A Comparison of Multiprocessor RTOS Implemented in Hardware and Software

Tobias Samuelsson and Mikael Åkerholm
Department of Computer Science and Engineering
Mälardalen University
Västerås, Sweden
{tsn98026, mam98008}@student.mdh.se

Peter Nygren, Johan Stärner, and Lennart Lindh
Computer Architecture Laboratory
Mälardalen University
Västerås, Sweden
{peter.nygren, johan.starner, lennart.lindh}@mdh.se

Abstract
The performance demands on real-time platforms increase, mostly because real-time applications become larger and more complex. Two reaserch directions with purpose to meet the increasing performance demands in real-time systems are utiliziation of multiprocessor platforms and hardware operating system kernels. In this article we show 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, when measuring the communication latency between different nodes. We also present an optimization that has been applied to the software kernel. The improved software kernel showed in some cases even better performance than the hardware kernel.

1 INTRODUCTION

Real-time systems are computing systems that have a time critical nature. When a certain event occurs in the environment, the real-time system must react with the correct response within a certain time. An Operating System (OS) is in simplest form a system program that provides a higher level of abstraction to the application programmer than the bare hardware. The most distinguishing feature with a Real-Time Operating System (RTOS) compared to such an OS is its deterministic and predictive time management methods. 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. Examples of such hardware include graphic accelerators and network cards; another application is the Real-Time Unit (RTU) [10]. The RTU contains typical RTOS functionality that has been moved from software into hardware. In this way it is possible to decrease the RTOS overhead as for instance scheduling, time management and communication primitives, since it is processed by parallel hardware.

In this paper, we present a comparison between a centralized software kernel and the RTU, both executing on the same multiprocessor system. The comparison is achieved by an own series of benchmarks with ingredients from other well-known benchmarks and focus is on performance of basic OS functionality. We made an own benchmark series because there is no existing benchmark adapted for the target system. If we had ported an existing benchmark the comparison with other target systems would anyhow be discussable, since system-calls and other implementation details differ. Another motivation is that we could choose according to us the most interesting benchmarks. 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 is that we can execute the same benchmark code and thereby eliminate implementation differences and strive towards a fair hardware/software 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 hardware implementation.

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 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 while we present many smaller independent measurements. In [11], a comparison between a software kernel and a hardware kernel on a single processor system has been done. In [5] the authors present a simulation based comparison of three different systems; one of the systems is also included in this benchmark.

The outline of the paper is as follows: 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, which also is applicable to the hardware kernel. In the 5th section we discuss the paper and the 6th and final section contains suggestions of future work.

1.1 *Common Hardware Platform*

When comparing different RTOS kernels it is important that the hardware configuration is equivalent, in order to easily obtain comparable results. The hardware platform used by both kernels in this paper is the SARA system described in [6]. 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 Motorla 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, however in a CPCI system there is always one special system-slot. This slot has a special processor board (system-board) that handles the arbitration, clock-distribution, etc on the CPCI bus. 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 system board, even if two tasks residing on the same slave board are communicating.

There are two kinds of PCI buses in the system. All boards have a local PCI bus that is connected to the CPCI bus through a PCI-PCI bridge, see figure 1. The system board has a transparent bridge, while the other boards have bridges that translate addresses on one bus to another address on the other bus, in purpose to utilize the whole local address range.

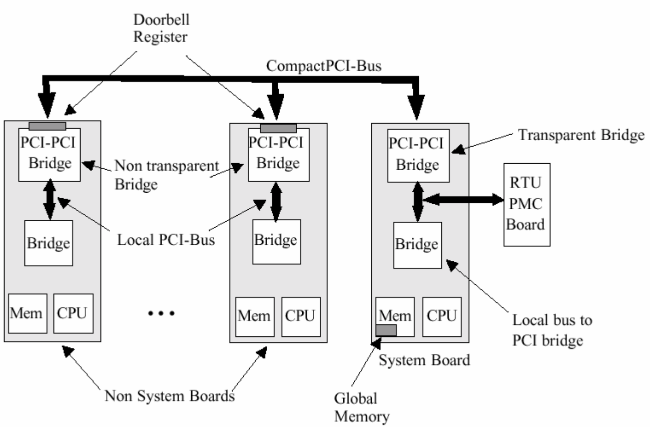

Figure 1, block-diagram of the SARA system

1.2 *Hardware Kernel*

The RTU is a special purpose hardware that performs real-time operating system functions. Examples of such functions are scheduling, time-management and semaphore handling. The task-scheduling algorithm is priority-based and supports preemption of tasks. A Virtual Communication Bus (VCB) [8] that uses the global memory on the system board is used for interprocess communication and synchronization between tasks in the system. The message-queues are priority based and if an incoming message has higher priority than the receiving task, the priority of the receiving task is raised during the message transaction. By this mechanism, the priority inversion problem [12] is avoided. For further details related to the RTU, please refer to [10].

