Dynamic Hinting: Collaborative Real-Time Resource Management for Reactive Embedded Systems

Marcel Baunach

*Department of Computer Engineering, University of Würzburg, Am Hubland, 97074 Würzburg, Germany

Abstract

The increasing complexity of today’s reactive embedded applications can rapidly result in reduced real-time capabilities of the underlying hard and software. As an example for this paper we’ll refer to the specific and growing demands on the severely resource constrained sensor nodes in sensor/actuator networks (SANet). While preemptive operating systems are one way to retain acceptable reactivity within highly dynamic environments, their concurrency paradigm commonly leads to severe resource management problems, caused by the coexistence of tasks with interfering and even varying requirements. To counteract these problems, we present the novel Dynamic Hinting approach for maintaining good reactivity in typically resource constrained sensor/actuator systems by efficient combination of preemptive task scheduling and collaborative resource allocation. With respect to task priorities, our technique significantly improves classical methods for handling priority inversions (and deadlocks where required) under both short- and long-term resource allocations. Furthermore, we facilitate compositional software design by providing independently developed tasks with runtime information for yet collaborative and reflective resource sharing – e.g. by means of time-utility-functions. In many cases this even allows to reduce blocking delays as otherwise imposed by bounded priority inversion.

Keywords: real-time resource management, reflection, collaboration, prioritized/preemptive tasks, embedded systems, SANet

1. Introduction

The ever increasing size, pervasiveness and demands of today’s sensor/actuator networks (SANet) significantly boost the complexity of the underlying nodes. To still allow an easy and unobtrusive deployment in large numbers while keeping costs, power consumption and maintenance efforts low, these embedded sensor/actuator systems are – and will probably remain – rather small in size, computationally weak and severely resource constrained. Thus, modular hardware and software concepts (e.g. service oriented programming abstractions [1, 2]) are more and more used to manage their design, implementation and operation. Obviously, adequate interaction between the various modules is essential to avoid typical compositional problems. Beside task scheduling [3], directly related issues comprise dynamic resource sharing or even real-time operation [4]. Concerning this, we find that current research in the field of severely resource constrained systems is still too limited to static design concepts. As already stated in [5], next generation embedded systems will be more frequently used as reactive real-time platforms in highly dynamic environments. Here, the true system load varies considerably and can hardly be predicted a priori during development. In fact, we also expect a clear focus shift from almost pure sensing in classic sensor networks (SNs) toward additional and intense pro-activity in SANet applications. Integrated control systems, precise and DSP based on-demand measurements, time synchronization and recording services, real-time routing, etc. are just a few examples. Then, preemptive and prioritized tasks are required for fast response on various (sporadic and periodic) events but further complicate resource assignment and reactivity. This is especially true for open systems where both real-time and non-real-time tasks must coexist in order to reduce hardware overhead, energy issues and deployment effort.

1.1. Summary and Structure of this Paper

In this contribution we present the novel Dynamic Hinting approach for collaborative resource sharing and real-time operation within preemptive operating systems. Dynamic hinting improves compositional software design by granting independently implemented tasks a collaborative, but mutually exclusive use of shared resources. Thereby we support both periodic and aperiodic tasks (e.g. sporadically triggered in event driven designs). As often suggested (e.g. in [6]), we take advantage of the resource manager’s enormous runtime knowledge about the system’s current requirements. This information is carefully selected and forwarded to exactly those tasks, which currently block the execution of more relevant tasks by a resource allocation. In turn, these so called hints allow blocking (and this also includes deadlocked) tasks to adapt to the current resource demands, and finally to contribute to the system’s overall reactivity and stability. In some cases, even delays which would otherwise occur due to bounded priority inversion can be reduced. Furthermore, accounting for the task priorities as defined by the developer is simplified. The decision between following or ignoring a hint is made by each task autonomously.
and dynamically at runtime, e.g. by use of appropriate time-
utility-functions [7]. In our opinion, the central weakness of all
resource management approaches we found so far is, that tasks
are not aware of their (varying) influence on the remaining sys-
tem, and thus cannot collaborate adequately. In this respect,
dynamic hinting follows classic reflection concepts [6, 8]: It in-
troduces a new policy into operating system kernels, by which
programs can become ‘self-aware’ and may change their be-
behavior according to their own current requirements as well as
to the system’s demands. Thus, our concept is not limited to
embedded systems and the SANet domain in particular, but can
be applied to real-time operation in general.

This work primarily presents dynamic hinting as extension
for the priority inheritance protocol (PIP) [9]. This way, we
focus on long-term resource allocations (as frequently re-
quired for e.g. complex objects or hardware devices [10]) and avoid
some related shortcomings of similar techniques. Since our
concept may also be combined with most other policies and
systems where task blocking can occur, we also compare re-
results from using it with the priority ceiling protocol (PCP) [9].

Though both original protocols face several problems and failed
almost completely within some of our test beds, dynamic hint-
ing always allowed a significantly higher resource load and im-
proved task progress/utility. Just limited by the CPU speed, our
approach improved both resource allocation delays and task re-
activity to be very close to the achievable best case values (just
≈1% below in several cases). In fact, by the nature of both clas-
sic protocols, extending PIP commonly achieved a much better
performance gain than extending PCP.

While this article is in general based on [11], it signifi-
cantly extends our previous work (which considered PIP only)
by a deep analysis of using PCP with our collaborative ap-
proach. We also present an entirely novel method for signaling
blocking tasks asynchronously about their spurious influence
even if these are currently executing arbitrary application code.

Another point is the improved discussion about time-utility-
functions which can be used to decide between collaborative
and egotistic task behavior dynamically at runtime. Further ad-
ditions comprise, among others, an improved method for deter-
mining critical resources (i.e. the ones which cause the blocking
of higher priority tasks), the discussion of further potential ap-
plication areas and relationships to reflective operating system
concepts, various considerations about the impact on the pro-
graming model, and novel test bed results which we obtained
from implementation updates.

The remaining work is organized as follows: Initially, we
will further motivate the need for real-time and preemptive op-
erating systems in reactive embedded systems and sensor/ac-
tuator platforms in particular. Then, we’ll outline some re-
lated techniques from existing work. The concepts and details
about our new approach will form the central part of this paper.

Furthermore, an exemplary implementation of dynamic hinting
will show that – despite of the problem’s complexity – it is effi-
ciently applicable even for low performance devices like sensor
nodes. Therefore, we will also present some application exam-
pies and the impact on the programming model before perfor-
mance results from real-world test beds close this paper.

2. Motivation and Requirements

An operating system has significant influence on the over-
all system performance since it coordinates task interactions as
well as the access to shared operational resources like hardware
components or data structures. For some of them, exclusive
access must be granted at least temporarily to avoid task race
conditions, resulting malfunctions or even system breakdown.
Unfortunately, resource assignment in complex, modular sys-
tems with concurrently running tasks is hard to manage during
development and runtime. This is particularly true, if tasks are
allowed to use virtually any available resource in any order and
if they may even require exclusive access to several of them at
the same time. As long as allocation times remain relatively
short or if the runtime requirements are roughly predictable, ef-
cient methods already exist (→Sections 3, 4). However, if
long-term allocations collide with sporadic but time critical on-
demand allocations in dynamic environments, smart adaptive
techniques are needed to still provide good reactivity.

Whereas the main resource of each computing system, the
CPU, is often managed by the task scheduler in a preemp-
tive way, we believe that the operating system should also co-
ordinate access to other exclusive resources (where automatic
preemption is often not that easy) by contributing appropriate
mechanisms. Though some techniques were already imple-
mented for resource constrained platforms, most of them do not
address the specific aspects of reactive real-time operation.

2.1. Terminology and The Problem Model.

In the context of this work we denote a resource as preemp-
tive if the resource manager is authorized at any time to tem-
porarily withdraw the resource from a task and if it is also ca-
pable of returning it in its previous state. In contrast, a non-
preemptive resource must always be released voluntarily by its
current owner task. Apart from the preemptive CPU, we con-
side all other resources as non-preemptive.

Additionally, we distinguish between short-term and long-
term resource allocations: While the first term means that a
process does not suspend itself while holding the resource (e.g.
lock a data structure, process the contained data, and release it
eventually), the latter permits self-suspensions (e.g. lock a hard-
ware component and release the CPU until an interrupt request
occurs).

Regarding these definitions, our optimization target for dy-
namic hinting is twofold: First, we aim on on-demand re-
source deallocations to reduce the duration of (bounded and
unbounded) priority inversions, and to help software designers
in avoiding prophylactical (e.g. regular/frequent) deallocations
just to (maybe) serve other tasks. Second, we intend to im-
prove the interleaved execution of preemptive tasks through the
provision of a generous but priority aware resource assignment
policy, which relieves the resource manager from the need to
maintain a so called safe-state on the resource pool. By com-
bining both goals we achieve that persisting (long and short-
term) resource allocations do not circumvent the assignment of
further (free) resources, and finally observe a significantly improved task progression and utility compared to PIP and the conservative PCP in particular.

Compared to the classical problem model, which is commonly used in PIP/PCP related literature and where a task locks/allocates a resource and won’t unlock/release/deallocate it until its work with this resource has been entirely finished, our modified problem model relaxes this constraint: We intentionally introduce the option for an early but strictly task-controlled release of non-preemptive resources if these cause the blocking of a task which is higher prioritized than the one which currently keeps the resource locked. In fact, the resulting the on-demand resource handover is the key to our concept’s success and easily outperforms PIP, PCP, and similar resource sharing protocols.

