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 prohibitively expensive.

In this paper, we propose a hybrid particle-swarm optimization technique for integrated software allocation on heterogeneous 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 reliability 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 synthesized from a real-world automotive benchmark. The evaluation makes comparative analysis of integer-linear programming method and various hybrid algorithms on metrics such as optimality, efficiency and scalability.

1. Introduction

The automotive electrical/electronic system executes several safety-critical software functions (or software applications), e.g.,throttle control, brake-by-wire control, traction control, etc. Moreover, it is a distributed architecture, thus executes some applications on multiple electroniccomputing units (ECU). The automotive functionality is getting complex, e.g., modern cars support hundreds of software applications and executes millions of lines of codes, therefore, efficient partitioning of the distributed software functionality is crucial to ensure software extensibility, that is to support current and future software growth. In this regard, the main concern in embedded system design including in the automotive

design includes the optimization of power and energy, which has been researched at different levels such as electronic circuit design, dynamic power and energy management, software/hardware partitioning, software applications allocation, etc. In this paper, we propose power-efficient allocation of distributed software applications on heterogeneous computing units.

In distributed computing, the risk of software functionality failure is greater due to higher transient and permanent faults, thus maximizing reliability of the distributed system is desirable. In the safety-critical design, the software applications are required to meet reliability goals in order to assure correct operation of the software over some period of time. The most common way to maximize reliability is by applying *fault tolerance*, that is via redundant software and hardware components. However, fault tolerance requires additional computation resources, and consumes more power and energy. Therefore, the software allocation should consider meeting the reliability goals besides optimizing the power consumption of distributed software applications, since different software allocation satisfying the reliability goals could deliver different power consumption.

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 [3] approaches. 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., guarantee optimal solutions, nevertheless, they do not scale to large-scale problems [5]. Moreover, applying exact methods on-non linear problems, which are prevalent in practice, is prohibitively expensive. Our previous work on solving the software allocation problem [6], we demonstrate the limitation of integer-linear programming (ILP) [7] using exact method by the CPLEX solver. Similarly, 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 guarantee of optimality [8, 9]. 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" [10].

Metaheuristics has found wide applications in many domains, e.g., cellular networks, cloud computing, software design, etc [11]. Many of the existing metaheuristic algorithms are nature inspired, e.g., genetic algorithm, evolutionary algorithms, simulated annehealing, ant colony, particle-swarm optimization, etc. Applications of metaheuristics on the software allocation of real-time systems are in the early stages, nevertheless, there exist some work, e.g., by Qin-Ma et al. [12] on maximizing reliability of distributed computing systems using hanybee algorithm, maximizing reliability of distributed systems using hill-climbing particle-swarm optimization by Yin et al. [13], 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. Sequences of runnables deployed on the same unit or network 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* [14]. The propagation of signals over multirate chains result in undersampling/oversampling effects, which makes end-to-end timing analysis difficult [15]. In order to maximize software applications reliability and meet their reliability goals, we implement fault tolerance.

The contributions of our work are summarized 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. [16]. 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

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

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. 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 [17].

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 [17].

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 [18] 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

Figure 1 illustrates the system model in consideration which consists of AUTOSAR software applications $A = \{a_i : i = 1, \ldots, n_A\}$ partitioned on an

execution platform $\langle \mathcal{N}, \mathcal{B} \rangle$, where $\mathcal{N} = \{n_h : h = 1, \ldots, n_{\mathcal{N}}\}$ are the computing unit, $\mathcal{B}$the shared network bus. The applications have user-defined requirements (RL,EE,CL), where the tuple elements respectively refer to the reliability goals (or requirement), end-to-end timing requirements, and criticality levels. The execution platform has provisions (or capabilities) (HZ,PW,FR), where the tuple elements respectively refer to the processor speed and the power-consumption specifications of the computing units, and the failure rates of the computing units and the network bus.

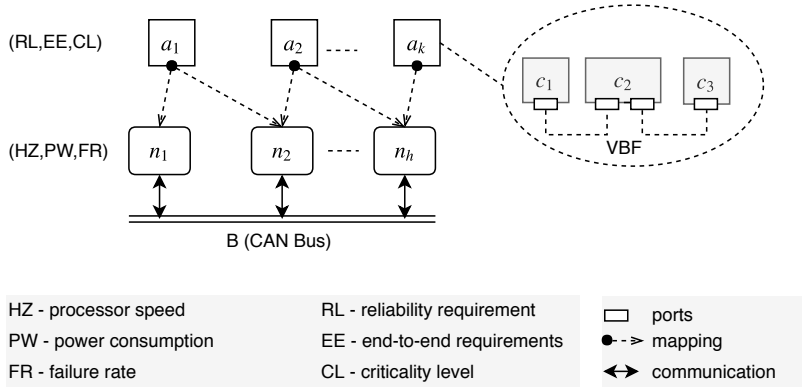


Figure 1: System Model.

Notations. For easy reading, we introduce the main notations used throught the paper as shown in Table 1.

