Performance Evaluations and Chip-Space Requirements of a Multithreaded Java Microcontroller

J. Kreuzinger, R. Zulauf, A. Schulz
Th. Ungerer, M. Pfeffer
Institute for Computer Design
and Fault Tolerance
University of Karlsruhe, Germany

U. Brinkschulte, C. Krakowski
Institute for Process Control
Automation and Robotics
University of Karlsruhe, Germany

Abstract

This paper introduces a multithreaded Java processor kernel which is specifically designed as core processor of a microcontroller or system-on-a-chip. Handling of external real-time events is performed through multithreading. Real-time Java threads are used as interrupt service threads (ISTs) instead of interrupt service routines (ISRs). Our proposed Komodo microcontroller supports multiple ISTs with zero-cycle context switching overhead. Simulation results show a performance gain by multithreading of about 1.4. A VHDL-based hardware design aimed at a Xilinx FPGA yields chip-space requirements of about 55000 gates for a four-threaded processor kernel.

1. Introduction

The wide-spread market of embedded real-time systems prefers microcontrollers over general-purpose processors. Real-time event handling, rapid context switching, and small memory requirements are essential goals in microcontroller design. State-of-the-art embedded systems are programmed in assembly or C/C++ languages. Java features several advantages with respect to real-time systems: The object orientation of the Java supports easy programming, reusability, and robustness. Java bytecode is portable, of small size, and secure. However, because of its high hardware requirements and unpredictable real-time behavior, the use of Java is hardly possible in embedded real-time systems. Both problems are solved with the design of a multithreaded Java-microcontroller enhanced by an adapted JVM.

Java processors execute Java bytecode instructions directly in hardware raising the performance of Java applications and decreasing the hardware requirements compared to a Java interpreter or Just-In-Time compiler. Such Java processor proposals are Sun's picoJava-I and II ([1], [2]), the JEM-1 [3], the Delft Java processor [4], the PSC1000 [5], and the Java Silicon Machine JSM [6]. Hardware support to decouple bytecode translation and execution is proposed in [7].

A multithreaded processor is characterized by the ability to simultaneously execute instructions of different threads within the processor pipeline. Multiple on-chip register sets are employed to reach an extremely fast context switch. Multithreading techniques are proposed in processor architecture research to mask latencies of instructions of the presently executing thread by execution of instructions of other threads [8]. Recently, multithreading has also been proposed for event-handling of internal events ([9], [10], [11]). However, the fast context switching ability of multithreading has rarely been explored in context of microcontrollers for handling of external hardware events. Contemporary microprocessors and microcontrollers activate Interrupt-Service-Routines (ISRs) for event handling. Assuming a multithreaded processor core allows to trigger so-called Interrupt-Service-Threads (ISTs) instead of ISRs for event handling. In this case an occurring event activates an assigned thread.

Our Komodo project explores the suitability of multithreading techniques in embedded real-time systems. We propose multithreading as an event handling mechanism that allows efficient handling of simultaneous overlapping events with hard real-time requirements. Moreover, we explore our methods in context of a real-time Java system. Our Komodo microcontroller ([12],[13]) starts with a Java processor core. In enhancement, the real-time issue is supported by applying the multithreading technique to reach a zero cycle context switching overhead and by hardware support for advanced real-time scheduling schemes.

The next section presents our Komodo microcontroller, section three our performance results, and section four the chip-space requirements based on a hardware synthesis.

2. The proposed Komodo microcontroller

Because of its application for embedded systems, the processor core of the Komodo microcontroller is kept at the hardware level of a simple scalar processor with a four-stage pipeline. As shown in Fig. 1, the microcontroller core consists of an instruction-fetch unit, a decode unit, a memory access unit (MEM) and an execution unit (ALU). Four stack register sets are provided on the processor chip to support up to four threads. A signal unit triggers IST execution due to external signals.

