Application Specific Real-Time Microkernel in Hardware

Susanna Nordström {susanna.nordstrom@realfast.se}, Lennart Lindh {lennart.lindh@realfast.se}, Lars Johansson, Tobias Skoglund

Department of Computer Science and Electronics, Mälardalen University, Västerås, Sweden

Abstract—**Real-time operating system kernels in embedded systems need to be configurable. Unfortunately many of today's commercial real-time kernels are monolithic. These optimized code packages are difficult to change and maintain. This is motivated mainly to achieve short response time and easy access of debugging information.**

To solve the drawbacks of the monolithic real-time kernels, the microkernel structure was introduced. The microkernel approach has been discussed for having disadvantages regarding performance. We have implemented a prototype that shows how a modular microkernel architecture in hardware can be used to speed up task management, semaphores and flags for a commercial real-time kernel. The implementation shows both the adaptability of the hardware kernel and the performance speed up associated with hardware implementations.

This article demonstrates that a hardware based real-time kernel can keep or increase the performance of a monolithic structured real-time operating system, while improving system modularity.

Index Terms—**Real time systems, Operating system kernels, Field Programmable gate arrays, multitasking**

I. Introduction

Real-time operating systems used in today's embedded systems need to reflect the modularity of the underlying hardware and the increasing demand for adding and replacing functionality in the system. [1] A real-time operating system structured with the microkernel concept fulfils these requirements. [7]

Monolithic kernels are large in size and the structure, or lack of structure, makes them difficult to change and maintain without affecting other parts of the kernel.

The microkernel approach is based on the idea of only placing essential core real-time operating system functions in the kernel, and other functionality is designed in modules that communicate through the kernel via minimal well-defined interfaces. The microkernel approach results in easy re-configurable systems without the need to rebuild the kernel.

The first generation of microkernels suffered from poor performance, which led to bad reputation of this kernel structure. In the second-generation microkernels, performance has increased and is no longer a problem. [3]

The client-server message passing idea of the microkernel structure results in more context switching compared to monolithic systems. [3] Implementing the real-time kernel in hardware, following the microkernel structure approach, results in a microkernel with the same or better performance as monolithic structured systems; but without the latencies that microkernel systems has been discussed for suffering from. In addition to decreasing the size of the software footprint, the RTU hardware solution also realizes the possibility to draw benefits from hardware characteristics such as parallelism, determinism and unburden of the CPU.

The goal of this work is to show that a microkernel-structured kernel in hardware can be ported to an existing monolithic real-time operating system, and how it affects the performance of the system-calls. Minimal changes of the microkernel are done.

The paper is organized as follows. Section II describes previous work and related work, divided in the aspects of hardware support in real-time systems and microkernel structured real-time systems. In section III challenges and consideration of the implemented prototype is described followed by the experimental results in section IV. Finally, our conclusions are presented in section V.

II. Published Work

The real-time kernel in hardware, called real-time unit (RTU), was developed at Mälardalen Real-Time Research Centre, Mälardalen University, Sweden. It has been a topic for research in both uni- and multiprocessor projects at the Computer Architecture Laboratory for many years, by Lindh et al. [2], [8], [9], [14] In the RTU, the scheduling, inter process communication, interrupt management, resource management, synchronization and time management control are implemented in hardware (using VHDL). The hardware implementation is utilized through memory mapped registers and is used together with a software driver of 2 Kb code (also called API, Application Programmers Interface) which makes it possible for the programmer to utilize the hardware, i.e. handle the service calls to the kernel.

Previous work [2], [8], [9], [14] has shown the benefits of having the RTOS kernel activities implemented in hardware; system overhead is decreased resulting in improved predictability and response time, the CPU load and memory footprint is reduced and less cache misses are seen.

The RTU has been further developed into a commercial product in the form of an Intellectual Property component, named Sierra, by the company RealFast AB, Vasteras, Sweden [13].

To begin with, related work performed in the aspect of hardware support for real-time systems will be mentioned. Mooney et al have implemented a configurable hardware scheduler in [6]. The scheduler provides three scheduling disciplines: priority-based, rate monotonic and earliest deadline first. The hardware implementation eliminates the scheduling and the time-tick processing overhead from the real-time operating system.

