Dynamic Hinting: Real-Time Resource Management in Wireless Sensor/Actor Networks

Marcel Baunach
University of Wuerzburg, Germany

Dynamic Hinting: Real-Time Resource Management in Wireless Sensor/Actor Networks

Marcel Baunach
Department of Computer Engineering, University of Würzburg, Am Hubland, 97074 Würzburg, Germany
Email: baunach@informatik.uni-wuerzburg.de

***Abstract*—Increasing complexity of today's WSAN applications can rapidly result in reduced real-time capabilities of the underlying sensor nodes. Using preemptive operating systems is one way to retain acceptable reactivity within highly dynamic environments but commonly leads to severe resource management problems. We present the *Dynamic Hinting* approach for maintaining good system reactivity by efficient combination of preemptive task scheduling and cooperative resource allocation. With respect to task priorities, our technique significantly improves classical methods for handling priority inversions 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 resource sharing. In some cases this even allows to improve blocking delays as otherwise imposed by bounded priority inversion.**

I. Introduction

The ever increasing size, pervasiveness and demands on today's wireless sensor/actor networks (WSAN) significantly boost the complexity of the underlying nodes. Thus, modular hardware and software concepts (e.g. service oriented programming abstractions) are more and more used to manage design and operation of these embedded systems. Then, adequate interaction between the various modules is essential to avoid typical compositional problems. Beside task scheduling [1], directly related issues comprise resource sharing or even real-time operation [2]. Concerning this, we find that current WSN research is still too limited to static design concepts. Yet, as already stated in [3], next generation embedded systems will be more and more used as reactive real-time platforms in highly dynamic environments where the true system load varies and can not be predicted a priori during development. In fact, we also expect a clear focus shift from pure sensing in classic WSNs toward additional pro-activity in WSAN applications (e.g. integrated control systems, precise on demand measurements, time services, etc.). Then, preemptive and prioritized tasks are required for fast response on various events but further complicate resource assignment and reactivity.

In this paper we present the *Dynamic Hinting* approach for cooperative resource sharing and real-time operation within preemptive operating systems. Dynamic hinting enables compositional software design by allowing independently implemented tasks a collaborative use of shared resources. As often suggested (e.g. in [4]), 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 even deadlocked) tasks to adapt to 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, and 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 [5]. To allow long-term allocations (as frequently required for hardware) dynamic hinting extends the classic priority inheritance protocol and avoids some related shortcomings of other techniques.

This paper is organized as follows: Initially, we will further motivate the need for real-time and preemptive operating systems in sensor networking. Then, we'll outline some related concepts 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 efficiently usable even for low performance devices like sensor nodes. Therefore, we will present application examples and the impact on the programming model before performance results from real-world test beds close this paper.

II. Motivation and Requirements

An operating system has significant influence on the overall system performance since it coordinates task interactions as well as the access to operational resources like hardware components or data structures. In many cases, these resources need to be shared among several tasks. 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 systems 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, efficient methods already exist (→Sections III, IV). 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 processor, is often managed by the task scheduler in a *preemptive* way, we believe that the operating system should also coordinate access to other, *exclusive* resources by contributing appropriate mechanisms. Indeed, several resource management techniques were already implemented for sensor nodes but most of them do not address the special aspects of reactive real-time operation.

A. Requirements for Sensor/Actor Networking

During our research and practical work, we found that reactivity and pro-activity in modern WSAN applications requires quite sophisticated real-time and resource concepts. We'll give just one short example from one of our real-world applications for motivating this need toward the reader:

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. 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 further radio packets. Concurrent to this communication task, sensor tasks often use exactly the same bus for continuous streaming of data (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. 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. The regular release of a long-term resource could be a solution. However, this strategy might impose considerable overhead when deallocation and re-allocation are expensive in time and energy. Where data streams often require explicit termination (trailers) and initiation (headers), resources might require a time-consuming (de)initialization procedure upon each (de)allocation. Using server tasks for managed operation of such resources is also no universal option if the performance for this abstraction is simply not available.

In addition to sporadic resource sharing, we found the use of dynamic task priorities convenient for adaptation to changing environmental conditions. However, this complicates resource sharing even further but should be considered. After motivating our requirements for real-time resource management in WSAN systems, we'll now overview some related work.

III. Related Work in WSN/WSAN Systems

Non-preemptive systems with run-to-completion tasks are very common in sensor/actor networking 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 virtualizing of resources like in the TinyOS component concept [6]. However, large resource hierarchies might result in severe intercommunication overhead and reduced overall performance.

