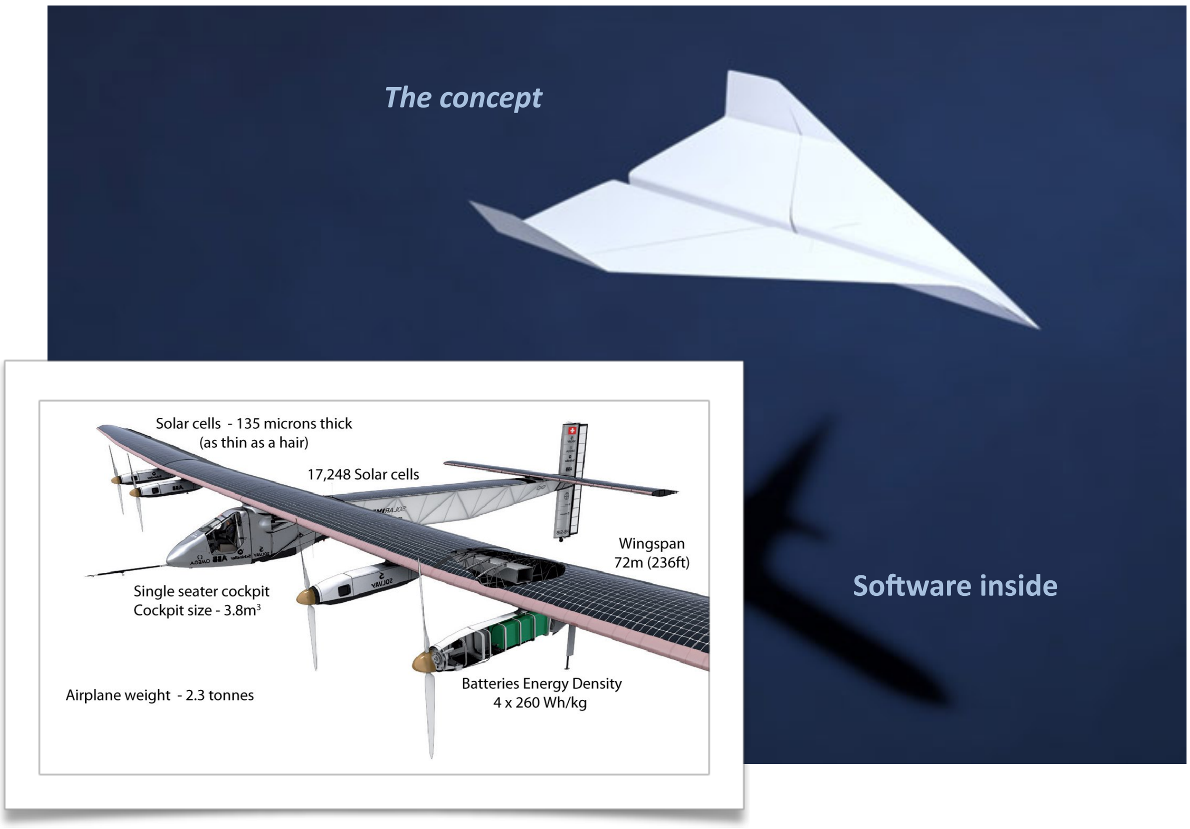
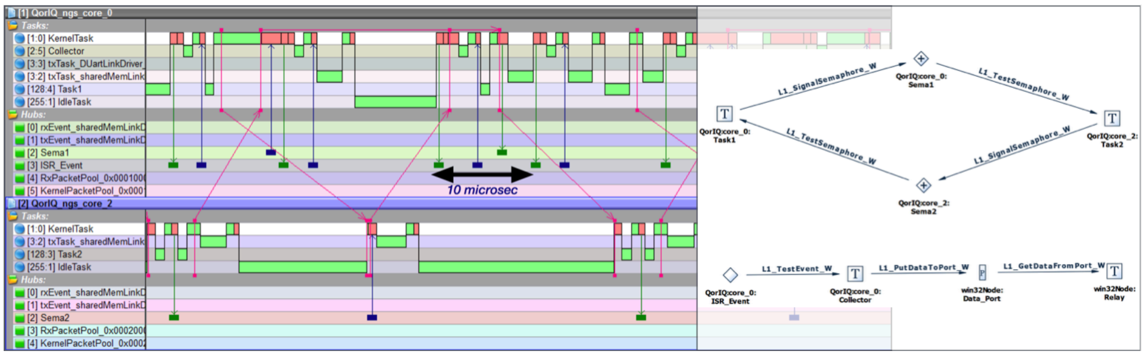
March 2017


The concept
Software inside
Solar cells - 135 microns thick (as thin as a hair)
17,248 Solar cells
Wingspan 72m (236ft)
Single seater cockpit Cockpit size - 3.8m³
Airplane weight - 2.3 tonnes
Batteries Energy Density 4 x 260 Wh/kg

ALTREONIC "FROM DEEP SPACE TO DEEP SEA"

QoS AND REAL TIME REQUIREMENTS FOR EMBEDDED MANY- AND MULTICORE SYSTEMS - UPDATED



SYSTEMS ENGINEERING FOR SMARTIES

Published by:
Altreonic NV
Gemeentestraat 61A B1
B3210 Linden
Belgium

www.altreonic.com
info.request (@) altreonic.com

March - 2017

Author: Eric Verhulst, Bernhard Sputh

Contact: goedelseries@altreonic.com

2nd publication in the Gödel Series:

SYSTEMS ENGINEERING FOR SMARTIES©

QoS and Real Time Requirements for Embedded Many- and Multicore Systems

Table of Contents

Preface

This booklet is the second of the **Gödel* Series**, with the subtitle "Systems Engineering for Smarties". The aim of this series is to explain in an accessible way some important aspects of trustworthy systems engineering with each booklet covering a specific domain.

The first publication is entitled "**Trustworthy Systems Engineering with GoedelWorks**" and explains the high level framework Altreonic applies to the domain of systems engineering. It discusses a generic model that applies to any process and project development. It explains the 16 necessary but sufficient concepts. This model was applied to the import of the project flow of the ASIL (Automotive Safety Integrity Level) project of Flanders's Drive whereby a common process was developed based on the IEC-61508, IEC-62061, ISO-DIS-26262, ISO-13849, ISO-DIS-25119 and ISO-15998 safety standards covering the automotive on-highway, off-highway and machinery domain.

The second publication is entitled "**QoS and Real Time Requirements for Embedded Many- and Multicore Systems**". It explains the principles behind real-time scheduling for embedded real-time systems whereby meeting the real-time constraints often is a top level safety requirement. What distinguishes this booklet is that it also deals with systems that have multiple processors (on-chip or connected over a network). The complexity and challenges on such targets mean that the system must now schedule all available resources, such as communication backbones, peripherals and energy. In combination with new functional needs this results in new approaches focusing on the Quality of Service and requiring specific support from the hardware.

The name of Gödel (as in GoedelWorks) was chosen because Kurt Gödel's theorems have fundamentally altered the way mathematics and logic was approached, now almost 80 years ago. The attentive reader will also recognise Heisenberg, Einstein and Wittgenstein on the front page. What all these great thinkers really did was to create clarity in something that looked very complex. And while it required a lot of hard thinking on their side, it resulted in a very concise and elegant theorem or formula. One can even say that any domain or subject that still looks complex is really a problem domain that is not yet fully understood. We hope to achieve something similar, be it less revolutionary, for the systems engineering domain and it's always good to have intellectual beacons to provide guidance.

The Gödel Series publications are freely downloadable from our web site. Further titles in the planning will cover topics of Real-Time programming, Formal Methods and Safety Analysis methods. Copying of content is freely permitted provided the source is referenced. As these booklets will be updated based on feedback from our readers, feel free to contact us at goedelseries @ altreonic.com.

Eric Verhulst,

CEO/CTO Altreonic NV

*: pronunciation [ˈkʊʁt ˈɡøːdəl] (🔊 listen)

1. Introduction

In this booklet we discuss the requirements and specifications for a Real-Time Operating System (RTOS) from the point of view of its capabilities to support embedded applications in meeting safety, particularly **real-time requirements on modern many/multicore systems**.

As the booklet is related to a **multiprocessor and distributed real-time operating system** this is rather unique as most RTOS are designed for single processor systems and if not, they assume a shared memory architecture. In such cases, the RTOS is only concerned with the local scheduling on each processor with interprocessor synchronisation and communication being left to a middleware layer. In view of modern **many/multicore architectures** this is no longer adequate as resources are globally shared on the highly integrated chip and any activity on any processor has a potential impact on the scheduling on another processor. We outline how simple real-time requirements are addressed by using static scheduling schemes. While predictable they are prone to catastrophic failure, which in the case of the highly concentrated functionality and performance in many/multicore chips can be catastrophic for the whole system. Moreover, their behaviour is increasingly statistical in nature, hence soft real-time rather than hard real-time. We show how a more dynamic approach can make better use of the available resources and can allow fault containment as well as recovery from errors. This extends the traditional real-time requirements into a wider concept that we call **Quality of Service (QoS).**

real-time

adj

(Electronics & Computer Science / Computer Science) denoting or relating to a data-processing system in which a computer receives constantly changing data, such as information relating to air-traffic control, travel booking systems, etc., and processes it sufficiently rapidly to be able to control the source of the data

Collins English Dictionary – Complete and Unabridged © HarperCollins Publishers 1991, 1994, 1998, 2000, 2003

In order to meet QoS levels, system components (on-chip or distributed) must meet certain criteria. We call this the **Assured Reliability and Resilience Level (ARRL)** and link it with the QoS. We use the formally developed network-centric VirtuosoNext™ as a reference.

2. Background of VirtuosoNext

The initial purpose for developing VirtuosoNext [1] was to provide a soft- ware runtime environment supporting a coherent and unified systems engineering methodology based on "**Interacting Entities**", currently further developed and commercialised by Altreonic [2]. In this methodology requirements result in concrete specifications that are fulfilled in the architectural domain by concrete "entities" or sets of entities. Entities can be decomposed as well as grouped to fulfill the specifications. In order to do so, we also need to define "interactions", basically the actions that coordinate the entities. In practice these interactions can be seen as protocols whereby the entities synchronise and exchange data.

Interactions and entities are first of all abstractions used during the modelling phase. As such, a specified functionality can first be simulated as part of a simulation model, critical properties can be formally verified using formal techniques and finally an implementation architecture can be defined using the architectural modelling tools of the target domain. In our case we try to keep the semantics unified from early requirements till implementation. In the targeted embedded systems domain this means that the final architecture is likely a concurrent or parallel software program running on one or more programmable processors. Some functionality might be implemented on specific hardware entities. Such entities will be integrated in the input or output subsystem or will be designed as co-processing blocks. In most cases these hardware entities will be controlled from a software driver running on a processor.

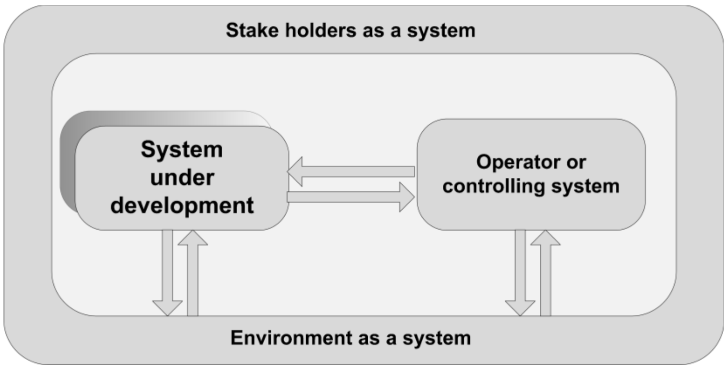


Figure 1 The context of systems engineering

In an embedded system, and in most systems, two additional systems must be taken into consideration. The first one is the "**environment**" in which the embedded system is placed. This will often generate inputs to the system or accept outputs from it or it will influence the operating conditions, not necessarily in a fully predictable way. A second system that is often present is the "**operator**", who also will generate inputs or process the outputs. If this is a human operator, we have to deal with an entity whose behaviour is not necessarily always predictable. Often the "operator" might be another embedded system and then the behaviour should be more predictable, at least if well specified. However, systems are layered. If we "open" the embedded system or consider the system under development with its environment and its operator as a new system, we can see that each system can be a component in a larger system and often it will be composed itself of "subsystem components", resulting in specific requirements in order to reuse them. For this paper we stay at the level where such components are programmable processors or software implemented functions.

The use of a concurrent (parallel by extension) programming paradigm embodied in an RTOS is a natural consequence of the interacting entities paradigm. Programming in a concurrent way implies that the abstract entities (that fulfill specifications) are mapped onto RTOS "**tasks**" (also called processes or threads in the literature) and that interactions are mapped onto services used by the tasks to synchronize and to communicate. In principle, this abstract model maps equally well to hardware as to software but we focus here on the soft- ware. The target domain ranges from small single chip micro-controllers over multi-core CPUs to widely distributed heterogeneous systems that include support for legacy technology. The goal is to program such systems in a **transparent way**, independently of the processor or communication medium used.

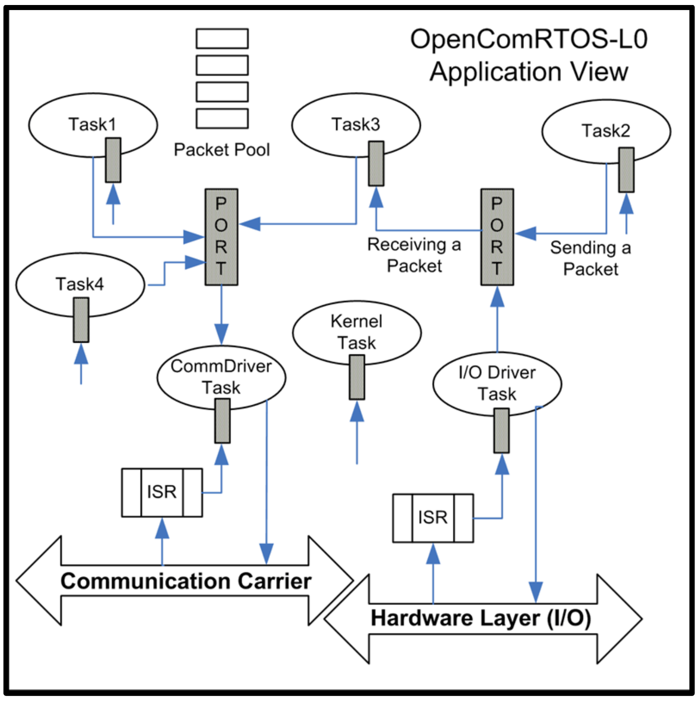
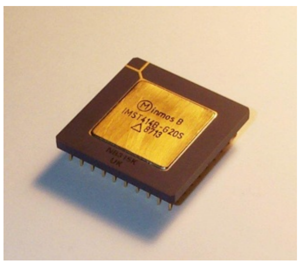


Figure 2 Interacting Entities mapped onto RTOS tasks and services

3. Early requirements derived from the Virtuoso RTOS



Precursor to VirtuosoNext were OpenComRTOS and the **Virtuoso RTOS** [3]. It had its origin in the pioneering INMOS transputer [4,5], a partial hardware implementation of **Hoare's Communicating Sequential Processes (CSP) process algebra** [6]. Later Virtuoso was ported to traditional processors but mostly parallel DSPs. The transputer was a rather unusual RISC like processor with unique support for on- chip concurrency and inter-processor communication. On-chip it had a scheduler with two priority levels, each level supporting round-robin scheduling between the compile time generated processes. It also had hardware support for inter- process communication and synchronization using "channels". For distributed, embedded real-time applications, it raised two major issues:

- Two levels of priority are not enough for hard real-time applications. Typically at least **32 levels of priority** are needed with full support for pre-emption and priority inheritance.
- **Topology independence**: although the transputer had interprocessor links, the communication between processors had to be manually routed at the application level. The issue is here mostly one of maintenance. Every little change in the topology could result in major reprogramming efforts.

