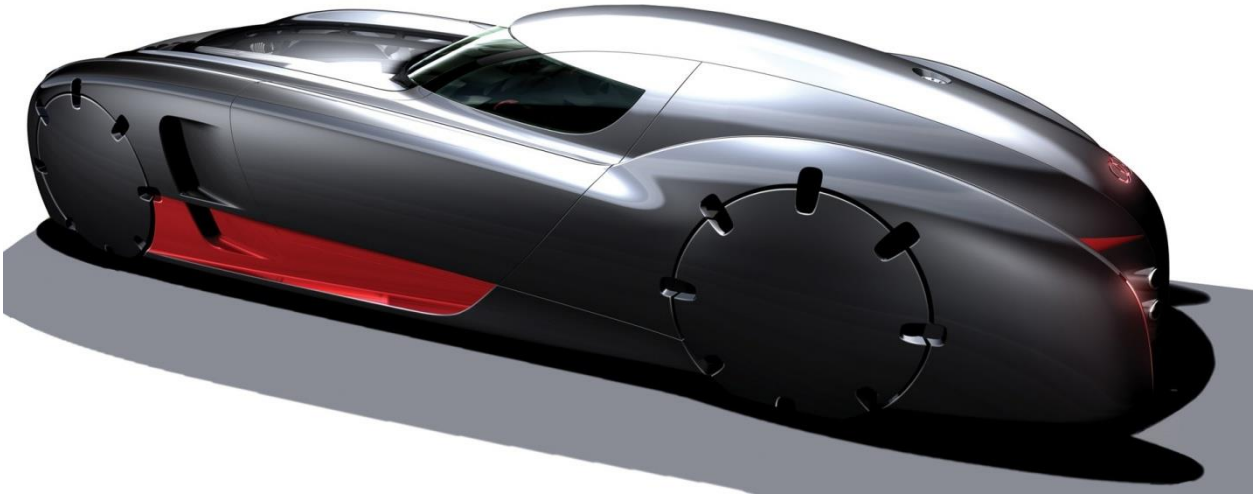
April 2013

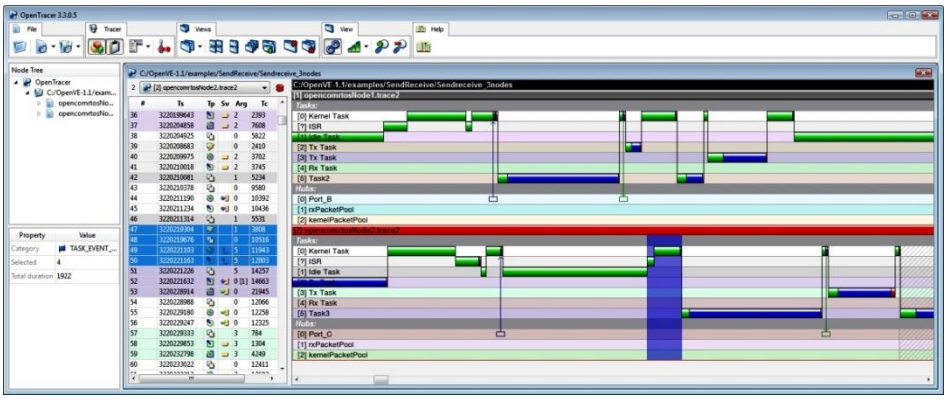


Altreonic "From Deep Space to Deep Sea"

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

First publication in the Gödel Series:

Systems Engineering for Smarties



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

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

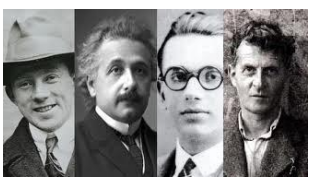
April- 2013

Author: Eric Verhulst

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 ˈgøːdəl] (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).**

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 OpenComRTOS as a reference.

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

1.1 Background of OpenComRTOS

The initial purpose for developing OpenComRTOS [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.

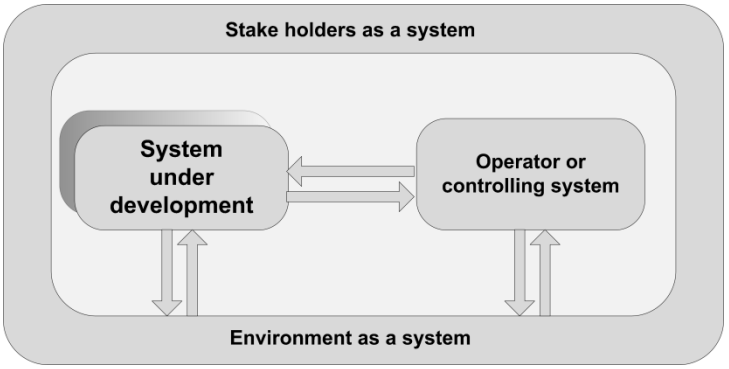


Figure 1 The context of systems engineering

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.

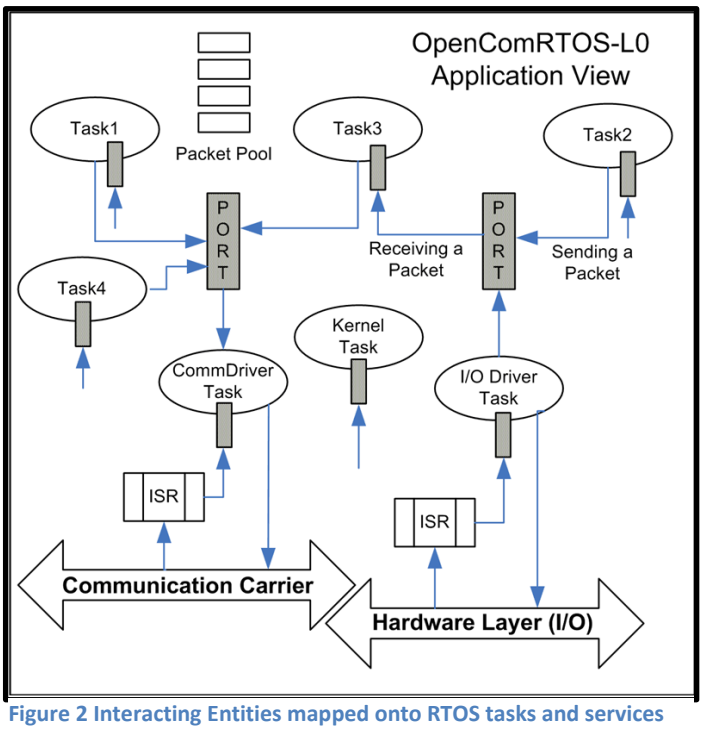
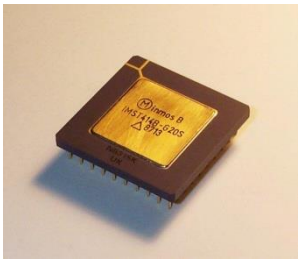


Figure 2 Interacting Entities mapped onto RTOS tasks and services

1.2 Early requirements derived from the Virtuoso RTOS



A precursor to OpenComRTOS was 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.

