International Journal of Automotive Technology, Vol. 16, No. 4, pp. 635–644 (2015)
DOI 10.1007/s12239–015–0065–7

pISSN 1229–9138/ eISSN 1976–3832

GUARANTEEING END-TO-END DEADLINES FOR AUTOSAR-BASED AUTOMOTIVE SOFTWARE

H. YOON[1)] and M. RYU[2)*]

[1)]Department of Electrical Engineering and Computer Engineering, Hanyang University, Seoul 133-791, Korea
[2)]Department of Computer Science and Engineering, Hanyang University, Seoul 133-791, Korea

(Received 23 May 2014; Revised 9 November 2014; Accepted 16 November 2014)

ABSTRACT–Automotive electrical/electronic (E/E) architectures are evolving towards a complex software intensive distributed system. However, current technology and practice in the automotive industry do not adequately address the increasing complexity of software, which hinders accomplishing reliable, maintainable, high-quality software within time and budget constraints. Although automotive open system architecture (AUTOSAR) addresses many software issues such as system software services, application interfaces, and communication middleware, it largely focuses on the implementation aspect without sufficiently addressing the design aspect. In this paper, we present a novel approach to guaranteeing end-to-end deadlines for AUTOSAR-based automotive systems in the early design stage. Our approach, we call zero slack priority assignment (ZSPA), decomposes end-to-end deadlines into local per-task deadlines and finds a feasible scheduling solution leveraging the Audsley's optimal priority assignment algorithm. Our simulations show that ZSPA outperforms existing methods, heuristic optimized priority assignment (HOPA) (Garcia and Harbour, 1995) and the genetic algorithm (GA) (Azketa *et al*., 2011). Specifically, ZSPA shows up to 20% higher success rate in finding feasible solutions than HOPA and GA. The computational complexity of ZSPA is $O(n^2)$, whereas HOPA and GA have unbounded running time.

KEY WORDS : End-to-end deadlines, AUTOSAR, Automotive software, Scheduling, Priority assignment

NOMENCLATURE

τ	: task
S	: a set of tasks
C	: a set of task chains
N	: a set of nodes
R	: release time of task chain
E	: worst-case execution time of task chain
D	: end-to-end deadline of task chain
T	: period and minimum inter-arrvial time of task chain
e	: worst-case execution time of task
d	: deadline of task
o	: offset of task
p	: priority level of task
r	: response time of task
b	: longest time of certain task's critical section
I	: interference time of task
B	: blocking time of task
δ	: slack of task
l	: size of CAN bus message
s	: bit time of CAN bus

**Corresponding author*. e-mail: msryu@hanyang.ac.kr

1. INTRODUCTION

Automotive electrical/electronic (E/E) architectures are evolving towards a complex software intensive distributed system. A high-end car has more than ten million lines of software code running on 70 distributed electronic control units (ECUs), and 90% of advanced automotive functions like electronic stability program (ESP) and adaptive cruise control (ACC) are driven by software. The reason is that electronics has become indispensable to automotive technology and software plays a crucial role in implementing advanced functionality using electronics technology.

However, current technology and practice in the automotive industry do not adequately address the increasing complexity of software, which hinders accomplishing reliable, maintainable, high-quality software within time and budget constraints. Recently, automotive open system architecture (AUTOSAR) has been developed as a standard software architecture, jointly by automobile manufactures, part suppliers and tool vendors. AUTOSAR addresses many software issues including system software services, application interfaces and communication middleware, thereby allowing us to achieve key goals such as component-based development, efficient integration from multiple part suppliers, improved portability and maintainability, etc. However, the AUTOSAR standard largely focuses on the implementation aspect without

sufficiently addressing the design aspect. Thus, there exists a significant lack of practical design methods and guidance to overcome the challenges posed by the increased software complexity.

A key challenge for automotive system development is to guarantee end-to-end deadlines in the early design stage. Automotive functions like ESP and ACC are often implemented as distributed tasks running on different ECUs, while having stringent end-to-end deadlines that must be met to ensure desired control quality, safety and comfort. Figure 1 shows an example in which automotive functions have been partitioned into multiple AUTOSAR tasks with end-to-end deadlines on distributed ECUs. From a scheduling point of view, guaranteeing the end-to-end deadlines creates two major problems. First, end-to-end deadlines should be decomposed into local per-task deadlines that are consistent with the end-to-end deadlines. Second, feasible scheduling priorities should be found and assigned to distributed tasks so that every local per-task deadline is guaranteed by the AUTOSAR scheduler on each ECU (Shin and Sunwoo, 2007). Unfortunately, despite the significant body of research effort in the real-time scheduling community (Baker, 2003; Bake, 2005; Davis and Burns, 2011; Hou *et al*., 1994; Lee, 1994; Liu and Layland, 1973; Monnier *et al*., 1988; Samii *et al*., 2009), this problem of guaranteeing end-to-end deadlines has not received enough attention and remains as a major concern for automotive system design.

Few researchers addressed this distributed scheduling problem with end-to-end deadlines, which is known to be NP-hard (Bettati and Liu, 1922; Tindell *et al*., 1992). Garcia *et al*. proposed a heuristic approach, called heuristic optimized priority assignment (HOPA), that attempts to find feasible scheduling priorities for distributed tasks according to their deadlines. Azketa *et al*. presented a genetic algorithm (GA) for feasible priority assignment as an improvement over HOPA (Garcia and Harbour, 1995). Unfortunately, HOPA and GA have a very high computational demand and the quality of a solution obtained by HOPA or GA in a limited amount of computation time cannot be predicted or guaranteed. There exist some techniques that address a partial aspect of the problem of guaranteeing end-to-end deadlines. Kao *et al.* proposed several heuristic algorithms for decomposing end-to-end deadlines into local per-task deadlines (Kao and Garcia-Molina, 1997). Audsley proposed an optimal priority assignment algorithm for tasks with offsets on a uniprocessor system (Audsley, 1991).