1.3 *Software Kernel*

The software kernel is designed to act like the RTU, with respect to API and general semantics. The processor on the system board executes ordinary tasks as well as a kernel that handles RTOS functions for all tasks and processors in the system. The software kernel is written in PPC750 assembler, C and C++, compiled with GNU Compiler Collection version 2.95.1 [2] without optimization flags. Small parts of the code are compiler dependent, since inline assembler and macros are used. This paper contains a brief description of the central parts: scheduling, interprocess communication and clock management.

The software kernel is centralized; thereby the kernel on the system board schedules the whole system. The scheduling algorithm is priority driven like the algorithm in the RTU and supports pre-emption of tasks. Scheduling decisions are enforced through interrupts from the kernel on the master node to all processors. 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 Task Control Block (TCB) associated with each of its own tasks. The TCB contains a reference to the procedure that represents the task, a stack pointer and the processor registers needed to store and reload the state for the task to obey the decisions made by the scheduler.

The VCB connected to the RTU consists of a software layer containing the API against a hardware layer performing message transactions. The software kernel replaces the hardware layer while the interface layer remains untouched. 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.

Since the system board hosts the kernel, all decisions are made on the system board and therefore it is the only board that must provide timekeeping methods. This is a gain with the centralized implementation, because we can avoid costly clock synchronization and clock-tick interrupts on the slave boards. Clock-tick interrupts are created with the standard methods provided by the processor on the master board, and this is a point where the software kernel differs from the RTU. The RTU has configurable clock-tick resolution, however the default and used interval between two interrupts is 10 μs. But using such a resolution in the software version leaves no time to other activities than clock tick administration. Instead the interval between two ticks is set to 100 ms, this seems to be to 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. Therefore the kernel can be considered as event driven during the benchmarks and clock-tick at 100 ms or 5 ms does not affect the results. The gain with a 100 ms period is that it is possible to manually start different tasks and nodes during the same period.

2 BENCHMARK METHODS

The benchmark is aimed to serve as comparison between the hardware kernel and the software kernel. Both kernels have the same programming interface, although the software kernel has a slightly reduced version.

The benchmark series is built with own ideas and parts from the established benchmarks: Rhealstone [4], SSU [7] and Distributed Hartstone [3]. All tests have been repeated five times in order to compute the average result. In real-time context it is often the worst-case times that are considered, but this benchmark is performance oriented so it is the average times that are analyzed. The authours are avare of that five times is not enough to prove statistical confidence, but regarded as enough to consitute bases of this comparison. In the following sub-sections, the idea and experimental setup of each independent benchmark is explained.

2.1 Create Task

This test case is taken from the SSU benchmark and the time it takes to create a task is measured. The timekeeping starts when a task is going to be created and stops when the task has been created. Conditions that may affect the task creation time are the number of already created tasks and where in the system the task is located. 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.

2.2.1 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 the bandwidth in most systems depends of the message size, it is 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. The total amount of data is constant 10 kB and is divided into different number of messages in each independent test.

2.2.2 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, it is easier to take the two necessary timestamps on the same node to achieve a reliable result. 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.

2.2.3 RTOS Overhead

The method used in this benchmark is a common benchmark method and is for instance used in [1]. The comparison is based on executing an ordinary application and measuring the time it takes. As applications the classic problem The Traveling Salesman (TSP) was chosen. It is not a typical application executed by a RTOS, it should instead be considered only as synthetic workload. It is considered beyond the scope of this paper to describe the implementation to its fully, it should be enough information to know that the TSP implementation communicates through many small messages.

3 RESULT

Each benchmark test have been performed with several configurations, for instance we have run the Create Task test both on the master node and a slave node. This implies that we have too many graphs to present, so we show according to us the most important graphs. We will analyze all the results, including the graphs that are not presented in this paper.

