Power-aware Allocation of Fault-tolerant AUTOSAR Systems with End-to-end Timing Constraints via Hybrid Particle Swarm Optimization

Abstract

The growing complexity of automotive functionality has attracted revolutionary computing architectures such as mixed-criticality design, which enables effective consolidation of software applications with different criticality on shared execution platforms. The allocation of mixed-critical software application on the execution platform plays an important role in the design of predictable and resource-constrained real-time software systems that are required to meet end-to-end timing and reliability requirements, and preserve critical system resources such as power and energy of embedded systems. Due to the non-linearity of real-time scheduling, complexity of reliability analysis, and large-scale software systems, exact methods, e.g., branch and bound, are prohobitively expensive.

In this paper, we propose a hybrid particle-swarm optimization technique for integrated software allocation on heteroginuous computing units to optimize total power consumption while meeting end-to-end timing and reliability requirements. Our optimization approach takes into consideration fault-tolerance to maximize reliability of software applications to meet reliaiblity goals. To reduce the overhead of fault-tolerance especially on end-to-end timing analysis, we propose an approximation technique. The proposed approach is evaluated using a range of different software applications that are synthesize from real-world automotive benchmark. The evaluation makes comparative analysis of integer-linear programming method and various hybrid algorithms on metrices such as optimality, efficiency and scalability.

1. Introduction

Over the last decades, the complexity of automotive functionality has increased tremendously, that is, the electical/electronic platform is running more and more automotive functions (or software applications), with millions lines of code, over complex network infrastructure. Moreover, some automotive applications are computationally intensive, e.g., the computer-vision detection in self-driving vehicles using deep learning. Thus, there is a need for powerful computing architectures that can accommodate the current and future demands of software applications. The distributed computing in automotive systems enables deployment of automotive applications on multiple units to realize complex functionality, e.g., end-to-end behavior, which have strict timing constraints. Moreover, consolidation of applications on the same execution platform for efficiency has gained interest in the automotive domain, also known as *mixed-criticality* design, whicc is after demonstrated the avionics domain. The distributed computing of applications and mixed-criticality design are interesting phenomena in the evolution of automotive systems development. In the context of software allocation, they bring two main challengs: i) real-time analysis is complex in distributed environment due to independently executing functions on the different units. Considering independent failures of the computing units, the distriburted architecture provides an opportunity to improve the reliability of the software applications by replicating functionality on multiple units; ii) the allocation of distributed software applciations is normally NP hard, and therefore finding optimal solutions is exponential.

Software allocation is a well-researched area in the domain of embedded systems, including in hardware/software co-design [1], platform-based system design [2] and the Y-chart design approach [3]. It is a type of job-shop problem with constraints, and therefore finding an optimal solution, in the general case, is NP-hard [4]. The methods to solve such problems can be *exact* or *heuristic.* The exact methods, e.g., branch and bound, dynamic programming, etc., gurantee optimal solutions, neverthless, they do not scale to large-scale problems [5]. Moreover, applying exact methods on-non linear problems, which are prevalent in practice, is prohobitively expensive. Our previous work on solving the software allocation problem [43], we demonstrate the limitation of integer-linear programming (ILP) [8] using exact method by the CPLEX solver . Similary, the scalability issues of exact methods on software allocation is indicated in several research [5]. In contrast, heuristic methods device a working technique to solve practical problems, which are usually large-scale, non-linear, without gurantee of optimality [6, 7]. A particular type of heuristic is *metaheuristics* which can be defined as "an iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space, learning strategies are used to structure information in order to find efficiently near-optimal solutions." [**?**].

Metaheuristics has found wide applications in many domains, e.g., cellular networks, cloud computing, softwar design, etc [**?**]. Many of the existing metaheuristic algorithms are nature inspired, e.g., genetic algorithm, evolutionary algorithms, simulated annehealing, ant colony, paticle-swarm optimization, etc. Applications of metaheuristics on the software allocation of real-time systems are in the early stages, neverthless, there exist some work, e.g., by Qin-Ma et al. [**?**] on maximizing reliability of distributed computing sytems using hanybee algorithm, maximizing reliability of distributed systems using hill-climbing particle-swarm optimization by Yin et al. [35], etc. In this work, we apply differential evolution and hybrid particle-swarm optimization algorithms on a fault-tolerant distributed software applications to optimize the total power consumption of a distributed system. The software applications are developed using the AUTOSAR software components that are implemented by periodically activated runnables. Sequencies of runnables deployed on the same unit or netowrk of units realize end-to-end functionality, also known as *cause-effect* chains. The chains are triggered by different sampling rates, also known as *multirate* [12]. The propagation of signals over multirate chains result in undersampling/oversampling effects, which makes end-to-end timing analysis difficult [13]. In order to maximize software applications reliability and meet their reliability goals, we implement fault-tolerance.

The contributions of our work are summerized as follows: (i) we provide a fitness function with end-to-end timing and reliability constraints of the software applications, (ii) due to the overhead of fault-tolerance, we propose an approximation algorithm to reduce the end-to-end delays of computation, (iii) we provide performance comparison of the ILP method with CPLEX, differential evolution, and hybrid particle-swarm optimization algorithms with differential evolution, hill-climbing and stochastic hill-climbing algorithms. Our approach is evaluated on synthetic automotive applications that are generated according to the real-world automotive benchmark proposed by Kramer et al. [14]. In the evaluation, we show comparative performance of the various optimization algorithms interms of quality of solutions (or optimality), computation time, and stability of the algorithms, for small and large software allocation problems. The tool applied in the evaluation is publicly accessible from BitBucket[1].

The rest of the paper is organized as follows: Section 2 provides a brief overview of AUTOSAR, emphasizing on end-to-end timing and reliability modeling, and software allocation, Section 3 describes the AUTOSAR system model, including timing analysis, reliability and power-consumption assumptions. Formulation of the software allocation problems is presented in Section 4, which consist of the timing and reliability constraints and minimization of the total power consumption, followed by formulation of the optimization problem. We show how to solve the optimization problem using metaheuristics in Section 5. The evaluation of our proposed methods are demonstrated in Section 6 using the automotive benchmark. Our work is compared to related works in Section 8. Finally, we conclude the paper in Section 9, and outline the possible future work.

2. AUTOSAR

The AUTomotive Open System ARchitecture (AUTOSAR) partnership has defined the open standard AUTOSAR for automotive software architecture that enables manufacturers, suppliers, and tool developers to adopt shared development specifications, while allowing sufficient space for competitiveness. The specifications state standards and development methodologies on how to manage the growing complexity of Electronic/Electrical (E/E) systems, which take into account the flexibility of software development, portability of software applications, dependability, efficiency, etc., of automotive solutions.

[1]`https://bitbucket.org/nasmdh/archsynapp/src/master/`

The conceptual separation of software applications from their infrastructure (or execution platform) is an important attribute of AUTOSAR and is realized through different functional abstractions [16].

2.1. Software Application

According to AUTOSAR, software applications are realized on different functional abstractions. The top-most functional abstraction, that is the Virtual Function Bus (VFB), defines a software application over a virtual communication bus using software components that communicate with each other via standard interfaces of various communication semantics. The behavior of a software component is realized by one or more atomic programs known as *Runnables*, which are entities that are scheduled for execution by the operating system and provide abstraction to operating system tasks, essentially enabling behavioral analysis of a software application at the VFB level. The Runtime Time Environment (RTE), which is the lower-level abstraction, realizes the communication between Runnables via RTE Application Programming Interface (API) calls that respond to events, e.g., timing. Furthermore, the RTE implementation provides software components with the access to basic software services, e.g., communication, micro-controller and ECU abstractions, etc., which are defined in the Basic Software (BSW) abstraction [16].

2.2. Timing and Reliability of Applications

The timing information of applications is a crucial input to the software allocation process. Among other extensions, the AUTOSAR Timing Extension specification [17] states the timing descriptions and constraints that can be imposed at the system-level via the *SystemTiming* element. The timing constraints realize the timing requirements on the observable occurrence of events of type *Timing Events*, e.g., Runnables execution time, and *Event Chains*, also referred to as *Cause-effect Chains* that denote the causal nature of the chain. In this work, we consider periodic events and cause-effect chains with different rates of execution (or activation patterns).

Although the importance of reliability is indicated in various AUTOSAR specifications via best practices, the lack of a comprehensive reliability design recommendations has opened an opportunity for flexible yet not standardized development approaches. In this paper, we consider application reliability as a user requirement and, in the allocation process, we aim at meeting the requirement via optimal placement and replication of software components.

3. System Model

The system model shown in Figure 1 is captured as 5-tuple $\langle A, E, \mapsto, Req, Pro \rangle$, where $A = \{a_k : k = 1, \ldots, n_A\}$ denotes AUTOSAR software applications, $E = \langle N, B \rangle$ an execution platform of $N = \{n_i : i = 1, \ldots, n_N\}$ heterogenous computation (or computing units) and a shared CAN bus B, $\mapsto: A \to N$ partitions the software applications to at least a single computing unit, and Req, Pro are assignment functions, respectively define each application the software application requirements and the resource provisions of the units. The function $Req : A \rightarrow (\mathrm{RL}, \mathrm{EE}, \mathrm{CL})$ assigns each application the reliability goal (or requirement), end-to-end timing requirements and criticality level. The function $Pro : N \rightarrow (\mathrm{HZ}, \mathrm{PW}, \mathrm{FR})$ assigns each computing unit, respectively the processor speed, power consumption and failure rate.

The end-to-end timing requirements are the timing constraints over the end-to-end functional behaviors of the applications. These requirements are often specified on the *cause-effect chains* consisting of software components and (potentially network messages) within the application. The reliability requirement is the expected reliability goal over a period of time t in which no failure of the application experienced, and the criticality level signifies the importance of an application over other applications that have lower criticality levels, thus prioritizing the application during resource contention. The criticality levels are defined systematically, e.g., following the hazard analysis according to the ISO 26262 standard for functional safety in road vehicles [18].

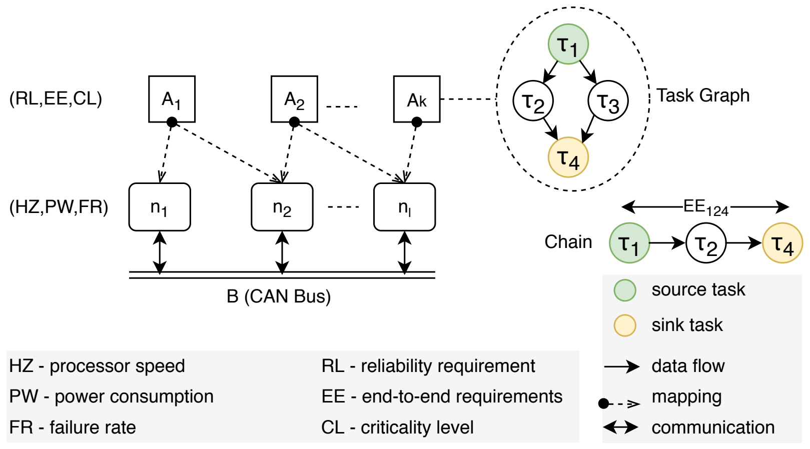


Figure 1: System Model.

3.1. Software Application

The software application represents an independent and self-contained user-defined software functionality, e.g., x-by-wire, electronic throttle control, flight control, etc. The application is assumed to be developed using AUTOSAR software components, which define functional behavior, resource requirements, etc. The functional behavior of the components are implemented using AUTOSAR Runnables, which are schedulable pieces of codes similar to tasks. We assume the runnables are periodically activated with support for multiple worst-case executions that correspond to the different computation processor types. Unlike tasks, runnables' functional and extra-functional properties, e.g., timing, memory requirements, are part of the AUTOSAR software component specification.

Notation	Description
/* Application related */	
• $A = \{a_i : i = 1, \dots, n_A\}$ denote software applications, modeled as directed acyclic graphs of runnables, where $\mathcal{V}(a_k)$ are the nodes and $\mathcal{E}(a_k)$ the directed edges of the graph a_k.	
• $C^{a_k} = \{c_i^{a_k} : i = 1, \dots, n_C\}$	software-component types used in a_k **
• $Q_i^{a_k} = \{q_{i,j}^{a_k} : j = 1, \dots, n_{Q_i}\}$	component replicas of type $c_i^{(a_k)}$
• $R_i^{a_k} = \{r_{i,j}^{a_k} : j = 1, \dots, n_{R_i}\}$	runnables of $c_i^{(a_k)}$
• $T_i^{a_k} = \{r_{i,j}^{a_k} : j = 1, \dots, n_{T_i}\}$	tasks mapped to $c_i^{(a_k)}$
/* Execution platform related */	
• $N = \{n_i : i = 1, \dots, n_N\}$	computation (or computing) nodes
• $M = \{m_i : i = 1, \dots, n_M\}$	messages on the CAN bus
• τ, c, m, γ denote iterator variables, respectively for task, component, chain and node, e.g., $\forall \tau \in T^{a_k}$.***	
/* Mapping related */	
• $\mathbf{x} = \{\mathbf{x}_k : k = 1, \dots, n_{\mathbf{x}}\} : \bigcup Q_i^{a_k} \mapsto M$	a mapping vector from $Q^{(a_k)}$ to M
• k, i, j denote iterator index-variables, respectively for the mapping vector $\mathbf{x}$, and rows and columns of the matrix $\mathbf{x}^{(k)}$, e.g., $x_{ij}^{a_k}$.***	
• $B = \{b_i : i = 1, \dots, n_B\}$ directed acyclic graphs of tasks, where b_k refines a_k using merging rules.	
• $\Gamma^{a_k} = \{\Gamma_i^{a_k} : i = 1, \dots, n_\Gamma\}$	end-to-end chains
• $\Gamma_i^{(a_k)} = (e_i)_{i=1}^{Z}$	a chain of $e \in V(g_\tau^{a_k}) \cup M$
/* Functions related */	
• $Power(\mathbf{x})$	total power consumption of A in $\mathbf{x}$
• $Reliability_a(\mathbf{x})$	application reliability of $a \in A$ in $\mathbf{x}$
• $ResponseTime_\tau(\mathbf{x})$	response time of $\tau \in V(g_\tau^{a_k})(\mathbf{x})$
• $Delay_\gamma(\mathbf{x})$	age delay of $\gamma \in \Gamma^{a_k}$ in $\mathbf{x}$

*Note: the total elements in a set S is denoted by n_S, e.g., n_A denotes the number of applications in the set S, essentially it refers to its cardinality.
** For readability, we prefer to use $S_i^{(a_k)}$ in place of $S_i^{a_k}$.
*** IFor other uses of the iterators, they are defined in the context.