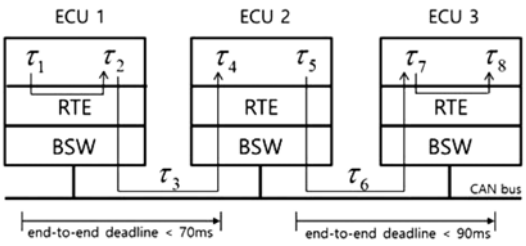

Figure 1. Automotive functions distributed as AUTOSAR tasks on multiple ECUs.

In this paper, we present a novel approach to guaranteeing end-to-end deadlines in AUTOSAR-based automotive systems. We first describe a basic approach, we call fixed slack priority assignment (FSPA), which is a direct combination of the deadline decomposition algorithms by Kao *et al.* (Kao and Garcia-Molina, 1997) and the priority assignment algorithm by Audsley (Audsley, 1991). We then present an improved version, called zero slack priority assignment (ZSPA). While FSPA separately solves each problem, i.e., deadline decomposition and priority assignment, ZSPA treats the two problems at the same time in a unified manner allowing dynamic adjustment of task deadlines during priority assignment. Our simulations show that ZSPA outperforms existing methods, HOPA (Garcia and Harbour, 1995) and GA (Azketa *et al*., 2011). Specifically, ZSPA shows up to 20% higher success rate in finding feasible solutions than previous approaches. The computational complexity of ZSPA is $O(n^2)$, which is much more efficient than the exhaustive algorithm used in HOPA and the genetic algorithm used in GA.

The rest of paper is organized as follows. Section 2 describes the background and models for the problem of guaranteeing end-to-end deadlines. Section 3 introduces FSPA as a basic solution and Section 4 presents ZSPA as an improved version of FSPA. Section 5 provides evaluation results. Finally, Section 6 concludes the paper and discusses future research directions.

2. BACKGROUND AND SYSTEM MODELS

In this section, we first provide a brief overview of the AUTOSAR standard, and then describe our system model.

2.1. Overview of AUTOSAR Standard

The AUTOSAR standard defines a layered architecture that consists of three layers, application, runtime environment (RTE) and basic software (BSW) layers.

The application layer contains software components (SWCs) that are the smallest unit of application software composition. The purpose of SWCs is to support component-based software development, which allows us to compose an automotive function by assembling SWCs. Each SWC possesses ports and runnables. Ports are communication interfaces through which SWCs can communicate with each other. Runnables are executable code that can be invoked through runnable interfaces by other entities such as OS tasks. In AUTOSAR, OS tasks have separate threads of control and are scheduled by the OS scheduler. Note that OS tasks describe the AUTOSAR software's dynamic behavior whereas SWCs describe AUTOSAR software's static structure.

The RTE layer serves as middleware that decouples

application SWCs from the underlying BSW and hardware. This layer handles information exchanges between application SWCs and connects application SWCs to the underlying ECU hardware.

The BSW layer resembles a traditional OS layer. The main purpose of BSW is to offer various OS services to the RTE layer through application programming interfaces (APIs). The main services of BSW include task management, task scheduling using a fixed-priority scheduler or a predefined schedule table, interrupt service routine (ISR) handling and alarm management.

Based on the above architecture, an application can be developed in four steps. The first step is to create input descriptions including information about the inner structure and hardware resource requirements of each SWC, hardware characteristics of each ECU, the overall system structure including network topology and communication channels. The second step is to configure the overall system structure including mapping of SWCs to ECUs and specifying communication messages on in-vehicle network (IVN). The third step is to specify each ECU's functionality. This includes runnable-to-task mapping, task scheduling and configuration of RTE and BSW. The last step is to specify the implementation of each ECU and generate software executables using a set of AUTOSAR tools such as a RTE generator and an OS generator.

2.2. Task Chain Model and Timing Attributes

As mentioned above, automotive functions are often developed as a distributed system and have stringent end-to-end deadline constraints to ensure control quality and vehicle safety. To capture key aspects of typical automotive functions, we model each automotive function as a chain of producer/consumer tasks that execute on distributed processors with an end-to-end deadline constraint (Mitra and Ramanathan, 1993).

Let $S = \{\tau_1, \tau_2, \ldots, \tau_n\}$ and $C = \{C_1, C_2, \ldots, C_m\}$ be a set of tasks and a set of task chains, respectively. A task chain can be expressed by a sequence of tasks in S denoted by $\tau_a \ldots \rightarrow \tau_i \rightarrow \tau_j \ldots \rightarrow \tau_q$. For each task chain C_k, its release time, worst case execution time (WCET), and end-to-end deadline are denoted by R_k, E_k and D_k respectively. Release time R_k is the earliest possible time for task chain C_k to start its execution. Execution time E_k is the sum of WCETs of all member tasks that form C_k. Let e_i be WCET of τ_i. Execution time E_k is then defined by $\sum_{\tau_i \in C_k} e_i$. End-to-end deadline D_k is the maximum time allowed from the release time R_k until the completion of all of its member tasks.

For a task chain C_k that is activated periodically, we denote its period by T_k. All member tasks of C_k are considered to have the same period as C_k. For a task chain C_k that is activated sporadically, we denote its minimum inter-arrival time by T_k and all member tasks of C_k have the same value. For a task chain C_k that runs once, it is considered as an aperiodic task chain and all the tasks in C_k are also considered as aperiodic tasks.

