Title

Addressing Extensibility and Fault Tolerance in CAN-based Automotive Systems

ISBN
978-1-4503-4984-0

Authors
Liang, H.
Wang, Z.
Zheng, B.
et al.

Publication Date
2017-10-19

Addressing Extensibility and Fault Tolerance in CAN-based Automotive Systems

Special Session Paper

Hengyi Liang
University of California, Riverside
Riverside, CA
hlian010@ucr.edu

Zhilu Wang
University of California, Riverside
Riverside, CA
zwang055@ucr.edu

Bowen Zheng
University of California, Riverside
Riverside, CA
bzhen003@ucr.edu

Qi Zhu
University of California, Riverside
Riverside, CA
qzhu@ece.ucr.edu

ABSTRACT

The design of automotive electronic systems needs to address a variety of important objectives, including safety, performance, fault tolerance, reliability, security, extensibility, etc. To obtain a feasible design, timing constraints must be satisfied and latencies of certain functional paths should not exceed their deadlines. From functionality perspective, soft errors caused by transient or intermittent faults need to be detected and recovered with *fault tolerance* techniques. Moreover, during the lifetime of a vehicle design or even the same car, updates are often needed to add new features or fix bugs in existing ones. It is therefore critical to improve the design *extensibility* for accommodating such updates without incurring major redesign and re-verification cost. In this work, we discuss the metrics for measuring latency, fault tolerance and extensibility, and present a simulated annealing based algorithm to search the design space with respect to them. Experimental results on industrial and synthetic examples demonstrate clear trade-offs among these objectives, and hence the importance of quantitatively analyzing such trade-offs and exploring the design space with automation tools.

CCS CONCEPTS

• **Computer systems organization → Embedded systems**; **Embedded software**;

ACM Reference Format:
Hengyi Liang, Zhilu Wang, Bowen Zheng, and Qi Zhu. 2017. Addressing Extensibility and Fault Tolerance in CAN-based Automotive Systems. In *Proceedings of NOCS '17, Seoul, Republic of Korea, October 19–20, 2017,* 8 pages. https://doi.org/10.1145/3130218.3130233

1 INTRODUCTION

The design of automotive electronic systems has become increasingly challenging due to large design space and stringent design requirements. The development of autonomous and semi-autonomous features, as well as vehicle connectivity functionality, requires more complex automotive software and hardware. In addition, the underlying architecture platform is shifting from the traditional *federated architecture*, where each function is deployed to one ECU (Electronic Control Unit) and provided as a black-box by a Tier-1 supplier, to the *integrated architecture*, where one function can be distributed over multiple ECUs and multiple functions can be supported by one ECU.

Model-based design (MBD) methodology has been proposed to address the deign challenges in complex systems such as vehicles and avionic systems [13, 14]. In MBD, system functionality is first captured with formal or semi-formal models for early-stage analysis and validation. These functional models are then mapped onto an architectural platform (often also captured with models) for software or hardware implementation. For automotive electronic systems, this mapping/synthesis process involves generating software tasks from functional models (sometimes through another layer of runnables), allocating tasks onto ECUs connected with buses (such as CAN [7, 15, 22] or FlexRay [3, 16]), and scheduling the execution of tasks and the transmission of bus messages (Figure 1). During this process, a variety of design objectives, such as safety, performance, fault tolerance, reliability, security and extensibility, need to be addressed.

Extensibility: A major challenge in vehicle design is to cope with software and hardware evolutions over the lifetime of a design or across multiple versions in the same product family or even for the same car. Updates such as adding new application software, reallocating some software among ECUs, or adding a new ECU are needed to fix bugs and provide new functionality. Due to the fast development of automotive applications, such updates (especially software updates) are expected to be more frequent. For instance, Tesla has already been able to carry out regular software updates over-the-air since version 8.1 [23].

However, small changes in software and hardware may cause big and unexpected changes in system timing and functionality. It is often necessary to re-verify and re-certify the entire system, which

NOCS '17, October 19–20, 2017, Seoul, Republic of Korea

ACM ISBN 978-1-4503-4984-0/17/10…$15.00
https://doi.org/10.1145/3130218.3130233

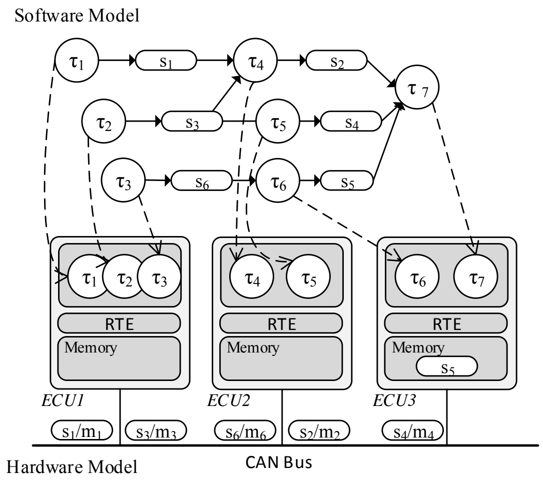


Figure 1: Mapping of software tasks onto ECUs connected with a CAN bus. The signals communicated between tasks are mapped to either local communication through memory or CAN messages.

could lead to prohibitively expensive costs and undermine system availability and reliability. Therefore, it is important to improve the extensibility of designs so that future updates can be accommodated without incurring major redesign and re-verification cost. This is a challenging goal, especially due to the sharing and contention of software functions over limited computation and communication resources.

