# End-to-end Latency Analysis of Cause-effect Chains in an Engine Management System

Junchul Choi, Donghyun Kang, and Soonhoi Ha[1]
Department of Computer Science and Engineering, Seoul National University, Seoul, Korea,
Email: {hinomk2, kangdongh, sha}@iris.snu.ac.kr

***Abstract*—An engine management system consists of periodic or sporadic real-time tasks. A task is a set of runnables that may be fully preemptive or partially at runnable boundaries. A cause-effect chain is defined as a chain of runnables that are connected by the read/write dependency. We propose a novel analytical technique to estimate the end-to-end latency of a cause-effect chain by considering conservatively estimated schedule time bounds of associated runnables. The proposed approach is verified with an industrial-strength automotive benchmark.**

***Index Terms*—engine management system, real-time systems, end-to-end latency, cause-effect chain, schedule time bound**

## I. Introduction

The complexity of modern automotive systems has drastically grown due to the introduction of multi-core execution platform and a large quantity of innovative functionalities such as powertrain systems, chassis control applications, and Advanced Driver Assistance Systems (ADAS). In the design of automotive systems, real-time scheduling analysis is an important step in order to fulfill strict timing constraints of safety-relevant automotive applications. End-to-end response times from sensors to actuators must be bounded, because too long response time may cause a fatal accident.

A popular approach in the industry for verification of the real-time embedded systems is to use simulation tools such as Timing Architects [1] or Trace Analyzer [2]. Since a simulation-based approach does not consider all possible scenarios, it usually under-estimates the worst-case latency. A model checking approach [3] accurately estimates the worst-case response time (WCRT), however, it is not scalable due to exponential time complexity. Recently compositional approaches such as SymTA/S [4] have been developed to overcome the scalability problem by performing the analysis in a modular manner. However, they tend to make over-estimation, especially for distributed systems that run dependent tasks.

Due to the divergence between academic research and industrial practice, industries still have difficulties in finding an approach that can cope with the complexity of the dynamic behavior in their software systems. To cope with this difficulty, Bosch GmbH recently initiated a challenge for an industry-strength benchmark, a full-blown performance model of a modern engine management system [5]. In this paper, we tackle the timing analysis problem that tightly estimates the upper bound of the end-to-end latencies of cause-effect chains in the engine management system benchmark, where a cause-effect chain is an end-to-end process flow from a sensor to an actuator. To this end, the proposed approach computes the scheduling time bounds of each task considering the task dependency, which is inspired by the scheduling time bound analysis (STBA) [6]. Then the end-to-end latency of a cause-effect chain is estimated by considering the schedule time bounds. To estimate the memory contention delay during the latency, we compute the worst-case resource demand from each processing element based on the event stream model similarly to [7] and [8], but we improve the estimation accuracy by accounting for the scheduling pattern of tasks.

For the automotive benchmark addressed in this paper, an approach based on MAST [9] is presented in [10]. Another approach based on model-checking is presented in [11]. Since the system model of both approaches is not able to express the system behavior of the benchmark fully, both approaches approximate some timing behaviors conservatively, which incurs over-estimation in the result. Between them, we compare the proposed technique with the approach [10] in the experiment.

## II. Engine management system benchmark

In the benchmark [5], the system contains a simplified microcontroller architecture with four symmetric cores. Each core $A_i^C$ has its own local memory $A_i^L$. A crossbar network is used for the interconnection among cores and a global memory $A^G$. The system-wide frequency is 200 MHz.

A task $\tau_i$ is a basic mapping unit onto a core, and task-to-core mapping is fixed and given. The core $\tau_i$ is mapped to is denoted by $m_i$. A task is invoked either periodically or sporadically. $I_P$ and $I_S$ denote a set of periodic tasks and a set of sporadic tasks, respectively. The minimum and the maximum initiation interval for each task $\tau_i$ are denoted as $p_i^l$ and $p_i^u$. Then $p_i^l = p_i^u$ if $\tau_i \in I_P$. All tasks are simultaneously initiated at the system activation time. The basic timing requirement for task $\tau_i$ is to finish execution before its deadline denoted by $d_i$ and $d_i$ is equal to $p_i^l$.

A distinct priority is assigned to each task by giving a unique index in the descending priority order; task $\tau_i$ has a higher priority than $\tau_j$ if $i < j$. A task $\tau_i$ is scheduled by either preemptive or cooperative fixed priority scheduling policy. $S_P$ and $S_C$ denote a set of preemptive tasks and a set of cooperative tasks, respectively. A task $\tau_i \in S_P$ can preempt lower priority tasks at any time, whereas a task $\tau_i \in S_C$ can preempt lower priority cooperative tasks at the boundary of runnable executions [12].