In this paper, we restrict our attention to preemptive static-priority scheduling specified by the AUTOSAR standard. Each task τ_i is associated with a priority denoted by p_i for scheduling on each AUTOSAR-based ECU. Note that AUTOSAR also supports the notion of schedule table, which allows us to create a schedule in advance and later execute the predefined schedule using timers and alarms. Although we do not consider the schedule table scheme in this paper, we believe that our approach is also applicable since our results can help find a feasible schedule.

Note that as shown in Figure 1, two types of communication can exist in a task chain, intra-ECU communication and inter-ECU communication. The cost of intra-ECU communication is negligible when compared to the cost of inter-ECU communication that requires significant amount of time due to IVN protocol processing. In this paper, we ignore the cost of intra-ECU communication and treat inter-ECU communications in a way similar to AUTOSAR tasks. For inter-ECU communications, we assume a priority-based IVN protocol such as controller area network (CAN). In CAN, messages are associated with static priorities and message scheduling is performed based on their priorities. Since this is not much different from the priority assignment and scheduling mechanism for OS tasks running on ECUs, we will not distinguish between OS tasks and CAN messages. Unless noted explicitly, we will refer to both AUTOSAR tasks and CAN messages as tasks and use the same notation τ_i. Similarly, we will not distinguish between ECUs and CAN bus, and refer to them as processing nodes.

2.3. Time-Triggered Task Synchronization

Producer/consumer relationships in a task chain impose precedence constraints—a consumer task can start its execution only after its producer task completes—on its member tasks. In general, there are two ways of satisfying precedence constraints in a distributed system, event-triggered or time-triggered synchronization. An event-triggered approach uses signals or messages to notify a consumer task of its producer's completion. The consumer task is allowed to start its execution only after it confirms the producer's completion by receiving a signal or message from the producer (Tindell and Burns, 1994; Tindell and Clark, 1994).

A time-triggered approach does not rely on signals or messages, but enforces time frames in which precedence-constrained tasks are allowed to run. For a task chain C_k consisting of q tasks, its end-to-end deadline D_k is divided into q local per-task deadlines. Each task τ_i is then assigned a local per-task deadline denoted by d_i. Let o_i be the offset of τ_i, which is the earliest possible release time of τ_i. Let τ_j be the consumer task of τ_i, i.e., $\ldots \rightarrow \tau_i \rightarrow \tau_j \ldots$. The precedence constraint can then be satisfied by having $o_i + d_i \leq o_j$ and ensuring that τ_i and τ_j meet their respective offset and deadline requirements. Note that this time-triggered approach may suffer from unnecessary delay since

a consumer task has to wait until its offset even though the producer task has already completed.

In this paper, we consider a time-triggered approach for the following reasons. First, an event-triggered approach makes it very difficult to accurately analyze the timing behavior of the system due to the pessimism in the worst-case response time analysis (Sun and Liu, 1996). Pessimistic response time analysis often results in over-design and/or under-utilization of system resources, the penalty of which may be much higher than the expected performance gain from prompt task activation in event-triggered approaches. Second, an event-triggered approach involves significant runtime overheads caused by exchanging notification signals or messages between producer and consumer tasks, whereas time-triggered approaches allow simple communication and interface protocols between system components. Third, a time-triggered approach also allows for easier management of complicated timing issues that can arise during the design and implementation of real-time systems (Ryu *et al*., 1997). For example, a time-triggered approach can facilitate early detection of timing-related errors and provide the notion of temporal firewalls for the purpose of fault isolation (Kopetz and Bauer, 2003).

3. PRIORITY ASSIGNMENT UNDER STATIC END-TO-END DEADLINE DECOMPOSITION

Using a time-triggered synchronization scheme requires us to solve two important problems, decomposing end-to-end deadlines into local per-task deadlines and determining feasible scheduling priorities to meet the local per-task deadlines. There has been notable work for each problem. Kao *et al.* proposed several efficient techniques for end-to-end deadline decomposition (Kao and Garcia-Molina, 1997). Audsley proposed an optimal priority assignment algorithm for tasks with offset constraints (Audsley, 1991). In this section, we introduce a basic approach that directly combines these two techniques.

3.1. Static End-to-End Deadline Decomposition

The problem of end-to-end deadline decomposition has been studied by Kao *et al.* (Kao and Garcia-Molina, 1997). The authors proposed two effective policies for end-to-end deadline decomposition, equal slack (EQS) and equal flexibility (EQF).

The EQS policy divides the task chain's slack $D_k - \Sigma_{\tau_i \in C_k} e_i$ equally so that each member task can have its fair share of the task chain's slack. Thus, for a task chain with q tasks represented by $C_k : \tau_1 \ldots \rightarrow \tau_1 \rightarrow \tau_{i+1} \ldots \rightarrow \tau_q$, we can determine task offsets and deadlines as follows.

$$d_i = e_i + \frac{D_k - \Sigma_{\tau_j \in c_k} e_j}{q} \quad (for\ 1 \le i \le q)$$
$$o_i = R_k \ for\ \ i = 1 \text{ and } o_i = o_{i-1} + d_{i-1} \ for\ \ 2 \le i \le q \tag{1}$$

The EQF policy assigns each task a weighted share of the task chain's slack such that $\frac{e_i}{d_i} = \frac{\Sigma_{e_j \in C_k} e_j}{D_k}$ for each member task $\tau_i \in C_k$. Thus, EQF assigns task deadlines in proportion to task execution times.

$$d_i = \frac{e_i}{\Sigma_{\tau_j \in C_k} e_j} D_k \quad (for\ 1 \le i \le q)$$
$$o_i = R_k \ for\ i = 1 \text{ and } o_i = o_{i-1} + d_{i-1} \ for\ \ 2 \le i \le q \tag{2}$$

3.2. Optimal Priority Assignment

Once task offsets and deadlines have been assigned, feasible scheduling priorities need to be determined. To this end, we introduce the Audsley's optimal priority assignment algorithm.