2.2. Requirements

During our research and practical work, we found that reactivity and pro-activity in modern embedded and SANet applications requires quite sophisticated real-time and resource concepts. We’ll give just few examples from our real-world applications:

A radio protocol task commonly requires long-term allocation of the used transceiver in combination with relatively short but sporadic access to the interconnection bus. Obviously, both resources need specific configuration, and thus are non-preemptive. Although the task might suspend itself while holding the transceiver and waiting for certain events, using the bus becomes time critical when radio transmission slots must be obeyed or when a receive buffer must be read and cleared quickly to allow the reception of successive radio packets. Consequent to this communication task, sensor tasks often use exactly the same interconnection bus for data transmission or even continuous streaming (e.g. from an ADC to some external consumer). Again, both resources are non-preemptive, but this time the bus is also locked in a long-term allocation (maybe also spanning over self-suspensions). The resulting compositional problem is already hard to solve. Even if task priorities can be selected carefully to indicate the desired relevance of each task, their compliance can not be guaranteed. Instead, knowledge about the overall system load (including further tasks) must be incorporated manually into the code – if this is possible at all. The regular release of a long-term resource could be one solution. However, this might impose considerable overhead when deallocation and re-allocation are expensive in time and energy, which is also quite often a valuable and highly critical resource. Where data streams often require explicit termination (trailers) and initiation (headers), resources might require a time-consuming (de)initialization procedure.

In addition to sporadic resource sharing, we found the support for adjustable task priorities convenient and useful. Beside a sensitive adaptation of tasks to changing environmental conditions, this would also allow server tasks to adopt to the priority of their (most relevant) clients at runtime. Though this complicates resource sharing even further, it should also be considered. A specific example is the sharing of hardware among virtual networks on routers or nodes. The idea is to largely isolate corresponding virtual subsystems on the same device for maintainability, security and safety reasons [12]. These, however have changing QoS demands and in consequence need flexible access to still shared I/O ports. While variable priorities can signal their relative importance, the prompt adaptation remains problematic: techniques for coordinating the mutual requirements of separate subsystems on-demand are still rarely found. With the increasing availability of multi-core architectures in embedded and SANet areas [10, 13], this must be considered even more carefully to efficiently utilize their capabilities.

In general, fast reactions on internal and external events (e.g. interrupts) are frequently required within similar scenarios but often suffer from severe delays due to currently blocked resources. Then, their fast handover to the reacting task is quite essential and should at least be facilitated by the resource manager without damaging the atomicity of depending operations. After motivating our requirements for real-time resource management we’ll overview some related work.

3. Related Work

*Non-preemptive systems* with run-to-completion tasks are very common in sensor/actuator systems, and prevent some conflicts implicitly since their executions can’t be interleaved. If tasks need to hold exclusive access to certain resources over several runs, a frequent approach is to implement server tasks or stateful function libraries for these resources. Such an abstraction layer allows sharing or even virtualization of resources like in the TinyOS component concept [14]. However, large resource hierarchies might result in severe inter-communication overhead and reduced overall performance, then.

Additionally, run-to-completion tasks often provide bad reactivity to sporadic events since they can not easily be suspended for any more important action. Indeed, interrupt handlers might react quickly, but using resources therein is seldom wise since these might currently be unavailable, and the attempt could block the whole system for quite some time. Event-driven systems like TinyOS solve this problem by simply posting an appropriate handler task during the interrupt service routine. However, its true execution delay is unknown or at least non-deterministic, and again depends on the currently running task and the scheduling policy. Note, that the non-preemptive use of the CPU can already lead to some kind of priority inversion, then (→Section 4). Therefore, TinyOS and Contiki [15], which natively also runs non-preemptive tasks, both support so called *TOSThreads*, and *protothreads* respectively. These are preemptive but lack priorities and resource management entirely.

*Preemptive systems* potentially provide much better reactivity. Here, a task can be suspended at any time for a more important action implemented in another task. Therefore individual priorities commonly define each task’s relevance. Yet, preemption yields no instant advantage if the important action requires a shared resource which is exclusively held by a less important task. Resulting problems like priority inversion [4] might lead to thwarting of high priority tasks and even deadlocks may occur. To cope with some of these issues, well studied protocols
like priority ceiling (PCP), highest locker (HLP), priority inheritance (PIP) [9, 16] or the stack resource policy (SRP) [17] can be found in literature and are implemented in some operating systems. Beside causing some runtime overhead, these techniques also suffer from certain weaknesses addressed in the next section. Moreover, preemptive sensor network operating systems like e.g. MANTIS [18], RETOS [19] or SenSmart [20] do not consider real-time or resource related problems at all.

4. Resource Management in Preemptive Systems

Priority inversion is an inherent problem of preemptive, prioritized tasks. Here, the competition for a single, non-preemptive resource may already lead to (temporary) blocking of a high priority task \( H \), if it requests a resource which is currently allocated by a lower prioritized task \( L \). Figure 1a shows this direct dependency problem which is known as bounded priority inversion. If an additional task \( M \) with medium priority prevents \( L \) from running and thus from releasing the resource, this indirect dependency is known as unbounded priority inversion (Fig. 1b) and might even result in the final suspension of \( H \). In both cases, the task priorities defined by the developer are not obeyed as desired, leading to unexpected behavior, reduced reactivity and real-time capability of the overall system.

By this, e.g. the Mars pathfinder mission almost failed [21]. The already mentioned PIP, PCP and HLP techniques face this problem by adjusting task priorities dynamically at runtime according to the current resource assignment situation. SRP works in a similar way, but commonly assigns preemption levels with regard to relative deadlines. From these alternatives, we selected the priority inheritance protocol as basic technique for our dynamic hinting approach. Though priority ceiling, highest locker, and the stack resource policy inherently prevent deadlock situations, and even restrict the maximum allocation delay (of initially unknown duration) to at most one first requested resource per task, these techniques imply some serious problems. First they need initial information about each task’s worst case resource requirements, which are often not even constant over several runs, but depend on various conditions and the code execution flow. Furthermore, arbitrary allocation nesting is hard to manage, and most notably, long-term resource allocations can immediately cause disastrous effects. Except for the last problem, we won’t go into detail about these techniques but refer to [4, 9, 22] and Section 7 instead.

For deadlock avoidance, HLP, PCP and SRP use a rather conservative policy when adjusting task priorities (or preemption levels) and deciding if a resource request is granted or denied (see Section 7). In some cases this rapidly leads to a problem often referred to as avoidance-related-inversion [22]: If a task \( T_1 \) with current priority \( p(T_1) \) holds a resource, PCP at least refuses the assignment of any remaining but free resource \( r \) to any other task \( T_2 \) with \( p(T_2) \leq p(T_1) \). Within some implementations, HLP implicitly acts in a similar way by avoiding the execution of such a task \( T_2 \) entirely [16]. Although these implicit and anticipatory reservations would ensure a fast assignment of further resources to \( T_1 \), they are critical in many respects. First, \( T_2 \) is rejected even if \( r \) will not be allocated by \( T_1 \) for a long time. Second, for performance in many implementations, \( T_2 \) will also be rejected if it does not even share a single resource with \( T_1 \). Finally, the penalty is even worse if the protocol already raised \( p(T_1) \) above its original base priority \( P_{T_1} \), while in fact, \( T_2 \) was specified to be truly more relevant than \( T_1 \) (i.e. \( P_{T_2} > P_{T_1} \)). In summary, a task implicitly prevents other tasks with equal or lower priority from being served while it simply holds a (maybe rarely shared) resource. When recalling our motivation for both real-time resource requests and long-term resource allocations (spanning over task self-suspensions) from Section 2.2, it becomes obvious that such a behavior is highly critical, and thus undesired. Another problem with PCP, HLP, and SRP is that dynamic base priorities are hard to manage within the resource manager. For large task systems in particular, this feature will either cause significant algorithmic runtime effort to avoid the comeback of deadlocks, or it must be disabled while tasks hold resources.

In comparison, PIP is much more generous when granting resource requests and sometimes gains a better average case performance. Here, requests for free resources are always granted immediately: The successful allocation of a resource \( r \) by a task \( L \) will initially leave \( L \)’s priority unmodified. Then, as soon as a task \( H \) with higher priority requests \( r \), \( L \) will be raised to the priority of \( H \). This avoids unbounded priority inversion and allows \( L \) a fast deallocation of \( r \). By doing so \( L \)’s priority is reduced again, \( H \) obtains \( r \) and is finally resumed.

Unfortunately, PIP may lead to chains of resource blocked tasks (chained blocking) and even deadlocks may occur. Whether these can still be prevented entirely depends on the applied resource management policy [23]. However, we will show that both shortcomings can be handled by our extension in an efficient manner. In our opinion, complete deadlock avoidance is not very practical in most embedded applications, anyway. Beside the frequently recommended Banker’s Algorithm [24] which maintains a so called safe state for avoiding deadlocks, another strategy is to require the resource allocation order to be fixed and inverse to the release order. In many scenarios both is neither possible nor desired. Where the first might unnecessarily block (high priority) tasks, the latter can even cause resources to remain allocated longer than really required by the task logic. So, deadlock detection and recovery is often needed.

Terminating a spurious low priority task or withdrawing its resources for the benefit of a more important one is critical, too. While undoing the work so far, this might also leave the resources in an undefined state, making a handover problematic. Just imaging the disastrous withdrawal of a currently active DMA or radio controller! Counteracting with checkpointing and roll-backs of whole tasks and involved resource states...
is hard or even impossible. Even if rarely used, this would produce enormous system load and memory overhead issues on smaller embedded systems. Not to mention the increased power consumption which is of special importance for energy constrained systems like sensor nodes.

