A Comparison of Multiprocessor Real-Time Operating Systems Implemented in Hardware and Software

Tobias Samuelsson
ABB Robotics
Västerås, Sweden
tobias.samulesson@se.abb.com

Mikael Åkerholm, Peter Nygren, Johan Stärner, Lennart Lindh
Department of Computer Engineering, Mälardalen University
Västerås, Sweden
{mikael.akerholm, peter.nygren, johan.starner, lennart.lindh}@mdh.se

Abstract

In this paper we present a performance comparison between a real-time multiprocessor kernel implemented in hardware and a corresponding kernel implemented in software. The hardware kernel showed overall better performance. For instance the speedup achieved with the hardware kernel was up to 2.6 times. We also present an optimization that has been applied to the software kernel, which in some cases showed even better performance than the hardware kernel.

1 Introduction

As real-time applications become larger and more complex every year the demands on real-time platforms also increase, which motivates moving real-time applications onto multiprocessor systems. One method to obtain even more performance is to utilize special purpose hardware, an example Real-Time Unit (RTU) [8]. The RTU contain Real-Time Operating System (RTOS) services implemented in hardware. The work in this paper investigates the gain with utilizing the RTU in a specific multiprocessor system, through a benchmark with a corresponding software kernel. The benchmark is composed by parts from well-known benchmarks and the focus is on performance of basic OS functionality. The software kernel has been especially implemented for this comparison; it offers the same but slightly reduced Application Interface (API) as the RTU. The gain with utilizing the same API is that we can execute the same benchmark code and thereby eliminate implementation differences and strive towards a fair comparison. Furthermore we apply an optimization to the software kernel, which should minimize some of the weaknesses. To explore the potential with the new version of software kernel, some of its benchmark results are compared with the results achieved with the former version as well as with the RTU. The purpose is to find the differences with implementing kernels in software and hardware. The expectation is clearly that the hardware kernel outperforms the software kernel. But how much can we assume to gain when utilizing a hardware RTOS? Can we unveil any weaknesses with the current hardware implementation?

A similar comparison has been made in [1]. The differences are firstly divergent hardware/software API; furthermore the author relies on executing one covering workload. In [9], a comparison between a software kernel and a hardware kernel on a single processor system has been done. In [4] the authors present a simulation based comparison. The remaining part of section 1 firstly introduces the common hardware platform followed by descriptions of the two kernels. Section 2 defines the benchmark method and section 3 presents and discusses the results. In section 4 we shortly present an optimization of the software kernel. In section 5 we discuss the paper and the 6^{th} section contains suggestions for future work.

1.1 Common Hardware Platform

In order to obtain comparable results, it is important that the hardware configuration is equivalent. The hardware platform used by both kernels in this paper is the SARA system described in [5]. The hardware architecture of the SARA system can be divided into local CPU boards, bus-arbitrator, global RAM and I/O. Figure 1 shows a schematic picture of the SARA system. Each board has a Motorola PPC750 processor, running at 367 MHz and the boards are connected to each other with a Compact PCI bus (CPCI). The CPCI bus offers eight slots for CPU boards, of which one is a special system slot hosting the system board. Communication and synchronization between different processes in the system is performed through a global memory that resides on the system board. This implies that all communication between tasks go through the sys-

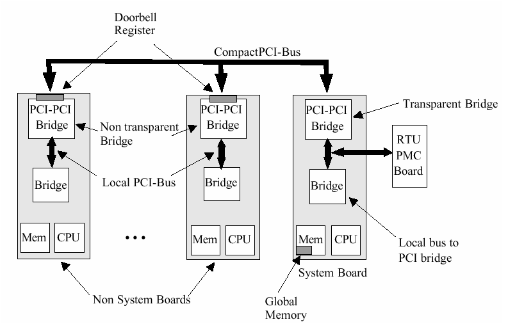

Figure 1. block diagram of the SARA system with RTU

tem board, even if two tasks residing on the same slave board are communicating.