Fault tolerance: Soft errors caused by transient or intermittent faults have become a major design concern, because of the continuous scaling of technology, high energy cosmic particles and radiation from the application environment [2, 24]. Fault tolerance techniques are greatly needed to detect and recover errors such as application crashes, illegal branches and silent data corruption. In this paper, similarly as in [8, 26], we focus on two categories of error detection techniques: embedded error detection (EED) and explicit output comparison (EOC). More specifically, EED includes a variety of error detection techniques such as instruction signature checking, control flow check (CFC) and watchdog timers [17]. EOC detects errors through explicit redundancy of task execution. For instance, the same program can be executed twice and output mismatch indicates occurred error(s) [8]. Choosing EOC or EED techniques for specific tasks could significantly improve system's ability to tolerate soft errors.

In addition to extensibility and fault tolerance, there are often timing constraints that must be satisfied to ensure functional correctness and system safety, such as task execution deadlines, message transmission deadlines, and latency deadlines along functional paths.

In this work, we discuss the metrics for measuring extensibility, fault tolerance and latency, optimize them in task allocation and scheduling, and analyze their trade-offs. We consider automotive systems that are based on CAN, the prevalent bus protocol currently in vehicles. For tasks on the same ECU, the communication is through local memory and very fast. For tasks on different ECUs, the communication is through CAN bus messages/frames and the transmission time is much longer.

Intuitively, maximizing extensibility or fault tolerance may lead to a more "balanced" task allocation and therefore more bus messages and longer path latencies. To quantitatively evaluate such trade-offs, we first define timing models for software tasks, messages and schedulability constraints, and metrics for extensibility and fault tolerance. We then optimize these metrics with a simulated annealing based approach, and conduct experiments with industrial and synthetic examples.

The rest of the paper is organized as follows. In Section 2, we discuss previous work on extensibility and fault tolerance. In Section 3, we introduce our system models on timing/latency, extensibility and fault tolerance. In Section 4, we demonstrate the trade-offs among these objectives with an illustrating example, and then introduce our simulated annealing-based algorithm for optimizing them. We present experimental results and discuss our findings in Section 5, and conclude the paper in Section 6.

2 RELATED WORK

In the literature, a number of studies have addressed robustness, scalability, flexibility and extensibility of real-time embedded system. The notions of these objectives could sometimes be obfuscated since they all relate to system's capability of accommodating changes (which could come from variations or updates). For instance, in [25], scalability refers to how well a system can handle task execution time increases. In [1], flexibility describes system's ability to add additional tasks without impeding existing ones.

Various viewpoints and definitions have also been proposed for system extensibility. In [25], Yerraballi et al. develop a method to find an optimal execution time scaling factor for all tasks in a given subset while ensuring system schedulability. In [21], novel definitions of sustainability and extensibility for FlexRay-based communication systems are presented, which can then be combined with CAN-based system. In [10], although no formal definition of extensibility is provided, an original approach utilizing contract-based design is proposed to negotiate among contracts for software updates. In this work, we adopt the task-level extensibility metric from [27], which measures how much task execution time can be increased without violating design constraints.

Regarding fault tolerance, many error detection techniques, such as triple modular redundancy, watchdog timers and instruction signature checking, have been proposed [4, 5, 11, 12, 17–20]. For instance, in [12, 20], Izosimov et al. employ process re-execution and replication to tolerate transient faults and then extend their algorithm by checkpointing with rollback recovery. In [11], they develop a heuristic algorithm to trade-off between hardware hardening and re-execution in software. In [4, 5], Burns et al. propose schedulability analysis and priority assignment with embedded error detection techniques.

In our previous work [26], we formulate the impact of EOC and EED on system timing for different platform configurations. An

MILP (mixed integer linear programming) model is then developed to explore task allocation and scheduling, together with the selections of error detection techniques for individual tasks.

3 SYSTEM MODEL

In our system model, the CAN-based architectural platform includes a set of p ECUs $E = \{e_1, e_2, \dots, e_p\}$ connected through a CAN bus. The functional model is represented as a task graph $\mathcal{G} = \{\mathcal{T}, \mathcal{S}\}$, where $\mathcal{T} = \{\tau_1, \tau_2, \dots, \tau_n\}$ is the set of tasks and $\mathcal{S} = \{s_1, s_2, \dots, s_m\}$ is the set of signals that impose data dependency and execution order among tasks.

We assume all tasks are invoked periodically and scheduled based on static priorities with preemptions allowed. Each task τ_i has its own activation period T_{τ_i}, worst-case execution time (WCET) c_{τ_i} and priority p_{τ_i}. If two tasks are allocated to the same ECU, signals are transmitted through local memory and we assume the communication delay is negligible. If two dependent tasks are mapped onto different ECUs, data will be exchanged through messages/frames on the CAN bus. The set of CAN messages is denoted as $\mathcal{M} = \{m_1, m_2, \dots, m_q\}$.

In the task graph, a path is an interleaving sequence of tasks and signals denoted as $p = [\tau_{r_1}, s_{r_1}, \tau_{r_2}, s_{r_2}, \dots, s_{r_{k-1}}, \tau_{r_k}]$. τ_{r_1}, the source node of the path, is usually triggered by external events such as sensor inputs. The sink node τ_{r_k} is often the task that activates actuators. It is possible that multiple paths exist between a source task and a sink task.

3.1 Worst-case End-to-end Path Latency

We define worst case end-to-end latency l_p of a path p as the maximum time delay needed for the input changes on the source node to be propagated to the outputs of the sink node. To ensure system safety and performance, a deadline d_p may be imposed on l_p, i.e. $l_p \leq d_p$. The computation of l_p requires the computation of worst-case response time for tasks and messages along the path, as explained in below.