2 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.

2.1 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.

2.2 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.

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.

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?

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.

2.3 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 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.

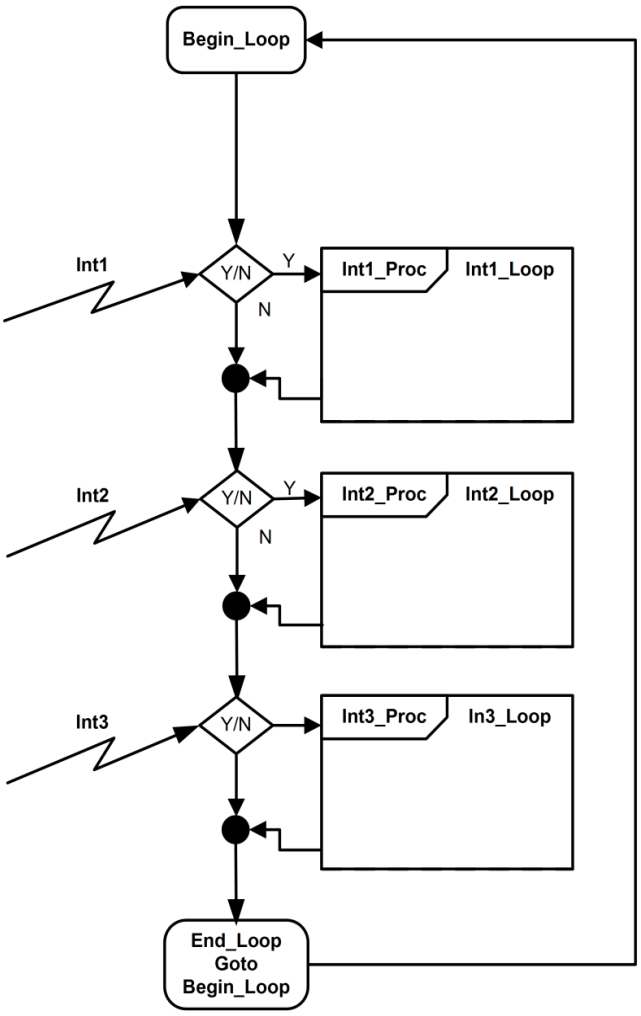


Figure 3 Superloop schedule with three interrupt sources

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)**. OpenComRTOS 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.

2.4 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:

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

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;

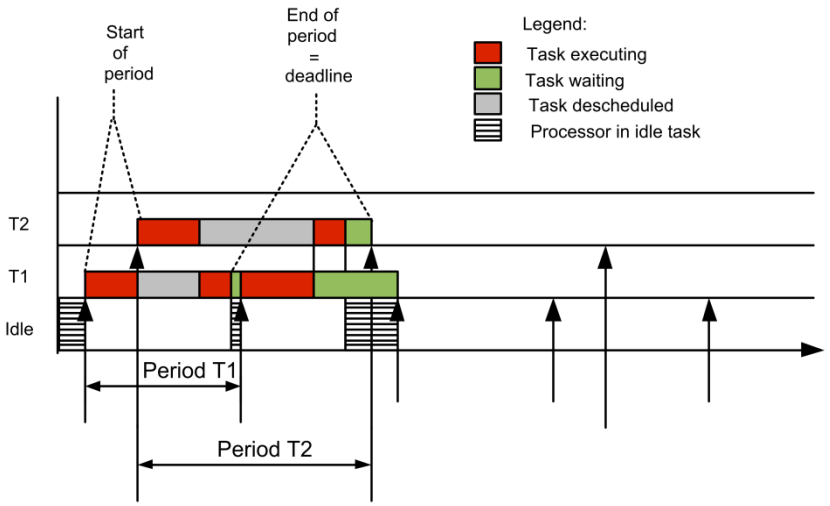


Figure 4 Two periodic tasks scheduled with RMA

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.

2.5 The application of RMA in OpenComRTOS

In OpenComRTOS 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 OpenComRTOS supports multi-processor systems. Hence priorities are considered a system-wide scheduling parameter and the DMA should still hold locally on each processor.

OpenComRTOS 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. OpenComRTOS provides this complemented with a runtime tracing function allowing profiling the temporal behaviour at runtime.

2.6 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.