Definition 1 (AUTOSAR Software Application). [2] We define an AUTOSAR software appplication as a set of communicating AUTOSAR software components, which is modeled as a *directed acyclic vertex-weighted* graph $\langle V, L, w, \rangle$ of periodic runnables, where V denote runnables nodes, $a_{ij} \in L$ a directed edge from τ_i to τ_j and $i \neq j$ and denotes data dependency in the direction of the edge. The function $w : V \rightarrow (e_i, D, P, \mathcal{N})$ assigns computation cost such as worst-case execution time (WCET) e_i on the unit $n_i \in \mathcal{N}$, deadline D and period P.

According to mixed-criticality design [21], several applications can be executed on the same computing unit(s) and can share the on-board network, e.g., the CAN bus. Since the applications can be of different criticality, the runtime framework should provide an isolation mechanism both at the computing and network layers, which guarantees non interferenace execution of the higher-criticality applications from the lower-criticality applications. The application of mixed-criticality is widely researched in avionics, for instance to isolate flight control functionality from management related functionality. In the AUTOSAR developmenh method, the runtime enviornment, i.e., RTE, should provide the isolation mechanismrequirements, the computing nodes and the network should provide an isolation mechanism in order to avoid interference of lower-criticality applications on higher-criticality applications. Mixed-criticality is attracting automotive, due increasing functional complexity.

3.2. Scheduling Mixd-critical AUTOSAR Applications

The applications are scheduled on the heterogeneous execution platform by considering their respective requirements such as the criticality levels, reliability requirements, and end-to-end timing requirements. There are several techniques in the literature that deal with the scheduling of mixed-criticality applications on *uniprocessor* systems [21]. In our problem, though distributed applications, each task is mapped to a single computing node and the mapping is static. In this case, the schedulability of tasks can be performed per computing node, following the uniprocessor scheduling. Therefore, in the context of this work, the distributed applications are schedulable if the tasks, messages and the cause-effect chains meet their respective timing requirements, but also meet reliability requirements under the limited resources of the execution platform.

In this work, we consider the *partitioned criticality (PA)* technique to schedule the mixed-criticality applications, which basically prioritizes higher critical applications over their lower-criticality counterparts. In contrast to other techniques, PA does not require a runtime monitoring of tasks, e.g., using servers [22, 23, 24], though less efficient. Note: other scheduling techniques together with the PA technique can be used with our approach.

[2] Note: only relevant concepts of the official AUTOSAR software application definition is assumed to avoid unnecessary complexity.

$r_{i,j}^{(a_k)}$	m_h	(e_h, P)	$\prec r_{i,j}^{(a_k)}$
1,1	1	$(1, 10)$	2,1
2,1	1	$(1, 5)$	
2,2	1	$(1, 15)$	
3,1	2	$(1, 20)$	
4,1	3	$(1, 10)$	
5,1	3	$(1, 20)$	

Table 1: Runnables Timing Specifications.

$\tau_i^{(a_k)}$	$\bigcup r_{i,j}^{a_k}$	(e_h, P)
1	1,2;2,1	$(2, 5)$
2	2,2	$(1, 15)$
3	3,1	$(1, 20)$
4	4,1;5,1	$(1, 10)$

Table 2: Tasks-Runnables Mappings.

3.3. Refine AUTOSAR Application Using Tasks Graphs

According to the AUTOSAR specification [28], runnables are mapped to tasks, and the tasks execute the runnables respecting the formers' timing specifications. In the mapping process, one or more runnables can be merged to optimize the runtime execution by reducing the number of schedulable tasks. Thereore, through the mappings, eventually runnables graphs are refined by tasks graphs as shown in Figure 2 (b). In this work, the following rules are applied in order to merge any runnables $a, b \in V(a_k)$ to $v \in V(b_k)$, that is only if the following rules satisfy: (i) the runnables are co-hosted in the same computing node, that is $a \mapsto n \wedge b \mapsto n$, where $n \in N$; (ii) activation periods of the runnables are the same, i.e., $P_a = P_b$

If the rules are satisfied, the task's timing specifications are set as follows: i) the WCET of the task is set to the sum of the WCET of the runnables, $e_i^v = e_i^a + e_i^b$, ii) the period and deadline of the task is set to the least-common multiple (LCM) of the runnables' periods, $P_v = D_v = lcm(P_a, P_b)$. Otherwise, runnables are not merged, instead, each runnable that is not merged is mapped to a task while preserving the timing specifications of runnables.

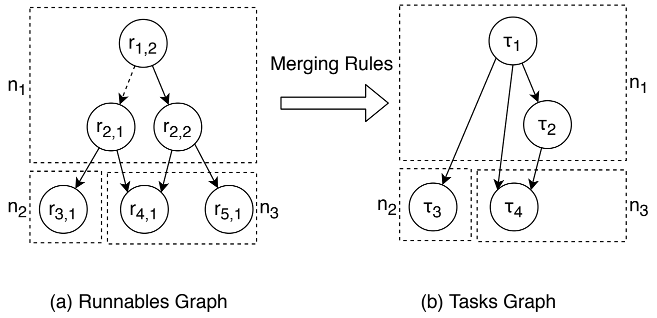


Figure 2: Example of a Software Application, Modeled as Directed Acyclic Graph, where $\dashrightarrow$, $\dashrightarrow$ denote triggering link, data-flow link, respectively.

3.4. Scheduling Tasks and Messages

We assume tasks are scheduled using the *fixed-priority preemptive scheduling polity* (FPPS) [**?**]. Initally, applications are priortized based on their criticality levels followng the PA technique, and within each application the tasks are prioritized according to the *deadline monotonic* (DM) priorities assignment.

$$cri(b_i) > cri(b_j) \implies \forall \tau_1, \in V(b_i) \forall \tau_2 \in V(b_j)\ Pri(\tau_1) > Pri(\tau_2),$$

where $\forall i, j : 1, ..., n_A \wedge i \neq j$, *cri* and *pri* are predicates which detemine the critiality and priority of tasks τ_1, τ_2, respectively; $V(b_i), V(b_j)$ are returns the tasks nodes, respectively for b_i and a_j.

The schedulability of tasks is checked by the classical response-time analysis shown in Equation (1) [25, 25], which computes the worst-case response time of each task, denoted by $delta_\tau$. According to the analysis, if the response time of each task is less than or equal to its deadline, that is $\delta_\tau \leq Deadline_\tau$, the taskset is schedulable otherwise it is not.

$$R_\tau = C_\tau + \sum_{j \in hp(\tau)} \left\lceil \frac{R_\tau}{P_j} \right\rceil C_j, \tag{1}$$

where C_τ, C_j are exection times of the lower and higher tasks, respectivel; $hp(\tau)$ is the predicate that returns the higher-priority tasks of task c_τ.

The messages in the CAN bus are scheduled using a fixed, non-preemptive scheduling policy. Similar to the tasks, the priority of messages followes the PA techniques to achieve the mixed-criticality requirement. This can easily be achived by inheriting the priority of sender task, that is $pri(m) = pri(\tau) | \tau = pre(m)$, where $pre(m)$ finds the sender task. The schdulability of messages is checked using the classical response-time analysis of the CAN network, presented by Rob Davis et. al [?] as shown in Equation (2). Thus, the worst-case response time of a message is computed as the summation of its *jitter* time (that is, the time taken by the sender task to queue for transmittion) J_m, the *interferance* time (that is, the message delay in the queue) w_m, and its *transmission* time (that is, the longest time for a signal or data to be transmitted) c_m.

$$R_m = J_m + w_m + c_m \tag{2}$$

$$w_m = B_m + \sum_{\forall k \in hp(m)} \left\lceil \frac{w_m + \tau_{bit}}{P_k} \right\rceil c_k \tag{3}$$

$$B_m = \max_{\forall k \in lp(m)} (c_k), \tag{4}$$

Note: we assume no jitter, therefore, the interferenece formula is reduced as shown in Equation (3), where B_m is the blocking time caused by the lower-priority messages using the CAN bus (since it is non-preemptive) and is computed by Equation (4); $hp(m)$ finds the higher-priority messages, which delay the trasmission of the message m in the queue as well as in the transmission.

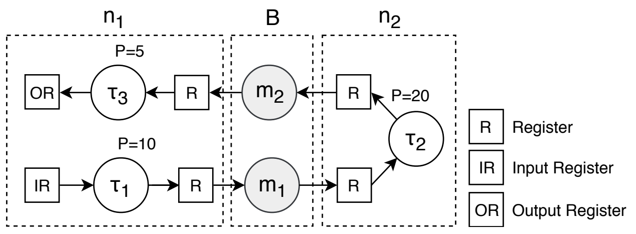


Figure 3: A Cause-effect Chain, mapped on nodes n_1 and n_2.

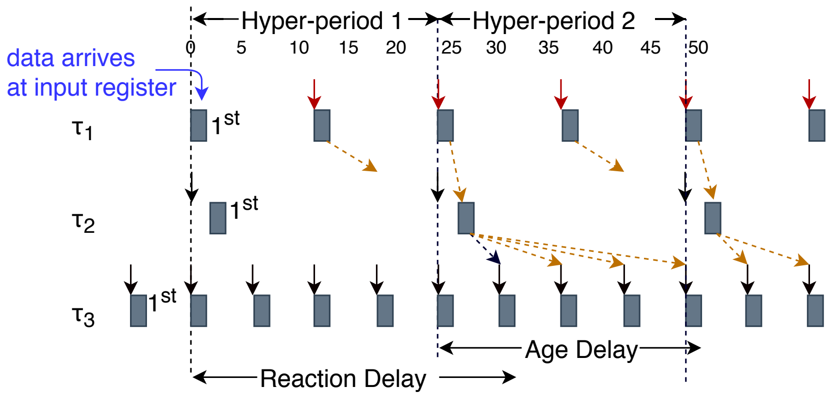


Figure 4: Reaction and Age Delays of the Cause-effect Chain, Shown in Figure 3.

3.5. Scheduling Cause-effect Chains

The software application can be considered as a set of *cause-effect chains* $\Gamma^{a_k} = \{\Gamma_i^{a_k} : i = 1, \ldots, n_\Gamma\}$. They represent sequences of actions triggered usually by external events (or causal actions or stimuli) and produce corresponding effects (or responses), e.g., pressing a rotary-wheel to activate a cruise control system, pressing a brake pedal to slow down a car, etc. The end-to-end requirements put upper-bounds on the duration of the stimuli-response elapse time. A chain is a directed paths in the task graph $(\tau_1, \tau_2, ..., \tau_n)$, where $\tau_* \in V(g_\tau)$, τ_1 is source task and τ_n is the sink task. An example of a cause-effect chain is shown in Figure 3, which consists of three independently clocked tasks τ_1, τ_2 and τ_3, and messages m_1 and m_2. It uses single-register buffers for communication, which is a common practice in control systems design, e.g., automotive software applications [26].

The end-to-end delay in a chain is the duration between the reading of data from the input register by the source task $Source(\Gamma_i^{a_k})$ to the writing of same data to the output register by the last (or sink) task $Sink(\Gamma_i^{a_k})$. Since we assume the chains consist of independently clocked tasks, the delay usually varies due to the undersampling and oversampling effects in the chains. In this work, we are particularly interested in the two types of delays that are widely used in the automotive and similar systems, namely the *age delay* and the *reaction delay.*

The difference between the two types of delays is demonstrated in Figure **??**. The tasks τ_1 and τ_2 execute on node n_1, whereas task τ_3 executes on node n_2. Note: τ_2 communicates with τ_3 via a CAN bus, which is not shown in the figure for simplicity. The red inverted arrows in the figure represent the reading of data from the input register, whereas the dashed-curve arrows represent the timed paths through which the data propagates from the input to the output of the chain. Thus, the age delay is the time elapsed between a stimulus and its corresponding latest non-overwritten response, i.e., between the 3^{rd} instance of τ_1 and the 10^{th} instance of τ_3. It is frequently used in the control systems applications where freshness of data is paramount, e.g., braking a car over a bounded time. And, the reaction delay is the earliest time the system takes to respond to a stimulus that "just missed" the read access at the input of the chain. Assume that data arrives just after the start of the 1^{st} instance of τ_1 execution. The data corresponding to this event is not read by the current instance of τ_1. In fact, the data will be read by the 2^{nd} instance of τ_1. The earliest effect of this data at the output of the chain will appear at the 7^{th} instance of τ_3, which represents the reaction delay. This delay is useful in the body-electronics domain where first reaction to events is important, e.g., in the button-to-reaction applications. For detailed discussion of the different delay semantics, we direct the reader to check research work by Mubeen et al. [13]. The age delay is computed using Equation (5) and Equation (6) for a single node and multiple nodes, respectively.

$$\Delta^{sub}(\Gamma) = \alpha(sink(\Gamma)) - \alpha(source(\Gamma)) + \delta(sink(\Gamma)) \qquad \text{single unit} \qquad (5)$$

$$\Delta(\Gamma) = \sum_{i \in I_\Gamma} \Delta^{sub}(i) + \sum_{j \in J_\Gamma} \delta^{msg}(j), \qquad \text{muliple units} \qquad (6)$$

where $\alpha(\tau)$ computes the activation of the task τ, based on the age-delay semanics.

