Benchmarking of a Real-Time System that utilises a booster

J. Furunäs
Mälardalen Real-Time Research Centre
Mälardalen University, Sweden
e-mail: johan.furunas@mdh.se

Abstract *Application speedups can be achieved by speeding up the hardware, e.g. by utilising an Operating System co-processor. The paper present results from a comparison of executing benchmark programs on a system with and without a co-processor, but which otherwise are identical. The observed speedup is mostly due to the fact that there is no need for clock tick administration, in the system with co-processor, and that the booster scheduler is faster than the software based one. In general the system calls are faster without the co-processor for the benchmark considered in this paper. This is mostly due to the slow bus accesses when communicating with the co-processor.*

Keywords: Real-Time Operating System co-processor, benchmark.

1. Introduction

It is not always possible to achieve increased performance by upgrading a processor, supporting faster clock frequencies, or utilising multiple application processors. Faster clock frequencies are limited by physical laws and are dependent of the silicon manufacturing techniques. The utilisation of multiple application processors may be limited by the possibility of making the application parallel and the architecture used, e.g. in an architecture with a shared bus, accessing the bus may be a bottleneck. On the other hand the developer of control systems more and more utilises Commercial Off The Shelf (COTS) components, e.g. computer boards, processors, I/O cards etc, and are therefore not able to optimise the system performance so easy. One solution to increase performance is to decrease the administration, e.g. scheduling, clock-tick administration etc., by utilising a co-processor. There are several examples of utilising special purpose RTOS co-processors [11] [12] [13] [14] [16] [8], or standard processor RTOS co-processors [9] [10]. The benefit of utilising special purpose hardware compared to utilising a standard processor is that it can be designed to be more predictable and to have greater performance, due to utilisation of parallel hardware. Additionally the flexibility of utilising a standard processor is not as distinguished as it was before the invention of flexible hardware, e.g. Field Programmable Gate Array (FPGA). This paper focuses on a special purpose scheduling co-processor called Booster [1], which is a commercial scheduler based on the Real-Time Unit research [18].

The motivation for this paper is to show that it is possible to increase performance, both in a single processor system and a multiprocessor system, through the utilisation of a Booster co-processor. Further motivation is to show that one must prevent bottlenecks e.g. slow busses etc. to efficiently utilise a RTOS co-processor. Results presented are based on benchmarks of a model of a telecommunication application running on:

1. A processor supervised by a commercial single processor RTOS.
2. A processor supervised by a RTOS with Booster support.
3. Two processors supervised by a RTOS with Booster support.

The following application components are measured:

- The application response time, i.e. how fast the application program completes.
- RTOS overhead for the clock tick administration, i.e. the RTOS overhead handling the scheduling of time events such as scheduling periodic- and delayed tasks etc.

The paper is organised as follows. In section 2 the hardware architecture of the benchmark is

described and in section 3 the Booster co-processor is described. Section 4 describes the benchmark application and section 5 describes the method of measurement used. Section 6 presents the benchmark results and section 7 discusses some ideas on how to improve the use of the Booster. Finally, section 8 concludes the paper and presents some ideas on future work.

2. Hardware Architecture

The system that runs the benchmark is a CPX2208 [4] 8 slot Compact PCI [3] chassis with one MCP750 [6] board, two MCPN750 [7] boards and a Booster (cf. Figure 1). The MCP750 is a system master board that is placed in the first slot of the CPX2208 and the booster is placed into its PMC [7] slot.

