Collaborative Memory Management for Reactive Sensor/Actor Systems

Marcel Baunach
University of Wuerzburg, Germany

Copyright © 2010 IEEE. Reprinted from *5th IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp)*.

This material is posted here with permission of the IEEE. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org.

By choosing to view this document, you agree to all provisions of the copyright laws protecting it.

Collaborative Memory Management for Reactive Sensor/Actor Systems

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 imposes demanding challenges on the underlying system design. This especially affects real-time operation, resource sharing and memory usage. Using preemptive task systems is one way to retain acceptable reactivity within highly dynamic environments. Yet, since memory is commonly rare and can often not be assigned statically, this rapidly leads to severe memory management problems among tasks with interfering and even varying requirements. Finding an allocator which suitably adapts to changing conditions while covering both issues is generally hard. We present the novel *CoMem* approach for maintaining high reactivity and efficient memory usage within such systems. With respect to task priorities and the typically limited performance and resources of sensor nodes, our technique facilitates compositional software design by providing tasks with runtime information for yet collaborative and reflective memory sharing. Thereby, we require no special hardware-support like MMUs but operate entirely software-based. An evaluation will show that our approach can still allow allocation delays to be close to the best case and inversely proportional to the requester's priority.**

***Index Terms*—dynamic memory, preemptive/prioritized tasks, reflection, embedded systems, real-time**

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 [1] and fine grained code updates [2]) are more and more used to manage design and operation of these embedded systems. Then, adequate interaction between these modules is essential to handle typical compositional problems like task scheduling, resource sharing or even real-time operation [3]. Concerning this, we find that current WSAN research is still too limited to static design concepts. As already stated in [4], next generation embedded systems will be more frequently used as reactive real-time platforms in highly dynamic environments. Here, the true system load varies considerably and can hardly be predicted a priori during development. 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 the novel CoMem approach for collaborative 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) influence on the remaining system and thus, cannot collaborate adequately. In this respect, CoMem follows classic reflection concepts [5], [6] and introduces a new policy into operating system kernels, by which programs can become 'self-aware' and may change their behavior according to their own current requirements and the system's demands. As often suggested [5], 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. In turn, these so called *hints* allow 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. The decision between following or ignoring a hint is made by each task autonomously and dynamically at runtime, e.g. by use of appropriate time-utility-functions [7]. Finally, CoMem is not limited to embedded systems and the WSAN domain, but can be applied to real-time operation in general.

We'll initially review some related techniques from existing work before details about our new approach will form the central part of this paper. Our reference implementation of CoMem will show that – despite of the problem's complexity – it is efficiently usable 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.

II. Related Work

Dynamic memory management is subject to intense research efforts and plays an important role in current software design [8], [9], [10], [11]. 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 due to frequent (de)allocations. Unfortunately, heap methods suffer from a few 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 [12]:

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 executing related functions as fast as possible or even in deterministic time.
F3 *Maximize error detection* or even avoid tasks to corrupt data by illegal access to unassigned memory sections.
F4 *Maximize tunability* to account for dynamic and task specific requirements like real-time operation.
F5 *Maximize portability and compatibility* by just relying on 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 memory blocks.

However, according to [13], 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 reorganization by the memory manager is hard or even impossible then, since it lacks information about the actual memory usage by the current owner tasks.

Thus, the use of dynamic memory is largely avoided for time or safety critical systems [10]. For these, F1 and F2 must be extended to support guaranteed allocation and the knowledge about the allocators WCRT. At least, real-time operating systems use so called *pools* of fixed-size memory blocks (low external, high internal fragmentation) and constant allocator execution time. In contrast, blocks of arbitrary size commonly provide more flexibility (less internal, more external fragmentation) at higher costs 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 on memory reorganization in case of allocation failures, we won't go into detail about these techniques but refer to [10], [13] instead.

