Memory allocation for low-power real-time embedded microcontroller: a case study

Zhishen Zhang, Yuwen Shen, Binqi Sun, Tomasz Kloda, Marco Caccamo
Technical University of Munich, Germany

***Abstract*—Memory allocation of instructions and data can affect the program execution speed. This paper tests various memory-intensive benchmarks under different memory allocations on a Cortex-M4-based microcontroller and solves the allocation problem using integer linear programming.**

***Index Terms*—Cortex-M, core-coupled memory, real-time**

I. Introduction

Low-power microcontrollers, by design, limit the energy-draining memory features. Typically, in many such devices, when a performance boost is needed for time-critical routines, only little core-coupled memories can be found. In this paper, we report our experience in harnessing core-coupled and standard memory components to improve processor utilization on the example of a popular embedded microcontroller.

II. Microcontroller

The STMicroelectronics' `STM32F3 DISCOVERY` (`STM32F303VC`) microcontroller is based on the Arm Cortex-M4 32-bit RISC core operating at a frequency of up to 72 MHz and incorporates three embedded memories: 256 KB of *Flash* memory, 40 KB of *SRAM*, and 8 KB of *CCM-RAM* (core-coupled memory RAM). The *CCM-RAM* and *SRAM* can be accessed in read/write at CPU clock speed (*i.e.,* zero wait-state). The *CCM-RAM* is typically used to speed up computation-intensive routines by executing their code at the maximum CPU clock frequency. This brings a significant decrease in the execution time compared to code executed from the *Flash* memory, where each instruction fetch has to be adjusted with additional wait-states with respect to the CPU clock frequency: zero up to 24 MHz, one from 24 to 48 MHz, and two wait-states from 48 to 72 MHz. Additionally, placing data in *SRAM* and instructions in *CCM-RAM* allows fetching instructions while data is being loaded or stored since they are accessed using separate buses (*i.e.,* Harvard configuration, see Figure 1). On the contrary, placing both data and instructions in *CCM-RAM* can lead to a bus contention as the *CCM-RAM* interface is shared between data and instructions. In this regard, the *Flash* memory with two interfaces, I-code and D-code, for reading instructions and data can avoid bus contention. However, as explained above, the *Flash* memory incurs wait-state penalties for higher CPU clock frequencies. Unlike the *SRAM* and *Flash*, the *CCM-RAM* has no DMA access on this device [1].

Marco Caccamo was supported by an Alexander von Humboldt Professorship endowed by the German Federal Ministry of Education and Research.

Zhishen Zhang and Yuwen Shen contributed equally to this work.

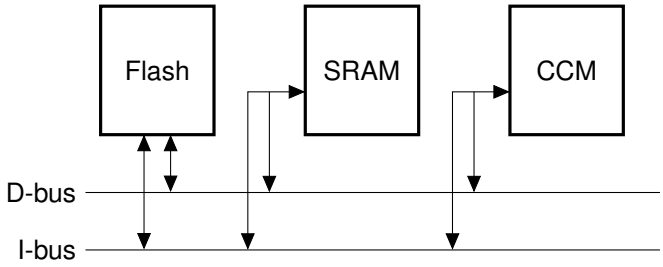

Fig. 1. STM32F3 device bus matrix architecture simplified view.

III. Benchmarks

We use a set of benchmarks representative of real-world embedded applications to evaluate the performance of different memory allocations. Benchmarks are derived from the *MiBench* [2], *TACLe* [3], and *Mälardalen* [4] test suites or developed for the purpose of this paper. Due to the microcontroller memory size limitation, lack of the underlying operating system and libraries, the benchmarks were ported to bare metal[1]. We specify code, inputs, and read-only (*ro*) data size.