Task worst-case response time: In our model, tasks running on the same ECU are scheduled based on static priorities with preemptions (commonly supported by OSEK standard and its derivatives). The execution of a task is subject to the interferences from higher priority tasks on the same ECU. Therefore, the worst-case response time r_{τ_i} of a task τ_i, which represents the longest time delay needed to complete the task after its activation, can be calculated as follows (similarly as in [9, 28]):

$$r_{\tau_i} = c_{\tau_i} + \sum_{\tau_j \in hp(\tau_i)} \left\lceil \frac{r_{\tau_i}}{T_{\tau_j}} \right\rceil c_{\tau_j} \tag{1}$$

where $hp(\tau_i)$ denotes the set of higher priority tasks on the same ECU. The second term represents the interferences from these higher priority tasks within the response time. This formula can be solved with an iterative numerical method.

Message worst-case response time: In our model, when two tasks communicating through signals are allocated to different ECUs, their communication signals are packed into messages and transmitted over the CAN bus. We further assume each signal s_i is mapped to its own message m_i. The transmission delays of these messages contribute significantly to the path latencies, and can be calculated similarly as tasks. Slightly different from the preemptive task scheduling policy though, CAN bus employs a fixed priority non-preemptive scheduling. Thus, a CAN message may suffer from additional blocking delay caused by lower priority messages, which can be approximated with the largest possible transmission time among all messages transfered on the same CAN bus. Equation (2) below is the formula for calculating message worst-case response time r_{m_i}, where B_{max} is the largest blocking time and c_{m_i} is the worst-case transmission time of the messages.

$$r_{m_i} = c_{m_i} + B_{max} + \sum_{m_j \in hp(m_i)} \left\lceil \frac{r_{m_i} - c_{m_i}}{T_{m_j}} \right\rceil c_{m_j} \tag{2}$$

Path latency: The worst-case end-to-end path latency l_p of path p is the summation of the periods and worst-case response times of all tasks and global signals (i.e., signals that are packed into CAN messages) on the path, as shown below in Equation (3). GS is the set of global signals. Note that in our model, a global signal has the same worst-case response time as its corresponding message, i.e. $r_{s_i} = r_{m_i}$. The periods are taken into account because of the asynchronous communication nature.

$$l_p = \sum_{\tau_i \in p} (r_{\tau_i} + T_{\tau_i}) + \sum_{s_i \in p \wedge s_i \in GS} (r_{s_i} + T_{s_i}) \tag{3}$$

Schedulability: In this work, a system is *schedulable* if all the timing constraints shown below in (4) to (6) are met. Constraint (4) ensures that the response time of every task is not greater than its deadline, which equals to its period in our model. Similarly, Constraint (5) ensures that every message is transmitted within its period. Constraint (6) ensures that the end-to-end latency of every path will not exceed its deadline.

$$\forall \tau_i \in \mathcal{T}, r_{\tau_i} \leq T_{\tau_i} \tag{4}$$

$$\forall m_j \in \mathcal{M}, r_{m_j} \leq T_{m_j} \tag{5}$$

$$\forall p_k \in \mathcal{P}, l_{p_k} \leq d_{p_k} \tag{6}$$

3.2 Task Level Extensibility

We adopt the task level extensibility metric from [27], which measures how much task WCET can be increased without violating design constraints. More specifically, we calculate system extensibility as the weighted sum of each task's maximum possible increase of its WCET:

$$\mathcal{E} = \frac{1}{|\mathcal{T}|} \sum_{\tau_i \in \mathcal{T}} w_{\tau_i} \frac{\Delta c_{\tau_i}}{T_{\tau_i}} \tag{7}$$

where w_{τ_i} is a predetermined value that indicates how likely a task's WCET might be increased in future updates. Δc_{τ_i} is the maximum possible increase of task WCET c_{τ_i} without violating design constraints (i.e., schedulability constraints (4) to (6) in this work), while all other system configurations remain unchanged.

A binary search based algorithm is used to compute the extensibility, as shown in Algorithm 1. In this algorithm, $\mathcal{E}$ denotes the system extensibility and is initialized to zero. For every task τ_i, we use binary search to calculate how much its WCET c_{τ_i} can be increased, as shown from line 2 to line 11. During the binary search, the lower bound lb is initially set to 1, representing the normalized factor with respect to the original WCET; while the upper bound ub is initially set to T_{τ_i}/c_{τ_i}, representing the normalized factor with

Algorithm 1: System Extensibility Computation

```
1:  $\mathcal{E} = 0$;
2:  for all task $\tau_i \in \mathcal{T}$ do
3:      $lb = 1; ub = T_{\tau_i}/c_{\tau_i}; c_{original} = c_{\tau_i}$
4:      while $ub - lb > \epsilon$ do
5:          $mid = (lb + ub)/2$;
6:          $c_{\tau_i} = mid * c_{original}$;
7:          $isSched$ = checkTaskSched();
8:          if $isSched == true$ then
9:              $lb = mid$;
10:         else
11:             $ub = mid$;
12:     $\mathcal{E}$ += $w_{\tau_i} * (mid - 1) * c_{original}/T_{\tau i}; c_{\tau_i} = c_{original}$
13: return $\mathcal{E}/|\mathcal{T}|$;
```

respect to the task deadline/period. The iterations end when the upper bound and lower bound meet within ϵ. Inside each iteration, we calculate the middle value *mid* (line 5) and update WCET c_{τ_i} (line 6). Then, function $checkTaskSched()$ updates all the response times of lower priority tasks, and checks whether any schedulability constraint has been violated (line 7). If the system is schedulable, we continue search the upper half (i.e., trying larger value for the execution time), otherwise we search the lower half. The system extensibility is the weighted sum of all tasks.

