CoStack: Collaborative Stack Sharing for Real-Time Embedded Systems

Fabian Mauroner
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
Email: mauroner@tugraz.at

Marcel Baunach
Institute of Technical Informatics
Graz University of Technology
Graz, Austria
Email: baunach@tugraz.at

Abstract—**Embedded real-time systems are targeting for economical stack memory usage and predictable execution flows, what is challenging to unify. In this paper, we propose CoStack, a collaborative stack sharing approach across tasks. CoStack allows defining a collaborative stack memory that can be used by a higher prioritized task if the stack runs out-of-memory. Thus, CoStack virtually reduces the stack memory consumption, leading to a lower memory requirement, and concurrently remains predictable, what is desired for real-time systems. This paper presents an experimental evaluation of CoStack, the synthesized results in a Field Programmable Gate Array (FPGA) and some implementation details of CoStack.**

***Keywords*–Embedded Systems; Stack handling; Operating-System-Awareness; FPGA implementation**

I. Introduction

Modern real-time embedded systems require, due to their growing complexity and flexibility, evermore memory to fulfill all their challenging requirements. However, embedded system's memory is a restricted resource; thereby, it has to be used in an efficient way.

Global variables are always available for the complete embedded system's life, but local variables are only available and allocated on demand. Therefore, local variables are dynamically allocated on a specific space in the memory, named *stack frame*. The *stack pointer*, indicating the threshold of valid and non-valid data in the stack frame, grows if a local variable is allocated or if Central Processing Unit (CPU) registers are temporally stored (e.g., on function prologue and epilogue). The stack pointer shrinks if the local variables or temporally stored registers are not needed anymore.

In state-of-the-art real-time Operating Systems (OSs) [1][2] for each task an own individual stack frame is allocated; although, it is not fully utilized simultaneously. Therefore, [3][4] show approaches to use a common shared stack frame among all tasks. This reduces the overall stack memory consumption, but restricts the schedulability of all tasks. The reason therefor is that a stack cannot grow if the task's stack pointer is not on the top of the common stack frame. Otherwise, it would destroy data from another task in the common stack frame. Therefore, an individual stack frame is assigned to the tasks in real-time systems by accepting an increasing overall memory consumption. The reason therefor is, for real-time systems, the schedulability and satisfaction of real-time constraints is an essential requirement. This has to be improved to reduce the required computation power and CPU frequency, to reduce the costs and power of the developed embedded real-time system.

Consequently, both, efficient stack memory and schedulability must be unified to improve the memory consumption and to fulfill all the real-time requirements.

Observations showed [5] that not each task fully utilizes its individual stack frame simultaneously, wherefore the approach to share stack memory space among all tasks is used. To avoid the restriction of the schedulability, it must be avoided or at least timely bound that a task is blocked for requesting stack memory. That is possible with the concept of address virtualization [6]. However, this concept requires an additional hardware component, namely the Memory Management Unit (MMU). It translates all Virtual Memory (VM) addresses into Physical Memory (PM) addresses. However, an MMU is only rarely found in embedded systems, because it requires a lot of power and it introduces non-predictable memory accesses. That has to be avoided for real-time systems, which aim for predictability. Therefore, in [7] we presented a dynamic stack sharing approach, which uses VM addresses only for the stack memory and ensures a predictable stack memory access and stack pointer adjustment if the underlying memory architecture behaves predictable, too.

Since the memory is a scarce resource, the software developer has to use it sparsely. However, sometimes it is not possible to optimize the code (e.g., to use an algorithm with less stack memory consumption). Thus, to avoid an out-of-stack, the software developer must guarantee enough stack memory spaces for all tasks at any time. Otherwise, a task would be unpredictably blocked and deadlines may be violated.

A. Related work

In the recent years, there has been done a lot of research to reduce the stack memory consumption, with completely different approaches:

Wang *et al.* proposed the preemptive threshold scheduling in [8], which is used in the ThreadX real-time OS [9]. It is based on Rate Monotonic (RM) scheduling and extends each task with a threshold priority, beyond its nominal priority. If a task is scheduled, its threshold priority is the new priority that must be exceeded to preempt the task by another task's nominal priority. This solution leads to non-preemptive task groups and these are able to use a common shared stack frame. Further, it may improve the schedulability compared to the standard RM scheduling. Nevertheless, sharing the stack memory is not possible between non-preemptive task groups and/or preemptive tasks; thus, sharing the stack memory is limited.

