RESEARCH DRAFT PAPER FOR PREREVIEW

Separation of concerns for resilient embedded real-time

Bernhard H.C. Sputh* and Eric Verhulst

*Correspondence:
bernhard.sputh@altreonic.com
Altreonic NV, Gemeentestraat 61A bus 1, 3210 Linden, Belgium
Full list of author information is available at the end of the article

Abstract

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

Keywords: real-time; multicore; RTOS; scheduling; space partitioning; time partioning; ARINC-653; VirtuosoNext

1 Introduction

1.1 What does"real-time" mean?

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

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

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

1.2 What does "priority" mean?

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

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

Flexibility and modularity was introduced by using a more dynamic approach , e.g. by introducing cooperative (often round-robin or time-sliced) scheduling. The best approach however is priority based preemptive scheduling. Using Rate or Deadline Monotonic Analysis [1, 2] one can prove that no deadlines will be missed, provided a maximum CPU workload is respected (typically about 70 percent, but it can be higher), if all tasks are independent and if they are scheduled in order of priority with the priority being assigned as a function of the execution frequency. This is the classical definition of priority based scheduling in the context of embedded real-time. Of course, in practice tasks are seldom independent (as they exchange information) but it works very well in practice. This being said, some people associate priority with "urgency" (as in a "high priority message") but it should be clear that this intuitive notion entails manually changing the execution schedule and hence, it can have undesired side-effects. An important aspect of priority based scheduling is that it decouples the processing logic from the processors it executes on. If the priorities are correctly assigned and if the maximum CPU load is observed as a constraint, then no deadline will be missed as a consequence, even when executing on processors that run slower of faster.

1.3 Real-time and priority on modern multi-core SoCs

Classical time based scheduling is based on knowing the WCET. As no execution time can be longer than the WCET, a schedule can be calculated. Hence, the question is how to obtain the WCET. Traditionally, one can simply execute the program in a loop and measure it. The issue is that this is like testing. How do we know we measured the real WCET or was it a measurement that is still below the WCET? Another approach is to use a very detailed simulation model of the hardware. The question remains essentially the same. Is it the real WCET or the best estimate of it? On modern SoCs, this is practically elusive. Modern SoCs can have 1000's of interrupt sources, I/O logic of various complexity, can have multiple shared buses, DMA engines, multiple memory banks and multiple processing cores. The best performance is obtained by using the caches (who's behaviour is very hard to model) but as they are small, a cache miss is not unlikely wereas external memory often

will be a factor 10 or more slower than the CPU. As a result, WCET measurements are always a worst case estimate and unfortunately, they can easily be a factor 100 worse than the average execution times. In addition, same changes to the code will result in small changes to the execution time. While safety critical systems designs freeze the code once it has been approved, it also means that a single change can result in expensive re-validation and testing, even if the change is small and it effects are well isolated. This makes using WCETs impractical, at least on GHz modern processors. Paradoxically, hard real-time is much easier on much slower MHz microcontrollers.

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

1.4 When is an RTOS really real-time?

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

1. A context switch. On modern processors the context can be extensive (100's of registers not just covering the CPU registers but also a myriad of status registers of on-chip resources such as the MMU). The larger it is, the higher the latency to switch from one task to another.
2. Interrupt handling. On modern processors, interrupt handlers can be very complex with potentially tens of interrupts arriving quasi simultaneously. Handling in order of priority is possible if the hardware has provisions. The need to disable interrupts in critical code sections has an impact on the system latency.
3. Implementing memory management and protection. This can considerably impact on the context switch times.
4. Handling caches. Especially when dealing with peripherals, cache flushing might be needed. This can considerably increase the execution time.
5. RTOS Kernel services. These services allow tasks to synchronise and to communicate. To safeguard the real-time behavior, this should happen in order of priority, so that no higher priority task is kept waiting by lower priority tasks. Priority inheritance is further a must to reduce blocking times. All kernel operations must be strictly bounded in execution time and being independent of e.g. the size of the system. This is the area where most general purpose OS and even some quasi-RTOS fail.
6. I/O handling. With many smart peripherals integrated on the chip (often as black boxes), the timing behavior is often not very predictable.

7 Interprocessor communication. As tasks are distributed over multiple processors (on- or off-chip), the communication can introduce a serious latency. Any delay on one processing node can delay the execution of a task on another node.