When considering WSAN operating systems, only few support dynamic memory for arbitrary use by application tasks: TinyOS 2.x [14] supports a semi-dynamic pool approach in which a fixed number of blocks can be statically assigned to a task. Tasks can release their blocks to the pool and reallocate blocks as long as the initial number is not exceeded. Contiki [2] offers dynamic memory for storing variables of dynamically loaded modules. In SOS [15] a block based first-fit scheme with $32 \times 16,\ 16 \times 32,\ 4 \times 128$ byte is used to store module variables and messages. MantisOS [16] uses best-fit to allocate arbitrary size blocks for thread stacks and the networking subsystem only. In [8] a combination of sequential fits, segregated free lists and the buddy system is proposed for Nano-Qplus. SensorOS [17] supports a pool based approach for messages and a buddy system for blocks of any size. To find a suitable allocator for concrete WSAN applications, SDMA [11] 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 [2]. Beside SensorOS, no OS provides the arbitrary use of dynamic memory for tasks. In particular, none provides any mean for dynamic memory reorganization 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 reorganization in small embedded systems without brute force methods like (energy intensive) swapping, memory revocation or task termination with potentially critical side effects.

III. The CoMem Approach

Reflection based task cooperation is a powerful strategy to share resources on-demand and "upwards" along with the task priorities. We adapt the strengths and benefits for the special case of dynamic memory allocation. This section explains the central idea, design and implementation decisions behind our new CoMem concept by addressing the specific problems.

A. Dynamic Hints for On-Demand Resource Sharing

Our CoMem approach is generally based on Dynamic Hinting [18], 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 blocking 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) [19] – reliably improves and stabilizes the overall system performance. Therefore, the approach reduces priority inversions, resource allocation delays and even recovers from deadlocks where required.

According to PIP, 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 task itself 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 but 'catches' an early wakeup to react on its blocking influence (→Fig. 1a).
- Hint Handler: When *ready* (i.e. 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

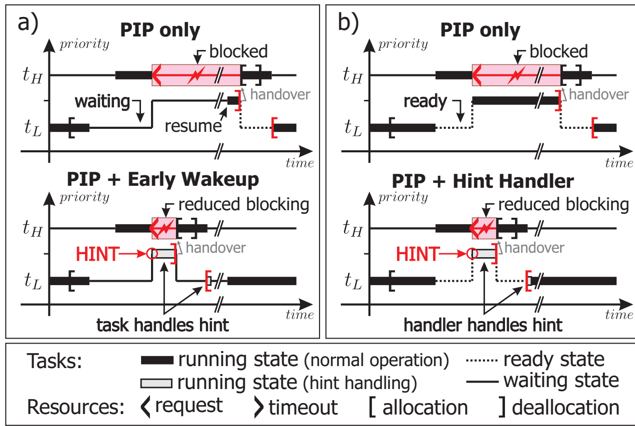

Figure 1. Hint handling when task t_L blocks in a) *waiting* or b) *ready* state

kernels, hint handlers allow to operate literally non-preemptive 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 and *decide* between following or ignoring the hint. When following:

1. Save the resource state and stop its operation.
2. Release the resource. *This will immediately cause an implicit task preemption due to the resource handover* to the higher priority task which could not be served until now.
3. Re-allocate the resource upon resumption (asap).
4. Restore the resource state and restart its operation.

B. 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 nodes 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 Section V).

So, the first problem is indirect *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 is rejected and we see a priority inversion.