In [10], Chu *et al.* proposed to use a binary translation and a specific kernel by using VM addressed stacks in embedded systems. Thus, their approach allocates only that memory that is required, but the authors showed that for each stack pointer access and adaptation the required run-time is enormous.

Yi *et al.* [11] showed an approach, where a tool analyzes the stack consumption of each task and modifies the developed code on compile time. The modified code calls the *on demand stack* library at each prologue and epilogue of a function. In their use case scenario, the stack memory consumption indeed reduces; however, the execution time increases by $10\,\%$.

Other solutions proposed to allocate the stack memory on the heap. In older works, as for instance [12], on each function prologue and epilogue the heap allocation and deallocation is called, respectively. However, these calls lead to a large run-time overhead for each function.

Works as [13][14] also allocate the stack on the heap. With analyses at compile time, they are able to reduce the large run-time overhead for allocating and deallocating stack memory on the heap. Nevertheless, run-time checks are still required to allocate and deallocate the stack memory.

Middha *et al.* [5] propose to allocate an individual stack frame to each task. If a task overflows (i.e., out-of-stack) its individual stack frame, their approach allocates unused stack memory in another task's individual stack frame. Without code optimization, their run-time consumption increases by around $23\,\%$; with an optimization about $3\,\%$. However, the optimized solution restricts programming features as function pointers and recursive functions.

None of the mentioned works is able to free stack memory voluntarily for a higher prioritized task if no stack memory is available. In [15], Baunach proposed CoMem that is a collaborative memory management in the heap memory for dynamic memory. There, each memory block (in the heap) is handled as a system resource. If a task requests the memory block and is used by another lower prioritized task, the lower prioritized task will be informed. That task has then the control to free the memory block or not.

CoMem enables a collaborative usage of memory blocks in the heap but not for the stack memory. Therefore, in this paper we present *CoStack*, a collaborative stack sharing concept based on a hardware extension, which enables the reduction of the whole stack memory consumption by defining parts in the source code in which the stack memory is collaborative. With the state-of-the-art approaches, there is no possibility to give a higher prioritized task the advantage to use the stack memory instead of a lower prioritized task that owns collaborative stack memory that might voluntarily be released. CoStack ensures the allocation of the required stack memory for higher prioritized tasks by freeing collaborative stack memory from lower prioritized tasks if an out-of-stack condition would result. CoStack does not require a code analysis at compile time; therefore, it is possible to use our approach also in highly dynamic environments, as the Internet of Things (IoT), Industry 4.0, or automotive applications.

The rest of the paper is organized as follows: First, Section II describes in detail the fundamentals of CoStack. Second, Section III analyses the memory improvement and shows the schedulability analysis. Next, Section IV demonstrates implementation aspects in our development platform. Section V shows the CoStack evaluation with an example and the synthesized results for a Field Programmable Gate Array (FPGA). Last, we summarize this work in Section VI.

II. Collaborative Stack Sharing

The collaborative stack sharing approach is based on StackMMU presented in [7]. First, we introduce the terminology and the system assumption. Second, we introduce the fundamentals of StackMMU. Third, we show the extension of the basic StackMMU, which is needed for providing the required information to the hardware. Last, the collaborative stack sharing approach, CoStack, is described in detail.

A. Terminology and Assumption

For CoStack we assume a Reduced Instruction Set Computer (RISC) load/store single-core CPU with Control Status Registers (CSRs), which are CPU registers accessible by specific instructions. Further, we define a multi-tasking system with $\tau \in T$ as a task in the set of tasks T. For each task τ we define, the priority p_τ, the Worst Case Execution Time (WCET) with C_τ defining the time executing on the CPU, and the period or minimum interarrival time with T_τ for periodic or sporadic tasks, respectively. The longest time, in which a priority inversion [16] occurs (i.e., a lower prioritized task runs instead of a higher prioritized task) is denoted as the blocking time B_τ of task τ. Further, we assume that the context switch in the OS and the OS itself consumes no computation time. The stack usage $\sigma_\tau(t) \in \mathbb{N}_0$ defines the stack memory consumption of task τ at time $t \in \mathbb{N}_0$, what can be analyzed with static code analyzers.