Above observations resulted in the adoption in the Virtuoso RTOS of following architectural principles:

- Use of **255 levels of priority** with full pre-emption capability.
- Development of **traditional RTOS services** like events, semaphores, fifos, mailboxes, resource and memory maps.
- Use of **command and data packets** to provide for system level communication.
- Use of **system wide identifiers** and no local pointers to provide for **topology independent programming.**
- Packets carry a **priority inherited** from the generating task.
- Support for **priority inheritance** in the scheduler.

4. Real-time embedded programming

While most programming is concerned with performance (often expressed in terms of achievable throughput), real-time is then often equated to "fast enough". In the embedded domain however, the system will often interact with the physical world whereby stringent time requirements must be met or the system can fail. In such systems, the reactive behaviour is most important and must always be achieved in addition to the logical correctness of the application. Such systems are often called "**hard" real-time** in contrast with **"soft" real-time** systems whereby the timing properties are statistical in nature.

quality of service definition

communications, networking

(QoS) The performance properties of a network service, possibly including throughput, transit delay, priority.

Some protocols allow packets or streams to include QoS requirements.

4.1. Why real-time?

It can be argued that an architectural paradigm based on entities and interactions does not need any notion of real-time. Indeed, the **temporal properties** can be considered as mostly **orthogonal** to the **"logical" behaviour** of a system. In the embedded domain (and most of the systems we use have embedded aspects), we are dealing with real-world interactions and time is part of it. Signals that the embedded system must process arrive in real-time and must be dealt with before the next set of signals arrives. Similarly, the embedded system will act on it surroundings and real-time requirements apply. Implicitly, we assume here that **sampling theory** is applied. Sampling theory dictates that we should at least sample at twice the bandwidth of the signal. Similarly, when we apply output or control signals this must also be done with a rate at least equal to twice the bandwidth. If the controlled subsystem has a mechanical mass and its properties such that inertia determines the dynamic behaviour, we similarly must take into account its time constant. Sometimes, the output timing can be rather demanding. An example is audio processing. Our human ear is very sensitive to phase-shifts so that even when the bandwidth requirements are met, the jitter requirements are stringent enough that hardware support might be needed.

The purpose of an RTOS is to give the engineer the means to meet such real-time requirements at the same time as he is meeting the architectural ones (as explained before: mapping abstract entities into concrete tasks). Timely behaviour is then a property of the tasks in a specific execution context. This allows designing and verifying a real-time system without having to look into the details of the algorithms executed by the tasks. The only information needed is what resources the tasks use (e.g. time in the form of processing cycles and memory). Executing the task on another processor does not change the algorithm, just the timing and memory used. Similarly, a concurrent program in itself doesn't need to be real-time (it's a matter of defining the parameters differently). However, it is very convenient that a concurrent program that was designed to handle real-time, can also handle **time-independent programming**, e.g. for simulation purposes. The opposite is often not true.

4.2. Why a simple loop is often not enough

It is useful for the remainder of this paper to present in short our view on embedded real-time programming. The reader can find a wide range of literature related to real-time and embedded programming elsewhere if he wants to investigate in more depth.

Let's start with the term "**real-time**". The intuitive notion of real-time is often a subjective one using terms like "fast" or "fast enough". Such systems can often be considered as **"soft" real-time**, because the real-time criteria are not clearly defined and are often statistical. However, when the system that must be controlled is physical, often the deadlines will be absolute. An example of a soft real-time system is a video system. The processing rate is determined by the frame rate, often a minimum of 25 Hz and determined by the minimum rate needed for the eye to perceive the frames as a continuous image. The human eye will itself filter out late arriving frames and can even tolerate a missing frame. Even more soft real-time are on-line transaction systems. Users expect them to respond with e.g. one second, but accept that occasionally it takes tens of seconds. Of course, if a soft real-time application repeatedly violates the expected real-time properties the **Quality of Service** will suffer and at some point that will be considered a failure as well.

On the other hand hard real-time systems that miss deadlines can cause physical damage or worse, can result in deadly consequences if the application is **safety critical**, even when a "fail-safe" mode has been designed in. Typical examples are dynamic positioning systems, machine control, drive-by-wire and fly-by-wire systems. In these cases often the term "**hard real-time**" is used to differentiate. From the point of view of the requirements, hard real-time means ”**predictable**” and ”**guaranteed**” and a single deadline miss is considered a failure whether its design can tolerate some deviations or not.

Two conclusions can be drawn. First of all, a hard real-time system can provide “soft” real-time behaviour, but the opposite is not true. Secondly, when safety critical, a hard real-time system must remain predictable even in the presence of faults. In the worst case it could fail, but the probability from this happening must be low enough to be considered an acceptable risk.

Strictly speaking, no RTOS is needed to achieve real-time behaviour in an embedded system. It all depends on the **complexity** of the application and on the additional requirements. E.g. if the system only has to periodically read samples from a sensor, do some processing and transmit the processed values, a simple loop that is executed forever will be sufficient. Sources of complexity are for example:

- Putting the processor to sleep in between processing to conserve energy.
- Managing several 100’s of sensors.
- Executing a high number of other tasks with different time constraints.
- Detecting a failure in the sensor circuit.
- Detecting a fault in the processor.

Such requirements are difficult if not impossible to handle when a simple polling loop is used, but as most processors will have support for interrupt handling, the developer can separate the I/O from the processing. This essentially means that most embedded systems will have a “hardware” level of priorities and a “software” level of priorities. The highest priority level is provided by the **Interrupt Service Routines** that effectively interrupts the lower priority (background) loop. However, the extra functionalities listed above might already require multiple interrupts and priorities. The sleep mode of the processor requires that the circuit generates an interrupt to wake up the processor and a timer supporting a time- out mechanism might be needed for detecting a failure. Also the transmission of the processed values might require some interrupts. Hence the question arises how each interrupt must be prioritised. In the simple example given, this is not much of an issue as long as we assume that the system is periodic and always has spare time between samples. What happens however if multiple interrupt sources are present and if they can be triggered at any moment in time, even simultaneously?

4.3. Superloops and static scheduling

When multiple interrupt sources are present, a simple solution is to distribute interrupt handling and processing over the available interrupt service routines and the main polling loop. The **separation between “handling” and “processing” of interrupts** is essential because interrupts will be disabled when an Interrupt Service Routine is entered and worse, the hardware might be designed in such a way that the data is only available for a short period of time. Hence, while an interrupt is being handled, the hardware must have a mechanism for holding arriving interrupts, else they will be lost and in the worst case, the application can fail. Therefore interrupt handling should be kept as short as possible. On the other hand in the polling loop, the program will repeatedly test for the presence of the interrupt and when enabled execute the corresponding processing function.

The issue is that such testing and processing must be done in sequence and that the program cannot progress unless the interrupt has arrived. Hence, if all interrupts are to be seen and processed, a **static schedule** must be calculated and the peripheral hardware must be configured to be compatible with it. Such a schedule is not necessarily feasible, e.g. when the arrival rates of the interrupts have a wide span and don't follow a harmonic periodicity. In addition, the polling will waste processing cycles that could be used for useful processing and worst, if for some reason the interrupt does not arrive, the whole system can become blocked. From a safety point of view, such a polling loop has **no built-in graceful degradation**. In addition, even when no errors occur, a small change in the application can result in the need to

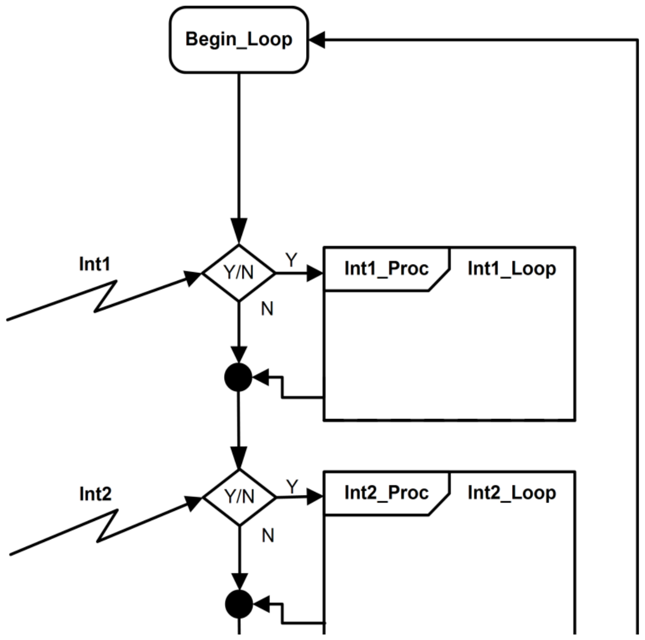
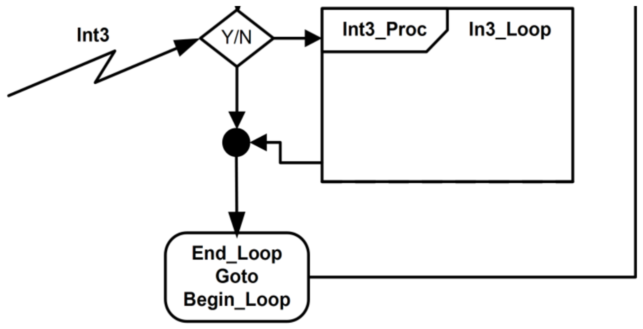


Figure 3 Superloop scheduling with three interrupt sources



recalculate the whole schedule or in the worst case can result in the application no longer being schedulable. What we need is a separation of concerns. The **logic of processing should be made independent of its behaviour in time**. With a sequential loop (on a sequential processor), this is not possible because the state space is shared amongst all processing functions and in addition the time behaviour depends on the temporal behaviour of the rest of the processing functions. What is needed is a mechanism that divides the global state space into local state spaces. There are two ways to achieve this:

- **Dedicating a processor** to each "local" processing function.
- Creating a mechanism that **separates the state spaces**, even when executed on the same processor.

The first solution has as side-effect that interprocessor communication can now become an issue (because communication media are also shared resources). The second solution creates the concept of "**virtualisation**", in essence a mechanism whereby each local processing function has virtually access to the full state of the processor. Note that this is only really possible because time is allocated to each virtual state space and this essentially means that to meet the real-time requirements at system level, this allocation of time must be carefully done to meet all real-time constraints.

The two solutions introduce both the notion of "**concurrency**", whether physical or virtual. Most real-time applications will however have "**interactions**" (e.g. passing data or synchronisation of a state that was reached) between the local state spaces. In line with the need for separation of concerns, we need a mechanism that "virtualises" these interactions independently of whether they take place on different processors or on the same processor.

And last but not least, while we separated the time behaviour from the logical behaviour, hard real-time systems still need a mechanism for handling time. This mechanism is called **scheduling**. We have seen a static version of it in the previous sections, called static scheduling. It assumes perfect knowledge about the system when it is built and assumes that the system's operating parameters are static and will never change. As outlined, this is seldom the case, certainly when failure conditions are taken into account. In general, a more dynamic scheduling mechanism is preferred. The scheduling can be based on a measurement of time or on the time already used. The most widely used mechanism is based on priorities, a ranking of the processing functions based on an analysis that combines the periodicity and the relative processing load. This mechanism is called **Rate Monotonic Scheduling (RMS)**. VirtuosoNext is a RTOS based on the assumption that a **Rate Monotonic Analysis (RMA)** is executed, resulting in a system wide priority ranking of the scheduled application functions. Nevertheless, the design allows for the implementation of different scheduling policies.

4.4. Rate Monotonic Analysis

RMA was first put forward in 1973 by Liu and Layland [7]. Although it doesn't solve all issues it provides a good framework that is simple and most of the time it is applicable. The algorithm states that given N tasks with a fixed workload that must be active with a fixed periodicity (with the beginning of the next period being considered as the deadline for the previous period), all deadlines will be met if the total processor workload remains below a value of 69% and a pre-emptive scheduler is used with each task receiving a priority that is higher if the task has a higher periodicity. The **upper bound of 69%** is obtained for an infinite number of tasks. For a finite number of tasks and especially when the periods are harmonic, the upper bound can be a lot higher, often even observed to be above 95%. Figure 4 illustrates RMA scheduling of two tasks. In general RMA defines the scheduleability criterion (on a single processor) as follows:

$$\sum_{j=1}^{n} \left(Cj/Tj\right) \le U(n) = n \,.\, (2^{\frac{1}{n}} - 1)$$

Figure 3 Superloop schedule with three interrupt sources

with:

- C_j being the worst case execution time of $task_j$;
- T_j being the execution time of $task_j$;
- U(n) being the worst case utilisation with n tasks;

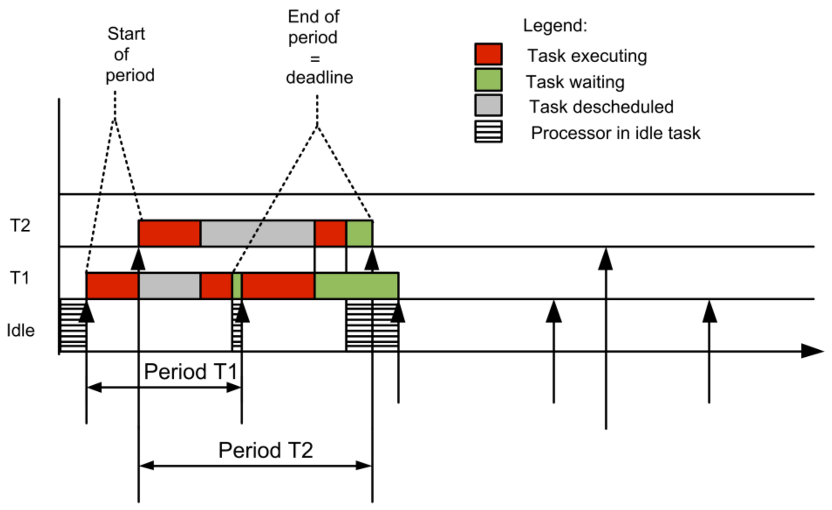


Figure 4 Two periodic tasks scheduled with RMA

According to the equation a system with one task has an utilisation of 1.0 (U(1) = 1.0). For an unlimited number of tasks the utilisation converges to **0.69** ($U(\infty) = 0.69$).

In practice the results of the first RMA algorithm are a (pessimistic) approximation and rely on some assumptions that are seldom met in real applications. For example, all tasks are assumed to be independent (hence they all are activated on independent events and do not synchronise or communicate with other tasks, nor do they share any resources). Also task activation is assumed to be instantaneous and the processor provides a fixed processing power (hence no cache effects). Even if often the 69% level is used as a maximum load in any case, this means that to remain on the safe side, it is often better to keep the overall CPU load lower than the figure obtained. On the other hand, if only a few tasks are used and the interactions are limited, often the application will miss no deadline even if the processing load is higher than 69%. The CPU load can also be higher if the periodicity of the tasks is harmonic. Hence RMA has to be seen as a **guideline** that must be complimented with a detailed analysis, profiling and especially measures to give the application more margin. It should also be pointed out that if a RMA schedule misses deadlines for the lower priority tasks that the higher priority tasks can meet their deadlines. This property of pre-emptive priority based scheduling is e.g. useful for creating a highest priority task that is only activated when exceptions have to be handled.

A very detailed and comprehensive analysis of RMA is given in Briand and Roy [8]. It also discusses the follow-up RMA algorithms that were developed later on taking into account realities like blocking times (using shared resources), inter-task dependencies and distributed systems. In all cases this results in higher boundaries for the CPU workload. The most important change to the basic RMA algorithm is that for determining the task priorities, one should not use the full period but the pseudo period that is derived by taking into account that the deadline of a task happens often before its period has expired. This is called **Deadline Monotonic Analysis (DMA)**. More extensive descriptions as well as algorithms for schedulability analysis for a wide range of RMA scheduling policies can be found in Ref [9].

It must be said however that for distributed systems no real RMA algorithm exists, although tools like MAST [10] allow verifying that a given schedule is feasible. In practice a system design with adequate priorities will give good assurance that all deadlines can be met.

