Design and Benchmarking of Real-Time Multiprocessor Operating System Kernels

Mikael Åkerholm and Tobias Samuelsson,
Masters' thesis

The Department of Computer Science and Engineering,
Mälardalen University, June 2002.

Foreword

We are cheerful to present our work to candidate for the degree of Master of Science in Computer Engineering. The work has been divided into two parts of equal weight and therefore this examination paper contains two papers that can be studied independently. The first paper is a survey and evaluation of existing real-time multiprocessor kernels. In the opening sections of the second paper we introduce an own implemenatation of such a kernel. The main part of the second paper presents a benchmark series, wich purpose is to compare our implemenatation with an existing hardware implementation.

Table of Contents

A Survey and Evaluation of Real-Time Multiprocessor Operating System Kernels

Mikael Åkerholm and Tobias Samuelsson
Preparatory study for masters' thesis.

ABSTRACT

Multiprocessor architectures, operating systems and real-time technologies are all interesting and highly advanced topics. Real-time demands inject an additional correctness criterion into computer systems. It is not just the result that is important, timing issues also have to be considered. A multiprocessor system is able to provide more performance than today's fastest single processor solution, and it is often in multiprocessor systems the latest technology is introduced. The operating system is without hesitation the most important software of all system programs in a real-time multiprocessor system.

The main limitations and concerns reported so far from the rather young research area of real-time issues in multiprocessor systems, mainly consists of schedulability problems and anomalies with old single processor scheduling algorithms. The possibilities with moving real-time applications onto multiprocessor platforms on the other hand weight more; scalability, robustness, more and cheaper computing power are the general advantages. That real-time operating systems with multiprocessor support will become a desired product in the near future is a highly realistic prediction, since the requirements and complexity of real-time applications increases rapidly.

In this survey we review and evaluate both commercial and research solutions that addresses all three attention-grabbing areas in a homogeneous manner. The paper first identifies the major design goals and key issues in multiprocessor real-time operating systems, to follow up with a set of case studies where the identified issues are unveiled.

The Department of Computer Science and Engineering,
Mälardalen University, March 2002.

Table of contents

Introduction

The purpose of this survey is to review some of the major concepts concerning Real-Time Operating Systems (RTOS) aimed for parallel or distributed hardware platforms, roughly reflecting the state of the art in the early beginning of the 21st century. The work was carried out as a preparatory study to a masters' thesis project at the department of computer science and engineering at Mälardalen University.

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 interval. In ordinary systems it is typically the value of the output that determines the correctness, in real-time systems we also have the timing issues to consider. A correct output produced to late or to early could often be useless, or even dangerous. In [STA00], it is concluded that 98% of the microprocessors produced in 1998 where for real-time or embedded systems use, and towards the end of 2000 it is predicted to be up against 100%. Examples of applications that require real-time computing include:

- Vehicle control systems
- Industrial automation systems
- Telecommunication systems
- Military systems
- Railway switching systems
- Forestry automation systems

The purpose with an RTOS is to simplify the development process of real-time systems, by providing an interface with a higher abstraction level than the bare hardware architecture offers. The most distinguishing features with a RTOS compared to an ordinary Operating System (OS) are the deterministic and predictive time management. An ordinary OS often tries to perform all actions with average throughput in mind, this methods minimizes the average case at the cost of the worst-case. An RTOS must try to handle all system calls and task switches in a predictive and analysable manner, i.e. often by providing a known worst-case behaviour.

The demand for distributed or parallel hardware architectures in real-time systems as other systems is mainly due to the fact that the applications requires more than a single processor can offer. Complexity in real-time systems increases more rapidly than the performance of the microprocessors increases. A special demand exists in some real-time systems; since many real-time applications are distributed by its nature it is motivated to use a distributed control system. Consider industrial manufacturing pipelines with several robots, the robots is performing independent work but it is easy to understand that the robotics must be synchronized in some way. Many slightly different interpretations of the difference between a parallel and a distributed hardware platform exist. In this survey the difference is defined by the tightness of the connections in the

machine, if a processor in the machine requires at least one common and shared resource or device to be working properly it is a parallel machine. In a distributed architecture it is assumed that a processor can be taken out together with its own private devices and resources, without ruin the possibility to use the rest of the system and the detached processor as two independent computers. The term multiprocessor is sometimes used to address only parallel computers, but in some cases both distributed and parallel architectures, it should be no problem to distinguish between the cases when that is necessary. Finally the term multi computer is a true distributed architecture.

The authors are not aware of any surveys with the same objectives as this, but it exists many papers and articles specialized towards a narrower subject covered by this survey and surveys with slightly different objectives. In [YAN97], the authors presents a survey directed towards RTOS, but no RTOS is described in detail it is rather a design issue survey. A chapter in [BUT97] presents some RTOS, described as a survey, but the selection criteria differs from ours. The survey is not directed through multiprocessor or distributed hardware platforms, although some RTOS with support for such target platforms are mentioned. A operating system survey that is directed towards multiprocessor platforms but not real-time kernels is [GOP93].

The outline of the survey is as follows. In the design issues section we will try to discover and explain the most important issues in multiprocessor RTOS and how they can be compared to each other. The next section is a case study of a set of selected RTOS with multiprocessor support, each RTOS central concepts will be presented and an evaluation will follow every RTOS description. The final section concludes the survey and discusses future work.

Design issues

In this section a framework for description and evaluation of real-time multiprocessor operating systems are presented. We are aware of that many important aspects of an operating system have been left out, mainly due to the need of limit the survey. The issues we have left out include areas such as structure and I/O. The structure of operating systems is typically referred to as monolithic, layered or client-server based. Areas sorting under I/O are disks, user interactions and environment sampling. I/O, structure and most of the other issues we are not addressing are described in [TAN92]. The three central issues, which describe almost all essential constructs in a real-time operating system, are:

- Scheduling
- Memory management
- Interprocess communication

In the first part of this section an explanatory presentation of the three mentioned issues is given, together with an introduction to some of the presented algorithms and ideas designed for usage within the three specific areas. In the end of the section benchmarks as method for practical comparison of RTOS:es are reviewed. Although this survey only evaluates different RTOS:es theoretically, a well and fair performed benchmark is often the most reliable basis for a comparison.

Scheduling

The first issue to address when talking about scheduling algorithms for real-time systems is the task management. With task management, we basically mean attributes associated to tasks. This is an important issue since different scheduling algorithms require different task attributes, and different real-world problems are often easiest to express with different task attributes.

Task management

The basic terms concerning tasks and their attributes are easiest explained by an example. A real-time system is a system that interacts with the environment by performing pre-defined actions on events within a certain time. The action of a special event is typically defined in a *task* and within a certain time forms the *deadline* for a task. A real-time task can be classified as *periodic* or *aperiodic* depending on its arrival pattern and as *soft* or *hard* based on its deadline. Tasks with regular arrival times are called periodic and tasks with irregular arrival times are aperiodic tasks. Each of the hard tasks must complete execution before some fixed time has elapsed since its request, i.e. finish before its deadline. Soft tasks do not have any demands in time, which means that soft tasks do not have any deadlines. Other attributes associated with a real-time task that usually are mentioned in scheduling and task management contexts are for instance:

- *Worst Case Execution Time* (WCET) – is the maximum time necessary for the processor to execute the task without interruption.
- *Release time* – is the time at which a task becomes ready for execution.

- *Sporadic task* – is an aperiodic task with a known *minimum interarrival time* (MINT), that is the minimum time between two activations.
- *Precedence constraints* – are constraints about the order of task execution, some tasks must execute in a defined order.

In almost all real-time operating systems, the information about a task is stored in a data structure, called the *task control block* (TCB). The TCB typically contains the task state, the desired or required set of attributes from those defined above, a pointer to the procedure that represents the task and a stack pointer. If the scheduling algorithm is preemptive the TCB must contain everything that is needed to store and reload the state for the task (registers etc). The rest of the content in the TCB varies from one RTOS to another; many of the special features of an RTOS will affect the TCB.

In almost all operating systems, a task can at any point in time be in one of the following *states: Running, Ready* or *Waiting*. The states may have different names in different operating systems, but the semantic meaning is always the same. Additional states exist in most operating systems, but these three states are the most important and basic states. Only one task per processor can be in the Running state at any instance in time, it is this task that currently uses the processor. A task that has all that is needed to execute, but for any reason waits for another task is said to be in the Ready state. Finally a task that misses something, a shared resource, waiting for an external event or waiting for its release time etc is said to be in the Waiting state.

Scheduling algorithms

Scheduling real-time systems are all about guaranteeing the temporal constraints (deadlines, release times and so on). Two main approaches for real-time scheduling exist. On one side we have off-line scheduling, where all scheduling decisions is calculated by the system designer before runtime and stored in a runtime dispatch table. The other approach is on-line scheduling, where all scheduling decisions is calculated by the scheduling algorithm at run-time. Throughout this text both the terms on-line and off-line scheduling as well as runtime and pre-runtime scheduling are used.

Which algorithm that is best suited depends on the scheduling problem to solve. For instance algorithms based on the off-line scheduling approach are more deterministic, and it is easy to prove and show that a task will meet its deadline since the methods in some sense applies the “proof-by-construction” approach. Off-line scheduling methods can also solve tough scheduling problems with high CPU utilisation and complicated precedence constraints. An off-line scheduler can spend long time in finding a suitable schedule, since the system is not up and running and no deadlines will be missed during the search, but at run-time the only scheduling mechanism we need is a simple dispatcher that performs a table lookup. On the other hand, we need to know almost everything about the systems

timing constraints to be able to create a suitable schedule before run-time. Algorithms based on the on-line scheduling approach in general offer higher flexibility and are better to adopt changes in the environment, at a higher cost for calculating on-line scheduling decisions and they cannot offer the same degree of provable determinism. The higher flexibility offered by on-line scheduling approaches cause that most existing algorithms that schedule aperiodic tasks use the on-line scheduling approach.

Most off-line scheduling algorithms that have been implemented are based on some kind of search technique with applied heuristics. Examples of such scheduling algorithms are *A** and *IDA*,* they are analysed and described in [FOH94] (chapter 3). Other examples are *branch-and-bound* [RAM90] and *meta* [FOH97]. According to [JXU93] most practitioners working on safety-critical hard real-time systems, that uses the off-line scheduling approach have been observed doing the schedules by hand instead of letting a computer search for a schedule, the result (the by hand created schedule) is often hard to verify and maintain. Examples of on-line scheduling algorithms that handle periodic tasks are *Earliest Deadline First* (EDF) [LAY73]*,* and *Rate Monotonic* (RM) [LAY73], both introduced in the early seventies. Most on-line scheduling algorithms use one of these two algorithms as base algorithm. A sample of on-line algorithms that handles both periodic and aperiodic tasks is *Sporadic Server* (SS) [LEH93], *Robust Earliest Deadline (RED)* [BUT93] and the *Total Bandwidth Server* (TBS) [BUT94].

Multiprocessor scheduling

The multiprocessor scheduling anomalies that do not exist in the single processor case and must be considered when constructing a multiprocessor scheduling algorithm are in brief that; the schedule length can be increased by:

- Increased number of processors
- Reduced task execution times
- Weaker precedence constraints

As a consequence when using multiprocessor platforms, algorithms that have been showed to be optimal in any sense on a uniprocessor system are often not optimal in a multiprocessor system. For instance EDF has been showed to be an optimal algorithm, under certain conditions in [DEZ74] for a uniprocessor system. In the multiprocessor environment, EDF and other under any conditions optimal on-line algorithms fails to be optimal, in [MOK83] it is showed that no algorithm can be optimal in an on-line scheduled multiprocessor system, with or without precedence and mutual exclusion constraints. Multiprocessor architectures combined with real-time scheduling are therefore a delicate problem and an ongoing research area. It is clearly easier for an off-line scheduler to be optimal, since the time that such a scheduler may consume is potentially unlimited. For instance consider an off-line scheduler that tries all possible combinations on all processors, if a schedule that solves the problem exists this scheduler will find it.

Hence we can say that such a scheduler would be optimal, but it will not be practically useable since the time required finding the schedule also are not unlimited. When implementing an off-line algorithm for finding a schedule that solves a specific scheduling problem, it is easy to understand that the total time spent finding a schedule simplified is a formula as below.

*(Time not finding a schedule * Number of failures) + (Time for finding * 1)*

Through a quick look at the formula we can see that the biggest time is spent in (not finding any schedule), hence it is at least as important for an off-line scheduling algorithm to fast detect that there is no solution as actually finding a schedule. This is the main reason why no one uses our intuitive, although clearly optimal suggestion of pre-runtime scheduling algorithm.

In [STA98] the authors identifies three phases in the scheduling procedure of a multiprocessor system, and the text is directed towards on-line scheduling. The phases also seem to be suitable for an off-line scheduler. In the off-line scheduler case phases 1 and 2 would be executed off-line, and only phase 3 would be executed on-line.

1. Allocation – the assignment of tasks and resources to the appropriate nodes or processors in the system.
2. Scheduling – ordering the execution of tasks and network communication such that timing constraints are met and the consistency of recourses is maintained.
3. Dispatching – executing the tasks in conformance with the scheduler's decisions.

To continue the opened discussion and get a little bit more concrete examples on a run-time, a pre-run-time and a combined approach will be given. However these algorithms assumes a lot of parameters and may be targets for some modifications and simplifications before they are applicable in the real world.

An on-line method

A holistic approach based on EDF is presented in [STA98]; the method is focused on how to perform the analyses so that we can guarantee deadlines on events. In the end, real-time scheduling is about being able to guarantee the temporal behaviour of a system. The method is an adoption of the holistic approach based on static rate monotonic priorities developed in [TIN94]. The assumptions limits the algorithm to be used anywhere without modifications, for instance tasks are statically allocated to a node, and a token-ring based communication medium is assumed. The original method, from [TIN94] has been adopted for use on other communication networks as well. For instance in [NOR00] examples on how to use the method with rate monotonic priorities and a CAN bus is showed.

Each node has a ready queue that is scheduled by EDF and the key to overcome the otherwise hard global analysis is *attribute inheritance*. Every message inherits two of the sending tasks temporal attributes, the period and the *release jitter*. The release jitter is simply defined to be the difference between the sender tasks earliest start-time and worst-case response-time. The period that is inherited is simply the period of the task. Based on that simple notation it is possible to perform *end-to-end* response time analysis of a transaction. The iterative analysis formula can be expressed as following:

1. Set all initial release jitter to zero.
2. Compute the worst-case response-time for tasks and messages for each host processor and network separately.
3. Compute the worst-case jitter that the response-times in step 2 can generate, go to step 2 if the jitter does not change, else we are done.

Each of the different computations in the steps 1, 2 and 3 are quite simple summations and can be found in [STA98], however they are iterative and with many tasks it may take a while to achieve convergence.

An off-line method

A heuristic method for multiprocessor systems dealing with hard periodic tasks is the *Slack method* [ALT98]. Many of the offline methods only deal with this task type. The method is divided into two sub-steps named Graph Reduction and CP-Mapping. In reality we can conclude that the division into three steps easiest describes the algorithm, since two sub steps are included in the Graph Reduction step. Here comes a brief overview of the method, for a complete and detailed description with proofs and formal mathematical definitions refer to [ALT98].

During the initial Graph Reduction step, transactions (consisting of several tasks) are treated separately. Each transaction graph is taken as input to the step.

1. First the Graph Reduction step tries to reduce each transaction graph to a CP graph, by a method named *critical path clustering*. Critical path clustering is about finding the *critical path* of a transaction. The critical path is in short defined as the path with the smallest slack, and the smallest slack of a path is in turn defined as the minimum slack of all vertices of a path. The slack is determined as the time between a tasks deadline and the summation of the computation time and the release time, i.e. the time a task can be delayed without missing its deadline.
2. Secondly the Graph Reduction step tries to fill slack intervals in the generated CP graphs, with other paths. This is an initial step to let different paths execute on the same processor.
3. Finally the CP-Mapping step is executed. In this step the different CP-graphs, which represents one transaction each are mapped onto a physical processor. A CP-graph is a drastically reduced form of the initial problem formulation; the two reductions executed in the Graph Reduction step are successful in most cases. Hence an optimal algorithm is used for the

mapping in this step, by optimal we mean an algorithm that finds a solution if a solution exists. It is simple to construct an optimal algorithm and here *backtracking* is the choice, i.e. if any deadlines are missed during an assignment, backtrack to the next possible assignment.

A hybrid method, integration of off-line and on-line

A detailed explanation and description of the *Slot Shifting* algorithm can be found in [FOH95], here comes an introductory description. The Slot Shifting algorithm is an algorithm that combines both on-line and off-line scheduling. Easily we can state that the Slot Shifting algorithm handles periodic tasks with the off-line approach and that the aperiodic tasks are on-line scheduled using the remaining capacity. The algorithm uses time slots as the basic abstraction for the system time; a slot is an interval of time with a fixed length.

The off-line preparations for slot shifting is basically about creating and ordinary off-line schedule and determining spare capacities, i.e. how many time units (slots) we can move a tasks execution without jeopardizing its deadline. The task assignment to different nodes in the system is done manually or with another algorithm, task assignment is not specified in the original algorithm. Further on the network between the different nodes are assumed to be a time slotted architecture, i.e. token-ring, TDMA etc. The messages on the network is integrated in the off-line scheduling procedure. The time model is said to be discrete, since both the communication medium and the scheduling algorithm uses the same globally synchronized time slots.

The online scheduler is invoked after each slot and checks whether we have any aperiodic tasks to schedule. If aperiodic tasks are pending, the scheduler performs an on-line guarantee test. The scheduler tries to shift the execution of the periodic tasks as much as possible without risking their deadlines, using the off-line calculated spare capacities. If we have enough spare capacity to serve the aperiodic requests, we execute them else they are rejected.

Memory management

When implementing a suitable memory abstraction in the operating system, the first thing to study is what the hardware offers. Many attempts for classification of hardware memory systems exist. In this survey it is the terms *Uniform Memory Access* (UMA), *Non Uniform Memory Access* (NUMA) and *NO ReMote Access system* (NORMA) that are used.

- In UMA architectures, memory access times are equal for all processes to the whole address space. A common design technique for those systems is processors connected to a bus, and a global shared memory connected to the same bus.
- The NUMA systems also offers a single shared address space that is visible for all processors, but the access times for a processor to different

memory region differs. A common design technique is processor boards with own memory modules attached to a shared bus.

- NORMA architectures on the other hand do not offer a global shared address space. Each processor just accesses its own address space. Typically these architectures consist of loosely coupled independent computers connected through ordinary *Local Area Network* (LAN) technologies, often referred to as clusters. But bus based NORMA systems are also common.

The software layer

Usually the operating system implements some memory abstractions above the abstraction offered by the hardware. Typically *protection*, *virtual memory*, *allocation* and *de-allocation* primitives are implemented in the operating system. Although some parts of the protection and virtual memory support often are implemented in hardware, it is the operating system that configures the hardware and implements own abstractions. Protection is about associating attributes with memory segments like read-only access or read-write access. Virtual memory means that we use more memory than is physically available. This is accomplished by introducing another bigger but considerably slower level in the memory hierarchy, i.e. a disk device. Allocation and de-allocation is about allocating and de-allocating memory dynamically during run-time. Another example of typical implementation in the software layer is another memory abstraction than the hardware offers, for instance NUMA from NORMA, an example of this is described under the interprocess communication section.

Real-time kernels

Real-time kernels memory management services are often simple and primitive, almost non-existent. Virtual memory are for instance often considered as a dangerous and unreliable feature, and therefore not implemented. This is easy to understand because of the unpredictable long access times to a memory page on a disk device. It is easy to agree with the authors of [BEN01], when they state that most of the proposed software algorithms and hardware support in commercial processors for memory management are optimized for average performance, and not for predictable worst-case behaviour. Dynamic allocation of memory is also avoided, since the behaviour is both temporal and functional unpredictable. We cannot guarantee that memory will be available, and not how long it will take for the kernel to find the necessary amount of memory. Another task may allocate memory, but some semantic fault may cause the task to never de-allocate the memory. Such tasks are said to leek memory. To be completely secure against memory leaking tasks, real-time kernels often do not implement dynamic memory handling. According to this it sounds like memory management in real-time systems is rather boring, but it is not, some more fancy techniques have been presented recently.

In [BEN01], the authors develop theories for integration of virtual memory and the other functions a modern processor offers through its Memory Management Unit (MMU). Their approach is motivated by the fact that many safety-critical real-time applications consist of both hard and soft real-time tasks, i.e. hard or soft deadlines. Suitable processors for their approach are processors, which have support for some kind of partitioning of the address space. The introduction of *real-time address spaces* is the proposed solution. Tasks with soft deadlines can utilize all fancy mechanisms offered by the processor, and resides within an ordinary address space. For instance we can have a monitor task with a soft deadline that sometimes need virtual memory. While tasks within a real-time address space does not use any virtual memory or caches. Those tasks need a predictable memory access time. The definition given in [BEN01] of a real-time address space is quoted below.

"An address space is a real-time address space if: The worst-case execution time of the virtual-to-physical translation for all pages that do not result in a page fault in the address space is known."

The principle for an implementation of real-time address spaces is described for PowerPC, MIPS and StrongArm in [BEN01].

Research efforts that pay attention to real-time systems and dynamic memory management, currently seems to be driven by the need of virtual and dynamic memory in programming languages like java and C++. It is possible to use a hardware module for memory allocation; in this way we could get predictable allocation time. The *Active Memory Module* (AAM) [SRI00] is such a device. All allocations are bounded to 14 clock cycles; a fast and constant allocation time makes this device suitable for real-time applications. The AAM module was developed for the use of Java in real-time applications. The garbage collection and dynamic memory allocations naturally existing in a Java program jeopardizes the execution times. The garbage collection and dynamic allocations in Java utilizes only heap memory, so the AAM module only administrates the heap memory. The RAM module and the AAM module is physically separated, an AAM module consists of memory modules and a controller.

Interprocess Communication

Communication is a central component in any operating system; especially in multiprocessor operating systems communication has the same importance as the instruction set has in a uniprocessor system. Co-operating processes or threads often communicate and synchronize. The execution by one certain process can affect another process by communication. Generally there are two different types of interprocess communication (IPC) for multiprocessor systems: message passing and shared memory.

Each of these communication models was developed for a different class of multiprocessors, reflecting the assumptions about the underlying hardware. The shared memory model is typically associated with tightly coupled shared memory multiprocessors (UMA, NUMA), while the message passing model is typically associated with distributed memory multiprocessors or distributed multicomputers (NORMA), which is a network of workstations.

Performance comparisons between message passing and direct use of shared memory have been done in [LEB92]. The authors claim that both communication models have performance advantages and the factors that influence the choice of model may not be known at compile-time. Their conclusions were mainly that the advantage with the shared memory model is load balancing, while the main advantage with the message-passing model is locality. Today both communication models are in use on both classes of machines.

In a shared memory multiprocessor, message passing is a more abstract form of communicating than accessing shared memory locations. Message passing subsumes communication, buffering and synchronization. Multiprocessor operating systems have experimented with a large number of various communication abstractions [GOP93], including *ports*, *mailboxes*, *links* etc. In other words these abstractions are kernel-handled message buffers. The two basic message-passing primitives in such abstractions are *send* and *receive*. Sends and receives may provide synchronous and/or asynchronous communication and they may be blocking (a process does not wait for the communication to complete) or non-blocking (a process waits for the communication to complete).

Computing large or complex data structures may be inefficient or difficult using message passing in a distributed memory multiprocessor or distributed multicomputer. A solution to this problem is to use coherent distributed memories, based on a message passing mechanism. As an example of such a system we refer to *Mirage* [FLE89]. Mirage is a protocol that hides the distributed memory network boundaries for the application programmers, i.e. used when the shared memory model are desirable on a distributed memory multiprocessor.

The difference between communication in a real-time system and a conventional (not real-time) system is their different system requirements. The major desirable characteristic in a conventional system is performance, expressed in throughput, average response time or latency. Real-time tasks must not only produce the correct results, they also have to be produced *on time*. However, this cannot be achieved by just providing a fast communication media. In conventional systems, IPC can be unpredictable due to the potentially unlimited blocking time of applications synchronizing or waiting for messages. So the communication in real-time systems must not only be fast, it must also be predictable and deterministic, in order to guarantee hard tasks by using some form of analysis before the start-up of the system (pre-runtime). A method to make IPC efficient is

to implement hardware support for such functionality [FUR01]. The main drawback with IPC mechanisms implemented in special purpose hardware is the cost of extra hardware.

Resource reclaiming

A shared resource is a software structure that can be used by more than one process to advance its execution. Any operating system that supports shared resources must provide any resource access protocol to ensure consistency of the data in the shared resources. In other words the operating system must guarantee mutual exclusion (for instance by providing semaphores) among competing tasks, so if two or more tasks have resource constraints, they must be synchronized. Intertask communication is a critical issue in real-time systems, since the fact that shared resources may cause *priority inversion* [BUT97] and unbounded blocking on processes' executions. A priority inversion occurs when a high-priority task requests a shared resource that is used by a low-priority task. This means that the high-priority task must wait for the low-priority task to finish its execution, i.e. the high-priority task executes with a lower priority than the low-priority task. Since the duration of priority inversion in general is unbounded, this jeopardizes the predictability of tasks execution. There are many different approaches to solve the priority inversion problem. The Priority Inheritance and the priority ceiling protocol are two semaphore protocols where the maximum blocking time for a task can be calculated.

Priority Inheritance Protocol

The priority inheritance protocol, proposed by Sha, Rajkumar and Lehoczky in [LEH90], minimizes the blocking time of a high-priority task by increasing the priority of the low-priority task when the high-priority task becomes blocked. A task can only hold semaphores during the execution, i.e. when a task has finish its execution it is not allowed to hold any semaphores.
Definition of the Priority inheritance protocol:

1. Task A executes and tries to obtain semaphore S. If semaphore S is locked, task A is blocked because it cannot lock the semaphore, if not task A locks semaphore S. When A unlocks S, the task with the highest priority that is blocked by A becomes ready.
2. Task A uses its assigned priority during execution unless it has locked semaphore S and blocks higher-priority tasks. If task A blocks higher-priority tasks, it will execute with the highest priority of the tasks that is blocked by A (A inherits the highest priority). When A unlocks semaphore S, A will return to its original priority.
3. Priority inheritance is transitive. Assume three tasks A, B and C in descending priority order. If task C blocks B and B blocks A, task C will receive task A's priority.

A drawback with the priority inheritance protocol is that it does not deal with blocking-chains and deadlocks.

Deadlock

Deadlocks are best described with an example. Assume two tasks A and B that both want semaphores S1 and S2. Task A has high priority and task B has low priority. Task A attempts to lock semaphore S1 before S2, while B attempts to lock S2 before S1. We have the situation illustrated by Figure 1.

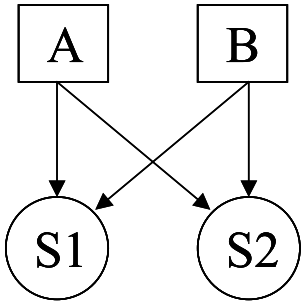


Figure 1, A and B are tasks, S1 and S2 are semaphores.

A deadlock scenario:

1. All semaphores are free.
2. Task B locks S2.
3. Task A preempts B and locks S1.
4. Task A attempts to lock S2, but S2 is locked so A is blocked.
5. Task B attempts to lock S1, but S1 is locked so B is blocked.
6. Both tasks are blocked, waiting for each other. A deadlock has arisen.

A solution to the deadlock problem is to use the Priority Ceiling Protocol, which is described in the next section.

Priority Ceiling Protocol

The priority ceiling protocol [LEH90] minimizes the blocking time of high-priority tasks by preventing blocking-chains. Furthermore this protocol prevents deadlocks.