Assume $\Gamma \in \Gamma^{a_k}$ is a chain, if the chain is mapped on a single node, the age delay is a mere difference between the activation of the sink task α_{τ_j} and the activation of the source task α_{τ_i} plus the worst-case response time of the sink task δ_{τ_j} in the longest timed path according to the semantics of the age delay. On the other hand, if the chain is mapped to multiple nodes, the age delay Δ_{multi} can be compositionally computed [27] as follows: the chain is partitioned into a set of sub-chains per node, indicated by the predicate $subch(\Gamma)$, and for each sub-chain $a \in subch(\Gamma)$, the age delay is computed recursively using the same method to used to compute the age delay for a single node, and the result is added to the response-times of the messages involved in the chain, $msg(\Gamma)$.

3.6. Reliability of Software Applications

Redundancy is the most common way to increase the reliability of a system. In this context, *software application reliability* refers to the probability that a software application functions correctly by the time t, or within the time interval

n_1	n_2	n_3
$q_{1,1}$	$q_{2,1}$	
$q_{3,1}$		

(a) Without Replication.

n_1	n_2	n_3
$q_{1,1}$	$q_{2,1}$	$q_{2,2}$
$q_{3,1}$	$q_{1,2}$	$q_{3,2}$

(b) With Replication.

Figure 5: Partition of the Software Application a_1.

$[0, t]$ [29]. Redandancy can be implemented according to different schemes, such as hot stand-by, cold stand-by, etc [30], in this work, we consider the hot-standby scheme, where replicated components maintain the same state. However, only the *primary* replicas act on the environment, e.g., activating an actuator. The primary software component is the one in operation and is indicated by $q_{i,1}^{(a_k)}$, the secondary software component, which is in the stand-by, by $q_{i,2}^{(a_k)}$, etc., for a software application A_k. Note: the software components are replicated unless the reliability requirements of applications are satisfied.

In this work the details of the redundancy scheme are abstracted away under the following assumptions: (i) Hot stand-by redundancy technique is used for the replacement of failed components, which are identical and are allocated on different nodes, ii) software components need to be replicated if the application's reliability requirement is not met without replication, otherwise they are not replicated; (ii) the time needed to detect and replace a faulty component is considered negligible and will not be taken into account in the response time analysis of tasks and delay calculation of cause-effect chains; (iii) Because of its simplicity, the mechanism for detection and replacement of faulty components will be considered fault-free, and therefore will not be included in the reliability calculations

Under these assumptions, the reliability of a software application is equivalent to the reliability of the execuiton platform such as the computg units the communication bus, if any, on wich the application is deployed. The reliability of a computing node or the bus is calculated using $e^{\lambda t}$, where λ is an exponentially distributed failure-rate. However, the reliability calcultion of the execution platform that services a software application is not trivial in the case with replication, e.g., the series-parallel reliability approach cannot be applied in the general case, due to the *functional* interdependency created between computing nodes as the result. To demonstrate the functional interdependency, let us assume a software application A_k, having component configurations with and without replication as shown Table 5b and 5a, respectively, where $q_{i,j}^{a_k}$ is the j^{th} software-component replica of software-component type $c_i^{a_k} \in C_i^{a_k}$. Note: for readability of the example, we remove the superscript k.

The reliability of the software application without replication forms a series path, indicated by the reliability block diagram (RBD) of Figure 6a, hence is computed as products of the reliability of n_1, n_2 and B. However, with replication, two computing nodes can form as series and parallel to service the sofware application, e.g., due to $q_{1,1}$ and $q_{2,2}$ or $q_{3,2}$, n_1 and n_3 make

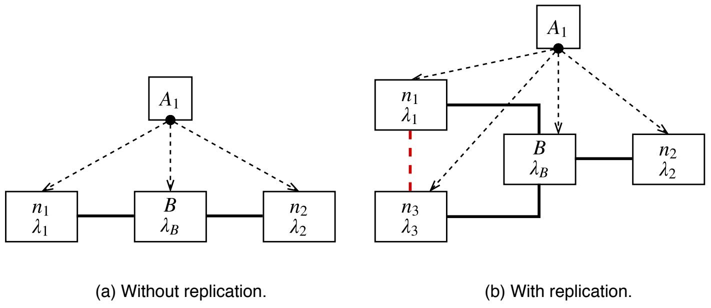


Figure 6: Reliability Block Diagrams of the Software Application.

series, and due to $q_{3,1}$ and $q_{3,2}$, the nodes make parallel, to relialize partional functionality of the application. In this case, the series-parallel diagram depicted in Figure 6b does not accurately capture the reliaiblity calculation of the application with replication. Note: the red-dashed line between n_1 and n_3 indicates the possibility of the computing nodes becoming series. To address this problem, we use the exact reliability calculation technique based on the enumeration of the different failure states of the computing nodes, that is the different failure states of the execution platform is enumerated exaustively, and subsequently, the total probability the software application functions is computed, which is discussed in great length in Subsection 4.3.

4. Problem Definition of Software Applications

The software applications A are partitioned on the execution platform $\langle N, B\rangle$ by satisfing the user-defined software applications requirements such as applications reliability RL^{a_k}, end-to-end timing requirements of chains $EE_i^{a_k}$ and applications criticality cl^{a_k}, and by meeting design constrains such as placement restrictions committed by the designer. Essentially, the partitioning is achieved by mapping software-component replicas (or software components) of applications effectively on the computing units N'. The partitioning should result in a reduced total power-consumption of the applications $Power(\mathbf{x})$, which is achieved by selecting lower-power computing units $N' \subseteq N$, where $\mathbf{x}$ is a possible mapping matrix, with $x_{ij}^{a_k}$representing the mapping of the software component $q_{i,j}^{(a_k)}$ to the compuing unit n_h, where $h = x_{ij}^{b_k}$. Note: under this partition, the reliability of applications $Reliability_{A_k}(\mathbf{x})$, the timing of chains $Delay_{\Gamma_i^{a_k}}(\mathbf{x})$ should satify the cooresponding requirements.

The power consumption of applications as well as the delay of chains depend on the tasks graphs $\bigcup g_{\tau}^{a_k}$, as well as the mapping of the tasks nodes to computing units and the links to the network bus. In this case, the tasks graphs has be constructed from the partition $\mathbf{x}$. However, if the tasks graphs are apriori to the software partitioning problem, which can be the case if no runnables from different components map to a single task, only the tasks nodes and the links are updated with the mapping information.

4.1. Task Graph Formulation

The task graph formulation implies the generation of a task graph b_k from a runnables graph a_k in $\mathbf{x}$, and involves two steps: (i) updating the mapping information of the runnables according to $\mathbf{x}$, that is for every component type $c_i^{a_k}$ mapped to $\mathbf{x}$, we reterive its runnables $R_{c_i}^{a_k}$. Subsequently, we update the computing units $\mathcal{N}_r^{a_k}$ to which the runnables are mapped as show in Equation (7), in linear-time complexity $O(\sum n_{C^{a_k}} * D)$, where $n_{C^{a_k}}$ are the number of component types in a_k , D is the maximum degree of replication allowed; (ii) the runnables graph is traversed iteratively to constract the corresponding tasks graph by applying the merging rules as shown in Equation (9), using the method described in Subsection **??**.

$$\forall ij \forall r \in R_{c_i}^{a_k}\ \mathcal{N}_r^{a_k} = n_h, \text{ where } h = x_{ij}^{b_k} \qquad k = 1, ..., n_A \tag{7}$$

$$a_k(\mathbf{x}) \xrightarrow{\text{Eqn. (13);Merging Rules}} b_k(\mathbf{x}) \qquad k = 1, ..., n_A \tag{8}$$

where $\mathcal{V}(a_k)$ is nodes of the runnables graph a_k .

However, if the tasks graph is apriori, that is, the runnables-to-tasks is static, only the nodes of the tasks graphs are updated with the mappings information using Equation (9). In the AUTOSAR development, the tasks are usually fixed and the mappings from runnables to tasks is usually static.

$$\forall ij \forall r \in T_{c_i}^{a_k}\ \mathcal{N}_\tau^{a_k} = n_h, \text{ where } h = x_{ij}^{b_k} \qquad k = 1, ..., n_A \tag{9}$$

4.2. Total Power Consumption

Power consumption refers to the energy usage of electronic components in an integrated circuit, e.g., processor, memory, I/O devices, etc., per time unit. There are several models (or techniques) to estimate the power consumption of a computing node. In this work, we use a technique based on processor load (or utilization) to estimate the average power consumption of a computinh node. Specifically, we use the linear polynomial model proposed by Fan et al. [31], which is shown in (10). The model states that the power consumption of a node is directly proportional to its load, and is inductively formulated from experimental results:

$$\mathcal{P}(u) = P_{idle} + (P_{busy} - P_{idle}) * u, \tag{10}$$

where u is the utilization of a computing unit, P_{idle} and P_{busy}, respectively refer to the power consumption measured at minimum and maximum processor loads. The measurements can be obtained by running performance benchmark suits, e.g., MiBench [32], AutoBench [33], etc. The utilization of the computing units for a partition $\mathbf{x}$ is computed as a sum of the utilization of their respective constituent tasks as shown in Equation (12). Finally, the total power-consumption of the applications is the sum of the power-consumption of the units as shown in Equation (13).

$$\mathcal{U}(\tau) = \frac{\text{WCET}_\tau}{P_\tau} \tag{11}$$

$$(u_1, ..., u_{n_N}) = \sum_{k=1}^{n_A} \sum_{\tau \in V(b_k)(\mathbf{x})} (\mathcal{U}(\tau) | Node_\tau = n_h, h = 1, .., n_N) \tag{12}$$

$$\mathcal{P}_{total}(\mathbf{x}) = \sum_{h=1}^{n_N} u_h(\mathbf{x}) \tag{13}$$

where $\mathcal{U}$, in Equation (11), computes utlization of task τ as a ration of its worst-case execution time and period, and u_h is utilization of node n_h.

4.3. Software-Applications Reliability Constraints

The applications reliability constraints ensure the mapping $\mathbf{x}$ satisfies the user-defined reliability requirements, that is $\forall k$ $Reliability_{A_k}(\mathbf{x}) \leq RelReq_{A_k}$. The reliability of each application is computed over t period of time from the computing units $N^{(a_k)}$ and the shared network bus B, where $N^{(a_k)}$ hosts a_k . The reliability is computed assuming exponentially distributed and constant failure rates of the units λ_{n_h} as well as the network bus λ_B. Thus, the reliability of an application is computed as a product of the reliability of the units and the network bus as shown using Equation (14). Note: if application does not use the shared bus $Reliability_B = 1$. Equation (15) finds the units $N^{(a_k)}$ that the application a_k uses by traversing the partition $\mathbf{x}$ in linear time.

$$Reliability_{a_k}(\mathbf{x}) = Reliability_{N^{a_k}}(\mathbf{x}) * Reliability_B \tag{14}$$

$$N^{a_k} = \{e \in N | \forall ij \; e = m_h\}, \text{ where } h = x_{ij}^{(k)} \tag{15}$$

Note: we assume applications are mutuallye exclusive, that is no shared components exist between any two applications, therefore, we can safety calculate the reliability of applications independently. Consequently, to increase readability, we remove the superscript (A_k) in the rest of this subsection.

The reliability of the units is $Reliability_N(\mathbf{x}) = e^{-\lambda_N(\mathbf{x})t}$, where is $\lambda_N(\mathbf{x})$ is the failure rate of an N-unit system over the partition $\mathbf{x}$. The system failure-rate is computed using the state enumeration as shown in [34], which is an exact technique to calculate reliability, as opposed to using series-parallel technique - motivated in Subsection 3.6. By applying the state enumeration technique, the system failure-rate can be defined as the probability a software application *fails* in the probability space $\langle \Omega, \xi, p, f \rangle$.

- $\Omega = \{0, 1\}$ are the possible outcomes (or states) of a computing unit. Assume the Boolean variable $s_h \rightarrow \Omega$, which indicates the state of n_h, then $s_h = 0$ indicates n_h fails and $s_h = 0$ indicates n_h operates. Thus, for computing units $N = \{n_1, .., n_{n_N}\}$, the states of the units (or configuration) is indicated by the N-cardinality set $S = \{s_1, ..., s_{n_N}\}$.

Units Config. $s \in \xi$	Probability p_s	Comonent Status $\forall i \; s_{c_i}$	Application Status f_s
$\{0,0,0\}$	0.0000000000	$\{0,0,0\}$	0
$\{0,0,1\}$	0.0000000099	$\{0,0,1\}$	0
$\{0,1,0\}$	0.0000000099	$\{1,0,0\}$	0
$\{0,1,1\}$	0.0000999800	$\{1,1,1\}$	1
$\{1,0,0\}$	0.0000000099	$\{1,0,1\}$	0
$\{1,0,1\}$	0.0000999800	$\{1,1,1\}$	1
$\{1,1,0\}$	0.0000999800	$\{1,1,1\}$	1
$\{1,1,1\}$	0.9997000299	$\{1,1,1\}$	1

Table 3: Example of Application Reliability Calculation using State Enumeration Over 10-years Operational Lifetime: an Application with Component Types $C = \{c_1, c_2, c_3\}$, Replicas $Q = \{c_{1,1}, c_{1,2}; c_{2,1}, c_{2,2}; c_{3,1}, c_{3,2}\}$ Partitioned on $N = \{n_1, n_2, n_3\}$ according to Figure (5), the Variable $s_{c_i} \in \{0,1\}$ Indicates if the Replicas of Type c_i Fails or Functions, Respectively.