Audsley showed that rate monotonic (RM) and deadline monotonic (DM) priority assignment schemes are not optimal for tasks with offsets (Audsley, 1991). Audsley constructed counter examples that are not schedulable by RM or DM priority assignment. Audsley also showed that those counter examples can be made schedulable through a different priority assignment, and proposed an efficient optimal algorithm. Note that by examining every possible combination of priority assignment, we may find a feasible priority ordering. However, this naïve approach has exponential complexity, since there exist *n!* combinations. Fortunately, Audsley developed an efficient optimal algorithm that examines $n^2 + n$ combinations.

Consider a set of n tasks running on a single processor, denoted by $Z = \{ \tau_1, \tau_2, \ldots, \tau_1, \ldots, \tau_n \}$. The following theorems provide key properties needed to develop an optimal priority assignment algorithm for fixed-priority preemptive scheduling on a uniprocessor system.

Theorem 1. If τ_i is assigned the lowest priority level, n, and is infeasible, no priority assignment function that assigns τ_i priority level n produces a feasible assignment.

Proof. See Theorem 1 in (Audsley *et al.*, 1991).

Theorem 2. If τ_i is assigned the lowest priority level, n, and is feasible, then if a feasible priority ordering for Z exists, an ordering with τ_i assigned the lowest priority exists.

Proof. See Theorem 2 in (Audsley *et al.*, 1991).

Theorem 3. Let the tasks assigned priority levels $i,i+1,\ldots, n$ by assignment function Fx be feasible under that priority ordering. If there exists a feasible priority ordering for Z, there exists a feasible priority ordering that assigns the same tasks to the same priority levels as Fx.

Proof. See Theorem 3 in (Audsley *et al.*, 1991).

An optimal priority assignment algorithm can be easily developed using the above theorems. Based on Theorem 1 and 2, we can assign priority levels starting from the lowest priority. We first seek a task that is feasible at the lowest priority level, n, treating other tasks as higher priority tasks. If we can find one, we proceed to find other task that is feasible at the next higher priority, $n-1$, also treating the

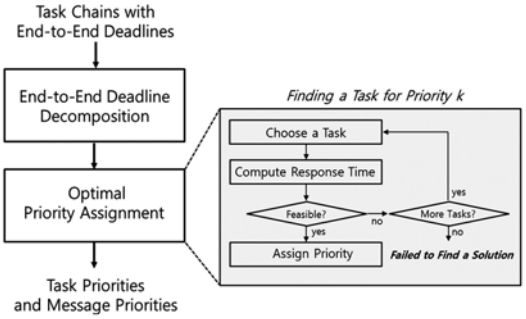

Figure 2. Overall design of FSPA.

remaining priority-unassigned tasks as higher priority tasks. This priority assignment continues until we reach the highest priority. In doing so, if we fail to find a task that is infeasible at a certain priority level, then we conclude that the given task set is not feasible.

Note that the optimal algorithm above has been developed for a single processor preemptive scheduling while we have multiple processors, i.e., ECUs, and IVN messages are nonpreemptible. In spite of these differences, we can directly apply the optimal algorithm to our problem. First, we can run the optimal algorithm separately for each ECU and IVN, since tasks running on different ECUs can be treated independently of the others by virtue of end-to-end deadline decomposition and time-triggered synchronization. Second, George *et al*. (1996) showed that the above algorithm is optimal even for non-preemptive scheduling.

A direct combination of the Audsley's algorithm with one of the deadline decomposition methods provides a simple solution, we call fixed slack priority assignment (FSPA). Let S_k be the set of tasks allocated to k th ECU or IVN. Figure 2 shows the overall design of FSPA and the following pseudo code shows the FSPA algorithm with EQS.

Algorithm FSPA-EQS

```
begin
  for each processing node N_j with an n-task set S_j
    for each task τ_i in S_j
      determine d_i according to EQS;
    endfor
    for each priority level k from n to 1
      for each task τ_i in S_j
        if τ_i is schedulable at priority level k
          p_i = k;
          S_j = S_j − {τ_i};
          break;
        endif
      endfor
      if no task is schedulable at priority level k
        exit;
      endif
    endfor
end
```

4. PRIORITY ASSIGNMENT WITH DYNAMIC SLACK ADJUSTMENT

In this section, we describe zero slack priority assignment (ZSPA), which can significantly improve upon FSPA.

4.1. Dynamic Slack Adjustment

Despite the optimality of Audsley's algorithm, the previous FSPA method may fail to find a feasible solution even if one exists. The main reason is that FSPA treats the two problems, end-to-end deadline decomposition and priority assignment, separately rather than having the two problems tangled together. Once all task deadlines are assigned, they remain fixed during the priority assignment phase. This may prevent us from finding a feasible solution that can be found with a slight change to some local per-task deadline.

We can overcome this limitation by dynamically adjusting local per-task deadlines when finding a task for each priority level. A key idea is to take advantage of exact slack time that each task may produce. For each task, the exact slack time is defined as the maximum amount of time the task can be delayed without missing its deadline. When a task is assigned a feasible priority, we can compute its safe slack time via worst-case response time analysis. We can then reduce the task's deadline to the obtained worst-case response time, which allows us to save the slack time for later use. Later, when some task deems to miss its deadline, we can use the saved slack time to extend the deadline so that the task can be made schedulable.

4.2. Task Selection Criterion

Reallocation of slack time offers more opportunities to find a feasible solution, but this also raises another question about how to effectively manage slack time.