So far, we presented our requirements for real-time aware resource management along with some already available concepts and systems for SANets and general embedded applications. Additionally, though the central idea behind dynamic hinting can also be combined with some other techniques, we motivated our decision for using the priority inheritance protocol as basis for our new approach, since it inherently allows long-term resource allocations without avoidance-related-inversions.

5. Resource Management and Dynamic Hinting

This section presents the details about our novel resource management approach. The basic idea behind dynamic hinting might be applied as integral concept for many (embedded) real-time operating systems if these support truly preemptive and prioritized tasks plus a timing concept that allows temporally limited resource requests. For our reference implementation we extended SmartOS [25], since it offers quite common characteristics, and thus is a good representative for the adaptation of similar systems. In addition, it is available for several microcontroller architectures, provides appropriate task, timing and resource basics, and finally allowed an easy integration.

Since SmartOS was developed for reactive systems, it inherently supports fully preemptive and prioritized tasks. Each task has its individual and dynamic base priority. This satisfies our request for easy adaptation to changing demands and environmental conditions at runtime. Furthermore, the kernel maintains a local system time and enables tasks to suspend themselves for/until a specified time (sleep). Most important for dynamic hinting, the integrated timing concept allows temporally limited waiting for events and resources with a certain (relative) timeout or (absolute) deadline. While this already provides a simple method for deadlock recovery, tasks may react on resource allocation failures and event imponderabilities without jamming the whole system (e.g. by spinning request loops).

We’ll now formalize the extended resource management policy of SmartOS (5.1) along with our new approach for combining priority inheritance (5.2) and dynamic hinting (5.3, 5.4). See Section 7 for a comparison to the priority ceiling variant.

5.1. Extended SmartOS Specifications

S1 Each SmartOS system consists of a set of preemptive tasks \( T \) \((|T| \geq 1)\) and non-preemptive resources \( R \) \((|R| \geq 0)\). Each task \( t \in T \) is executed for the whole system runtime and can neither be started dynamically nor terminate entirely. As depicted in Figure 2, it is always in one of the following states: ready, running, or waiting (for a resource, an event or simply for some time to elapse). On startup, all tasks are in ready state.

S2 Each task \( t \) has a base priority \( P_t \) which is defined at compile time and can be changed at runtime. We additionally introduce an active priority \( p(t) \geq P_t \) which is assigned by our resource management approach. At system start \( \forall t \in T : p(t) = P_t \) holds. The scheduler always selects a task with highest active priority in ready state for execution. For tasks with equal priorities, either round-robin or run-to-completion scheduling can be selected at compile time.

S3 State transitions can be triggered in three ways (→ Fig. 2):

a) self-suspension: the running task transits to waiting state by either sleeping, requesting a resource which is currently allocated by another task, or by waiting for a not yet occurred event.

b) self-preemption: the running task transits to ready state by releasing a resource or by invoking an event for which a higher prioritized task is already waiting. Reducing its own base priority might also do so.

c) passive: a task’s state is changed due to any other task’s operation (e.g. releasing a resource) or if its timeout for waiting/sleeping has expired.

S4 All resources in \( R \) are treated as non-preemptive and will never be withdrawn. Once assigned, each owner task is responsible for releasing its resources.

S5 If a task requests a currently free resource, it immediately succeeds and remains running. Otherwise it transits to waiting state. Waiting can be limited by specification of a timeout. A task remains in waiting state until it receives the resource or the timeout is reached. Then, depending on the other tasks, it continues to ready or running state.

S6 Any task may allocate any resource several times and must release it as often. Requests for already self-allocated resources are granted immediately without suspension.

S7 Each task \( t \in T \) may wait for at most one single resource \( r \in R \) at the same time. The awaited resource is

\[
\alpha_t := \begin{cases} 0 & \text{if } t \text{ awaits no resource} \\ r \in R & \text{if } t \text{ awaits } r \end{cases}
\]

S8 Several tasks \( T_r \subseteq T \) may await the same resource \( r \in R \) at the same time (due to S5, S6; \( T_r \neq T \)):

\[
T_r := \{ t \in T \mid \alpha_t = r \}.
\]
Section 3.4.1 Priority Inheritance and Deadlock Occurrence

Each resource \( r \in R \) may be assigned to at most one owner task \( t \in T \) at the same time. The owner task is

\[
\sigma_r := \begin{cases} 
\emptyset & \text{if } r \text{ is not assigned to any task} \\
t & \text{if } r \text{ is assigned to } t 
\end{cases}
\]

Section 3.4.1.1 Resource allocation

Each task \( t \in T \) may hold exclusive access to several resources \( R_t \subseteq R \) at the same time:

\[
R_t := \{ r \in R \mid \sigma_r = t \}.
\]

Allocation and deallocation orders of resources are arbitrary and independent from each other.

Section 3.4.1.2 Resource deallocation

If a task \( t \in T \) releases a resource \( r \in R_t \) entirely, \( r \) will directly be handed over to task \( u \in T_r \) with highest active priority \( p(u) \). On equal priorities, just the one which requested \( r \) first will receive it and leave waiting state (\( \rightarrow \text{S3c} \)).

In consequence to these specifications, dealing with deadlocks will be required at runtime, since the four Coffman conditions [23] are fulfilled and deadlock prevention is not possible:

- Mutual exclusion (by S9),
- Hold and wait (by S7, S10),
- Non-preemptive resources (by S4), and
- Circular waits (by S5, S10).

5.2. Priority Inheritance and Deadlock Occurrence

After disclosing the specifications, we'll now define the resource-await-queue (RAQ) as central data structure for our resource management approach.

**Definition:** The resource-await-queue \( A(t) \) of a task \( t \in T \) is an alternating list of tasks and resources for the representation of currently existing task-resource dependencies (\( \rightarrow \text{Fig. 3a} \)):

\[
A(t) := (t, r_t, \sigma_{(t)}, \alpha_{(t)}, \sigma_{(\alpha)}, \alpha_{(\alpha)}, \sigma_{(\alpha)}, \ldots)
\]

Thereby, two structural properties become obvious:

1. For each task \( t \), \( A(t) \) is well-defined, since \( \forall x \in A(t) : \text{outdeg}(x) \leq 1 \) due to S7 and S9.
2. Two or more RAQs may converge (\( \rightarrow \text{Fig. 3b} \)), since \( \forall x \in A(t) : \text{indeg}(x) \geq 0 \) due to S8 and S10.

Furthermore, we show an important fact about RAQs:

**Lemma 1.** \( \forall t \in T : A(t) \) ends either in a task or in a cycle.

**Proof.** Assume \( A(t) \) ends in a resource \( r \in R \). Then, \( \sigma_r = \emptyset \) and \( \exists u \in T : \sigma_r = r \). This means, that \( u \) would await \( r \) even though \( r \) is free – a conflict to S5 and S11.

Lemma 1 directly leads to some observations:

1. If \( A(t) \) and \( A(u) \) contain at least one common element \( x \in R \cup T \), they also contain at least one common task \( v \in T \). Finally, \( A(v) \) controls further execution of both \( t \) and \( u \). Within the example in Figure 3b, \( t_1 \ldots t_6 \) depend on \( A(t_6) \) and finally on \( t_7 \).

2. If \( A(t) \) does not end in a cycle, only its last task can be in ready or running state. All others are currently waiting.

These observations are exactly the critical point when dealing with resource management under real-time conditions. The tail of a RAQ (cycles will be addressed later) always prevents all other tasks therein from running because of at least one certain resource. Until now, this was regarded entirely independent from any task priorities. However, we actually want all tasks to be scheduled close to their intended base priorities, but we also mentioned in Section 4, that this is not always possible due to priority inversions. Therefore, we adapt each task’s active priority to the resource situation:

Each task \( t \in T \) always receives at least the maximum active priority of all tasks \( u \neq t \) it currently blocks \( (t \in A(u)) \). Indeed, \( t \) blocks all its preceding tasks \( A(u) \) and we want \( t \) to release its resources quickly to grant a fast resumption of more important tasks. Thus, as shown in Figure 4, we initially define

\[
w(r) := \begin{cases} 
0 & \text{if } T_r = \emptyset \text{ and } \text{indeg}(r) = 0 \\
\max \{ p(t) \mid t \in T_r \} & \text{if } T_r \neq \emptyset \text{ and } \text{indeg}(r) \geq 1 
\end{cases}
\]

as the maximum active priority of all tasks \( t \) currently waiting for resource \( r \). Hence, the optimum active priority \( p(t) \) for each task \( t \) has its lower bound limited by its allocated resources at

\[
W(t) := \begin{cases} 
0 & \text{if } R_t = \emptyset \text{ and } \text{indeg}(t) = 0 \\
\max \{ w(r) \mid r \in R_t \} & \text{if } R_t \neq \emptyset \text{ and } \text{indeg}(t) \geq 1 
\end{cases}
\]

Furthermore, \( p(t) \) is always limited to the bottom by \( t \)'s own base priority \( P_t \). Finally, \( t \)'s active priority computes as

\[
p(t) := \max \{ P_t, W(t) \} \geq P_t.
\]

This does not only solve the priority selection for priority inheritance but also leads to

**Lemma 2.** Each RAQ \( A(t) \) is always partially ordered by active priorities and thus its tail has highest active priority.