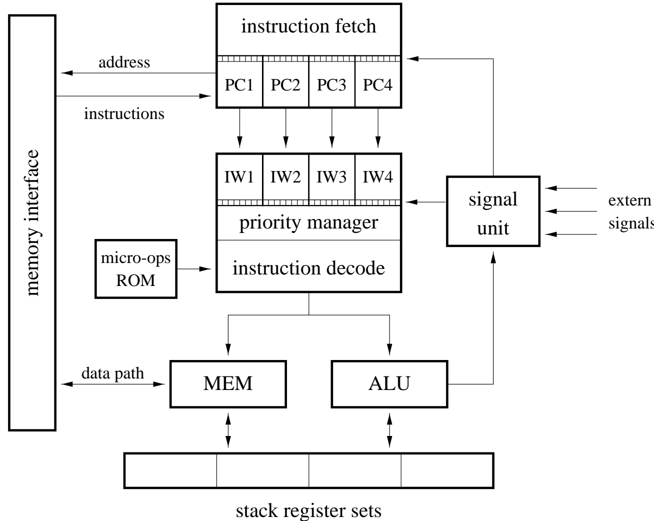

Figure 1: **Block diagram of the Komodo microcontroller**

The instruction fetch unit holds four program counters (PC) with dedicated status bits (e.g. thread active/suspended), each PC is assigned to another thread. Four byte portions are fetched over the memory interface and put in the according instruction window (IW). Several instructions may be contained in the fetch portion, because of the average bytecode length of 1.8 bytes. Instructions are fetched depending on the status bits and fill levels of the IWs.

The instruction decode unit contains the above mentioned IWs, dedicated status bits (e.g. priority) and counters. A priority manager decides subject to the bits and counters from which IW the next instruction will be decoded. We defined several priority schemes [14] to handle real-time requirements. The priority manager applies one of the implemented thread priority schemes for IW selection. A bytecode instruction is decoded either to a single micro-op, a sequence of micro-ops, or a trap routine is called. Each opcode is propagated through the pipeline together with its thread id. Opcodes from multiple threads can be simultaneously present in the different pipeline stages.

The instructions for memory access are executed by the MEM unit. If the memory interface only permits one access each cycle, an arbiter is needed for instruction fetch and data access. All other instructions are executed by the ALU unit. The result is written back to the stack register set of the according thread in the last pipeline stage.

External signals are delivered to the signal unit from the peripheral components of the microcontroller core as e.g. timer, counter, or serial interface. By the occurrence of such a signal the corresponding IST is activated. As soon as an IST activation ends its assigned real-time thread is suspended and its status is stored. An external signal may activate the same thread again.

To avoid pipeline stalls, instructions from other threads can be fed into the pipeline. Idle times may result from branches or memory accesses. The decode unit predicts the latency after such an instruction, and proceeds with instructions from other IWs. There is no overhead for such a context switch, because there are sufficient instructions already loaded in the instruction windows. At each cycle four bytes are fetched, but on average only 1.8 bytes are needed by the decode unit. This leads to mostly stuffed instruction windows. No save/restore of registers or removal of instructions from the pipeline is needed, because each thread has it's own stack register set.

A zero cycle context switch is also possible in processor cores with different instruction set architectures, as long as the instruction fetch bandwidth is exceeds the issue bandwidth.

Because of the unpredictability of cache accesses, a non-cached memory access is preferred for microcontrollers in real-time applications. The emerging load latencies are bridged by scheduling instructions of other threads by the priority manager. Therefore a cache is omitted from our Komodo microcontroller.

3. Simulator-based Performance Evaluation

For the evaluation of our design, we use an execution-based simulator of our Komodo microcontroller with an adapted JVM to run several benchmarks. A typical bytecode mix is derived from the CaffeineMark 3.0. Additionally we use three real-time applications, an impulse counter, a PID element, and

a FFT algorithm. Four threads are active and scheduled by a fixed priority until they are blocked.

Our first experiment estimates the performance gain by bridging latencies—the basic ability of multithreaded processors. We assume processor clock cycles of 100MHz or less and fast SRAM storage for memory thus eliminating load/store latencies. Under these assumptions the only latencies stem from branches, that take two cycles overhead in the pipeline to resolve and fetch from the new PC. The results pointed out a *gain* of 1.28 with:

$$gain = \frac{\text{cycles needed for sequential execution}}{\text{cycles needed by multithreaded execution}}$$

The second experiment additionally regards latencies of data accesses. We assume a large data memory with DRAM modules instead of costly SRAM chips. Due to the need for available instructions we assume a small instruction memory implemented with SRAM chips or as an on-chip memory. This yields instruction fetch without latency. We vary the share of load/store instructions and their latency. The results of this situation are shown in figure 2. The different graphs show the gain with respect to memory latency (in cycles). Each graph represents a program with different percentages of l/s-instructions (10, 20, 40, 50, 70, 90%) respectively the impulse counter (imp), PID element (pid), and fast fourier transformation algorithm (fft). The results in Fig 2 show a significant performance gain which is reached only by the multithreading ability of latency hiding and not by parallel processing.