B. StackMMU

The StackMMU [7] uses VM addresses for the stack memory. Thus, each task's stack pointer points to a virtual address and StackMMU translates that address to a PM address. For that, the memory is divided into a common area and into a stack area, as depicted in Figure 1. In the common area,

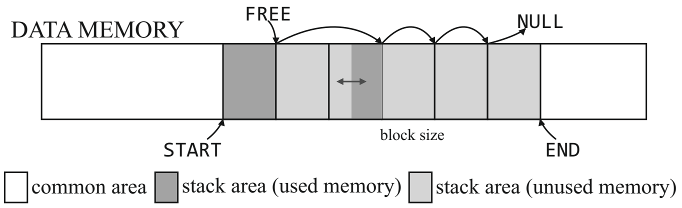


Figure 1. Memory layout of the StackMMU.

all the global data of a task is stored and accessed by PM addresses. The memory for the stack is located, in the stack area. Thereby, the stack area is again divided into blocks, called *pages*. Each page has the same size and is configurable including the start and end of the whole stack area at startup. A linked list contains all available pages, whereby the `FREE` register points to the first free available page. Thus, if a task requires more stack memory and exceeds the available memory in the last appended page, the StackMMU assigns a new page to that task. Thereby, the hardware uses the register `FREE` and updates the Task Control Block (TCB) of the task with the address of the new allocated page. On a stack memory access, first, the hardware reads the PM base address in the

TCB; and second, it accesses the content in the stack area. If the stack page is not required anymore, the hardware frees the page and updates all the required pointers (i.e., `FREE` and the pointer to the next free page in the page). Before a specific stack operation (i.e., growth, access, or shrinkage) is executed, all three operations are performing a read memory operation leading to an additional memory access. However, if the memory access is deterministic, as in many embedded systems, the whole stack operation is also executed in a predictable time as desired for real-time systems.

C. MultiStackMMU

StackMMU restricts the size of the stack pointer change for growing and shrinking in one instruction to the size of a stack page. That limitation is handled by a modified compiler. However, we purged that limitation with MultiStackMMU.

Here, if a stack grows or shrinks more than one page, the hardware extension performs the assignment or deassigment of pages one after another, respectively. This leads to a longer run-time to perform these operations; nevertheless, the execution time remains predictable if the memory accesses itself are deterministic.

Through this extension, the modified compiler, which limited the stack pointer change to one stack page size, is not required anymore. Besides, the hardware is now aware of the number of required pages on a stack memory request, necessary for performing our collaborative stack sharing approach CoStack.

D. CoStack

As long as in the stack area are available more unused pages than required by a task, the pages are properly assigned to the task by the MultiStackMMU approach. However, if there are not enough available pages, an out-of-stack condition occurs. An out-of-stack condition is handled by the OS; nevertheless, the task will be unpredictably blocked until stack memory is available. This would lead to a reduction of the system performance and may lead to real-time constraint violations. Thus, to reduce the possibility of an out-of-stack condition, enough stack memory must be provided.

Instead of blocking the task as long as not enough stack memory is available, CoStack deallocates stack memory from tasks that define their used stack memory as collaborative.

If CoStack detects that the required stack memory pages exceed the free available pages, a *collaborate exception* is triggered and the OS must handle it. Thereby, the stack growth is aborted by the hardware and the OS saves the context of the task. After rescheduling that task, the stack growth instruction will be repeated (i.e., the program counter of the stack growth instruction is stored in the TCB). In the exception handler, the OS searches a lower prioritized task that provides collaborative stack memory. Then, the OS modifies the program counter and schedules the collaborative task. The collaborative task frees the collaborative stack memory and yields itself to return to the OS. There, the scheduler selects the highest prioritized runnable task, which may be the same as the previous one. If a task, that requires stack memory, is scheduled and enough unused pages are available, the required pages are assigned to the task. Otherwise, once again a collaborate exception is triggered and the OS iterates through the tasks as before to search a collaborative task. In case that there are no lower prioritized tasks with collaborative stack memory, the OS has to define a handling strategy to handle this failure as in standard out-of-stack approaches.

Figure 2c demonstrates how a code part can be tagged with collaborative stack. The macros in Figure 2a, Figure 2b, and Figure 2d generate the code for handling CoStack properly. Additionally to the tagged code, which uses a collaborative stack memory, a handler is defined. The handler is only executed if the collaborative stack memory was deallocated for performing some clean-up work, similar to the catch primitive in programming languages such as Java or C++.