3.1. Software Applications

A software application represents an independent and self-contained user-defined software functionality, e.g., x-by-wire, electronic throttle control, flight control, etc. In AUTOSAR, software applications are developed using AUTOSAR *Software Component* (SW-C), which is design-time concept that represents the lowest-level hierarchical element in software architecture of the software application, hence is atomic, and therefore is mapped to a single computing unit. It is implented by the AUTOSAR runnables, which are schedulable pieces of codes (or objects), and contains specifications of timing behavior and computation resources needed by the runnables. We formally represent the AUTOSAR software application as follows:

Definition 1 (AUTOSAR Software Application). [2] We represent it as a *directed acyclic vertex-weighted* graph $\langle V, L, w \rangle$ of periodic runnables, where V denotes runnable nodes, $a_{ij} \in L$ a data-dependency link from r_i to r_j, where

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

Notation	Description
• Related to software applications	
$A = \{a_i : i = 1, \ldots, n_A\}$	AUTOSAR software applications*
$a_k = \langle V, E, w \rangle$	a software application, modeled as directed acyclic graph of runnables
$\mathcal{V}(a_k) \in R_i, \mathcal{E}(a_k)$	nodes and links of a_k, respectively
$\Gamma = \{\Gamma_i : i = 1, \ldots, n_\Gamma\}$	end-to-end chains of a_k
Γ_i	a path in $\mathcal{V}(a)$, denotes a chain
• Related to software components	
$C = \{c_i : i = 1, \ldots, n_C\}$	software-component types used in a_k
$Q_i = \{q_{i,j} : j = 1, \ldots, n_{Q_i}\}$	component replicas of type c_i
$R_i = \{r_{i,j} : j = 1, \ldots, n_{R_i}\}$	runnables of c_i
• Related to the execution platform	
$N = \{n_i : i = 1, \ldots, n_N\}$	computation (or computing) nodes
B	the shared network bus
• Related to the mapping	
$\mathbf{x} = \{\mathbf{x}^{a_k} : i = 1, \ldots, n_\mathbf{x}\}$	a mapping vector from Q^{A_k} to N
x^k_{ij}	the mapping of c_i to n_l, where $l = x^k_{ij}$
b_k	a software application, modeled as directed acyclic graph of tasks, refines a_k
$T_i = \{\tau_{i,j} : j = 1, \ldots, n_{T_i}\}$	tasks mapped to c_i
• Related to the optimization	
$Power(\mathbf{x})$	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 $\tau \in V(g_\tau)(\mathbf{x})$
$Delay_\gamma(\mathbf{x})$	age delay of $\gamma \in \Gamma^{A_k}$ in $\mathbf{x}$

Table 1: The Main Notations Used Throughtout the Paper.
* 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 A, essentially it refers to its cardinality of the set.

$i \neq j$. The assignment function $w : V \rightarrow (\mathrm{E}_i, \mathrm{D}, \mathrm{P}, \mathrm{N})$ sets each runnable nodes the computation cost such as worst-case execution times (WCET) $\mathrm{E} = \{\mathrm{E}_h : h = 1, ..., n_N\}$, deadline D and period P, where $\mathrm{E}_h \in \mathrm{E}$ is the WCET of r on the computing unit $n_h \in N$.

A path in the graph $\Gamma_l = r_i \rightarrow, ..., \rightarrow, r_j$ represents a *cause-effect* chain, where $r_i, r_j \in \mathcal{V}(a_k)$ are source and sink of the chain, e.g., consider the example from Figure the appication a_1 in Figure 2, $r_1 \rightarrow r_3 \rightarrow r_5$ is the chain, and r_1, r_5, respectively are the source and the sink of the chain. A chain is a subfunctionality of the application that is triggered by a stimulus (or stimuli), e.g., pressing a brake pedal, and the corresponding response, e.g., vehicle braking to the desired speed level. It usually has a user-defined timing requirement, known as *end-to-end* timing requirement (EE), which puts an upper-bound on the duration between the stimulus and the corresponding response of executing

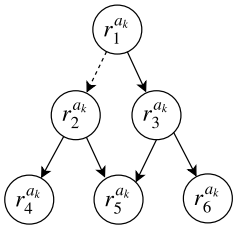


Figure 2: Software Application Example, Modeled as DAG of Runnables.

c_i	r_j	$(\mathrm{E}_h, \mathrm{P} = \mathrm{D})$	$\prec r_i$
1	1	$(0.1, 10)$	2
2	2	$(0.1, 20)$	
2	3	$(0.1, 15)$	
2	4	$(0.1, 8)$	
3	5	$(0.1, 12)$	
3	6	$(0.1, 6)$	

Table 2: Timing Specifications of the Runnables from Figure 2

the chain. The set of chains are represented as $\Gamma = \{\Gamma_i : i = 1, \ldots, n_\Gamma\}$. Note: each runnable is subscribes to at least one chain.

3.2. Scheduling Software Applications

We assume software applications share the execution platform such as the computing units and the on-board network bus, following the mixed-criticality design [19]. Thus, software applications with different software criticality should be isolated in order to prevent interference of lower-criticality applications on higher-critical applications. Assume the braking functionality of a vehicle, which is safety-criticality, is distributed over multiple computing units. Another application with infotainment functionality also shares the units. The mixed-criticality design to this problem ensures both applications are schedulable during absence of errors, however, the braking application must also be schedulable in cases of overrun due to errors, e.g., by degrading or halting the infotainment application. There are several techniques in the literature that deal with the scheduling of mixed-criticality applications on *uniprocessor* systems [19]. In this work, we consider the *partitioned criticality (PA)* (or criticality-as-priority assignment, CAPA) technique to schedule the mixed-criticality applications, which prioritizes applications based on criticality, rather than *deadline* as used by the deadline monotoic assignment [20]. In contrast to other techniques, CAPA is easy to implement and does not require a runtime monitoring, e.g., using servers [21, 22, 23], though not efficient[3]. Thus, the software applications are schedulable according to CAPA if the runnables, messages, and chains in the of the applications are schedulable, that is, they meet their corresponding deadlines.