- *Pointer Chase*: traversing a linked list, using linear and random access patterns, code: 62 B, no inputs, *ro*: 8 KB,
- *Memory Copy*: copying bytes from one to another memory area, code: 68 B, inputs: 0.4-8 KB, no *ro*,
- *Sine Lookup Table*: finding the sine value through array indexing, code: 228 B, inputs: 0.4-8 KB, *ro*: 5.6 KB,
- *Bubble* and *Quick Sort*: sorting algorithms, code: 172 B and 1.28 KB, inputs: 0.8-4 KB and 2-3.5 KB, no *ro*,
- *Binary Search*: finding the position of a given value within a sorted array, code: 128 B, inputs: 4 B, *ro*: 4 KB,
- *Fast Fourier Transform*: computing the discrete Fourier transform, code: 1 KB, inputs 0-4 KB, no *ro*,
- *ADPCM-Encode* and *Decode*: analog signal compression and decompression, code: 3.08 and 2.75 KB, inputs: 0-0.8 KB and 0-0.4 KB, *ro*: 1 KB,
- *RSA-Encryption* and *Decryption*: key encrypting and decrypting algorithm, code: 145 and 172 B, inputs: 0.1-0.5 KB and 0.8-4 KB, no *ro*,
- *Kalman Filter*: estimating unknown variables based on a series of uncertain measurements, code: 1.62 KB, inputs: 0.4-8 KB, no *ro*.

The stack usage is within the range of 16 to 208 B.

[1]Available at: https://github.com/ZhishenZ/Cortex-M4-Benchmarks

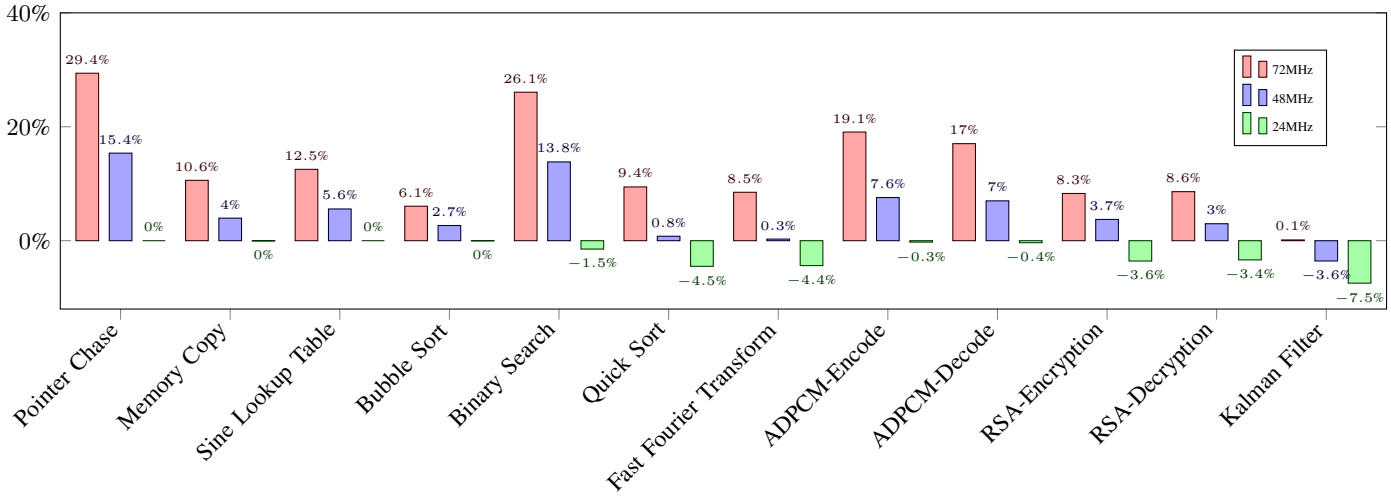

Fig. 2. Execution time decrease by moving instructions from the *Flash* to the *CCM-RAM* (read-only data in *Flash*).

IV. Evaluation

We run a set of experiments to evaluate the impact on the execution time of: i) placing the code into *Flash*/*CCM-RAM* and ii) placing the read-only data into *Flash*/*SRAM*/*CCM-RAM*. We perform the experiments for each benchmark under the CPU clock frequencies of 24, 48, and 72 MHz. We denote by $wcet^{p,d}$ the measured execution time with instructions in $p \in \{\boldsymbol{F}lash, \boldsymbol{C}CM{-}RAM\}$ and read-only data in $d \in \{\boldsymbol{F}lash, \boldsymbol{S}RAM, \boldsymbol{C}CM{-}RAM\}$. Jitter in the measured execution times is negligible.