Our idea is to maintain the task chain's slack, the sum of saved slack from the member tasks, as much as possible when selecting a task for priority assignment. Specifically, we choose the task with the largest slack so that the task

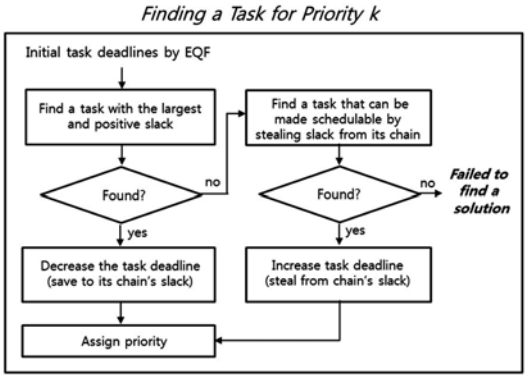

Figure 3. Overall flow of ZSPA.

chain's slack can be maximized. For instance, when there are multiple candidate tasks that are schedulable at priority level k, we choose the one that would produce the largest slack, and then we save the task's slack in its task chain's slack. Slack saving can be done simply by decreasing the task's deadline and increasing the task chain's slack by the same amount.

When there is no candidate task that is schedulable at priority level k, i.e., all tasks have negative slack values, we attempt to find a task that can be made schedulable by taking some slack from its task chain's slack. If we have sufficient slack in the task chain, then we simply extend the task's deadline by the absolute amount of the task's negative slack so that the task can be made schedulable, and decrease the task chain's slack by the same amount.

When some task cannot be made schedule because its chain's slack is not enough, the algorithm terminates concluding that the given task set is not schedulable. The overall design of the proposed method is given in Figure 3.

Let δ_i be the slack of task τ_i. Now our priority assignment algorithm is formally described as below.

Algorithm ZSPA

```
begin
  for each task τ_i in S = {τ_1, τ_2, …, τ_n}
    determine d_i according to EQF;
  endfor
  let Q = φ and T = S;
  for each task chain C_j in C = {C_1, C_2, …, C_m}
    Δ_j = 0;
  endfor
  for each priority level k from n to 1
    while T ≠ φ
      find a task τ_x with the maximum slack;
      T = T – {τ_x};
      C_y = find_task_chain(τ_x);
      if τ_x has non-negative slack
        p_x = k;
        Δ_y = Δ_y + δ_x ;
        d_x = d_x – δ_x ;
        T = T + Q;
        Q = φ ;
        break;
      else      /* τ_x has negative slack */
        d_x = d_x + Δ_y ;
        if τ_x is schedulable
          Δ_y = 0;
          p_x = k;
          Δ_y = Δ_y + δ_x ;
          d_x = d_x – δ_x ;
          T = T + Q ;
          Q = φ ;
          break;
        else      /* τ_x is not schedulable */
          d_x = d_x – Δ_y ;
          Q = Q+{τ} ;
        endif
      endif
    endwhile
    if no task is schedulable at priority level k
      exit;
    endif
  endfor
end
```

4.3. Schedulability Test and Response Time Analysis

The above priority assignment algorithm requires a schedulability test to check if a chosen task is schedulable at priority level k. An exact schedulability test is required for the above algorithm to remain optimal, and there exist exact schedulability tests for tasks with offsets (Audsley *et al.*, 1991). However, the computational complexity of exact schedulability tests is exponential, which in turn would make the complexity of above priority assignment algorithm exponential.

In this paper, we use an approximate schedulability test based on the notion of critical instant, which has complexity of O(n). With time-triggered synchronization, we only need to check if every task τ_i in task chain C_k completes within its time frame $[o_i, o_i + d_i]$.

Let N_j be the ECU or IVN where τ_i is allocated and S_j be the set of all tasks allocated to N_j. To check if τ_i is schedulable, we just need to consider τ_i 's execution interval $[o_i, o_i + d_i]$. During this interval, other tasks $\tau_i \in S_j$ may interfere with τ_i's execution. Let I_i be the interference time caused by other tasks $\tau_i \in S_j$. Furthermore, as AUTOSAR uses priority ceiling protocol (PCP) for task synchronization, a task may experience synchronization blocking denoted by B_i^{sync}. The response time r_i of τ_i can then be expressed as the sum of its own execution time e_i, interference time I_i and synchronization blocking B_i^{sync}.

$$r_i = e_i + I_i + B_i^{sync} \tag{3}$$

Since low priority tasks cannot affect τ_i 's execution during the interval $[o_i, o_i + d_i]$, only the high priority tasks can contribute to interference time I_i. Let $S_j^{high(\tau)} \in S_j$ be the set of tasks that have higher priorities than τ_i. The interference time I_i can then be bounded by[1]

$$I_i \le \sum_{\tau_k \in S_j^{high(\tau_i)}} \left\lceil \frac{d_i}{T_x} \right\rceil \cdot e_x \tag{4}$$

Synchronization blocking B_i^{sync} can also be determined by examining critical section durations. Let b_x be the longest duration of critical sections accessed by $\tau_x \in S_j$. Synchronization blocking B_i^{sync} is bounded by

[1] Note: $[x]$ evaluates to the smallest integer equal to or greater than x.

$$B_i^{sync} \le \max\{b_x | \tau_x \in S_j, \tau_x \neq \tau_i\} \quad (5)$$

Thus, we can compute the worst-case response time of τ_i using Equation (3). Now we can check the schedulability of $C = \{C_1, C_2, \ldots, C_m\}$ by running the following test for each task chain $C_k \in C$.

