	istore_1	1	store the TOS in local 1
89	dup	1	duplicate TOS
90	dup_x1	5	complex stack manipulation
96	iadd	1	integer addition
153	ifeq	4	conditional branch

Table 2.2: Execution time of simple bytecodes in cycles

of bytecode ldc2_w in clock cycles:

$$t_{ldc2\_w} = 17 + \begin{cases} r_{ws} - 2 & : \quad r_{ws} > 2 \\ 0 & : \quad r_{ws} \leq 2 \end{cases} + \begin{cases} r_{ws} - 1 & : \quad r_{ws} > 1 \\ 0 & : \quad r_{ws} \leq 1 \end{cases}$$

Thus, for a memory with two cycles access time ($r_{ws} = 1$), as we use it for a 100 MHz version of JOP with a 15 ns SRAM, the wait state is completely hidden by microcode instructions for this bytecode.

Memory access time also determines the cache load time on a miss. For the current implementation the cache load time is calculated as follows: the wait state c_{ws} for a single word cache load is:

$$c_{ws} = \begin{cases} r_{ws} - 1 & : \quad r_{ws} > 1 \\ 0 & : \quad r_{ws} \leq 1 \end{cases}$$

On a method invoke or return the bytecode has to be loaded into the cache on a cache miss. The load time l is:

$$l = \begin{cases} 6 + (n+1)(2 + c_{ws}) & : \quad \text{cache miss} \\ 4 & : \quad \text{cach hit} \end{cases}$$

where n is the length of the method in number of 32-bit words. For short methods the load time of the method on a cache miss, or part of it, is hidden by microcode execution. As an example the exact execution time for the bytecode invokestatic is:

$$\begin{aligned} t &= 74 + r + \begin{cases} r_{ws} - 3 & : \quad r_{ws} > 3 \\ 0 & : \quad r_{ws} \leq 3 \end{cases} + \begin{cases} r_{ws} - 2 & : \quad r_{ws} > 2 \\ 4 & : \quad r_{ws} \leq 2 \end{cases} \\ &+ \begin{cases} l - 37 & : \quad l > 37 \\ 0 & : \quad l \leq 37 \end{cases} \end{aligned}$$

For invokestatic a cache load time l of up to 37 cycles is completely hidden. For the example SRAM timing the cache load of methods up to 36 bytes long is hidden. The WCET analysis tool, as described in the next section, knows the length of the invoked method and can therefore calculate the time for the invoke instruction cycle accurate.

```
final static int N = 5;

static void sort(int[] a) {

    int i, j, v1, v2;
    // loop count = N-1
    for (i=N-1; i>0; --i) {
        // loop count = (N-1)*N/2
        for (j=1; j<=i; ++j) {
            v1 = a[j-1];
            v2 = a[j];
            if (v1 > v2) {
                a[j] = v1;
                a[j-1] = v2;
            }
        }
    }
}
```

Figure 2.7: Bubble Sort test program for the WCET analysis

2.4.5 WCET Analysis of the Java Application

We conclude this section with a worst-case analysis (now at the bytecode level) of Java applications. First we provide manual analysis on a simple example and than a brief description of the automation through a WCET analyzer tool.

An Example

In this section we perform manually a worst and best case analysis of a classic example, the Bubble Sort algorithm. The values calculated are compared with the measurements of the execution time on JOP on all permutations of the input data. Figure 2.7 shows the test program in Java. The algorithm contains two nested loops and one condition. We use an array of five elements to perform the measurements for all permutations (i.e. $5! = 120$) of the input data. The number of iterations of the outer loop is one less than the array size: $c_1 = N - 1$, in this case four. The inner loop is executed $c_2 = \sum_{i=1}^{c_1} i = c_1(c_1+1)/2$ times, i.e. ten times in our example.

The annotated control flow graph (CFG) of the example is shown in Figure 2.8. The edges contain labels showing how often the path between two nodes is taken. We can identify the outer loop, containing the blocks B2, B3, B4 and B8. The inner loop consists of blocks B4, B5, B6 and B7. Block B6 is executed when the condition of the if statement is true. The path from B5 to B7 is the only path that depends on the input data.

In Table 2.3 the basic blocks with the start address (Addr.) and their execution time (Cycles) in clock cycles and the worst and best case execution frequency (Count) is given. The values in the forth and sixth columns (Count) of Table 2.3 are derived from the CFG and show how often the basic blocks are executed in the worst and best cases. The WCET and BCET value for each block is calculated by multiplying the clock cycles by the execution frequency. The overall WCET and BCET values are calculated by summing the values of the individual blocks B1 to B8. The last block (B9) is omitted, as the measurement does not contain the return statement.

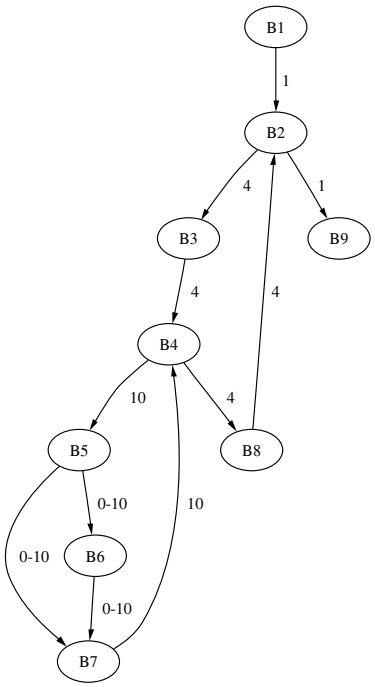


Figure 2.8: The control flow graph of the Bubble Sort example

			WCET		BCET	
Block	Addr.	Cycles	Count	Total	Count	Total
B1	0:	2	1	2	1	2
B2	2:	5	5	25	5	25
B3	6:	2	4	8	4	8
B4	8:	6	14	84	14	84
B5	13:	74	10	740	10	740
B6	30:	73	10	730	0	0
B7	41:	15	10	150	10	150
B8	47:	15	4	60	4	60
B9	53:		1		1	
Execution time calculated				1799		1069
Execution time measured				1799		1069

Table 2.3: WCET and BCET in clock cycles of the basic blocks

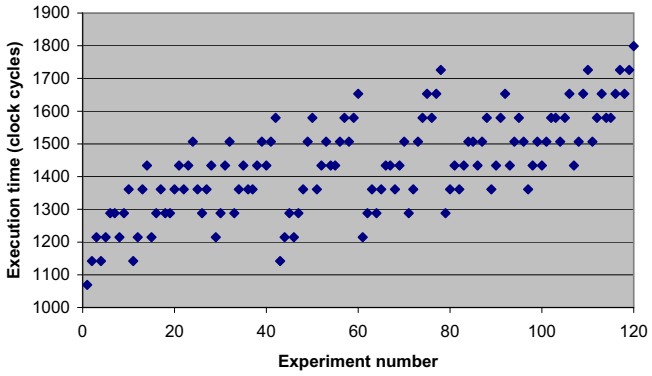


Figure 2.9: Execution time in clock cycles of the Bubble Sort program for all 120 permutations of the input data

The execution time of the program is measured using the cycle counter in JOP. The current time is taken at both the entry of the method and at the end, resulting in a measurement spanning from block B1 to the beginning of block B9. The last statement, the return, is not part of the measurement. The difference between these two values (less the additional 8 cycles introduced by the measurement itself) is given as the execution time in clock cycles (the last row in Table 2.3). The measured WCET and BCET values are exactly the same as the calculated values.

In Figure 2.9, the measured execution times for all 120 permutations of the input data are shown. The vertical axis shows the execution time in clock cycles and the horizontal axis the number of the test run. The first input sample is an already sorted array and results in the lowest execution time. The last sample is the worst-case value resulting from the reversely ordered input data. We can also see the 11 different execution times that result from executing basic block B6 (which performs the element exchange and takes 73 clock cycles) between 0 and 10 times.

WCET Analyzer

In [53] we have presented a static WCET analysis tool for Java. During the high-level analysis the the relevant information is extracted from the class files. The control flow graph (CFG) of the basic blocks[9] is extracted from the bytecodes. Annotations for the loop counts are extracted from comments in the source. Furthermore, the class hierarchy is examined to find all possible targets for a method invoke.

The tool performs the low-level analysis at the bytecode level. The behavior of the method cache is integrated for a simpler form (a two block cache). The well known execution times of the different bytecodes (see Section 2.4.4) simplifies this part of the WCET analysis, which is usually the most complex one, to a great extent. As there are no pipeline dependencies the calculation of the execution time for a basic block is merely just adding the individual cycles for each instruction.

The actual calculation of the WCET is transformed to an integer linear programming problem, a well known technique for WCET analysis [43, 29]. We performed the WCET analysis on several benchmarks (see Table 2.4). We also *measured* the WCET values for the benchmarks. It has to be noted that we actually cannot measure the real WCET. If we could measure it, we would not need to perform the WCET analysis at all. The measurement gives us an idea of the pessimism of the analyzed WCET. The benchmarks Lift and Kfl are real-world examples that are in industrial use. Kfl

[9] A basic block is a sequence of instructions without any jumps or jump targets within this sequence.

Program	Description	LOC
crc	CRC calculation for short messages	8
robot	A simple line follower robot	111
Lift	A lift controler	635
Kfl	*Kippfahrleitung* application	1366
UdpIp	UDP/IP benchmark	1297

Table 2.4: WCET benchmark examples

Program	Measured (cycle)	Estimated (cycle)	Pessimism (ratio)
crc	1552	1620	1.04
robot	736	775	1.05
Lift	7214	11249	1.56
Kfl	13334	28763	2.16
UdpIp	11823	219569	18.57

Table 2.5: Measured and estimated WCETs with results in clock cycles

and UdpIp are also part of an embedded Java benchmark suit that is used in Section 2.6.

Table 2.5 shows the measured execution time and the analyzed WCET. The last column gives an idea of the pessimism of the WCET analysis. For very simple programs, such as crc and robot, the pessimism is quite low. For the Lift example it is in an acceptable range. The difference between the measurement and the analysis in the Kfl example results from the fact that our measurement does not cover the WCET path. The large conservatism in UdpIp results from the loop bound in the IP and UDP checksum calculation. It is set for a 1500 byte packet buffer, but the payload in the benchmark is only 8 bytes. The last two examples also show the issue when a real-time application is developed without a WCET analysis tool available.

The WCET analysis tool, with the help of loop annotations, provides WCET values for the schedulability analysis. Besides the calculation of the WCET the tool provides user feedback by generating bytecode listings with timing information and a graphical representation of the CFG with timing and frequency information. This representation of the WCET path through the code can guide the developer to write WCET aware real-time code.

2.4.6 Discussion

The Bubble Sort example and experiments with the WCET analyzer tool have demonstrated that we have achieved our goal: JOP is a simple target for the WCET analysis. Most bytecodes have a single execution time (WCET = BCET), and the WCET of a task (the analysis at the bytecode level) depends only on the control flow. No pipeline or data dependencies complicate the low-level part of the WCET analysis.

The same analysis is not possible for other Java processors. Either the information on the bytecode execution time is missing[10] or some processor features (e.g., the high variability of the latency for a

[10]We tried hard to get this information for the aJile processor.

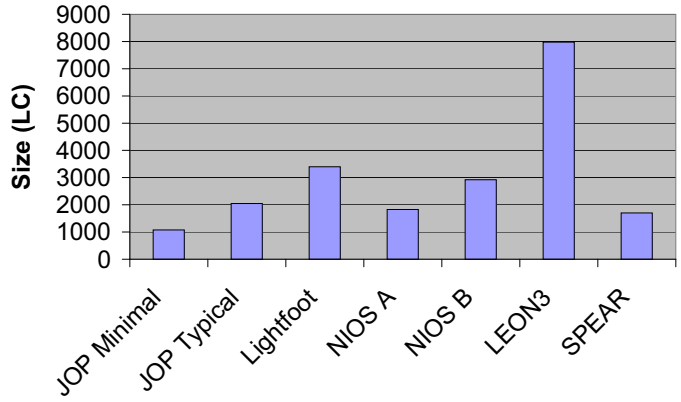


Figure 2.10: Size in logic cells (LC) of different soft-core processors

trap in picoJava) would result in very conservative WCET estimates. Another example that prohibits exact analysis is the mechanism to automatically fill and spill the stack cache in picoJava. The time when the memory (cache) is occupied by this spill/fill action depends on a long instruction history. Also the fill level of the 16-byte-deep prefetch buffer, which is needed for instruction folding, depends on the execution history. All this automatically buffering features have to be modeled quite conservative. A pragmatic solution is to assume empty buffers at the start of a basic block. As basic blocks are quite short most of the buffering/prefetching does not help to lower the WCET.

Only for the Cjip processor the execution time is well documented [24]. However, as seen in Section 2.6.2, the *measured* execution time of some bytecodes is *higher* than the documented values. Therefore the documentation is not complete to provide a safe processor model of the Cjip for the WCET analysis.

2.5 Resource Usage

Cost is an important issue for embedded systems. The cost of a chip is directly related to the die size (the cost per die is roughly proportional to the square of the die area [21]). Processors for embedded systems are therefore optimized for minimum chip size. In this section, we will compare JOP with different processors in terms of size.

One major design objective in the development of JOP was to create a small system that can be implemented in a low-cost FPGA. Figure 2.10 and Table 2.6 show the resource usage for different configurations of JOP and different soft-core processors implemented in an Altera EP1C6 FPGA [3]. Estimating equivalent gate counts for designs in an FPGA is problematic. It is therefore better to compare the two basic structures, Logic Cells (LC) and embedded memory blocks. The maximum frequency for all soft-core processors is in the same technology or normalized (SPEAR) to the technology.

All configurations of JOP contain the on-chip microcode memory, the 1 KB stack cache, a 1 KB method cache, a memory interface to a 32-bit static RAM, and an 8-bit FLASH interface for the Java program and the FPGA configuration data. The minimum configuration implements multiplication and the shift operations in microcode. In the typical configuration, these operations are implemented as a sequential Booth multiplier and a single-cycle barrel shifter. The typical configuration also contains some useful I/O devices such as an UART and a timer with interrupt logic for multi-threading.

Processor	Resources (LC)	Memory (KB)	fmax (MHz)
JOP Minimal	1077	3.25	98
JOP Typical	2049	3.25	100
Lightfoot[11][9]	3400	4	40
NIOS A [2]	1828	6.2	120
NIOS B [2]	2923	5.5	119
LEON3 [15]	7978	10.9	35
SPEAR[12][10]	1700	8	80

Table 2.6: Size and maximum frequency of FPGA soft-core processors

The typical configuration of JOP consumes about 30% of the LCs in a Cyclone EP1C6, thus leaving enough resources free for application-specific logic.

As a reference, NIOS [2], Altera's popular RISC soft-core, is also included in Table 2.6. NIOS has a 16-bit instruction set, a 5-stage pipeline and can be configured with a 16 or 32-bit datapath. Version A is the minimum configuration of NIOS. Version B adds an external memory interface, multiplication support and a timer. Version A is comparable with the minimal configuration of JOP, and Version B with its typical configuration.

LEON3 [15], the open-source implementation of the SPARC V8 architecture, has been ported to the exact same hardware that was used for the JOP numbers. LEON3 is a representative of a RISC processor that is used in embedded real-time systems (e.g., by ESA for space missions).

SPEAR [10] (Scalable Processor for Embedded Applications in Real-time Environments) is a 16-bit processor with deterministic execution times. SPEAR contains predicated instructions to support single-path programming [39]. SPEAR is included in the list as it is also a processor designed for real-time systems.

To prove that the VHDL code for JOP is as portable as possible, JOP was also implemented in a Xilinx Spartan-3 FPGA [66]. Only the instantiation and initialization code for the on-chip memories is vendor-specific, whilst the rest of the VHDL code can be shared for the different targets. JOP consumes about the same LC count (1844 LCs) in the Spartan device, but has a slower clock frequency (83 MHz).

From this comparison we can see that we have achieved our objective of designing a small processor. The commercial Java processor, Lightfoot, consumes 1.7 times the logic resources of JOP in the typical configuration (with a lower clock frequency). A typical 32-bit RISC processor (NIOS) consumes about 1.5 times (LEON about four times) the resources of JOP. However, the NIOS processor can be clocked 20% faster than JOP in the same technology. The vendor independent and open-source RISC processor LEON can be clocked only with 35% of JOP's frequency. The only processor that is similar in size is SPEAR. However, while SPEAR is a 16-bit processor, JOP contains a 32-bit datapath.

[11] The data for the Lightfoot processor is taken from the data sheet [9]. The frequency used is that in a Virtex-II device from Xilinx. JOP can be clocked at 100 MHz in the Virtex-II device, making this comparison valid.

[12] As SPEAR uses internal memory blocks in asynchronous mode it is not possible to synthesize it without modification for the Cyclone FPGA. The clock frequency of SPEAR in an Altera Cyclone is an estimate based on following facts: SPEAR can be clocked at 40 MHz in an APEX device and JOP can be clocked at 50 MHz in the same device.

2.6 Performance

In this section, we will evaluate the performance of JOP in relation to other embedded Java systems. Although JOP is intended as a processor with a low WCET for all operations, its general performance is still important.

2.6.1 General Performance

Running benchmarks is problematic, both generally and especially in the case of embedded systems. The best benchmark would be the application that is intended to run on the system being tested. To get comparable results SPEC provides benchmarks for various systems. However, the one for Java, the SPECjvm98 [55], needs more functionality than what is usually available in a CLDC compliant device (e.g., a filesystem and java.net). Some benchmarks from the SPECjvm98 suits also need several MB of heap.

Due to the absence of a *standard* Java benchmark for embedded systems, a small benchmark suite that should run on even the smallest device is provided here. It contains several micro-benchmarks for evaluating the number of clock cycles for single bytecodes or short sequences of bytecodes, and two application benchmarks.

To provide a realistic workload for embedded systems, a real-time application was adapted to create the first application benchmark (Kfl). The application is taken from one of the nodes of a distributed motor control system [45] (the first industrial application of JOP). The application is written as a cyclic executive. A simulation of both the environment (sensors and actors) and the communication system (commands from the master station) forms part of the benchmark, so as to simulate the real-world workload. The second application benchmark is an adaptation of a tiny TCP/IP stack for embedded Java. This benchmark contains two UDP server/clients, exchanging messages via a loopback device. The Kfl benchmark consists of 511 methods and 14 KB code, the UDP/IP benchmark of 508 methods and 13 KB code (including the supporting library).

As we will see, there is a great variation in processing power across different embedded systems. To cater for this variation, all benchmarks are 'self adjusting'. Each benchmark consists of an aspect that is benchmarked in a loop and an 'overhead' loop that contains any overheads from the benchmark that should be subtracted from the result (this feature is designed for the micro-benchmarks). The loop count adapts itself until the benchmark runs for more than a second. The number of iterations per second is then calculated, which means that higher values indicate better performance.

All the benchmarks measure how often a function is executed per second. In the Kfl benchmark, this function contains the main loop of the application that is executed in a periodic cycle in the original application. In the benchmark the wait for the next period is omitted, so that the time measured solely represents execution time. The UDP benchmark contains the generation of a request, transmitting it through the UDP/IP stack, generating the answer and transmitting it back as a benchmark function. The iteration count is the number of received answers per second.

The accuracy of the measurement depends on the resolution of the system time. For the measurements under Linux, the system time has a resolution of 10ms, resulting in an inaccuracy of 1%. The accuracy of the system time on leJOS, TINI and the aJile is not known, but is considered to be in the same range. For JOP, a μs counter is used for time measurement.

The following list gives a brief description of the Java systems that were benchmarked:

JOP is implemented in a Cyclone FPGA [3], running at 100 MHz. The main memory is a 32-bit SRAM (15ns) with an access time of 2 clock cycles. The benchmarked configuration of JOP contains a 4 KB method cache organized in 16 blocks.

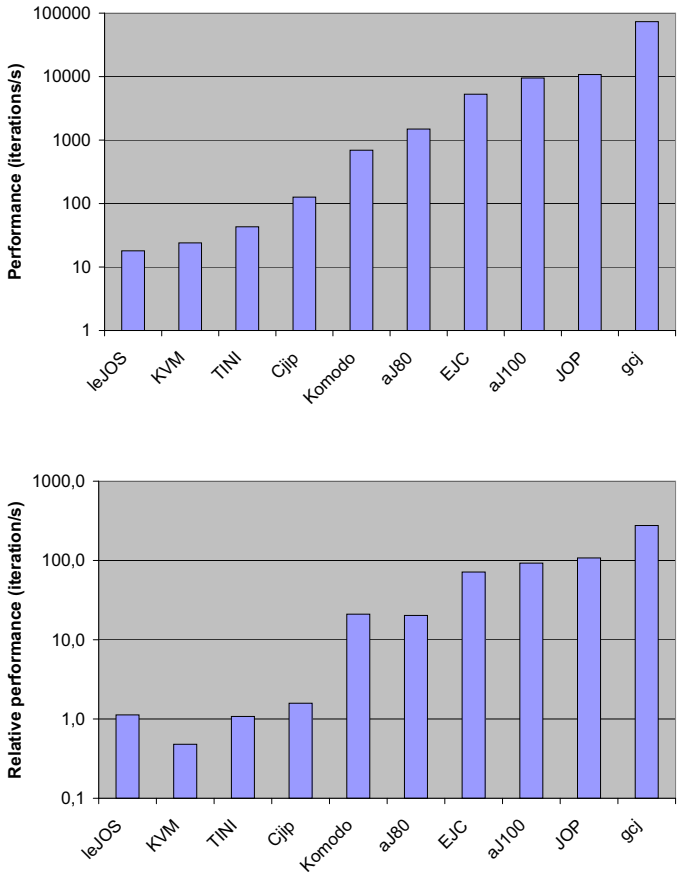


Figure 2.11: Performance comparison of different Java systems with application benchmarks. The diagrams show the geometric mean of the two benchmarks in iterations per second – a higher value means higher performance. The top diagram shows absolute performance, while the bottom diagram shows the result scaled to 1 MHz clock frequency.

leJOS As an example for a low-end embedded device we use the RCX robot controller from the LEGO MindStorms series. It contains a 16-bit Hitachi H8300 microcontroller [22], running at 16 MHz. leJOS [54] is a tiny interpreting JVM for the RCX.

KVM is a port of the Sun's KVM that is part of the Connected Limited Device Configuration (CLDC) [57] to Alteras NIOS II processor on MicroC Linux. NIOS is implemented on a Cyclone FPGA and clocked with 50 MHz. Besides the different clock frequency this is a good comparison of an interpreting JVM running in the same FPGA as JOP.

TINI is an enhanced 8051 clone running a software JVM. The results were taken from a custom board with a 20 MHz crystal, and the chip's PLL is set to a factor of 2.

Cjip The measured system [23] is a replacement of the TINI board and contains a Cjip [25] clocked with 80 MHz and 8 MB DRAM.

The benchmark results of **Komodo** were obtained by Matthias Pfeffer [37] on a cycle-accurate simulation of Komodo.

aJile's JEMCore is a direct-execution Java processor that is available in two different versions: the **aJ80** and the **aJ100** [1]. The aJ100 provides a generic 8-bit, 16-bit or 32-bit external bus interface,

	Frequency (MHz)	Kfl	UDP/IP	Geom. Mean (Iterations/s)	Scaled
JOP	100	17111	6781	10772	108
leJOS	16	25	13	18	1.1
TINI	40	64	29	43	1.1
KVM	50	36	16	24	0.5
Cjip	80	176	91	127	1.6
Komodo	33	924	520	693	21
aJ80	74	2221	1004	1493	20
aJ100	103	14148	6415	9527	92
EJC	74	9893	2822	5284	71
gcj	266	139884	38460	73348	276
MB	100	3792			

Table 2.7: Application benchmarks on different Java systems. The table shows the benchmark results in iterations per second – a higher value means higher performance.

while the aJ80 only provides an 8-bit interface.

The **EJC** (Embedded Java Controller) platform [12] is a typical example of a JIT system on a RISC processor. The system is based on a 32-bit ARM720T processor running at 74 MHz. It contains up to 64 MB SDRAM and up to 16 MB of NOR flash.

gcj is the GNU compiler for Java. This configuration represents the batch compiler solution, running on a 266 MHz Pentium MMX under Linux.

MB is the realization of Java on a RISC processor for an FPGA (Xilinx MicroBlaze [65]). Java is compiled to C with a Java compiler for real-time systems [35] and the C program is compiled with the standard GNU toolchain.

It would be interesting to include the other soft-core Java processors (Moon, Lightfoot, and FemtoJava) in this comparison. However, it was not possible to obtain the benchmark data. The company that produced Moon seems to be disappeared and FemtoJava could not run all benchmarks.

In Figure 2.11, the geometric mean of the two application benchmarks is shown. The unit used for the result is iterations per second. Note that the vertical axis is logarithmic, in order to obtain useful figures to show the great variation in performance. The top diagram shows absolute performance, while the bottom diagram shows the same results scaled to a 1 MHz clock frequency. The results of the application benchmarks and the geometric mean are shown in Table 2.7.

It should be noted that scaling to a single clock frequency could prove problematic. The relation between processor clock frequency and memory access time cannot always be maintained. To give an example, if we were to increase the results of the 100 MHz JOP to 1 GHz, this would also involve reducing the memory access time from 15 ns to 1.5 ns. Processors with 1 GHz clock frequency are already available, but the fastest asynchronous SRAM to date has an access time of 10 ns.

2.6.2 Discussion

When comparing JOP and the aJile processor against leJOS, KVM, and TINI, we can see that a Java processor is up to 500 times faster than an interpreting JVM on a standard processor for an embedded system. The average performance of JOP is even better than a JIT-compiler solution on an embedded system, as represented by the EJC system.

	JOP	leJOS	TINI	Cjip	Komodo	aJ80	aJ100
iload iadd	2	836	789	55	8	38	8
iinc	8	422	388	46	4	41	11
ldc	9	1340	1128	670	40	67	9
if_icmplt taken	6	1609	1265	157	24	42	18
if_icmplt n/taken	6	1520	1211	132	24	40	14
getfield	22	1879	2398	320	48	142	23
getstatic	15	1676	4463	3911	80	102	15
iaload	36	1082	1543	139	28	74	13
invoke	128	4759	6495	5772	384	349	112
invoke static	100	3875	5869	5479	680	271	92
invoke interface	144	5094	6797	5908	1617	531	148

Table 2.8: Execution time in clock cycles for various JVM bytecodes

Even when scaled to the same clock frequency, the compiling JVM on a PC (gcj) is much faster than either embedded solution. However, the kernel of the application is smaller than 4 KB [49]. It therefore fits in the level one cache of the Pentium MMX. For a comparison with a Pentium class processor we would need a larger application.

JOP is about 7 times faster than the aJ80 Java processor on the popular JStamp board. However, the aJ80 processor only contains an 8-bit memory interface, and suffers from this bottleneck. The SaJe system contains the aJ100 with 32-bit, 10 ns SRAMs. JOP with its 15 ns SRAMs is about 12% faster than the aJ100 processor.

The MicroBlaze system is a representation of a Java batch-compilation system for a RISC processor. MicroBlaze is configured with the same cache[13] as JOP and clocked at the same frequency. JOP is about four times faster than this solution, thus showing that native execution of Java bytecodes is faster than batch-compiled Java on a similar system. However, the results of the MicroBlaze solution are at a preliminary stage[14], as the Java2C compiler [35] is still under development.

The micro-benchmarks are intended to give insight into the implementation of the JVM. In Table 2.8, we can see the execution time in clock cycles of various bytecodes. As almost all bytecodes manipulate the stack, it is not possible to measure the execution time for a single bytecode in the benchmark loop. The single bytecode would trash the stack. As a minimum requirement, a second instruction is necessary in the loop to reverse the stack operation.

For JOP we can deduce that the WCET for simple bytecodes is also the average execution time. We can see that the combination of iload and iadd executes in two cycles, which means that each of these two operations is executed in a single cycle. The iinc bytecode is one of the few instructions that do not manipulate the stack and can be measured alone. As iinc is not implemented in hardware, we have a total of 8 cycles that are executed in microcode. It is fair to assume that this comprises too great an overhead for an instruction that is found in every iterative loop with an integer index. However, the decision to implement this instruction in microcode was derived from the observation that the dynamic instruction count for iinc is only 2% [51].

[13] The MicroBlaze with a 8 KB data and 8 KB instruction cache is about 1.5 times faster than JOP. However, a 16 KB memory is not available in low-cost FPGAs and is an unbalanced system with respect to the LC/memory relation.

[14] As not all language constructs can be compiled, only the Kfl benchmark was measured. Therefore, the bars for MicroBlaze are missing in Figure 2.11.

The sequence for the branch benchmark (if_icmplt) contains the two load instructions that push the arguments onto the stack. The arguments are then consumed by the branch instruction. This benchmark verifies that a branch requires a constant four cycles on JOP, whether it is taken or not.

The Cjip implements the JVM with a stack oriented instruction set. It is the only example (except JOP) where the instruction set is documented *including* the execution time [24]. We will therefore check some of the results with the numbers provided in the documentation. The execution time is given in ns, assuming a 66 MHz clock. The execution time for the basic integer add operation is given as 180 ns resulting in 12 cycles. The load of a local variable (when it is one of the first four) takes 35 cycles. In the micro-benchmark we measure 55 cycles instead of the theoretical 47 (iadd + iload_n). We assume that we have to add some cycles for the fetch of the bytecodes from memory.

For compiling versions of the JVM, these micro-benchmarks do not produce useful results. The compiler performs optimizations that make it impossible to measure execution times at this fine a granularity.

2.7 Conclusion

In this paper, we presented a brief overview of the concepts for a real-time Java processor, called JOP, and the evaluation of this architecture. We have seen that JOP is the smallest hardware realization of the JVM available to date. Due to the efficient implementation of the stack architecture, JOP is also smaller than a *comparable* RISC processor in an FPGA. Implemented in an FPGA, JOP has the highest clock frequency of all known Java processors.

We performed the WCET analysis of the implemented JVM at the microcode level. This analysis provides the WCET and BCET values for the individual bytecodes. We have also shown that there are no dependencies between individual bytecodes. This feature, in combination with the method cache [49], makes JOP an easy target for low-level WCET analysis of Java applications. As far as we know, JOP is the only Java processor for which the WCET of the bytecodes is known and documented.

We compared JOP against several embedded Java systems and, as a reference, with Java on a standard PC. A Java processor is up to 500 times faster than an interpreting JVM on a standard processor for an embedded system. JOP is about seven times faster than the aJ80 Java processor and about 12% faster than the aJ100. Preliminary results using compiled Java for a RISC processor in an FPGA, with a similar resource usage and maximum clock frequency to JOP, showed that native execution of Java bytecodes is faster than compiled Java.

The proposed processor has been used with success to implement several commercial real-time applications. JOP is open-source and all design files are available at `http://www.jopdesign.com/`.

Acknowledgment

The author thanks the anonymous reviewers for their insightful and detailed comments on the first version of the paper. The comments have helped to shape this paper and clarify ambiguous sections in the first version. The author also thanks Andreas Steininger and Peter Puschner for their support during the PhD thesis and comments on this paper.

Bibliography

[1] aJile. aj-100 real-time low power Java processor. preliminary data sheet, 2000.

[2] Altera. Nios soft core embedded processor, ver. 1. data sheet, June 2000.

[3] Altera. Cyclone FPGA Family Data Sheet, ver. 1.2, April 2003.

[4] ARM. Jazelle technology: ARM acceleration technology for the Java platform. white paper, 2004.

[5] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.

[6] Antonio Carlos Beck and Luigi Carro. Low power java processor for embedded applications. In *Proceedings of the 12th IFIP International Conference on Very Large Scale Integration*, December 2003.

[7] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[8] Cyrille Comar, Gary Dismukes, and Franco Gasperoni. Targeting GNAT to the Java virtual machine. In *TRI-Ada '97: Proceedings of the conference on TRI-Ada '97*, pages 149–161, New York, NY, USA, 1997. ACM Press.

[9] DCT. Lightfoot 32-bit Java processor core. data sheet, September 2001.

[10] Martin Delvai, Wolfgang Huber, Peter Puschner, and Andreas Steininger. Processor support for temporal predictability – the spear design example. In *Proceedings of the 15th Euromicro International Conference on Real-Time Systems*, Jul. 2003.

[11] S. Dey, P. Sanchez, D. Panigrahi, L. Chen, C. Taylor, and K. Sekar. Using a soft core in a SOC design: Experiences with picoJava. *IEEE Design and Test of Computers*, 17(3):60–71, July 2000.

[12] EJC. The ejc (embedded java controller) platform. Available at http://www.embedded-web.com/index.html.

[13] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

[14] Jakob Engblom, Andreas Ermedahl, Mikael Södin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, V4(4):437–455, August 2003.

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

[16] David Gregg, James Power, and John Waldron. Benchmarking the java virtual architecture - the specjvm98 benchmark suite. In N. Vijaykrishnan and M. Wolczko, editors, *Java Microarchitectures*, pages 1–18. Kluwer Academic, 2002.

[17] Flavius Gruian, Per Andersson, Krzysztof Kuchcinski, and Martin Schoeberl. Automatic generation of application-specific systems based on a micro-programmed java core. In *Proceedings of the 20th ACM Symposium on Applied Computing, Embedded Systems track*, Santa Fee, New Mexico, March 2005.

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

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

[20] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.

[21] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 3rd ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303, 2002.

[22] Hitachi. Hitachi single-chip microcomputer h8/3297 series. Hardware Manual.

[23] Imsys. Snap, simple network application platform. Available at http://www.imsys.se/.

[24] Imsys. ISAJ reference 2.0, January 2001.

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

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

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

[28] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, V34(3):195–227, November 2006.

[29] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.

[30] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[31] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *RTSS '99: Proceedings of the 20th IEEE Real-Time Systems Symposium*, page 12, Washington, DC, USA, 1999. IEEE Computer Society.

[32] Nazomi. JA 108 product brief. Available at http://www.nazomi.com.

[33] K. Nilsen, L. Carnahan, and M. Ruark. Requirements for real-time extensions for the Java platform. Available at http://www.nist.gov/rt-java/, September 1999.

[34] Kelvin D. Nilsen and Bernt Rygg. Worst-case execution time analysis on modern processors. *SIGPLAN Not.*, 30(11):20–30, 1995.

[35] Anders Nilsson. Compiling java for real-time systems. Licentiate thesis, Dept. of Computer Science, Lund University, May 2004.

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

[37] Matthias Pfeffer. *Ein echtzeitfähiges Java-System für einen mehrfädigen Java-Mikrocontroller*. PhD thesis, University of Augsburg, 2000.

[38] PTSC. Ignite processor brochure, rev 1.0. Available at http://www.ptsc.com.

[39] Peter Puschner. Experiments with WCET-oriented programming and the single-path architecture. In *Proc. 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, Feb. 2005.

[40] Peter Puschner and Alan Burns. A review of worst-case execution-time analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, 2000.

[41] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, page 85, Washington, DC, USA, 2002. IEEE Computer Society.

[42] Peter Puschner and Christian Koza. Calculating the maximum execution time of real-time programs. *Real-Time Syst.*, 1(2):159–176, 1989.

[43] Peter Puschner and Anton Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, Jul. 1997.

[44] Sven Gestegard Robertz and Roger Henriksson. Time-triggered garbage collection: robust and adaptive real-time GC scheduling for embedded systems. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 93–102, New York, NY, USA, 2003. ACM Press.

[45] Martin Schoeberl. Using a Java optimized processor in a real world application. In *Proceedings of the First Workshop on Intelligent Solutions in Embedded Systems (WISES 2003)*, pages 165–176, Austria, Vienna, June 2003.

[46] Martin Schoeberl. Design rationale of a processor architecture for predictable real-time execution of Java programs. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.

[47] Martin Schoeberl. Real-time scheduling on a Java processor. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications (RTCSA 2004)*, Gothenburg, Sweden, August 2004.