An important observation is also that a rigorous and static design might not always give the most safe system if the first missed deadline results in catastrophic behaviour. Many real-world systems can tolerate missed deadlines if these misses have a low probability and if they are spread in time (not bursty). Of course, this means that the system design must take this into account. A classical example is a brake-by-wire system. It must be designed for the maximum speed of the car and hence often the maximum rate will be used all the time. Even at this highest rate, there will be margin as the time constant of the mechanical system will be lower. If the car then operates at a lower speed, the control rate can be lowered as well and missing control signals from time to time (but not in continuous bursts) will in the worst case only lower the ”quality” of braking, but this is often not catastrophic.

4.5. The application of RMA in VirtuosoNext

In VirtuosoNext it was decided to support priority based pre-emptive scheduling as the standard scheduling policy. In [8] this is called **Highest Priority First**. Every task can be assigned its own priority based on an off-line Deadline Monotonic Analysis (DMA). It must be said however that DMA assumes that all tasks execute on a single processor, whereas VirtuosoNext supports multi-processor systems. Hence priorities are considered a system-wide scheduling parameter and the DMA should still hold locally on each processor.

VirtuosoNext was also designed to clearly separate Interrupt handling (in ISRs) and interrupt processing (in a task). Good design practice dictates that a minimum time is spent in interrupt handling. This improves the responsiveness of the system and hence, because interprocessor communication often requires fast interrupt handling, it will reduce the latencies. The latter is especially important for multiprocessor systems as the processing can be distributed over several processors and the scheduling delay includes communication delays. Similarly, in the design of a network-centric RTOS it was recognised that delays can also be the result from implementation artefacts. Hence, any activity in the RTOS or its system level drivers is done in order of priority. This **minimises the point- to-point latency**. Typical cases where this can be important are waiting lists and interprocessor communication. This means that one should be able to ignore the different scheduling latencies as the communication delay can be more important (especially on slow-speed networks). This latency is a combination of several factors that are difficult to quantify. Factors are: communication load, communication set-up time, transmission delay and receiver latency. Therefore, good profiling tools are a necessity. DMA then provides a good approximation and starting point. For extreme processor loads (typically when the task's individual processing time is of the same order of magnitude as the system latencies), this assumption does not hold and often only static scheduling or dedicating processors to such loads is the only acceptable solution.

A small note however on the assignment of the priorities. In our case, these are assigned at design time and the scheduler is a straightforward **Highest Priority First one**. Research on dynamic priority assignment [11] have shown that algorithms that use **Earliest Deadline First** (EDF) algorithms (the priority becomes higher during execution for the tasks whose deadline is the nearest) can tolerate a workload of up to 100%. There are however three reasons why this option was not further considered. The first one is that the implementation of an EDF scheduler is not trivial because measuring how far a task is from its deadline requires that the hardware supports measuring this. As this is often not the case, one has to fall back on software based solutions that periodically record the task's progress. For reasons of software overhead, this must be done with a reasonable frequency; typically about one millisecond which means that fine-grain microsecond EDF is not feasible (one millisecond can be quite long for a lot of embedded applications). The second reason is that no algorithms are known that allow calculating the EDF schedule on a distributed or multicore target. The third, but fundamental reason is that an **EDF schedule has no graceful degradation**. If a task continues beyond its deadline, it can bring the whole system down by starvation, whereas a static priority scheme will still allow higher priority tasks to run. The highest priority task can be activated by a time-out mechanism so that it can terminate such a run-away task before the other, still well behaving tasks are starved. Hence, if EDF scheduling is used, it is better to restrict this to a maximum priority level within a standard priority based scheduling scheme. A similar observation will be made in the next section when discussing priority inheritance schemes.

A general remark must be made here. An RTOS in itself does not guarantee that all real-time requirements will be met. Designers must use **schedulability analysis** and other analyses like simulation and profiling to verify this before the application tasks are executed. However, an RTOS must provide the right support for executing the selected schedule. In general, this means a consequent scheduling policy based on priorities with pre-emption capability and with support for priority inheritance. VirtuosoNext provides this complemented with a runtime tracing function allowing profiling the temporal behaviour at runtime.

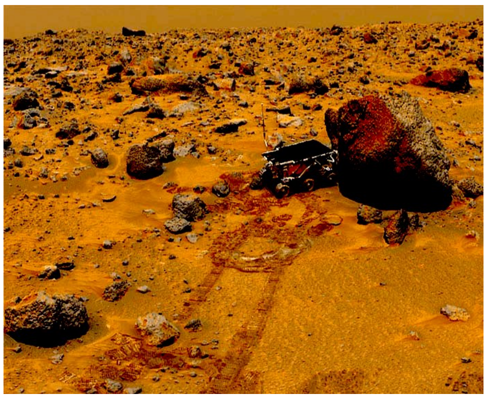
4.6. The issue of priority inversion and its inadequate solution

A major issue that has a serious impact on predictability is the presence of shared resources in an embedded system. A **shared resource** is often associated with a critical section or an access protocol. The latter are needed to assure that only one task at a time can modify the status of the shared resource. Examples are:

- A shared buffer that must be read out before new data is written into it,
- hardware status registers that set a peripheral in a specific state,
- a peripheral that can handle only one request at a time.

Note that a shared resource is a concept at a higher level of abstraction than the physical level but it will often be associated with it. It can be used to protect a critical section (e.g. the update of pointers in a datastructure) but it is not a critical section in itself. The critical section is a sequence of steps of the updating algorithm that must be done in an atomic way to guarantee that the datastructures remain coherent. It should also not be confused with disabling interrupts on a processor. The latter is a hardware mechanism that is processor specific and is designed to prevent other external interrupts from interfering with the intended program sequence.

In the context of a concurrent program, resource locking means that the system assigns temporarily **ownership of the resource** to a specific task until this task releases the resource. If more than one task requests to use the same resource, the second and subsequent requesting tasks cannot continue and will be blocked until the resource is released by its current owner. During the time a task owns a resource, it can become descheduled, e.g. because another higher priority task becomes active, the task requests a second resource, the peripheral associated with the resource is delayed itself or the task needs to synchronise with another task that has lower priority. In all cases, the resource owning tasks and other waiting tasks can be blocked from progressing which means that deadline violations become possible even if the priorities were correctly assigned and the application is schedulable with known blocking times. A very important conclusion to draw at this point is that a good design will try to limit the blocking times as much as possible and should avoid the need to protect the access to resources at all. This might require a change in the architecture of the system but from the reliability and safety point of view this is a cheap preventive measure.



The real issue comes in when we also analyse what can happen as a function of the assigned priorities. Assume a high priority task requests a resource that is owned by a low priority task. As it is a low priority task, middle priority tasks that are ready to run will pre-empt the lower priority task and if they have lengthy processing times, they will block the high priority task even if they don't need the resource at all. This problem is called the **priority inversion problem** and was made famous in 1977 when the Mars Pathfinder [12] kept resetting itself as a result of a continuously missed deadline, which was caused by a classical case of priority inversion as described above whereby priority inheritance was disabled by default in the RTOS.

Is there a cure for this problem (assuming that the system architect did his best in minimising the need for resource locking)? The answer is unfortunately no, but **the symptoms can be relieved**. The solution is actually very simple. When the system detects that a task with a higher priority than the one currently owning the resource is requesting it, it temporarily boosts the priority of the current owner task, so that it can proceed further. Priority inversion will be avoided. In practice different algorithms were tried out, but in general the only change made is that the boosting of the priority is limited to a certain application specific ceiling priority. Else, the scheduling order of other tasks requiring a different set of resources can be affected as well. Using the ceiling level, we can also guarantee that higher priority tasks (like monitor tasks) will run when activated and not being blocked by a lower priority task that was boosted.

If we analyse the issue of blocking in the context of a real system, we can see however that the priority inheritance algorithm does not fully solve the blocking issue. It relieves the symptoms by reducing the blocking times but a good design can maybe avoid them in the first place. The resource blocking issue is part of a more general issue. In essence, a concurrent real-time system is full of implicit resource requests. For example, if a high priority task is waiting to synchronise with a lower priority task, should the kernel also not boost its priority? To make it worse, if such a task is further dependent on other tasks and we would boost the priority can this not result in a snowball effect whereby task priorities are boosted for all tasks and of course, we would have no gain. Or assume that the task is waiting for a memory block while a lower priority task owns such a memory block. Or assume that a task acquires a resource, which makes it ready and is put on the ready list. But while it waits to be scheduled a higher priority task becomes ready first and requests the same resource, which means that the first task that was ready should be descheduled again and the resource given to the higher priority one.

While all these observations are correct, often such situations can be contained by a good architectural design. The major issue is that implementing this extra resource management functionality is not for free and the tests they require are executed every time, resulting in a non-negligible overhead. The conclusion is that in practice resource based protection must be avoided by design and that priority inheritance support is best limited to the traditional blocking situations. In the case of the implicit resource blocks, if they pose an issue to the application, they can be reduced to a classical priority inversion problem by associating a resource with the implicit resource. E.g. if a memory block is critical, associate a resource at the application level and normal support for priority inheritance will limit the blocking time. Else make sure that the system has additional memory blocks available from the beginning.

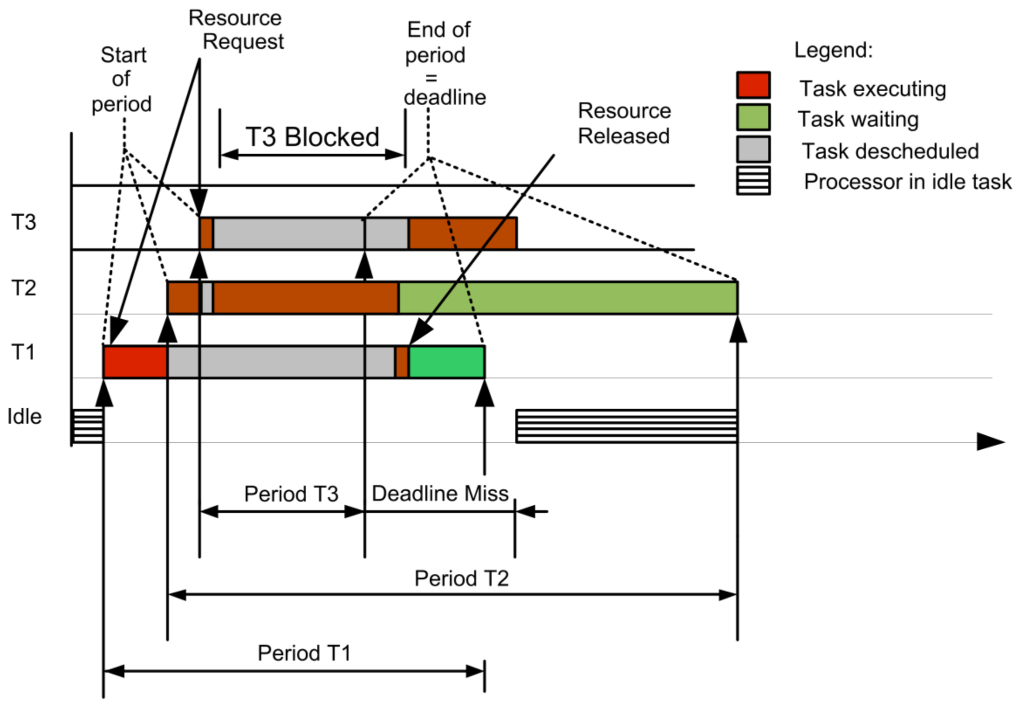
4.7. Distributed priority inheritance in VirtuosoNext

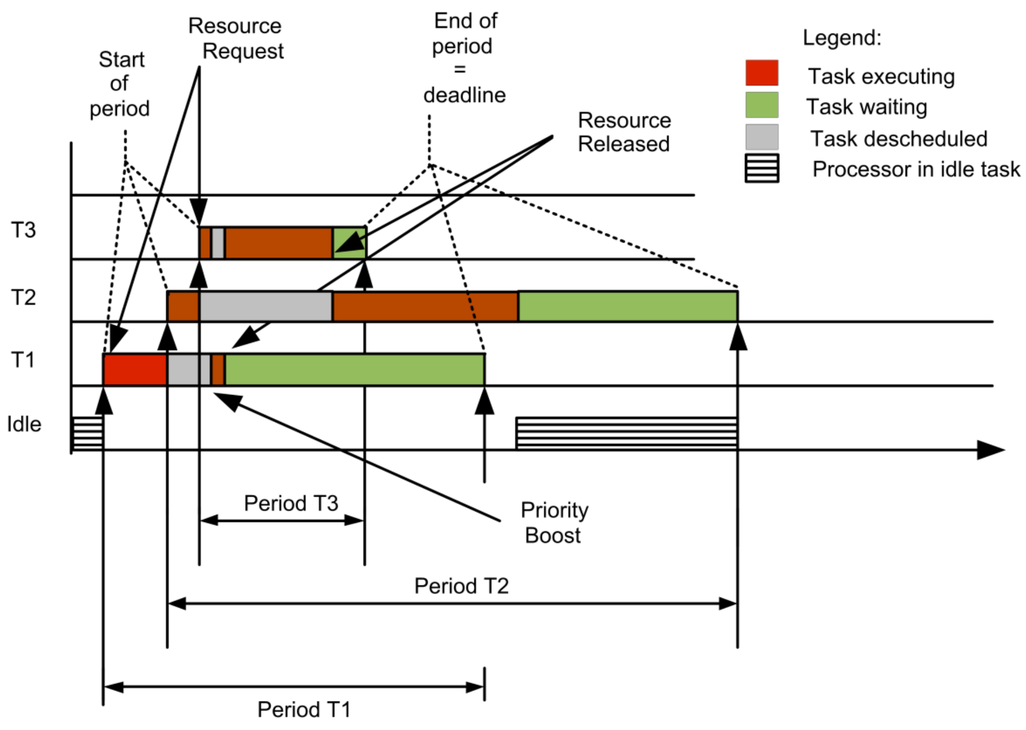
While implementing support for priority inheritance for a single processor RTOS is straightforward, implementing it in a distributed RTOS is more complicated because task states are distributed and change over time. On a single processor, the RTOS scheduler will examine the resource current owner when a task is requesting a resource. If the owner's priority is lower, then it will put the requesting task in the resource waiting list and boost the priority of the current owner. When the owner releases the resource, the RTOS scheduler will assign it to the highest priority task in the waiting list.

On a multi-processor system (single chip many/multicore, networked), on each node the kernel scheduler is managing the resources residing on its node. Requests for the resource can come from local tasks, remote tasks, the owner task can be residing on the same node or on another node, can be waiting on still another node or it can be in transit from one node to another. Hence, the local kernel scheduler must determine where the task resides at the moment of the resource request, send a priority boost request to the node where the task is residing and when the resource is released, lower the owner task's priority to its original priority.