TABLE I
Base execution times $wcet^{F,F}$ (ms) at 72 / 48 / 24 MHz.

Pointer Chase	209.73	240.68	370.76	FFT	3.85	5.04	9.16
Memory Copy	0.24	0.33	0.64	ADPCM-Enc.	6.08	7.88	14.37
Sine Lookup	180.99	244.32	448.17	ADPCM-Dec.	4.28	5.66	10.39
Bubble Sort	177.49	257.03	500.77	RSA-Enc.	5.63	7.20	12.28
Binary Search	0.00586	0.00708	0.01133	RSA-Dec.	46.31	58.99	102.15
Quick Sort	10.44	14.08	25.78	Kalman Filter	66.75	80.56	139.09

A. Code segment memory allocation

We first measure the execution time of each benchmark when its code is run from *Flash* $wcet^{\boldsymbol{F},F}$ (see Table I) and then when its code is run from *CCM-RAM* $wcet^{\boldsymbol{C},F}$. The read-only data are stored in the *Flash* in both cases, and both measurements are taken under the same CPU clock frequency. We calculate the relative decrease in execution time as $(wcet^{\boldsymbol{F},F} - wcet^{\boldsymbol{C},F})/wcet^{\boldsymbol{F},F}$. Figure 2 shows the relative decrease in execution times for all benchmarks under different CPU clock speeds. For the highest frequency (72 MHz), the execution time can be reduced by more than 25% for the most memory-intensive benchmarks (*e.g.,* Pointer Chase or Binary Search). For the lowest frequency (24 MHz), placing the instructions into the *CCM-RAM* can have a detrimental effect on the execution times (*e.g.,* Quick Sort or Kalman Filter). Indeed, at this frequency, while instruction fetches have the same latency for both memories, the *Flash*, in contrast to the *CCM-RAM*, can pipeline the instructions and data (*e.g.,* literals or constants stored in the code section).

B. Read-only data memory allocation

In this experiment, we evaluate the impact of the read-only data (*e.g., lookup tables*) allocation on the execution time.

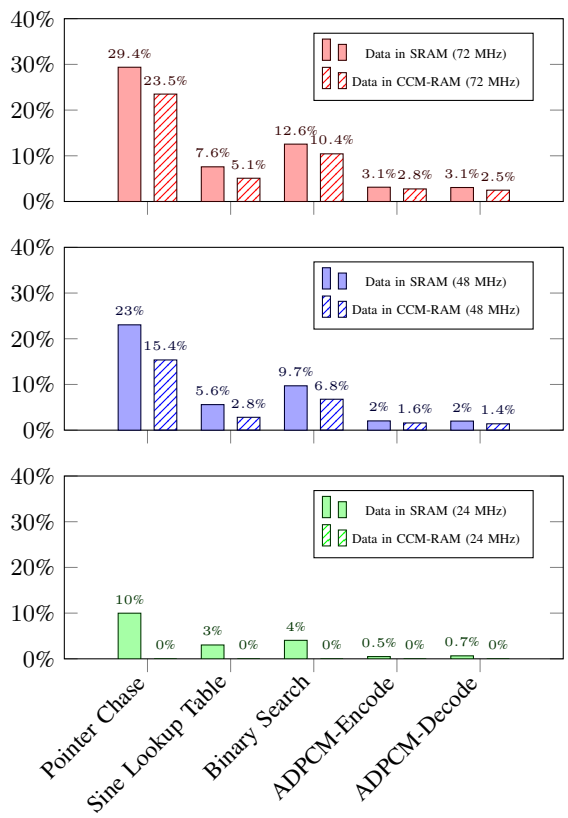

Fig. 3. Execution time decrease by moving read-only data from the *Flash* to the *SRAM* or *CCM-RAM* (instructions in *Flash*).

