CoMem: Collaborative Memory Management for Real-Time Operation within Reactive Sensor/Actor Networks

Marcel Baunach

To cite this version:

Marcel Baunach. CoMem: Collaborative Memory Management for Real-Time Operation within Reactive Sensor/Actor Networks. 18th International Conference on Real-Time and Network Systems, Nov 2010, Toulouse, France. pp.79-88. hal-00544512

HAL Id: hal-00544512
https://hal.science/hal-00544512

Submitted on 8 Dec 2010

CoMem: Collaborative Memory Management for Real-Time Operation within Reactive Sensor/Actor Networks

Marcel Baunach
Department of Computer Engineering, University of Würzburg, Germany
baunach@informatik.uni-wuerzburg.de

Abstract

Increasing complexity and modularity of today's WSAN applications impose demanding challenges on the system design. This especially affects real-time operation, resource sharing and dynamic memory management. Preemptive task systems are one way to retain good reactivity within dynamic environments. Yet, since memory is often too rare for static assignment, this rapidly leads to severe compositional problems among tasks with interfering and even varying requirements. We present our novel CoMem approach for high reactivity and efficient memory usage in such systems. With respect to task priorities and the typically limited resources of sensor nodes, we facilitate compositional software design by providing tasks with runtime information for yet collaborative and self-reflective memory sharing. Thereby, we require no special hardware-support like MMUs but operate entirely software-based.

1 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 [11] and fine grained code updates [7]) are more and more used to manage design and operation of these embedded systems. Then, adequate interaction between the modules is essential to handle typical compositional problems like task scheduling, resource sharing or even real-time operation [14]. In this respect, we find that current WSAN research is still too limited to static design concepts. As already stated in [9], 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 during development. Then, preemptive and prioritized tasks are required for fast response on various (sporadic and periodic) events but further complicate memory management and reactivity. This is especially true for open systems where real-time and non real-time tasks coexist in order to reduce hardware overhead, energy issues and deployment effort.

In this paper we present our novel CoMem approach for collaborative heap memory sharing and real-time operation within preemptive operating systems. It improves compositional software design by providing independently implemented tasks with information about their current influence on each other. In our opinion, the central weakness of all memory management approaches we found so far is, that tasks are not aware of their (varying) impact on the remaining system, and thus cannot collaborate adequately. In this respect, CoMem follows classic reflection concepts [1, 21], and introduces a new policy into software and operating systems, by which programs can become 'self-aware' and change their behavior according to their own current requirements and the system's demands. As often suggested [1], we take advantage of the resource and memory manager's enormous runtime knowledge about each task's current requirements. This information is carefully selected and forwarded to exactly those tasks, which currently block the execution of more relevant tasks. By creating a bidirectional communication link between memory manager and tasks, passing these so called *hints* allows blocking tasks to adapt to the current memory demands and finally to contribute to the system's overall progress, reactivity and stability. In this collaborative manner, CoMem also accounts for task priorities as defined by the developer. For hard real-time (RT) constraints, the allocation time can be bounded by using a special RT memory layout. Nevertheless, the decision between following or ignoring a hint is always made by each task autonomously and dynamically at runtime, e.g. by use of appropriate time-utility-functions [19]. Finally, CoMem is not limited to embedded systems and the WSAN domain, but can be applied to real-time operation in general.

This paper is organized as follows: Initially, we'll review some related techniques from existing work before details about our new approach will form the central part of this paper. An exemplary implementation of CoMem will show that – despite of the problem's complexity – it is efficiently applicable even for low performance devices like sensor nodes. Therefore, we will also present some application examples and the impact on the programming model before performance results from real-world test beds close this paper.

2 Related work

Dynamic memory management is subject to intense research efforts and plays an important role in current software design [17, 16, 15, 22]. Yet, most concepts limit their focus on developing an allocator, which assigns the available heap space in a way to reject as few requests as possible in spite of high dynamics and frequent (de)allocations. Unfortunately, heap methods suffer from some inherent flaws, stemming entirely from fragmentation. For multitasking systems in particular, there is a lack of scalability due to competition for shared heap space. Thus, a good allocator should support and balance a number of features for allocation of the memory blocks [16, 13]:

F1 *Minimize space* by not wasting it, i.e. allocate as little memory as possible while keeping fragmentation low.

F2 *Minimize time and overhead* by fast or even deterministic execution of related functions.

F3 *Maximize error detection* or even avoid tasks to corrupt data by illegal access to foreign blocks.

F4 *Maximize tuneability* to account for dynamic and task specific requirements like real-time operation.

F5 *Maximize portability and compatibility* by using few but widely supported hardware and software features.

F6 *Minimize anomalies* to support good average case performance when using default settings.

F7 *Maximize locality* by neighboring related blocks.

F8 *Avoid trivializing assumptions*, making progress and success easy by imposing unreasonable restrictions.

However, according to [25], any allocator can face situations where continuous free memory is short while the total amount of free space would be sufficient to serve an allocation request. Especially for systems without MMU or virtual address space, a centralized heap re-organization by the memory manager is hard or even impossible then, since it lacks information about critical dependencies and the actual memory usage by the current owner tasks.

Thus, the use of dynamic memory is largely avoided for time or safety critical systems [15]. For these, F1 and F2 must be extended to provide a spatial and temporal allocation guarantee, i.e. the knowledge about the allocators WCET. If not avoidable, real-time operating systems often support so called *pools* of fixed-size memory blocks (low external, high internal fragmentation) and constant allocator execution time – at least in case of success. In contrast, blocks of arbitrary size commonly provide more flexibility (less internal, more external fragmentation) at higher management effort, and might theoretically partition the usually small heap space more efficiently. Depending on the internal heap organization, four central techniques are commonly distinguished: Sequential fits, segregated free lists, buddy systems and bitmap fits. Since we focus primarily on real-time support and memory re-organization in case of allocation failures, we won't go into detail about these techniques, but refer to [15, 25] instead.

When considering WSAN operating systems, only few support dynamic memory for arbitrary use by application tasks: TinyOS 2.x [24] supports a semi-dynamic pool approach in which a fixed number of blocks can be statically assigned to a task. At runtime, tasks can release their blocks to the pool and reallocate blocks as long as the initial number is not exceeded. Contiki [7] offers dynamic memory for storing variables of dynamically loaded modules. In SOS [10] a block based first-fit scheme with 32×16, 16×32, 4×128 bytes is used to store module variables and messages. MantisOS [6] uses best-fit to allocate arbitrary size blocks for thread stacks and the networking subsystem only. In [17] a combination of sequential fits, segregated free lists and the buddy system is proposed for Nano-Qplus. SensorOS [12] supports a pool based approach for messages and a buddy system for blocks of any size. To find a suitable allocator for concrete WSN applications, SDMA [22] uses simulation for comparing several candidates by various metrics.

