

Toward a Processor Core for Real-Time Capable Autonomic Systems

Sascha Uhrig, Stefan Maier, Theo Ungerer

Institute of Computer Science

University of Augsburg

86159 Augsburg

Germany

Tel.: +498215982353

Fax.: +498215982359

{uhrig, maier, ungerer}@informatik.uni-augsburg.de

Members of the HiPEAC network of excellence

Abstract—

This paper proposes a processor core that allows to support the autonomic computing principles in embedded hard-real-time systems. The simultaneous multithreaded CAR-Core processor features hardware-integrated scheduling schemes that isolate the hard-real-time thread from non-real-time threads. It is binary compatible with Infineon's TriCore processor and designed as IP core for a system-on-chip. The challenge for the processor design is to implement simultaneous multithreading such that a thread cannot influence the timing behavior of another thread in order to allow predictable thread execution times. Therefore new instruction issue and data memory access techniques are proposed. The autonomic computing requirements shall be implemented by autonomic managers running as helper threads in own thread slots concurrent to the real-time application. The autonomic manager threads monitor the application and decide if self-configuration, self-healing, self-optimization, or self-protection must be triggered.

Keywords Autonomic computing, organic computing, multithreading, real-time scheduling, helper threads

I. INTRODUCTION

Autonomic Computing [1], [3] has been introduced by IBM at the beginning of this millennium. The basic idea is to make computing systems behave more like organic entities, which adapt to new challenges, heal themselves after injuries, and protect themselves against attacks. The Organic Computing Initiative (<http://www.organic-computing.org>) defines an Organic Computing system as “technical system which adapts dynamically to the current conditions of its environment. It is self-organizing, self-configuring, self-healing, self-protecting, self-explaining and context-aware”. While Autonomic Computing focuses mainly on servers and computing centers, Organic Computing aims at the development of robust, flexible and highly adaptive embedded systems.

Our solution is to fulfil the Autonomic and Organic Computing (AC/OC) and the hard-real-time demands at the hardware level by combining a multithreaded processor core with appropriate hardware scheduling within a SoC. Multithreaded processor architectures support the execution of multithreaded

programs by special hardware attributes like multiple register sets, multiple program counters and a special pipeline design to allow the mixed pipelined execution of instructions from different threads [7]. Multithreading has so far been mainly implemented to increase processor performance by latency hiding.

The multithreaded Komodo microcontroller [4] has been developed to explore the properties of hardware multithreading with hardware-integrated real-time scheduling for embedded real-time systems. Its core processor is designed as single-issue Java processor which is able to run real-time garbage collection and a dynamic real-time class loader as helper threads concurrent to hard-real-time threads. Infineon incorporated multithreading into its TriCore 2 signal processor [6] extending the single-threaded superscalar TriCore 1 processor. The TriCore 2 features two thread slots running the alternate thread from scratch memory if the main thread suffers an instruction cache miss. The multithreaded application-specific extension (MT-ASE) of MIPS proposes to use multithreading on the ASIC level [5].

A basic requirement for the processor design is derived from the demand of real-time capability: a significant WCET analysis should be possible. Moreover, the performance requirements of complex control algorithms in embedded applications demand more performance than traditionally simple processors for real-time systems can provide. An implementation of the AC/OC principles requires additional computing power, the ability to monitor the application threads, and a temporal isolation from the real-time application threads. Therefore a new processor core has to be developed which satisfies the performance demands as well as the requirement of countable execution times. The design presented in this work sets aside caches and speculative execution, but uses multithreading in combination with a real-time scheduling to match the postulated properties. Due to the real-time requirements, several techniques like real-time scheduling, split-phase loads, and interruptible microcodes must be integrated (see section IV).

The paper is organized as follows: The next section briefly describes the CAR-SoC project in which the autonomic/organic

computing principles and the presented processor core are pursued. Section III presents CAR-SoC—the aimed overall hardware solution. Section IV focuses on the main part of the paper, the design of the multithreaded processor core called CarCore, followed by its evaluation (section V) and the conclusions.

