# Towards Collaborative Resource Sharing under Real-Time Conditions in Multitasking and Multicore Environments

Marcel Baunach
University of Wuerzburg, Germany

# Towards Collaborative Resource Sharing under Real-Time Conditions in Multitasking and Multicore Environments

Marcel Baunach
University of Würzburg, Germany
Chair for Technical Computer Science
baunach@informatik.uni-wuerzburg.de

## Abstract

*Today's embedded system designs demand for an ever increasing integration density of various services on common platforms. Especially within highly dynamic environments where severe real-time demands must be met, sharing exclusive resources among these concurrently running subsystems is a hard compositional problem. Classic approaches suffer from the fact that tasks or cores are not aware about their mutual influences and existing resource conflicts, and thus cannot collaborate efficiently in critical situations. We eliminate this flaw, and present the novel DynamicHinting method for sharing exclusive resources on-demand among prioritized tasks on both common and different cores: Hints will be issued by the resource manager(s) to indicate e.g. priority inversions, and to provide blocking tasks with the time, knowledge, and CPU power to resolve the conflicts in time.*

## 1. Introduction

The still increasing pervasiveness and ubiquity of today's embedded devices comes along with rapidly increasing demands on the underlying hardware, software, and networking subsystems. Driven by strong market pulls various (emerging) technologies will continue to draw our special research attention regarding quite common and quite diverse requirements: While Gartner [9] just identified e.g. location based services and augmented reality as "highly beneficial" throughout the next decade, autonomous vehicles and mobile robotics are even more long-termed but considered to be "transformational" then.

In our opinion, turning the associated expectations into reality demands for a symbiotic research in two main directions: Wireless Sensor Networks (WSN) and Embedded Control Systems (ECS). The first rely on rather resource constrained but cheap and autonomously operating devices: Deployed in arbitrary numbers they take the part of remotely monitoring and interconnecting the environment as well as the participating systems through sensors and robust communication concepts. The latter, in turn, make use of significantly more powerful and application-tailored hardware to finally interact with the environment through various types of actuators and adaptive control algorithms.

The distinct challenges thus relate to optimizing energy consumption, service coverage, or sensor data aggregation for the WSN domain, and to safety, security, or computational power for the ECS domain. The common challenges, however, result from the highly dynamic environments in which these distributed or even (partially) mobile systems are intended to operate: Periodic as well as sporadic events demand for a high reactivity to reflect both soft and hard real-time constraints. Especially in the medical, military, and avionics sector, their violation might not only lead to simple system failures or quality degradations, but also to severe consequences for humans and equipment. Further enforced through the demand for compositionality and modularity regarding various co-existing services and software subsystems on a single platform, the most central problem can in general be identified as a resource sharing issue among concurrently running "jobs" with time-critical objectives. This is exactly where DynamicHinting applies.

## 2. Related work: Integration and isolation

While a reasonable system design is always required first, various solution approaches exist to counteract the just mentioned compositionality problem at various implementation levels: On the software side, these strategies often rely on *preemptive operating systems* to *integrate* prioritized tasks, and to support dynamic resource sharing at runtime. The developers (and the tasks, respectively) are commonly aware of the fact that they have to share e.g. the CPU and other operational resources, and thus make use of e.g. static preemption points [6, 19] or dynamic priority inheritance strategies [16] to reduce priority inversions [20] or to avoid/prevent deadlocks. In contrast, *virtualization and hypervisor techniques* attempt to largely *isolate* simultaneously hosted (operating) systems through static resource assignments [8, 10]. At best, the hosted guest systems do not even know about each other then. Unfortunately, the aim for "true independence" is not always attainable if sharing truly unique resources (like e.g.

memory, IRQ controllers, peripheral buses, etc.) cannot be avoided. Thus, partitioning techniques like [14] periodically reassign sets of resources to task groups. The so called paravirtualization even partially revokes the idea of a strict isolation, and puts some concurrency awareness into specifically adapted kernels [5, 17]. However, sharing resources among them remains hard in general, and only a few (and mostly commercial) concepts exist [12, 18, 15]. This is especially true for hard real-time operation where a far-reaching or even strict isolation is often a prerequisite for the certification of mission critical systems [1].

Regarding most hypervisor architectures, well-known concepts from traditional OS resource managers can often be found adapted at the virtualization level: For instance, explicit resource requests can be passed through the kernel using so called hypercalls to drivers which reside within the hypervisor. Another strategy lets one server task or hosted guest OS acquire a particular resource exclusively to provide corresponding services. At runtime, these will be accessed through unified communication channels, like e.g. shared memory or virtual ethernet. Apart from the management overhead and the indeterministic response times for both techniques, task or service request priorities commonly have no meaning across the guests' borders, and even non-detectable priority inversions might eventually occur. For multicore architectures, these might even evolve and persist across the cores.