8 Respect the implicit priority levels of a real-time application, starting from the hardware (being clocked synchronously, it can never be made to wait). In order of priority (hence in time), interrupts must be handled first and interrupt servicing must be kept as short as possible. The kernel task and its scheduler as well as the driver tasks are the next priority level because any delay at this level, will result in a later execution of the application tasks. And finally, the application tasks. These levels of priority, often go hand in hand with typical time constraints, sub-microsecond for the interrupt handling, microseconds for the kernel and driver tasks, tens of microseconds to several hundred of milliseconds for the application tasks.

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

1.5 Resource blocking, Priority Inheritance and fairness policies

A major issue that haunts real-time applications is the sharing of resources or the presence of long critical sections (that only allow one task at a time). Above section has introduced some of them (like e.g. a context switch) but as the blocking is relatively short, these are generally not considered as critical. Resource locking is typically needed when synchronising with a slower peripheral or to ensure data integrity. The issue is widely described in the literature and was made famous by the Mars Rover reset issue [3]. There is no real solution to it but generally speaking it requires a careful design of the application by minimising the need for resource locking and by the availability of priority inheritance with a ceiling priority support in the kernel scheduler. This reduces the impact of the blocking to a minimum but as said, it remains a best effort strategy. One should also note that modern multicore processors exhibit “milder” forms of resource locking or at least they have functional features that increase the latency. Examples are shared memory (often with multiple wait states), the resulting cache coherency and cash flushing operations but also the multitude of interrupt sources that in the end are all pipelined to a single interrupt to the CPU. And last but not least, as processing chips became faster and memory cheaper, the applications became more complex with more elaborate algorithms. This can result in so-called “greedy” tasks executing converging algorithms that can block other tasks. To reduce this issue, such task must be run at lower priority but with time slicing/round-robin scheduling as a “fairness” policy to avoid blocking other tasks.

1.6 Complexity dictates orthogonality, for logic and for time

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

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

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

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

2.1 If you can't change the hardware, adapt the programming model

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

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

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

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

Note that this model does not consider time as a scheduling parameter but as an external measure (by using e.g. a clock). The time behavior is the result of the programming logic being executed on a given (stochastic) hardware. One can record its progress in time by observing the external clock but the clock itself does not change the execution logic. Note that this programming model is not new. It is largely what all RTOS put forward. It was formally formulated in Process Algebra's like Hoare's CSP [4] , later extended to support timing behavior but this was not really successful. We also note that Leslie Lamport wrote about declaring that time doesn't need a special treatment in programs [5, 6, 7]. It is sufficient to declare a global variable representing it.

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

3 Introduction to VirtuosoNext

The guidelines outlined in previous sections have been applied in the real-time and concurrent programming model of VirtuosoNext Designer. A front end graphical

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

The architecture of the RTOS kernel itself has a long history. The prioritised packet switching architecture was one of its original concepts as it allowed transparent programming of parallel processing targets. The latest versions of the kernel were redeveloped using formal techniques, which resulted in a drastic code size reduction (a factor 5 to 10 depending on the target) and the introduction of the generic Hub interaction entity. The hub concept presents itself as a classical service to the programmer (semaphores, fifos, etc.) but also allows to add very specific services like a proxy hub for using dedicated on-chip logical units. [8]

While the static memory model is beneficial for performance, better safety and security protection is possible by exploiting advanced protection mechanisms like MMUs and MPUs, as available on modern processors. On target processors that allow it, this has resulted in a specific version of the RTOS kernel that isolates each task as well as various memory regions in a fully protected zone. Measurements show that the resulting overhead is very modest as well as in execution time a well as in additional code size. The major impact is on the required data memory as the MMU requires the data sections to be aligned with specific minimum blocks. Priority based preemptive scheduling is still used to achieve (hard) real-time behavior but with an option to assign time constraints (earliest starting time, latest termination time as well as use of CPU cycles) on top of the priority based scheduling. If these conditions are violated they are signalled to the kernel allowing to recover from the run-time faults.

