On Reducing Busy Waiting in AUTOSAR via Task-Release-Delta-based Runnable Reordering

Robert Höttger, Burkhard Igel
University of Applied Sciences and Arts Dortmund
Pimes Research Department
Otto-Hahn-Str. 23, 44227 Dortmund, Germany
{robert.hoettger, igel}@fh-dortmund.de

Olaf Spinczyk
Technische Universität Dortmund
Department of Computer Science 12
Otto-Hahn-Str. 16, 44221 Dortmund, Germany
olaf.spinczyk@tu-dortmund.de

***Abstract*—The increasing amount of innovative software technologies in the automotive domain comes with challenges regarding inevitable distributed multi-core and many-core methodologies. Approaches for general purpose solutions have been studied over decades but do not completely meet the specific constraints (e. g. timing, safety, reliability, affinity, etc.) for AUTOSAR compliant applications. AUTOSAR utilizes a spinlock mechanism in combination with the priority ceiling protocol in order to provide mutually exclusive access to shared resources. The essential disadvantages of spinlocks are unpredictable task response times on the one hand and wasted computation time caused by busy waiting periods on the other hand.**

In this paper, we propose a concept of task-release-delta-based runnable reordering for the purpose of sequentializing parallel accesses to shared resources, resulting in reduced task response times, improved timing predictability, and increased parallel efficiency respectively. To achieve this, runnables that represent smallest executable program parts in AUTOSAR are reordered based on precedence constraints. Our experiments among industrial use cases show that task response times can be reduced by up to 18,2%.

I. INTRODUCTION

Due to the increasing demands of assistant systems, safe autonomous driving, its complex and dynamic behavior, and the corresponding need for computing power in the automotive domain, multi-core and many-core approaches need to efficiently utilize parallel and distributed resources. While facing the migration of legacy single core software to multicore platforms in AUTOSAR[1], not only schedulability, distribution, or allocation under timing, safety, precedence, or affinity constraints have to be investigated, but also shared resource issues like priority inversion, starvation, deadlocks, or nested blocking behavior which are crucial for validating correct software behavior and optimizing different goals, e. g. energy consumption, throughput, costs, reliability, etc.

In terms of AUTOSAR, software components realize certain functionalities and consist of runnables that are grouped to schedulable tasks. Tasks can be mapped to different cores while corresponding memory exchanged between tasks is usually protected by spinlocks. Tasks, mapping, and scheduling are statically defined once during application configuration [1]. Hence, on-line runnable reordering is infeasible and would involve disproportionate overheads. However, Kluge *et al.* have shown in [2] that on-line task filtering can be used in AUTOSAR to efficiently schedule tasks while considering resource conflicts and active tasks. This can be extended to schedule task instances with specific (off-line calculated) runnable orders to any point in time to reduce busy waiting periods while not violating any precedence constraints.

In general, managing mutual exclusion of shared resources has been addressed by various locking protocols and evolved to meet different challenges, e. g. dynamic priorities or nested global critical sections etc. Using spinlocks instead of semaphores has proven to maintain cache affinity and avoid scheduler invocations as well as context switches [3]. Despite that, applications using spinlock mechanisms suffer from busy waiting in case of resource blocking that occurs when a runnable's request to a lock is rejected since another runnable is already holding the lock. Hence, spinlocks are the prior choice for inherently short critical sections only. Considering modern model-based engineering approaches, significant knowledge of causal dependencies within software functions can be used for reordering runnables within a task to sequentialize shared resource accesses and reduce unnecessary blocking, i. e., busy waiting and the tasks' response times correspondingly.

We present the TDRR (Task-Release-Delta-based Runnable Reordering) approach, that is implemented along the APP4MC tool platform[2]. TDRR provides runnable orders for tasks to reduce busy waiting according to a specific system state while considering runnable dependencies such that functional behavior is retained and no revalidation is necessary.

The remainder of this paper is structured as follows: The next Section II gives a brief overview of related work and introduces the environment this work has been based on. Section III defines the system model and all necessary notations required for the presented TDRR approach of Section IV. After illustrating some special cases among hypothetical examples, Section V further presents results gained from industrial automotive applications. Finally, Section VI outlines benefits and challenges of the approach whereas Section VII concludes this paper.

II. RELATED WORK

APP4MC forms the basis of our TDRR implementation and has recently (June 2016) been published as an open

[1]AUTomotive Open System ARchitecture: http://www.autosar.org

[2]As part of the AMALTHEA4PUBLIC project, this work has been funded by the Federal Ministry of Education and Research - BMBF, under funding no. $01|S14029K$

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

source Eclipse project[3]. With its current project members (20), associated partners (12) and its advisory board members (8), including different OEMs, Tier 1 & 2 suppliers as well as consultant companies and research institutes, APP4MC provides a commonly accepted model basis and serves as an open, extensible development platform to integrate, combine, and analyse various tooling. The data model is called AMALTHEA and is used in the remainder of this paper. It extends AUTOSAR by various timing elements and necessary constraint artifacts. We further utilize APP4MC parallelization processes [4] to identify parallel partitions and map them to hardware elements [5] based on model properties, configurations, and analyses. TDRR is applied to such partitioned and mapped models.