In general, static allocator selections will hardly be optimal and cannot be easily adapted in case of dynamic changes to the application code [7]. Beside SensorOS, no OS provides the arbitrary use of dynamic memory for tasks. In particular, none provides any mean for dynamic memory organization in case of time-critical sporadic requests or priority inversions, when low priority tasks block higher priority tasks by any memory allocation. To the best of our knowledge, no OS exists to support on-demand memory re-organization in small embedded systems without brute force methods like (energy intensive) swapping, memory revocation or task termination with possibly critical side effects. This is exactly where CoMem applies.

3 The CoMem approach

Reflection based task collaboration [1] is a mighty tool to share resources on-demand and "upwards" along with the task priorities [2]. We adapt the strengths and benefits for the special case of dynamic memory allocation. By addressing the specific problems and the feature requests from Section 2, we'll now present the central idea, design and implementation decisions behind our new concept.

3.1 Dynamic hints for on-demand resource sharing

Our CoMem approach is generally based on Dynamic Hinting [2], a technique for collaborative sharing of arbitrary resources among prioritized and preemptive tasks. As central idea, dynamic hinting analyzes emerging task/resource conflicts at runtime and provides spurious tasks with information about how they can help to improve the reactivity and progress of more relevant tasks. The combination with blocking based priority inheritance techniques – e.g. the basic Priority Inheritance Protocol (PIP) [20] – reliably improves and stabilizes the overall system performance. Therefore, the approach reduces priority inversions, resource allocation delays and even recovers from deadlocks where required.

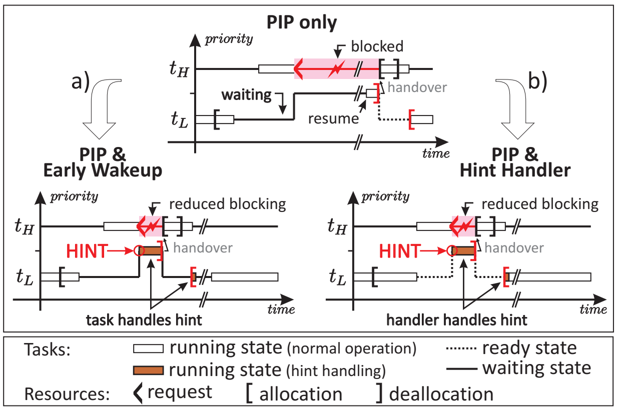


Figure 1. Hint handling when task t_L blocks while in a) waiting state or b) ready state

According to the PIP policy, a task t's *active priority* $p(t)$ is raised to $p(v)$ iff t blocks at least one other task v with truly higher *active priority* $p(v) > p(t)$ by means of at least one so called *critical resource*. Only then, dynamic hinting immediately passes a hint indicating this priority inversion to t and 'asks' for releasing at least one critical resource quickly. While this facilitates the on-demand release and handover of blocked resources, passing such hints is not trivial in preemptive systems, since from the blocker's view, this happens quasi-asynchronously and regardless of its current situation, task state or code position. Given that a blocking task can be in ready or even waiting state while a new blocking comes up, two techniques are relevant for our CoMem approach:

- Early Wakeup: When in *waiting* state (i.e. suspended by a blocking function), t will immediately be scheduled again and transit to running state. The resumed function will return an indicator value to signal this special situation. The impact on the programming model is similar to exception handling in various programming languages: A task 'tries' to e.g. sleep or wait for an event but 'catches' an early wakeup to react on its blocking influences (→Fig. 1a).
- Hint Handler: When in *ready* state (i.e. just preempted by another task) a task-specific hint handler is injected into t's execution. These handlers operate entirely transparent to the regular task. Similar to the CPU scheduler in preemptive kernels, hint handlers allow to operate literally non-pre-emptive resources in a quasi-preemptive way (→Fig. 1b).

In both cases, hints are passed instantly and only when blocking really occurs. Since dynamic hinting is a reflective approach, hint handling always follows the same procedure: Query the critical resource r_c and decide between following or ignoring the hint. When following:

1. Save r_c's state (if necessary) and stop its operation. Respect WCRTs when required, i.e. if real-time contracts must be obeyed as described in Section 3.3.
2. Release r_c. *This will immediately cause an implicit task self-preemption due to the resource handover.*
3. Re-allocate r_c upon resumption.
4. Restore r_c's state and restart its operation.

3.2 CoMem for dynamic memory allocation

Since memory is commonly a very scarce resource in small embedded systems, it needs to be shared among tasks to achieve a higher integration density for future, versatile systems and WSAN applications. This is already true, if some tasks run rather seldom and a static memory allocation would leave valuable space unused for long periods. Nevertheless, rarely running tasks might also be subject to tight timing constraints and request memory only upon certain events (e.g. triggered by environmental interactions, see feature F4 and Section 5).

In this context, the first problem is *priority inversion* concerning such an request. Commonly, this term is used upon blocking on ordinary resources. However, the heap memory will become partitioned and fragmented during system runtime and the number of (potentially disturbing) blocks is highly variable. In such cases an ordinary resource for managing mutually exclusive access is insufficient. Instead, so called *virtual resources* are used to internally split the complete (and otherwise monolithic) memory for use by several tasks. As an example, Figure 2a shows such a scenario: Tasks t_A, t_B, t_D hold memory blocks protected by the virtual resource r_H. Thus, t_C's request cannot be satisfied and we see a priority inversion.

Simply using e.g. PIP for raising $p(t_A), p(t_B)$, and potentially accelerate their deallocation, imposes some questions: Which one should be adapted? Raising just one blocking task might select the wrong one. Raising all blocking tasks means setting them to equal priorities $p(t_A) = p(t_B) = P_{t_C}$ and leads to round-robin or run-to-completion scheduling despite of intentionally different *base priorities* $P_{t_A} \leq P_{t_B} \leq P_{t_C}$. In fact, t_C could be served if either lower prioritized task t_A or t_B would release or just relocate its memory block. Yet, in common approaches, tasks do not know about their spurious influences and thus cannot react adequately. In turn, developers tend to retry until the allocation succeeds.