4 Partitioning for safety and security

While we assumed in the beginning of the paper that the software itself is error-free, it still runs on potentially failing hardware in a potentially unsecure environment. If a system is safety critical, the designer must make sure it remains safe and secure, even when faults happen or when the system's security is breached. Both result in erroneous states of the system. How much and what type of measures the designer must take largely depends on the safety and security risks resulting from these fault or breaches. At the system level, we can consider them as failures. Dealing with such failures can be done by applying redundancy and diversity [9]. In order to better utilise the performance offered by advanced multi-core processors, an approach that is often taken is to "partition" the software in space and time on the processor. The goal is to isolate the different partitions so that no fault can propagate beyond the boundaries of its allocated partition and jeopardise the software executing in another partition. It must be noted that such partitioning is only possible on processors with the right hardware support, such as MPUs, MMUs and reprogrammable timers. The

principles of space and time partitioning are outlined below followed by a novel fine grain partitioning approach implemented in VirtuosoNext without jeopardising the real-time support, as evidensed in section 5.

4.1 Space Partitioning

The standard approach for space partitioning is derived from the hypervisor approach, initially developed on server systems allowing to run multiple server applications (e.g. webservers) interleaved in time, which each application running on top of a virtual machine (isolated in space by using the protection offered by the MMU). This approach typically timeslices between the different applications but seriously impacts on the real-time response. In VirtuosoNext, an application is composed of multiple interacting tasks, with each task running in its own memory space, also protected by the MMU, but in order of its priority. This fine-grain task level space partitioning compared to the process or application level space partitioning above has the advantage that it allows a much finer level of partitioning which can be as small as a single line of code, but typically the function providing the task entry point, without jeopardising the real-time capability, an issue that traditional hypervisor type approaches have. In process level space partitioning the data of individual threads of a process are shared, which means they can corrupt each other's data. This is not the case when using the fine grain partitioning support of VirtuosoNext, whereby each task runs in User-Mode and is only permitted to access its own memory (allocated at compile time), as well as explicitly shared memory in the form of global variables. This also prevents direct access to the underlying hardware for which the application task can call the trusted services of the underlying RTOS kernel and its driver layer. As in VirtuosoNext device drivers are implemented as tasks, the user can also develop them in Supervisor-mode.

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

4.2 Time Partitioning

VirtuosoNext does not provide a classical time partitioning (read: time slicing) implementation as seen in hypervisors, instead like its predecessor OpenComRTOS [10] it provides system wide (distributed) priority based preemptive scheduling at all levels. This means that a high priority request from task A on Node 1 for a Service provided at Node 23 will be treated everywhere as a high priority request. This means that with the exception of some memory and scheduling overhead, VirtuosoNext provides the responsiveness of a traditional RTOS, albeit in a distributed implementation and classical scheduling theories remain valid. Rather than allocating fixed and isolating time slots (that put a serious lower boundary on the reaction time of the system), VirtuosoNext provides support for restricting the scheduling in time of tasks on top of the priority based scheduling. For example, tasks can be defined with earliest starting time and latest termination times or with a maximum CPU cycles budget. The kernel task continuously monitors these tasks specific

boundary conditions. Note however, that such boundary conditions are often not needed, unless stringent safety requirements impose them. The rationale for this statement is further elaborated.

4.3 Current practice: ARINC-653

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

4.4 ARM-Cortex-M3

The ARM-Cortex-M3 [11] is a widely used micro controller architecture, typically is clocked at 50 to a few 100 MHz. It provides a simple MPU which allows to specify 8 address regions the task running in user-mode is allowed to access. tasks running in supervisor-mode may always access the whole address space. At runtime it is easy to reconfigure the MPU during a context switch. The downside of this simple MPU is that it requires that the memory regions have sizes of 2^n, and that the starting address is aligned to the size. This causes additional complexity when preparing the linker script for an application as the user must manually properly align the memory regions. In VirtuosoNext we follow a different approach by linking the application twice, the first time to determine the real size of the different regions which is then used by the Section-Analyser tool to generate a linker script where the different memory regions are properly aligned.

4.5 ARM-Cortex-A9

The ARM-Cortex-A9 [12], in our case clocked at 700MHz, provides an MMU which works with two, user maintained, page tables, each table supports a different page-size (4kB, and 1MB). During the context switch the page table is updated to make the pages of the current task inaccessible and the pages of the next task accessible. This is a computationally complex process and thus rather expensive.