- $\xi = \Omega^S$ are elementary events that correspond to the possible configurations of the units N, therefore, the events are mutually exclusive. Consider $N = \{n_1, n_2, n_3\}$, Table (3) shows the their possible configurations ξ. Assume the configuration $s \in \xi = \{0,1,0\}$, it shows n_1 and n_3 fail as indicated by $s_1 = 0, s_3 = 0$, respectively, and n_2 operates as indicated by $s_2 = 1$.

- $p : \xi \to [0,1]$ assings the configurations probabilities using

$$\forall s \in \xi \; p_s = \prod_{h=1}^{n_N} \lambda_{n_h} * (1 - s_h) + (1 - \lambda_{n_h}) * s_h$$

  where λ_{n_h} is the failure-rate of n_h. The probability p_s is the product of the probability of having the state s_h, which is λ_{n_h} if n_h fails, otherwise, $(1 - \lambda_{n_h})$ if n_h operates.

- $f : \xi \to \{0,1\}$ determines the status of the applciation in each state $s \in \xi$, that is $f_s = 0$ means the application fails, otherwise, $f_s = 1$ means the application operates, at the sate s.

Definition 2 (Software Application Failure). A software application fails in the configuration $s \in \xi$ if there exists a component type c_i where all of its replicas Q_i *fail*, otherwise, it functions, as shown using Equation (16). The component replica $q_i, j \in Q_i$ of type c_i fails if n_h fails, that is $s_h = 0$.

$$f_s(\mathbf{x}) = \begin{cases} 0 & \text{if } \exists i \; c_i | \forall j \; s_h = 0 \\ 1 & \text{otherwise} \end{cases} \qquad \text{where } h = x_{ij} \tag{16}$$

Thus, the failure rate of the N-unit system $\lambda_N(\mathbf{x})$ is the sum of the probabilities in which the application fails, that is

$$\lambda_N(\mathbf{x}) = \sum_{s\in\xi|f_s(\mathbf{x})=0} p_s(\mathbf{x})$$

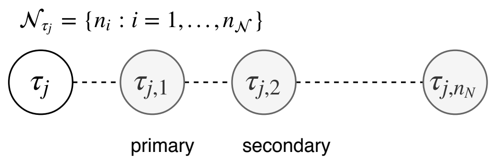
Example 1 (Reliability Calculation). *Let us assume we want to calculate the reliability of the application in Table 3 over a 10-year (or 87600h) operational lifetime. The reliability of the units is* $Reliability_N = e^{-\lambda_N t} = 0.99736671$*, where* $\lambda_N = p_1 + p_2 + p_3 + p_5 = 0.0000000301$*. Assume* $\lambda_B = 0.00000001$*, hence* $Reliability_B = e^{-\lambda_B t} = 0.99912438$*. Then, the reliability of the application is* $Reliability_N * Reliability_B = 0.99649339932$*.*

4.4. Chains Timing Constraints

The chains timing constraints over $\mathbf{x}$ ensure that the delays of chains meet thier respective end-to-end requirements, that is $\forall k \forall \gamma \in \Gamma^{a_k}\ Delay_\gamma(\mathbf{x}) \leq \mathrm{EE}^{a_k}_\gamma$. The delays calculations require that the tasks and messages in the chains meet their deadlines. Since, each task is used in at least one chain, we check the schedulability of all tasks and the message before we calculate the chains timing constraints. The schedulability of the tasks and the message is checked using the worst-case response-time anlaysis presented in Subsection **??** and Subsection **??**, respectively. Moreover, the schedulability of the chains is checked using the age-delay analysis shown in Subsection **??**.

4.4.1. Tasks Timing Constraints

The tasks timing constraints ensure that the tasks in the distributed system meet their respective deadlines. Tasks replicas mapped to the same unit do not improve realiability of the applications since we assume a permanent failure of computing units would lead to the total failure of the software running on the units. Consequently, the replicas except one are discarded, hence exists only a maximum of one replica per computing unit. The replica can be identified from other replicas based on the unit to which it is mapped. Assume $\tau \in \mathcal{V}(b_k)$ is a task node and $\mathcal{N}^{a_k}_\tau$ is the set of units to which its replicas can be mapped. Thus, the replicas of τ are represented a paired-set $\mathcal{R} = \{(\tau, n) | n \in \mathcal{N}^{a_k}_\tau\}$, where (τ, n) is the replica of τ mapped to n.


$\mathcal{N}_{\tau_j} = \{n_i : i = 1, \ldots, n_{\mathcal{N}}\}$
τ_j
$\tau_{j,1}$
$\tau_{j,2}$
τ_{j,n_N}
primary
secondary

To constract the tasks timing constraints, first we arrange the tasks per computing unit, T_{n_h}, so that the tasks graphs are not traversed everytime the response-time of tasks is computed. So, we traverse the tasks graphs, consequently in each task node $\tau \in V(b_k(\mathbf{x}))$, we check if the replica $\tau_{i,j}$ is mapped to $n_h \in N$. If the condition is true, we collect the task node τ_i as shown in Equation (19).

$$T_{n_h} = \{\tau \in \mathcal{V}(b_k(\mathbf{x})) | \forall j \in \mathcal{N}_{\tau_i}^{b_k} \; j = n_h\} \qquad \text{for all } h = 1, ..., n_N, \tag{17}$$

where where $h = x_{ij}^{b_k}$. The complexity of this equation, considering an adjucency matrix implementation, is exponential time $O(\sum |\mathcal{V}(b_k)| * n_N^2)$, where $\mathcal{V}(b_k)$ is tasks nodes in b_k, and N is the total computing units in the distributed system.

Subsequently, we calculate the response time of each task $\tau \in T_m$ in $\mathbf{x}$ by invoking the response-time analyais formula, and construct the tasks timing constraints as shown in Equation (18).

$$\forall \tau \in T_n \; \mathit{ResponseTime}_\tau(\mathbf{x}) \leq \mathit{Deadline}_\tau \tag{18}$$

4.4.2. Messages Timing Constraints

The messages timing constraints ensure that the response-time of message in the CAN bus meet their respective deadlines, which is equal to their periods. The messages M in $\mathbf{x}$ are determined from the communication links of the tasks graphs. Assume $(t1, t2) \in \mathcal{E}(b_k)$ is a link in the task graph. Due to replication, the link represents a set of sub-links $\mathcal{R}_{t2}^{b_k} \times \mathcal{R}_{t2}^{b_k}$ that communicate the replicas of $t1$, $\mathcal{R}_{t1}^{b_k}$, and the replicas of $t2$, $\mathcal{R}_{t2}^{b_k}$. Thus, if the task $(t1, n1) \in \mathcal{R}_{t1}^{b_k}$ and $(t2, n2) \in \mathcal{R}_{t2}^{b_k}$ are mapped to the same computing units, that is $n1 = n2$, the link is not mapped on the bus, rather uses a shared variable or inter-process communication. Whereas, if the tasks are mapped on different units, the link is mapped to a bus, hence a message is used to establish the communication between the tasks. Equation (19) shows how the messages in the distribured systems are determined by traversing the sub-links of the tasks graphs,

$$M = \{m_i | \forall (t1, t2) \in \mathcal{E}(b_k(\mathbf{x})) \forall (i, j) \in \mathcal{R}_{t1}^{b_k} \times \mathcal{R}_{t2}^{b_k} \; n_i \neq n_j\}, \tag{19}$$

where (i, j) is a sub-link of $(t1, t2)$, m_i is the message that the replica i uses to communicate with the replica j. We assume the message arrival behavior is periodic, similar to $t1$, i.e., $P_{m_i} = P_{t1}$. Likewise, it herits its criticality from $t1$, i.e., $\mathrm{CL}_{m_i} = \mathrm{CL}_{t1}$, so that the criticality is preserved at the network, as well.

4.4.3. Delay Constraints

The delay calculation of chains is multiplicity due to replication. Assume the chain $\Gamma = (\tau_i)_{i=1}^l = (\tau_1, ..., \tau_l)$, then the set of chains with replication is a cartesian product of the tasks replicas (or the tasks nodes mapping to computing units according to $\mathbf{x}$) in the chain, that is $\Gamma^*(\mathbf{x}) = \mathcal{R}_{\tau_1}^{b_k} \times, ..., \times \mathcal{R}_{\tau_l}^{b_k}$, where l is the chain length. Assume we want to calculate the delay of the chain $\gamma \in \Gamma^*$ compositionally, where $\gamma = (t_i)_{i=1}^l = (t_1, ..., t_l)$: first we identify the subchains I and messages J of the chain. The subchains I are subsets of the chain γ where the communication between the sender and receiver tasks of the chain use a network bus. That is, if t_i is the sender task, and its receiver task t_{i+1} is

$\gamma \in \Gamma^*$	$i \in I_\gamma$	$j \in J_\gamma$
$(\tau_1, n_1) \to (\tau_2, n_1) \to (\tau_4, n_2)$	$(\tau_1, n_1) \to (\tau_2, n_1), (\tau_4, n_2)$	$m_{(\tau_2,n_1),(\tau_4,n_2)}$
$(\tau_1, n_1) \to (\tau_2, n_1) \to (\tau_4, n_3)$	$(\tau_1, n_1) \to (\tau_2, n_1), (\tau_4, n_3)$	$m_{(\tau_2,n_1),(\tau_4,n_3)}$
$(\tau_1, n_2) \to (\tau_2, n_2) \to (\tau_4, n_2)$	$(\tau_1, n_2) \to (\tau_2, n_2) \to (\tau_4, n_2)$	$\emptyset$
$(\tau_1, n_2) \to (\tau_2, n_2) \to (\tau_4, n_3)$	$(\tau_1, n_2) \to (\tau_2, n_2), (\tau_4, n_3)$	$m_{(\tau_2,n_2),(\tau_4,n_3)}$

Table 4: Chains-with-Replication of Degree 2 for the Chain $\Gamma = \tau_1 \to \tau_2 \to \tau_4$, its Subchains I and Messages J.

mapped to a different unit, i.e., $n_{t_i} \neq n_{t_{i+1}}$, then $(t_h)_{h=i'}^{i} \in I$ is a subchain of γ and $m_{t_i} \in J$ is the message used by the subchain, where $0 \leq i' \leq i$, captured by the expression $(I; J) = \{(t_i)_{i=0}^{l-1}; m_{t_i} | n_{t_i} \neq n_{t_{i+1}}\}$.

Thus, the delay $\Delta_\gamma(\mathbf{x})$ for a mapping $\mathbf{x}$ is computed as the sum of the age delays of its subchains and the response-times of the messages,

$$\Delta_\gamma(\mathbf{x}) = \sum_{i \in I_\gamma(\mathbf{x})} \Delta_i^{sub}(\mathbf{x}) + \sum_{j \in J_\gamma(\mathbf{x})} \delta_i^{msg}(\mathbf{x}),$$

according to the age-delay formula shown in Equation (6), where Δ^{sub}, δ^{msg} are the functions that compute the age delay of i subchain, and the response-time of j message, respectively. Thus, the chains timing constraints are formulated over $\mathbf{x}$ and applications A using Equation (20).

$$\forall \gamma \in \Gamma^{a_k} \ \Delta_\gamma^{a_k}(\mathbf{x}) \leq \mathrm{EE}_\gamma^{a_k}, \tag{20}$$

Example 2 (Delay Calculation). *Consider the chain $\Gamma = \tau_1 \to \tau_2 \to \tau_4$ from Figure 2 where τ_1 and τ_2 realize the component types c_1, and τ_4 realizes c_2. The mapping of the components is shown in Figure 5 (b), i.e., with replication. Thus, the units to which τ_1 and τ_2 are mapped are $\mathcal{R}_{\tau_l}^{b_k} = \mathcal{R}_{\tau_2}^{b_k} = \{n_1, n_2\}$, and τ_4 to $\mathcal{R}_{\tau_4}^{b_k} = \{n_2, n_3\}$, by infering the mappings of respective components. Table 4 illustrates how compute the chains, considering replication of degree 2, which is $\Gamma^* = \mathcal{R}_{\tau_1}^{b_k} \times \mathcal{R}_{\tau_2}^{b_k} \times \mathcal{R}_{\tau_4}^{b_k}$, and also how to compute the subchains and messages of each chain $\gamma \in \Gamma^*$. The delays of the subchains is computed according to the age-delay semantics demonstrated in Subsection 3.5.*

4.5. Approximation of Delay Calculation

Due to the replication, the number of chains-with-replication per chain Γ grows exponentially as the degree of the replication D linearly increases, $|\Gamma|^D$. Likewise, the length of the chain has a polynomial effect as it grows. Moreover, the age-delay calculation is exhaustive search as demonstrated in the example. For these main reasons, the delays computation is normall expesive. Such computationally expensive operations are challenging for metaheuristics, due meta-heuristic algorithms compute large-space candidate solutions. Therefore, we propose an appoximation algorithm to efficiently compute the delays by selecting only the chain-with-repications that use the buses the most.

4.6. Software Allocation Optimization

The software allocation is defined as a single-objective optimization problem. The objective function $Power(\mathbf{x})$ is a cost function which minimizes the total power consumption of the software applications as deployed in the heterogenuous computing units, where $\mathbf{x}$ is the decision variable (or solution) of the optimization. The cost function is formulated in Equation 21, with inequality constraints shown by Equation (22, 23,24). The constraints ensure the solution meet the reliability requirements, the tasks deadlines, and the chains end-to-end requirements. Furthermore, the overlapping constrain shown in Equation (**??**) ensure that replicas are not allocated to the same computing units.

$$\min_{\mathbf{x}\in X}\ Power(\mathbf{x}) \qquad \text{subjected to:} \tag{21}$$

$$Reliability_{A_k}(\mathbf{x}) \leq RelReq_{A_k} \qquad \text{forall } k = 1, ..., n_{A_k} \tag{22}$$

$$\forall \tau \in T_{m_h}\ ResponseTime_{\tau}(\mathbf{x}) \leq Deadline_{\tau} \qquad \text{forall } h = 1, ..., n_M \tag{23}$$

$$\forall \gamma \in \Gamma^{a_k}\ Delay_{\gamma}(\mathbf{x}) \leq E2eReq_{\gamma} \qquad \text{forall } k = 1, ..., n_A \tag{24}$$

$$\tag{25}$$

where X is the search space of the problem, $\mathbf{x} \in X$ is a feasible solution, and $x_{ij}^{b_k} \in \mathbf{x}$ is a mapping of a component $q_{i,j}^{a_k}$ to the node m_h, where $h = x_{ij}^{b_k}$

In the next section, we discuss our proposed method to address the considered optimization problem.