Simply using e.g. PIP for raising $p(t_A), p(t_B)$ and potentially accelerating 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

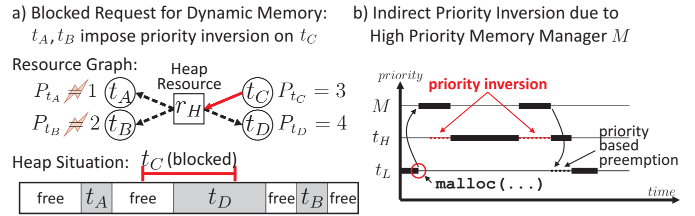

Figure 2. Critical Scenarios during Dynamic Memory Management

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 blocking 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. This would still cause significant CPU load upon short periods and potentially miss sufficiently large free memory areas upon long periods. Anyway, the memory manager does not know that t_C actually still waits for memory between the polls and can neither serve t_C nor reserve memory. If supported, another load intensive option are lock-free methods like [9]. To minimize the load by currently not serveable tasks, our approach uses a task-blocking `malloc()` function and transfers the memory organization to the memory manager subsystem M. In turn, we have to

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

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 high priority task t_H ($p(t_L) < p(t_H)$) 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. Yet, 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. 3).

This way, we provide the memory management subsystem with information about how long we are willing to wait in worst case and simultaneously supply a defined amount of time for reorganization of the heap space. τ will also be passed to the blocking tasks which in turn can use it within their time-utility-functions [7] (→ Section V-B, feature F4).

Finally, the heap (re)organization policy is a critical core element within all memory managers and was already considered in many ways, e.g. [10], [13]. Beside task termination, two elementary options exist and are supported by CoMem:

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

We still 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 reorganization which belong to lower prioritized tasks and would lead to sufficient continuous space to serve higher prioritized requesters. Furthermore, relocation takes precedence over release while the latter is always more powerful.

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 cannot be updated automatically and would still transfer data from/to old blocks. 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, we advise them 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 reorganization by means of some helper functions like e.g. `relocate()` and `free()` from Fig. 3.

Before heading to the technical details and the impact on the programming model, we'll outline our design decisions:

1. Persisting memory allocations must not prevent further requests. Then, CoMem always knows about all system wide requirements and can produce adequate hints/suggestions.
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, that as long as no MMU is available, our concept cannot *protect* memory against unauthorized access but only coordinate its exclusive sharing (→feature F3).

IV. 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 *Smart*OS [20] since it fulfills these requirements. Beyond, it offers quite common characteristics, and thus is a good representative for the adaptation of similar systems. In addition, it is available for several MCU architectures like MSP430, AVR, SuperH (→feature F5).

A. SmartOS 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. Beside 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 [18] and Section III-A.

B. CoMem Implementation Details

Next, we'll show how to achieve the CoMem design considerations from Section III-B. 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:

```
1 typedef struct {
    unsigned int size;     //1W: size in machine words
3   volatile int *base;    //1W: (absolute) start address
    Resource_t   broker;   //2W: associated resource
5   advise_t     advise;   //1W: What to do upon a hint
    MCB_t        *next;    //1W: linked list pointer
7 } MCB_t;                 // Total RAM size: 6W
```

Since we want tasks be be informed *immediately* if they block a higher priority task due to a dynamic memory allocation, we associate one *Smart*OS resource – a so called *broker* – with each allocated memory block. The broker grants directed influence by the resource manager on the task holding the block. While this communication option is entirely missing in all memory management systems we found so far, it does not only allow the targeted generation of hints (via dynamic hinting) but also provides two important advantages:

1. We implicitly adapt the underlying resource management policy (e.g. PIP) for the memory management subsystem. Thus, 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.

In contrast to many other approaches which maintain a list of free memory areas [16], CoMem uses a linked list of