3.3 Soft Error Tolerance Model

We consider two major soft error detection techniques, i.e. embedded error detection (EED) and explicit output comparison (EOC). Usually, EED covers part of the total errors with additional computation overhead (which depends on specific application and implementation method). For instance, state-of-the-art CFC techniques may cover 70% of total errors. EOC can achieve almost 100% error detection at the cost of 100% execution time overhead (temporal redundancy) or 100% resource overhead (spatial redundancy). In this work, we assume EOC detection rate is 100%, similarly as in [26].

System error coverage: During the hyperperiod T_{hyper} of a task set $\mathcal{T}$ (i.e., the least common multiple of the task periods), a total number of $K \geq 0$ errors may occur. System error coverage is then defined as the probability that all errors are either i) detected and recovered within hyperperiod while all timing constraints are satisfied or ii) happened during idle time [26].

Let t_{eoc}, t_{eed}, t_{none} denote the accumulative time needed by tasks employing EOC, EED and no error detection technique, respectively. t_{idle} denotes the total idle time. An exact analysis of system error coverage depends on the specific error occurrence profile and timing pattern, and is hard to capture with a closed form formulation. For simplicity, on a single ECU, we assume that K arbitrary errors of uniform distribution may occur during a hyperperiod. The system error coverage P is then approximated as:

$$P \approx \sum_{i=0}^{K} \sum_{j=0}^{i} \binom{K}{i} \binom{i}{j} (\frac{\alpha \cdot t_{eed}}{T_{hyper}})^j (\frac{\beta \cdot t_{eoc}}{T_{hyper}})^{i-j} (\frac{t_{idle}}{T_{hyper}})^{K-i} \tag{8}$$

where α and β represent the error detection rate of EED and EOC, respectively.

Task execution time with error detection and recovery: As we mentioned, EED and EOC come with additional computation overhead. We characterize a task using error detection technique with $C_{dec_{\tau_i}}$, which denotes the time for execution and error detection of task τ_i. For EOC, $C_{dec_{\tau_i}} = 2C_{\tau_i} + \Lambda_i$ if a temporal redundancy approach is used. Λ_i denotes the time for comparing outputs. If we duplicate the execution of same task on different cores (i.e. a spatial redundancy approach), $C_{dec_{\tau_i}} = C_{\tau_i} + \Lambda_i$. For EED, since we run a task with built-in detection, $C_{dec_{\tau_i}} = C_{\tau_i} + \Delta C_{\tau_i}$, where ΔC_{τ_i} is the increased timing cost for EED. $C_{rec_{\tau_i}}$ is the error recovery time for task τ_i if the error is detected. We assume the re-execution of a task is scheduled immediately if error(s) is detected. $C_{rec_{\tau_i}} = C_{\tau_i}$ for EOC, while $C_{rec_{\tau_i}} = C_{\tau_i} + \Delta C_{\tau_i}$ for EED.

Worst-case response time analysis with error detection: To analyze the response time of task with EED/EOC technique, we need to integrate error detection time and recovery time into Equation (1). For this, we employ two binary ρ_i and o_i to distinguish error detection strategy. ρ_i is 1 if either EED or EOC is employed for task τ_i and 0 otherwise. o_i is 1 if EOC if used, and 0 if EED is used. We rewrite $C_{dec_{\tau_i}}$ and $C_{rec_{\tau_i}}$ as following:

$$C_{dec_{\tau_i}} = C_{\tau_i} + [o_{\tau_i}(C_{\tau_i} + \Lambda_{\tau_i}) + (1 - o_{\tau_i})\Delta C_{\tau_i}]\rho_{\tau_i} \tag{9a}$$

$$C_{rec_{\tau_i}} = (C_{\tau_i} + (1 - o_{\tau_i})\Delta C_{\tau_i})\rho_{\tau_i} \tag{9b}$$

Let r_{τ_i,τ_j} denote worst-case response time for task τ_i when error(s) occurs during the execution of task τ_j. Task τ_i will be blocked by task τ_j's recovery time if τ_j has higher priority. Follow the same idea of Equation (1), we have:

$$r_{\tau_i,\tau_j} = C_{dec_{\tau_i}} + C_{rec_{\tau_j}} p_{\tau_i,\tau_j} + \sum_{\tau_k \in \mathcal{T} \wedge \tau_i \neq \tau_j} \left\lceil \frac{r_{\tau_i,\tau_j}}{T_{\tau_k}} \right\rceil C_{dec_{\tau_k}} p_{\tau_i,\tau_j} \tag{10}$$

where p_{τ_i,τ_j} denotes the relative priority between task τ_i and τ_j. Considering a complete task with K errors, the response time of task τ_i can be formulated as:

$$\begin{aligned} r_{\tau_i} = & \sum_{e_l \in E} a_{\tau_i,e_l} C_{dec_{\tau_i}} + K \max_{\tau_j \in \mathcal{T}} \{ \sum_{e_l \in E} C_{rec_{\tau_j}} p_{\tau_i,\tau_j} h_{\tau_i,\tau_j,e_l} \} \\ & + \sum_{\substack{\tau_k \in \mathcal{T} \\ \wedge \tau_i \neq \tau_j}} \sum_{e_l \in E} \left\lceil \frac{r_{\tau_i,\tau_j}}{T_{\tau_k}} \right\rceil C_{dec_{\tau_k}} p_{\tau_i,\tau_j} h_{\tau_i,\tau_j,e_l} \end{aligned} \tag{11}$$

where Boolean variable a_{τ_i,e_l} is 1 if task τ_i is assigned to core e_l and 0 otherwise. h_{τ_i,τ_j,e_l} is 1 if task τ_i and τ_j are on the same core e_l and 0 otherwise. To ensure each tasks is only mapped to one ECU, the following relations must be enforced:

$$\sum_{e_l \in E} a_{\tau_i,e_l} = 1 \tag{12}$$

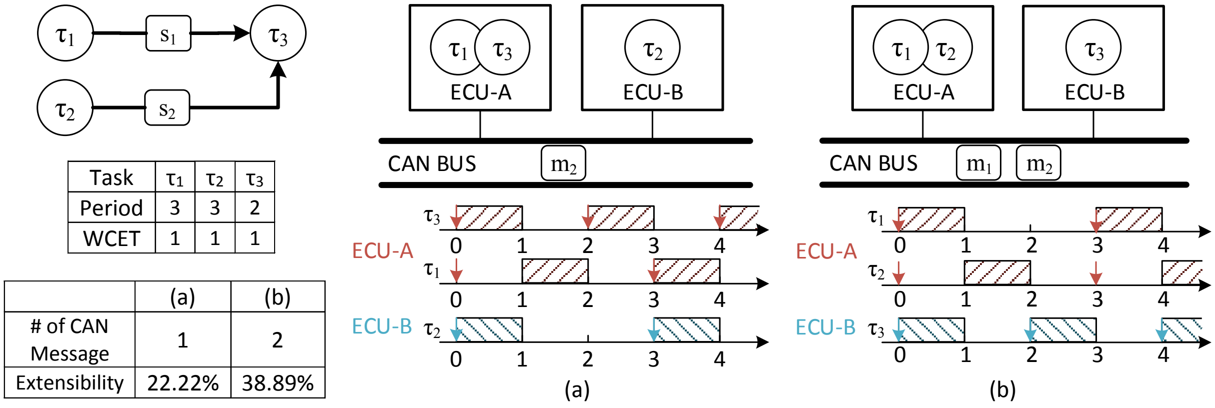
$$a_{\tau_i,e_l} + a_{\tau_j,e_l} - 1 \leq h_{\tau_i,\tau_j,e_l} \tag{13}$$

$$h_{\tau_i,\tau_j,e_l} \leq a_{\tau_i,e_l} \tag{14}$$

$$h_{\tau_i,\tau_j,e_l} \leq a_{\tau_j,e_l} \tag{15}$$

4 OPTIMIZATION AND TRADE-OFFS AMONG OBJECTIVES

Based on the models introduced in Section 3, we quantitatively analyze the trade-offs among extensibility, fault tolerance and latency (as well as other metrics related to communication cost, such as the number of CAN messages and the bus utilization). In this section,



Task	τ1	τ2	τ3
Period	3	3	2
WCET	1	1	1

	(a)	(b)
# of CAN Message	1	2
Extensibility	22.22%	38.89%

Figure 2: An illustrating example showing the trade-off between extensibility and communication cost (i.e., number of messages and path latencies), under different task mapping choices: (a) task τ_1 and task τ_3 mapped to ECU-A while task τ_2 mapped to ECU-B, and (b) task τ_1 and task τ_2 mapped to ECU-A while task τ_3 mapped to ECU-B.

we will first demonstrate such trade-offs with an illustrating example, and then introduce a simulated annealing based algorithm for optimizing these different design objectives/metrics.

4.1 Illustrating Example

Figure 2 shows how the mapping of three tasks onto a CAN-based platform with two ECUs can affect system extensibility and communication cost (measured by the number of CAN messages or path latencies). The task graph, task WCETs and periods are shown on the left side. Task τ_1 and τ_2 send their output to task τ_3.

In mapping (a), task τ_1 and τ_3 are mapped to the same ECU-A, while τ_2 is mapped to ECU-B. We assign higher priority to τ_3 than τ_1, based on the Rate Monotonic policy. Although there is still 16.7% utilization left on ECU-A, this mapping makes it impossible to increase the execution time for either τ_1 or τ_3, i.e., their extensibility is zero. On ECU-B, task τ_2 can increase its WCET by 2 time unit. Thus, the total system extensibility is $2/3/3 = 22.2\%$. In terms of communication, only message m_2 needs to be transmitted over CAN bus and the latency on path τ_1 to τ_3 should be relatively short.

In mapping (b), task τ_2 and τ_3 are swapped. On ECU-A, task τ_1 and τ_2 have the same period and WCET, and we assume τ_1 has the higher priority. The maximum increase of WCET for either task is 1 time unit. The system extensibility is calculate as $(1/3 + 1/3 + 1/2)/3 = 38.9\%$. In terms of communication, messages m_1 and m_2 need to be transmitted over the CAN bus, and the latency on path τ_1 to τ_3 is also higher than mapping (a).

In this example, we clearly see the trade-off between extensibility and communication cost (i.e., latency or number of messages). Next, we will introduce how we can optimize these design objectives.

4.2 Objective Function and Constraints

We optimize an objective function (16) that includes extensibility, fault tolerance and communication cost, by exploring allocation, priority assignment and error detection technique (EED or EOC or none detection) for each task. The communication cost could be measured by total path latency, number of CAN messages or bus utilization.

$Cost_{ext}$, $Cost_{ft}$, $Cost_{com}$ in (16) are costs for extensibility, fault tolerance and communication, respectively. For instance, $Cost_{ext} = 1 - \mathcal{E}$, where $\mathcal{E}$ is the system extensibility in (7). Note that the higher the extensibility, the lower the cost $Cost_{ext}$ is. λ, μ and γ are weights and can be tuned to trade off these objectives.