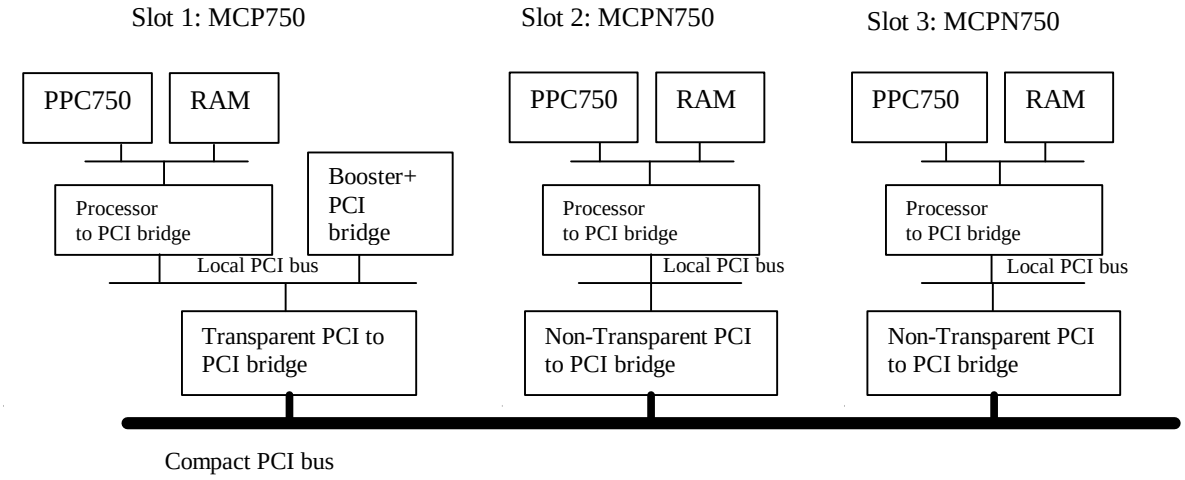

Figure 1: Compact PCI system with one MCP750 board (with a booster) and two MCPN750 boards.

The memory on the MCP750 board is configured as a global memory, which means that the MCPN750 boards are able to access that memory.

The benchmark application is either executed on one of the MCPN750 boards, with or without booster assistance, or shared between two MCPN750 boards that are assisted by a booster.

3. Booster

The booster is a hardware scheduler that schedule processes on one, two or three processors, interacted via register accesses. An RTOS has been implemented that utilises the booster, over the interconnection bus (cf. fig. 2). The RTOS has the same Application Interface (API) as the commercial pure software based RTOS utilised in the benchmark i.e. the booster RTOS is a direct descendent of the pure software one. Different APIs, e.g. POSIX [15] and OSE [17] etc., have been implemented for the booster without any changes in the booster hardware.

As described above the booster is mainly a scheduler, but by adding components with other functionality, e.g. semaphores, watchdogs etc. one can speedup other functionality related to a RTOS. An example of a RTOS co-processor with more functionality is the Real-Time Unit (RTU) [18].
The Booster have 17 registers, were 12 are processor specific registers i.e. 4 per processor. These registers are for handling service calls, round robin times and showing current running-process identity etc. for respective processor. The other 5 registers are shared between all processors and are mainly used for time management, e.g. periodic processes, system time etc. To measure time in the benchmark, the Booster timer register is

utilised as a 1 µs timer. For more details about Booster see [1].

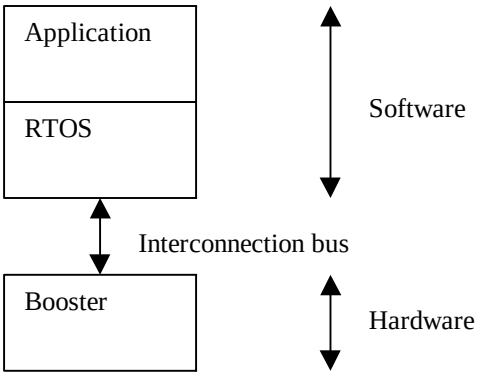

Figure 2: Application with RTOS in software and booster in hardware.