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 and again depends on the currently running task and the scheduling policy. Notice, that the non-preemptive use of the CPU can already lead to priority inversion then (→Section IV). Therefore, TinyOS and Contiki [7], 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, this feature also complicates resource sharing: Preemption yields no instant advantage if the action requires a shared resource which is exclusively held by a less important task. Resulting problems like priority inversion [8] 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) or *priority inheritance* (PIP) [9], [10] are found in some embedded operating systems. Beside causing some runtime overhead, these techniques also suffer from certain weaknesses addressed in the next section. Furthermore, preemptive WSN operating systems like e.g. MANTIS [11] or RETOS [12] do not consider real-time or resource related problems at all.

IV. 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 currently allocated by a lower priority 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.

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. From these alternatives, we selected the priority inheritance protocol as basic technique for our dynamic hinting approach.

Though priority ceiling and highest locker inherently prevent deadlock situations and even restrict the maximum allocation delay to at most one blocked resource per task, both techniques imply a serious problem regarding long-term resource allocations. Except for this problem, we won't go into detail about these techniques here but refer to [2], [9], [13] instead.

For deadlock avoidance, HLP and PCP use a rather conservative policy when adjusting task priorities and deciding if a resource request is granted or denied. In some cases this rapidly leads to a problem often referred to as *avoidance-related-inversion* [13]: If a task T_1 with current priority $p(T_1)$ holds a resource, PCP refuses the assignment of any remaining but free resource r to any other task T_2 with $p(T_2) \leq p(T_1)$. HLP implicitly acts in a similar way by avoiding the execution of such a task T_2 entirely [10]. Although these implicit and anticipatory reservations would allow a fast assignment of further resources to T_1, it is 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 raised $p(T_1)$ above its original *base priority* while in fact, T_2 is specified to be truly more relevant than T_1. In summary, a task implicitly blocks all other tasks with equal or lower priority while it simply holds a (maybe rarely shared) resource. When recalling our motivation for both real-time and long-term resource allocations from Section II-A, it becomes obvious that such a behavior is highly critical. Another problem with PCP/HLP is that dynamic *base priorities* are hard to implement. This option would either bring back deadlocks, or it must be disabled for tasks which currently hold or request a resource. Yet, this flexibility is interesting for reflecting changing environmental conditions or for server tasks which adopt to the priority of their (most relevant) clients.

Compared to PCP and HLP, 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.

Indeed, 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 [14]. However, we will show that both shortcomings can be handled by our extension in an efficient manner. In our opinion, deadlock avoidance is not very practical in most WSAN applications. E.g. the frequently recommended Banker's algorithm starts a new task only if its worst case resource requirements can still be satisfied when all other running tasks also claim their demands entirely. Maintaining this so called *safe state* implies two major problems. First, a (time critical) task might not start promptly when required. Second, two tasks that *might* require the same unique resource may never run interleaved. Another strategy is to require the allocation order for resources to be fixed and inverse to their release order. In many scenarios this is neither possible nor desired. The constraint might 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. While undoing the work so far, it might also leave the resources in an undefined state making a handover problematic. 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 on typical sensor nodes.

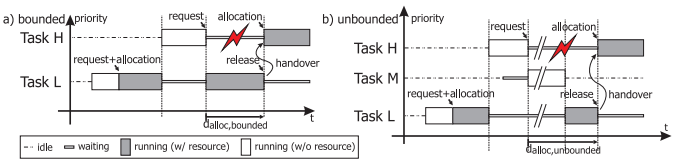

Figure 1. Priority Inversion (bounded and unbounded)

So far, we presented our requirements for real-time aware resource management in WSAN applications along with some already available concepts and systems. Additionally, 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.

V. Resource Management and Dynamic Hinting

This section presents the details about our resource management approach. In fact, 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 *Smart*OS [15] since it is available for several sensor nodes, provides appropriate task, timing and resource basics, and thus allowed an easy integration.

Since *Smart*OS 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. This way, tasks may react on resource allocation failures and event imponderabilities without blocking the whole system. Beside, this already provides a simple method for deadlock recovery.

We'll now formalize the extended resource management policy of *Smart*OS (V-A) along with our new approach for priority inheritance (V-B) and dynamic hinting (V-C).