A task $\tau_i$ consists of a set of runnables $\{r_{i,j} \mid 1 \leq j \leq |\tau_i|\}$ where runnable $r_{i,j}$ is a unit of execution and $|\tau_i|$ means

[1]Corresponding author

978-3-9819263-0-9/DATE18/©2018 EDAA

the number of runnables in the task. Runnables in a task are executed sequentially on the mapped core in the ascending index order. The lower and the upper bound of the execution time of $r_{i,j}$, denoted $c^l_{i,j}$ and $c^u_{i,j}$, are specified assuming that code is executed directly from core-exclusive flashes without contention. Note that memory access delay is not included in the execution times. The runnables are assumed to read all required data at the beginning of their execution and write back the results after execution is completed. We assume that when a runnable attempts to access a memory, no preemption is allowed until the resource request is processed.

A cause-effect chain $CEC_i$ defines a chain of runnables that are connected by the read/write dependency with labels. Note that there are no cyclic dependencies between tasks within a cause-effect chain. Due to the potential different task periods, data may get lost (undersampling) or get duplicated (oversampling). An end-to-end(E2E) latency of a cause-effect chain is defined as the maximum time duration between the first input that may be undersampled and the first output generated from the corresponding or later input. This semantic is same as the *reaction time constraint* of the AUTOSAR [13].

Data is specified by a set of labels and all labels are assumed to be stored in the global memory. Read and write accesses have symmetric memory access times. When accessing the global memory, crossbar transfer takes 8 cycles and access to global memory takes 1 cycle. The accesses are assumed to be arbitrated according to the FIFO policy in the global memory.

We aim to conservatively estimate the upper bound of the response time of each task $\tau_i$, denoted as $L_{\tau_i}$, and end-to-end latency of cause-effect chain $CEC_i$, denoted as $L_{CEC_i}$, as tightly as possible.

## III. Proposed Timing Analysis Technique

### A. Schedule Time Bound Analysis

In this subsection, we derive two schedule time bounds $LB^s_c(i,j)$ and $UB^f_c(i,j)$ which are the lower and the upper bound of the latency between the release time of a runnable $r_{c,i}$ to the start time and the finish time of a runnable $r_{c,j}$, respectively, where $1 \le i \le j \le |\tau_c|$.

**Start time lower bound:** For $LB^s_c(i,j)$ computation, we consider the minimum interference from the other runnables. Assuming that no task is blocked by a low priority task, $LB^s(r_{c,i}, r_{c,j})$ can be formulated as follows:

$$LB^s_c(i,j) = \sum_{k=i}^{j-1} c^l_{c,k} + \sum_{\tau_h \in hp(\tau_c)} \left\lceil \frac{\delta_h}{p^u_h} \right\rceil \cdot C^l_h \tag{1}$$

where $hp(\tau_c) = \{\tau_h | m_h = m_c, c > h\}$, $C^l_h = \sum_{k=1}^{|\tau_h|} c^l_{h,k}$, and $\delta_h = \max(0, LB^s_c(i,j) - (p^u_h - C^l_h) + 1)$.

**Finish time upper bound:** For $UB^f_c(i,j)$ computation, we have to compute the maximum interference among tasks. We formulate $UB^f_c(i,j)$ for a preemptive task $\tau_c \in S_P$ as follows:

$$UB^f_c(i,j) = \sum_{k=i}^{j} c^u_{c,k} + \sum_{\tau_h \in hp(\tau_c)} \left\lceil \frac{UB^f_c(i,j)}{p^l_h} \right\rceil \cdot C^u_h \tag{2}$$

where $C^u_h = \sum_{k=1}^{|\tau_h|} c^u_{h,k}$. For a cooperative task $\tau_c$, the release of $\tau_c$ can be blocked by at most one runnable execution of a lower priority task mapped on the same core. Higher priority cooperative tasks released after the start time of a runnable $r_{c,j}$ do not affect on the finish time. We need to formulate the upper bound of the latency between the release time of a runnable $r_{c,i}$ to the start time of a runnable $r_{c,j}$, denoted $UB^s_c(i,j)$ as follows:

$$UB^s_c(i,j) = B_c + \sum_{k=i}^{j-1} c^u_{c,k} + \sum_{\tau_h \in hp(\tau_c)} \left\lceil \frac{UB^s_c(i,j)+1}{p^l_h} \right\rceil \cdot C^u_h \tag{3}$$