4.6 Freescale T2080

The Freescale T2080 [13] consists of 4 PowerPC e6500 cores, clocked at 1.8GHz, of which each provides two threads, thus the SoC provides 8 logical cores. Each

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

4.7 Texas Instruments TMS320C6678

The Texas Instruments TMS320C6678 [14] has 8 physical cores, clocked at 1.25GHz but these provide neither an MPU nor an MMU. However, there are SoC-Level MPUs which can be used to isolate memory regions from cores and peripherals on the SoC. By default every core / peripheral has access to the complete address space, and in the MPUs one explicitly blocks peripherals from accessing a certain region. This is the opposite approach from the other architectures discussed in this paper. Thus on this target the isolation is only possible on the core level. However, the impact is minimal as this only requires an initial setup and no modification afterwards.

5 Results

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

5.1 Semaphore Loop Times

Space partitioning also affects runtime performance because the context of a task now includes also the information about the memory regions it is allowed to access. This becomes visible when comparing the time the system takes to perform a Semaphore-Loop (two tasks, two semaphore Hubs with one loop requiring eight context switches per loop). Table 1 gives the semaphore loop times measured for the different targets for non-partitioned and partitioned implementations, except for the C6678. For the ARM-Cortex-M3 and the T2080 the enabling space partitioning has only a moderate impact of 10% while of the ARM-Cortex-A9 the space partitioning has an impact of 30%.

5.2 Interrupt Handling Latency

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

Note that the interrupt latency is really a histogram as it depends on what other applications are active on the processing node. To simulate such a stress pattern,

a semaphore loop (see next paragraph) is scheduled in parallel with the interrupt latency measurement. The semaphore loop is a very good stress load as it continuously disables interrupts for short interval when the Kernel task executes the semaphore services and context switches.

With space partitioning enabled the IRQ to Task latencies, shown in Table 2, generally increase between the protected and non-protected versions, especially the latency maximum increases substantially due to the fact that this is caused by disabling interrupts while performing a context switch. The context switch for the protected version is usually more complex than for the non-protected version. Similarly, the IRQ to Task latency, shown in Table 3, increases when enabling space partitioning.

5.3 Code Size

The fine-grain space partitioning implementation of VirtuosoNext is lightweight both in code size and in runtime impact. Table 4 shows code size of VirtuosoNext partitioned and non-partitioned. The code sizes were obtained by building the same application using all available Services (compiled with Os). We see that for the ARM-Cortex-M3 and ARM-Cortex-A9 the code size increases by more than 30% while for the T2080 the increment is in the area of 3%. The reason for this is that for the ARM-Cortex platforms a new context switch had to be developed when we implemented the space-partitioning while for the T2080 almost the same context switch is used for both variants.

5.4 Multicore SoCs, T2080 vs TMS320C6678

The TI-C6678 [14] and the Freescale T2080 [13] SoCs have both 8 logical cores. The T2080 has four physical cores supporting two threads each, while the C6678 has 8 physical cores. In case of the T2080 Each core has its private 32kB L1-Data and L1-Instruction Caches, which are shared among the threads, and the L2 Cache is shared among all cores. In contrast the C6678 provides each core with its own L2 cache of 256kB which furthermore can be configured as normal memory. This seriously reduces the memory access overhead and thus improves the real-time capability. On the other hand, the C6678 has less memory protection logic using a MPU, so that the protection is practically at the core level. The T2080 offers fine grain memory protection using its MMU, so that each block of 4kB can be protected.

5.5 The ultimate real-time stress test

In order to verify the approach the interrupt latency test above was modified to use a timer with a 10μs periodicity. This results in a 100% CPU load, taken up for 27.53% by the two tasks running the semaphore loop, the kernel task, a collector task and the timer ISR. Each of the tasks are memory protected. Figure 3 shows a representation of the trace data collected on core-0, while running the measurement. The trace shows the scheduling of the different tasks, with the Kernel-Task being on top, as it has the highest priority and the Idle-Task at the bottom due to it having the lowest priority. The measurements show no degradation of the real-time performance.