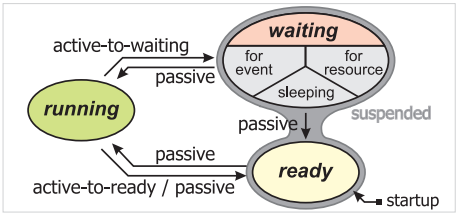

Figure 2. Task State Transitions under *Smart*OS

A. Extended SmartOS Specifications

S1 Each *Smart*OS system consists of a set of preemptive tasks T ($|T| \geq 1$) and resources R ($|R| \geq 0$). Each task is executed for the whole system runtime and can neither be started dynamically nor terminate entirely. Instead it is always in one of the following states (→Fig. 2): *ready, running*, or *waiting*. 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) active-to-waiting: the *running* task changes its own state to *waiting* by sleeping, requesting a resource which is currently allocated by another task, or by waiting for a not yet occurred event.
- b) active-to-ready: the *running* task might transit 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 Resources are non-preemptive. 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 specifying 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 $\alpha_t \in R$ at the same time:

$$\alpha_t = \begin{cases} \emptyset & \text{if } t \text{ awaits no resource} \\ r \in R & \text{if } t \text{ awaits } r \end{cases}$$

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

$$T_r = \{t \in T | \alpha_t = r\}.$$

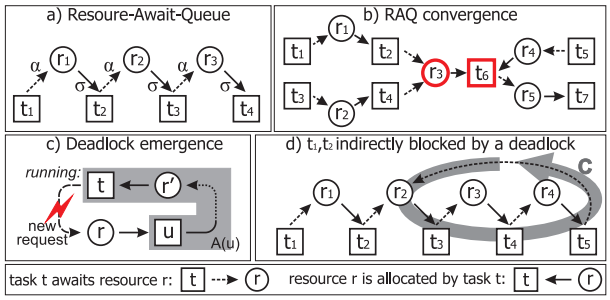

Figure 3. Examples for Resource-Await-Queues

S9 Each resource $r \in R$ may be assigned to at most one owner task $\sigma_r \in T$ at the same time:

$$\sigma_r = \begin{cases} \emptyset & \text{if } r \text{ is not assigned to any task} \\ t \in T & \text{if } r \text{ is assigned to } t \end{cases}$$

S10 Each task t may exclusively hold several resources $R_t \subseteq R$ at the same time:

$$R_t = \{r \in R | \sigma_r = t\}.$$

Allocation and deallocation order of resources are arbitrary and independent.

S11 If a task 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, the one which requested r first will receive it and leave waiting state (→S3c).

In consequence to these specifications, dealing with deadlocks will be required at runtime since the four Coffman conditions [14] 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 S10).

B. Priority Inheritance and Deadlocks 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 (→Fig. 3):

$$A(t) := \left(t, \underbrace{\alpha_t}_{\in R}, \underbrace{\sigma_{(\alpha_t)}}_{=u \in T}, \underbrace{\alpha_u}_{\in R}, \underbrace{\sigma_{(\alpha_u)}}_{=v \in T}, \alpha_v, \sigma_{(\alpha_v)}, \ldots \right)$$

Thereby, two structural properties become obvious:

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

Further 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 with a resource $r \in R$. Then, $\sigma_r = \emptyset$ and $\exists_{u \in T} \alpha_u = r$. 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, $t_1 \dots t_6$ 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 IV, 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* tasks in $A(u)$ and we want t to release its resources quickly to grant a fast resumption of more important tasks (→Fig. 4). Initially, we define

$$w(r) := \begin{cases} 0 & \text{if } T_r = \emptyset \quad (indeg(r) = 0) \\ \max\{p(t) | t \in T_r\} & \text{if } T_r \neq \emptyset \quad (indeg(r) \geq 1) \end{cases} \quad (1)$$

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

$$W(t) := \begin{cases} 0 & \text{if } R_t = \emptyset \quad (indeg(t) = 0) \\ \max\{w(r) | r \in R_t\} & \text{if } R_t \neq \emptyset \quad (indeg(t) \geq 1) \end{cases} \quad (2)$$

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

$$p(t) := \max\{P_t, W(t)\} \geq P_t. \quad (3)$$

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:

$$\forall_{u,v \in A(t) \cap T} : \exists_{r \in A(t) \cap R, \alpha_u = r, \sigma_r = v} \Rightarrow u \in T_r, r \in R_v$$

$$\Rightarrow p(v) \overset{(Eq.\ 3)}{\geq} W(v) \overset{(Eq.\ 2)}{\geq} w(r) \overset{(Eq.\ 1)}{\geq} p(u) \qquad \square$$

Again, Figure 4 gives an example. For now, deadlocks still require closer examination:
If $A(t)$ contains a cycle C (→Fig. 3d), then:

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

Let's consider the consequences: By requesting a resource r currently allocated by $u = \sigma_r$, a task t 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.