II. THE CAR-SOC PROJECT

State-of-the-art embedded computing systems with limitations in space and energy consumption are manufactured as so-called System on Chip (SoC), where all electronic components are placed on a single chip. CAR-SoC (Connective Autonomic Real-time System on Chip) is a new SoC approach aiming to integrate hardware and software for high performance embedded computing with respect to further requirements:

- Connectivity to enable several SoCs to dynamically form networks. Future embedded systems will consist of multiple small computing components which cooperate to solve a common task. The principles of autonomic/organic computing can require a change of the network topology in case of a node failure or a new node in the network.
- Autonomic and Organic Computing (AC/OC) aim at improved controllability of complex systems and require future systems to fulfil the self-x properties, i.e. self-configuration, self-healing, self-optimization, and self-protection. Future systems will therefore act more independently, flexible and autonomously, i.e. they will be more life-like. For these tasks, additional computing power besides the actual application is required.
- Real-time capabilities (hard, firm, or soft) to keep the timing requirements demanded by many embedded applications.

The CAR-SoC uses so-called helper threads which are running in own hardware thread slots concurrent to the application to implement AC/OC managers at system software and middleware level. These managers monitor the application and decide if self-configuration, self-healing, self-optimization, or self-protection techniques must be triggered at their respective levels. A special real-time scheduling scheme (Guaranteed Percentage scheduling as proposed in [4] is used as starting point) isolates the AC/OC manager threads from the application's real-time thread(s).

Figure 1 shows the different layers of the CAR-SoC System architecture which consists of the hardware layer, a system software layer, a middleware layer, and the application itself. The system basis is the CAR-SoC chip, which combines a multithreaded processor core with reconfigurable hardware. The reconfigurable part is used to adapt the SoC to specific tasks and to accelerate dataflow oriented applications.

The next upper level is the system software which is expanded by helper threads. Two separate AC/OC managers monitor the application on middleware and system software level. AC/OC management on the local (i.e. node) level is realized by a closed control loop consisting of local monitoring, local AC/OC management and local resource management. Changes detected by monitoring are reflected by the AC/OC manager and adapted by the resource manager. A similar control loop can be found on the global (i.e. middleware) level. This control loop consists of global monitoring, global AC/OC manager

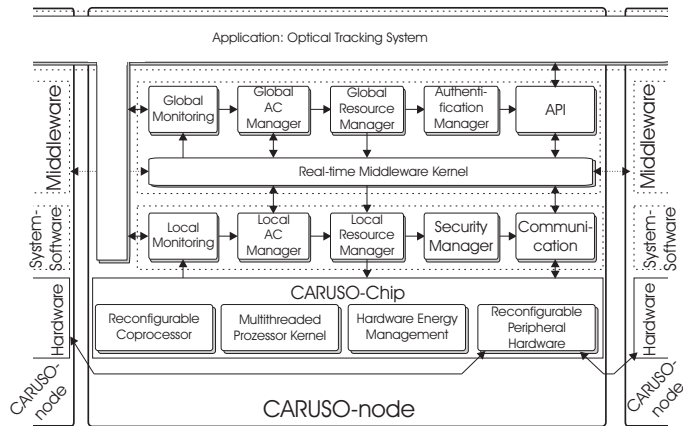


Fig. 1. The CAR-SoC project layers

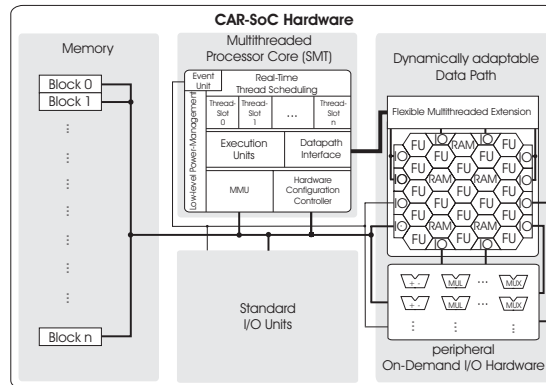


Fig. 2. The CAR-SoC chip

and global resource manager. It is responsible to handle the distributed AC/OC management, e.g. migration of services, deactivation of failing chips, optimization of network topology, etc. The middleware is therefore responsible for the global system management and optimization.