Using e.g. plain C-functionality within preemptive systems would result in spinning loops calling `malloc()`, and cause the unintentional (and maybe infinite) blocking of lower prioritized tasks. If the underlying operating system supports timing control for tasks, spinning might be relaxed by periodic polling for free memory. While this would still cause significant CPU load upon short periods, it can potentially miss sufficiently large free memory areas upon long periods. Anyway, the memory manager does not know that a task t actually still waits for memory between the polls, and can neither serve t nor reserve memory. If supported, another load intensive option are lock-free methods like [16]. To minimize the CPU load by currently not serviceable tasks, our approach uses a task-blocking `malloc()` function and transfers the memory

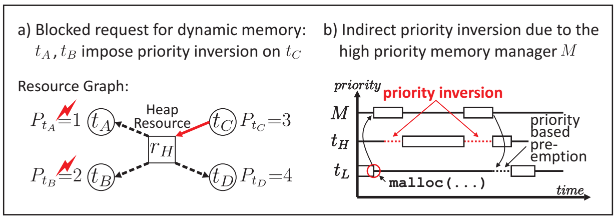

a) Blocked request for dynamic memory:
t_A, t_B impose priority inversion on t_C
Resource Graph:
$P_{t_A}=1$
t_A
Heap Resource
r_H
t_C
$P_{t_C}=3$
$P_{t_B}=2$
t_B
t_D
$P_{t_D}=4$
b) Indirect priority inversion due to the
high priority memory manager M
priority
priority inversion
M
t_H
t_L
priority based pre-emption
malloc(...)
time

Figure 2. Critical scenarios during dynamic memory management

organization to the memory manager subsystem M. In turn, we have to

a) find a suitable strategy for the heap (re)organization,
b) limit the blocking to a certain timeout τ (as often requested and useful within reactive systems),
c) decide whether this subsystem M itself is a task (server), a kernel function (syscall), or if it entirely operates within the context of each task (library).

Let's start with c. If the memory manager M has higher priority than ordinary application tasks, indirect priority inversion would still emerge from handling each request immediately and independently from the requester's priority. As Figure 2b shows, this would allow a low priority task t_L to implicitly slow down a higher priority task t_H by simply calling `malloc()`. To avoid this problem, it is at least wise to design the memory manager as server task t_M, and adapt its base priority P_{t_M} dynamically to the maximum active priority of all tasks it currently has to serve. To further reduce overhead in terms of task count, context switches, stack space, etc., we decided to execute the memory management functions entirely within the context of the calling tasks. In addition, this will implicitly treat the corresponding operations with adequate priority in relation to other tasks (→features F2, F4).

To allow temporally limited blocking, we extended our `malloc()` function by a timeout parameter τ (→Fig. 4). This way, we provide the memory management subsystem with information about how long we are willing to wait in worst case, and at the same time we supply a defined amount of time for re-organization of the heap space.

Finally, the heap (re)organization policy is indeed a critical core element within all memory managers and was already considered in many ways, e.g. [15, 25]. Beside task termination, two elementary options exist and are also supported by CoMem:

- release memory blocks (e.g. dismiss or swap data)
- relocate memory blocks (e.g. for compaction)

For both, we need to discuss which blocks to select and how to treat them adequately with respect to their current owner task. Within our concept, only these blocks are considered for re-organization which belong to lower prioritized tasks and would lead to sufficient continuous space for serving higher prioritized requesters. If sufficient, relocation is less damaging and takes precedence over release while the latter is at least as effective.

It is important to notice, that revoking or moving memory *without* signaling this to the owner task is complicated or even impossible in most cases. Not even data structures which are just accessed relative to the block base addresses (like stacks) can simply be relocated: expired addresses might still reside in registers or CPU stages, then. Much worse, affected peripherals like e.g. DMA controllers can often not be updated automatically and would still transfer data from/to old addresses. In such situations not even task termination and restart is a valid solution. Instead, this can only be handled by the owner task which has complete knowledge about the memory usage and *all* dependencies.

Thus, the central idea of CoMem is to inform those tasks which cause the denial of memory for higher prioritized tasks. Along with the hint, the requester's remaining timeout and active priority will also be passed to the blocking tasks and can be used within their time-utility-functions [19] (→feature F4). Furthermore, we advise the blockers whether releasing or relocating their memory blocks would solve this problem most suitably and thus account for the reactivity and progress of more relevant tasks. In fact, this triggers a self-controlled but on-demand heap re-organization by means of some helper functions like e.g. `relocate()` and `free()` from Fig. 4.

Before heading to the RT aspects, the technical details and the impact on the programming model, we'll summarize our design decisions while recalling feature F8:

1. Persisting allocations must not prevent further requests. Then, CoMem always knows about all system wide requirements and can generate adequate hints.
2. Extending `malloc()` by a timeout τ for limited waiting gives blocking tasks the time to react on a hint.
3. Executing the memory management functions directly within the callers' task contexts reduces overhead and implicitly reflects the task priorities.

Please note: As long as no MMU is available, our concept cannot *protect* memory against unauthorized access but only coordinate its exclusive sharing (→feature F3).

3.3 Hard real-time heap organization

Until now, the acceptance of hints is neither guaranteed nor bounded in execution. While we'll already obtain remarkable results without further efforts (→Section 5), this is only acceptable for non/soft real-time operation. Thus, we define a special *RT heap layout* and a *contract scheme* to assure timely memory allocations for real-time tasks with hard timeouts.

Given a set of time critical memory blocks $\hat{M}$ with a finite maximum allocation timeout $A(\hat{m}) > 0$ for each $\hat{m} \in \hat{M}$. The directed *RT Memory-Graph* $G_{\hat{M}} = (\hat{M}, \alpha)$ specifies the potential simultaneous allocation of blocks:

$$\alpha := \{(\hat{a}, \hat{b}) \mid A(\hat{a}) \leq A(\hat{b}) \ \wedge \ \text{simult. alloc. of } \hat{a}, \hat{b} \text{ possible}\}$$

As Figure 3 already shows, $G_{\hat{M}}$ is used to pre-layout the heap space so that any valid allocation of $\hat{M}$ is supported. Note that the memory is not assigned statically

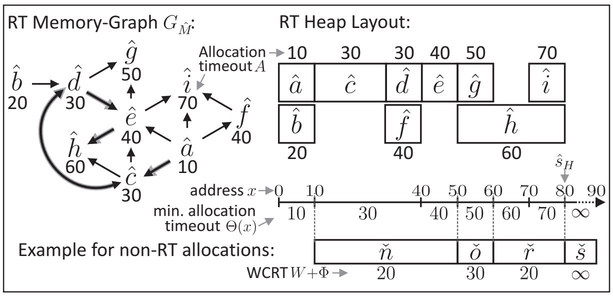

RT Memory-Graph $G_{\hat{M}}$:
RT Heap Layout:
$\hat{b}$ 20
$\hat{d}$ 30
$\hat{g}$ 50
$\hat{e}$ 40
$\hat{h}$ 60
$\hat{c}$ 30
$\hat{i}$ 70
$\hat{a}$ 10
$\hat{f}$ 40
Allocation timeout A
10 30 30 40 50 70
$\hat{a}$ $\hat{c}$ $\hat{d}$ $\hat{e}$ $\hat{g}$ $\hat{i}$
$\hat{b}$ $\hat{f}$ $\hat{h}$
20 40 60
$\hat{s}_H$
address x
0 10 40 50 60 70 80 90
min. allocation timeout $\Theta(x)$
10 30 40 50 60 70 ∞
Example for non-RT allocations:
$\check{n}$ $\check{o}$ $\check{r}$ $\check{s}$
WCRT $W+\Phi$
20 30 20 ∞

Figure 3. Heap organization for RT memory

but only reserved while being shared with non real-time tasks at runtime. In addition, for each heap address x

$$\Theta(x) := \begin{cases} \min\{A(\hat{m})\} & x \text{ is reserved for } \hat{m} \in \hat{M} \\ \infty & x \text{ is not reserved for } \hat{M} \end{cases} \qquad (1)$$

is the minimal allocation timeout for all RT blocks spanning over x, or ∞ if x is not reserved.