The first term $B_c$ indicates the maximum blocking from a lower priority task. $B_c = \max_{\tau_{l,k} \in \cup lp(\tau_c)} c^u_{l,k}$ where $lp(\tau_c) = \{\tau_l | m_l = m_c, c < l, \tau_l \in S_C\}$, only when $i = 1$ since any lower priority task cannot start after the first runnable starts. If $i > 1$, $B_c = 0$. Note that $UB^s_c(i,j)+1$ is used in the third term to include the higher priority tasks released between the finish of the $(j-1)$-th runnable and the start of the $(j)$-th runnable. Then $UB^f_c(i,j)$ can be estimated as follows:

$$\begin{aligned} UB^f_c(i,j) =& B_c + \sum_{k=i}^{j} c^u_{c,k} + \sum_{\tau_h \in hp(\tau_c) \cap S_P} \left\lceil \frac{UB^f_c(i,j)}{p^l_h} \right\rceil \cdot C^u_h \\ &+ \sum_{\tau_h \in hp(\tau_c) \cap S_C} \left\lceil \frac{UB^s_c(i,j)}{p^l_h} \right\rceil \cdot C^u_h \end{aligned} \tag{4}$$

All requests of higher priority preemptive tasks within $UB^f_c(i,j)$ are accounted while the requests of higher priority cooperative tasks after $r_{c,j}$ starts are excluded.

Finally, we can compute the estimated end-to-end latency of a task $\tau_c$ as $L_{\tau_c} = UB^f_c(1, |\tau_i|)$.

### B. Memory Contention Delay Analysis

Since memory accesses are arbitrated according to the FIFO policy and a core is assumed to be blocked during memory access, a naive way to estimate the worst-case access delay is to assume that every access experiences blocking of maximum queued accesses from all other cores (one access per each core). To find a tighter bound of memory access delay, however, we analyze the maximum number of memory accesses issued by tasks in each core within any time window of size $\Delta t$ by adopting the event stream model [14]. For the detailed explanation and equations, please refer to [14].

For a given time window of size $\Delta t$, we compute the maximum possible memory access requests from each core. For each core, we have to find out a task execution scenario that produces the maximum memory access requests within the time window. To this end, we test all time distributions of the time window to tasks running in the core. For each distribution, we can compute the overall memory demand in the time window by summing the maximum memory demand from each task when it executes up to the distributed amount of time. By computing the lower and upper bound of *net* execution time that a task may take within a time window $\Delta t$, we can eliminate the infeasible time distribution.

We define two functions $t_i^{min}(\Delta t)$ and $t_i^{max}(\Delta t)$ that represent the minimum and the maximum execution time amount a task $\tau_i$ may take within a time window $\Delta t$, respectively. To maximize the number of accesses within a time window, we assume task execution scenarios that a task is invoked with the minimum execution time $\mathcal{C}_i^l$ and the minimum initiation interval $p_i^l$. The minimum *net* execution time $t_i^{min}(\Delta t)$ is found when the time window starts immediately after the earliest finish time of the first instance and the start times of all subsequent instances are maximally delayed by $L_{\tau_i} - \mathcal{C}_i^l$. On the contrary, the amount of execution time is maximized in time window $\Delta t$ when an instance starts as late as possible to finish at its end-to-end latency $L_{\tau_i}$ and subsequent instances start immediately at their request time.

For a tight estimation of resource demand bound, we additionally consider how many instances may exist in a time window $\Delta t$. The authors in [14] found that in case the number of task instances laid in the time window is maximized, the *net* execution time may not reach up to the maximum net execution time $t_i^{max}(\Delta t)$. Hence we compare two cases to find the maximum possible resource demand; (1) the case the number of task instances is maximized and (2) the case the net execution time is maximized. We denote the maximum number of $\tau_i$ instances laid in the time window $\Delta t$ as $\overrightarrow{n_i}(\Delta t)$, the maximum *net* execution time when the number of task instances is maximized as $\overrightarrow{t_i^{max}}(\Delta t)$, and the number of instances when the net execution time is maximized as $n_i(\Delta t)$. Then, we formulate memory access bound function $D_{A_i^C,A^G}(\Delta t)$ which finds the maximum number of accesses from a core $A_i^C$ to global memory $A^G$ within any time window of size $\Delta t$. $D_{A_i^C,A^G}(\Delta t)$ is formulated as follows:

$$D_{A_i^C,A^G}(\Delta t) = \tag{5}$$
$$\max\left\{ \sum_{m_k=A_i^C} D_{\tau_k,A^G}(t_k,\Delta t) \;\middle|\; \begin{array}{c} \sum_{m_k=A_i^C} t_k = \Delta t, \\ \forall_{m_k=A_i^C} t_k \geq t_k^{min}(\Delta t) \end{array} \right\}$$

$$D_{\tau_k,A^G}(t_k,\Delta t) = \max\begin{pmatrix} \eta_{k,A^G}^{e(n_k(\Delta t))}(\min(t_k, t_k^{max}(\Delta t))) \\ \eta_{k,A^G}^{e(\overrightarrow{n_k}(\Delta t))}(\min(t_k, \overrightarrow{t_k^{max}}(\Delta t))) \end{pmatrix} \tag{6}$$

where $t_k$ is the partitioned time to task $\tau_k$ between $t_k^{min}(\Delta t)$ and $t_k^{max}(\Delta t)$, and $\eta_{k,A^G}^n(t)$ is the maximum number of resource accesses that may be issued from $n$ instances of a task $\tau_k$ to a memory $A^G$ when the net execution time of $\tau_i$ does not exceed $t$ time units. $D_{A_i^C,A^G}(\Delta t)$ is used to bound the arbitration delay during the latency computed by equations (2), (3), and (4).

### C. End-to-end latency of a cause-effect chain

For brevity, we define two variables $BCST(r_{c,i}) = LB_c^s(1,i)$ and $WCFT(r_{c,i}) = UB_c^f(1,i)$ which mean a lower bound of start time of $r_{c,i}$ and an upper bound of finish time of $r_{c,i}$, respectively.

Fig. 1 shows three example cause-effect chains and the activation patterns of five tasks are summarized in Fig. 1 (a).

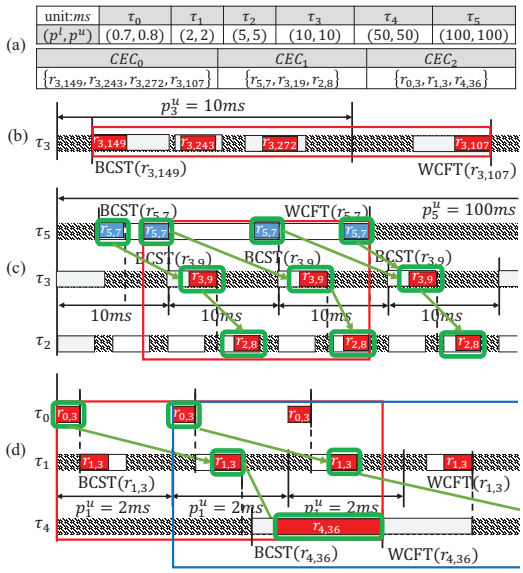


Fig. 1. End-to-end latency computation of three example cause-effect chains $CEC_0$ (b), $CEC_1$ (c), and $CEC_2$ (d). A white box indicates the schedule time bound of a runnable while a red or a blue box is a runnable execution.

A cause-effect chain $CEC_0$ in Fig. 1 (b) consists of four runnables in the same task $\tau_3$. In this case, we have to analyze how many task instances are involved in the chain. If the (i+1)-th runnable of the chain has a smaller index than the i-th runnable, labels written by the i-th runnable will be read by the (i+1)-th runnable in the next task instance. In Fig. 1 (b), two task instances are involved in the chain since index decrease appears only once ($r_{3,272}, r_{3,107}$). If one task instance covers the cause-effect chain, the end-to-end latency can be computed as $UB_c^f(b,e)$ where $r_{c,b}$ and $r_{c,e}$ are the first and the last $\tau_c$ runnables in the chain. Otherwise, the worst-case end-to-end latency becomes the distance from the BCST of the first runnable to the WCFT of the last runnable plus the task period involved in the chain, which gives $p_3^u + WCFT(r_{3,107})$ - $BCST(r_{3,149})$ for Fig. 1 (b). When a chain is a sequence of sub-chains each of which consists of runnables in a same task, we compute the latency of each sub-chain with this approach.