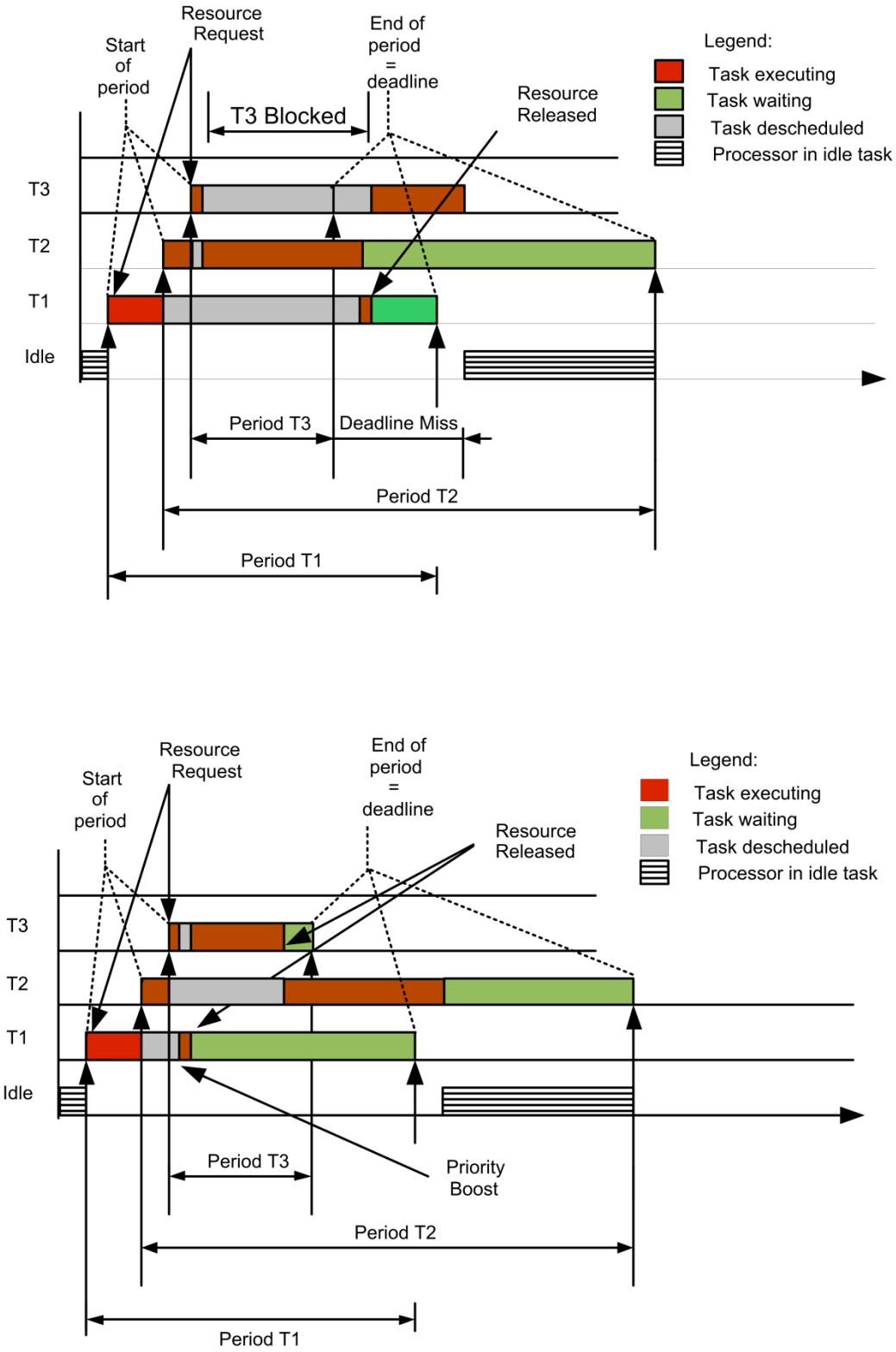


Figure 5 Three tasks sharing a resource first without and then with priority inheritance support

2.7 Distributed priority inheritance in OpenComRTOS

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 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.

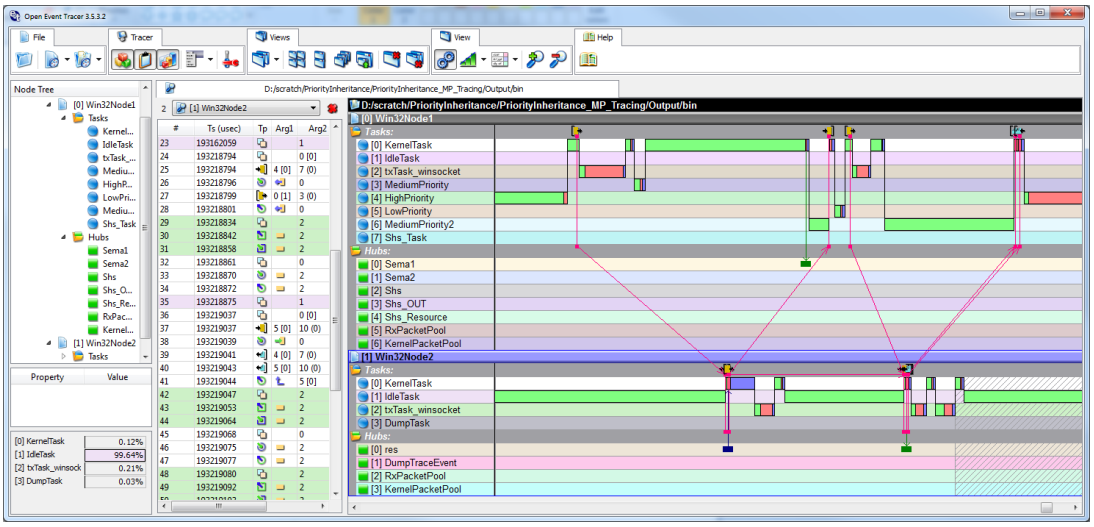


Figure 6 Event trace of distributed priority inheritance in OpenComRTOS

2.8 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 OpenComRTOS 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 OpenComRTOS 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 OpenComRTOS project, but the fact that OpenComRTOS supports programming a multicore and distributed system in a transparent way facilitates addressing such requirements.

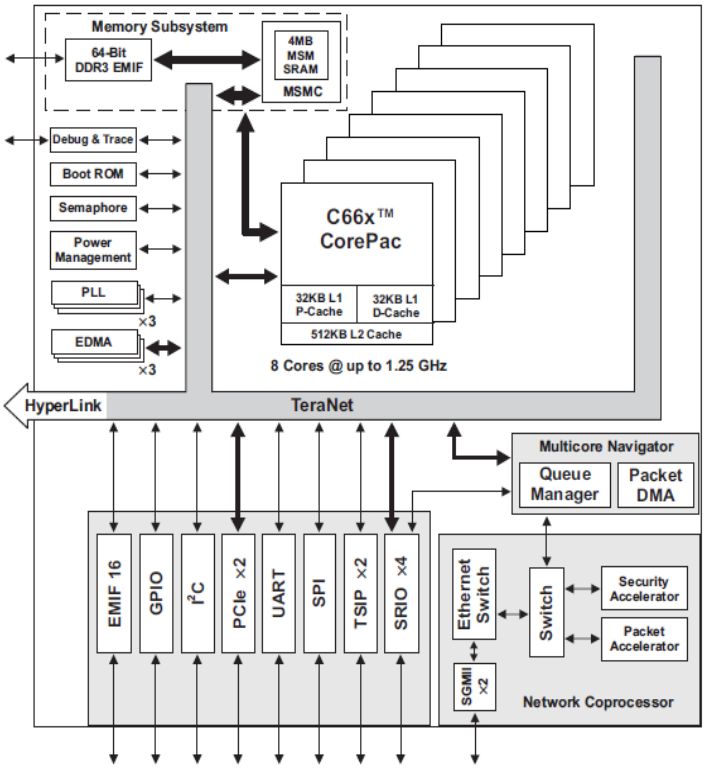


Figure 7 Advanced multicore chip: Texas Instruments C6678 DSP

3 Hard real-time on advanced multi-core chips

3.1 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.