As requested by feature F1, our goal is to efficiently share the heap space between the RT blocks $\hat{M}$ and non-RT blocks $\check{M}$ with $\hat{M} \cap \check{M} = \emptyset$. To avoid colliding requests for RT blocks but still obtain large continuous areas of free memory for non real-time tasks, the *RT Heap-Layout* is organized at startup or compile-time as follows:

C1 No two different blocks $\hat{m}_1, \hat{m}_2 \in \hat{M}$ with $(\hat{m}_1, \hat{m}_2) \in \alpha$ may span over a common heap address x.

C2 Reservations for the RT blocks $\hat{M}$ must be partially ordered by $A(\hat{m})$ as follows: $\forall_{x \leq y} : \Theta(x) \leq \Theta(y)$.

Both is easily achieved in $O(|\hat{M}| \log |\hat{M}|)$ by sorting $\hat{M}$ by ascending $A(\hat{m})$ and placing these blocks successively at the lowest address permitted by C1 and C2. While C1 already solves the memory allocation for the RT blocks $\hat{M}$, we'll show how C2 simplifies the assignment of blocks $\check{M}$ to non-RT tasks.

While $\hat{M}$ must obviously be known during RT layout generation[1], non-RT blocks $\check{M}$ need not to be known then. However, to allow the emergence of hints toward blocking non-RT tasks at runtime, we assume a strict priority based separation between RT and non-RT tasks[1]:

$$\forall_{\check{m} \in \check{M}, \hat{m} \in \hat{M}} : P_{\sigma(\check{m})} < P_{\sigma(\hat{m})}$$

Additionally, upon each allocation attempt, the owner $\sigma(\check{m})$ must provide an individual contract offer by specifying the WCRT $W(\check{m}) > 0$ of its hint handling routine for clearing $\check{m}$ (by either *free* or *relocate*). If the WCRT is unknown, ∞ must be specified. The offer is negotiated by the memory manager, and allows an appropriate placement of the non-RT blocks:

When considering some temporal overhead Φ for the memory management itself, the lowest possible base address $x_{\min}$ for any $\check{m} \in \check{M}$ is

$$x_{\min}(\check{m}) := \min\{x \mid \Theta(x) \geq W(\check{m}) + \Phi\}. \qquad (2)$$

Since several non-RT blocks may share a RT block's range (e.g. $\check{o}, \check{r}, \hat{h}$ in Fig. 3), we have to be careful with overlappings. Thus, the base address selection is done as follows:

C3 Place $\check{m}$ at an address $x \geq x_{\min}(\check{m})$ so that the space for any real-time block $\hat{m}$ can be freed within $A(\hat{m})$:

$$\forall_{\hat{m} \in \hat{M}} : \sum_{\substack{m \in \check{M} \\ m \text{ overlaps } \hat{m}}} W(m) + \Phi \leq A(\hat{m}) \qquad (3)$$

This way, all disturbing non-RT blocks can be removed for the guaranteed timely success of any RT task's request.

Furthermore, we largely avoid placing any $\check{m}$ over boundaries of real-time blocks to reduce the chance for collisions and frequent hint handling. Profiling allocation frequencies and durations might be used for further optimization but is omitted here. If there is currently no place for $\check{m}$, the memory manager tries to hint any other allocated $\check{m}' \in \check{M}$ with sufficient $W(\check{m}') + \Phi \leq A(\check{m})$ and lower owner priority. On success, $\check{m}$ is allocated while C3 must still be followed. If an allocation is not possible within $A(\check{m})$, the request is rejected (timeout).

To ensure the real-time feasibility of our approach, the heap space s_H must finally be sufficient for the RT blocks under C1 and C2. Therefore, we use the block sizes $|\hat{m}|$ as node weights in $G_{\hat{M}}$ and select the longest acyclic path $P_{\hat{M}}$ to compute a lower bound $\hat{s}_H$ for the heap size s_H:

$$\hat{s}_H := |P_{\hat{M}}| = \sum_{\hat{m} \in P} |\hat{m}| \qquad (4)$$

In addition, some extra space $\check{s}_H$ must be estimated and reserved for those non-RT blocks which cannot be entirely co-located with RT blocks due to C3. For each block $\check{m} \in \check{M}$ of known size $|\check{m}|$, the still unallocatable fraction is

$$u(\check{m}) := \max\{0, |\check{m}| - (\hat{s}_H - x_{\min}(\check{m}))\}, \text{ and}$$

$$\check{s}_H := \max\{u(\check{m}) \mid \check{m} \in \check{M}\} \qquad (5)$$

is the lower bound for the extra space. Finally,

$$s_H \geq \hat{s}_H + \check{s}_H \qquad (6)$$

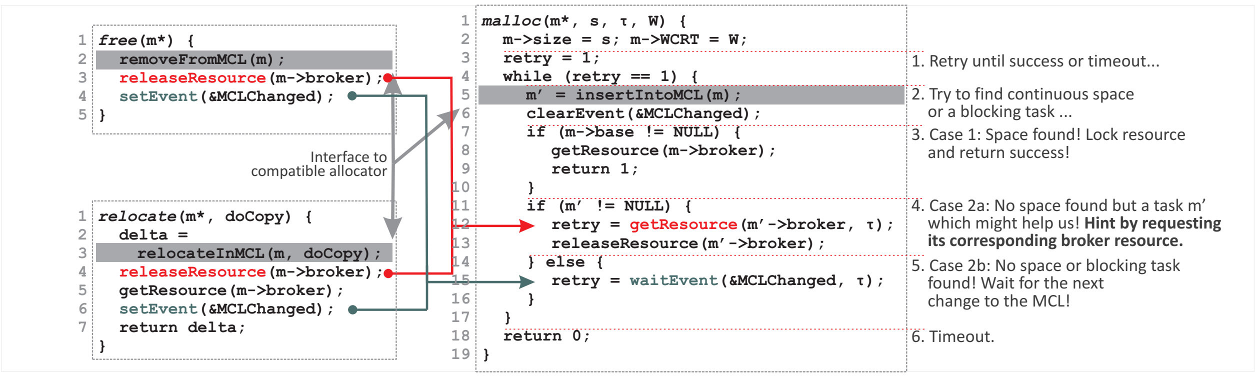
must be chosen as heap size to be sufficient in terms of (timely) allocations for all (RT) blocks. Still, allocations can never be granted at all for non-RT blocks of initially unknown size.