## 3. Scope of this work

With regard to the real-time demands in open architectures with both time-critical *and* non-time-critical subsystems, we present an entirely novel collaboration concept for sharing exclusive resources among tasks and cores in multi-tasking and multi-core environments. The central idea behind our approach combines time-awareness and situation-awareness to express (unavoidable) resource dependencies, and to resolve related conflicts on-demand. Additional contracts among the tasks or cores ensure worst case allocation times (WCAT) where required:

In short, low priority tasks get notified through so called *hints* as soon as they block at least one task with higher priority. While these tasks might even reside on separate cores and within separate OS instances, specific time-utility-functions (TUF) [13] can then be applied by the blocking task to decide from existing contracts and other application specific parameters about whether to ignore the hint or to collaborate by releasing the resource in question prematurely. In any case, (except for the CPU on each individual core) resources will never be withdrawn, but must always be released voluntarily by their current owner tasks.

With respect to the consequently required hardware/-software co-design, the scope of this contribution is twofold:

1. Regarding the purely software based integration of concurrently running code on a single core, we will first present the embedded real-time operating system *Smart*OS in Section 4. In combination with the central DynamicHinting concept from Section 5 its special focus aims on providing a unified temporal semantics and a resource related dependency awareness for fully preemptive and prioritized tasks.

2. Regarding the hardware-assisted virtualization of such exclusive resources among multiple OS instances on several cores, an additional hardware component is proposed in Section 6 to natively extend our collaboration concept: Designed as a central resource manager, it forwards the resource related dependency awareness between the cores to carefully breach their initially strict isolation just where and only when really required.

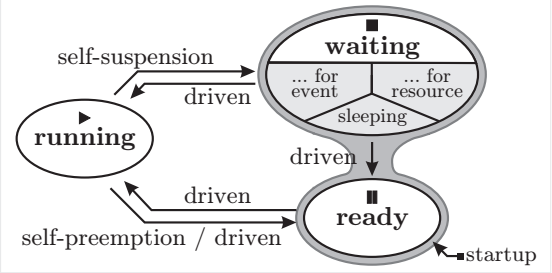



Figure 1: The *Smart*OS task states

In the remainder of this paper, we consistently present the technical concepts for generating and receiving hints in hardware and software. In particular, we will also discuss the impact on the application design and implementation, and show some selected real-world test bench results.

## 4. The *Smart*OS operating system

Since the theoretical and algorithmic background on *Smart*OS has already been introduced in [3], this section will just provide a brief introduction on the most central kernel concepts (i.e., time, tasks, events, mutexes, semaphores, exceptions, and IRQ handlers):

Designed for both small MCUs (WSN domain) and full-grown CPUs (ECS domain), *Smart*OS features a hybrid exo/micro kernel architecture, and incorporates several advantages of both classes. While it supports arbitrary resource allocations (exo), it provides native abstractions only for a small set of system components (micro). Most drivers must thus be provided by the application.

Each application is organized as a system of concurrently running and fully preemptive **tasks** with individual stack areas and dynamic *base priorities*. According to Figure 1 these tasks always transit between three states: running (i.e., executed), waiting (i.e., suspended), and ready (i.e., preempted), with ready being the initial state. Atomic sections are intentionally not allowed for their unpredictable duration. Though the tasks are initially

independent from each other, they may develop dynamic dependencies at runtime (e.g. through resource sharing or inter-task-communication): With respect to these dependencies the scheduler consequently provides various synchronization primitives and also maintains a dynamic *active priority* for each task: The tasks with highest active priority in ready state will be selected for execution in either round-robin or cooperative manner.

Explicit inter-task communication and environmental interaction is provided via so called **events** which can be invoked by the tasks and by application specific **IRQ handlers**. While the initial interrupt acceptance is always centralized at the kernel level, their actual handling is commonly requested through events and accomplished at task level.[1]

Inter-task synchronization is provided through so called mutexes and semaphores: While **mutexes** are available to protect code sequences from interleaved execution (critical sections), **semaphores** coordinate the dynamic access to temporally shared but exclusively assigned resources. Once assigned, these will never be withdrawn from a task, but must always be released voluntarily. The resource manager supports physical resources (e.g. buses) as well as abstract resources (e.g. data structures) under both long-term and short-term allocation. In the scope of reactive embedded systems we consider both types to be indispensable for the operation in highly dynamic environments: *Long-term allocations* allow tasks to suspend themselves while holding a resource (e.g. allocate a transceiver and wait for a related IRQ). In contrast, *short-term allocations* will operate on resources without intermediate self-suspension (e.g. lock a data structure just for immediate processing). In any case, the *Smart*OS resource manager coordinates pending allocation requests with respect to each involved task's priority, and applies modified priority inheritance protocols (PIP and PCP) for collaborative resource sharing as described next within this paper.[2]

Another considerable concept in the domain of embedded operating systems is the native support for task specific **exception** handling. This allows reacting on unforeseeable system conditions, and to separate the program logic from the error handling. In the tradition of higher level languages, the kernel API allows emulating the typical try/catch structure as shown in Listing 3 at the cost of some task stack for saving its state at the beginning of each (nested) try block. In the special context of collaborative resource sharing we'll also see these exceptions useful for synchronizing the resolution of asynchronously emerging resource conflicts to the execution flow of the blocking task in Section 5.