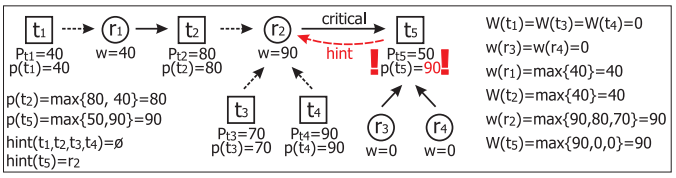

Figure 4. Example for Priority Inheritance and Dynamic Hinting

As requested in Section II, the formalization of our resource management policy and the priority inheritance protocol supports arbitrary and independent resource (de)allocation orders plus dynamic base priorities (→Section VI). Next we will present our dynamic hinting approach for improving bounded priority inversion, chained blocking and deadlock situations.

C. 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 II, IV 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 allocation succeeds. The resulting (active) loops or long timeouts might not only block other tasks but even worse, they simply shift the problem back from system level to task level. 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 (→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 a RAQ and reduces direct, chained or deadlock blocking of at least one higher priority task (→Fig. 5).
Therefore, two preconditions must be fulfilled:

1) An ongoing resource allocation must never prevent any task from requesting any resource. Otherwise, our approach lacks knowledge about the system requirements.
2) A spurious task must receive the time and opportunity to react on a hint. In our case, PIP provides the priority and the limited waiting of other tasks provides the time.

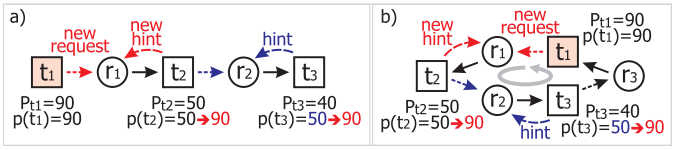

Figure 5. Dynamic Hinting Examples: a) Chain, b) Deadlock

The first step for determining hints is to identify the *critical resources* for each task t. These currently define $p(t)$ and thus, they directly or indirectly cause the blocking of the most important tasks in $A(t)$ with base priority truly above P_t:

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

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

$$crit(t) \neq \emptyset \Leftrightarrow p(t) > P_t. \quad (5)$$

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 task by releasing any $r \in crit(t)$. Yet, our approach always selects the hint as follows (→Fig. 4, 5):

$$hint(t) := r \in crit(t), r \text{ was requested last.} \quad (6)$$

Then, if t releases its hint, this resource is directly passed to its first requester (→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 (→S3b,c). As soon as t is scheduled again, it can immediately re-request the just released resource to continue its operation quickly. In any case, the untimely release of a hint resolved a priority inversion and accounted for the intended task base priorities.

The example in Figure 4 shows $crit(t_5) = \{r_2\}$ since $p(t_5) > P_{t_5}$ was defined by t_4's request for r_2. Releasing r_2 would instantly relax $p(t_5) := P_{t_5}$. Then t_4 is served and scheduled since it is indeed the task with highest priority but currently blocked by t_5. The allocation timeout t_4 specified for r_2 grants t_5 the time to cooperate as described. If t_5 follows its hint r_2 prior to its regular release, it indeed improves the bounded priority inversion toward t_4. Furthermore, t_5 also improves the reactivity of t_2 and t_3 since these tasks are also more relevant ($P_{t_2} > P_{t_3} > P_{t_5}$) and will receive r_2 right after t_4.

For better understanding of the implementation details in Section VI and the integration with PIP, we'll briefly address the situations in which a hint must be updated (→Eq. 4, 6).

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

Two issues are obvious: First, $hint(t)$ changes each time when PIP updates $p(t)$. Second, $hint(t)$ often changes while t itself is not running. However, it *might* become running then.

Thus, we'll now describe the ways in which a task may receive and handle its hints: First, an *Explicit Query* (`getHint(...)`) can be done at distinct points in time or at 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 (→Fig. 5b) and during many long-term allocations, where tasks e.g. stream data or wait for some events/interrupts while holding a resource (→Section II). Beside this severe weakness, the manual effort and code pollution would be immense.