1.2 Hardware Kernel

The RTU is a RTOS kernel implemented in a Field Programmable Gate Array (FPGA) with purpose to increase the performance and predictability. As shown in figure 1, the RTU is attached to the local PCI bus on the system board. The RTU moves scheduling, Inter Process Communication (IPC), semaphore handling and time management from software to hardware. The task scheduling algorithm is premptive and based on Fixed Priority Scheduling (FPS). The RTU has one ready queue for each CPU and there are two events that can perform a taskswitch. The task itself can signal a taskswitch and the scheduler can interrupt an executing task when a task with higher priority is ready to execute. To communicate with a slave board the RTU generates an interrupt through the doorbell register on the slave CPU board. The interrupts to the system board is generated through the standard interrupt lines provided by the PCI bus.

The IPC and synchronization between tasks that the RTU provides, consist of a Virtual Communication Bus (VCB) [6], which is integrated with the RTU. The VCB provides a message passing mechanism that allows task-to-task communication locally on one CPU as well as between different CPUs. The VCB consists of two layers; one hardware layer and one software layer. The hardware layer contains message transaction functionality and is implemented in the same FPGA as the RTU. The software layer runs on the CPU on the system board and contains the API against the hardware layer. The message queues are priority based and placed in the global memory on the system board, but handled by the RTU. If an incoming message has higher priority than the receiving task, the priority of the receiving task is raised during the message transaction.

The interface to the RTU is read- and writeable registers, which makes it easy to port to different platforms. The RTU has registers that store information about the different CPUs and tasks. Each task in the system has a Task Control Block (TCB). One part of each TCB is implemented in software, and another in hardware in the RTU. The RTU has a maximum of 256 TCBs so the maximum number of tasks that the RTU supports is 256 tasks. The TCBs in the RTU have registers to store state, priority, round robin time, pointer to a function, stack pointer, stack size and which CPU the particular task will execute on. The round robin time is an extension to FPS scheduling, the purpose it that tasks with equal priority can utilze this mechanism to share a processor. If the round robin time is set, the associated task will execute the specified time and then it is interrupted by the RTU to leave time for other tasks with equal priority. For further details related to the RTU, please refer to [8].

1.3 Software Kernel

The software kernel acts like the RTU, with respect to API and general semantics. The CPU on the system board executes ordinary tasks as well as the software kernel. The software kernel utilizes the same platform as shown in figure 1, except that there is no RTU attached to the system board. The software kernel is written in PPC750 assembler, C and C++, compiled with GNU Compiler Collection version 2.95.1.

The software kernel is centralized, thereby the kernel on the system board schedules the whole system. The scheduling algorithm is priority driven and premeptive like the algorithm in the RTU. Scheduling decisions are enforced through interrupts. Each task executes only on the node where it was created and registered, so no task migration is allowed. The scheduler holds the current priority and state of all tasks in the system, while each node has a TCB associated with each of its own tasks. The TCB is the same as in the RTU except that the complete TCB is implemented in software, i.e. the TCB registers in the RTU are moved to software. The message-queues are placed in the global memory on the system board and to protect them, a priority driven semaphore-protocol has been implemented.

With a centralized implementation costly clock synchronization and clock-tick interrupts on the slave boards can be avoided. Clock-tick interrupts are used only on the master board, and the resultion differs from the RTU. The RTU has configurable clock-tick resolution, however the default and used interval between two interrupts is 10 s. Using such a resolution in the software version leaves no time to other activities. Instead the interval between two ticks is set to 100 ms. This seems to be too long during normal circumstances, since a kernel should be able to handle delays for about 10 ms. But as a consequence of many system calls, such as sleep, send and receive the kernel executes the scheduling algorithm and can postpone any pending clock-tick interrupts another 100 ms. As a result when executing system-call intensive applications, the software kernel becomes completely event driven. All benchmarks in this paper are system-call intensive or measure only the time for a specific system-call. The motivation for using a 100 ms period in a benchmark situation is that it is possible to manually start different tasks and nodes during the same period.