Each semaphore S is assigned a priority ceiling. The value of the ceiling is equal to the priority of the task that has highest priority among those tasks that want to lock semaphore S.

Definition of the priority ceiling protocol:

1. Task A executes and attempts to lock semaphore S. Let S* be the semaphore with highest priority ceiling among those semaphores that currently are locked. Task A cannot lock semaphore S if it is already locked, so A is blocked. If semaphore S is not locked, A locks S in case A's priority is higher than the priority ceiling for S*. The task with

highest priority among those tasks that are blocked by A, becomes ready when A unlocks S.

2. Task A uses its assigned priority during the execution in case it has not locked a semaphore S and blocks higher-priority tasks. If task A blocks higher-priority tasks, it will execute with the highest priority of the tasks that is blocked by A (A inherits the highest priority). When A unlocks semaphore S, A will return to its original priority.

Interrupts

Interrupts generated by external I/O devices causes a big problem for the predictability of real-time systems, because if they are not properly handled they may introduce unbounded delays during process execution. In most operating systems, the arrival of an interrupt starts the execution of an interrupt service routine (driver), dedicated to the management of the certain device. With this approach, all hardware details of the device are encapsulated inside the driver. An interrupt service routine should be as small as possible, i.e. handle the device that generated the interrupt in order to minimize the blocking time of ordinary tasks. Often this means that the interrupt service routine reads or writes a value to the device and then just acknowledges the interrupt. Typically the interrupt service routine often sends the received value to a task that further serves the value.

In many operating systems, interrupts are served using fixed priority schemes, i.e. each driver is scheduled based on a static priority, which is higher than the process priorities. This is motivated by the fact that I/O devices normally deals with real-time constraints, whereas most application programs do not. An important real-time issue with interrupt handling is systems maximum interrupt latency, i.e. how long time can the system turn off all interrupts (interrupt disable).

Next a brief description three different approaches will be given.

1. This approach eliminates interrupt interference by disabling all interrupts except the timer interrupts, which are necessary for the system. Since no interrupts are allowed, application tasks handle the external devices through polling. This strategy requires that the application tasks have direct access to the I/O devices that they want to handle. All device-dependent functions can be encapsulated in a set of library functions; so different application tasks can include different libraries. An advantage with this function is that the kernel does not need to be modified when new I/O devices are added. The main drawbacks with this approach are the response times of external devices and the processor utilization on I/O operations. A system that has adopted this approach is RK, which is a hard real-time kernel for multi-sensor applications [LEE88].

2. The second approach resembles the previous one, because all interrupts except from the timer are disabled. Instead of using application tasks for handling devices, this method uses dedicated kernel routines to serve devices. The timer periodically activates the kernel routines. This strategy eliminates the unbounded delays due to the execution of the interrupt drivers and all device-dependent functions are encapsulated in kernel routines and do not need to be known to the application tasks. In comparison with the previous approach this strategy has a little higher system overhead, because of the required communication between the kernel routines and the application tasks for exchanging I/O data. This approach has for instance been adopted in the MARS system [KOP89].

3. The third approach enables all interrupts, but the drivers must be reduced to the least as possible. The only purpose of the drivers is to activate a certain task that will handle the management of the device, i.e. the driver do not handle the device directly but only activates a dedicated task. The device management tasks are scheduled and guaranteed as all ordinary tasks. The major advantages with this method in comparison with the previous described approaches are the elimination of the busy wait I/O operations, that a control task can have a higher priority than a device-handling task. A system that has been adopted this approach is the SPRING system [MOL90], which is a distributed real-time operating system for large complex applications with hard timing constraints.

Benchmarking real-time operating systems

Benchmark programs measure the relative speed of computers, algorithms or different language implementations. Usually they are used when performance comparisons are to be carried out. For instance, benchmarking can be used to compare a software implementation with a hardware implementation, since the approach: using special-purpose hardware for increasing the performance and predictability of a system is widely used today. Another way of using benchmarks is when a novel system with performance requirements has been developed. In such case, the benchmark program works as a software verification tool. A problem with benchmarking of real-time operating systems is the lack of standards for measuring performance [ACH91, WEI99]. Especially when evaluating real-time kernels from different vendors, the tests often have to be rewritten, because the kernels have interface differences.

Monitoring

In order to benchmark a system, the system has to be monitored, i.e. gathering run-time information for performance measurements. Any attempt to gain more information about a system may intrusive the system temporal and/or functional. This problem has been referred to as the *probe-effect*. Probes are necessary for monitoring; without probes it is impossible to measure a systems performance. When the programmer has monitored a system by using probes, the probes cannot

be removed in all cases. Because, an elimination of the probes may also introduce the probe-effect. That means that the probe-effect may occur either when probes are inserted into the system or when probes are removed from the system.

A system may have performance requirements. In such case, the designers must first estimate the performance of the architecture during the design phase, and when the system has been tested, measure and evaluate the real performances and compare them to the performance requirements. In [CAL98], the authors have classified different kind of performances into different categories, see figure 2. Dynamic performances are classified into external performances that concern the behaviour of the system observed from its interface (response time, throughput etc.) and internal performances (process execution time, bus utilization, etc.). The dynamic performances are the most difficult to deal with.

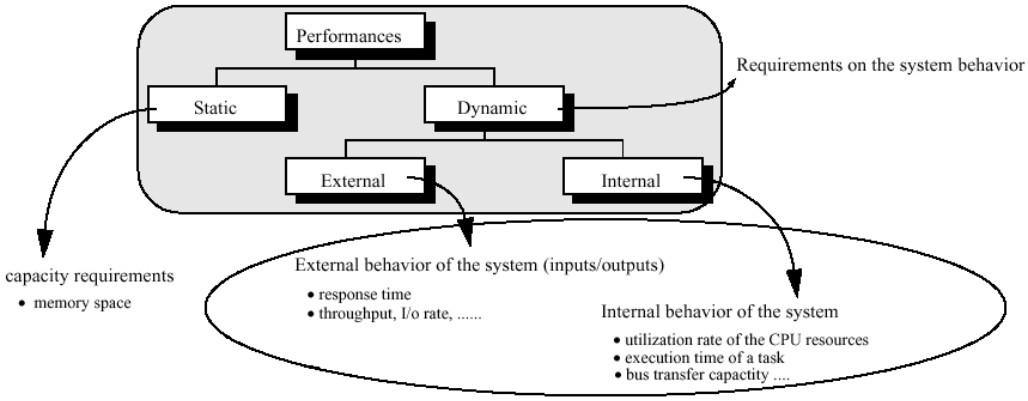


Figure 2, Different categories of performances.

The problem of collecting relevant information from distributed real-time systems can be classified in three categories: hardware monitoring, software monitoring and hybrid monitoring.

The hardware monitoring is based on connecting probes to the hardware system in order to observe its behaviour without disturbing it. The probes can for instance be logic analyzers or emulators. The main drawbacks according to [CAL98] are:

- Often the development of a specific hardware and software to monitor it are required.
- The abstraction-level of the collected information is very low, which makes it difficult to interpret.
- With future VLSI components including all the parts (CPU, memory, FPGA) in the same chip (system-on-chip), this technique does not work.

The software approach consists of adding a set of extra instructions to the software, in order to collect all useful information during runtime. This technique

can be fairly target independent, with focus on the hardware architecture. It also provides a high level of abstraction of the collected information. The major drawback with this approach is the disturbance on the system, affected by the information gathering. Both the temporal and system behaviour may be disturbed. For real-time applications, the disturbance may end in both wrong results and not fulfilled timing constraints.

The hybrid approach is a combination of software monitoring and hardware monitoring. It is based on a few instructions in the software code that selects adequate information. The information is collected with a specific hardware, and then transmitted to a host system [CAL98].

A hardware approach that overcomes the main drawbacks with hardware monitoring listed in [CAL98], is MAMon (Multipurpose/Multiprocessor Application Monitor) [SHO01], which is an ongoing research project at Mälardalen University. MAMon can be integrated in a System-on-Chip (SoC) design and it increases the level of abstraction. The current version of the monitoring system can both monitor the logic-level and the system-level in both single and multiprocessor systems. The monitoring system is able to monitor in a completely passive manner without probe effect, assuming an ideal target system. Ideal target systems are systems were a small hardware component could be integrated in the hardware as a SoC or a hardware kernel as the RTU [ADO96]. Today many SoC applications are hard to verify and optimize, they are often monitored from the register transfer level (RTL) or even the gate-level. Bugs are however easier to find in the system-level since fewer events/s occurs on that level [SHO01] as illustrated in figure 3, this motivates a top-down debugging strategy. MAMon allows us monitoring the system-level, this might be important in future SoC applications since complexity is increasing with a tremendous speed.

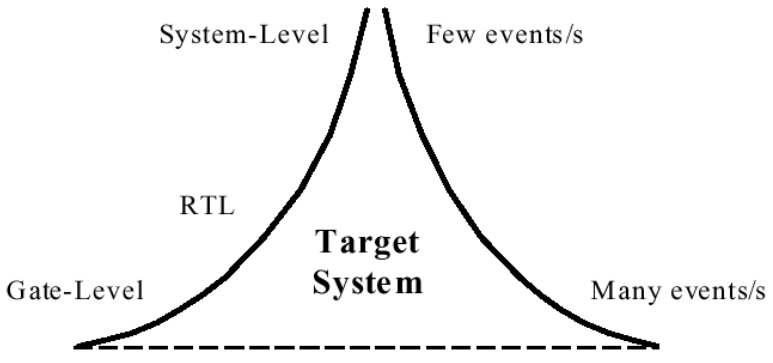


Figure 3, Events/s in different abstraction levels.

Testing vs. benchmarking

Software testing is often associated with the terms *validation* and *verification*. Validation is the process of checking that a programs specification fulfils the customers' requirements. Software testing is a kind of verification of a program or system, with consideration to the specification. In general, two methods can be applied in the verification process, namely testing and formal methods. Today, testing is the state-of-practice, although formal methods seem to be a powerful tool, many problems must be solved before formal methods can replace testing in complex real-time systems. In [THA00], the author explains the main limitations with today's formal methods. For instance it is at least as error-prone to express systems functionality in a formal mathematical language as writing the code, and it is also hard to verify that it is the "real system" that is modeled. The interested reader can consult [BAR92], for more information on how to use formal methods in the software verification process.

Software testing is used in the verification-process of a program or system, with consideration to the specification. Testing may be divided into two stages. First, a kind of testing strategy is developed. The testing strategy describes how testing is to be performed, i.e. how to select input data, what information have to be collected and how to collect and analyze this information. Second, the testing strategy is applied to a program, which results in a test of that program.

Benchmarking is used with completely different goals in mind, than verification. The purpose with a benchmark is to create basis for comparisons of systems, or products with similar properties [KAM96]. In other words a suitable benchmark must include and cover the most typical properties of the target applications.

Benchmark examples

In this section some suitable *benchmarks for real-time operating systems* are presented and described. Classes of benchmark techniques that are skipped in our selection include *microprocessor-oriented benchmarks*. When designing a real-time or embedded system, microprocessor-oriented benchmarks are used to assist the designer in the comparison between different hardware platforms. Some examples of microprocessor-oriented benchmarks can be found in the Whetstone [CUR76] and the Dhrystone [WEI84] benchmarks. These benchmarks are widely used and have own metrics associated with them, whetstones and dhrystones. Both benchmarks are based on synthetic workloads, the Whetstone benchmark represents a typical scientific workload while the Dhrystone benchmarks represents a typical system program instruction stream. Instead of assisting the designer to choose the most suitable hardware platform, the following methods are used for performance measurements.

Rhealstone

The Rhealstone benchmark [KAR89] is a real-time microbenchmark, based on the idea that each real-time application is unique, in contrast to Dhrystone and Whetstone. First of all, six categories of components that are crucial to the performance of real-time systems are measured and summarized:

1. *Task switching time* – The average time the system takes to switch between two independent and active tasks of equal priority
2. *Preemption time* – The average time it takes a higher priority task to take the control of the system from a low priority task. In general this event occur when a high priority task becomes ready.
3. *Interrupt latency time* – The time between the CPU's receipt of an interrupt request and the execution of the first instruction in the interrupt service routine (ISR).
4. *Semaphore shuffling time* – The delay of a task releasing a semaphore, and the activation of another task waiting for that semaphore. No other tasks are scheduled in between.
5. *Deadlock breaking time* – The average time it takes to resolve a deadlock, which occurs when a high priority task preempts a low priority task that hold a resource needed by the high priority task.
6. *Datagram throughput time* – The number of kilobytes per second one task can send to another, by using kernel primitives, i.e. the average interprocess communication speed.

Measurement of these Rhealstone components yields a set of time values. In order to perform overall comparisons between different real-time systems, the values have to be combined into a single Rhealstone performance number. The following steps are necessary to achieve a single Rhealstone number:

1. Express all measured time values in the same unit (seconds).
2. Compute the arithmetic mean of the different components.
3. Arithmetically invert the mean (from step 2), to obtain the number of Rhealstones per second.

[KAR90]

The calculated Rhealstone number treats all the Rhealstone components as equally important parameters of real-time performance. In other words, this number is good when evaluating a real-time system performance without a particular application in mind. When a real-time system is dedicated to a certain type of application, it is possible to calculate an *application-specific Rhealstone* number. This number is based on weighted components, i.e. each component is given unequal weights. For instance, an application may be interruptdriven and don not use semaphores at all. The steps for calculating application-specific Rhealstones are:

1. Estimate the relative frequency of each Rhealstone component's presence, and assign coefficients proportional to the frequencies.

2. Compute a weighted average of the Rhealstone components, and the invert it to get a result expressed in application-specific Rhealstones per second.

[KAR90]

For further details about computing application-specific Rhealstone numbers look in [KAR90], because in contains some improvements in comparison with the original proposal [KAR89].

Distributed Hartstone

Distributed Hartstone [KAM91] is an extension of the Hartstone benchmark [WEI90]. To be short the Hartstone benchmark is a benchmark aimed for real-time systems, which consists of 5 test series each with a synthetic workload aimed to test different task types.

The distributed Hartstone benchmark suite is developed for use in distributed real-time systems, and is well adopted to evaluate the performance of an RTOS. The distributed Hartstone benchmark as the Hartstone benchmark is based on a synthetic workload that represents the typical instruction stream of a scientific application. The benchmark consists of different task sets, which evaluates the performance of one particular feature each. In each task set, all factors are kept constant except one, which is the varied factor. The varied factor in a task set is typically an execution time or the number of task or messages, however the common factor is that the factors all can be increased to infinitely. In this way a distributed Hartstone benchmark always reaches the breaking point of the system, which is the point where the first deadline miss is experienced. In all the different tests with different task sets, it is the breaking point that yields the distributed Hartstone performance measure. The different task sets or measurements that forms the performance measure are:

- DSHcl Series: Communication latency
  The end-to-end communication delay is measured, it is an important metric and we can all agree with that it is preferable to be able to distinguish between different systems in this area. A good system in this test should be able to provide a bounded worst-case time, and shortest possible delay to messages from high-prioritized tasks.
- DSHpq Series: Priority queuing
  The scheduling of messages is tested; in ordinary systems it is often FIFO queues that handle the messages. But in a real-time system this is often not a suitable solution, since a high frequency or a fast activity cannot be delayed by an arbitrary number of other activities. This test is aimed to test if the message passing algorithms can avoid priority inversions and how successful the algorithm is on handling message priorities.
- DSNpp Series: Preemptability of the protocol engine
  If a low priority message is being handled by the protocol engine, this test is concerned about how long time it will take to switch over to an recently arrived high priority message.

- DScb Series: Communication bandwidth
  The communication bandwidth is a metric on how much traffic the communication medium can handle, included in this metric is also all the operating system processing required for communication.
- DSHmc Series: Media Contention
  This measure is concerned about the lower levels of the communication protocol. A suitable medium access protocol should provide a higher degree of service to high-prioritized messages. Even in the medium access layer priority inversion should be provided.

SSU

The Superconducting Super Collider Laboratory (SSCL) has made performance-measurements on four different real-time kernels [ACH91]. The products were all from different vendors, but in order to compare and evaluate the different offerings, they were tested on the same hardware platform. The measurements falls into two categories: real-time and non real-time. The non real-time category contains the throughput measurements, including process creation/termination times, interprocess communication facilities involving messages, semaphores, shared memory and memory allocation/deallocation. Measurements classified as real-time, are for instance context switch times and interrupt latencies.
Each test was executed several times in order to compute the average time to complete a test. Then the entire measurement was repeated a number of times, to measure the minimum and maximum average values.

The different measurements that were performed are:

1. *Create/Delete task* – This is the time it takes to create and delete a task. As soon a task I created, it deletes itself. The measurement includes two task context switches and the time it takes to create, start and delete a task.
2. *Ping suspend/resume task* – A task with low priority resumes a suspended high priority task. The high priority task immediately suspends itself. This measurement includes two task context switches and the time it takes to suspend and resume a task.
3. *Suspend/Resume task* – This test is identical to the previous test, except that this test does not include any task context switches, since in this test a high priority task suspends and resumes a suspended low priority task.
4. *Ping semaphore* – Two tasks with equal priorities communicate with each other through semaphores, i.e. they are competing for a semaphore.
5. *Getting/Releasing semaphore* – This test measure the time it takes to get and immediately release a semaphore within the same task context.
6. *Queue Fill, Drain, Fill Urgent* – The time it takes to fill a queue and drain the queue is measured (two different tests). Then the tests are repeated with priority messages. The messages are sent to the head of the queue.

7. *Queue Fill/Drain* –A task sends a message to a queue, which the sending task immediately receives on the same queue. No task context switch will occur nor there is any pending queue operations. In other words, the time measured includes context switches, queue pends and sending and receiving a message.
8. *Allocating/Deallocating memory* – This test measure the time it takes to allocate a number of buffers from a memory partition and the time it takes to return those buffers to the partition.

The conclusion that the authors of [ACH91] presented was that standards adherence makes code more portable. In order to perform measurements on the different kernels, they had to rewrite all the tests for all the kernels, because the interfaces were different. In other words, the tests were custom-written for each target platform.

Case Studies

In this section some case studies of implemented multiprocessor real-time operating system kernels are presented. As introduction to this chapter our selection criteria's, and the used evaluation model are presented.

Selection criteria

The overall requirements for the selected operating systems are support for real-time applications and multiprocessor management. The multiprocessor support should preferably be for NUMA or NORMA architectures, rather than only UMA systems. While no specific requirements of the real-time support are specified, it could be any mix between soft and hard real-time support, and anything the inventors of the operating system want to refer to as real-time.

In [BUT97], the authors divide current operating systems having real-time characteristics into three main categories, division (1):

1. Priority-based kernels for embedded applications
2. Real-time extended time-sharing operating systems
3. Research operating systems

Another division that exists is (2):

1. Commercial
2. Open source
3. Research

The selection of operating systems in this paper strives to cover both the listings above. An overview and classification of the studied operating systems are shown in table 1.

RTOS	Classification according to (1)	Classification according to (2)
CHAOS^{arc}	3	3
CHIMERA	3	3
MARS	3	3
MontaVista Linux	2	1,2
OSE	1	1
RT-MACH	2, 3	3
RTEMS	1	1,2
SPRING	3	3
SARA	1,3	3

Table 1, RTOS classification table.

It is easy to conclude that the research operating systems are over-represented, but it is also these results and ideas that are most well described and include the most advanced solutions. Most of the information available for many commercial operating systems tend to be of more advertising nature, than honest descriptions

on chosen algorithms and results. We have to make clear that the commercial RTOS we picked out, were selected because of the opposite. It was possible to find non-advertising documentation concerning OSE. On the other hand many of the published papers related to research operating systems, tend to describe only the parts that their research are focused on. In other words it is sometimes hard to get a general picture of a research system, but often easy to find results according to small special areas. Our responses to the documentation related to the open source operating systems is mixed. It was a lot of pages written, and it was possible to find what we searched for. But the really deep descriptions of methods and algorithms were missing; instead they referred to the actual code. The code describes everything, but it can be bothersome to understand the details.

Some commercial RTOS that are not investigated in this survey are VxWorks [WIN02], QNX [QNX02] and VRTX [MEN02]; all of them seem to be interesting. On the open source scene we have for instance not investigated eCos [RED02], a Linux distribution for real-time usage. The research systems we have left out include RK [LEE88], HARTIK [GBU93] and Asterix [THA01]. The reasons why we have not included the mentioned RTOS:es in the survey are for some RTOS:es limited multiprocessor support and concerning some other it has been hard to find relevant documentation. But the main reason is the need to limit the survey, due to the limited time a masters' thesis project contains.

Evaluation model

The chosen evaluation model, strives for the possibility that the readers themselves, shall be able to compare the different operating systems. An obvious way to achieve this goal is to evaluate many issues and then let the reader combine the issues freely. The studied issues will be: scheduling, memory management and interprocess communication. Each issue will be evaluated and graded from 1 to 5 according to the three keywords listed below, where 5 is the best. With this method, the result will be 3 grades for each operating system and studied issue.

- Determinism – guarantee possibilities.
- Inventiveness – extraordinary solutions.
- Usefulness – typically flexibility or portability.

In this way the reader can create a lot of ranking orders, for instance it is possible to find the operating system with highest overall rank, or the operating system with the most deterministic scheduling algorithm, or the most useful interprocess communication methods. A summary of the evaluation is presented last in this chapter and, the motivations are presented in an evaluation section bounded to every investigated RTOS.

Examples of what we will reward can be found in the tables (2, 3 and 4) below for each issue and comparison keyword. Note that the listed examples in the tables

are only positive qualities; the absence or negation of the mentioned qualities will lead to a lower grade. A motivation to each assigned grade will be given.

Category	Scheduling
(1)	WCET estimations possibilities. Off line analysis possibilities.
(2)	Task placement algorithms. Multilevel scheduling.
(3)	Reasonable task attributes, (easy translation of real-world problems). High potential CPU utilisation. Flexibility in the number of supported scheduling algorithms.

Table 2, Scheduling evaluation.

Category	Memory management
(1)	Bounded worst-case access time.
(2)	Virtual memory and dynamic allocation. Cache support.
(3)	Dynamic and virtual memory.

Table 3, Memory management evaluation.

Category	Interprocess Communication
(1)	Bounded worst-case transmission time. Resource reclaiming protocols. Interrupt handling.
(2)	New or conceptually different methods.
(3)	Number of supported methods (message passing, shared memory, remote procedure calls etc). Potentially fast communication. Interrupt handling.

Table 4, Interprocess Communication evaluation.

CHAOS^{arc}

CHAOS^{arc} [GHE93] is an object oriented real-time kernel of the CHAOS [SCH87] family. The CHAOS family of operating systems is structured in a layer model with three layers. The first layer or the layer closest to the hardware is CHAOS^{base}, which is a machine dependent component that implements some basic operating system abstractions as threads, memory handling and synchronisation primitives. Above CHAOS^{base} follows CHAOS^{min}, in this layer the object-oriented approach is implemented with abstractions of classes, objects and invocations. In the top layer the policies of the different operating systems in the CHAOS family are implemented. The most interesting set of policies or operating system constructed with CHAOS^{base} and CHAOS^{min} are CHAOS^{arc} according to the authors of [SCH90].

The hardware requirements of the operating system are dynamic since the hardware dependent module CHAOS^{base} has been implemented or at least could be implemented on several different architectures. The implementation presented in this paper is running on a 32 node GP1000 BBN Butterfly [CRO85]. The Butterfly is a shared-memory parallel processor, with each processor node mainly consisting of a MC68020 processor, 4MB RAM and a co-processor called *Processor Node Controller* (PNC) which handles shared memory requests. The interconnection between processor nodes is a 32 Mbit/s per path multistage switch. The operating systems itself is running on every processor in the system.

The main contribution with CHAOS^{arc} is the use of an object and class based environment for the programmer, when defining the application. Just as an object oriented language, CHAOS^{arc} let the programmer define classes that represents a desired behaviour and the use of objects that is an instance of a class.

Four built in classes defines all primitive objects *Abstract Data Type* (ADT), *Threaded Abstract Data Type* (TADT), *monitor* and *task*. An ADT defines a passive object without execution threads or synchronization for concurrent calls. When calling an object of the ADT class, the method is executed in the address space of the caller and without synchronization of concurrent calls. When an object from the TADT class is called, a new execution thread without automatic synchronization of concurrent calls is created. A monitor is an object without execution threads that only allows one single call to be active at a time. Finally a task is an object with a single execution thread, all calls to a task are serialized and executed in the context of the task. These four primitive classes are implemented in the CHAOS^{min} layer. But the CHAOS^{arc} layer offers definition of more complex classes that are built upon these four primitive classes. A complex object can be represented as in figure 4, by a scheduler a state and a server.

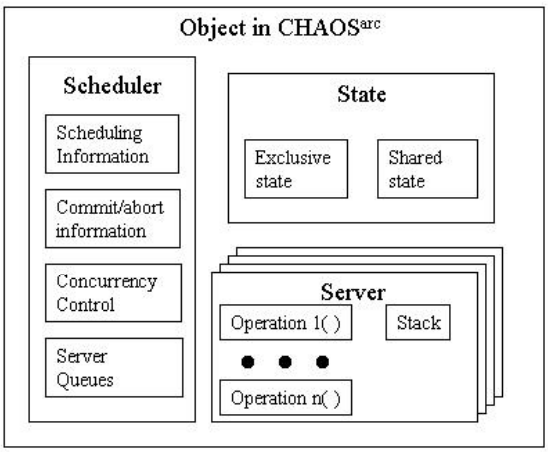


Figure 4, Object representation in CHAOS^{arc}.

The objects state is partitioned into a number of components, each of which can either be of shared or exclusive type. The shared component is directly accessible to all invocations of the same object, while an exclusive component is unique for every invocation of an object. But if the invocation commits the shared component is atomically copied back to the objects state, on the contrary when an invocation is aborted the result of exclusive components is discarded.

The objects scheduler receives and schedules all invocations to the object. The scheduling decisions by the objects scheduler is based on parameters passed with the invocation call (see section communication). The objects scheduling component is described in section scheduling.

The objects servers are simply threads that are able to execute any of the operations that the objects interface offers. Threads can either be created dynamically during run-time or static at object initialisation.

Scheduling

Multiprocessor scheduling is performed by the CHAOS^{arc} policies. The policies can be configured different for every invocation of an operation and is carried out by the objects scheduler. The multiprocessor scheduling is performed in cooperation with the thread schedulers residing on every processor in the system. On this object level the processor placement of the task is determined. If the operation requires any locks the objects scheduler takes care of that problem, calculating times until locks can be achieved etc. Unfortunately the research around CHAOS^{arc} was not focused on this type of multilevel scheduling (i.e., invocation scheduling followed by thread scheduling) so the exact algorithms used for the invocation scheduling were not described in detail.

The thread scheduling is distributed across the machine with one scheduler per processor. The scheduling algorithms [SCH89] basic data structure is a doubly linked list, called a slot list. The slot list records which threads have been scheduled during each of the time periods. The threads are preemptive scheduled with EDF (Earliest Deadline First)[LAY73], and as close to their start times as possible.

When a thread has to be scheduled, a feasibility test is performed. The test is a simple search for execution time. It starts with a slot compatible with the threads' start times, and ends at a slot compatible with the threads' deadline or when the total length of available time slots is equal to the threads execution time. If enough available time is found the thread is scheduled and the slot list is updated, else the thread is reported un-schedulable. The slot list can be accessed through a balanced binary tree; therefore a slot with a particular start time can be located in O(log n) time.

Once a thread has passed the test and is going to be scheduled it has nothing to do with the slot list anymore. Instead all scheduled threads are put in another list and scheduled with EDF. This list includes all threads, even threads that have an earliest start time later than the current time. This is the reason why the list cannot be a queue, since the first thread might not have a feasible start time. So the final complexity of the feasibility test and the EDF scheduler is in the worst case reported to be O(n log n).