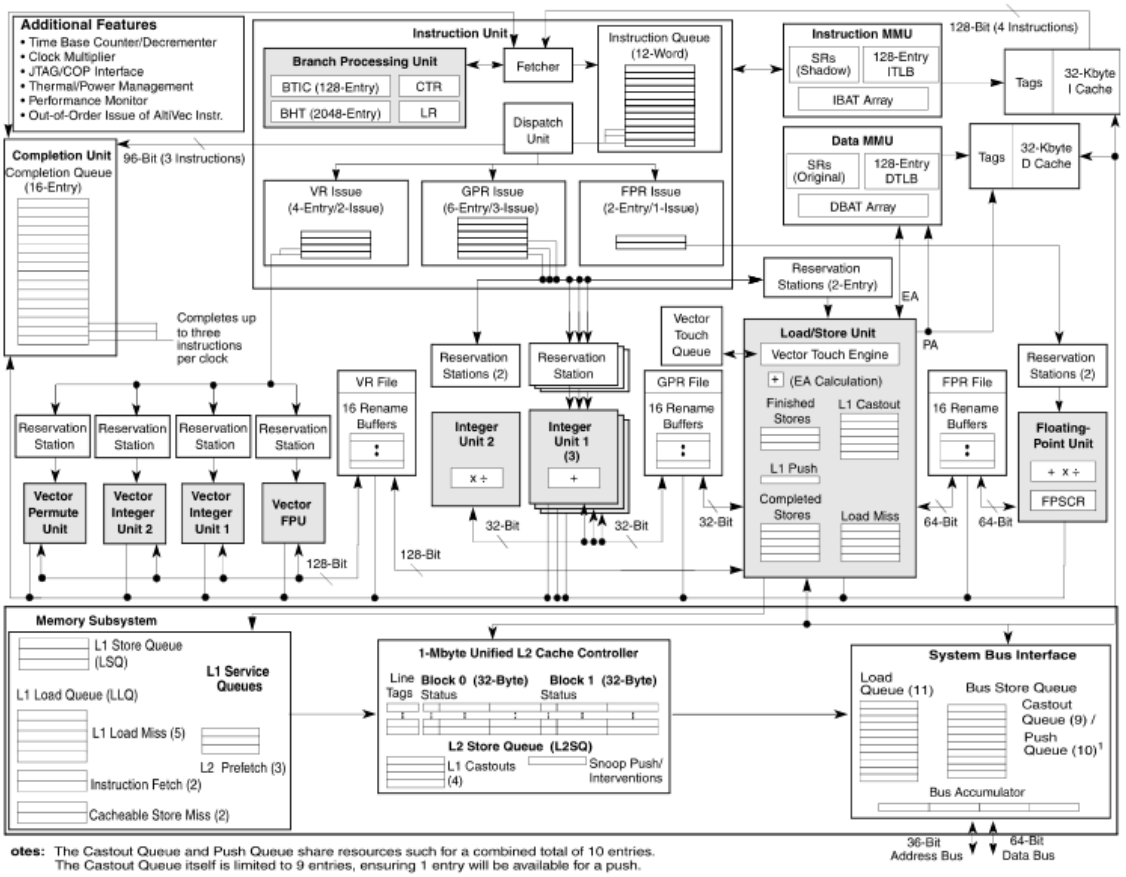


Figure 8 Freescale PowerPC 744X block diagram

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.

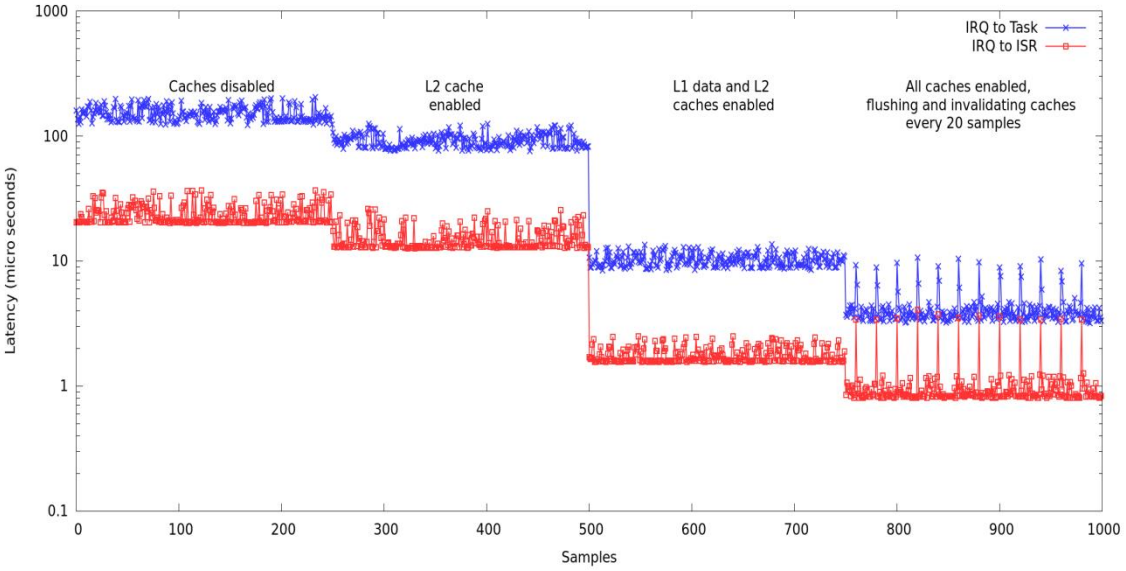


Figure 9 Interrupt latency to ISR and task level on a 1 GHz PowerPC (logarithmic scale)

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.