**Proof.**

\[
\forall u, v \in A(t) \cap T : \exists r \in A(t) \cap R, \alpha_u = r, \sigma_v = u \Rightarrow u \in T_r, r \in R_v \\
\Rightarrow p(v) \geq W(r) \geq w(r) \geq p(u)
\]
Lemma 3. If any $A(t)$ contains a cyclic subsequence $C$, then:

1. $C$ is a deadlock cycle,  (proof by $S5$, $\forall_{u \in C \cap T} \alpha_v \notin \emptyset$)
2. $A(t)$ contains at least two tasks,  (proof by $S6$)
3. $A(t)$ contains no other cycle,  (proof by $S7$)
4. all tasks in $A(t)$ are suspended,  (proof by Lemma 1)
5. $\forall_{u, v \in C \cap T} \ p(u) = p(v)$.  (proof by Lemma 2)

Let’s consider the consequences: By requesting a resource $r$ currently allocated by $u = \sigma$, a task $t$ only produces a deadlock if it already holds another resource $r' \in A(u)$ (Fig. 3c). Then, $A(u)$ and $A(t)$ contain exactly the same elements. Thus, deadlocks are first considered when a task requests a resource.

As requested in Section 2, the formalization of our resource management policy along with the priority inheritance protocol, supports arbitrary and independent resource (de)allocation orders plus dynamic base priorities (Fig. 4). These currently define $A(t)$ (Fig. 4). These currently define $A(t)$ for each task $t \in T$.

5.3. The Dynamic Hinting Approach

The central objective of dynamic hinting is to allow tasks the collaborative sharing of exclusive resources while, at the same time, it supports them to closely comply with their intended base priorities. We already introduced the related problems in compositional task systems, but also motivated in Sections 2 and 4 why we accept and handle them dynamically at runtime.

Many conservative resource management systems try to avoid deadlocks by simply refusing a resource request immediately if it would cause an allocation cycle. Others accept at least chained blocking and simply suspend the requester $h$ until it can be served. In our opinion, both methods are not satisfying since exactly the just rejected or suspended task $h$ alone has to cope with the situation. This is especially annoying if $h$ is truly more important than at least one other task $t$ in the just averted cycle or extended chain. Then, this results in a violation of base priorities ($p_t < p_h$). Furthermore, resources are usually indispensable when requested, and thus, tasks tend to retry infinitely until the request succeeds. The resulting (active/spinning) loops or long timeouts might not only starve other tasks, but even worse, they simply shift the problem back from system level to task level without further information about how to proceed effectively. Indeed, the task-resource-dependencies and RAQ structures are highly dynamic and depend on the system wide allocation order. Hence, another task therein might react much better than $h$ if it knew about the situation. Unfortunately, tasks are commonly not aware about their spurious influence and so the RAQs are commonly reduced successively, beginning at their very end ($\rightarrow$Lemma 2). This is exactly where dynamic hinting applies.

Our approach provides runtime information for each task about which resource it should release to improve the overall system reactivity and liveliness. Considering these so called hints is always optional for each task. But if followed, it definitely breaks an RAQ and reduces direct, chained or deadlock blocking of at least one higher priority task (Fig. 4, 5).

Therefore, two preconditions must be fulfilled:

1. An ongoing resource allocation shall not prevent any task from requesting any resource. Otherwise, our approach lacks knowledge about the overall system requirements. Within our specifications, this is provided by S2, S5, and the priority inheritance protocol in general.
2. A spurious task must receive the time and opportunity to react on a hint. Here, the possibility to request resources with arbitrary timeout ($\rightarrow$S5) provides the time for the current owner, and Eq. 3 provides the appropriate priority.

The first step for determining hints is to identify the $critical$ resources for each task $t$ ($\rightarrow$Fig. 4). These currently define $p(t)$ and thus, they directly or indirectly cause the blocking of any task $h$ with higher base priority $p_h > p_t$ and $t \in A(h)$:

$$crit(t) := \begin{cases} \emptyset & \text{if } p(t) = p_h \\ \{r \in R_t | w(r) > P_r\} & \text{if } p(t) > p_h \end{cases} \quad (4)$$

According to Eq. 3, the presence of a critical resource for a task $t \in T$ implies a raised active priority and vice versa:

$$crit(t) \neq \emptyset \Leftrightarrow p(t) > p_t$$

Then, $p(t)$ was raised by at least one pending resource request of a task $u$ with $p(u) = p(t) > p_t$. In turn, $t$ can reduce the blocking of at least one more important task by releasing any $r \in crit(t)$. Finally, $t$’s hint can be selected in many ways, e.g. with regard to (remaining) timeouts, priority thresholds or RAQ characteristics. Yet, our approach always selects the resource $r \in crit(t)$ which blocks a most important task, and thus defines $p(t) := w(r)$, as follows ($\rightarrow$Fig. 4):

$\text{Please note: Though limited to at most one blocked resource per task (}$∀t \in T : |A(t)| \leq 3$, this problem also exists for e.g. PCP and HLP.$\text{)}$
$\text{hint}(t) := r \in \text{crit}(t)$ with $p(t) = w(r), r$ was requested last. (5)

This information is signaled to $t$ (if desired, further critical resources can be queried by the task upon processing the hint).

As soon as $t$ releases the indicated resource, it will be directly passed to its first requester ($\rightarrow$S11), w.l.o.g. $u$. Next, $p(t)$ is updated by Eq. 1-3 and $u$ is scheduled promptly. This is true since then $u$ holds the highest priority of all tasks in ready state and $t$ did let $u$ pass by ($\rightarrow$S3b,c). As soon as $t$ is scheduled again, it can immediately re-request the just released resource to continue its operation quickly ($\rightarrow$Section 5.4). If, however, there is still another hint for $t$, our approach will signal this situation again. In any case, the ultimate release of a hint definitely resolves a priority inversion and accounts for the intended task base priorities.

The example in Figure 4 shows $\text{crit}(t_5) = \{r_2, r_4\}$ since both allocations block the execution of more important tasks. In particular, $p(t_5) > P_i$ was defined by $t_4$’s request for $r_2$. Releasing $r_2$ would instantly relax $p(t_5) := w(r_4)$. Then, $t_4$ is served and scheduled immediately since it indeed is the task with highest priority but currently still blocked by $t_5$. The allocation timeout which $t_4$ specified for $r_2$ grants $t_5$ the time to collaborate as described. If $t_5$ follows its hint $r_2$ prior to its regular release, it even shortens the bounded priority inversion toward $t_4$. At the same time, $t_5$ also improves the reactivity of $t_2$ and $t_3$ since these tasks are also more relevant ($P_2 > P_3 > P_h$) and will receive $r_2$ right after $t_4$. Finally, $t_5$ is resumed and can either re-request $r_2$ or follow another hint $r_4$ to speed up the also more important task $t_6$ ($P_6 > P_h$).

For a better understanding of the implementation details in Section 6 and the integration with PIP, we’ll briefly address the situations in which a task’s hint must be updated via Eq. 4, 5:

1. A new $\text{hint}(t)$ evolves if another task $u$ with $p(u) > p(t)$ requests any resource $r \in R_t$ while $p(t) = P_i$.
2. An already existing $\text{hint}(t)$ changes if another task $u$ with $p(u) \geq p(t)$ requests any resource $r \in R_t$ while $p(t) > P_i$. It also changes or voids if another task’s timeout for the hint is reached or if it releases it.

Two issues are obvious: First, $\text{hint}(t)$ may change each time when $p(t)$ is updated. Second, $\text{hint}(t)$ often changes while $t$ itself is not running. However, $t$ might become running then ($\rightarrow$S3c) and must get informed about its spurious influence.

5.4. Receiving and Processing Hints

A special problem with dynamic resource sharing is, that task blocking can occur at virtually any time. From the blocker’s view, this happens quasi-asynchronously and regardless of its current situation, task state or code position. Thus, we’ll now describe three ways in which a task may receive and handle its hints quickly, to help and speed up more important tasks:

First, an Explicit Query (getHint(...)) can be done regularly or at distinct code positions where its handling would be possible at all. However, by using this explicit method a task can never react as long as it is in waiting state. Yet, this is exactly the case upon deadlocks ($\rightarrow$Fig. 5b) and during many long-term allocations, where tasks e.g. sleep occasionally or wait for some events/interrupts while holding a resource ($\rightarrow$Sections 2, 7). Then, their reaction is unpredictable and deferred until the next query. Beside this severe weakness, the implementation effort and code pollution might be immense.

Second, we introduce a much better strategy called Early Wakeup. When enabled, all functions by which a task suspends itself to waiting state ($\rightarrow$S3a) may return early upon a new or changed hint. A dedicated return value will indicate this special situation. The effort and impact on the programming model is similar to exception handling in various programming languages: A task ‘tries’ to e.g. sleep but ‘catches’ an early wakeup to react on its blocking influence. This way, coping with hints can be done instantly and it is entirely limited to the cases when they really occur. Figure 6a shows an example for a sleeping task $L$ (in waiting state) which nevertheless reduces the blocking delay of a higher prioritized task $H$ promptly.

The use of early wakeup can be selected and tuned individually by each task $t$ and for each self-suspension. Therefore, we added a threshold parameter $\varphi$ to the involved kernel functions:

```
int sleep(deadline | timeout, \varphi)
int waitEvent(event*, deadline | timeout, \varphi)
int getResource(resource*, deadline | timeout, \varphi)
```

Then, a self-suspending function will only return early if

$$\varphi \neq 0 \land p(t) > P_1 \land p(t) \geq \varphi,$$

i.e. if priority inheritance raised the caller’s priority $p(t)$ to at least the specified threshold $\varphi$. In particular, these functions will also return right after calling if a hint is already available.