5.6 The impact of multicore communication

In an additional stress test, the semaphore loop was distributed over two cores with the timer still at 10 microseconds. This introduces additional latency and CPU load due to the driver tasks (which is transparent for the application developer). The measurement task has the second highest priority after the Kernel-Task. The minimal latencies stayed the same, however the maximal IRQ to ISR latency increased to 3.471μs while the maximal IRQ to Task latency increased to 9.633μs. The long IRQ to Task latency can be explained by the fact that the RX-Part of the communication is handled inside an ISR, which has a highest priority than the Kernel-Task. Figure 4 shows a representation of the trace data collected on core-0 and core-2, while running the measurement.

5.7 The impact of runtime faults

While a safe program should implement its core functions with all necessary guarding logic (e.g. tesing for limit values, unspecified states, etc.) runtime behavior can still make the program fail catastrophically especially as embedded system communicate with the outside world. A typical case is e.g. a divide-by-zero fault. Other faults can be hardware indiced, e.g. by bitflips of bus errors. Such faults will result in processor exceptions, essentially a state to which the hardware is designed to transition to when such an exception occurs. Most often this leaves the currently executing code in an erroneous state from which the only recovery possible is executing a reboot of the program. In VirtuosoNext, one the failing task needs to be restarted. The kernel task will intercept the exception, call a preprogrammed abort handler and restart the task afresh. The developer can recover the previous state from a saved area in memory if this is essential for the application. Tests using a deliberate memory access outside the allowed adresses have show that this allows to recover from such faults with essentially no catastrophic impact. This is a form of temporal redundancy using a spatial duplicate. A typical abort handler is very short and upon termination, the kernel will simple re-initialise the failing task so that the application can continue with no noticeable delay. The recovery delay is measured in microseconds, 2.3 microseconds to be precise on a Freescale QorIQ T2080/1 processor running at 1.8 GHz with all tasks protected in memory.

6 Discussion

7 Conclusions and recommendations for multicore SoC design

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

The concurrent programming model that was put forward to address the issues on the software side is also applicable to the hardware. If CPUs are decoupled (by having large enough local memory and caches), the impact of shared memory is

vastly reduced. Being able to lock code in cache (hence it acts like fast SRAM), greatly improves the performance as well as the statistical execution spread. A side conclusion is that it pays of to have a smaller code size.

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

While the tests have confirmed the wide variance in real-time performance of different hardware architectures, we have also shown that applying good design principles in developing the RTOS can largely bridge the gap, even on processors that are less suitable for real-time. Overall by pursuing a programming model that implements it to its logical consequences in the RTOS design on modern processors we have shown that hard real-time can be combined to achieve more resilience at system level than is possible with the tradional approach.

Competing interests
The authors declare that they have no competing interests.

Author's contributions
Text for this section ...

Acknowledgements
This work is supported by the EUROCPS NoFiST (Novel Fine Grain Space and Time Partitioning for a Mixed Criticality Platform) project.

References

1. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM **20**(1), 46–61 (1973). doi:10.1145/321738.321743
2. Audsley, N.C.: Deadline Monotonic Scheduling (1990)
3. Risat Mahmud, P.: Mars Pathfinder: Priority Inversion Problem (2014)
4. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1985)
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978). doi:10.1145/359545.359563
6. Lamport, L.: "sometime" is sometimes "not never": On the temporal logic of programs. In: Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '80, pp. 174–185. ACM, New York, NY, USA (1980). doi:10.1145/567446.567463. http://doi.acm.org/10.1145/567446.567463
7. Lamport, L.: In: Borrione, D., Paul, W. (eds.) Real-Time Model Checking Is Really Simple, pp. 162–175. Springer, Berlin, Heidelberg (2005). doi:10.1007/11560548_14
8. Sputh, B.H.C., Verhulst, E., Mezhuyev, V.: OpenComRTOS: Formally developed RTOS for Heterogeneous Systems. In: Embedded World Conference 2010 (2010)
9. Verhulst, E., Sputh, B., Van Schaik, P.: Antifragility: systems engineering at its best. Journal of Reliable Intelligent Environments **1**(2), 101–121 (2015). doi:10.1007/s40860-015-0013-3
10. Verhulst, E., Boute, R.T., Faria, J.M.S., Sputh, B.H.C., Mezhuyev, V.: Formal Development of a Network-Centric RTOS. Software Engineering for Reliable Embedded Systems. Springer, Amsterdam Netherlands (2011)
11. ARM: ARM Cortex-M3 Processor Technical Reference Manual, Revision r2p1 edn. (2015). ARM
12. ARM: ARM Cortex-A9 Processor Technical Reference Manual, Revision r4p1 edn. (2012). ARM
13. NXP: QorIQ T2080 Reference Manual. NXP. Document Number: T2080RM Rev. 3, 11/2016
14. Texas Instruments: TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor (Rev. C). Texas Instruments. http://www.ti.com/lit/ds/symlink/tms320c6678.pdf