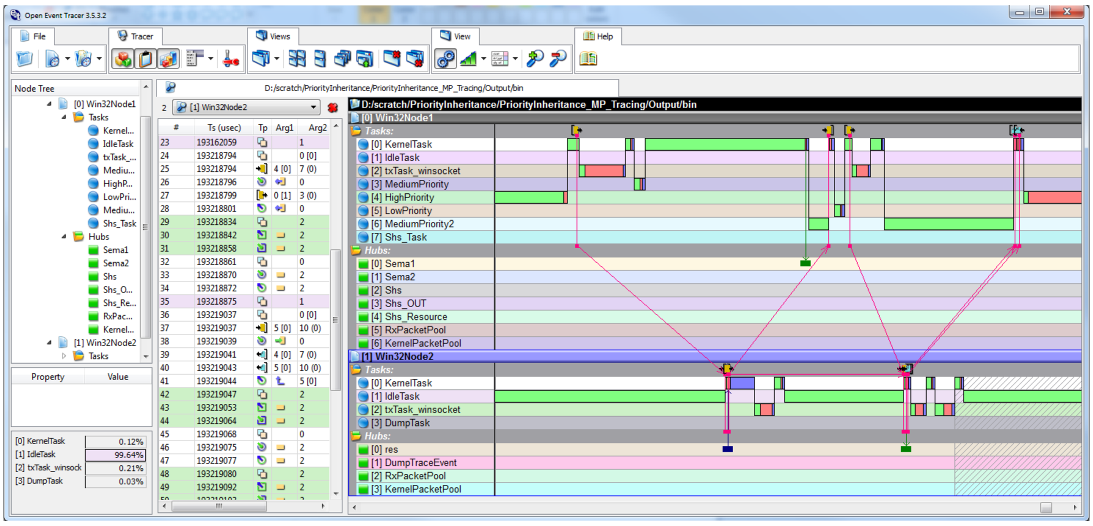
While this approach works well, the inherent communication delay of the inter-node communication can result in side-effects. For example the owner task might have issued a

Figure 5 Three tasks sharing a resource first withput and then with priority inheritance support





request that is forwarded to another node just before the boost request arrives. In practice this means that the **distributed priority inheritance** implementation is a **best effort** approach. The effects are mitigated if all tasks involved have a relative high priority, whereby the blocking is less problematic and if the network has relatively low communication delays. The boosting and reduction of the blocking time is the greatest when the owner task has a relatively low priority versus the requesting task. In that case, the prioritised communication layer will automatically assure that the priority boost request arrives first. This also means that such a priority boosting mechanism is most useful if the use of the resource is relatively long, i.e. longer than the transmission latency in the network.



#	Ts (usec)	Tp	Arg1	Arg2
23	193162059			1
24	193218794			0 [0]
25	193218794		4 [0]	7 (0)
26	193218796			0
27	193218799		0 [1]	3 (0)
28	193218801			0
29	193218834			2
30	193218842			2
31	193218858			2
32	193218861			0
33	193218870			2
34	193218872			2
35	193218875			1
36	193219037			0 [0]
37	193219037		5 [0]	10 (0)
38	193219039			0
39	193219041		4 [0]	7 (0)
40	193219043		5 [0]	10 (0)
41	193219044			5 [0]
42	193219047			2
43	193219053			2
44	193219064			2
45	193219068			0
46	193219075			2
47	193219077			2
48	193219080			2
49	193219092			2

[0] KernelTask	0.12%
[1] IdleTask	99.64%
[2] txTask_winsock	0.21%
[3] DumpTask	0.03%

Figure 6 Event trace of distributed priority inheritance in VirtuosoNext

4.8. Next generation requirements

In the first part of this paper, we have limited ourselves to the handling of real-time requirements. An unspoken assumption was that the system is fully defined at compile time. For most embedded applications this is the case. However, as applications are becoming more **dynamic and adaptive, the complexity increases** as well. In such applications, meeting stringent real-time requirements is still often a prime requirement but it is not sufficient. The real-time requirements will have to be met when multiple applications execute simultaneously with a variable amount of available resources. In the extreme, this also means in the presence of faults resulting in a number of resources no longer being available on a permanent or temporary basis.

We will illustrate this with two use cases for which the network-centric VirtuosoNext could provide the system level software.

The first use case is a **next generation electric vehicle**. Such a vehicle will be fully controlled by software and electronic components ("drive-by-wire") and likely have a distributed power and wheel control architecture whereby for each wheel traction control is combined with active suspension control, stability, anti-slip control and even braking. Many components can fail or show intermittent failures, e.g. sensors can fail, wires can break, connectors can give micro-cuts (very short absence of electrical contact due to vibrations), memory can become corrupted, processors can fail, etc. While the design should be robust enough to make such failures very low probability events, over the lifetime of the car such occurrences are certain. Practically speaking this means that while the system can be designed assuming that all resources are always available; the designer must provide additional operating modes that take into account that some resources are not available for meeting all requirements. In the simplest case this can mean that when one wheel controller fails, the processing is immediately redistributed over the three still fully functional units. Or this can mean that the system switches to a degraded mode of operation with a different set of tasks using less compute intensive algorithms.

The second use case is a **next generation mobile platform**. It is envisioned that such a platform will have tens of processing nodes, execute multiple application functions with some functions showing a variable processing load depending on the data being processed (typical for multimedia and image processing). In the worst case, the processing load can even surpass temporarily the available processing power. On the other hand such applications can often tolerate a few missed deadlines. However, such a mobile platform loaded with a dynamic set of tasks, poses additional constraints. E.g. when using wireless connections, bandwidth will vary over time, processing power might be variable because of voltage and frequency scaling techniques to minimise power consumption and available memory will vary depending on the use by other applications. Many of the processors used for such applications are so-called many or multicore chips are essentially chips with in silicon networks (NoCs) over which CPUs as well as high performance peripherals are connected. The NoC as well as the peripherals, the on-chip as well as off-chip memory are all resources that can be shared. In the Figure 6 such an advanced multicore chip supported by VirtuosoNext is shown. Newer versions also include a quad-core ARM processor.

What these two use cases illustrate is that an embedded real-time application is becoming more challenging for following reasons:

- Applications can **no longer** be **fully statically** defined.
- Some applications have a **variable processing** load.
- The system software must not only schedule processing time as a resource, but also **other system resources** like bandwidth, processing power, memory and even power usage.
- The system will have **hard real-time constraints** as well as **soft real-time constraints**.
- The system will have **different "modes"** (each consisting of a coherent set of states).
- **Fault tolerance** is not to be considered as an exception but as a case where the system has less resources available.

The result is that such an embedded system becomes "**layered**" and time as a resource is not the only one that must be scheduled. Such a system will need to schedule the use of several types of resources, although the final criterion remains **meeting the various real-time requirements**. In the guaranteed mode of operation we find back the traditional real-time scheduling. Rate Monotonic Scheduling provides for meeting the time properties whereas compile-time analysis assures that all other resources are available. In the extreme case this includes providing for fault tolerance because the system has to be designed with enough redundant resources to cope with major failures.

The next layer is then a **best-effort mode** in which the properties are guaranteed most of the time, eventually with degraded service levels. For the time properties this means we enter the domain of soft real-time, but often at the application level this means that the system offers a statistically defined level of quality of service level. A typical example is generating an image with less resolution because not enough processing power was available during the frame time. In the extreme case this corresponds with a fail-safe mode of operation whereby the quality of services is reduced to a minimum level that is still sufficient to stop the system in a safe way.

Finally, the last layer is one where essentially **nothing is guaranteed**. The system will only make resources available if there are any left. Statistically, this can still be most of the time unless a critical resource like power is starting to fail, and the system then was designed to put the processor in a "sleep" mode to e.g. stretch battery time.

What we witness here is a transition from a statically defined hard real-time system with fully predictable time behaviour, but possibly catastrophically failing, towards a system where the design goal is defined as a **statistical quality of service (QoS)** at the application level. Such a system must still be able to meet hard real-time constraints in a predictable way but must also offer different operating modes corresponding with a graceful degradation of the services offered by the system as a whole. Practically speaking, when a processor fails, it will often be catastrophically although processors with a MMU (Memory Management Unit) and appropriate system software can contain the failure to the erroneous task or process without affecting the rest of the application, unless there are dependencies. Most embedded processors however will need a hard reset to recover from such a fault. Hence, such a system will need redundancy of hardware resources, be it as part of a distributed system, be it as part of a multicore chip.

These next generation requirements were not addressed in the original VirtuosoNext project, but the fact that VirtuosoNext supports programming a multicore and distributed system in a transparent way facilitates addressing such requirements.

5. An approach for QoS resource scheduling

In this section we make an attempt at developing the QoS domains as requirements resulting in concrete functional and architectural support to enable managing the diversity of multi/ many-core on-chip resources.

5.1. Formalising Quality of Service (QoS) domains

If a system needs more resources in a worst case application scenario than available, does it mean that it is not a feasible system? As we have seen above, this needs not to be the case. The step to make is to assign the priorities not only in terms of meeting real-time constraints (as dictated by RMS) but also as a function of the criticality level of the task. In essence, we can distinguish three levels whereby we map the criticality level to a QoS level:

- **QoS-3**: Tasks that must run and never miss any deadline: this is the **hard real-time** domain.
- **QoS-2**: Tasks that must run but can miss a deadline if not too often: this is the **soft real-time** domain of best effort.
- **QoS-1**: Tasks that must run but only when resources are left over: this is the domain of **no guarantees**.

We can formalise this further:

QoS-1 is the level where is no guarantee that there will be resources to provide the service.

> This implies tasks with no strict real-time constraints and often convenience functions. It also applies to tasks where the output is more or less time-independent. If no update can be calculated, the previous output will be sufficient. This does not mean that for a service to remain usable, that a certain level of updating must be possible. A typical example application is a video phone with a bad connection. In the worst case, the user can switch off the video transmission to improve the audio quality. Hence a fault like resource exhaustion does not result in a fatal condition but mostly in a lower level of service provided. The limit case is the one whereby the quality has so much deteriorated that it becomes fully unusable. Of course, this should not happen more often than specified. This might be the case when the system has been underspecified from the very beginning.

QoS-2 is the level where the tasks must produce a result within a statistically acceptable interval.

> This means that the tasks have no hard real-time constraints but should still meet them most of the time, hence we can define quality attributes like probability of reaching the deadline within a time interval, probability of successive misses, etc. Hence a fault like resource exhaustion results in a statistically predictable failure rate. Upon a fault, the application must define what an acceptable behaviour is. Typical behaviour is: abort and drop the result, extrapolate from a previous result, etc. Hence the service degradation has been specified and must be met.

QoS-3 is the level whereby the system does not tolerate missing a deadline.

If such a fault occurs, all service can be lost. While the consequences are application specific, the application must be capable to capture the fault and prevent it from generating system errors, switch the system to a safe state or initiate actions to restart the system. This is typically the domain of safety, often requiring hardware support. We can distinguish two subdomains depending on the hardware architecture. If no redundancy is available, the system must be brought into a safe state after the fault happened. Hence, it is part of the system specification. If hardware redundancy is available, then the redundancy can be used to still provide a valid output. Hence, the system will have degraded but the service level will have been maintained. Of course, a subsequent fault can now be fatal; hence the safety assurance will now be lower.

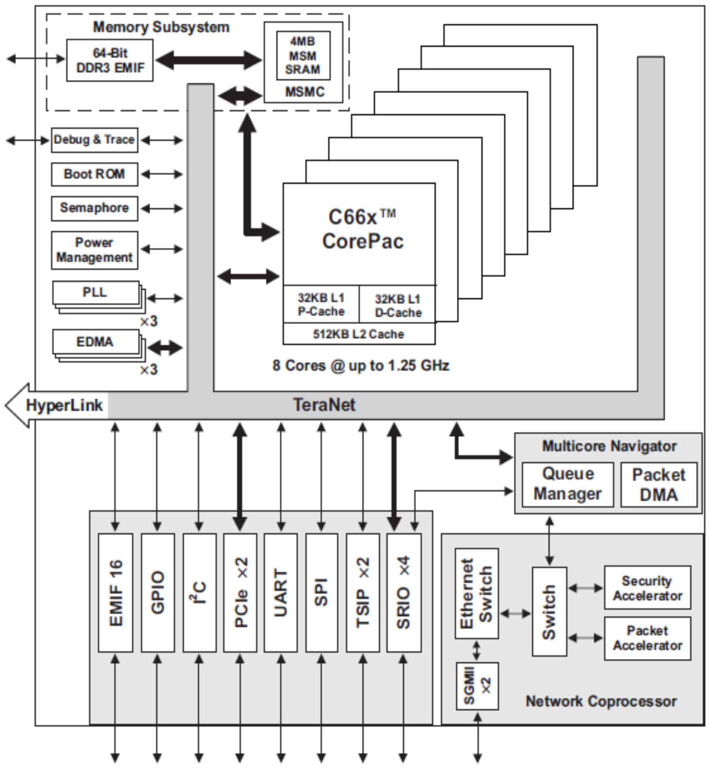


Figure 7 Advanced multicore chip: Texas Instruments C6678 DSP

Note that above classification is very generic and does not prescribe in detail how the system should handle the faults. This is often application specific. However it shows that in general a system can host several applications or functions with a different level of QoS. It also points to a different approach in safety design. Rather than making sure in a static way that an application has all the resources defined at build time, we only need to guarantee this for QoS3 level applications. Fault tolerance can be considered as its limit case. If the system has serious issues with resource exhaustion, then all resources should be assigned to meet QoS3 specifications. In the worst case, this means keeping the system alive as long as possible with minimal resources to prevent greater catastrophic failures.

This is in line with the concept of Rate Monotonic Scheduling whereby priority is used to assign automatically the CPU time to the highest priority tasks and whereby priority inheritance is used to unblock the resources as fast as possible so that higher priority task can use them.

5.2. Isolation for error propagation prevention

Given that we have different QoS levels, a clear requirement is that errors in one level must not result in errors in another, in particularly higher QoS level. But also inside each level, measures must be taken to prevent error propagation from one function to another. This should be pursued in a systematic way and requires several layers of defense.

At the programming level: **Error-free code**. The first objective to achieve is avoiding that the software itself generates errors that were introduced during its development. While extensive testing can uncover many of these, to increase the assurance formal modelling and verification is a must for safety critical applications.

Defensive programming: The second objective is to protect the software from runtime generated errors. This is mostly related to the numerical domains. Data values must remain in a valid range at the input, processing and output stage. While this is also a concern of developing error-free code, data can also receive wrong values due to hardware faults (e.g. corrupted bits due to external radiation of a power supply glitch). Several techniques can be used to mitigate the effects, ranging from plausibility checks, clamping the data to limit values, using redundancy or using coded programming. In general, this also means adopting a programming style that avoids dynamically changing code and data at runtime. Static code that verifies at compile time that all resources needed are available before the code is started, is certainly a good strategy for many safety critical embedded systems.

At the processor level: Current processor designs are still largely based on the von Neumann architecture whereby the ALU sequentially executes code thereby reading and writing data residing in a global address space. This in conflict in terms of error propagation with the RTOS programming model that can be seen as set of interacting functions, each having their own workspace (and hence called tasks in a RTOS). When a task is executing, it is written in the assumption that it has access to all on-chip resources and executes independently whereby the RTOS kernel isolates it from the lower level hardware details. We see here the emergence of virtualisation. To make this safe and secure, no task shall be allowed to modify code or data belonging to other tasks, except under protection of the kernel task using its services. On many micro-controllers there is no hardware support available, so only verified software can reduce the probability of this happening to a minimum. More advanced micro-controllers will have some form of memory protection (MPU) that allows restrict memory access to specified regions (often with a granularity of a few Kbytes) and will have a user as well as supervisor mode, whereby in user mode certain operations are prohibited. High-end processors will have Memory Management support (MMU) whereby the MMU helps in executing code in a virtual linear address space and helps to isolate user applications from each other. MMUs are however complex and resource intensive, whereby the granularity is fairly large tens of Kbytes). Latest developments have added so-called hypervisor support. Hereby the processor I/O space is virtualised, reducing the probability of corruption by competing processes. All these hardware techniques assist the software layer, but increase the complexity and decrease the hard-real time capability, as e.g. extra latencies are introduced.

At the **processor's architectural level**: A simple and straightforward strategy is to develop the processing hardware in such a way that each application has its own dedicated CPU core. This technique is not new as it allows using dedicated and optimised CPU cores for the application at hand. In addition, if each core has its own local memory, often the clock frequency can be reduced, which is beneficial in terms of energy consumption. However the designer is here confronted with physical constraints. First of all, memory now becomes the critical resource. In additional memory technology has followed CPU clocks in size but not in speed. Secondly, in the end processors are I/O bound and there is only a maximum of pins that can be put on a chip package. Hence, memory as well as I/O devices have become shared resources, even in the case of redundancy.

At the **system's architectural level**: From above, one can see that to reach the highest level of QoS, dedicated hardware is the most trustworthy solution. In the ideal case, we have different chips for each application. While this is no longer very costly, it moves partly the problem to the PCB domain where the probability of failures due to mechanical and chemical stress is higher than in the chip package. However, it is often the only way to mitigate common mode failures (example: power supply issues) and with an adequate programming model, it allows heterogeneous redundancy.