$$r_i \le d_i \quad \text{for every} \quad \tau_i \in C_k \quad (6)$$

For IVN messages that are nonpreemptive, we introduce a different schedulability test. Since a high priority task cannot preempt a low priority task, it may be delayed by at most the maximum of low priority task execution times. Let B_i^{block} be the maximum possible blocking time of τ_i due to nonpreemptibility.

$$B_i^{block} = \max\{e_x | \tau_x \in C_k \text{ and } \tau_x \neq \tau_i\} \quad (7)$$

$$r_i = e_i + I_i + B_i^{block} \quad (8)$$

5. PERFORMANCE EVALUATION

We implemented four different priority assignment methods including FSPA and ZSPA, and compared their performance via simulations. For our experiments, we implemented mHOPA and mGA, slightly modified versions of HOPA and GA, respectively. mHOPA initially decomposes end-to-end deadlines using the EQF method. It then iteratively performs DM priority assignment and end-to-end deadline decomposition based on response time calculation until it finds a feasible solution. mGA uses 50 priority assignment solutions obtained by mHOPA as an initial population for genetic operations. mGA applies genetic operations including selection, mutation, and crossover to those 50 initial populations producing new 50 candidate solutions.

The original HOPA and GA models do not explicitly have the notion of local per-task deadlines and perform holistic end-to-end schedulability analysis. For instance, HOPA derives task deadlines by decomposing end-to-end deadlines, but the purpose of this is to use them for DM priority assignment. Once a priority assignment is done, HOPA checks end-to-end deadlines rather than examining individual task deadlines. For our experiments, we slightly modified original HOPA and GA methods and performed schedulability tests on a local per-task deadline basis. We then ran mHOPA and mGA only for 1,200 seconds since they have unbounded running times.

Table 1. Parameters for task chains.

Length	Execution time	Period	Deadline
[1, 7]	$\sum_{\tau_i \in C} e_i$	[400, 600]	P

Table 2. Parameters for tasks.

Task location	Execution time	Period	Block time
ECU	[10, 50]	P	[0, 5]
IVN	[5, 10]	P	[0, 5]

We randomly generated synthetic workloads varying key numbers of realistic workload. The number of ECUs was randomly specified from a range of 5 through 10. The number of task chains was set as 10. The number of tasks in each chain, total sum of execution times, period, and deadline were randomly chosen from specified ranges shown in Table 1. For each task, the mapping to ECU, execution time, and blocking duration were also randomly chosen from the specified ranges shown in Table 2.

Figure 4 shows the percentage of schedulable solutions found by each method over varying utilization values of task chains. Here, the utilization of task chain C_k is defined

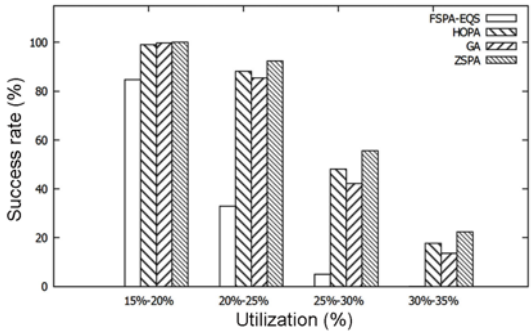

Figure 4. Performance over varying utilization values.

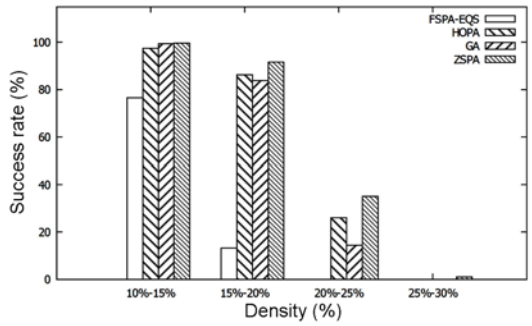

Figure 5. Performance over varying density values.

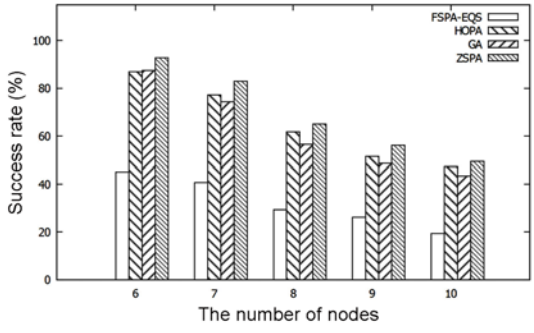

Figure 6. Performance over varying number of nodes.

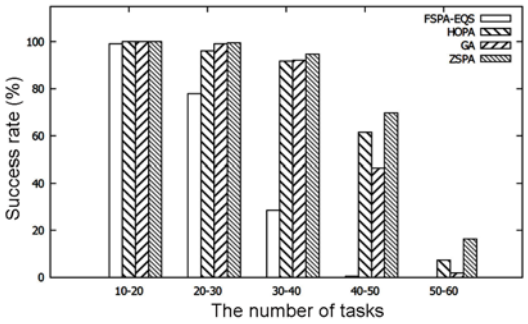

Figure 7. Performance over varying number of tasks.

as E_k/T_k. In our experiments, all the methods show performance decreases as the utilization increases, and there exist only a slight performance difference between the methods when the utilization is low, i.e., less than 20%. However, as the utilization increases, the performance gap increases and ZSPA outperforms other methods up to 8% and 13% higher than mHOPA and mGA, respectively, when the utilization is between 25% and 30%. On the other hand, the performance of FSPA rapidly drops as the utilization increases, for example 0.58% success rate when the utilization is higher than 30%. Similar results can be observed for density values as shown in Figure 5.

Figure 6 and 7 compare the performance in terms of the number of processors and the number of tasks. In all cases, ZSPA outperforms other methods by up to 20% and 10% higher than mGA and mHOPA (when the number of tasks is 40 to 50), respectively.