Memory management

Memory management is described in [SCH87], because of the heavy use of dynamic memory during object creation and invocation; it is obvious that CHAOS^{arc} needs some support for predictive dynamic memory. The solution is based on the observation that the possible memory requests are a finite set. The supported types of requests are:

- Status block's of known sizes, that the operating system uses to maintain an objects status during an invocation
- Parameter block's, which are used for parameter passing during an invocation
- Memory used for the representation of objects and processes

To provide predictable access times for allocation of these three types of memory, the kernel pre allocates memory blocks and store the blocks in four memory pools. Two pools for kernel usage and two pools for application usage, this prevents that memory usage from the application affect the kernel. The main contribution with this is that the kernel is able to provide predictable response times, since sizes and maximum amount of memory blocks needed by the kernel is predictable.

For application code it is worse, but it is possible to pre determine the memory usage. When memory blocks are deallocated, they are returned to the pool from

which they originated. A possibility is therefore to begin the application code with allocating a certain number of memory blocks with a certain size. Then stick to this size when allocating memory, and never allocate more memory blocks at the same instance in time than the number of allocated memory blocks pre allocated in the beginning.

Interprocess communication

The interaction possibilities between objects are: explicit synchronization with the use of synchronization points, invocation of each other's operations and atomic computations spanning multiple objects.

Synchronization points are the method for expressing events. A synchronization point is an ADT with internal data structures consisting of:

- A control block, which represents the state (enabled, disabled etc).
- The connected queue reflects permanent dependencies between synchronization points.
- The delayed queue represents dependencies among multiple invocations among a single synchronization point.
- An enabled synchronization point expresses an internal or external event and can be used for object invocations or any other synchronization event.

Invoking a method or operation of an object often requires both data transfers (for parameters) and control transfers (for the operation's code). To be able to handle different interaction patterns typical to real-time systems, CHAOS^{arc} supports a number of invocation modes and to the modes associated attributes. The different invocation modes are async, sync, periodic, event, stream and fast. To give a short and general view of the invocation semantics and possibilities only a few modes and attributes associated to each mode are mentioned here in table 5, but the interested reader are referred to [GHE93] for a detailed description. For example the async mode has 15 different invocation attributes with default values.

Mode	**Semantics**	**Example of Attributes**
Async	Sporadic (one time) invocation that can execute in parallel with invoker.	Hard or soft deadline and atomicity.
Periodic	Causes the operation to be executed periodically.	Deadline attributes and period time.
Event	Invokes the operation every time a certain synchronization point is reached.	Trigger synchronization point.

Table 5, example of modes their semantics and attributes.

The implementation of all functionality that extends CHAOS^{min} to CHAOS^{arc} is actually implemented with the primitive objects offered by CHAOS^{min}. Examples of the added functionality are invocations and atomic computations. The invocations are thereby extendable and a user of the operating system can implement arbitrary invocation semantics.

Atomic computations

An atomic computation is in CHAOS^{arc} an abstraction for a hierarchy of atomic invocations of object operations. Namely, an atomic computation represents a group of object invocations with common timing, consistency and recovery requirements. The whole computation (sequence of operations) is guaranteed to end in either commit or abort. When committed, external observers can view the result of the computation. When aborted the result is discarded and the system state is restored. The success or failure of an atomic computation is determined by timing constraints (e.g., deadline for the computation), or by the consistency requirements that follows when the computation shall be atomic.

Conclusions

CHAOS^{arc} seems to differ from other RTOS in many ways at a first look. The biggest differences lies in the upper layers of the layered system, the object oriented approach and the lack of ordinary message passing or shared memory. The primitives offered by the CHAOS^{base} layer seem to be quite similar to other RTOS primitives, threads, scheduling etc. It is actually this layer that performs all operations in the end. All interactions between objects and the objects responses to the environment, maps into ordinary thread scheduling.

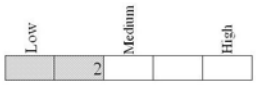
A question that arises is how to perform scheduling analyses and map the environments constraints into the objects, we are sure that the inventors have solutions to this. But the method differs from other known and well-understood methods and terms, in the RTOS domain.

Another question is their interpretation of atomic computations, a sequence of invocations that either are committed or aborted. It is easy to understand the need of such transactions in database systems and similar facilities. But in a real-time system that interacts with the environment all the time, is it suitable or even possible to abort operations? When we have told the controlled physical object to turn left, increase the speed or decrease the temperature, it is a little bit late to abort the operation.

Evaluation

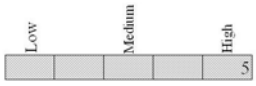
Scheduling

- *Determinism*



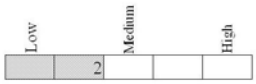
The on-line mechanism, with a guarantee algorithm and EDF scheduling is analysable. But what happens if a task cannot be guaranteed, probably the invocation or instansation fails. The off-line analysis of the object-oriented approach is not presented either, is it possible to analyse which tasks that is going to be created during runtime from an object point of view?

- *Inventiveness*



Multilevel scheduling and a form of task migration, when an object of the ADT class is invoked is a pretty fancy solution.

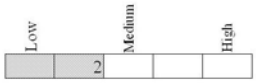
- *Usefulness*



It is a tempting idea to use an object hierarchy instead of ordinary tasks, but it seems to be harder to translate real-world problems into this model. Although we have a lot of configurable invocation attributes and semantics, but is that enough? Real-time engineers are used to the task model and this revolutionary solution will probably just make it harder to create an application.

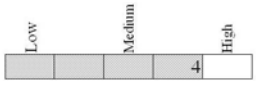
Memory management

- *Determinism*



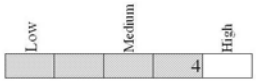
The authors claim that it is possible to pre determine the memory usage for the application and, in this way, know sizes and quantity of memory blocks that the application uses. We have to remember that CHAOS^{arc} implements a strict object oriented policy, that includes dynamic memory allocations that are in some sense hidden for the programmer. A worst-case scenario on the other hand is easy to create, and likeley to occour. If the application allocates various sizes of memory blocks, remember that using different sizes of parameter or return value fields should be enough.

- *Inventiveness*



The system pools together with dynamic memory usage, provides a base for the implementation of a object oriented approach. It is also an solution that add additional primitives compared to a basic solution.

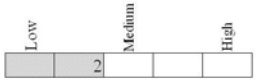
- *Usefulness*



The memory management system in CHAOS[arc] would be useful in other RTOSes where the dynamic memory usage is more controlled and easy to predict. Used in such RTOSes the system could be a way to provide dynamic memory. And of course in CHAOS[arc], the memory management system is a requirement.

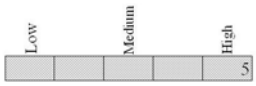
Interprocess communication

- *Determinism*



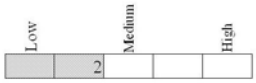
It cannot be guaranteed that all invocations of type *fast* will succeed. The other *non-fast* invocations are queued can utilize dynamic memory, and remote procedure calls, in other words hard to predict.

- *Inventiveness*



A very extraordinary solution, it is easy to associate IPC with shared memory or message passing. Here no such abstractions are avalible, although the invocations in the end must map onto messages or shared memory in some sense.

- *Usefulness*



We can easily conclude that the communication mechanism with object invocations are closely realated to remote procedure calls, which are useful. Therefore the object invocation mechanism must be useful to, but we miss alternatives in CHAOS[arc].

Chimera

Chimera [KHO92] is a multiprocessor real-time operating system designed especially to support the development of software for robotic and automation systems. The advantage with Chimera over other real-time operating systems is that it provides features that are necessary for quickly developing reconfigurable and reusable code [CHI02]. The environment consists of one or more real-time processing units (currently Ironics 68020 CPU boards) connected to a VME bus on a SUN 3 workstation. The workstation provides a high-level computing environment with editors, compilers, debuggers and so on. Real-time applications are developed at the SUN workstation and then executed on the Ironics boards. The Sun and Ironics CPU's has the same 68020-based architecture, so no extra cross-compilers/linkers are needed. Chimera code is compiled and linked with an ordinary SUN C compiler and linker. This guarantees that the code will have exactly the same behaviour when executed on the SUN CPU or the Ironic CPU's. The Chimera software consists of two parts; one part that runs on the Ironics and another part that runs on the SUN workstation. User tasks execute concurrently on one or more CPU's and communicate with each other through local shared memory and local semaphores. The Ironic boards have distributed dual ported memory, i.e. can be accessed by both the Ironics and SUN CPU's. This makes the download from the SUN to Ironics very simple.

According to the developers of Chimera, a predictable system must handle all errors in a certain fashion. The best solution if it is possible is to call an error handler that corrects the error. An intermediate solution is to operate the system with degraded performance, which is often necessary with autonomous systems. Last of all the system must be shutdown.
Chimera has support for both deadline-failure handling (timing errors) and handling of non-timing errors. The *global error handling mechanism* is a powerful mechanism in Chimera. It allows the user to develop applications without explicitly check the return value if it is an error. Because whenever an error is detected an *error signal* invokes the error handling mechanism. By default, a detailed error message is printed and the task is aborted. This mechanism allows the error messages to be very specific.

An optional failure handler is called when a task fails to complete its deadline, insufficient CPU time is available or when the maximum estimated CPU time for a task has surpassed. The failure handler can be programmed to run at either the same priority as the task that misses its deadline or at a different priority. The user can define own error handlers and alter the default action of the system. The authors of [KHO92] claim that the deadline failure handling mechanism is essential in predictable systems, because the estimating of a tasks execution time is often really difficult. Especially when the hardware has mechanisms for increasing the average performance, for instance caches and pipelines. Such hardware is often used in real-time systems, which implies that the execution

times cannot be predicted accurately enough. Underestimating of a worst-case execution time can cause a disaster.

Scheduling

Chimera provides typical RTOS kernel task management features, such as creating, suspending, restarting, preempting and scheduling. The kernel schedules the tasks using virtual timers, all based on a hardware timer. Chimera supports both static and dynamic scheduling of real-time tasks. The default scheduler supports the rate monotonic scheduling algorithm (static scheduling), the earliest-deadline-first scheduling algorithm (dynamic scheduling) and the maximum-urgency-first scheduling algorithm (static and dynamic scheduling). This algorithm provides predictable dynamic scheduling possibilities. The scheduler is designed as a module that can easily be replaced by any user-defined scheduler. That allows Chimera to be used in many different applications, without being restricted by the default scheduler. The authors to [KHO92] claims that most real-time scheduling theory concentrates in ensuring tasks always meet their deadlines. In addition, Chimera has a deadline failure handling mechanism, which calls an exception handler when a task fails to meet its deadline.

Memory management

The Ironic boards have distributed dual ported memory, i.e. can be accessed by both the Ironics and SUN CPU's. The VME bus has many different address spaces, and each CPU addresses the spaces in different ways. Memory can be dynamically allocated by using express mail, which is explained in the next section. Memory can either be allocated locally or remotely on other processing units or memory boards (NUMA). The Chimera kernel was designed to provide much of the functionality of a true multi-tasking operating system, while preserving the response time of a dedicated real-time processor. This was made possible by eliminating for instance inter-process security, a large process space and virtual memory.

Interprocess communication

A task can communicate or synchronize with any other task through local shared memory, high-performance local semaphores or user signals. Local semaphores are either used to synchronize tasks or provide mutual exclusion during critical sections. *User signals* are an alternate way of synchronizing tasks, allowing the receiving task to be interrupted when the signal arrives, instead of polling as done with local semaphores.

Many different types of interprocess communication and synchronization mechanisms are built in as layers, in purpose to simplifying the development of complex applications.

- *Express mail*: The express mail mechanism is a high-speed communication protocol that was developed especially for backplane communication. It is the lowest communication layer. Express mails are handled by server tasks that run on each CPU. The server task monitors

the express mails, translates symbolic names into pointers and translates addresses within various address spaces on the VME bus.

- *Global Shared Memory*: The VME bus has many different address spaces, and each CPU addresses the spaces in different ways. Memory can be dynamically allocated by using express mail. Memory can either be allocated locally or remotely on other processing units or memory boards.
- *Spin-locks*: The spin-locks use atomic *test & set* instructions, in purpose to provide mutual exclusion of a shared data. The use the polling technique to obtain the lock, which could waste a lot of CPU time. But generally they require the least amount of overhead, in comparison with other synchronized IPC mechanism [HOF89].
- *Remote Semaphores*: Chimera provides both local and remote semaphores, which allow several tasks on different CPU's to use semaphores. The remote semaphores use *test & set* to get hold of a lock. A task that tries to obtain a lock that is occupied will be blocked. When the lock is released, the blocked task is invoked.
- *Priority Message Passing*: The priority message passing system uses the express mail to initialise message queues. The user defines lengths of queues. Typed messages of variable length can be sent between tasks on different or the same processor. The message queues can be sorted using *first-in-first-out*, *last-in-first-out* or *highest-priority-first* algorithms.
- *Global State Variable Table Mechanism*: This is a mechanism that allows multiple CPU's to control tasks that are co-operating, by means of state variable tables. One global table and one local copy of the table for each task that requires access are created. I.e. they share state information and they can update the states in a correct way. Tasks update the local copies periodically and they do always make use of the local copy.
- *Multiprocessor Servo Control*: One task in the system can take control of some or all processing units. The task can then control the execution of other tasks and even spawn new tasks to any processing unit.
- *Extended File System*: Instead of having a separate disk file system, the real-time system uses the file systems on the host workstation. A task can therefore perform file operations as any process on the host workstation. All remote operations are transparent to the user.
- *Host Procedure Calls*: Tasks running on any processing unit can perform procedure calls to the host workstation, i.e. execute routines on the host workstation.
- *Host Workstation Integration*: The host workstation is transparent to the system, which means that the host workstation is totally integrated into the real-time environment. The host workstation has all the same features as the processing units, for example it can use semaphores and global shared memory.
- *Special Purpose Processors*: Special purpose processors can be added to the system to increase the performance for specialized computations. The

hardware interface is independent, which simplifies the integration of special purpose processors. Since special purpose processors do not run a kernel, they are treated as slaves in the system.

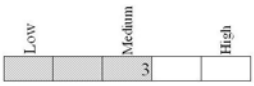
Conclusions

The most distinguish feature with Chimera is the SUN workstation that is integrated into the environment. The workstation's purpose is to provide a high-level computing environment with editors, compilers, debuggers and so on, in order to simplify the development of applications. When an application has been developed, it is downloaded to and executed on an Ironic board. In [CHI02] the authors claim that a deadline failure handling mechanism is essential in predictable systems, because the estimating of a tasks execution time is often really difficult. But is not the estimating of a tasks execution time easy in a predictable system? The reason for making a real-time predictable is the desirable prediction of a task's behaviour, and therefore predictable execution times, which are easy to estimate offline.

Evaluation

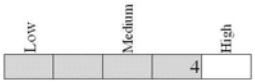
Scheduling

- *Determinism*



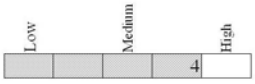
The scheduling algorithms provided are well known and off-line analysable. The WCET seems to be hard to estimate, due to the memory management and IPC. However Chimera has a deadline failure handler mechanism, which is an advantage when the estimation of WCET is hard.

- *Inventiveness*



Chimera has a deadline failure handler mechanism, which calls an exception handler when a task fails to meet its deadline. No direct multilevel scheduling is provided, but the *Multiprocessor Servo Control* mechanism allows a task to take control of some or all processing units. The task can then control the execution of other tasks and even spawn new tasks to any processing unit.

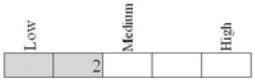
- *Usefulness*



The number of different supported scheduling algorithms make make Chimera flexible and the EDF algorithm makes the potential CPU utilisation really high.

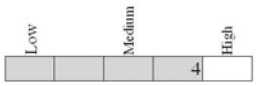
Memory management

- *Determinism*



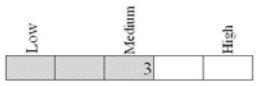
Dynamic memory allocation may jeopardize the determinism and another feature is the possibility to allocate memory remotely on other processing boards.

- *Inventiveness*



The fact that memory can either be allocated locally or remotely on other processing units makes Chimera flexible, but this feature could jeopardize the determinism of the system. Chimera has own version of *malloc*, which allocates memory dynamically.

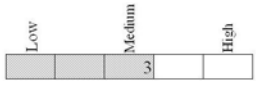
- *Usefulness*



The dynamic memory allocation possibility increases the usefulness of this system.

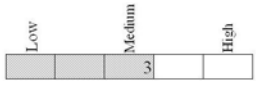
Interprocess communication

- *Determinism*



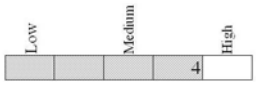
Chimera was designed to minimize communication overheads, which aims for good predictability through smaller variations in worst and best case.

- *Inventiveness*



A task running on any processing unit can perform remote procedure calls to the host workstation. This feature makes the file system on the host workstation available for all tasks running on any processing unit.

- *Usefulness*



Many different communication and synchronisation mechanisms aims for flexability.

MARS

Maintainable real time system (MARS) [KOP85, DAM89, KOP89] is a distributed operating system built upon standalone computers, with some custom built hardware on each node (component). Each node consists of a CPU (MC68000), a custom built LAN controller (LANCE), a custom-built clock synchronization unit (CSU) and some I/O units. Every node also has an identical copy of the kernel. Theoretically MARS has almost no restrictions in the maximum number of nodes and every cluster of MARS nodes can be connected to another MARS cluster through special interface nodes, which forward all messages from one MARS bus to another. However the main contribution with MARS, which makes MARS different from other distributed real-time operating systems is its deterministic behaviour under all conditions. MARS is completely off-line scheduled; even the message bus is pre runtime scheduled. This combined with a time driven dispatch policy gives predictability even under peak load situations. The other factor that aid for the determinism of a MARS system is a high degree of fault-tolerance. Even if a MARS system experiences failure on CPU's etc the system can still be able to deliver full service. Figure 5, shows a schematic picture of the MARS system. The I-Component (Interface-Component) in the figure is an ordinary node and has its interface against the MARS bus as all other nodes, but is actually representing another cluster.

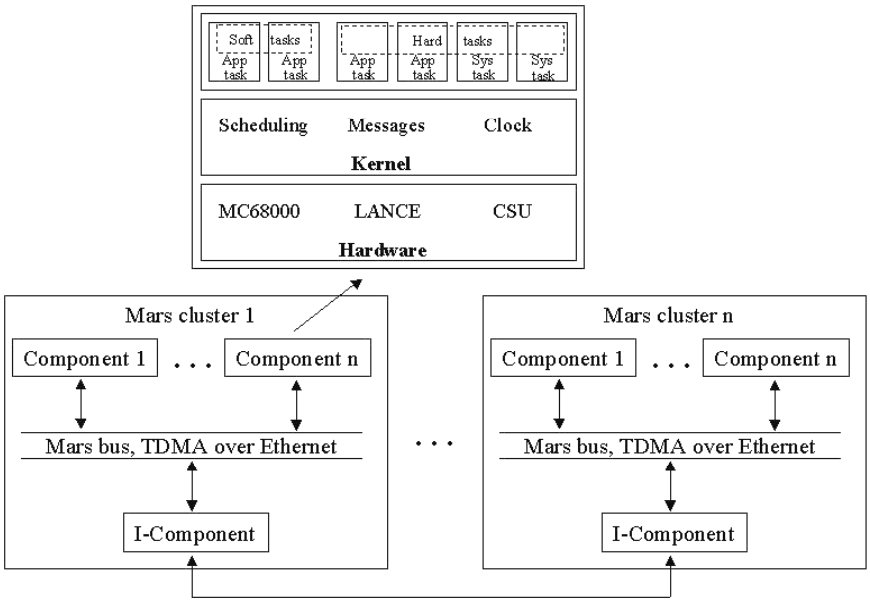


Figure 5, Schematic picture of the MARS system.

Scheduling

MARS supports both hard and soft tasks. Hard tasks in MARS are defined as periodic tasks with a hard deadline, within which it has to be completed. The set of hard tasks consists of both application tasks and system tasks, where system tasks perform specific functions of the kernel. The system tasks include time synchronization and protocol conversations to and from MARS messages and RS-232 strings for example. Soft tasks are tasks that are not subject to strict deadlines; usually soft tasks are aperiodic tasks. They are scheduled in background under low load conditions.

The complete system is off line scheduled, the off-line task and bus scheduler calculate the schedules for tasks and messages before runtime and store the schedules in runtime-scheduling tables residing on each node. The schedules in the runtime-scheduling table consist of a cycle, which satisfies all expressed constraints and should be repeated when executed once. If no such cycle is found a redesign must be done. The on line scheduler is simple and fast; the only job to do during runtime is to make a table lookup and dispatch the task found.
The scheduling algorithm used by the off-line task, takes attributes as WCET (MAXT), and *total transaction time* (Mart) for the transaction that the task is included in. A schedule is produced, based on a heuristic search strategy that calculates task urgency according to estimations of the time necessary to complete the transaction.
Each node can have several different schedules, used in different points of time or in different phases of the application. For instance the starting phase of an application may require a completely different task set than the remaining part of the application. A change of schedule or mode can be simultaneously trigged on all operational nodes in the system by messages.
The off-line scheduling principle requires that the complete system behaviour is analysed and known before runtime, but for some applications that cannot be satisfied. In such cases the Slot Shifting algorithm can be used as scheduling algorithm instead. The Slot Shifting algorithm shows great performance with aperiodic tasks, although the guidelines of the scheduling decisions are calculated off-line. This is achieved by calculating spare capacities in intervals off-line, which simply are unused capacities. Then at run-time the algorithm tries to use the spare capacities for aperiodic tasks. Tools for estimating the WCET of a task given the code support the designers since that estimation is critical. This estimation requires bounded loops and no recursion. All tasks must be present at the node before runtime, no task migration allowed.

Memory management

Memory management in MARS is easy to describe, theoretically you could say that there is no memory management. MARS does not support any dynamic allocation, virtual memory or other potentially un-deterministic memory management features. For example every message that is going to be transmitted has an own buffer that is pre-allocated (at compile time) and own by the operating system, instead of having a task to allocate dynamic memory. We would like to describe memory management in MARS as a compiler and hardware issue.

Interprocess communication

Every message is transmitted n times over n parallel busses or sequentially n times on one bus or any combination between the two extremes. The loss of n-1 messages is therefore tolerable. This redundancy is implemented in the lower levels of the communication protocol; notice that this solution does not increase the communication time when a message is lost, as for example a retransmission protocol does.

The communication medium is the MARS bus, which is an off-line scheduled TDMA bus. This medium access protocol provides a collision free, deterministic and load-independent access to the Ethernet. The semantics for the messages is comparable to global variables and they are called state-messages. State-messages are not consumed when read, so several tasks can read the same message and a new version of a message updates the previous. The messages carry state information about the state of the environment that has been observed at a given point in time. All messages have an identical structure, with a standard header that contains mainly time stamps, a constant but application dependent length of the body and a standard trailer that contains a checksum. The send of a message is non-blocking, just as writing a global variable.

Every message has a validity time associated with it due to the real-time applications temporal constraints. Two time stamps from the CSU are also attached to every message (when sent and received) in this way timing errors can be detected, and the validity time can be measured. When the validity time is expired, the message is discarded by the operating system. Since every node knows which message to expect in every TDMA slot, implicit flow control and error detection between sender and receiver(s) exists.

Clock synchronization

Since MARS is a true distributed system with nodes consisting of standalone computers and is time driven, the need of a global time base is obvious. Each MARS node has its own real-time clock with a resolution of 1 μs. The clock synchronization algorithm is based on message passing, since all messages in MARS have timestamps from the senders CSU and the receivers CSU. Each node can record time differences to the other nodes periodically. Based on that computation a correction term for the own clock can be computed by the Fault-Tolerant Average Algorithm (FTA) [KOP87].

Interrupt handling

All interrupts except the clock interrupt from the CSU are disabled. Allowing each device interrupt the CPU would jeopardize the hard tasks deadlines and cause an unpredictable run-time behaviour. Many other RTOS use a priority scheme for the interrupts, but that solution has also been discarded since high priority devices would be favoured and the on-line scheduler just is a dispatcher and could cause deadline misses with aperiodic activities. The clock interrupt handler polls the devices instead, but not on every clock interrupt.

Conclusions

MARS seems to be a very fault-tolerant operating system. MARS achieves fault tolerance both logically and physically [KOP89]. Logical fault tolerance is achieved by sending each message n times sequentially on the same channel or parallel over more channels, and by a fault-tolerant clock synchronization algorithm, that can tolerate a known number of faulty clocks. Physical fault tolerance is reached with hardware redundancy; each MARS node has one or more identical active replica(s) and more than one active redundant communication bus are supported. The amount of fault tolerance needed by the application decides the number of redundant nodes and busses. Each node also has some self-checking properties and thereby fails silently (delivers correct result or no result at all).

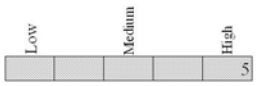
If trying to be pessimistic and critical, it is easy to understand that the fail-silent property must be hard to implement. The redundancy within a FTU assumes that components fail silently, so what happens if a component fail anyway (*Byzantine failure* or undiscovered failure etc)? It would maybe be more fault-tolerant to use some kind of voting mechanism within a FTU, like the TMR (Triple Modular Redundancy) algorithm for instance. The static scheduling requires that the application is well known before run-time. Some questions that arises are:

- What if the application is not even built?
- What if the applications environment changes during run-time?
- What if the initial timing hypothesis seems to be wrong?

Evaluation

Scheduling

- *Determinism*

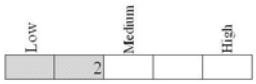


The off-line scheduling principle, guarantees that no deadlines will be missed. Interrupts and other possibilities for instability are disabled or designed away.

- *Inventiveness*

Different schemes for different times, and the use of a off-line task compensates for the otherwise simple on-line mechanisms.

- *Usefulness*



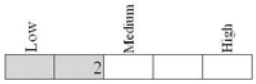
Static scheduling have many requirements on the application, but we can have complicated precedence constraints etc.

Memory management

- *Determinism*

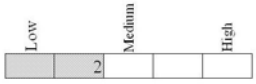
MARS does not provide any construct that jeopardizes memory access times, i.e. dynamic or virtual memory. Even message buffers have to be allocated before run-time, and all messages in a MARS system has the same size.

- *Inventiveness*



No extra ordinary solutions here, just the simplest.

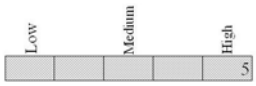
- *Usefulness*



The memory management system is useful in hard real-time environments. But it is not flexible or provide any new ideas, such things that this comparison give credits for.

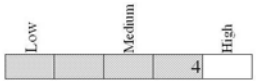
Interprocess communication

- *Determinism*



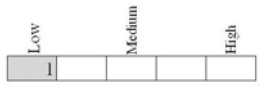
Off-line scheduled messages, and the redundant properties, aims for determinism even when messages are lost.

- *Inventiveness*



Redundant communication buses, the implicit flow-control achieved with the messages and the implicit use of all messages for clock synchronization purposes are exceptional solutions.

- *Usefulness*



Off-line scheduled messages limit the usefulness to very special applications.

MontaVista Linux

MontaVista Linux [WEI01] is a commercial open-source Linux distribution with multiprocessor and real-time support. A popular approach to build a real-time Linux distribution is to make use of the RT-Linux [YOD97] kernel. To make it clear MontaVista Linux is not based on the RT-Linux kernel, the RT-Linux approach is just described here as orientation.