2 Benchmark Methods

Both kernels in the benchmark have the same API, although the software kernel has a slightly reduced version. The benchmark series is built with own ideas and parts from the benchmarks: Rhealstone [3], SSU [4] and Distributed Hartstone [2]. All tests have been repeated five times in order to compute the average result. Five times was regarded as enough to constitute basis for the comparison since each observation point had a very small variation range.

2.1 Create Task

This test case is taken from the SSU benchmark and it is the time it takes to create one task that is measured. The timekeeping starts when a task is going to be created and stops when the task has been created. One condition that may affect the task creation time are the number of already created tasks if the RTOS has to traverse list structures with existing tasks, another is where in the system the task is located due to access time variations caused by physically varying distances. Therefore the number of already created tasks is varied between 0 and 16 and the test is performed on both the master and a slave node independently. In complex multiprocessor systems, 16 tasks are still very low number of tasks. The aim for this benchmark is to find out if the number of already created tasks affects the create task time, therefore the maximum of 16 tasks has been considered as enough.

2.2 Taskswitch

This test case has been influenced by the Rhealstone benchmark and it measures the time to switch between two independent and active tasks with equal priority. The taskswitch time is crucial to the performance in a real-time system. In this test the terms for variation are the number of simultaneously active tasks and the placement of tasks, i.e. on master or slave node. Practically we measure the time between that a task calls for a manual taskswitch, until the next task becomes executing. The number of active tasks is varied between 2 and 16, and the tests are performed on both the master and a slave node independently.

2.3 Communication Bandwidth

Variants of this measurement are included in Rhealstone and Distributed Hartstone. This measurement aims to measure the number of bytes per second one task can send to another. The communication bandwidth may be different for tasks hosted by different processors and tasks hosted by the same processor. For this reason independent benchmarks with the receiving and sending task hosted by different processors are performed. Because this benchmark focuses on effective bandwidth, which is the bandwidth a user can utilize, bytes and processing that belongs to the RTOS or communication protocol are redundant. Therefore the effective bandwidth in most systems depends of the message size, the same amount of data divided into more messages gives more redundant bytes, and the message size must be a target for variation. The experimental setup used for this benchmark relies on measuring the time it takes to send a fixed amount of raw data between two tasks, with no other communication present. The timekeeping starts when the first message is sent by the sending task, and ends when the last message is received by the receiving task.

2.4 Communication Latency

The end-to-end communication latency is also measured in Distributed Hartstone. Using different message sizes between 1 and 128 bytes with regular intervals between the two extremes, tests the effect of message size. The effect of task placement is considered by letting the sending and receiving tasks be hosted by different nodes in independent test series. The implementation of this benchmark simply consists of measuring the time from which the message is sent by the sending task until it is received by the receiving task, in this implementation it was most suitable to measure the roundtrip delay. Since we have no accurate external clock and no clock synchronization between the nodes. The first timestamp is taken on the sender side when a message is sent and the second timestamp when the sender receives an acknowledgement from the receiver.

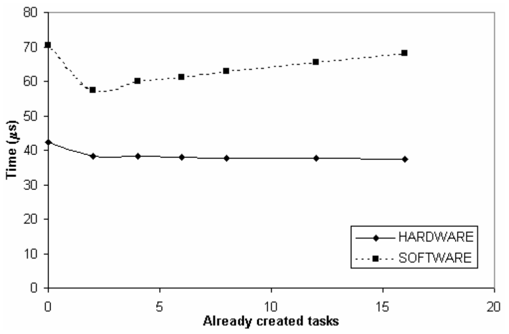

Figure 2. create task benchmark on slave node

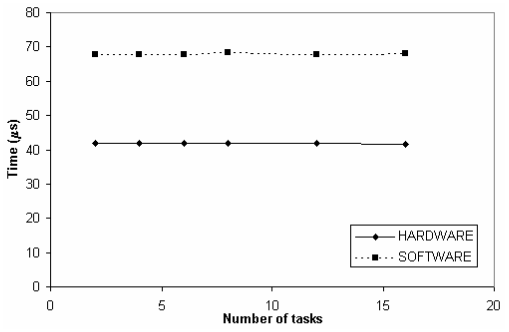

Figure 3. taskswitch benchmark on slave node

3 Result

In the following subsections the results are presented, but due to space limitations we discuss a subset.