Finally, **time** and time-awareness is another inherent design concept in *Smart*OS, and the most central foundation of any application software: Apart from a local system time with a unified resolution of 1 µs the kernel provides a highly precise IRQ timestamping mechanism with a symmetric precision interval around the actual IRQ trigger. The API's temporal semantic forwards the notion of time to the application tasks, and provides non-blocking versions of all kernel functions which might not complete immediately: Sleeping is equally supported as the specification of absolute deadlines or relative timeouts for waiting on events or for bounding resource requests.

From the operating system's point of view each application's reactivity initially relies on best effort scheduling with respect to the task priorities. Since real-time violations can always be traced back to (unresolved) resource sharing conflicts, additional reactivity demands can be specified at application level (contracts), and will reliably be satisfied by limiting the worst case allocation time of resources via DynamicHinting (collaboration). As an extensive example, the special use case of dynamic memory management under hard real-time conditions can be found in [4].

## 5. Collaborative resource sharing among tasks

Starting with the software integration, this section presents the *Smart*OS hint generation and hint passing policy within a single kernel instance, and introduces a novel way to handle the indicated resource conflicts efficiently with respect to the current task and system situation. From the various compositional challenges we will just address the problem of deadlocks (where required) and bounded or unbounded priority inversion: Here, a low priority task blocks the execution of a higher priority task through a lasting resource allocation which cannot be withdrawn.[3]

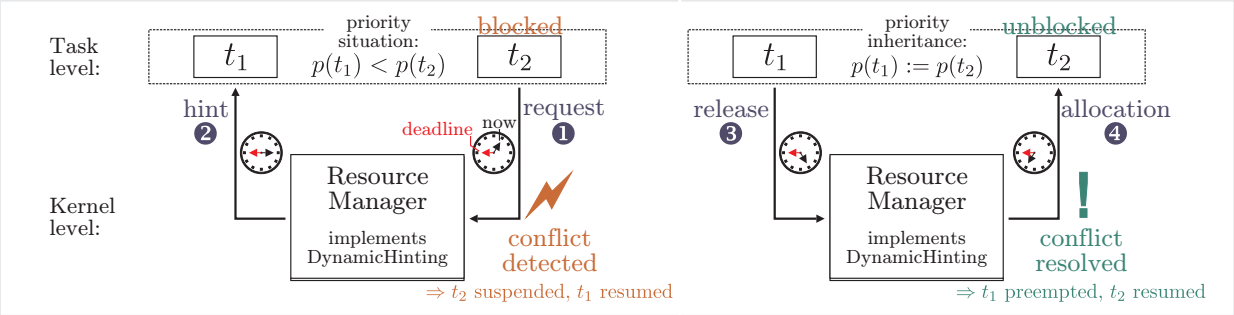
Classic approaches to solve this problem comprise e.g. HLP (highest locker protocol), PCP (priority ceiling protocol), PIP (priority inheritance protocol) [16, 7], or SRP (stack resource policy) [2]: While these either implicitly or explicitly raise the priority of every blocking task to at least the highest priority of the tasks it currently blocks, the inherent flaw of this common strategy is, that the tasks will not realize their own impact on the remaining system! Thus, no concept is able to resolve bounded priority inversion, which can be a true killer scenario for any time critical but currently not grantable resource request. Beyond, each concept also has its specific weakness:

For its relatively simple implementation overhead and deadlock avoidance capability, HLP is often found in effectively used operating systems. However, it does not only lack the ability to interleave equal priority tasks, but

[1] Though not discussed here, this unification provides a deterministic handling latency.

[2] True resource access protection is initially not available due to missing hardware assistance on most supported COTS microcontrollers (like e.g. TI MSP430, Renesas SuperH, Atmel AVR).

[3] In 1997, the Mars Pathfinder mission had almost failed because of such a priority inversion on an inappropriately shared data bus [11].




(a) Resource allocation attempt: $t_2$ calls getResource($R$, $\Delta_{\text{DL}}$)

(b) On-demand resource handover: $t_1$ calls releaseResource($R$)

Figure 2: A delayed resource allocation, and related task/kernel interactions

also prohibits self-suspensions during lasting resource allocations. Thus it "blocks on preemption" and the resulting "inheritance related starvation" of lower priority tasks renders it unsuitable for long-term allocations.

Within the more elaborate PCP a lasting allocation by a task does not starve lower priority tasks. The policy "blocks on request", but applies a very restrictive resource assignment policy to maintain a so called "safe state" for deadlock avoidance. Though quite desirable, this feature also renders the approach unsuitable for long-term allocations, since these are likely to immediately result in the "avoidance related rejection" of further requests even for currently free resources. For the same effect SRP is also not suitable for long-term allocations, since its specific stack sharing feature would be corrupted then. PIP is more generous in general, and it will never reject requests for currently free resources. Thus, it is the most appropriate candidate for long-term allocations. However, as a considerable drawback, it suffers from so called "chain blocking", and even deadlocks might occur. Yet, both problems will be solved by DynamicHinting as addressed in [3]. Further considerations as well as a discussion of more synchronization alternatives can also be found there.