In order to meet the challenges of embedded multi- and many-core systems in terms of managing accesses to shared resources, the use of different semaphore- or spinlock-based locking protocols was found practical. Suspension-based semaphore protocols were introduced by Rajkumar *et al.* in [6] and were advanced to the priority ceiling protocol (PCP) to bound priority inversion as well as prevent chained blocking and deadlocks. The PCP is used to ensure local mutual exclusion across tasks on one processor in AUTOSAR.

For protecting shared resources among tasks executed on different processors, AUTOSAR utilizes spinlocks that can potentially lead to deadlocks and priority inversion. These issues are avoided for non-nested locks via either returning an error to the task requesting an already hold lock or via suspending all interrupts such that the *locking* task can not be preempted. Both approaches are ineffective regarding implementation overhead and starvation [7]. With our TDRR mechanism, we want to fill the gap between semaphore-based protocols and spinlock-based protocols, such that the latter can also be applied to applications with longer critical sections without creating significant *spinning* times.

Since TDRR considers initial runnable orders, partitioning and task splitting are required and have major influence on the efficiency of the presented approach. In [8], [9], and [4], precedence-constrained-based task splitting respectively partitioning is presented for the purpose of parallelizing sequential legacy code based on the AMALTHEA model. The generated tasks execute optimally by a specified quality attribute for minimizing execution time in case release times of tasks are equal. However, many precedence constraints across tasks result in synchronization periods (idle waiting) as soon as tasks, that communicate with each other, are released at different points in time. Lowinski *et al.* [8] [9] use HLEFT list scheduling extended by greedy heuristics to either maximize speedup (earliest execution), reduce cross partition precedence constraints (maximal parents), or increase the gradient (*slackness*, minimal distance) of created partitions. Precedence constraints are ensured by synchronization points, to which runnables may have to wait in case a synchronization point has no gradient. In [4], critical path and earliest start schedule partitioning strategies are explained. Precedence-constraint-based partitioning is required for TDRR to ensure the same task behavior of tasks executed in parallel based on causal runnable dependencies. Partitioning in APP4MC can be influenced by its configuration parameters, runnable pairing or -separation constraints as well as cyclic dependency dissolutions. Other partitioning strategies, such as bin-packing in combination with a least-loaded algorithm presented by Monot *et al.* in [10], are applicable to TDRR, but are subject to revalidation due to the fact that precedence constraints can be violated. This violation is caused by the assumption that runnables executed on different processors do not share resources that does not hold for the newest AUTOSAR standard, since inter-task dependencies can exist for *global* resources.

[3]Available at: http://projects.eclipse.org/projects/technology.app4mc

In contrast to RunPar [1], that schedules tasks sequentially, TDRR assumes that runnables of different tasks can be executed concurrently resulting in less idle intervals. Moreover, the supertask approach recently presented in [11] improves response time (measured upon speedup) for a single instance of a task but has significant disadvantages due to the least common multiple period of tasks contained in the supertask. Consequently, runnables of tasks with higher periods are scheduled more often and the total system load is increased. Additionally, assuming indivisible periods like 5ms or 7ms, the period definition for a *merged* supertask must not be the hyper period i. e. least common multiple.

III. PRELIMINARIES AND SYSTEM MODEL

Instead of introducing another protocol, we estimate to which situation *conflicts* between tasks, i. e. concurrent accesses to a shared resource, can occur and define runnable orders for these situations to reduce conflicts as much as possible. Such situations, that result in busy waiting, are identified via investigating task release time differences, initial runnable orders and resource accesses. New runnable orders are calculated for the latter released task for such conflict situation based on precedence constraints and the system model. The approach's benefit is the analysis of concurrent accesses to shared memory at early design steps, its application to AUTOSAR models, and improved system performance via reducing task response times. The TDRR idea is shown in Figure 1.

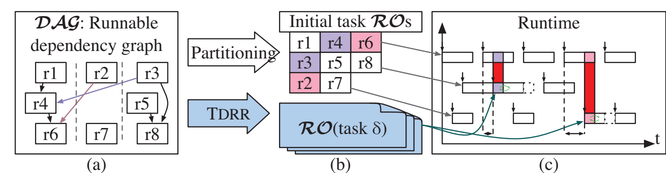

Fig. 1: TDRR idea to reduce busy waiting

The runnable dependency graph $\mathcal{DAG}$ (a) is used to form tasks with their initial runnable order during the partitioning and to calculate additional runnable orders based on specific release time delta values via our TDRR approach (b). If the specific release delta values are detected at runtime of the program (c), the according runnable orders replace the initial order.

In the following, the system model for calculating metrics required to identify runnable orders is defined.