According to AUTOSAR [24], runnables are high-level schedulable objects that are mapped to tasks, and subsequently are scheduled by the AUTOSAR operating system [25]. Next, we explain the mapping rules applied in this work.

[3]Note: scheduling techniques other than the PA technique can be used with our approach to schedule the applications.

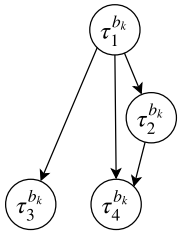


Figure 3: A Software Application Modeled as Directed Acyclic Graph.

τ_i	$\bigcup r_j$	$(\mathrm{E}_h, \mathrm{P} = \mathrm{D})$
1	1,2	$(0.2, 10)$
2	3	$(0.1, 15)$
3	4	$(0.1, 8)$
4	5,6	$(0.1, 6)$

Figure 4: Tasks Timing Specifications after Merging.

3.2.1. Runnables-to-Tasks Mapping/Transformation

In the mapping process, one or more runnables can be merged to optimize the runtime execution by reducing the number of tasks scheduled by the OS. In this work, we merge the runnables $a, b \in \mathcal{V}(a_k)$ into the task $v \in \mathcal{V}(b_k)$ if the following two conditions are satisfied: (i) $(a, b) \in \mathcal{E}(a_k)$ is a link in the graph, (ii) the period of a is a factor of b, or vice versa , otherwise, each runnable is mapped to a separate task, inheriting the timing specifications of the runnable. If the merging conditions are met, the timing specifications of v are set as follows: (i) the WCET of the task is set to the sum of the WCET of the runnables, i.e., $\mathrm{E}_h^v = \mathrm{E}_h^a + \mathrm{E}_h^b$, for all $h : 1, ..., n_N$, (ii) the period and deadline of the task is set to the minimum of the runnables' periods, $\mathrm{P}_v = \mathrm{D}_v = min(\mathrm{P}_a, \mathrm{P}_b)$

The AUTOSAR software applications is shedulable if and only if its task graph is scheduable, that is each task node meets its deadline, each communication between task nodes (that is, the message) meets its deadline, and the chain meets end-to-end requirement. The schedulability analysis of the tasks, messages and chains used in this work are explained next.

3.2.2. Scheduling of Tasks and Messages

Following the CAPA scheduling technique, the tasks in the distributed system are assigned priorities according to their criticality, thus the higher the application's criticaity, the higher the priority its tasks acquire

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

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; $\mathcal{V}(b_i), \mathcal{V}(b_j)$ returns the tasks of b_i and a_j, respectively.

The tasks are scheduled using the *fixed-priority preemptive scheduling policy* (FPPS) [26], and the schedulability analysis, that is the necessary and sufficiency testing for schedulability, is conducted via the classical response-time analysis (RTA) as shown by Equation (1) [20, 20]. According to the analysis, if the response time of a task is less than or equal to its deadline, that tasks a $\delta_\tau \leq Deadline_\tau$, the task is schedulable, otherwise 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 priority 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 follows the CAPA technique to achieve the mixed-criticality requirement. This can easily be achived by inheriting the priority of each sender task communicating over the bus $pri(m)$ to the successor message $suc(\tau)$, that is $pri(m) = pri(\tau)|\tau = suc(m)$. The schdulability of messages is checked using the classical response-time analysis of the CAN network using Equation (2), as presented by Rob Davis et. al [27]. The worst-case response time of a message is computed as the sum of its *jitter* time (that is, the time taken by the sender task to queue for transmittion) J_m, the *interference* 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 and in the transmission.

3.2.3. Scheduling Cause-effect Chains

The chain consists of indpendently clocked tasks, which results in undersampling/oversampling effects. As a result, a single propagating across the chain is characterized by having various delays, which are discussed in detail by Feiertag et al. [28] in the context of single-register buffer communication, which is a common practice in control systems design, e.g., automotive software applications [29]. In this work, we demonstrate the two widely used semantics in the automotive domain: *age* delay and *reaction* dealy, and consider only the age delay in the analysis. The age delay is the time elapsed between the data arriving a the input register (which is the stimulus) and its corresponding latest, non-overwritten output at the output register OR (which is the response). 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.

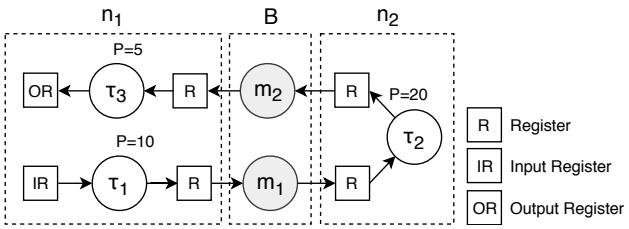


Figure 5: A Cause-effect Chain, mapped on nodes n_1 and n_2.