In contrast to these classic techniques, our novel collaboration concept establishes an additional conflict-awareness at task level to let even intrinsically exclusive resources become preemptive on-demand. In compliance with the exokernel philosophy, the central contribution of DynamicHinting is to provide tasks with direct access to exclusive but temporally shared resources. In terms of the allocation latency, it therefore aims on closely reflecting their base priorities as defined by the developer or as adapted at runtime. However, just indicated by the resource manager, the ultimate decision between resolving a dynamically emerging resource conflict either immediately, with some tolerable delay, or not at all, is shifted from OS level to task level (or from hardware level to OS level in Section 6). Though the resource manager detects the conflicts first, the reason for this intentional relocation of competencies is, that the current resource owner has the most complete knowledge about the consequences of an early release *and* about how to terminate a resource properly (i.e., with the fewest side-effects). Eventually, such an early release and on-demand handover can significantly reduce bounded priority inversions and resolve deadlocks. Finally, keeping the allocation delays roughly proportional to the task base priorities means an entirely new improvement for time-critical open systems which do not withdraw (temporally exclusive) resources by force.

### 5.1. Usage and example scenario

Exemplified by an initially not satisfiable resource request, Figure 2 shows some typical task/kernel interactions within our collaborative concept to still provide the allocation in time: First, the high priority task $t_2$ requests a resource $R$ from the resource manager (❶). Unfortunately the request cannot be served immediately due to the lasting allocation of $R$ by another task $t_1$. However, since $t_2$ bounded the request by the specification of an absolute deadline $\Delta_{\text{DL}}$ it initially remains blocked in suspended/waiting state. Meanwhile, the resource manager identified the blocking task $t_1$ with a lower active priority $p(t_1) < p(t_2)$. Within our example the applied priority inheritance policy will immediately raise $t_1$'s active priority to $p(t_1) := p(t_2)$. Next, $t_1$ will be resumed to running state, and receives a hint (❷) which indicates its disturbing influence on the system.

Since according to the *Smart*OS specifications resources are always exclusive and non-preemptive, only $t_1$ itself is authorized to instruct or perform modifications to its allocated resource $R$. Being aware of the resource conflict, the received hint is evaluated by $t_1$, and possibly triggers a self-controlled release of $R$ (❸). Most important, this will commonly include an appropriate finalization of the resource usage: A shared bus for example might require the proper termination of a currently running stream as well as the deactivation of an autonomously operating on-chip peripheral (e.g. a DMA controller) which would otherwise continue to transmit data over the already released bus. Under guidance of the resource manager, the entire operation finally leads to a resource handover which allows to serve and unblock $t_2$ (❹). Of course, $t_1$'s priority will also be adapted (i.e., reduced) again according to the applied priority inheritance method. Regarding poten-

```
1  OS_DECLARE_TASK(T, 200, 100); // stack size = 200 words
2                                // base priority = 100
3  OS_TASKENTRY(T) {
4
5    /* prepare for catching an exception */
6    Exception_t e;
7
8    /* lock a resource in a long-term allocation */
9    getResource(&R);
10
11   /* use the resource infinitely unless another
12      task demands for it */
13   while (1) {
14
15     TRY {              // prepare to react on hints
16       ADCFunction();   // throws EX_EW directly
17       DSPFunction();   // throws EX_HH via hint handler
18
19     } CATCH (e) {      // synchronize on hints
20       switch (e) {     // exception specific code ...
21         case EX_HH:    // ... for finalizing ...
22         case EX_EW:    // ... the resource usage
23       }
24       releaseResource(&R);  // collaborate
25 /* THE TASK T WILL DEFINITELY BE PREEMPTED HERE SINCE
26    THE RESOURCE R WILL BE HANDED OVER TO SERVE AND
27    RESUME A STILL BLOCKED TASK WITH HIGHER PRIORITY */
28       getResource(&R);      // re-allocate ASAP
29     }
30
31   }
32 }
```

(a) A task $T$ with hint handling capability

```
33 /* A hint handler which can be injected into a
34    task execution flow */
35 OS_DHHANDLER(HH) {
36   /* conditional collaboration (contract based) */
37   Resource_t *hint = getHint();
38   if ( TUF(hint, contract) == COLLABORATE ) {
39     THROW EX_HH;
40   }
41 }
42
43
44 /* Some CPU intense function which can
45    nevertheless be interrupted by a hint */
46 void DSPFunction() {
47   setDHH(&HH, φ);   // enable hint handler
48   someHighCPULoad();
49   setDHH(NULL);     // disable hint handler
50 }
51
52
53 /* Some ADC sampling function.
54    Sleeps for 10ms between the samples,
55    but can wake up early for hints */
56 void ADCFunction() {
57   while (someCondition) {
58     sampleSensorData();
59     if (sleep(10000, φ) == HINT) {
60       /* unconditional collaboration (always) */
61       THROW EX_EW;
62     }
63   }
64 }
```

(b) Hint reception and synchronization via exceptions

Figure 3: Using *Smart*OS exceptions to synchronize on dynamic hints

tial real-time constraints for $t_2$, a previously defined contract might have engaged $t_1$ to react on the hint before $t_2$'s allocation deadline has been reached.