E.g. both new requests of $t_1$ in Figures 5a,b will immediately resume $t_2$ if it has early wakeup enabled. Then, $t_2$’s own request for $r_2$ is withdrawn. If not enabled, $t_3$ may wake up early, instead. The same applies if $t_2$ refuses to release its new hint $r_1$ but simply requests $r_2$ again. So, beginning with the most promising task, i.e. the one which could reduce the blocking most, the hint is forwarded, and obviously a single collaborative task in a chain or cycle is already sufficient to improve or recover from the situation.
Third, the last method for receiving hints are so called Hint Handlers. So far, tasks in waiting state can use early wakeup and running tasks can use explicit querying. The advantage of both methods is, that the hints can be handled directly within the task code, i.e. at positions where the developer knows the current task and resource situation, and can react in an appropriate way. A problem remains for tasks which are seldom in waiting state or execute code which can not be modified for explicit querying. Indeed, code modifications are often not desired or simply neither allowed (rights) nor possible (closed source) at all. Then, a blocking task’s priority is still raised according to PIP, but again the reactivity of other tasks remains reduced, since the resource deallocation is deferred at least until the next query or wakeup occurs. Our solution is to allow the resource manager to inject a special routine into the task execution flow, which is capable of releasing the critical resource temporarily with respect to the current task situation. When enabled, these hint handlers are similar to interrupt service routines since they can also be invoked asynchronously at any code position. Figure 6b shows an example: Here, the high priority task \( H \) preempts the low priority task \( L \) and requests a resource which is held by \( L \). Now, the scheduler does not simply resume \( L \) after raising its priority \( p(L) \) but instead it starts \( L \)'s (task specific) hint handler. Since a handler is executed directly within its task’s context, it is also allowed to operate on the critical resource just like the task itself. In particular, it may release/realocate resources (\( \rightarrow S4 \)), and also use dynamic hinting for solving further potential conflicts. As soon as a handler returns, its hinted task (or handler, respectively) is resumed where it was preempted before.

To make the hint handler’s execution entirely transparent and invisible to its corresponding task, it must fulfill two preconditions: It must know how to release the hinted resource at the current time and it must be able to recover its previous state. Then, similar to the CPU scheduler in preemptive kernels, a hint handler allows to operate literally non-preemptive resources in a quasi-preemptive way. Sometimes, similar techniques are also provided by drivers. However, these are commonly neither task specific nor able to handle resource dependencies properly. During our work, we found that it is very common, that further resources depend on a critical resource and must also be handled. In fact, the current owner can do this best, since it has the required knowledge for providing an all-embracing solution.

Though it would be useful to associate one hint handler per task-resource-combination, our current implementation w.l.o.g. allows one handler per task. Similar to early wakeup, the handler can be activated by specification of a priority threshold \( \varphi \):

\[
\text{void setDHH(handler*)} \quad // \text{register hint handler function} \\
\text{void enaDHH(p);} \quad // \text{set threshold (0 to disable)}
\]

This way, each task can also adjust and fine-tune its individual acceptance for hints to the priority of the task it blocks. Again, the hint handler is only invoked if Eq. 6 holds.

Another option is to introduce a real-time priority threshold by initially defining \( \varphi \) equal for all tasks. This inherently limits any potential collaboration to situations where tasks (directly or indirectly) block any real-time task \( t_{rt} \) with \( P_{t_{rt}} \geq \varphi \).

<table>
<thead>
<tr>
<th>Property Matrix for the three Hint Reception Methods</th>
<th>Explicit Querying</th>
<th>Early Wakeup</th>
<th>Hint Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option while in running state:</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>Option while in waiting state:</td>
<td>yes</td>
<td>on resumption</td>
<td>yes</td>
</tr>
<tr>
<td>Expected code modifications:</td>
<td>many</td>
<td>few</td>
<td>handler only</td>
</tr>
<tr>
<td>Execution of handling code:</td>
<td>sync</td>
<td>sync</td>
<td>async</td>
</tr>
<tr>
<td>Delay until the hint is received:</td>
<td>query</td>
<td>prompt</td>
<td>prompt</td>
</tr>
<tr>
<td>Task &amp; resource specific:</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>TUFs applicable:</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 1: Property Matrix for the three Hint Reception Methods

Of course, priority thresholds are not the only useful metric for deciding between collaborative or egoistic behavior. Thus, beside the hinted resource itself, we grant each task / access to some further information: Its current (raised) priority \( p(t) \), a flag indicating that a deadlock situation might persist if the hint is not followed, and the absolute time when the hint expires due to its latest request timeout:

\[
\text{Resource* getHint(Priority_t* p, boolean* DL, Time_t* T0);}
\]

This information is of special interest for applying time-utility-functions as proposed in [7]. It allows each task to reflectively relate its own demands and utility to other tasks. E.g. a hint could be ignored, if the remaining timeout is still sufficient to complete the own operation regularly or if it is too short for a timely deallocation. Otherwise it is either always followed, or the decision depends on further factors like the priority difference, task states, energy effort, etc. The test bed in section 7.2 gives a concrete example for such decision criterions.

By providing the three presented techniques for hint reception, our approach covers all potential states in which a task can be when it starts blocking another more important one (\( \rightarrow \) Fig. 2, 6). Table 1 gives a summary. In any case, handling a hint always follows the same procedure:

1. Query the critical resource, and decide between following or ignoring the hint. When following:
   (a) Save the resource state (if necessary) and stop its operation.
   (b) Release the resource. \( \text{This will cause an implicit task self-preemption (} \rightarrow S3b) \text{ due to the resource handover (} \rightarrow S11). \)
   (c) Re-allocate the resource upon resumption.
   (d) Restore the resource state and restart its operation.

In this section we showed, how dynamic hinting can help to reduce resource allocation delays and even to recover from deadlocks. By following hints from the resource manager, tasks can collaborate implicitly without explicit knowledge of each other. Initially, our approach gives no guarantee about that, but depends on the behavior of the involved tasks. Indeed, deadlocks might consolidate and blocking persists if all involved tasks behave egoistic and ignore their hints. However, a single collaborative task is already sufficient for effective improvement of these problems. In fact, contracts between tasks may be used to regulate hint handling, and account for stable system behavior. When considering our demands on the resource policy, a soft advise is a good chance to facilitate reflective operation. This also avoids complex brute force recovery methods.
which often cause high system load and energy consumption along with still reduced reactivity. Examples and performance tests follow in Section 7.

6. Implementation Details

This section shows the central algorithmic aspects and complexity details about the just described techniques within our reference implementation. Regarding the tight performance and memory constraints of many embedded systems, resource usage is limited to two central functions and timeout handling. The related code is atomic and, since RAQs and task priorities might get changed, always terminates by calling the scheduler. The problem is how to efficiently select each task’s active priority and dynamic hint simultaneously. Thus, we first consider the situations in which \( p(t) \) might change at all:

1. \( p(t) \) might rise, if a task \( u \neq t \) requests a resource \( r \in R_t \).
2. \( p(t) \) might fall, if \( t \) itself releases a resource \( r \in R_t \) or if a task \( u \neq t \) waiting for a resource \( r' \in R_t \) times out.
3. \( v(r) \) of several tasks \( v \) might change if any base priority \( P_v \) is changed at runtime while \( v \in A(u) \). We’ll omit further details here, since this just requires the application of Eq. 3 to all tasks in \( A(u) \).

Before heading to the functional details, we’ll consider the computational complexities for \( w(r) \) from Eq. 1 and \( W(t) \) from Eq. 2: Since the internal representation for each \( T_r \) is a priority queue (of active priorities), retrieving \( w(r) \) is in \( O(1) \). In contrast, each \( R_t \) is a list and thus \( W(t) \) is in \( O(\text{indeg}(t)) \).

6.1. Resource allocation: \( \text{getResource(resou}}\text{e, timeout, } \varphi \)

This function either returns 1 (success), 0 (timeout) or -1 (hint) to its caller \( t \). Two basic conditions must be considered:

1. If \( r \) is free (\( \sigma_r = \emptyset \)) or already allocated by \( t \) (\( \sigma_r = t \)), the request succeeds immediately without suspending \( t \).
   Updating any priorities or dynamic hints is not required.
2. If \( r \) is occupied by \( \sigma_r \neq t, t \) is suspended. Then, priority management and hint selection is done.

Case 1 is obvious due to Lemma 1 and S6. In particular, \( t \) is at most tail of other RAQs. So, the partial order of all RAQs remains implicitly valid. Complexity: \( O(1) \).

For case 2, \( p(t) \) remains unaffected but \( t \’s \) state is changed from running to waiting. Furthermore, \( \alpha_t := r \) and \( t \) is inserted into \( T_r := T_r \cup \{t\} \). Thus, \( w(r)_{\text{new}} \geq w(r)_{\text{old}} \).

If \( w(r)_{\text{new}} > p(\sigma_r) \), the partial order for \( A(t) \) is violated and must be fixed by raising \( p(\sigma_r) := p(t) \). In this case, \( \text{hint}(\sigma_r) := r \) is also updated since \( p(\sigma_r) \) is now limited by \( r \).
Both changes might propagate over further task-resource-dependencies, and so we iterate over \( A(t) \) until a task \( u \in A(t) \) with \( p(u) \geq p(t) \) is found. Complexity: \( O(|A(t)|) \).

According to Section 5.4, we already resume the execution of the first \( \text{waiting} \) task \( v \in A(t) \) with early wakeup enabled and for which Eq. 6 is true. If \( v \) follows the hint, dynamic hinting was already successful. Otherwise, \( v \) will simply continue its execution or restart its own just aborted request. By doing so, a hint can be passed to the next task in \( A(v) \).

6.2. Resource deallocation: \( \text{releaseResource(resource)} \)

Releasing a resource \( r \) is always initiated by its owner task \( t = \sigma_r \rightarrow S4) \). If \( t \) holds \( r \) several times, one is freed and no priority adjustments are required. The same is true if \( r \) frees \( r \) entirely while \( T_r = \emptyset \) since then, \( w(r) = 0 \) and \( p(t) \) was obviously not defined by \( r \). Complexity: \( O(1) \).