We now introduce a much better strategy called *Early Wakeup*. When enabled, all functions by which a task suspends itself (→S3a) may return early upon a new or changed hint. Then, a dedicated return value will indicate this special situation. This way, coping with hints can be done instantly and it is entirely limited to the cases when they really occur. The use of early wakeup can be selected and tuned individually by each task t and for each self-suspension. Therefore, we extended the involved functions by an additional parameter φ:

```
int sleep(deadline | timeout, φ)
int waitEvent(event, deadline | timeout, φ)
int getResource(resource, deadline | timeout, φ)
```

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

$$\varphi \neq 0 \ \wedge \ p(t) > P_t \ \wedge \ p(t) \geq \varphi \quad (7)$$

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

E.g. both new requests in Figure 5a,b will immediately resume t_2 if it has early wakeup enabled. Then, its request for r_2 is withdrawn. Otherwise, or if t_2 then refuses to release its hint r_1 and simply requests r_2 again, t_3 may wake up early. Obviously, a single cooperative task in a chain or cycle is already sufficient to improve or recover from the situation.

Of course, priority thresholds are not the only useful metric for deciding between cooperative or egoistic behavior. Thus, beside the hinted resource r, we grant each task t 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 at which the hint r expires due to the latest request timeout:

```
Resource* getHint(Priority_t* p, boolean* DL, Time_t* TO);
```

The latter is of special interest for applying time-utility-functions as proposed in [5]. These allow to relate the remaining allocation time to the still remaining timeout. E.g. the timeout might suffice to still complete the current operation or it might be that short that the required time for releasing the hint would exceed it anyway.

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

We showed, that 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. Yet, our approach gives no guarantee about that, but depends on the behavior of the involved tasks. However, a single cooperative task is already sufficient for effective handling of these problems. When considering our demands on the resource policy, a soft advise is a good chance to avoid complex brute force recovery methods. Usage examples and corresponding test bench results follow in Section VII.

VI. Implementation Details

This section shows some central implementation details of the just described techniques. 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, let's 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) $p(v)$ of several tasks v might change if any base priority P_u is changed at runtime while $v \in A(u)$. This is rather simple and we'll omit the details here.

Before heading to the functional details, we'll consider the computational complexities for $w(r)$ and $W(t)$: Since the internal representation for each T_r is a priority queue, retrieving $w(r)$ is in $O(1)$. In contrast, each R_t is a list and thus $W(t)$ is in $O(indeg(t))$.

1) Resource allocation: getResource(resource, timeout, φ): This function either returns 1 (success), 0 (timeout) or -1 (hint). Here, two basic conditions must be considered:

a) 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.
b) If r is occupied by $\sigma_r \neq t$, t is suspended and priority management is done.

Case a) 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 b) $p(t)$ 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, $w(r)_{new} \geq w(r)_{old}$.

If $w(r)_{new} > p(\sigma_r)$, the partial order for $A(t)$ is violated and must be fixed by setting $p(\sigma_r) := p(t)$. In this case, $hint(\sigma_r) := r$ 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: $O(|A(t)|)$.

As we have seen in Section V-C, in case of early wakeup, we already stop at the first task v for which Eq. 7 is true and resume its execution. If v follows the hint, *dynamic hinting* was successful. Otherwise v will restart the just aborted request and by doing so, the hint will be passed to the next task in $A(v)$.

2) Resource deallocation: releaseResource(resource): Releasing a resource r is always initiated by its owner $t = \sigma_r$ ($\rightarrow$S4). If t holds r several times, one is freed and no priority adjustments are required. The same is true if t frees r entirely while $T_r = \emptyset$ since then, $p(t)$ was obviously not defined by r ($w(r) = 0$). 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, $\sigma_r := v, \alpha_v := \emptyset, T_r := T_r \backslash \{v\}$ and finally $w(r)_{new} \leq w(r)_{old}$. 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 $hint(t)$ might need an update according to Eq. 3-6. As desired, $P_t \leq p(t)_{new} \leq p(t)_{old}$ finally holds. Complexity: $O(indeg(t))$.

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, $\exists!_{v \in A(t)} : v = \sigma_r$ ($\rightarrow$Lemma 1) and $p(t) \leq p(v)$ ($\rightarrow$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 $hint(v)$ 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. 3-6 until the partial order is satisfied again in $O(|A(v)|) = O(|A(\sigma_r)|)$.

VII. Real-World Applications and Test Beds

For analyzing our approach of combining temporally limited resource requests, the priority inheritance protocol and dynamic hinting, we extended the *Smart*OS kernel as described. The implementation was done for Texas Instrument's MSP430 [16] family of microprocessors, since these are found on a large variety of sensor nodes like e.g. TelosB [17]. Requiring 4 kB of ROM and 150 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 *Smart*OS timeline with a resolution of 1 µs.