Figure 2a shows how the collaborative stack mechanism is performed. First, it checks if the currently running task already defines a collaborative code part. If so, the collaborative code part is already a part of an upper tagged collaborative code part and the `try` part is immediately executed. Otherwise, the handler address and the collaborate frame pointer are stored in the task's TCB. Additionally, the callee saved registers are stored on the stack, to restore them if the collaboration must be performed. Afterwards, the `try` part is executed. If there was no other task requiring the collaborative stack, the callee saved registers on the stack are discarded because the program flow was not manipulated. Otherwise, the program counter continues at the label `COLLABORATE` after a context switch. There, the callee saved registers are restored and the collaborative stack memory space is freed, by using the stored collaborate frame pointer. After that, the collaborate frame pointer in the TCB is cleared and the syscall `yield()` is called for a self-preemption to allow the OS to schedule a higher prioritized task that may wait for stack memory.

III. Analysis

In this section, we analyze the memory utilization of the collaborative stack sharing approach CoStack and compare it with the StackMMU approach. Further, we show the schedulability analysis of CoStack.

A. Memory Consumption

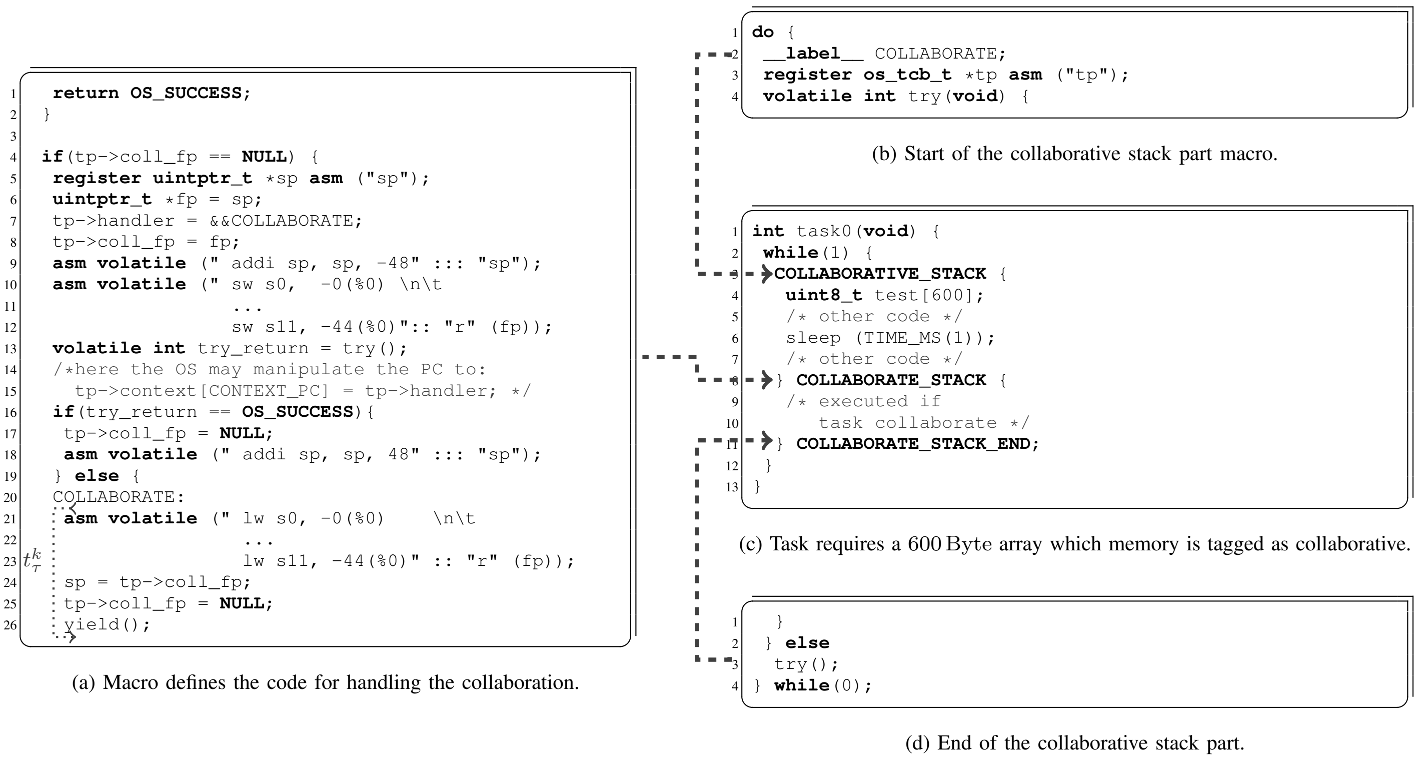
In the StackMMU approach, the stack grows and shrinks with the stack page size `page_size`. There, the size of the stack changes over time t, depending on the required memory $\sigma_\tau(t)$ by a task τ at time t. Thus, the stack memory consumption of the whole system U is calculated as follows:

$$U(t, \texttt{page\_size}) = \sum_{\tau \in T} \left\lceil \frac{\sigma_\tau(t)}{\texttt{page\_size}} \right\rceil \cdot \texttt{page\_size} \tag{1}$$