4. Application

A common telecommunication application, implementing the central transitions in a telecom switch, has been chosen as benchmark. The application, can be build as follows from Ericsson UAB [2] (A Swedish telecom company).
There are x number of rings consisting of y number of prioritised workload processes with w number of workload loops and z number of iterations in a workload loop. Each ring represents a telephone call connection. Every process within a ring has identical priority and the priority is increased by one for each new ring. When a process finish z iterations of a workload loop it sends a signal buffer to the next process in the ring and pend until a signal buffer is sent to it i.e. from the previous process in the ring. After w number of workload loops the workload process suspends itself.
The x and y parameters controls the number of workload processes in the system and are used to load the RTOS kernel. W and z controls the process workload. By tuning x, y, w and z it is possible to affect the workload, which also affect the idle process execution time. A low workload is when the idle process gets a lot of execution time.
There are three types of RTOS configurations that the benchmark is tested on, namely.

1. A uniprocessor system based on a commercial widely utilised RTOS (the name of which cannot be disclosed) that utilises local memory for code and global memory for data. One MCPN750 board executes the application.
2. A uniprocessor system that utilises local memory for code, global memory for data and the same RTOS as above is utilising a Booster. One MCPN750 board executes the application.
3. A multiprocessor system that utilises local memory for code, global memory for data and the RTOS is utilising a Booster. Two MCPN750 boards share the execution of the application.

To be able to compare the three configurations the following has been added to the application.

- A high priority process is included to create the rings, start the rings, and to present the results.
- The idle process increases an idle loop variable each time it is executed.
- Code that samples the Booster timer before and after an RTOS service call i.e. services call time.
- Make the ring processes send the service call times to the high priority process, which will present the times.

5. Method of measurement

To measure the processor utilisation factor the response time upon finished work is measured. The response time (cf. figure 3) is measured between starting the ring until the idle process starts, which is done by sampling the booster timer register configured as a 1 µs timer.
To measure the clock tick administration overhead the idle process increments a counter continuously. By knowing how much time it takes to count up to a certain number, without clock tick interruption, and how much time idle is executing it is possible to calculate the clock tick administration time (cf. example below).
Example:
Idle has counted to 5222424. To count up to 198000 (without clock tick) it takes 0.9 seconds. The Idle time is then 5222424/(198000/0.9)=23.8 seconds. Idle starts after 5.2 second i.e. the work processes have finished their work. The time that idle and work processes have to disposal is 29.7 seconds i.e. the time the highest prioritised process is delayed. Idle total time, including clock tick administration is 29.7 - 5.2 = 24.5

seconds. The clock tick time is then 24.5 - 23.8 = 0.68 seconds, which makes 0.68/24.5 = 2.77 % i.e. 2.77 % is clock tick administration during the considered 24.5 seconds period.

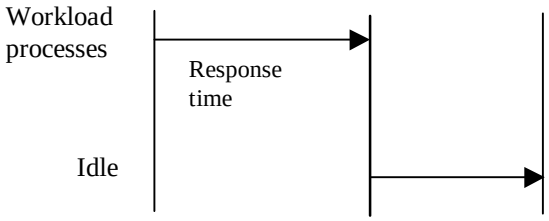

Figure 3: Response time upon finished work.

The RTOS overhead for send, receive alloc and free are measured by sampling (reading the Booster timer) the time before and after the respective RTOS calls. To show the effects of utilising an RTOS co-processor in a uniprocessor and multiprocessor system, the benchmark is executed on the respective system.

6. Results

This section presents the results of the benchmark test, including.

- The application response time, i.e. how fast the application completes.
- RTOS overhead for clock tick administration.
- Effects of utilising an RTOS co-processor in a uniprocessor system and multiprocessor system.