Consider the chain $\tau_1 \rightarrow \tau_2 \rightarrow \tau_4$ of the task graph from Figure 2, and assume it is mapped on the computing units n_1 and n_2 as illustrated in Figure 5, and communicate using the single-register buffers. The tasks τ_1 and τ_2 execute on node n_1, whereas task τ_3 executes on node n_2. The input data can arrive at any instance in the input register IR.

The execution behavior of the chain is illustrated in Figure 6 over two hyper-periods. Note: the message between τ_2 and τ_3 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 the 3^{rd} instance of τ_1 and the 10^{th} instance of τ_3. It is frequently used in the control systems where freshness of data is paramount, e.g., braking a car over a bounded time. 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. [15].

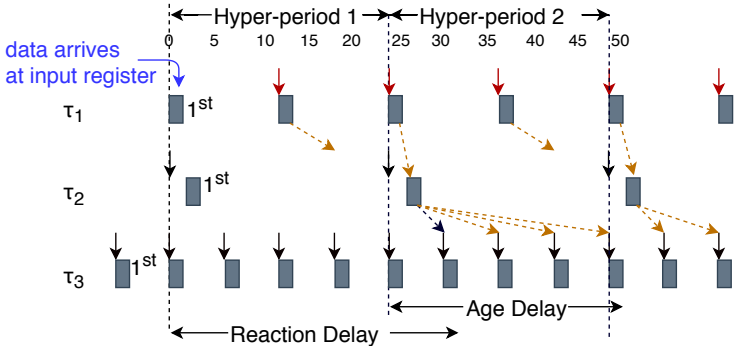


Figure 6: Reaction and Age Delays of the Cause-effect Chain, Shown in Figure **??**.

Thus, if the chain is mapped to a single computing unit, the age delay Δ^{sub} is computed using Equation (5), that is by taking the difference between the activations of the sink and the source taks, plus the response-time of the sink task. Otherwise, if it is mapped to multiple units and the bus, the delay Δ is computed compositionally by identifying the subchains using Equation (6). In the latter case, the response-time of messages δ^{msg}, and the "just missed" case that is by adding the period of the successor task $P_{suc(j)}$ are taken into consideration.

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

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

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

3.3. Reliability of Software Applications

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 $[0, t]$ [30]. Redundancy is the most common way for implementing fault tolerance and increasing the reliability of a system. Redundancy can be implemented according to different schemes, such as hot stand-by, cold stand-by, etc. [31], which differ on the number of replicas that are active as well as the methods for detection and compensation of faulty replicas. In our system model, we consider the hot-standby scheme, where replicated components maintain the same state, but only one replica (the so-called *primary*) effectively acts on the environment, for instance issuing an input. The primary software component will be denoted as $q_{i,1}$, whereas the secondary software component, which is in the stand-by, by $q_{i,2}$, etc., for a software application A_k.

In this work the details of the redundancy scheme are abstracted away under the following assumptions: (i) Software does not contain design errors. This has two implications: first, that hardware elements, i.e. computing nodes and communication buses, are the only causes of failure and, second, that introduction of N-version programming is not required. Different replicas of the same software component execute exactly the same program. (ii) Hot stand-by redundancy (also known as Primary/backup) is used for detection and replacement of failed components. (iii) Software components need to be replicated only if the application's reliability requirement is not met without replication, otherwise they are not replicated. (iv) 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; (v) 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

Note that under these assumptions, the reliability of a software application is equivalent to the reliability of the platform on which it is deployed. The

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 7: Allocation of the Software Components.

reliability of a computing unit (and of the bus) can be easily calculated as $e^{\lambda t}$, where λ is an exponentially distributed failure-rate. However, calculating the reliability of the whole execution platform is not trivial for the case with replication. In particular, the traditional series-parallel reliability approach cannot be applied because of the *functional* interdependencies created between computing nodes as the result of replication and allocation. To illustrate the difficulty, let us assume a software application A_k, having component configurations with and without replication as shown Table 7b and 7a, respectively, where $q_{i,j}$ is the j^{th} software-component replica of software-component type $c_i \in C_i$. 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 8a, 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 software application, e.g., due to $q_{1,1}$ and $q_{2,2}$ or $q_{3,2}$, n_1 and n_3 make series, and due to $q_{3,1}$ and $q_{3,2}$, the nodes make parallel, to realize a partial functionality of the application. In this case, the series-parallel diagram depicted in Figure 8b does not accurately capture the reliability 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 overcome this problem, we will use an exact technique for reliability calculation based on the enumeration of the different failure states of the computing nodes; that is, the different failure states of the execution platform are enumerated exhaustively, and subsequently, the total probability the software application functions is computed. This technique will be discussed in great length in Subsection 4.5.

4. Software-to-Hardware Allocation Problem

The software applications are allocated on the execution platform by mapping the software-component replicas (or software components) $Q_i^{A_k}$ to the computing units $\mathcal{N}$. The software-to-hardware allocation problem is to find the mapping $\mathbf{x} : Q_i^{A_k} \rightarrow \mathcal{N}$ that satisfies the user-defined requirements of the software applications, but also minimize the total power consumption of the applications, where $\mathbf{x}$ is a possible mapping matrix, and x_{ij}^k represents the mapping of the software component $q_{i,j}^{A_k}$ to the compuing unit n_h, where $h = x_{ij}^k$, of the application A_k.