5. Solution using Hybrid Particle Swarm Optimization (PSO)

In our previous work, we provided an ILP model for the same optimization problem, then the CPLEX solver returned optimal solutions to problems in the range *small* and *medim*, where the small problem refers to a software application with software components less than 10, chains less than 30, and similarly the medium problem refers to applications with components less than 15, and chains less than 40. The problems specifications are stipulated from the real automotive benchmark proposed by Kramel et al. [14]. However, the ILP approach, as also shown for similar problems, suffered from the scalability problem of large software allocation problems. In this section, we propose a metaheuristic approach based on the particle-swarm optimization (PSO), evolutionary, differential evolution (DE), hybrid PSO with DE, hill-climbing and stochastic hill-climbing.

Metaheuristics does not guarantee optimal solutions, nevertheless, the solutions can be good enough (or acceptable) in practice. Thus, although the power consumption of the applications may not be optimal, the solution can be deemed acceptable. PSO has been applied to solve a wide range of problems, including a task allocation problem [35], and DE is shown to scale well for problems with high dimensions. In fact, PSO and DE are used together for improved performance in several optimization problems [**?**], likewise, PSO is used with local search techniques such as Hill climbing to intensify the search [**?**]. Finally, we evaluate the different meta-heuristic methods based on solution quality and computation time for different software allocation problems.

5.1. Particle Swarm Optimization

PSO is a population-based technique proposed by Eberhart and Kennedy in 1995 to study social behavior, as inspired by natural swarm intelligence observed from the flocking of birds and schooling of fishes [36]. Since then, it is extended in order to address various metaheuristic optimization challenges, such as intensification, diversification, convergence analysis, local optima, parameter tuning, computation time, etc. It is successfully applied on several complex real-world problems, e.g., diagnosis and classification of diseases, efficient engineering designs, tuning control design parameters, scheduling problems, etc [37].

In PSO, the population (or swarm) $PN = \{p_1, p_2, \ldots, p_N\}$ is a collection of particles $p_i \in PN$, organized according to a certain population topology [38]. A particle has a position $\mathbf{x}$ and a velocity $\mathbf{v}$, which denote current location and direction of the particle's motion, and current momentum, respectively. It is a memory-based technique, that is, it remembers the best performance of every particle as well as the best performance of the swarm $\mathbf{z}$ in order to plan for the next move of the particles, where $\mathbf{y}, \mathbf{z}$ are position vectors and have the same dimensions as $\mathbf{x}$. The velocity of a particle is the resultant vector of its current velocity and the particles attraction vectors $(\mathbf{y}-\mathbf{x}), (\mathbf{z}-\mathbf{x})$, respectively, known as *cognitive* and *social* components of the particle's velocity formula, as shown in Equation 26. The attraction vectors impose force of attraction on the particle to move closer to their respective components. Thus, the next position of a particle is the resultant of its current position and its next velocity as shown in Equation (27).

$$\mathbf{v} \leftarrow \omega\mathbf{v} + c_1 Rand() \circ (\mathbf{z} - \mathbf{x}) + c_2 Rand() \circ (\mathbf{z} - \mathbf{x}) \tag{26}$$

$$\mathbf{x} \leftarrow \mathbf{x} + \mathbf{v} \tag{27}$$

where ω is the weight of the velocity, also known as *inertia coefficient*, and controls the convergence of the algorithm, c_1, c_2 are acceleration coefficients and controls the weight of attraction towards the cognitive and social components, respectively, $Rand() \in U(0, 1)$ is a random function, along the acceleration coefficients, is element-wise multiplied with the components to improve diversity of the search by introducing stochastic behavior.

Although PSO was originally proposed for continuous problem, it is applied to discrete problems successfully as well. In the latter case, the solutions are represented by *0-1* integer variables [39] or integer-linear by approximation to the nearest integer values [40], which is the representation employed adopted in our problem as it is compact, hence fewer decision variables. Accordingly, after the new position (or candidate solution) is determined, following Equations 26 and 27, the solution is discretized by rounding off the its elements to the nearest integer values, that is $\mathbf{x} \leftarrow [\mathbf{x}]$.

$$\begin{bmatrix}1 & 0 & 0\\0 & 1 & 0\\1 & 1 & 0\end{bmatrix}\begin{bmatrix}0 & 1 & 0\\0 & 0 & 1\\0 & 0 & 1\end{bmatrix}$$

Figure 7: Binary (0-1) Representation.

$$\begin{bmatrix}1 & 2\\2 & 3\\1 & 3\end{bmatrix}$$

Figure 8: Integer Representation.

Figure 9: Solution Representations for Components $\{c_1, c_2, c_3\}$ Mapped to Computiong Nodes $\{n_1, n_2, n_3\}$ based on Table 5b.

A meta-heuristic algorithm comprises two major parts: solution representation and an objective function. The solution representation shows the data structure that is used to represent each point in the problem space, and it has a significant impact on the performance of the meta-heuristic algorithm. The fitness function is used to evaluate the quality of candidate solutions based on their fitness to meet the problem objectives. A solution that delivers lower power consumption and violates less constraints is indicated by a lower fitness value. In the following, we describe the solution representation and the objective function proposed in our solution framework to run the meta-heuristic algorithms.

5.2. Solution Representation

The software allocation is a type of Assignment optimization problem, as such, the solutions are discrete values. There are two commonly used solution encodings (or representations) are binary (0-1) and integer. The binary variable indicates if a component is allocated to a computing node or not. In the integer representation, the variable indicate the computing-node identifier to which the component is allocated. The two representations are demonstrated in Figure 9 using the example provided in Figure 5.

In this work, we consider the integer representation due to efficient encoding (much fewer variables) as shown in Figure 28, and it is computationally more efficient, considering the various operations in the optimization problem, e.g., the binary representation shown in Figure 9 uses a pair of metrices to represent the primary and secondary replicas using 18 binary variables versus only 6 variables used in the case of integer representation shown in Figure 8. The solution to the allocation problem is represented by a vector-matrix $\mathbf{x} = \{\mathbf{x}^{A_k} : k = 1, ..., N_a\}$ as shown in Equation (28), where $\mathbf{x}^{(k)}$ is a matrix of size $n_C \times K$, and $x_{ij}^{(a_k)} = h \in \{1, \ldots, n_N\}$ denotes the mapping of the software-component replica $q_{i,j}^{(a_k)}$ to the computinG node n_h.

$$\mathbf{x}^{(k)} = \begin{bmatrix} x_{11}^{(k)} & x_{12}^{(k)} & \cdots & x_{1K}^{(k)} \\ x_{21}^{(k)} & x_{22}^{(k)} & \cdots & x_{2K}^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N_c1}^{(k)} & x_{N_c2}^{(k)} & \cdots & x_{N_cK}^{(k)} \end{bmatrix} \tag{28}$$

5.3. Fitness Function

The fitness function $f : \mathbf{x} \to \mathbb{R}$ is a type of objective function that summerizes the contriutions of the decision variables via real numbers. The fitness value is used to compare feasible solutions, that is the higher the fitness, the better. In the context of metaheuristics, it is highly desirable to integrate the goal function and all constraints into one function that can be used as a fitness function. [cite to talbi2009metaheuristics and faragardi2018efficient]. Thus, it combines the objective function, which is the power-consumption minimization, with the constraints such as the reliability and timing constraints into a single function by using a penality function.

The benefit of using a single function, including all penalty functions, is to provide a metric to distinguish between two unfeasible solutions. For example, let us assume that $\mathbf{x}_1$ and $\mathbf{x}_1$ are two different solutions for the allocation problem while both violate some constraints of the problem. Let us also assume that solution $\mathbf{x}_1$ slightly violates only one constraint, whereas solution $\mathbf{x}_1$ significantly violates multiple constraints. If the heuristic algorithm can perceive the difference between $\mathbf{x}_1$ and $\mathbf{x}_2$ in terms of being far away from a feasible solution, the fitness function guides the search toward a feasible solution more efficiently, compared to the case that the heuristic algorithm only knows that they are both infeasible. The integration of the goal function with all the penalty functions is a promising solution to provide knowledge about how far an unfeasible solution is from a feasible solution.

Consequently, the original constrained optimization problem is transformed into unconstrained optimization problem, by extending the objective function $P(\mathbf{x})$ with the constrains, represented by a set of *penalty functions* $\{\phi_{reliability}(\mathbf{x}),\ \phi_{deadline}(\mathbf{x}),\ \phi_{e2e}(\mathbf{x}), \phi_{rep}(\mathbf{x})\}$. The first penalty function corresponds to the reliability constraint which returns 0 if the reliability constrain is not violated, otherwise returns a positive number denoting how far the reliability constraint is violated. The further the violation, the higher value of the penalty function. Similarly, the other penalty functions correspond to the deadline, the end-to-end timing requirement, and the replication constraints, respectively. I ndeed, the penalty function penalizes the candidate solution by increasing its fitness (for our minimization problem), thus discriminating the solution. Section **??** explains how our solution framework formulates the penalty functions.

The fitness function $f(x)$ is computed as follows.

$$\min_{\mathbf{x}\in X}\ f(\mathbf{x}) = P(\mathbf{x}) + \beta_1\phi_{reliability}(\mathbf{x}) + \beta_2\phi_{deadline}(\mathbf{x}) + \beta_3\phi_{e2e}(\mathbf{x}) + \beta_4\phi_{rep}(\mathbf{x}) \tag{29}$$

where β_1 to β_4 are penalty coefficients used to tune the weight of the penalty functions with regard to the range of the objective function. In Section 5.5, the proper value of the penalty coefficients is discussed in more details.

5.4. Penalty Function

$$\phi_{reliability}(\mathbf{x}) = \sum_{k=1}^{n_{A_k}} Max\{0, Reliability_{A_k}(\mathbf{x}) - RelReq_{A_k}\} \tag{30}$$

$$\phi_{deadline}(\mathbf{x}) = \sum_{\forall\tau\in T_{m_h}} Max\{0, ResponseTime_{\tau}(\mathbf{x}) - Deadline_{\tau}\} \tag{31}$$

$$\phi_{e2e}(\mathbf{x}) = \sum_{\forall\gamma\in\Gamma^{a_k}} Max\{0, Delay_{\gamma}(\mathbf{x}) - E2eReq_{\gamma}\} \tag{32}$$

5.5. Penalty Coefficients

To calculate the value of the penalty coefficient ($\beta_1 to \beta_4$) we use the same analytical approach proposed in[cite to faragardi2018efficient] where the value of each penalty coefficient determined separately with respect to the relative proportion of the range of the penalty function to the range of the objective function (which is P(x) in our problem). Indeed, the penalty coefficient should be determined such that all the feasible solutions have a lower fitness value in comparison to infeasible solutions, meaning that all the feasible solutions are always preferred to an infeasible solution [**?**]. On the other hand, the penalty coefficient should not be extremely large since it hinders the search algorithm to search among infeasible solutions to find a way to reach the global optimum [**?**].

To calculate the minimum value of β_1 we consider two solutions for the problem. Solution 1 has the best power consumption (denoted by P^{min}) while it just infinitesimally violates the reliability constraint. Solution 2 has the worst possible value of $P(x)$ (denoted by P^{max}), while it satisfies the reliability constraint. We expect that Solution 2 has a lower fitness value than that of solution 1. Accordingly,

$$P^{min} + \beta_1 \times Min\{PenaltyValue\} > P^{max} + 0 \tag{33}$$

Let us assume that (i) $P^{min} = 0$, (ii) P^{max} is set equal to the total power consumption of all nodes when they are fully utilized, and (iii) $Min\{PenaltyValue\} = 10^{-8}$ which is the minimum value of $\phi_{reliability}$ in an unfeasible solution. Hence,

$$\beta_1 > 10^8 \times P^{max} \tag{34}$$

Our experiments also verify this theoretical discussion. We observed that when $\beta_1 = 10^8 \times P^{max}$, we always converge to a feasible solution and when it is set to a lower value, in some experiments, we converge to an infeasible solution. We also observed that when β_1 is set to a significantly higher value, the deviation from the best fitness value found in multiple experiments goes up, and the average fitness value is increased, thereby, the quality of the solutions is decreased.

Similarly, for the other penalty coefficients we use these calculations which result in

$$\beta_2 > 1 \times P^{max} \tag{35}$$

$$\beta_3 > 1 \times P^{max} \tag{36}$$

$$\beta_4 > 1 \times P^{max} \tag{37}$$

It should be noted that the minimum violation of $\phi_{deadline}$, ϕ_{e2e}, and ϕ_{rep} is one.

5.6. Hybrid PSO

The canonical PSO technique uses the constriction factors to balance exploitation and exploration of the search space, that is to deliver better quality solutions. Nevertheless, it still suffers from local minima especially for complex and large problems that exhibit especially multimodal behavior. Hybridization of PSO is one the most widely studied approach in the improvement of the the PSO technique. Basically, it combines other optimization techniques, for instance to intensify local search, and improve diversification by introducing stochastic search. However, hybridization of PSO usually incurs additional computation time. Therefore, the benefit of hybridization has to be studied carefully in conjunction to computation time. Moreover, it should not complicate the user-configurable parameters, to be inline with the philosophy of PSO for ease-of-use.