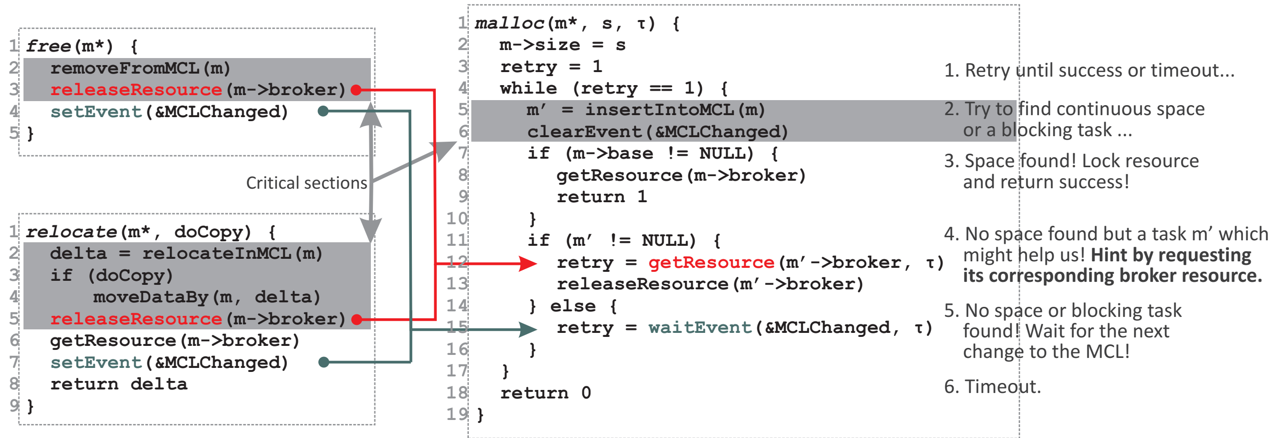

Figure 3. CoMem Function Interactions

MCBs for currently allocated blocks. Internally, this Memory Control List (MCL) is sorted by base address pointers 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 restricted sensor node performance and still provided good results within our testbeds. In fact, allocated blocks must be scanned anyway to select one for reorganization. Complexity: $O(n)$.

Since CoMem's public functions are executed concurrently in each callers' context, access to the MCL is protected by a unique *Smart*OS semaphore which results in critical (not atomic!) sections (→Fig. 3).

For allocations via `malloc()`, we supply three parameters:

1) An MCB m for managing the block. Since MCBs are supplied by tasks as required, the CoMem library needs not to reserve a fixed number in advance.
2) A requested size s for the continuous block.
3) A deadline 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). On success (L7), the corresponding broker resource m_b is locked by the caller and we are done. Since m_b 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 deadline τ has expired, it also releases m'_b temporarily (`free()`:L3, `relocate()`:L5) to indicate the changed memory situation and to trigger a new retry for m. If the deadline expired, retrying stops and `malloc()` returns 0 (L18). If no blocking task/block was found (L14), `malloc()` waits for the next modification to the heap space. Again, if this happens within the deadline one more retry is triggered, otherwise 0 is returned to also indicate the timeout (L18).

Accordingly, `free()` and `relocate()` are rather simple:

`free()` simply removes the specified MCB m from the MCL and releases the broker resource m_b. Finally, it indicates the MCL modification by setting the corresponding event.

`relocate()` searches a new location for the supplied block m (cyclic next-fit) by which more continuous free space becomes available (L2). If requested, it moves the data to the new location. Finally, it temporarily releases its own broker resource m_b (L5/6) and triggers the `MCLChanged`-event (L7) to resume waiting tasks. The data shift is returned in bytes.

The remaining problem is how to *reasonably* select a blocking MCB m' for generating a hint on. While scanning the MCL for free space, we search for two types of MCBs: The first one would at least produce the requested space if it was relocated and the other one if it would be released entirely. In consequence, m'_{advise} will be set to either $relocate$ or $release$ while $relocate$ 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 advise for a suitable reaction. When considering the blocked task, a release is always at least as effective as a relocation.

V. Real-World Applications and Test Beds

For analyzing our CoMem approach of combining temporally limited memory requests, on demand heap reorganization via dynamic hints and the priority inheritance protocol, we extended *Smart*OS as described. The implementation was done for Texas Instrument's MSP430 [21] family of microprocessors, since these are found on a large variety of sensor nodes. Requiring $4 + 1$ kB of ROM and $40 + 8$ B of RAM for the whole kernel and the CoMem library, the typically small memory footprint of sensor nodes was considered carefully to leave sufficient room for the actual application. Our test scenarios were executed on SNoW[5] sensor nodes [22] 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.