The Industrial TRON (μITRON) is a subproject of The Real-time Operating System Nucleus project (TRON). In [11], Nakano et al. presents a solution that consists of a hardware part, called "silicon TRON", and a software part, called μITRON. The Silicon TRON together with the μITRON is called a "Silicon OS". The hardware part implements the scheduler with system call functions and the software provides other system call functions and interface processes between applications and the hardware part. Like the RTU, this solution also communicates with the CPU through register and interrupts for task switch. The time measurements on system calls regarding flags and semaphores showed that the system call processing time in hardware could be reduced by 130 to 1880 compared to a conventional implementation in software, i.e. two to three orders of magnitude speed enhancement.

In [12], a hardware architecture for real-time operating systems support using special hardware components implemented in one FPGA, called the F-timer, is presented. The F-Timer is a co-processor that communicates with the microprocessor and releases the processor of the tasks time management. A similar software solution, based on a micro controller, was created for comparison. Measuring the performance, the conclusion was that the software solution was 18 times worse than the F-Timer hardware architecture solution.

Kohout et al. [5] presents a real-time task manager (RTM), a processor extension that implements scheduling, time management and event management with the purpose to minimize real-time operating system performance drawbacks. The RTM was implemented to handle these functionality in μC/OS with the result of reducing processing overhead with 60 to 90%.

Related work in the aspect of microkernel structures of operating systems and real-time operating systems has been performed by Engel et al. who stated in [1] that microkernel structured systems have an advantage over standard embedded real-time operating systems in system-on-chip (SoC). This is because of the microkernel associated modularity, encapsulation of functionality and the adaptability to changes in the design partitioning during development in SoC.

The work of Liedkte et al. in [3] presented that microkernel based systems are usable in practice with good performance by porting a microkernel designed kernel, the L4 kernel, to the monolithic structured Linux kernel. Linux operating system was implemented on top of L4.

As mentioned, work has been done in the area of hardware support of real-time systems and the microkernel real-time operating system structure approach. We have combined these aspects by looking into the microkernel structured real-time kernel in hardware (RTU).

III. Implementation

The configuration of the RTU used in this work is a uni-processor solution that consists of the internal components in fig. 1; an interface, a scheduler, an interrupt handler, a resource manager and a time manager. The RTU as a whole supports 16 tasks, 8 priority levels, 8 external interrupts, 16 semaphores, 8 flags, 4 watchdogs and timers for delay and periodic start of tasks. The scheduling concept in the RTU is a priority-based pre-emptive algorithm. The processor is interrupted only when a task switch is about to occur. (This configuration is equal to the Sierra IP component [13]).

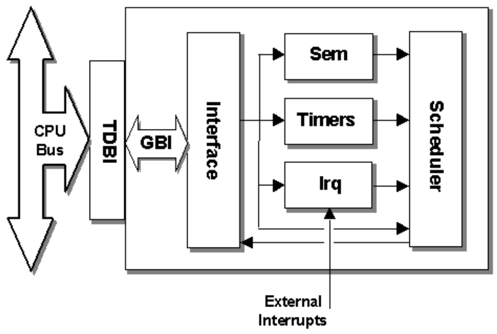

Fig. 1. RTU architecture. GBI – Generic Bus Interface, TDBI – Technology Dependent Bus Interface, i.e. the MicroBlaze On-Chip Peripheral Bus in this implementation. [13]

The implementation that has been done is adapting the existing real-time kernel in hardware, as hardware support for the commercial real-time operating system μC/OS-II by Micrium [10]. The μC/OS-II kernel is implemented completely in software and is a well-used real-time operating system for embedded systems. It is written as a monolithic kernel in the language of ANSI C with a minor part in assembler for context switching.

In the prototype implementation, the RTU has been implemented to replace the scheduling and partial task management, semaphore handling and flag handling in μC/OS-II.

In order to measure the result, benchmarking of service call response time has been executed to a system running μC/OS-II and another system running μC/OS-II on top of the RTU. The measurements have been carried out with a timer component connected to the system bus

The hardware/software implementation has been developed in Xilinx Embedded Development Kit (EDK) 6.2i with the MicroBlaze CPU @24 MHz in a 1 million gate Xilinx Virtex-II FPGA [15].

The implementation of the different parts is described below in the aspects of limitations and considerations of the porting of µC/OS-II and RTU functionality.

A. Task management

In µC/OS-II, the task management is performed by an algorithm, which results in constant time for scheduling, regardless of number of created tasks in the system. Further, the µC/OS-II data structure for events implicates a limitation of the possibility to have several tasks at same priority. This means that the maximum number of tasks in the prototype implementation must be the same as the maximum number of priorities in the RTU, which is 8. No changes to the RTU were made regarding task management. Since the RTU has no support for dynamic change of priority, the µC/OS-II OSTaskChangePrio() call was not supported in the prototype. For the same reason, mutual exclusion semaphores were also excluded. All other service calls regarding task management was however implemented. The data structure for scheduling could be removed since this is handled by the RTU.