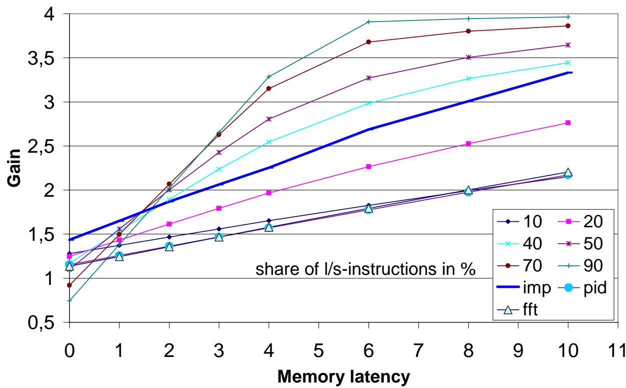

Figure 2: **Benchmark with different share and latency of l/s-instructions**

Next we assume a single memory interface which prefers data access over instruction fetch and a bandwidth of 32 bit. We assume latency of zero cycles for instruction fetch, one cycle for data access, and two cycles for branches. The share of branches is 15.8% and of load/store-instructions 20% which is typical for many applications.

For a zero cycle context switch, it is necessary to keep the instruction windows stuffed. To reach this goal, four instruction fetch strategies are investigated: (1) only the fill level of the IWs is taken into account, (2) the fill level and the priority of the thread are evaluated, (3) fill level and recently taken branches are taken into account, because after a taken branch the IW is empty, or (4) a combination of all three techniques is used. Figure 3 shows only a minimal discrepancy between these strategies and therefore the simplest fetch strategy (only fill level) is sufficient.

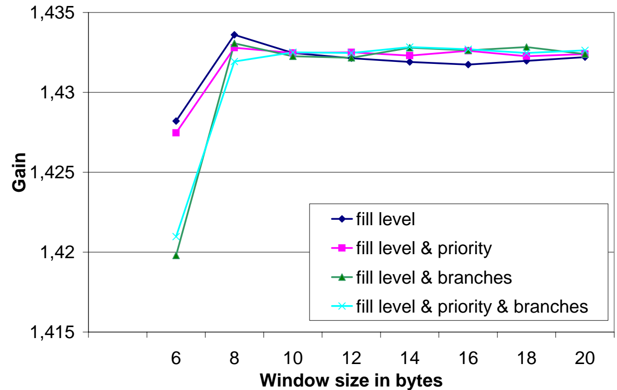

Figure 3: **Benchmark with different fetch strategies**

To find an optimal size of the instruction windows, we ran several tests with different sizes. We found out, that the fetch bandwidth plus two is the minimum window size and fetch bandwidth plus six is about the optimum size.

4. Hardware Design

Based on the simulator results, a VHDL hardware design was performed and implemented on a FPGA. We use the Xilinx Foundation Series development tool for the design and synthesis of our first version of the Komodo microcontroller. A FPGA board with a Xilinx XC4036XL was available for testing the design.

Due to design complexity and restrictions of the number of CLBs we simplify our processor model. First of all we reduce the instruction set to about 50 hardware supported bytecodes. In detail, we only implement some branch and basic arithmetic instructions (no multiply or division), instructions for stack manipulation, instructions to access an array to test our micro-ops unit, and extended bytecodes to load and store values in memory. All instructions work on

integers only. Our priority manager only performs a simple cycle-by-cycle switching between the threads. More complex instruction scheduling strategies will be implemented later.

Technical data	
data bit width	32 bit
address space	17 bit
instruction window size	8 bytes
stack size	16 entries
external frequency	20 MHz
internal frequency	5 MHz
number of CLBs	1 296
number of flip-flops	853
number of latches	109
gate count	~27 200

Table 1: **Technical data of the pipeline**

Table 1 shows the parameters and synthesis results of our single-threaded pipeline design and table 2 shows the chip-space requirements for each unit. Most of it is required by the execution unit and instruction window and decode unit. The stack unit is kept small, because of its small size of 16 entries.

unit	# CLB	# FF	# lat	# gates
fetch	60	78	0	~1 600
IW/decode	445	151	40	~6 700
execute	506	181	0	~7 760
write back	27	43	0	~550
stack	154	279	0	~8 040
misc	104	51	3	~2 550
total	1296	853	109	~27 200

Table 2: **Chip-space of the different units**

Multithreading requires additional space for multiple PCs in the instruction fetch unit, for the instruction windows in the decode unit, and for the multiplication of the stack register sets. Additionally, a tag for binding of an in-flight instruction to its thread is needed in each pipeline stage.

To determine chip-space requirements for the multithreaded Komodo processor, we implement the multithreaded pipeline. The results of the synthesis in figure 4 show that the bulk of the chip-space is occupied by the additional stack register sets. This will be even more dramatic by more reasonable stack sizes of about 64 entries.