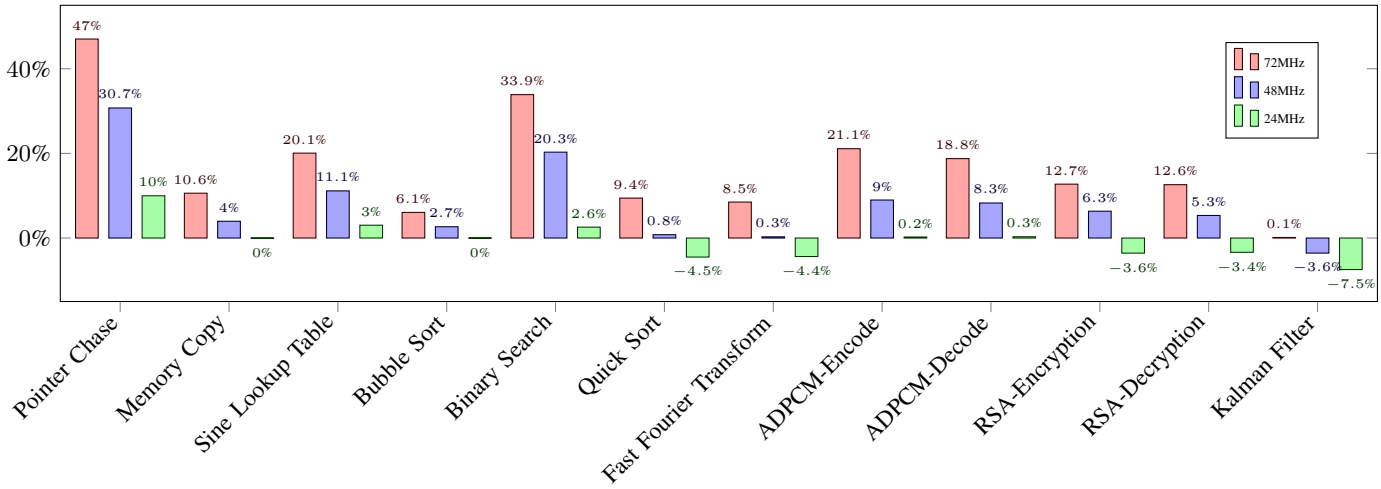

Fig. 4. Execution time decrease by moving instructions from the *Flash* to the *CCM-RAM* and read-only data from the *Flash* to the *SRAM*.

For each benchmark with a sufficiently large read-only data segment, we measure its execution times $wcet^{F,\boldsymbol{F}}$, $wcet^{F,\boldsymbol{S}}$, $wcet^{F,\boldsymbol{C}}$ when such a segment is allocated respectively to: i) *Flash*, ii) *SRAM*, and iii) *CCM-RAM*. The instructions are always stored in the *Flash*, and all three measurements are taken under the same CPU clock frequency. We calculate the relative decrease in execution time as $(wcet^{F,\boldsymbol{F}} - wcet^{F,\boldsymbol{C}})/wcet^{F,\boldsymbol{F}}$ for *CCM-RAM* and $(wcet^{F,\boldsymbol{F}} - wcet^{F,\boldsymbol{S}})/wcet^{F,\boldsymbol{F}}$ for *SRAM*. Figure 3 shows the relative decreases in execution times for all benchmarks under different CPU clock speeds. The maximum achievable speedup is 29.4% for *SRAM* and 23.5% for *CCM-RAM*. In general, allocating read-only data in *SRAM* is faster than in *CCM-RAM*.

C. Best allocation

We now combine the best allocation for instructions (*i.e., CCM-RAM*) and read-only data (*i.e., SRAM*). For each benchmark, we measure its execution time $wcet^{\boldsymbol{C},\boldsymbol{S}}$ when run from the *CCM-RAM* with read-only data stored in the *SRAM* and compare it with the base case $wcet^{\boldsymbol{F},\boldsymbol{F}}$ when both instructions and read-only data are in the *Flash*. We compute the relative decrease in execution time as $(wcet^{\boldsymbol{F},\boldsymbol{F}} - wcet^{\boldsymbol{C},\boldsymbol{S}})/wcet^{\boldsymbol{F},\boldsymbol{F}}$ and show the results in Figure 4. The boost can reach up to 47% at 72 MHz.