To conclude we can see that the various defense mechanisms are intertwined and a good trade-off decision will depend as well on the application as on the available hardware. However, two factors dominate. The first one is that an adequate programming model is a precondition. The second one is that simple hardware error detection and protection mechanisms can be very beneficial. In all cases is the key challenge to share the available resources in the best possible way

5.3. The trade-offs involved when selecting the resource quantum

When resources are available, a mechanism must be provided to share them amongst the competing application functions. The simplest way is to associate a logical resource lock (managed by the RTOS kernel) with each physical resource. The question is to know how much of the global resources like bandwidth, memory or energy should be allocated. We call such a resource part a **resource quantum**. The trade-offs to be made are multiple:

- The quantum must be large enough to offset the overhead associated with allocating a quantum.
- The quantum should be small enough to avoid starvation for other application functions.

We can illustrate this with the case of a shared communication link between two or more nodes. Assuming we transmit and receive a communication unit of N bytes at a time, following parameters are of importance:

- The **set-up time** of a communication.
- The **transmit time** of a communication.

- The **maximum bandwidth** of the communication medium.
- The **communication overhead** per transmitted byte (e.g. due to headers and extra control bytes).
- The arbitration overhead and communication scheduling delay.
- The **reception time** of a communication.

If we assume that such a communication unit consists of a packet, itself composed of a header and payload, then the set-up times will be equal for all packets but the transmission time depends on the payload. The smaller the packets, the more packets we can send per unit of time, but the lower the bandwidth at the application level. Moreover the larger the packet, the longer the blocking time. A similar situation can be found when using time-slicing. Minimum time-slices are needed to keep the overhead acceptable, but longer time slices will reduce the responsiveness of the system. The issue is that the quantum size will be application and system specific, hence no optimum solution can be found beforehand. The solution is to be found in an iterative approach whereby first approximations and feedback from runtime profiling is used to tune the quantum size until a better value is obtained. Note however that these values also depend on the other applications being executed and hence optimal values can fluctuate at runtime.

In an experimental set-up using the Intel experimental 48-core SCC chip, the communication latency and bandwidth were measured using VirtuosoNext. The results were compared with the Texas Instruments 8-core C6678 DSP. While both processors target different applications, it is clear that the additional wait states and shared communication infrastructure are the root cause of an important communication bottle neck. This is in particular true for the Intel chip whereas the C6678 has much better support for on- and off-chip data moving (using separate busses, DMA engines and local caches that can be locked to act like zero wait state SRAM. The interested reader can consult the paper [14]

5.4. Maintaining maximum QoS by graceful degradation and recovery

The next problem to tackle is to define scheduling strategies that allow to keep a maximum of **QoS when faults occur** whereby resources become depleted. This is complex because the decisions must be swiftly taken, often based on incomplete information. We can define following rules (amongst others):

- Applications with QoS level N have priority over applications with a QoS level lower than N.
- QoS-3 requires redundancy.
- QoS-2 must have an abort mechanism for safely releasing resources.
- If the fault is intermittent, then recovery can be attempted.

In this scheme, the scheduler must be guaranteed to always have enough resources to exercise control over the applications. If not, a clearing of the faulty state and reinitialisation of the failing system unit is often the only option. Note that this scheme also generates a requirement for the inter-node interactions. They must exhibit the same QoS level as the highest level needed for the RTOS scheduler.

6. Hard real-time and caching on advanced multi-core chips

6.1. Effects of caching on predictable timings

A side-effect of the very tight integration of components on a single chip is that scheduling becomes **increasingly less predictable**. This is largely due to the mismatch between the CPU clock speed, the speed of the external memory and the arbitrating logic that manages the peripherals. The total memory often needs to be large as available processing speed often grows in line with the code size and the data to be processed. The industry has adopted two main approaches to tackle this issue. The most obvious one is to allow more computational concurrency so that the scheduler can switch to another context while the communication happens. In the ideal case this requires the use of DMA engines and a fast context switching support. The most often used technique is to use small (by technological necessity) but fast internal caches. When code and data are in the cache close to maximum performance is obtained, but as caches are limited in size, this is not guaranteed all the time. When code and data are not in the cache, deviations of a factor 20 to 100 are not uncommon.

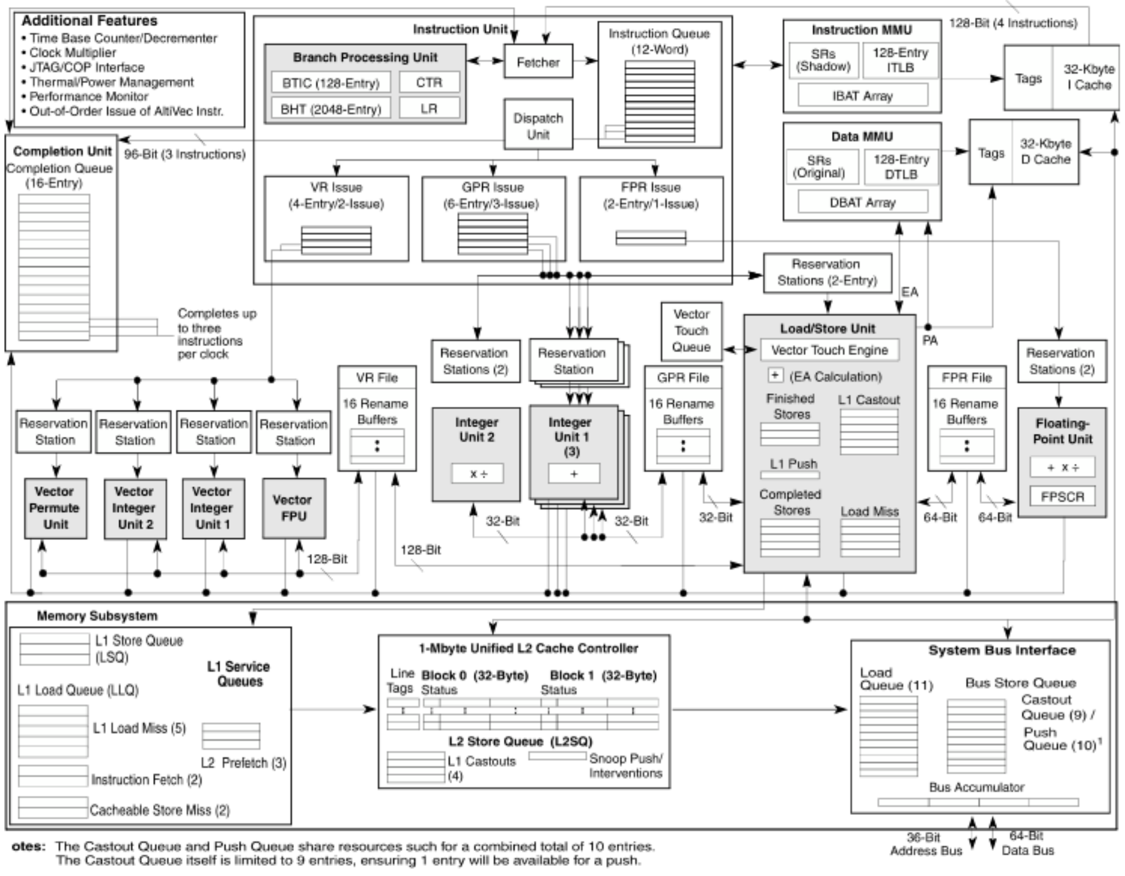


Figure 8 Freescale PowerPC 744X block diagram

The following graph shows interrupt latency measurements on a 1 GHz PowerPC (e600 family), whereby for the sake of the measurements the cache is periodically flushed. The measurements were taken by setting a hardware timer that periodically interrupts the processor with stress loading the CPU using a semaphore loop (which has very low processing load but invokes constantly context switches). The time taken to read the timer value in the

ISR, subsequently in a waiting task is recorded. This time interval is called the interrupt latency. Four cases were measured and plotted (logarithmic scale):

- All caching disabled.
- L1 Data cache and L2 cache enabled.
- L2 and L1 data and program cache enabled.
- L2 and L1 cache enabled, but the caches are periodically flushed.

The first observation to make is that the enabling of the caches has a serious impact on the performance. When all caches are enabled, the performance gain is about a factor of 50. This is mainly due to the access speed to the external memory. On the other hand, the cache flushing has less impact than one would expect. This is due to the architectural implementation of the cache logic whereby a complete cashline (32 bytes) is cashed whenever non-cached data or code is accessed. Hence, this mechanism mitigates the effect although we still see that interrupt latencies jump with a factor 5 (but way below the values with no caching).

This was confirmed by executing a few measurements using the on-chip performance monitor unit. This unit allows measuring the time more accurately and also records instruction and data caches misses.

- Ten data accesses (read operations) from consecutive memory locations take 122 instructions executed in 167 CPU cycles when caches are enabled.
- Ten data accesses from non-consecutive locations takes 140 instructions executed in 8959 cycles when both caches are disabled.
- Ten data accesses from non-consecutive locations takes 137 instructions executed in 1234 cycles when all caches are fully enabled.

We noticed that even the hardware based performance monitor recorded erratic results for the

Figure 9 Interrupt latency to ISR and task level on a 1 GHz PowerPC (logarithmic scale)

number of instructions but also for e.g. instruction misses, while the code easily fitted in L1 cache and no other software was loaded. This statistical spread would be worse when multiple cores share the same memory and multiple peripheral devices are active. This spread cannot be mitigated unless one changes the hardware architecture. Hence a more statistical approach to QoS scheduling is needed as well.

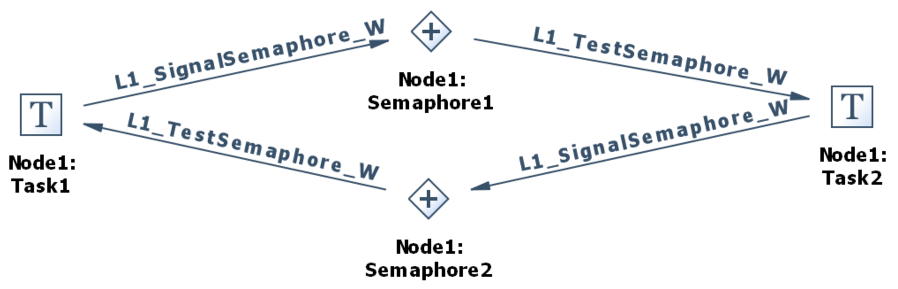


Figure 9 Semaphore example used for benchmarking

Next a measurement was done using task interaction using RTOS kernel services. The test program is like in Figure 10 whereby each test used a different type of VirtuosoNext hub (port, event, semaphore, fifo), once with a 1024 bytes datatransfer and once with no data transfer. The timings are in microseconds and averaged over 1000 loops with 2 tasks using the service. Hence, each loop consists of 4 task switches and 4 task-to kernel interactions.

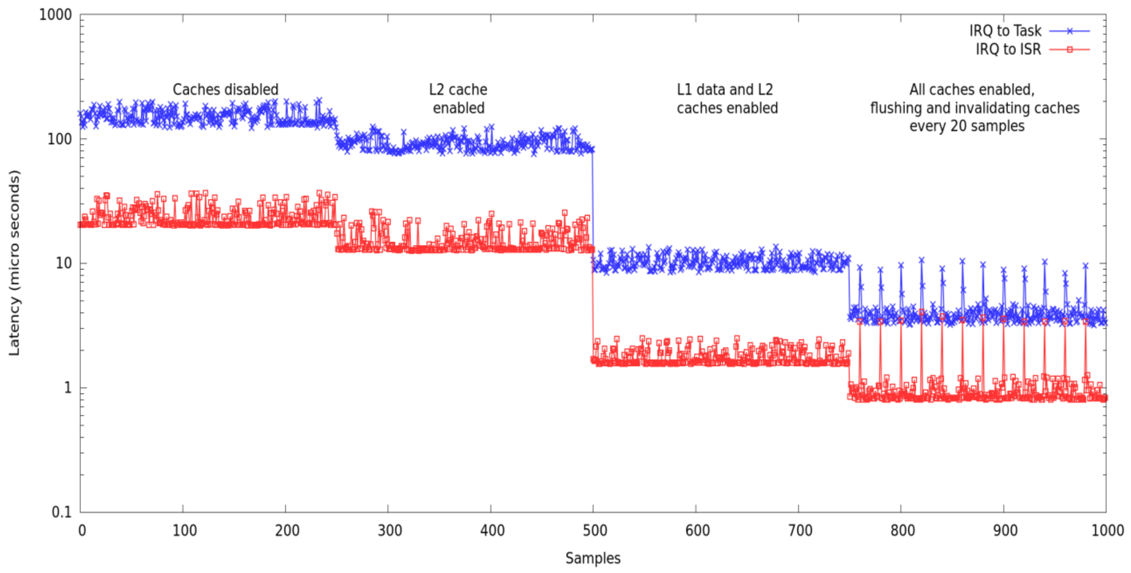
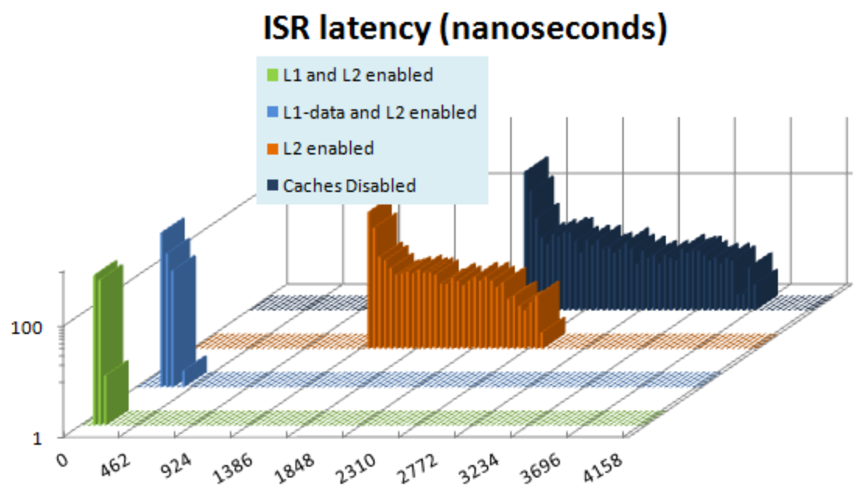


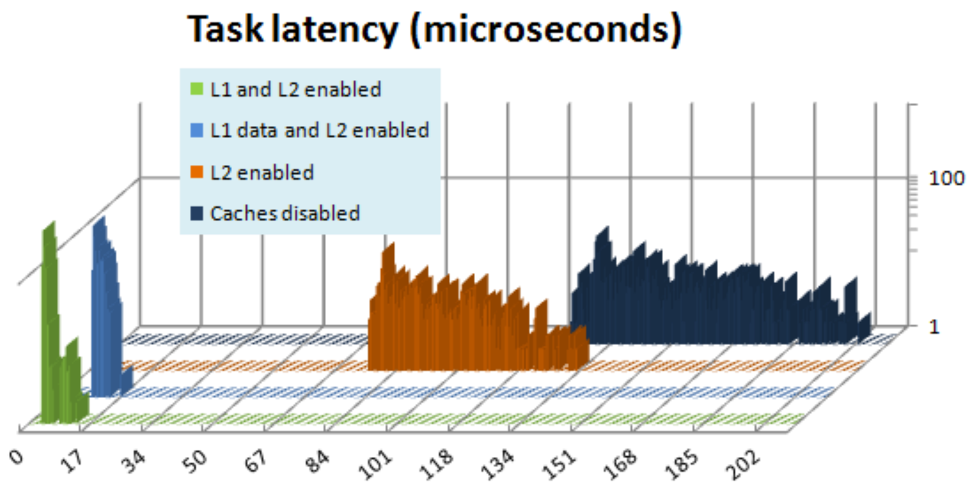
Table 1 Loop times for different caches setting and hub services (in microseconds)

Data size	**Interaction type**	**Caches disabled**	**L2 enabled**	**L1 data cache and L2 enabled**	**L1 data and program and L2 enabled**
1024	Port	1333	341	73.5	10.9
	Event	216	102	15.0	3.97
	Semaphore	216	102	15.1	3.97
	Fifo	1844	551	103	14.4
0	Port	215	103	15.0	3.98
	Event	213	101	14.9	3.97
	Semaphore	213	101	14.9	3.95
	Fifo	216	103	15.2	4.07

What we can notice is that the impact of code or data not being in cache is quite dramatic for the temporal behaviour. The difference ranges from a factor of about 30 to about 130. This is not only the impact of the slow external memory (133 MHz SDRAM) but also due to the presence of an on-board controller that arbitrates in a round-robin fashion between the external peripherals, the processor and the memory. The latter is an important observation. Worst Case Execution Times are often determined by using a detailed simulation model of the processor. While the CPUs themselves are becoming very complex, using every trick to provide higher peak performance, the integration of peripherals on the chip make that almost impossible, partly because the details of the design are not known. In addition, the processor (or rather the chip) will be integrated with other chips on a board, often arbitrating between the processor chip, on-board external memory and external peripherals at a slower clock speed than the internal clock of the processor chip. Often this will include undocumented firmware (e.g. in ASICs or FPGA). Hence, not only are worst case timings now difficult to predict, they also are board and application specific. In the measurements above, worst case timings for task latency were obtained of more than 200 microseconds (no caching) versus 4 microseconds (all caching enabled). These 200 microseconds were based on 1000 samples and is likely still below



the real worst case timing if a prolonged test would be executed in the context of a real



application. To mitigate this risk, it is clear that a static design is no longer adequate and more

Figure 11 Interrupt to ISR latency

statistical approach must be taken.