The optimization is subject to the schedulability constraints in (4) to (6), and possible constraint on each design objective. For instance, there could be upper bounds EXT_{max}, FT_{max} and COM_{max} on each cost, as shown below.

$$\min \quad \lambda * Cost_{ext} + \mu * Cost_{ft} + \gamma * Cost_{com} \tag{16}$$

$$s.t. \quad 0 \geq \lambda \geq 1, 0 \geq \mu \geq 1, 0 \geq \gamma \geq 1 \tag{17}$$

$$Cost_{ext} \leq EXT_{max} \tag{18}$$

$$Cost_{ft} \leq FT_{max} \tag{19}$$

$$Cost_{com} \leq COM_{max} \tag{20}$$

4.3 Simulated Annealing

We developed a simulated annealing based algorithm for the above optimization, as shown in Algorithm 2. For the initial configuration, tasks are randomly allocated to ECUs and scheduled using the Rate Monotonic policy. T represents current simulation temperature, T^* is the final temperature and η is the cooling factor. K^* is the maximum number of iterations within each temperature.

During each iteration, function *randomChange* modifies current solution A_{cur} into a candidate solution A_{new} by randomly performing one of the following operations: i) changing the allocation of a task from one ECU to another, ii) swapping the priorities of two tasks on the same ECU, or iii) changing the error detection technique of a task. Function $ComputeObjective(A_{new})$ then computes corresponding cost C_{new} as defined in (16), and function $checkSched(A_{new})$ determines the schedulability of this candidate solution. Note that if the candidate solution is infeasible, we add a penalty Δ to the cost instead of rejecting the solution directly. Function $P(C_{cur}, C_{new}, T)$ then computes the acceptance possibility of A_{new}. A_{cur} and C_{cur} keep track of the latest solution and its cost. A_{opt} stores the best solution. The simulated annealing procedure stops when temperature reaches the predefined value T^*.

Algorithm 2: Simulated Annealing for Task Allocation, Scheduling and Error Detection Technique Selection

```
Construct initial configuration.
while $T \geq T^*$ do
    while $K < K^*$ do
        $A_{new}$ = randomChange($A_{cur}$)
        $C_{new}$ = ComputeObjective($A_{new}$)
        $isSched$ = checkSched($A_{new}$)
        if $isSched == false$ then
            $C_{new} = C_{new} + \Delta$
        if $C_{new} < C_{cur}$ then
            $A_{cur} = A_{new}, C_{cur} = C_{new}$
            if $isSched = true$ then
                $A_{opt} = A_{cur}$
        else if P($C_{cur}$, $C_{new}$, $T$) > rand() then
            $A_{cur} = A_{new}, C_{cur} = C_{new}$
        $K = K + 1$
    $T = T * \eta$
return $A_{opt}$
```

5 EXPERIMENTAL RESULTS

We conducted experiments on an industrial case and a set of synthetic examples. The industrial case is derived from an experimental vehicle subsystem and contains 41 tasks communicating through 81 signals. The subsystem involves distributed functions collecting data from 360° sensors to actuators. All the periods and task WCETs are given in the industrial case. We also use the TGFF tool [6] to generate a set of synthetic examples with random periods and WCETs. We impose end-to-end latency deadlines on selected critical paths. The examples are tested for a number of different platform configurations. For the industrial case, a minimum of 5 ECUs is needed for finding feasible solutions.

5.1 Extensibility vs. Latency

We first explore the trade-off between extensibility and communication cost, which is measured by the total critical path latency (i.e., the sum of end-to-end latencies for all selected critical paths). We conduct optimizations using our simulated annealing approach (Algorithm 2). More specifically, for extensibility optimization, we set $\lambda = 1$ and $\mu = \gamma = 0$ in the objective function (16). For latency optimization, we set $\lambda = \mu = 0$ and $\gamma = 1$. We carry out these optimizations for platform configurations containing 5 to 10 ECUs, and record both extensibility and latency for each optimization. The comparison results for the industrial case are shown in Figure 3, with yellow bars on the right in both sub-figures representing extensibility optimization results and blue bars on the left in both sub-figures representing latency optimization results.

We can clearly see a trade-off between extensibility and latency. Extensibility optimization indeed significantly improves the extensibility metric over latency optimization, but leads to longer total latency. Intuitively, optimizing extensibility leads to more "balanced" allocation of tasks on ECUs and thus more CAN messages (rather than communication through local memory) and longer total latency. In our experiments, extensibility optimization results have over 70 signals mapped to CAN messages while latency optimization results only have 10-20 messages. Furthermore, we can see that the extensibility increases with more ECUs. This is as expected since the tasks get more timing slacks with lower average ECU utilization.

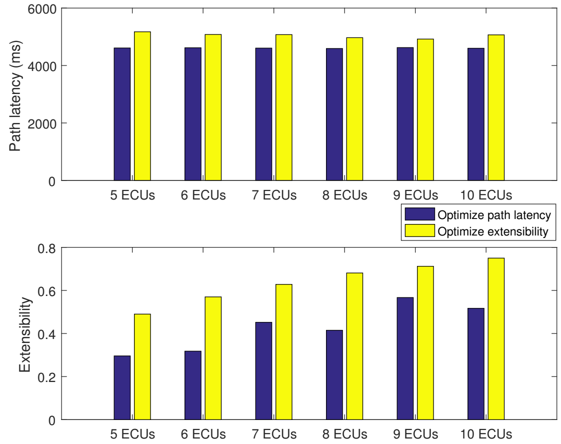


Figure 3: Comparison between extensibility optimization and latency optimization for industrial case.

5.2 Error Coverage vs. Latency

We then explore the trade-off between error coverage (fault tolerance) and latency. We employ EED and the temporal redundancy model of EOC as described in Section 3.3, and we set $\lambda = \gamma = 0$ and $\mu = 1$ for error coverage optimization. Figure 4 shows the comparison between error coverage optimization (yellow bars on the right) and latency optimization (blue bars on the left) for the industrial case. The trade-off between the two is also very clear. Intuitively, more balanced allocation of tasks leads to more timing slacks for tasks to add error detection techniques, but results in longer latency. We can also see the error coverage increases with more ECUs.