We also conducted a case study to evaluate our approaches with anti-lock brake system (ABS) and engine control system (ECS) examples (Asberg *et al.*, 2009). The examples have been adapted into our system model as shown in Figure 8. There are three task chains, ABS-control, ECS-injection, and ECS-ignition, which run on five distributed ECUs and CAN bus. The tasks of each task chain and their attributes are given in Table 3, 4, and 5, respectively.

In the original examples, Asberg *et al.* did not consider

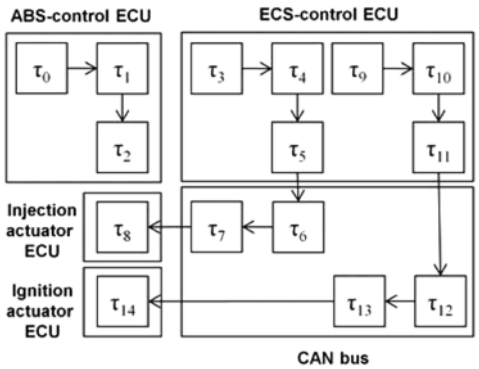

Figure 8. Target system model for ABS and ECS.

Table 3. ABS task chain and task attributes.

Name	WCET (ms)	ECU
Sensor (τ_0)	0.1	ABS-control
Control (τ_1)	0.5	ABS-control
Actuator (τ_2)	0.1	ABS-control

Table 4. ECS-injection task chain and task attributes.

Name	WCET (ms)	ECU
Engine (τ_3)	0.300	ECS-control
Throttle (τ_4)	0.200	ECS-control
Injection (τ_5)	2.500	ECS-control
CAN message #1 (τ_6)	1.282	CAN bus
CAN message #2 (τ_7)	1.282	CAN bus
Injection-Actuator (τ_8)	1200	Injection-actuator

Table 5. ECS-ignition task chain and task attributes.

Name	WCET (ms)	ECU
Engine (τ_9)	0.300	ECS-control
Throttle (τ_{10})	0.200	ECS-control
Ignition (τ_{11})	2.500	ECS-control
CAN message #1 (τ_{12})	1.282	CAN bus
CAN message #2 (τ_{13})	1.282	CAN bus
Ignition-Actuator (τ_{14})	3.5	Ignition-actuator

inter-ECU communication. In this case study, we included a CAN bus and assumed appropriate CAN messages for inter-ECU communications. Note that communication delay of a CAN message can be computed by the following (Tindell and Burns, 1994).[2]

$$e = \left(\left\lfloor \frac{34 + 8 \cdot l}{5} \right\rfloor + 47 + 8 + l \right) \cdot s \tag{9}$$

where l denotes the size of message and s is the bit time of CAN bus.

For this study, we have considered 1 Mbit/sec CAN bus and 16 bytes for each CAN message payload, which leads to the worst-case communication delay of 1.282 milliseconds for a single CAN message transmission.

To evaluate the efficacy of each priority assignment algorithm, we calculated and compared worst-case response times of solutions produced by different priority assignment algorithms. Before describing the comparison, let us first define density of task chain C_k as the sum of WCETs of its member tasks divided by the task chain's end-to-end deadline, i.e., E_k/D_k. Thus, a high density value indicates that the task chain has a tight end-to-end deadline, and vice

[2] Note: $\lfloor x \rfloor$ evaluates to the largest integer equal to or smaller than x.

Table 6. Worst-case response time (ms) at density 0.5.

Task-chain	FSPA-EQS	mHOPA	mGA	ZSPA
ABS	not schedulable	1.266	1.259	1.000
ECS-injection	not schedulable	33.117	32.642	26.656
ECS-ignition	not schedulable	17.075	16.802	17.310

Table 7. Worst-case response time (ms) at density 0.4.

Task-chain	FSPA-EQS	mHOPA	mGA	ZSPA
ABS	not schedulable	1.487	1.536	1.000
ECS-injection	not schedulable	41.751	36.209	24.092
ECS-ignition	not schedulable	20.185	19.729	21.156

Table 8. Worst-case response time (ms) at density 0.3.

Task-chain	FSPA-EQS	mHOPA	mGA	ZSPA
ABS	1.788	1.925	2.026	1.000
ECS-injection	51.714	36.971	45.344	22.810
ECS-ignition	26.684	19.566	24.132	21.156

versa. We started with 0.5 as an initial density value and imposed a corresponding end-to-end deadline to each task chain, i.e., an end-to-end deadline twice the sum of WCETs. We then ran priority assignment algorithms, obtained priorities and calculated worst-case response times. We repeated these steps with different density values, 0.4, 0.3, 0.2, and 0.1. The following tables 6 – 9 show the worst-case response times of each priority assignment algorithm. As seen in the tables, ZSPA produces better response times in most cases.

6. CONCLUSION

In this paper, we presented a novel approach to guaranteeing end-to-end deadlines in automotive systems. Our approach, we call ZSPA, treats deadline decomposition and priority assignment in a unified manner leveraging the Audsley's optimal algorithm.

Table 9. Worst-case response time (ms) at density 0.2.

Task-chain	FSPA-EQS	mHOPA	mGA	ZSPA
ABS	2.566	3.100	2.767	1.000
ECS-injection	73.109	39.820	50.292	22.810
ECS-ignition	39.274	31.320	26.972	21.156

Table 10. Worst-case response time (ms) at density 0.1.