### 5.2. Receiving and processing hints

As we have just seen from the example, the combination of temporally bounded resource requests, dynamic hints, optional contracts, and the priority inheritance protocol provides a blocking task with the necessary time, knowledge, objective, and CPU power to resolve the situation in the most appropriate way.[4] However, apart from this conceptional point of view, the technical realization bears some problems: Since hints can possibly emerge during each resource request of a currently running task, a potential blocker task can then itself be in either ready or waiting state (see Figure 1). If the blocker is even executed on another core, it can also be in running state then. From the blocker's point of view, and comparable to interrupts, hints must thus be passed and processed asynchronously. While DynamicHinting supports three techniques to notify a *Smart*OS task, the impact on the actual task implementation depends on the application design, but can be unified and synchronized through exceptions.

For a comprehensive introduction of the three hint notification and reception strategies we refer to Listing 3:

1. **Explicit querying** allows a task to explicitly and intentionally poll the resource manager for a persisting resource conflict. While this option is rather inconvenient to apply, it is likely to pollute the code, and it generates high CPU load if a high reactivity on hints must be ensured. In fact, it is only recommended to query the actual hint in case the task is already aware of its spurious influence (→ Line 37).

2. **Early wakeup** will resume a blocking task in waiting state early. Related functions like e.g. sleep, getResource, or waitEvent provide a special return value to distinguish between success, failure, or hint. Besides, they allow the specification of a priority threshold $\varphi$, and will only accept a hint in case the blocked task has equal or higher priority (→ Line 59). While periodic polling is avoided entirely now, the CPU load will decrease significantly and the impact on the code implementation is minimized.

3. **Hint handlers** are comparable to IRQ handlers, and will immediately be injected into a blocking task's execution flow. This option is particularly suitable for tasks with high CPU utilization and infrequent self-suspensions. In contrast to IRQ handlers, the hint handlers are task specific, and can be (de)activated or changed arbitrarily at runtime (→ Line 47, 49). While the specific priority threshold $\varphi$ behaves analogous to before, the nested execution of hint handlers can also be (de)activated but will consume additional stack space. Apart, the handlers

[4] Though PCP would in general be compatible with our collaboration concept, we recommend to combine it with PIP due to its better support for long-term allocations.

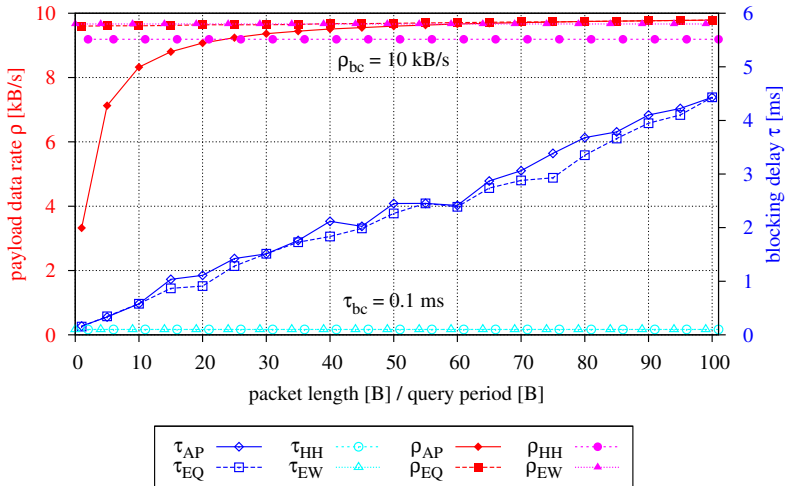



Figure 4: Streaming test: Atomic packets (AP) vs. DynamicHinting (EQ/EW/HH)

will always be executed in the context of the task for which they are called, and thus have the same access rights on the task's currently allocated resources.

Once received, the hint can either be handled entirely within the hint handler or in response to the early wakeup. However, a centralized processing might be more convenient in many cases, and thus we recommend to only make the decision about the collaboration immediately, and possibly throw an exception for a synchronized resource deallocation. Once more, Listing 3 gives an example: Encapsulated in an infinitely executed *Smart*OS try block (→ Line 15) the task $T$ calls two functions with different CPU utilization characteristics: The ADCFunction (→ Line 56) sleeps frequently and thus makes use of early wakeup. The DSPFunction (→ Line 46) generates high CPU load and thus registers a hint handler first. On its execution, the hint handler evaluates both the hint and a contract by means of a time-utility-function (TUF) [13] in Line 38. In case of no collaboration, it will simply return, and the task will continue where the handler has been injected before (i.e., somewhere between the lines 47 and 49). Otherwise the exception will bring the task execution back to the innermost catch block in Line 19. Although the early wakeup in Line 59 will always collaborate in our example, the exception it possibly throws has the same effect: Independent from the code position where the hint has been received asynchronously, the catch block will synchronize this special situation and finalize the resource usage in Line 20. The subsequent deallocation in Line 24 will immediately hand the resource over to the blocked task which caused the hint. This other task will be resumed immediately then, and our task from Listing 3 will consequently be preempted. Yet, on its next resumption it will re-request the voluntarily released resource in Line 28 and continue or start over.