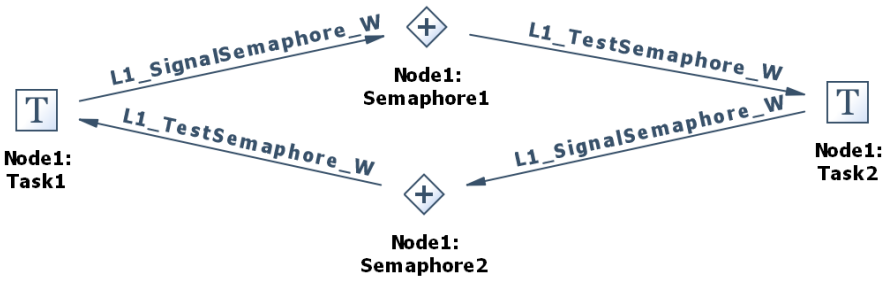


Figure 10 Semaphore example used for benchmarking

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

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

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 OpenComRTOS 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.

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 statistical approach must be taken.

As a summary, the ISR and task interrupt latencies histograms (limited to 1000 measurements) were plotted on single graphs with the different cache modes enabled.

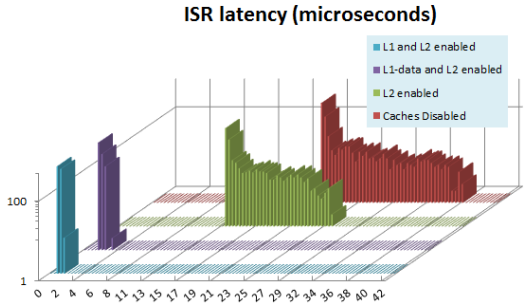


Figure 11 Interrupt to ISR latency

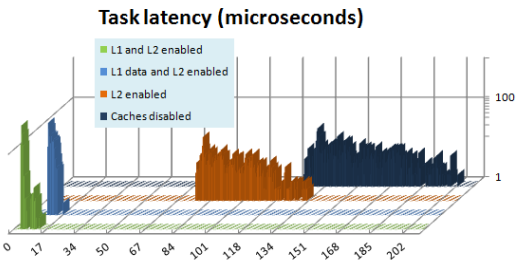


Figure 12 Interrupt to task latency

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).

3.2 Inter Core Communication Performance

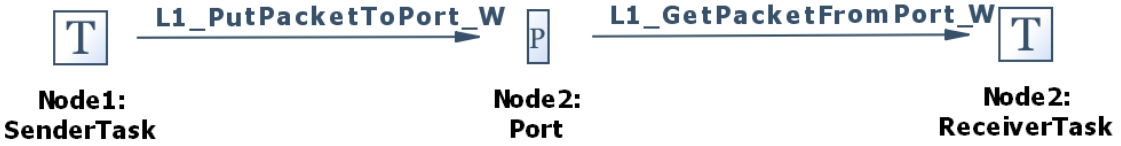


Figure 13 Application Diagram for the Throughput Measurement

To measure the application level inter core communication throughput, i.e. the usable Task-to-Task bandwidth when developing an application, we performed the following measurements. The benchmark system consists of two Tasks: a sender task and a receiver task, communicating using an intermediate OpenComRTOS port-hub. The figure shows the application diagram of the system. The sender task sends a Packet to the Port-Hub from which the receiver task receives it. The Port-Hub interactions are done using waiting semantics, which means that the sender task has to wait until the receiver task has synchronised with it in the Port-Hub. The port-hub copies the payload data contained in the packet from the Sender-Task to the L1-Packet from the Receiver-Task, and then sends acknowledgement packets to both Tasks. We measured how long it takes the receiver task to receive 1000 times a data packet of a specific size. To

perform the initial synchronisation the receiver task waits for a first communication to take place before determining the start time. Please note that the sender task and receiver task synchronise in the port-hub, thus the sender task can only send the next packet, after it has received the acknowledgement packet that the previous transfer was performed successfully.

In the following section we outline measurements done on two different types of multi-core targets. The firs one is the Intel 48-core experimental chip and the second one the already mentioned Texas Instruments C6678 8-core DSP.

3.2.1 Intel SCC

The Intel SCC is composed of 48 Pentium cores (running at 533 MHz), each with 16 kB data and program caches and 256 kB L2 cache. Each tile, which consists of two cores, provides a 16 kB large Message Passing Buffer (MPB). In the link driver implementation we assigned each core of the tile 8 kB of this buffer, which it uses as an input port for the OpenComRTOS drivers. This means that each core reads the messages meant for it from its part of the MPB. To send a message each core writes the message directly into the MPB of the core the message is intended for, i.e. we establish a full mesh on the Intel-SCC, leaving all the routing decisions to the underlying routing network. Inside the MPB the data is organised using a lock free ring buffer implementation, where the writer and reader Task do not need to lock each other out. However, it is still necessary to prevent that more than one writer tries to gain access to the MPB in parallel, thus there is one locking operation involved. The lock is represented by an atomic variable. Having an RTOS means that it is necessary to inform the reader core that new data has arrived, this is achieved by the writer-node issuing an Interrupt Request (IRQ) to the reader-node. Upon receiving the IRQ, the reader-node reads out the data, translates the transfer packet into a local packet and then passes it to the Kernel-Task for processing.

3.2.2 Texas Instruments C6678 8-core DSP.

The TI-C6678 contains 8 cores running at 1 GHz, Figure 7 gives a block diagram of the chip. Each core has 32 kB L1 cache for data and program and an additional L2 cache of 512 kB (used as SRAM). The 8 cores share also a fast 4 MB SRAM and external DDR3 memory. The chip has also an on-chip queue management system, Ethernet switch, DMA and SRIO amongst other, all connected over a fast TeraNet switching network. The complexity is high and the chip has about 1000 interrupt sources and a 3 level interrupt controller. Impressed, we called it a ”RoC" (Rack On a Chip). The inter core communication was implemented by using the Queue Management Sub System (QMSS) available in the chip and exchanging pointers to blocks of the 4 MB shared SRAM. The queues that are used for this purpose can issue an interrupt to their CPU if data has arrived, providing an easy scheme to communicate.