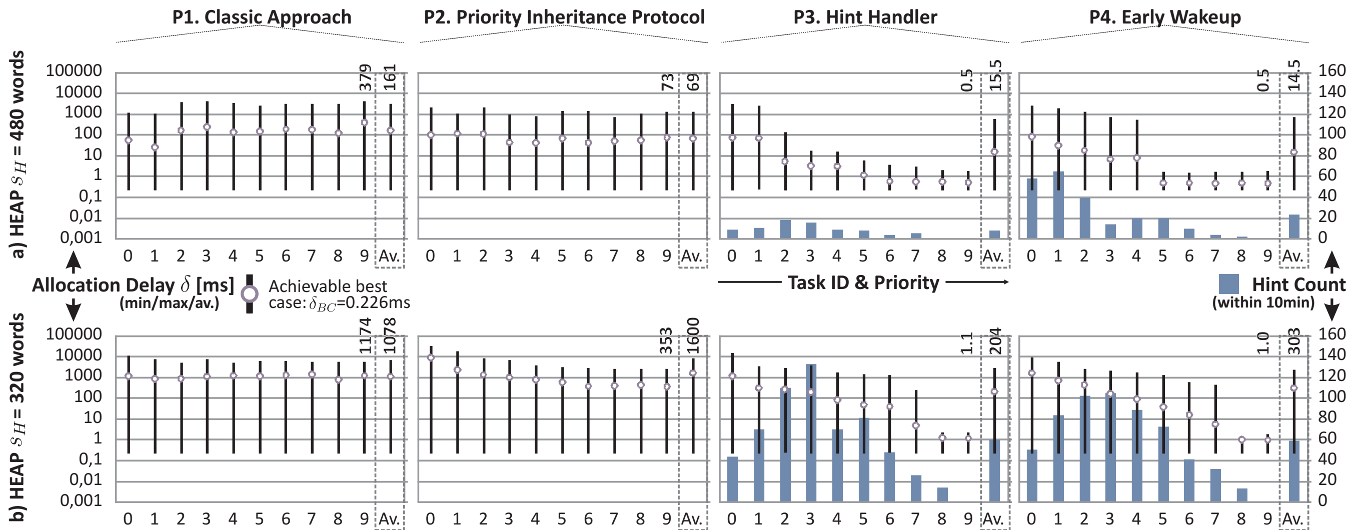

Figure 4. CoMem stresstest results for different policies and heap sizes (ascending base priorities for $n = 10$ tasks)

A. Dynamic Memory Stresstest

The first scenario analyzes our approach under extreme conditions with n tasks $t_0 \cdots t_{n-1}$ and many concurrent memory requests. Ascending base priorities $P_{t_i} = i$ were assigned 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 of step (1), the CPU time of 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 used an infinite deadline $\tau = \infty$ 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: Here, we omitted the request for a blocking task's broker resource during `malloc()` (L12). Instead we always waited for the `MCLChanged` event if no continuous space was found. This avoided hints and the chance for collaborative memory sharing entirely.
- P2 PIP only: We implemented `malloc()` as shown in Fig. 3 but simply ignored the emerging hints. Though a blocking task did not collaborate explicitly, its active priority was at least raised to the priority of the task it blocked and it received more CPU time for step (3), then.
- P3 Hint Handlers: This time, each task supplied a hint handler for immediate injection into its own execution when blocking a higher prioritized task.
- P4 Early Wakeup: Finally, we implemented the tasks to sleep while holding a memory block. Yet, tasks were resumed immediately when blocking a higher prioritized task.