If \( T_r \neq \emptyset \) and \( t \) releases \( r \) entirely, the resource is directly handed over to a task \( v \in T_r \) with highest active priority \( \rightarrow S11) \). Thus, \( \sigma_r := v, \alpha_v := \emptyset, T_r := T_r \setminus \{v\} \) and finally \( w(r)_{\text{new}} \leq w(r)_{\text{old}} \). Yet, \( p(v) \leq p(t) \) still holds due to Lemma 2 and priority management remains to be done. First, \( p(v) \) of the new owner remains unaffected. But if \( p(v) = p(t) \), \( p(v) \) might have defined \( p(t) \) in the past and thus \( p(t) \) and \( \text{hint}(t) \) might need an update according to Eq. 3-5. As desired, \( p_t \leq p(t)_{\text{new}} \leq p(t)_{\text{old}} \) finally holds. Complexity: \( O(\text{indeg}(t)) = O(|R_t|) \).

6.3. Timeouts

If a task \( t \’s \) request for a resource \( r = \alpha_t \) times out, we have to check and possibly update the priorities and hints for several tasks in \( A(t) \). Indeed, \( \exists v \in A(t) : v = \sigma_r \rightarrow \text{Lemma 1} \) and \( p(t) \leq p(v) \rightarrow \text{Lemma 2} \).

If \( p(t) < p(v) \), \( t \) and \( v \) are neither on a common cycle nor was \( p(v) \) defined by \( p(t) \). Hence, neither \( p(v) \) nor \( \text{hint}(v) \) need updates and we are done in \( O(1) \). If \( p(t) = p(v) \), \( p(v) \) is currently limited at least by \( p(t) \). In this case, all tasks in \( A(v) \) need to be checked and updated iteratively by Eq. 3-5 until the partial order is satisfied again in \( O(|A(v)|) = O(|A(\sigma_r)|) \).

6.4. Hint Handlers

A hint handler is an ordinary function without any parameters or return value. To be transparent and avoid lasting modifications of the task’s stack or register set (which it uses), it always operates according to callee-saves strategy. If a task \( t \) transits from ready to running state (\( \rightarrow \text{Fig. 2} \) and \( \text{hint}(t) \neq \emptyset \), \( t \’s \) handler is injected. Therefore, its address is pushed on the stack, just above the original task return address. In consequence, the CPU starts the handler, and finally returns to the original task without any further context switch.

6.5. Memory and Response-Time Overhead

Since the main data structure (i.e., the RAQ) must be maintained anyway in order to support the classic PIP/PCP protocols, there is almost no additional RAM overhead for the dynamic hinting approach itself. In fact each task’s control block contains 3 additional machine words: a flag field, a pointer for the associated hint handler, and another pointer for the current critical resource (i.e. the hint). Of course the additional ROM requirements depend on the code for processing the hints.

The absolute response time overhead for passing a hint depends on the underlying hardware, but equals the time for an ordinary context switch since invoking a hint handler or resuming a task early is nothing else than switching to the corresponding task context (\( \approx 35 \) µs on the test bed hardware from Section 7).
7. Real-World Applications and Test Beds

For analyzing our approach of combining temporally limited resource requests, the priority inheritance protocol (PIP) and dynamic hinting, we extended the SmartOS kernel as described. In addition, we also implemented the priority ceiling protocol (PCP) according to [9]. This allowed us to perform a comprehensive and meaningful comparison between two very common strategies with different claims (e.g. generous resource assignment for PIP and deadlock prevention for PCP). Since PCP showed significantly less performance in most test beds, we omit deeper (algorithmic and implementation) details here but refer to [9] instead. However, for its integration, some specifications from section 5.1 were slightly modified as follows:

- **Modifies S2:** At compile time, each resource \( r \in R \) receives its individual ceiling priority

\[
c(r) = \begin{cases} 
0 & \text{if } r \text{ is unused} \\
\max\{P_t | t \text{ might request } r\} & \text{otherwise}
\end{cases}
\]

In consequence, task base priorities are not dynamic any more. Otherwise, this would not only require the recomputation of ceiling priorities at runtime but also jeopardize the deadlock-freedom of PCP when changing \( P_t \) of any \( t \in T \) during last resource allocations.

- **Replaces S5:** If a task \( t \in T \) requests a resource \( r \in R \), it immediately succeeds and remains running if it already owns \( r \) (\( \sigma_r = t \)) or if (according to the PCP policy) both

1. \( \sigma_r = \emptyset \) (\( r \) is free), and
2. \( p(t) > \max\{c(s) | s \in R \land \sigma_s \notin \{\emptyset, t\}\} \) (no other task holds any resource with ceiling priority equal or above the requester’s active priority)

hold. Otherwise, \( t \) transits to waiting state until it receives the resource or the timeout is reached. Yet, a blocking task \( v \) inherits at least \( t \text{’} s \) active priority: \( p(v) \geq p(t) \).

- **Replaces S11:** If a task \( t \in T \) releases a resource \( r \in R_u \), the system checks if it can serve another task \( u \) waiting for a resource \( s = \alpha_u \) (this is not necessarily \( r \)) which was previously not granted due to the PCP policy. If so, \( \sigma_s := u \) and \( u \) leaves waiting state. Anyway, \( p(t) \) is updated according to PCP.

Both implementations were done for Texas Instrument’s MSP430 [26] family of microprocessors, since these are found on a large variety of sensor nodes like e.g. TelosB [27], GumSense [28] or SnoW3 [29]. Requiring 4 kB of ROM and 40 B of RAM for the whole kernel, the typically low computational performance and small memory of sensor nodes was considered carefully to leave sufficient room for the actual application. Our test scenarios were executed on boards with an MSP430F1611 MCU running at 8 MHz. For detailed performance analysis at runtime, we used the integrated SmartOS timeline with a resolution of 1 μs. Note, that beside the definition of the ceiling priorities for PCP, the used test bed sources were exactly identical. Just the kernel was adapted to apply either PIP or PCP.

7.1. Test Bed I - Continuous Data Streaming

Our first test bed considers a problem we had in one of our real SANet applications. It addresses a quite frequently encountered situation: A task \( S \) is used to continuously transfer some data over a shared bus \( b \) to an external device. The stream is rather long (or even infinite) but could be interrupted and resumed at any time for more important communication over the same bus. Then however, it always needs some bus setup plus a complex header/trailer for proper initiation/termination. During the transfer, \( S \) obviously requires exclusive access to \( b \).

A common solution is to split the stream payload into atomic packets. Then, \( S \) would terminate the stream and release the bus temporarily after each packet. This way, other tasks may receive the bus regularly. However, since \( S \) does not really know if it currently blocks a more relevant task, the temporary stream interruption and release of \( b \) might be completely unnecessary. It is also obvious that the selected packet length has significant influence on the extent of potentially resulting bounded priority inversions as declared in Section 4. By using short (long) packets, the overhead increases (decreases) while improving (degrading) the reactivity of higher prioritized tasks when these request \( b \). In fact, a fixed length is often selected during development with regard to the individual application requirements. These must be known exactly, then.

Using a server task for coordinating the bus access might even result in slightly worse performance due to client-server communication overhead. The mentioned problems remain the same but are concentrated at the server which commonly also creates atomic packets or grants exclusive bus reservations. Finally, dynamic hinting provides two improvements. Since our approach knows about pending bus requests, \( S \) could query its current blocking state periodically and react only if necessary. This time the query interval must still be selected carefully but the overhead for useless stream interruption is already avoided. The additional use of hint handlers or early wakeup even gains the desired reactivity, as both instantly inform \( S \) if it blocks a task with truly higher priority. Therefore, these options must be enabled during the entire transmission or at least for the delay/suspension between two subsequent data words.

For the concrete application we had to stream 8 bit ADC data sampled at 10 kHz over an SPI bus. The overhead for each header and trailer was 1 byte. Beside, a radio transceiver \( R \) and a motor controller \( M \) shared the same SPI bus (at different settings) for short communication. Yet, both associated tasks \( R, M \) had to process sporadic events (av. inter-arrival time: \( \approx 5 \text{ ms} \)) and were much more time and safety critical, since especially failures in the motor control were disastrous. So, we defined \( P_M < \varphi = 100 < P_R < P_M \). For our application and testbed, we implemented the streaming task \( S \) to operate the bus (at best effort) in two completely different ways.