As a summary, the ISR and task interrupt latencies were plotted on single graphs with the different cache modes enabled. Note that the vertical axes are logarithmically scaled.

These graphs clearly show that the caches reduce the absolute interrupt latencies, but also the statistical spread, although the first sample measurement is always 5 to 10 times longer than the subsequent measurements (as we noticed in the time series when caches are flushed).

Figure 12 Interupt to task latency

6.2. QoS and ARRL

While we expressed QoS as an application level property, it was often linked with the capability to meet real-time deadlines, often associated with the safety properties of a system. These safety properties are expressed as **SIL levels (Safety Integrity Levels)** [13]. These SIL levels express that the system design has drastically reduced the probability of residual errors resulting in potential hazards (whereby people can become hurt of killed). In this sense, QoS is a broader concept that encompasses safety issues besides other issues like security and availability of service. As we outlined, reaching a certain QoS level requires an approach on several fronts, often at the functional or development process level. Hence, it makes sense to develop a classification that gives us the requirements that must be met to reach a certain QoS level. We called this the **Assured Reliability and Resilience Levels or ARRL** for short. There are defined as follows:

re·li·abil·i·ty

Definition of RELIABILITY

1: the quality or state of being reliable

2: the extent to which an experiment, test, or measuring procedure yields the same results on repeated trials

- **ARRL-0**: Nothing is guaranteed (”**use as is**”).
- **ARRL-1**: The functionality is **guaranteed as far as it was tested**. This leaves the untested cases as a potential domain of errors.
- **ARRL-2**: The functionality is guaranteed in all cases **as far as no fault occurs**. This requires **formal evidence** covering all system states.
- **ARRL-3**: The functionality is **fail-safe** (errors are not propagated) or switches to a reduced operational mode upon a fault. The **fault behavior is predictable as well as the next state after the fault**. This means that fault modes are part of the initial design specifications. This requires fault detection mechanisms as well monitoring so that errors are contained and the system can be brought into a controlled state again.
- **ARRL-4**: If a major fault occurs, the **functionality is maintained** and the system is degraded to the ARRL-3 level. Transient faults are masked out. This requires **redundancy**, e.g. TMR (Triple Modular Redundancy).
- **ARRL-5**: To cope with **residual common mode failures**, the TMR is implemented using heterogeneous redundancy.

re·sil·ience

Definition of RESILIENCE

1: the capability of a strained body to recover its size and shape after deformation caused especially by compressive stress

2: an ability to recover from or adjust easily to

When comparing with the QoS levels, one can see that a component or system must meet minimum levels of ARRL to enable a minimum QoS level whereby we consider resource exhaustion as a fault.

- **QoS-1 requires a minimum of ARRL-2**
- **QoS-2 requires a minimum of ARRL-3**
- **QoS-3 requires a minimum of ARRL-4.**

Hence, the ARRL levels allow us to define rules and requirements that components must be met in order to be usable for meeting a specified QoS level.

The implications:

Current single chip designs have shared resources, hence only ARRL-3 can be reached, whereas to reach ARRL-4 and ARRL-5, each processor must have a dedicated set of resources (CPU, memory, power, ...). ARRL-4 or ARRL-5 level must also be reached for the inter node communication mechanism.

safety-
critical system definition

A computer, electronic or electromechanical system whose failure may cause injury or death to human beings

E.g. an aircraft or nuclear power station.

Common tools used in the design of safety critical systems are redundancy and formal methods.

7. Partitioning and separation of concerns for embedded real-time

Many embedded applications, specifically safety-critical ones, have strict real-time constraints. In the very worst case, missing a deadline can be catastrophic. Therefore, many approaches have been developed and successfully deployed whereby time is explicitly used to schedule the application tasks. A very important design parameter is a guaranteed Worst Case Execution Time (WCET). While this approach can be justified partly for historical reasons but also for reasons of simplicity, modern many-core processors pose a significant challenge as the chips combine multiple tightly coupled processing cores, fast caches to alleviate slow memory and complex peripherals. All these elements result in a statistical execution behavior whereby a measure like WCET is no longer practical. In this paper we advocate that this situation requires a different approach to programming, i.e. one based on events and concurrency with time no longer being a strict design parameter but rather a consequence of the program execution. It is a consequence of applying a separation of concerns to execution in space and time. Benchmarks obtained with the latest version of VirtuosoNext Designer, a fine-grain partitioning multi-core RTOS, illustrate that this is not only feasible but also with no compromise on the real-time behavior.

7.1.What does``real-time'' mean?

Similarly to the term ``priority'' the term ``real-time'' has for decades been extensively used in the embedded domain, yet it is still often a misunderstood one. The rigorous definition is the one of "hard real-time", i.e. the guarantee that no specified deadline will ever be missed. In contrast, ``soft real-time'' is meant to indicate that it is sufficient to meet the deadlines most of the time, with occasional misses being acceptable (which implies that burst misses are not acceptable).

This definition is already an indication that meeting a deadline is not only a matter of executing the critical code before the deadline is reached. There can be multiple reasons why this can fail. Programming errors, runtime errors, but also the hardware behavior influences the execution of the critical code. What's important is that the execution is predictable and trustworthy. For the sake of argument, let's assume that the code has been proven correct, i.e. formally verified. This leaves only the hardware or the environment as sources of deviating behavior in time.

Coming back to the requirement of correctness, in general the programming logic will be time independent. The same program code can run on any processor being clocked at any speed. Time enters as a design constraint.

7.2.What does ``priority'' mean?

While simple embedded applications processing an input to generate a single output can often be programmed as a simple loop, scaling it up to a so-called superloop to handle multiple events with multiple outputs, often executing at different rates, is not only tricky, it is not very resilient. If one of the subloops has an issue, the whole application is jeopardized.

Time based scheduling improves on the superloop approach by having a programmable time base that triggers the execution of the different functions. This approach can be very predictable and is relatively easy to verify, but it is not very flexible as any change can have an impact on the complete time schedule.

Flexibility and modularity was introduced by using a more dynamic approach , e.g. by introducing cooperative (often round-robin or time-sliced) scheduling. The best approach however is priority based preemptive scheduling. Using Rate or Deadline Monotonic Analysis [7, 11, 15] one can prove that no deadlines will be missed, provided a maximum CPU workload is respected (typically about 70 percent, but it can be higher), if all tasks are independent and if they are scheduled in order of priority with the priority being assigned as a function of the execution frequency. This is the classical definition of priority based scheduling in the context of embedded real-time. Of course, in practice tasks are seldom independent (as they exchange information) but it works very well in practice. This being said, some people associate priority with ``urgency'' (as in a ``high priority message'') but it should be clear that this intuitive notion entails manually changing the execution schedule and hence, it can have undesired side-effects. An important aspect of priority based scheduling is that it decouples the processing logic from the processors it executes on. If the priorities are correctly assigned and if the maximum CPU load is observed as a constraint, then no deadline will be missed as a consequence.

7.3.Real-time and priority on modern multi-core SoCs

Classical time based scheduling is based on knowing the WCET. As no execution time can be longer than the WCET, a schedule can be calculated. Hence, the question is how to obtain the WCET. Traditionally, one can simply execute the program in a loop and measure it. The issue is that this is like testing. How do we know we measured the real WCET or was it a measurement that is still below the WCET? Another approach is to use a very detailed simulation model of the hardware. The question remains essentially the same. Is it the real WCET or the best estimate of it?

On modern SoCs, this is practically elusive. Modern SoCs can have 1000's of interrupt sources, I/O logic of various complexity, can have multiple shared buses, DMA engines, multiple memory banks and multiple processing cores. The best performance is obtained by using the caches (who's behavior is very hard to model) but as they are small, a cache miss is not unlikely whereas external memory often will be a factor 10 or more slower than the CPU. As a result, WCET measurements are always a worst case estimate and unfortunately, they can easily be a factor 100 worse than the average execution times. In addition, same changes to the code will result in small changes to the execution time. While safety critical systems designs freeze the code once it has been approved, it also means that a single change can result in expensive re-validation and testing, even if the change is small and it effects are well isolated. This makes using WCETs impractical, at least on GHz modern processors. Paradoxically, hard real-time is much easier on much slower MHz microcontrollers.

From above, the reader should have become aware that dynamic scheduling was introduced to handle the complexity of handling multiple events with potentially very different timing constraints. By introducing tasks as independent execution units (basically, a function with a private workspace), we have raised the level of abstraction. By using priority to determine the order of execution at runtime, we have decoupled the logical behavior from the timing properties of the underlying hardware. The resulting execution profile is an event-driven partial order in time, not a strict order in time. In practice, tasks do not execute independently and therefore RTOSes are used not only to provide priority based scheduling but also to provide the synchronization and communication services. This also means that the RTOS itself must be designed so that the real-time behavior is always guaranteed.

7.4.When is an RTOS really real-time?

In order to provide its services to the application tasks in a way that meets real-time requirements, a RTOS kernel typically has to implement the following functions:

- A context switch. On modern processors the context can be extensive (100's of registers not just covering the CPU registers but also a myriad of status registers of on-chip resources such as the MMU). The larger it is, the higher the latency to switch from one task to another.
- Interrupt handling. On modern processors, interrupt handlers can be very complex with potentially tens of interrupts arriving quasi simultaneously. Handling in order of priority is possible if the hardware has provisions. The need to disable interrupts in critical code sections has an impact on the system latency.
- Implementing memory management and protection. This can considerably impact on the context switch times.
- Handling caches. Especially when dealing with peripherals, cache flushing might be needed. This can considerably increase the execution time.
- RTOS Kernel services. These services allow tasks to synchronize and to communicate. To safeguard the real-time behavior, this should happen in order of priority, so that no higher priority task is kept waiting by lower priority tasks. All kernel operations must be strictly bounded in execution time and being independent of e.g. the size of the system. This is the area where most general purpose OS and even some quasi-RTOS fail.
- I/O handling. With many smart peripherals integrated on the chip (often as black boxes), the timing behavior is often not very predictable.
- Interprocessor communication. As tasks are distributed over multiple processors (on- or off-chip), the communication can introduce a serious latency. Any delay on one processing node can delay the execution of a task on another node.
- Respect the inherent priority levels of a real-time application, starting from the hardware (being clocked synchronously, it can never be made to wait). In order of priority (hence in time), interrupts must be handled first and interrupt servicing must be kept as short as possible. The kernel task and its scheduler as well as the driver tasks are the next priority level because any delay at this level, will result in a later execution of the application tasks. And finally, the application tasks. These levels of priority, often go hand in hand with typical time constraints, sub-microsecond for the interrupt handling, microseconds for the kernel and driver tasks, tens of microseconds to several hundred of milliseconds for the application tasks.

Many of the functions above entail that the RTOS kernel keeps track of waiting lists. A real real-time RTOS kernel will guarantee that this waiting behavior is strictly in order of priority and independent of e.g. the number of tasks in the system. This also applies to the communication that often has to share the communication medium, hence access in order of priority must be guaranteed. As a communication medium is also a resource that must be shared, packetization is a must to limit the blocking time.

7.5.Resource blocking, Priority Inheritance and fairness policies

A major issue that haunts real-time applications is the sharing of resources or the presence of long critical sections (that only allow one task at a time). Above section has introduced some of them (like e.g. a context switch) but as the blocking is relatively short, these are generally not considered as critical. Resource locking is typically needed when synchronising with a slower peripheral or to ensure data integrity. The issue is widely described in the literature and was made famous by the Mars Rover reset issue [16]. There is no real solution to it but generally speaking it requires a careful design of the application by minimizing the need for resource locking and by the availability of priority inheritance with a ceiling priority support in the kernel scheduler. This reduces the impact of the blocking to a minimum but as said, it remains a best effort strategy.

One should also note that modern multi-core processors exhibit ``milder'' forms of resource locking or at least they have functional features that increase the latency. Examples are shared memory (often with multiple wait states), the resulting cache coherency and cash flushing operations but also the multitude of interrupt sources that in the end are all pipelined to a single interrupt to the CPU. And last but not least, as processing chips became faster and memory cheaper, the applications became more complex with more elaborate algorithms. This can result in so-called ``greedy'' tasks executing converging algorithms that can block other tasks. To reduce this issue, such task must be run at lower priority but with time slicing/round-robin scheduling as a ``fairness'' policy to avoid blocking other tasks.

7.6.Complexity dictates orthogonality, for logic and for time}

Above sections should have made it clear that advanced multi-core processors enable more advanced processing intensive applications, but also result in a large increase in complexity and stochastic execution. The latter is practically incompatible with keeping track of the timing behavior in the application, e.g. by trying to find a static schedule. The result is a need to decouple the various elements. Complexity of the system state machine is reduced by splitting it up in smaller execution units (hence introducing a concurrent programming model) and by taking out the time dependencies (by using rate monotonic scheduling based on priorities). In other words, the application logic must be decoupled from the time behavior. How do we then guarantee deadlines? Partly, by a feasibility analysis based on average timings (e.g. obtained on a simulator) and by observing the time behavior from outside the application (read: testing). What are the pre-conditions: the time-independent logical behavior has to be correct and the execution profile in time must have a narrow enough spread in time.

7.6.1.Can hard-real time be relaxed without becoming too soft?

While some real-time applications, for example high speed digital signal processing or high speed control might not tolerate missing deadlines at all, many real-world applications can easily tolerate missing a deadline from time to time.

When processing signal data Nyquist's theorema applies and a good design will over-sample. Missing a sample or reading it a bit later or sooner mainly introduces some numerical noise that is later on filtered out. The result is not a failure but a lower quality level. The same applies to applying control signals. This is best illustrated by considering for example a braking system. Does it matter if the brake controller applies the actuator command a few microseconds too late or too soon if the time constant of the controlled system (due to the mass inertia) is measured in milliseconds? It would only matter if the actuator command went missing for a few time intervals greater than the time constants of the controlled system. Missing a single deadline makes the system software soft real-time for a single time period. Missing it in consecutive bursts however can be considered a failure. Most likely, not because the timings were ill defined or because of schedule errors but because a deeper logical error manifested itself, or the hardware had a temporary hick-up.

7.6.2.If you can't change the hardware, adapt the programming model}

If execution times on advanced SoCs become stochastic, what can we do? We adapt the programming model. Firstly, execution is not triggered by a precisely timed event, but by the event of its arrival, which can be sooner or later than the nominal scheduled time of the event. This is also consistent with a hardware reality whereby some jitter is always present. What can we do if that is not sufficient for the application requirements? As the deadlines are present in the I/O domain, one can still adapt the peripheral circuits. These can be very precisely timed because the logic is simple. One can over-sample, buffer the data and read it out triggered by a local precise clock. This can be done with a precision of nanoseconds.