For Figure 3, $P_{\hat{M}} = \hat{a}, \hat{c}, \hat{d}, \hat{e}, \hat{h}$. Thus $s_H \geq \hat{s}_H + \check{s}_H = |P_{\hat{M}}| + u(\check{s}) = 80 + 10 = 90$ must be selected.

4 CoMem implementation and usage

This section presents the implementation details about our novel memory management approach. The basic idea behind CoMem 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

[1] which is commonly the case for hard real-time operation, anyway



```
free(m*) {
  removeFromMCL(m);
  releaseResource(m->broker);
  setEvent(&MCLChanged);
}
```

```
relocate(m*, doCopy) {
  delta =
    relocateInMCL(m, doCopy);
  releaseResource(m->broker);
  getResource(m->broker);
  setEvent(&MCLChanged);
  return delta;
}
```

```
 1 malloc(m*, s, τ, W) {
 2   m->size = s; m->WCRT = W;
 3   retry = 1;
 4   while (retry == 1) {
 5     m′ = insertIntoMCL(m);
 6     clearEvent(&MCLChanged);
 7     if (m->base != NULL) {
 8       getResource(m->broker);
 9       return 1;
10     }
11     if (m′ != NULL) {
12       retry = getResource(m′->broker, τ);
13       releaseResource(m′->broker);
14     } else {
15       retry = waitEvent(&MCLChanged, τ);
16     }
17   }
18   return 0;
19 }
```

1. Retry until success or timeout...
2. Try to find continuous space or a blocking task ...
3. Case 1: Space found! Lock resource and return success!
4. Case 2a: No space found but a task m’ which might help us! **Hint by requesting its corresponding broker resource.**
5. Case 2b: No space or blocking task found! Wait for the next change to the MCL!
6. Timeout.

Figure 4. CoMem function interactions

*Smart*OS [4] since it fulfills these requirements. As requested by feature F5, it offers quite common characteristics, and thus is a good representative for the adaptation of similar systems. Beyond, it is also available for several MCU architectures like MSP430, AVR and SuperH.

4.1 *Smart*OS overview

The *Smart*OS kernel maintains a local system time and 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. Each task t has its individual and dynamic *base priority* P_t and an *active priority* $p(t)$ when using PIP for resource sharing. In general, each task may wait for at most one event or resource at the same time but it may hold several resources simultaneously. Allocation and deallocation orders are always arbitrary and independent. Apart from the CPU, resources are always treated as non-preemptive and will never be withdrawn. Once assigned, each owner task is responsible for releasing its resources. For on-demand resource handover, dynamic hinting was integrated as presented in [2] and Section 3.1.

4.2 CoMem implementation details

Next, we’ll show how to achieve our design considerations from Section 3. Regarding the tight performance and memory constraints of many embedded systems, CoMem is limited to three central functions and one Memory Control Block (MCB) for each dynamic memory block:

```
typedef struct {
  unsigned int size;     //1W: size in machine words
  volatile int *base;    //1W: block start address
                         //    (fixed for RT blocks)
  Resource_t   broker;   //2W: associated resource
  Time_t       WCRT;     //2W: -1 for unknown or ∞
  advice_t     advice;   //1W: what to do upon a hint
  MCB_t        *next;    //1W: linked list pointer
} MCB_t;                 // Total RAM size: 8W
```

Since we want tasks be be informed *immediately* if they block a higher priority task due to a dynamic memory allocation, we can simply use the dynamic hinting concept for this. Indeed, we associate one *Smart*OS resource – a so called *broker resource* – with each allocated memory block and implicitly obtain two important advantages:

1. We adapt the underlying resource management policy (e.g. PIP) for the memory management: All system resources and memory blocks are treated in the same way and respect the task priorities equally.
2. CoMem can be implemented as library and does not produce additional overhead within the kernel.
3. We create a communication link from the memory manager back to the owners of allocated blocks.

In general, CoMem is a two layer approach consisting of collaboration and allocation. Figure 4 shows the central code of the collaboration concept. Since it calls an interface compatible allocator internally, even variable allocation strategies and real-time metrics can be used.

In contrast to many other approaches which maintain a list of free memory areas [6], our allocator uses a linked list of MCBs for currently allocated blocks. Internally, this Memory Control List (MCL) is sorted by base addresses and thus allows linear scanning for continuous free areas of sufficient size for new requests. Though other data structures might scale better for many simultaneous allocations, a simple list’s low complexity is in line with the typically weak sensor nodes and still provided good performance within our testbeds. In fact, allocated blocks must be scanned anyway to select one for re-organization in case of insufficient free space. Complexity: $O(n)$.
For non-RT blocks, `malloc()` requires four parameters:

- An MCB m for managing the block. Since MCBs are supplied by the tasks as required, the CoMem library needs not to reserve a fixed number in advance.
- The block size s, the WCRT W for hint handling, and a timeout τ for limited waiting in case of currently insufficient continuous free space.

Internally, **`malloc(...)`** loops until the request succeeds or the timeout is reached (Line L4): Initially each retry attempts to insert the new block into the MCL (first-fit, L5) while always considering C3. On success (L7), the corresponding broker-resource b_m is locked by the caller and we are done. Since b_m belongs to the block owner $\sigma(m)$ then, it is sufficient for another task with higher priority to request this very resource if it is blocked by $\sigma(m)$. Indeed, this is exactly what happens if sufficient space is not available but a disturbing memory block m' was found (L11). By the resource request (L12), PIP adapts the active priority $p(\sigma(m'))$ of the blocking owner $\sigma(m')$. If dynamic hinting is enabled, the resource manager immediately passes a hint to $\sigma(m')$ to indicate its disturbing influence. If $\sigma(m')$ reacts by releasing/relocating its block m' before the timeout τ has expired, it also releases m'_b temporarily (`free()`:L3, `relocate()`:L4) to indicate the changed memory situation and to trigger a new retry for m. If no spurious task/block was found (L14), `malloc()` waits for the next modification to the heap space. Again, one more retry is triggered if there is still some time left. If the timeout has expired, `malloc()` stops and returns 0 to indicate the failure (L18).