[48] Martin Schoeberl. Restrictions of Java for embedded real-time systems. In *Proceedings of the 7th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2004)*, pages 93–100, Vienna, Austria, May 2004.

[49] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[50] Martin Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

[51] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[52] Martin Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006.

[53] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th international workshop on Java technologies for real-time and embedded systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[54] Jose Solorzano. leJOS: Java based os for lego RCX. Available at: http://lejos.sourceforge.net/.

[55] SPEC. The spec jvm98 benchmark suite. Available at http://www.spec.org/, August 1998.

[56] O. Strom, K. Svarstad, and E. J. Aas. On the utilization of Java technology in embedded systems. *Design Automation for Embedded Systems*, 8(1):87–106, 2003.

[57] Sun. Java 2 platform, micro edition (J2ME). Available at: http://java.sun.com/j2me/docs/.

[58] Sun. *picoJava-II Microarchitecture Guide*. Sun Microsystems, March 1999.

[59] Sun. *picoJava-II Programmer's Reference Manual*. Sun Microsystems, March 1999.

[60] S. Tucker Taft. Programming the internet in Ada 95. In *Ada-Europe '96: Proceedings of the 1996 Ada-Europe International Conference on Reliable Software Technologies*, pages 1–16, London, UK, 1996. Springer-Verlag.

[61] Stephan Thesing. *Safe and Precise Worst-Case ExecutionTime Prediction by Abstract Interpretation of Pipeline Models*. PhD thesis, University of Saarland, 2004.

[62] Vulcan. Moon v1.0. data sheet, January 2000.

[63] Vulcan. Moon2 - 32 bit native Java technology-based processor. product folder, 2003.

[64] Andy Wellings. Is Java augmented with the RTSJ a better real-time systems implementation technology than Ada 95? *Ada Lett.*, XXIII(4):16–21, 2003.

[65] Xilinx. Microblaze processor reference guide, edk v6.2 edition. data sheet, December 2003.

[66] Xilinx. Spartan-3 FPGA family: Complete data sheet, ver. 1.2, January 2005.

3 Non-blocking Real-Time Garbage Collection

Trans. on Embedded Computing Sys., 26 pages, accepted 2009, ACM

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

Abstract

A real-time garbage collector has to fulfill two basic properties: ensure that programs with bounded allocation rates do not run out of memory and provide short blocking times. Even for incremental garbage collectors, two major sources of blocking exist, namely root scanning and heap compaction. Finding root nodes of an object graph is an integral part of tracing garbage collectors and cannot be circumvented. Heap compaction is necessary to avoid probably unbounded heap fragmentation, which in turn would lead to unacceptably high memory consumption. In this paper, we propose solutions to both issues.

Thread stacks are local to a thread, and root scanning therefore only needs to be atomic with respect to the thread whose stack is scanned. This fact can be utilized by either blocking only the thread whose stack is scanned, or by delegating the responsibility for root scanning to the application threads. The latter solution eliminates blocking due to root scanning completely. The impact of this solution on the execution time of a garbage collector is shown for two different variants of such a root scanning algorithm.

During heap compaction, objects are copied. Copying is usually performed atomically to avoid interference with application threads, which could render the state of an object inconsistent. Copying of large objects and especially large arrays introduces long blocking times that are unacceptable for real-time systems. In this paper an interruptible copy unit is presented that implements non-blocking object copy. The unit can be interrupted after a single word move.

We evaluate a real-time garbage collector that uses the proposed techniques on a Java processor. With this garbage collector, it is possible to run high priority hard real-time tasks at 10 kHz parallel to the garbage collection task on a 100 MHz system.

3.1 Introduction

Garbage collection (GC) is a feature of modern object oriented languages, such as Java and C#, that increases programmer productivity and program safety. However, dynamic memory management is usually avoided in hard real-time systems. Even the real-time specification for Java (RTSJ) [6], which targets soft real-time systems, defines an additional memory model, with immortal and scoped memory, to avoid GC.

However, the memory model introduced by the RTSJ is unusual to most programmers. It also requires that the Java virtual machine (JVM) checks all assignments to references. If a program does not adhere to the specified model, run-time exceptions are triggered. Arguably, this is a different

level of safety than most Java programmers would expect. Therefore, much research activity is spent to enable GC in real-time systems.

In a system with a concurrent garbage collector, the GC thread and the mutators (i.e., the application threads, which mutate the object graph) have to synchronize their work. Several operations (e.g., barrier code, stack scanning, and object copy) need to be performed atomically. Stack scanning and object copy in atomic sections can introduce considerable blocking times. In this paper, we propose two solutions that eliminate the blocking of these two tasks.

On the software side, we integrated the proposed solutions into a copying GC algorithm. The hardware portions of the presented approaches were implemented in the Java processor JOP [33], which runs at 100 MHz. This platform was used to evaluate the usefulness of our concepts. It is possible to run a 10 kHz high priority task without a single deadline miss with ongoing GC. The maximum task frequency is limited by the scheduler and not by the garbage collector. It has to be noted that the proposed root scanning strategy and copy unit are not JOP specific. The copy unit can also be integrated in a standard RISC processor that executes compiled Java.

This paper is based on prior work on non-blocking root scanning [27] and non-blocking object copy [35]. The evaluation section provides the results for the combination of both concepts. The paper is organized as follows: in the remainder of this section, we discuss the issues to be solved in the areas of root scanning and object copy in a real-time garbage collector. Section 3.2 provides an overview of the related work in these fields. In Section 3.3, our solutions to make root scanning preemptible are presented. A hardware unit to allow non-blocking copying of objects is proposed in Section 3.4. Section 3.5 provides details of our implementation, which is then evaluated in Section 3.6. Section 3.7 concludes the paper and provides an outlook on future work.

3.1.1 Root Scanning

Tracing garbage collectors traverse the object graph to identify the set of reachable objects. The starting point for this tracing is the *root set*, a set of objects which is known to be directly accessible. On the one hand, these are references in global (static in Java) variables, on the other hand these are references that are local to a thread. The latter comprise the references in a thread's runtime stack and thread-local CPU registers. The garbage collector must ensure that its view of the root set is consistent before it can proceed, otherwise objects could be erroneously reclaimed.

For stop-the-world garbage collectors, the consistency of the object graph is trivially ensured. Incremental garbage collectors however require the use of *barriers*, which enforce the consistency of marking and the root set [5, 10, 39, 42]. While barriers are an efficient solution for the global root set, they are considered to be too inefficient to keep the local root sets consistent. Even frequent instructions like storing a reference to a local variable would have to be guarded by such a barrier, which would cause a considerable overhead and make it difficult if not impossible to compute tight bounds for the worst-case execution time (WCET). The usual solution to this problem is to scan the stacks of all threads in a single atomic step and stall the application threads while doing so.[1] The atomicity entails that the garbage collector may not be preempted while it scans a thread's stack, which in turn causes a considerable release jitter even for high-priority threads.

However, the atomicity is only necessary w. r. t. the thread whose stack is scanned, because a thread can only modify its own stack. If the garbage collector scans a thread's stack, the thread must not execute and atomicity has to be enforced. Other mutator threads are allowed to preempt the stack scanning thread. If a thread scans its own stack, it is not necessary to prohibit the preemption of the

[1] The former implies the latter on uniprocessors, but not on multi-processors.

thread – when the thread continues to execute, the stack is still in the same state and the thread can proceed with the scanning without special action. Consequently, preemption latencies due to root scanning can be avoided. With such a strategy, it is also possible to minimize the overhead for root scanning. It can be scheduled in advance such that the local root set is small at the time of scanning.

In this paper, we present two solutions for periodic and sporadic threads that make use of this approach, and evaluate their trade-offs. Furthermore, we show how the worst case time until all threads have scanned their stacks can be computed.

3.1.2 Object Copy

Heap fragmentation is one of the main reasons to avoid dynamic memory management in hard real-time systems and safety critical systems. The worst-case memory consumption within a fragmented heap [41] is too high to be acceptable. A garbage collector that performs heap compaction as part of the collection task eludes this fragmentation issue.

Heap compaction comes at a cost: objects need to be moved in the heap. This object copy consumes processor execution time, memory bandwidth, and needs to be performed atomically. We can accept the first two cost factors as a trade-off for safer real-time programs. However, the blocking time introduced by the atomic copy operation can be in the range of milliseconds on actual systems. This value can be too high for many real-time applications.

In this paper we propose a memory unit for non-blocking object copy. The memory copy is performed independent of the activity in the CPU, similar to a direct memory access (DMA) unit. The copy unit executes at the priority of the GC thread. When a higher priority thread becomes ready, the copy unit is interrupted. The memory unit stores the state of the copy task. The object field and array access is also performed by this memory unit. When a field of an object under copy is accessed by the mutator, the memory unit redirects the access to the correct version of the object: to the original object when the field has not yet been copied or to the destination object when the field has already been copied.

3.2 Related Work

Real-time GC research dates back to the 1970s where collectors for LISP and ML have been developed. Therefore, a vast number of papers on real-time GC have been published. A good introduction to GC techniques can be found in Wilson's survey [40] and in [16].

3.2.1 Root Scanning

The idea of delegating local root scans to the mutator threads was proposed by [12] and [11]. They point out that this allows for more efficient code and reduces the disruptiveness of GC. Mutator threads should check an appropriate flag from time to time and then scan their local root set. However, the authors remain vague on when the mutators should check this flag and do not investigate the effect of various choices. As they aim for efficiency rather than real-time properties, they do not consider a thread model with known periods and deadlines.

[18] coined the term "sliding view" for the independent scanning of local thread states in a reference counting garbage collector. This scheme was later extended to a mark-sweep garbage collector [2]. Again, these works do not consider the implications on the timing of a real-time system.

The approach presented in this paper builds to some degree on an approach by [36]. They propose a thread model which does not support blocking for I/O and where threads cannot retain a local state

across periods. They also propose that the garbage collector runs at the lowest priority, which entails that the stacks of all threads are empty when a GC cycle starts. Consequently, the garbage collector only needs to consider the global root set.

Yuasa introduces a *return barrier* in [43]. In a first step, the garbage collector scans the topmost stack frames of all threads atomically. Then it continues to scan one frame at a time. When a thread returns to a frame that has not yet been scanned, it scans it by itself. Return instructions consequently carry an overhead for the respective check. Furthermore, the proposed policy makes it difficult to compute tight WCET bounds, because it is difficult to predict when a scan by a thread is necessary. A further critical issue is that the topmost frames of *all* threads have to be scanned in a single atomic step. Therefore, the worst-case blocking time increases with the number of threads. An overhead of 2 to 10 percent due to the necessary checks for the return barrier is reported in [43]. Depending on the configuration, 10 to 50 μs were measured as worst-case blocking time on a 200 MHz Pentium Pro processor for two single-threaded benchmarks.

[8] propose a strategy for lowering the overhead and blocking of stack scanning. The mutator thread marks the activated stack frames and the garbage collector scans only those frames which have been activated since the last scan. However, this technique is only useful for the average case. In the worst case, it is still necessary to scan the whole stack atomically.

In the JamaicaVM's garbage collector, the mutator threads are responsible of keeping their root set up to date in "root arrays" [38]. The average overhead for keeping these root arrays up to date is estimated as 11.8%.

3.2.2 Object Copy

The JamaicaVM takes a simple approach to avoid blocking times due to object copying: it avoids moving objects at all [37]. Objects and arrays are split into fix sized blocks and are never moved. This approach trades external fragmentation for internal fragmentation. However, the internal fragmentation can be bounded.

The Metronome garbage collector splits arrays, similar to the JamaicaVM approach, into small chunks called Arraylets [4]. Metronome compacts the heap to avoid fragmentation and the Arraylets reduce blocking time on the copy of large arrays. Both approaches, the JamaicaVM garbage collector and Metronome, have to pay the price of a more complex (and time consuming) array access. The defragmentation algorithm in Metronome evacuates the objects from almost empty pages to nearly full pages [3]. This minimizes the amount of data to be moved, and the overall effort for defragmentation. However, it still requires to atomically move considerable amounts of data.

Another approach to allow interruption of GC copy is to perform field writes to both copies of the object or array [15]. This approach slows down write access, but those are less common than read accesses. The writes to the two copies must be performed atomically to ensure the consistency of the data. An additional pointer is also needed between the two copies of the object. We consider the overhead for establishing the atomicity for the two writes too high for this solution to be practical. [21] propose a garbage collector where the mutator is allowed to modify the original copy of the objects. All writes are recorded in a mutation log and the garbage collector has to apply the writes from this log after updating the pointer(s) to the new object copy.

The clever usage of atomic two-field compare-and-swap (CAS) operations for an incremental object copy is proposed by [25]. During the copy process, an object is expanded to an intermediate wide version and an uninitialized narrow version in tospace. The wide version is protected by CAS operations. However, this solution introduces some overheads to the mutator field access especially during the copy process. In the worst case, the mutator has to expand the object to the wide version on a

field write. [26] explored two more variants of using CAS for consistent object copying, which rely on a probabilistic understanding of time bounds. Furthermore, it is admitted for the original variant that "In a small probability worst-case race scenario, repeated writes to a field in the expanded object may cause the copier to be postponed indefinitely." As a hard real-time system has to guarantee time bounds also in worst-case scenarios, we do not consider these approaches to be suitable for such systems.

[22] propose hardware support, the object-space manager (OSM), for real-time garbage collector on a standard RISC processor. The concurrent garbage collector is based on [5], but the concurrency is of finer grain than the original Baker algorithm as it allows the mutator to continue during the object copy. The OSM redirects field access to the correct location for an object that is currently being copied. [29] extend the OSM to a GC memory module where a local microprocessor performs the GC work. In the paper the performance of standard C++ dynamic memory management is compared against garbage collected C++. The authors conclude that C++ with the hardware supported garbage collection performs comparable with traditional C++.

One argument against hardware support for GC might be that standard processors will never include GC specific instructions. However, Azul Systems has included a read barrier in their RISC based chip-multiprocessor system [9]. The read barrier looks like a standard load instruction, but tests the TLB if a page is a GC-protected page. GC-protected pages contain objects that are already moved. The read barrier instruction is executed after a reference load. If the reference points into a GC-protected page a user-mode trap handler corrects the stale reference to the forwarded reference.

[20] presents a hardware implementation of Baker's read-barrier [5] in an object-based RISC processor. The cost of the read-barrier is between 5 and 50 clock cycles. The resulting minimum mutator utilization (MMU) for a time quantum of 1 ms was measured to be 55%. For a real-time task with a period of 1 kHz the resulting overhead is about a factor of 2. We consider the 50 cycles, even if they are quite low, too expensive for a read-barrier and use the Brooks-style [7] indirection instead.

The solution proposed by Meyer for object-oriented systems also contains a GC coprocessor in the same chip. Close interaction between the RISC pipeline and the GC coprocessor allow the redirection for field access in the correct semi-space with a concurrent object copy. The hardware cost of this feature is given as an additional word for the back-link in every pointer register and every attribute cache line. The only additional runtime cost is on an attribute cache miss. In that case, two instead of one memory accesses resolve the cache miss. It is not explicitly described in the paper when the GC coprocessor performs the object copy. We assume that the memory copy is performed in parallel with the execution of the RISC pipeline. In that case, the GC unit *steals* memory bandwidth from the application thread. Our copy unit, in contrast, respects thread priorities and has no influence on the WCET of hard real-time threads.

The Java processor SHAP [44], with a pipeline and cache architecture based on the architecture of JOP, contains a memory management unit with a hardware garbage collector. That unit redirects field and array access during a copy operation of the GC unit.

The three hardware-assisted GC proposals [22, 20, 44] do not address the influence of the copy hardware on the WCET of the mutator threads. It is known that background DMA complicates WCET analysis. In our proposal, we allow object copy only when the GC thread is running. Therefore, that task is simple to integrate into the schedulability analysis. Scheduling the GC thread at low priority and providing an interruptible (non-blocking) object copy result in 100% utilization for high priority real-time tasks.

3.3 Preemptible Root Scanning

Due to the volatile nature of a thread's stack, the garbage collector and the mutator thread must cooperate for proper scanning. If a thread executes arbitrary code while its stack is scanned, the consistency of the retrieved data cannot be guaranteed. Therefore, a thread is usually suspended during a stack scan. In order to ensure the consistency of the root set, the stack is scanned atomically to avoid preemption of the garbage collector and inhibit the execution of the respective thread.

When the GC thread scans a stack it is not allowed to be preempted by that thread. The runtime stacks of any two threads are however disjoint – otherwise they could not execute independently of one another. Therefore, preemption by any other mutator thread is not an issue. When inhibiting the preemption of the garbage collector only for the thread whose stack is scanned, a thread will only suffer blocking time due to the scanning of its own stack. A high priority thread, which has probably a shallow call tree, will not suffer from the scanning of deeper stacks of more complex tasks. The protection of the scan phase can be achieved by integrating parts of the GC logic with the scheduler. During stack scanning only the corresponding mutator thread is blocked.

We generalize this idea by moving the stack scanning task to the mutator threads. Each thread scans its own stack at the end of its period. In that case mutual exclusion is trivially enforced: the thread performs either mutator work or stack scanning. The garbage collector initializes a GC period as usual. It then sets a flag to signal the threads that they shall scan their stacks at the end of their period. When all threads have acknowledged the stack scan, the garbage collector can scan the static variables and proceed with tracing the object graph. Why the static variables are scanned after the local variables is discussed in Section 3.3.1.

By using such a scheme, it is not necessary to enforce the atomicity of a stack scan. Furthermore, the overhead for a stack scan is low; at the end of each period, the stack is typically small if not even empty. Such a scheme also simplifies exact stack scanning, because stack scanning takes place only at a few predefined instants. Instead of determining the stack layout for every point at which a thread might be preempted, it is only necessary to compute the layout for the end of a period. The required amount of data is reduced considerably as well, which lowers the memory overhead for exact stack scanning.

3.3.1 Consequences

In [27], we proved that delegating the scanning of the thread-local root sets to the mutator threads does not void the correctness of our garbage collector. It has to be assured that no reference can remain undetected by the garbage collector. Such a situation could happen, if a reference migrates from a not-yet-scanned local variable to a local variable that already has been scanned. We could prove that in such a situation, the proposed GC algorithm can compute the root set correctly. Threads can exchange data only through static variables and object fields. An appropriate write barrier can therefore make migrating references visible to the garbage collector. The proof revealed some other interesting issues, which we address in the following.

Write Barrier

Formal reasoning showed that a Yuasa-style snapshot-at-beginning barrier is sufficient to ensure the correctness of the GC, if new objects are allocated gray in terms of Dijkstra's tri-color abstraction [10]. The idea behind this is that a snapshot-at-beginning barrier allows to approximate the history of the object graph if no object is black. On the one hand, overwritten references are marked gray,

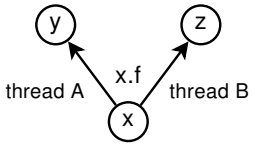


Figure 3.1: Threads may have an inconsistent view of the object graph

i.e., they are visible to the garbage collector. On the other hand, the garbage collector follows the most recent state of the object graph during tracing. Therefore, the whole history of the object graph is visible to the garbage collector. If an object is black, it is not considered by the garbage collector for tracing, and its actual state would remain invisible to the garbage collector.

The usual solution to keep the view of the heap consistent for such garbage collectors is a *double barrier* [1]. It requires that the write barrier pushes both the old and the new value onto the mark stack during root scanning. W. r. t. predictability, a snapshot-at-beginning barrier is superior to a double barrier, because only zero or one references may be pushed onto the mark stack. For a double barrier, zero, one or two references may be pushed. Obviously, the latter has a higher variability in its execution time.

We are aware of the fact that allocating new objects gray is against "common knowledge", especially for a copying garbage collector. However, in the case of our GC algorithm (it is described in detail in Section 3.5.1), the impact of this can be kept considerably lower than for other garbage collectors. The notion of gray objects mainly refers to their status w. r. t. tracing the object graph. With our garbage collector, it is possible to allocate a new object in tospace and to also push it onto the mark-stack (one may think of these objects as being "anthracite"). The copying step is skipped for such objects, while tracing takes place as normal. This leaves us with a trade-off between a double barrier and some additional tracing effort for the garbage collector. The temporal variability of the allocation is slightly increased, because new objects are pushed onto the mark stack only during root scanning. Otherwise, we would not be able to ensure that tracing ever finishes. However, the increase of the temporal variability is small, compared to the overall costs of allocation.

It has to be noted that reading the old value and writing the new value in the write barrier has to be atomic, which is the case in our implementation. When guaranteeing this atomicity is too expensive, the double barrier is an alternative solution. [2]

Memory Model

Figure 3.1 shows a situation, where two threads, A and B, have an inconsistent view of the object graph. While for thread A the field x.f references object y, the same field references object z for thread B. Such a situation is acceptable in the Java memory model [13], but poses problems for a garbage collector, because it would leave either object y or object z unvisited. Proper synchronization of course eliminates such coherence issues, but the authors consider correctness of synchronization to be an unreasonably strong precondition for GC. Flawed synchronization should not cause a failure of the garbage collector.

Inconsistent views of the object graph originate from the fact that threads are allowed to cache data locally. On uniprocessors, cache coherence is not an issue – all threads share the same cache – but thread-local registers may be used to store reference fields. As these registers are scanned for the computation of the local root set of a thread, it is ensured that references cached in registers are

[2] We thank Bertrand Delsart who pointed out this detail during the presentation at the JTRES 2008.

visited as well as references stored in the heap. It is therefore safe to assume that all threads have a consistent view of the object graph.

On multi-processors, cache coherence must be ensured to allow consistent tracing of the object graph. It is beyond the scope of this paper how the required degree of cache coherence can be achieved efficiently.

Static Variables

For our proof, we modeled static variables with an immutable root, which points to a virtual array that contains the static variables. This virtual array can then be handled like any other object and the scanning of static variables becomes part of the marking phase. As marking has to take place after root scanning, static variables have to be scanned after the local root sets.

There is also a more pragmatic reason for this – it is easy to construct an example where scanning static references before scanning local variables breaks the consistency of a garbage collector that uses a snapshot-at-beginning write barrier. Consider the case where during the scanning of static variables a reference is transferred from a local variable to a static variable that already has been scanned. The value of the local variable might be lost until it is scanned, and the new value of the static variable is not visible to the garbage collector. The variable already has been scanned, and the snapshot-at-beginning barrier retains the old value, but does not treat the new one. Therefore, the respective object may erroneously appear unreachable to the garbage collector. The consequence of this is that static variable have to be scanned after the local root sets have been scanned.

3.3.2 Execution Time Bounds

Functional correctness is not the only concern for real-time systems: the effects on the timing behavior of the GC thread also have to be analyzed. We found two solutions to apply the theoretical results: The first solution, which is described in Section 3.3.2, can be applied only to periodic tasks. The second solution can be applied to sporadic tasks as well; it is described in Section 3.3.2. The two solutions also provide a trade-off in terms of timing and memory overheads.

Thread Model

We assume that all threads are either periodic or have at least a known deadline. This is a reasonable assumption for real-time threads: it is impossible to decide whether a task delivers its result on time if no deadline or period is known.[3]

The thread model has five states: CREATED, READY, WAITING, BLOCKED and DEAD. Initially, a thread is in state CREATED. When a thread gets available for execution, it goes to the READY state. When it has finished execution for a period it becomes WAITING. At the start of the next period, it goes to the READY state again. If a thread terminates, it becomes DEAD. Threads are in state BLOCKED while they wait for locks or I/O operations. The time between the instant at which a thread becomes READY until it goes to state WAITING must be bounded – if it is not WAITING when its deadline arrives, it has missed the deadline. Figure 3.2 visualizes the possible state transitions of the thread model.

For the calculation of the execution time bounds, we assume that threads scan their stack when they become WAITING. For periodic tasks in the RTSJ [6], this can be done implicitly when wait-

[3] For threads without a known deadline, the garbage collector can fall back to blocking the thread and scanning the stack itself.

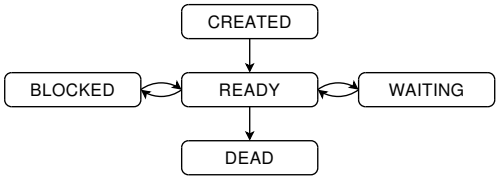


Figure 3.2: Thread Model

ForNextPeriod() is invoked. There is no need to change the application code. If no such method needs to be called by tasks, the scanning can be integrated into the scheduler. In the current version of the RTSJ, sporadic threads do not invoke such a method; their stack is however empty when they do not execute, which in turn makes root scanning trivial. The overhead for stack scanning of course has to be taken into account for calculating the WCET of tasks.

We assume that the GC thread runs at the lowest priority in the system. On the one hand, a garbage collector usually has a long period (and deadline), compared to other real-time tasks. It follows from scheduling theory that it should have a low priority [19].

Solution for Periodic Tasks

For periodic threads, the time between two releases is known and the time between two successive calls of waitForNextPeriod() is bounded. For this solution, the individual tasks push the references of the local root set onto the mark stack of the garbage collector if an appropriate flag is set. The garbage collector must wait until all tasks have acknowledged the scan before it can proceed. In the worst case, a task has become WAITING very early in its period when the garbage collector starts execution and becomes WAITING very late in its next period.

Let R_i be the worst-case response time of a thread τ_i, Q_i its best case response time and T_i its period. The response time of a task is the time between the instant at which a thread becomes READY until it goes to the WAITING state again. $C_{stackscan}$ is the worst case time until all threads have scanned their local root set and the garbage collector may proceed. $C_{stackscan}$ can be computed as follows:

$$C_{stackscan} = \max_{i \geq 0}(T_i - Q_i + R_i) \tag{3.1}$$

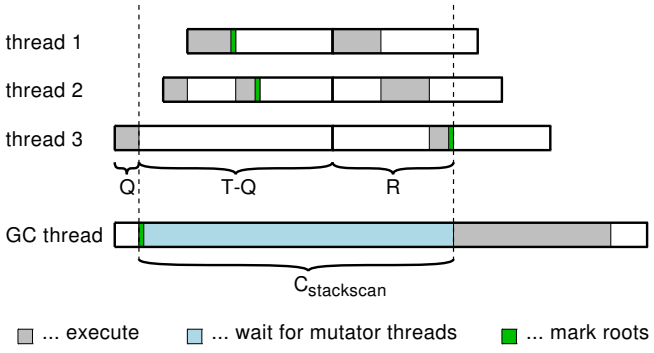
Figure 3.3(a) visualizes the formula above. It shows that the worst case between two completions of a thread is $T - Q + R$. Consequently, this is the longest time the garbage collector must wait for this thread.

To avoid the computation of the best and worst-case response times – especially the former is typically unknown –, this can be simplified to

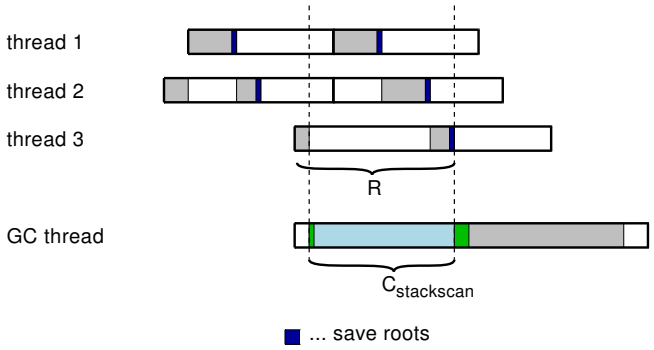
$$C_{stackscan} = 2T_{max} \tag{3.2}$$

Generalized Solution

The considerations for periodic tasks cannot be applied to sporadic tasks in the general case. For sporadic tasks, the minimum inter-arrival time is known, but usually not the maximum inter-arrival time. Therefore, the worst case time until the garbage collector may proceed is potentially unbounded. A similar issue occurs with threads that have a very long period; for such threads, $C_{stackscan}$ for the simple solution may become prohibitively large.



(a) Solution for periodic tasks



(b) Generalized solution

Figure 3.3: Visualization of the WCETs for root scanning

The stack of a thread is only modified if the respective thread executes. Therefore, the garbage collector can reuse data from previous scans and only needs to wait for threads which may have executed since their last scan. These are – apart from the initialization and destruction of threads – the threads which are not in state WAITING.

We adapt the root scanning scheme such that threads save their stack on every call of waitForNextPeriod() to a *root array*. For WAITING threads, the content of the root array from their last scan is used by the garbage collector; for all other threads, the garbage collector waits until they have updated their root array. With this scheme, it is sufficient to take into account the worst-case response time for the execution time of stack scanning.

$$C_{stackscan} = R_{max} \tag{3.3}$$

This time can be further improved: the garbage collector can only execute if no other thread is READY. If threads scan their stacks when becoming BLOCKED, the garbage collector can therefore never encounter threads that have executed since their last stack scan. Trivially this is also the case, if a system does not support blocking operations at all. As the information in the root arrays is always consistent when the garbage collector executes, it is never necessary to wait for any thread to scan its stack and

$$C_{stackscan} = 0 \tag{3.4}$$

Enforcing scanning upon blocking requires more effort than enforcing it upon waiting. It entails that the implementation of wait() needs to be changed accordingly. Depending on the organization of a JVM, this may or may not be possible.

Discussion

As pointed out by [28] and [30], the allocation rate and the size of the heap determine the maximum GC period. The proposed solutions introduce a waiting time for the garbage collector and therefore may make it necessary to increase its period. Such an increase may lead to a situation where it cannot be guaranteed anymore that the garbage collector can cope with the allocation rate of a system.

$C_{stackscan}$ has to be added to the response time of the GC thread; the impact of this delay depends on the thread periods. If the response time of the garbage collector is far greater than the periods of the mutator threads, the relative impact is small. If there is some slack between the maximum and the actual GC period, the effect can probably be hidden.

An advantage of the generalized solution is that $C_{stackscan}$ is considerably smaller than for the simple solution. The downside of the generalized solution is however that a dedicated memory area is needed to save the roots of the individual threads. It is not possible anymore to let the mutator thread push its root set onto the mark stack. To avoid blocking in this scheme, it is necessary to use two memory areas for each thread to allow for double buffering. If the maximum number of roots is unknown, each of these areas occupies as much memory as the stack.

The overhead for completing the root scanning is larger on the garbage collectors side for the generalized solution. This is due to the fact that the garbage collector itself has to push the references onto the mark stack. On the threads' side, the overhead is slightly smaller, because the content of the stack only has to be transferred to the root array, without performing any computations. However, the increased overhead for scanning is most likely far smaller than the time that is spent on waiting for the other threads. It is mandatory to use such a scheme for sporadic tasks; applying it to periodic tasks as well allows to trade time to wait for a root scan with additional memory consumption.

Figure 3.3 compares the worst case scenarios of the solution for periodic tasks and the generalized solution. For the generalized solution, the threads save their stack in a root array at the end of each period. The garbage collector only has to wait for threads that have executed since their last scan. In Figure 3.3(b), thread 3 is BLOCKED when the garbage collector starts execution, but does not scan its stack. Therefore, the garbage collector has to wait for this thread. In the worst case, this waiting time equals the maximum response time.

It is possible to mix root scanning strategies to find an optimal solution. For high frequency threads, jitter is usually very important, and the waiting time of the solution for periodic threads may be negligible. For medium frequency threads, the generalized solution with an impact in the order of one period may be a better trade-off. Low frequency threads are probably less sensitive to jitter and root scanning by the garbage collector may not hurt them. However, it is not possible to propose a generic solution to this problem without knowledge about the properties of the whole system.

3.4 Non-blocking Object Copy

Copying large arrays and objects in a compacting garbage collector attributes to the largest blocking times. To avoid losing updates on the object during copying (write to fields that are already copied), it is usually performed atomically. To avoid those long blocking times in a real-time garbage collector, we propose an interruptible copy unit. The copy unit has two important properties:

- It can be preempted at single word copy boundaries
- The copy process is executed at the GC thread priority

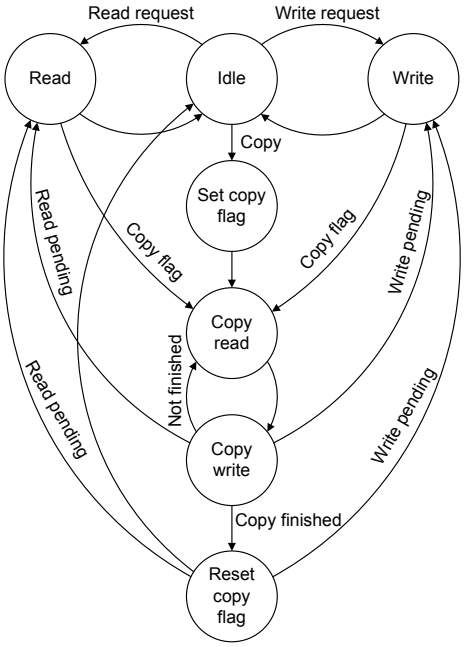


Figure 3.4: Memory controller state machine with background copy

A real-time garbage collector needs to be interruptible by higher priority threads. If the copy task is performed by the hardware, which works autonomously in its own hardware thread, the hardware also needs to be interrupted on a thread switch. Furthermore, the copy task needs to be resumed at the correct time, i.e., when no thread with a priority higher than the GC thread priority is ready.

A simplified solution is to start the copy as a background DMA operation and let the GC thread wait for completion before continuing the GC work. However, this background activity, even when interruptible at word boundaries, changes the WCET of high priority threads. It *steals* memory cycles from those threads. The copy unit starts at idle cycles, but it will still block incoming read or write requests from the real-time threads during the copy. Therefore, it will delay most of the load and store instructions.

Figure 3.4 shows a simplified state diagram of the memory controller that performs the background copy. From the *idle* state either a normal read, normal write or a start of the copy task is performed. The states of the copy task are: start with setting a flag that an object copy is pending, perform the copy (via states *copy read* and *copy write*), and end with the reset of the copy flag. After each write in the copy loop the CPU is checked for an outstanding read or write request. In that case the copy task stops and that request is fulfilled. A stopped copy is resumed from states *read* and *write* if the copy flag is set.

For time predictability we need a complete stop of the copy task on a software thread switch (from the GC thread to an application thread). Two solutions are possible: (a) integrate the control of the copy task into the scheduler, or (b) let the copy unit itself detect a thread switch.

For the first solution the stopping of the copy unit is integrated into the scheduler. On a non-GC thread dispatch, the scheduler has to explicitly stop the copy task. However, this approach needs integration of GC related work into the scheduler, which is not possible in all JVMs.

The second approach is to interrupt the copy task by a normal memory operation (read or write). Interruption can be detected by the memory unit by a pending read or write request. During the object

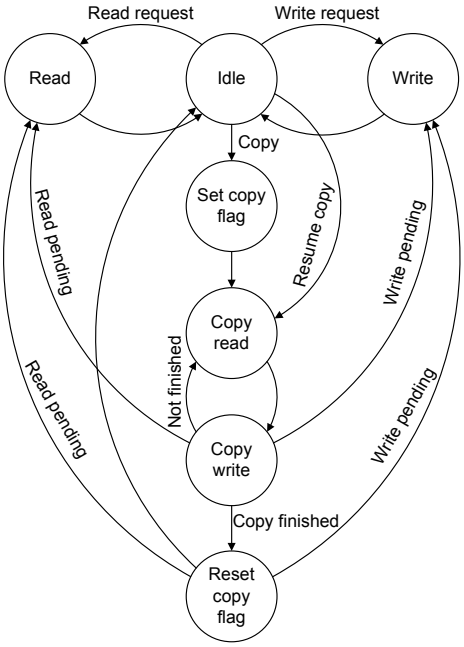


Figure 3.5: Memory controller state machine with interruptible copy

copy the GC thread performs a busy wait on the status of the copy. Therefore, the GC thread does not access main memory at this time. If the memory unit recognizes a read or write request it comes from an application thread that interrupted the GC thread. That request is the signal to stop copying. The state machine for this behavior is depicted in Figure 3.5. As in the former state machine the copy loop can be interrupted by a pending read or write request. The difference is that there is no automatic transition from the *read* and *write* state back to the copy loop. The copy task needs to be explicitly resumed from the processor, as indicated by the transition from *idle* to *copy read*.