3.1 Create Task

As shown in figure 2, the hardware kernel is faster than the software kernel. A task that is created has to be registered to the scheduler, in both cases and this is achieved by a synchronous system call. By synchronous we mean that the caller has to wait for an acknowledgement that is delivered after the call is processed. Also as predicted the software kernel has longer latencies with increasing number of already created tasks. That depends on the list management latencies that increase with increasing number of tasks. The cache memory effects are also visible in the test, since the first call is more time consuming than the second. As figure 2 shows, the effect of caches are bigger with the software kernel than with the RTU (greater slope). The RTU is implemented in a separate hardware module and cannot be affected by the processor caches, but still RTU interface code and parts of the tasks TCBs are affected by caches. The create task benchmark on the master node resulted in another relationship, the form of the graphs were essentially the same but the software kernel was faster than the hardware kernel. In the software case the tasks are created completely locally on the master node, but when using the hardware kernel

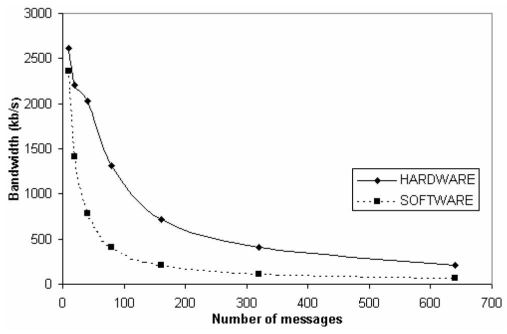

Figure 4. communication bandwidth benchmark, sender on slave and receiver on master node

we suffer of PCI access latencies.

3.2 Taskswitch

When switching tasks on a slave node the hardware kernel is much faster than the software kernel as we can see in figure 3. This is explained by the fact that functionality implemented in hardware often are faster than corresponding software solutions. Notice that the software kernel is not affected by the number of tasks in this test, because the two tasks that are involved in the taskswitch, have the highest priorities in the system and therefore no list traversing has to be done. When this benchmark was executed on the master node, the difference between the two kernels was much smaller. The software kernel was faster, due to the PCI latencies of the RTU against local switches on the master node.

3.3 Communication Bandwidth

When meassuring the bandwidth between two tasks on the master node the software kernel shows best result. But as figure 4 shows the bandwidth is greater with the hardware kernel when communicating across node boundaries. The software kernel seems to be more efficient with increasing number of messages, in relation to the hardware kernel. The reason is that the scheduler in the software kernel performs a speculation after many system-calls, when the result is to not switch task it is a waste of time. A situation when the speculations are successful is when many send and receive calls are processed in a rapid sequence.

3.4 Communication Latency

Figure 5 shows the communication latencies between tasks hosted by different nodes. The sending task resides on a slave node and the receiving task resides on the master node. The test makes clear that the

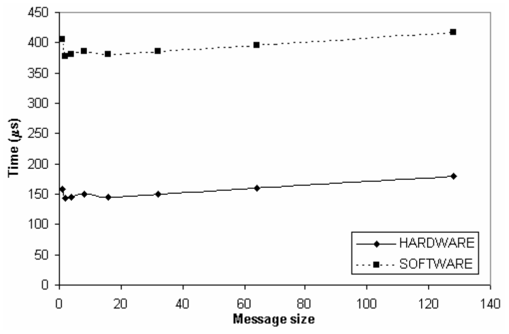

Figure 5. communication latency benchmark, sender on slave and receiver on master node