Below is a simplified pseudo code example of how the prototype implementation is carried out in the µC/OS-II system call OSTaskCreate():

```
Turn RTU taskswitch_off()
  If priority is available and valid
      Reserve priority
      Turn RTU taskswitch_on()
      Initiate Task Control Block (TCB)
      If TCB initiation was successful
          call RTU task_create()
  Return status
```

The other system calls implementations are carried out in similar way, i.e. calling RTU system calls in a µC/OS-II system call in appropriate locations.

B. Semaphore and flag management

µC/OS-II supports both binary and counting semaphores while the RTU only supports binary semaphores. For this reason only binary semaphores are supported in the prototype. In order not to remove the µC/OS-II query-functionality for semaphore handling, the data structure was preserved and the hardware support was limited to the use of task switch, delay- and undelay calls in connection with semaphore handling.

No changes to the RTU were made regarding semaphore management.

All µC/OS-II flag functionality could be supported except for the timeout functionality in connection with waiting for a flag.

The number of flags in the RTU was extended to handle 8 flags and the ability to handle disjunctive synchronization was added in hardware part and API.

C. Time management

µC/OS-II supports both delay in clock ticks and time. The RTU only supports delay in clock ticks and hence only clock-ticks is supported in this prototype. Another difference is the maximum value of clock ticks. In µC/OS-II the highest value is 65534, in the RTU it is 1024. This is because of the size of the amount of possible transferable data through the current bus interface. In this implementation we chose not to extend the RTU value. The problem is solved with several calls to the RTU call being made instead.

TABLE I
MEASUREMENTS

System Call	µC/OS-II (ticks)	µC/OS-II/RTU (ticks)	Difference (ticks)	frac-tion of ex.time
OSTaskCreate	498	356	142	0.71
OSTaskSuspend	586	412	174	0.70
OSTaskResume	346	315	31	0.91
Manual taskswitch	357	367	-10	1.03
OSFlagCreate	96	133	-37	1.39
OSFlagPost	158	123	35	0.78
OSFlagPost[1]	343	123	220	0.36
OSFlagPost[2]	455	123	332	0.27
OSFlagPost[3]	529	467	62	0.88
OSFlagPend[4]	410	353	57	0.86
OSFlagPend[5]	142	128	14	0.90
OSSemCreate	98	90	8	0.92
OSSenPend	98	91	7	0.93
OSSemPost	92	81	11	0.88
OSSemPost[6]	229	182	47	0.79
OSSemPost[7]	451	531	-80	1.18
OSSemPend[8]	370	417	-47	1.13

[1] One task with power priority waiting for flags
[2] Two tasks with lower priority waiting for flags
[3] One task with higher priority waiting for flags
[4] One task pending, task switch occur
[5] One task pending, task switch do not occur
[6] One task with lower priority waiting for semaphore
[7] One task with higher priority waiting for semaphore
[8] Semaphore not available, task switch occur

IV. EXPERIMENTAL RESULTS

Benchmarking of service call response time was executed to a system running µC/OS-II and another system running µC/OS-II on top of the RTU. The experimental results of the measurements can be seen in table 1 [4].

The most extreme positive result is the modified OSFlagPost system call. It was measured to be executed in less than a third of the time of the original µC/OS-II system call. This is because the execution time increases with the number of tasks waiting for the posted flags in the original µC/OS-II call. This is not the case in the prototype implementation with the RTU support. In the RTU, the number of tasks waiting for posted flags does not affect the execution time, since the processor is not involved in the flag queue handling. In the prototype, the µC/OS-II code was severely reduced in size when most of the functionality could be executed in the RTU hardware instead. The only time the processor will be interrupted is when a task switch is necessary.

The most extreme negative result was the modified OSFlagCreate() system call. It was measured to have increased the execution time with a factor of 1.39 of the original µC/OS-II system call. The code in the system call performs mainly initialization of data structures. Most of this code was preserved in the modified system call. This is because the concerned data structures are used in the µC/OS-II fault checking functionality and the RTU does not support this. Further, the RTU call flag_set() had to be executed to initialize the flags to a specified value. This results in an overhead compared to the original system call, because the execution time of RTU system call is added to the original µC/OS-II code.

