Priority aware Resource Management for Real-Time Operation in Wireless Sensor/Actor Networks

Marcel Baunach
Department of Computer Engineering, University of 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 outline our *dynamic hinting* approach for maintaining high 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 boosts 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, directly related issues comprise resource sharing or even real-time operation. Concerning this mixture, we find that current WSN research is still too restricted to static design concepts. As already stated in [1], 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. In fact, we also expect a clear focus shift from pure sensing in classic WSNs towards additional pro-activity in WSAN applications (e.g. integrated control systems, precise on demand measurements, etc.). Then, preemptive and prioritized tasks are required for reliable and fast response on various events.

We present the *dynamic hinting* approach for cooperative resource sharing and real-time operation within preemptive operating systems. As often suggested [2], we take advantage of the resource manager's enormous runtime knowledge about the system's current resource requirements. This information is carefully selected and forwarded to those tasks, which currently block 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.

II. Motivation and Requirements

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 allocate virtually any resource mix in any order or if long-term allocations collide with sporadic and time critical on demand allocations. During our research we found that reactivity and pro-activity in modern WSAN applications requires quite sophisticated real-time and smart adaptive resource concepts. We'll just give a short example from a real-world application:

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 data streaming. 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 one 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 or stateful libraries for managed operation of such resources is also no universal solution since this would just relocate the problem and cause additional overhead.

III. Related Work in WSN/WSAN Systems

Non-preemptive systems with run-to-completion tasks are very common in the WSN domain and prevent some resource conflicts implicitly since task executions can't be interleaved. Yet, this often causes bad reactivity to sporadic events. Therefore, e.g. TinyOS [3] and Contiki [4] support preemptive extensions but then lack priorities and resource management entirely. *Preemptive systems* potentially provide much better

reactivity since a task can be preempted for a more important action implemented in another task. Yet, preemption yields no instant advantage if the action requires a shared resource which is exclusively held by a less important task. Resulting problems like bounded or unbounded priority inversion [5] might lead to thwarting of high priority tasks and even deadlocks may occur. In any case, the task priorities defined by the developer are not obeyed as desired. To cope with some of these issues, well studied protocols like *priority ceiling*, *highest locker* or *priority inheritance* [6] are found in some embedded operating systems. We selected the priority inheritance protocol (PIP) as basic technique for our approach. Again, preemptive WSN operating systems like MANTIS [7] or RETOS [8] do not consider real-time or resource related problems at all.

IV. Resource Management and Dynamic Hinting

The central objective of our approach is to allow tasks $t \in T$ the collaborative sharing of exclusive resources. At the same time, it supports them to closely comply with their intended *base priorities* P_t. The basic idea behind *dynamic hinting* might be applied as integral concept for many real-time operating systems if these support three central features:

1) truly preemptive and prioritized tasks,
2) non-preemptive (i.e. exclusive) resources,
3) temporally limited resource requests (e.g. via deadline).

While the first two features can be found quite often, the last requires a special timing concept. Then however, tasks can request resources which are still held by other tasks. In this case, a requester is suspended until the resource is released (and handed over) or until the timeout occurs (and the request is denied). Though this allows to cope with allocation failures, it can also induce long resource request chains (Fig. 1, 2a). In case of infinite timeouts, even deadlocks may occur (Fig. 2b). These are already critical if two tasks mutually request a resource which is held by the other one, respectively.

Thus, 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 resource chains and simply suspend each requester task 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 l in the just averted cycle or extended chain. It also results in a violation of base priorities ($P_l < P_h$). 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, task-resource-dependencies are highly dynamic and depend on the system wide allocation order during runtime. Hence, another task 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 allocation chains are commonly reduced successively, beginning at their very end. This is exactly where *dynamic hinting* applies.

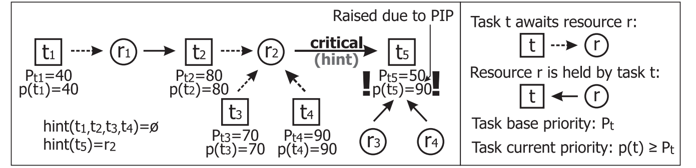

Figure 1. Example for Priority Inheritance and *Dynamic Hinting*

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 reduces direct, chained or deadlock blocking of at least one higher priority task (→Fig. 1, 2).
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 necessary possibility and priority (1) and the limited waiting of other tasks provides the time (2). PIP adjusts task priorities dynamically at runtime and according to the current resource assignment situation. It selects each task's l *current priority* $p(l) \geq P_l$ to be at least as high as the current priority $p(h)$ of the highest prioritized task h it currently blocks by virtue of a resource allocation.

Then, the first step for determining hints is to identify the *critical resources* for each task l. These currently define $p(l) \gneq P_l$ and thus, they directly or indirectly cause the blocking of more important tasks with base priority truly above P_l:

$$crit(t) := \{r \in R | r \text{ defines } p(t) \text{ by PIP}\} \quad (1)$$

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. 1, 2):

$$hint(t) := r \in crit(t), \; r \text{ was requested last.} \quad (2)$$

Then, if l releases its hinted resource r, it is directly passed to its first requester, w.l.o.g h. Next, $p(l)$ is updated by PIP and h is scheduled promptly. This is true since then h holds the highest priority of all tasks in ready state and l did let h pass by. As soon as l is scheduled again, it can immediately re-request r 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 1 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

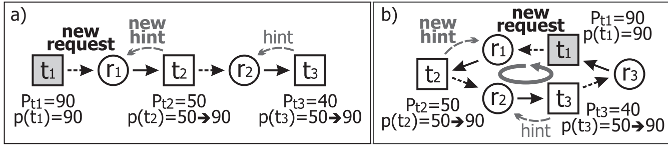

Figure 2. Dynamic Hinting Examples: a) Chain, b) Deadlock

more relevant ($P_{t_2} > P_{t_3} > P_{t_5}$) and will receive r_2 right after t_4.