Let S denote the totally assigned stack space. To avoid an out-of-stack condition, the stack memory consumption U is not allowed to exceed the totally assigned stack memory S. Otherwise a task would be unpredictably blocked what restricts the schedulability:

$$\forall t \in \mathbb{N}_0 : \quad U(t, \texttt{page\_size}) \leq S \tag{2}$$

With the introduction of the collaborative stack sharing approach, each task τ may define some collaborative stack memory $\kappa_\tau(t)$ at time $t \in \mathbb{N}_0$. This collaborative stack memory κ is available for all higher prioritized tasks, leading



```
  return OS_SUCCESS;
 }

 if(tp->coll_fp == NULL) {
  register uintptr_t *sp asm ("sp");
  uintptr_t *fp = sp;
  tp->handler = &&COLLABORATE;
  tp->coll_fp = fp;
  asm volatile (" addi sp, sp, -48" ::: "sp");
  asm volatile (" sw s0,  -0(%0) \n\t
                  ...
                  sw s11, -44(%0)":: "r" (fp));
  volatile int try_return = try();
  /*here the OS may manipulate the PC to:
    tp->context[CONTEXT_PC] = tp->handler; */
  if(try_return == OS_SUCCESS){
   tp->coll_fp = NULL;
   asm volatile (" addi sp, sp, 48" ::: "sp");
  } else {
  COLLABORATE:
   asm volatile (" lw s0, -0(%0)    \n\t
                   ...
                   lw s11, -44(%0)" :: "r" (fp));
   sp = tp->coll_fp;
   tp->coll_fp = NULL;
   yield();
```

(a) Macro defines the code for handling the collaboration.

```
do {
 __label__ COLLABORATE;
 register os_tcb_t *tp asm ("tp");
 volatile int try(void) {
```

(b) Start of the collaborative stack part macro.

```
int task0(void) {
 while(1) {
  COLLABORATIVE_STACK {
   uint8_t test[600];
   /* other code */
   sleep (TIME_MS(1));
   /* other code */
  } COLLABORATE_STACK {
   /* executed if
      task collaborate */
  } COLLABORATE_STACK_END;
 }
}
```

(c) Task requires a 600 Byte array which memory is tagged as collaborative.

```
  }
 } else
  try();
} while(0);
```

(d) End of the collaborative stack part.

Figure 2. Macros for the collaborative stack sharing approach with a usage example.

to the available collaborative stack pages K_τ at time t for task τ.

$$K_\tau(t) = \sum_{\forall i: p_{\tau_i} < p_\tau} \left\lfloor \frac{\kappa_{\tau_i}(t)}{\texttt{page\_size}} \right\rfloor \cdot \texttt{page\_size} \quad (3)$$

As mentioned in Section II, if a higher prioritized task requires non-available stack memory but collaborative stack memory is available, the collaborative task deallocates its collaborative stack memory to make it available for the higher prioritized task. This means that the stack memory consumption is virtually reduced, leading to the next equation that must be hold to avoid an out-of-stack condition:

$$\forall t \in \mathbb{N}_0, \forall \tau \in T: \quad U(t, \texttt{page\_size}) - K_\tau(t) \leq S \quad (4)$$

Thus, CoStack contributes to the reduction of the totally assigned stack memory S by collaboratively sharing stack memory as shown by comparing the equation (2) with (4).

B. Schedulability Analysis

The freeing of the collaborative stack memory is performed in the respective lower prioritized tasks. Thus, a priority inversion occurs: The lower prioritized task frees the stack memory and blocks the higher prioritized task because the required stack memory is not available. Thus, we are investigating the maximum blocking time B_τ of task τ for the RM schedulability analysis [16]:

$$\frac{C_{\tau_0}}{T_{\tau_0}} + ... + \frac{C_{\tau_{n-1}}}{T_{\tau n-1}} + \max\left(\frac{B_{\tau_0}}{T_{\tau_0}}, ..., \frac{B_{\tau_{n-1}}}{T_{\tau_{n-1}}}\right) \leq n(2^{\frac{1}{n}} - 1) \quad (5)$$