A label set $\mathcal{L} = \{l_0, ..., l_\gamma\}$ consists of labels l_g that denote, e. g., memory that can be written or read by runnables.

A runnable set $\mathcal{R} = \{r_0, ..., r_\iota\}$ defines runnables $r_i = \{\text{ins}_i, \mathcal{L}_{\text{w},i}, \mathcal{L}_{\text{r},i}\}, i \leq \iota$ that consist of an instructions value $\text{ins}_i \in \mathbb{N}$, a write label set $\mathcal{L}_{\text{w},i} \subseteq \mathcal{L}$, and a read label set $\mathcal{L}_{\text{r},i} \subseteq \mathcal{L}$. The instruction value ins is constant and defines the weight of a runnable and quantifies the computation to perform the runnable's function. Dynamic instructions can be also modeled upon upper and lower deviation bounds but are assumed to be constant for this paper's scope.

An edge set $\mathcal{E} = \{e_{r_\omega \to r_\rho} \mid (\mathcal{L}_{\text{w},\omega} \cap \mathcal{L}_{\text{r},\rho}) \neq \emptyset \; \forall \; \omega \neq \rho\}$, with $\omega \leq \iota, \rho \leq \iota$, defines precedence constraints and all edges are derived from runnable's label accesses: if the same label is written by a runnable r_ω and read by another runnable $r_\rho \neq r_\omega$ there exists exactly one edge $e_{r_\omega \to r_\rho}$. An edge $e_{r_\omega \to r_\rho}$ consequently defines a linear order such that the source runnable r_ω can be denoted as a predecessor and a target runnable r_ρ can be denoted as a successor: $e_{r_\omega \to r_\rho} \Rightarrow r_\omega \prec r_\rho$. Such order is crucial to not be violated and a predecessor has to be always executed before its successor. This precedence constraint is required to ensure end-to-end latency bounds and forms an essential difference compared with classical software parallelization [11]. An algorithm to derive edges from label accesses is also given in [9].

A ProcessPrototype set $\mathcal{P} = \{p_0, ..., p_\mu\}, \; \mu \leq \iota$ groups runnables such that a ProcessPrototype $p_m \subset \mathcal{R}$ consists of a runnable subset and each runnable is contained in exactly one ProcessPrototype:

$$\forall \; p_i, p_j \in \mathcal{P} : (p_i \cap p_j) = \emptyset; \quad \text{ with } i \neq j, \bigcup_{m=0}^{\mu} p_m = \mathcal{R}$$

A task set $\mathcal{T} = \{t_0, ..., t_\mu\}$ defines for each task t_m a permutation of p_m without repetition with $|t_m| = |p_m|$. Tasks have a static call sequence of runnables in APP4MC whereas our TDRR approach assumes not a static order but specific permutations such that different task *instances* may have different call sequences, i. e. permutations. Any two concatenated runnables $r_i, r_{i+1} \in t$ are binary related via the strict order $r_i \prec r_{i+1}$.

Based on these AUTOSAR models, the following metrics can be calculated that are required to form TDRR runnable orders.

Firstly, for each runnable we define its position in its contained task via the initial start time $\text{ist}_r = \sum_i \text{ins}_{r_i} \; : \; r_i \prec r$ and the initial end time $\text{iet}_r = \sum_i \text{ins}_{r_i} \; : \; r_i \succ r$. These values are used later in Algorithm 1.

A runnable directed acyclic graph is denoted as $\mathcal{DAG} = \{\mathcal{R}, \mathcal{E}\}$ and consists of a runnable set and an edge set. The graph is adjusted to be acyclic and Tasks $\mathcal{T}$ are formed via the partitioning strategies CPP or ESSP described in [4]. The graph structure is used for deriving to what extent a runnable can be shifted within a task while not violating dependencies.

The set $\text{PRS}_r = \{\text{pseq}_0, ...\}$ denotes preceding runnable sequences of r: each sequence pseq in PRS_r is an ordered list of concatenated runnables, i. e., there is an entry runnable that has no predecessor (no incoming edge) and one successor, that is concatenated through runnables until PRS_r sequence's last runnable's successor is r.

$\text{SRS}_r = \{\text{sseq}_0, ...\}$ denotes succeeding runnable sequences of r: each sequence sseq in SRS_r is an ordered list of runnables, that are concatenated, i. e., the first element of a SRS_r sequence is a direct successor of r, that is concatenated through runnables until an exit runnable that has no successor (no outgoing edge) forms the end of the sequence.