5.3 Extensibility vs. Error Coverage

We study the relation between extensibility and error coverage, by mapping the industrial case onto a platform of 5 ECUs with an average ECU utilization of 58%. Note that extensibility and error coverage are not always mutually exclusive. Both metrics get better when more time slack is available. However, applying the slack to error detection techniques does take away some capability to accommodate future changes. Table 1 shows the trade-off between extensibility and error coverage, when error coverage optimization is performed with minimum extensibility set to 0, 0.1, 0.2, 0.3, 0.4 and 0.5, respectively.

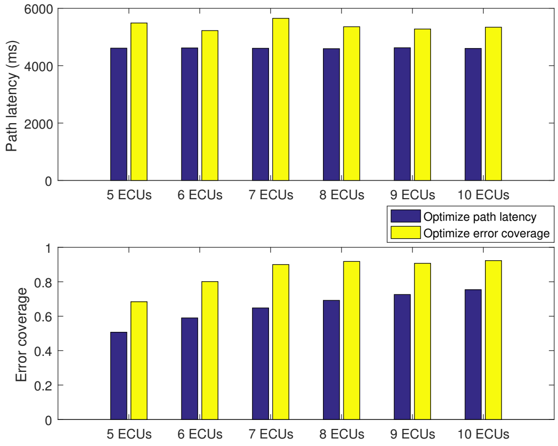


Figure 4: Comparison between error coverage (fault tolerance) optimization and latency optimization for industrial case.

Table 1: Optimizing error coverage at different minimum extensibility requirement for industrial case.

min. ext.	0	0.1	0.2	0.3	0.4	0.5
error coverage	0.5775	0.5176	0.5122	0.4568	0.4472	0.4089
extensibility	0.1354	0.1582	0.2162	0.3027	0.4036	0.5605

5.4 Optimization of All Three Objectives

We then optimize all three objectives (extensibility, error coverage / fault tolerance, latency / communication cost) by setting $\lambda = \mu = \gamma = 1$ in (16). $Cost_{ext}$, $Cost_{ft}$ and $Cost_{com}$ are all normalized. In particular, $Cost_{ext} = 1 - \mathcal{E}$. $Cost_{ft}$ is as defined in Equation (8). $Cost_{com}$ represents the cost of total latency, and is defined as $Cost_{com} = (Lat - Lat_{lb})/(Lat_{ub} - Lat_{lb})$. $Lat = \sum_{p_k \in \mathcal{P}} l_{p_k}$ is the total latency, $Lat_{lb} = \sum_{p_k \in \mathcal{P}} \sum_{\tau_i \in p_k} (T_{\tau_i} + c_{\tau_i})$ is a lower bound for the total latency, and $Lat_{max} = 2 * \sum_{p_k \in \mathcal{P}} (\sum_{\tau_i \in p_k} T_{\tau_i} + \sum_{s_j \in p_k} T_{s_j})$ is an upper bound.

Table 2: System total critical path latency, extensibility and error coverage in the solutions from optimizing each individual objective and from optimizing all three objectives (industrial case).

5 ECUs			
	Path Latency	Extensibility	Error Coverage
Opt. Path Latency	4718.02	0.296	0.629
Opt. Extensibility	5174.52	0.490	0.507
Opt. Error Coverage	5488.06	0.323	0.684
Opt. All	5206.46	0.418	0.699

8 ECUs			
	Path Latency	Extensibility	Error Coverage
Opt. Path Latency	4761.50	0.415	0.856
Opt. Extensibility	4969.90	0.681	0.692
Opt. Error Coverage	5357.31	0.485	0.918
Opt. All	5221.79	0.633	0.806

As shown in Table 2, optimizing all three objectives provides more balanced solutions, when compared with optimizing for each individual objective. Such results are not surprising qualitatively, but the quantitative comparison should facilitate designers to make design choices.

5.5 Impact of ECU Speed and Number of Tasks

Finally, we study how extensibility is affected by the ECU computation speed and the number of tasks. We generate a set of synthetic examples with different number of tasks and map them to a platform with 5 ECUs. We scale all task WCETs by a factor of 1X, 1.5X and 2X to model different ECU computation speed while task periods remain unchanged. Figure 5 demonstrates the quantitative impact of ECU speed and number of tasks on system extensibility.

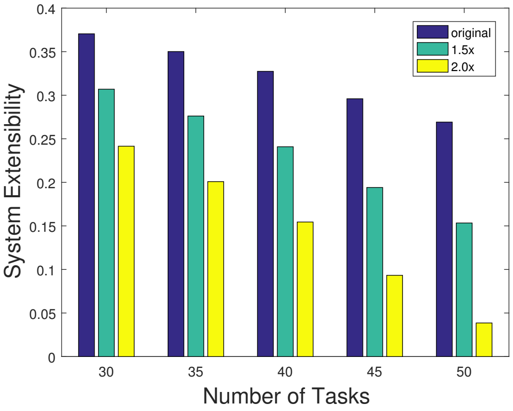


Figure 5: System extensibility under different ECU speed and number of tasks (synthetic examples).

6 CONCLUSION

In this work, we quantitatively analyze the trade-offs among extensibility, fault tolerance and latency for CAN-based automotive electronic systems. We introduce metrics for defining these three objectives and present a simulated annealing based algorithm for optimizing them. The clear trade-offs among these objectives demonstrate the need to develop design automation methods for facilitating the design space exploration in automotive systems.

7 ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation grants CCF-1553757, CCF-1646381 and CNS-1646641, and the Office of Naval Research grant N00014-14-1-0815.

REFERENCES

[1] Iain Bate and Paul Emberson. 2006. Incorporating scenarios and heuristics to improve flexibility in real-time embedded systems. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 221–230.