### 5.3. Use case: A shared field bus

This section presents some test bed results from one of our real-world WSN applications: Here, a sensor task attempts to continuously stream captured and preprocessed data over an exclusively shared peripheral bus. Even though it strives for a high data rate, it operates the bus at "best-effort". Besides, it has a lower base priority than two concurrently running time-critical tasks. These are event-driven and require the same bus to sporadically access a radio transceiver and a stepper motor driver: Violating their real-time demands has to be avoided carefully, since this could easily lead to lost radio packets and impaired actuator control characteristics.

Figure 4 shows both the average bus allocation and task blocking delay $\tau$ for the real-time critical tasks, and the achieved data rate $\rho$ for the sensor task's stream when using different implementation strategies. The achievable best case values (no resource conflict, no stream interruption) are indicated as $\tau_{\text{bc}}$ and $\rho_{\text{bc}}$, respectively.

1. First we implemented the sensor task to divide the stream into atomic packets (AP), and to release the resource after each packet transmission: While an increasing (decreasing) packet length resulted in less (more) management overhead and an increased (decreased) data rate $\rho_{\text{AP}}$, it also increased (decreased) the allocation delay $\tau_{\text{AP}}$ for the time-critical tasks. Finding a reasonable trade-off for selecting an adequate packet length proved to be hard.

2. Using explicit querying (EQ) to only release the bus after each packet when really required, already improved the situation: The resource management overhead was kept at its minimum now, and the data rate $\rho_{\text{EQ}}$ could consequently settle close the achievable best case $\rho_{\text{bc}}$. However, the continuous polling for a hint still generated considerable CPU load, and the frequency still affected the allocation delay $\tau_{\text{EQ}}$ depending on the packet length.

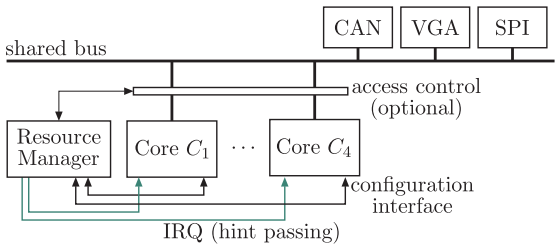



Figure 5: The DynamicHinting multi-core setup

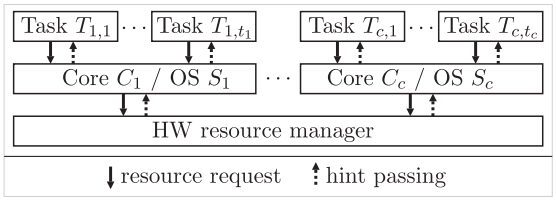



Figure 6: The hierarchical DynamicHinting setup for multi-core and multi-tasking systems

3. Following the general idea from Listing 3, we implemented both a hint handler (HH) *and* early wakeup (EW) for the sensor task. Since this combination did immediately generate a hint as soon as the bus was requested by a real-time task, we did not have to divide the stream into atomic packets any more. Instead we simply sent a trailer to indicate a stream termination on-demand, and released the bus thereafter. This approach kept both the resource management overhead and the CPU load of the sensor task minimal, and it even simplified the implementation of the task logic. As a result, both the data rates $\rho_{\text{EW}}$ and $\rho_{\text{HH}}$ did stabilize close to the achievable maximum $\rho_{\text{bc}}$, and the allocation delays $\tau_{\text{EW}}$ and $\tau_{\text{HH}}$ could also be kept close to the achievable best case $\tau_{\text{bc}}$.

## 6. Collaborative resource sharing among cores

According to the integration of DynamicHinting into the *Smart*OS kernel, we also extended this concept to even lower system levels by providing a hardware component for coordinating the shared access on exclusive resources among various CPU cores. Our test implementation makes use of a Spartan-3A FPGA where we synthesized our resource manager and four cores of a simple 32 Bit CPU architecture with a five-stage instruction execution cycle. Figure 5 gives an overview:

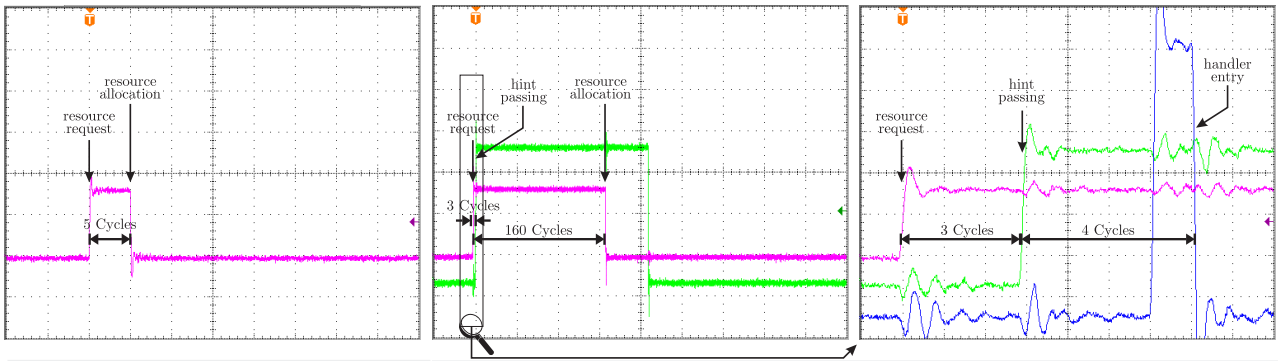
The extended CPU instruction set provides an `OUT a,v` instruction which communicates with the resource manager over its (multi-port) configuration interface: Depending on the specified address `a` a core can (de)activate DynamicHinting entirely for the specific resource which is encoded in `v`, request and release it, or query the resource's allocation state. In turn, the resource manager makes use of core-specific IRQ signals to indicate an emerging resource conflict by passing a hint to a currently blocking core. When used in the context of a virtualization solution, the OS on the specific core must thus be aware of this communication channel. In fact, this complies perfectly to the centralized IRQ acceptance under *Smart*OS ($\rightarrow$ Section 4). Comparable to the hint selection within the OS kernel, the destination core of the IRQ depends on the cores' priorities. These can either be fixed, or be derived from the priority of the currently running task on the specific core. While the first option would probably be used for most system architectures with far-reaching isolation, the latter can be used to make the selected priority inheritance technique transparent across the core borders. Then however, the OS is once more reliable for passing (retrieving) this priority information `v` to (from) the resource manager via an `OUT a,v` command to yet another dedicated address `a`.

Processing a hint IRQ can be done in two ways: If the OS itself is the owner of the hinted resource, or if it can withdraw resources from the tasks, it might terminate their operation and return them for a handover. Otherwise, e.g. for *Smart*OS, it forwards the hint to the current resource owner task, and returns the resource to the hardware resource manager as soon as the task has released it. While this hierarchical approach is depicted in Figure 6, forwarding and processing the hint is done as described in Section 5. Again, contracts can be negotiated by the operating systems and the tasks to define temporal resource (de)allocation boundaries.

Up to now – and comparable to the implementation of DynamicHinting within the *Smart*OS kernel – the hardware resource manager is also used for coordinating the access to resources only; but not for their protection. While this was acceptable for the OS design where we had to support hardware without appropriate protection mechanisms, this deficit can be solved now: If the resources are e.g. attached to a shared bus, as depicted in Figure 5, an optional access control unit can be provided by the hardware resource manager to control and possibly seal off any unauthorized access.

### 6.1. Use case: A shared field bus

Comparable to the purely software based stream test from section 5.3, we configured the four CPU cores $C_1 \ldots C_4$ from Figure 5 to have increasing priorities. While core $C_1$ generated frequent but low priority output to the shared SPI bus (e.g. for a best effort stream), the core $C_3$ sporadically had to transmit more relevant information over the same bus. Figure 7 outlines the core interactions and shows performance results: As expected, the allocation of a currently free resource takes exactly 5 cycles on our five-stage CPU, since issuing the corresponding `OUT` command means nothing else than an ordinary CPU instruction. In case of a currently allocated resource it takes 3 cycles to have the hardware resource manager trigger the hint IRQ, and another 4 cycles to let the tar-




(a) Allocation of a free resource (minimal delay)

(b) Allocation of a currently allocated resource (conflict resolution)

Figure 7: Hardware support and core interactions for allocating a shared resource. Delays are given in CPU cycles.

get core execute the first instruction of the IRQ handler. The values so far are constant for our hardware implementation. In contrast, the time for actually releasing the notified resource is initially indeterministic, but depends once more on the hint handling strategy and on optional contracts: For our setup we demanded the core $C_1$ to always collaborate in case it received a hint concerning the shared SPI bus resource. The resulting resource allocation delay for core $C_3$ was measured to be perfectly constant at 160 cycles. Using e.g. 50 MHz as CPU frequency finally resulted in a resource allocation delay of 0.1 µs for the free bus resource, and 3.2 µs for the already allocated but collaboratively shared bus resource.

As we have seen from this use case, implementing DynamicHinting as peripheral resource manager module for multiple cores allows the direct resource utilization by application tasks. For virtualized systems in particular, this might in the long run help to avoid access indirections over additional drivers within a potential hypervisor.

## 7. Conclusion and outlook

In this paper we presented an entirely novel and unified resource sharing concept for open multi-tasking and multi-core environments with real-time demands: DynamicHinting initially allows combining the semantics of task priorities, time, and events to simultaneously support time-triggered and event-driven execution models for both periodic and sporadic tasks:

At runtime the involved resource managers develop a complete knowledge about currently lasting resource allocations and pending resource requests. In case of a currently not assignable resource they suspend the requester, and perform priority inheritance on the resource owner according to a selected policy. A blocking task will thus inherit at least the priority of the just suspended task. As the most central innovation of our approach, the blocker will also be notified immediately through a hint to become aware of the situation and its spurious influence.