A RT-Linux based operating system is built with an approach that in theory disables all common Linux functionality, and replaces it with a new RT-Linux kernel. The RT-Linux kernel is aimed for use in hard real-time systems, and supports only single processor machines. The common Linux kernel is used to boot the system and directly after the boot sequence the RT-Linux kernel takes control of the system. Although the two kernels coexist, it is the RT-Linux kernel that is in charge. The idea is that it shall be possible to let non real-time tasks take advantage of the rich Linux API, i.e. monitoring tasks and soft tasks. The tasks that are dispatched by the RT-Linux kernel always have higher priority, but use a simple and real-time limited API. In the original description [YOD97] real-time tasks are scheduled by fixed priority, earliest deadline or rate monotonic, configurable by choosing a scheduling module. No dynamic or virtual memory is used, and a single processor message based interprocess communication mechanism that adopts real-time theories is provided.

MontaVista Linux uses another approach than the common kernel substitution approach; instead the Linux kernel itself is tuned to provide real-time support. The scheduling algorithm and interrupt handling are the main targets for modification. MontaVista Linux has support for almost all common processor platforms and as other commercial RTOS tries to fit with almost all possible applications. The multiprocessor support seems to be directed towards back plane networking with CompactPCI busses, examples of that are given in [BWE01]. It should therefore be possible to use MontaVista Linux on hardware architectures like the SARA system (described in this survey), since the CompactPCI bus interface are supported.

Scheduling

The supported task type is the standard Linux thread, with a boosted priority scheme. Standard Linux threads are allowed to have priorities ranging between 0-99, MontaVista Linux defines priorities up to 128. The threads are free to take advantage of the whole Linux API. The scheduling algorithm is the main improvement over the standard Linux system; actually the real-time support breaks down into the preemption patch, and a second scheduling level [WEI01].

The contribution with the preemption patch is that it does not break or stretch the standard Linux API. Until recently it was not considered possible by the Linux community to create a fully preemptable Linux distribution, without limit or

extend the standard libraries and systems calls. The solution that the MontaVista distribution has come up with resides from a special Linux dialect, the Symmetric Multi Processing (SMP) version of Linux. The SMP Linux kernel according to [WEI01] already offers a highly preemptable response. Although it is not intended for use in real-time systems, it is here the preemption patch comes into the picture. The patch modifies the spin-lock construct offered by the SMP kernel, i.e. the standard IPC construct, to become a preemption lock. This construct is then used to protect critical sections. Additionally the preemption patch creates possibilities for fast response times, through interrupts. When an interrupt has occurred, the patch modifies the kernel to allow rescheduling on return from interrupt if a new task has become ready. The main preemption techniques is taken from the SMP distribution, although the MontaVista distribution is not intended to be used on SMP:s, since the main communication primitive is removed.

The scheduling algorithm is based on fixed priorities of 128 levels and overrides the standard timesharing scheduler provided by Linux. The scheduling algorithm is transparent, it simply means that if no real-time tasks are ready for execution the control is given to the standard scheduling algorithm. In this way real-time and non real-time tasks can coexist, but the non real-time tasks are only scheduled when there are idle time left.

Memory management

It is the standard Linux memory management routines that are offered. That include dynamic allocations, and virtual memory optimized for throughput. No special treatment of real-time tasks is implemented.

Interprocess communication

The standard Linux communication primitives are used. It is a rich variety of possibilities for IPC on a Linux system, since modules and libraries are freely distributed. Included from the beginning in the MontaVista distribution is for instance several primitives for semaphores, several message passing systems, one shared memory system, two signal systems and a watchdog mechanism. Although none of the included mechanisms are intended for real-time systems.

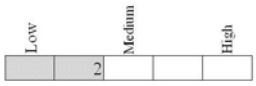
Conclusions

The MontaVista Linux distribution is aimed for real-time systems, today the real-time support is limited. However a reasonable prediction is that Linux will be used more and more in embedded and real-time systems. Papers and news groups discussing how to port from a conventional RTOS to MontaVista Linux are for instance easy to find and it is a lot of peoples involved in Linux related projects. MontaVista Linux cannot in its present release be treated as a RTOS with hard real-time support, but it is probably useable in more soft real-time applications with the rich and among programmers well known Linux API.

Evaluation

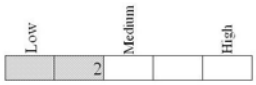
Scheduling

- *Determinism*



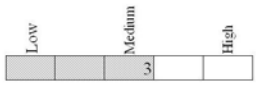
The scheduling algorithm itself is maybe one of the most used algorithms in the real-time domain. The WCET:s may be hard to estimate, since underlying routines for memory management and IPC does not provide bounded execution times.

- *Inventiveness*



No extraordinary solution, the contribution is the modification of a system call and the return from interrupt routine provided by a patch.

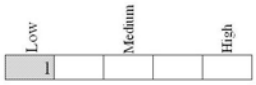
- *Usefulness*



It is surely a useful scheduling algorithm, it is this type of priority driven algorithms that have been used in most commercial RTOS in the past. But we are restricted to only this algorithm and, we cannot use this RTOS in a hard real-time application.

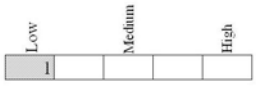
Memory management

- *Determinism*



Any efforts for a deterministic memory management system have not been done, we have dynamic allocation and virtual memory optimized for throughput.

- *Inventiveness*



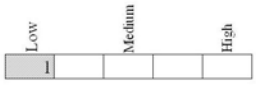
None of the offered constructs are implemented especially for this distribution. It is open source code, shipped with almost all Linux distributions.

- *Usefulness*

On the other hand it is very useful, many programmers are used to the standard Linux API. It is also easy to extend the system with other freely distributed packages.

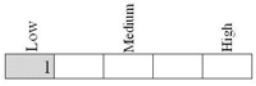
Interprocess communication

- *Determinism*



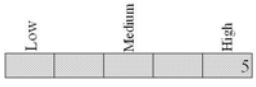
No bounded delays can be provided.

- *Inventiveness*



Standard solutions shipped with almost all Linux distributions.

- *Usefulness*



A broad range of communication possibilities are available for Linux today, it is easy and free to use them.

OSE

OSE [ENE1] is a commercial general-purpose real-time operating system, merchandised by ENEA OSE systems. OSE is the first commercial RTOS, that is IEC 61508 certified [ENE3]. As most commercial operating systems, OSE supports different hardware configurations (processors etc) and can offer solutions for almost any purpose, examples include a web-server as interface between the embedded system and the internet and a soft kernel that are able to simulate a complete system before run-time.

The set of supported processor's seems to be almost all that is common used in embedded and real-time systems (Arm, Motorola, MIPS, Lucent, Mitsubishi, Intel, IBM etc), and OSE supports single processor solutions as well as scalable distributed solutions. In the distributed case, an image of the kernel is placed on every CPU in the system and the kernels communicate through a message-passing paradigm. Task migration is not supported. Instead tasks belong to the node where they are created, so no task placement algorithm or global scheduler exists.

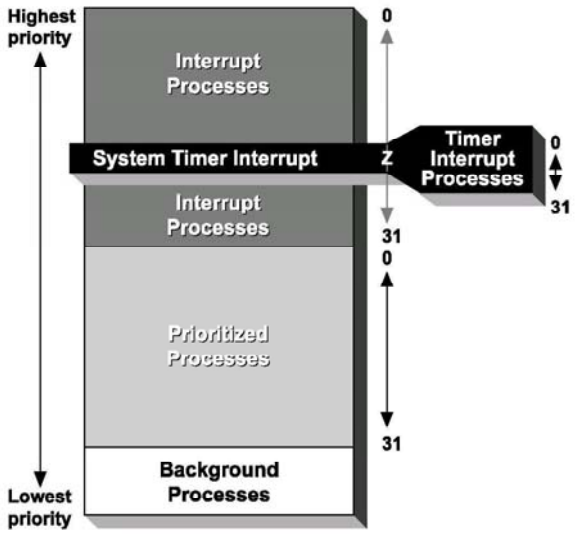


Figure 6, processes priority.

Scheduling

The scheduling algorithm [ENE1] is preemptive and priority based. In OSE terminology, processes are approximately the same as the tasks introduced in the design issues section. The supported process types are interrupt processes, timer interrupt processes, prioritized processes, background processes and phantom

processes. The different process types have an internal priority as shown in figure 6, except for the background processes the priorities inside a certain type can be from 0-31. Note that the periodic processes (timer interrupt processes) always run with priority Z from the dispatchers point of view, which is the priority of the system timer interrupt. Prioritized and background processes has for every priority level a round-robin queue containing all ready processes of that priority level. The first process in the queue is the process currently running on that priority level.

- Interrupt processes are called in response to a hardware interrupt or a software event. These processes has the highest priorities in the system, therefore an interrupt process can only be interrupted by another interrupt process with higher priority. 32 different interrupt priority levels exist. Interrupt processes become ready, executes and finally terminates.
- Timer-interrupt processes are executed with the same priority level as the system timer interrupt, but internally has 32 different priorities. These processes are OSE's support for periodic tasks. Their invocation semantics are the same as ordinary interrupt processes, becomes ready, executes and terminates.
- Prioritized processes is said to be the most common type of process in OSE, they are written as infinite loops that will run until they become interrupted by a process with higher priority or suspend themselves. Prioritized processes have 32 different priorities, but it is still possible to have more than 32 prioritized processes. For each priority level that contains prioritized processes, the Kernel has a round-robin queue, containing all ready processes of the current priority level. The first process in the queue is the process currently running on that priority level. Each process on that priority level, share the levels execution time like an ordinary time-sharing operating system.
- Background processes have the lowest priority in the system and run in a true time-sharing mode against each others (no internal priorities), they are like prioritized processes written as infinite loops. A background process cannot suspend itself, if several background processes exists, they always run until their round-robin time slice has expired.
- Phantom processes does not contain any code, they are used in conjunction with redirection tables to form a logical channel when communicating outside the target system (CPU). A phantom process is used as an image of the receiver processes on the sender's node. More about phantom processes in communication section.

Memory management

Memory management [ENE1] in OSE contributes with the possibility to have several memory groups; the different memory groups are called memory pools. A pool is an area of memory, which message buffers, stacks and kernel areas are allocated. In all possible configurations there is always one and only one global

memory pool called the system pool. The system pool is the first pool created and its existence is crucial for the kernel, if this pool gets corrupted the whole system will crash. It is possible to build a system where all processes allocate their memory from the global system pool. From a safety point of view, it is better to create local pools for user created process. This is the desired way of user memory handling, OSE also support grouping processes together into logical blocks, some system calls work on entire blocks rather than single processes (start, kill etc). It may be a structured and clear approach to design an OSE application, with process groups allocated in different memory pools.

Interprocess communication

Communication in OSE [KAL, ENE2] seems to be ideally constructed for single processor solutions, but with the possibility to support almost any multiprocessor configuration. The possibilities for communication and synchronization between processes in OSE are message passing and semaphores, but in a multiprocessor environment semaphores cannot be used. However messages are not handled in the same way in a single processor environment, as in a multiprocessor environment. A kind of optimized message semantics (concerning message copying) is used in the first case. OSE seems to support some kind of remote procedure calls also, but it is just mentioned as quickest in [ENE1].

Semaphores are only visible within a single processor and can be divided into two classes, namely fast semaphores and “ordinary” semaphores. Fast semaphores are owned by a single process and hence only can be used by that process. All processes can access an ordinary semaphore, but there is no access protocol for avoidance of priority inversion present.

The recommended communication method in OSE is message passing. OSE processes communicate directly to each other through intelligent messages also referred to as OSE signals. A message in OSE can only have one owner, this ensure the integrity of the message. This mechanism is designed to avoid the use of global memories (mail boxes and semaphores) for communication. The semantics of the messages or the message passing method is named direct messages, as the name unveils the messages are sent directly to a process and not to a mailbox implemented in global memory or something similar. This is of course one of the reasons, why a message only has one owner. The process addresses the messages with a receiver task rather than with an id of an intervened message queue owned by the RTOS. The real advantage with this method is potentially less data copying. In other message passing systems with a mailbox owned by the RTOS, the messages are first often copied from the senders memory area to a memory area owned by the RTOS, then a second copy from the RTOS memory area to the receivers memory area when the receiver reads the message. In the direct message-passing paradigm the maximum number of copies will be one (from the sender to the receiver), but in many cases there will be no copying at all. The desired semantics of the messages is just pointer copying, the

"payload" never actually moves in this way. This must require hard restrictions on the memory management, since tasks actually are poking in each other's memory areas. Of course the data will be copied when tasks are in different physical memory domains, when trying to pass a message across memory address spaces it is recognized by OSE and copying has to be performed but just one copy as mentioned above.

The OSE message-passing concept is totally transparent to the location of the communicating processes. This is implemented with phantom processes in OSE, e.g. processes without any executable code. The link-handlers on each node are used to set up logical channels, through creating phantom processes, which on the sender's node represent the receiving process and on the receiver's node the sender process. The only purpose of the phantom processes on the sender's node is to hold a redirection table, so that all messages sent to it will be redirected to the link handler instead. The link handler is then responsible for sending these messages to the link handler in the target where the receiver resides. The phantom process on the receiver's node are used to send messages in the other direction, a schematic picture can be seen in figure 7, the logical channel between process A on target 1 and process B on target 2 is represented with the dotted arrow. The phantom processes A' and B' and the link handlers LH are used for the real message transport.

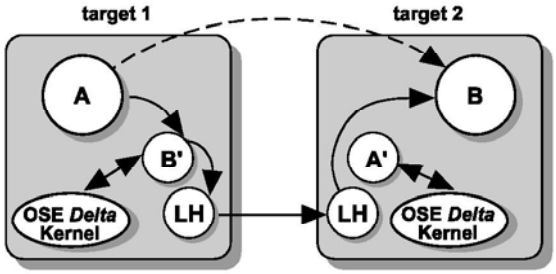


Figure 7, Tasks communication over memory domains.

Conclusions

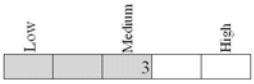
OSE appears to be a typical commercial RTOS, with support for a lot of hardware configurations and a priority driven scheduling algorithm. The message passing mechanism with active messages seems to be the special feature that OSE offers. It seems to be a fast method and we avoids a lot of copying, semantically speaking the implementation is totally transparent to the location of the communicating processes. But in a hard real-time environment we cannot treat the placement of the communicating processes in OSE as arbitrary. The communication time will increase drastically between tasks on the same node and tasks on different physical nodes. Overall the communication mechanisms within a node appears to be more careful constructed than the global communication

mechanisms, collisions on the network and other unexpected communication disturbances seems to be a question for the application designer.

Evaluation

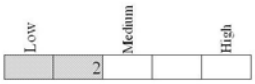
Scheduling

- *Determinism*



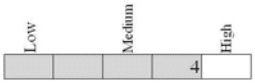
The priorities can be set according to RM, and the analysis can be performed with RM analysis. The problems would probably be to make exact WCET analysis, mainly because of unbounded message delays and the memory management. Another problem is the interrupt handling, interrupts have always the highest priority. This is a potential hazard, if any device that causes interrupts fail, the system will probably break down.

- *Inventiveness*



Nothing extraordinary here, a simple priority based scheduler.

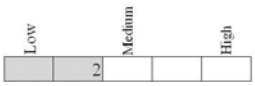
- *Usefulness*



OSE is probably easy to use, and it seems to be easy to adopt real-world problem to the different processes. We have the possibility to use different process types, the question is if that is a possibility or a limit. For instance an interrupt process must always be of higher priority than a prioritized process. We also have a limit in the maximum number of processes, since we have a limited number of priorities.

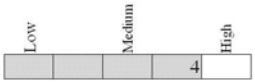
Memory management

- *Determinism*



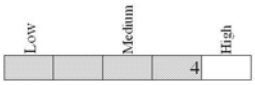
Dynamic memory allocations, without any efforts of additional determinism.

- *Inventiveness*



OSE have with real-time measurements an advanced memory management system, which allows dynamic allocation, and protection within different memory groups.

- *Usefulness*

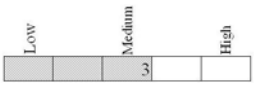


The memory management system in OSE is probably useful, dynamic allocations are always useful. But it is the protection mechanism, with memory pool system calls that seems to be the most useful mechanism. Usually memory pools and address spaces are defined off-line, but in OSE it seems to be possible to configure these during

runtime.

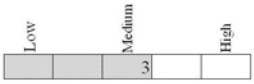
Interprocess communication

- *Determinism*



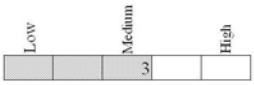
Bounded propagation times, or any collision avoidance methods are not supported. Although the communication mechanism is what is typically provided.

- *Inventiveness*



The direct message-passing concept is an excellent solution that minimizes the data copying and protects address spaces from each other. Although it is not so revolutionary, it is ordinary messages that require additional processes to propagate between two address spaces.

- *Usefulness*



OSE provides what is necessary and nothing more. Although the semaphores could be usable between different address spaces, it is a reasonable limit.

RT-Mach

RT-Mach [NAK90] is a further development of Mach [ACC86], which is a research operating system with accomplished research goals back in 1994. The RT-Mach project is still a living research project; the philosophy behind RT-Mach is firmly based on real-time scheduling theory and in particular on priority-driven preemptive scheduling. RT-Mach extends this philosophy by adding a fundamental OS notion of temporal protection that enables the timing behaviour of applications to be isolated from one another. One of the design goals with the original Mach operating system was high portability, it seems like this goal was achieved particularly through a memory management system that separates the machine dependent and independent parts in an extremely clear and unusual way [TAN92]. Because of the high portability of the underlying Mach operating system, one of the goals with RT-Mach was to provide a common real-time computing environment in various machine architectures including single board, multi processor and distributed real time systems [NAK90].

Scheduling

The active entities in RT-Mach are threads [NAK90], processes also mentioned as tasks in our references are passive entities. A thread can be defined for a real-time or non real-time activity. All threads real-time or not must at least be specified by a procedure name and a stack descriptor, which specifies the size and address of the private stack region. The real-time threads, which we are interested in, have a number of additional attributes. For a real-time thread, timing attributes must be defined by a timing attribute descriptor. Other attributes are hard or soft based on its deadline and periodic or aperiodic based on the nature of its activity. A periodic threads timing attributes is mainly defined by WCET, deadline, period time, start time and phase offset. The meaning of this parameters in a periodic thread is as follows, a new instantiation of the thread will be scheduled at its start time and then repeat the activity every multiple of the period time. The phase offset is used to adjust a ready time within each period. The thread will execute a maximum time of WCET and must at all activations finish before its deadline. An aperiodic threads timing attributes is defined by WCET, deadline, worst-case interarrival time. The meaning of WCET and deadline are the same as for a periodic thread; the worst-case interarrival time expresses the minimum time between two activations of the thread.

The scheduling algorithm(s) in RT-Mach [ARA93, NAK90] has been heavily influenced by the goal of the predecessor Mach to run on multiprocessor architectures, additional abstractions, as processor sets and task allocation exist. Every CPU in a multiprocessor system can be assigned to a processor set, which is an operating system and scheduling abstraction. Each processor belongs exactly to one processor set and a processor set is the set of one or more processors. Threads can also be assigned to processor sets, and it is within the assigned processor set that a certain thread is scheduled. A thread cannot migrate to another

processor set, but it is a form of thread migration that occurs inside a processor set even if the designers never mention or claim that. Notice that the processor set solution also works for one processor. Each processor set has a run queue; it is from this queue a task to execute is chosen regarding on the processor sets scheduling policy. A certain thread can execute on different processors inside the processor set.
Different scheduling policies can be applied to different processor sets, *Rate Monotonic*, *Earliest Deadline*, *Fixed Priority*, *Deferrable Server* and *Sporadic Server* have at least been implemented together with the round-robin time slice inheritance from Mach.
The scheduling algorithm for a processor set can be changed during runtime by user level code; the scheduling policy representation inside a processor set is a self-contained object. The meaning of self-contained is that it is separated from the actual dispatch routine, so a change of the scheduler is just a change of the run queue manager. The actual change of schedule algorithm for a user thread is two system calls, one to get the scheduler object and a second to set a new schedule object.
Above the actual scheduling algorithm RT-Mach has another algorithm for bandwidth allocation and overload handling, ITDS (Integrated Time Driven Scheduler). By bandwidth reservation for the hard activities, the ITDS algorithm knows how much time there is available for activities with soft deadlines.

Memory management

An effort to more deterministic memory management than the inheritance from Mach offers is described in [NAK90]. Mach has an unsuitable memory management technique for real-time usage called *lazy evaluation*. The problem with this technique is that it causes unbounded delays. For instance if a thread allocates a region of memory, the system does not allocate the object unless the thread touches the region and causes a page-fault. To make it possible to create more deterministic accesses, a “patch” in form of a system call is provided. The call named *vm_vire* makes sure that the portion of memory is pre-allocated, or allocated without the lazy evaluation technique.

Interprocess communication

Shared memory is a natural way of communication in RT-Mach, since no special mechanism is needed for threads created inside the same task to share memory objects, they all share the same address space automatically. Remember that threads are the only active entities and that tasks are passive, so it not unlikely to imagine a system where one process creates all threads.

Synchronization (mutual exclusion) through locks is described in [NAK90] and was included in the design the RT-Mach. Mutual exclusion is an important feature since all threads created inside the same process shares the same resources. The lock and unlock pair provide a priority inheritance mechanism to avoid priority

inversions. The mutual exclusion in RT-Mach between threads is therefore deterministic and suitable for real-time computing.

A predictable and deterministic communication protocol suited for use in a hard real-time environment was on the other hand not included in the design goals of RT-Mach. The predecessor Mach supports a variety of communication techniques (message passing, remote procedure calls, byte streams etc), but these are optimized for throughput rather than determinism and are therefore not included in the presentation of RT-Mach in [NAK90]. According to [NAK93] the original Mach communication primitives are heavily used in RT-Mach anyway. A method (RT-IPC) for implementation of a suitable communication protocol in RT-Mach has been proposed in [NAK93], and it is this technique that is described here. RT-IPC is implemented above the original message passing method and uses the same interface, but operates in another name space and logical port. A name space in the RT-Mach kernel has the same semantic meaning as a namespace in a regular C++ program. In figure 8, the two different namespaces of a communication port is illustrated.

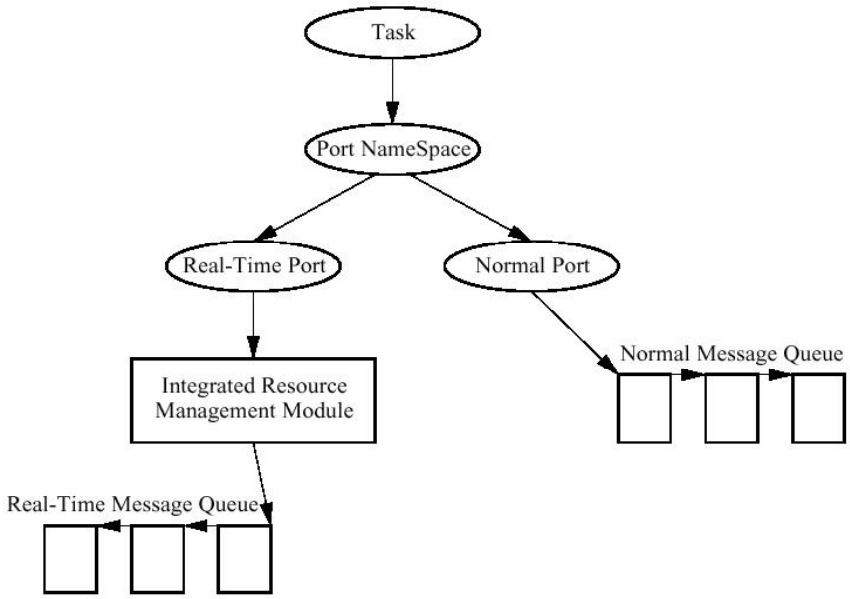


Figure 8, In RT-Mach, it is possible for coexistence of real-time messages and ordinary messages.

The added real-time features, which aim for determinism when using a real-time port, are listed below.

- Message buffers must be pre-allocated (static or dynamic allocation), to avoid unpredictable allocation delays.
- Priority inheritance to the message-sending server. Messages that are going to be sent are first transferred to a message sending server, the

server inheritances the highest priority of all the sending threads at any instance in time.
- Message queue ordering. The message queue to the sending server is ordered in priority of the messages and each messages priority is the same as the sending threads.
- Priority hand-off. The receiver propagates the priority of the sender.

Conclusions

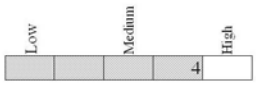
RT-Mach is surprisingly well suited for real-time applications, although the building base was an ordinary time-sharing system. The large amount of scheduling algorithms provided and the possibility to use different scheduling algorithms for different task groups seems to be the main contribution. In this way we potentially have more freedom when trying to translate the requirements of different activities into tasks. It should also be possible to compare and evaluate different scheduling algorithms practically, since we are able to hold all other factors except the scheduling algorithm constant.

Efforts to make the communication and memory handling mechanisms more predictable have been made, that is good. But any worst-case scenarios or any attempts to show the bounded delays were not presented.

Evaluation

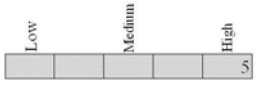
Scheduling

- *Determinism*



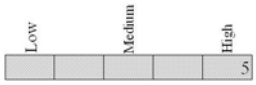
The scheduling algorithms are all off-line analyzable, and we have overload handling through the ITDS algorithm. The WCET estimation should be partially possible, although the memory management system seems to be a little bit weak form this point of view. The problem may lie in the thread migration inside a processor set.

- *Inventiveness*



Task migration inside a processor set, change of scheduling algorithm during runtime, different scheduling algorithms on different processor sets and global scheduling gives highest score here.

- *Usefulness*



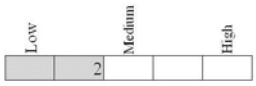
All desired task types, and very high flexibility. The flexibility is achieved through the possibility of different scheduling algorithms on different processor sets and the possibility to change it during run-time.

Memory management

- *Determinism*

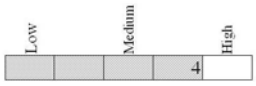
Efforts to more predictive memory management have been implemented, but on the other hand we still have dynamic allocations, and other inheritances optimized for throughput rather than determinism.

- *Inventiveness*



Here we have an unsuitable memory system that have been adopted to real-time use, with a patch in form of an additional system call. That is not inventive; it is rather an *ad hoc* solution.

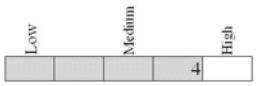
- *Usefulness*



It is useful with dynamic memory allocation, but the need of an additional primitive is not a desired feature. The memory system separates machine dependent and independent parts in a clear and unusual way that is clearly useful. This separation is said to make the whole operating system more portable than others.

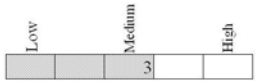
Interprocess communication

- *Determinism*



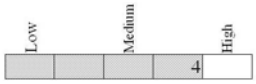
The priority inversion problem is considered and cares to real time aspects are taken. The proposed communication protocol seems to provide a deterministic access to the communication medium. A potential problem is to let ordinary traffic coexist with real-time traffic, of course we can avoid that in the design but it is still a hazard.

- *Inventiveness*



The contribution is namespaces, which show us how to extend an existing message passing mechanism into a more real-time suited one.

- *Usefulness*



Message passing, shared memory and synchronization primitives are available. More than necessary.