III. CAR-SOC DESIGN

The bottom layer of the project is formed by the CAR-SoC chip itself. The chip carries the CarCore, a multithreaded processor core, an array of reconfigurable functional units, standard I/O units, and memory blocks. Figure 2 shows a block diagram of the CAR-SoC chip. The reconfigurable array will be integrated to speed-up dataflow oriented applications and to realize complex communication protocols (e.g. CRC calculation). As standard I/O units several simple communication interfaces like UARTs, IIC or CAN are imaginable. Also a high speed chip-to-chip communication via LVDS is possible.

IV. CARCORE PROCESSOR

The CAR-SoC processor core, called CarCore, is derived from Infineon's TriCore 1 microcontroller core. The CarCore is fundamentally binary compatible to the TriCore and their two basic pipelines are very similar [2]. The CarCore microarchitecture extends the TriCore by the ability to execute up to four threads in an overlapped parallel fashion. Figure 3 shows the

block diagram of the CarCore with its two pipelines, the eight register sets (one register per pipeline and per hardware thread slot), and the scheduler.

Due to the real-time capability of the CAR-SoC and a predictable runtime behaviour, we avoided caches and speculative execution. Neither of the four threads can influence the runtime behaviour of the other threads (except for software control mechanisms like mutual exclusion). The aim is to use several thread slots as helper threads for autonomic management while the real-time application is running in the other thread slots.

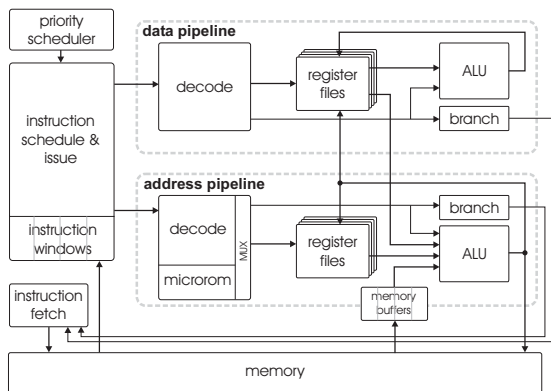


Fig. 3. The CarCore

A. Simultaneous Multithreading

Infineon’s TriCore 2 possesses two separated register sets and two pipelines for address and data calculation. In most cases a data and an address instruction can be executed in parallel. The CarCore extends this technique by the simultaneous multithreading paradigm: Instructions from different threads can be simultaneously issued to both pipelines of the CarCore. Thereby, the restriction given by the TriCore that only a leading data instruction and the following address instruction of the same thread can be issued in parallel to both pipelines. Instructions of different threads always can be issued to the two pipelines.

B. Microcode and memory accesses

Several TriCore instructions (e.g. the CALL and RETURN instructions) are very complex and require the address pipeline for several cycles. If such an instruction is issued in a multithreaded processor, the address pipeline would be blocked. Hence, the predictability of the timing behavior of the other threads would be destroyed. The solution in the multithreaded CarCore is to implement these complex instructions as interruptible sequences of microinstructions. Besides the four program counters four microinstruction pointers are available—one program counter and one microinstruction pointer for each thread. This technique allows to interrupt the execution of one thread’s microprogram by the execution of either machine instructions or microinstructions from another thread—a requirement for hard-real-time.

Due to long memory latencies a thread could lock the load/store interface which means it potentially blocks other

higher prioritized threads. To avoid this problem, the CarCore implements so-called split-phase load accesses i.e. address calculation and data write back is divided into two parts. In between the data received from memory is stored in a load buffer for later insertion into the register file. Each thread slot has got its own load buffer.

Both techniques follow the aim that no thread is able to disturb the runtime behavior of another thread. This feature forms the basis for the real-time scheduling described in the next section.