A. Test Bed I - Continuous Data Streaming

Our first test bed considers a problem we had in one of our real WSAN control 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 suspended 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 needs 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 IV. 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 value 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 also commonly 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 early wakeup even provides the desired reactivity as it hints S *instantly* if it blocks a task with truly higher base priority. Therefore, this option must simply be enabled for the delay/suspension between two subsequent transmissions of 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 ms) and were much more time and safety critical since especially failures in the motor control were disastrous. So, we defined $P_S < \varphi = 100 < P_R < P_M$. To further reduce the CPU load we 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:

```
1  void streamData() {    // executed in context of task S
   int stop = 0;
3  /* start stream */
   getResource(&SPI, INFINITE, 0);
5  cfgBus(); header(); startDMA();
   while (stop != 1) {       // 1 indicates stop event
7    /* Wait infinitely for the stopStream event.
        Enable early wakeup if raised >= φ = 100. */
9    stop = waitEvent(&stopStream, INFINITE, φ);
     if (stop == -1) {       // hint received!
11     Resource_t *hint = getHint(NULL, NULL, NULL);
       if (hint == &SPI) {   // conditional hint handling
13       /* stop stream and release resource quickly */
         stopDMA(); trailer(); releaseResource(hint);
15 /* --- THE TASK WILL BE SUSPENDED HERE SINCE AT ---
     --- LEAST ONE OTHER TASK WAITS FOR THE HINT ---- */
17       /* continue stream as soon as possible */
         getResource(hint, INFINITE, 0);
19       cfgBus(); header(); startDMA(); } } }
   /* stop stream */
21 stopDMA(); trailer(); releaseResource(&SPI);
}
```

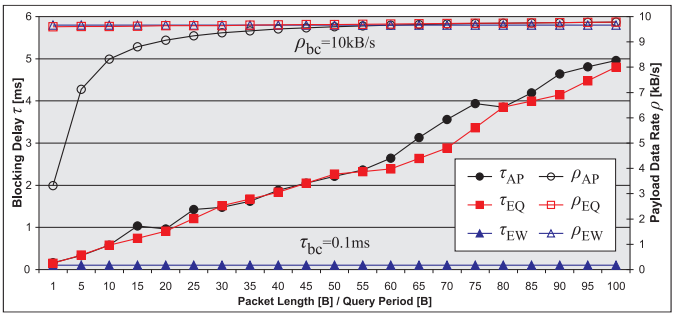

Figure 6. Streamtest: Packet Oriented (AP) vs. Dynamic Hinting (EQ/EW)

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.

```
send(nextDataWord);
stop=sleep(100, φ); // delay 100µs, early wakeup if p ≥ φ
```

Streaming data while simultaneously running some sporadic but highly reactive tasks might already cause extreme system load for low performance embedded systems like sensor nodes. Yet, our testbed results 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). Figure 6 shows the results in terms of the average blocking delay τ of the real-time tasks and the achieved payload data rate ρ of the streaming task. Due to the fixed trailer length and sampling rate, the best case blocking delay τ_{bc}=100 µs and the best case payload data rate ρ_{bc}=10 kB/s.

As expected for the packet oriented design, its throughput ρ_{AP} improves while the blocking delay τ_{AP} degrades rapidly with increasing packet length. When using dynamic hinting with periodic explicit querying, ρ_{EQ} remains nearly constant and close to the achievable maximum ρ_{bc}. However, the blocking delay τ_{EQ} almost matches τ_{AP} and is also not satisfying for long periods. Finally, when using early wakeup, the data rate is still held high and additionally the blocking delay is kept extremely low. Indeed, $\rho_{EW} \approx \rho_{bc}$ and $\tau_{EW} \approx \tau_{bc}$. For better comparability, ρ_{EW} and τ_{EW} are visible as horizontal lines in Figure 6. Yet, early wakeup is independent from any block length or query period.

This stream test showed practical results from a common and realistic application scenario. However, only few tasks and one shared resource were involved.

B. 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. Inspired by the well-known dining philosophers problem, where philosophers (tasks) over and over compete for cutlery (resources) that would allow them to eat, we modified the scenario to be more complex.

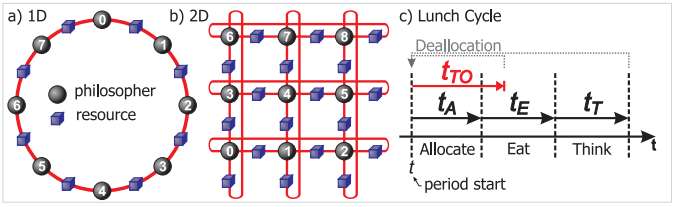

Figure 7. The Dining Philosophers Problem: 1D, 2D, The Lunch Cycle

First, we extended the classic one dimensional problem to two and three dimensions. As Figures 7a,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 $P_{0,\dots,m-1}$ 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 blocks at least one of their shared resources. Finally, the number of potential allocation cycles (deadlocks) increases significantly along with the dimension and task count.