RTEMS

RTEMS [OAR00] is a free, open-source real-time kernel that provides a high performance environment for embedded systems. Initially RTEMS stood for the Real-Time Executive for Missile Systems but as it became clear that the working range extended far beyond missiles, the "M" changed to mean Military. There are two implementations of RTEMS, one Ada implementation and one C implementation. The C version changed the "M" to mean Multiprocessor while the Ada version remained with the "M" meaning Military.
The main issue during the development of RTEMS was portability. RTEMS is designed to isolate the hardware dependencies in specific board support packages. Therefore real-time applications should be easily ported to different processors, for instance Intel i80386 and above, Motorola MC68xxx, PowerPC and SPARC.
RTEMS adopts an object-oriented model, which increase the reusability and extendibility of code. Tasks, message queues, semaphores, memory regions, memory partitions, timers, ports and rate monotonic periods are all objects that can be dynamically created, deleted and manipulated.

Scheduling

The task manager in RTEMS provides a comprehensive set of directives to create, delete and administrate tasks. All tasks have priorities that are used by the scheduler. A task is defined as the smallest thread of execution that can compete on its own for system resources. During system initialization a TCB is allocated to each task. The TCB contains all information that is pertinent to the execution of the task. A task can support either preemption or non-preemption. A task that supports preemption leaves the processor to a higher priority task that is ready, even if the lower priority task is in the execution state, i.e. the task has not finish the execution. A task that has disabled preemption retains the control of the processor as long as it is in the execution state.
RTEMS supports 255 levels of priorities, and several tasks are allowed to have the same priority. A *timeslicing* component is used by the scheduler to determine how to allocate the processor to tasks of equal priority. If timeslicing is enabled, then the time a task can execute is limited. The processor is then allocated to another ready task of equal priority, i.e. round robin scheduling within individual priority groups. If timeslicing is disabled, then the task will execute until another higher priority task becomes ready. An interrupt level component is used to determine which interrupts will be enabled during execution of a task. This is done by setting an interrupt-level for each task. Another component is the asynchronous processing component, which is used to determine when received signals are to be processed by the task. This component does only affect tasks that have established a routine to process asynchronous signals. If signal processing is enabled, signals set to the task will be processed next time the task executes, otherwise all signals received will remain posted until signal processing is enabled.

RTEMS' scheduling concept is to provide immediate response to specific external events. The scheduler allocates the processor using a priority-based, preemptive algorithm extended to provide timeslicing within individual priority groups. The goal with the RTEMS algorithm is to guarantee that the executing task on a processor at any instance of time is the one with the highest priority among all tasks in the ready state. The user of the system assigns priority levels to the tasks when they are created. The priority levels can also be altered during run-time. A mechanism for altering the RTEMS scheduling algorithm is called manual round robin. This allows a task to give up the processor and immediately returned to the ready queue. If no other task with the same priority is ready to run, then the executing task will not give up the processor.

A rate monotonic manager is provided, which facilities the manage (of execution) of periodic tasks. This manager was designed to support application programmers that want to utilize the *Rate Monotonic Scheduling Algorithm* (RM) to guarantee that all periodic tasks will meet their deadlines, even under transient overload conditions, by means of schedulability rules for RM. If there are several ready tasks of equal priority level, the task that have been ready longest time will execute first. The RM manager definitions of different task types:

- Periodic task – tasks that executes at regular intervals (periods). Periodic tasks have hard deadlines, which are the same as their periods.
- Aperiodic task – tasks that executes at irregular intervals with soft deadlines. That means that the deadlines are not rigid, but adequate response times are desirable.
- Sporadic task – aperiodic tasks with hard deadlines and minimum interarrival times.

All tasks with hard deadlines (periodic and sporadic tasks) are typically referred as critical tasks, while tasks with soft deadlines (aperiodic tasks) are referred as non-critical tasks. The critical tasks are scheduled using RMS, and the non-critical tasks are scheduled as background tasks, i.e. by assigning priorities such that the lowest priority critical task has a higher priority level than the highest priority non-critical task. The motivation to this type of scheduling is that all critical tasks must be guaranteed execution (using the RM schedulability analysis), even under transient overload, while schedulability is not guaranteed for non-critical tasks.

Memory management

A processor may support any combination of memory models ranging from pure physical addressing to complex demand paged virtual memory systems. Regardless of the support from the processor RTEMS supports only a flat memory model, which ranges contiguously over the processor's available address space. RTEMS does not support segmentation or virtual memory of any kind.
The RTEMS memory manager provides dynamic memory allocation and address translation. The dynamic memory allocation is required by applications with memory requirements that vary during execution. The address translation

mechanism is used by applications that share memory with other processors. The owner of a shared memory accesses the memory using internal addresses, while other processors must use external addresses.

Interprocess communication

RTEMS provide different managers that is dedicated to communication and synchronization:

- Semaphore
- Message Queue
- Event
- Signal

The semaphore manager is used when mutual exclusion of one or more shared resources is necessary. Both binary and counting semaphores are supported by RTEMS. A binary semaphore is restricted to either zero or one, while counting semaphores are restricted to all positive integer values. A counting semaphore is typically used to control access to pool of two or more shared resources. RTEMS support both the Priority Inheritance and the Priority Ceiling protocol to solve the priority inversion problem.

The message manager supports both communication and synchronization between different tasks. A message is a variable length buffer where information can be stored. The message queues can contain variable number of messages and they are sorted in FIFO order, with the exception of *urgent* messages that can be placed at the head of a queue. Tasks can either do a (block) wait for a message to arrive at a queue or poll a queue for the arrival of a message.

The event manager provides a high performance synchronization intertask communication mechanism. A task uses an event flag to inform another task of the occurrence of significant situation.

The signal manager supports asynchronous communication and is typically used for exception handling. The directives provided are establish an asynchronous signal routine (ASR) and send signal to a task.

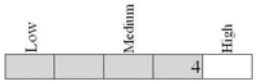
Conclusions

RTEMS is probably one of the most distinguishing operating system in this paper. The fact that RTEMS is a free, open-source real-time kernel is the main motivation. The main issue during the development of RTEMS was portability. RTEMS isolates the hardware dependencies in specific board support packages. So real-time applications is easily ported to different processor families.

Evaluation

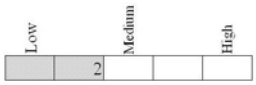
Scheduling

- *Determinism*



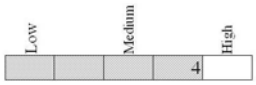
Off-line analysis is possible, we can also disable interrupts during task execution. We can also set priorities on interrupts, higher, lower or in between tasks. Although interrupts exist and for instance dynamic memory allocation may jeopardize the WCET estimation.

- *Inventiveness*



Nothing special here. Just a simple priority based scheduler.

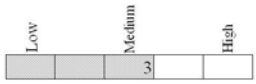
- *Usefulness*



In RTEMS it seems to be easy to adopt real-world problems. We can use interrupts freely, although we miss real freedom in the choice of scheduling algorithms etc.

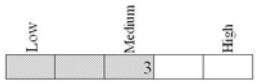
Memory management

- *Determinism*



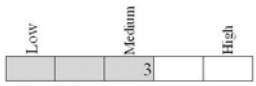
RTEMS does not support segmentation or virtual memory of any kind. But dynamic memory allocations are a little problem.

- *Inventiveness*



RTEMS supports dynamic allocation of memory.

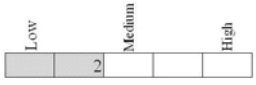
- *Usefulness*



Dynamic memory is an advantage when a tasks memory requirements change during runtime.

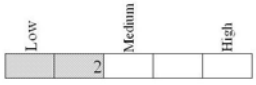
Interprocess communication

- *Determinism*



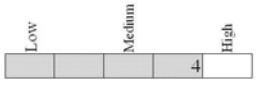
The fact that the message-queues are sorted in FIFO order, may jeopardize the determinism of the message passing mechanism. The transmission times are not bounded.

- *Inventiveness*



No distinguishing mechanism is provided.

- *Usefulness*



Several different communication and synchronization primitives aim for a high grade of usefulness.

SARA

The hardware architecture of the SARA system [FUR99] is divided into local CPU board, bus arbitrator, global RAM, I/O and *Real-Time Unit* (RTU) [FUR95, ADO96]. The RTU is a co-processor that performs real-time operating system functions. The RTU is further investigated in the scheduling section. The processor boards, RTU and other facilities 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 CPU-board (system board) that handles the arbitration, clock-distribution, etc on the back plane. An overview of the SARA-system is shown in figure 9.

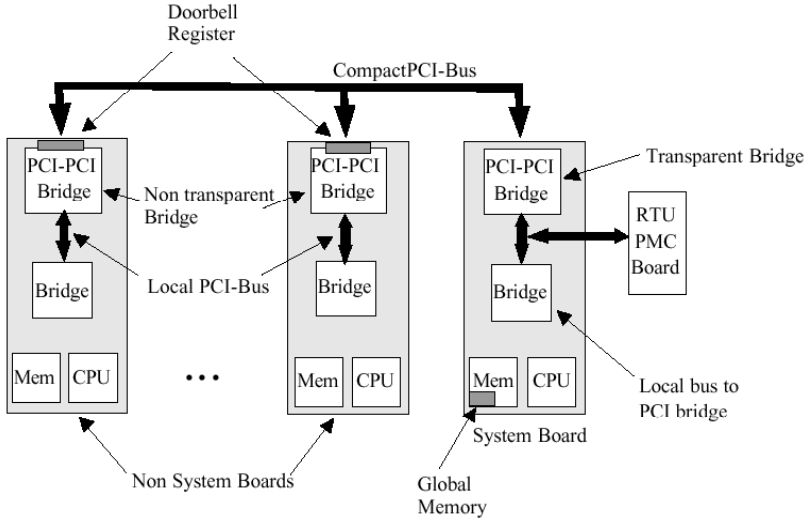


Figure 9: Block diagram of SARA system.

The RTU is attached to the local PCI bus on the system board, when the RTU signals something for instance task-switch, it will generate an interrupt on the local PCI bridge on the system board. As the local bus on the system board is attached with a transparent bridge to the global CPCI bus, all interrupts will become visible for all other boards. The problem with this is that the interrupts is signaled through four interrupt-lines that are available on the CPCI bus. If more than four boards are inserted, some boards have to share interrupts and this may cause latencies. The SARA system provides a solution for the latencies through 16 bit wide doorbell registers in all non-transparent bridges; all doorbell registers in the system have a unique address. When the RTU generates an interrupt, it generates a write-cycle from the local bus where it is hosted to all doorbell registers it wants to access. The interrupts to the system board is generated through the standard interrupt lines. The doorbell registers not only provide a solution to the latencies with shared interrupts, through them it is also possible to use 2^{16} different interrupts.

Scheduling

As the applications become larger and more complex, the demands on real-time kernels increase every year. A real-time kernel must give short and predictable response. For the purpose of meeting these demands in time, the RTU, which is a real-time kernel coprocessor, has been implemented in hardware [FUR95]. By implementing the kernel in hardware, the performance and determinism can be improved.

The RTU is a small single- or multiprocessor multitasking real-time kernel. It can handle 1 to 3 processors, 64 tasks and 8 priority levels. The interface to the RTU is read- and writeable registers, which makes it easy to port to different types of processors. (See the register model section for further details).

Advantages of using a hardware kernel:

- Flexible, the only software segment, which must be rewritten on a new processor type, is the assembler code for taskswitch.
- The performance increases. The scheduling of a task is done within 5 to 80 clock cycles, and the scheduling doesn't load the processor.
- Deterministic execution of the service instructions, which make it easier to calculate the execution time for a system.
- No clock tick interrupts are needed, because the RTU handles the task scheduling.
- The response time decreases for all service calls, because the RTU is designed of parallel hardware.
- Easier software development, because the kernel code doesn't have to be executed by the processor.
- Easier understand ability for the system, because the real-time kernel is separated from the RAM.
- A more safe execution of the real-time kernel, because no interference between the kernel functions are possible (designed in physical separated parts).
- No interrupt handler for external interrupts has to be implemented in software.
- Easier to debug without affecting it, because the service calls can be logged on the bus.

The RTU consists of several units, where each unit represents certain functionality. Depending on what kind of system the RTU is going to be used in, it can be configured with different units, i.e. it depends on the service requirements. The simplest RTU consists only a scheduler. Functionalities available (1999):

- **Scheduler**: The scheduling algorithm is priority-based and supports preemption of tasks. The goal is to ensure that the task, which is

executing on the processor at an arbitrary time, is the one with the highest priority among all tasks in the ready-queue. Only one task can execute on a CPU at a time, so the maximum number of executing tasks is the same as number of CPU's. The RTU has several ready queues, one ready-queue for each CPU and a "global" ready queue, i.e. it contains task that can be executed at any of the connected CPU's. The scheduler compares the own and the global queue in parallel, for each CPU in order to execute the task with highest priority. There are two events that can do a taskswitch. The task itself can request a switch and the scheduler can interrupt the executing task when a task with higher priority is ready.

- **Delay**: Holds a task in the delay queue until the delay time has expired. Then the task is placed in the ready queue.
- **Periodic start**: Holds a task in the periodic queue until the periodic time has expired. Then the task is placed in the ready queue.
- **Watchdog**: A watchdog task is suspended until the watchdog timer has expired 100 ms. After 100 ms has expired, the task becomes ready to execute.
- **Semaphores**: The RTU can hold four tasks in the semaphore queue. It is a FIFO queue, which means that the first task in the queue gets the semaphore, when it is released (free). When a task gets the semaphore, it is removed from the semaphore queue and sent to the ready queue.
- **Event flags**: the RTU can hold four tasks in the event flag queue. When the flag is set, all tasks in the queue are sent to the ready queue.
- **External interrupts**: Holds a task until an interrupt corresponding to the task's interrupt level occurs. Then the task is sent to the ready queue.
- **Interface**: The RTU has this I/O interface between the real-time functions and the PCI bus.

Register model

Each processor have three dedicated registers: cpu_control_register, next_task_id and cpu_status_register. The RTU has two registers that hold the overall information: rtu_status_register and rtu_control_register. Finally there are two more registers: svc_instruction_register and svc_semaphore_register. They are both service-call registers (SVC). The register address is calculated by adding a base address to the address offset in the table 6.

Register name	Address offset (hex)	Read/Write	Size
CPU_STATUS_REGISTER (0 to 2)	0, 2, 4	R	16 bit
RTU_STATUS_REGISTER	6	R	16 bit
RTU_CONTROL_REGISTER	8	r/w	16 bit
NEXT_TASK_ID (0 to 2)	A, C, E	R	16 bit
SVC_INSTRUCTION_REGISTER	10	W	16 bit
SVC_SEMAPHORE_REGISTER	12	r/w	8 bit
CPU_CONTROL_REGISTER (0 to 2)	16, 18, 1A	r/w	16 bit

Table 6, register offsets.

Memory management

Communication and synchronization between different processes in the system is performed through the global memory that resides on the system board. The global memory is also used for TCB and stacks. As shown in figure 1, 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. The system board has a transparent bridge, while other boards have bridges that remap addresses on one bus to another address on the other bus. With this non-transparent bridges address collisions on the CPCI bus can be avoided and all boards can use it's full address range on the local PCI bus.

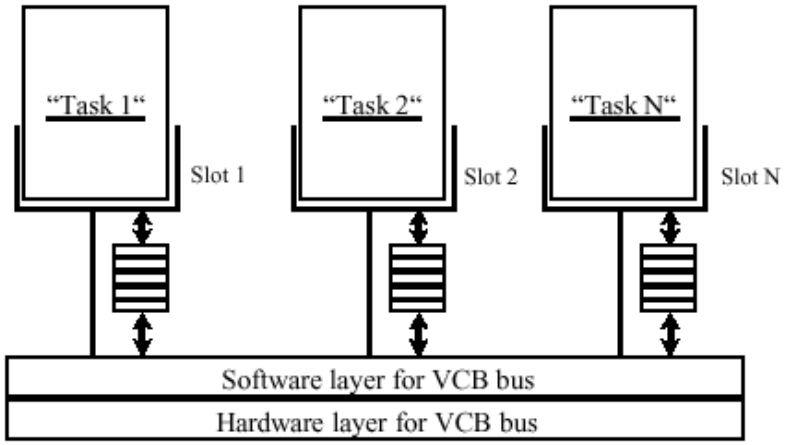


Figure 10: Logical picture of a VCB system

Interprocess communication

A *Virtual Communication Bus* (VCB) that uses the global memory on the system board is used for inter process communication and synchronization between tasks in the system [NYG00]. As the name says, the VCB is just a virtual bus that uses the physical CPCI bus and the global memory on the system board. The VCB provides a message passing mechanism that allows task-to-task communication locally on one CPU as well as between several different CPUs. The logical architecture of a system with a VCB bus is shown in figure 10.

The VCB bus is divided in two layers. The lower hardware layer consists of base primitives and is implemented and integrated in a FPGA (Field Programmable Gate Array). The upper layer is implemented in software and it provides different types of functionality from the bus. When a task wants to communicate on the VCB bus, it has to connect to the virtual bus. This is done by allocating one VCB-slot. When a task allocates a slot, it has to decide a message-sorting algorithm that will sort the message-queue. The two available sorting algorithms are *First In First Out* (FIFO) and a priority-based algorithm (highest priority first). The sender of a message has to set a priority to the message. VCB provides support for both synchronous and asynchronous communication. A task that is connected to a

VCB-slot can communicate with all the other tasks in the system. The functions this bus supports are for instance: *send*, *receive*, *broadcast*, *send and wait*, *multicast* and *subscribe*. It is the hardware layer that performs the job, when a call to the VCB is made. This feature speeds up the message passing compared to similar implementations in software [RTU00].

When a task allocates a VCB slot, a message-sorting algorithm must be chosen. It is a choice between two algorithms, namely FIFO and PRIORITY based sorting. If FIFO sorting is chosen, we will have an ordinary queue, the message received first will be read first. It is PRIORITY that is interesting in real-time applications. If the PRIORITY sorting algorithm is chosen the incoming message will be placed in the queue depending on its priority. If an incoming message has higher priority than the receiving task, the priority of the receiving task will be raised to the same priority as the message. The priority of the task will be restored to its origin when the message has been taken care of. By raising the priority of the receiving task when high priority messages arrives priority inversion is avoided.

Conclusions

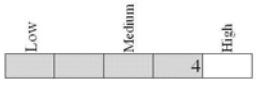
The SARA system together with the RTU provides an interesting solution that has proved to be efficient [FUR00] in comparison with some commercially available solutions. The speciality is of course the large amount of typical software routines that has moved into hardware. According to us the algorithms implemented in the RTU are of typical soft real-time class, but an interesting question arises when moving such algorithms into hardware. Is it possible that algorithms that usually cannot provide bounded response times etc, can be moved to hardware and then provide this? It seems like almost all the system calls have bounded worst-case response times; this together with the possibility of high "useful" CPU utilization could make the system suitable for hard real-time applications.

A potential problem with the SARA system today is the non-deterministic arbitration for the CPCI bus. Besides this problem, the fact that the bus is used for both system calls to the RTU and interprocess communication may cause new and undiscovered race conditions. Conflict between messages and system calls is not a problem in an ordinary software implemented RTOS.

Evaluation

Scheduling

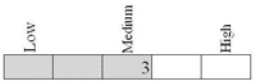
- *Determinism*



Static priorities suggest the use of for instance RM, and it is analyzable. The interrupts are also mapped into priority-based tasks. The execution of the scheduling algorithm does not require any CPU time either.

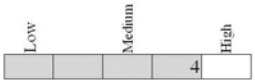
- *Inventiveness*

The algorithm itself is not fancy, but the hardware



implementation compensates for that.

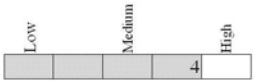
- *Usefulness*



Certainly, easy to map a real-world problem. This together with the possible speedup with a hardware implementation aims for a higher grade.

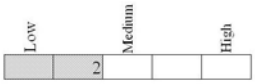
Memory management

- *Determinism*



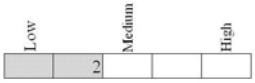
The SARA system does not provide any virtual or dynamic memory. Although access times to the shared memory on the system board may be varying, it is not supposed to be addressed as a part of the memory, instead through the message passing mechanism.

- *Inventiveness*



The memory management system on the SARA system does not provide any fancy solutions.

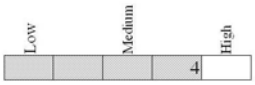
- *Usefulness*



No virtual memory or dynamic allocations are supported, in other words nothing that increases the usefulness.

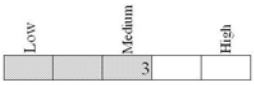
Interprocess communication

- *Determinism*



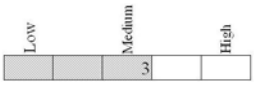
The priority based message queues are a nice real-time feature; together with a more deterministic hardware implementation and priority inversion handling a higher grade is motivated.

- *Inventiveness*



Nothing new, but not a copy of another solution either.

- *Usefulness*



Certainly useful, the speedup achieved with a hardware implementation is a desired feature. But a higher grade here would be achieved with more communication possibilities.

Spring

Spring [RAM91] is a distributed real-time operating system for large complex applications with hard timing constraints. According to the inventors, to ensure predictability, the design and implementation of all levels of the system must be integrated, i.e. a predictable architecture facilitates the construction of a predictable operating system, which is necessary to build predictable application software. Spring is a complete real-time system with among other things a partially customized hardware, a predictable operating system and a compiler.
The Spring system is physically distributed and consists of multiprocessor nodes connected through a network, where each multiprocessor contains one or more application processors (AP), one or more system processors and an I/O subsystem, as in the block diagram in figure 11.

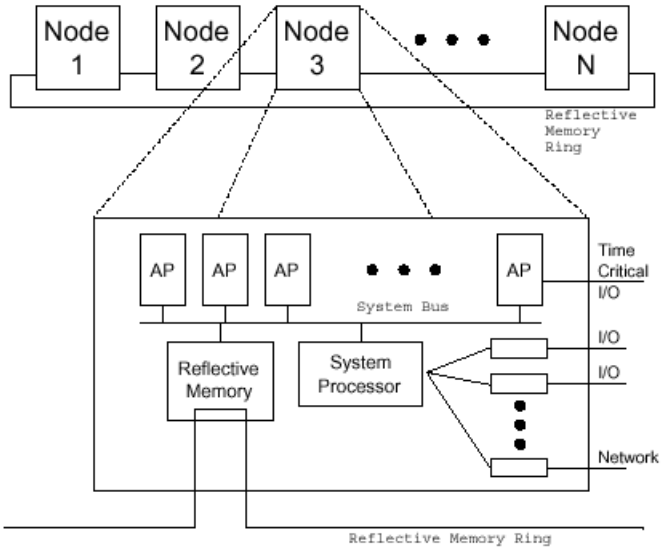


Figure 11, hierachical block diagram of a Spring system.

- Application processors execute previously guaranteed application tasks.
- The system processor executes the scheduling algorithm and other operating system tasks. The advantage with a physical separation between system activities and application activities is that the system overhead on the application processors is reduced. It also removes unpredictable delays since the application processors are not affected by external interrupts.
- The I/O subsystem handles non-critical I/O, slow I/O devices and fast sensors.

Each processing unit within a node consists of a commercial Motorola 68020-based MVME136A board [MOL90]. The MVME136A boards have the typical shared bus multiprocessors features, for instance an asynchronous bus interface

and a local memory. A part of this memory is used for storing programs and private data, and can only be accessed by the local processor. The rest of the memory can either be accessed by the local processor or remotely over the VME bus by another processor.

Scheduling

The scheduling is the most distinguishing feature of the Spring Kernel, and the mechanism is divided into four levels:

- At the lowest level, there is one type of dispatcher running on each application processor and another type running on the system processors. The application dispatchers simply removes next ready task from a system task table that contains previously guaranteed tasks arranged in the proper order for each application processor. The dispatcher on the system processor allows the periodic execution of systems tasks.
- The second level consists of a local scheduler that is resident on the system processor. It is responsible for dynamically guaranteeing the schedulability of a task set on an especial application processor. The scheduler produces a task table that is passed to the application processor.
- The third level is a distributed scheduler that tries to find an alternative node to execute any task that cannot be locally guaranteed.
- The fourth scheduling level is a metalevel controller that adapts various parameters of the scheduling algorithm to the different load conditions.

[BUT97]

Spring tasks are characterized by many different parameters. The user has to specify a worst-case execution time, a deadline, and interarrival time, a task-type (critical, essential or unessential), preemptive or non-preemptive, an importance level (value), a list of resources, a precedence graph and a list of nodes on which the task code will be loaded. The scheduling algorithm uses this information to find a feasible schedule.

A task is not defined as a large process, but rather a compiler generated non-preemptable piece of a process [NIE93, BUR93]. This is necessary because the compiler takes resource needs into account when creating small and predictable tasks from larger processes written by an application programmer.
A task holds a requested resource as long as it executes, i.e. each task acquires resources before it begins and releases the resources upon completion. The assignment of tasks to processors is initially done statically to avoid unpredictable delays and improve speed. A task can be loaded on more than one processor during runtime, so if an overload occurs, a task can be executed on another processor without a large overhead. Tasks with precedence constraints that share a single deadline are placed together in a *task group*.

Tasks are classified based on importance and timing requirements. The importance of a task is the same as the value gained by the system when the task

completes before its deadline. Timing requirements represent the real-time properties of a task, for instance periodic or aperiodic execution, hard or soft deadlines, while some tasks may not have any explicit timing requirements.
Based on importance and timing requirements, three different types of tasks are defined: critical tasks, essential tasks and unessential tasks.

- Critical tasks are those tasks that must complete their deadlines. A critical task that misses its deadline might occur a catastrophic result. The number of truly critical tasks is usually relatively small compared to the total number of tasks in the system.
- Essential tasks are those tasks that have timing constraints and are necessary to the operation of the system. A deadline miss does not cause catastrophic result, but the systems performance will be degraded. The number of essential tasks is normally large in complex control applications, and they must be treated dynamically because it is impossible to reserve enough resources for all of them [RAM91]. The Spring Kernel provides an on-line, dynamic algorithm for this type of tasks.
- Unessential tasks execute when they do not affect critical or essential tasks, and they may or may not have deadlines, i.e. they are executed in background. Maintenance functions and long-range planning tasks belong to this category.

The Spring scheduling algorithm [NIE93, BUR93] dynamically guarantees the execution of newly arrived tasks depending on the current load. The feasibility is determined based on many issues, such as timing constraints, mutual exclusions on shared resources, precedence relations, preemption properties and fault-tolerant requirements. The algorithm uses a heuristic function to reduce the search space and find a result in a polynomial time, since the problem is NP-hard. The *heuristic function* H is applied to each of the tasks that are waiting for to be scheduled. The task with the smallest value is selected to extend the current schedule. It is easy to modify the heuristic function. The value that the function determines can for instance be arrival time (the algorithm will work as *first come first served*), absolute deadline (the algorithm will work as *earliest deadline first*) and computation time (the algorithm will work as *shortest job first*).

Each task has to declare a binary array of resources $R_i = [R_1(i),\ldots,R_r(i)]$. If a specific resource is not used by a task, the tasks element in the binary array is set to zero $R_k(\mathrm{i}) = 0$, and the element is set to one if it is used. For each resource, the algorithm determines the earliest time the resource is available, denoted as EAT_k (Earliest Available Time). The earliest start time $T_{est}(i)$ that a task τ_i can start the execution without blocking any shared resources is:

$T_{est}(i) = \max[a_i, \max(EAT_k)]$,

Where a_i is the arrival time of τ_i. Once T_{est} as been calculated for all of the tasks, the heuristic function selects the task with smallest value of T_{est}. The complexity of the Spring scheduling algorithm is $O(n^2)$, where n denotes number nodes.

Precedence relations can be handled by a factor called eligibility. A task becomes eligible for execution when all tasks in front of the current task in the precedence graph have completed. A task that is not eligible cannot be selected for extending a partial schedule.