In CoStack, the blocking time compounds on some administrative work and the freeing of the stack memory. Thereby, the time for freeing the stack memory is not constant, because MultiStackMMU is based on pages, which must be released one after another. Therefore, we are defining the collaborate time $t^k_{\tau\prime}(t)$ of task $\tau\prime$ at time t (see Figure 2a), which defines the time required by the collaborative task $\tau\prime$ to perform all the operations to free $\tau\prime$'s collaborative stack memory. Consequently, the blocking time B_τ of task τ can be calculated as follows:

$$B_\tau := \sum_{\forall i: p_{\tau_i} < p_\tau} \max_{\forall t \in \mathbb{N}} \{t^k_{\tau_i}(t)\} \quad (6)$$

The blocking time B_τ is the sum of the longest collaborate time $t^k_{\tau_i}$ of all lower prioritized tasks τ_i. Thus, the blocking time highly depends on the requested stack memory, the collaborative stack memory of each collaborative task, the number of lower prioritized collaborative tasks, and the time when the stack memory is requested.

IV. Implementation Details

We implemented the collaborative stack sharing approach into our *mosart*MCU research platform, running with the *mosart*MCU-OS.

A. mosartMCU

The *mosart*MCU (i.e., Multi-Core Operating-System-Aware Real-Time MCU) project implements OS awareness into embedded multi-core systems. The open RISC-V [17] architecture, maintained by the University of California of Berkeley, is the specification of the softcore *mosart*MCU. The *mosart*MCU is based on the offered open source Verilog implementation vScale. vScale implements all the 32 Bit integers and the multiplication/division instructions [18] and executes the instruction in a three stage pipeline. The specification specifies 32 registers whereas the compiler does not use the register `tp`. That register indicates the TCB, which contains information about the task including its priority, of the currently running task. We extended the basic implementation with an automatic read operation, triggered by the hardware if the register `tp` is changed. Therefore, the hardware is always aware of the currently running task's priority. In parallel to the normal execution, a read operation is automatically performed by using an additional connection to the data memory through a dual-port memory. This dual-port memory is also used by some other extensions (e.g., [19]). Further, the CPU specification defines three different operating modes, whereas the *mosart*MCU supports only the non-privileged *user*-mode and the privileged *kernel*-mode. These operating modes define permissions for some instructions and for accessing CSRs, which are hardware registers used to configure and to get information from the Microcontroller Unit (MCU).

B. mosartMCU-OS

The *mosart*MCU-OS is a real-time OS supporting the OS-awareness extension of the *mosart*MCU. Besides other OS-awareness concepts, it only has to initialize some CSRs (i.e., stack area) and the references of the next free available pages at startup, for supporting MultiStackMMU. After that, the MultiStackMMU operates transparent to the OS. However, the OS must be extended to support our proposed collaborative stack sharing approach.

C. Collaborative Stack Management

In a CSR, the hardware provides the number of required stack pages after a collaborate exception. The exception handler stores the number of required pages into the TCB of the currently running task. After executing the exception handler, the OS does its management work and selects a task according to the RM scheduling policy. Before leaving the OS, by restoring all the task's registers, the OS checks if the scheduled task requires more pages than available. The number of available pages is provided by the hardware through another CSR register. If there are more pages available than required, the OS lefts and resumes with the scheduled task. Otherwise, the OS searches, starting by the lowest prioritized task, a collaborative task that is recognized by the information in the TCB. If a collaborative task is found, the scheduler selects that task for executing. Thereby, the scheduler changes the program counter to continue at the collaborate handler, also stored in the TCB. The collaborate code restores its callee saved registers, frees the collaborate stack memory, and yields itself to return to the OS.

V. Experimental Evaluation

We experimentally evaluated the collaborative stack sharing approach in the *mosart*MCU, running with 50 MHz in a Xilinx Artix-7 FPGA. First, we demonstrate the cooperative stack sharing, measured with an oscilloscope; and second, we show the synthesized results for the FPGA.