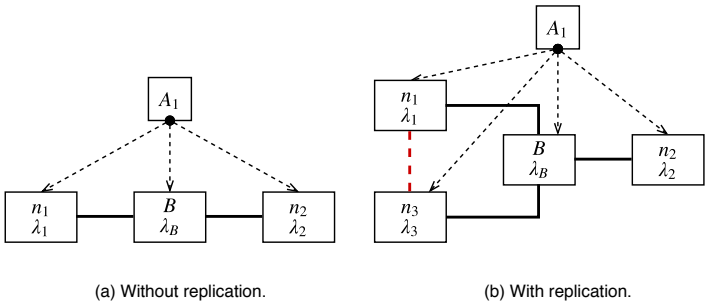


Figure 8: Reliability Block Diagrams of the Software Application.

4.1. Total Power Consumption

The total power consumption of the applications $\mathcal{P}_{total}(\mathbf{x})$ is computed as the sum of the power consumption of the computing units to which the applications are allocated. The power consumption of a unit is computed according to the linear model proposed by Fan et al. [32] as shown by Equation (7), which is directly proportional to its load (or utilization) and is inductively formulated from experimental results.

$$\mathcal{P}(u) = P_{idle} + (P_{busy} - P_{idle}) * u, \tag{7}$$

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 load. The parameters of the model can be obtained by running performance benchmark suits, e.g., MiBench [33], AutoBench [34], etc.

The utilization of a computing unit is computed as a sum of the utilization of the tasks mapped to it T_{n_h}, which are identified by traversing the mapping elements-wise x_{ij}^k as shown in Equation (8), where T_i refers to the tasks implementing the component type $c_i^{A_k}$.

$$T_{n_h} = \{T_i^{A_k} | \forall kij \ h = x_{ij}^k\} \text{ forall } h = 1, ..., n_N \tag{8}$$

Then, the utilization of the units, indicated by the vector $(u_1, ..., u_{n_N})$, is

$$(u_1, ..., u_{n_N})(\mathbf{x}) = \sum_{\tau \in T_{n_h}} \mathcal{U}(\tau, n_h), \text{ for all } h = 1, .., n_N,$$

where $\mathcal{U}(\tau, n) = WCET_{\tau,n}/P_\tau$ computes utilization of the task τ on the unit n. Thus, the total power consumption of the applications is formulated as

$$\mathcal{P}_{total}(\mathbf{x}) = \sum_{h=1}^{n_N} \mathcal{P}(u_h(\mathbf{x})) \text{ for all } h = 1, .., n_N, \tag{9}$$

4.2. Tasks and Messages Timing Constraints

The tasks timing constraints ensure that the tasks in the distributed system meet their respective deadlines, that is $\forall \tau \in T_{n_h}\ ResponseTime_\tau(\mathbf{x}) \leq Deadline_\tau$ for all $h = 1, ..., n_N$, following Equation (8). Similarly, the messages timing constraints ensure that the response-time of message in the CAN bus meet their respective deadlines.

The set of messages in the bus is determind by traversing the edges of the tasks graphs. If the edge relates tasks located on different units, a message is used to communicate across the bus, otherwise, no message is used. However, due to replication, the edge may represent a set of subedges, where each subedge relates tasks replicas on both sides of the edge. Consider $\mathcal{R}_\tau^{b_k} = \{(\tau, n) \in \tau \times N_\tau^{A_k}\}$ is the set of the replicas of type $\tau \in \mathcal{V}(b_k)$, now assume $(t1, t2) \in \mathcal{E}(b_k)$ is an edge in the task graph and $\mathcal{R}_{t2}^{b_k}, \mathcal{R}_{t2}^{b_k}$ are the replicas of type $t1$ and $t2$, respectively. The set of subedges between $t1$ and $t2$ $\mathcal{R}_{t2}^{b_k} \times \mathcal{R}_{t2}^{b_k}$, and the set of messages in these subedges are where the latter relates tasks replicas located on different nodes. By extension, the set of message in the bus iM determined as,

$$M = \{m_{i,j} | \forall (t1, t2) \in \mathcal{E}(b_k(\mathbf{x})) \forall (i, j) \in \mathcal{R}_{t1}^{b_k} \times \mathcal{R}_{t2}^{b_k}\ n_i \neq n_j\}, \tag{10}$$

where (i, j) is a sub-link of $(t1, t2)$, m_{ij} is the message that the replica i uses to communicate with the replica j, $n_i, n_j \in \mathcal{N}$.

We assume the messages inherit the timing and criticality specifications of the sending tasks, thus $P_{m_i} = P_{t1}$, $\mathrm{CL}_{m_i} = \mathrm{CL}_{t1}$.

4.3. End-to-end Timing Constraints

The end-to-end timing constraints over $\mathbf{x}$ ensure that the delays of chains meet thier respective end-to-end requirements, that is $\forall \gamma \in \Gamma^{A_k}\ Delay_\gamma(\mathbf{x}) \leq \mathrm{EE}_\gamma^{A_k}$ for all $k = 1, ..., n_A$. Note: The constrains assumes the tasks and messages constraints are satisfied.