A cause-effect chain $CEC_1$ in Fig. 1 (c) consists of three runnables with different activation patterns. In this case, we consider the schedule time bound of the first runnable ($BCST(r_{5,7})$, $WCFT(r_{5,7})$) and examine all possible BCSTs of the second runnable $r_{3,19}$ that may appear after the first runnable. In the example of Fig. 1 (c), there are three possible BCSTs of $r_{3,19}$. If we consider a pair of runnables only, the worst-case scenario is that the second runnable starts just before the first runnable finishes and the label written by the first runnable is read by the second runnable at the latest in the next task instance. Based on this observation we define a set of starting points of the first runnable as shown in blue color in the figure where the start time of the first runnable coincides with a possible BCST of the second runnable. For the subsequent pair of runnables, we need to consider the schedule time bound of the successor and the WCFT of the predecessor. If the WCFT of the predecessor lies

TABLE I
END-TO-END LATENCIES OF TASKS AND CAUSE-EFFECT CHAINS IN THE ENGINE MANAGEMENT SYSTEM BENCHMARK (UNIT: MS)

| | | proposed | | MAST | |
|---|---|---|---|---|---|
| | | 200MHz | 300MHz | 300MHz | $d$ |
| CORE0 | ISR_10 ($\tau_0$) | 0.03 | 0.02 | 0.02 | 0.70 |
| | ISR_5 ($\tau_1$) | 0.29 | 0.19 | 0.20 | 0.90 |
| | ISR_6 ($\tau_2$) | 0.32 | 0.22 | 0.22 | 1.10 |
| | ISR_4 ($\tau_3$) | 0.69 | 0.46 | 0.47 | 1.50 |
| | ISR_8 ($\tau_4$) | 1.32 | 0.66 | 0.67 | 1.70 |
| | ISR_7 ($\tau_5$) | 2.67 | 1.08 | 1.09 | 4.90 |
| | ISR_11 ($\tau_6$) | 4.23 | 1.30 | 1.32 | 5.00 |
| | ISR_9 ($\tau_7$) | $\infty$ | 2.23 | 2.27 | 6.00 |
| CORE1 | Task_1ms ($\tau_{11}$) | 0.78 | 0.52 | 0.54 | 1.00 |
| | Angle_Sync ($\tau_{12}$) | $\infty$ | 4.69 | 6.60 | 6.66 |
| CORE2 | Task_2ms ($\tau_{13}$) | 0.41 | 0.27 | 0.29 | 2.00 |
| | Task_5ms ($\tau_{14}$) | 1.35 | 0.90 | 0.92 | 5.00 |
| | Task_20ms ($\tau_{16}$) | 18.80 | 9.92 | 11.15 | 20.00 |
| | Task_50ms ($\tau_{17}$) | $\infty$ | 13.16 | 13.57 | 50.00 |
| | Task_100ms ($\tau_{18}$) | $\infty$ | 30.77 | 32.74 | 100.00 |
| | Task_200ms ($\tau_{19}$) | $\infty$ | 30.87 | 32.87 | 200.00 |
| | Task_1000ms ($\tau_{20}$) | $\infty$ | 30.97 | 33.03 | 1000.00 |
| CORE3 | ISR_1 ($\tau_8$) | 0.04 | 0.02 | 0.03 | 9.50 |
| | ISR_2 ($\tau_9$) | 0.06 | 0.04 | 0.04 | 9.50 |
| | ISR_3 ($\tau_{10}$) | 0.08 | 0.05 | 0.06 | 9.50 |
| | Task_10ms ($\tau_{15}$) | $\infty$ | 8.04 | 8.45 | 10.00 |
| EffectChain_1 | | $\infty$ | 11.35 | 12.87 | |
| EffectChain_2 | | $\infty$ | 12.84 | 24.67 | |
| EffectChain_3 | | $\infty$ | 62.64 | 62.91 | |

in the schedule time bound of the successor, the label written by the predecessor should be read by the successor runnable at the latest in the next task instance. For each candidate starting point of the first runnable, the figure shows the longest cause-effect chain by green arrows where red and blue boxes mean the executions of runnables. Among all candidates, we find one that gives the worst-case latency. In Fig. 1 (c) the chain from the second starting point, which is represented by a red bounding box, has the longest latency among four candidates.