V. A Case Study in Real-time Scheduling

A. Problem formulation

We consider a task set τ consisting of n periodic benchmark tasks running on the studied microcontroller. Each task $\tau_i \in \tau$ has a period T_i and different worst-case execution times $wcet_i^{p,d}$ depending on different program and read-only data allocations (*i.e.,* $p \in \{F, C\}, d \in \{F, C, S\}$)[2]. The utilization of a task is defined as $U_i^{p,d} = wcet_i^{p,d}/T_i$, and the utilization of the task set is calculated as $\sum_{i=1}^{n} U_i^{p,d}$. Particularly, we denote base utilization as $U = \sum_{i=1}^{n} U_i^{F,F}$, where all tasks' program and read-only data are allocated to *Flash*. Our goal is to minimize the task set utilization by determining the optimal allocation of each task's program and read-only data under memory capacity constraints.

To this end, a 0-1 integer linear programming (ILP) model is developed as follows with the notations given in Table II.

TABLE II
Table of Notations.

Notation	Implication
Decision variables	
$x_i^{p,d}$	1, if task τ_i's program is allocated to p and task τ_i's data is allocated to d; 0, otherwise; $p \in \{F, C\}, d \in \{F, C, S\}$
Problem data	
$U_i^{p,d}$	utilization of task τ_i when allocating program to p and read-only data to d; $p \in \{F, C\}, d \in \{F, C, S\}$
m_i^P, m_i^D, m_i^I	size of task τ_i's program, read-only data, and inputs;
M^F, M^C, M^S	capacity of *Flash*, *CCM-RAM* and *SRAM*.

$$\text{min.} \quad \sum_{i=1}^{n} \sum_{\forall p, \forall d} U_i^{p,d} \cdot x_i^{p,d} \tag{1}$$

$$\text{s.t.} \quad \sum_{i=1}^{n} \left(\sum_{\forall d} m_i^P x_i^{\diamond,d} + \sum_{\forall p} m_i^D x_i^{p,\diamond} \right) \leq M^{\diamond}, \ \diamond \in \{F, C\} \tag{2}$$

$$\sum_{i=1}^{n} \left(m_i^I + \sum_{\forall p} m_i^D x_i^{p,S} \right) \leq M^S \tag{3}$$

$$\sum_{\forall p, \forall d} x_i^{p,d} = 1, \quad \forall i \tag{4}$$

$$x_i^{p,d} \in \{0, 1\}, \quad \forall i, p, d \tag{5}$$

Objective (1) minimizes the utilization of the task set. Constraints (2) and (3) ensure the capacity limits of *Flash*, *CCM-RAM*, and *SRAM*. Constraints (4) guarantee each task's

[2]Task inputs are assumed to be allocated to *SRAM* statically.

program or read-only data is allocated to only one memory type. Constraints (5) indicate binary decision variables.

B. Numerical study

We evaluate the developed ILP model on randomly generated benchmark task sets with three varied parameters: (1) number of tasks $n \in \{8, 16, 32, 64\}$, (2) base utilization $U \in [0.1, 1.5]$ with step 0.1, (3) CPU clock frequency $f \in \{24\,\text{MHz}, 48\,\text{MHz}, 72\,\text{MHz}\}$. For each combination of the above parameters, we generate 100 random task sets. The generation of each task set follows three steps. First, we sample n tasks from the benchmark set under clock frequency f, and take task parameters $(wcet_i^{p,d}, m_i^P, m_i^D, m_i^I)$ from the sampled benchmarks. Second, we use *UUnifast* [5] to generate the base utilization $U_i^{F,F}$ for each task τ_i such that the total base utilization equals U. Finally, we calculate the period of each task by $T_i = wcet^{F,F}/U_i^{F,F}$.

For each generated task set, we solve the ILP model using a state-of-the-art mathematical programming solver Gurobi 9.5.1 [6] on a server equipped with AMD EPYC 7763 CPU. After obtaining the optimal solution of the model, we calculate the optimal objective value U^* and the utilization reduction ratio $\beta^U = (U - U^*)/U$. For each combination of n and f, we show the average utilization reduction ratio over all tested base utilization in Figure 5.