At the individual start time t of each philosopher's period (→Fig. 7c), 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_T. 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.

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 $t + t_{TO}$ for each part r of the cutlery while considering early wakeup as follows:

```
allocationResult = getResource(r, t + t_TO, φ);
```

For our analysis, we applied dynamic hinting in different ways:

1. PIP: We disabled the hints completely ($\varphi = 0$) to study the performance of the pure priority inheritance protocol.
2. PIP+DH: The philosophers were *always* cooperative and released each hinted resource on demand ($\varphi = 1$).
3. PIP+DH+TUF: We applied a time-utility-function [5] for dynamic runtime decision as follows: For each required resource an average allocation timeout $\frac{t_{TO}}{2n}$ can be accepted. Thus, whenever a philosopher received a hint it checked if the remaining timeout was sufficient to allocate its still required resources R' in average case. If $t_{TO,remain} > |R'| \cdot \frac{t_{TO}}{2n}$ the hint was followed, otherwise it was ignored. Still, $\varphi = 1$ was specified.

During hint re-allocation in both 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 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 commonly not aborted.

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

- Tasks / Dimensions: $m \in \{4^1, 9^1, 16^1, 2^2, 3^2, 4^2, 2^3, 3^3\}$
  $\Rightarrow$ 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.

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. 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 even 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 $l_{max} = \frac{t_{testbench}}{t_A + t_E + t_T}$ 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.

Figures 8a,b show the results (average over 10 runs à 20 min.) for philosophers with increasing base priorities $P_{P_i} = 1+i$ and common values for $t_{TO} = t_E = t_T = 500$ 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 as expected (→Fig. 8a). However, the values exhibit a clear jitter and the average lunch cycles for all tasks settled at $\approx$ 47%. The jitter is even worse for the allocation delay $t_{A,av}$ in Figure 8b. 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 cooperative resource sharing by means of dynamic hinting instantly improved all results while obeying the philosophers' base priorities even better. First, when following *each* hint, deadlocks are obviously avoided entirely (→Section V-C). Beyond, the average lunch count increased to $\approx$ 79% at signifantly less jitter. This is especially true for the allocation delays $t_{A,av}$ which are more stable around 16% of t_{TO}.

Finally, by using the TUF described above, we observed additional improvements in most setups. Now, the philosophers are only cooperative 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 cooperativeness, the number of

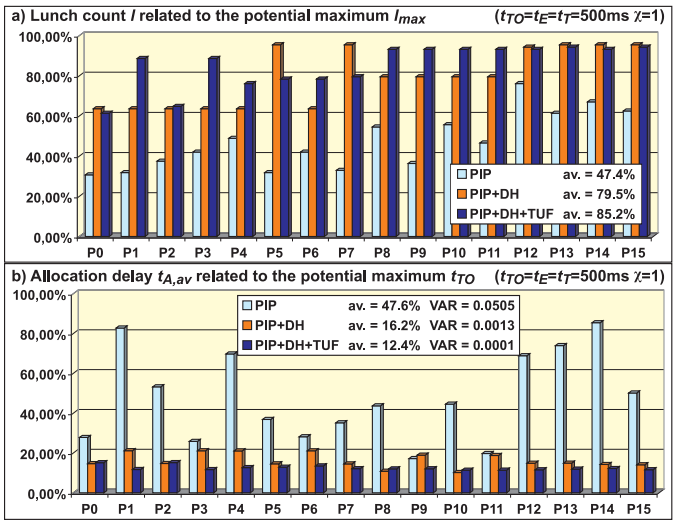

Figure 8. Dining Philosophers Test Bed Results

deadlock situations might also rise again. In our testbed we counted ≈ 2 deadlocks per minute indeed, but nevertheless improved the overall lunch count to ≈ 85% of the potential maximum (→Fig. 8a). The allocation delays $t_{A,av}$ were also reduced further and stabilized even better (→Fig. 8b).

Unfortunately, for massive underload/overload setups, the TUF yielded only slight improvements of the lunch count compared to the PIP+DH method. Then, the load was either manageable anyway or it was simply too extreme. However, in any setups, dynamic hinting was always significantly better than 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.