The remaining problem is how to *reasonably* select a blocking MCB m' for generating a hint on. While scanning the MCL for free space (L5), we search for two types of MCBs: The first would at least produce the requested space if it was relocated and the other one if it would be released entirely. In consequence, m'_{advice} will be set to either $relocate$ or $release$ while the first takes precedence and the corresponding block with the lowest priority owner is selected for hinting. Thus, along with the hint, its owner also receives the advice for a suitable reaction. When considering the blocked task, a release is always at least as effective as a relocation.

For any RT block $\hat{m}$, its size s, base address and allocation timeout $\tau = A(\hat{m})$ is fixed within the RT heap layout. Then, the reserved dedicated space is also cleared by dynamic hints and finally assigned to the caller. If all WCRTs of the affected non-RT tasks are held, the hard timeout τ is also safe.

Finally, `free()` and `relocate()` are rather simple: **`free(m)`** simply removes the specified MCB m from the MCL and releases the broker resource b_m. Finally, it triggers the corresponding event to indicate the MCL update. **`relocate(m)`** seeks a new location for the supplied block m (cyclic next-fit) by which more continuous free space becomes available (L3). If requested, it also moves the data. Finally, it temporarily releases its own broker-resource b_m (L4/5) and triggers an event (L6) to resume waiting tasks. For subsequent address updates by the caller, the data shift is returned in bytes.

5 Real-world applications and test beds

To analyze our collaborative heap management concept of combining temporally limited memory requests, on demand heap re-organization and the priority inheritance protocol, we implemented CoMem as described. We used *Smart*OS for the Texas Instrument's MSP430 [23] family of microprocessors, since these are found on a large variety of sensor nodes. Requiring $4 + 1$ kB of ROM and $40 + 16$ B of RAM for the whole OS kernel and the CoMem library, the typically small memory of sensor nodes was considered carefully to leave sufficient space for the actual application. Our test scenarios were executed on SNoW[5] sensor nodes [3] with an MSP430F1611 MCU (10 kB RAM, 48 kB ROM) running at 8 MHz. For detailed performance analysis at runtime, we used the integrated *Smart*OS timeline with a resolution of 1 µs.

5.1 Dynamic memory stresstest

The first scenario analyzes our approach under extreme conditions with n tasks $t_0, \ldots, t_{n-1}$ and many concurrent memory requests. For this test we omitted dedicated RT blocks to study the performance under arbitrary allocations by tasks without detailed timing specifications (see feature F6). Instead, we simply assigned ascending base priorities $P_{t_i} = i$ and each task executed the same code repeatedly: (1) sleep, (2) request dynamic memory, (3) operate on the memory, (4) release the memory.

The duration Δ_s of step (1), the operation time Δ_c for step (3) and the size of the requested memory blocks were randomized for each iteration. This way, we obtained significant heap space fragmentation and task blocking which needed handling at runtime. Though we specified infinite timeouts τ and WCRTs W for allocation, each task measured the execution time δ of `malloc()` and logged its minimum, maximum and average allocation delays $\delta_{min}, \delta_{max}, \delta_{av}$. Furthermore, it registered the number of received hints. For comparing the allocation delays in relation to the task priorities, we applied two non-collaborative and two collaborative policies P1-P4:

P1 Classic: We omitted the request for a blocking task's broker resource during `malloc()` (L12). Instead we always waited for heap modifications (L15) if no continuous space was found. This avoided PIP, hints and the chance for collaborative memory sharing entirely and is comparable to many common approaches.

P2 PIP only: We implemented `malloc()` as shown in Fig. 4 but simply ignored the emerging hints. Though a blocking task did not collaborate explicitly then, its active priority was at least raised to the priority of the task it blocked and it received CPU time for step (3) more quickly.

P3 Hint Handlers: This time, each task supplied a hint handler for immediate injection into its own execution flow when blocking a higher prioritized task. This simulates blocking while in ready/preempted state.

P4 Early Wakeup: Finally, the tasks did sleep while holding a memory block. Yet, they were resumed immediately when blocking a higher prioritized task. This simulates blocking while in waiting/suspended state.

For collaboration under P3 and P4, a task t_L treated its hints as follows: First, t_L stopped the operation on its memory block. Depending on the advice from the CoMem subsystem, t_L either called `free()` or `relocate()`. As intended, this caused the immediate allocation success and the scheduling of a directly blocked task t_H with higher priority. This is always true since t_H then held the highest priority of all tasks in ready state and t_L did let t_H 'pass by'. When scheduled again, t_L

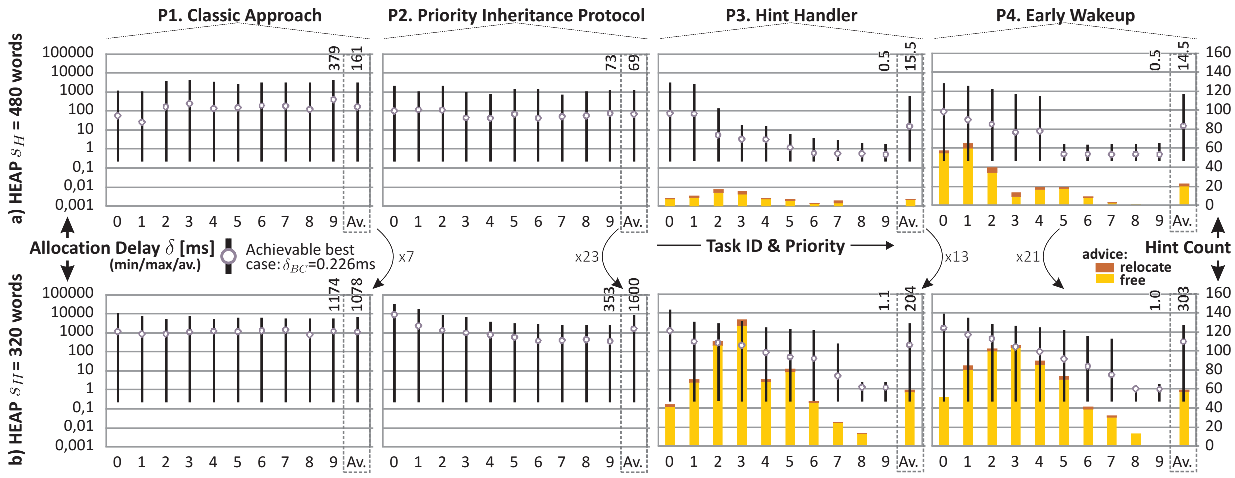


Figure 5. CoMem stresstest results for different heap sizes (asc. base priorities for $n = 10$ tasks)

tried to continue or restart its operation quickly. In case of $relocate$ it reused the shifted block. In case of $free$ it re-requested a new block of the old size. Please note that the data continuity within the memory blocks was not considered by this test (see Section 5.2 instead). Just the allocation delay was analyzed for reactivity and response time evaluation.