The remaining question is how to resume the copy task? Similar to the stopping of the copy unit, two solutions are possible: (a) the scheduler resumes the copy task, or (b) the GC thread performs the resume. The scheduler integration works as follows: When the GC thread is about to be rescheduled, the scheduler has to resume the copy operation as well. This approach is only possible when the scheduler has knowledge about the thread types (mutator or GC thread).

The proposed solution lets the GC thread resume the copy task when getting rescheduled. To perform this function, the GC thread needs to know that it was preempted – an information that is usually not available for a thread. However, the copy unit preserves this information and the state *interrupted* can be queried by the GC thread from the copy unit in the copy loop.

Listing 3.1 shows the copy code in the garbage collector. The GC thread kicks off the copy task with startCopy() and performs a busy wait till the copy task is finished – copyFinished() returns true. Within the loop, the state of the copy state machine is checked with copyInterrupted() and the copy task is resumed if necessary. On a resume, the copy unit just continues to copy the object; it is not a restart of the copy task, as in [14], that can result in starvation of the GC copy. It has to be noted that this busy waiting loop does not consume any memory bandwidth. The code is executed from the instruction cache, stack operations are performed in the stack cache, and all state queries go via an on-chip bus directly to the memory controller. The memory controller can perform the copy at maximum speed during the garbage collector busy wait. At the end of the copying process, the

```
startCopy(src, dst, size);
while (!copyFinished()) {
    if (copyInterrupted()) {
        resumeCopy();
    }
}
synchronized (GC.mutex) {
    updateHandle(handle, dst);
}
```

Listing 3.1: Busy waiting copy loop in the GC thread with a copy resume

reference to the object in the handle is updated atomically. The copy unit still redirects the access to the correct location to avoid any race condition. The redirection is updated at the next object copy (with startCopy()).

A further simplification of the copy unit is possible when the GC thread triggers only single word copies in a tight loop. The copy process is automatically preempted when the GC thread gets preempted. No resume is necessary due to the incremental copy trigger and the polling for the finished copy task can be omitted. The disadvantage of this simplification is the slower copy of the object.

3.5 Implementation

We implemented the proposed non-blocking copy unit in the Java processor JOP [33]. JOP was designed from scratch as a real-time processor [31] to simplify the low-level part of WCET analysis. The main benefit of a Java processor for real-time Java is the possibility to perform WCET analysis at bytecode level [34].

In the following section, the GC algorithm that is part of the JOP runtime environment is briefly described. It has to be noted that the proposed copy unit is independent of the processor platform and also independent from the GC algorithm.

The described GC algorithm is intended for hard real-time systems where allocation rate and object lifetime can be analyzed. As a fallback, when the analysis was wrong, an allocation will be blocked till the GC has freed enough memory. The real-time GCs, which are part of practically all available RTSJ implementations, are usually optimized for soft or mixed real-time applications. These GCs support applications which are not analyzable by (self-)tuning of GC parameters.

3.5.1 The GC Algorithm

The collector for JOP is a concurrent copy collector [30, 36] based on the garbage collectors of [5] and [10]. Baker's expensive read-barrier is avoided by using a write barrier and performing the object copy in the collector thread. Therefore, the collector is concurrent and resembles the collectors presented by [39] and [10]. The collector and the mutator are synchronized by a read and a write barrier. A Brooks-style [7] forwarding directs the access to the object either into tospace or fromspace. Indirection through the forwarding pointer is implemented in hardware and is therefore an atomic operation. On a standard uniprocessor preemption points are common practice for short critical section to synchronized mutator and GC threads. The forwarding pointer is kept in a separate handle area, as proposed by [23]. The separate handle area reduces the space overheads, because only one pointer is needed for both object copies. Furthermore, the indirection pointer does not need

```
private static void putfield_ref(int ref, int value, int index) {
    synchronized (GC.mutex) {
        // snapshot-at-beginning barrier
        int oldVal = Native.getField(ref, index);
        // Is it white?
        if (oldVal!=0 && Native.rdMem(oldVal+GC.OFF_SPACE)!=GC.toSpace) {
            // mark gray
            GC.push(oldVal);
        }
        // assign value
        Native.putField(ref, value, index);
    }
}
```

Listing 3.2: Snapshot-at-beginning write barrier in JOP's JVM

to be copied. The handle also contains other object related data, such as type information, and the mark list. The objects in the heap only contain the fields and no object header. It has to be noted that the size of the handle area needs to be chosen according to the application characteristics.

The second synchronization barrier is a *snapshot-at-beginning* write barrier as proposed by [42]. A snapshot-at-beginning write barrier synchronizes the mutator with the collector on a reference store into a static field, an object field, or an array. The *to be overwritten* field is shaded gray as shown in Listing 3.2. An object is shaded gray by pushing the reference of the object onto the mark stack.[4] Further scanning and copying into tospace – coloring it black – is left to the GC thread. One field in the handle area is used to implement the mark stack as a simple linked list.

This write barrier and appropriate stack scanning allow using expensive write barriers only for reference field access (putfield, putstatic, and aastore in Java bytecode). Local variables and the operand stack need no barrier protection.

Note that field and array access is implemented in hardware on JOP. Only write accesses to reference fields need to be protected by the write barrier, which is implemented in software. During class linking all write operations to reference fields (putfield and putstatic when accessing reference fields) are replaced by JVM internal bytecodes (e.g., putfield_ref) to execute the write barrier code as shown in Listing 3.2.

The methods of class Native are JVM internal methods needed to implement part of the JVM in Java. The methods are replaced by regular or JVM internal bytecodes during class linking. Methods getField(ref, index) and putField(ref, value, index) map to the JVM bytecodes getfield and putfield. The method rdMem() is an example of an internal JVM bytecode and performs a memory read. The null pointer check for putfield_ref is implicitly performed by the hardware implementation of getfield that is executed by Native.getField(). The hardware implementation of getfield triggers an exception interrupt when the reference is null. The implementation of the write barrier shows how a bytecode is substituted by a special version (pufield_ref), but uses in the Java method the hardware implementation of that bytecode (Native.putField()).

In principle, this write barrier could also be implemented in microcode to avoid the expensive invoke of a Java method. However, the interaction with the garbage collector, which is written in Java, is simplified by the Java implementation. As a future optimization we intend to inline the write barrier code.

[4] Although the garbage collector is a copying collector a mark stack is needed to perform the object copy in the GC thread and not by the mutator.

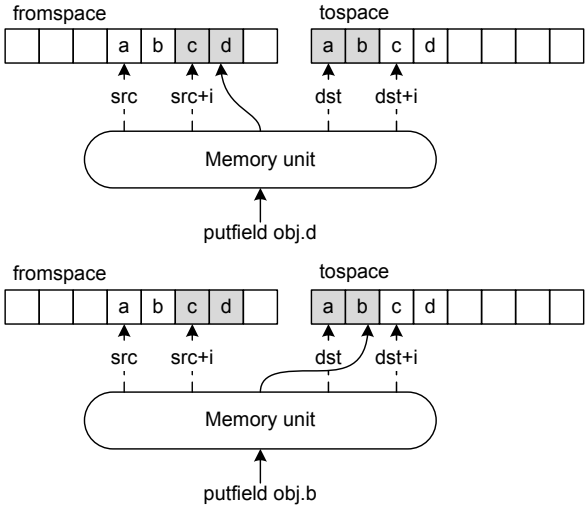


Figure 3.6: Redirection of a putfield operation by the memory unit.

The collector runs in its own thread and the priority is assigned according to the deadline, which equals the period of the GC cycle. As the GC period is usually longer than the mutator task deadlines, the garbage collector runs at the lowest priority. When a high priority task becomes ready, the GC thread will be preempted. Atomic operations of the garbage collector are protected simply by turning the timer interrupt off.[5] Those atomic sections lead to release jitter of the real-time tasks and shall be minimized. It has to be noted that the GC protection with interrupt disabling is not an option for multiprocessor systems.

3.5.2 Root Scanning

The implementation of the new root scanning strategies is straight forward. The logic for stack scanning is inserted into the implementation of waitForNextPeriod(). The garbage collector is modified such that it waits for the application threads to scan their stacks instead of doing it itself. A third change is the gray allocation of new objects during the root scanning phase, which is not necessary for atomic stack scanning. In total, less than 100 lines of Java code are specific to the root scanning strategies proposed in Section 3.3.2.

3.5.3 The Memory Controller

The memory controller in JOP already implements the field and array access in hardware. The hardware implementation of those functions reduces the overhead of the read-barrier (the handle indirection) and speeds up null pointer and bounds checks [32]. This memory controller is extended with a copy function and the redirection of field and array accesses to the correct part of the object.

Figure 3.6 shows an example of the write access to an object that is under copy from address src to address dst. The index i points to the next word that will be moved. The object contains four fields (a, b, c, and d). Gray memory cells show the current locations of the fields. Fields a and b are already in tospace, fields c and d are in the original object in fromspace. The upper figure shows the access

[5] If interrupt handlers are allowed to change the object graph those interrupts also need to be disabled.

to field d that goes to the original object. The lower figure shows the redirection of the access to field b into the tospace copy of the object.

We have implemented the simplified version of the copy unit with the simple interaction with the GC thread. Instead of kicking off the whole copy task once and resuming it after preemption, the copy task is continually triggered for individual words in the garbage collector loop. The following code fragment shows that loop.

```
for (i=0; i<size; i++) {
    Native.memCopy(dst, src, i);
}
```

The method memCopy() is mapped to a JVM internal bytecode and triggers the hardware to perform a single word copy from src to dst at offset i. Note that this loop is not protected by a synchronized block and can be preempted when a high priority thread becomes ready. The copy task is preempted implicitly as well. When the GC thread is running again, it just continues to copy the object.

The advantage of our implementation is a simple state machine in the memory unit and less hardware resource consumption. The disadvantage is the slower copying of the object. A hardware implementation of the copy operation could perform a single word copy in 5 cycles (two cycles to read the word and 3 cycles to write the word) on the actual platform. Copy of a single word with the simplified solution takes 27 cycles: 12 cycles are spent in the JVM internal bytecode and 15 cycles are loop overhead and pushing the arguments for memCopy() onto the operand stack. The maximum blocking time of the copy operation is the execution of the internal bytecode,[6] therefore, 12 clock cycles.

One important feature of the memory controller is the redirection of field and array access to the correct copy of the object. Field and array access are already part of the memory unit [32]. Therefore, the pointer of the access just needs to be compared with the pointer of the object currently copied and the index with the copy pointer. If the index is higher than the copy pointer the access is performed normal – the pointer in the handle indirection points to the old copy until the whole copy is performed. The handle is updated afterwards atomically by the GC thread. If the access goes to a field or array element that is already copied, the access is redirected. To speedup the redirection, the memory unit precalculates the distance between the old copy and the new copy of the object at the start of the copy operation. This offset is simply added at the effective address calculation when a redirection is necessary.

The redirection is performed in the same cycle as the effective address calculation. Therefore, field and array access takes the same time as in the original implementation. The calculation of the offset and the redirection is carefully designed to avoid introduction of a slow critical path in the memory unit that would reduce the maximum operation frequency of the processor.

The hardware resource consumption of the copy unit is moderate. The additional registers, adders, and multiplexors in the memory unit consume 322 additional logic cells (LC). This is about 10% of the complete processor. However, it doubled the size of the memory unit from 301 LC to 623 LC. The memory unit is now almost as large as the execution unit (679 LC).

3.6 Evaluation

For the evaluation we used following hardware setup: JOP implemented in an FPGA and configured for 100 MHz.[7] JOP is configured with 4 KB instruction cache and 1 KB stack cache. The main

[6] Interrupts are only accepted at bytecode boundaries.

[7] The actual synthesis results with medium effort on optimization for the low-cost Altera Cyclone-I FPGA is 97 MHz.

memory consists of 1 MB static RAM with 15 ns access time, resulting in a single word read access in two clock cycles and a single word write access in three clock cycles.

For jitter measurements, we used 6 different tasks, which are similar to the tasks presented in [27] and [35]. We chose to unify these two slightly different experiments, so we can present consistent figures throughout all aspects of our evaluation. Rate monotonic priority ordering is used to determine the tasks' priorities. The task properties are described in the following and subsumed in Table 3.1. The figures presented in Tables 3.2, 3.3, 3.4, and 3.5 were obtained by measuring the maximum release jitter of the highest priority thread during a run of 15 minutes. For the measurements, we slightly modified the periods of the threads. We used prime numbers (e.g., 2003 μs instead of 2000 μs) to avoid a regular phasing of the threads, which could have led to too optimistic results.

The most important thread w. r. t. the measurements is the high-frequency task τ_{hf} with a frequency of 10 kHz. It computes its own release jitter and does nothing else. This task has the highest priority of all tasks and all jitter figures in this section refer to the release jitter of this thread.

Two more threads, τ_p and τ_c, act as producer/consumer pair exchanging arrays. τ_p produces one array every two milliseconds and τ_c consumes the available arrays every 10 milliseconds. A simple list is used to pass the objects from τ_p to τ_c. These threads have the second- and third-highest priorities in the system. $\tau_{p'}$ is a variant of τ_p, which uses a preallocated pool of objects instead of dynamic allocation. It is used to evaluate the behavior of the system if no GC takes place or if GC could not cope with the allocation rates.

τ_s is a thread which occupies the stack such that it is not empty when waitForNextPeriod() is invoked. Consequently, a strategy as proposed by [36] cannot be used if this thread is part of the task set, because the strategy assumes that the threads' stacks are empty at the time of root scanning. It is also the thread with the deepest stack. The period of τ_s is 15 ms.

To record the measurements, we used a logging thread τ_{log} with a period of 25 ms. It prints results every 40$^{\text{th}}$ iteration, i.e., once per second. The artificially short period is necessary to keep the waiting time for the *scan* strategy sufficiently low. The GC thread, τ_{gc}, has a period of 50 ms and is consequently the lowest-priority thread in the system. The GC period was chosen shorter than necessary to force the GC thread to run practically as a background thread. This setting maximizes the interference between the GC thread and the mutator threads.

The careful reader will note that the release jitter in Tables 3.2, 3.3, 3.4, and 3.5 exceeds the period of τ_{hf} for some measurements. In the scheduler we used for this experiment, we chose not to adjust the following release times in these cases. On the one hand, this may lead to queuing up of releases, heavily overloading the system. On the other hand, this allows the threads to "catch up" in the following releases and avoids the release times to drift off. At least for our experiment, the latter behavior is more useful, because a single deadline miss then does not affect the measurement for all following releases.

We evaluated four different root scanning strategies. The strategy labeled *base* in Tables 3.2, 3.3, 3.4, and 3.5 scans the stacks of all threads in one atomic step. The *single* strategy scans one stack at a time atomically. The *scan* and *save* strategies implement the solution for periodic tasks and the generalized solution as described in Section 3.3.2. For the *scan* strategy, tasks push their local root set onto the mark stack at the end of their period. For the *save* strategy, tasks save their stack into root arrays, and the garbage collector pushes the references onto the mark stack.

Tables 3.2 and 3.3 show results for a fixed task set and various arrays sizes. The size of the arrays that τ_p and τ_c exchange is shown in the first column. The task set for arrays of up to 4 KB is $\{\tau_{hf}, \tau_p, \tau_c, \tau_{log}, \tau_{gc}\}$. For larger arrays, it was necessary to use the task set $\{\tau_{hf}, \tau_{p'}, \tau_c, \tau_{log}, \tau_{gc}\}$, i.e., preallocated arrays are used. The garbage collector could not keep up with the allocation rates with dynamic allocation. Still, it tries to garbage collect the preallocated arrays and therefore has an

Thread	Period	Deadline	Priority
τ_{hf}	100 μs	100 μs	6
τ_p	2 ms	2 ms	5
τ_c	10 ms	10 ms	4
τ_s	15 ms	15 ms	3
τ_{log}	25 ms	25 ms	2
τ_{gc}	50 ms	50 ms	1

Table 3.1: Thread properties of the test programs

Array Size	Jitter (μs)			
	base	single	scan	save
256 B	536	136	82	91
512 B	533	130	82	91
1 KB	537	135	84	90
2 KB	535	142	128	134
4 KB	531	244	241	237
8 KB	537	447	445	455
16 KB	856	857	856	866
32 KB	1677	1671	1677	1685
64 KB	3313	3316	3311	3323

Table 3.2: Release jitter with blocking copy, task set $\{\tau_{hf}, \tau_p, \tau_c, \tau_{log}, \tau_{gc}\}$, varying array sizes

effect on the system behavior. While Table 3.2 presents the numbers for a system that copies objects atomically, the results in Table 3.3 were obtained on a system with a copy unit.

The *base* strategy yields a release jitter of around 535 μs, for arrays of up to 8 KB. The jitter due to the root scanning is large enough to hide the jitter that is caused by the atomic copying of the arrays up to this size. The *single* strategy lowers the release jitter to around 135 μs for small arrays. In Table 3.2, the jitter increases linearly with the array size, starting at 2 KB, to up to 3316 μs. The copy unit removes this effect, such that the *single* strategy can achieve a release jitter of around 130 μs for all array sizes. The results for the *scan* and *save* strategies are similar: both lower the jitter to around 85 μs for small arrays. Again, the jitter for larger arrays increases without the copy unit. Table 3.3 shows that with the copy unit low jitter can be achieved also for large arrays. Tables 3.2 and 3.3 show that the copy unit and the new root scanning strategies allow to achieve low release jitter for high frequency threads.

One question to be answered as well is in how far GC affects the release jitter when comparing it to a system without GC. In Tables 3.4 and 3.5, various task sets are compared. The first column displays the labels for the task sets, the following six columns indicate which tasks are part of a specific task set. For the measurements in the column labeled *no GC*, τ_p is replaced with $\tau_{p'}$, which uses a pool of preallocated objects instead of dynamic allocation, but otherwise is equivalent to τ_p. For these measurements, τ_{gc} is not part of the task set, while it is part of the task set for all other measurements. Therefore, the figures in the *no GC* column indicate how a system without GC would behave. The size of the arrays that are passed between τ_p and τ_c is 4 KB for all measurements

Array Size	Jitter (μs)			
	base	single	scan	save
256 B	532	132	73	86
512 B	532	125	73	75
1 KB	528	126	73	75
2 KB	527	125	73	74
4 KB	527	131	73	86
8 KB	526	131	73	86
16 KB	526	126	75	86
32 KB	526	124	72	86
64 KB	525	122	71	74

Table 3.3: Release jitter with copy unit, task set $\{\tau_{hf}, \tau_p, \tau_c, \tau_{log}, \tau_{gc}\}$, varying array sizes

Task Set	Threads					Jitter (μs)				
	τ_{hf}	τ_p	τ_c	τ_s	τ_{log}	no GC	base	single	scan	save
A	✓			✓	✓	76	517	199	78	86
B	✓	✓	✓		✓	70	531	244	241	237
C	✓	✓	✓	✓	✓	84	683	242	230	247

Table 3.4: Release jitter with blocking copy, varying task sets, array size 4 KB

in Tables 3.4 and 3.5. Figure 3.7 shows a graphic representation of the numbers in Table 3.5; the evaluated task sets and strategies are the same.

As in Tables 3.2 and 3.3, the *base* strategy introduces a considerable amount of jitter. Up to 683 μs of release jitter could be observed for task set *C*, which is almost one order of magnitude larger than the jitter without GC. Scanning the stacks of individual threads atomically results in a lower jitter, but the atomic array copying results in up to 244 μs in Table 3.4. This jitter is removed in Table 3.5, resulting in up to 198 μs of jitter for the *single* strategy. Comparing the task sets with and without τ_s confirms that the effect depends on the size of the largest thread stack.

When comparing the system without dynamic memory allocation to the *scan* and *save* strategies, the jitter is increased by less than 20 μs. Up to 75 μs of jitter can be observed without GC, and must therefore be attributed to scheduling and synchronization in the application logic. Future work will have to concentrate on this area to allow the scheduling of real-time threads with periods of less than 100 μs.

Task Set	Threads					Jitter (μs)				
	τ_{hf}	τ_p	τ_c	τ_s	τ_{log}	no GC	base	single	scan	save
A	✓			✓	✓	70	531	196	80	74
B	✓	✓	✓		✓	69	527	131	73	86
C	✓	✓	✓	✓	✓	75	683	198	82	92

Table 3.5: Release jitter with copy unit, varying task sets, array size 4 KB

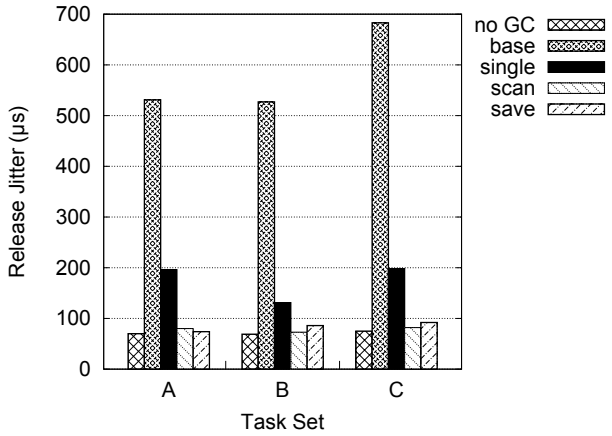


Figure 3.7: Release jitter with copy unit, varying task sets, array size 4 KB

The new strategies *scan* and *save* slightly degrade the scheduling quality. Up to 82 μs of jitter could be observed for the *scan* strategy, up to 92 μs for the *save* strategy. As we made only minimal changes to the scheduler, and no significant atomic sections were introduced, we had expected only a smaller deviation in these results. However, further research will be necessary to find the source of the jitter increase.

3.6.1 Discussion

The results presented in the previous section demonstrate that traditional root scanning techniques are inadequate when considering high frequency real-time threads. Scanning all thread stacks in a single atomic step easily increases the release jitter by one order of magnitude, scanning one thread stack at a time atomically almost triples the jitter. Of course, the effects depend on the actual application, but they tend to become worse with larger applications.

The copying of large arrays is also a key factor in achieving a high scheduling quality. The impact of atomic copying grows linearly with the array sizes; without appropriate measures to reduce this effect, it can easily introduce an unacceptable amount of jitter.

One important result derived from the Tables 3.4 and 3.5 is that it is necessary to use both a non-blocking root scanning strategy and a non-blocking copy mechanism to achieve low jitter. Improved root scanning is futile if the copying of large arrays introduces considerable jitter. Lowering the preemption latency of array copying to the granularity of single words is rendered useless if whole thread stacks are scanned in one atomic step.

The results presented in Figure 3.7 demonstrate that the the GC techniques proposed in this paper can lower the jitter to almost the same level as in a system without GC. For the evaluated system, scheduling is the largest source of jitter. For high frequency tasks, it is therefore necessary to improve the scheduler; the garbage collector is not the limiting factor anymore.

Of course, GC does not come for free. It introduces memory and performance overheads and may therefore make it necessary to use more expensive hardware for a given system. On the other hand, dynamic memory management increases programmer productivity and program safety. The low intrusiveness of the proposed GC mechanisms allows deciding on this trade-off without sacrificing scheduling quality.

3.7 Conclusion and Outlook

We investigated the root scanning phase of GC and could show three important properties: First, atomicity for stack scanning is only necessary w. r. t. the thread whose stack is scanned. Second, atomicity is not required at all if mutator threads scan their own stack. And third, a snapshot-at-beginning write barrier is sufficient to allow complete decoupling of local stack scans.

Furthermore, we provided two approaches how these theoretical properties can be utilized and showed the implications on the execution time of a garbage collector. The first approach can only be applied to periodic tasks and delays the garbage collector by up to two times the longest task period. The second approach is more general and has a smaller impact on the execution time of the garbage collector, but has a higher memory overhead.

In this paper, we also proposed and evaluated a hardware extension to eliminate the blocking time due to atomic copying of large arrays. A copy unit performs the object and array copy and redirects field and array access to the correct version of the object or array. An important feature of the proposed copy unit is scheduling the copy task at GC priority. Therefore, a high priority real-time thread can interrupt the copy task at single word copy boundaries. As the copy task is completely interrupted (no background activity), it does not influence the WCET of real-time threads.

An evaluation of the proposed solutions confirmed the theoretical results. Jitter of high priority threads, which can be attributed to GC, could be reduced considerably. The impact of the new root scanning strategies on the jitter due to scheduling and synchronization however still needs to be analyzed.

Future work will investigate if a tighter coupling of scheduling and root scanning is profitable. Merging the root arrays of the generalized solution with the memory areas for the thread contexts could lower the memory consumption without impairing the performance.

Exact stack scanning has not been handled in this paper. The proposed solutions lower the overhead for exact scanning, but tools to make use of this need to be developed. Furthermore, for hard real-time systems the execution time of the GC task needs to be bounded. We consider WCET analysis of the GC as future work.

The current implementation of our concepts is based on a uniprocessor. We plan to implement them also in the chip-multiprocessor (CMP) version of JOP. The copy unit needs to redirect access from all processors during the copy. Therefore, part of the functionality has to be placed after the memory arbiter. In a CMP setting with a time-sliced arbiter [24] the bandwidth is reserved for the copy task – the copy unit will act just like another CPU. In that case the copy task does not need to be interrupted as proposed for the uniprocessor version. With regard to our root scanning approach, we are confident that the theoretical basis is applicable to CMP systems. Actual implementations may however offer new obstacles as well as new opportunities, especially in the area of cache consistency.

Acknowledgement

We thank the reviewers for the detailed comments, which have helped to clarify the description of the presented ideas. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

Bibliography

[1] Joshua Auerbach, David F. Bacon, Bob Blainey, Perry Cheng, Michael Dawson, Mike Fulton, David Grove, Darren Hart, and Mark Stoodley. Design and implementation of a comprehensive real-time java virtual machine. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 249–258, New York, NY, USA, 2007. ACM.

[2] Hezi Azatchi, Yossi Levanoni, Harel Paz, and Erez Petrank. An on-the-fly mark and sweep garbage collector based on sliding view. In *OOPSLA'03 ACM Conference on Object-Oriented Systems, Languages and Applications*, ACM SIGPLAN Notices, Anaheim, CA, November 2003. ACM Press.

[3] David F. Bacon, Perry Cheng, and V.T. Rajan. Controlling fragmentation and space consumption in the Metronome, a real-time garbage collector for Java. In LCTES [17].

[4] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collecor with low overhead and consistent utilization. In *Conference Record of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, New Orleans, LA, January 2003. ACM Press.

[5] Henry G. Baker. List processing in real-time on a serial computer. *Communications of the ACM*, 21(4):280–94, 1978. Also AI Laboratory Working Paper 139, 1977.

[6] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[7] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In Guy L. Steele, editor, *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming*, pages 256–262, Austin, TX, August 1984. ACM Press.

[8] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *Proceedings of SIGPLAN'98 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, Montreal, June 1998. ACM Press.

[9] Cliff Click, Gil Tene, and Michael Wolf. The pauseless GC algorithm. In Michael Hind and Jan Vitek, editors, *Proceedings of the 1st International Conference on Virtual Execution Environments, VEE 2005, Chicago, IL, USA, June 11-12, 2005*, pages 46–56. ACM, 2005.

[10] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

[11] Damien Doligez and Georges Gonthier. Portable, unobtrusive garbage collection for multiprocessor systems. In *Conference Record of the Twenty-first Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, Portland, OR, January 1994. ACM Press.

[12] Damien Doligez and Xavier Leroy. A concurrent generational garbage collector for a multithreaded implementation of ML. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, ACM SIGPLAN Notices, pages 113–123. ACM Press, January 1993.

[13] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley Professional, Boston, Mass., 2005.

[14] Flavius Gruian and Zoran Salcic. Designing a concurrent hardware garbage collector for small embedded systems. In *Proceedings of Advances in Computer Systems Architecture: 10th Asia-Pacific Conference, ACSAC 2005*, pages 281–294. Springer-Verlag GmbH, October 2005.

[15] Lorenz Huelsbergen and James R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Fourth Annual ACM Symposium on Principles and Practice of Parallel Programming*, volume 28(7) of *ACM SIGPLAN Notices*, pages 73–82, San Diego, CA, May 1993. ACM Press.

[16] Richard E. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. Wiley, Chichester, July 1996. With a chapter on Distributed Garbage Collection by R. Lins.

[17] *ACM SIGPLAN 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'2003)*, San Diego, CA, June 2003. ACM Press.

[18] Yossi Levanoni and Erez Petrank. An on-the-fly reference counting garbage collector for Java. In *OOPSLA'01 ACM Conference on Object-Oriented Systems, Languages and Applications*, volume 36(10) of *ACM SIGPLAN Notices*, Tampa, FL, October 2001. ACM Press.

[19] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[20] Matthias Meyer. A true hardware read barrier. In J. Eliot B. Moss, editor, *ISMM'06 Proceedings of the Fourth International Symposium on Memory Management*, pages 3–16, Ottawa, Canada, June 2006. ACM Press.

[21] Scott M. Nettles and James W. O'Toole. Real-time replication-based garbage collection. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, volume 28(6) of *ACM SIGPLAN Notices*, Carnegie Mellon University, USA, June 1993. ACM Press.

[22] Kelvin D. Nilsen and William J. Schmidt. Cost-effective object-space management for hardware-assisted real-time garbage collection. *Letters on Programming Language and Systems*, 1(4):338–354, December 1992.

[23] S. C. North and John H. Reppy. Concurrent garbage collection on stock hardware. In Gilles Kahn, editor, *Record of the 1987 Conference on Functional Programming and Computer Architecture*, volume 274 of *Lecture Notes in Computer Science*, pages 113–133, Portland, Oregon, September 1987. Springer-Verlag.

[24] Christof Pitter. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, pages 115–122, Santa Clara, USA, September 2008. ACM Press.

[25] Filip Pizlo, Daniel Frampton, Erez Petrank, and Bjarne Steensgard. STOPLESS: A real-time garbage collector for multiprocessors. In Mooly Sagiv, editor, *ISMM'07 Proceedings of the Fifth International Symposium on Memory Management*, pages 159–172, Montréal, Canada, October 2007. ACM Press.

[26] Filip Pizlo, Erez Petrank, and Bjarne Steensgaard. A study of concurrent real-time garbage collectors. In *Proceedings of SIGPLAN 2008 Conference on Programming Languages Design and Implementation*, ACM SIGPLAN Notices, pages 33–44, Tucson, AZ, June 2008. ACM Press.

[27] Wolfgang Puffitsch and Martin Schoeberl. Non-blocking root scanning for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, September 2008.

[28] Sven Gestegøard Robertz and Roger Henriksson. Time-triggered garbage collection — robust and adaptive real-time GC scheduling for embedded systems. In LCTES [17].

[29] William J. Schmidt and Kelvin D. Nilsen. Performance of a hardware-assisted real-time garbage collector. In *ASPLOS-VI: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 76–85, New York, NY, USA, 1994. ACM Press.

[30] Martin Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006.

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

[32] Martin Schoeberl. Architecture for object oriented programming languages. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 57–62, Vienna, Austria, September 2007. ACM Press.

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

[34] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[35] Martin Schoeberl and Wolfgang Puffitsch. Non-blocking object copy for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, September 2008.

[36] Martin Schoeberl and Jan Vitek. Garbage collection for safety critical Java. In *Fifth International WOrkshop on Java Technologies for Real-Time Systems (JTRES)*, pages 85–93, Vienna, Austria, September 2007. ACM Press.

[37] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Compilers, Architectures and Synthesis for Embedded Systems (CASES2000)*, San Jose, November 2000.

[38] Fridtjof Siebert. Constant-time root scanning for deterministic garbage collection. In *Tenth International Conference on Compiler Construction (CC2001)*, Genoa, April 2001.

[39] Guy L. Steele. Multiprocessing compactifying garbage collection. *Communications of the ACM*, 18(9):495–508, September 1975.

[40] Paul R. Wilson. Uniprocessor garbage collection techniques. Technical report, University of Texas, January 1994. Expanded version of the IWMM92 paper.

[41] Paul R. Wilson and Mark S. Johnstone. Truly real-time non-copying garbage collection. In Eliot Moss, Paul R. Wilson, and Benjamin Zorn, editors, *OOPSLA/ECOOP '93 Workshop on Garbage Collection in Object-Oriented Systems*, October 1993.

[42] Taichi Yuasa. Real-time garbage collection on general-purpose machines. *Journal of Systems and Software*, 11(3):181–198, 1990.

[43] Taichi Yuasa. Return barrier. In *Proceedings of the International Lisp Conference 2002*, 2002.

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

4 A Hardware Abstraction Layer in Java

Trans. on Embedded Computing Sys., 42 pages, accepted 2009, ACM

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

Stephan Korsholm
Department of Computer Science
Aalborg University, Denmark
stk@cs.aau.dk

Tomas Kalibera
Department of Computer Science
Purdue University, USA
kalibera@cs.purdue.edu

Anders P. Ravn
Department of Computer Science
Aalborg University, Denmark
apr@cs.aau.dk

Abstract

Embedded systems use specialized hardware devices to interact with their environment, and since they have to be dependable, it is attractive to use a modern, type-safe programming language like Java to develop programs for them. Standard Java, as a platform independent language, delegates access to devices, direct memory access, and interrupt handling to some underlying operating system or kernel, but in the embedded systems domain resources are scarce and a Java virtual machine (JVM) without an underlying middleware is an attractive architecture. The contribution of this paper is a proposal for Java packages with hardware objects and interrupt handlers that interface to such a JVM. We provide implementations of the proposal directly in hardware, as extensions of standard interpreters, and finally with an operating system middleware. The latter solution is mainly seen as a migration path allowing Java programs to coexist with legacy system components. An important aspect of the proposal is that it is compatible with the Real-Time Specification for Java (RTSJ).

4.1 Introduction

When developing software for an embedded system, for instance an instrument, it is necessary to control specialized hardware devices, for instance a heating element or an interferometer mirror. These

devices are typically interfaced to the processor through device registers and may use interrupts to synchronize with the processor. In order to make the programs easier to understand, it is convenient to introduce a *hardware abstraction layer* (HAL), where access to device registers and synchronization through interrupts are hidden from conventional program components. A HAL defines an interface in terms of the constructs of the programming language used to develop the application. Thus, the challenge is to develop an abstraction that gives efficient access to the hardware, while staying within the computational model provided by the programming language.

Our first ideas on a HAL for Java have been published in [29] and [18]. This paper combines the two papers, provides a much wider background of related work, gives two additional experimental implementations, and gives performance measurements that allow an assessment of the efficiency of the implementations. The remainder of this section introduces the concepts of the Java based HAL.

4.1.1 Java for Embedded Systems

Over the nearly 15 years of its existence Java has become a popular programming language for desktop and server applications. The concept of the Java virtual machine (JVM) as the execution platform enables portability of Java applications. The language, its API specification, as well as JVM implementations have matured; Java is today employed in large scale industrial applications. The automatic memory management takes away a burden from the application programmers and together with type safety helps to isolate problems and, to some extent, even run untrusted code. It also enhances security – attacks like stack overflow are not possible. Java integrates threading support and dynamic loading into the language, making these features easily accessible on different platforms. The Java language and JVM specifications are proven by different implementations on different platforms, making it relatively easy to write platform independent Java programs that run on different JVM implementations and underlying OS/hardware. Java has a standard API for a wide range of libraries, the use of which is thus again platform independent. With the ubiquity of Java, it is easy to find qualified programmers which know the language, and there is strong tool support for the whole development process. According to an experimental study [23], Java has lower bug rates and higher productivity rates than C++. Indeed, some of these features come at a price of larger footprint (the virtual machine is a non-trivial piece of code), typically higher memory requirements, and sometimes degraded performance, but this cost is accepted in industry.