Memory management

Figure 11 shows that each node is a distributed memory multiprocessor with non-uniform (NUMA) memory access times. Local access is significantly faster, since it does not use the system bus. Each node has a 2 MB reflective memory that is used to support predictable IPC.
Traditional memory management with support for virtual memory introduces several sources of unpredictability. Each memory reference is subject to three possible delays: page fault, TLB loading and TLB translation [NAH92]. Each of these delays makes it harder to calculate worst-case execution times that are not to pessimistic. Therefore the Spring memory management unit (MMU) has two following basic ideas:

1. Avoid page faults by preallocating, at process creation time, a physical page for every used page in a program's address space and loading that page in memory.
1. Explicitly manage the contents of the translation look-aside buffer (TLB) to ensure that all memory references experience TLB hits.

The first idea eliminates unpredictability due to page faults, and the second eliminates unpredictability due to TLB misses. Since every memory reference results in a TLB hit, the MMU influence on memory references time is predictable. [NAH92] describes how the memory management unit is implemented.

Spring has memory-management primitives that create various resource segments that are completely memory-resident, for instance code, stacks, task control blocks, task descriptors, local data, global data and ports. The kernel allocates all the required segments when a task begins execution.

Interprocess communication

Due to hard real-time requirements, all communication must be predictable. This include bounding of execution times of IPC primitives, network protocol processing and message propagation delay. The Spring IPC system supports both synchronous and asynchronous message passing through ports, which are kernel-protected memory objects. Processes can communicate by placing messages into ports and removing messages from ports. The ports are typed by the kind of process that uses them (guaranteed hard real-time, soft real-time or non real-time)

and according to the communication they are used for, namely synchronous and asynchronous. Ports have bounded capacity and the messages have fixed sizes. Messages can have deadlines, which determine when they must be delivered to a port.

Interrupt handling

Peripheral I/O devices are divided into two classes: fast and slow I/O devices. The system processor handles fast I/O devices, so the application processors are not affected. Interrupts from fast I/O devices are treated as a new task that is subject to the guarantee routine like any other task in the system. Slow I/O devices are multiplexed through a front-end dedicated processor (I/O processor) controlled by a commercial operating system. The guarantee algorithm does not affect device drivers running on the I/O processor, although they can activate critical or essential tasks.

Conclusions

Predictability is a keyword that come back again and over again when talking about the Spring system's design issues. It is obvious that a hard real-time system with hard timing constraints must be predictable to manage the application's timing requirements. In order to achieve predictability, all levels of the system must be predictable, i.e. a predictable architecture facilitates the construction of a predictable operating system, which is necessary to build predictable application software.

A distinguishing feature with Spring in comparison with other multiprocessor real-time systems is the system processor, which only executes scheduling and system tasks. The application tasks execute on application processors. The advantage with a physical separation between system activities and application activities is that the system overhead on the application processors is reduced. It also removes unpredictable delays since the application processors are not affected by external interrupts. It is certainly a powerful feature to achieve predictability in a real-time system. Another solution to achieve predictability is to eliminate some or all interrupts, instead of separating system activities from application activities.

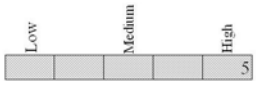
Spring is a distributed real-time system for large complex applications with hard timing constraints, but it does not provide any fault tolerance.

- What will happen if a critical task misses its deadline?
- The task migration may also be a problem. What will happen with globally rejected tasks, i.e. tasks that cannot be guaranteed?
- What if acknowledgements get lost, i.e. when a task is migrated to another node? Inconsistent state?

Evaluation

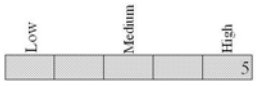
Scheduling

- *Determinism*



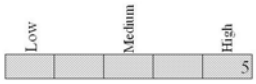
All levels of the system have been designed to ensure predictability. Offline analysis is possible for tasks with hard deadlines. Tasks that have timing constraints but are not necessary to the operation of the system are treated dynamically, with an online guarantee algorithm. Combined with a hardware implementation, this is a deterministic solution.

- *Inventiveness*



Spring supports both multilevel scheduling and task migration. Again this together with a hardware implementation aim for the highest grade.

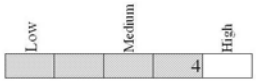
- *Usefulness*



Spring has all desired task types, and a very high flexibility. The flexibility is achieved through the possibility of different scheduling algorithms.

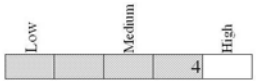
Memory management

- *Determinism*



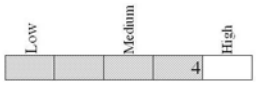
The Spring memory management unit (MMU) eliminates unpredictability due to page faults and TLB misses. So every memory reference results in a TLB hit, the MMU influence on memory references time is predictable.

- *Inventiveness*



Spring supports virtual memory and dynamic memory allocation.

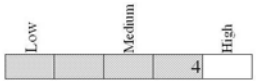
- *Usefulness*



Both virtual memory and dynamic memory allocation make Spring flexible.

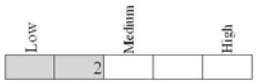
Interprocess communication

- *Determinism*

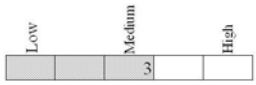


Spring have bounded execution times of IPC primitives, network protocol processing and message propagation delay. The IPC system provides a message passing mechanism that uses ports, which are kernel-protected memory objects. The ports can be typed by the kind of communication they are used for, i.e. guaranteed hard real-time, soft real-time or non real-time.

- *Inventiveness* No extraordinary features are provided.



- *Usefulness* The system processor handles fast I/O devices, so the application processors are not affected.



Summary of evaluations

In table 7, a summary containing all the evaluation parts of all reviewed RTOS:es is presented. As before each of the areas scheduling, memory management and interprocess communication are graded within the three topics (D)eterminism, (I)nventiveness and (U)sefulness. A lot of grades give the reader the opportunity to self create ranking orders focusing on the issue that is the most interesting for his/hers own purposes.

	Scheduling			Memory management			Interprocess communication		
	D	I	U	D	I	U	D	I	U
CHAOS[arc]	2	5	2	2	4	4	3	5	2
Chimera	3	4	4	2	4	3	3	3	4
MARS	5	3	2	5	2	2	5	4	1
MontaVista Linux	2	2	3	1	1	5	1	1	5
OSE	3	2	4	2	4	4	3	3	3
RT-Mach	4	5	5	3	2	4	4	3	4
RTEMS	4	2	4	3	3	3	2	2	4
SARA	4	3	4	4	2	2	4	3	3
Spring	5	5	5	4	4	4	4	2	3

Table 7, Summary of all evaluation parts.

Conclusions and Future Work

When having reviewed a considerable amount of material related to multiprocessor RTOS, it is tempting to predict a "silver-bullet" solution that is the best solution for all real-time multiprocessor systems. Unfortunately, as with many other issues about computer science, an RTOS that always performs best, when used with different hardware platforms and in different purposes, does not exist. I.e. a RTOS must be designed (both software and hardware) considering the real-world requirements. For instance, safety critical real-time systems have to be predictable so that we are able to guarantee all timing constraints. While some other real-time systems are allowed to have deadline-misses. These real-time applications are often referred to, as applications with soft real-time demands. In these systems many ordinary operating system constructs optimized for throughput and processor utilization can be suitable. This is obvious since we always want to use the processor to its maximum, but algorithms developed for hard real-time systems often limits the utilization, because the focus is on completing all temporal constraints whatever it costs.

When designing an RTOS, there are many trade-offs that have to be considered. Concentrating the focus on real-time, one important trade-off is determinism and flexibility/high average performance. But we cannot say that one solution is better than the other, it is application dependent. In this survey, MARS is the most deterministic operating system, and therefore most suitable for safety critical applications. MARS is completely offline scheduled, even the communication is offline scheduled. This feature makes MARS totally predictable and deterministic, but it is not flexible at all, since the applications behavior must be known before runtime, in order to schedule execution and communication. The RTOS:es found among the most flexible are more suitable for applications with soft demands. The application can be faster developed, we do not have to analyze the whole behavior before runtime and we can use more advanced system calls. We can also potentially achieve higher processor utilization with more released and throughput oriented algorithms for scheduling, communication and memory management.

Concerning multiprocessor support in RTOS, it is possible to distinguish between RTOS:es constructed for multiprocessor platforms and RTOS:es constructed for single processor machines, with lately added multiprocessor support. As with all attempts to generalize exceptions exists, but the scheduling algorithms and IPC routines can be treated as some kind of indication. A RTOS originally constructed for multiprocessor platforms, should through the scheduling algorithm take advantage of the possibilities with multiprocessor platforms, or all the IPC routines should be constructed for communication between different nodes. While many of the RTOS:es originally aimed for single processor machines does not bother about the number of processors in the scheduling procedures, and have different IPC routines for tasks that resides on the same node and tasks communicating across node boundaries.

As future work, an operating system will be designed and implemented, with the survey as a preparatory study. The operating system will be implemented in software with the same interface as the hardware implemented RTU, and operate on the SARA system. Both described in the case studies section. A fair benchmark between the two operating systems should be easy to achieve, since the same interface is used.

Acknowledgements

Thanks goes to Peter Nygren and Johan Stärner, both currently PhD students at the department of computer science and engineering at Mälardalen University, for their supportive work as supervisors and fruitful comments on how to improve the survey.

References

[ACC86] M. Accetta et al., Mach: A new kernel foundation for unix development, In Proceedings of the Usenix Summer Conference, July, 1986.

[ACH91] S, Acharya et al., Overview of Real-Time Kernels at the Superconducting Super Collider Laboratory, Particle Accelerator Conference, 1991.

[ADO96] J. Adomat et al., Real-Time Kernel in Hardware RTU: A step towards deterministic and high performance real-time systems, In 8th Euromicro Workshop on Real-Time Systems, June 1996.

[ALT98] P. Altenbernd and H. Hansson, The Slack Method: A New Method for Static Allocation of Hard Real-Time Tasks. Kluwer Journal on Real-Time Systems, 1998.

[ARA93] H. Arakawa et al., Modeling and Validation of the Real-Time Mach Scheduler, Proceedings of the ACM SIGMETRICS conference on Measurement and modeling of computer systems, June 1993.

[BAR92] L. Barroca and J. McDermid, Formal Methods: Use and Relevance for Development of Safety-Critical Systems, The Computer Journal, 1992.

[BEN01] M. Bennet and N Audsley, Predictable and Efficient Virtual Addressing for Safety-Critical Real-Time Systems, 2001.

[BUR93] W. Burleson et al., The Spring Scheduling Co-Processor: A Scheduling Accelerator, In Proceedings of IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1993.

[BUT93] G. Butazzo, RED: A Robust Earliest deadline scheduling algorithm, Proc. of 3rd International Workshop on Responsive Computing Systems, 1993.

[BUT94] G. Butazzo and M. Spuri, Efficient aperiodic service under earliest deadline scheduling, In proc. 15th Real-Time Systems Synposium, Dec. 1994.

[BUT97] G. Buttazzo, Hard Real-Time Computing Systems – Predictable Scheduling Algorithms and Applications, Kluwer Academic Publishers, ISBN 0-7923-9994-3, 1997.

[BWE01]B. Weinberg, Open Availability Architecture: Building Highly Available Systems with Linux and CompactPCI, MontaVista Software White Paper, 2001.

[CAL98] J. Calvez, Performance Monitoring and Assessment of Embedded Hw/Sw Systems, In Journal “Design Automation for Embedded Systems”, Kluwer Academic Publishers, 1998.

[CHI02] Chimera homepage: www-2.cs.cmu.edu/~aml/chimera/chimera.html

[CRO85] Crowther W et al., The Butterfly parallel processor, IEEE Computer Architecture Technical Committee Newsl., December., 1985.

[CUR76] H. J. Curnow and B.A. Wichmann, A synthetic benchmark, Computer Journal, January 1976.

[DAM89] A. Damm et al., The Real-Time Operating System of MARS, Operating Systems Review, July 1989.
[DEZ74] M. Dertouzos, Control robotics: The procedural control of physical process, in proceedings of IFIP congress, 1974.
[ENE1] Enea OSE Systems, User's Guide / R1.0.0 (Chapter 2 OSE Concepts).
[ENE2] Enea OSE Systems, The Direct Message Passing RTOS – A Better Way!, White Paper.
[ENE3] Enea OSE Systems, Safety Critical Applications.
[FLE89] B. Fleisch and G. Popek, Mirage: A Coherent Distributed Shared Memory Design, 1989.
[FOH94] G. Fohler, Flexibility in Statically Scheduled Hard Real-Time Systems, Dissertation Technische Universität Wien, April 1994.
[FOH95] G. Fohler, Joint Scheduling of Distributed Complex Periodic and Hard Aperiodic Tasks in Statically Scheduled Systems, In Proceedings of the IEEE Real-Time Systems Symposium, December 1995.
[FOH97] G. Fohler and K. Ramamritham, Static scheduling of pipelined periodic tasks in distributed real-time systems, In Proceedings of 9th Euromicro Workshop on Real Time Systems, 1997.
[FUR95] J. Furunäs, RTU94 – Real Time Unit 1994, Bachelor Thesis, department of Computer Engineering, University of Mälardalen, 1995.
[FUR99] J. Furunäs et al., Flexible Multiprocessor computer Systems, In CAD & Computer Graphics, December 1999.
[FUR00] J. Furunäs, Benchmarking of a Real-Time System that utilises a booster, International Conference on Parallel and Distributed Processing Techniques and Application, 2000.
[FUR01] J. Furunäs, Interprocess Communication Utilising Special Purpose Hardware, Licentiate thesis, Mälardalen University Press and Department of Information Technology, Uppsala University, December 2001.
[GBU93] G. Buttazzo, Hartik: A real-time kernel for robotics applications, In proc. IEEE Real-Time Systems Symposium, December 1993.
[GHE93] A. Gheith and K. Schwan, CHAOS^{arc}: Kernel Support for Multiweight Objects, Invocations, and Atomicity in Real-Time Multiprocessor Applications, ACM Transactions on Computer Engineering, February 1993.
[GOP93] P. Gopinath et al., A Survey of Multiprocessor Operating System Kernels (DRAFT), Technical report, Georgia Institute of Technology, College of Computing, November 1993.
[HOF89] R. Hoffman et al., CHIMERA: A Real-Time Programming Environment, for Manipulator Control, In Proceedings of IEEE International Conference on Robotics and Automation, May 1989.
[JXU93] J. Xu, Multiprocessor Scheduling of Processes with release Times, Deadlines, Precedence, and Exclusion Relations, IEEE Transaction on software engineering, February 1993.
[KAL] D. Kalinsky, Direct Message Passing for High Availability Software Design, OSE Systems, Inc.
[KAM91] N. 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.
[KAM96] N. Kamenoff, One Approach for Generalization of Real-Time Distributed Systems Benchmarking, Proceedings of the 4th WPDRTS, 1996.
[KAR89] R. Kar and K. Porter, Rhealstone - a Real-Time Benchmarking Proposal, Dr. Dobbs' Journal, February 1989.
[KAR90] R. Kar, Implementing the Rhealstone Real-Time Benchmark, Dr. Dobbs' Journal, April 1990.

[KHO92] P. Khosla et al., The Chimera II Real-Time Operating System for Advanced Sensor-Based Control Applications, IEEE Transactions on Systems, Man and Cybernetics, 1992.
[KOP85] H. Kopetz and W. Merker, The Architechture Of MARS, Proceedings of 15th Fault-tolerant Computing Symposium, June 1985.
[KOP87] H. Kopetz and W. Oshenreiter, Clock Synchronization in Distributed Real-Time Systems, IEEE Transactions on Computer Engineering, August 1987.
[KOP89] H. Kopetz et al., Distributed Fault-Tolerant real-Time Systems: The MARS Approach, IEEE micro, February 1989.
[LAY73] J. Layland and C Liu, Scheduling algorithms for multiprogramming in hard real-time environments, Journal of the ACM, January 1973.
[LEB92) T. LeBlanc and E. Markatos, Shared Memory vs. Message Passing in Shared Memory Multiprocessors, 1992.
[LEE88] I. Lee et al., Rk: A real-time kernel for a distributed system with predictable response, Dept. of Computer Science, Univesity of Pennsylvania, October 1988.
[LEH90] J. Lehoczky et al., Priority inheritance protocols: An approach to real-time synchronization, IEEE Transactions on Computers, September 1990.
[LEH93] J. Lehoczky et al., On line scheduling for Hard-Real-Time Systems, The Journal of Real-Time Systems 1, 1993.
[MEN02] Mentor Graphics homepage, http://www.mentor.com.
[MOK83] A. MOK, Fundamental design problem of distributed systems for the hard real-time environment, Dissertation Camebridge, May 1983.
[MOL90] L. Molesky et al., Implementing a Predictable Real-Time Multiprocessor Kernel – The Spring Kernel, May 1990.
[NAH92] E. Nahum et al., Architecture and OS Support for Predictable Real-Time Systems, March 1992.
[NAK90] T. Nakajima et al., Real-Time Mach: Towards a Predictable Real-Time System, In Proceedings of USENIX 1st Mach Workshop, October 1990.
[NAK93] T. Nakajima et al., RT-IPC: An IPC Extension for Real-Time Mach, In Proceedings of the 2nd Microkernel and Other kernel Architectures, USENIX, 1993
[NIE93] D. Niehaus et al., The Spring Scheduling Co-Processor: Design, Use, and Performance, In proceedings of Real-Time Systems Symposium, 1993.
[NOR00] Christer Norström et al., Robusta realtidssystem, version: rts-bok00-11, August 2000.
[NYG00] 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.
[OAR00] On-Line Applications Research Corporation (OAR), RTEMS C User's Guide, Edition 1 for RTEMS 4.5.0, 2000.
[QNX02] QNX homepage, http://www.qnx.com.
[RAM90] K. Ramamritham, Allocation and scheduling of complex periodic tasks, in proceedings of the 10th International Conference on Distributed Computing Systems, 1990.
[RAM91] K. Ramamritham and J. A. Stankovic, The Spring Kernel: A new paradigm for Real-Time Systems, IEEE Software, May 1991.
[RED02] Red Hat homepage, http://www.redhat.com.
[RTU00] 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.
[SCH87] K. Schwan et al., Chaos – kernel support for objects in the real-time domain, IEEE Transactions on Computer Engineering, July 1987.

[SCH90] K. Schwan et al., From CHAOS^{min} to CHAOS^{arc}: A family of Real-time Kernels, In Proceedings of the IEEE Real-Time Systems Symposium, December 1990.

[SCH89] K. Schwan and H. Zhou, Optimum preemptive scheduling for hard real-time system: Toward real-time threads, Technical report, GIT, July 1989.

[SHO01] M. El Shobaki and Lennart Lindh, A Hardware and Software Monitor for High-Level System-on-Chip Verification, Proceedings of the IEEE International Symposium on Quality Electronic Design (IEEE), San Jose, CA, USA 2001.

[SRI00] W. Srisa-an et al., Active Memory: Garbage-Collected Memory for Embedded Systems, the second Annual Workshop on Hardware Support for Objects and Microarchitechtures for Java, September 2000.

[STA00] J. Stankovic, A Special issue on: Ubiquitous Computing, volume 1, issue 4, November 2000.

[STA98] J. Stankovic et al., Deadline scheduling for real-time systems: EDF and related algorithms, Kluwer international series in engineering and computer science, ISBN 0-7923-8269-2, 1998.

[TAN92] A. Tanenbaum, Modern Operating Systems, Prentice Hall, Inc, ISBN 0-13-595752-4, 1992.

[THA00] H. Thane, Monitoring, Testing and Debugging of Distributed Real-Time Systems, Doctoral Thesis, KTH, Stockholm, Sweden, May 2000.

[THA01] H. Thane et al., The Asterix Real-Time Kernel, 13th Euromicro International Conference on Real-Time Systems, 2001.

[TIN94]K. Tindell and J. Clark, Holistic Schedulability Analysis for Distributed Real-Time Systems, Real-Time Systems Journal 9, 1995.

[WEI84] R. P. Weicker, Dhrystone: A Synthetic Systems Programming Benchmark, Communications of the ACM, 1984.

[WEI90] N. H. Weiderman, Hartstone: Synthetic Benchmark Requirements for Hard Real-Time Applications, Proceedings of the working group on Ada performance issues, 1990.

[WEI01] B. Weinberg and C. Lundholm, Embedded Linux – Ready for Real-Time, MontaVista Software White Paper, 2001.

[WIN02] WindRiver homepage, http://www.windriver.com.

[YAN97] L. Yanbing et al., Real-time operating systems for embedded computing**,** In proc. Of IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1997.

[YOD97] Victor Yodaiken, The RT-Linux Approach to Real-Time, A short position paper, 1997.

Introduction and Benchmarking of Competitive Real-Time Multiprocessor Kernels

Mikael Åkerholm and Tobias Samuelsson,
Masters' thesis project

ABSTRACT

As the demands on real-time applications increase and they become more complex every year, the demands of real-time platforms also grow larger every year. Today both good performance and correct timing are to be desired in a real-time system. The motivations for using multiprocessor systems are scalability, robustness and performance. No single processor solution is able to provide more computing power than a multiprocessor system built of the same processors. The operating system is without hesitation the most important software of all system programs in a real-time multiprocessor system. A real-time system must be predictable in order to offer correct timing. One way to achieve both performance and determinism is to implement the kernel in hardware.

In this paper we introduce MPSWOS, which is a multiprocessor operating system kernel implemented in software. With the purpose to find out the differences between kernels implemented in hardware and kernels implemented in software, we compare MPSWOS with an existing kernel in hardware. The kernels operatates on the same hardware architecture and have the same application interface, so the only difference is the kernel implementation. This paper mainly contain a practical comparison between the different kernels, in the form of a benchmark.

The Department of Computer Science and Engineering,
Mälardalen University, June 2002.

Table of contents

Introduction

This paper is the second part of our masters' thesis project presented to the department of computer science and engineering at Mälardalen University. The first part of this thesis evaluates and compares different selected multiprocessor real-time operating system kernels as a prepearatory study. The preparatory study for this thesis has been used as base for this the second part, which assignment was to design and develop a real-time operating system in software with the same application interface (API) as the RTU [RTU00] and then benchmark and compare it with the existing SARA system [FUR99] equipped with a RTU. The RTU is a RTOS co-processor, developed at Mälardalen University.

As real-time applications become larger and therefore more complex every year, the demands on performance and timing correctness increase. The fact that no single processor solution is able to provide more computing power than a multiprocessor system built of the same processors is one motivation to use multiprocessors. The performance of multiple processors may be limited by the hardware architecture. For instance, in a shared bus multiprocessor system, accessing the bus may limit the performance of the system. One solution to obtain more performance is to decrease the administration of the system e.g. scheduling, clock-tick management and so on, by utilizing special purpose hardware. The RTU is an example of such solution. Not only the performance is increased, by utilizing parallel hardware the determinism is also better.

The new software OS introduced in this paper utilizes the same hardware architecture and has the same application interface (API) as the existing SARA system with RTU. The name of the new software OS is MPSWOS, which is a shortening for Multiprocessor SoftWare Operating System. The motivation to this paper is to show the differences between utilizing a co-processor and utilizing a standard processor.

A similar comparison has been made in [FUR00], but the differences are the software operating system and the benchmark programs. In [RIZ01], a comparison between a software kernel and a hardware kernel on a single processor system has been done.

The outline of this paper is as follows. First we introduce the design of MPSWOS and the motivations behind the selection of the design. In that section we will also explain the differences between the RTU and MPSWOS. The next section is benchmark, which contains a practical comparison between the different kernels. Different benchmarks have been performed, in order to achieve values for communication bandwidth, communication latency, OS-overhead and so on. Both the results and the motivation to them are presented. The final section concludes the paper with conclusions and future work suggestions.

Design of MPSWOS

The hardware architecture MPSWOS run on is a SARA system without RTU. The SARA system is divided into local CPU board, bus arbitrator, global RAM and I/O. The processor 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 CPU-board (system board) that handles the arbitration, clock-distribution, etc on the back plane. An overview of the SARA-system that MPSWOS runs on is shown in figure 1.

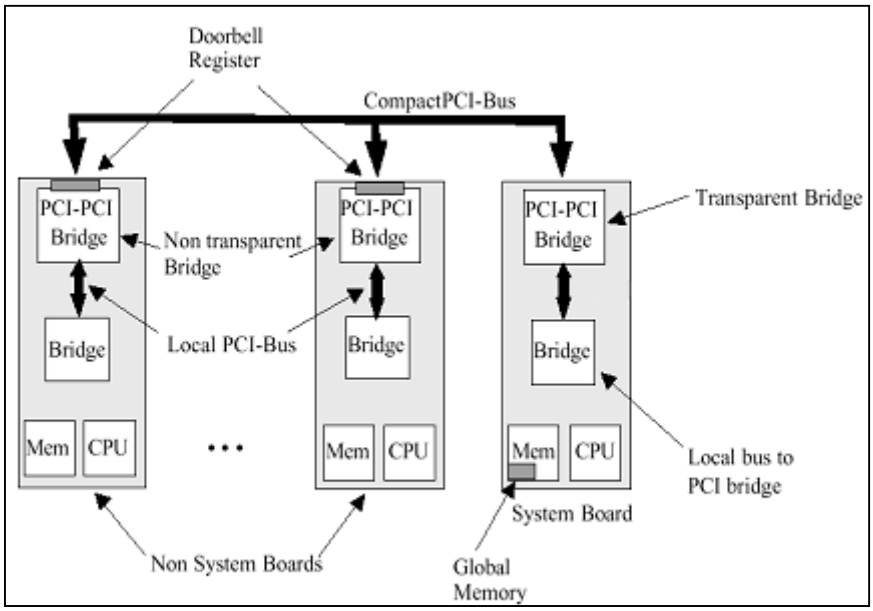


Figure 1, block diagram of the SARA system (without RTU)

Scheduling

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 SARA system equipped with an RTU, MPSWOS consists of several schedulers. The system board has a complete scheduler while the slave boards have simpler schedulers. The idea with several different schedulers can for instance be found in the Spring system [RAM91]. In general it is possible to compare the non-system boards (slaves) in the SARA system with the application processors in Spring. The main difference is that the scheduler on the slave boards

in the MPSWOS system has some more functionality than the dispatchers in the Spring system. Since the system board has a different scheduler than the slave boards, they do have different versions of the operating system.

The different operating systems are communicating by sending interrupts to each other. Parameters are passed through the global memory. Each CPU board has one parameter area in the global memory to use when sending interrupt to the system board. This implies that some kind of protection is needed; i.e. only one task per board is allowed to use the parameter area at the same time. The lock we have chosen is a load-locked/store-conditional (LL-SC) lock provided by the hardware. The first instruction (load-locked) loads a synchronization variable into a register. It may be followed by arbitrary instructions that modify the value in the register. The last instruction (store-conditional) tries to write the register back to memory location, if and if only no other process has written to that location. In other words, these instructions allow us to implement a range of atomic read-modify-write operations. The task placement is offline scheduled and the tasks can only execute on one node. For instance when a task is placed on the system board, it can only execute on the system board.
The task management and scheduling algorithm are the same as in the original SARA system, i.e. pre-emptive priority-driven scheduler, which guarantees that the task with highest priority is executing (at any instance of time).

Each slave board has a local ready queue and blocked queue, while the system board has semaphore-blocked queues, a waiting queue, a local ready queue and a blocked queue, see figure 2. In order to avoid clock-synchronization, the master node (system board) handles all timing, i.e. the master node is the only node that has clock interrupts. The waiting queue contains both master and slave tasks. On each clock interrupt, the master checks the waiting queue, i.e. is there any task that becomes ready to execute? If a slave-task becomes ready, an interrupt to the slave is sent. When a slave-task makes a delay, an interrupt to the master is sent and the first task in the ready queue is dispatched.