We configured the test bed using several task counts n, timings Δ_s and Δ_c, heap sizes s_H, and randomized block sizes s_B under the policies described above. Since the results always showed similar main characteristics, we just present the analysis for $n = 10$ tasks, block sizes $s_B \in \{32, 64\}$ words, heap sizes $s_H \in \{320, 480, 640\}$ words, $\Delta_s \in [0.2, 1.0]$s, and $\Delta_c \in [1.0, 5.0]$s. Each setup was executed for 10 min.

As expected, all allocations succeeded immediately when sufficient heap space $s_H = 640$ words was available to serve all requests even in the worst case. Though static memory assignments would suit much better then, we did this cross-check to see if the influence on the CPU load is already observable: Indeed, while the hint count remained 0, the average allocation delay already settled around δ_{av}=280 µs for each task and policy. In comparison, the best case execution time of `malloc()` (only one task and immediate success without preemption) was δ_{BC}=226 µs.

Selecting $s_H := 10 \cdot \frac{32+64}{2} = 480$ words (the required heap size for the average case) already shows the benefits of our collaborative approach (→Fig 5a). While the non-collaborative policies deliver almost uniform average allocation delays around 161 ms (P1) and 69 ms (P2), both do not reflect the tasks' intended base priorities at all. In contrast, using hints manages to reliably signal tasks about their spurious influence and allows them to react adequately. Considering the average *and* maximal allocation delays, the task priorities are visibly reflected by both collaborative policies P3 and P4. By following their hints, low priority tasks obviously allow higher priority tasks to achieve short allocation delays. Compared to P1 and P2, not even t_0 suffers from significantly increased delays, while several high priority tasks are very close to the achievable best case of δ_{BC}=226 µs, now. In average, δ_{av} roughly improved by factors 11 and 5, respectively.

Reducing $s_H := 10 \cdot 32 = 320$ words increases competition and allocation delays to be even more demanding (→Fig 5b). Still, the different task priorities are not visible for P1. In this regard, the sole PIP showed slight improvements for P2 under this heavy load. However, their average allocation delay increased by factor 7 and even 23, respectively. Since blocking occurs more often now, the hint count also increases significantly for the collaborative policies. Yet, these still manage to serve tasks according to their intended relevance: the two most important ones still achieve an average delay of $\delta_{av} \approx 1$ ms while even the lowest prioritized ones are still at least as reactive as with the non-collaborative approaches. Again, similar results are also visible for δ_{max}, which is quite notable.

This testbed addressed allocation delays for dynamic memory in case of sporadic requests and varying task priorities. We pointed out that dynamic dependencies (via broker resources) between blocking and blocked tasks can already reduce these delays in general and account for the specific task priorities in particular. While PIP already showed rudimentary success for heavy load situations, hints boosted this effect significantly. They even allowed almost best case delays for high priority (and maybe real-time) tasks though special RT specifications were omitted.

5.2 Real-world example under real-time conditions

The second testbed considers a problem from one of our real-world projects. The infrastructure of our ultrasound based indoor vehicle tracking system SNoW Bat [5] comprises several static anchors as references for the applied localization algorithms. These anchors run six preemptive tasks for several software modules (radio communication, sensor reading, etc). Two tasks are exceptionally memory intensive: t_{US} performs the ultrasound chirp detection, recording and processing for e.g. time-of-flight calculation. Each time it uses a capture compare unit to trigger an ADC/DMA combination which in turn samples

heap memory	t_{US} mode	t_{US} hint handling	$\delta_{max}(t_{RC})$
free	-	-	226 µs
alloc. by t_{US}	idle	just free the memory	1301 µs
alloc. by t_{US}	sampling	stop DMA/ADC & free	1351 µs
alloc. by t_{US}	DSP	abort DSP & free	1342 µs
alloc. by t_{US}	sampling/DSP	free after measurement	141284 µs

Table 1. Memory allocation delays within SNoW Bat

the chirp signal into a buffer of 4 kB. Then, a DSP algorithm operates on the sampled data.

In parallel, each node runs a task t_{RC} for remote node management and software updates. Compared to other parts of the system, this service is rarely used. But as soon as a new firmware image (max. 48 kB for an MSP430) is announced via radio, t_{RC} requests $n \cdot 256$ B of RAM and successively fills this buffer with image fragments received by radio. Then, the buffer is transferred to an external flash memory (block size: 256 B). This is repeated until the entire image was received. For optimizing the data rate and energy consumption, n should be as large as possible. This reduces frequent switching of the SPI-communication between MCU and radio or flash as well as the spacing delay between successive radio packets. Further, the external flash consumes less time and energy when accessed less frequently but for longer burst writes. In fact we use $n = 20$ and thus require 5 kB for the buffer.

From the 10 kB of the controller's RAM, the kernel and tasks require about 4 kB of static memory. The remaining 6 kB can be used as heap space. Thus, the chirp sampling buffer $\check{m}_{US}$ (4 kB) and the image data buffer $\hat{m}_{RC}$ (5 kB) must dynamically share their memory. In fact, aborting or even missing a chirp detection is not that critical: The node will be available for later measurements and other nodes are still available, too. Yet, missing an image fragment is highly critical indeed! Though some safety strategies are applied, an incomplete reception causes expensive retransmissions and write accesses to the external flash. Thus, t_{RC} imposes a hard upper bound $A(\hat{m}_{RC})$ for its memory allocation delay.

This real-time demand can easily be solved with our CoMem approach: Since t_{US} requires its buffer quite frequently (up to 3 Hz) but tries to limit the overhead for frequent re-initialization, it allocates the memory at system start and configures the DSP process and DMA controller according to the assigned base address. Since t_{RC} is more time-critical, it receives a higher base priority $P_{t_{RC}} > P_{t_{US}}$. As soon as t_{RC} requests dynamic memory, CoMem immediately passes a hint to t_{US}. If the sampling buffer $\check{m}_{US}$ is currently not in use, it is simply released to serve t_{RC} quickly. Otherwise, t_{US} initiates an untimely but controlled abortion of the current measurement. In particular, this includes adequate handling of active ADC and DMA operations. Since CoMem implicitly applies PIP, it raises $p(t_{US}) := P_{t_{RC}}$ while t_{RC} blocks on its allocation request. After memory deallocation, PIP will reduce $p(t_{US}) := P_{t_{US}} < P_{t_{RC}}$ again and t_{RC} is served and scheduled promptly. In turn, t_{US} will re-request its sampling memory as soon as possible for further measurements – and will receive it when t_{RC} has completed the image reception.