Recent real-time Java virtual machines based on the Real-Time Specification for Java (RTSJ) provide controlled and safe memory allocation. Also there are platforms for less critical systems with real-time garbage collectors. Thus, Java is ready to make its way into the embedded systems domain. Mobile phones, PDAs, or set-top boxes run Java Micro Edition, a Java platform with a restricted set of standard Java libraries. Real-time Java has been and is being evaluated as a potential future platform for space avionics both by NASA and ESA space agencies. Some Java features are even more important for embedded than for desktop systems because of missing features of the underlying platform. For instance the RTEMS operating system used by ESA for space missions does not support hardware memory protection even for CPUs that do support it (like LEON3, a CPU for ESA space missions). With Java's type safety hardware protection is not needed to spatially isolate applications. Moreover, RTEMS does not support dynamic libraries, but Java can load classes dynamically.

Many embedded applications require very small platforms, therefore it is interesting to remove as much as possible of an underlying operating system or kernel, where a major part of code is dedicated to handling devices. Furthermore, Java is considered as the future language for safety-critical systems [15]. As certification of safety-critical systems is very expensive, the usual approach is to minimize the code base and supporting tools. Using two languages (e.g., C for programming device

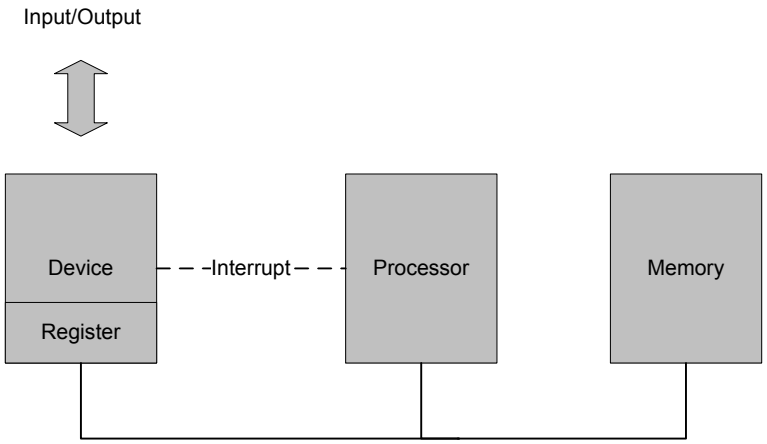


Figure 4.1: The hardware: a bus connects a processor to device registers and memory, and an interrupt bus connects devices to a processor

handling in the kernel and Java for implementing the processing of data) increases the complexity of generating a safety case. A Java only system reduces the complexity of the tool support and therefore the certification effort. Even in less critical systems the same issues will show up as decreased productivity and dependability of the software. Thus it makes sense to investigate a general solution that interfaces Java to the hardware platform; that is the objective of the work presented here.

4.1.2 Hardware Assumptions

The hardware platform is built up along one or more buses – in small systems typically only one – that connect the processor with memory and device controllers. Device controllers have reserved some part of the address space of a bus for its device registers. They are accessible for the processor as well, either through special I/O instructions or by ordinary instructions when the address space is the same as the one for addressing memory, a so called memory mapped I/O solution. In some cases the device controller will have direct memory access (DMA) as well, for instance for high speed transfer of blocks of data. Thus the basic communication paradigm between a controller and the processor is shared memory through the device registers and/or through DMA. With these facilities only, synchronization has to be done by testing and setting flags, which means that the processor has to engage in some form of busy waiting. This is eliminated by extending the system with an interrupt bus, where device controllers can generate a signal that interrupts the normal flow of execution in the processor and direct it to an interrupt handling program. Since communication is through shared data structures, the processor and the controllers need a locking mechanism; therefore interrupts can be enabled or disabled by the processor through an interrupt control unit. The typical hardware organization is summarized in Figure 4.1.

4.1.3 A Computational Model

In order to develop a HAL, the device registers and interrupt facilities must be mapped to programming language constructs, such that their use corresponds to the computational model underlying the language. In the following we give simple device examples which illustrate the solution we propose for doing it for Java.

```
public final class ParallelPort {
    public volatile int data;
    public volatile int control;
}

int inval, outval;
myport = JVMMechanism.getParallelPort();
...
inval = myport.data;
myport.data = outval;
```

Figure 4.2: The parallel port device as a simple Java class

Hardware Objects

Consider a simple parallel input/output (PIO) device controlling a set of input and output pins. The PIO uses two registers: the *data register* and the *control register*. Writing to the data register stores the value into an internal latch that drives the output pins. Reading from the data register returns the value that is present on the input pins. The control register configures the direction for each PIO pin. When bit *n* in the control register is set to 1, pin *n* drives out the value of bit *n* of the data register. A 0 at bit *n* in the control register configures pin *n* as input pin. At reset the port is usually configured as input port – a safe default configuration.

In an object oriented language the most natural way to represent a device is as an object – the *hardware object*. Figure 4.2 shows a class definition, object instantiation, and use of the hardware object for the simple parallel port. An instance of the class ParallelPort is the hardware object that represents the PIO. The reference myport points to the hardware object. To provide this convenient representation of devices as objects, a JVM internal mechanism is needed to access the device registers via object fields and to *create* the device object and receive a reference to it. We elaborate on the idea of hardware objects in Section 4.3.1 and present implementations in Section 4.4.

Interrupts

When we consider an interrupt, it must invoke some program code in a method that handles it. We need to map the interruption of normal execution to some language concept, and here the concept of an asynchronous event is useful. The resulting computational model for the programmer is shown in Figure 4.3. The signals are external, asynchronous events that map to interrupts.

A layered implementation of this model with a kernel close to the hardware and applications on top has been very useful in general purpose programming. Here one may even extend the kernel to manage resources and provide protection mechanisms such that applications are safe from one another, as for instance when implementing trusted interoperable computing platforms [12]. Yet there is a price to pay which may make the solution less suitable for embedded systems: adding new device drivers is an error-prone activity [7], and protection mechanisms impose a heavy overhead on context switching when accessing devices.

The alternative we propose is to use Java directly since it already supports multithreading and use methods in the special InterruptHandler objects to handle interrupts. The idea is illustrated in

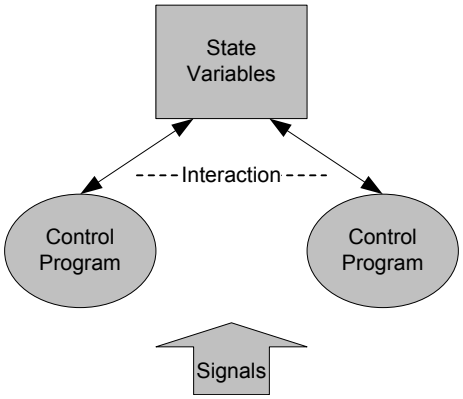


Figure 4.3: Computational model: several threads of execution communicate via shared state variables and receive signals.

Figure 4.4, and the details, including synchronization and interaction with the interrupt control, are elaborated in Section 4.3.2. Implementations are found in Section 4.4.

4.1.4 Mapping Between Java and the Hardware

The proposed interfacing from hardware to Java does not require language extensions. The Java concepts of packages, classes and synchronized objects turn out to be powerful enough to formulate the desired abstractions. The mapping is done at the level of the JVM. The JVM already provides typical OS functions handling:

- Address space and memory management
- Thread management
- Inter-process communication

These parts need to be modified so they cater for interfaces to the hardware.

Yet, the architectures of JVMs for embedded systems are more diverse than on desktop or server systems. Figure 4.5 shows variations of Java implementations in embedded systems and an example of the control flow for a web server application. The standard approach with a JVM running on top of an operating system (OS) is shown in sub-figure (a).

A JVM without an OS is shown in sub-figure (b). This solution is often called *running on the bare metal*. The JVM acts as the OS and provides thread scheduling and low-level access to the hardware. In this case the network stack can be written entirely in Java. JNode[1] is an approach to implement the OS entirely in Java. This solution has become popular even in server applications.[2]

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

Figure 4.5 shows also the data and control flow from the application down to the hardware. The example consists of a web server and an Internet connection via Ethernet. In case (a) the application web server talks with java.net in the Java library. The flow goes down via a native interface to the

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

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

```
public class RS232ReceiveInterruptHandler extends InterruptHandler {
    private RS232 rs232;
    private InterruptControl interruptControl;

    private byte UartRxBuffer[];
    private short UartRxWrPtr;

    ...

    protected void handle() {

        synchronized(this) {
            UartRxBuffer[UartRxWrPtr++] = rs232.P0_UART_RX_TX_REG;
            if (UartRxWrPtr >= UartRxBuffer.length) UartRxWrPtr = 0;
        }
        rs232.P0_CLEAR_RX_INT_REG = 0;
        interruptControl.RESET_INT_PENDING_REG = RS232.CLR_UART_RX_INT_PENDING;
    }
}
```

Figure 4.4: An example interrupt handler for an RS232 interface. On an interrupt the method handle() is invoked. The private objects rs232 and interruptControl are hardware objects that represent the device registers and the interrupt control unit.

TCP/IP implementation and the link layer device driver within the OS (usually written in C). The device driver talks with the Ethernet chip. In (b) the OS layer is omitted: the TCP/IP layer and the link layer device driver are now part of the Java library. In (c) the JVM is part of the hardware layer, and direct access from the link layer driver to the Ethernet hardware is mandatory.

With our proposed HAL, as shown in Figure 4.6, the native interface within the JVM in (a) and (b) disappears. Note how the network stack moves up from the OS layer to the Java library in example (a). All three versions show a pure Java implementation of the whole network stack. The Java code is the same for all three solutions. Version (b) and (c) benefit from hardware objects and interrupt handlers in Java as access to the Ethernet device is required from Java source code. In Section 4.5 we show a simple web server application implemented completely in Java as evaluation of our approach.

4.1.5 Contributions

The key contribution of this paper is a proposal for a Java HAL that can run on the bare metal while still being *safe*. This idea is investigated in quite a number of places which are discussed in the related work section where we comment on our initial ideas as well. In summary, the proposal gives an interface to hardware that has the following benefits:

Object-oriented An object representing a device is the most natural integration into an object oriented language, and a method invocation to a synchronized object is a direct representation of an interrupt.

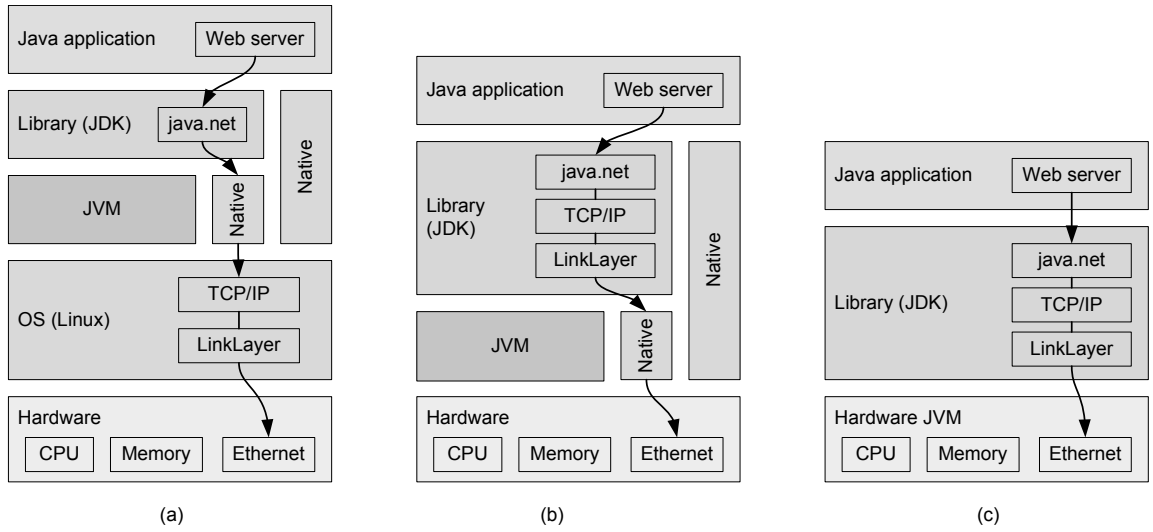


Figure 4.5: Configurations for an embedded JVM: (a) standard layers for Java with an operating system – equivalent to desktop configurations, (b) a JVM on the bare metal, and (c) a JVM as a Java processor

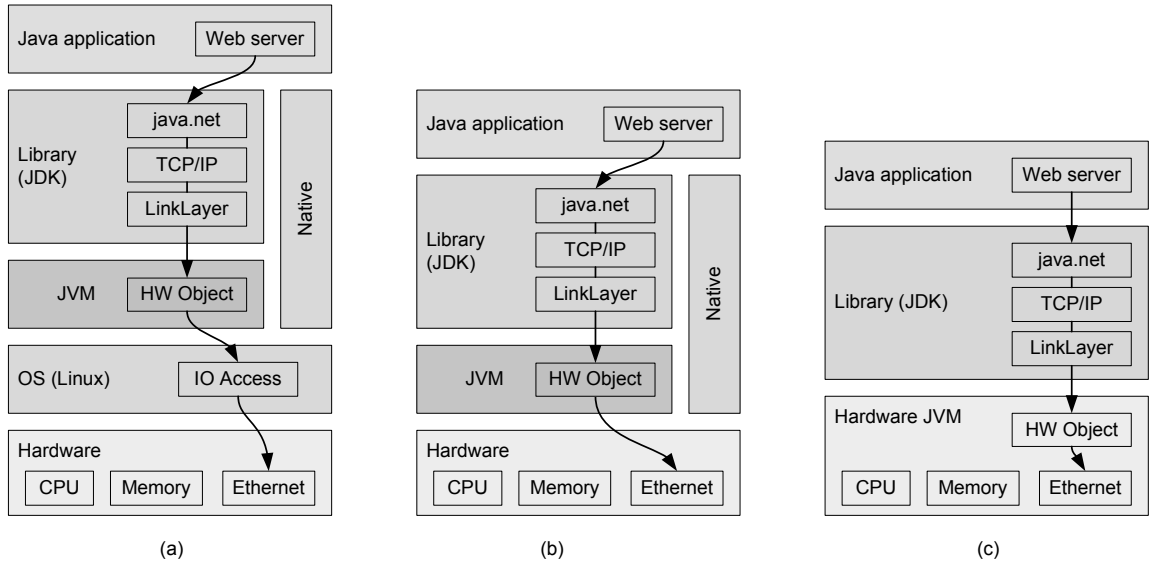


Figure 4.6: Configurations for an embedded JVM with hardware objects and interrupt handlers: (a) standard layers for Java with an operating system – equivalent to desktop configurations, (b) a JVM on the bare metal, and (c) a JVM as a Java processor

Safe The safety of Java is not compromised. Hardware objects map object fields to device registers. With a correct class that represents the device, access to it is safe. Hardware objects can be created only by a factory residing in a special package.

Generic The definition of a hardware object and an interrupt handler is independent of the JVM. Therefore, a common standard for different platforms can be defined.

Efficient Device register access is performed by single bytecodes getfield and putfield. We avoid expensive native calls. The handlers are first level handlers; there is no delay through event queues.

The proposed Java HAL would not be useful if it had to be modified for each particular kind of JVM; thus a second contribution of this paper is a number of prototype implementations illustrating the architectures presented in Figure 4.6: implementations in Kaffe [36] and OVM [2] represent the architecture with an OS (sub-figure (a)), the implementation in SimpleRTJ [25] represents the bare metal solution (sub-figure (b)), and the implementation in JOP [28] represents the Java processor solution (sub-figure (c)).

Finally, we must not forget the claim for efficiency, and therefore the paper ends with some performance measurements that indicate that the HAL layer is generally as efficient as native calls to C code external to the JVM.

4.2 Related Work

Already in the 1970s it was recognized that an operating system might not be the optimal solution for special purpose applications. Device access was integrated into high level programming languages like Concurrent Pascal [13, 24] and Modula (Modula-2) [37, 38] along with a number of similar languages, e.g., UCSD Pascal. They were meant to eliminate the need for operating systems and were successfully used in a variety of applications. The programming language Ada, which has been dominant in defence and space applications till this day, may be seen as a continuation of these developments. The advent of inexpensive microprocessors, from the mid 1980s and on, lead to a regression to assembly and C programming. The hardware platforms were small with limited resources and the developers were mostly electronic engineers, who viewed them as electronic controllers. Program structure was not considered a major issue in development. Nevertheless, the microcomputer has grown, and is now far more powerful than the minicomputer that it replaced. With powerful processors and an abundance of memory, the ambitions for the functionality of embedded systems grow, and programming becomes a major issue because it may turn out to be the bottleneck in development. Consequently, there is a renewed interest in this line of research.

An excellent overview of historical solutions to access hardware devices from and implement interrupt handlers in high-level languages, including C, is presented in Chapter 15 of [5]. The solution to device register access in Modula-1 (Ch. 15.3) is very much like C; however the constructs are safer because they are encapsulated in modules. Interrupt handlers are represented by threads that block to wait for the interrupt. In Ada (Ch 15.4) the representation of individual fields in registers can be described precisely by *representation* classes, while the corresponding structure is bound to a location using the Address attribute. An interrupt is represented in the current version of Ada by a protected procedure, although initially represented (Ada 83) by task entry calls.

The main ideas in having device objects are thus found in the earlier safe languages, and our contribution is to align them with a Java model, and in particular, as discussed in Section 4.4, imple-

mentation in a JVM. From the Ada experience we learn that direct handling of interrupts is a desired feature.

4.2.1 The Real-Time Specification for Java

The Real-Time Specification for Java (RTSJ) [4] defines a JVM extension which allows better timeliness control compared to a standard JVM. The core features are: fixed priority scheduling, monitors which prevent priority inversion, scoped memory for objects with limited lifetime, immortal memory for objects that are never finalized, and asynchronous events with CPU time consumption control.

The RTSJ also defines an API for direct access to physical memory, including hardware registers. Essentially one uses RawMemoryAccess at the level of primitive data types. Although the solution is efficient, this representation of physical memory is not object oriented, and there are some safety issues: When one raw memory area represents an address range where several devices are mapped to, there is no protection between them. Yet, a type safe layer with support for representing individual registers can be implemented on top of the RTSJ API.

The RTSJ specification suggests that asynchronous events are used for interrupt handling. Yet, it neither specifies an API for interrupt control nor semantics of the handlers. Any interrupt handling application thus relies on some proprietary API and proprietary event handler semantics. Second level interrupt handling can be implemented within the RTSJ with an AsyncEvent that is bound to a *happening*. The happening is a string constant that represents an interrupt, but the meaning is implementation dependent. An AsyncEventHandler or BoundAsyncEventHandler can be added as handler for the event. Also an AsyncEventHandler can be added via a POSIXSignalHandler to handle POSIX signals. An interrupt handler, written in C, can then use one of the two available POSIX user signals.

RTSJ offers facilities very much in line with Modula or Ada for encapsulating memory-mapped device registers. However, we are not aware of any RTSJ implementation that implements RawMemoryAccess and AsyncEvent with support for low-level device access and interrupt handling. Our solution could be used as specification of such an extension. It would still leave the first level interrupt handling hidden in an implementation; therefore an interesting idea is to define and implement a two-level scheduler for the RTSJ. It should provide the first level interrupt handling for asynchronous events bound to interrupts and delegate other asynchronous events to an underlying second level scheduler, which could be the standard fixed priority preemptive scheduler. This would be a fully RTSJ compliant implementation of our proposal.

4.2.2 Hardware Interface in JVMs

The aJile Java processor [1] uses native functions to access devices. Interrupts are handled by registering a handler for an interrupt source (e.g., a GPIO pin). Systronix suggests[3] to keep the handler short, as it runs with interrupts disabled, and delegate the real handling to a thread. The thread waits on an object with ceiling priority set to the interrupt priority. The handler just notifies the waiting thread through this monitor. When the thread is unblocked and holds the monitor, effectively all interrupts are disabled.

Komodo [20] is a multithreaded Java processor targeting real-time systems. On top of the multiprocessing pipeline the concept of interrupt service threads is implemented. For each interrupt one thread slot is reserved for the interrupt service thread. It is unblocked by the signaling unit when an interrupt occurs. A dedicated thread slot on a fine-grain multithreading processor results in a very

[3] A template can be found at `http://practicalembeddedjava.com/tutorials/aJileISR.html`

short latency for the interrupt service routine. No thread state needs to be saved. However, this comes at the cost to store the complete state for the interrupt service thread in the hardware. In the case of Komodo, the state consists of an instruction window and the on-chip stack memory. Devices are represented by Komodo specific I/O classes.

Muvium [6] is an ahead-of-time compiling JVM solution for very resource constrained microcontrollers (Microchip PIC). Muvium uses an Abstract Peripheral Toolkit (APT) to represent devices. APT is based on an event driven model for interaction with the external world. Device interrupts and periodic activations are represented by events. Internally, events are mapped to threads with priority dispatched by a preemptive scheduler. APT contains a large collection of classes to represent devices common in embedded systems.

In summary, access to device registers is handled in both aJile, Komodo, and Muvium by abstracting them into library classes with access methods. This leaves the implementation to the particular JVM and does not give the option of programming them at the Java level. It means that extension with new devices involve programming at different levels, which we aim to avoid. Interrupt handling in aJile is essentially first level, but with the twist that it may be interpreted as RTSJ event handling, although the firing mechanism is atypical. Our mechanism would free this binding and allow other forms of programmed notification, or even leaving out notification altogether. Muvium follows the line of RTSJ and has a hidden first level interrupt handling. Komodo has a solution with first level handling through a full context switch; this is very close to the solution advocated in Modula 1, but it has in general a larger overhead than we would want to incur.

4.2.3 Java Operating Systems

The JX Operating System [8] is a microkernel system written mostly in Java. The system consists of components which run in *domains*, each domain having its own garbage collector, threads, and a scheduler. There is one global preemptive scheduler that schedules the domain schedulers which can be both preemptive and non-preemptive. Inter-domain communication is only possible through communication channels exported by services. Low level access to the physical memory, memory mapped device registers, and I/O ports are provided by the core ("zero") domain services, implemented in C. At the Java level ports and memory areas are represented by objects, and registers are methods of these objects. Memory is read and written by access methods of Memory objects. Higher layers of Java interfaces provide type safe access to the registers; the low level access is not type safe.

Interrupt handlers in JX are written in Java and are run through portals – they can reside in any domain. Interrupt handlers cannot interrupt the garbage collector (the GC disables interrupts), run with CPU interrupts disabled, must not block, and can only allocate a restricted amount of memory from a reserved per domain heap. Execution time of interrupt handlers can be monitored: on a deadline violation the handler is aborted and the interrupt source disabled. The first level handlers can unblock a waiting second level thread either directly or via setting a state of a AtomicVariable synchronization primitive.

The Java New Operating System Design Effort (JNode[4]) [22] is an OS written in Java where the JVM serves as the OS. Drivers are written entirely in Java. Device access is performed via native function calls. A first level interrupt handler, written in assembler, unblocks a Java interrupt thread. From this thread the device driver level interrupt handler is invoked with interrupts disabled. Some device drivers implement a synchronized handleInterrupt(int irq) and use the driver object to signal the

[4] http://jnode.org/

upper layer with notifyAll(). During garbage collection all threads are stopped including the interrupt threads.

The Squawk VM [34], now available open-source,[5] is a platform for wireless sensors. Squawk is mostly written in Java and runs without an OS. Device drivers are written in Java and use a form of peek and poke interface to access the device registers. Interrupt handling is supported by a device driver thread that waits for an event from the JVM. The first level handler, written in assembler, disables the interrupt and notifies the JVM. On a rescheduling point the JVM resumes the device driver thread. It has to re-enable the interrupt. The interrupt latency depends on the rescheduling point and on the activity of the garbage collector. For a single device driver thread an average case latency of 0.1 ms is reported. For a realistic workload with an active garbage collector a worst-case latency of 13 ms has been observed.

Our proposed constructs should be able to support the Java operating systems. For JX we observe that the concepts are very similar for interrupt handling, and actually for device registers as well. A difference is that we make device objects distinct from memory objects which should give better possibilities for porting to architectures with separate I/O-buses. JNode is more traditional and hides first level interrupt handling and device accesses in the JVM, which may be less portable than our implementation. The Squawk solution has to have a very small footprint, but on the other hand it can probably rely on having few devices. Device objects would be at least as efficient as the peeks and pokes, and interrupt routines may eliminate the need for multithreading for simple systems, e.g., with cyclic executives. Overall, we conclude that our proposed constructs will make implementation of a Java OS more efficient and perhaps more portable.

4.2.4 TinyOS and Singularity

TinyOS [16] is an operating system designed for low-power, wireless sensor networks. TinyOS is not a a traditional OS, but provides a framework of components that are linked with the application code. The component-based programming model is supported by nesC [10], a dialect of C. TinyOS components provide following abstractions: *commands* represent requests for a service of a component; *events* signal the completion of a service; and *tasks* are functions executed non-preemptive by the TinyOS scheduler. Events also represent interrupts and preempt tasks. An event handler may post a task for further processing, which is similar to a 2nd level interrupt handler.

I/O devices are encapsulated in components and the standard distribution of TinyOS includes a rich set of standard I/O devices. A Hardware Presentation Layer (HPL) abstracts the platform specific access to the hardware (either memory or port mapped). Our proposed HAL is similar to the HPL, but represents the I/O devices as Java objects. A further abstractions into I/O components can be built above our presented Java HAL.

Singularity [17] is a research OS based on a runtime managed language (an extension of C#) to build a software platform with the main goal to be dependable. A small HAL (IoPorts, IoDma, IoIrq, and IoMemory) provides access to PC hardware. C# style attributes (similar to Java annotations) on fields are used to define the mapping of class fields to I/O ports and memory addresses. The Singularity OS clearly uses device objects and interrupt handlers, thus demonstrating that the ideas presented here transfer to a language like C#.

[5] https://squawk.dev.java.net/

```
public abstract class HardwareObject {
    HardwareObject() {};
}
```

Figure 4.7: The marker class for hardware objects

4.2.5 Summary

In our analysis of related work we see that our contribution is a selection, adaptation, refinement, and implementation of ideas from earlier languages and platforms for Java. A crucial point, where we have spent much time, is to have a clear interface between the Java layer and the JVM. Here we have used the lessons from the Java OS and the JVM interfaces. Finally, it has been a concern to be consistent with the RTSJ because this standard and adaptations of it are the instruments for developing embedded real-time software in Java.

4.3 The Hardware Abstraction Layer

In the following section the hardware abstraction layer for Java is defined. Low-level access to devices is performed via hardware objects. Synchronization with a device can be performed with interrupt handlers implemented in Java. Finally, portability of hardware objects, interrupt handlers, and device drivers is supported by a generic configuration mechanism.

4.3.1 Device Access

Hardware objects map object fields to device registers. Therefore, field access with bytecodes putfield and getfield accesses device registers. With a correct class that represents a device, access to it is safe – it is not possible to read or write to an arbitrary memory address. A memory area (e.g., a video frame buffer) represented by an array is protected by Java's array bounds check.

In a C based system the access to I/O devices can either be represented by a C struct (similar to the class shown in Figure 4.2) for memory mapped I/O devices or needs to be accessed by function calls on systems with a separate I/O address space. With the hardware object abstraction in Java the JVM can represent an I/O device as a class independent of the underlying low-level I/O mechanism. Furthermore, the strong typing of Java avoids hard to find programming errors due to wrong pointer casts or wrong pointer arithmetic.

All hardware classes have to extend the abstract class HardwareObject (see Figure 4.7). This empty class serves as type marker. Some implementations use it to distinguish between plain objects and hardware objects for the field access. The package visible only constructor disallows creation of hardware objects by the application code that resides in a different package. Figure 4.8 shows a class representing a serial port with a status register and a data register. The status register contains flags for receive register full and transmit register empty; the data register is the receive and transmit buffer. Additionally, we define device specific constants (bit masks for the status register) in the class for the serial port. All fields represent device registers that can change due to activity of the hardware device. Therefore, they must be declared volatile.

In this example we have included some convenience methods to access the hardware object. However, for a clear separation of concerns, the hardware object represents only the device state (the registers). We do not add instance fields to represent additional state, i.e., mixing device registers

```
public final class SerialPort extends HardwareObject {

    public static final int MASK_TDRE = 1;
    public static final int MASK_RDRF = 2;

    public volatile int status;
    public volatile int data;

    public void init(int baudRate) {...}
    public boolean rxFull() {...}
    public boolean txEmpty() {...}
}
```

Figure 4.8: A serial port class with device methods

with heap elements. We cannot implement a complete device driver within a hardware object; instead a complete device driver owns a number of private hardware objects along with data structures for buffering, and it defines interrupt handlers and methods for accessing its state from application processes. For device specific operations, such as initialization of the device, methods in hardware objects are useful.

Usually each device is represented by exactly one hardware object (see Section 4.3.3). However, there might be use cases where this restriction should be relaxed. Consider a device where some registers should be accessed by system code only and some other by application code. In JOP there is such a device: a system device that contains a 1 MHz counter, a corresponding timer interrupt, and a watchdog port. The timer interrupt is programmed relative to the counter and used by the real-time scheduler – a JVM internal service. However, access to the counter can be useful for the application code. Access to the watchdog register is required from the application level. The watchdog is used for a sign-of-life from the application. If not triggered every second the complete system is restarted. For this example it is useful to represent one hardware device by two *different* classes – one for system code and one for application code. We can protect system registers by private fields in the hardware object for the application. Figure 4.9 shows the two class definitions that represent the same hardware device for system and application code respectively. Note how we changed the access to the timer interrupt register to private for the application hardware object.

Another option, shown in class AppGetterSetter, is to declare all fields private for the application object and use setter and getter methods. They add an abstraction on top of hardware objects and use the hardware object to implement their functionality. Thus we still do not need to invoke native functions.

Use of hardware objects is straightforward. After obtaining a reference to the object all that has to be done (or can be done) is to read from and write to the object fields. Figure 4.10 shows an example of client code. The example is a *Hello World* program using low-level access to a serial port via a hardware object. Creation of hardware objects is more complex and described in Section 4.3.3. Furthermore, it is JVM specific and Section 4.4 describes implementations in four different JVMs.

For devices that use DMA (e.g., video frame buffer, disk, and network I/O buffers) we map that memory area to Java arrays. Arrays in Java provide access to raw memory in an elegant way: the access is simple and safe due to the array bounds checking done by the JVM. Hardware arrays can be *used* by the JVM to *implement* higher-level abstractions from the RTSJ such as RawMemory or

```
public final class SysCounter extends HardwareObject {

    public volatile int counter;
    public volatile int timer;
    public volatile int wd;
}

public final class AppCounter extends HardwareObject {

    public volatile int counter;
    private volatile int timer;
    public volatile int wd;
}

public final class AppGetterSetter extends HardwareObject {

    private volatile int counter;
    private volatile int timer;
    private volatile int wd;

    public int getCounter() {
        return counter;
    }

    public void setWd(boolean val) {
        wd = val ? 1 : 0;
    }
}
```

Figure 4.9: System and application classes, one with visibility protection and one with setter and getter methods, for a single hardware device

```
import com.jopdesign.io.*;

public class Example {

    public static void main(String[] args) {

        BaseBoard fact = BaseBoard.getBaseFactory();
        SerialPort sp = fact.getSerialPort();

        String hello = "Hello World!";

        for (int i=0; i<hello.length(); ++i) {
            // busy wait on transmit buffer empty
            while ((sp.status & SerialPort.MASK_TDRE) == 0)
                ;
            // write a character
            sp.data = hello.charAt(i);
        }
    }
}
```

Figure 4.10: A 'Hello World' example with low-level device access via a hardware object

scoped memory.

Interaction between the garbage collector (GC) and hardware objects needs to be crafted into the JVM. We do not want to *collect* hardware objects. The hardware object should not be scanned for references.[6] This is permissible when only primitive types are used in the class definition for hardware objects – the GC scans only reference fields. To avoid collecting hardware objects, we *mark* the object to be skipped by the GC. The type inheritance from HardwareObject can be used as the marker.

4.3.2 Interrupt Handling

An interrupt service routine (ISR) can be integrated with Java in two different ways: as a first level *handler* or a second level *event* handler.

ISR handler The interrupt is a method call initiated by the device. Usually this abstraction is supported in hardware by the processor and called a first level handler.

ISR event The interrupt is represented by an asynchronous notification directed to a thread that is unblocked from a wait-state. This is also called deferred interrupt handling.

An overview of the dispatching properties of the two approaches is given in Table 4.1. The ISR handler approach needs only two context switches and the priority is set by the hardware. With the ISR event approach, handlers are scheduled at software priorities. The initial first level handler,

[6] If a hardware coprocessor, represented by a hardware object, itself manipulates an object on the heap and holds the only reference to that object it has to be scanned by the GC.

ISR	Context switches	Priorities
Handler	2	Hardware
Event	3–4	Software

Table 4.1: Dispatching properties of different ISR strategies

running at hardware priority, fires the event for the event handler. Also the first level handler will notify the scheduler. In the best case three context switches are necessary: one to the first level handler, one to the ISR event handler, and one back to the interrupted thread. If the ISR handler has a lower priority than the interrupted thread, an additional context switch from the first level handler back to the interrupted thread is necessary.

Another possibility is to represent an interrupt as a thread that is released by the interrupt. Direct support by the hardware (e.g., the interrupt service thread in Komodo [20]) gives fast interrupt response times. However, standard processors support only the handler model directly.

Direct handling of interrupts in Java requires the JVM to be prepared to be interrupted. In an interpreting JVM an initial handler will reenter the JVM to execute the Java handler. A compiling JVM or a Java processor can directly invoke a Java method as response to the interrupt. A compiled Java method can be registered directly in the ISR dispatch table.

If an internal scheduler is used (also called *green threads*) the JVM will need some refactoring in order to support asynchronous method invocation. Usually JVMs control the rescheduling at the JVM level to provide a lightweight protection of JVM internal data structures. These preemption points are called pollchecks or yield points; also some or all can be GC preemption points. In fact the preemption points resemble cooperative scheduling at the JVM level and use priority for synchronization. This approach works only for uniprocessor systems, for multiprocessors explicit synchronization has to be introduced.

In both cases there might be critical sections in the JVM where reentry cannot be allowed. To solve this problem the JVM must disable interrupts around critical non-reenterable sections. The more fine grained this disabling of interrupts can be done, the more responsive to interrupts the system will be.

One could opt for second level handlers only. An interrupt fires and releases an associated schedulable object (handler). Once released, the handler will be scheduled by the JVM scheduler according to the release parameters. This is the RTSJ approach. The advantage is that interrupt handling is done in the context of a normal Java thread and scheduled as any other thread running on the system. The drawback is that there will be a delay from the occurrence of the interrupt until the thread gets scheduled. Additionally, the meaning of interrupt priorities, levels and masks used by the hardware may not map directly to scheduling parameters supported by the JVM scheduler.

In the following we focus on the ISR handler approach, because it allows programming the other paradigms within Java.