Task-chain	FSPA-EQS	mHOPA	mGA	ZSPA
ABS	4.900	5.132	5.854	1.000
ECS-injection	149.294	89.910	92.602	23.810
ECS-ignition	77.044	64.013	63.310	24.656

Our simulations show that ZSPA outperforms existing methods, HOPA (Garcia and Harbour, 1995) and GA (Azketa *et al.*, 2011) as well as FSPA. ZSPA shows up to 20% higher success rate in finding feasible solutions than previous approaches. The computational complexity of ZSPA is $O(n^2)$, which is more efficient than the exhaustive algorithm used in HOPA and GA.

There remain several future research directions. First, we will explore a more accurate method for schedulability analysis than the approximate one used in this work. Second, we are also planning to implement a timing design tool for automotive system development that can be easily integrated with existing AUTOSAR tools.

ACKNOWLEDGEMENT–This research was partly supported by the MSIP (Ministry of Science, ICT&Future Planning), Korea, under the ITRC (Information Technology Research Center) support program (NIPA-2013-H0301-13-2007) supervised by the NIPA (National IT Industry Promotion Agency), and the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2014R1A1A20558 13).

REFERENCES

Asberg, M., Behnam, M., Nemati, F. and Nolte, T. (2009). Towards hierarchical scheduling in AUTOSAR. *Proc. Emerging Technologies and Factory Automation*, 1–8.

Audsley, N. C. (1991). Optimal Priority Assignment and Feasibility of Static Priority Tasks with Arbitrary Start Times. Dept. Computer Science, University of York. Technical Report YCS 164.

Audsley, N. C., Bruns, A., Richardson, M. F. and Wellings, A. J. (1991). Hard real-time scheduling: The deadline-monotonic approach. *Proc. IEEE Workshop on Real-Time Operating Systems and Software*, 133–137.

Azketa, E., Uribe, J. P., Marcos, M., Almeida, L. and Gutierrez, J. J. (2011). Permutational genetic algorithm for the optimized assignment of priorities to tasks and messages in distributed real-time systems. *Proc. IEEE 10th Trust, Security and Privacy in Computing and Communications*, 958–965.

Baker, T. P. (2003). Multiprocessor EDF and deadline monotonic schedulability analysis. *Proc. IEEE 24th Real-Time Systems Symp.*, 120–129.

Baker, T. P. (2005). Comparison of Empirical Success Rates of Global vs. Partitioned Fixed-Priority and EDF Scheduling for Hard Real Time. Technical Report. TR-050601.

Bettati, R. and Liu, J. W. S. (1992). End-to-end scheduling to meet deadlines in distributed systems. *Proc. 12th Distributed Computing Systems*, 452–459.

Davis, R. I. and Burns, A. (2011). A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.* **43**, **4**, Article No. 35.

Garcia, J. J. G. and Harbour, M. G. (1995). Optimized priority assignment for tasks and messages in distributed

hard real-time systems. *Proc. 3rd Workshop on Parallel and Distributed Real-Time Systems*, 124–132.

George, L., Rivierre, N. and Spuri, M. (1996). Preemptive and Non-Preemptive Real-Time Uniprocessor Scheduling. INRIA Research Report, No. 2966.

Hou, E. S. H., Ansari, N. and Hong, R. (1994). A genetic algorithm for multiprocessor scheduling. *IEEE Trans. Parallel and Distributed Systems* **5**, **2**, 113–120.

Kao, B. and Garcia-Molina, H. (1997). Deadline assignment in a distributed soft real-time system. *IEEE Trans. Parallel and Distributed Systems* **8**, **12**, 1268–1274.

Kopetz, H. and Bauer, G. (2003). The time-triggered architecture. *Proc. IEEE* **91**, **1**, 112–126.

Lee, S. K. (1994). On-line multiprocessor scheduling algorithms for real-time tasks. *Proc. IEEE Region 10's 9th Annual Int. Conf.*, 607–611.

Liu, C. L. and Layland, J. W. (1973). Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* **20**, **1**, 46–61.

Mitra, H. and Ramanathan, P. (1993). A genetic approach for scheduling non-preemptive tasks with precedence and deadline constraints. *Proc. 26th System Sciences*, **2**, 556–564.

Monnier, Y., Beauvais, J. P. and Deplanche, A. M. (1998). A genetic algorithm for scheduling tasks in a real-time distributed system. *Proc. 24th Euromicro Conf.*, **2**, 708–714.

Ryu, M., Hong, S. and Saksena, M. (1997). Streamlining real-time controller design: from performance specifications to end-to-end timing constraints. *Proc. IEEE Real-Time Technology and Applications Symp.*, 91–99.

Samii, S., Yanfei, Y., Zebo, P., Eles, P. and Yuanping, Z. (2009). Immune genetic algorithms for optimization of task priorities and flexRay frame identifiers. *Proc. IEEE 15th Embedded and Real-Time Computing Systems and Applications*, 486–493.

Shin, M. and Sunwoo, M. (2007). Optimal period and priority assignment for a networked control system scheduled by a fixed priority scheduling system. *Int. J. Automotive Technology* **8**, **1**, 39–48.

Sun, J. and Liu, J. W. S. (1996). Bounding completion times of jobs with arbitrary release times and variable execution times. *Proc. IEEE 17th Real-Time Systems Symp.*, 2–12.

Tindell, K. W., Burns, A. and Wellings, A. J. (1992). Allocating hard real-time tasks: An NP-hard problem made easy. *J. Real-Time Systems* **4**, **2**, 145–165.

Tindell, K. and Burns, A. (1994). *Guaranteed Message Latencies for Distributed Safety-critical Hard Real-time Control Networks*. Department of Computer Science. University of York. York.

Tindell, K. W. and Clark, J. (1994). Holistic schedulability analysis for distributed hard real-time systems. *J. Microprocessing and Microporgramming – Parallel Processing in Embedded Real-time Systems* **40**, **2–3**, 117–134.