To reduce its CPU load, we first used a DMA channel for sending the ADC values continuously to the bus controller. Thus, the streaming task \( S \) simply had to allocate and configure the bus resource for a new stream. After starting the DMA transfer, \( S \) did sleep until an event signaled to finalize the stream or a hint occurred. The following example code
shows the relevant implementation details for $S$ when using early wakeup. The hint handling itself is highlighted and implemented according to Section 5.4. Most notably, it also includes a proper handling of the DMA resource which depends on the operation of the shared bus:

```c
void streamData() {  // executed in context of task $S$
    int stop = 0;
    // start stream */
    getResource(&SPI, INFINITE, 0);
    cfgBus(); header(); startDMA();
    // */ Try*/ to wait infinitely for the stopStream event.
    Enable early wakeup for $p(S)$ raised >> $\varphi = 100$.*/
    stop = waitEvent(&stopStream, INFINITE, $\varphi$);
    if (stop == -1) {  // "Catch" a hint!
        Resource_t * hint = getHint(NULL, NULL, NULL);
        if (hint == &SPI) {  // conditional hint handling
            stop = 0;
            stopDMA(); trailer(); releaseResource(&SPI);
        }  // */ THE TASK WILL BE PREEMPTED HERE SINCE AT ---*
        // LEAST ONE OTHER TASK WAITS FOR THE RELEASE --*/
        get_resource(hint, INFINITE, 0);
        cfgBus(); header(); startDMA();
    }
    // */ stop stream */
    stopDMA(); trailer(); releaseResource(&SPI);
}
```

In fact, the code is very similar when not using a DMA but sending the data words directly. Then, the DMA related functions can be removed and line 9 can be replaced by e.g.

```c
SPI_TX(nextData());  // send next data word
stop = sleep(100, $\varphi$);  // delay 100 ms, early wakeup if $p(S)$ $\geq$ $\varphi$
```

In our second implementation, we required the streaming task to also process the data before sending it over the bus. Therefore, it fetched each sample from the ADC and applied some functions (from a closed source library) first. The additional CPU load significantly reduced the sleeping time (in waiting state) between two data words, and in consequence it additionally limited the use of early wakeup. Thus, we implemented a hint handler and associated it with the streaming task $S$. This way, the resource manager was able to successfully hint $S$ at any code position by simply injecting the handler. The handler operated fully transparent to $S$ and temporarily released the critical bus resource $b$ – even during the execution of the original foreign code. Again, we selected $\varphi = 100$. For better understanding, the following code example makes exclusive use of the hint handler:

```c
void streamData() {  // executed in context of task $S$
    // register and enable dynamic hint handler */
    setDHH(DHH_Stream); enableDHH(p);
    // */ start stream */
    getResource(&SPI, INFINITE, 0); cfgBus(); header();
    // */ get, process & send data until stopStream occurs */
    while (!checkEvent(&stopStream) SPI_TX(nextData()));
    // */ done. disable hint handler and stop the stream */
    disableDHH(); trailer(); releaseResource(&SPI);
}
```

```c
OS_DHHANDLER(DHH_Stream) {  // task specific handler
    Resource_t * hint = getHint(NULL, NULL, NULL);
    if (hint == &SPI) {  // conditional hint handling
        trailer(); releaseResource(hint);
    }  // */ THE TASK WILL BE PREEMPTED HERE SINCE AT ---*
    // LEAST ONE OTHER TASK WAITS FOR THE RELEASE --*/
    get_resource(hint, INFINITE, 0); cfgBus(); header();
}
```

Processing data while simultaneously running some sporadic but highly reactive tasks might already cause extreme system load for low performance embedded systems like sensor nodes. In particular, if resources beside the CPU must also be shared. Yet, the test bed results will show, that our approach can still gain good reactivity and high throughput without manual task tuning. First, we implemented the application with atomic fixed-length packets (AP), then we used dynamic hinting with explicit querying (EQ), and finally we activated early wakeup (EW) and hint handlers (HH).

For our analysis we consider the PIP based system first. Figure 7 shows the results in terms of the average blocking delay $\tau$ of the real-time tasks and the achieved payload data rate $\rho$ of the streaming task. Due to the fixed trailer length and sampling rate, the best case blocking delay $\tau_{bc}=100$ $\mu$s and the best case payload data rate $\rho_{bc}=10$ kB/s. As expected for the packet oriented design without dynamic hinting, its throughput $\rho_{AP}$ improves while the blocking delay $\tau_{AP}$ degrades rapidly with increasing packet length. When using dynamic hinting with periodic explicit querying and on-demand release only, $\rho_{EQ}$ remains nearly constant and close to the achievable maximum $\rho_{bc}$. On the other hand, the selected period causes the blocking delay to behave analogous to the fixed packet version: $\tau_{EQ}$ almost matches $\tau_{AP}$ and is also not satisfying for long periods. When using early wakeup with the DMA version, the data rate is still held high, and additionally the blocking delay (which also includes stopping the DMA) is kept extremely low. Indeed, $\rho_{EW}$ $\approx$ $\rho_{bc}$ and $\tau_{EW}$ $\approx$ $\tau_{bc}$. Finally, when using the hint handler with the data processing implementation, we still obtain roughly the same reactivity for the real-time tasks: $\tau_{HH}$ $\approx$ $\tau_{EW}$ $\approx$ $\tau_{bc}$. Yet, we can observe a slightly decreased (but still constant) data rate $\rho_{HH}$ close to $\rho_{bc}$. The reason is not the bus or resource management itself, but the fact that the real-time tasks produce variable CPU load which sometimes slows down the data processing within the streaming task. Anyhow, their deadlines were never violated during our tests.

For better comparability, the data rates $\rho_{EW}$, $\rho_{HH}$ and blocking delays $\tau_{EW}$, $\tau_{HH}$ are visible as horizontal lines in Figure 7, though early wakeup and hint handlers are independent from any block length or query period, but occur just on demand.

It is worthwhile to note, that in this test bed (since it shares just a single resource among only three tasks), PIP behaves very similar to PCP and produces the same priority assignments, blocking and program flow. Therefore, the evaluation of PIP
implicitly applies to PCP, too. In fact, the observed differences always ranged at ± 0.037 ms for \( \tau \) and ± 0.044 kB/s for \( \rho \).

This stream test showed practical results from a real and quite common application scenario\(^2\). It comprised resource dependencies as well as resource blocking, which was mainly caused by one running, sleeping, waiting or simply preempted task. Our approach was able to significantly improve both, PIP and PCP to achieve almost maximal data rates and minimal blocking delays. However, only few tasks and one shared resource were involved.

7.2. Test Bed II - Dining Philosophers Stress Test

The next step is a tough stress test comprising many tasks, resources and deadlock pitfalls. Though being more synthetic, the resulting test application still considers real world demands and allows a deep analysis of our approach under extreme conditions. Furthermore, it demonstrates the use of time-utility-functions. Inspired by the well-known dining philosophers problem [30], where philosophers (tasks) over and over compete for cutlery (resources) that would allow them to eat, we modified the scenario to be more complex.

First, we extended the classic one dimensional problem to two and three dimensions. As Figures 8a,b show, this causes more extensive task-resource-dependencies and boosts competition between the philosophers as well as the overall system load. For \( m \) philosopher tasks \( P_0,...,m-1 \) in an \( n \)-dimensional setup, the system will contain \( |R| = n \cdot m \) resources and each task requires \( 2 \cdot n \) resources for eating. Then, each one directly bars its \( 2 \cdot n \) neighbors from also doing so since it exclusively holds at least one of their shared resources. Finally, the number of potential allocation cycles (deadlocks) significantly increases along with the dimension and task count.

At the individual start time \( t_0 \) of each philosopher’s period (→Fig. 8c), the corresponding task tries to quickly allocate its required resources. The entire allocation attempt is temporally limited to \( t_{TO} \). If the timeout \( t_{TO} \) is reached, the philosopher gives up, releases all resources it allocated so far, and restarts its lunch cycle. On success, the allocation delay \( t_A \) is logged. Then the task consumes some fixed time \( t_E \) for eating and finally releases its resources before thinking for a fixed time \( t_F \). The relationship to real-world embedded applications are tasks executing repeated actions for which they require the CPU and some exclusive resources with a certain period stability.

\(^2\)Note that this scenario closely relates to the theoretical Sleeping Barber Problem [30] with \( S \) being the barber sleeping in his chair \( b \) until customers \( R, M \) demand a hair cut and thus also require \( b \). Yet, priorities are involved here and \( S \) does not necessarily sleep while holding \( b \).

Figure 8: The Dining Philosophers Problem: 1D, 2D, The Lunch Cycle

Figure 9: The Philosophers’ Utility (top) and Time-Utility-Function (bottom)

Again, the most interesting point is the applied resource allocation concept. Similar to common application logic, each philosopher requests its resources in a fixed order. Therefore, it specifies the same absolute timeout \( t_0 + t_{TO} \) for each part \( r \) of the cutlery while enabling early wakeup as follows:

For our analysis, we applied dynamic hinting with both, PCP and PIP in the following ways:

1. PIP / PCP: We disabled the hints completely (\( \varphi = 0 \)) to study the performance of the pure priority inheritance/ceiling protocol.

2. PIP+EW / PCP+EW: The philosophers used early wakeup (\( \varphi = 1 \)) during their resource allocation stage. They were always collaborative and released each hinted resource on demand.

3. PIP+EW+TUF / PCP+EW+TUF: We applied a time-utility-function [7] for dynamic runtime decision as follows: Initially, we considered the allocation timeout to be hard, and specified each task’s utility at time \( t \) as binary function

\[
u(t, t_{TO}) := \begin{cases} 1 & \text{if } t \leq t_0 + t_{TO} \\ 0 & \text{otherwise} \end{cases}
\]

Next, for each of the \( 2n \) required resources an average allocation timeout \( t_{TO} \) can be accepted. Thus, whenever a task received a hint, it checked if its own remaining timeout \( t_{remain} = t_0 + t_{TO} - t \) was sufficient to allocate (in average case) its still required resources \( R' \) plus the one which would be released when accepting the hint. To also resolve deadlocks we used the time-utility-function

\[
f(t_{TO}, t_{remain}, R', n, \text{deadlock}) := \begin{cases} 1 & \text{if } (t_{remain} > (|R'| + 1) \cdot \frac{t_{TO}}{2n}) \\ \text{or deadlock == true} & 0 \text{ otherwise} \end{cases}
\]

to decide between following (1) or ignoring (0) the hint. Figure 9 shows the corresponding graphs of \( u \) and \( f \) as functions of \( t_{remain} \). Still, \( \varphi = 1 \) was used, and further hints during the re-allocation attempts were considered in the same way.
During hint re-allocation in both variants 2. and 3., further hints were considered in the same way, respectively. In summary, we implemented each philosopher to potentially set back its entire meal for other more important tasks. But as soon as it has started eating, it won’t stop for anybody else. It’s just the same in many real applications: a complex process might be deferred in time for the benefit of a more important task. But when in progress once, it is not aborted.