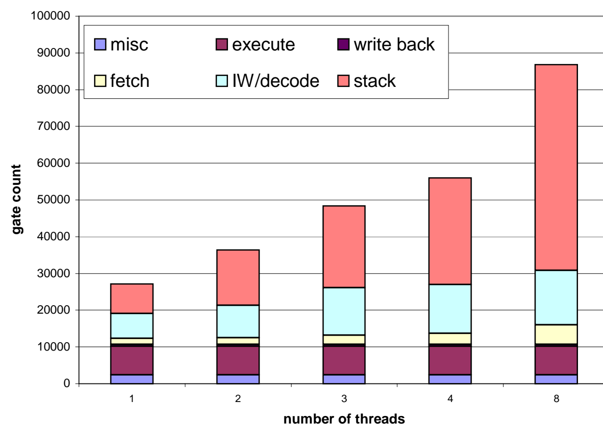

Figure 4: **Chip-space of multithreaded pipeline**

5. Conclusions

This paper describes a multithreaded Java microcontroller, several simulation based benchmark results that show the performance gain possible through multithreading, hardware parameter optimizations, and chip-space requirements of the multithreaded versus the single-threaded processor configurations.

A complete system with hardware, operating system (JVM), middleware and application (automated guided vehicles) is under construction. We are working on optimizations of our hardware design and further improvement of the adapted JVM.

6. References

[1] J. O'Connor and M. Tremblay. *PicoJava-I: The Java Virtual Machine in Hardware.* IEEE Micro, pages 45–53, March/April 1997.

[2] Sun. *PicoJava-II Microarchitecture Guide.* Sun microsystems, Part No.: 960-1160-11, March 1999.

[3] Wolfe. *First Java-specific Chip Takes Wing* Electronic Engineering Times, April 1997, http://www.techweb.com/wire/news/ ↩ 1997/09/0922java.html

[4] C.J. Glossner, S. Vassiliadis. *The Delft-Java Engine: An Introduction* Euro-Par 97, Passau, Germany, August 1997, 766-770.

[5] Patriot Scientific Corporation. *PSC1000 Microprocessor homepage* http://www.ptsc.com/psc1000/

[6] H. Ploog, R. Kraudelt, N. Bannow, T. Rachui, F. Golatowski, D. Timmermann. *A Two step Approach in the Development of a Java Silicon Machine (JSM) for Small Embedded Systems.* Workshop on Hardware Support for Objects and Microarchitectures for Java, Austin, Texas, October 10, 1999, in conjunction with ICCD'99.

[7] R. Radhakrishnan, D. Talla, L.K. John. *Allowing for ILP in an Embedded Java Processor.* 27th Annual International Symposium on Computer Architecture, ISCA'27, Vancouver, Canada, June 10-14, 2000, 294-305.

[8] J. Silc, B. Robic, and T. Ungerer. *Processor Architecture: From Dataflow to Superscalar and Beyond.* Springer-Verlag, Heidelberg, 1999.

[9] R. S. Chappell, J. Stark, S. P. Kim, S. K. Reinhardt, Y. N. Patt. *Simultaneous Subordinate Microthreading (SSMT).* ISCA'26 Proceedings, Atlanta, Georgia, Vol 27, No 2, pp. 186-195, May 1999.

[10] S. W. Keckler, A. Chang, W. S. Lee, W. J. Dally. *Concurrent Event Handling through Multithreading.* IEEE Transactions on computers, Vol 48, No 9, pp. 903-916, September 1999

[11] C.B. Zilles, J.S. Emer, G.S. Sohi. *The Use of Multithreading for Exception Handling* MICRO-32, Haifa, November 1999, 219-229.

[12] U. Brinkschulte, C. Krakowski, J. Kreuzinger, R. Marston, and T. Ungerer. *The Komodo Project: Thread-Based Event Handling Supported by a Multithreaded Java Microcontroller.* 25th EUROMICRO Conference, Milano, September 1999.

[13] U. Brinkschulte, C. Krakowski, J. Kreuzinger, T. Ungerer. *A Multithreaded Java Microcontroller for Thread-oriented Real-time Event-Handling.* 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT '99), Newport Beach, Ca., pp. 34-39, October 1999.

[14] U. Brinkschulte, C. Krakowski, J. Kreuzinger, T. Ungerer. *Interrupt Service Threads - A New Approach to Handle Multiple Hard Real-Time Events on a Multithreaded Microcontroller.* RTSS, Phoenix, December 1999.