Hardware Properties

We assume interrupt hardware as it is found in most computer architectures: interrupts have a fixed priority associated with them – they are set with a *solder iron*. Furthermore, interrupts can be globally disabled. In most systems the first level handler is called with interrupts globally disabled. To allow nested interrupts – being able to interrupt the handler by a higher priority interrupt as in preemptive scheduling – the handler has to enable interrupts again. However, to avoid priority inversion between handlers only interrupts with a higher priority will be enabled, either by setting the interrupt level

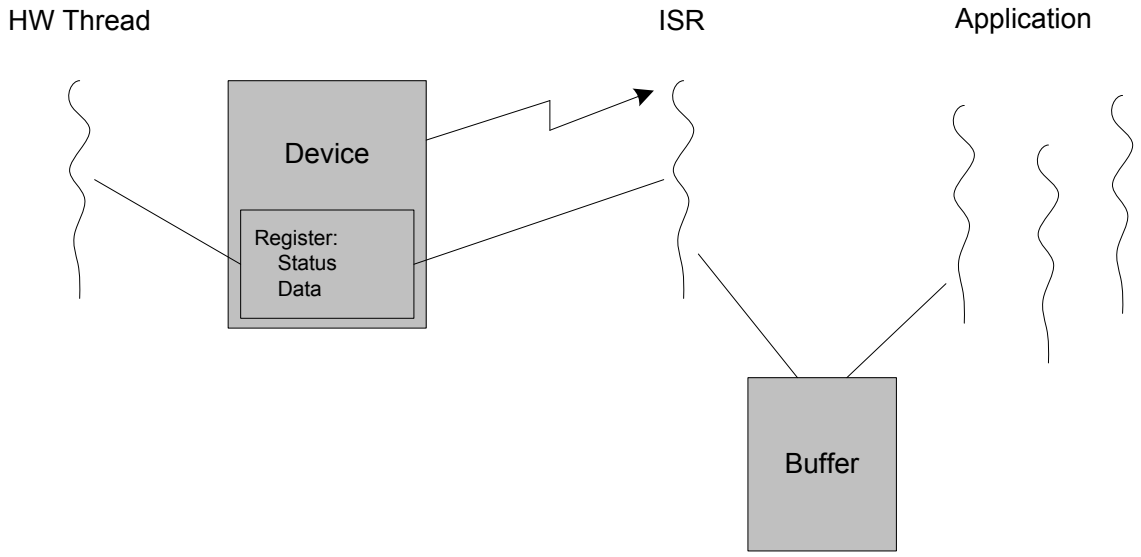


Figure 4.11: Threads and shared data

or setting the interrupt mask. Software threads are scheduled by a timer interrupt and usually have a lower priority than interrupt handlers (the timer interrupt has the lowest priority of all interrupts). Therefore, an interrupt handler is never preempted by a software thread.

Mutual exclusion between an interrupt handler and a software thread is ensured by disabling interrupts: either all interrupts or selectively. Again, to avoid priority inversion, only interrupts of a higher priority than the interrupt that shares the data with the software thread can be enabled. This mechanism is in effect the priority ceiling emulation protocol [32], sometimes called immediate ceiling protocol. It has the virtue that it eliminates the need for explicit locks (or Java monitors) on shared objects. Note that mutual exclusion with interrupt disabling works only in a uniprocessor setting. A simple solution for multiprocessors is to run the interrupt handler and associated software threads on the same processor core. A more involved scheme would be to use spin-locks between the processors.

When a device asserts an interrupt request line, the interrupt controller notifies the processor. The processor stops execution of the current thread. A partial thread context (program counter and processor status register) is saved. Then the ISR is looked up in the interrupt vector table and a jump is performed to the first instruction of the ISR. The handler usually saves additional thread context (e.g. the register file). It is also possible to switch to a new stack area. This is important for embedded systems where the stack sizes for all threads need to be determined at link time.

Synchronization

Java supports synchronization between Java threads with the synchronized keyword, either as a means of synchronizing access to a block of statements or to an entire method. In the general case this existing synchronization support is not sufficient to synchronize between interrupt handlers and threads.

Figure 4.11 shows the interacting active processes and the shared data in a scenario involving the handling of an interrupt. Conceptually three threads interact: (1) a hardware device thread represent-

ing the device activity, (2) the ISR, and (3) the application or device driver thread. These three share two types of data:

Device data The hardware thread and ISR share access to the device registers of the device signaling the interrupt

Application data The ISR and application or device driver share access to e.g., a buffer conveying information about the interrupt to the application

Regardless of which interrupt handling approach is used in Java, synchronization between the ISR and the device registers must be handled in an ad hoc way. In general there is no guarantee that the device has not changed the data in its registers; but if the ISR can be run to completion within the minimum inter-arrival time of the interrupt the content of the device registers can be trusted.

For synchronization between the ISR and the application (or device driver) the following mechanisms are available. When the ISR handler runs as a software thread, standard synchronization with object monitors can be used. When using the ISR handler approach, the handler is no longer scheduled by the normal Java scheduler, but by the hardware. While the handler is running, all other executable elements are suspended, including the scheduler. This means that the ISR cannot be suspended, must not block, or must not block via a language level synchronization mechanism; the ISR must run to completion in order not to freeze the system. This means that when the handler runs, it is guaranteed that the application will not get scheduled. It follows that the handler can access data shared with the application without synchronizing with the application. As the access to the shared data by the interrupt handler is not explicitly protected by a synchronized method or block, the shared data needs to be declared volatile.

On the other hand the application must synchronize with the ISR because the ISR may be dispatched at any point. To ensure mutual exclusion we redefine the semantics of the monitor associated with an InterruptHandler object: acquisition of the monitor disables all interrupts of the same and lower priority; release of the monitor enables the interrupts again. This procedure ensures that the software thread cannot be interrupted by the interrupt handler when accessing shared data.

Using the Interrupt Handler

Figure 4.12 shows an example of an interrupt handler for the serial port receiver interrupt. The method handle() is the interrupt handler method and needs no synchronization as it cannot be interrupted by a software thread. However, the shared data needs to be declared volatile as it is changed by the device driver thread. Method read() is invoked by the device driver thread and the shared data is protected by the InterruptHandler monitor. The serial port interrupt handler uses the hardware object SerialPort to access the device.

Garbage Collection

When using the ISR handler approach it is not feasible to let interrupt handlers be paused during a lengthy stop-the-world collection. Using this GC strategy the entire heap is collected at once and it is not interleaved with execution. The collector can safely assume that data required to perform the collection will not change during the collection, and an interrupt handler shall not change data used by the GC to complete the collection. In the general case, this means that the interrupt handler is not allowed to create new objects, or change the graph of live objects.

```
public class SerialHandler extends InterruptHandler {

    // A hardware object represents the serial device
    private SerialPort sp;

    final static int BUF_SIZE = 32;
    private volatile byte buffer[];
    private volatile int wrPtr, rdPtr;

    public SerialHandler(SerialPort sp) {
        this.sp = sp;
        buffer = new byte[BUF_SIZE];
        wrPtr = rdPtr = 0;
    }

    // This method is scheduled by the hardware
    public void handle() {
        byte val = (byte) sp.data;
        // check for buffer full
        if ((wrPtr+1)%BUF_SIZE!=rdPtr) {
            buffer[wrPtr++] = val;
        }
        if (wrPtr>=BUF_SIZE) wrPtr=0;
        // enable interrupts again
        enableInterrupt();
    }

    // This method is invoked by the driver thread
    synchronized public int read() {
        if (rdPtr!=wrPtr) {
            int val = ((int) buffer[rdPtr++]) & 0xff;
            if (rdPtr>=BUF_SIZE) rdPtr=0;
            return val;
        } else {
            return -1;        // empty buffer
        }
    }
}
```

Figure 4.12: An interrupt handler for a serial port receive interrupt

```
package com.board-vendor.io;

public class IOSystem {

    // some JVM mechanism to create the hardware objects
    private static ParallelPort pp = jvmPPCreate();
    private static SerialPort sp = jvmSPCreate(0);
    private static SerialPort gps = jvmSPCreate(1);

    public static ParallelPort getParallelPort() {
        return pp;
    }

    public static SerialPort getSerialPort() {..}
    public static SerialPort getGpsPort() {..}
}
```

Figure 4.13: A factory with static methods for Singleton hardware objects

With an incremental GC the heap is collected in small incremental steps. Write barriers in the mutator threads and non-preemption sections in the GC thread synchronize the view of the object graph between the mutator threads and the GC thread. With incremental collection it is possible to allow object allocation and changing references inside an interrupt handler (as it is allowed in any normal thread). With a real-time GC the maximum blocking time due to GC synchronization with the mutator threads must be known.

Interruption of the GC during an object move can result in access to a stale copy of the object inside the handler. A solution to this problem is to allow for pinning of objects reachable by the handler (similar to immortal memory in the RTSJ). Concurrent collectors have to solve this issue for threads anyway. The simplest approach is to disable interrupt handling during the object copy. As this operation can be quite long for large arrays, several approaches to split the array into smaller chunks have been proposed [33] and [3]. A Java processor may support incremental array copying with redirection of the access to the correct part of the array [30]. Another solution is to abort the object copy when writing to the object. It is also possible to use replication – during an incremental copy operation, writes are performed on both from-space and to-space object replicas, while reads are performed on the from-space replica.

4.3.3 Generic Configuration

An important issue for a HAL is a safe abstraction of device configurations. A definition of factories to create hardware and interrupt objects should be provided by board vendors. This configuration is isolated with the help of Java packages – only the objects and the factory methods are visible. The configuration abstraction is independent of the JVM. A device or interrupt can be represented by an identical hardware or interrupt object for different JVMs. Therefore, device drivers written in Java are JVM independent.

```
public class BaseBoard {

    private final static int SERIAL_ADDRESS = ...;
    private SerialPort serial;
    BaseBoard() {
        serial = (SerialPort) jvmHWOCreate(SERIAL_ADDRESS);
    };
    static BaseBoard single = new BaseBoard();
    public static BaseBoard getBaseFactory() {
        return single;
    }
    public SerialPort getSerialPort() { return serial; }

    // here comes the JVM internal mechanism
    Object jvmHWOCreate(int address) {...}
}

public class ExtendedBoard extends BaseBoard {

    private final static int GPS_ADDRESS = ...;
    private final static int PARALLEL_ADDRESS = ...;
    private SerialPort gps;
    private ParallelPort parallel;
    ExtendedBoard() {
        gps = (SerialPort) jvmHWOCreate(GPS_ADDRESS);
        parallel = (ParallelPort) jvmHWOCreate(PARALLEL_ADDRESS);
    };
    static ExtendedBoard single = new ExtendedBoard();
    public static ExtendedBoard getExtendedFactory() {
        return single;
    }
    public SerialPort getGpsPort() { return gps; }
    public ParallelPort getParallelPort() { return parallel; }
}
```

Figure 4.14: A base class of a hardware object factory and a factory subclass

Hardware Object Creation

The idea to represent each device by a single object or array is straightforward, the remaining question is: How are those objects created? An object that represents a device is a typical Singleton [9]. Only a single object should map to one instance of a device. Therefore, hardware objects cannot be instantiated by a simple new: (1) they have to be mapped by some JVM mechanism to the device registers and (2) each device instance is represented by a single object.

Each device object is created by its own factory method. The collection of these methods is the board configuration, which itself is also a Singleton (the application runs on a single board). The

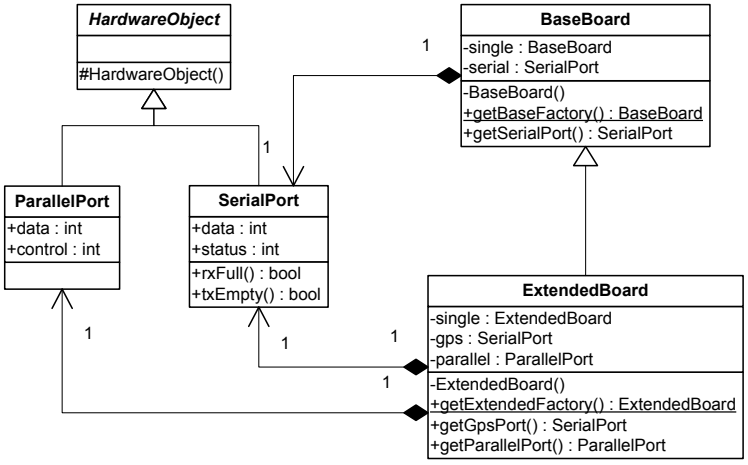


Figure 4.15: Device object classes and board factory classes

Singleton property of the configuration is enforced by a class that contains only static methods. Figure 4.13 shows an example for such a class. The class IOSystem represents a system with three devices: a parallel port, as discussed before to interact with the environment, and two serial ports: one for program download and one which is an interface to a GPS receiver.

This approach is simple, but not very flexible. Consider a vendor who provides boards in slightly different configurations (e.g., with different number of serial ports). With the above approach each board requires a different (or additional) IOSystem class that lists all devices. A more elegant solution is proposed in the next section.

Board Configurations

Replacing the static factory methods by instance methods avoids code duplication; inheritance then gives configurations. With a factory object we represent the common subset of I/O devices by a base class and the variants as subclasses.

However, the factory object itself must still be a Singleton. Therefore the board specific factory object is created at class initialization and is retrieved by a static method. Figure 4.14 shows an example of a base factory and a derived factory. Note how getBaseFactory() is used to get a single instance of the factory. We have applied the idea of a factory two times: the first factory generates an object that represents the board configuration. That object is itself a factory that generates the objects that interface to the hardware device.

The shown example base factory represents the minimum configuration with a single serial port for communication (mapped to System.in and System.out) represented by a SerialPort. The derived configuration ExtendedBoard contains an additional serial port for a GPS receiver and a parallel port for external control.

Furthermore, we show in Figure 4.14 a different way to incorporate the JVM mechanism in the factory: we define well known constants (the memory addresses of the devices) in the factory and let the native function jvmHWOCreate() return the correct device type.

Figure 4.15 gives a summary example of hardware object classes and the corresponding factory classes as an UML class diagram. The figure shows that all device classes subclass the abstract class HardwareObject. Figure 4.15 represents the simple abstraction as it is seen by the user of hardware objects.

```
abstract public class InterruptHandler implements Runnable {
    ...

    public InterruptHandler(int index) { ... };

    protected void startInterrupt() { ... };
    protected void endInterrupt() { ... };

    protected void disableInterrupt() { ... };
    protected void enableInterrupt() { ... };
    protected void disableLocalCPUInterrupts() { ... };
    protected void enableLocalCPUInterrupts() { ... };

    public void register() { ... };
    public void unregister() { ... };

    abstract public void handle() { ... };

    public void run() {
        startInterrupt();
        handle();
        endInterrupt();
    }
}
```

Figure 4.16: Base class for the interrupt handlers

Interrupt Handler Registration

We provide a base interrupt handling API that can be used both for non-RTSJ and RTSJ interrupt handling. The base class that is extended by an interrupt handler is shown in Figure 4.16. The handle() method contains the device server code. Interrupt control operations that have to be invoked before serving the device (i.e. interrupt masking and acknowledging) and after serving the device (i.e. interrupt re-enabling) are hidden in the run() method of the base InterruptHandler, which is invoked when the interrupt occurs.

The base implementation of InterruptHandler also provides methods for enabling and disabling a particular interrupt or all local CPU interrupts and a special monitor implementation for synchronization between an interrupt handler thread and an application thread. Moreover, it provides methods for non-RTSJ registering and deregistering the handler with the hardware interrupt source.

Registration of a RTSJ interrupt handler requires more steps (see Figure 4.17). The InterruptHandler instance serves as the RTSJ logic for a (bound) asynchronous event handler, which is added as handler to an asynchronous event which then is bound to the interrupt source.

```
ih = new SerialInterruptHandler(); // logic of new BAEH

serialFirstLevelEvent = new AsyncEvent();
serialFirstLevelEvent.addHandler(
    new BoundAsyncEventHandler( null, null, null, null, null, false, ih )
    );

serialFirstLevelEvent.bindTo("INT4");
```

Figure 4.17: Creation and registration of a RTSJ interrupt handler

4.3.4 Perspective

An interesting topic is to define a common standard for hardware objects and interrupt handlers for different platforms. If different device types (hardware chips) that do not share a common register layout are used for a similar function, the hardware objects will be different. However, if the structure of the devices is similar, as it is the case for the serial port on the three different platforms used for the implementation (see Section 4.4), the driver code that *uses* the hardware object is identical.

If the same chip (e.g., the 8250 type and compatible 16x50 devices found in all PCs for the serial port) is used in different platforms, the hardware object and the device driver, which also implements the interrupt handler, can be shared. The hardware object, the interrupt handler, and the visible API of the factory classes are independent of the JVM and the OS. Only the *implementation* of the factory methods is JVM specific. Therefore, the JVM independent HAL can be used to start the development of drivers for a Java OS on any JVM that supports the proposed HAL.

4.3.5 Summary

Hardware objects are easy to use for a programmer, and the corresponding definitions are comparatively easy to define for a hardware designer or manufacturer. For a standardized HAL architecture we proposed factory patterns. As shown, interrupt handlers are easy to use for a programmer that knows the underlying hardware paradigm, and the definitions are comparatively easy to develop for a hardware designer or manufacturer, for instance using the patterns outlined in this section. Hardware objects and interrupt handler infrastructure have a few subtle implementation points which are discussed in the next section.

4.4 Implementation

We have implemented the core concepts on four different JVMs[7] to validate the proposed Java HAL. Table 4.2 classifies the four execution environments according to two important properties: (1) whether they run on bare metal or on top of an OS and (2) whether Java code is interpreted or executed natively. Thereby we cover the whole implementation spectrum with our four implementations. Even though the suggested Java HAL is intended for systems running on bare metal, we include systems running on top of an OS because most existing JVMs still require an OS, and in order for them to migrate incrementally to run directly on the hardware they can benefit from supporting a Java HAL.

[7] On JOP the implementation of the Java HAL is already in use in production code.

	Direct (no OS)	Indirect (OS)
Interpreted	SimpleRTJ	Kaffe VM
Native	JOP	OVM

Table 4.2: Embedded Java Architectures

In the direct implementation a JVM without an OS is extended with I/O functionality. The indirect implementation represents an abstraction mismatch – we actually re-map the concepts. Related to Figure 4.6 in the introduction, OVM and Kaffe represent configuration (a), SimpleRTJ configuration (b), and JOP configuration (c).

The SimpleRTJ JVM [25] is a small, interpreting JVM that does not require an OS. JOP [26, 28] is a Java processor executing Java bytecodes directly in hardware. Kaffe JVM [36] is a complete, full featured JVM supporting both interpretation and JIT compilation; in our experiments with Kaffe we have used interpretative execution only. The OVM JVM [2] is an execution environment for Java that supports compilation of Java bytecodes into the C language, and via a C compiler into native machine instructions for the target hardware. Hardware objects have also been implemented in the research JVM, CACAO [19, 29].

In the following we provide the different implementation approaches that are necessary for the very different JVMs. Implementing hardware objects was straightforward for most JVMs; it took about one day to implement them in JOP. In Kaffe, after familiarizing us with the structure of the JVM, it took about half a day of pair programming.

Interrupt handling in Java is straightforward in a JVM not running on top of an OS (JOP and SimpleRTJ). Kaffe and OVM both run under vanilla Linux or the real-time version Xenomai Linux [39]. Both versions use a distinct user/kernel mode and it is not possible to register a user level method as interrupt handler. Therefore, we used threads at different levels to simulate the Java handler approach. The result is that the actual Java handler is the 3rd or even 4th level handler. This solution introduces quite a lot of overheads due to the many context switches. However, it is intended to provide a stepping stone to allow device drivers in Java; the goal is a real-time JVM that runs on the bare hardware.

In this section we provide more implementation details than usual to help other JVM developers to add a HAL to their JVM. The techniques used for the JVMs can probably not be used directly. However, the solutions (or sometimes work-arounds) presented here should give enough insight to guide other JVM developers.

4.4.1 SimpleRTJ

The SimpleRTJ JVM is a small, simple, and portable JVM. We have ported it to run on the bare metal of a small 16 bit microcontroller. We have successfully implemented the support for hardware objects in the SimpleRTJ JVM. For interrupt handling we use the ISR handler approach described in Section 4.3.2. Adding support for hardware objects was straightforward, but adding support for interrupt handling required more work.

Hardware Objects

Given an instance of a hardware object as shown in Figure 4.18 one must calculate the base address of the I/O port range, the offset to the actual I/O port, and the width of the port at runtime. We

```
public final class SerialPort extends HardwareObject {
    // LSR (Line Status Register)
    public volatile int status;
    // Data register
    public volatile int data;
    ...
}
```

Figure 4.18: A simple hardware object

```
SerialPort createSerialPort(int baseAddress ) {
    SerialPort sp = new SerialPort(baseAddress);
    return sp;
}
```

Figure 4.19: Creating a simple hardware object

have chosen to store the base address of the I/O port range in a field in the common super-class for all hardware objects (HardwareObject). The hardware object factory passes the platform and device specific base address to the constructor when creating instances of hardware objects (see Figure 4.19).

In the put/getfield bytecodes the base address is retrieved from the object instance. The I/O port offset is calculated from the offset of the field being accessed: in the example in Figure 4.18 status has an offset of 0 whereas data has an offset of 4. The width of the field being accessed is the same as the width of the field type. Using these values the SimpleRTJ JVM is able to access the device register for either read or write.

Interrupt Handler

The SimpleRTJ JVM uses a simple stop-the-world garbage collection scheme. This means that within handlers, we prohibit use of the new keyword and writing references to the heap. These restrictions can be enforced at runtime by throwing a pre-allocated exception or at class loading by an analysis of the handler method. Additionally we have turned off the compaction phase of the GC to avoid the problems with moving objects mentioned in Section 4.3.2.

The SimpleRTJ JVM implements thread scheduling within the JVM. This means that it had to be refactored to allow for reentering the JVM from inside the first level interrupt handler. We got rid of all global state (all global variables) used by the JVM and instead allocate shared data on the C stack. For all parts of the JVM to still be able to access the shared data we pass around a single pointer to that data. In fact we start a new JVM for the interrupt handler with a temporary (small) Java heap and a temporary (small) Java stack. Currently we use 512 bytes for each of these items, which have proven sufficient for running non-trivial interrupt handlers so far.

The major part of the work was making the JVM reentrant. The effort will vary from one JVM implementation to another, but since global state is a bad idea in any case JVMs of high quality use very little global state. Using these changes we have experimented with handling the serial port receive interrupt.

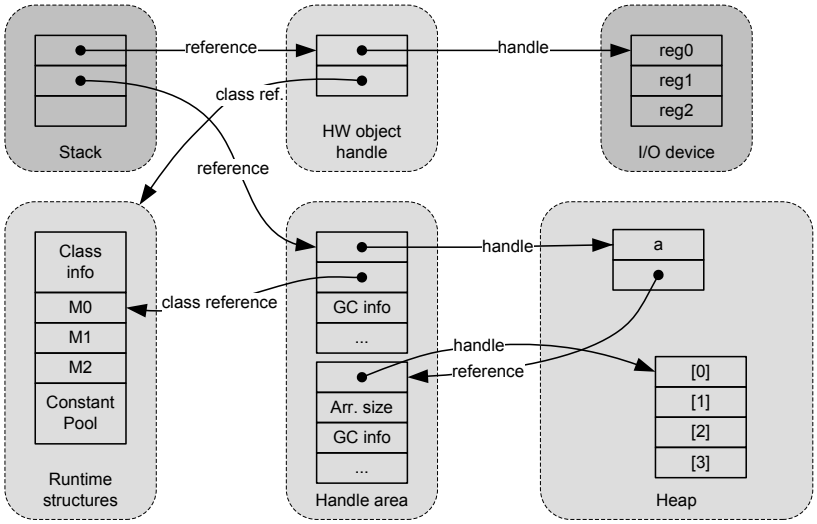


Figure 4.20: Memory layout of the JOP JVM

4.4.2 JOP

JOP is a Java processor intended for hard real-time systems [26, 28]. All architectural features have been carefully designed to be time-predictable with minimal impact on average case performance. We have implemented the proposed HAL in the JVM for JOP. No changes inside the JVM (the microcode in JOP) were necessary. Only the creation of the hardware objects needs a JOP specific factory.

Hardware Objects

In JOP, objects and arrays are referenced through an indirection called *handle*. This indirection is a lightweight read barrier for the compacting real-time GC [27, 31]. All handles for objects in the heap are located in a distinct memory region, the handle area. Besides the indirection to the *real* object the handle contains auxiliary data, such as a reference to the class information, the array length, and GC related data. Figure 4.20 shows an example with a small object that contains two fields and an integer array of length 4. The object and the array on the heap just contain the data and no additional hidden fields. This object layout greatly simplifies our object to device mapping. We just need a handle where the indirection points to the memory mapped device registers instead of into the heap. This configuration is shown in the upper part of Figure 4.20. Note that we do not need the GC information for the hardware object handles. The factory, which creates the hardware objects, implements this indirection.

As described in Section 4.3.3 we do not allow applications to create hardware objects; the constructor is private (or package visible). Figure 4.21 shows part of the hardware object factory that creates the hardware object SerialPort. Two static fields (SP_PTR and SP_MTAB) are used to store the handle to the serial port object. The first field is initialized with the base address of the I/O device; the second field contains a pointer to the class information.[8] The address of the static field SP_PTR is returned as the reference to the serial port object.

[8] In JOP's JVM the class reference is a pointer to the method table to speed-up the invoke instruction. Therefore, the name is XX_MTAB.

```
package com.jopdesign.io;

public class BaseFactory {

    // static fields for the handle of the hardware object
    private static int SP_PTR;
    private static int SP_MTAB;

    private SerialPort sp;

    IOFactory() {
        sp = (SerialPort) makeHWObject(new SerialPort(), Const.IO_UART1_BASE, 0);
    };

    ...

    // That's the JOP version of the JVM mechanism
    private static Object makeHWObject(Object o, int address, int idx) {
        int cp = Native.rdIntMem(Const.RAM_CP);
        return JVMHelp.makeHWObject(o, address, idx, cp);
    }
}

package com.jopdesign.sys;

public class JVMHelp {

    public static Object makeHWObject(Object o, int address, int idx, int cp) {
        // usage of native methods is allowed here as
        // we are in the JVM system package
        int ref = Native.toInt(o);
        // fill in the handle in the two static fields
        // and return the address of the handle as a
        // Java object
        return Native.toObject(address);
    }
}
```

Figure 4.21: Part of a factory and the helper method for the hardware object creation in the factory

The class reference for the hardware object is obtained by creating a *normal* instance of SerialPort with new on the heap and copying the pointer to the class information. To avoid using native methods in the factory class we delegate JVM internal work to a helper class in the JVM system package as shown in Figure 4.21. That helper method returns the address of the static field SP_PTR as reference to the hardware object. All methods in class Native, a JOP system class, are *native*[9] methods for low-level functions – the code we want to avoid in application code. Method toInt(Object o) defeats Java's type safety and returns a reference as an int. Method toObject(int addr) is the inverse function to map an address to a Java reference. Low-level memory access methods are used to manipulate the JVM data structures.

To disallow the creation with new in normal application code, the visibility is set to package. However, the package visibility of the hardware object constructor is a minor issue. To access private static fields of an arbitrary class from the system class we had to change the runtime class information: we added a pointer to the first static primitive field of that class. As addresses of static fields get resolved at class linking, no such reference was needed so far.

Interrupt Handler

The original JOP [26, 28] was a very puristic hard real-time processor. There existed only one interrupt – the programmable timer interrupt as time is the primary source for hard real-time events. All I/O requests were handled by periodic threads that polled for pending input data or free output buffers. During the course of this research we have added an interrupt controller to JOP and the necessary software layers.

Interrupts and exact exceptions are considered the hard part in the implementation of a processor pipeline [14]. The pipeline has to be drained and the complete processor state saved. In JOP there is a translation stage between Java bytecodes and the JOP internal microcode [28]. On a pending interrupt (or exception generated by the hardware) we use this translation stage to insert a special bytecode in the instruction stream. This approach keeps the interrupt completely transparent to the core pipeline. The special bytecode that is unused by the JVM specification [21] is handled in JOP as any other bytecode: execute microcode, invoke a special method from a helper class, or execute Java bytecode from JVM.java. In our implementation we invoke the special method interrupt() from a JVM helper class.

The implemented interrupt controller (IC) is priority based. The number of interrupt sources can be configured. Each interrupt can be triggered in software by a IC register write as well. There is one global interrupt enable and each interrupt line can be enabled or disabled locally. The interrupt is forwarded to the bytecode/microcode translation stage with the interrupt number. When accepted by this stage, the interrupt is acknowledged and the global enable flag cleared. This feature avoids immediate handling of an arriving higher priority interrupt during the first part of the handler. The interrupts have to be enabled again by the handler at a *convenient* time. All interrupts are mapped to the same special bytecode. Therefore, we perform the dispatch of the correct handler in Java. On an interrupt the static method interrupt() from a system internal class gets invoked. The method reads the interrupt number and performs the dispatch to the registered Runnable as illustrated in Figure 4.22. Note how a hardware object of type SysDevice is used to read the interrupt number.

The timer interrupt, used for the real-time scheduler, is located at index 0. The scheduler is just a plain interrupt handler that gets registered at mission start at index 0. At system startup, the table of Runnables is initialized with dummy handlers. The application code provides the handler via a

[9] There are no *real* native functions in JOP – bytecode is the native instruction set. The very few native methods in class Native are replaced by special, unused bytecodes during class linking.

```
static Runnable ih[] = new Runnable[Const.NUM_INTERRUPTS];
static SysDevice sys = IOFactory.getFactory().getSysDevice();

static void interrupt() {

    ih[sys.intNr].run();
}
```

Figure 4.22: Interrupt dispatch with the static interrupt() method in the JVM helper class

```
public class InterruptHandler implements Runnable {

    public static void main(String[] args) {

        InterruptHandler ih = new InterruptHandler();
        IOFactory fact = IOFactory.getFactory();
        // register the handler
        fact.registerInterruptHandler(1, ih);
        // enable interrupt 1
        fact.enableInterrupt(1);
        .....
    }

    public void run() {
        System.out.println("Interrupt fired!");
    }
}
```

Figure 4.23: An example Java interrupt handler as Runnable

class that implements Runnable and registers that class for an interrupt number. We reuse the factory presented in Section 4.3.3. Figure 4.23 shows a simple example of an interrupt handler implemented in Java.

For interrupts that should be handled by an event handler under the control of the scheduler, the following steps need to be performed on JOP:

1. Create a SwEvent with the correct priority that performs the second level interrupt handler work

2. Create a short first level interrupt handler as Runnable that invokes fire() of the corresponding software event handler

3. Register the first level interrupt handler as shown in Figure 4.23 and start the real-time scheduler

In Section 4.5 we evaluate the different latencies of first and second level interrupt handlers on JOP.

4.4.3 Kaffe

Kaffe is an open-source[10] implementation of the JVM which makes it possible to add support for hardware objects and interrupt handlers. Kaffe requires a fully fledged OS such as Linux to compile and run. Although ports of Kaffe exist on uCLinux we have not been able to find a bare metal version of Kaffe. Thus even though we managed to add support of hardware objects and interrupt handling to Kaffe, it still cannot be used without an OS.

Hardware Objects

Hardware objects have been implemented in the same manner as in the SimpleRTJ, described in Section 4.4.1.

Interrupt Handler

Since Kaffe runs under Linux we cannot directly support the ISR handler approach. Instead we used the ISR event approach in which a thread blocks waiting for the interrupt to occur. It turned out that the main implementation effort was spent in the signaling of an interrupt occurrence from the kernel space to the user space.

We wrote a special Linux kernel module in the form of a character device. Through proper invocations of ioctl() it is possible to let the module install a handler for an interrupt (e.g. the serial interrupt, normally on IRQ 7). Then the Kaffe VM can make a blocking call to read() on the proper device. Finally the installed kernel handler will release the user space application from the blocked call when an interrupt occurs.

Using this strategy we have performed non-trivial experiments implementing a full interrupt handler for the serial interrupt in Java. Still, the elaborate setup requiring a special purpose kernel device is far from our ultimate goal of running a JVM on the bare metal. Nevertheless the experiment has given valuable experience with interrupt handlers and hardware objects at the Java language level.

4.4.4 OVM

OVM [2] is a research JVM allowing many configurations; it is primarily targeted at implementing a large subset of RTSJ while maintaining reasonable performance. OVM uses ahead of time compilation via the C language: it translates both application and VM bytecodes to C, including all classes that might be later loaded dynamically at run-time. The C code is then compiled by GCC.

Hardware Objects

To compile Java bytecode into a C program, the OVM's Java-to-C compiler internally converts the bytecode into an intermediate representation (IR) which is similar to the bytecode, but includes more codes. Transformations at the IR level are both optimizations and operations necessary for correct execution, such as insertion of null-pointer checks. The produced IR is then translated into C, allowing the C compiler to perform additional optimizations. Transformations at the IR level, which is similar to the bytecode, are also typical in other JVM implementations, such as Sun's HotSpot.

We base our access to hardware objects on IR instruction transformations. We introduce two new instructions outb and inb for byte-wide access to I/O ports. Then we employ OVM's instruction rewriting framework to translate accesses to hardware object fields, putfield and getfield instructions,

[10] http://www.kaffe.org/

Reading from the device register serial.data, saving the result to the stack

original bytecode	*stack content*		*modified bytecode*	*stack content*
	{serial}			{serial}
GETFIELD data	{io port address}		GETFIELD data	{io port address}
		$\Longrightarrow$	INB	{inval}

Writing a value on the stack into the device register serial.data

original bytecode	*stack content*		*modified bytecode*	*stack content*
	{serial}, {outval}			{serial}, {outval}
PUTFIELD data	*empty*	$\Longrightarrow$	SWAP	{outval}, {serial}
		$\Longrightarrow$	GETFIELD data	{outval}, {io port address}
		$\Longrightarrow$	OUTB	*empty*

Figure 4.24: Translation of bytecode for access to regular fields into bytecode for access to I/O port registers.

into sequences centered around outb and inb where appropriate. We did not implement word-wide or double-word wide access modes supported by a x86 CPU. We discuss how this could be done at the end of this section.

To minimize changes to the OVM code we keep the memory layout of hardware objects as if they were ordinary objects, and store port addresses into the fields representing the respective hardware I/O ports. Explained with the example from Figure 4.18, the instruction rewriting algorithm proceeds as follows: SerialPort is a subclass of HardwareObject; hence it is a hardware object, and thus accesses to all its public volatile int fields, status and data, are translated to port accesses to I/O addresses stored in those fields.

The translation (Figure 4.24) is very simple. In case of reads we append our new inb instruction after the corresponding getfield instruction in the IR: getfield will store the I/O address on the stack and inb will replace it by a value read from this I/O address. In case of writes we replace the corresponding putfield instruction by a sequence of swap, getfield, and outb. The swap rotates the two top elements on stack, leaving the hardware object reference on top of the stack and the value to store to the I/O port below it, The getfield replaces the object reference by the corresponding I/O address, and outb writes the value to the I/O port.

The critical part of hardware object creation is to set I/O addresses into hardware object fields. Our approach allows a method to turn off the special handling of hardware objects. In a hardware object factory method accesses to hardware object fields are handled as if they were fields of regular objects; we simply store I/O addresses to the fields.

A method can turn off the special handling of hardware objects with a marker exception mechanism which is a natural solution within OVM. The method declares to throw a PragmaNoHWIOReg-

istersAccess exception. This exception is neither thrown nor caught, but the OVM IR level rewriter detects the declaration and disables rewriting accordingly. As the exception extends RuntimeException, it does not need to be declared in interfaces or in code calling factory methods. In Java 1.5, not supported by OVM, a standard substitute to the marker exception would be method annotation.

Our solution depends on the representation of byte-wide registers by 16-bit fields to hold the I/O address. However, it could still be extended to support multiple-width accesses to I/O ports (byte, 16-bit, and 32-bit) as follows: 32-bit I/O registers are represented by Java long fields, 16-bit I/O registers by Java int fields, and byte-wide I/O registers by Java short fields. The correct access width will be chosen by the IR rewriter based on the field type.

Interrupt Handler

Low-level support depends heavily on scheduling and preemption. For our experiments we chose the uni-processor x86 OVM configuration with green threads running as a single Linux process. The green threads, delayed I/O operations, and handlers of asynchronous events, such as POSIX signals, are only scheduled at well-defined points (*pollchecks*) which are by default at back-branches at bytecode level and indirectly at Java-level blocking calls (I/O operations, synchronization calls, etc). When no thread is ready to run, the OVM scheduler waits for events using the POSIX select call.