Time-shift values of runnables are defined by $\text{tsv}_r = \text{lstart}_r - \text{estart}_r$, where a runnable's earliest start value (estart) is calculated via the maximal preceding sequence's instructions sum, and the latest start value (lstart) is calculated via the difference of the $\mathcal{DAG}$'s total instructions sum, the runnable's instructions, and the maximal succeeding sequence's instruction sum, respectively:

$$\text{estart}_r = \max_{\text{pseq}} \sum_i \text{ins}_i : \quad \forall i \text{ with } r_i \in \text{pseq}$$

$$\text{lstart}_r = \sum_j \text{ins}_j - \text{ins}_r - \max_{\text{sseq}} \sum_i \text{ins}_i : \quad \forall j \text{ with } r_j \in \mathcal{DAG}, \; \forall i \text{ with } r_i \in \text{sseq}$$

The time-shift values are required to both, perform the partitioning and to decide about conflicting runnable's position in a task such that precedence constraints, derived from the edge set $\mathcal{E}$, are not violated. For the latter case, the $\mathcal{DAG}$ is reduced to runnables and edges contained in a single task only (further described at Figure 3).

A set of conflicting labels is defined by:

$$\mathcal{L}_c = \{l_0, ..., l_\eta \mid l_e \in (\mathcal{L}_{w,i} \cup \mathcal{L}_{r,i}) \cap (\mathcal{L}_{w,j} \cup \mathcal{L}_{r,j}) \text{ with } (r_i \in t_i) \cap (r_j \notin t_i)\} \subset \mathcal{L}, \eta \leq \gamma, i \neq j$$

Hence, each label $l_e \in \mathcal{L}_c$ is accessed by at least two runnables contained in different tasks. Accesses to these labels can be derived from AUTOSAR notations `ReleaseResource()` and `GetResource()`.

A conflict set $\mathcal{C} = \{c_0, ..., c_\kappa\}$ defines for each conflict $c_k = \{l_e, \mathcal{T}_{c_k}\}$ a label $l_e \in \mathcal{L}_c$ that is contained in at most one conflict ($\forall \; l_e \; \exists! \; c$), and all tasks $\mathcal{T}_{c_k}$, that contain at least one runnable accessing l_e:

$$\mathcal{T}_{c_k} = \{t_i, ...| \; l_e \in (\mathcal{L}_{\omega,i} \cup \mathcal{L}_{\rho,i}) \text{ for at least one } r_i \in t_i\} \subset \mathcal{T}$$

Since runnable orders ($\mathcal{RO}$) are calculated for specific release time differences of conflicting tasks, a Δ_c matrix is defined that identifies the release time deltas between tasks to which conflicts occur:

$$\Delta_c = \begin{pmatrix} \delta_{t_0 \to t_0} & ... & \delta_{t_0 \to t_n} \\ \vdots & \ddots & \vdots \\ \delta_{t_n \to t_0} & ... & \delta_{t_n \to t_n} \end{pmatrix} : n = |\mathcal{T}|$$

A δ element is defined by a set of intervals: $\delta_{t_i \to t_j} = \{(a, b), ... \mid a < b\}$, such that if t_i was released within any of the intervals $\{(a, b), ...\}$ earlier than t_j, a busy waiting period occurs at either t_i or t_j. With the

common standard for set notation, an open interval (a, b) excludes the lower limit a and the upper limit b such that the release time of t_i, denoted as $\mathrm{RT}(t_i)$, must be $(a < \mathrm{RT}_{t_i} < b)$ for busy waiting to occur. The first δ index t_i is the row of Δ_c and the second index t_j is the column. An interval in $\delta_{t_i \to t_j}$ is calculated via $a = \mathrm{iet}_{r_i} - \mathrm{ist}_{r_j}$ and $b = \mathrm{iet}_{r_j} - \mathrm{ist}_{r_i}$ for $\mathrm{iet}_{r_i} > \mathrm{ist}_{r_j}$, i. e. if the intervals overlap, or via $a = \mathrm{ist}_{r_j} - \mathrm{iet}_{r_i}$ and $b = \mathrm{iet}_{r_j} - \mathrm{ist}_{r_i}$ if the intervals do not overlap. For both cases it applies that r_i is contained in a different conflicting tasks than r_j and $\mathrm{ist}_{r_i} \leq \mathrm{ist}_{r_j}$.

At runtime, if $(a \leq (\mathrm{RT}(t_j) - \mathrm{RT}(t_i)) \leq b)$ for any $(a, b) \in \delta_{t_i \to t_j}$, a busy waiting would appear. The actual release times of tasks vary and depend on various circumstances, e. g. the system state, the scheduler, etc. Each δ element may consist of an interval set in case the conflict's label is accessed multiple times by the corresponding task(s) or a different label is subject to a conflict between the same tasks. An empty set element denotes that the corresponding tasks do not conflict in any way, e. g. $\delta_{t_0 \to t_1} = \emptyset$ means that if t_0 was released earlier than t_1, no conflict occurs between them.

The following Figure 2 illustrates an example case showing three tasks (t_0)–(t_2), where rectangles denote runnables (same instruction amounts for simplicity) and the label L denotes an access to a shared resource. As soon as two runnables access the same label at the same time (vertically), a conflict occurs denoted in a vertical rectangle. Task release delta values, to which such conflicts occur, are indicated below the horizontal time axis. The upper diagram shows initial $\mathcal{RO}$ execution resulting in busy waiting and the lower diagram presents task executions for the same release delta values without any busy waiting due to reordered runnables. For this case, it is assumed that there are no precedence constraints between the runnables and that task release times are recorded during runtime upon a common time basis.