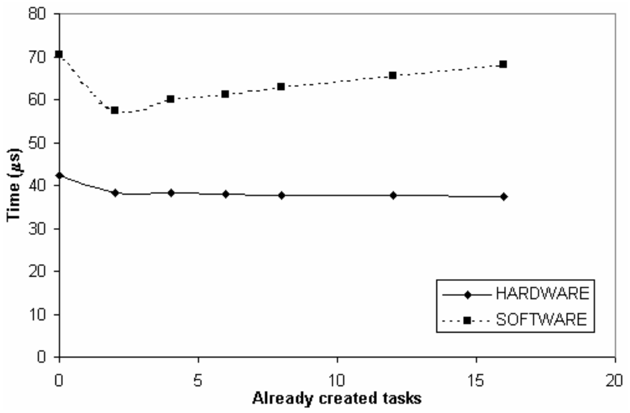

Figure 2, create task benchmark on slave node

3.1 Create Task

The results for the Create task benchmark on a slave node are shown in figure 2. The x-axis indicates the number of already created tasks in the system. The measured time is the time it takes to create one task. As we can see the hardware kernel is faster than the software kernel. A task that is created has to be registered to the scheduler, in both cases this is achieved by a synchronous system-call. By synchronous we mean that the caller has to wait for an acknowledgement that is delivered when the call is processed, so this benchmark shows that the software kernel has longer latencies. 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. In any case it is possible to create tasks with fixed latencies in the software case, but we have not focused on code optimization. The cache memory effects are also visible in the test, since the first call is more time consuming than the second. The RTU is inplemented in hardware and cannot be affedted by caches, but it still have TCB's and RTU interface code implemented in software, and these parts are affected.

The Create Task benchmark on the master node resulted in another relationship, the form of the graphs was essentially the same but the software kernel was faster than the hardware kernel. This is due to that in the software case the tasks are created completely locally on the master node, but when using the hardware kernel we suffer of PCI 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 algorithms implemented in parallel special purpose hardware often are faster than algorithms implemented in software. 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, has the highest priority in the system. When this benchmark was executed on 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.

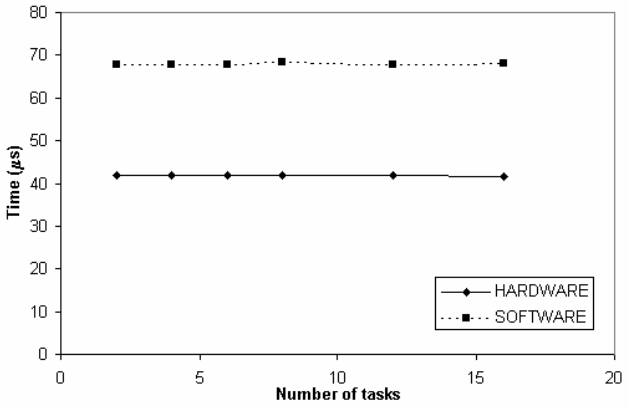

Figure 3, taskswitch benchmark on slave node

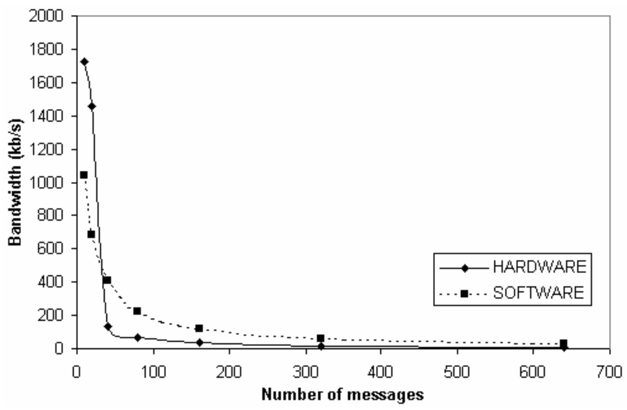

Figure 4, communication bandwidth benchmark, sender and receiver on master node

3.3 Communication Bandwidth

In figure 4 the bandwidth between tasks residing on the master node is compared, and figure 5 shows the bandwidth between a sending task residing on a slave node and a receiving task residing on the master node. This benchmark shows that the bandwidth between two tasks hosted by the master node is sometimes greater with the software kernel, but when communicating across node boundaries the bandwidth is always greater with the hardware kernel. The software kernel seems to be more efficient with increasing number of messages, in relation to the hardware kernel. The reason is that the software kernel executes the scheduling algorithm as a speculation after many system-calls, including send and receive message. These speculations can be considered as successful when the result is to switch tasks, otherwise they are waste of time. When there are lots of successful speculations the software kernel is efficient. Such a situation is when many send and receive are processed in a rapid sequence. The hardware kernel does these speculations also, but do not load the system processors because they are executed in parallel hardware.