There are some other aspects that affect the service call response time besides those already mentioned in the extreme cases. Whenever a taskswitch is included in a service call the processor has to be interrupted. The RTU does not help much in this case, because the taskswitch has to be done in the CPU by software control. However, when clock-tick occurs, scheduling decisions has to be made and delays for tasks have to be decremented. All of this is performed by the RTU hardware in the prototype implementation without involvement of the CPU. In the unaided µC/OS-II, this is done in software, which means the processor will be interrupted for every clock tick. This results in overhead. The size of the overhead depends on how often the ticks occur and number of tasks existing in the system.

All pend-calls include a timeout (RTU task_delay call). Since the maximum timeout value is higher in the µC/OS-II call than in corresponding RTU call, overhead is introduced when the RTU has to be called several times.

V. Conclusions

Shortest service call response time was measured in those cases where the RTU hardware was closely compatible to the µC/OS-II software. In order to reach further response time improvements when the RTU hardware differs the most from the µC/OS-II software, the RTU has to be adjusted. Suggestions of adjustments would be to change the bus interface and extend the amount of transferable data to 32 bits. This would increase the size of number of clock ticks possible to set in a timeout, which would consequently shorten the response time in associated service calls when repetitive calls to RTU timeout function can be avoided. Another adjustment would be to add dynamic change of priority in order to further support the µC/OS-II functionality.

Our conclusion is that the principal structure of the RTU can be preserved and the changes suggested concerns only extending of support of functionality.

Finally, hardware support is motivated in embedded systems where the application uses the real-time operating system extensively and when higher performance, smaller software footprint and determinism are of great importance. Having the hardware support structured in a microkernel approach adds flexibility to the system without decrease of performance.

References

[1] F. Engel, G. Heiser, I. KuZ, S. M. Petters and S. Ruocco, "Operating Systems on SoCs: A Good Idea?," in ERTSI in conjunction with 25th IEEE RTSS04, Lisbon, Portugal, December 2004.

[2] Furunäs, J. "Benchmarking of a Real-Time System that utilises a booster." In International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA200), June, 2000.

[3] H. Härtig, M. Hohmuth, J. Liedkte, S. Schönberg and J. Wolter, "The performance of µ-Kernel-Based Systems", in proceedings of the 16th ACM symposium on Operating Systems Principles, p 66-77, Saint Malo, France, 1997.

[4] L. Johansson and T. Samuelsson, "Integration of an Ultra-fast Real-Time Accelerator in the Real-Time Operating System µC/OS-II", Master Thesis report, Malardalen University, Vasteras, Sweden, October 2004.

[5] P. Kohout, B. Ganesh and B. Jacob, "Hardware Support for Real-Time Operating Systems", in Conference on Hardware/Software codesign and system synthesis of contents, p.45-51, Newport Beach, USA, 2003.

[6] P. Kuacharoen, M. A. Shalan and V. J. Mooney III, "A Configurable Hardware Scheduler for Real-Time Systems," in Precedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA´03), p 96-101, Las Vegas, USA, June 2003.

[7] J. Liedkte, "Toward Real Microkernels," in Communicatons of the ACM, vol. 39, No 9, September 1996.

[8] L. Lindh, and F. Stanischewski, "FASTCHART – A Fast Time Deterministic CPU and Hardware Based Real-Time-Kernel." In IEEE, Euromicro workshop on Real-Time Systems, June 1991.

[9] L. Lindh, T. Klevin, and J. Furunäs, "Scalable Architecture for Real-Time Applications – SARA". Swedish National Real-Time Conference SNART99 Linköping, Sweden, August, 1999.

[10] Micrium, Inc, www.micrium.com, (2005)

[11] T. Nakano, Y. Komatsudaira, A. Shiomi and M. Imai. VLSI Implementation of a Real-time Operating System. Proc. of ASPDAC '97, pp. 679-680, January, 1997.

[12] A. Parisoto, A. Souza, L. Carro, M. Pontremoli, C. Pereira, and A. Suzim, "F-Timer: dedicated FPGA to real-time systems design support. In Real-Time Systems", In Proceedings in the Ninth Euromicro Workshop, p.35 – 40, June, 1997.

[13] RealFast Intellectual Property, Vasteras, Sweden, www.realfast.se/sierra (2005)

[14] T. Samuelsson, M. Åkerholm, P. Nygren, J. Stärner and L. Lindh, "A Comparison of Multiprocessor Real-Time Operating Systems Implemented in Hardware and Software." International Workshop on Advanced Real-Time Operating System Services (ARTOSS), Porto, Portugal, 2003.

[15] Xilinx, Inc. www.xilinx.com, (2005)