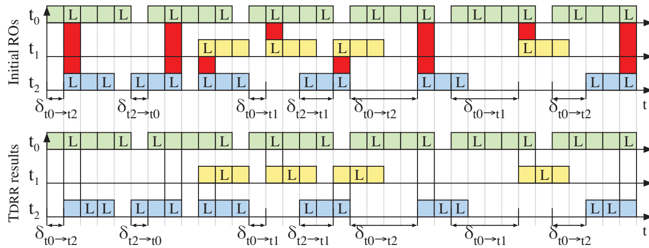

Fig. 2: Three conflicting tasks, five label accesses, 8 task delta values resulting in busy waiting

The corresponding delta matrix is:

$$\Delta_c = \begin{pmatrix} 0 & \{(0,2),(3,5)\} & \{(0,5)\} \\ \emptyset & 0 & \{(0,1)\} \\ \{(0,2)\} & \{(0,3)\} & 0 \end{pmatrix}$$

Intervals regarding the same label are merged such that overlapping or adjacent intervals are combined to their outermost values. For all combined intervals in a Δ_c column (different conflicting labels and tasks), all interleaving time frames are separated within the conflict interval set $\mathrm{CI}_{t_m} = \{\mathrm{ci}_{t_i \to t_m}, ...\}$. As soon as other intervals start or end, new ci entries are created, e. g. intervals $\delta_{t_i \to t_m} = \{(0,4),(3,5)\}$ result in $\mathrm{ci}_{t_i \to t_m} = \{(0,3),(3,4),(4,5)\}$. This is necessary to consider all conflicts occuring at a specific combination of released (and executing) tasks that conflict with t_m and to calculate $\mathcal{RO}$s for all respective combinations (CI result is given from line 2 in Algorithm 1). Specific values of a conflict interval are addressed via dot notation, e. g. $\mathrm{ci}.a$ for the interval ci's start value a.

Finally, we define assignable runnables to a specific position in t_m by:
$\mathrm{AR}(\mathrm{pos}, t_m) = \{r_i, ...|\ \mathrm{estart}_{r_i} \leq \mathrm{pos} \leq \mathrm{lstart}_{r_i}, \mathrm{st}_{r_i} \geq \mathrm{pos}, r_i \notin \mathcal{T}_{c_k}\} \subset t_j$. As described earlier, estart_{r_i} is defined by the task's runnable dependency $\mathcal{DAG}$ and equals $\sum_i r_i\ :\ r_i \in \mathrm{PRS}_{r_i}$, whereas st_{r_i} corresponds the start time in the current $\mathcal{RO}$. This additional constraint ($\mathrm{st}_{r_i} \geq \mathrm{pos}$) is necessary to not reassign a runnable that has already been moved to a position prior to the current pos value.

Having the above metrics, $\mathcal{RO}$s can be calculated that define for each interval ci in $\delta \in \Delta_c$ and corresponding conflicting tasks a runnable order for a latter released task. Reordering runnables within a task that has already progressed is certainly less effective since shifting potential is reduced by already executed runnables.

IV. TDRR Approach

A task's permutation depends on the task release delta matrix Δ_c, the runnable dependency graph $\mathcal{DAG}$, and the initial runnable orders $\mathcal{RO}_{\mathrm{init},t_j}$ provided by, e. g., partitioning approaches presented in [4], runnable activation groups, or other task creation methodologies. At each release of a task t_m, the system is observed whether there are tasks executing that fulfill the three conditions: (1) they share a conflicting label with t_m, (2) they were released earlier than t_m and still execute, and (3) their release time difference is within at least one δ conflict range (cf. Table I).

TABLE I: Conditions to identify conflicting tasks

(1)	$((\mathcal{L}_{t_i} \cap \mathcal{L}_{t_m}) \neq \emptyset) \subseteq \mathcal{CL},\ i \neq m$
(2)	$(\mathrm{RT}(t_i) < \mathrm{RT}(t_m)) \wedge (I_{t_i} + \mathrm{RT}(t_i) > \mathrm{RT}(t_m))$
(3)	$(a \leq (\mathrm{RT}(t_m) - \mathrm{RT}(t_i)) \leq b)$ for at least one Interval $(a, b) \in \delta_{t_i \to t_m}$

If all three conditions are given, the latter task t_m is released with a runnable order that minimizes busy waiting, calculated via Algorithm 1. These runnable orders also consider multiple conflicting tasks of $\mathcal{T}_{c_k}$ that are executing upon the release of t_m. We propose the following greedy Algorithm 1 that shifts conflict unaffected runnables to conflicting task intervals based on their tsv property. Calculating $\mathcal{RO}$s for each ci value ensures that all conflicts (labels), all release δ values and all tasks are considered. It may occur that a ci value concerns many overlapping δ values such that $\mathcal{RO}$s can not be calculated to eliminate all busy waiting for a specific situation. Nevertheless, lines 3+ try to reduce this as much as precedence constraints allow it. The first loop at line 1 iterates among all tasks, and the second loop at line 3 addresses all