Figures

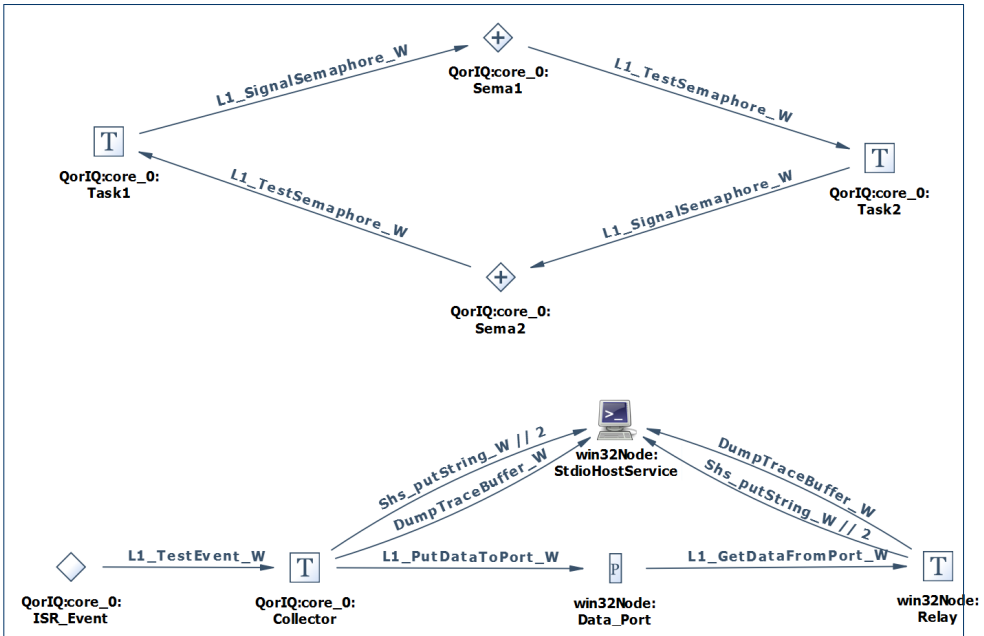

Figure 1 Interrupt Latency Measurement Application Diagram

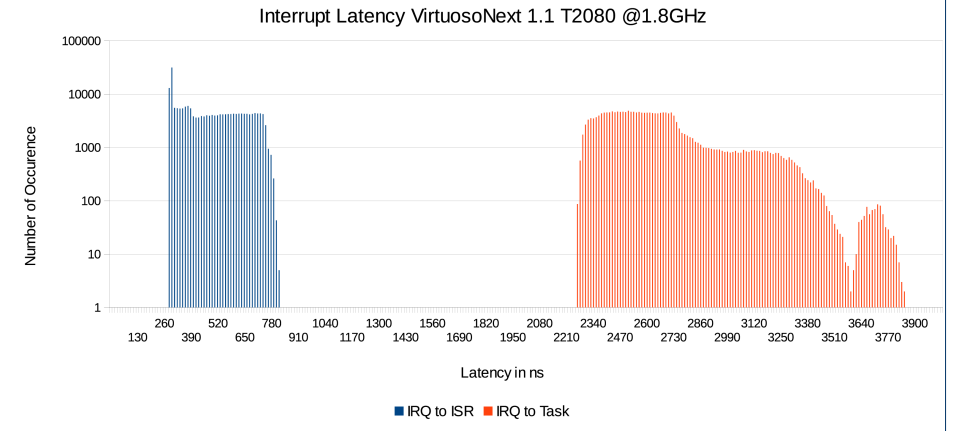

Figure 2 Interrupt Latency Histogram on Freescale T2080 @ 1.8GHz (logarithmic scale).