The delay calculation of a chain Γ is multiplicity Γ^* due to replication. Consdier the chain $\Gamma = (\tau_1, ..., \tau_l)$, 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 thez sender task, and its receiver task t_{i+1} is 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,

$$\Delta_\gamma(\mathbf{x}) = \sum_{i \in I_\gamma(\mathbf{x})} \Delta_i^{sub}(\mathbf{x}) + \sum_{j \in J_\gamma(\mathbf{x})} \Big[\delta_j^{msg}(\mathbf{x}) + P_{suc(j)}\Big],$$

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

Table 3: Chains-with-Replication of Degree 2 for the Chain $\Gamma = \tau_1 \rightarrow \tau_2 \rightarrow \tau_4$, its Subchains I and Messages J.

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 (11).

$$\forall \gamma \in \Gamma^{A_k} \ \Delta_\gamma^{A_k}(\mathbf{x}) \leq \mathrm{EE}_\gamma^{A_k}, \tag{11}$$

Example 1 (Delay Calculation). *Consider the chain $\Gamma = \tau_1 \rightarrow \tau_2 \rightarrow \tau_4$ from Figure 3 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 7 (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 3 illustrates how to 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 3.2.3.*

4.4. 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 on the number of replicated chains. Moreover, the age-delay calculation is an exhaustive search as demonstrated in Subsection 3.2.3. For these main reasons, the age delay computation is sometimes prohobitively expensive considering the meta-heuristic algorithms, which compute large-space candidate solutions over thousands iterations. Therefore, we propose an approximation algorithm to efficiently compute the delays based on the worst-case age delay explained in the following lemma:

Lemma 1 (Worst-case Edge Delay). *The worst-case age delay of a chain mapped to a single computing unit follows the maximization of the age delay formula shown in Equation (). Thus, the source task reads from the buffer as early as possible and the sink task reads from predecessor task as late as possible. Moreover, the read operation in between the chain, we assume just miss and late exectuion betwore the deadline of the executing task, which means the worst-case age delay becomes the summation fo the .... which occurs between the earliest*

activation of the source task and the latest activation of the sink task plus the maximum worst-case response time of the sink task of a chain. The earliest activation occurs when the source task start time concedes its release time, and latest activation time of the sink task occurs not latter than the second hyper-period since the data is picked up by the source task.

The lemma provides the worst-age delay of a chain mapped to a single unit, however, the chains with the replication can mapped to over a network. So, the worst age delay considers the messages time and the time considered for the just miss from the receiving end of the task across the network. Therefore, the worst-case edge delay of the chain Γ is calculated as follows: first we identify the chain-with-replication $\gamma \in \Gamma^*$ that uses the network frequently, then the worst-case edge delay is the summation of the summation of the worst-case edge delay mapped to a single unit $2H$, the response times of the messages $\sum \delta^{msg}(\mathbf{x})$, and the "just miss" times $\sum P_{suc(j)}(\mathbf{x})$.

$$\Delta^{worst} = 2H + \sum_{j \in J_\gamma(\mathbf{x})} \Big[\delta_j^{msg}(\mathbf{x}) + P_{suc(j)}(\mathbf{x})\Big], \tag{12}$$

where where $H = lcm(\gamma)$ is the worst-case age delay over a single unit, Δ^{worst} is the worst-case age delay over multiple units, and $\exists\gamma \in \Gamma^* \ \max|J_\gamma|$ is the chain-with-replication that uses the network frequently.

4.5. 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 (13). Note: if application does not use the shared bus $Reliability_B = 1$. Equation (14) finds the units N^{A_k} that the application $a_k$uses by traversing the partition $\mathbf{x}$ in linear time.

$$Reliability_{a_k}(\mathbf{x}) = Reliability_{N^{A_k}}(\mathbf{x}) * Reliability_B \tag{13}$$

$$N^{A_k} = \{e \in N | \forall ij \ e = m_h\}, \text{ where } h = x_{ij}^{(k)} \tag{14}$$

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 [35], which is an exact technique to calculate reliability, as opposed to using series-parallel technique -

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 4: 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 (7), the Variable $s_{c_i} \in \{0, 1\}$ Indicates if the Replicas of Type c_i Fails or Functions, Respectively.

motivated in Subsection 3.3. 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 $S = \{s_1, ..., s_{n_N}\}$.

- $\xi = \Omega^S$ 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 (4) 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 $s_2 = 1$.

- $p : \xi \rightarrow [0, 1]$ 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.

- $f : \xi \rightarrow \{0, 1\}$ 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 (15). The component replica $q_i, j \in Q_i$ of type c_i fails if n_h fails, that is $s_h = 0$.

$$f_s(\mathbf{x}) = \begin{cases} 0 & \text{if } \exists i \ c_i | \forall j \ s_h = 0 \\ 1 & \text{otherwise} \end{cases} \qquad \text{where } h = x_{ij} \tag{15}$$

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 2 (Reliability Calculation). *Let us assume we want to calculate the reliability of the application in Table 4 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* $Reliability_N * Reliability_B = 0.99649339932$*.*

4.6. 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 16, with inequality constraints shown by Equation (17, 18,19). 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.