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

We configured the test bed using several task counts n, 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 of $s_B \in \{32, 64\}$ words and heap sizes of $s_H \in \{320, 480, 640\}$ words. 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 benefit of our collaborative approaches (→Fig 4a). While the non-collaborative approaches deliver almost uniform average allocation delays around 161 ms (P1) and 69 ms (P2), both do not reflect the task's intended base priorities at all. In contrast, using hints manages to reliably signal tasks about their blocking 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 blocking, 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 10 and 5, respectively.

Reducing $s_H := 10 \cdot 32 = 320$ words increases competition and allocation delays to be even more demanding (→Fig 4b). Still, the different task priorities are not visible for P1, while 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 concepts. Yet, these still manage to serve tasks according to their intended relevance: the two most important ones still achieve an average delay of $\delta_{av} \approx 1\,\mathrm{ms}$ 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}. Please note that each task's hint count highly depends on its number of successful memory allocations. Since requests from low priority tasks are granted less frequently, these also generate less hints. Medium priority tasks are served more often, tend to cause more blockings and finally receive more hints. While requests from high priority tasks are also commonly granted, these rarely block still higher prioritized tasks. In fact, the peak can be shifted towards the low priority tasks by increasing the sleep time (1) between two allocations while increasing the operation time (3) reverses this effect.

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 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 and allowed almost best case delays for high priority (real-time) tasks.

B. Real World Example

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 [23] comprises several static anchors as references for the applied localization algorithms. These anchors run six preemptive tasks for several software modules (i.e. radio communication, temperature compensation, etc). Two tasks are exceptionally memory intensive: t_{US} runs a DSP algorithm for ultrasound chirp detection, recording and processing – i.e. calculating the time-of-flight between a synchronizing radio packet and the corresponding chirp. Each time it uses a capture compare unit to trigger an ADC/DMA combination which in turn samples the chirp signal into a buffer of $4\,\mathrm{kB}$. Then the DSP calculates the arrival of the first wavefront with a precision of ≈ 5 µs.

In parallel, each node runs a task t_{RC} for a remote management and software update system. Compared to other parts of the system, this service is rarely used. But as soon as a new firmware image is announced via radio, t_{RC} requests $n \cdot 256\,\mathrm{B}$ of RAM and successively fills this buffer with image fragments of $256\,\mathrm{B}$ each. After n radio packets, the buffer is transferred to an external flash memory (block size: $256\,\mathrm{B}$). 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 radio and flash as well as the spacing delay between successive radio packets. Furthermore, the external flash consumes less time and energy when powered up less frequently but for longer burst writes. In fact we use $n = 20$ and thus require $5\,\mathrm{kB}$ for the buffer.

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 I
MEMORY ALLOCATION DELAYS WITHIN SNOW BAT

From the $10\,\mathrm{kB}$ of the controller's RAM, kernel and tasks require about $4\,\mathrm{kB}$. The remaining $6\,\mathrm{kB}$ are used as heap space for dynamic memory. Thus, the chirp sampling buffer ($4\,\mathrm{kB}$) and the image data buffer ($5\,\mathrm{kB}$) must be allocated dynamically. In fact, aborting or not even starting a chirp detection is not that critical: The node will be available for later measurements. Missing an image fragment is highly critical indeed! Though ACKs and other safety strategies are applied, an incomplete reception causes expensive retransmissions and write accesses to the external flash. Thus, t_{RC} imposes an upper bound 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 $3\,\mathrm{Hz}$), it allocates the memory at system start and configures the DSP process and DMA controller according to the assigned base address. Yet, t_{RC} is more time-critical and thus receives a higher base priority $P_{t_{RC}} > P_{t_{US}}$. As soon as t_{RC} requests dynamic memory, t_{US} is 'hinted' immediately. If the sampling buffer is currently not in use, it is simply released to serve t_{RC} quickly. Otherwise, t_{US} requests the blocked task's remaining allocation deadline τ for application of a simple time-utility-function: If τ won't allow to release the memory in time, t_{US} simply ignores the hint and continues its operation – leading to an unavoidable deadline violation for the blocked task t_{RC}. Otherwise, it 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 to raise $p(t_{US}) \geq P_{t_{RC}}$ while t_{RC} blocks on `malloc()`, the WCRT of t_{US}'s hint handling defines the minimal tolerable delay between image announcement and the first fragment. 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 I 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 handover which is just limited by the required time for aborting any currently running operation. Then, we observed $\delta_{max} \leq 1.4$ ms and can finally use a memory allocation deadline of $\tau = 2$ ms for t_{RC}.