3.2.3 Measurements on the Intel-SCC

When distributing the tasks over different Nodes in the system, the data will be transferred between the two nodes using link drivers and using the on-chip communication mechanism. These link drivers transfer only the used part of the data part of the packets. We measured the following different system setups, with different payload sizes:

- Single-core: In this setup all tasks and the hub are on the same core. Thus no inter core communication is involved.

- Multi-core: Afterwards the benchmark was distributed over two nodes, in the following way:
  - node1: Sender task
  - node2: receiver task and Port-Hub

In this setup we measured with different numbers of hops (see (Manual:Intel:SccProgGuide for details) between the two cores:

- No-Hop: Node1 on core 10 and Node2 on core 11
- Hop: Node1 on core 10 and Node2 on core 8
- 8-Hops: Node1 on core 10 and Node2 on core 36

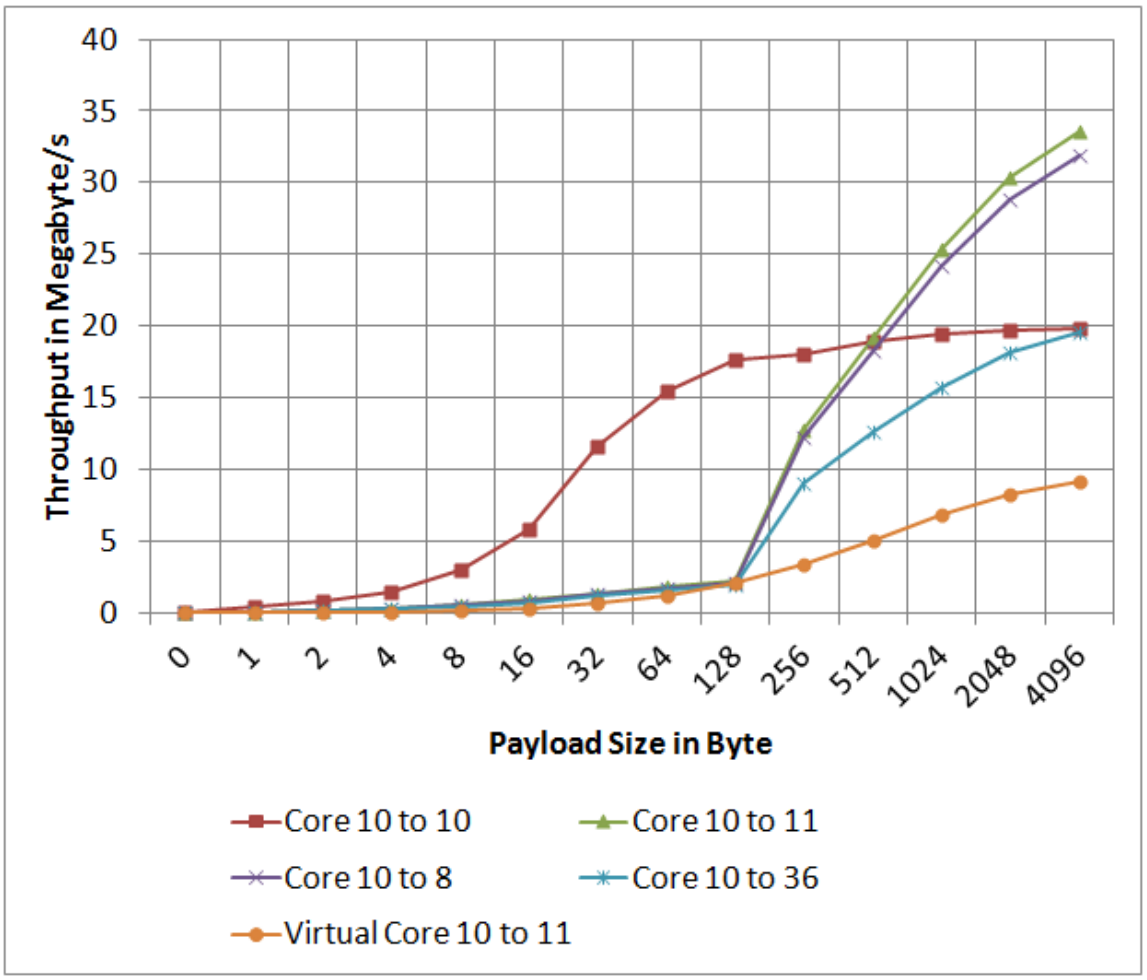


Figure 14 Intel-SCC Throughput over Packet Payload Size

This figure gives the measured results for the different systems. What sticks out is that the single core example goes into saturation at around 20 Mbyte per second, while the distributed versions achieve a higher throughput of up to 33 Mbyte per second.

There is also a strange jump in throughput from payload sizes 128 byte to 256 byte, for the distributed version, which we do not observe in the single core version. Furthermore, we see a strong influence of the routing network which nearly halves the throughput between the No-

Hop and the 8-Hop versions, thus the location of the Nodes and their distance matters on the Intel-SCC.

The curve labelled `Virtual Core 10 to 11' is moving the data, by transferring the ownership of a shared buffer from core 10 to core 11. This is done by transferring the buffer information (address, size, resource-lock-id) from core 10 to core 11 using a port-hub.

Once core 11 has this information it locks a resource, to avoid unintentional access, copies the data, and then releases the lock. The achieved throughput is about half of what we achieved in the single core version.

The reason for this is that the buffer is placed in shared memory which halves the achievable throughput. The throughput of the bare version, i.e. without OpenComRTOS running, just a main and small runtime layer, drops from 17.4 Mbyte per second, when copying from private memory to private memory, to 10.3 Mbyte per second when copying from shared memory to private memory.