Secondly, connect the events to their corresponding processing functions. Shrink the global state machine, by decoupling events and processing functions resulting in smaller state machines. They only share state information at clearly programmed interactions, reducing the risk of spillover of state from one state machine to another. The result is a typical concurrent programming model with tasks (sometimes mislabeled as threads or processes). These tasks are written as much as possible in a time independent way. Hence, when written in a portable programming language, they can execute on any target at any rate. Their execution in time depends on the triggering of events (typically: interrupts, kernel synchronization and communication services). Each task however should be computationally stable and logically correct. Failing time properties are not the result of a failure in the scheduling model, but of a logical failure of the code (numerical properties or state machine related properties).

Thirdly, apply priority based scheduling. The critical reader will object that Rate Monotonic Scheduling is not a guarantee because of the simplified assumptions in the theory. This is correct. But it provides a very good first starting point. Profiling, applying a sufficient CPU load and time margins as well as specifying ``safety'' functions as part of the application are most often sufficient to handle the rare cases. If that is not sufficient (because the safety requirements are very high), then the only remedy is to simplify the design itself, eventually by splitting it over more than one processing node, reducing possible interference and complexity.

Fourthly, verify the programming logic. Most time related failing of applications are not directly related to time, but to how time is represented in the software or by the code itself being erroneous. A typical issue is that time is represented in the hardware as a discrete value limited integer. In the program it becomes a typed number that can be manipulated, resulting in e.g. rounding errors, underflow or overflow, etc. Decisions in the programming logic can hence be based on erroneous values, resulting in a failing program, especially with time-outs in unforeseen circumstances.

Note that this model does not consider time as a scheduling parameter but as an external measure (by using e.g. a clock). The time behavior is the result of the programming logic being executed on a given (stochastic) hardware. One can record its progress in time by observing the external clock but the clock itself does not change the execution logic.

Note that this programming model is not new. It is largely what all RTOS put forward. It was formally formulated in Process Algebra's like Hoare's CSP [17], later extended to support timing behavior but this was not really successful. We also note that Leslie Lamport wrote about declaring that time doesn't need a special treatment in programs [18,19, 20]. It is sufficient to declare a global variable representing it.

It should be noted that some years ago, asynchronous logic was developed that exhibited this approach even in hardware. This allows for example to change the processing chip supply voltage at runtime, resulting in the processor running slower or faster but not failing, without any change to the program code. While this technology failed on the market (mainly by lack of competitive development tools), it was very promising for resilient processing.

7.6.3. Implementation in VirtuosoNext

The guidelines outlined in previous sections have been applied in the real-time and concurrent programming model of VirtuosoNext Designer. A front end graphical modeling tool is used to specify the multi-processing hardware topology as well as the concurrent application tasks and interaction entities (called ``hubs''). The latter can be moved in the system by remapping them to a different processing node. Important is that from the high level description, code generators and parsers take care of generating all bring-up code as well as all system data structures. As this is a compile time operation, the compiled code becomes a static image in memory. The benefit is a smaller code size (from a few kBytes to about 35 kBytes for the most complex version) as well as a reduction of the error-prone programming as the developer only has to add the application specific code.

The architecture of the RTOS kernel itself has a long history. The prioritised packet switching architecture was one of its original concepts as it allowed transparent programming of parallel processing targets. The latest versions of the kernel were redeveloped using formal techniques, which resulted in a drastic code size reduction (a factor 5 to 10 depending on the target) and the introduction of the generic Hub interaction entity. The hub concept presents itself as a classical service to the programmer (semaphores, fifos, etc.) but also allows to add very specific services like a proxy hub for using dedicated on-chip logical units. [1]

While the static memory model is beneficial for performance, better safety and security protection is possible by exploiting advanced protection mechanisms like MMUs and MPUs, as available on modern processors. On target processors that allow it, this has resulted in a specific version of the RTOS kernel that isolates each task as well as various memory regions in a fully protected zone. Measurements show that the resulting overhead is very modest as well as in execution time a well as in additional code size. The major impact is on the required data memory as the MMU requires the data sections to be aligned with specific minimum blocks.

Priority based preemptive scheduling is still used to achieve (hard) real-time behavior but with an option to assign time constraints (earliest starting time, latest termination time as well as use of CPU cycles) on top of the priority based scheduling. If these conditions are violated they are signaled to the kernel allowing to recover from the run-time faults.

7.7.Partitioning for safety and security

While we assumed in the beginning of the paper that the software itself is error-free, it still runs on potentially failing hardware in a potentially insecure environment. If a system is safety critical, the designer must make sure it remains safe and secure, even when faults happen or when the system's security is breached. Both result in erroneous states of the system. How much and what type of measures the designer must take largely depends on the safety and security risks resulting from these fault or breaches. At the system level, we can consider them as failures.

Dealing with such failures can be done by applying redundancy and diversity [22] . In order to better utilize the performance offered by advanced multi-core processors, an approach that is often taken is to "partition" the software in space and time on the processor. The goal is to isolate the different partitions so that no fault can propagate beyond the boundaries of its allocated partition and jeopardize the software executing in another partition. It must be noted that such partitioning is only possible on processors with the right hardware support, such as MPUs, MMUs and reprogrammable timers.

The principles of space and time partitioning are outlined below followed by a novel fine grain partitioning approach implemented in VirtuosoNext without jeopardizing the real-time support, as evidenced in section \ref{section:benchmarks}.

7.7.1.Space Partitioning

The standard approach for space partitioning is derived from the hypervisor approach, initially developed on server systems allowing to run multiple server applications (e.g. webservers) interleaved in time, which each application running on top of a virtual machine (isolated in space by using the protection offered by the MMU). This approach typically timeslices between the different applications but seriously impacts on the real-time response.

In VirtuosoNext, each task runs in its own memory space, also protected by the MMU, but in order of its priority. This fine-grain task level space partitioning compared to the process or application level space partitioning above has the advantage that it allows a much finer level of partitioning which can be as small as a single line of code, but typically the function providing the task entry point, without jeopardizing the real-time capability, an issue that traditional hypervisor type approaches have. In process level space partitioning the data of individual threads of a process are shared, which means they can corrupt each others data. This is not the case when using the fine grain partitioning support of VirtuosoNext, whereby each task runs in User-Mode and is only permitted to access its own memory (allocated at compile time), as well as explicitly shared memory in the form of global variables. This also prevents direct access to the underlying hardware for which the application task can call the trusted services of the underlying RTOS kernel and its driver layer. As in VirtuosoNext device drivers are implemented as tasks, the user can also develop them in Supervisor-mode.

With VirtuosoNext the application is now split explicitly between a trusted and a non-trusted zone. The trusted zone contains the qualified kernel task and the driver layers. The untrusted zone contains the application tasks that can use the services provided by the trusted zone. In this case the kernel task can be fully trusted as it underwent a qualification process.

7.7.2.Time Partitioning

VirtuosoNext does not provide a classical time partitioning (read: time slicing) implementation as seen in hypervisors, instead like its predecessor OpenComRTOS [1] it provides system wide (distributed) priority based preemptive scheduling at all levels. This means that a high priority request from task A on Node 1 for a Service provided at Node 23 will be treated everywhere as a high priority request. This means that with the exception of some memory and scheduling overhead, VirtuosoNext provides the responsiveness of a traditional RTOS, albeit in a distributed implementation and classical scheduling theories remain valid. Rather than allocating fixed and isolating time slots (that put a serious lower boundary on the reaction time of the system), VirtuosoNext provides support for restricting the scheduling in time of tasks on top of the priority based scheduling. For example, tasks can be defined with earliest starting time and latest termination times or with a maximum CPU cycles budget. The kernel task continuously monitors these tasks specific boundary conditions. Note however, that such boundary conditions are often not needed, unless stringent safety requirements impose them. The rationale for this statement is further elaborated.

7.7.3.Current practice: ARINC-653

Commonly used Space and Time Partitioning (RT)OS are often based on the ARINC-653 specification. It defines a standardised approach on how to configure and specify the various partitions as well the interface functions on how to program applications. Most RTOS vendors offer a compliant implementation that runs the (RT)OS kernel on top of a time-slicing hypervisor. The hypervisor has two functions. Firstly it allocates timeslots to the partitions and hence isolates the partitions from each other in the time domain. Each partition will also execute in a protected memory region, hence isolating the partitions in space. Inside a partition, the tasks can be scheduled according to priority. Secondly, it isolates the application partitions from the I/O domain by providing a visualization layer. As mentioned before, the approach jeopardizes on the hard real-time performance because of the timeslicing (often tens of milliseconds) and offers coarse grain space partitioning (although on processors with less sophisticated memory protection this is often the only option). In the next sections, we discuss some selected processors and how VirtuosoNext exploits their hardware support to offer fine-grain partitioning in combination with hard real-time support.

7.7.1.ARM-Cortex-M3

The ARM-Cortex-M3 [23] is a widely used micro controller architecture, typically is clocked at 50 to a few 100 MHz. It provides a simple MPU which allows to specify 8 address regions the task running in user-mode is allowed to access. tasks running in supervisor-mode may always access the whole address space. At runtime it is easy to reconfigure the MPU during a context switch. The downside of this simple MPU is that it requires that the memory regions have sizes of 2^n, and that the starting address is aligned to the size. This causes additional complexity when preparing the linker script for an application as the user must manually properly align the memory regions. In VirtuosoNext we follow a different approach by linking the application twice, the first time to determine the real size of the different regions which is then used by our Section-Analyser tool to generate a linker script where the different memory regions are properly aligned.

7.7.2.ARM-Cortex-A9

The ARM-Cortex-A9 [24], in our case clocked at 700MHz, provides an MMU which works with two, user maintained, page tables, each table supports a different page-size (4kB, and 1MB). During the context switch the page table is updated to make the pages of the current task inaccessible and the pages of the next task accessible. This is a computationally complex process and thus rather expensive.

7.7.3.Freescale T2080

The Freescale T2080 [25] consists of 4 PowerPC e6500 cores, clocked at 1.8GHz, of which each provides two threads, thus the SoC provides 8 logical cores. Each logical core provides its own MMU, however this MMU does not use a page table in main memory instead the page table is stored inside the MMU. The MMU supports pages-sizes of $4kB \cdot 2^n$ Each entry can be assigned a translation ID and all pages must be aligned to a 4kB boundary. Each MMU entry has a 14bit Translation-ID which that is compared to the specified Process-ID, limiting access to the memory specified by the MMU entry. To properly align the page tables, we use a Section Analyser tool, and a multi stage build process, like for the ARM-Cortex-M3. A Project Generator assigns each task a dedicated Process-ID. The computational complexity of reconfiguring the MMU during a context switch is limited to changing the data and bss segment entries for the next task, the overhead being independent of the size.

7.7.1.Texas Instruments TMS320C6678

The Texas Instruments TMS320C6678 [26] has 8 physical cores, clocked at 1.25GHz but these provide neither an MPU nor an MMU. However, there are SoC-Level MPUs which can be used to isolate memory regions from cores and peripherals on the SoC. By default every core / peripheral has access to the complete address space, and in the MPUs one explicitly blocks peripherals from accessing a certain region. This is the opposite approach from the other architectures discussed in this paper. Thus on this target the isolation is only possible on the core level. However, the impact is minimal as this only requires an initial setup and no modification afterwards.

7.1.Benchmarks on single and multi-core targets

In this section we give the results of the Semaphore Loop and Interrupt latencies for the different architectures listed previously.

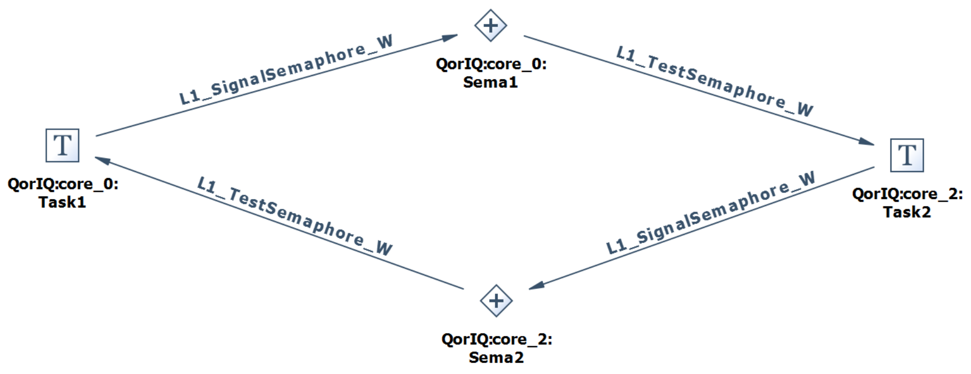
7.1.1.Semaphore Loop Times

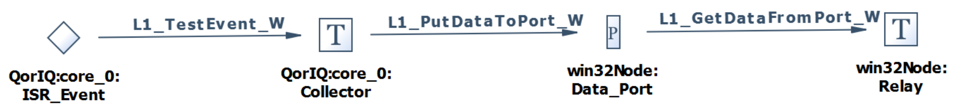
Space partitioning also affects runtime performance because the context of a task now includes also the information about the memory regions it is allowed to access. This becomes visible when comparing the time the system takes to perform a Semaphore-Loop (two tasks, two semaphore Hubs with one loop requiring eight context switches per loop). The table below gives the semaphore loop times measured for the different targets for non-partitioned and partitioned implementations, except for the C6678. For the ARM-Cortex-M3 and the T2080 the enabling space partitioning has only a moderate impact of about 10% while of the ARM-Cortex-A9 the space partitioning has an impact of about 30%.

Table 4 Semaphore Loop Times in microseconds

Target	Non-Partioned	Partitioned
ARM-Cortex-M3 (@50MHz)	54.60	58.90
ARM-Cortex-A9 (@700MHz)	23.65	30.39
C6678 (@1.25GHz)	2.81	n.a.
T2080 (@1.8GHz)	5.64	6.01

7.1.1.Interrupt Handling Latency





In addition to fast context switching a RTOS must also be able to react predictably and with very low latency to external events, so called Interrupts. In the case of VirtuososNext we define two Interrupt Latencies of interest. The first one is the IRQ (Interrupt Request) to ISR (Interrupt Service Routine) Latency, the second is the IRQ to task Latency.

Note that the interrupt latency is really a histogram as it depends on what other applications are active on the processing node. To simulate such a stress pattern, a semaphore loop (see next paragraph) is scheduled in parallel with the interrupt latency measurement. The semaphore loop is a very good stress load as it continuously disables interrupts for short interval when the Kernel task executes the semaphore services and context switches.

With space partitioning enabled the IRQ to Task latencies, shown in Table \ref{table:irq2isr}, generally increase between the protected and non-protected versions, especially the latency maximum increases substantially due to the fact that this is caused by disabling interrupts while performing a context switch. The context switch for the protected version is usually more complex than for the non-protected version. Similarly, the IRQ to Task latency increases when enabling space partitioning.

Table 5 Minimal and maximal IRQ to ISR Latencies in nanoseconds

Target	Non-Partioned	Partitioned
ARM-Cortex-M3 (@50MHz)	780 - 2500	960 - 4920
ARM-Cortex-A9 (@700MHz)	100 - 314	138 - 1150
C6678 (@1.25GHz)	160 - 260	n.a.
T2080 (@1.8GHz)	286 - 793	286 - 819

Table 6 Minimal and maximal IRQ to Task Latencies in microseconds

Target	Non-Partioned	Partitioned
ARM-Cortex-M3 (@50MHz)	15.0 - 35.0	16.0 - 39.0
ARM-Cortex-A9 (@700MHz)	0.994 - 2.182	1.726 - 4.228
C6678 (@1.25GHz)	936	n.a.
T2080 (@1.8GHz)	2.158 - 3.705	2.262 - 3.848

Interrupt Latency Measurement Application Diagram

Interrupt Latency Histogram on Freescale T2080 @ 1.8GHz (logarithmic scale

7.1.2.Code Size

The fine-grain space partitioning implementation of VirtuosoNext is lightweight both in code size and in runtime impact. Table \ref{table:codesize} shows code size of VirtuosoNext partitioned and non-partitioned. The code sizes were obtained by building the same application using all available Services (compiled with Os). We see that for the ARM-Cortex-M3 and ARM-Cortex-A9 the code size increases by more than 30\% while for the T2080 the increment is in the area of 3\%. The reason for this is that for the ARM-Cortex platforms a new context switch had to be developed when we implemented the space-partitioning while for the T2080 almost the same context switch is used for both variants. Note that the code sizes given include the runtime overhead of the compiler and the system initialisation.