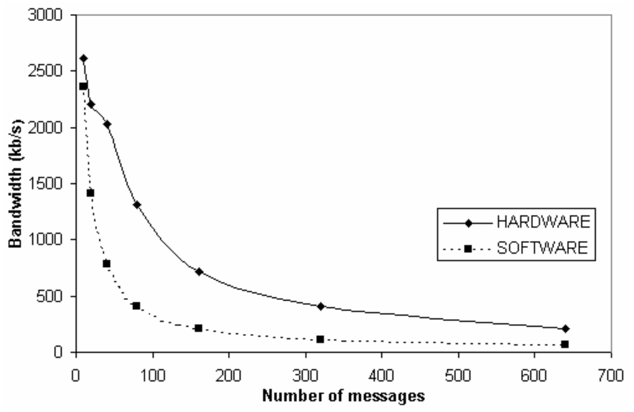

Figure 5, communication bandwidth benchmark, sender on slave node and receiver on master node

3.4 Communication Latency

Figure 6 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 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 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. This indicates that the hardware is a more deterministic solution, and in this way decreases the demands on real-time application programmers since it is easier to estimate bounded latencies that are not too pessimistic.

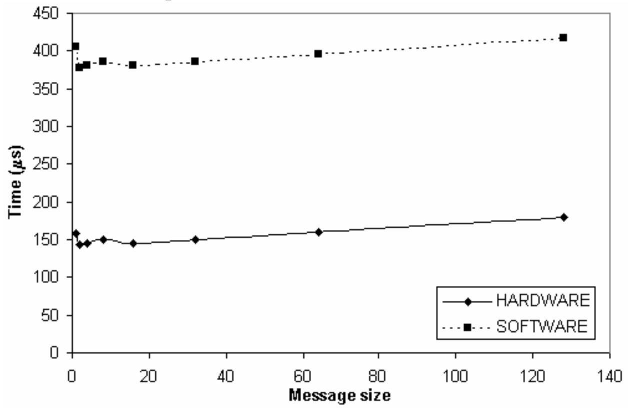

Figure 6, communication latency benchmark, sender on slave node and receiver on master node

3.5 RTOS Overhead

In figure 7, we can see when running the TSP application and exercising the whole system with messages as synchronization, the hardware kernel is faster. The reason why the hardware kernel is faster than the software kernel is larger communication bandwidth, shorter communication latencies and faster taskswitches.

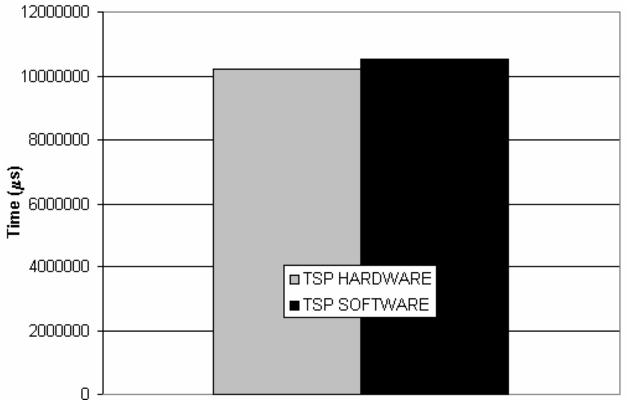

Figure 7, RTOS overhead benchmark, running TSP

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 lie in time-consuming system-calls from tasks on slave nodes. As long as tasks are executing on the system board, which also 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. Note that this optimization is not only possible to introduce in the software kernel, but also in some form in the hardware kernel. For a more detailed description of this kernel and complete benchmark results refer to [14].