3.2.4 Impact of core distance on timings

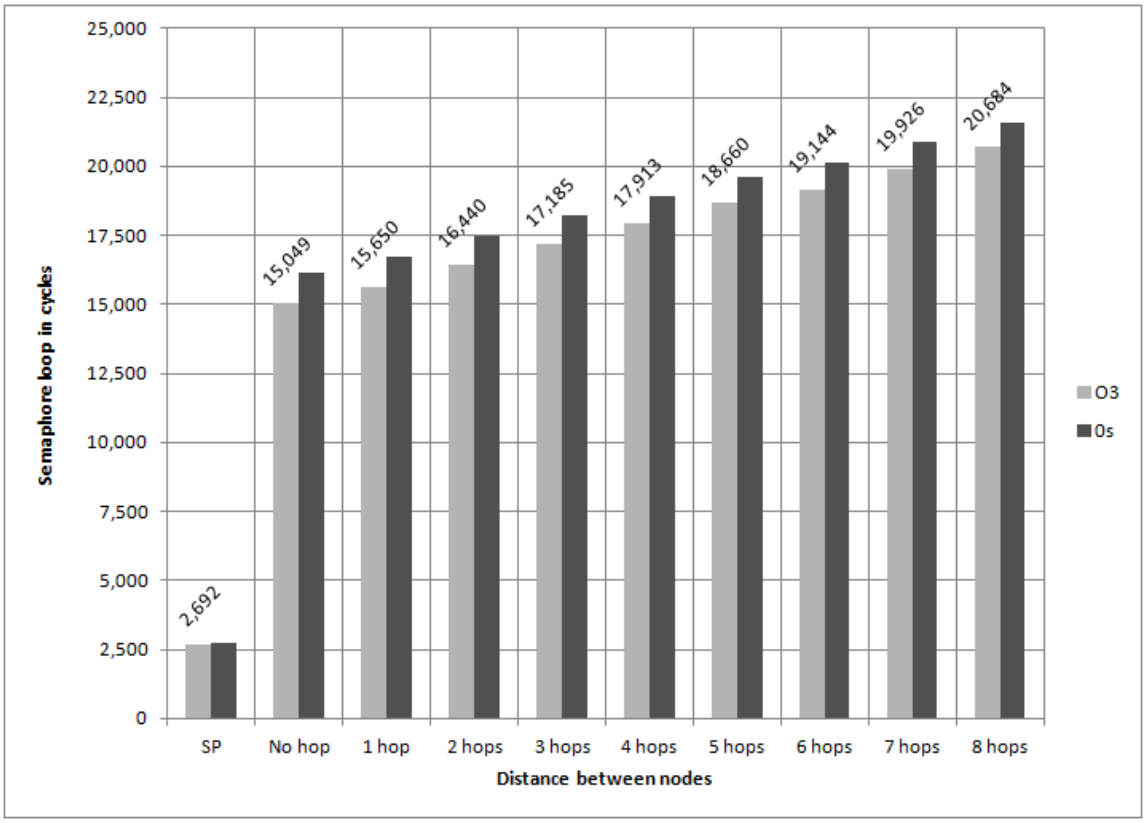


Figure 15 Multicore semaphore loop test on Intel SCC showing hop delay

While on the TI chip all cores can directly communicate (there are only 8 of them) The Intel SCC provides an additional test possibility because the 48 cores communicate over a NoC with routers. These routers introduce addition ``hop'' delays. We have measured the semaphore loop for all possibilities (from 0 hops for directly connected cores to 8 hops for those furthest

apart). The semaphore loop times then range from 15049 cycles to 20684 cycles (compiled with -O3). In itself, these timings are quite reasonable given the extra hop delays, that range from 280 to 385 cycles in one direction (calculated by dividing the extra hop delay of a semaphore loop by 2). This hop delay is not only due to communication latency but also to extra context switching by the driver and the Kernel-Task, interrupt handling and the need to invalidate the cache.

3.2.5 Measurements on the Texas Instruments C6678 8 core DSP

The TI-C6678 evaluation board available to us was clocked at 1 GHz, thus all measurements were done at this frequency. Another point to mention is that In the initial tests none of the DMA units provided by the TI-C6678 have been used for these measurements, thus the DSP-Core had to spend all its cycles to move the data.