As we have seen from Section 5, the combination of temporally bounded resource requests, dynamic hints, optional contracts, and the priority inheritance protocol already provides a blocking task with the necessary time, knowledge, objective, and CPU power to resolve the conflict in the most appropriate way. The question about how to design the mentioned contracts and the TUFs for reliably bounding the collaboration delay has already been discussed in [3, 4]. This paper added the synchronized and unified handling of hints through exceptions as well as the entire hardware realization of the approach. Our hierarchical extension from Section 6 to even pass hints between the cores and the tasks thereon bridges the gap between virtualization (isolation) and real-time kernels (integration) by introducing an additional hardware resource manager that is compatible to the resource manager in the guest operating system *Smart*OS.

Although we currently apply our collaborative resource sharing concept primarily for tasks within the *Smart*OS multitasking kernel, the FPGA implementation from Section 6.1 did already show convincing results for sharing resources on-demand between (operating) systems on several CPU cores. Thus, as subject of our current work, further improvements aim on a theoretical feasibility analysis for dynamically defined contracts and the corresponding worst case allocation times across core borders. While we would certainly not recommend to *excessively* soften the guest isolation paradigm in hypervisor systems, a hardware-assisted virtualization can definitely benefit from DynamicHinting for sharing at least those resources collaboratively which cannot be assigned strictly exclusive due to various reasons. In this regard, we consider the co-design of kernel and hypervisor architectures toward a compatible mechanism as a great opportunity for managing the growing complexity of today's embedded systems.

## References

[1] Airlines Electronic Engineering Committee. *ARINC Specification 653, Avionics Application Software Standard Interface*, 2006.

[2] T. P. Baker. A Stack-Based Resource Allocation Policy for Realtime Processes. In *IEEE Real-Time Systems Symposium*, pages 191–200, 1990.

[3] M. Baunach. Dynamic Hinting: Collaborative Real-Time Resource Management for Reactive Embedded Systems. *Journal of Systems Architecture*, 57:799–814, Oct. 2011.

[4] M. Baunach. CoMem: Collaborative Memory Management for Real-Time Operation within Reactive Sensor/Actor Networks. *Real-Time Systems*, 48:75–100, Aug. 2012.

[5] E. Bini, G. Buttazzo, J. Eker, S. Schorr, R. Guerra, G. Fohler, K.-E. Arzen, V. R. Segovia, and C. Scordino. Resource Management on Multicore Systems: The ACTORS Approach. *IEEE Micro*, 31:72–81, 2011.

[6] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh. Worst-Case Response Time Analysis of Real-Time Tasks under Fixed-Priority Scheduling with deferred Preemption. *Real-Time Systems*, 42:63–119, 2009.

[7] A. M. K. Cheng and J. Ras. The Implementation of the Priority Ceiling Protocol in Ada-2005. *Ada Lett.*, XXVII(1):24–39, 2007.

[8] J. Fornaeus. Device Hypervisors. In *47th Design Automation Conference (DAC 2010)*, pages 114–119, New York, NY, USA, 2010. ACM.

[9] Gartner Inc. *The Gartner, Inc. Hype Cycle and Priority Matrix*, Aug. 2011.

[10] G. Heiser. The Role of Virtualization in Embedded Systems. In *1st Workshop on Isolation and Integration in Embedded Systems (IIES 2008)*, Apr. 2008.

[11] M. Jones. What Happened on Mars?, 1997.

[12] J. Kiszka. Towards Linux as a Real-Time Hypervisor. In *11th Real Time Linux Workshop*, 2009.

[13] P. Li, B. Ravindran, and E. D. Jensen. Adaptive Time-Critical Resource Management Using Time/Utility Functions: Past, Present, and Future. *Computer Software and Applications Conference*, 2:12–13, 2004.

[14] A. K. Mok, A. X. Feng, and D. Chen. Resource Partition for Real-Time Systems. In *7th IEEE Real-Time Technology and Applications Symposium (RTAS 2001)*, pages 75–84, 2001.

[15] W. River. The Wind River Hypervisor. Web site http://windriver.com/products/hypervisor/, 2012.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *Transactions on Computers*, 39(9):1175–1185, 1990.

[17] A. Tavares, A. Didimo, S. Montenegro, T. Gomes, J. Cabral, P. Cardoso, and E. Ekpanyapong. RodosVisor – an object-oriented and customizable hypervisor: The CPU virtualization. In *1st IFAC Conference on Embedded Systems, Computational Intelligence and Telematics in Control (CESCIT 2012)*, pages 200–205, Apr. 2012.

[18] S. Xi, J. Wilson, C. Lu, and C. D. Gill. RT-Xen: Towards Real-Time Hypervisor Scheduling in XEN. In *11th International Conference on Embedded Software (EMSOFT 2011)*, pages 39–48. ACM, Oct. 2011.

[19] G. Yao, G. Buttazzo, and M. Bertogna. Feasibility Analysis under Fixed Priority Scheduling with Fixed Preemption Points. In *16th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2010)*, pages 71–80, Washington, DC, USA, 2010. IEEE Computer Society.

[20] Özalp Babaoğlu, K. Marzullo, and F. B. Schneider. A Formalization of Priority Inversion. *Real-Time Systems*, 5(4):285–303, 1993.