C. Scheduling

As thread scheduling the guaranteed percentage (GP) scheduling introduced within the Komodo project [4] will be applied. It allows a hermetic temporal isolation of the scheduled threads. The desired percentage of execution time is assigned to each thread and will be guaranteed during short periods of time (e.g. 100 execution cycles). The only restriction is that the total amount of assigned percentages (for hard-real-time threads) must not exceed 100%.

Due to the multithreaded execution and its latency bridging, more than the total number of 100% of execution cycles are available. These add-on cycles can be used for non-real-time applications or for the AC/OC management.

In the current prototype simulator the GP scheduling is not integrated yet. Instead of that, we used two more simple scheduling schemes:

- The fixed priority scheme (FPP) associates fixed priorities to the thread slots depending on the thread slot number, i.e. thread 0 has the highest and thread 3 the lowest priority.
- The round-robin (RR) scheduling assigns the highest priority to another thread in a circular manner each cycle.

V. EVALUATION

For evaluation we implemented a cycle-accurate simulator of the CarCore processor in System-C. We used both simple scheduling schemes (FPP and RR) and compared them against each other and against a serial single-threaded execution. We used several small self-made benchmarks which are executed one to four times in parallel, each in its own thread slot:

Bsort: An array of strings is sorted by the well-known bubble sort algorithm. It generates high load in the address pipeline.

Qsort: The same array is sorted by the recursive quick sort algorithm. Many of the complex call and return instructions are executed.

FFT: An integer based version of the Fast Fourier Transformation is performed on emulated input signals. The FFT requires a higher computational power compared to the other benchmarks and is most similar to typical applications for high performance microcontrollers.

PID: This benchmark executes a proportional-integral-differential regulator algorithm. Like the Bsort benchmark most of the instructions are address-related instructions. It is a typical application for embedded systems.

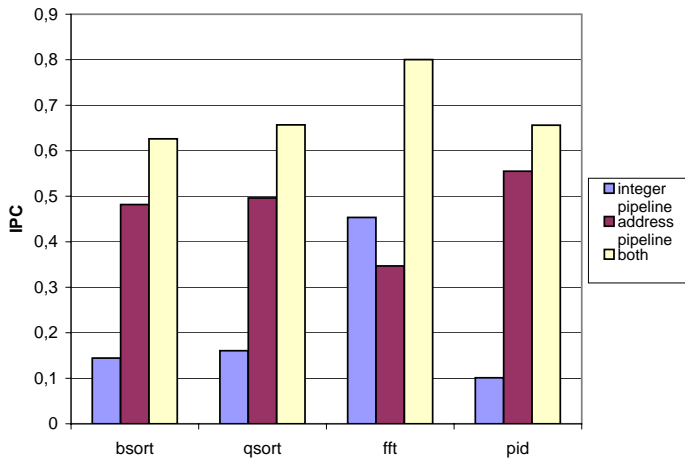


Fig. 4. IPC and pipeline usage of the different benchmarks

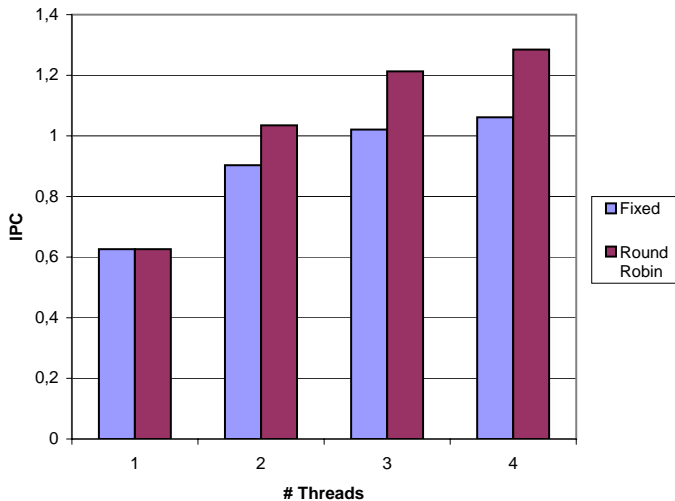


Fig. 5. IPC of the *bsort* benchmark running in multiple thread slots

The performance evaluation with one thread (figure 4) shows low usage of the processor pipeline due to latencies resulting mainly from microcode instructions executed in the address pipeline. Compared to the other benchmarks the computational intensive FFT benchmark takes most advantage from the separated address and data pipelines.