[2] Robert C Baumann. 2005. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability* 5, 3 (2005), 305–316.

[3] FlexRay Consortium et al. 2005. FlexRay communications system-protocol specification. *Version* 2, 1 (2005), 198–207.

[4] G de A Lima and Alan Burns. 2001. An effective schedulability analysis for fault-tolerant hard real-time systems. In *Proceedings of 13th Euromicro Conference on Real-Time Systems*. 209–216.

[5] G de A Lima and Alan Burns. 2003. An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems. *IEEE Trans. Comput.* 52, 10 (2003), 1332–1346.

[6] Robert P Dick, David L Rhodes, and Wayne Wolf. 1998. TGFF: task graphs for free. In *Proceedings of the 6th International Workshop on Hardware/Software Codesign*. IEEE Computer Society, 97–101.

[7] L-B Fredriksson. 2002. CAN for critical embedded automotive networks. *IEEE Micro* 22, 4 (2002), 28–35.

[8] Yue Gao, Sandeep K Gupta, and Melvin A Breuer. 2013. Using explicit output comparisons for fault tolerant scheduling (FTS) on modern high-performance processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 927–932.

[9] Michael Gonzalez Harbour, Mark H. Klein, and John P. Lehoczky. 1994. Timing analysis for fixed-priority scheduling of hard real-time systems. *IEEE Transactions on Software Engineering* 20, 1 (1994), 13–28.

[10] Sönke Holthusen, Sophie Quinton, Ina Schaefer, Johannes Schlatow, and Martin Wegner. 2016. Using Multi-Viewpoint Contracts for Negotiation of Embedded Software Updates. *arXiv preprint arXiv:1606.00504* (2016).

[11] Viacheslav Izosimov, Ilia Polian, Paul Pop, Petru Eles, and Zebo Peng. 2009. Analysis and optimization of fault-tolerant embedded systems with hardened processors. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 682–687.

[12] Viacheslav Izosimov, Paul Pop, Petru Eles, and Zebo Peng. 2005. Design optimization of time-and cost-constrained fault-tolerant distributed embedded systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 864–869.

[13] Jeff C Jensen, Danica H Chang, and Edward A Lee. 2011. A model-based design methodology for cyber-physical systems. In *2011 7th International Wireless Communications and Mobile Computing Conference (IWCMC)*. IEEE, 1666–1671.

[14] Gabor Karsai, Janos Sztipanovits, Akos Ledeczi, and Ted Bapty. 2003. Model-integrated development of embedded software. *Proc. IEEE* 91, 1 (2003), 145–164.

[15] Gabriel Leen and Donal Heffernan. 2002. Expanding automotive electronic systems. *Computer* 35, 1 (2002), 88–93.

[16] Rainer Makowitz and Christopher Temple. 2006. FlexRay-a communication network for automotive control systems. In *2006 IEEE International Workshop on Factory Communication Systems*. 207–212.

[17] Seyed Ghassem Miremadi, Johan Karlsson, Ulf Gunneflo, and Jan Torin. 1992. Two Software Techniques for On-line Error Detection. In *22nd International Symposium on Fault-Tolerant Computing (FTCS)*. 328–335.

[18] Nahmsuk Oh, Philip P Shirvani, and Edward J McCluskey. 2002. Control-flow checking by software signatures. *IEEE Transactions on Reliability* 51, 1 (2002), 111–122.

[19] András Pataricza, István Majzik, Wolfgang Hohl, and Joachim Hönig. 1993. Watchdog processors in parallel systems. *Microprocessing and Microprogramming* 39, 2-5 (1993), 69–74.

[20] Paul Pop, Viacheslav Izosimov, Petru Eles, and Zebo Peng. 2009. Design optimization of time-and cost-constrained fault-tolerant embedded systems with checkpointing and replication. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 17, 3 (2009), 389–402.

[21] Reinhard Schneider, Dip Goswami, Samarjit Chakraborty, Unmesh Bordoloi, Petru Eles, and Zebo Peng. 2011. On the quantification of sustainability and extensibility of FlexRay schedules. In *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. IEEE, 375–380.

[22] CAN Specification. 1991. 2.0 B. *Robert Bosch GmbH, Stuttgart, Germany* (1991), 401–413.

[23] Tesla. 2017. Tesla Over-the-Air Software Update. http://www.teslamotors.com/support/software-updates. (2017).

[24] Christopher Weaver, Joel Emer, Shubhendu S Mukherjee, and Steven K Reinhardt. 2004. Techniques to reduce the soft error rate of a high-performance microprocessor. In *ACM SIGARCH Computer Architecture News*, Vol. 32. IEEE Computer Society, 264.

[25] Ramesh Yerraballi and Ravi Mukkamalla. 1996. Scalability in real-time systems with end-to-end requirements. *Journal of Systems Architecture* 42, 6-7 (1996), 409–429.

[26] Bowen Zheng, Yue Gao, Qi Zhu, and Sandeep Gupta. 2015. Analysis and optimization of soft error tolerance strategies for real-time systems. In *Proceedings of the 10th International Conference on Hardware/Software Codesign and System Synthesis*. IEEE Press, 55–64.

[27] Qi Zhu, Yang Yang, Marco Di Natale, Eelco Scholte, and Alberto Sangiovanni-Vincentelli. 2010. Optimizing the software architecture for extensibility in hard real-time distributed systems. *IEEE Transactions on Industrial Informatics* 6, 4 (2010), 621–636.

[28] Qi Zhu, Haibo Zeng, Wei Zheng, Marco Di Natale, and Alberto Sangiovanni-Vincentelli. 2012. Optimization of task allocation and priority assignment in hard real-time distributed systems. *ACM Transactions on Embedded Computing Systems (TECS)* 11, 4 (2012), 85.