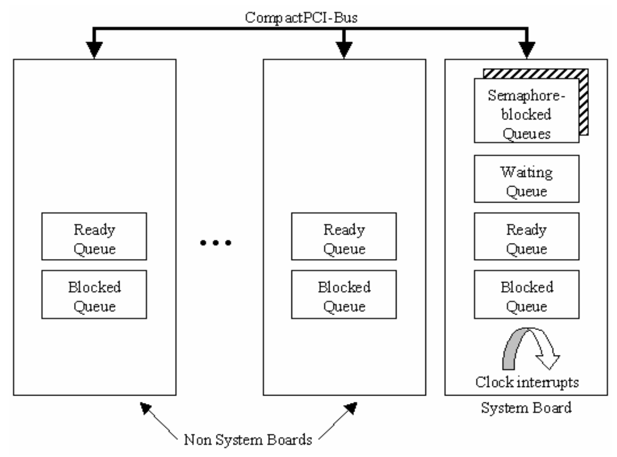

Figure 8, placement of scheduling queues in the optimized software kernel

A central part of a multiprocessor RTOS is the placement of the processor schedulers and different task queues. It is a choice between schedulers residing on the different processors, a centralized scheduler, perhaps with dedicated application and system processors or any combined approach. 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 [9] and in RT-Mach [13].

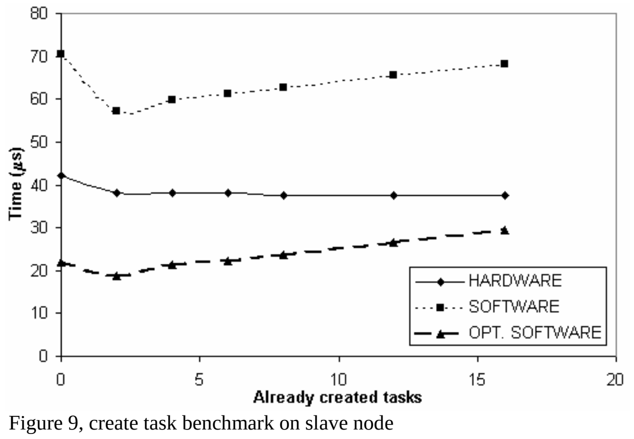

Figure 9, create task benchmark on slave node

Each slave board has a local ready queue and local blocked queue, while the system board also have semaphore queues and a waiting queue, see figure 8. The waiting queue contains both master and slave tasks sorted on wakeup time. On each scheduler invocation, the master checks the waiting queue for ready tasks. If a slave task becomes ready, an interrupt to the slave is sent. When a slave task executes a delay, an interrupt to the master is sent and the first task in the ready queue is dispatched from its local ready queue. Since the master board handles all timing, clock distribution and synchronization are avoided, i.e. the master node is the only node that has clock interrupts.

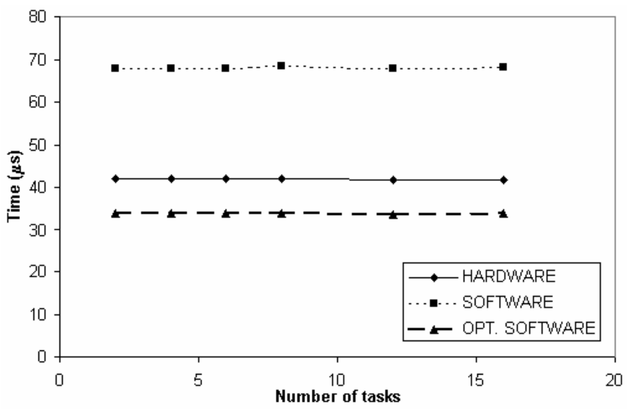

Figure 10, taskswitch benchmark on slave node

Figure 9, 10 and 11 show three examples of benchmark tests when the semi-distributed kernel shows clearly better performance than the centralized software kernel. The semi-distributed software kernel shows even better performance than the hardware kernel in figure 9 and 10. The result in figure 9 is expected, since the tasks are created locally and we do not need any communication with PCI devices. 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 already created tasks, and the reason to this behavior is the list management that increase with increasing number of tasks.

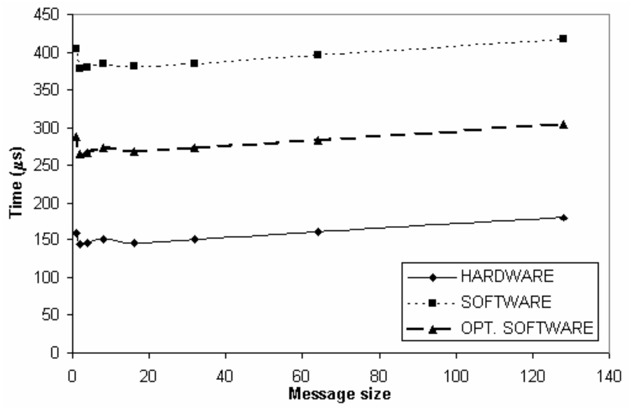

Figure 11, communication latency benchmark, sender on slave node and receiver on master node

The motivation to figure 10, when the semi-distributed software kernel is faster than the hardware kernel, is the taskswitches that can be handled locally on a slave node with the optimization. Each node manages its own queue of active 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 11, the optimized software kernel has shorter communication latencies than the centralized software kernel, but the hardware kernel still shows the best result.