Algorithm 1: TDRR Algorithm

Data: $\Delta_c, \mathcal{DAG}, \mathcal{RO}_{\text{init}} = \mathcal{T}$
Result: $\mathcal{RO}(t_m, \text{ci})$

1 **forall** $t_m \in \mathcal{T}$ with at least one $\delta_{t_i \to t_m} \in \Delta_c \neq \emptyset$ **do**
2 $\quad \text{CI}_{t_m}.addAll\left(\text{getInterlIntervals}\left(\bigcup_{t_i \in \mathcal{T}} \delta_{t_i \to t_m}, \mathcal{RO}_{\text{init}}\right)\right)$;
3 $\quad$ **forall** $\text{ci} \in \text{CI}_{t_m}$ **do**
4 $\quad\quad \text{pos} \leftarrow \text{ci}.a$;
5 $\quad\quad$ **while** $\text{pos} < \text{ci}.b$ **do**
6 $\quad\quad\quad$ get $\text{AR}(\text{pos}, t_m)$;
7 $\quad\quad\quad$ **if** $|AR| > 0$ **then**
8 $\quad\quad\quad\quad$ sort AR by increasing tsv_r and decreasing ins_r;
9 $\quad\quad\quad\quad$ move first element $\text{ar} \in \text{AR}$ from st_{ar} to pos;
10 $\quad\quad\quad\quad \text{pos} \ \leftarrow \text{pos} + \text{ins}_{\text{ar}}$;
11 $\quad\quad\quad$ **else**
12 $\quad\quad\quad\quad \text{pos} \leftarrow \text{pos} + 1$;
13 $\quad\quad\quad$ **end**
14 $\quad\quad$ **end**
15 $\quad$ **end**
16 **end**

conflicting intervals and corresponding situations provided by line 2 for each of these tasks. Line 5 finally considers all positions within these intervals. Line 6 identifies assignable runnables to a specific position as stated earlier and line 9 shifts the most effective runnable (identified by line 8) one after another towards pos and correspondingly moves the conflicting runnable backwards until the problematic runnable left the conflict interval ($\text{pos} > \text{ci}.b$ cf. line 5). The slots are kept as small as possible since line 12 increments the position value by a single instruction in case no runnable is assignable to pos due to the runnable dependency graph structure ($\mathcal{DAG}$). Consequently, runnable reordering is performed on the most fine grained level as possible. Regarding line 12, it has to be added that the implementation further considers whether no runnable can be be placed within the according conflicting interval. In this situation, the initial runnable order is kept to not create 'empty' time intervals within the according task. Line 8 ensures efficiency since runnables assignable to a short time frame are selected prior to more flexible runnables with a greater tsv value.

Figure 3 illustrates an example runnable $\mathcal{DAG}$ for a task (a), a corresponding initial runnable order $\mathcal{RO}_{\text{init}}$ (b) where two runnables r_0 and r_3 are assumed to conflict with another task, and a $\mathcal{RO}$ representing the result of Algorithm 1 in (c). Dependencies are shown as arrows and runnables as rectangles with a specific width that represents their relative instruction amounts.

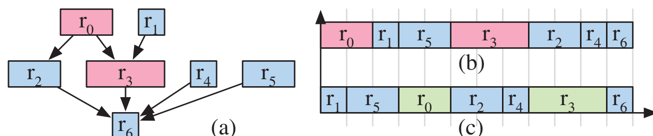

Fig. 3: (a) example $\mathcal{DAG}$, (b) initial $\mathcal{RO}$, (c) adapted $\mathcal{RO}$ no conflicts

Line 6 of Algorithm 1 identifies (with $r(\text{tsv}_r, \text{ins}_r)$) initially $\text{AR} = \{r_1(7,1), r_5(9,2)$ and $r_4(10,1)\}$ for $\text{pos} = st_{r_0} = 0$ since $r_2, r_3,$ and r_6 cannot execute before r_0. Since r_1 has the lowest shift value tsv_r, it is selected to be moved to $\text{pos} = 0$. The second iteration for $\text{pos} = 1$ identifies $\text{AR} = \{r_4, r_5\}$ without r_1 since $\text{st}_{r_1} = 0 < \text{pos} = 1$. r_5 is selected since $\text{tsv}_{r_5} = 9 < 10 = \text{tsv}_{r_4}$. Subsequently, pos is increased to 6 by line 12 where the conflicting runnable r_3 has been shifted to. Since only r_2 is available, it is swapped with r_3 and the algorithm finishes since line 5 $\text{pos} = 8 \geq 8 = \text{iet}_{r_3}$. The final resulting $\mathcal{RO}$ is shown in Figure 3 (c).