As OS we use Xenomai RT Linux [39, 11]. Xenomai tasks, which are in fact user-space Linux threads, can run either in the Xenomai primary domain or in the Xenomai secondary domain. In the primary domain they are scheduled by the Xenomai scheduler, isolated from the Linux kernel. In the secondary domain Xenomai tasks behave as regular real-time Linux threads. Tasks can switch to the primary domain at any time, but are automatically switched back to the secondary domain whenever they invoke a Linux system call. A single Linux process can have threads of different types: regular Linux threads, Xenomai primary domain tasks, and Xenomai secondary domain tasks. Primary domain tasks can wait on hardware interrupts with a higher priority than the Linux kernel. The Xenomai API provides the interrupts using the ISR event handler approach and supports *virtualization* of basic interrupt operations – disabling and enabling a particular interrupt or all local CPU interrupts. These operations have the same semantics as real interrupts, and disabling/enabling a particular one leads to the corresponding operation being performed at the hardware level.

Before our extension, OVM ran as a single Linux process with a single (native Linux) thread, a *main OVM thread*. This native thread implemented Java green threads. To support interrupts we add additional threads to the OVM process: for each interrupt source handled in OVM we dynamically add an interrupt listener thread running in the Xenomai primary domain. The mechanism that leads to invocation of the Java interrupt handler thread is illustrated in Figure 4.25.

Upon receiving an interrupt, the listener thread marks the pending interrupt in a data structure shared with the main OVM thread. When it reaches a pollcheck, it discovers that an interrupt is pending. The scheduler then immediately wakes-up and schedules the Java green thread that is waiting for the interrupt (IRQ server thread in the figure). To simulate the first level ISR handler approach, this green thread invokes some handler method. In a non-RTSJ scenario the green thread invokes the run() method of the associated InterruptHandler (see Figure 4.16). In an RTSJ scenario (not shown in Figure 4.25), a specialized thread fires an asynchronous event bound to the particular interrupt source. It invokes the fire() method of the respective RTSJ's AsyncEvent. As mentioned in Section 4.3.3 the RTSJ logic of AsyncEventHandler (AEH) registered to this event should be an instance of InterruptHandler in order to allow the interrupt handling code to access basic interrupt handling operations.

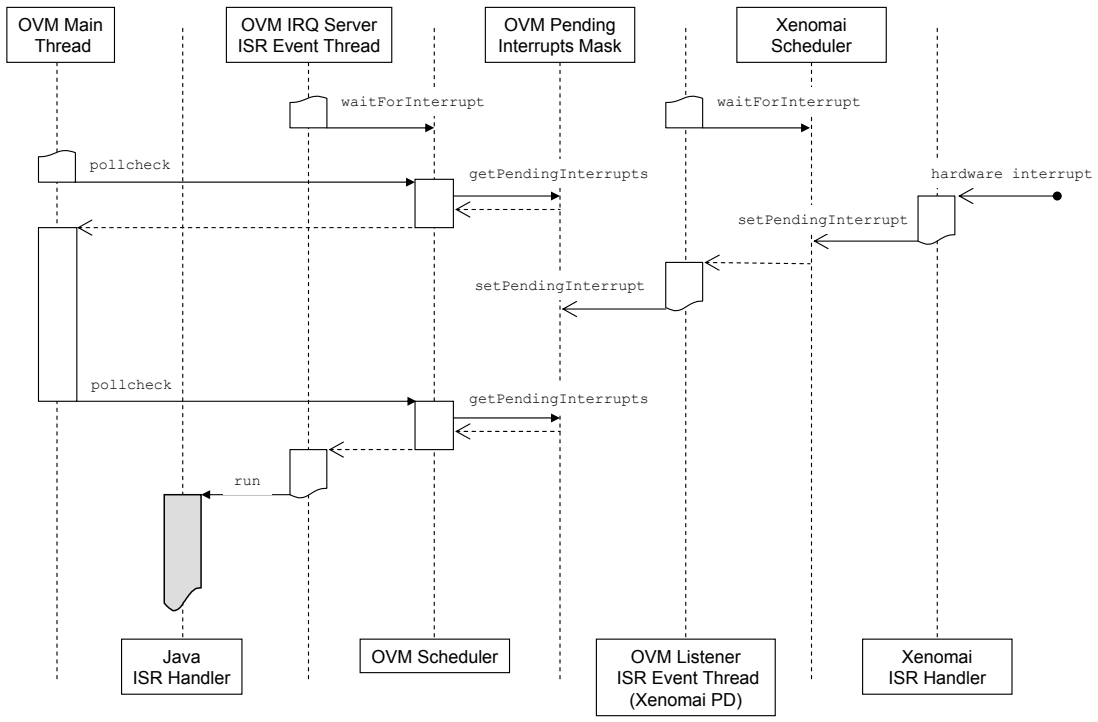


Figure 4.25: Invocation of a Java interrupt handler under OVM/Xenomai.

As just explained, our first level InterruptHandlers virtualize the interrupt handling operations for interrupt enabling, disabling, etc. Therefore, we have two levels of interrupt virtualization, one is provided by Xenomai to our listener thread, and the other one, on top of the first one, is provided by the OVM runtime to the InterruptHandler instance. In particular, disabling/enabling of local CPU interrupts is emulated, hardware interrupts are disabled/enabled and interrupt completion is performed at the interrupt controller level (via the Xenomai API), and interrupt start is emulated; it only tells the listener thread that the interrupt was received.

The RTSJ scheduling features (deadline checking, inter-arrival time checking, delaying of sporadic events) related to release of the AEH should not require any further adaptations for interrupt handling. We could not test these features as OVM does not implement them.

OVM uses *thin monitors* which means that a monitor is only instantiated (*inflated*) when a thread has to block on acquiring it. This semantic does not match to what we need – disable the interrupt when the monitor is acquired to prevent the handler from interrupting. Our solution provides a special implementation of a monitor for interrupt handlers and inflate it in the constructor of InterruptHandler. This way we do not have to modify the monitorenter and monitorexit instructions and we do not slow down regular thin monitors (non-interrupt based synchronization).

4.4.5 Summary

Support for hardware objects (see Section 4.3.1) and interrupt handling (see Section 4.3.2) to all four JVMs relies on common techniques. Accessing device registers through hardware objects extends

the interpretation of the bytecodes putfield and getfield or redirects the pointer to the object. If these bytecodes are extended to identify the field being accessed as inside a hardware object, the implementation can use this information. Similarly, the implementation of interrupt handling requires changes to the bytecodes monitorenter and monitorexit or pre-inflating a specialized implementation of a Java monitor. In case of the bytecode extension, the extended codes specify if the monitor being acquired belongs to an interrupt handler object. If so, the implementation of the actual monitor acquisition must be changed to disable/enable interrupts. Whether dealing with hardware or interrupt objects, we used the same approach of letting the hardware object and interrupt handler classes inherit from the super classes HardwareObject and InterruptHandler respectively.

For JVMs that need a special treatment of bytecodes putfield and getfield (SimpleRTJ, Kaffe, and OVM) bytecode rewriting at runtime can be used to avoid the additional check of the object type. This is a standard approach (called *quick* bytecodes in the first JVM specification) in JVMs to speedup field access of resolved classes.

Historically, registers of most x86 I/O devices are mapped to a dedicated I/O address space, which is accessed using dedicated instructions – port read and port writes. Fortunately, both the processor and Linux allow user-space applications running with administrator privileges to use these instructions and access the ports directly via iopl, inb, and outb calls. For both the Kaffe and OVM implementations we have implemented bytecode instructions putfield and getfield accessing hardware object fields by calls to iopl, inb, and outb.

Linux does not allow user-space applications to handle hardware interrupts. Only kernel space functionality is allowed to register interrupt handlers. We have overcome this issue in two different ways:

- For Kaffe we have written a special purpose kernel module through which the user space application (the Kaffe VM) can register interest in interrupts and get notified about interrupt occurrence.
- For OVM we have used the Xenomai real-time extension to Linux. Xenomai extends the Linux kernel to allow for the creation of real-time threads and allows user space code to wait for interrupt occurrences.

Both these work-arounds allow an incremental transition of the JVMs and the related development libraries into a direct (bare metal) execution environment. In that case the work-arounds would no longer be needed.

If a compiling JVM is used (either as JIT or ahead-of-time) the compiler needs to be aware of the special treatment of hardware objects and monitors on interrupt handlers. One issue which we did not face in our implementations was the alignment of object fields. When device registers are represented by differently sized integer fields, the compiler needs to pack the data structure.

The restrictions within an interrupt handler are JVM dependent. If an interruptible, real-time GC is used (as in OVM and JOP) objects can be allocated in the handler and the object graph may be changed. For a JVM with a stop-the-world GC (SimpleRTJ and Kaffe) allocations are not allowed because the handler can interrupt the GC.

4.5 Evaluation and Conclusion

Having implemented the Java HAL on four different JVMs we evaluate it on a several test applications, including a tiny web server, and measure the performance of hardware accesses via hardware objects and the latency of Java interrupt handlers.

4.5.1 Qualitative Observations

For first tests we implemented a serial port driver with hardware objects and interrupt handlers. As the structure of the device registers is exactly the same on a PC, the platform for SimpleRTJ, and JOP, we were able to use the exact same definition of the hardware object SerialPort and the test programs on all four systems.

Using the serial device we run an embedded TCP/IP stack, implemented completely in Java, over a SLIP connection. The TCP/IP stack contains a tiny web server and we serve web pages with a Java only solution similar to the one shown in the introduction in Figure 4.6. The TCP/IP stack, the tiny web server, and the hardware object for the serial port are the same for all platforms. The only difference is in the hardware object creation with the platform dependent factory implementations. The web server uses hardware objects and polling to access the serial device.

A Serial Driver in Java

For testing the interrupt handling infrastructure in OVM we implemented a serial interrupt based driver in Java and a demo application that sends back the data received through a serial interface. The driver part of the application is a full-duplex driver with support for hardware flow control and with detection of various error states reported by the hardware. The driver uses two circular buffers, one for receiving and the other for sending. The user part of the driver implements blocking getChar and putChar calls, which have (short) critical sections protected by the interrupt-disabling monitor. To reduce latencies the getChar call sets the DSR flag to immediately allow receiving more data and the putChar, after putting the character into the sending buffer, initiates immediately the sending, if this is not currently being done already by the interrupt machinery. The driver supports serial ports with a FIFO buffer. The user part of the demo application implements the loop-back using getChar and putChar. The user part is a RTSJ AsyncEventHandler which is fired when a new character is received. From a Java perspective this is a 2nd level interrupt handler, invoked after the corresponding serial event is fired from the 1st level handler. To test the API described in the paper we implemented two versions that differ in how the first level handler is bound to the interrupt: (a) a RTSJ style version where the first level handler is also a RTSJ event handler bound using bindTo to the JVM provided 1st level serial event, and (b) a non-RTSJ style version where the 1st level handler is registered using a InterruptHandler.register call. We have stress-tested the demo application and the underlying modified OVM infrastructure by sending large files to it through the serial interface and checked that they were returned intact.

The HAL in Daily Use

The original idea for hardware objects evolved during development of low-level software on the JOP platform. The abstraction with read and write functions and using constants to represent I/O addresses just *felt* wrong with Java. Currently hardware objects are used all over in different projects with JOP. Old code has been refactored to some extent, but new low-level code uses only hardware objects. By now low-level I/O is integrated into the language, e.g., auto completion in the Eclipse IDE makes it easy to access the factory methods and fields in the hardware object.

For experiments with an on-chip memory for thread-local scope caching [35] in the context of a chip-multiprocessor version of JOP, the hardware array abstraction greatly simplified the task. The on-chip memory is mapped to a hardware array and the RTSJ based scoped memory uses it. Creation of an object within this special scope is implemented in Java and is safe because the array bounds checks are performed by the JVM.

JNI vs Hardware Objects

JNI provides a way to access the hardware without changing the code of the JVM. Nevertheless, with a lack of commonly agreed API, using it for each application would be redundant and error prone. It would also add dependencies to the application: hardware platform and the operating system (the C API for accessing the hardware is not standardized). The build process is complicated by adding C code to it as well. Moreover, the system needs to support shared libraries, which is not always the case for embedded operating systems (example is RTEMS, used by ESA).

In addition, JNI is typically too heavy-weight to implement trivial calls such as port or memory access efficiently (no GC interaction, no pointers, no threads interaction, no blocking). Even JVMs that implement JNI usually have some other internal light-weight native interface which is the natural choice for hardware access. This leads us back to a Java HAL as illustrated here.

OVM Specific Experience

Before the addition of hardware objects, OVM did not allow hardware access because it did not and does not have JNI or any other native interface for user Java code. OVM has a simplified native interface for the virtual machine code which indeed we used when implementing the hardware objects. This native interface can as well be used to modify OVM to implement user level access to hardware via regular method calls. We have done this to implement a benchmark to measure HWO/native overheads (later in this section). As far as simple port access is concerned, none of the solutions is strictly better from the point of the JVM: the bytecode manipulation to implement hardware objects was easy, as well as adding code to propagate native port I/O calls to user code. Thanks to ahead-of-time compilation and the simplicity of the native interface, the access overhead is the same.

The OVM compiler is fortunately not "too smart" so it does not get in the way of supporting hardware objects: if a field is declared volatile side-effects of reading of that field are not a problem for any part of the system.

The API for interrupt handling added to OVM allows full control over interrupts, typically available only to the operating system. The serial port test application has shown that, at least for a simple device; it really allows us to write a driver. An interesting feature of this configuration is that OVM runs in user space and therefore it greatly simplifies development and debugging of Java-only device drivers for embedded platforms.

4.5.2 Performance

Our main objective for hardware objects is a clean object oriented interface to hardware devices. Performance of device register access is an important goal for relatively slow embedded processors; thus we focus on that in the following. It matters less on general purpose processors where the slow I/O bus essentially limits the access time.

Measurement Methodology

Execution time measurement of single instructions is only possible on simple in-order pipelines when a cycle counter is available. On a modern super-scalar architecture, where hundreds of instructions are in flight each clock cycle, direct execution time measurement becomes impossible. Therefore, we performed a bandwidth based measurement. We measure how many I/O instructions per second can be executed in a tight loop. The benchmark program is self-adapting and increases the loop count exponentially till the measurement run for more than one second and the iterations per second are

	JOP		OVM		SimpleRTJ		Kaffe	
	read	write	read	write	read	write	read	write
native	5	6	5517	5393	2588	1123	11841	11511
HW Object	13	15	5506	5335	3956	3418	9571	9394

Table 4.3: Access time to a device register in clock cycles

reported. To compensate for the loop overhead we perform an overhead measurement of the loop and subtract that overhead from the I/O measurement. The I/O bandwidth b is obtained as follows:

$$b = \frac{cnt}{t_{test} - t_{ovhd}}$$

Figure 4.26 shows the measurement loop for the read operation in method test() and the overhead loop in method overhead(). In the comment above the method the bytecodes of the loop kernel is shown. We can see that the difference between the two loops is the single bytecode getfield that performs the read request.

Execution Time

In Table 4.3 we compare the access time with native functions to the access via hardware objects. The execution time is given in clock cycles. We scale the measured I/O bandwidth b with the clock frequency f of the system under test by $n = \frac{f}{b}$.

We have run the measurements on a 100 MHz version of JOP. As JOP is a simple pipeline, we can also measure short bytecode instruction sequences with the cycle counter. Those measurements provided the exact same values as the ones given by our benchmark, such that they validated our approach. On JOP the native access is faster than using hardware objects because a native access is a special bytecode and not a native function call. The special bytecode accesses memory directly where the bytecodes putfield and getfield perform a null pointer check and indirection through the handle for the field access. Despite the slower I/O access via hardware objects on JOP, the access is fast enough for all currently available devices. Therefore, we will change all device drivers to use hardware objects.

The measurement for OVM was run on a Dell Precision 380 (Intel Pentium 4, 3.8 GHz, 3G RAM, 2M 8-way set associative L2 cache) with Linux (Ubuntu 7.10, Linux 2.6.24.3 with Xenomai-RT patch). OVM was compiled without Xenomai support and the generated virtual machine was compiled with all optimizations enabled. As I/O port we used the printer port. Access to the I/O port via a hardware object is just slightly faster than access via native methods. This was expected as the slow I/O bus dominates the access time.

On the SimpleRTJ JVM the native access is faster than access to hardware objects. The reason is that the JVM does not implement JNI, but has its own proprietary, more efficient, way to invoke native methods. It is done in a pre-linking phase where the invokestatic bytecode is instrumented with information to allow an immediate invocation of the target native function. On the other hand, using hardware objects needs a field lookup that is more time consuming than invoking a static method. With bytecode-level optimization at class load time it would be possible to avoid the expensive field lookup.

We measured the I/O performance with Kaffe on an Intel Core 2 Duo T7300, 2.00 GHz with Linux 2.6.24 (Fedora Core 8). We used access to the serial port for the measurement. On the interpreting

```
public class HwoRead extends BenchMark {

    SysDevice sys = IOFactory.getFactory().getSysDevice();

/*  Bytecodes in the loop kernel
    ILOAD 3
    ILOAD 4
    IADD
    ALOAD 2
    GETFIELD com/jopdesign/io/SysDevice.uscntTimer : I
    IADD
    ISTORE 3
*/
    public int test(int cnt) {

        SysDevice s = sys;
        int a = 0;
        int b = 123;
        int i;

        for (i=0; i<cnt; ++i) {
            a = a+b+s.uscntTimer;
        }
        return a;
    }

/*  Bytecodes in the loop kernel
    ILOAD 3
    ILOAD 4
    IADD
    ILOAD 2
    IADD
    ISTORE 3
*/
    public int overhead(int cnt) {

        int xxx = 456;
        int a = 0;
        int b = 123;
        int i;

        for (i=0; i<cnt; ++i) {
            a = a+b+xxx;
        }
        return a;
    }
}
```

Figure 4.26: Benchmark for the read operation measurement

Kaffe JVM we notice a difference between the native access and hardware object access. Hardware objects are around 20% faster.

Summary

For practical purposes the overhead on using hardware objects is insignificant. In some cases there may even be an improvement in performance. The benefits in terms of safe and structured code should make this a very attractive option for Java developers.

4.5.3 Interrupt Handler Latency

Latency on JOP

To measure interrupt latency on JOP we use a periodic thread and an interrupt handler. The periodic thread records the value of the cycle counter and triggers the interrupt. In the handler the counter is read again and the difference between the two is the measured interrupt latency. A plain interrupt handler as Runnable takes a constant 234 clock cycles (or 2.3 µs for a 100 MHz JOP system) between the interrupt occurrence and the execution of the first bytecode in the handler. This quite large time is the result of two method invocations for the interrupt handling: (1) invocation of the system method interrupt() and (2) invocation of the actual handler. For more time critical interrupts the handler code can be integrated in the system method. In that case the latency drops down to 0.78 µs. For very low latency interrupts the interrupt controller can be changed to emit different bytecodes depending on the interrupt number, then we avoid the dispatch in software and can implement the interrupt handler in microcode.

We have integrated the two-level interrupt handling at the application level. We set up two threads: one periodic thread, that triggers the interrupt, and a higher priority event thread that acts as second level interrupt handler and performs the handler work. The first level handler just invokes fire() for this second level handler and returns. The second level handler gets scheduled according to the priority. With this setup the interrupt handling latency is 33 µs. We verified this time by measuring the time between fire of the software event and the execution of the first instruction in the handler directly from the periodic thread. This took 29 µs and is the overhead due to the scheduler. The value is consistent with the measurements in [31]. There we measured a minimum useful period of 50 µs for a high priority periodic task.

The runtime environment of JOP contains a concurrent real-time GC [31]. The GC can be interrupted at a very fine granularity. During sections that are not preemptive (data structure manipulation for a new and write-barriers on a reference field write) interrupts are simply turned off. The copy of objects and arrays during the compaction phase can be interrupted by a thread or interrupt handler [30]. Therefore, the maximum blocking time is in the atomic section of the thread scheduler and not in the GC.

Latency on OVM/Xenomai

For measuring OVM/Xenomai interrupt latencies, we have extended an existing interrupt latency benchmark, written by Jan Kiszka from the Xenomai team [39]. The benchmark uses two machines connected over a serial line. The *log* machine, running a regular Linux kernel, toggles the RTS state of the serial line and measures the time it takes for the *target* machine to toggle it back.

To minimize measuring overhead the *log* machine uses only polling and disables local CPU interrupts while measuring. Individual measurements are stored in memory and dumped at shutdown

	Median (µs)	3rd Quartile (µs)	95% Quantile (µs)	Maximum (µs)
Polling	3	3	3	8
Kernel	14	16	16	21
Hard	14	16	16	21
User	17	19	19	24
Ovm	59	59	61	203

Table 4.4: Interrupt (and polling) latencies in microseconds.

so that they can be analyzed offline. We have made 400,000 measurements in each experiment, reporting only the last 100,000 (this was to warm-up the benchmark, including memory storage for the results). The *log* machine toggles the RTS state regularly with a given period.

We have tested 5 versions of the benchmark on the *target* machine: a polling version written in C (*polling*), a kernel-space interrupt handler in C/Xenomai running out of control of the Linux scheduler (*kernel*), a hard-realtime kernel-space interrupt handler running out of control of both the Xenomai scheduler and the Linux scheduler (*hard*), a user-space interrupt handler written in C/Xenomai (*user*), and finally an interrupt handler written in Java/OVM/Xenomai (*ovm*).

The results are shown in Table 4.4. The median latency is 3 µs for polling, 14 µs for both kernel-space handlers (hard and kernel), 17 µs for user-space C handler (user), and 59 µs for Java handler in OVM (ovm). Note that the table shows that the overhead of using interrupts over polling is larger than the overhead of handling interrupts in user-space over kernel-space. The maximum latency of OVM was 203 µs, due to infrequent pauses. Their frequency is so low that the measured 95% quantile is only 61 µs.

The experiment was run on Dell Precision 380 (Intel Pentium 4 3.8 GHz, 3G RAM, 2M 8-way set associative L2 cache) with Linux (Ubuntu 7.10, Linux 2.6.24.3 with Xenomai-RT patch). As Xenomai is still under active development we had to use Xenomai workarounds and bugfixes, mostly provided by Xenomai developers, to make OVM on Xenomai work.

Summary

The overhead for implementing interrupt handlers is very acceptable since interrupts are used to signal relatively infrequently occurring events like end of transmission, loss of carrier etc. With a reasonable work division between first level and second level handlers, the proposal does not introduce dramatic blocking terms in a real-time schedulability analysis, and thus it is suitable for embedded systems.

4.5.4 Discussion

Safety Aspects

Hardware objects map object fields to the device registers. When the class that represents a device is correct, access to it is safe – it is not possible to read from or write to an arbitrary memory address. A memory area represented by an array is protected by Java's array bounds check.

Portability

It is obvious that hardware objects are platform dependent; after all the idea is to have an interface to the bare metal. Nevertheless, hardware objects give device manufacturers an opportunity to supply supporting factory implementations that fit into Java's object-oriented framework and thus cater for developers of embedded software. If the same device is used on different platforms, the hardware object is portable. Therefore, standard hardware objects can evolve.

Compatibility with the RTSJ Standard

As shown for the OVM implementation, the proposed HAL is compatible with the RTSJ standard. We consider it to be a very important point since many existing systems have been developed using such platforms or subsets thereof. In further development of such applications existing and future interfacing to devices may be refactored using the proposed HAL. It will make the code safer and more structured and may assist in possible ports to new platforms.

4.5.5 Perspective

The many examples in the text show that we achieved a representation of the hardware close to being platform independent. Also, they show that it is possible to implement system level functionality in Java. As future work we consider to add devices drivers for common devices such as network interfaces[11] and hard disc controllers. On top of these drivers we will implement a file system and other typical OS related services towards our final goal of a Java only system.

An interesting question is whether a common set of *standard* hardware objects is definable. The SerialPort was a lucky example. Although the internals of the JVMs and the hardware were different one compatible hardware object worked on all platforms. It should be feasible that a chip manufacturer provides, beside the data sheet that describes the registers, a Java class for the register definitions of that chip. This definition can be reused in all systems that use that chip, independent of the JVM or OS.

Another interesting idea is to define the interaction between the GC and hardware objects. We stated that the GC should not collect hardware objects. If we relax this restriction we can redefine the semantics of collecting an object: on running the finalizer for a hardware object the device can be put into sleep mode.

Acknowledgement

We wish to thank Andy Wellings for his insightful comments on an earlier version of the paper. We also thank the reviewers for their detailed comments that helped to enhance the original submission. The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement number 216682 (JEOPARD).

[11] A device driver for a CS8900 based network chip is already part of the Java TCP/IP stack.

Bibliography

[1] aJile. aj-100 real-time low power Java processor. preliminary data sheet, 2000.

[2] Austin Armbruster, Jason Baker, Antonio Cunei, Chapman Flack, David Holmes, Filip Pizlo, Edward Pla, Marek Prochazka, and Jan Vitek. A real-time Java virtual machine with applications in avionics. *Trans. on Embedded Computing Sys.*, 7(1):1–49, 2007.

[3] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *POPL '03: Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 285–298, New York, NY, USA, 2003. ACM Press.

[4] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[5] Alan Burns and Andrew J. Wellings. *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley Longman Publishing Co., Inc., 3rd edition, 2001.

[6] James Caska. micro [μ] virtual-machine. Available at `http://muvium.com/`, accessed 2009.

[7] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. An empirical study of operating systems errors. *SIGOPS Oper. Syst. Rev.*, 35(5):73–88, 2001.

[8] Meik Felser, Michael Golm, Christian Wawersich, and Jürgen Kleinöder. The JX operating system. In *Proceedings of the USENIX Annual Technical Conference*, pages 45–58, 2002.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1994.

[10] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[11] Philippe Gerum. Xenomai - implementing a RTOS emulation framework on GNU/Linux. `http://www.xenomai.org/documentation/branches/v2.4.x/pdf/xenomai.pdf`, 2004.

[12] Trusted Computing Group. Trusted computing. Available at `https://www.trustedcomputinggroup.org/`, May 2008.

[13] Per Brinch Hansen. *The Architecture of Concurrent Programs*. Prentice-Hall Series in Automatic Computing. Prentice-Hall, 1977.

[14] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 3rd ed.* Morgan Kaufmann Publishers Inc., Palo Alto, CA 94303, 2002.

[15] Thomas Henties, James J. Hunt, Doug Locke, Kelvin Nilsen, Martin Schoeberl, and Jan Vitek. Java for safety-critical applications. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems (SafeCert 2009)*, Mar. 2009.

[16] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX), ACM SIGPLAN*, pages 93–104, Cambridge, MA, November 2000. ACM. Published as Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX), ACM SIGPLAN, volume 35, number 11.

[17] Galen Hunt, James R. Larus, Martin Abadi, Mark Aiken, Paul Barham, Manuel Fahndrich, Chris Hawblitzel, Orion Hodson, Steven Levi, Nick Murphy, Bjarne Steensgaard, David Tarditi, Ted Wobber, and Brian D. Zill. An overview of the singularity project. Technical Report MSR-TR-2005-135, Microsoft Research (MSR), October 2005.

[18] Stephan Korsholm, Martin Schoeberl, and Anders P. Ravn. Interrupt handlers in Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.

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

[20] Jochen Kreuzinger, Uwe Brinkschulte, Matthias Pfeffer, Sascha Uhrig, and Theo Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.

[21] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, USA, second edition, 1999.

[22] Sebastian Lohmeier. Jini on the Jnode Java os. Online article at `http://monochromata.de/jnodejini.html`, June 2005.

[23] Geoffrey Phipps. Comparing observed bug and productivity rates for java and c++. *Softw. Pract. Exper.*, 29(4):345–358, 1999.

[24] Anders P. Ravn. Device monitors. *IEEE Transactions on Software Engineering*, 6(1):49–53, January 1980.

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

[26] Martin Schoeberl. *JOP: A Java Optimized Processor for Embedded Real-Time Systems*. PhD thesis, Vienna University of Technology, 2005.

[27] Martin Schoeberl. Real-time garbage collection for Java. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC 2006)*, pages 424–432, Gyeongju, Korea, April 2006. IEEE.

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

[29] Martin Schoeberl, Stephan Korsholm, Christian Thalinger, and Anders P. Ravn. Hardware objects for Java. In *Proceedings of the 11th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2008)*, Orlando, Florida, USA, May 2008. IEEE Computer Society.

[30] Martin Schoeberl and Wolfgang Puffitsch. Non-blocking object copy for real-time garbage collection. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*. ACM Press, September 2008.

[31] Martin Schoeberl and Jan Vitek. Garbage collection for safety critical Java. In *Proceedings of the 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007)*, pages 85–93, Vienna, Austria, September 2007. ACM Press.

[32] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Comput.*, 39(9):1175–1185, 1990.

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

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

[35] Andy Wellings and Martin Schoeberl. Thread-local scope caching for real-time Java. In *Proceedings of the 12th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2009)*, Tokyo, Japan, March 2009. IEEE Computer Society.

[36] Tim Wilkinson. Kaffe – a virtual machine to run java code. Available at http://www.kaffe.org, 1996.

[37] Niklaus Wirth. Design and implementation of modula. *Software - Practice and Experience*, 7:3–84, 1977.

[38] Niklaus Wirth. *Programming in Modula-2*. Springer Verlag, 1982.

[39] Xenomai developers. Xenomai: Real-time framework for Linux. `http://www.xenomai.org`, 2008.

5 Time-predictable Computer Architecture

EURASIP Journal on Embedded Systems, Volume 2009, Article ID 758480, 17 pages, 2009, Hindawi

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

Abstract

Today's general-purpose processors are optimized for maximum throughput. Real-time systems need a processor with both a reasonable and a known worst-case execution time (WCET). Features such as pipelines with instruction dependencies, caches, branch prediction, and out-of-order execution complicate WCET analysis and lead to very conservative estimates. In this paper, we evaluate the issues of current architectures with respect to WCET analysis. Then we propose solutions for a time-predictable computer architecture. The proposed architecture is evaluated with implementation of some features in a Java processor. The resulting processor is a good target for WCET analysis and still performs well in the average case.

5.1 Introduction

Standard computer architecture is driven by the following paradigm: *Make the common case fast and the uncommon case correct* [26]. However, this design approach leads to architectures where the worst-case execution time (WCET) is high and hard to predict by static analysis. For real-time systems we have to design architectures with the following paradigm: *Make the worst case fast and the whole system easy to analyze*.

Classic enhancements in computer architectures are: pipelining, instruction and data caching, dynamic branch prediction, out-of-order execution, speculative execution, and fine-grained chip multithreading. These features are increasingly harder to model for the low-level WCET analysis. Execution history is the key to performance enhancements, but also the main issue for WCET analysis. Thus we need techniques to manage the execution history.

Pipelines shall be simple, with minimum dependencies between instructions. It is agreed that caches are mandatory to bridge the gap between processor speed and memory access time. Caches in general, and particularly data caches, are usually hard to analyze statically. Therefore, we are introducing caches that are organized to speed-up execution time and provide tight WCET bounds. We propose three different caches: (1) an instruction cache for full methods, (2) a stack cache and, (3) a small, fully associative buffer for heap access. Furthermore, the integration of a program- or compiler-managed scratchpad memory can help to tighten bounds for hard to analyze memory access patterns.

Out-of-order execution and speculation result in processor models that are too complex for WCET analysis. We argue that the transistors are better used on chip multiprocessors (CMP) with simple in-order pipelines. Real-time systems are naturally multithreaded and thus map well to the explicit parallelism of chip multiprocessors.

We propose a multiprocessor model with one processor per thread. Thread switching and schedulability analysis for each individual core disappears, but the access to the shared resource main memory still needs to be scheduled.

We have implemented most of the proposed concepts for evaluation in a Java processor. The Java processor JOP [61] is intended for real-time and safety critical applications written in a modern object oriented language. It has to be noted that all concepts can also be applied to a standard RISC processor. The following list points out the key arguments for a time-predictable computer architecture:

- There is a mismatch between performance oriented computer architectures and worst-case analyzability.
- Complex features result in increasingly complex models.
- Caches, a very important feature for high performance, need new organization.
- Thread level parallelism is natural in embedded systems. Exploration of this parallelism with simple chip multiprocessors is a valuable option.
- One thread per processor obviates the classic schedulability analysis and introduces scheduling of memory access.

Catching up with WCET analysis of features that enhance the average case performance is not an option for future real-time systems. We need a sea change and shall take the constructive approach by designing computer architectures where predictable timing is a first order design factor.

The contributions of the paper are twofold: (1) an extensive overview is given of processor features that make WCET estimation difficult; (2) solutions for a time-predictable architecture that can be implemented in RISC, CISC, or VLIW style processors are provided. The implementations of some of the proposed concepts in the context of a Java processor, as described in the evaluation section, have been previously published in [57] and [58].

The paper is organized as follows: Section 5.2 presents related work on real-time architectures. In Section 5.3, we describe the main issues that hamper tight WCET estimates of actual processors. We propose solutions for these issues in Section 5.4. In Section 5.5, we evaluate the proposed time-predictable computer architecture with an implementation of a Java processor in an FPGA. Section 5.6 concludes the paper.

5.2 Related Work

Bate et al. [7] discuss the usage of modern processors in safety critical applications. They compare commercial off-the-shelf (COTS) processors with a customized processor developed specifically for the safety critical domain. While COTS processors benefit from a large user base and the resulting maturity of the design process, customized processors provide following advantages:

- Design in conjunction with the safety argument

- Design for good worst-case performance
- Using only features that can be easily analyzed
- The processor can be treated as a *white box* during verification and testing

Despite these advantages, few research projects exist in the field of WCET optimized hardware. Thiele and Wilhelm [69] argue that a new research discipline is needed for time-predictable embedded systems to "match implementation concepts with techniques to improve analyzability".

Similarly, Edwards and Lee argue: "It is time for a new era of processors whose temporal behavior is as easily controlled as their logical function" [15]. A first simulation of their PRET architecture is presented in [40]. PRET implements the SPARC V8 instruction set architecture (ISA) in a six-stage pipeline and performs chip level multithreading for six threads to eliminate data forwarding and branch prediction. Scratchpad memories are used instead of instruction and data caches. The shared main memory is accessed via a TDMA scheme, called memory wheel, similar to the TDMA based arbiter used in the JOP CMP system [46]. The SPARC ISA is extended with a *deadline* instruction that stalls the current thread until the deadline is reached. This instruction is used to perform time based, instead of lock based, synchronization for access to shared data.

Berg et al. identify the following design principles for a time-predictable processor: "... recoverability from information loss in the analysis, minimal variation of the instruction timing, non-interference between processor components, deterministic processor behavior, and comprehensive documentation" [8]. The authors propose a processor architecture that meets these design principles. The processor is a classic five-stage RISC pipeline with minimal changes in the instruction set: it handles function calls with an explicit instruction for simpler reconstruction of the control flow graph and construction of 32-bit immediate values with two instructions to avoid immediate values in the code segment. The memory system has to be organized in Harvard-style with dedicated busses to the FLASH memory for the code and the SRAM memory for the data. The replacement strategy of caches has to be least-recently used (LRU).

Heckmann et al. provide examples of problematic processor features in [25]. The most problematic features found are the replacement strategies for set-associative caches. A pseudo-round-robin replacement strategy of the 4-way set-associative cache in the ColdFire MCF 5307 effectively renders the associativity useless for WCET analysis. The use of a single 2-bit counter for the whole cache destroys age information within the cache sets. The analysis of that cache results in effectively modeling only a quarter of the cache as a direct mapped cache. Similarly, a pseudo-LRU replacement strategy for an 8-way set-associative cache of the PowerPC 750/755 uses an age counter for each set. Here, only half of the cache is modeled by the analysis. Slightly more complex pipelines, with branch prediction and out-of-order execution, need an integrated pipeline and cache analysis to provide useful WCET bounds. Such an integrated analysis is complex and also demanding with respect to the computational effort. In conclusion Heckmann et al. suggest the following restrictions for time-predictable processors: (1) separate data and instruction caches; (2) locally deterministic update strategies for caches; (3) static branch prediction; and (4) limited out-of-order execution. The authors argue for restriction of processor features of actual processors (of the time) for embedded systems, but do not provide suggestions for *additional* or *alternative* features for a time-predictable processor.