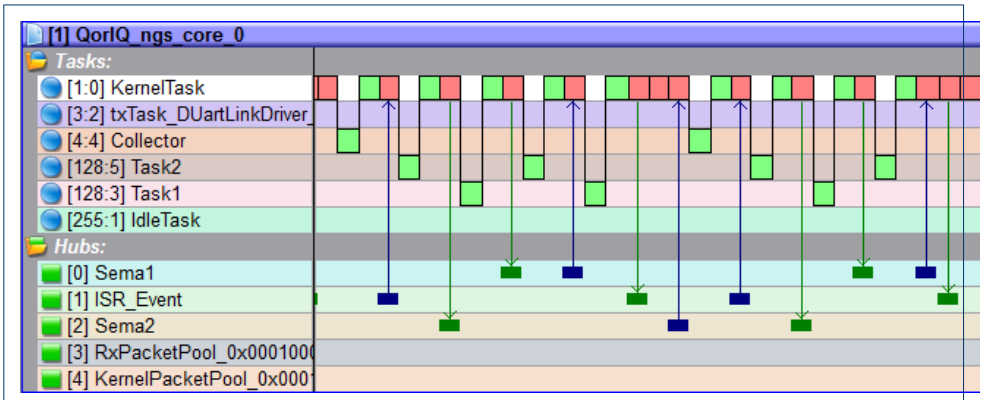

Figure 3 Trace of the Interrupt Latency Measurement with a 10μs ticker and a single CPU Semaphore Loop.

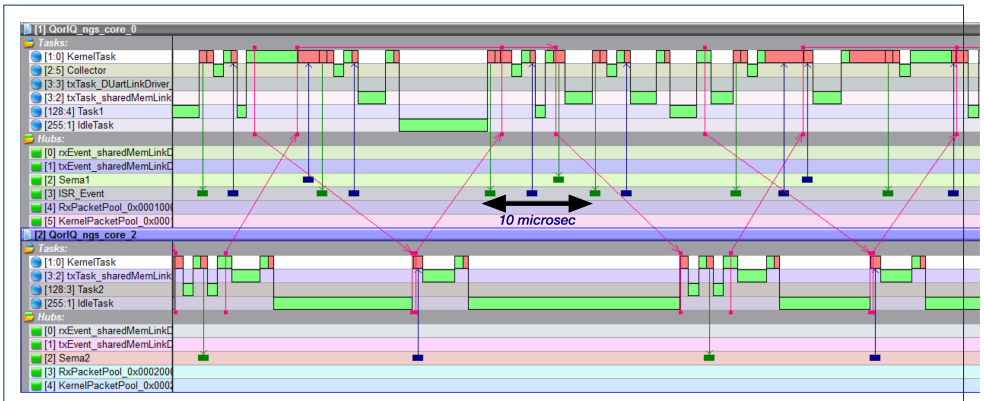

Figure 4 Trace of the Interrupt Latency Measurement with a 10μs ticker and a Semaphore Loop distributed over core-0 and core-2.

Tables

Table 1 Semaphore Loop Times, non-partitioned and partitioned.

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

Table 2 Minimal and maximal IRQ to ISR Latencies, non-partitioned and partitioned.

Target	Non-Partitioned	Partitioned
ARM-Cortex-M3 (@50MHz)	780ns – 2,500ns	960ns – 4,920ns
ARM-Cortex-A9 (@700MHz)	100ns – 314ns	138ns – 1,150ns
C6678 (@1.25GHz)	160ns – 260ns	NA
T2080 (@1.8GHz)	286ns – 793ns	286ns – 819ns

Table 3 Minimal and maximal IRQ to Task Latencies, non-partitioned and partitioned.

Target	Non-Partitioned	Partitioned
ARM-Cortex-M3 (@50MHz)	15μs – 35μs	16.00μs – 39μs
ARM-Cortex-A9 (@700MHz)	0.994μs – 2.182μs	1.726μs – 4.228μs
C6678 (@1.25GHz)	0.936μs – 1.728μs	NA
T2080 (@1.8GHz)	2.158μs – 3.705μs	2.262μs – 3.848μs

Table 4 Code size, non-partitioned and partitioned.

Target	Non-Partitioned	Partitioned
ARM-Cortex-M3 (@50MHz)	8,656B	11,564B
ARM-Cortex-A9 (@700MHz)	15,144B	21,844B
C6678 (@1.25GHz)	26,448B	NA
T2080 (@1.8GHz)	37,224B	38.504B