For each of the three presented alternatives, we inspected all 48 test bed setups from the following configuration space:

- Tasks / Dimensions: $m \in \{4^1, 9^1, 16^1, 2^2, 3^2, 4^2, 2^3, 3^3\}$
  \[ \Rightarrow \text{Resources: } |R| \in \{4, 9, 16, 8, 18, 32, 24, 81\} \]
- Resource allocation timeouts [ms]: $t_{TO} \in \{500, 1000, \infty\}$
- Eat and think duration [ms]: $t_E, t_T \in \{500, 1000\}$

Essentially, the resulting basic characteristics were the same for any value of $m$. Hence, we’ll just present a small but representative selection for $m = 4^2 = 16$ philosophers (→Figure 8b).

Obviously, a thinking philosopher allows each of its neighbors to eat. Thus, $\chi = \frac{t_E}{t_T}$ is an indicator for the average system load. For $\chi = 1$, the tasks might perfectly interleave their eat/think processes if this is allowed by the resource assignment policy. However, due to various overhead (e.g. context switches, etc.), this is never visible in a real run. Instead, the whole system already faces a slight overload condition, then. This overload increases with $\chi > 1$ and turns into underload for $\chi < 1$.

As first metric for the achieved performance, we counted the number of each philosopher’s successful lunch cycles $l$ in relation to the possible maximum $l_{\max} = t_{\text{max}} \cdot t_E + t_T$ with $t_A = 0$. Second, we considered the average allocation delay $t_{A,av}$ in relation to its maximum $t_{TO}$. As third metric we counted the number of deadlock situations during each run. The presented results were averaged over 10 runs à 20 min.

Again, we start our analysis with the PIP based system. Figures 10a,b show the results for philosophers with increasing base priorities $P_{Pi} = 1 + i$ and identical $t_{TO} = t_E = t_T = 500$ ms. Since $\chi = 1$, a lunch count close to 100% might be possible.

Using the pure priority inheritance protocol already supports the increasing task priorities partially as expected (→Fig. 10a). However, the values exhibit a clear jitter and the average lunch cycles for all tasks settled at 47.4%. The jitter is even worse for the allocation delay $t_{A,av}$ in Figure 10b (the best case $t_{A,bc} = 4.0\%$ evolves if no blocking occurs while just the task load and allocation overhead is considered). In almost all of our setups this phenomenon occurred when using PIP only. Obviously, the high variance arises from the tasks’ missing knowledge about each others requirements. The same is true for the system wide deadlock count which reached an average of $\approx 161$ per minute.

Using collaborative resource sharing by means of dynamic hinting instantly improved all results while obeying the philosophers’ base priorities much better. First, when following each hint, deadlocks are obviously avoided entirely (→Section 5.4). Beyond, the average lunch count increased to 79.5% at significantly less jitter. This is especially true for the allocation delays $t_{A,av}$ which are more stable around 16.2% of $T_{TO}$.

Finally, by also applying the TUF described above, we observed additional improvements in most setups. Now, the philosophers will only collaborate if they can afford it. Let’s consider the consequences: Along with falling priority, tasks tend to receive more hints and less CPU time. Thus, they also tend to get ever closer to their allocation timeout $t_{TO}$ and behave more egoistic when short in time. This results in a slight reduction of lunch counts for high priority philosophers but significantly increases the lunch count for the low prioritized ones. Due to the selective collaboration, the number of deadlock situations might also rise again. In our testbed we counted $\approx 2$ deadlocks per minute indeed, but nevertheless the overall lunch count further improved to 85.2\% of the potential maximum. The allocation delays $t_{A,av}$ were also reduced further and stabilized even better around 12.4\% of $t_{TO}$.

Unfortunately, for massive underload/overload setups, the TUF yielded only slight improvements for the lunch count compared to the PIP+EW method. Then, the load was either man-

---

**Figure 10:** Dining Philosophers Test Bed under PIP (2D, 4×4 Philosophers)

**Figure 11:** Dining Philosophers Test Bed under PCP (2D, 4×4 Philosophers)
Figure 12: Dining Philosophers Test Bed Comparison (2D, 4×4 Philosophers)

ageable anyway or it was simply too extreme. However, in any setup, dynamic hinting was always significantly better than the pure PIP. Especially when using $T_{TO} = \infty$, the pure PIP always got stuck in deadlocks while our approach recovered reliably and still achieved good results (for $T_{TO} = \infty$, the TUF considered deadlock situations only).

Though extending PIP yielded much better results, we’ll finally address and compare the PCP based system. The mentioned problems from Section 4 are particularly obvious within this test bed. PCP’s conservative policy sometimes leads to unnecessary denial of free resources. Here, this avoidance-related-inversion causes PCP to only serve the highest priority tasks which mutually pass their resource to each other, then. For comparison, Figures 11 and 10 are based on the same settings.

When choosing $T_T = \lambda \cdot T_E$ and thus $\chi = \frac{1}{\lambda}$ each thinking philosopher allows $[\lambda]$ others to start their lunch. As expected, exactly the $[\lambda] + 1$ tasks with highest priority will settle and eat alternately, leading to the final starvation of the others (→ Fig. 11a, $\lambda = 1$). In theory, the allocation delay is minimal for perfect task interleaving. Indeed, Figure 11b shows values close to the PIP version, but only for non-starved tasks. Yet, these are also not optimal due to the OS overhead and the high task load.

To allow eating for all philosophers under PCP, we had to choose $T_T = (m - 1) \cdot T_E$, i.e. a load of just $\chi = \frac{1}{17} = 6.7\%$ (→ Fig. 12). However, $l_{av}$ yielded $\approx 96.3\%$, then. To also achieve $l_{av} \geq 96.3\%$ under PIP, it was sufficient to reduce the load from 100% to $\chi = 66.7\%$. Then, the average allocation delay $T_{A,av}$ even reached stable $4.8\% \approx T_{A,bc}$ under PIP compared to unstable $85.3\% \gg T_{A,bc}$ under PCP.

The direct comparison shows two issues: First, compared to the stream test, the usage of dynamic hinting under PCP provides less advantage for the philosopher test. Since PCP is inherently too restrictive and prophylactically denies low priority tasks access to requested resources, these block others less frequently. Thus, hints are also rarely generated and the advantage of our concept vanishes. Second, PIP avoids unnecessary denials due to its generous policy and serves tasks more often. Though resulting problems like chained blocking or deadlocks are serious then, these produce hints when necessary and are handled and resolved excellently by dynamic hinting at runtime.

As our test beds show, choosing PIP as underlying protocol for our novel approach allows to combine the advantages of PIP and PCP dynamically: tasks receive resource allocation delays according to their relative priorities while still achieving a high progress rate. In summary, the introduced reflection and collaboration concept always achieves significantly better results than the sole classic conventional techniques.

8. Conclusion and Outlook

In this paper, we introduced the novel dynamic hinting approach for collaborative resource sharing among preemptive tasks in reactive systems. We showed, that dynamic hinting – preferably in combination with the priority inheritance protocol and temporally bounded resource requests – can help to improve and stabilize the overall system performance. Therefore, our concept helps to reduce resource allocation delays and to recover from deadlocks at runtime. In particular, the individual task base priorities are considered carefully to keep each task’s performance and reactivity close to its intended relevance. The basic idea is to analyze emerging task-resource conflicts at runtime and to provide blocking tasks with information about how they can improve the reactivity and progress of more relevant tasks. Therefore, and depending on their current state (running, ready, waiting), we also introduced three central techniques for passing hints to them when required. In any case, our reflective approach allows each task to dynamically decide between collaborative or egoistic behavior with respect to its current conditions and other tasks’ requirements. By following the hints from the resource manager, tasks can implicitly collaborate without explicit knowledge of each other. However, dynamic hinting can initially not guarantee any time limits since these highly depend on the behavior of the involved tasks. Yet, even if used sparingly, our approach is at least equal and in most cases even significantly better compared to non-collaborative operation.

Our test beds and the integration of all presented concepts into the preemptive real-time operating system SmartOS showed, that the effective use of prioritized tasks for creating reactive systems is even possible on resource constrained and computationally weak devices like sensor nodes. Of course, a well-thought application design still remains elementary, but compositional software design is already facilitated. In general, our approach is not necessarily limited to networked sensor/actuator systems but may also extend other embedded systems.

Concerning real-world applications, we already integrated dynamic hinting into a SANet based indoor localization and vehicle steering system, where we achieved a considerably higher localization frequency and path precision due to faster event handling and data transmissions. Most recently, we applied it for collaborative memory management in open real-time systems [31]. Here, task-controlled heap re-organization in case of memory shortages avoided brute force memory withdrawals, data loss, and critical system states, while still considering priorities and allocation deadlines. At present we are working...
on more sophisticated concepts for adjusting the acceptance of hints to the task and system situation. In particular, we see improvements concerning the hint selection itself and the application of TUFs by considering more application specific factors. For example, this would allow us to relate the tight and varying energy reserves of autonomously operating wireless sensor/actuator systems to their reactivity requirements. Also, we plan to evaluate the use of dynamic hinting for remote resource management in distributed [32] and multi-core systems [13, 33], where blocking may induce hints between the subsystems. When considering database systems or software transactional memory (STM) techniques [34], an improvement might also emerge from controlled on-demand interruptions of transactional memory (STM) techniques [34], an improvement might also emerge from controlled on-demand interruptions of interfering operations instead of common, but quite problematic aborts and restarts. Finally, another more general focus is the application of software model checking (cf. [35]) for the validation of systems using our new approach.

[16] A. M. K. Cheng, J. Ras, The implementation of the Priority Cell-