The third case shown in Fig. 1 (d) is the case that the cause-effect chain starts with a sporadic task: the first runnable in $CEC_2$ belongs to a sporadic task $\tau_0$. Since the sporadic task may start anytime, we find the worst-case scenario in which the finish time of $r_{0,3}$ is aligned with the best case start time of the first $r_{1,3}$ instance. Then the end-to-end latency from $r_{0,3}$ to $r_{1,3}$ is bounded by $UBL^f(r_{0,3}, r_{0,3}) + p_1^u + WCFT(r_{1,3}) - BCST(r_{1,3})$. Note that we need to check only one starting point, which makes the finish time of the sub-chain be aligned with the best case start time of next sub-chain, unlike the case of periodic tasks in Fig. 1 (c). We repeat this computation for all instances of the first periodic task in the chain within the hyper-period of tasks. In Fig. 1 (d), $\tau_1$ is the first periodic task. If we repeat computation for all $\tau_1$ instances, the maximum latency occurs with the third $\tau_1$ instance since labels written by the third $r_{1,3}$ instance is missed by the first $r_{4,36}$ instance.

## IV. EXPERIMENTS

The estimated end-to-end latencies of all tasks and cause-effect chains from the proposed technique and the MAST approach [10] are compared in Table. I. If the operating frequency of every core is set to 200 MHz, 7 out of 21 tasks in the benchmark model are unschedulable, which are denoted as $\infty$ in the table, since core utilizations are too high: utilizations are 97%, 133.5%, 106.8%, and 117.9% for each core.

When the clock frequency is increased to 300MHz to make all tasks schedulable, the proposed technique shows tighter estimation than the MAST for all tasks and chains as summarized in Table I. On average, the MAST shows 10.45% larger estimates than the proposed technique. Especially the proposed technique shows almost half latency compared to the MAST for EffectChain_2 which contains only periodic tasks as in Fig. 1 (c). Since we consider the possible schedules of runnables within the hyper-period unlike the MAST, the effectiveness of using the schedule time bounds is confirmed.

## V. CONCLUSION

We propose a novel analytical method to estimate the end-to-end latency of a cause-effect chain with consideration of the worst-case memory access delay. We consider the schedule time bounds of runnables for tight estimation. Memory access bound functions are described to find the maximum possible arbitration delay with arrival curve analysis. Since the engine management system benchmark is defined by an automotive industry [5], we believe that the proposed technique is applicable for the similar practical industry-strength benchmark.

## ACKNOWLEDGMENT

This research was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIP) (No. NRF-2016R1A2B3012662).

## REFERENCES

[1] "Timing architects tool suite." [Online]. Available: http://www.timing-architects.com/
[2] "Trace analyzer." [Online]. Available: https://auto.luxoft.com/uth/timing-analysis-tools/
[3] K. Lampka, S. Perathoner, and L. Thiele, "Analytic real-time analysis and timed automata: A hybrid method for analyzing embedded real-time systems," in *Proceedings of EMSOFT '09*, 2009, pp. 107–116.
[4] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *IEE Proceedings - Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, Mar 2005.
[5] "2016 formals methods and timing verification (fmtv) challenge, co-located with the 7th waters," https://waters2016.inria.fr/challenge/.
[6] J. Kim, H. Oh, J. Choi, H. Ha, and S. Ha, "A novel analytical method for worst case response time estimation of distributed embedded systems," in *Design Automation Conference (DAC)*, May 2013, pp. 1–10.
[7] M. Negrean and R. Ernst, "Response-time analysis for non-preemptive scheduling in multi-core systems with shared resources," in *SIES '12*, June 2012, pp. 191–200.
[8] S. Schliecker and R. Ernst, "Real-time performance analysis of multiprocessor systems with shared memory," *ACM Trans. Embed. Comput. Syst.*, vol. 10, no. 2, pp. 22:1–22:27, Jan. 2011.
[9] "Mast: Modeling and anlysis suite for real-time applications." [Online]. Available: http://mast.unican.es/
[10] J. M. Rivas Concepcin, J. J. Gutirrez, J. Medina, and M. Harbour, "Calculating latencies in an engine management system using response time analysis with mast," in *7th WATERS, FMTV Challenge*, Jul. 2016.
[11] I. Stierand, P. Reinkemeier, S. Gerwinn, and T. Peikenkamp, "Computational analysis of complex real-time systems fmtv 2016 verification challenge," in *7th WATERS, FMTV Challenge*, Jul. 2016.
[12] AUTOSAR, "Autosar specification of rte software." [Online]. Available: https://www.autosar.org/
[13] AUTOSAR, "Autosar specification of timing extensions." [Online]. Available: https://www.autosar.org/
[14] D. Kang, J. Choi, and S. Ha, "Worst case delay analysis of shared resource access in partitioned multi-core systems," in *International Symposium on Embedded Systems for Real-time Multimedia*, Oct. 2017.