The application response time is decreased when an RTOS co-processor is used. Table 1 shows the response time of the benchmark application with workload loop value 250000. The first column holds the number of workload processes used and the other columns holds the response times (in seconds) for respectively configuration. Each configuration can be described as follows:

- 32 ms/1 ms (GM) = RTOS without co-processor running with 32 ms/1 ms clock ticks, all data is located in a globally accessed memory and the code is located in a locally accessed memory.
- 32 ms/2 ms (LM) = RTOS without co-processor running with 32 ms/2 ms clock ticks, both code and data is located in a locally accessed memory.
- 32 ms/1 ms ($) = RTOS without co-processor running with 32 ms/2 ms clock ticks, both code and data is located in cache memory.
- Boost1 (GM) = one processor with RTOS co-processor, all data is located in a globally accessed memory and the code is located in a locally accessed memory.
- Boost2 (GM) = two processors with RTOS co-processor, all data is located in a globally accessed memory and the code is located in a locally accessed memory.
- Boost1 (LM) = one processor with RTOS co-processor, both code and data is located in a locally accessed memory.
- Boost1 ($) = one processor with RTOS co-processor, both code and data is located in cache memory.

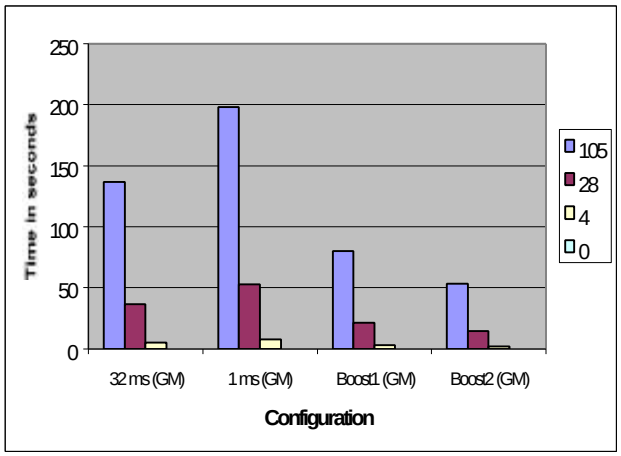

Figure 4: Response time, global memory configuration.

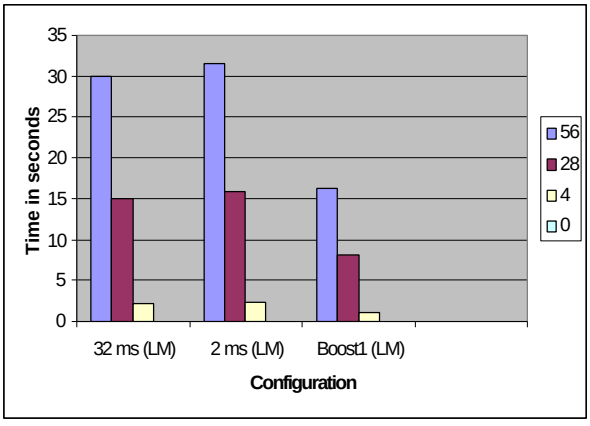

Figure5: Response time, local memory configuration.

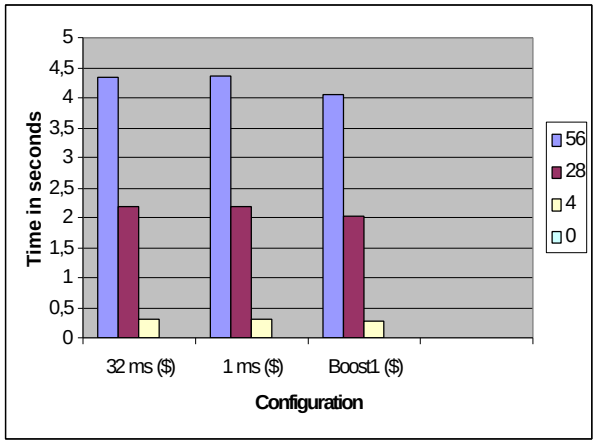

Figure 6: Response time, cache memory configuration.