This test bed showed, that CoMem allows tasks to coordinate sporadic memory requirements without explicit communication. Our approach provides sufficient information (via hints) and adequate task priorities (via PIP) to allow tasks an reflective resolution (via TUFs) of their blocking influence. In fact, a blocking task needs not to know which task it blocks. 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.

VI. Conclusion and Outlook

In this paper, we introduced the 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 reducing allocation delays. In particular, individual task base priorities are considered carefully to keep each task's progress and reactivity close to its intended relevance. The basic idea is to analyze emerging task/memory conflicts at runtime and to provide blocking tasks with information about how to safely reduce the blocking of more relevant tasks. By following these hints, tasks can implicitly collaborate without explicit knowledge of each other. This even reduces bounded priority inversions and achieves allocation delays which are just limited by the pure handover overhead (i.e. deallocation and reallocation). The reflective concept even allows each task to decide dynamically between collaborative or egoistic behavior with respect to its current conditions and other tasks' requirements. However, CoMem can not guarantee any time limits since these highly depend on the behavior of the involved tasks. Yet, even if used sparsely, our approach is definitely better compared to non-collaborative operation. Thus, a well-thought application design still remains elementary, but compositional software is already facilitated. In fact, our approach is not limited to the WSAN domain but may also extend other embedded systems.

The test beds and the integration of our novel concept into the real-time operating system *Smart*OS showed, that the effective use of preemptive tasks for creating reactive 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.

At present we are working on improved hint generation and application of TUFs by considering more application specific factors like programming-by-contract for limiting the WCRT of hint handlers. Another area is the evaluation of CoMem within multi-core systems [24], [25] where blocking may induce hints between the subsystems.

References

[1] 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*, vol. 9, no. 3, Mar. 2009.

[2] 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*.

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

[4] 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.

[5] 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.

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

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

[8] 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*. ACM, 2007.

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

[10] M. Masmano, I. Ripoll, P. Balbastre, and A. Crespo, "A constant-time dynamic storage allocator for real-time systems," *Real-Time Syst.*, vol. 40, no. 2, pp. 149–179, 2008.

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

[12] D. Lea, "A memory allocator," http://g.oswego.edu/dl/html/malloc.html.

[13] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles, "Dynamic storage allocation: A survey and critical review." Springer-Verlag, 1995.

[14] U. Berkeley, "TinyOS," Web site http://www.tinyos.net/, 2010.

[15] 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*. New York, NY, USA: ACM, 2005, pp. 163–176.

[16] 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.*, vol. 10, no. 4. Kluwer Academic Publishers, 2005.

[17] 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.

[18] 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*, 2009, pp. 31–40.

[19] 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.

[20] 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.

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

[22] M. Baunach, R. Kolla, and C. Mühlberger, "SNoW^5: A versatile ultra low power modular node for wireless ad hoc sensor networking," in *5. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze*, 17.–18. Jul. 2006.

[23] M. Baunach, R. Kolla, and C. Muehlberger, "SNoW Bat: A high precise WSN based location system," Univ. of Wuerzburg, Tech. Rep. 424, 2007.

[24] 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*. Washington, DC, USA: IEEE Computer Society, 2008.

[25] A. Easwaran and B. Andersson, "Resource sharing in global fixed-priority preemptive multiprocessor scheduling," in *IEEE Real-Time Systems Symposium*, T. P. Baker, Ed. IEEE Computer Society, 2009.