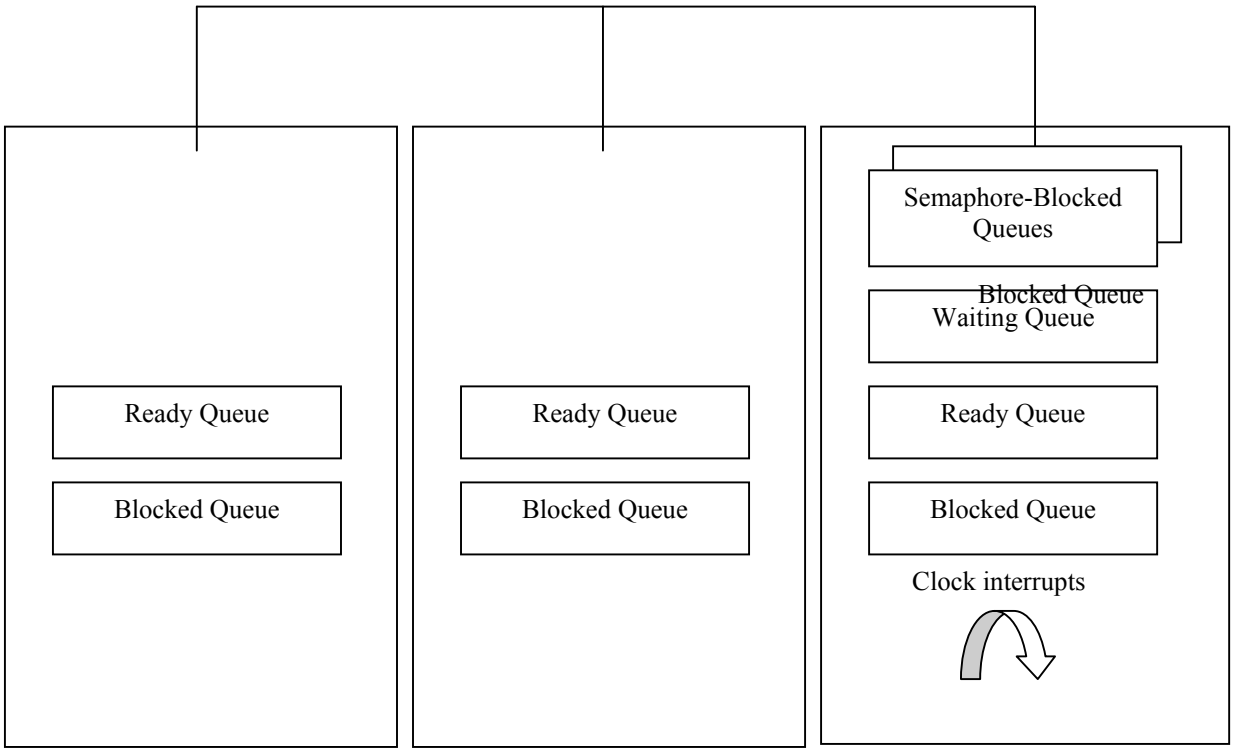


Figure 2, Placement of Scheduling queues

The state transition diagram for tasks are shown in figure 3. The different states are *ready, executing, waiting, blocked and semaphore-blocked*. The *OS* text in figure 3 means that this event is caused by the operating system.

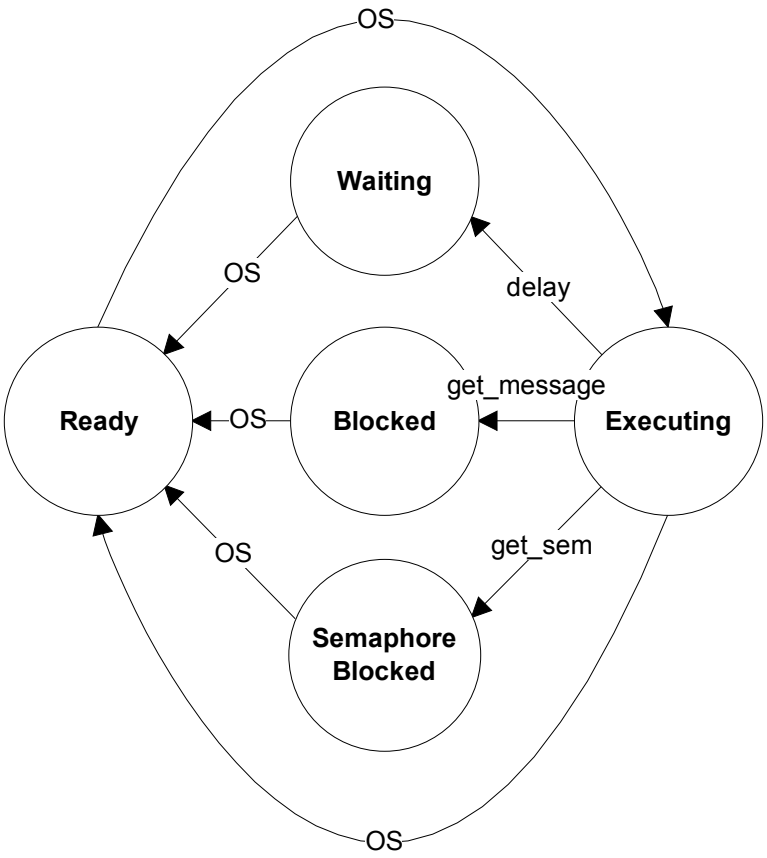


Figure 3, state transition diagram for tasks

Memory management

Communication and synchronization between different processes in the system is performed through the global memory that resides on the system board. Global memory areas can be defined on any CPU board. As shown in figure 1, 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. The system board has a transparent bridge, while the other boards have bridges that remap addresses on one bus to another address on the other bus. With this non-transparent bridges, address collisions on the CPCI bus can be avoided and all boards can use its full address range on the local PCI bus.

MPSWOS has the same memory settings as the original SARA system, i.e. global memory is residing on the system board. With this solution, all communication between tasks goes through the system board, even if two tasks that are communicating reside on slave boards. We have an idea to relieve the pressure on the system board. But due to lack of time, we have implemented the simpler

solution that has the same memory settings as the original SARA system. The idea was to define global memory areas on each CPU board. We wanted 16 MB physically on each node, starting at adress 0x1000000 and ending at 0x1FFFFFF. That will result in a 48 MB continous memory, when using three CPU boards. Since the slave boards have non-transparent bridges, they are able to translate an address on one bus to another address on another bus. By setting the non-transparent bridges, this solution is feasible. The idea is shortly described in the *Future Work* section.

Interprocess communication

A Virtual Communication Bus (VCB) that uses the global memory on the system board is used for inter process communication and synchronization between tasks in the system [NYG00]. As the name says, the VCB is just a virtual bus that uses the physical CPCI bus and the global memory on the system board. The VCB provides a message passing mechanism that allows task-to-task communication locally on one CPU as well as between several different CPUs. The logical architecture of a system with a VCB bus is shown in figure 4. The VCB bus is divided in two layers. The lower hardware layer consists of base primitives and is implemented and integrated in a FPGA (Field Programmable Gate Array). The upper layer is implemented in software and it provides different types of functionality from the bus. When a task wants to communicate on the VCB bus, it has to connect to the virtual bus. This is done by allocating a VCB-slot. The sender of a message has to set a priority to the message. VCB provides support for both synchronous and asynchronous communication. A task that is connected to a VCB-slot can communicate with all the other tasks in the system. The functions this bus supports are for instance: send, receive, broadcast, send and wait, multicast and subscribe. It is the hardware layer that performs the job, when a call to the VCB is made. This feature speeds up the message passing compared to similar implementations in software [RTU00].

In MPSWOS, the lower hardware layer has been replaced by a software layer. This means that the VCB interface to the user has not been changed, see figure X. Only the most important methods have been implemented, such as send, receive and so on.

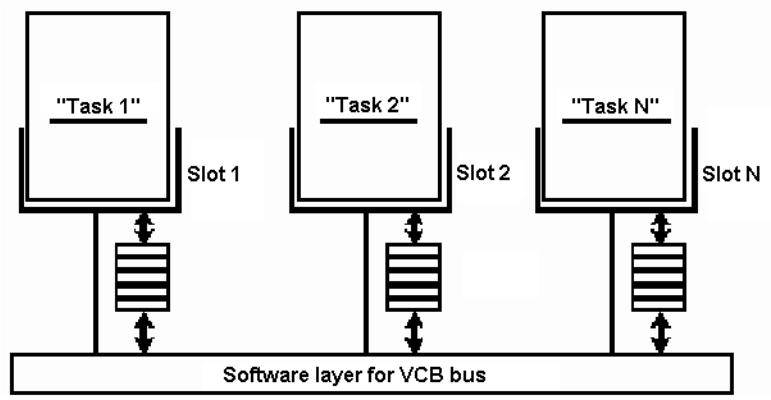


Figure 4, the VCB communication model

The message-queues are placed in the global memory on the system board. Each slot has a message-queue. To protect the message-queues, i.e. guarantee mutual exclusion, a semaphore-protocol has been implemented. The system board handles the semaphores. Each message-queue (slot) has a semaphore, and each semaphore has a semaphore blocked-queue residing in the global memory. A task that tries to take a semaphore that is occupied, is placed in the semaphore blocked-queue. The semaphore blocked-queues are sorted by priority.

Clockmanagement

Since the system board handles the waiting-queue, the non-system boards need no clock interrupts, i.e. no clock-synchronization or distribution is needed. Clock-tick interrupts is taken when decrementer exception occurs, i.e. when the decrementer register is equal to zero and no exception with higher priority exists. The decrementer register is decremented at 16.75MHz, one fourth of the bus clock rate. When a decrementer exception is taken, instruction fetching resumes at offset 0x0000_0900, from the base 0x0000_0000 or 0xFFFF_0000 configurable in the processor machine state-register.

The system board has a time base facility (TB), which is a 64-bit structure that consists of two 32-bit registers, time base upper (TBU) and time base lower (TBL).

Benchmarking

The benchmark presented here is aimed to serve as comparison between a software and a hardware operating system. The software operating system is the MPSWOS described earlier in this paper; the hardware operating system is the SARAOS or the RTU [RTU00]. Both operating systems have the same programming interface, although the software operating system has a slightly reduced version.

The benchmark series is built with own ideas and parts from several well-known benchmarks: Rhealstone [KAR89], SSU [ACH91] and Distributed Hartstone [KAM91]. It was the most suitable solution to create an own benchmark series that gives the freedom of measuring what we feel are important, if using another benchmark we would have to port the benchmark to our operating system and that would require almost the same amount of work. The different test cases that are used are first explained both the idea and the implementation, at the end of each test case the results are presented.

All tests have been executed five times in order to compute the average result. Worth to notice is that all times expressed in the results is expressed in the one fourth of the processor bus clock speed. To transform the times to seconds one should multiply the time with 1/16,7 MHz, since the processor bus clock speed is 67 MHz on all nodes in the system.

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. All tasks in both operating systems are created at the system start-up, so this test is classified as non real-time and it is not critical to the system. The reason to this measurement is the comparison between the hardware and software implementation that is desirable. Conditions that may affect the task creation time are the number of already created tasks and where in the system the task is located.

Method

The practical part of this benchmark is pretty straightforward, just measure the time for the system call representing create task. 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.

Result

The results are presented with the two graphs below. As we can see the software operating system is faster on both the master node in figure 5, and the slave node in figure 6. That is an expected result, since the tasks are created locally and we do not need any communication with PCI devices. The hardware operating system

on the other hand is a PCI device with enclosed latencies. Also as predicted the software operating system has longer latencies with increasing number of already created tasks. That is because of the list management latencies that increase with increasing number of tasks. The cache memory effects are also visible in the test, since the first task create call take more time than the second.

It is confusing that both the hardware and the software operating system is faster on the slave node than the master node, which is a result we cannot explain. The hardware operating system should be slower on the slave node, since it is physically longer distance between the slave processor and the operating system than between the master node and the operating system. We have verified the result with repeated measurements and we have also used the MAMON [SHO01] monitoring device, which indicates the same result. Appendix 1 contains tables with listings of the results from this benchmark.

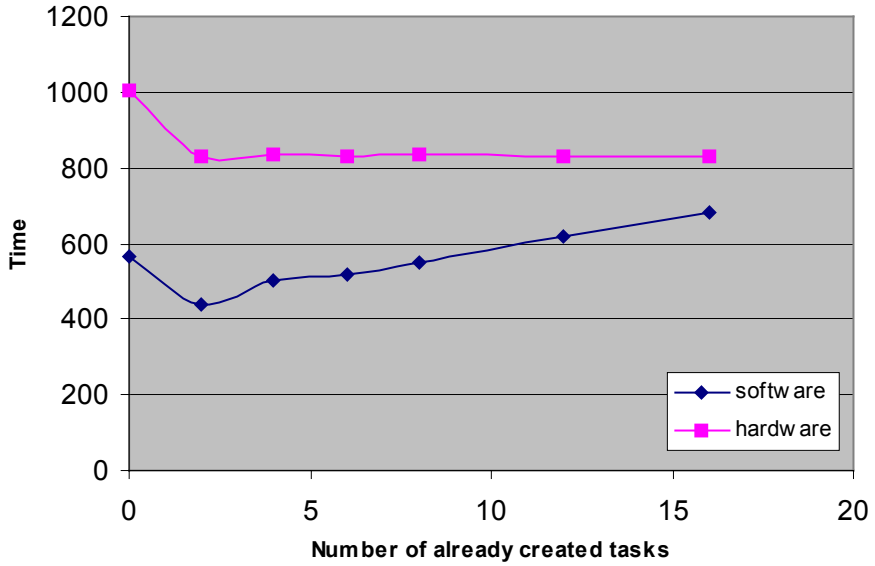


Fig 5, Create task benchmarks on the master node for both operating systems.

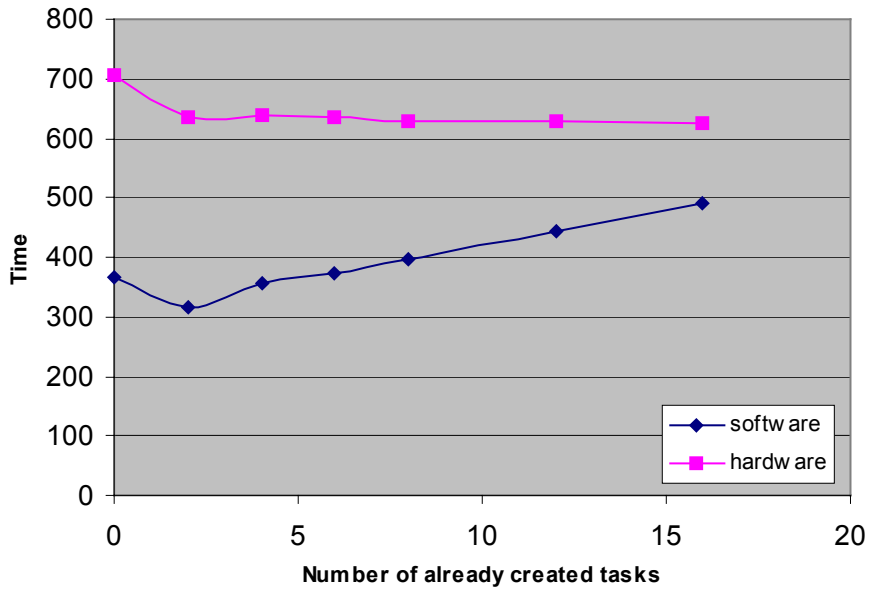


Fig 6, Create task benchmarks on the slave node for both operating systems.

Task switch

This test case has been influenced by the Rhealstone benchmark and it measures the time the system takes to switch between two independent and active tasks with equal priority. The task switch 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.

Method

Measuring the time between that a task calls for a manual task switch, until the next task becomes executing is a reasonable method of measurement. 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.

Result

In this test the software operating system is faster than the hardware operating system, that is because of the task switches that can be handled locally on a node. Each node manages its own queue of active tasks. But in the hardware case the operating system queues are managed centrally in the operating system core, residing as a PCI device, which causes access latencies to the queues. Notice that the software operating system is not affected by the number of tasks in this test, which is because of the two tasks that are involved in the switch, has the highest priority in the system. In figure 7, the result from the test on the master node is compared and figure 8 summaries the result from the slave node. In appendix 1 tables with complete result listings are presented.

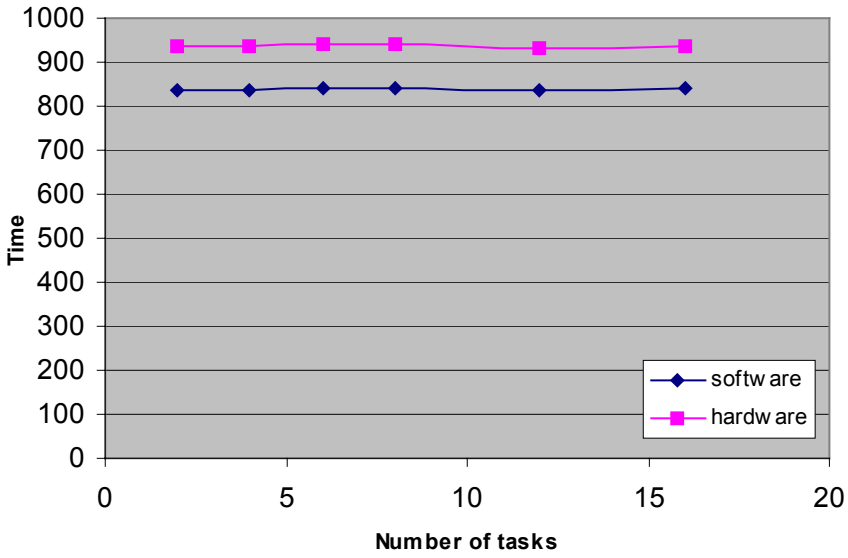


Fig 7, Task switch benchmarks on the master node for both operating systems.

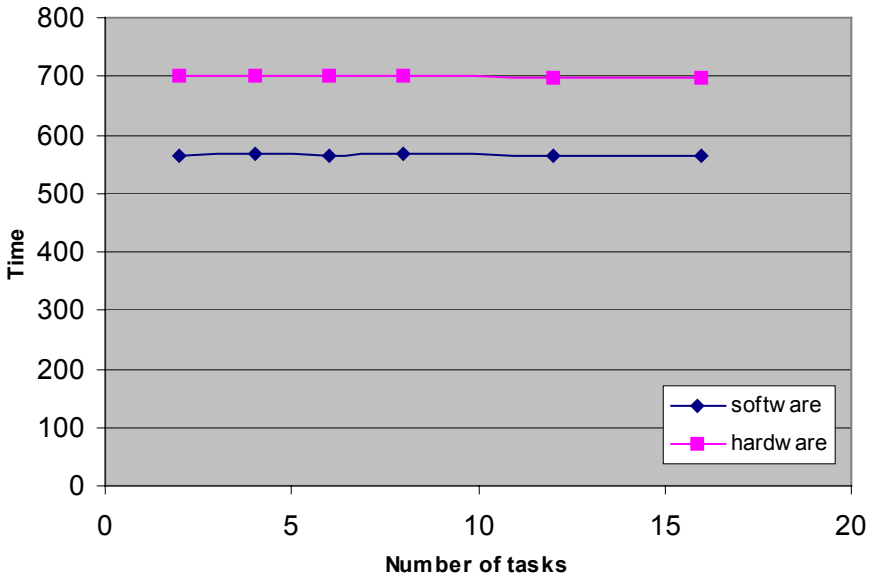


Fig 8, Task switch benchmarks on a slave node for both operating systems.

RTOS overhead

The method used in this benchmark is for instance used in [FUR00]. The paper presents a performance comparison between systems that utilises hardware operating systems and ordinary operating systems. The comparison is based on executing an application. In this test case two different kinds of applications are

going to be used as variation to the measurement, one application that communicates heavily and another that does not any exchange messages.

Method

Practically this measurement should be possible by comparing execution times between the hardware implemented and the software implemented RTOS, and it is not really necessary to know the exact utilisation of the application. As applications the classic problems Nqueens and Travelling Salesman was chosen.

Nqueens is the problem to find all the positions of *n* queens on an n*n chessboard so that no two queens attack each other (not on the same column or diagonal). The base algorithm to solve the problem is a recursive algorithm that finds all solutions, by trying all solutions.

The Travelling Salesman Problem (TSP) is a classic combinatorial problem. It is also practical since it is the basics for things like scheduling planes and personnel at an airplane company. Given are *n* cities and a symmetric matrix dist[1:n,1:n]. The value in dist[i,j] is the distance from city i to j. A salesman starts in city 1 and wishes to visit every city exactly once, ending back in city 1. The problem is to determine a path that minimizes the distance the salesman must travel. The algorithm to solve the problem is based on a work pool that resides on the master node, while the two slaves fetches pieces of work and report results back to the master. In this test the whole system is exercised.

Result

In figure 9, we can see that for Nqueens the result is essentially the same for both the hardware and the software operating system. But when running the TSP application and exercising the whole system with messages as synchronisation, the hardware operating system is faster. In appendix 1, the complete result listings are included.

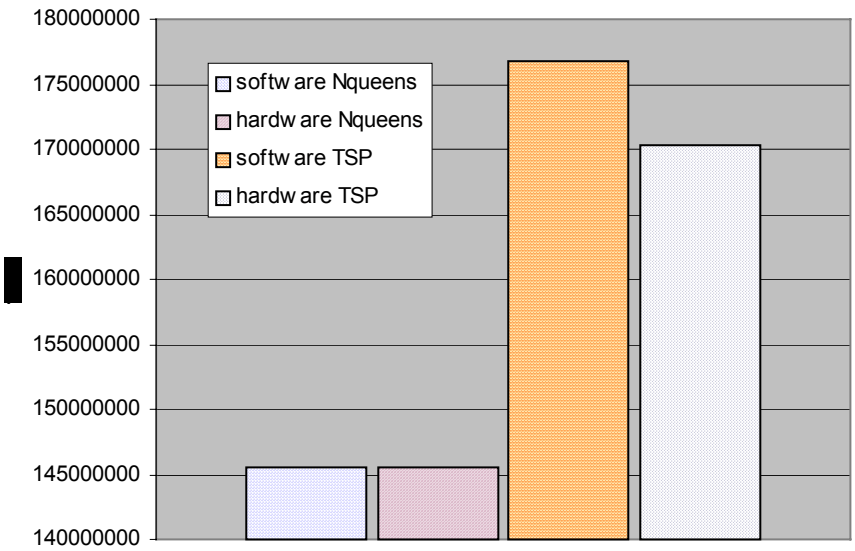


Figure 9, an application benchmark between the two operating systems.

Communication bandwidth

Variants of this measurement are included in Rhealstone and Distributed Hartstone. This measurement aims to measure the number of kilobytes per second one task can send to another. Included in this metric is not only the limitation of the physical communication medium, but also the operating system overhead required for sending messages. The communication bandwidth may be different for tasks hosted by different processors and tasks hosted by the same processor. For this reason the result will be reported by the communication bandwidth between:

- Two tasks hosted by the master node
- Two tasks hosted by a slave node
- The sender on a slave node, and the receiver on the master node

The bandwidth is also in most systems depending of the message size. The message size is therefore a target for variation.

Method

Practically the measure can be carried out by measuring the time it takes to send a fixed amount of raw data between two tasks, with no other communication present. The time is from which the first message is sent by the sending task, until the last message is received by the receiving task. The number of messages the data is dived into is varied between 10 and 640, and the amount of data is 10240 bytes. This gives message sizes varying between 16 and 1024 bytes. The communicating tasks placement is also varied in independent tests.

Result

This benchmark shows that the bandwidth between tasks hosted by the same node is greater with the software operating system, but when communicating across node boundaries the bandwidth is greater with the hardware operating system. The result is probably depending of the rather complicated message semantics with priority boosting of the receiving task. When that priority change is taking place on the same node, i.e the sender and the receiver are on the same node. The software operating system is more efficient, since the enclosed latencies with PCI accesses can be avoided. But when communicating across node boundaries the hardware implementation is more efficient. The priority boosting occurs inside the hardware without any processor involvement, while the software operating system processors must communicate and synchronize with each other to achieve the priority boosting.

The three figures 10, 11 and 12 summarises the bandwidth benchmarks. In figure 10 the bandwidth between tasks on the master node is compared. In figure 11 the bandwidth between two tasks on a slave node is compared, notice that the bandwidth in this test is lower than when both tasks are residing on the master node. The explanation is that the communication is implanted through a shared memory that resides on the master node; this gives the master node shorter access times. Finally figure 12 compares the bandwidth between tasks communicating across node boundaries. Complete results are presented in appendix 1.

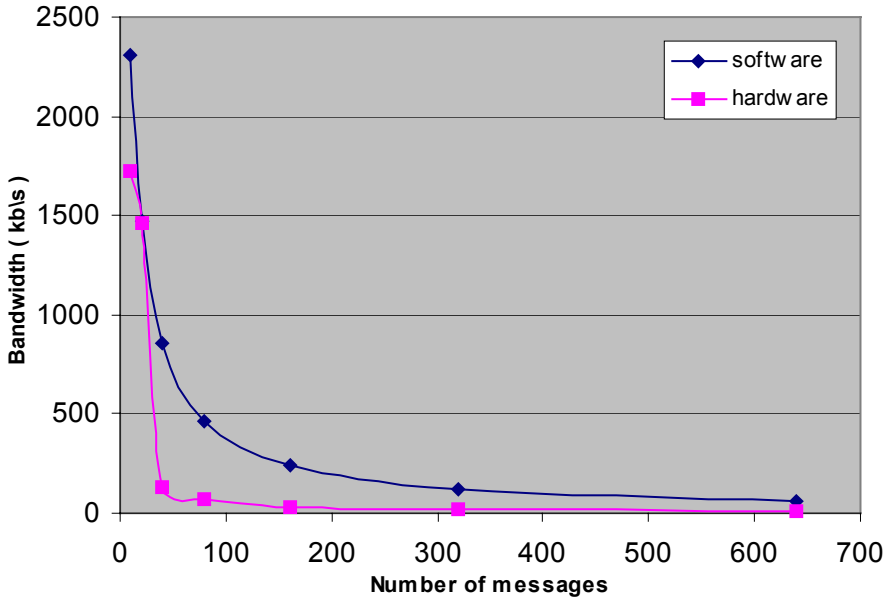


Figure 10, bandwidth between two tasks hosted by the master node for both operating systems.

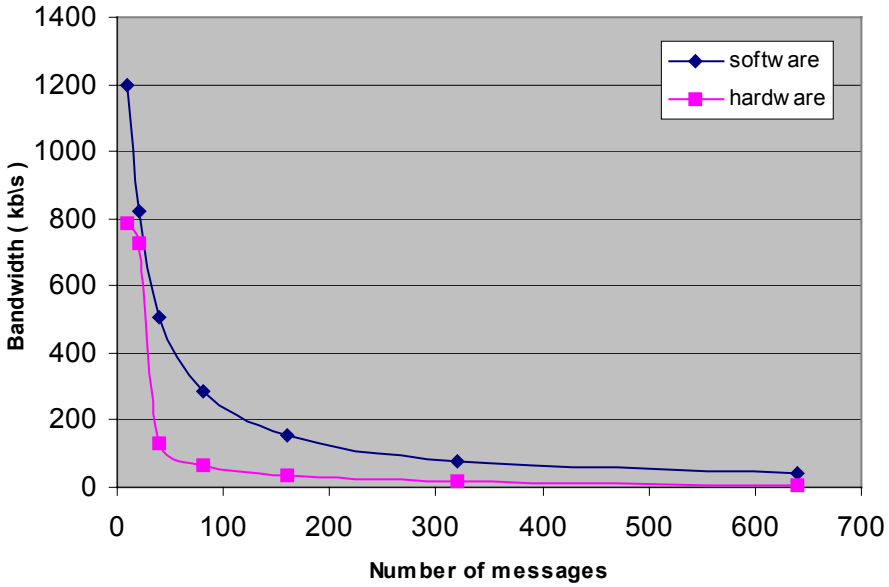


Figure 11, bandwidth between two tasks hosted by a slave node for both operating systems.