However, multiple executions of the same benchmark in more than one thread slot increases efficiency. Latency bridging leads to a performance gain of up to 100 % (*bsort*, 4 threads, and round robin scheduling, figure 5). But also the computing-intensive FFT benchmark gains up to 75 % of processor performance (4 threads, round robin, not shown in figure).

Without respect to the number of running threads the address pipeline comes out as the bottleneck of the implemented Car-Core microarchitecture. Many latencies during single threaded operation arise from microcode execution which represents half of the operations executed in the address pipeline. Using more than one active thread leads to the address pipeline working at its limit. As shown in figure 4 only the FFT benchmark has a higher IPC within the integer pipeline than within the address

pipeline.

An interesting observation is the worse performance of the FPP scheme compared to RR which happens because of its focus on the thread with the highest priority. This thread is always preferred to the others and its execution finishes first. After a thread has finished, less possibilities for latency bridging are available in the multithreaded processor. At last only one thread is executed and the performance drops. However, the user may prefer the FPP scheme, because it allows a WCET analysis for the thread with the highest priority. Therefore it is suitable for applications with hard-real-time requirements.

In contrast the RR scheduling scheme does not prefer a single thread. The threads finish nearly at the same time and the scheduler has the full choice until the end of the simulation is reached. None of the threads is real-time capable in case of the RR scheduling.

VI. CONCLUSIONS

We designed and evaluated a simultaneous multithreaded processor core for reconfigurable SoCs able to guarantee predictable thread timing despite simultaneous issue and execution of instructions of different threads. The processor core is designed to host hard-real-time threads in combination with autonomic/organic manager threads that monitor the real-time threads. This paper focuses on the processor core. Its evaluations show the effectiveness of the multithreading approach, but also the bottleneck in the address pipeline in particular due to the microcoded implementation of the complex CALL and RETURN instructions of the TriCore instruction set. An interesting observation is that a priority scheduling scheme as required to allow hard-real-time for a single thread leads to an early termination of this thread and therefore to a lower performance than a round robin scheduling because towards the end of the runtime not enough instructions of other threads are available to fill the latency gaps. An adaptation of the guaranteed percentage scheme of the Komodo processor to the CAR-core should avoid this performance decrease.

REFERENCES

- [1] Paul Horn. Autonomic Computing: IBM's Perspective on the State of Information Technology. IBM Manifesto, IBM Corporation., October 2001.
- [2] Infineon Technologies AG, München. *TC1130 User's Manual*, May 2004. Version 1.1.
- [3] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36(1):41–50, January 2003.
- [4] Jochen Kreuzinger, Alexander Schulz, Matthias Pfeffer, Theo Ungerer, Uwe Brinkschulte, and Christian Krakowski. Real-time Scheduling on Multithreaded Processors. In *7th International Conference on Real-Time Computing Systems and Applications (RTCSA 2000)*, Cheju Island, South Korea, pages 155–159, December 2000.
- [5] MIPS Technologies, Inc., Mountain View, CA, USA. MIPS Technologies Boosts System Performance Through New Multi Threading Extensions to MIPS Architecture. Press release, October 2003.
- [6] Erik Norden. A Multithreading Extension for Low-Power, Low-Cost Applications. Talk at: Embedded Processor Forum 2003, San José, CA, USA, May 2003.
- [7] Theo Ungerer, Borut Robič, and Jurij Šilc. A survey of processors with explicit multithreading. *ACM Computing Surveys*, 35(1):29–63, March 2003.