# of proc	32 ms (GM)	32 ms (LM)	32 ms ($)	1 ms (GM)
105	137,0			198,1
56		29,89	4,34	
28	36,34	14,9	2,17	52,82
4	5,22	2,14	0,31	7,54
0	1,4e-4	5,5e-5	4e-6	1,5e-4
# of proc	2 ms (LM)	1 ms ($)	Boost1 (GM)	
105			79,8	
56	31,62	4,35		
28	15,8	2,17	21,1	
4	2,26	0,31	3,04	
0	5,5e-5	5e-6	2,79e-4	
# of proc	Boost2 (GM)	Boost1 (LM)	Boost1 ($)	
105	53,2			
56		16,23	4,06	
28	14,3	8,12	2,03	
4	2,05	1,16	0,29	
0	1,4e-4	1,1e-4	2,4e-5	

Table 1: Response times in seconds.

Figure 4 to 6 shows the response time for the different configurations. The faster the memory system is the less the response time differences between using and not using a co-processor gets. When using cache and utilising a co-processor, the accesses to the co-processor are costly. With a logic-analyser connected to the processor bus and the PCI bus, write access times have been measured to 230 ns - 1130 and read accesses to 1360 ns - 3630 ns. The PCI bridges and other devices cause the access time variation. Since the PCI bridges have 32 access buffers an access can vary if the buffers gets full i.e. the bridges can't keep up with the processor bus. Additionally other devices, e.g. other processors (if multiprocessor system), Ethernet chips, may access the PCI bus delaying each other's accesses resulting in access time variations.

32 ms (GM)	32 ms (LM)	32 ms ($)	1 ms (GM)	2 ms (LM)
2,8	0,2	0,01	32,7	5,8
1 ms ($)	Boost1 (GM)	Boost2 (GM)	Boost1 (LM)	Boost1 ($)
0,1	0	0	0	0

Table 2: Clock tick administration in %.

Table 2 shows the clock tick administration for the different configurations. As seen, the clock tick administration is up to 32 %, when the commercial RTOS runs with 1 ms clock tick resolution and data located in a global memory accessed over the PCI-bus. In the booster case, clock tick administration is zero since the co-processor takes care of that. One can also see that when a faster memory system is used the clock tick administration decrease. Note that the booster RTOS is implemented to support multiprocessor systems and the commercial RTOS only supports single processor systems. Due to this fact the two RTOS:es are not totally comparable. The comparison would be more accurate if the booster RTOS only supported single processor systems. In the future this change will be implemented and the result from that is that greater speedup is expected.

7. Modifications

The results in previous section show that the PCI-bus accesses are very costly compared to local- or cache accesses, which means that performance could be improved if the co-processor would be located differently in a system. Below some ideas on where to locate a co-processor is described with remarks concerning easiness, scalability and performance.

- Integrate the booster with the CPU. *Easiness:* Is easy when building an own processor but not on COTS processor. *Scalability:* Is possible to scale when building a system on chip based on own

implemented processors else it doesn't scale well.
Performance: Here the greatest performance gains can be achieved, since the co-processor can perform the context switch and the RTOS instruction can be integrated in the processor instruction set.

- Place the booster on the processor bus.
Easiness: Simpler than first idea. One must probably design a new processor board since there are not many COTS boards that have an FPGA on the processor bus.
Scalability: Doesn't scale well.
Performance: Some improved performance but not as good as first idea.
- Place the booster on the processor bus and use snoop signals on the mcp750 to enforce coherency.
Easiness: Same as previous.
Scalability: Same as previous.
Performance: Some improved performance compared to previous but not as good as the first idea.
- Make the booster registers part of the L2 cache.
Easiness: Simpler than first idea. One must probably design a new processor board since there are not many COTS boards that have an FPGA on the L2 cache bus.
Scalability: Same as previous.
Performance: Some improved performance compared to previous but not as good as the first idea.
- Make the booster registers distributed to either L2 cache or local memory over a gigabit network (one for each CPU).
Easiness: Simpler than first idea. One must probably design a new processor board since there are not many COTS boards that have an FPGA on the L2 cache - or processor bus.
Scalability: Does scale well.
Performance: Same as previous.