A. CoStack Evaluation

This evaluation illustrates CoStack on a collaborative 600 Byte stack memory provided by task $\tau\prime$, once the stack memory is not available for the higher prioritized task τ. Figure 3 depicts the execution flow with the signal *os* showing

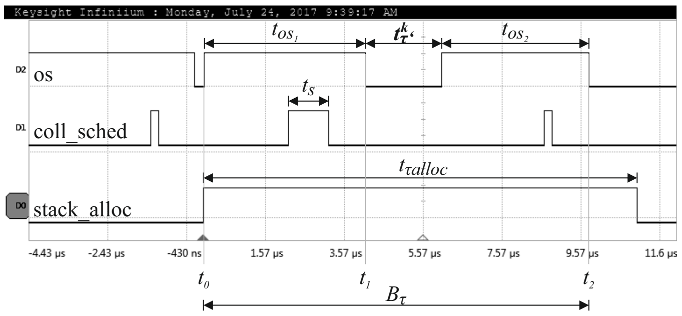


Figure 3. Execution flow example of CoStack.

the execution of the OS, the signal *coll_sched* demonstrating the part in the scheduler that is responsible for searching and scheduling a collaborative task, and the signal *stack_alloc* showing the time for allocating the required 600 Bytes stack memory to task τ. Table I lists all the measured times, and

TABLE I. Measured times for the example in Figure 3.

t_{os_1}	t_s	t_{os_2}	$t^k_{\tau\prime}$	B_τ	$t_{\tau alloc}$
3.74 µs	1.02 µs	4.08 µs	1.94 µs	9.76 µs	11.00 µs

following emphasizes some specific points in time of Figure 3:

- At time t_0, task τ requests 600 Byte stack memory. The memory is not available; therefore, the OS is called by a collaborate exception. There, the OS saves the context of task τ, does its management work including the work for scheduling the collaborative task $\tau\prime$ (i.e., t_s), and restores its context. All this in total consumes t_{os_1}.
- At time t_1, the OS returns, and task $\tau\prime$ is running. Instead of continuing with the previous preempted program counter it continues at the label `COLLABORATE`. There, its callee saved registers are restored, the collaborative stack memory $\kappa_{\tau\prime}$ is released, and the task yields to return to the OS, which again schedules task τ. These operations reflect the collaborate time $t^k_{\tau\prime}$.
- After leaving the OS (i.e., t_{OS_2}) on time t_2, there is enough stack memory available for task τ; therefore, the required stack memory is allocated to task τ.

The blocking time B_τ (i.e., $t^k_{\tau\prime}$ including the OS overheads t_{OS_1} and t_{OS_2}) and the stack allocation time $t_{\tau alloc}$ are not constant, because they depend on the number of collaborative tasks, the location of the task in the searching list, and the number of required and collaborate stack pages. However, the execution is still predictable because MultiStackMMU works predictable. Further, CoStack avoids an out-of-stack, because the collaborative task $\tau\prime$ voluntarily frees its collaborative stack memory $\kappa_{\tau\prime}$ for the higher prioritized task τ. Otherwise, task τ would not be able to execute, because no stack memory is

TABLE II. Resource comparison of the original and the extended *mosart*MCU.

	*mosart*MCU		
	Original	MultiStackMMU	CoStack
LUT slices	2799	4283	4298
FF slices	2078	2378	2378
max. frequency	73.992 MHz	63.339 MHz	63.391 MHz
Dynamic power	16 mW	21 mW	21 mW

available; consequently, this might result into an unpredictable blocking of task τ and to possibly violated real-time constraints.

B. Synthesized Results

The synthesized results, provided by the Xilinx Vivado 2017.3 development toolchain, for the Xilinx Artix-7 FPGA are listed in Table II. It compares Look Up Table (LUT) slices, Flip-Flop (FF) slices, the maximum achievable frequency, and the dynamic power of the original *mosart*MCU, and the extended versions MultiStackMMU and CoStack. If we compare the original *mosart*MCU with the extended versions, it is remarkable that the original version requires about 65 % and 87 % of LUTs and FFs, respectively. The increased resource utilization is caused by the VM address translation and some other registers that are required for handling the StackMMU approach. However, comparing the two extended *mosart*MCU versions, the resource utilization remains negligible the same.

For the maximal achievable frequency, and the dynamic power consumption the table represents a similar behavior. For the former, the large frequency reduction is caused by the implementation of the StackMMU in the already longest path of the original *mosart*MCU. For the latter, the dynamic power increases due to the additional required LUTs and FFs. However, for the two extended *mosart*MCU versions, the maximum achievable frequency remains almost the same and both require the same amount of power.