Table 1 shows the results for t_{RC}'s worst case allocation delays. If t_{US} would only release its memory after each complete measurement, t_{RC} would be blocked for $\delta_{max} \approx 141.3$ ms in worst case. Using hints from our CoMem approach allows an almost immediate memory handover which is just limited by some overhead Φ and the required time for aborting any currently running operation. Static analysis of the handler code revealed $W(\check{m}_{US}) \approx 1.3$ ms. Thus, we declared $\hat{m}_{RC}$ as RT memory block as described in Section 3.3. According to eq. (3), $A(\hat{m}_{RC}) \geq W(\check{m}_{US}) + \Phi = 1.3ms + 0.226ms$ also bounds the minimal tolerable delay Δ between image announcement and the first fragment. Finally, we selected $\Delta = 3ms$ and the hard timeout $\tau = 2ms = A(\hat{m}_{RC}) \geq 1.526$ ms. Indeed, we observed $\delta_{max} \leq 1.351$ ms during our tests and no timeout violation was detected. According to eq. (6), $s_H = |\hat{m}_{RC}| + 0 = 5$ kB was sufficient.

This test bed showed, that CoMem allows tasks to coordinate sporadic and time-critical memory requirements without explicit communication. In fact, a blocking task not even needs to know which task it blocks. Our approach provides sufficient information (via hints) and adequate task priorities (via PIP) to allow tasks an reflective resolution of their blocking influence. Beside the advantage of time aware on-demand memory handover in sporadic real-time systems, termination and reconfiguration of dependent resources (e.g. ADC and DMA) or subsystems (e.g. DSP) is limited to a minimum.

6 Conclusion and outlook

In this paper, we introduced our novel CoMem approach for collaborative memory sharing among preemptive tasks in reactive systems. We showed, that CoMem can help to improve and stabilize the overall system performance by optimizing memory allocation delays. Apart from F3 (protection) and F7 (locality) it also considers most feature requests from Section 2. In particular, individual task base priorities are considered carefully to keep each task's progress and reactivity close to its intended relevance. By analyzing emerging task/memory conflicts at runtime, we provide spurious tasks with information about how to reliably reduce the blocking of more relevant tasks. Following these hints allows tasks to collaborate implicitly without explicit knowledge of each other. This even reduces priority inversions and achieves memory allocation delays which are mainly limited by the pure handover overhead.

As a reflective concept, CoMem allows individual tasks to decide dynamically between collaborative or egoistic behavior with respect to their current conditions and other tasks' requirements. Thus, we initially can not guarantee any time allocation limits since these highly depend on the behavior of the blocking tasks. To still support hard allocation timeouts for RT tasks even if these share the heap with non-RT tasks in open systems, we introduced a special RT heap layout based on static timing specifications.

The test beds and the integration of our novel concept into the real-time operating system *Smart*OS showed, that the effective use of prioritized tasks for creating reactive open systems is even feasible on small embedded devices like sensor nodes: High priority tasks almost achieved the theoretical best case reactivity while low priority tasks did hardly lose performance. Even if used sparsely, our approach always proved to be better compared to non-collaborative operation. Though a well-thought application design still remains elementary, compositional software is already facilitated. In general, our approach is not necessarily limited to sensor/actor networking but may also extend other embedded systems.

At present we are working on improved hint generation and the application of TUFs by considering more application specific factors like remaining timeouts and allocation frequencies. Therefore we also research various allocator strategies to keep the chance for allocation failures low. Another area is the evaluation of CoMem for shared memory in multi-core systems [18, 8] where blocking may induce hints between the subsystems.

References

[1] 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.

[2] M. Baunach. Dynamic hinting: Real-time resource management in wireless sensor/actor networks. In *RTCSA '09: Proceedings of the 2009 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 31–40. IEEE Computer Society, 2009.

[3] M. Baunach, R. Kolla, and C. Mühlberger. SNoW^5: A versatile ultra low power modular node for wireless ad hoc sensor networking. In P. J. Marrón, editor, *5. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, Stuttgart, 17.–18. July 2006. Institut für Parallele und Verteilte Systeme.

[4] 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. July 2007.

[5] M. Baunach, R. Kolla, and C. Muehlberger. SNoW Bat: A high precise WSN based location system. Technical Report 424, Univ. of Wuerzburg, may 2007.

[6] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: an embedded multithreaded operating system for wireless micro sensor platforms. In *Mob. Netw. Appl.*, volume 10, pages 563–579, Hingham, MA, USA, 2005. Kluwer Academic Publishers.

[7] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *LCN 2004: 29th IEEE Int'l Conference on Local Computer Networks*. IEEE Computer Society, 2004.

[8] A. Easwaran and B. Andersson. Resource sharing in global fixed-priority preemptive multiprocessor scheduling. In T. P. Baker, editor, *IEEE Real-Time Systems Symposium*, pages 377–386. IEEE Computer Society, 2009.

[9] Giorgio C. Buttazzo. Real-Time Scheduling and Resource Management. In Lee et al. [14].

[10] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM.

[11] R. S. Kavi Kumar Khedo. A Service-Oriented Component-Based Middleware Architecture For Wireless Sensor Networks. *Int'l Journal of Computer Science and Network Security*, 9(3):174–182, Mar. 2009.

[12] M. Kuorilehto, T. Alho, M. Hännikäinen, and T. D. Hämäläinen. Sensoros: A new operating system for time critical WSN applications. In *SAMOS Workshop on Systems, Architectures, Modeling, and Simulation*. Springer, 2007.

[13] D. Lea. A memory allocator. Web http://g.oswego.edu/dl/html/malloc.html, 2010.

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

[15] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo. A constant-time dynamic storage allocator for real-time systems. *Real-Time Syst.*, 40(2):149–179, 2008.

[16] M. M. Michael. Scalable lock-free dynamic memory allocation. In *PLDI '04: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 35–46. ACM, 2004.

[17] H. Min, S. Yi, Y. Cho, and J. Hong. An efficient dynamic memory allocator for sensor operating systems. In *SAC '07: Proceedings of the 2007 ACM symposium on Applied computing*, pages 1159–1164, New York, NY, USA, 2007. ACM.

[18] S. Ohara, M. Suzuki, S. Saruwatari, and H. Morikawa. A prototype of a multi-core wireless sensor node for reducing power consumption. In *SAINT '08: International Symposium on Applications and the Internet*, pages 369–372, Washington, DC, USA, 2008. IEEE Computer Society.

[19] 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*, 2:12–13, 2004.

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

[21] J. A. Stankovic and K. Ramamritham. A reflective architecture for real-time operating systems. In *Advances in real-time systems*, pages 23–38. Prentice-Hall, Inc., 1995.

[22] G. Teng, K. Zheng, and W. Dong. SDMA: A simulation-driven dynamic memory allocator for wireless sensor networks. *Int'l Conference on Sensor Technologies and Applications*, 0, 2008.

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

[24] UC Berkeley. TinyOS. Web www.tinyos.net, 2010.

[25] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. pages 1–116. Springer-Verlag, 1995.