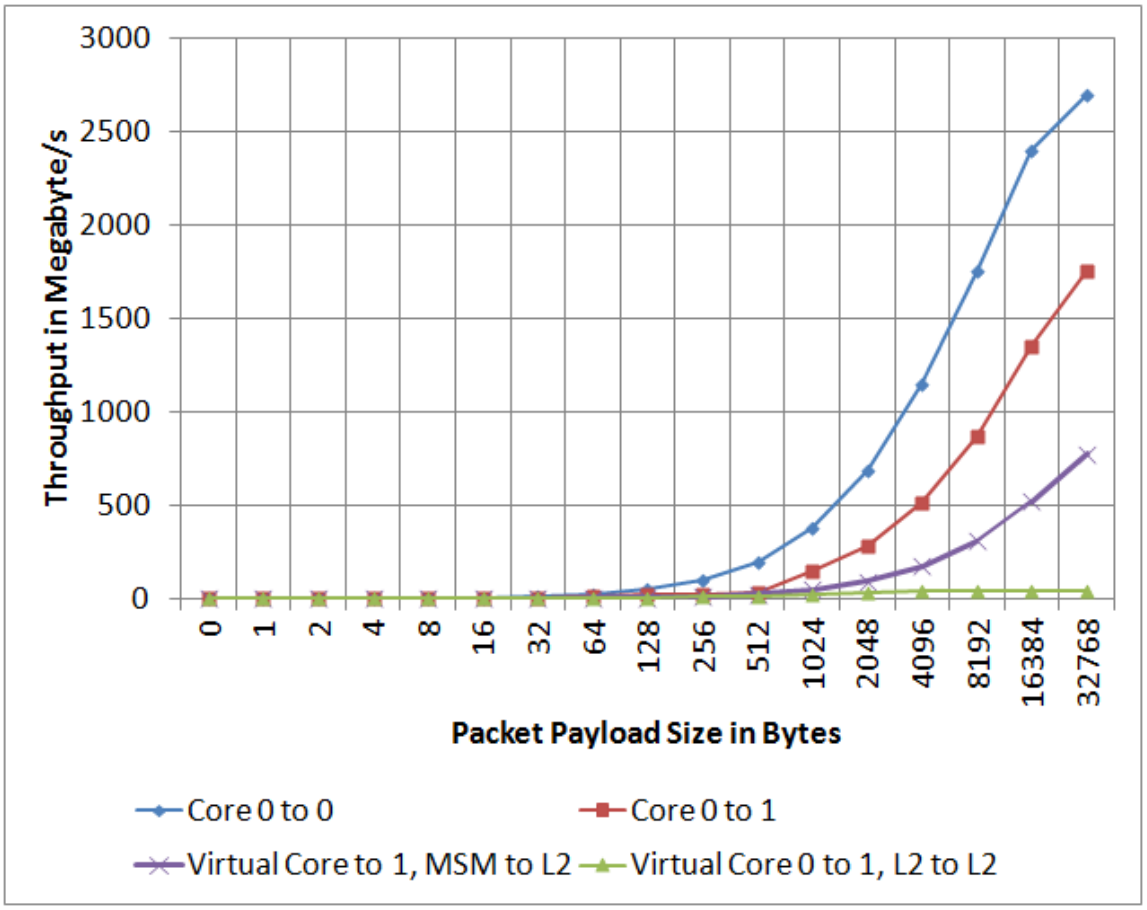


Figure 16 TI-C6678 Throughput over Packet Payload Size

The figure gives the throughput measurements for the TI-C6678 @ 1 GHz, for both the single core (`Core 0 to 0') and the distributed version (`Core 0 to 1'). The measurement setup is described as follow:. In case of the single core measurement, the data and the code were completely within the 512 Kbyte large L2-SRAM of core 0. This is possible because the architecture permits to use the L2 cache as SRAM. For the distributed version we used the Queue Management Sub System (QMSS) queues to transfer descriptors of transfer packets between the cores. The queues 652 and 653 were used, generating an interrupt when data is

pending on them. The shared transfer packets were located in the Multicore Shared Memory (MSM), constituting 4Mbyte of fast memory shared between the cores. This memory is part of the Multicore Shared Memory Controller (MSMC) , which interfaces the eight cores to external DDR-SRAM. For the single core version we achieve a top throughput of 2695 Mbytes per second using packets with 32 Kbyte payload. The distributed version achieved a maximum throughput of 1752 Mbytes per second with the same payload. In both cases we have not yet reached the saturation of the system, thus the total throughput will be higher, if we increase the packet payload size.

Like for the Intel-SCC we've also implemented a measurement of the virtual bandwidth, using a shared buffer. With a buffer size of 32 Kbyte we achieved a throughput of 772 Mbyte per second @ 1 GHz , when the shared buffer is located in the MSM, and we copied to the L2-SRAM of core 1 (`Virtual Core 0 to 1, MSM to L2'). If the shared buffer is located in the L2-SRAM of core 0 (`Virtual Core 0 to 1, L2 to L2'), the throughput we achieve is 45 Mbyte per second @ 1 GHz. Currently we investigate why the copy between the L2-SRAM of the cores does provide so little throughput.

When utilising the experimental driver for the EDMA3 peripherals of the TI-C6678, and EDMA3 unit we achieve a throughput of 4041 Mbyte per second with a buffer size of 128 Kbyte, transferred between two buffers in the L2-SRAM of core 0. The advantage of using the DMA unit over using the CPU for copying or moving data is that during the transfer the CPU can perform other Tasks, thus the transfer happens in parallel to the processing.

3.2.6 Conclusions from these measurements

The first part of the paper explained the design principles of OpenComRTOS that minimise the impact of the distributed processing on the hard real-time behaviour. Due to being built around the concept of prioritsed packet switching the performance degradation caused by additional middleware layers are avoided in OpenComRTOS systems. This not only results in a better performance, but also in smaller memory requirements, less power consumption and more real- time predictability. The architecture of OpenComRTOS is ideally suited for the multi/many cores systems such as the Intel-SCC and the TI-C6678, because it makes it very easy to use all processing power without having to worry about the details of the underlying hardware.

What has become clear in the performance measurements is that both the Intel-SCC and the TI-C6678 are complex architectures requiring a lot of attention to achieve best performance and predictable real-time behaviour. The developer must be very careful in placing data and code in memory and selecting the communication mechanism. In case of the Intel-SCC the access to the DDR3 memory has a very long latency with a minimum of 86 wait states, and is only available over the system wide shared routing network, which causes additional wait states. The approach taken in the TI-C6678 with a dedicated switching network (TeraNet) provides a much better throughput to the shared memory resources. Additionally, each core has its own 512 kB of L2-SRAM which can be used to store code and local data, an approach not possible in case of the Intel-SCC. A local RAM of 512 kB might sound little but for OpenComRTOS it is more than sufficient, due to its small code size of around 5 kB. This leaves in many cases sufficient space for user applications and device drivers.

The tests have also shown that shared memory presents some pitfalls, similar to the ones global variables represent in multi-threaded environments. Not only makes it the bus structure very complex, it also makes it slow compared with the speed of the CPUs and it poses more safety and security risks, e.g. the cache must also be invalidated at the right time. Therefore, having large and local low wait state memory for each core with a fast dedicated communication network set up in a point-to-point topology with DMAs improves performance, and improves reliability when this memory can be marked as private to the core, thus preventing external cores from accessing and potentially corrupting it. This is an important issue for safety and security critical systems. Finally, multi/manycore designers should be aware that concurrency even on a single core combined with low latency is beneficial as it allows to reduce the grain size of the computations without suffering much overhead. It also increases throughput by overlapping computation with communication.

The communication infrastructure provided by the TI-C6678, with its packetisation and hardware-queue support, is similar to the internal architecture of OpenComRTOS, whereby all interactions are implemented using packet exchanges.

Nevertheless, it is clear that by packing so many functions so closely on a single chip with many of the resources being shared amongst the on-chip CPUs and the influence of the cache operation make it very hard to develop applications with fully predictable temporal properties and low latencies. Hence the conclusion remains the same: the embedded developer must adapt hos programming model from a fully static but no longer predictable in time to a concurrent programing model that allows to absorb the small timing variations that are now unavoidable because of the complexity of the hardware sharing.

4 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.

4.1 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.

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 an use them.

4.2 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

4.3 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 OpenComRTOS. 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]

4.4 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 swiftlytaken, 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. QoS and ARRL

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

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:

- **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.

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-2 or ARRL-3 in the best case 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 then also be reached for the inter node communication mechanism.

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 misfortune or change

5 Conclusion

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.

6 References

6.1 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.altreonic.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. Intel Labs. The SCC Programmer's Guide, 2012. http://communities.intel.com/servlet/JiveServlet/downloadBody/5684-102-8-22523/SCCProgrammersGuide.pdf. 1, 4.4
16. Texas Instruments. TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor (Rev. C). http://www.ti.com/litv/pdf/sprs691c. 1

6.2 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.

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.

Second publication in the Gödel Series:

Systems Engineering for Smarties©