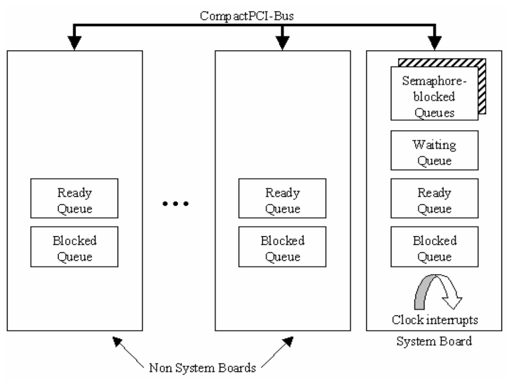

Figure 6. queues in the optimized software kernel

end-to-end delay also referred as the transmission latency is shorter with the hardware kernel in this test case. But as earlier the result depends on the locality of the involved tasks, the software kernel has shorter latency when both the sending and the receiving tasks are hosted by the master node which also hosts the kernel. The observed latencies for the hardware kernel are almost equal in all test cases, whereas the software kernel has shorter latencies when both tasks are residing on the master node and longer latencies when at least one of the two tasks resides on a slave node.

4 Optimization of The Software Kernel

As we could see in the benchmark results presented in the previous section, the main drawback with the software kernel seems to be time consuming system-calls from tasks on slave nodes. As long as tasks are executing on the system board which hosts the kernel, the software solution is often faster than the hardware and its enclosed PCI-access latencies. An optimization that should increase the performance of both kernels is to increase the locality of data and try to avoid PCI-accesses. In the remaining part of this section we will present an optimization of the software kernel that strives for higher utilization of data locality.

A central part of a multiprocessor RTOS is the placement of the processor schedulers and different task queues. Unlike the centralized RTU and the original software version, the improved software kernel consists of several schedulers. The system board has a complete scheduler while the slave boards have simpler schedulers or dispatchers. This quality makes the improved version semi-distributed, i.e. not pure centralized and not fully distributed. The idea with several different schedulers can for instance be found in the Spring system [7]. Each slave board has a local ready queue and local blocked queue, while the system board also has semaphore queues and a waiting queue, see figure 6.

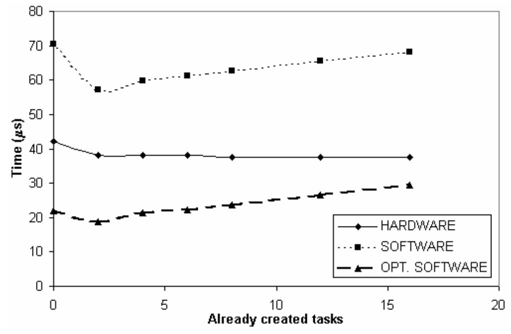

Figure 7. create task benchmark on slave node

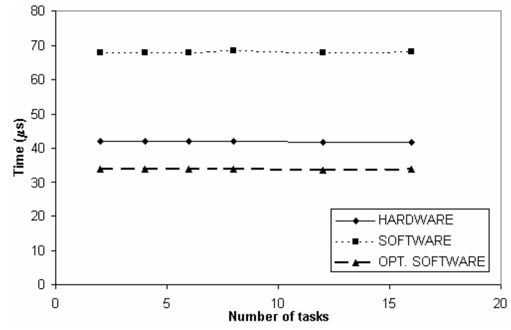

Figure 8. taskswitch benchmark on slave node

Figure 7, 8 and 9 show three examples of benchmark tests when the semi-distributed kernel shows better performance than the centralized software kernel. The semi-distributed software kernel shows even better performance than the hardware kernel in figure 7 and 9. The result in figure 7 is expected, since the tasks are created locally and we do not need any communication with PCI devices in the optimized software case. The hardware kernel on the other hand is a PCI device with enclosed latencies. Also as predicted the optimized software kernel has longer latencies with increasing number of created tasks, since and the list management overhead increases with increasing number of tasks.