Target	Non-Partioned	Partitioned
ARM-Cortex-M3 (@50MHz)	8,656	11,564
ARM-Cortex-A9 (@700MHz)	15,144	21,844
C6678 (@1.25GHz)	26,448	n.a.
T2080 (@1.8GHz)	37,224	38,504

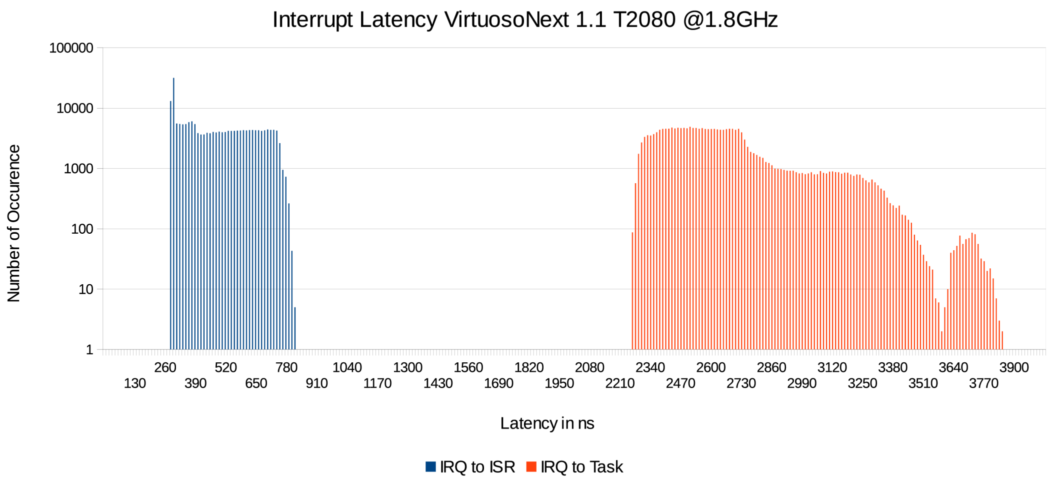
Code size in bytes, non-partitioned and partitioned.

7.1.3.Multi-core SoCs, T2080 vs TMS320C6678}

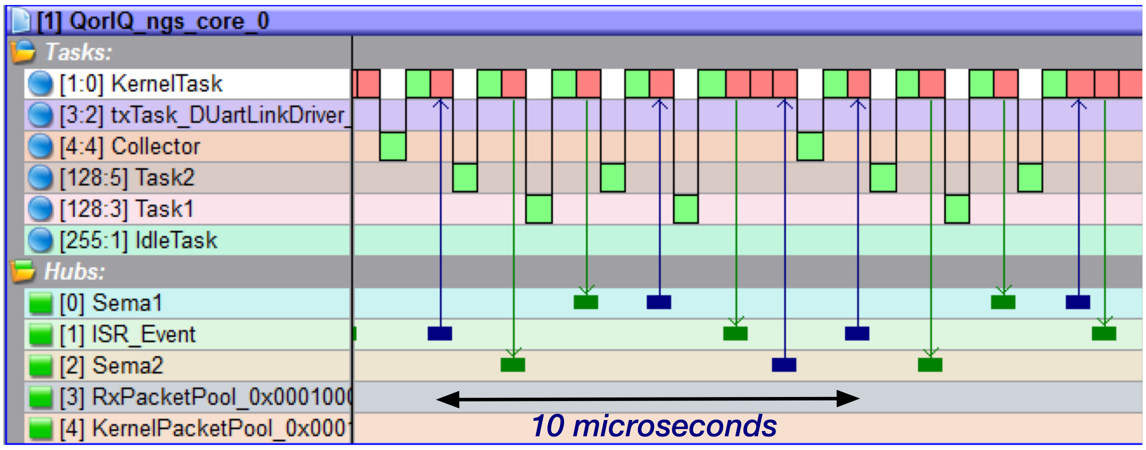
The TI-C6678 and the Freescale T2080 SoCs have both 8 logical cores. The T2080 has four physical cores supporting two threads each, while the C6678 has 8 physical cores. In case of the T2080 Each core has its private 32kB L1-Data and L1-Instruction Caches, which are shared among the threads, and the L2 Cache is shared among all cores. In contrast the C6678 provides each core with its own L2 cache of 256kB which furthermore can be configured as normal memory. This seriously reduces the memory access overhead and thus improves the real-time capability. On the other hand, the C6678 has less memory protection logic using a MPU, so that the protection is practically at the core level. The T2080 offers fine grain memory protection using its MMU, so that each block of 4kB can be protected.

7.1.4.The ultimate real-time stress test}

In order to verify the approach the interrupt latency test above was modified to use a timer with a 10 microseconds periodicity. This results in a 100% CPU load, taken up for 27.53% by the two tasks running the semaphore loop, the kernel task, a collector task and the timer ISR. Each of the tasks are memory protected. Below a screenshot of the Even Tracer shows the trace data



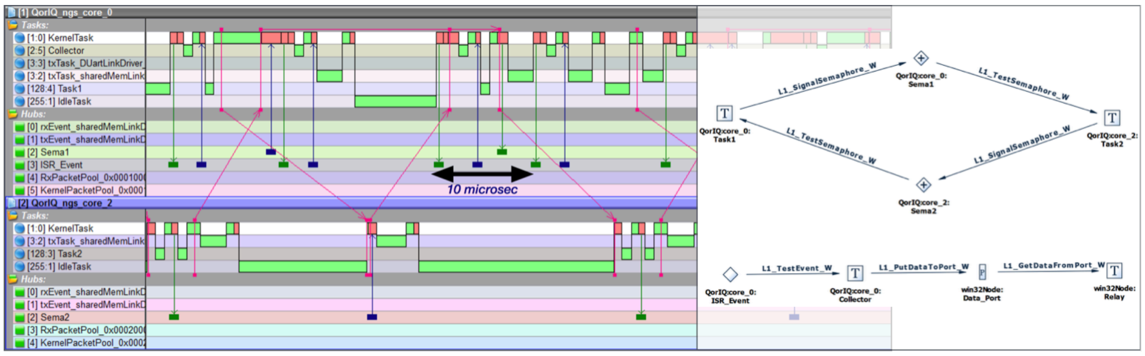
collected on core-0, while running the measurement. The trace shows the scheduling of the different tasks, with the Kernel-Task being on top, as it has the highest priority and the Idle-Task at the bottom due to it having the lowest priority. The measurements show no degradation of the real-time performance.



Trace of the Interrupt Latency Measurement with a 10 microsecond ticker and a single CPU Semaphore Loop.

7.1.5.The impact of multi-core communication}

In an additional stress test, the semaphore loop was distributed over two cores with the timer still at 10 microseconds. This introduces additional latency and CPU load due to the driver tasks (which is transparent for the application developer). The measurement task has the second highest priority after the Kernel-Task. The minimal latencies stayed the same, however the maximal IRQ to ISR latency increased to 3471 nanoseconds while the maximal IRQ to Task latency increased to 9.633 microseconds. The long IRQ to Task latency can be explained by the fact that the RX-Part of the communication is handled inside an ISR, which has a highest priority than the Kernel-Task. The Figure below shows a representation of the trace data collected on core-0 and core-2, while running the measurement.



Trace of the Interrupt Latency Measurement with a 10 microsends ticker and a Semaphore Loop distributed over core-0 and core-2.

7.2.Conclusions and recommendations for multi-core SoC design

The complexity of advanced multi-core chips was largely enabled by the law of Moore. The result is that chips are a lot less power hungry and allow much higher performance and functional density. Unfortunately, memory technology and the peripheral real-world often still runs at a much slower rate. Hence, caching and buffering are needed to reduced the mismatch in speed. The result is complexity and less predictability in time, partly because the design aims at peak performance (which is good for general purpose computing), less at bounded performance (which is needed for safety critical applications).

The concurrent programming model that was put forward to address the issues on the software side is also applicable to the hardware. If CPUs are decoupled (by having large enough local memory and caches), the impact of shared memory is vastly reduced. Being able to lock code in cache (hence it acts like fast SRAM), greatly improves the performance as well as the statistical execution spread. A side conclusion is that it pays of to have a smaller code size.

One should also not be afraid to use an heterogenous SoC architecture. The presence of up to 1000 interrupt sources, to be handled by a single CPU, is a clear indication that offloading the I/O work to small peripheral CPUs is beneficial for real-time, provided the programming effort is kept low by using a common high level API that is target independent. On the hardware side, the effort should be focused on reducing the latency (introduced by e.g. complex set-up and feature bloat). Examples of such heterogenous architectures are TI's OMAP family, that combine DSPs, ARM-M3 and -A9 in a single chip. While VirtuosoNext support all target CPUs, it is still a serious effort because of the complexity.

While the tests have confirmed the wide variance in real-time performance of different hardware architectures, we have also shown that applying good design principles in developing the RTOS can largely bridge the gap, even on processors that are less suitable for real-time. If the measurements show that the hard real-time constraints are difficult to satisfy, with a concurrent and portable programming model, one can easily redistribute the application or simply wait for the vendor to release their next generation faster chip.

8. Final conclusions

Modern advanced many/multicore chips introduce the need to take into account their complexity of shared resources and their statistical nature of executing applications. This means that traditional real-time and safety thinking (that assumes that everything is mostly static and predictable) is no longer fully applicable, unless the on-chip resources are seriously under-utilised. Nevertheless, they remain a necessary first-order approach that must be understood and taken into account.

This publication proposes to consider meeting real-time constraints as part of the QoS offered by an application, even in the presence of faults. The result is a scheme whereby **graceful degradation is defined as a design requirement**, especially in the presence of faults.

The design path to a working solution is to consider most, if not all on-chip shared resources, as resources for which the application functions compete at runtime constrained by their relative priority derived from their QoS level. In this case we can reuse the priority inheritance protocol for managing access to the resources. However, unless the chip designers have taken specific precautions, this means that for the higher safety levels physical partitioning is still a must. This publication does not yet present a complete solution on how to define the runtime scheduling and resource sharing parameters for a given application. We envision a process whereby first order approximations are derived from a static approach with runtime profiling allowing improving upon the selected parameters. However, it remains a trade-off exercise as full optimisation is not likely due to the statistical nature of the problem domain.

Finally, we have shown how different safety integrity levels (SIL) are related to quality of service (QoS), whereby a criterion (ARRL) was formulated that components must meet to be used in certain quality of service levels.

9. References

9.1. Further reading

1. E. Verhulst, R.T. Boute, J.M.S. Faria, B.H.C. Sputh, and V. Mezhuyev. Formal Development of a Network-Centric RTOS. Software Engineering for Reliable Embedded Systems. Springer, Amsterdam Netherlands, 2011.
2. Altreonic, January 2011. http://www.altrenoic.com
3. Eric Verhulst. Virtuoso : providing sub-microsecond context switching on dspswith a dedicated nanokernel. in international conference on signal processing applications and technology, santa clara september, 1993. 1993.
4. Wikipedia. Transputer — wikipedia, the free encyclopedia, 2011.
5. Inmos, January 2011. http://www.inmos.com, last visited: 20.01.2011.
6. C.A.R. Hoare. C.A.R. Hoare. Communicating Sequential Processes. Prentice-Hall, 1985.
7. C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-timeenvironment. J. ACM, 20:46–61, January 1973.
8. Loic P. Briand and Daniel M. Roy. Meeting Deadlines in Hard Real-Time Systems: The Rate Monotonic Approach. IEEE, 1999.
9. Mark Klein, Thomas Ralya, Bill Pollak, Ray Obenza, and Michael Gonzalez Harbour. A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems. Springer, August 1993.
10. Mast, January 2011. http://mast.unican.es, last visited: 20.01.2011.
11. A. Styenko. Real-Time Systems: Scheduling and Structure Af.Sc. Thesis. University of Toronto, 1985.
12. M.B. Jones. What really happened on mars, 1997.
13. IEC 61508 edition 2.0, 2010. [Online; accessed 19-March-2013].
14. A Formalised Real-time Concurrent Programming Model for Scalable Parallel Programming" authors Eric Verhulst, Bernhard H.C. Sputh at the Workshop on High-performance and Real-time Embedded Systems(HiRES 2013) January 23, 2013, Berlin, Germany. http://www.altreonic.com/content/altreonic-hires2013-workshop
[15] N. C. Audsley, "Deadline monotonic scheduling," 1990.
[16] P. Risat Mahmud, "Mars pathfinder: Priority inversion problem," 2014.
[17] C. A. R. Hoare, Communicating Sequential Processes. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1985.
[18] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," Commun. ACM, vol. 21, no. 7, pp. 558–565, Jul. 1978. [Online]. Available: http://doi.acm.org/10.1145/359545. 359563
[19] ——, ""sometime" is sometimes "not never": On the temporal logic of programs," in Proceedings of the 7th ACM SIGPLANSIGACT Symposium on Principles of Programming Languages, ser.
POPL '80. New York, NY, USA: ACM, 1980, pp. 174–185. [Online]. Available: http://doi.acm.org/10.1145/567446.567463
[20] ——, Real-Time Model Checking Is Really Simple. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 162–175. [Online]. Available: http://dx.doi.org/10.1007/11560548 14
[21] B. H. Sputh, E. Verhulst, and V. Mezhuyev, "OpenComRTOS: Formally developed RTOS for Heterogeneous Systems," in Embedded World Conference 2010, Mar. 2010.
[22] E. Verhulst, B. Sputh, and P. Van Schaik, "Antifragility: systems engineering at its best," Journal of Reliable Intelligent Environments, vol. 1, no. 2, pp. 101–121, 2015. [Online]. Available: http://dx.doi.org/10.1007/s40860-015-0013-3
[23] ARM Cortex-M3 Processor Technical Reference Manual, Revision r2p1 ed., ARM, 2015.
[24] ARM Cortex-A9 Processor Technical Reference Manual, Revisio r4p1 ed., ARM, 2012.
[25] QorIQ T2080 Reference Manual, NXP, document Number: T2080RM Rev. 3, 11/2016.

[26] TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor (Rev. C), Texas Instruments, http://www.ti.com/lit/ds/symlink/tms320c6678.pdf.

9.2. Acknowledgements

While GoedelWorks is a development of Altreonic Systems, part of the theoretical work was done in the following projects:

1. CRAFTERS. Artemis Project. ConstRaint and Application driven Framework for Tailoring Embedded Real-time Systems. Project website: http://www.crafters-project.org
2. Airbus, for lending us a PowerPC platform.
3. EUROCPS NoFiST (Novel Fine Grain Space and Time Partitioning for a Mixed Criticality Platform) project with Thales TRT.

What this booklet is all about

Developing real-time Embedded Systems engineering is becoming complex because we have now interconnected target system with many processors, often of a different type that contain tens if not hundreds of processors. The complexity also increases because the semiconductor developers can squeeze more and more on single chip. This requires that applications must share the on-chip resources while the temporal behaviour becomes more statistical in nature.

While traditional real-time scheduling techniques are still valid, there is a need to shift towards a scheduling approach based on Quality of Service (QoS) that includes the capability to continue the processing for the most important application parts even when some of the resources fail. By considering this, we have entered the domain of safety engineering whereby the ultimate QoS offered is the survival of the system. This introduces the concept of Assured Reliability and Resilience Level (ARRL) resulting in requirements to be met by components to meet the system level QoS requirements.