PSO is hybridized with several optimization techniques, such as Genetic Algorithm (GA), DE, local searches (e.g., Hill-climbing, gradient decent, etc.), ant colony, simulated annealing, etc. Of which, it is shown to perform better when hybridized with DE on constrained, discrete, large benchmarks. Furthermore, it is shown to perform better when hybridized with Hill-climbing (specifically *Steepest-descent* variant)for software allocation problem [] in particular. In this paper, we hybridize PSO with DE (DEPSO) and Hill-climbing (HCPSO) to the solve the software allocation problem as formulated in Equation (x). In the latter case, we also apply the stochastic variant of Hill-climbing (SHPSO) in order to offset stagnation of the steepest Hill-climbing when applied on large software allocation problems.

```
input  : PSO parameters, DE parameters
output: Software allocation solution sBest.x
Particles $P \leftarrow$ initPSO();
while termination criteria do
    $P \leftarrow$ComputePersonalBest($P$);
    sBest $\leftarrow$ComputeSwarmBest($P$);
    foreach $p \in P$ do
        computeParticleVelocity($p$) according to Equation (26);
        computeParticlePosition($p$) according to Equation (27);
    end
    if interval criteria then
        $P \leftarrow$ optimizeUsingDE($P$);
        // $P \leftarrow$ optimizeUsingHC($P$)
        // $P \leftarrow$ optimizeUsingSHC($P$)
    end
end
```

Algorithm 1: Hybrid PSO Algorithms.

5.7. Differential Evolution

Similar to PSO, Differential Evolution (DE) is a population-based metaheuristic technique for the global optimization which includes non-linear and non-differentiable problems. It was initially proposed by Storn and Price in 19995 [41], since then it has improved with regard to the different operators of DE such as mutation and crossover, and variants over population topology and hybridization [42]. It is a parallel search technique, therefore, is ideal for computationally intensive problems, and employs mutation and crossover operators that allow the search to skip local minima as opposed to PSO.

In every generation, the population undergoes mutation, crossover, and selection according to the formulas shown in Equation , (39), and (40), respectively. A mutant vector v is created from randomly selected elements $\{a, b, c\} \in PN$ according the mutation operation shown in Equation (38), that is by adding the base matrix to the weighted difference matrix $F \circ (b-c)$, where F controls the amplification of the $(\mathbf{b}-\mathbf{c})$ variation.

$$\mathbf{v} \leftarrow \mathbf{a} + F \circ (\mathbf{b} - \mathbf{c}) \tag{38}$$

$$u_{ik} \leftarrow \begin{cases} v_{ik} & \text{if } U(0,1) \leq CF \text{ and } h = (i * K + k) \\ x_{ik} & \text{if } U(0,1) > CF \text{ and } h \neq (i * K + j) \end{cases} \tag{39}$$

$$\mathbf{x} \leftarrow \begin{cases} \mathbf{u} & \text{if } f(\mathbf{u}) < f(\mathbf{x}) \text{ functions} \\ \mathbf{x} & \text{otherwise} \end{cases} \tag{40}$$

DE complements the classical PSO by introducing stochastic behavior via the evolutionary operators such as mutation, cross-over and selection. In this specific hybridization approach, we allow the DE algorithm to run intermittently for some number of generations before the next PSO generation starts.

5.7.1. Hill-climbing PSO

Hill-climbing is a popular local search based on the notion of *neighborhood*, that is, the candidate solution (or neighbor) that performs better is selected iteratively until no improvements can be made. The software allocation solution $\mathbf{x}$ is neighbor to $\mathbf{x'}$ if $\mathbf{x} = \mathbf{x'}$ except $\exists i, j |\ x_{ij} \neq x'_{ij}$, that is, a single mapping is different. In every iteration, the best neighbor is selected, and subsequently replaces the current candidate solution if it performs better, and continues until maximum iteration, this variant is known as Steepest-descent Hill-climbing (SHC).

Since SHC exhaustively checks all neighbors before moving to the next iteration, the computation time is high especially for high-dimensional problems. To offset this problem, we also apply the stochastic version of Hill-climbing. In the later case, the neighbor is selected randomly, first by selecting the dimension, that is the component c_{ij}, where $i = U(1, I)$ and $j = U(1, K)$, second, selecting the value, that is the node n_j, where $j = U(1, J)$. If the neighbor improves the current candidate solution sufficiently, the search moves to the next iteration, which is until no more improvements can be made.

6. Evaluation

In this section, we evaluate our proposed hybrid PSO algorithms for the allocation of software applications on heterogenous computing units, which conform to the system model presented in Section 3. The algorithms are evaluated against different specifications of automotive software applications and execution platforms with regard to effectiveness, stabilitity and scalability. The software-application specifications consist of the number of software components c, runnables r, tasks t and cause-effect chains g. The specifications are synthesized from the automotive benchmark proposed by Kramel et al. [14]. The benchmark indicates a strong correlation between runnables and cause-effect chains in terms of timing and activation patterns. It shows the timing specifications of runnables and their shares in an engine management system. Moreover, it shows the activation patterns of cause-effect chains, the runnables per activation and their shares in the system. The engine management system is one of the most complex automotive systems in the vehicular electical/electronic execution platform.

Software Applications Benchmark. Based on our experience in the automotive industry, the benchmark results are extrapolated to characterize different classes of automotive software applications specifications, that is by varying the parameters related to the software components, runnables, and cause-effect chains. The different classes of specifications range from Spec-I to Spec-V as shown in Table **??**. The specification classes are useful to evaluate and discusse the effectiveness and scalability of the different optimization algorithms.

Parameter	Spec.-I	Spec.-II	Spec.-III	Spec.-IV
Components c	≤ 10	≤ 15	≤ 20	≤ 80
Runnables r	≤ 50	≤ 100	≤ 500	≤ 1000
Tasks t	≤ 30	≤ 60	≤ 80	≤ 100
Cause-effect chains g	≤ 30	≤ 40	≤ 60	≤ 100
Activation-pattern	$\{2, 3, 4\}$			
share of activation-patterns	$\{0.7, 0.2, 0.1\}$			

Table 5: Specification of the Applications for Evaluation.

Parameter	Range
EE	$100n_\Gamma$
RL	0.99999999
CL	{A,B,C,D}

Table 6: Ranges of Values for Applications Requirements.

Parameter	Range
Nodes n_N	$4 - 10$
P_{min}, P_{max} (Watt)	$1 - 10$
λ_n (h^{-1})	$10^{-4} - 10^{-2}$
λ_B (h^{-1})	$10^{-4} - 10^{-2}$
Hz (MHz)	$80 - 800$

Table 7: Ranges of Values for Execution Platforms.

The first specification class Spec-I encompases small software applications with number of components less than 10, runnables less than 50, tasks 30, cause-effect chains less than 30. The Spec-I and Spec-II classes represent medium and large software applications, and the last specification class is introduced to strech the performance analysis.

Execution Platform Specifications. Likewise, the specifications for an execution platform consist of the processor speed, power specifications and failure rates of computing units, and we assume the range of values to these parameters as shown in Table 7.

Applications Requirements Specifications . Table **??** shows the range of values used in our experiment to specify the requirements of software applications, that include the end-to-end timing requirements EE of chains, the reliability requirement RL and the criticality level CL. The end-to-end requiremens are assumed as a function of length of the chain n_Γ, that is the longer the chain, the higher the number. The reliability range of safety-critical automotive application is usually given in higher degree of 9, for operation of over a long period of time, which implies almost no failure during the specified duration.

Evaluation Setup. The evaluation is conducted on a MacBook Pro laptop comptuer, with hardware specifications as follows: Intel Core i7 processor type, 2.6.GHz processor speed, 6 Cores , 9 MB L3 cache, and 16 GB memory.

Algorithm	Parameters Settings
PSO	Particle Swarm Optimization: learning factors $c_1 = c_2 = 1.49445 \in [0, 4]$, number of particles 40, iterations 5000
DE	Differential Evolution: crossover $CR = 0.5 \in [0, 1]$, scale factor $F = 0.7 \in [0, 2]$
PF	Penality Function: β_1=, β_2 =, β_3 =

Table 8: Parameters Settings of the Metaheuristic Optimization. .

6.1. Result

We conduted two experiments: i) the first experiment is designed to compare the performance such as convergence time, computation time, optimality (or quality of solutions), and stability of solutions of the meta-heuristic algorithms used in this paper, ii) the second experiment is designed to evaluate the overhead of increasing replication on the optimization especially on the computation of cause-effect chains, and also to evaluate the effect of the approximation algorithms proposed in Subsection x to reduce the overheadd and maybe trade-off with optimality of solutions.

Experiment 1. According to the specifications of the range discussed, we synthesized six optimization problems as shown in Table 9. The problems emulate the software allocation safety-critical distributed automotive applications on a CAN network of heterogenous computing nodes. The problems are identified by handlers of type $\langle c_i g_j n_i \rangle$ to improve readability, where the c, g, n variables indicate respectively the number of components, cause-effect chains and computing nodes. The $c_6 g_{10} n_4$ and $c_8 g_{20} n_6$ problems conform to Spec-I and denote a small (or light) optimization problem, the $c_{10} g_{20} n_8$ problems is based on Spec-II and denote a medium size problem, the $c_{20} g_{30} n_{10}$, $c_{50} g_{40} n_{20}$ and $c_{80} g_{60} n_{20}$ are based on Spec-III and denote large size problem. The optimization problems are executed each 30× using our ILP method proposed in [43] and the meta-heuristic algorithms presented in Section 5. The optimization parameters such as the penality function coefficient and the meta-heuristic parameters control the metaheuristics optimization, and their settings are shown in Table 8. The settings are obtained from literature as best practices of using the algorithms, as well as from our experimentaion of the algoriths with the problems at hand. Subsequently, we recorded the computation time, fitness values, power-consumption delivered by each algorithm.

Experiment 2. Usually the replication exerts heavy computation over the calculation of the cause-effect delays due its combinatorial nature. The approximation technique, which is presented in Subsection **??**, optimizes the calculation of the cause-effec chain delays in the presenece of replication. We executed the optimization problems $c_{50} g_{40} n_{20}$ and $c_{80} g_{60} n_{20}$ with 2 and 3 degrees of replication, and also with and without the approximation technique applied according to the specification in Table 10. The degree of replication indicates the multiplicity of each component in the software applications.

Identifier	Components c	Runnables r	Chains g	Nodes n
$c_6g_{10}n_4$	6	60	10	4
$c_8g_{20}n_6$	8	80	20	6
$c_{10}g_{20}n_8$	10	100	20	8
$c_{20}g_{30}n_{10}$	20	200	30	10
$c_{50}g_{40}n_{20}$	50	500	60	20
$c_{80}g_{60}n_{20}$	80	800	60	20

Table 9: Specifications of Optimization Problems.

Identifier	Chains g	Replication d	Problem Id.
$g_{30}d_2$	30	2	$c_{50}g_{40}n_{20}$
$g_{30}d_3$	30	3	$c_{50}g_{40}n_{20}$
$g_{30}d_2$	60	2	$c_{80}g_{60}n_{20}$
$g_{30}d_3$	60	3	$c_{80}g_{60}n_{20}$

Table 10: Specifications of Chains g and Degrees of Replication d, Used in Experiment 2.

6.2. Analysis

In this subsection, we analyze the results from experiement 1 and 2, respectively.

6.2.1. Analysis of Experiment 1

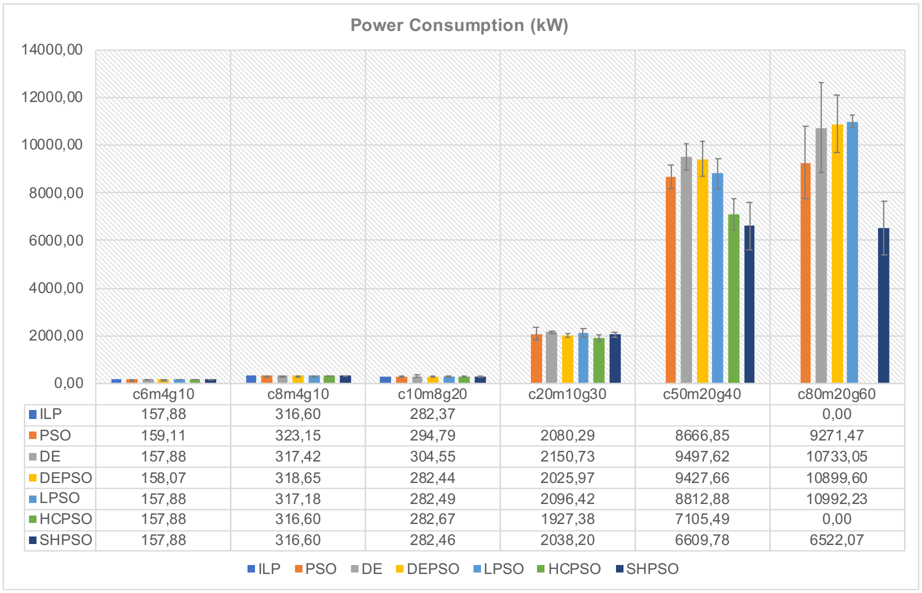
Table 11 shows a summary of the evaluation results from executing Experiment 1 such as the average and standard deviation of the computation times and fitness values, as well as the quality of solutions. The latter is determined by comparing the power-consumption outcomes delivered from each algorithms agains the optimal or best solutions found (or benchmarks), which are indicated by the **boldface** type. It simplly indicates how optimal or good the solution is as compared to the benchmark. In the first three optimization problems, the ILP is the benchmark since it returned optimal solutions. Similarly, the SHPSO is the benchmark in the problems $c_{20}g_{30}n_{10}$ and $c_{50}g_{40}n_{20}$, and SHPSO is the benchmark in the last probem $c_{80}g_{60}n_{20}$.

We analyze the results over three metrices: solution quality, computation time, and stability, in order.