The motivation to figure 8, when the semi-distributed software kernel is faster than the hardware kernel, is that taskswitches can be handled locally on a slave node. Each node manages its own queue of active

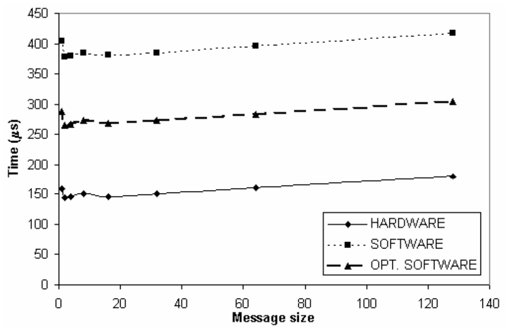

Figure 9. communication latency benchmark, sender on slave and receiver on master node

tasks. But in the hardware case the kernel queues are managed centrally in the kernel-core, residing as a PCI device, which causes access latencies to the queues. As shown in figure 9, the optimized software kernel has shorter communication latencies than the centralized software kernel, but the hardware kernel still shows the best result.

5 Discussion

In this paper we presented the results of a comparison between a centralized software kernel and the hardware kernel RTU, both upon the same multiprocessor system. The comparison was performed with different micro benchmarks. We showed that the performance of the RTU was in general better than the corresponding software solution.

We also showed an improvement of the software kernel. The improved software kernel showed better or equal performance results in comparison with the centralized software kernel. In some benchmark tests the improved software kernel showed even better performance than the RTU. It was not a fair comparison between hardware and software kernels since the implementation details differs, still it shows that potential to improve the RTU exist since it was possible to improve the corresponding software solution.

6 Future Work

As a future work section we present some suggestions to improve the RTU. Firstly we showed that it was possible to achieve a more efficient kernel by moving some of the intelligence closer to the tasks, and thereby get shorter access latencies. This is an essential part of an RTOS kernel since it interacts frequently with the tasks, but also a weakness with the RTU as combined with the SARA system today. One way is to move some of the functionality back into software so that some decisions can be made without PCI access latencies. Another way would be to provide a RTU for each processor in some form. A third option would be to find a better target system that in some way provides shorter access latencies, as for instance a System On Chip (SOC), where the RTU would have almost insignificant access latency compared to a PCI-device. Also since the scheduling algorithm does not load the application processors, it should be possible to use a really fancy and effective algorithm that cannot be used in software implementations.

Acknowledgements Thanks to all personnel at the department.Special thanks go to Leif Enblom for fruitful discussions and to Raimo Haukilahti and Tomas Lennvall for peer reviewing this paper.

References

[1] J. Furunäs. Benchmarking of a Real-Time System that Utilises a Booster. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA200)*, June 2000.

[2] N. I. Kamenoff and N. H. Weiderman. Hartstone distributed benchmark: requirements and definitions. In *Proceedings of the 12th IEEE Real-Time Systems Symposium, IEEE Computer Society Press*, pages 199–208, Dec 1991.

[3] R. Kar and K. Porter. Rhealstone - a Real-Time Benchmarking Proposal. *Dr. Dobbs Journal*, (2), Feb 1989.

[4] J. Lee, V. J. Mooney III, K. Ingström, A. Daleby, T. Klevin, and L. Lindh. A Comparison of the RTU Hardware RTOS with a Hardware/Software RTOS. In *Proceedings of the ASP-DAC 2003, Design Automation Conference*, pages 683 –688, Jan 2003.

[5] L. Lindh and T. Klevin. Scalable Architecture for Real-time Applications and use of bus-monitoring. In *Proceedings of Sixth International Conference on Real-Time Computing Systems and Applications, RTCSA '99*, pages 208–211, December 1999.

[6] P. Nygren and L. Lindh. Virtual Communication Bus with Hardware and Software Tasks in Real-Time System. In *Proceedings for the work in progress and industrial experience sessions, 12th Euromicro conference on Real-time systems*, page 3, June 2000.

[7] K. Ramamritham and J. A. Stankovic. The Spring Kernel: A new paradigm for Real-Time Systems. *IEEE Software*, 8(3), May 1991.

[8] RF RealFast AB. Real-Time Unit, A New Concept to Design Real-Time Systems with Standard Components. Technical report, RF RealFast AB, Dragverksg 138, S-724 74 Vasteras, Sweden, 2000.

[9] L. Rizvanovic. Comparison between Real time Operative systems in hardware and software. Master's thesis, Malardalen University, Vasteras, Sweden, 2001.