$$\min_{\mathbf{x} \in X} \ Power(\mathbf{x}) \qquad \text{subjected to:} \tag{16}$$

$$Reliability_{A_k}(\mathbf{x}) \leq RelReq_{A_k} \qquad \text{forall } k = 1, ..., n_{A_k} \tag{17}$$

$$\forall \tau \in T_{m_h} \ ResponseTime_\tau(\mathbf{x}) \leq Deadline_\tau \qquad \text{forall } h = 1, ..., n_M \tag{18}$$

$$\forall \gamma \in \Gamma^{A_k} \ Delay_\gamma(\mathbf{x}) \leq E2eReq_\gamma \qquad \text{forall } k = 1, ..., n_A \tag{19}$$

$$\tag{20}$$

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

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 *medium*, where the small problem refers to a software application with software components less than 10, chains less than 30, and he 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. [16]. However, the ILP approach, as also shown for similar problems, suffered from the scalability problem, that did not return soltuions for large-scale software allocation problems. In this section, we propose metaheuristics 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., that is, lthough 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 [13], 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 [36, 37], likewise, PSO is used with local search techniques such as Hill climbing to intensify the search [13]. 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 [38]. 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 [36]. 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 [39].

In PSO, the population (or swarm) $\mathcal{PN} = \{p_1, p_2, \ldots, p_N\}$ is a collection of particles, organized according to a certain population topology [40]. A particle has a position $\mathbf{p}$. and a velocity $\mathbf{v}$. PSO is memory-based, that is, it remembers the best position of a particle, $\mathbf{p}_{bst}$. Moreover, it remembers the best position of the swarm, $\mathbf{z}$. The particle moves to the global optima guided by its current velocity and its attraction vectors known as the *cognitive* and the *social* components as shown by Equation 21. The cognitive component $(\mathbf{p}_{bst} - \mathbf{p})$ attracts the particle towards its best position whereas the social component $(\mathbf{z} - \mathbf{p})$ attracts the particle towards the swarm best position. In fact, the next velocity of the particle is the resultant of the attraction components and the

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

Figure 9: Binary (0-1) Representation.

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

Figure 10: Integer Representation.

Figure 11: Solution Representations for Components $\{c_1, c_2, c_3\}$ Mapped to Computiong Nodes $\{n_1, n_2, n_3\}$ based on Table 7b.

current velocity. Thus, the next position of the particle is the resultant of its current position and its next velocity as shown by Equation (22).

$$\mathbf{v} \leftarrow \omega\mathbf{v} + c_1 Rand() \circ (\mathbf{p}_{bst} - \mathbf{p}) + c_2 Rand() \circ (\mathbf{z} - \mathbf{p}) \tag{21}$$

$$\mathbf{p} \leftarrow \mathbf{p} + \mathbf{v}, \tag{22}$$

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.

5.2. Solution Representation

The software allocation is a discrete problem, as such, the solutions are discrete values. The PSO was originally proposed for continuous problem, neverthless, it is applied to discrete problems successfully as well, e.g., to the sales man problem [41]. There are two commonly used solution representations of PSO for descrete problems: the binary (0-1) and integer representations, which are demonstrated in Figure 11 using the example provided in Figure 7. In the latter, the variable indicate the computing-unit identifier to which the component is allocated. The two representations are interchangeable.

In this work, we consider the integer representation due to efficient encoding (much fewer variables) as can be observed from Figure 23, and it is computationally more efficient considering our problem. Following the integer-representation approach, the solution is discretized by approximatings its constituents into the nearest integer values, that is $\mathbf{x} \leftarrow [\mathbf{x}]$, where $\mathbf{x}$= $\mathbf{p}$.

The solution to the allocation problem of the applications A is represented by a vector of $n_{C^{A_k}} \times D$ matrices $\mathbf{x} = \{\mathbf{x}^{a_k} : i = 1, \ldots, n_{\mathbf{x}}\}$, where $\mathbf{x}^{a_k}$ as shown by Equation (23) is the solution of a_k, and x^k_{ij}= $h \in \{1, \ldots, n_N\}$ denotes

the mapping of the software-component replica $q_{i,j}^k$ to the computing unit n_h.

$$\mathbf{x}^k = \begin{bmatrix} x_{11}^{(k)} & x_{12}^{(k)} & \dots & x_{1K}^{(k)} \\ x_{21}^{(k)} & x_{22}^{(k)} & \dots & x_{2K}^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N_c1}^{(k)} & x_{N_c2}^{(k)} & \dots & x_{N_cK}^{(k)} \end{bmatrix} \tag{23}$$

5.3. Fitness Function

The fitness function $f : \mathbf{x} \rightarrow \mathbb{R}$ is a type of objective function that summerizes the contributions 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 [42, 8]. 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 penalty 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 $Power(\mathbf{x})$ with the constrains, represented by a set of *penalty functions* $\{\phi_{reliability}(\mathbf{x}),\ \phi_{deadline}(\mathbf{x}),\ \phi_{e2e}(\mathbf{x})\}$. 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 and the end-to-end timing requirement, respectively. Indeed, the penalty function penalizes the candidate solution by increasing its fitness (for our minimization problem), thus discriminating the solution. The fitness function $f(x)$ is computed as follows.

$$\min_{\mathbf{x} \in X} \ f(\mathbf{x}) = Power(\mathbf{x}) + \beta_1 \phi_{rel}(\mathbf{x}) + \beta_2 \phi_{ddl}(\mathbf{x}) + \beta_3 \phi_{e2e}(\mathbf{x})), \tag{24}$$

where β_1 to β_3 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 values of the penalty coefficients are discussed in more details.