VIII. Conclusion and Outlook

In this paper, we introduced the *dynamic hinting* approach for cooperative resource sharing among preemptive tasks in reactive systems. We showed, that dynamic hinting in combination with priority inheritance 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 close to its intended relevance. The idea is to analyze emerging task-resource dependencies at runtime and to provide spurious tasks with information about how they could increase the reactivity of more relevant tasks. Nevertheless, our approach allows each task to dynamically decide between cooperative or egoistic behavior with respect to its current conditions and other tasks' requirements. By following the hints from the resource manager, tasks can collaborate implicitly without explicit knowledge of each other. However, our approach can not guarantee any time limits since these highly depend on the behavior of the involved tasks. Yet, even if used sparsely, it is always equal or better compared to non-cooperative operation.

Our test beds and the integration of all presented concepts into *Smart*OS showed, that the effective use of prioritized tasks for creating reactive systems is even possible on small embedded 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 sensor/actor networking but may also extend other embedded systems in general.

At present we are working on more sophisticated concepts for adjusting the acceptance of hints to the task and system situation. In particular, we want to improve the hint selection and the application of TUFs. Also, we plan to evaluate the use of dynamic hinting for remote resource management in distributed systems. Concerning real-world applications, we just integrated our approach into a WSAN based indoor localization and car control system where we achieved a considerably higher localization frequency and path precision. Another focus is the application of software model checking on systems using our new approach under *Smart*OS.

References

[1] Jacek Blazewicz, Klaus Ecker, Erwin Pesch, Günter Schmidt, and Jan Weglarz, *Handbook on Scheduling: Models and Methods for Advanced Planning (International Handbooks on Information Systems)*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.

[2] I. Lee, J. Y.-T. Leung, and S. H. Son, Eds., *Handbook of Real-Time and Embedded Systems*. CRC Press, 2007.

[3] Giorgio C. Buttazzo, "Real-Time Scheduling and Resource Management," in *Handbook of Real-Time and Embedded Systems*, I. Lee, J. Y.-T. Leung, and S. H. Son, Eds. CRC Press, 2007.

[4] N. Audsley, R. Gao, A. Patil, and P. Usher, "Efficient OS Resource Management for Distributed Embedded Real-Time Systems," in *Proceedings of Workshop on Operating Systems Platforms for Embedded Real-Time applications*, Dresden, Germany, Jul 2006.

[5] Peng Li, Binoy Ravindran, and E. Douglas Jensen, "Adaptive Time-Critical Resource Management Using Time/Utility Functions: Past, Present, and Future," *Computer Software and Applications Conference, Annual International*, vol. 2, pp. 12–13, 2004.

[6] U. Berkeley, "TinyOS," Web site http://www.tinyos.net/, 2004.

[7] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt, "Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors," in *LCN '04: 29th IEEE International Conference on Local Computer Networks*. IEEE Computer Society, 2004.

[8] O. Babaoğlu, K. Marzullo, and F. B. Schneider, "A formalization of priority inversion," *Real-Time Syst.*, vol. 5, no. 4, pp. 285–303, 1993.

[9] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Trans. Comput.*, vol. 39, no. 9, pp. 1175–1185, 1990.

[10] A. M. K. Cheng and J. Ras, "The implementation of the Priority Ceiling Protocol in Ada-2005," *Ada Lett.*, vol. XXVII, no. 1, pp. 24–39, 2007.

[11] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han, "MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms," *Mob. Netw. Appl.*, vol. 10, no. 4, 2005.

[12] Hojung Cha, Sukwon Choi, Inuk Jung, Hyoseung Kim, Hyojeong Shin, Jaehyun Yoo, and Chanmin Yoon, "RETOS: resilient, expandable, and threaded operating system for wireless sensor networks," in *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*. New York, NY, USA: ACM, 2007.

[13] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri, Eds., *Scheduling in Real-Time Systems*. Wiley, 2002.

[14] E. G. Coffman, M. Elphick, and A. Shoshani, "System Deadlocks," *ACM Comput. Surv.*, vol. 3, no. 2, pp. 67–78, 1971.

[15] M. Baunach, R. Kolla, and C. Mühlberger, "Introduction to a Small Modular Adept Real-Time Operating System," in *6. Fachgespräch Sensornetzwerke*. RWTH Aachen University, 16.–17. Jul. 2007.

[16] *MSP430x1xx Family User's Guide*, Texas Instruments Inc., 2006.

[17] J. Polastre, R. Szewczyk, and D. Culler, "Telos: Enabling Ultra-Low Power Wireless Research," in *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks*, 2005.