V. Experiments

To investigate benefits and industrial use, TDRR was applied to three AMALTHEA models: (1) a democar model [12], (2) an automotive anonymized model from industry (denoted as AIM), and (3) a publicly available model provided by [13] (denoted as FMTV). The amounts of runnables, labels, tasks, calculated conflicts, $\mathcal{RO}$s, and the maximal response time reductions are shown in Table II.

For the democar model, the runnable instructions are evenly 80000 except for two runnables with 160000 instructions each (in the following, corresponding values are reduced by factor 10000 for clarity reason). Initial tasks group runnables by their periodic activations of 5ms, 10ms, and 20ms. As soon as task splitting or partitioning is performed, the amount of conflicts increases since $\mathcal{DAG}$s are cut and more labels are shared between tasks. The initial three task configuration results in 26 labels used by more than a single task. These labels define the conflicts subject to TDRR investigation. This investigation results in $\delta_{t10ms \to t20ms} = \{(24, 56)\}$ and $\delta_{t10ms \to t5ms} = \{(56, 88), (96, 144)\}$ across all conflicts. Consequently, $\mathcal{RO}$s are calculated for t_{5ms} only, since (1) t_{5ms} and t_{20ms} do not share resources, (2) t_{20ms} consists of a single runnable, and (3) the later release of task t_{10ms} has no affect on t_{5ms} or t_{20ms}. The time wasted for busy waiting can be reduced by maximal 16 time units for the worst conflict case $\delta_{t10ms \to t5ms} = 64$ as shown in Figure 4.

The calculated runnable orders provide a conflict free execution across all task release delta values. If, for example, t_{10ms} is released 64 time units before the release of t_{5ms}, we derive that the first four runnables conflict due to their label accesses. While a busy waiting of the first runnable would as well shift all succeeding runnables, there can still occur further busy waiting (cf. Figure 4, $t_{5\text{ms,init}}$). The reordering of runnables optimally reduces the response time of task t_{5ms} by $\frac{16}{88} \approx 18,2\%$ such that no busy waiting occurs at all. This case is shown in Figure 4 where the letters in the rectangles represent acronyms for runnables and the accessed labels.

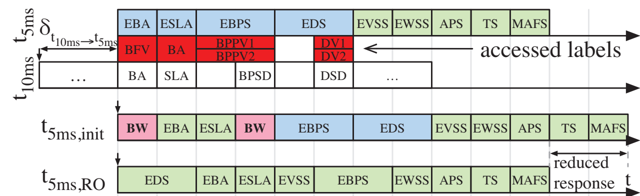

Fig. 4: TDRR applied to democar model [12], Case $\delta_{t10ms \to 5ms} = 64$, $\mathcal{RO}_{t_{5ms}}$ provided to save 16 busy waiting time units

Across all conflicting intervals, the 10 calculated $\mathcal{RO}$s eliminate busy waiting completely.

The second model comes from industry and has a greater amount of model elements compared with the democar model. Across all runnables, the total mean instruction sum is 4260966. For the 4159 conflicts, 430 critical release time delta values are derived to which 10470 $\mathcal{RO}$s are calculated to resolve all possible conflicts as much as possible. For the worst release delta value (to which the longest busy waiting time is predicted), the task's response time can be reduced by 12,5%.

The third investigated model (denoted as FMTV) is provided by Hamann *et al.* in [13]. The model has been published to provide a real world example upon application characteristics presented in [14]. In total, the model consumes 147712230 mean instructions across all runnables. The lowest instruction value is 364 and the highest 1792703. Here, we assume that the response time reduction achieved by TDRR is relatively small since the tasks may have already been optimized by simulation tools. Table II summarizes properties of the three examined models as well as maximal response time reductions for conflicting tasks.

TABLE II: TDRR results regarding three investigated models

Model	Democar	AIM	FMTV
Runnables	43	1297	1250
Labels	71	46929	10000
Tasks	3	77	21
Conflicts	17	4159	2249
Merged δ values	3	430	1100
$\mathcal{RO}$s	10	10470	1892
Max. response time reduction	18,2%	12,5%	2,2%

VI. Discussion

TDRR assumes the executing task's runnable orders from an initial solution. However, a task may already be released upon a different permutation due to another conflict. Consequently, the predicted release delta conflicts can occur at different delta values than expected. This fact has been omitted so far, since most conflicts were observed to appear without predecessor tasks being already reordered, but we will address these cases in upcoming work.

Another challenge that requires further investigation is the fact that runnables may be called in more than one task, resulting in tasks being multisets. In our examined models, this was only the case for the second (AIM) model.

In addition, using data conflict graphs [15] or runnable interaction graphs [16] could ease calculating $\mathcal{RO}$s. Since the amount of $\mathcal{RO}$s depends on the amount of $\mathcal{L}_c$, the partitioning heuristic presented in [4] could be extended by Systematic Memory based Simulated Annealing (SMSA) as presented in [16] to reduce the amount of calculated $\mathcal{RO}$s.