Solution Quality. In the 1^{st} optimization problem $c_6g_{10}n_4$, the ILP, DE, LPSO, HCPSO, SHPSO returned the optimal power consumption, which is 227KW, but DEPSO and PSO returned near optimal solutions with $> 99\%$ quality measures. In the 2^{nd} problem $c_8g_{20}n_6$, similar results are obtained from ILP, HCPSO and SHPSO, which are optimal, but this time, in contrast, DE, LPSO and DEPSO performed worse by less than 1% but better than PSO by 2%. In the 3^{rd} problem $c_{12}g_{20}n_8$, only ILP returned the optimal solution, followed by DEPSO, LPSO, HCPSO, SHPSO with near optimal solutions of $> 99\%$ quality measure,

Problem	Algorithm	Fitness		Time (ms)		Quality
		Mean	SD	Mean	SD	
$c_6g_{10}n_4$	**ILP**	227.88	0	309	57.74	100.00
	PSO	229.11	2.38	0.12	0.34	99.46
	DE	227.88	0	0.01	0	100.00
	DEPSO	228.07	0.31	0.09	0.01	99.92
	LPSO	227.88	0	0.02	0.02	100.00
	HCPSO	227.88	0	0.03	0	100.00
	SHPSO	227.88	0	0.13	0.03	100.00
$c_8g_{20}n_6$	**ILP**	406.6	0	4148.3	95.77	100.00
	PSO	415.15	12.4	0.07	0.15	97.94
	DE	407.42	1.05	0.03	0.02	99.80
	DEPSO	409.65	8.8	0.17	0.01	99.26
	LPSO	407.18	0.53	0.32	0.73	99.86
	HCPSO	406.6	0	0.13	0.06	100.00
	SHPSO	406.6	0	0.29	0.14	100.00
$c_{10}g_{20}n_8$	**ILP**	442.37	0	14049.1	150.84	100.00
	PSO	448.79	12.61	0.79	1.37	98.57
	DE	451.55	17.72	0.23	0.41	97.97
	DEPSO	442.44	0.19	1021.46	2263.76	99.98
	LPSO	442.49	0.17	1062.51	2338.73	99.97
	HCPSO	442.67	0.21	7.57	22.68	99.93
	SHPSO	442.46	0.19	10.73	61.31	99.98
$c_{20}g_{30}n_{10}$	ILP	NA	NA	NA	NA	NA
	PSO	64595.28	9544.82	11.27	9.73	65.74
	DE	53655.73	4134.84	22.15	7.95	79.14
	DEPSO	44055.97	4237.81	192.95	230.83	96.38
	LPSO	58603.42	6617.49	19.83	6.98	72.46
	HCPSO	42462.38	1643.71	247.05	104.36	100.00
	SHPSO	42558.2	2770.52	114.52	102.41	99.77
$c_{50}g_{60}n_{20}$	ILP	NA	NA	NA	NA	NA
	PSO	1298680.85	38557.68	1753.43	776.16	98.26
	DE	1460553.62	34599.66	571.43	248.46	87.37
	DEPSO	1384474.66	32550.41	4925.97	4809.57	92.17
	LPSO	1430847.88	32045.32	640.86	320.33	89.18
	HCPSO	1276036.05	65320.02	17445.87	15796.87	100.00
	SHPSO	1336679.78	98051.36	1074.4	339.83	95.46
$c_{80}g_{60}n_{20}$	ILP	NA	NA	NA	NA	NA
	PSO	2692638.14	46015.42	324.95	103.66	91.60
	DE	2737416.39	23780.06	716.97	207.19	90.10
	DEPSO	2604249.6	46945.89	4018.55	12.37	94.71
	LPSO	2650992.23	35813.35	1005.74	375.25	93.04
	HCPSO	NA	NA	NA	NA	NA
		2466535.41	89380.36	2147.79	357.58	100.00

Table 11: Fitness and Allocation Time of the ILP and the Metaheuristic Techniques, for the Increasing Sizes of the Software Allocation Problem.



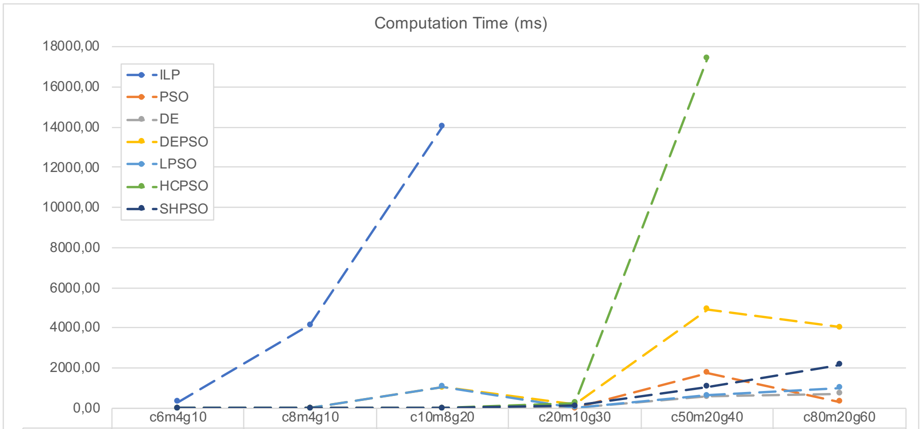
	c6m4g10	c8m4g10	c10m8g20	c20m10g30	c50m20g40	c80m20g60
ILP	157,88	316,60	282,37			0,00
PSO	159,11	323,15	294,79	2080,29	8666,85	9271,47
DE	157,88	317,42	304,55	2150,73	9497,62	10733,05
DEPSO	158,07	318,65	282,44	2025,97	9427,66	10899,60
LPSO	157,88	317,18	282,49	2096,42	8812,88	10992,23
HCPSO	157,88	316,60	282,67	1927,38	7105,49	0,00
SHPSO	157,88	316,60	282,46	2038,20	6609,78	6522,07

Figure 10: (Near) Optimal Power Consumption of the Different Software Allocation Problems.

and rest performed worse. In the last three optimization problems $c_{20}g_{30}n_{10}$, $c_{50}g_{40}n_{20}$ and $c_{80}g_{60}n_{20}$, ILP did not return solutions due to extremly large computation time, hence terminated manually. However, the hybrid algorithms based on hill-climbing such as HCPSO and SHCPSO performed well, followed by DEPSO in the problems $c_{20}g_{30}n_{10}$ and $c_{80}g_{60}n_{20}$. However, HCPSO failed to return solutions in the largest problem $c_{80}g_{60}n_{20}$ but its stochastic verion SHCPSO did.

Convergence Time. In the case of metaheuristics, the convergence time refers to the amount of time taken by the algorithm to return solutions before the steady state where new fitness values are observed. In this evalution, it is calculated over a maximum of 5000 iterations (or generations) only for the duration before steady period, which is bounded by 5 minutes. Note: the steady time, where no fitness values change within the maximum iterations, is not considered in the converge time. Figure 11 summarizes the computation time of the algorithms for the samples listed in Table 11. For the samples the ILP method returned solutions, the computation times are usually larger than the rest, which are in milliseconds for the 1^{st} sample and in seconds for the 2^{st} and 3^{rd}. The the meta-heuristic algorithms, the convergence time is in milliseconds for the first four samples, and is in seconds for the rest. However, the computation times of the meta-heuristic algorithms, which are not shown in the table usually took less than 50 minutes for the larest sample.

Solutions Stability. the PSO, DE are characterized by random search which enables exploration of higher dimensional problems possible. However,



	c6m4g10	c8m4g10	c10m8g20	c20m10g30	c50m20g40	c80m20g60
ILP	309,00	4148,30	14049,10			
PSO	0,12	0,07	0,79	11,27	1753,43	324,95
DE	0,01	0,03	0,23	22,15	571,43	716,97
DEPSO	0,09	0,17	1021,46	192,95	4925,97	4018,55
LPSO	0,02	0,32	1062,51	19,83	640,86	1005,74
HCPSO	0,03	0,13	7,57	247,05	17445,87	
SHPSO	0,13	0,29	10,73	114,52	1074,40	2147,79

Figure 11: Computation Time of the Various Algorithms for Solving Different Instances of the Software Allocation Problem.

somestime this creates instability in the solutions, that is for the same problem, it is possible to observe different performance, e.g., fitness values, computation time, etc. The stability of the solutions depend on the nature of the algorithms as well as the problems at hand. Therefore, it is crucial to evaluate the stability of the meta-heuristic algorithms used in this work. One way measuring the stability is using standar deviation, and Figure x and Figure y show the deviation of each algorithms for the (near) optimal power consumption of the different samples.

In general, with regard to quality of the solutions, the hybrid PSO with hill-climbing are more stable in the first three samples, but also DE and LPSO in 1^{st} sample, as compared to PSO and DEPSO. However, as the problem size increases to $1^{st}, 2^{nd}, 3^{rd}$, the hybrid PSO with hill-climbing performed worse and PSO and others improved. With regard to convergence time, the stability usually decreased, that is with uniformity for the PSO, DE, HCPSO and SHPSO, however, for the rest it is not uniform.

6.2.2. Analysis of Experiment 2

Table 12 shows results of executing experiment 2, which shows improvements of the computation time by applying the approximation algorithm in stead of the exact approach. In the case of the approximation, the delays are exaustively calculated in the presence of replication. However, the quality of the solutions are degraded as expected due to the approximation. Specifically, the result shown $61\% - 81\%$ computation time improvement over the exac method while facing quality degradation only for samples $g_{30}d_2$ and $g_{30}d_2$. The improvements are in seconds, which implies for a single usage (or run) of the meta-heuristic

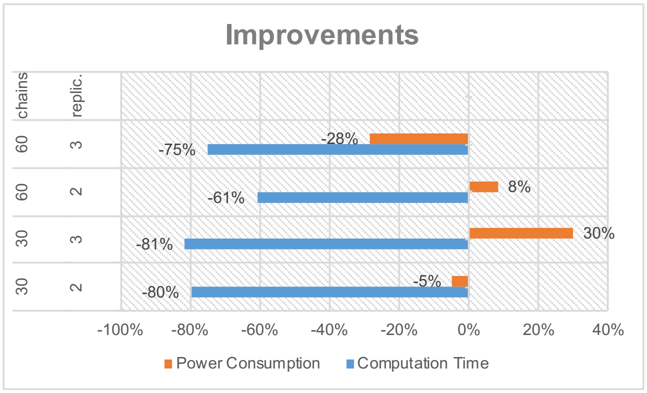


Figure 12: Effect of Approximate Algorithm over Delay Calculations with Replication.

optimization algorithms, it is not significant. However, considering practical systems design process, which requires several iterations, the commulative effect of the algorithms can negatively impact the responsivess to engineers. Thus, the improvements can be in trade-off with optimality of the solutions.

7. Discussion

8. Related Work

In a heterogeneous distributed system where computing nodes and communications links could have various failure rates, a reliability-aware allocation of tasks to nodes, and using links with the lowest failure rates can noticeably improve the system reliability [44][45][35][46]. Interleaving real-time constraints into the problem adds more complexity to reliability-aware task allocation in distributed systems [47]. As opposed to [9][5], we assume that software applications are multirate, which increase the difficulty of software allocation due the complexity of their timing analysis, and increased search space as the result of increasing timed paths of cause-effect chains. Furthermore, we assume a fault-tolerant system model.

Although improving reliability of the system using a reliability-aware task allocation does not impose extra hardware/software cost, in reliability-based design approach, redundancy (or replication) of software or hardware components is frequently applied to improve reliability. In such systems not only optimal allocation of software components (or replicas) should be taken into account but also the cardinality of the replicas should be limited for improved efficiency while meeting the desired reliability requirement. The integration of these two approaches (i.e., reliability-aware task allocation and application redundancy) is a promising technique to deal with high criticality of the system to fulfill the required reliability. For example, [48] proposes a heuristic algorithm to maximize reliability of a distributed system using task replication while at the same time minimizing the makespan of the given task set. Furthermore, in systems with replication, it uses the Minimal Cut Sets method, which is an approximate algorithm, to calculate reliability of a system. In contrast, we apply an exact method based on state enumeration, which is applicable to the problem size assumed in this work.

In our problem, power consumption is the other criterion of the optimization problem. Several research work exist on improving power consumption in real-time distributed systems. The research work [49] shows a survey of different methods on energy-aware scheduling of real-time systems, which categorizes the study into two major groups: i) Dynamic Voltage Scaling (DVS) [50][51], and ii) task consolidation to minimize the number of used computing and communication units [52], which is the approach followed in our work.

In the context of automotive systems, there are few works considering the reliability of a distributed system subject to real-time requirements of the automotive applications [53][54]. There are also other works discussing the allocation of software components onto nodes of a distributed real-time systems that consider other types of constraints other than reliability, for example, i) [55] which considers computation, communication and memory resources, and ii) [15] which proposes a genetic algorithm for a multi-criteria allocation of software components onto heterogeneous nodes including CPUs, GPUs, and FPGAs. Our approach also considers a hetrogeneous platform, i.e., nodes with different power consumption, failure-rate, and processor speed. In this work, we consider only the processor time; however, it can easily be extended to take into account different types of memory consumption that the software applications require.

9. Conclusions and Future Work

Software to hardware allocation plays an important role in the development of distributed and safety-critical embedded systems. Effective software allocation ensures that high-level software requirements such as timing and reliability are satisfied, and design and hardware constraints are met after allocation. In fault-tolerant multirate systems, finding an optimal allocation of a distributed software application is challenging, mainly due to the complexity of cause-effect chains' timing analysis, as well as the calculation of software application reliability. The timing analysis is complex due to oversampling and undersampling effects, caused by the different sampling rates, and the complexity of the reliability calculation is caused by the interdependency of the computation nodes due to replicas. Consequently, the formulation of the problem, to find an optimal solution, becomes non trivial.