The VISA approach [3] adapts a complex simultaneous multithreading processor that can be reconfigured to a simple single-issue pipeline. The complexity of the processor can be dynamically disabled at runtime. WCET analysis is performed for the simple pipeline. A task is divided into sub-tasks and each sub-task is assigned a checkpoint. The task is executed on the complex pipeline

and only if the checkpoint is missed the processor is switched to the simple mode. The checkpoint is inserted early enough to complete the sub-task on the simple pipeline before the deadline. The available slack time, when the task is executed on the fast, complex pipeline, is utilized for energy saving.

Puschner and Burns argue for a single-path programming style [53] that results in a constant execution time. In that case, WCET can easily be measured. However, this programming paradigm is quite uncommon and restrictive. Single-path programming can be inefficient when the control flow is data dependent. A processor, called SPEAR [14], was especially designed to evaluate the single-path programming paradigm. A single predicate bit can be set with a compare instruction whereby several instructions (e.g., move, arithmetic operations) can be predicated. The SPEAR implements a three-stage in-order pipeline and uses on-chip memories for instruction and data instead of caches.

Complex hardware and software architectures hinder hierarchical timing analysis [55]. A radical simplification of the whole system to avoid unwanted timing interactions is proposed – single path programming, execution of a single task/thread per core, simple in-order pipelines, and statically scheduled access to shared memory in CMPs.

Whitham argues that the execution time of a basic block has to be independent of the execution history [75]. As a consequence his MCGREP architecture reduces pipelining to two stages (fetch and execute) and avoids caches all together. To reduce the WCET, Whitham proposes to implement the time critical functions in microcode on a reconfigurable function unit (RFU). The main processor implements a RISC ISA as a microprogrammed, sequential processor. The interesting approach in MCGREP is that the RFUs implement the same architecture and microcode as the main CPU. Therefore, mapping a sequence of RISC instructions to microcode for one or several RFUs is straightforward. With several RFUs, it is possible to explicitly extract instruction level parallelism (ILP) from the original RISC code in a similar way to VLIW architectures.

Whitham and Audsley extend the MCGREP architecture with a trace scratchpad [76]. The trace scratchpad caches microcode and is placed after the decode stage. It is similar to a trace cache found in newer Pentium CPUs to cache the translated micro operations. The differences from a cache are that the execution from the trace scratchpad has to be explicitly started and the scratchpad has to be loaded under program control. The authors extract ILP at the microcode level and schedule the instructions statically – similar to a VLIW architecture.

5.3 WCET Analysis Issues

The WCET of tasks is the necessary input for schedulability analysis. Measuring the WCET is not a safe option. Only static WCET analysis can provide safe upper bounds of execution times.

WCET analysis can be separated into in high-level and low-level analysis. The high-level analysis is a mature research topic [39, 54, 21]. An overview of WCET related research can be found in [52] and [77]. The main issues that need to be solved are in the low-level analysis. The processors that can be analyzed are usually several generations behind actual architectures [19, 43, 25]. For example: Thesing models, in his PhD thesis [68] from 2004, the MPC755 variant of the PowerPC 750. The PowerPC 750 was introduced in 1997 and the MPC755 was *not recommended for new designs* in 2006.

The main issues in low-level analysis are features that increase average performance. All these features, such as multi-level caches, branch target buffer, out-of-order execution, and speculation, include a state that heavily depends on a large execution history. This caching of the execution history is actually fundamental for performance enhancements. However, it is this history that is

hard to model for WCET analysis. A long history leads to a state explosion for the final WCET calculation. Low-level WCET analysis thus usually performs simplifications and uses conservative estimates. One example of this conservative estimate is to classify a cache access as a miss, if the outcome of the cache access is unknown.

Lundqvist and Stenström have shown that this intuitive assumption can be wrong on dynamically scheduled microprocessors [42]. They provide an example of such a timing anomaly in which a cache hit can cause a longer execution time than a cache miss. The principles behind these timing anomalies are further elaborated in [74].

5.3.1 Pipeline Dependencies

Simple pipelines, similar to the original Berkeley/Stanford RISC design [45], are easy to model for WCET analysis. In a non-stalled pipeline, the execution time latency corresponds to the length of the pipeline. The effective execution time itself is only a single cycle. What makes pipeline analysis necessary are stalls introduced by dependencies within the pipeline. Those stalls are introduced by:

1. Data dependencies between instructions
2. Control dependencies between instructions

In one of the first RISC designs, the MIPS [27], these dependency hazards are explicitly exposed to the compiler. They have to be resolved by the compiler with instruction scheduling for delayed branches and for the single cycle delay between a memory load and the data use. Therefore, these effects are also recognized by the WCET tool. More advanced pipelines avoid exposing stalls from the ISA in order to avoid too many (compiler) target variations and retain binary compatibility between processor versions. Nevertheless, this information is needed for WCET analysis.

Dependencies within a basic block can be easily modeled. The challenge is to merge the effects from different basic blocks and across function boundaries. In [41], the *timing schema* [64] is extended to include the pipeline information. Timing schema is a tree based WCET analysis. After the determination of basic block execution times, the control flow graph is processed in a bottom-up manner until a final WCET bound is available. Branches are merged with the higher WCET bound as result. For the extension the pipeline is represented by reservation stations and the state at the head and tail of a basic block is considered when basic blocks are merged.

Pipelines with timing dependencies can result in an unbounded effect, called *long timing effect* (LTE) [17]. This means that an instruction far back in the history (longer than the pipeline length) influences the execution time of the current instruction. These LTEs can be negative or positive. A positive LTE means longer execution time. An instruction with a possible positive LTE needs a safe approximation of that effect for the pipeline analysis.

More complex pipelines can be analyzed with abstract interpretation, but the analysis time can become impractical. Berg et al. [8] report that up to 1000 states per instruction are needed for the model of the PowerPC 755. This processor was introduced in 1998 and we expect a considerable growth of the states that need to be tracked by abstract interpretation for newer processors.

5.3.2 Instruction Fetch

The instruction fetching is often decoupled from the main memory or the instruction cache by a prefetch unit. This unit fills the prefetch queue with instructions independently of the main pipeline. This form of prefetching is especially important for a variable length instruction set as the x86 ISA

or the bytecode instructions of the Java virtual machine (JVM). The fill status of the prefetch queue depends on the history of the instruction stream. The possible length of this history is unbounded. To model this queue for a WCET tool, we need to cut the maximum history and assume an empty queue at such a cut point.

In [80] the authors model the 4 byte long prefetch queue of an Intel 80188. Even for this simple prefetch queue, the authors have to perform some simplifications in their approach to handle the resulting complexity due to the interaction between the instruction execution and the instruction prefetch (the consuming and the producing end of the queue).

5.3.3 Caches

Between the middle of the 1980s and 2002, CPU performance increased by around 52% per year, but memory latency decreased only by 9% [26]. To bridge this growing gap between CPU and main memory performance, a memory hierarchy is used. Several layers with different tradeoffs between size, speed, and cost form that memory hierarchy. A typical hierarchy consists of:

1. Register file
2. Per processor level 1 instruction and data cache
3. On-chip, shared unified level 2 cache
4. Off-chip level 3 cache
5. Main memory
6. Hard disc for virtual memory

The only layer that is under the control of the compiler is the register file. The rest of the memory hierarchy is usually not visible – it is not part of the ISA abstraction. Placement of data in the different layers is performed automatically by the hardware for caches and by the OS for virtual memory management. The access time for a word located in a memory block paged out by the OS is several orders of magnitude higher than a level 1 cache hit. Even the access times to the level 1 cache and to the main memory differ by two orders of magnitudes.

Cache memories for the instructions and data are classic examples of the *make the common case fast* paradigm. Avoiding or ignoring this feature in real-time systems, due to its unpredictable behavior, results in a very pessimistic WCET bound. Much effort has been expended on research to integrate the instruction cache into the timing analysis of tasks [5, 24], on the cache's influence on task preemption [37, 12], and on integration of the cache analysis with the pipeline analysis [23]. The influence of different cache architectures on WCET analysis is described in [25].

A unified cache for data and instructions can easily destroy all the information on abstract cache states. Access to n unknown addresses in an n-way set-associative cache results in the state *unknown* for all cache lines. Modern processors usually have separate instruction and data caches for the level 1 cache. However, the level 2 cache is usually shared. Most CMP systems also share the level 2 cache between the different cores. The possible interactions between concurrent threads running on different cores are practically impossible to model.

Data caches are considerably harder to analyze than instruction caches. For some data accesses, especially for data allocated on the heap, the addresses cannot be predicted. However, access to the stack can be predicted statically. A data cache that caches heap and stack content suffers from the

same problem as a unified instruction and data cache: an unknown address for a heap access will evict one block from all sets in the abstract cache state and will increase the age of all cache blocks.

In a recent paper, Reineke et al. analyzed the predictability of different cache replacement policies [56]. It is shown that LRU performs best with respect to predictability. Pseudo-LRU and FIFO perform similarly. Both perform considerably worse than LRU. In an 8-way set-associative setting, Pseudo-LRU and FIFO take more than twice as long as LRU to recover from lost information.

5.3.4 Branch Prediction

Accurate branch prediction is of utmost importance to keep long pipelines filled. The penalty of a wrongly predicted conditional branch is typically almost as long as the pipeline length. Modern branch predictors guess the outcome primarily from results of earlier branches. They heavily rely on the execution history, an effect we want to avoid for a tight worst-case prediction. Global branch predictors and caches have a similar issue: as soon as a single index into the branch history is unknown, the whole information of branch prediction is lost for the analysis at that point.

Two-level branch predictors are not suitable for time-predictable architectures [18]; e.g., on the Pentium III, Pentium 4, and UltraSparc III a decrease in the number of loop iterations can actually result in an increase of the execution time. This is another form of timing anomaly [42].

Branch prediction also interferes with cache contents. When the analysis cannot anticipate the outcome of the prediction, both branch directions need to be considered for cache analysis.

5.3.5 Instruction Level Parallelism

Some microprocessors try to extract ILP from the instruction stream, i.e., execute more than one instruction per clock cycle. ILP extractions can be done either statically by the compiler or dynamically by the hardware.

Processors with static scheduled ILP are known as very long instruction word (VLIW) processors. The main issue of VLIW processors is that the pipeline details are exposed at the ISA. The compiler has to group parallel instructions and needs to consider pipeline constraints. Some processors rely on the compiler to resolve data dependencies and do not stall the pipeline. Therefore, each new generation of VLIW processors needs a new compiler back end. However, this issue is actually an advantage for low-level WCET analysis, as these details are needed for the pipeline analysis.

Dynamically scheduled, super-scalar microprocessors combine several parallel execution units with out-of-order execution to extract the ILP from the instruction stream. In current processors, about hundred instructions (e.g., 128 in the Pentium 4 [26]) can be in flight at each cycle. Analysis of a realistically sized application with an accurate processor model is thus (almost) impossible. Even modeling the pipeline states for basic blocks leads to a state space explosion. And modeling only basic blocks would result in very long penalties for the branches – on a later version of the Pentium 4, a simple instruction takes at least 31 clock cycles from fetch to retire [26].

Despite this complexity, in [38] a hypothetical out-of-order executing microprocessor is modeled for WCET analysis. Verification of the proposed approach on a real processor is missing. We think modeling out-of-order processors is practically not feasible.

5.3.6 Chip Multithreading

Dynamic extraction of ILP is limited to about two instructions per cycle on current processors, such as Pentium 4 and AMD Opteron [26]. Another path to speedup multithreaded workloads is the

extraction of thread-level parallelism (TLP). The concept of TLP in a single processor is quite old – it was used in the CDC 6600, a supercomputer from the 1960s – but is now being reconsidered in all desktop and server processors. Fine-grained multithreading can hide the latency of load/use hazards or a cache miss for one thread by the execution of other threads.

The main issue with multithreading in real-time systems arises when the execution time of one thread depends on the state of a different thread. The main source of timing interactions in a CMP comes from shared caches and shared main memory. In the worst case, all latency hiding has to be ignored by the analysis and the sum of the execution times of several threads is the same as the serial execution on a single-threaded CPU. In addition, multithreaded processors usually share the level 1 caches. Therefore, each thread invalidates the abstract cache state of the other threads.

Dynamic ILP and TLP can be combined for simultaneous multithreading (SMT). With this technique independent threads can be active in the same pipeline stage. This results in a higher utilization of processor resources that are already available for the ILP extraction. Modeling the fine-grained interaction of different SMT threads for WCET analysis seems, at least to the author, an intractable problem.

5.3.7 Chip Multiprocessors

Due to the power wall [26], CMP systems are now state-of-the-art in desktop and server processors. There are three different CMP systems: (1) multicore versions of super-scalar architectures (Intel/AMD), (2) multicore chips with simple RISC processors (Sun Niagara), and (3) the Cell architecture.

Mainstream desktop processors from Intel and AMD include two or four out-of-order executing processors. These processors are replications of the original, complex cores that share a level 2 cache and the memory bus. Cache coherence protocols on the chip keep the level 1 caches coherent and consistent. Furthermore, these cores also support SMT, sometimes also called hyper-threading.

Sun took a completely different approach with their Niagara T1 [34] by abandoning their super-scalar architecture. The T1 contains 8 simple RISC cores, each supporting 4 threads, scheduled round-robin. When a thread stalls due to a cache miss or a load-use dependency, it is skipped in the schedule. The first version of the chip contains a single floating point unit that is shared by all 8 processors. Each core implements a six-stage, single-issue pipeline similar to the original five-stage RISC pipeline. Such a simple pipeline brings WCET analysis back into consideration.

The Cell multiprocessor [28, 32, 33] takes an approach similar to a distributed memory multiprocessor. The Cell contains, beside a PowerPC microprocessor, 8 synergistic processors (SP). The SPs contain 256 KB on-chip memory that is incoherent with the main memory. The PowerPC, the 8 SPs, and the memory interface are connected via a network consisting of four independent rings. Communication between the cores in the network has to be setup explicitly. All memory management, e.g., transfer between SPs or between on-chip memory and main memory, is under program control, resulting in a new programming model. The time-predictable memory access to the on-chip memory and the in-order pipeline of the SPs should be a reasonable target for WCET analysis. The challenge is to include the explicit memory transfers between the cores and the main memory into the analysis.

Intel recently announced a CMP system named Larrabee [63]. Larrabee is intended as a replacement for graphic processing units from other vendors. It is notable that Intel uses several dual-issue, in-order x86 cores. They argue that for some workloads in-order pipelines are more power efficient than out-of-order cores. The design is based on the first Pentium processor, enhanced with multithreading support and vector instructions.

The main source of timing interactions in a CMP comes from the shared level 2 (and probably level 3) cache and the shared main memory. The shared memory provides an easy-to-use programming model at the cost of unpredictable access time to the data. A shared level 2 cache is practically not analyzable due to the inter-thread interference. This is the same issue as with multithreading with a shared level 1 cache.

Cache coherence protocols (bus snooping or directory based) enforce a coherent and consistent view of the main memory. These protocols exchange the cache information between all cores on each memory access and introduce a high variability of the cache access time even when the access is a cache hit.

Yan and Zhang analyze a shared instruction cache on a dual core system that executes two threads [78]. To restrict the set of conflicting cache blocks they introduce the category *always-except one hit* for level 2 cache blocks. Assuming thread A and B, a cache block c is classified as *always-except one hit* for thread A when: c is part of a loop in thread A, c conflicts with a block used by thread B, and the conflicting block in thread B is not part of a loop in thread B. However, the approach has two drawbacks: (1) for n threads/cores several categories (up to $n-1$) need to be introduced; (2) *not in a loop* is not a proper model for real-time threads as these are usually periodic.

The memory arbitration algorithm determines the worst-case access time to the main memory. Any fairness based arbitration is, at least, difficult to integrate into WCET analysis. Priority based arbitration can only guarantee access time for the core with the highest priority, because lower priority cores can be blocked indefinitely.

5.3.8 Documentation

To model the processor for the low-level analysis an accurate documentation of the processor internals is needed. However, this information is often not available or sometimes simply wrong [17]. For actual processors the documentation of the internals is usually not disclosed. Over time, due to reverse engineering and less competition with other processors, more information becomes available. This is probably another reason why WCET analysis is about 10 years behind the processor technology.

5.3.9 Summary

While conventional techniques in designing processor architectures increase average throughput, they are not feasible for real-time systems. The influence of these architectural enhancements is at best hardly WCET analyzable. From a survey of the literature, we found that modeling a new version of a microprocessor and finding all undocumented details is usually worth a full PhD thesis.

We argue that trying to catch up on the analysis side with the growing complexity of modern computer architectures is not feasible. A paradigm shift is necessary. Computer architecture has to be redefined or adapted for real-time systems. Predictable and *analyzable* execution time is of primary importance.

5.4 Time-predictable Architecture

We propose a computer architecture designed especially for real-time applications. We do not want to restrict features only, but we also want to actively add features that enhance performance and are time-predictable.

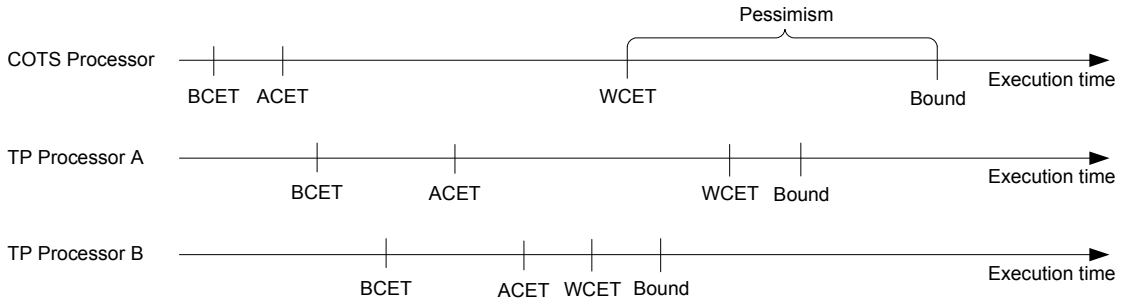


Figure 5.1: Distribution of the best-case, average-case and worst-case execution times and the WCET bound of a task on different architectures. A time-predictable processor has less pessimism and the average-case execution time is not important.

Figure 5.1 illustrates the aim of a time-predictable architecture, showing the distribution of the different execution times for a task: they are best-case execution time (BCET), average-case execution time (ACET), worst-case execution time (WCET), and the bound of the WCET that an analysis tool can provide. The difference between the actual WCET and the bound is caused by the pessimism of the analysis resulting from two factors: (a) certain information, e.g., infeasible execution paths, not being known statically and (b) the simplifications to make the analysis computationally practical. For example, infeasible execution paths may significantly impact the WCET bound, because the static analysis cannot prove that these paths may never be executed. Similarly, dynamic features such as speculative execution and pipelining often need to be modeled conservatively to prevent an explosion of the analysis complexity.

The first time line shows the distribution of the execution times for a commercial off-the-shelf (COTS) processor. The other two time lines show the distribution for two different time-predictable processors.

Variant A depicts a time-predictable processor with a higher BCET, ACET, and WCET than a standard processor. Although the WCET is higher than the WCET of the standard processor, the pessimism of the analysis is lower and the resulting WCET bound is lower as well. Even this type of processor is a better fit for hard real-time systems than today's standard processors.

Processor B shows an architecture where the BCET and ACET are further increased, but the WCET and the WCET bound are decreased. Our goal is to design an architecture with a low WCET bound. For hard real-time systems the likely increase in the ACET and BCET is acceptable, because the complete system needs to be designed to reduce the WCET. It should be noted that a processor designed for low WCET will never be as fast in the average case as a processor optimized for ACET. Those are two different design optimizations. We define a time-predictable processor as "under the assumption that only feasible execution paths are analyzed, a time-predictable processor's WCET bound is close to the real WCET."

In the following we propose time-predictable solutions or replacements, if possible, for the issues we identified in the last section. Table 5.1 summarizes the issues of standard processors for WCET analysis and the proposed architectural solutions.

5.4.1 Pipeline Dependencies

Pipelining is a major architectural feature to speed up program execution. Different stages of an instruction are overlapped and therefore executed in parallel. The theoretical throughput of a scalar

	Standard processor WCET issues	Time-predictable processor
Pipeline	Dependencies and shared state	Simple pipeline
Instruction fetch	Unbounded timing effects	Avoid prefetch queue, use double buffer
Caches	Replacement policy, abstract cache state destruction by unknown addresses	Method cache, stack cache, and a highly associative, small heap cache
Branch prediction	Long history in dynamic predictors	Static branch prediction
Super-scalar architectures	Timing anomalies	Avoid, instead use CMP and/or VLIW
Chip-multithreading	Inter-thread interference (cache, pipeline)	Avoid, instead use CMP
Chip-multiprocessors	Inter-core interference via shared memory, cache	TDMA scheduled memory access

Table 5.1: Architectural issues for WCET analysis of standard processors and proposed architectural solutions.

pipeline is one instruction per clock cycle.

In contrast to Whitham [75], we think that a time-predictable architecture should be pipelined. The pipeline should be simple and dependencies between instructions avoided, or at least minimized, to avoid unbounded timing effects.

5.4.2 Instruction Fetch

To avoid a prefetch queue, with probably unbounded execution-time dependencies over a stream of instructions, a fixed-length instruction set is recommended. Variable length instructions can complicate instruction cache analysis because an instruction can cross a block boundary. The method cache, as proposed in the following section, avoids this issue. Either all instructions of a function, independent of their length, are in the cache, or none of them.

Fetching variable sized instructions from the method cache can be performed in a single cycle. The method cache is split into two interleaved memories banks. Each of the two cache memories needs a read port wide enough for a maximum sized instruction. Accessing both memories concurrently with a clever address calculation overcomes the boundary issue for variable sized instruction access.

5.4.3 Caches

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

For the cache analysis the addresses of the memory accesses need to be predicted. The addresses for the instruction fetch are easy to determine and access to stack allocated data, e.g. function arguments and local variables, is also quite regular. The addresses can be predicted when the call tree is known.

The addresses for heap allocated data are very hard to predict statically – the addresses are only known during runtime (we found no publication that describes analysis of the data cache for heap allocated data). Without knowing the address, a single access influences all sets in the cache.

To avoid corruption of the abstract cache state in the analysis by data accesses, separate instruction and data caches are mandatory [25]. Furthermore, we propose to split the data cache into a cache for stack allocated data and a cache for global or heap allocated data. As stack allocated data is guaranteed thread local, the stack cache can be further simplified for CMP systems.

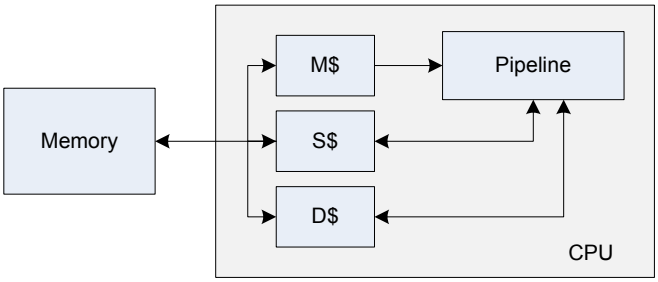


Figure 5.2: Cache configuration for the time-predictable architecture. The method cache M$ caches instructions of a full method/function. The data cache D$ is augmented by a stack cache S$ to avoid cache trashing of stack allocated data with heap allocated data.

For the instruction cache we propose a new form of organization where whole functions are loaded on a miss on call or return. Figure 5.2 shows the proposed organization of the three caches.

The Instruction Cache

We propose a new form of organization for the instruction cache: the *method cache* [57], which has a novel replacement policy. A whole function or method is loaded into the cache on a call or return. This cache fill strategy pools all the cache misses of a function. All instructions except call and return are guaranteed cache hits. Only the call tree needs to be analyzed during the cache analysis. With the proposed cache organization, the cache analysis can be performed independently of the pipeline analysis.

Filling the cache on call and return only removes another source of interference: there is no competition for the main memory access between instruction cache and data cache. In traditional architectures there is a subtle dependency between the instruction cache and memory access for a load or store instruction. For example, a load or store at the end of the processor pipeline competes with an instruction fetch that results in a cache miss. One of the two instructions is stalled for additional cycles by the other instruction. With a data cache, this situation can be even worse. The worst case scenario for the memory stall time for an instruction fetch or a data load is two miss penalties when both cache reads are a miss.

The main restriction of the method cache is that a whole method needs to fit into the cache. For larger methods, software and hardware based options are possible to resolve this issue. The compiler can split large methods into several shorter methods. At the hardware level there are two options for methods that are too large: the cache can be disabled or the method cache can be switched into a direct mapped mode.

If we avoid absolute jumps within a method we can use a relative program counter within the method and place a method at each position within the cache. This property is fulfilled with Java bytecode, but can also be enforced by the compiler for C code.

For a full method load into the cache, we need to know the length of the method. This information is available in the Java class file. For compiled C code this information can be provided in the executable. A simple convention, implemented in the linker, is to store the method length one word before the actual method start. In order to use the method cache in a RISC processor, the ISA is extended with a prefetch instruction to force the cache load. The prefetch instruction can be placed immediately before the call or return instruction. It can also be scheduled earlier to hide the cache load latency.

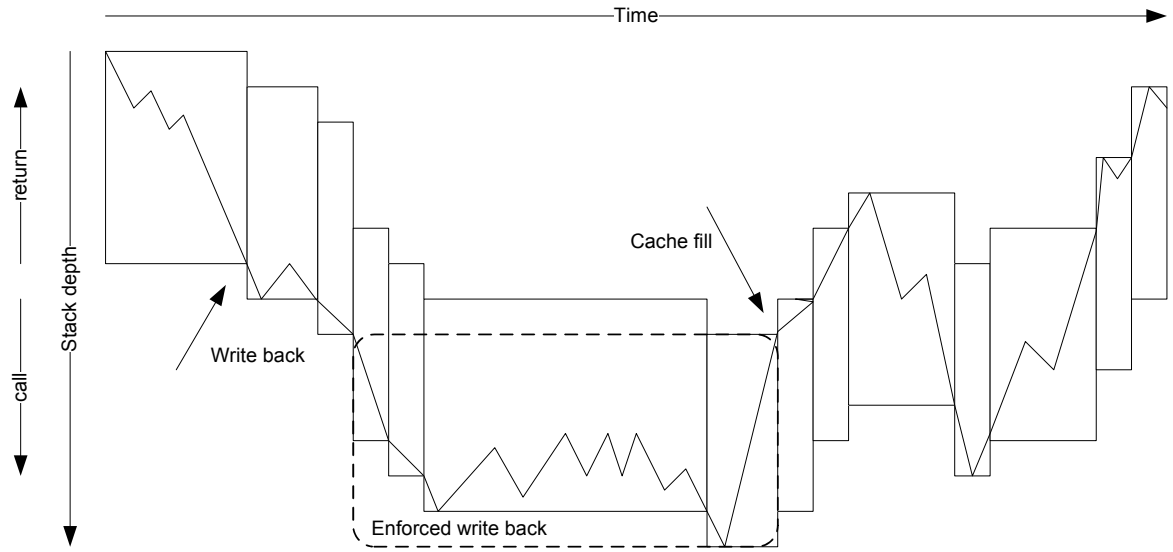


Figure 5.3: Stack usage for call and return and the resulting stack cache window. When the window overflows on a call a write back of old frames is necessary. The stack cache fill is caused by an underflow after a return. Enforcing a write back of the whole stack cache can guarantee hits for subsequent, more deeply nested functions.

The Stack Cache

Access patterns to stack allocated data are different from heap or static allocated data. Addresses into the stack are easy to predict statically because the allocation addresses of stack frames can be predicted by the analysis of the call tree. Furthermore, a new stack frame for a function call does not need to be cache consistent with the main memory. The involved cache blocks need no cache fill from the main memory.

To benefit from these properties for WCET analysis, we propose to split the data cache into a stack cache and a cache for static and heap allocated data (it is possible to further split the data cache into a cache for static data and heap data). The organization of the cache for static and heap allocated data, further referred to as data cache, will be proposed in the following section.

The regular access pattern to the stack cache will not benefit from set associativity. Therefore, the stack cache is a simple direct mapped cache. The stack contains local variables and the write frequency is higher than for other memory areas. The high frequency mandates a write back organization.

A stack cache is similar to a windowed register file as implemented in the Berkeley RISC processor [45]. A stack cache can be organized to exchange data with the main memory on a stack frame basis. When the cache overflows, which happens only during a call, the oldest frame or frames have to be moved to the memory. A frame needs to be loaded from the memory only when a function returns. Exchange with the main memory can be implemented in hardware, microcode, or with compiler visible machine instructions.

If the maximum call depth results in a stack that is smaller than the stack cache, all accesses will be a cache hit. A write back occurs first when the program reaches a call depth resulting in a wrap around within the cache. A cache miss can occur only when the program goes up in the call tree and needs access to a cache block that was evicted by a call down in the call tree.

Figure 5.3 shows the call and return behavior of a program over time and the changing stack cache window. The stack grows downwards in the figure. The dashed box shows a possibility to enforce a write back at some program point. The following stack changes fit into the enforced stack window and no memory transactions are necessary.

On a return, the previously used cache blocks can be marked empty because function local data is not accessible after the return (it could be accessed in C by returning a pointer to the stack data. However, this is undefined and considered poor programming practic). As a result, cache lines will never need to be written back on a cache wrap around after return. The stack cache activity can be summarized in the following way:

- A cache miss can only occur after a return. The first miss is at least *one cache size* away from a leaf in the call tree.
- Cache write back can only occur after a function call. The first write back is *one cache size* away from the root of the call tree.

We can make the misses and write backs more predictable by forcing them to occur at explicit points in the call tree. At these points, the cached stack frames are written back to the main memory and the whole stack cache is marked empty. If we place the flush points at function calls in the call tree that are within *one cache size* from the leaf functions, all cache accesses into that area are guaranteed hits. This algorithm can actually improve WCET because most of the execution time of a program is spent in inner loops further down the call tree.

Stack data is usually not shared between threads and no cache coherence and consistence protocol – the major bottleneck for CMP scaling – needs to be implemented for a CMP system.

The Data Cache

For conservatively written programs with statically allocated data, the address of the data is known after program linking. Value analysis results in a good prediction of read and write addresses. The addresses are the input for the cache analysis. In [20], control tasks, from a real-time benchmark provided by Airbus, were analyzed. For this benchmark 90% of the memory accesses were predicted precisely.

In a modern object oriented language, data is usually allocated on the heap. The address for these objects is only known at runtime. Even when using such a language in a conservative style, where all data is allocated during an initialization phase, it is not easy to predict the resulting addresses. The order of the allocations determines the addresses of the objects. When the order becomes unknown at one point in the initialization phase, the addresses for all following allocations cannot be determined precisely.

It is possible to analyze local cache effects with unknown addresses for an LRU set-associative cache. For an n-way associative cache, the history for n different addresses can be tracked. Because the addresses are unknown, a single access influences *all* sets in the cache. The analysis reduces the effective cache size to a single set.

The local analysis for the LRU based cache is illustrated by a small example with a four-word cache. The example cache allocates a cache block on a write. Table 5.2 shows a code fragment with access to heap allocated data (objects `a`, `b`, `c`, and `d`). The cache state after the load or store instruction is shown in the right section of the table. The leftmost column of the cache state represents the youngest element, the rightmost column the oldest (the LRU element). We assume a 4-way set-associative cache for the example. Therefore, we can locally track four different and *unknown*

		Cache state			
Instruction	Memory	youngest			oldest
a.v = 123;	store a.v	a.v	—	—	—
b.v = 456;	store b.v	b.v	a.v	—	—
c.v = b.v;	load b.v	b.v	a.v	—	—
	store c.v	c.v	b.v	a.v	—
d.v = b.v;	load b.v	b.v	c.v	a.v	—
	store d.v	d.v	b.v	c.v	a.v
b.v = a.v;	load a.v	a.v	d.v	b.v	c.v
	store b.v	b.v	a.v	d.v	c.v

Table 5.2: An example of analyzable accesses to three heap allocated objects with a four-word LRU cache. The cache content after the execution of a statement is depicted in the right section of the table.

addresses. After the first two constant assignments, we know that a.v and b.v are in the cache. The following load of b.v is trivially a hit and the store into c.v changes the cache content and the age of a.v and b.v. All following loads are hits and only change the age ordering of the cache elements. In this small example we dealt with four different and unknown addresses, but could classify all read accesses as hits for a four-word cache.

We propose to implement the cache architecture exactly as it results from this analysis – a small, fully associative cache with an LRU replacement policy. This cache organization is similar to the victim cache [31], which adds associativity to a direct mapped cache. A small, fully associative buffer holds discarded cache blocks. The replacement policy is LRU.

LRU is difficult to calculate in hardware and only possible for very small sets. Replacement of the oldest block gives an approximation of LRU. The resulting FIFO strategy can be used for larger caches. To offset the less predictable behavior of the FIFO replacement [56], the cache has to be much larger than an LRU based cache.

The Scratchpad Memory

A common method for avoiding data caches is an on-chip memory called scratchpad memory, which is under program control. This program managed memory entails a more complicated programming model, although it can be automatically partitioned [4, 71]. A similar approach for time-predictable caching is to lock cache blocks. The control of the cache locking [49] and the allocation of data in the scratchpad memory [72, 65] can be optimized for the WCET. A comparison between locked cache blocks and a scratchpad memory with respect to the WCET can be found in [50].

Exposing the scratchpad memory at the language level can further help to optimize the time-critical path of the application.

5.4.4 Branch Prediction

As the pipelines of current general-purpose processors become longer to support higher clock rates, the penalty of branches also increases. This is compensated by branch prediction logic with branch target buffers. However, the upper bound of the branch execution time is the same as without this feature.

Simple static branch prediction (e.g. backward branches are assumed taken, forward branches not taken) or compiler generated branch predictions are WCET analyzable options. One-level dynamic branch predictors can be analyzed [13]. The branch history table has to be separate from the instruction cache to allow independent modeling for the analysis.

5.4.5 Instruction Level Parallelism

Statically scheduled VLIW processors are an option for a time-predictable architecture. The balance between the VLIW width and the number of cores in a CMP system depends on the application domain. For control oriented applications, we assume that a dual-issue VLIW is a practical architecture. DSP related applications can probably fill more instruction slots with useful instructions.

Dynamically scheduled super-scalar architectures are not considered as an option for a time-predictable architecture. The amount of hardware that is needed to extract ILP from a single thread is better spent on a (VLIW based) CMP system.

5.4.6 Chip Multithreading

Fine-grained multithreading within the pipeline is in principle not an issue for WCET analysis. The scheduling algorithm of the threads needs to be known and must not depend on the state of the threads. Round-robin scheduling is a time-predictable option. The execution time for simple instructions simply increases by a factor equal to the number of threads. The benefit of hiding pipeline stalls due to data dependencies or branches results in a lower factor for these instructions. Execution of n tasks on an n-way multithreading pipeline takes less (predictable) time than executing these tasks serially on a single threaded processor. However, cache misses, even if a single cache miss could be hidden, result in interference between the different threads because the memory interface is a shared resource.

Fine-grained multithreading resolves the data dependencies for a thread within the pipeline: the thread is only active in a single pipeline stage. Therefore, the forwarding network can be completely removed from the processor. This is an important simplification of the pipeline because the forwarding multiplexer is often part of the critical path that restricts the maximum clock frequency.

To avoid cache thrashing, each thread needs – in addition to its own register file – its own instruction and data cache, which reduces the effectively shared transistors to the pipeline itself. We think that the cost is too high for the small performance enhancement. Therefore, also duplicating the pipeline – resulting in a CMP solution – will result in a better performance/cost factor.