Future work also addresses determining overheads caused by $\mathcal{RO}$ look ups during execution time and the memory required to store $\mathcal{RO}$s and the task release times. Also, the affect of frequent task preemptions to conflicts will be investigated along with a simulation tool and more models will be investigated e. g. regarding $\mathcal{DAG}$ generators.

Another interesting study is the direct comparison of spin-lock based TDRR with semaphore based approaches disregarding AUTOSAR. This could also lead to utilizing TDRR in other operating systems upon another abstraction below tasks to schedule subtasks more efficiently.

VII. Conclusion

Using TDRR to execute tasks with different off-line calculated runnable orders for the purpose of reducing busy waiting was successfully applied to AMALTHEA models. While some challenges remain to be addressed in future work, we were able to improve system performance by reducing task response times by up to 18,2%. To the best of our knowledge, this is the first approach to have a varying runnable order within a task in AUTOSAR, while preserving precedence constraints and not accompanying significant re-validation efforts for retaining the system's software behavior.

References

[1] S. Kehr, M. Panic, E. Quinones *et al.*, "RunPar : An Allocation Algorithm for Automotive Applications Exploiting Runnable Parallelism in Multicores," in *Proc. of the Int. Conf. on Hardware/Software Codesign and System Synthesis*. ACM, 2014, pp. 29:1–29:10.

[2] F. Kluge, C. Yu, J. Mische *et al.*, "Implementing AUTOSAR Scheduling and Resource Management on an Embedded SMT Processor," in *Proc. of th 12th Int. Workshop on Software and Compilers for Embedded Systems*. New York, NY, USA: ACM, 2009, pp. 33–42.

[3] A. Wieder and B. B. Brandenburg, "On Spin Locks in AUTOSAR: Blocking Analysis of FIFO, Unordered, and Priority-Ordered Spin Locks," in *Proc. of the IEEE 34th Real-Time Systems Symp., RTSS*, 2013, pp. 45–56.

[4] R. Hoettger, L. Krawczyk, and B. Igel, "Model-Based Automotive Partitioning and Mapping for Embedded Multicore Systems," *Int. Conf. on Parallel, Distributed Systems and Software Engineering, Istanbul, Turkey*, vol. 2, no. 1, pp. 2643–2649, 2015.

[5] L. Krawczyk, C. Wolff, and D. Fruhner, "Automated Distribution of Software to Multi-core Hardware in Model Based Embedded Systems Development," in *Proc. of the 21st Int. Conf. on Information and Software Technologies (ICIST)*. Springer Int., 2015, pp. 320–329.

[6] R. Rajkumar, L. Sha, and J. P. Lehoczky, "Real-Time Synchronization Protocols for Multiprocessors," in *Proc. of the 9th IEEE Real-Time Systems Symp. (RTSS'88)*, 1988, pp. 259–269.

[7] K. Lakshmanan, "AUTOSAR Extensions for Predictable Task Synchronization in MultiCore ECUs," *Proc. of the SAE World Congress and Exhibition*, 2011.

[8] M. Lowinski, D. Ziegenbein, and S. Glesner, "Partitioning Embedded Real-Time Control Software based on Communication Dependencies," in *Proc. of the Int. Workshop on Modelling in Automotive Software Engineering*, 2015, pp. 2–11.

[9] ——, "Splitting Tasks for Migrating Real-Time Automotive Applications to Multi-Core ECUs," in *11th IEEE Symp. on Industrial Embedded Systems*, 2016.

[10] A. Monot, N. Navet, B. Bavoux *et al.*, "Multi-source software on multicore automotive ECUs," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 10, pp. 3934–3942, 2012.

[11] S. Kehr, M. Panic, E. Quinones *et al.*, "Supertask : Maximizing Runnable-level Parallelism in AUTOSAR Applications," in *Design, Automation & Test in Europe (DATE)*, 2016, pp. 25–30.

[12] P. Frey, "A Timing Model for Real-Time Control-Systems and its Application on Simulation and Monitoring of AUTOSAR Systems Dissertation," PhD Thesis, Ulm University, 2010.

[13] A. Hamann, D. Ziegenbein, S. Kramer *et al.*, "FMTV 2016 Verification Challenge," *Robert Bosch GmbH*, 2016.

[14] S. Kramer, D. Ziegenbein, and A. Hamann, "Real World Automotive Benchmarks For Free," in *6th Int. Workshop on Analysis Tools [...] for Embedded and Real-time Systems (WATERS)*, 2015.

[15] K. O. Thabit, "Cache Management by the Compiler," PhD Thesis, Rice University, Houston, USA, 1982.

[16] H. R. Faragardi, K. Sandstr, and T. Nolte, "An Efficient Scheduling of AUTOSAR Runnables to Minimize Communication Cost in Multi-core Systems," in *7th Int. Symp. on Telecommunications*, 2014, pp. 41–48.