8. Conclusion & Future Work

This paper describes results of benchmarks of a real-time system, build on COTS components, with and without operating system co-processor. A common telecommunication application, implementing the central transitions in a telecom switch, has been chosen as benchmark. It has been shown that application speedups can bee achieved when utilising a co-processor. But the speedups can possibly be greater if locating the booster differently within a system. Also, greater speedups are expected when the co-processor RTOS is optimised for single processor systems. In the future that will be tested.
Some suggestions on where to locate the booster have been described and should be practically evaluated to prove the correctness of them.

Acknowledgements

Ericsson UAB supported this work.

References

[1] "BOOSTER RTU - Hardware Functional Specification ",RF RealFast AB, Dragverksg 138, S-724 74 Västerås, Sweden, http://www.realfast.se/

[2] Ericsson UAB, Älvsjö, Sweden.

[3] Compact PCI is a computer bus that is defined by PICMG (**P**CI **I**ndustrial **C**omputer **M**anufacturers **G**roup), http://www.picmg.com/gcompactpci.htm

[4] "CompactPCI CPX 2108/2208 Chassis Installation Guide CPX2108A/IH2", http://library.mcg.mot.com/mcg/hdwr_systems/@Generic__CollectionView

[5] "MCP750 CompactPCI Single Board Computer Programmer's Reference Guide", http://library.mcg.mot.com/mcg/hdwr_boards/@Generic__CollectionView

[6] "MCPN750 CompactPCI Single Board Computer Programming and Reference Guide", http://library.mcg.mot.com/mcg/hdwr_boards/@Generic__CollectionView

[7] **P**CI **M**ezzanine **C**ard is defined in IEEE P1386.1/Draft 2.0 (April 4, 1995), http://www.force.de/technology/draft/pmc_draft.pdf

[8] J. Hildebrandt, F. Golatowski, D.Timmermann, "Scheduling Coprocessor for Enhanced Least-Laxity-First Scheduling in Hard Real-Time Systems", 11th Euromicro Conference on Real-Time Systems, York, England, June 9-11, 1999.

[9] W. A. Halang, A. D. Stoyenko, "Constructing Predictable Real Time Systems", Kluwer Academic Publisher 1991.

[10] M. Colnaric, D. Verber, W. A. Halang, "Design of Embedded Hard Real-Time Applications with Predictable Behaviour", Real-Time Applications, Proceedings of the IEEE Workshop, 1993 .

[11] T. Nakano, A. Utama, M. Itabashi, A. Shiomi, M. Imai,"Hardware Implementatiuon of a Real-Time Operating System", Proceedings of the 12th TRON Project International Symposium 28 nov. -2 dec. 1995.

[12] J. Roos, "The Design of a Real-Time Coprocessor for Ada Tasking", Department of Computer Engineering, Lund University, P.O. Box 118,S-221 00 Lund, Sweden, June 21, 1989.

[13] L. D. Molesky, K. Ramamritham, C. Shen, J. A. Stankovic, G. Zlokapa, "Implementing a Predictable Real-Time Multiprocessor Kernel - The Spring Kernel", Department of Computer and Information Science University of Massachusetts, Amherst, MA 01003, USA, May 1990.

[14] Parisoto A., Souza Jr. A., Carro L., Pontremoli M., Pereira C., Suzim A.,"F-Timer: Dedicated FPGA to Real-Time Systems Design Support", Euromicro Workshop on Real-Time Systems, Toledo, Spain, June 11-13, 1997.

[15] [POSIX] Portable Operating System Interface Standard. http://standards.ieee.org/regauth/posix/index.html

[16] Lindh L.,"Utilization of Hardware Parallelism in Realizing Real Time Kernels", Doctoral Thesis, Department of Electronics Royal Institute of Technology S-100 44, Stockholm, Sweden, 1994.

[17] http://www.enea.com/

[18] J. Adomat, J. Furunäs, L. Lindh, J. Stärner." Real-Time Kernel in Hardware RTU: A Step Towards Deterministic and High-Performance Real-Time Systems". Proceedings of the 1996 Euromicro Workshop on Real-Time Systems, 12-14 June, L'Aquila, Italy.