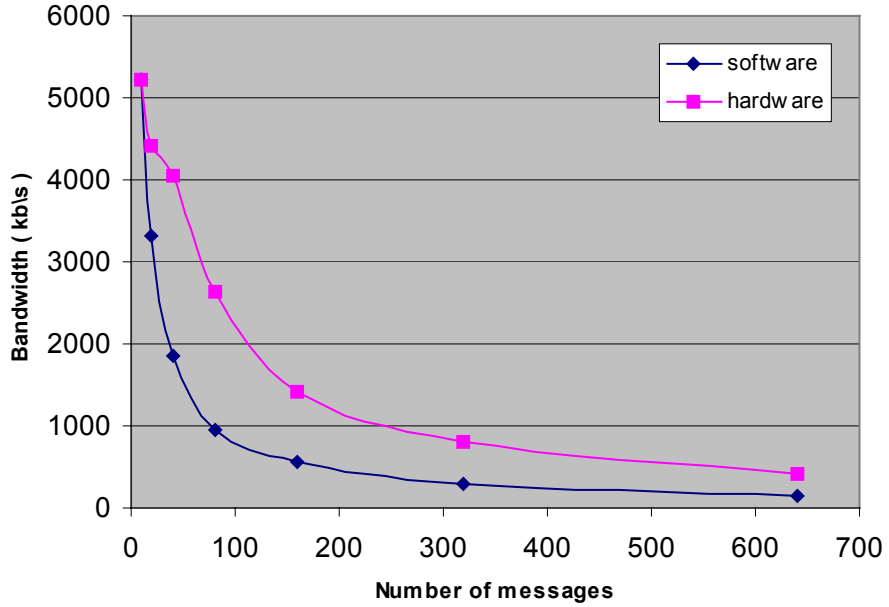


Figure 12, bandwidth between a sending task on a slave node and a receiving task on the master node.

Communication latency

The end-to-end communication latency is also measured in distributed Hartstone. This is an important metric in real-time systems, since many of the calculations used for guaranteeing deadlines use the end-to-end communication latency. A RTOS suitable for hard real-time systems should also be able to provide a bounded worst-case end-to-end delay for messages independent on other traffic.

The effect other traffic have on messages are tested by the message priorities measure also included in this benchmark. Using different message sizes with regular intervals between the two extremes tests the effect of message size. The effect of task placement will be taken care of by three different cases; the end-to-end delay will be reported by the delay between:

- Two tasks hosted by the master node.
- Two tasks hosted by a slave node.
- Sender and receiver on different nodes

Method

This measure simply consists of measuring the end-to-end delay, i.e. the time from which the message is sent by the sending task until it is received by the receiving task. In the case with the software operating system and communication across node boundaries, it is most suitable to measure the roundtrip delay. Since we have no accurate external clock and no clock synchronization between nodes, it is easiest to achieve an accurate result by taking the two necessary timestamps on the same node. The first timestamp is taken when a message is sent and the second timestamp when the receiver returns the message. The message sizes are varied between 1 and 128 bytes. The test is divided into three different independent measures, with sender and receivers placed as described above.

Result

This test shows that the end-to-end delay, also referred to, as the transmission latency is shorter with the hardware operating system. How come that the bandwidth and latency not follows each other, is an immediate reaction. The problem with the software operating system is that the scheduling routines are executed in a speculative manner, with the purpose to find out if we should switch task. A speculation takes place in association with many system calls, such as send and receive. These speculations surely take place in the hardware operating system too, but they do not steal any execution time on the systems processors. However, when the speculations are successful the software operating system also is successful. Such a case seems to be when a send and receive pattern is repeated, as in the bandwidth test.

The communication latency on the master node is compared in figure 13. Figure 14 shows a comparison of the latency on a slave node. Figure 15 shows the latency between tasks hosted by different nodes. Finally appendix 1 presents the

exact results. We can see that the differences are smallest on the master node, because that the software operating system routes the semaphore calls to the master node.

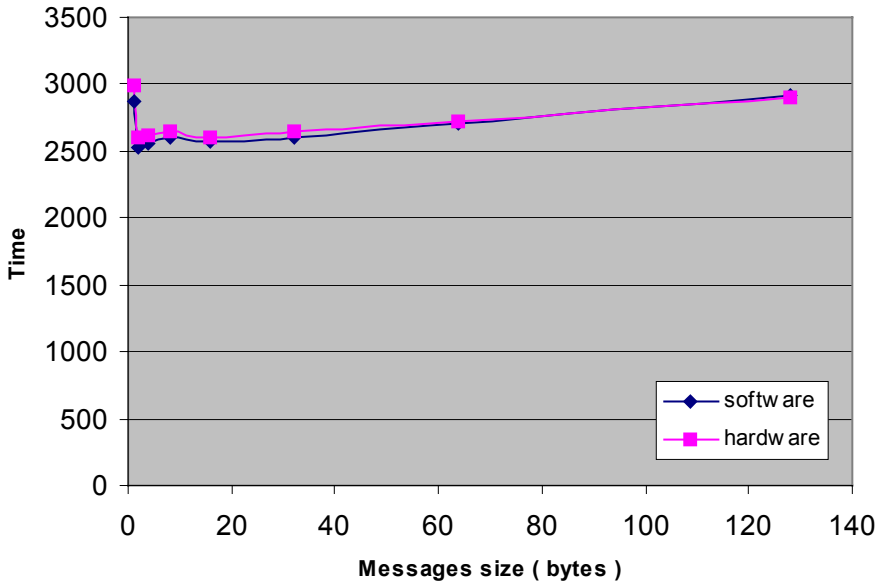


Figure 13, the communication latency between tasks residing on the master node for both operating systems.

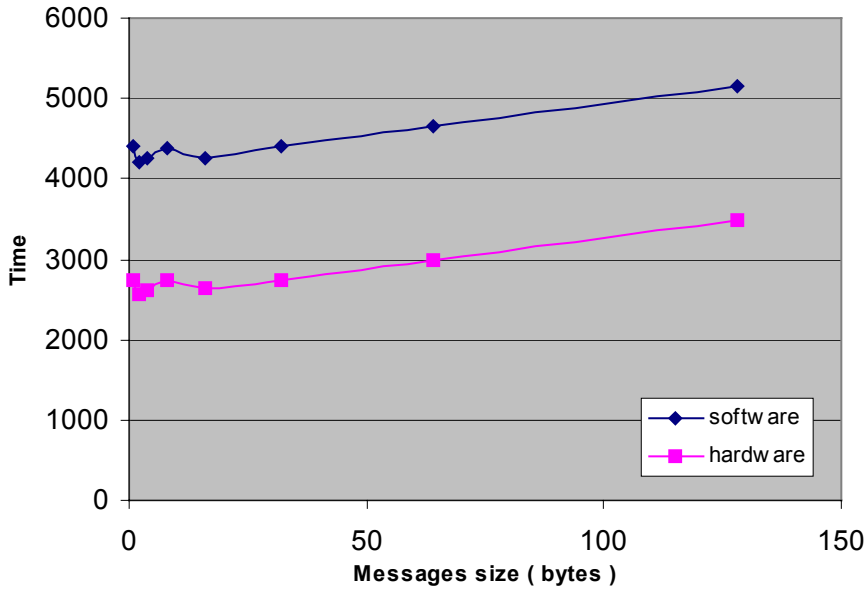


Figure 14, the communication latency between tasks residing a slave node for both operating systems.

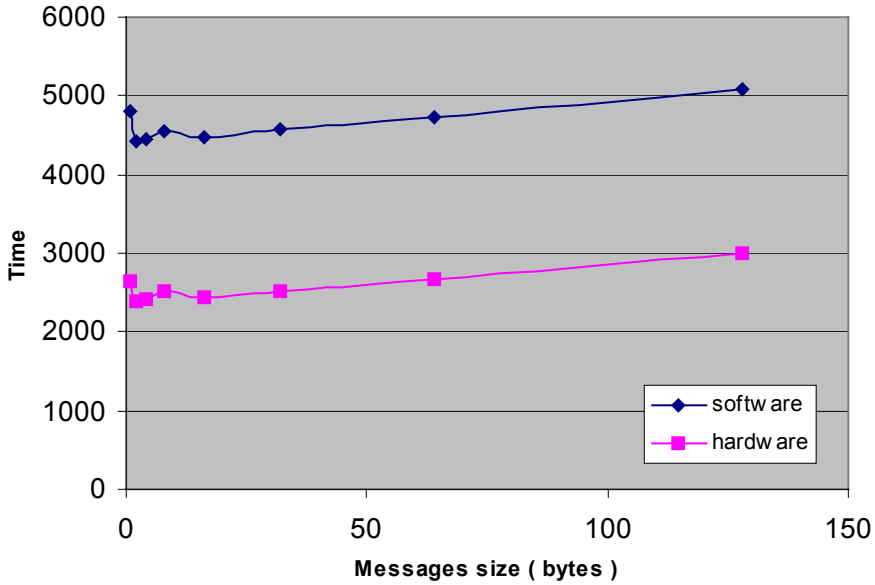


Figure 15, the communication latency between the sender residing on a slave node and a receiver residing on the master node for both operating systems.

Message priorities

In Distributed Hartstone, the message priority test is performed as a special real-time test. It is the scheduling of messages that is tested; in ordinary systems it is often First In First Out (FIFO) queues that handle the messages. But in a real-time system this is often not a suitable solution, since a high frequency or fast activity cannot be delayed by an arbitrary number of other activities. This test is aimed to test if the message passing algorithms can avoid priority inversions and how successful the algorithm is on handling message priorities. In the distributed Hartstone test, this benchmark has a result of yes or no type. To associate a number and thereby make the two solutions comparable, we aim to measure the time it takes to send a high-prioritised message, when the receiver have a huge number of low-prioritised messages pending.

Method

Practically an applicable method should be to let a task send as many low prioritised messages as possible. Simultaneously let another task send a high-prioritised message and compare the time it took with the ideal case, when no other traffic takes place. The message size is varied between 1 and 128 bytes. The placement of the sender and receiver is varied between, both on the master node, both on a slave node and the sender on a slave node and the receiver hosted by the master node. The task that sends low prioritised messages as fast as possible is always placed on an idle node. To achieve as much disturbing messages as possible it is important that this task never is interrupted.

Result

This test shows that the hardware operating system is much more successful than the software operating system in handling these situations. The problem with the software operating system seems to be that interrupts following message sends are queued in the interrupt handlers. Even if the message-passing algorithm itself is clearly priority driven, the messages are not delivered in correct order since the interrupts indicating message deliveries are routed in the sent order. Another problem is that all bridges between busses in the system are FIFO-driven and this may result in unpredictability's in both operating systems.

In figure 16 the results from the master node is shown. In the case with both the sender and the receiver on the slave node visualised in figure 17, the hardware operating system also shows an example of un-deterministic behaviour and the curve is a bit swingy. Figure 18 presents a comparison in the case with sender and receiver on different nodes. In appendix 1, the complete results are presented.

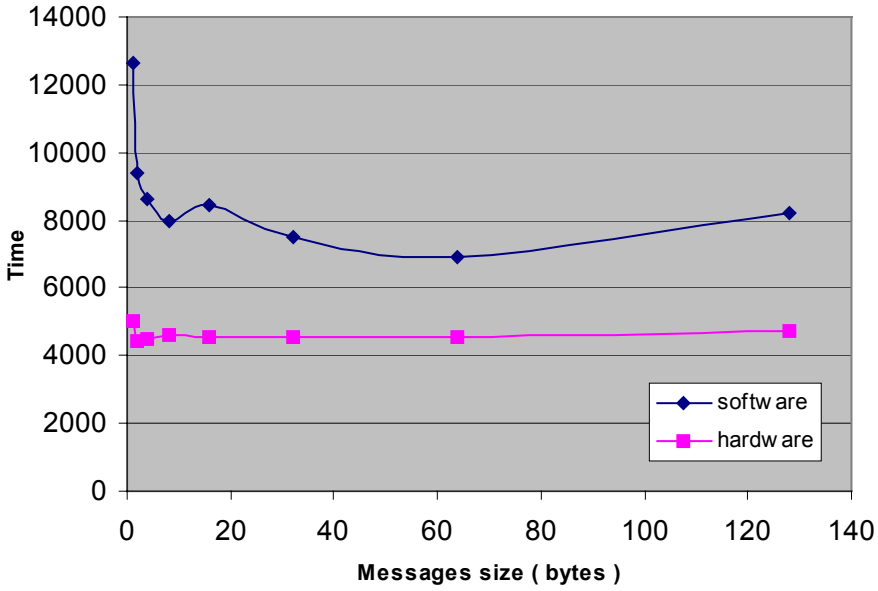


Figure 16, the message priority benchmark when the master node hosts both sender and receiver. A slave node sends the disturbing messages.

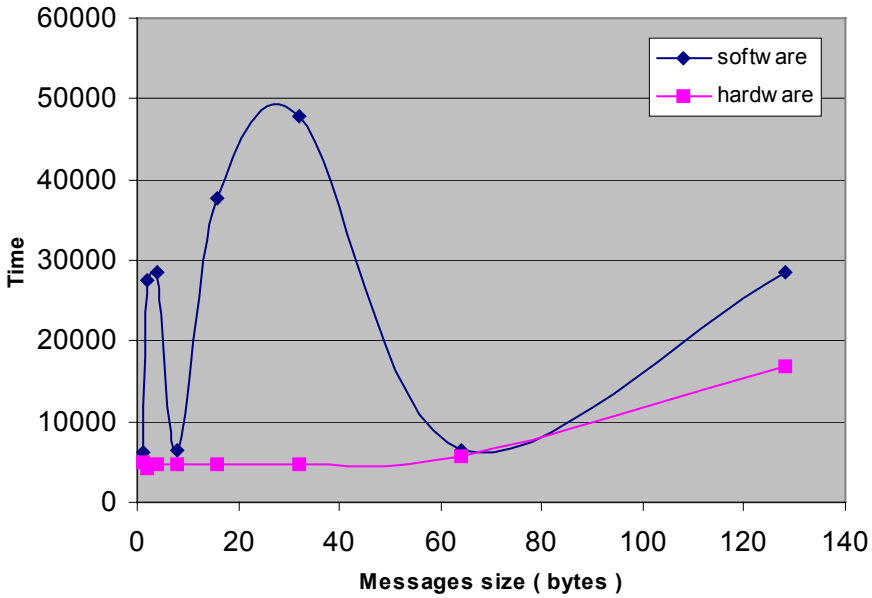


Figure 17, the message priority benchmark when a slave node hosts both sender and receiver. The master node sends the disturbing messages.

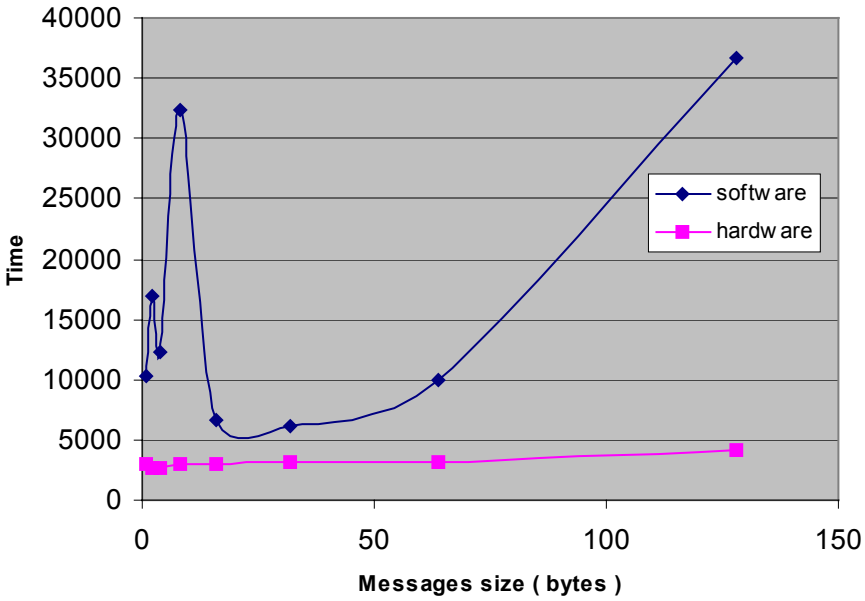


Figure 18, the message priority benchmark when the master node hosts the receiving task and a slave node hosts the sender. An idle slave node sends the disturbing messages.

Conclusions and future work

As summary and conclusion we end up with our opinions of advantages and disadvantages with the two operating systems.

The software operating system MPSWOS main advantage in comparison with the hardware operating system RTU is that some system calls can be executed in parallel. Intelligence has been moved from a centralized source to the system nodes, this implies that some decisions can be made faster and we can utilize the natural locality of data close to each node. An example is for instance the processor dispatch queues that has been moved to the respective node. It should also be easier to make small changes in the kernel since it is implemented in software instead of hardware.

The hardware operating system RTU has a great advantage over all software implementations, the execution of scheduling decisions and other operating system functionality does not load the systems 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. As we could see in our benchmarks, the main advantage with using the hardware implememantion in the SARA system today is that it is more deterministic than the software version. But notice that since scheduling decisions and many other system calls does not load the processors, the gain with a hardware implemenation becomes greater with more complicated algorithms and more tasks.

As a future work section we present some suggestions to improve the SARA and RTU implementations. The first and biggest proposal is to implement global memory areas on all nodes. Global memory areas on all CPU boards will relieve the pressure on the system board. As it works today, all communication goes through the system board, even when tasks that are located on slave boards are communicating with each other. Secondly since the scheduling algorithm does not load the application processors, make use of a really fancy and effective algorithm that hardly cannot be used in software implementations. Finally we think that an improvement of the interface between processors and the RTU should be useful, it is often big latencies in communicating across the PCI busses. As for example the dispatch queues could be moved to the processors, as in the software operating system.

Acknowledgements

Thanks goes to Peter Nygren and Johan Stärner, both currently PhD students at the department of computer science and engineering at Mälardalen University, for their supportive work as supervisors. A special thanks goes to Leif Enblom, for many fruitful discussions of ideas.

References

[ACH91] S, Acharya et al., Overview of Real-Time Kernels at the Superconducting Super Collider Laboratory, Particle Accelerator Conference, 1991.

[FUR99] J. Furunäs et al, Flexible Multiprocessor computer Systems, In CAD & Computer Graphics, December 1999.

[FUR00] J. Furunäs, Benchmarking of a Real-Time System that utilises a booster, International Conference on Parallel and Distributed Processing Techniques and Application, 2000.

[KAM91] N. 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.

[KAR89] R. Kar and K. Porter, Rhealstone - a Real-Time Benchmarking Proposal, Dr. Dobbs' Journal, February 1989.

[NYG00] 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.

[RAM91] K. Ramamritham and J. A. Stankovic, The Spring Kernel: A new paradigm for Real-Time Systems, IEEE Software, May 1991.

[RIZ01] L. Rizvanovic, Comparison between Real time Operative systems in hardware and software, Masters' thesis presented at Mälardalen University, Västerås, Sweden 2001.

[RTU00] 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.

[SHO01] M. El Shobaki and Lennart Lindh, A Hardware and Software Monitor for High-Level System-on-Chip Verification, Proceedings of the IEEE International Symposium on Quality Electronic Design (IEEE), San Jose, CA, USA 2001.

Appendix 1

Task create software master			
Number of tasks	min	max	average
0	563	567	564,2
2	437	439	438
4	497	502	500,8
6	519	522	520
8	550	552	551,2
12	612	622	616,2
16	675	686	681,4

Task create software slave			
Number of tasks	min	max	average
0	363	374	365,4
2	312	322	314,6
4	351	362	357,6
6	371	380	372,8
8	395	404	397,4
12	439	448	443,6
16	486	497	490,4

Task create hardware master			
Number of tasks	min	max	average
0	996	1010	1002,4
2	822	833	828,2
4	831	844	835
6	828	831	829,6
8	831	839	835,6
12	825	838	829,6
16	822	841	832,4

Task create hardware slave			
Number of tasks	min	max	average
0	703	711	706
2	627	649	636,8
4	633	646	638,4
6	629	640	636
8	627	631	627,8
12	625	634	627
16	624	626	624,6

Task switch software master			
Number of tasks	min	max	average
2	834	845	836,6
4	834	844	837,8
6	834	845	840,8
8	835	845	839,2
12	834	843	838,2
16	834	846	840

Task switch software slave			
Number of tasks	min	max	average
2	562	569	565,2
4	562	570	566
6	560	570	564,8
8	567	569	568
12	560	568	562,6
16	559	569	564,4

Task switch hardware master			
Number of tasks	min	max	average
2	931	943	936,6
4	921	946	936,2
6	937	943	939
8	930	943	938,8
12	919	945	932,2
16	930	941	937

Task switch hardware slave			
Number of tasks	min	max	average
2	696	707	700,4
4	692	719	701,8
6	695	706	701,4
8	686	715	699,8
12	694	702	698,6
16	694	701	696,8

RTOS overhead software			
Application	min	max	average
Nqueuens	145611618	145612170	145611973,2
TSP	176728413	176732478	176730136,2

RTOS overhead hardware			
Application	min	max	average
Nqueuens	145605961	145606827	145606411,8
TSP	170350982	170359110	170353193

Bandwidth software master/master			
Number of messages	min	max	average
10	2313,294735	2313,576405	2313,413661
20	1475,160664	1475,415211	1475,234479
40	851,8286062	852,0068556	851,9117848
80	461,6884046	461,7295417	461,701618
160	241,0366063	241,048158	241,0419744
320	123,1760355	123,1793184	123,177189
640	62,29388882	62,30896001	62,30290881

Bandwidth software slave/slave			
Number of messages	min	max	average
10	1196,044147	1197,199664	1196,648542
20	821,7074378	822,7590487	822,3121268
40	504,3293618	504,8027819	504,6273603
80	284,5831385	284,8206047	284,703793
160	152,2172386	152,2448838	152,2314403
320	78,86757002	78,93058219	78,89431595
640	40,14321302	40,16127638	40,15112246

Bandwidth software slave/master			
Number of messages	min	max	average
10	5205,247618	5210,402035	5208,212697
20	3305,939781	3307,28245	3306,630227
40	1851,046442	1851,778057	1851,228832
80	957,3334752	957,4889208	957,4095879
160	572,105316	572,5123243	572,2755495
320	296,4313759	298,7456763	297,5183925
640	153,7723493	154,200667	153,9889073

Bandwidth hardware master/master			
Number of messages	min	max	average
10	1726,655897	1728,278776	1727,741162
20	1456,180388	1457,495952	1456,843002
40	133,5179073	133,5260391	133,5215561
80	68,06516118	68,06860198	68,06696013
160	34,59892716	34,60068429	34,59995062
320	15,5208208	15,52104055	15,52096956
640	7,377148613	7,377244724	7,377214235

Bandwidth hardware slave/slave			
Number of messages	min	max	average
10	786,0449082	786,3629884	786,1526099
20	724,3646222	724,8220878	724,5868699
40	131,0706438	131,0796858	131,07629
80	67,42353602	67,42699201	67,42496625
160	34,43009879	34,43144366	34,43079202
320	15,50346835	15,50370449	15,50356365
640	7,375079968	7,375148035	7,375110121

Bandwidth hardware slave/master			
Number of messages	min	max	average
10	5122,300434	5253,863406	5224,245917
20	4292,854364	4565,814065	4409,570644
40	4038,016978	4063,781754	4048,15388
80	2620,691769	2647,633498	2635,179051
160	1423,429722	1426,760945	1424,821782
320	812,4435829	815,3216079	813,9432077
640	425,3533563	426,6932818	425,8482222

Latency software master/master			
Size of messages	min	max	average
1	2871	2881	2876,2
2	2529	2533	2531,2
4	2551	2560	2555,4
8	2602	2611	2606
16	2560	2572	2565,2
32	2603	2617	2609,2
64	2705	2716	2711,2
128	2911	2913	2912,6

Latency software slave/slave			
Size of messages	min	max	average
1	4390	4432	4408,4
2	4185	4233	4207,8
4	4239	4272	4260,8
8	4370	4390	4380
16	4238	4297	4267,8
32	4388	4406	4395
64	4623	4661	4650,4
128	5136	5175	5147

Latency software slave/master			
Size of messages	min	max	Average
1	4800,5	4825	4809,3
2	4406	4440	4423,9
4	4445	4481	4461,1
8	4556	4562	4558,9
16	4460	4491,5	4472,5
32	4552,5	4578,5	4566,3
64	4717,5	4749	4733,4
128	5076,5	5091	5085,2

Latency hardware master/master			
Size of messages	min	max	Average
1	2970	3004	2992,4
2	2592	2609	2601,4
4	2597	2625	2610,4
8	2641	2662	2652,4
16	2591	2611	2604,2
32	2643	2660	2649,6
64	2713	2739	2725
128	2902	2911	2905,4

Latency hardware slave/slave			
Size of messages	min	max	Average
1	2736	2743	2739,2
2	2558	2583	2572,2
4	2609	2633	2623
8	2737	2759	2747,6
16	2628	2649	2639,2
32	2733	2753	2745,8
64	2977	3002	2989,2
128	3469	3488	3475,6

Latency hardware slave/master			
Size of messages	min	max	Average
1	2633	2687	2650,1
2	2392	2405	2398,1
4	2406,5	2434,5	2422,8
8	2478,5	2515,5	2505,9
16	2421	2439,5	2428,1
32	2495,5	2522	2507,7
64	2655,5	2687	2672,2
128	2998,5	3021	3006,3

Message priorities software master/master			
Size of messages	min	max	Average
1	7100	14002	12615,4
2	9064	9513	9401,4
4	5195	9540	8629,4
8	5229	13159	7946,8
16	5203	13130	8468
32	5274	9535	7481,4
64	5379	9485	6939,6
128	5664	10667	8235,4

Message priorities software slave/slave			
Size of messages	min	max	Average
1	6142	6589	6270,4
2	7664	57593	27616,2
4	8817	57851	28411,2
8	6276	6604	6360,6
16	7727	57835	37706
32	7804	57887	47748,2
64	6449	6595	6498
128	8502	58490	28497,2

Message priorities software slave/master			
Size of messages	min	Max	Average
1	7147	19078,5	10218,2
2	7211	29961,5	16926,4
4	7055	21772,5	12314,8
8	4093,5	135553,5	32344,7
16	3597	9527,5	6571,5
32	4139	7104,5	6098,6
64	3722,5	16848	9985,3
128	5648,5	135579,5	36679

Message priorities hardware master/master			
Size of messages	min	Max	Average
1	4709	5708	5016,8
2	4261	4612	4429,2
4	4294	4588	4479,6
8	4367	4800	4586,8
16	4485	4677	4551,2
32	4466	4627	4536
64	4383	4665	4522,6
128	4577	4779	4704,8

Message priorities hardware slave/slave			
Size of messages	min	max	Average
1	4775	5015	4879,8
2	3851	4771	4325,8
4	3897	4926	4617,6
8	4636	4848	4698
16	4370	4979	4603,8
32	4315	5105	4797,8
64	5216	7270	5781,8
128	5509	60301	16736,2

Message priorities hardware slave/master			
Size of messages	min	max	Average
1	2581,5	3530	2965,3
2	2376,5	2988	2710,2
4	2420,5	2967,5	2596,5
8	2590	3314	3000,2
16	2764,5	3157,5	2929
32	2908	3411	3168,8
64	2750,5	4067	3095
128	3644	4664,5	4083,1