VI. Conclusion

Memory is a rare and expensive resource in embedded systems. This makes the economical usage of memory important. The stack memory has the potential to optimize the overall memory consumption because its size changes dynamically over the time. The paper proposes CoStack, an extension of the dynamically sharing stack memory concept StackMMU with collaborative stack memory. A task tags a code part with collaborative stack memory and if a higher prioritized task requires stack memory that is not available at that moment, the collaborative task deallocates the collaborative stack memory for enabling the higher prioritized task to continue. Our analysis shows that the whole stack memory requirement is virtually reduced. Further, we showed the impact of CoStack in the schedulability analysis for RM scheduling. The experimental evaluation shows that the synthesized results for the FPGA remain almost constant. Therefore, CoStack contributes to a reduction of the memory usage by introducing a collaborative stack sharing mechanism and remains predictable as aimed for real-time systems.

Acknowledgment

This research project was partially funded by AVL List GmbH and the Austrian Federal Ministry of Sciences, Research and Economy (bmwfw).

References

[1] "Micrium uC/OS-III," URL: https://www.micrium.com/rtos/ [accessed: 2018-02-27].

[2] "The FreeRTOS Kernel," URL: http://www.freertos.org/ [accessed: 2018-02-27].

[3] "Contiki: The Open Source OS for the Internet of Things," URL: http://www.contiki-os.org [accessed: 2018-02-27].

[4] T. P. Baker, "A stack-based resource allocation policy for realtime processes," in Proc. of the 11th IEEE Real-Time Systems Symposium (RTSS), Dec 1990, pp. 191–200.

[5] B. Middha, M. Simpson, and R. Barua, "MTSS: Multitask Stack Sharing for Embedded Systems," ACM Trans. on Embedded Computer Systems, vol. 7, 2008, pp. 41:1–46:37, ISSN: 0001-0782.

[6] J. Fotheringham, "Dynamic Storage Allocation in the Atlas Computer, Including an Automatic Use of a Backing Store," Communications of the ACM, vol. 4, 1961, pp. 435–436, ISSN: 0001-0782.

[7] F. Mauroner and M. Baunach, "StackMMU: Dynamic Stack Sharing for Embedded Systems," in Proc. of the 22nd IEEE Int. Conference on Emerging Technologies and Factory Automation (ETFA), Sep 2017, pp. 1–9.

[8] Y. Wang and M. Saksena, "Scheduling Fixed-Priority Tasks with Preemption Threshold," in Proc. of the 6th Int. Conference on Real-Time Computing Systems and Applications (RTCSA), 1999, pp. 328–335.

[9] "ThreadX," 2018, URL: https://rtos.com/solutions/threadx/real-time-operating-system/ [accessed: 2018-02-27].

[10] R. Chu, L. Gu, Y. Liu, M. Li, and X. Lu, "SenSmart: Adaptive Stack Management for Multitasking Sensor Networks," IEEE Trans. on Computers, vol. 62, 2013, pp. 137–150, ISSN: 0018-9340.

[11] S. Yi, S. Lee, Y. Cho, and J. Hong, OTL: On-Demand Thread Stack Allocation Scheme for Real-Time Sensor Operating Systems. Springer Berlin Heidelberg, 2007, pp. 905–912, in Computational Science – ICCS 2007, ISBN: 978-3-540-72590-9.

[12] E. A. Hauck and B. A. Dent, "Burroughs' B6500/B7500 Stack Mechanism," in Proc. of the Spring Joint Computer Conference, ser. AFIPS '68 (Spring), 1968, pp. 245–251.

[13] D. Grunwald and R. Neves, "Whole-program Optimization for Time and Space Efficient Threads," in Proc. of the 7th Int. Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ser. ASPLOS VII, 1996, pp. 50–59.

[14] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable Threads for Internet Services," in Proc. of the 9th ACM Symposium on Operating Systems Principles (SOSP), ser. SOSP '03, 2003, pp. 268–281.

[15] M. Baunach, "CoMem: collaborative memory management for real-time operation within reactive sensor/actor networks," Trans. on Real-Time Systems, vol. 48, 2012, pp. 75–100, ISSN: 1573-1383.

[16] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," IEEE Trans. on Computers, vol. 39, Sep 1990, pp. 1175–1185, ISSN: 0018-9340.

[17] "RISC-V," URL: https://riscv.org/ [accessed: 2018-02-27].

[18] A. Waterman, Y. Lee, R. Avizienis, D. Patterson, and K. Asanovic, The RISC-V Instruction Set Manual, 2016, URL: https://riscv.org/specifications/ [accessed: 2018-02-27].

[19] F. Mauroner and M. Baunach, "EventIRQ: An Event based and Priority aware IRQ handling for Multi-Tasking Environments," in Proc. of the 20th Euromicro Conference on Digital System Design (DSD), Aug 2017, pp. 102–110.