The optimized software kernel is not fully distributed, since the semaphore protocol and the IPC is still centralized. The semi-distributed software kernel will probably not perform better than the hardware kernel in this benchmark even if it was fully distributed, because a fully distributed kernel would require more synchronization in order to keep the different nodes apprehension of the system-state consistent.

5 DISCUSSION

In this paper we presented the results of a comparison between a centralized software kernel and the hardware kernel RTU, both used with the same multiprocessor system. The comparison was performed with different benchmark series with importance to real-time. We showed that the performance of the RTU was in general better than the corresponding software solution. A great advantage with the RTU is the execution of scheduling decisions and other operating system functionality that does not load the applications processors. A hardware implementation also makes it easier to create bounded execution times for system calls, which is desired in real-time systems when guaranteeing deadlines of events. But notice that since scheduling decisions and many other system calls does not load the processors, the gain with a hardware implementation becomes greater with more complicated algorithms and increased number of tasks.

We also showed an improvement of the software kernel that in addition could be applied to the RTU. By moving intelligence from a centralized source to the system nodes, this implied that some decisions could be made faster and we could utilize the natural locality of data close to each node. The main advantage with the improved software kernel in comparison with the RTU is that some system calls can be executed in parallel (on different nodes). The improved software kernel showed better or equal performance results in comparison with the centralized software kernel. In some benchmark tests the semi-distributed software kernel showed even better performance than the RTU.

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. A better target system should therefore in some way provide 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. Secondly since the scheduling algorithm does not load the application processors, it should be possible to use a really fancy and effective algorithm that hardly cannot be used in software implementations.

ACKNOWLEDGEMENTS

Thanks go to all personnel at the Computer Architecture Laboratory, for pushing and realizing this research. A special thanks goes to Leif Enblom, for many fruitful discussions of ideas. Raimo Haukilahti and Tomas Lennvall deserve special appreciation for reviewing this paper.

REFERENCES

[1] J. Furunäs, Benchmarking of a Real-Time System that utilises a booster, International Conference on Parallel and Distributed Processing Techniques and Application, 2000.

[2] GNU Compiler Collection, http://www.gnu.org/software/gcc

[3] 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, 1991.

[4] R. Kar and K. Porter, Rhealstone - a Real-Time Benchmarking Proposal, Dr. Dobbs' Journal, February 1989.

[5] 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 Asia and South Pacific Design Automation Conference, Japan, January 2003.

[6] L. Lindh, T. Klevin and J. Furunäs, Flexible Multiprocessor computer Systems, In CAD & Computer Graphics, December 1999.

[7] K. Low, S. Acharya, M. Allen, E. Faught, D. Haenni and C. Kalbfleisch, Overview of Real-Time Kernels at the Superconducting Super Collider Laboratory, Particle Accelerator Conference, 1991.

[8] 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, June 2000.

[9] K. Ramamritham and J. A. Stankovic, The Spring Kernel: A new paradigm for Real-Time Systems, IEEE Software, May 1991.

[10] RF RealFast AB, Real-Time Unit, A New Concept to Design Real-Time Systems with Standard Components, RF RealFast AB, Dragverksg 138, S-724 74 Västerås, Sweden, E-mail: realfast@realfast.se, 2000.

[11] L. Rizvanovic, Comparison between Real time Operative systems in hardware and software, Masters' thesis presented at Mälardalen University, Västerås, Sweden 2001.

[12] L. Sha, R. Rajkumar and J.P. Lehoczky, Priority inheritance protocols: An approach to real-time synchronization, IEEE Transactions on Computers, September 1990.

[13] H. Tokuda, T. Nakajima and P. Rao, Real-Time Mach: Towards a Predictable Real-Time System, In Proceedings of USENIX 1st Mach Workshop, October 1990.

[14] M. Åkerholm and T. Samuelsson, Design and Benchmarking of Real-Time Multiprocessor Operating System Kernels, Masters' thesis presented at Mälardalen University, Västerås, Sweden, June 2002.