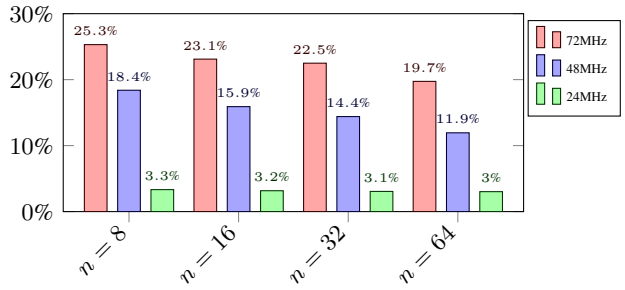

Fig. 5. Average utilization reduction ratio with optimal memory allocation.

From Figure 5, it can be seen that with optimal memory allocation we can achieve an average utilization reduction ratio of 25.3% when $n = 8, f = 72\,\text{MHz}$. This result is useful for real-time system design. Suppose we have a task set with base utilization $U = 1.2$, which is not schedulable under any scheduling policy. However, if we apply the optimal memory allocation, the task set becomes schedulable by the *Earliest Deadline First* since $U^* = 0.896$ is smaller than 1 [7]. Besides, the figure shows a trend that the utilization reduction ratio decreases with the increase of task number. This is as expected since a task set with more tasks tends to have a larger memory consumption, however, the memory capacity in the system is fixed in our experiments.

The running time of our ILP solver ranges from 1.5 ms to 20 ms as n ranges from 8 to 64.

VI. Related Work

Altmeyer et al. [8] proposed a branch-and-bound-based method for optimal cache partitioning for real-time tasks. The same search technique can be applied to the problem presented in this paper by considering more than one type of memory.

Several techniques in static analysis [9], [10] can be used to identify the program's sections resulting in a better reduction of its execution times if locked in faster memory. Moreover, dynamic locking approaches [11], [12] load the selected sections at run-time depending on the program's execution path.

If stack usage is considerably higher than reported in our benchmarks, the non-preemptive or limited-preemption approaches might be applied [13].

VII. Conclusions and Future Work

We presented a method for static memory allocation of data and instructions on low-power embedded microcontrollers.

This work can be extended by using *DMA* to load read-only data and instructions from the *Flash* to the *SRAM* and the *CCM-RAM*, respectively (only certain `STM32F3` devices have DMA support for *CCM-RAM* [1]). We also plan to consider a more fine-grained allocation with multiple code sections.

References

[1] "Use STM32F3/STM32G4 CCM SRAM with IAR Embedded Workbench®, Keil® MDK-ARM, STMicroelectronics STM32CubeIDE and other GNU-based toolchains," Application note, accessed: 2022-06-04.

[2] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the 4th Annual IEEE International Workshop on Workload Characterization. WWC-4*, 2001, pp. 3–14.

[3] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, ser. OpenAccess Series in Informatics (OASIcs), vol. 55, 2016, pp. 2:1–2:10.

[4] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future," *OpenAccess Series in Informatics*, vol. 15, pp. 136–146, 2010.

[5] E. Bini and G. C. Buttazzo, "Measuring the performance of schedulability tests," *Real-Time Systems*, vol. 30, no. 1, pp. 129–154, 2005.

[6] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2022. [Online]. Available: https://www.gurobi.com

[7] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *J. ACM*, vol. 20, no. 1, p. 46–61, Jan 1973.

[8] S. Altmeyer, R. Douma, W. Lunniss, and R. I. Davis, "On the Effectiveness of Cache Partitioning in Hard Real-Time Systems," *Real-Time Syst.*, vol. 52, no. 5, p. 598–643, Sep 2016.

[9] H. Falk and J. C. Kleinsorge, "Optimal static WCET-aware scratchpad allocation of program code," in *2009 46th ACM/IEEE Design Automation Conference*, 2009, pp. 732–737.

[10] H. Falk, S. Plazar, and H. Theiling, "Compile-time decided instruction cache locking using worst-case execution paths," in *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007, pp. 143–148.

[11] H. Ding, Y. Liang, and T. Mitra, "WCET-Centric dynamic instruction cache locking," in *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2014, pp. 1–6.

[12] J.-F. Deverge and I. Puaut, "WCET-Directed Dynamic Scratchpad Memory Allocation of Data," in *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, 2007, pp. 179–190.

[13] H. Zeng, M. Di Natale, and Q. Zhu, "Optimizing stack memory requirements for real-time embedded applications," in *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*, 2012, pp. 1–8.