SMT is not an option as the interaction between the threads is too complex to model.

5.4.7 Chip Multiprocessors

Embedded applications need to control and interact with the *real* world, a task that is inherently parallel. Therefore, these systems are good candidates for CMPs. We argue that the transistors required to implement super-scalar architectures are better used on complete replication of simple cores.

CMP systems share the access bandwidth to the main memory. To build a time-predictable CMP system, we need to schedule the access to the main memory in a predictable way. A predictable scheduling can only be time based, where each core receives a fixed time slice. This scheduling scheme is called time division multiple access (TDMA). The time slices do not need to be of equal

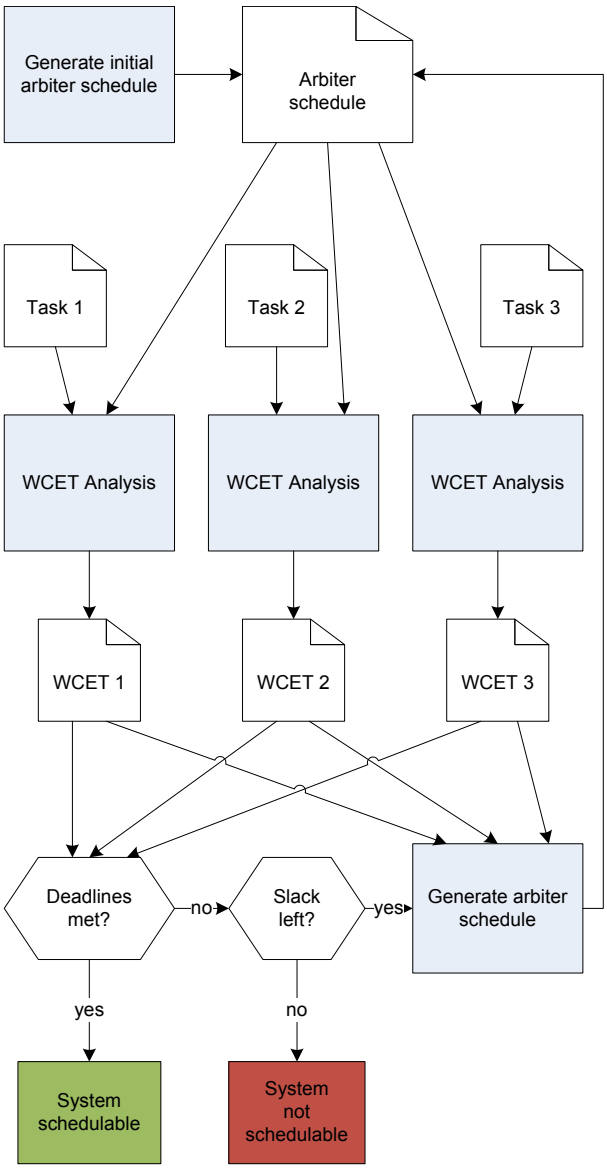


Figure 5.4: Tool flow for a CMP based real-time system with one task per core and a static arbiter schedule. If the deadlines are not met, the arbiter schedule is adapted according to the WCETs and deadlines of the tasks. After the update of the arbiter schedule the WCET of all tasks needs to be recalculated.

size. The execution time of un-cached loads and stores and the cache miss penalty depend on this schedule and therefore, for accurate WCET analysis, the complete schedule needs to be known.

Assuming that enough cores are available, we propose a CMP model with a single thread per processor. In that case thread switching and schedulability analysis for each individual core disappears. Since each processor executes only a single thread, the WCET of that thread can be as long as its deadline. When the period of a thread is equal to its deadline, 100% utilization of that core is feasible. For threads that have enough slack time left, we can increase the WCET by decreasing their share of the bandwidth on the memory bus. Other threads with tighter deadlines can, in turn, use the freed bandwidth and run faster. The usage of the shared resource main memory is adjusted by the TDMA schedule. The TDMA schedule itself is the input for WCET analysis for all threads. Finding a TDMA schedule, where all tasks meet their deadlines, is thus an iterative optimization problem.

Figure 5.4 shows the analysis tool flow for the proposed time-predictable CMP with three tasks. First, an initial arbiter schedule is generated, e.g., one with equal time slices. That schedule and the tasks are the input of WCET analysis performed for each task individually. If all tasks meet their deadline with the resulting WCETs, the system is schedulable. If some tasks do not meet their deadline and other tasks have some slack time available, the arbiter scheduler is adapted accordingly. WCET analysis is repeated, with the new arbiter schedule, until all tasks meet their deadlines or no slack time for an adaption of the arbiter schedule is available. In the latter case no schedule for the system is found.

5.4.8 Documentation

The hardware description language VHDL was originally developed to document the behavior of digital circuits. Today digital hardware can be synthesized from a VHDL description. Therefore, the VHDL code for the processor is the ideal form of documentation. VHDL code can also be simulated and all interactions between different components are observable.

An open-source design enables the WCET tool provider to check the real processor when the documentation is missing; documentation errors are also easier to find. Sun provides the Verilog source of their Niagra T1 [34] as open-source under the GNU GPL (`http://www.opensparc.net/opensparc-t1/downloads.html`).

5.5 Evaluation

In this section, we evaluate some of the proposed time-predictable architectural features with JOP [61], an implementation of a Java processor. We have chosen to natively support Java as it is the language which will be used for future safety critical systems [73, 30]. Java's intermediate representation, the Java class file, is analysis friendly and the type information can be reconstructed from the class file. Executing bytecodes – the instruction set of the Java virtual machine (JVM) – directly in the hardware allows WCET analysis at the bytecode level. The translation step from bytecode to machine code, which introduces timing inaccuracies, can be avoided.

5.5.1 The Java Processor JOP

The major design goal of JOP is the time-predictable execution of Java bytecodes [59]. All functional units, and especially the interactions between them, are carefully designed to avoid any timing dependency between bytecodes.

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

Table 5.3: Direct-mapped cache, average memory access time

JOP dynamically translates the Java bytecodes to a stack based microcode that can be executed in a three-stage pipeline. The translation takes exactly one cycle per bytecode. Compared to other forms of dynamic code translation, the scheme used in JOP does not add any variable latency to the execution time and is therefore time-predictable.

JOP contains a simple execution stage with the two topmost stack elements as discrete registers. No write back stage or forwarding logic is needed. The short pipeline (four stages) results in short conditional branch delays; a difficult to analyze branch prediction logic or a branch target buffer can be avoided.

All microcode instructions have a constant execution time of one cycle. No stalls are possible in the microcode pipeline. Loads and stores of object fields are handled explicitly. The absence of timing dependencies between bytecodes results in a simple processor model for the low-level WCET analysis.

The proposed architecture is open-source and all design files are available (`http://www.jopdesign.com/`). The instruction timing of the bytecodes is documented.

Method Cache

JOP contains the proposed method cache. The default configuration is 4 KB, divided into 16 blocks of 256 Bytes. The replacement strategy is FIFO.

WCET analysis of the method cache and of standard instruction caches is currently under development. Therefore, we perform only average case measurements for a comparison between a time-predictable cache organization and a standard cache organization. With a simulation of JOP, we measure the cache misses and miss penalties for different configurations of the method cache and a direct-mapped cache. The miss penalty and the resulting effect on the execution time depend on the main memory system. Therefore, we simulate three different memory technologies: static RAM (SRAM), synchronous DRAM (SDRAM), and double data rate (DDR) SDRAM. For the SRAM, a latency of 1 clock cycle and an access time of 2 clock cycles per 32-bit word are assumed. For the SDRAM, a latency of 5 cycles (3 cycles for the row address and 2 cycles for the CAS latency) is assumed. The SDRAM delivers one word (4 bytes) per cycle. The DDR SDRAM has a shorter latency of 4.5 cycles and transfers data on both the rising and falling edge of the clock signal.

The resulting miss cycles are scaled to the bandwidth consumed by the instruction fetch unit. The result is the number of cache fill cycles per fetched instruction byte. In other words: the average main memory access time in cycles per instruction byte. A value of 0.1 means that for every 10 fetched instruction bytes, one clock cycle is spent to fill the cache.

Cache size	Block size	SRAM	SDRAM	DDR
1 KB	16 B	0.36	0.21	0.12
1 KB	32 B	0.36	0.21	0.12
1 KB	64 B	0.36	0.22	0.12
1 KB	128 B	0.41	0.24	0.14
2 KB	32 B	0.06	0.04	0.02
2 KB	64 B	0.12	0.08	0.04
2 KB	128 B	0.19	0.11	0.06
2 KB	256 B	0.37	0.22	0.13

Table 5.4: Method cache, average memory access time

Table 5.3 shows the result for different configurations of a direct-mapped cache. For the evaluation we used an adapted version of the real-time application Kfl (the Kfl benchmark is also used in Section 5.5.4), which is a node in a distributed control application. As the embedded application is quite small (1366 LOC), we simulated small instruction caches. Which configuration performs best depends on the relationship between memory bandwidth and memory latency. The data in bold emphasize the best block size for the different memory technologies. As expected, memories with a higher latency and bandwidth perform better with larger block sizes. For small block sizes, the latency clearly dominates the access time. Although the SRAM has half the bandwidth of the SDRAM and a quarter of the DDR SDRAM, it is faster than the SDRAM memories with a block size of 8 byte. In most cases a block size of 16 bytes is fastest.

Table 5.4 shows the average memory access time per instruction byte for the method cache. Because we load full methods, we have chosen larger block sizes than for a standard cache. All configurations benefit from a memory system with a higher bandwidth. The method cache is less latency sensitive than the direct-mapped instruction cache. For the small 1 KB cache the access time is almost independent of the block size. The capacity misses dominate. From the 2 KB configuration we see that smaller block sizes result in less cache misses. However, smaller block sizes result in more hardware for the hit detection since the method cache is in effect fully associative. Therefore, we need a balance between the number of blocks and the performance.

The cache conflict is high for the small configuration with 1 KB cache. The direct-mapped cache, backed up with a low-latency main memory, performs better than the method cache. When high-latency memories are used, the method cache performs better than the direct mapped cache. This is expected as the long latency for a transfer is amortized when more data (the whole method) is filled in one request.

A small block size of 32 Bytes is needed in the 2 KB method cache to outperform the direct mapped cache with the low-latency main memory as represented by the SRAM. For higher latency memories (SDRAM and DDR), a method cache with a block size of 128 bytes outperforms the direct mapped instruction cache.

The comparison does not show if the method cache is more easily predictable than other cache solutions. It shows that caching full methods performs similarly to standard caching techniques.

Instruction	Cycles	Funtion
iconst_0	1	load constant 0 on TOS
bipush	2	load a byte constant on TOS
iload_0	1	load local variable 0 on TOS
iload	2	load a local variable on TOS
dup	1	duplicate TOS
iadd	1	integer addition
isub	1	integer subtraction
ifeq	4	conditional branch

Table 5.5: Execution time of simple bytecodes in cycles

Stack Cache

In JOP a simplified version of the proposed stack cache is implemented. The JVM uses the stack not only for the activation frame and for local variables, but also for operands. Therefore, the two top elements of the stack are implemented as registers [58]. With this configuration we can avoid the write-back pipeline stage.

The fill and spill between the stack cache and the main memory is simplified. The cache content is exchanged only on a thread switch. Therefore, the maximum call depth is restricted by the on-chip cache size. In a future version of JOP, we intend to relax this limitation. The cache fill will be performed on a return and the write back on invoke when necessary. A stack analysis tool will add a marker to the methods where a full cache write back shall be performed and the stack access in methods deeper in the call tree will be guaranteed hits. Heap allocated data and static fields are not cached in the current implementation of JOP.

Branch Prediction

In JOP, branch prediction is avoided. This results in pressure on the pipeline length. The micro-programmed core processor has a pipeline length of as little as three stages resulting in a branch execution time of three cycles in microcode. The two slots in the branch delay can be filled with instructions or *nop*. With the additional bytecode fetch and translation stage, the overall pipeline is four stages and results in a four cycle execution time for a bytecode branch.

5.5.2 WCET Analysis

Bytecode instructions that do not access memory have a constant execution time. Most simple bytecodes are executed in a single cycle. Table 5.5 shows example instructions and their timing. The access time to object, array, and class fields depends on the timing of the main memory. With a memory with r_{ws} wait states for a read access the execution time for, e.g. getfield is

$$t_{getfield} = 11 + 2r_{ws}$$

To demonstrate that JOP is amenable to WCET analysis, we have built an IPET based WCET analyzer [62]. While loop bounds are annotated at the source level, the analysis is performed at the bytecode level. Without dependencies between bytecodes, the pipeline analysis can be omitted. The execution time of basic blocks is calculated simply by adding the execution time of individual

bytecodes. For the method cache we have implemented a simplified analysis where only leaf nodes in the call tree are considered. A return from such a leaf node is a guaranteed hit. (The maximum method size is restricted to half of the cache size.) Invocation of a leaf node in a tight loop (without invocations of other methods) is classified as a miss for the first iteration and a hit for the following iterations. For small benchmarks the overestimation of the WCET is around 5%. For two real applications (Lift and Kfl) the analysis resulted in an overestimation of 56% and 116%. It should be noted that the overestimation is calculated by comparison with measurement based WCET estimation, which is not a safe approach.

Another indication that JOP is a WCET *friendly* design is that other real-time analysis projects use JOP as the primary target platform. Harmon has developed a tree based WCET analyzer for interactive back-annotation of WCET estimates into the program source [22]. Bogholm et al. have developed an integrated WCET and scheduling analysis tool based on model checking [10].

5.5.3 Comparison with picoJava

We compare the time-predictable JOP design with picoJava [66, 67], a Java processor designed for average case performance. Simple bytecodes are directly supported by the processor. Most of them execute in a single cycle. More complex bytecodes trap to a software routine. However, the invocation time of the trap depends on the cache state and is between 6 cycles in the best case and 426 cycles in the worst case – a factor in the order of two magnitudes. Some of the trapped instructions (e.g., invokevirtual) can be replaced at runtime by a *quick* version (e.g., invokevirtual_quick). This replacement results in different execution times for the first execution of some code and following executions.

To speedup sequences of stack operations, picoJava can fold several instructions into a RISC style three register operation, e.g., the sequence: load, load, add, store. This feature compensates for the inefficiency of a stack machine. However, the folding unit depends on a 16 byte instruction buffer with all the resulting unbounded timing effects of a prefetch queue.

picoJava implements a 64 word stack buffer as discrete registers. Spill and fill of that stack buffer is performed in background by the hardware. Therefore, the stack buffer closely interacts with the data cache. The interference between the folding unit, the instruction buffer, the instruction cache, the stack buffer, the data cache, and the memory interface causes complications in modeling picoJava for WCET analysis.

picoJava is about 8 times larger than JOP and can be clocked at less than half of the frequency of JOP in the same technology [51]. Therefore, the small size of a time-predictable architecture naturally leads to a CMP system.

5.5.4 Performance

One important question remains: is a time-predictable processor slow? We evaluate the average case performance of JOP by comparing it with other embedded Java systems: Java processors from industry and academia and two just-in-time (JIT) compiler based systems. For the comparison we use JavaBenchEmbedded, (available at `http://www.jopwiki.com/JavaBenchEmbedded`) a set of open-source Java benchmarks for embedded systems. Kfl and Lift are two real-world applications adapted with a simulation of the environment to run as stand-alone benchmarks. Udplp is a simple client/server test program that uses a TCP/IP stack written in Java.

	Kfl	UdpIp	Lift
Cjip	176	91	
jamuth	3400	1500	
EJC	9893	2882	
SHAP	11570	5764	12226
aJ100	14148	6415	
JOP	18275	8467	18649
picoJava	23813	11950	25444
CACAO/YARI	39742	17702	38437

Table 5.6: Application benchmark performance on different Java systems. The table shows the benchmark results in iterations per second – a higher value means higher performance.

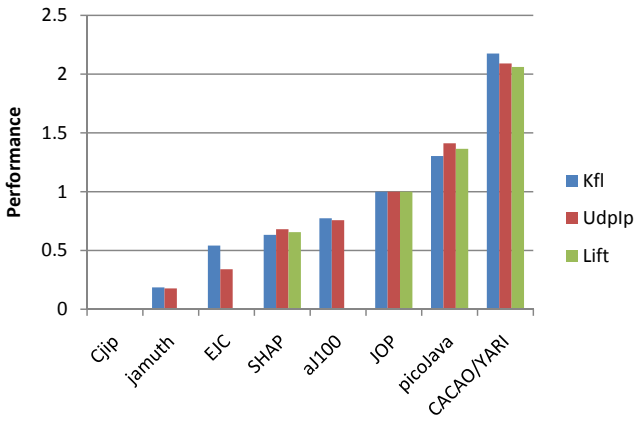


Figure 5.5: Performance comparison of different Java systems with embedded application benchmarks. The results are scaled to the performance of JOP

Table 5.6 shows the raw data of the performance measurements of different embedded Java systems for the three benchmarks. The numbers are iterations per second whereby a higher value represents better performance. Figure 5.5 shows the results scaled to the performance of JOP.

The numbers for JOP are taken from an implementation in the Altera Cyclone FPGA [2], running at 100 MHz. JOP is configured with a 4 KB method cache and a 1 KB stack cache.

Cjip [29] and aJ100 [1] are commercial Java processors, which are implemented in an ASIC and clocked at 80 and 100 MHz, respectively. Both cores do not cache instructions. The aj100 contains a 32 KB on-chip stack memory. jamuth [70] and SHAP [79] are Java processors that are implemented in an FPGA. jamuth is the commercial version of the Java processor Komodo [36], a research project for real-time chip multithreading. jamuth is configured with a 4 KB direct-mapped instruction cache for the measurements. The architecture of SHAP is based on JOP and enhanced with a hardware object manager. SHAP also implements the method cache [48]. The benchmark results for SHAP are taken from the SHAP website (`http://shap.inf.tu-dresden.de/`, accessed July, 2008); SHAP is configured with a 2 KB method cache and 2 KB stack cache.

picoJava [44] is a Java processor developed by Sun. picoJava is no longer produced and the second

Soft-Core	Logic Cells	Memory	Frequency
JOP	3,300	7.6 KB	100 MHz
YARI	6,668	18.9 KB	75 MHz
picoJava	27,560	47.6 KB	40 MHz

Table 5.7: Resource consumption and maximum operating frequency of JOP, YARI, and picoJava.

version (picoJava-II) was available as open-source Verilog code. Puffitsch implemented picoJava-II in an FPGA (Altera Cyclone-II) and the performance numbers are obtained from that implementation [51]. picoJava is configured with a direct-mapped instruction cache and a 2-way set-associative data cache. Both caches are 16 KB.

EJC [16] is an example of a JIT system on a RISC processor (32-bit ARM720T at 74 MHz). The ARM720T contains an 8 KB unified cache. To compare JOP with a JIT based system in exactly the same hardware we use the research JVM CACAO [35] on top of the MIPS compatible soft-core YARI [11]. YARI is configured with a 4-way set-associative instruction cache and a 4-way set-associative write-through data cache. Both caches are 8 KB.

The measurements do not provide a clear answer to the question of whether a time-predictable architecture is slow. JOP is about 33% faster than the commercial Java processor aJ100. However, picoJava is 36% faster than JOP and the JIT/RISC combination is about 111% faster than JOP. (The numbers of CACAO/YARI are from [11]. In the mean time YARI has been enhanced and outperforms JOP by a factor of 2.8.) We conclude that a time-predictable solution will never be as fast in the average case as a solution optimized for the average case.

5.5.5 Hardware Area and Clock Frequency

Table 5.7 compares the resource consumption and maximum clock frequency of a time-predictable processor (JOP), a standard MIPS architecture (YARI), and a complex Java processor (picoJava), when implemented in the same FPGA. The streamlined architecture of JOP results in a small design: JOP is half the size of the MIPS core YARI, and compared to picoJava consumes about 12% of the resources. JOP's size allows implementing a CMP version of JOP even in a low-cost FPGA. The simple pipeline of JOP achieves the highest clock frequency of the three designs. From the frequency comparison we can estimate that the maximum clock frequency of JOP in an ASIC will also be higher than a standard RISC pipeline in an ASIC.

5.5.6 JOP CMP System

We have implemented a CMP version of JOP with a fairness based arbiter [47]. All cores are allotted an equal share of the memory bandwidth. Each core has its own method cache and stack cache. Heap allocated data is not cached in this design.

When comparing a JOP CMP system against the complex Java processor picoJava, a dual core version of JOP is about 5% slower than a single picoJava core, but consumes only 22% of the chip resources. With four cores, JOP outperforms picoJava by 30% with size of 43% of picoJava.

A configurable TDMA arbiter for a time-predictable CMP system and the integration of the arbitration schedule into the WCET tool [62] is presented in [46].

5.5.7 Summary

A model of a processor with accurate timing information is essential for tight WCET analysis. The architecture of JOP and the microcode are designed with this in mind. Execution time of bytecodes is known cycle accurately [59]. It is possible to analyze the WCET at the bytecode level [9] without the uncertainties of an interpreting JVM [6] or generated native code from ahead-of-time compilers for Java.

5.6 Conclusion

In this paper, we argue for a time-predictable computer architecture for embedded real-time systems that supports WCET analysis. We have identified the problematic micro-architecture features of standard processors and provided alternative solutions when possible.

Dynamic features, which model a large execution history, are problematic for WCET analysis. Especially interferences between different features result in a state space explosion for the analysis. The proposed architecture is an in-order pipeline with minimized instruction dependencies. The cache memory consists of a method cache containing whole methods and a data cache that is split for stack allocated data and heap allocated data. The pipeline can be extended to a dual-issue pipeline when the instructions are compiler scheduled. For further performance enhancements, we propose a CMP system with time sliced arbitration of the main memory access. Running each task on its own core in a CMP system eliminates scheduling, and the related cache thrashing, from the analysis. The schedule of the memory access becomes an input for WCET analysis. With non-uniform time slices, the arbiter schedule can be adapted to balance the utilization of the individual cores.

The concept of the proposed architecture is evaluated by a real-time Java processor, called JOP. We have presented a brief overview of the architecture. A simple four-stage pipeline and microcoded implementation of JVM bytecodes result in a time-predictable architecture. The proposed method and stack caches are implemented in JOP. The resulting design makes JOP an easy target for the low-level WCET analysis of Java applications.

We compared JOP against several embedded Java systems. The result shows that a time-predictable computer architecture does not need to be slow. A streamlined, time-predictable processor design is quite small. Therefore, we can regain performance by the exploration of thread level parallelism in embedded applications with a replication of the processor in a CMP architecture.

The proposed processor has been used with success to implement several commercial real-time applications [60]. JOP is open-source under the GNU GPL and all design files and the documentation are available at `http://www.jopdesign.com/`.

We plan to implement some of the suggested architectural enhancements in a RISC based system in the future. We will implement the proposed stack cache and the method cache in YARI [11], an open-source, MIPS ISA compatible RISC implementation in an FPGA.

A scratchpad memory for JOP is implemented and the integration into the programming model is under investigation. We will add a small fully associative data cache to JOP. This cache will also serve as a buffer for a real-time transactional memory for the JOP CMP system. We will investigate whether a standard cache for static data is a practical solution for Java.

Acknowledgement

The author thanks Wolfgang Puffitsch and Florian Brandner for the productive discussions on the topic and suggestions for improvements of the paper.

Bibliography

[1] aJile. aj-100 real-time low power Java processor. preliminary data sheet, 2000.

[2] Altera. Cyclone FPGA Family Data Sheet, ver. 1.2, April 2003.

[3] Aravindh Anantaraman, Kiran Seth, Kaustubh Patil, E. Rotenberg, and F. Mueller. Virtual simple architecture (visa): exceeding the complexity limit in safe real-time systems. In *Computer Architecture, 2003. Proceedings. 30th Annual International Symposium on*, volume 31, 2 of *Computer Architecture News*, pages 350–361, New York, June 9–11 2003. ACM Press.

[4] Federico Angiolini, Luca Benini, and Alberto Caprara. Polynomial-time algorithm for on-chip scratchpad memory partitioning. In *Proceedings of the International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES-03)*, pages 318–326, New York, October 30 November 01 2003. ACM Press.

[5] Robert Arnold, Frank Mueller, David Whalley, and Marion Harmon. Bounding worst-case instruction cache performance. In *IEEE Real-Time Systems Symposium*, pages 172–181, 1994.

[6] Iain Bate, Guillem Bernat, Greg Murphy, and Peter Puschner. Low-level analysis of a portable Java byte code WCET analysis framework. In *Proc. 7th International Conference on Real-Time Computing Systems and Applications*, pages 39–48, Dec. 2000.

[7] Iain Bate, Philippa Conmy, Tim Kelly, and John A. McDermid. Use of modern processors in safety-critical applications. *The Computer Journal*, 44(6):531–543, 2001.

[8] Christoph Berg, Jakob Engblom, and Reinhard Wilhelm. Requirements for and design of a processor with predictable timing. In Lothar Thiele and Reinhard Wilhelm, editors, *Perspectives Workshop: Design of Systems with Predictable Behaviour*, number 03471 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2004. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

[9] Guillem Bernat, Alan Burns, and Andy Wellings. Portable worst-case execution time analysis using Java byte code. In *Proc. 12th EUROMICRO Conference on Real-time Systems*, Jun 2000.

[10] Thomas Bogholm, Henrik Kragh-Hansen, Petur Olsen, Bent Thomsen, and Kim G. Larsen. Model-based schedulability analysis of safety critical hard real-time java programs. In *Proceedings of the 6th international workshop on Java technologies for real-time and embedded systems (JTRES 2008)*, pages 106–114, New York, NY, USA, 2008. ACM.

[11] Florian Brandner, Tommy Thorn, and Martin Schoeberl. Embedded JIT compilation with CACAO on YARI. Technical Report RR 35/2008, Institute of Computer Engineering, Vienna University of Technology, Austria, June 2008.

[12] José V. Busquets-Mataix, Juan José Serrano, Rafael Ors, Pedro J. Gil, and Andy J. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *IEEE Real-Time Technology and Applications Symposium (RTAS '96)*, pages 204–213, Washington - Brussels - Tokyo, June 1996. IEEE Computer Society Press.

[13] Antoine Colin and Isabelle Puaut. Worst case execution time analysis for a processor with branch prediction. *Real-Time Systems*, 18(2/3):249–274, 2000.

[14] Martin Delvai, Wolfgang Huber, Peter Puschner, and Andreas Steininger. Processor support for temporal predictability – the SPEAR design example. In *Proceedings of the 15th Euromicro International Conference on Real-Time Systems*, Jul. 2003.

[15] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *DAC '07: Proceedings of the 44th annual conference on Design automation*, pages 264–265, New York, NY, USA, 2007. ACM.

[16] EJC. The ejc (embedded java controller) platform. Available at http://www.embedded-web.com/index.html.

[17] Jakob Engblom. *Processor Pipelines and Static Worst-Case Execution Time Analysis*. PhD thesis, Uppsala University, 2002.

[18] Jakob Engblom. Analysis of the execution time unpredictability caused by dynamic branch prediction. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 152–159. IEEE Computer Society, 2003.

[19] Jakob Engblom, Andreas Ermedahl, Mikael Södin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer (STTT)*, V4(4):437–455, August 2003.

[20] Christian Ferdinand, Reinhold Heckmann, Marc Langenbach, Florian Martin, Michael Schmidt, Henrik Theiling, Stephan Thesing, and Reinhard Wilhelm. Reliable and precise WCET determination for a real-life processor. In Thomas A. Henzinger and Christoph M. Kirsch, editors, *EMSOFT*, volume 2211 of *Lecture Notes in Computer Science*, pages 469–485. Springer, 2001.

[21] Jan Gustafsson. *Analyzing Execution-Time of Object-Oriented Programs Using Abstract Interpretation*. PhD thesis, Uppsala University, 2000.

[22] Trevor Harmon and Raymond Klefstad. Interactive back-annotation of worst-case execution time analysis for java microprocessors. In *Proceedings of the Thirteenth IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, August 2007.

[23] Christopher A. Healy, Robert D. Arnold, Frank Mueller, David B. Whalley, and Marion G. Harmon. Bounding pipeline and instruction cache performance. *IEEE Trans. Computers*, 48(1):53–70, 1999.

[24] Christopher A. Healy, David B. Whalley, and Marion G. Harmon. Integrating the timing analysis of pipelining and instruction caching. In *IEEE Real-Time Systems Symposium*, pages 288–297, 1995.

[25] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, Jul. 2003.

[26] John Hennessy and David Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[27] John L. Hennessy. VLSI processor architecture. *Computers, IEEE Transactions on*, C-33(12):1221–1246, Dec. 1984.

[28] H. Peter Hofstee. Power efficient processor architecture and the cell processor. In *Proceedings of the 11th International Conference on High-Performance Computer Architecture (HPCA-11 2005)*, pages 258–262, 2005.

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

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

[31] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, May 1990.

[32] James A. Kahle, Michael N. Day, H. Peter Hofstee, Charles R. Johns, Theodore R. Maeurer, and David J. Shippy. Introduction to the Cell multiprocessor. *j-IBM-JRD*, 49(4/5):589–604, 2005.

[33] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *Micro, IEEE*, 26:10–25, 2006.

[34] Poonacha Kongetira, Kathirgamar Aingaran, and Kunle Olukotun. Niagara: A 32-way multithreaded sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

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

[36] Jochen Kreuzinger, Uwe Brinkschulte, Matthias Pfeffer, Sascha Uhrig, and Theo Ungerer. Real-time event-handling and scheduling on a multithreaded Java microcontroller. *Microprocessors and Microsystems*, 27(1):19–31, 2003.

[37] Chang-Gun Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6):700–713, 1998.

[38] Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. Modeling out-of-order processors for WCET analysis. *Real-Time Systems*, V34(3):195–227, November 2006.

[39] Yau-Tsun Steven Li and Sharad Malik. Performance analysis of embedded software using implicit path enumeration. In *LCTES '95: Proceedings of the ACM SIGPLAN 1995 workshop on languages, compilers, & tools for real-time systems*, pages 88–98, New York, NY, USA, 1995. ACM Press.

[40] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. Predictable programming on a precision timed architecture. In Erik R. Altman, editor, *Proceedings of the International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2008)*, pages 137–146, Atlanta, GA, USA, October 2008. ACM.

[41] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong-Sang Kim. An accurate worst case timing analysis for RISC processors. *IEEE Transactions on Software Engineering*, 21(7):593–604, July 1995.

[42] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium (RTSS 1999)*, pages 12–21, Washington, DC, USA, 1999. IEEE Computer Society.

[43] Kelvin D. Nilsen and Bernt Rygg. Worst-case execution time analysis on modern processors. *SIGPLAN Not.*, 30(11):20–30, 1995.

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

[45] David A. Patterson. Reduced instruction set computers. *Commun. ACM*, 28(1):8–21, 1985.

[46] Christof Pitter. Time-predictable memory arbitration for a Java chip-multiprocessor. In *Proceedings of the 6th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2008)*, 2008.

[47] Christof Pitter and Martin Schoeberl. Performance evaluation of a Java chip-multiprocessor. In *Proceedings of the 3rd IEEE Symposium on Industrial Embedded Systems (SIES 2008)*, Jun. 2008.

[48] Thomas B. Preusser, Martin Zabel, and Rainer G. Spallek. Bump-pointer method caching for embedded java processors. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems (JTRES 2007)*, pages 206–210, New York, NY, USA, 2007. ACM.

[49] Isabelle Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 217–226, Washington, DC, USA, 2006. IEEE Computer Society.

[50] Isabelle Puaut and Christophe Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proceedings of the conference on Design, Automation and Test in Europe (DATE 2007)*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

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

[52] Peter Puschner and Alan Burns. A review of worst-case execution-time analysis (editorial). *Real-Time Systems*, 18(2/3):115–128, 2000.

[53] Peter Puschner and Alan Burns. Writing temporally predictable code. In *Proceedings of the The Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2002)*, pages 85–94, Washington, DC, USA, 2002. IEEE Computer Society.

[54] Peter Puschner and Anton Schedl. Computing maximum task execution times – a graph-based approach. *Journal of Real-Time Systems*, 13(1):67–91, Jul. 1997.

[55] Peter Puschner and Martin Schoeberl. On composable system timing, task timing, and WCET analysis. In *Proceedings of the 8th International Workshop on Worst-Case Execution Time (WCET) Analysis*, Prague, Czech Republic, July 2008.

[56] Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Journal of Real-Time Systems*, 37(2):99–122, Nov. 2007.

[57] Martin Schoeberl. A time predictable instruction cache for a Java processor. In *On the Move to Meaningful Internet Systems 2004: Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES 2004)*, volume 3292 of *LNCS*, pages 371–382, Agia Napa, Cyprus, October 2004. Springer.

[58] Martin Schoeberl. Design and implementation of an efficient stack machine. In *Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005)*, Denver, Colorado, USA, April 2005. IEEE.

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

[60] Martin Schoeberl. Application experiences with a real-time Java processor. In *Proceedings of the 17th IFAC World Congress*, Seoul, Korea, July 2008.

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

[62] Martin Schoeberl and Rasmus Pedersen. WCET analysis for a Java processor. In *Proceedings of the 4th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2006)*, pages 202–211, New York, NY, USA, 2006. ACM Press.

[63] Larry Seiler, Doug Carmean, Eric Sprangle, Tom Forsyth, Michael Abrash, Pradeep Dubey, Stephen Junkins, Adam Lake, Jeremy Sugerman, Robert Cavin, Roger Espasa, Ed Grochowski, Toni Juan, and Pat Hanrahan. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27(3):1–15, 2008.

[64] Alan C. Shaw. Reasoning about time in higher-level language software. *IEEE Trans. Softw. Eng.*, 15(7):875–889, 1989.

[65] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen. WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium (RTSS)*, pages 223–232. IEEE Computer Society, 2005.

[66] Sun. *picoJava-II Microarchitecture Guide*. Sun Microsystems, March 1999.

[67] Sun. *picoJava-II Programmer's Reference Manual*. Sun Microsystems, March 1999.

[68] Stephan Thesing. *Safe and Precise Worst-Case ExecutionTime Prediction by Abstract Interpretation of Pipeline Models*. PhD thesis, University of Saarland, 2004.

[69] Lothar Thiele and Reinhard Wilhelm. Design for timing predictability. *Real-Time Systems*, 28(2-3):157–177, 2004.

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

[71] Manish Verma and Peter Marwedel. Overlay techniques for scratchpad memories in low power embedded processors. *IEEE Trans. VLSI Syst*, 14(8):802–815, 2006.

[72] Lars Wehmeyer and Peter Marwedel. Influence of memory hierarchies on predictability for time constrained embedded software. In *Proceedings of Design, Automation and Test in Europe (DATE2005).*, pages 600–605 Vol. 1, March 2005.

[73] Andy Wellings. Is Java augmented with the RTSJ a better real-time systems implementation technology than Ada 95? *Ada Lett.*, XXIII(4):16–21, 2003.

[74] Ingomar Wenzel, Raimund Kirner, Peter Puschner, and Bernhard Rieder. Principles of timing anomalies in superscalar processors. In *Proceedings of the Fifth International Conference on Quality Software (QSIC2005)*, pages 295–306. IEEE Computer Society, 2005.

[75] Jack Whitham. *Real-time Processor Architectures for Worst Case Execution Time Reduction*. PhD thesis, University of York, 2008.

[76] Jack Whitham and Neil Audsley. Using trace scratchpads to reduce execution times in predictable real-time architectures. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, pages 305–316, April 2008.

[77] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution time problem – overview of methods and survey of tools. *Trans. on Embedded Computing Sys.*, 7(3):1–53, 2008.

[78] Jun Yan and Wei Zhang. WCET analysis for multi-core processors with shared L2 instruction caches. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium (RTAS 2008)*, pages 80–89, April 2008.

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

[80] N. Zhang, Alan Burns, and Mark Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, 1993.