In this work, we propose an ILP model of the software allocation problem for fault-tolerant multirate systems. The objective function of the optimization problem is minimization of power consumption with the aim of satisfying timing and reliability requirements, and meeting design and hardware constraints. The optimization problem involves linearization of the reliability model with piecewise functions, formulating the timing model using logical constraints, and limiting the number of replicas that can be used in the allocation. Furthermore, the allocation consider two cases of timing analysis: response time analysis and utilization bound.

Our approach is evaluated on synthetic automotive applications that are developed using the AUTOSAR standard, based on a real-world automotive benchmark. Although we consider automotive applications for the evaluation, the proposed approach is equally applicable to resource-constrained embedded systems, especially with timing, power and reliability requirements, in any other domain that are developed using the principles of model-based development and component-based software development. Our approach effectively applies to medium-sized automotive applications, but does not scale for complex applications. Considering similar system models, we plan to extend the current work with heuristic methods, e.g., genetic algorithms, simulated annealing, particle swarm optimization, etc., to handle large systems.

Acknowledgement

This work is supported by the Swedish Governmental Agency for Innovation Systems (Vinnova) through the VeriSpec project, and the Swedish Knowledge Foundation (KKS) through the projects HERO and DPAC.

References

[1] W. Wolf, A Decade of Hardware/ Software Codesign, Computer 36 (4) (2003) 38–43. doi:10.1109/MC.2003.1193227.

[2] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, M. Sgroi, Benefits and Challenges for Platform-based Design, in: Proceedings of the 41st annual conference on Design automation - DAC ’04, ACM Press, New York, USA, 2004, p. 409. doi:10.1145/996566.996684.

[3] B. Kienhuis, E. F. Deprettere, P. van der Wolf, K. Vissers, A Methodology to Design Programmable Embedded Systems, in: Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation — SAMOS, Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 18–37. doi:10.1007/3-540-45874-3_2.

[4] D. Fernández-Baca, Allocating Modules to Processors in a Distributed System, IEEE Transactions on Software Engineering 15 (11) (1989) 1427–1436. doi:10.1109/32.41334.

[5] S. E. Saidi, S. Cotard, K. Chaaban, K. Marteil, An ILP Approach for Mapping AUTOSAR Runnables on Multi-core Architectures, in: Proceedings of the 2015 Workshop on Rapid Simulation and Performance Evaluation Methods and Tools - RAPIDO ’15, ACM Press, New York, USA, 2015, pp. 1–8. doi:10.1145/2693433.2693439.

[6] H. R. faragardi, B. Lisper, K. Sandström, T. Nolte, A Resource Efficient Framework to Run Automotive Embedded Software on Multi-core ECUs, Journal of Systems and Software 139 (2018) 64–83. doi:10.1016/j.jss.2018.01.040.

[7] A. Bucaioni, L. Addazi, A. Cicchetti, F. Ciccozzi, R. Eramo, S. Mubeen, M. Sjodin, MoVES: A Model-driven Methodology for Vehicular Embedded Systems, IEEE Access 6 (2018) 6424–6445. doi:10.1109/ACCESS.2018.2789400.

[8] H. Bradley, Applied Mathematical Programming, Addison-Wesley, 1977. doi:http://agecon2.tamu.edu/people/faculty/mccarl-bruce/books.htm.

[9] E. Wozniak, A. Mehiaoui, C. Mraidha, S. Tucci-Piergiovanni, S. Gerard, An Optimization Approach for the Synthesis of AUTOSAR Architectures, in: IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, 2013. doi:10.1109/ETFA.2013.6647952.

[10] J. Fernandez, C. Galindo, I. García, System Engineering and Automation An Interactive Educational Approach, 2014. doi:10.1007/s13398-014-0173-7.2.

[11] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, R. Zainlinger, Distributed Fault-tolerant Real-Time Systems: the Mars Approach, IEEE Micro 9 (1) (1989) 25–40. doi:10.1109/40.16792.

[12] L. Vinet, A. Zhedanov, A "Missing" Family of Classical Orthogonal Polynomials, Computers as Components (2010) 528doi:10.1088/1751-8113/44/8/085201.
URL `http://arxiv.org/abs/1011.1669http://dx.doi.org/10.1088/1751-8113/44/8/085201`

[13] S. Mubeen, J. Mäki-Turja, M. Sjödin, Support for End-to-end Response-time and Delay Analysis in the Industrial Tool Suite: Issues, Experiences and A Case Study, Computer Science and Information Systems 10 (1) (2013) 453–482.

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

[15] I. Švogor, I. Crnkovic, N. Vrcek, An Extended Model for Multi-Criteria Software Component Allocation on a Heterogeneous Embedded Platform, Journal of computing and information technology 21 (4) (2014) 211–222.

[16] N. Naumann, AUTOSAR Runtime Environment and Virtual Function Bus, Hasso-Plattner-Institut, Tech. Rep.

[17] AUTOSAR, Specification of Timing Extensions, Tech. rep., AUTOSAR (2017).
URL `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_TPS_TimingExtensions.pdf`

[18] I. ISO, 26262: Road vehicles-Functional safety, Tech. rep., ISO/TC 22/SC 32 Electrical and electronic components and general system aspects (2011).

[19] S. d. C. Kung-Kiu Lau, What are Software Components?, World Scientific Publishing Company (June 29, 2017), 2017. doi:10.1142/9789813221888_0002.

[20] I. Crnkovic, M. Larsson, I. Ebrary, Building Reliable Component-based Software Systems (2002).

[21] S. Vestal, Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance, in: Proceedings - Real-Time Systems Symposium, 2007. doi:10.1109/RTSS.2007.47.

[22] L. Abeni, G. Buttazzo, Integrating multimedia applications in hard real-time systems, in: Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279), IEEE Comput. Soc, pp. 4–13. doi:10.1109/REAL.1998.739726.
URL `http://ieeexplore.ieee.org/document/739726/`

[23] M. Ashjaei, N. Khalilzad, S. Mubeen, M. Behnam, I. Sander, L. Almeida, T. Nolte, Designing end-to-end resource reservations in predictable distributed embedded systems, Real-Time Systemsdoi:10.1007/s11241-017-9283-6.

[24] R. Inam, N. Mahmud, M. Behnam, T. Nolte, M. Sjödin, The Multi-Resource Server for predictable execution on multi-core platforms, in: Real-Time Technology and Applications - Proceedings, Vol. 2014-Octob, 2014. doi:10.1109/RTAS.2014.6925986.

[25] S. K. Baruah, A. Burns, R. I. Davis, Response-time analysis for mixed criticality systems, in: Proceedings - Real-Time Systems Symposium, 2011. doi:10.1109/RTSS.2011.12.

[26] M. Becker, D. Dasari, S. Mubeen, M. Behnam, T. Nolte, End-to-end timing analysis of cause-effect chains in automotive embedded systems, Journal of Systems Architecturedoi:10.1016/j.sysarc.2017.09.004.

[27] N. Feiertag, K. Richter, J. Nordlander, J. Jonsson, A Compositional Framework for End-to-end Path Delay Calculation of Automotive Systems under Different Path Semantics, in: IEEE Real-Time Systems Symposium: 30/11/2009-03/12/2009, IEEE Communications Society, 2009.

[28] AUTOSAR, Specification of RTE Software, Tech. rep., AUTOSAR (2017). URL `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_RTE.pdf`

[29] A. Goel, Software Reliability Models: Assumptions, Limitations, and Applicability, IEEE Transactions on Software Engineering SE-11 (12) (1985) 1411–1423. doi:10.1109/TSE.1985.232177.

[30] E. Dubrova, Fault-Tolerant Design, Springer New York, New York, NY, 2013. doi:10.1007/978-1-4614-2113-9.

[31] X. Fan, W.-D. Weber, L. A. Barroso, Power Provisioning for a Warehouse-sized Computer, ACM SIGARCH Computer Architecture News 35 (2) (2007) 13. doi:10.1145/1273440.1250665.

[32] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, R. B. Brown, MiBench: A Free, Commercially Representative Embedded Bbenchmark Suite, in: 2001 IEEE International Workshop on Workload Characterization, WWC 2001, 2001, pp. 3–14. doi:10.1109/WWC.2001.990739.

[33] EMBC, AutoBench™ 2.0 - Performance Suite for Multicore Automotive Processors (2018).

[34] C. Lucet, J.-F. Manouvrier, Exact Methods to Compute Network Reliability, in: Statistical and Probabilistic Models in Reliability, Birkhäuser Boston, Boston, MA, 1999, pp. 279–294. doi:10.1007/978-1-4612-1782-4_20.

[35] P.-Y. Yin, S.-S. Yu, P.-P. Wang, Y.-T. Wang, Task Allocation for Maximizing Reliability of a Distributed System using Hybrid Particle Swarm Optimization, Journal of Systems and Software 80 (5) (2007) 724–735.

[36] J. Kennedy, R. Eberhart, C. A. C. Coello, G. T. Pulido, M. S. Lechuga, J. Kennedy, R. Eberhart, C. A. C. Coello, G. T. Pulido, M. S. Lechuga, F. Scholarpedia, Particle swarm optimization, Neural Networks, 1995. Proceedings., IEEE International Conference ondoi:10.1109/ICNN.1995.488968.

[37] R. Poli, An Analysis of Publications on Particle Swarm Optimisation Applications, Journal of Artificial Evolution and Applicationsdoi:10.1155/2008/685175.

[38] Q. Liu, W. Wei, H. Yuan, Z. H. Zhan, Y. Li, Topology selection for particle swarm optimization, Information Sciencesdoi:10.1016/j.ins.2016.04.050.

[39] J. Kennedy, R. Eberhart, A discrete binary version of the particle swarm algorithm, in: 1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation, Vol. 5, IEEE, pp.

4104–4108. doi:10.1109/ICSMC.1997.637339.
URL `http://ieeexplore.ieee.org/document/637339/`

[40] M. Clerc, Discrete particle swarm optimization, illustrated by the traveling salesman problem, in: New optimization techniques in engineering, 2000. doi:10.1007/978-3-540-39930-8_8.

[41] R. Storn, K. Price, Differential Evolution – A Simple and Efficient Heuristic for global Optimization over Continuous Spaces, Journal of Global Optimization 11 (4) (1997) 341–359. doi:10.1023/A:1008202821328.
URL `http://link.springer.com/10.1023/A:1008202821328`

[42] S. Das, S. S. Mullick, P. Suganthan, Recent advances in differential evolution – An updated survey, Swarm and Evolutionary Computation 27 (2016) 1–30. doi:10.1016/J.SWEVO.2016.01.004.
URL `https://www.sciencedirect.com/science/article/pii/S2210650216000146`

[43] N. Mahmud, G. Rodriguez-Navas, H. R. Faragardi, S. Mubeen, C. Seceleanu, Power-aware Allocation of Fault-tolerant Multi-rate AUTOSAR Applications, in: 25th Asia-Pacific Software Engineering Conference, 2018.
URL `http://www.es.mdh.se/publications/5222-`

[44] S. M. Shatz, J.-P. Wang, M. Goto, Task Allocation for Maximizing Reliability of Distributed Computer Systems, IEEE Transactions on Computers 41 (9) (1992) 1156–1168.

[45] S. Kartik, C. S. R. Murthy, Task Allocation Algorithms for Maximizing Reliability of Distributed Computing Systems, IEEE Transactions on computers 46 (6) (1997) 719–724.

[46] L. Zhang, K. Li, Y. Xu, J. Mei, F. Zhang, K. Li, Maximizing Reliability with Energy Conservation for Parallel Task Scheduling in a Heterogeneous Cluster, Information Sciences 319 (2015) 113–131.

[47] H. R. Faragardi, R. Shojaee, M. A. Keshtkar, H. Tabani, Optimal Task Allocation for Maximizing Reliability in Distributed Real-time Systems, in: Computer and Information Science (ICIS), 2013 IEEE/ACIS 12th International Conference On, IEEE, 2013, pp. 513–519.

[48] I. Assayad, A. Girault, H. Kalla, A Bi-criteria Scheduling Heuristic for Distributed Embedded Systems under Reliability and Real-time Constraints, in: Dependable Systems and Networks, 2004 International Conference on, IEEE, 2004, pp. 347–356.

[49] M. Bambagini, M. Marinoni, H. Aydin, G. Buttazzo, Energy-aware Scheduling for Real-time Systems: A survey, ACM Transactions on Embedded Computing Systems (TECS) 15 (1) (2016) 7.

[50] V. Devadas, H. Aydin, On the Interplay of Voltage/Frequency Scaling and Device Power Management for Frame-based Real-time Embedded Applications, IEEE Transactions on Computers 61 (1) (2012) 31–44.

[51] X. Wang, I. Khemaissia, M. Khalgui, Z. Li, O. Mosbahi, M. Zhou, Dynamic Low-power Reconfiguration of Real-time Systems with Periodic and Probabilistic Tasks, IEEE Transactions on Automation Science and Engineering 12 (1) (2015) 258–271.

[52] H. R. Faragardi, A. Rajabi, R. Shojaee, T. Nolte, Towards Energy-aware Resource Scheduling to Maximize Reliability in Cloud Computing Systems, in: High Performance Computing and Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), 2013 IEEE 10th International Conference on, IEEE, 2013, pp. 1469–1479.

[53] S. Islam, R. Lindstrom, N. Suri, Dependability Driven Integration of Mixed Criticality SW Components, in: Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on, IEEE, 2006, pp. 11–pp.

[54] J. Kim, G. Bhatia, R. R. Rajkumar, M. Jochim, An Autosar-compliant Automotive Platform for Mmeeting Reliability and Timing Constraints, Tech. rep., SAE Technical Paper (2011).

[55] S. Wang, J. R. Merrick, K. G. Shin, Component Allocation with Multiple Resource Constraints for Large Embedded Real-time Software Design, in: IEEE 10th Real-Time and Embedded Technology and Applications Symposium, 2004., IEEE, 2004, pp. 219–226.