Next, we'll describe two exemplary ways in which a task may receive and handle its hints: First, an *explicit query* can be done at distinct points in time or at code positions where its handling would be possible at all. Then however, a task can never react as long as it is in *waiting state*. Yet, this is exactly the case upon deadlocks and during many long-term allocations, where tasks e.g. wait for some events/interrupts while holding a resource. Beside this severe weakness, the manual effort and code pollution would be immense.

Thus, we recommend a much better strategy called *early wakeup*. When enabled, all functions by which a task suspends itself 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 threshold parameter φ:

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

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

$$(\varphi \neq 0 \ \wedge \ p(t) > P_t \ \wedge \ p(t) \geq \varphi), \quad (3)$$

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 Fig. 2a,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 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 this situation.

Of course, priority thresholds are not the only useful metric for a task to decide 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 [9]. These allow to relate the remaining allocation time to the still remaining timeout. Another option is to introduce a real-time priority threshold by initially defining φ equal for all tasks. This inherently limits the potential co-operativeness to situations where tasks (directly or indirectly) block any real-time task t_R with $P_{t_R} \geq \varphi$.

V. Real-World Applications and Test Bed

For analyzing our approach of combining temporally limited resource requests, the priority inheritance protocol and *dynamic hinting*, we extended the *Smart*OS [10] kernel as described since it is available for several sensor nodes, provides appropriate task, timing and resource basics, and thus allowed an easy integration. The implementation was done for Texas Instrument's MSP430 family of microprocessors, since these are found on a large variety of sensor nodes. Requiring $4\,\mathrm{kB}$ of ROM and $150\,\mathrm{B}$ 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 bed considers a quite frequent problem we also had in one of our real WSAN control applications: A task S continuously transfers some data over a shared bus b to an external device. The stream is rather long (or even infinite) but can be suspended and resumed at any time for more important communication over the same bus. Therefore, it always needs some bus setup plus a complex header/trailer for proper initiation/termination. During the transfer, S 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 interruption might be completely unnecessary. Furthermore, the selected packet length defines the duration of potentially resulting priority inversions. By using short (long) packets, the overhead increases (decreases) while improving (degrading) the reactivity of higher prioritized tasks when these request b. Commonly, a fixed length is 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.

Dynamic hinting provides two options for improvements. Since our approach knows about pending bus requests, S could query its current blocking state periodically and react only if necessary. Though the query interval must still be selected carefully, the overhead for useless stream interruption is already avoided! The additional use of *early wakeup* finally improves the reactivity as it hints S *instantly* and *only* if it blocks a task with truly higher base priority.

For the concrete application we had to stream $8\,\mathrm{bit}$ ADC data sampled at $10\,\mathrm{kHz}$ 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 had to process sporadic events (av. inter-arrival time: $\approx 5\,\mathrm{ms}$) and were much more time and safety critical since especially failures in the motor control were disastrous. So, we defined

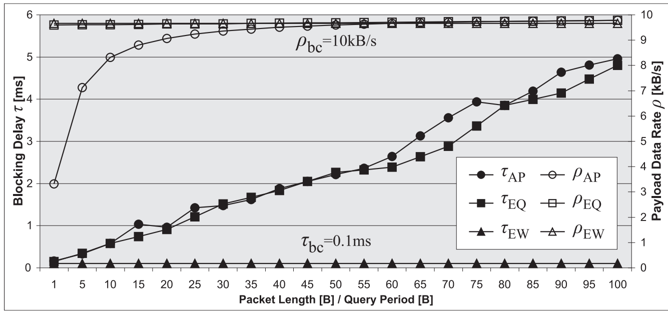

Figure 3. Streamtest: Packet Oriented vs. *Dynamic Hinting*

$P_S < \varphi = 100 < P_R < P_M$. To reduce CPU load we used a DMA channel between ADC and the bus controller. Thus, 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*:

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

Streaming data simultaneously to sporadic but highly reactive tasks might already cause extreme system load for low performance embedded systems like sensor nodes. Yet, the 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). Fig. 3 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 values are τ_{bc}=100 ns and 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. However, the blocking delay τ_{EQ} almost matches τ_{AP} and is also not satisfying for long periods (for short ones, the task causes higher CPU load). When using *early wakeup*, the data rate is still held high while 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 Fig. 3. Yet, *early wakeup* is independent from any packet length or query period.

VI. Conclusion and Outlook

In this paper, we outlined the *dynamic hinting* approach for cooperative resource sharing among preemptive tasks in reactive systems. In particular, the individual task base priorities are considered carefully to keep each task's performance close to its intended relevance. Therefore we analyze emerging task-resource dependencies at runtime and provide spurious tasks with information about how they could increase the reactivity of more relevant tasks or to recover from deadlocks. Thereby, tasks can collaborate even without explicit knowledge of each other. Nevertheless, each one can decide dynamically between cooperative or egoistic behavior with respect to its current conditions and other tasks' requirements.

While our approach is not necessarily limited to the WSAN domain, our implementation and test bed showed, that using compositional software design and prioritized tasks allows to create quite reactive systems even on small embedded devices.

At present we research 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.

References

[1] 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.

[2] 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.

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

[4] 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.

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

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.