5.4. Penalty Function

$$\phi_{rel}(\mathbf{x}) = \sum_{k=1}^{n_A} \max\{0, Reliability_{a_k}(\mathbf{x}) - \mathrm{RL}_{a_k}\} \tag{25}$$

$$\phi_{ddl}(\mathbf{x}) = \sum_{\forall \tau \in T_{m_h}} \max\{0, ResponseTime_{\tau}(\mathbf{x}) - \mathrm{DL}_{\tau}\} \tag{26}$$

$$\phi_{e2e}(\mathbf{x}) = \sum_{\forall \gamma \in \Gamma^{A_k}} \max\{0, Delay_{\gamma}(\mathbf{x}) - \mathrm{EE}_{\gamma}\} \tag{27}$$

5.5. Penalty Coefficients

To calculate the values of the penalty coefficients β_1, β_2 and β_3, we use the similar analytical approach proposed in [43], where the value of each penalty coefficient is determined separately with respect to the relative proportion of the range of the penalty function to the range of the objective function, which is $Power(x)$ in our problem. Indeed, the penalty coefficients should be determined such that all the feasible solutions have a lower fitness value in comparison to the infeasible solutions, meaning that all the feasible solutions are always preferred to infeasible solutions [8]. On the other hand, the penalty coefficients 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 [42].

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

$$\mathcal{P}^{min} + \beta_1 \times \min\{PenaltyValue\} > \mathcal{P}^{max} + 0$$

Let us assume that (i) $Power^{min} = 0$, (ii) $\mathcal{P}^{max}$ 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,

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

Our experiments also verify this theoretical discussion. We observed that when $\beta_1 = 10^8 \mathcal{P}^{max}$, 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 similar calculations, which result in $\beta_2 > 1 \times \mathcal{P}^{max}$ and $\beta_3 > 1 \times \mathcal{P}^{max}$. Note: the minimum violation of ϕ_{ddl} and ϕ_{e2e} is one.

5.6. Hybrid Particle Swarm Optimization

The canonical PSO technique uses the constriction factors to balance exploitation and exploration of the search space to get closer to the global optima, hence improving solution quality. Nevertheless, it still suffers from premature convergence or local minima especially when applied on complex and large problems [44]. Its hybridization is proven to perform better in many cases [45]. In particilar, it is shown to perform better in the tasks assignment problem, that is when hybridized with, e.g., the genetic algorithm [**?**], the hill-climbing [13], simulated annealing [**?**], differential evolution [**?**], etc. As compared to the hybridization with genetic, the hybridization with hill-climbing, HCPSO is shown to perform better by Yin et al. [13] for the tasks allocation problem to maximize reliability of distributed systems.

In this work, we apply HCPSOto the probelm at hand, and to tackle its stagnation when applied on large problems, we apply HCPSO. Moreover, we hybridize PSOwith the differential evolution technique, DEPSO, to improve diversification due the mutation and cross-over operators of the differential evolution. Algorithm x show the pseudocode of the hybrid PSO, where line x to y shows application of the hybridization. In every iteration of PSO, the algorithm used in the hybridization is invoked whenever the interval criteria is met, which is intemittent.

input : PSO parameters, DE parameters
output: Software allocation solution sBest.**x**

```
Particles P ← initPSO();
while termination criteria do
    P ←ComputePersonalBest(P);
    sBest ←ComputeSwarmBest(P);
    foreach p ∈ P do
        computeParticleVelocity(p) according to Equation (21);
        computeParticlePosition(p) according to Equation (22);
    end
    if interval criteria then
        P ← optimizeUsingDE(P);
        // P ← optimizeUsingHC(P)
        // P ← optimizeUsingSHC(P)
    end
end
```

Algorithm 1: Hybrid PSO Pseudocode.

5.7. Differential Evolution

Similar to PSO, the differential evolution technique [46, 47] is a population-based meta-heuristic algorithm and employs a fixed set of particles (or agents) to travere the search space. Similar to the genetic algorithm, it uses mutation, crossover and selection operators unlike PSO. It creates each agent $\mathbf{x}$ a mutant $\mathbf{v}$ out of three other random agents from the population, **a,b,c**, and differential weight $F \in [0, 2]$, as shown in Equation (28). The mutant undergoes the crossover process as indicated by Equation (29), where $CF \in [0, 1]$ is the crossover probability, which creates solution $\mathbf{u}$. If the solution $\mathbf{u}$ performs better than the agent, it is selected thus replaces the agent $\mathbf{x}$ as shown in Equation (30).

$$\mathbf{v} \leftarrow \mathbf{a} + F \circ (\mathbf{b} - \mathbf{c}) \tag{28}$$

$$\mathbf{u} \leftarrow crossOver(\mathbf{v}, \mathbf{x}, CF, F) \tag{29}$$

$$\mathbf{x} \leftarrow \begin{cases} \mathbf{u} & \text{if } f(\mathbf{u}) < f(\mathbf{x}) \text{ functions} \\ \mathbf{x} & \text{otherwise} \end{cases} \tag{30}$$

The hybridization with DE helps PSO escape local minima due to the additional stochastic behavior introduced by the differential evolution operators.

5.7.1. Stochastic 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.

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.

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. [16]. 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. 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 6 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

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

Table 7: Ranges of Values for Execution Platforms.

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. .

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.

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 $c_{20} g_{30} n_{10}$, $c_{50} g_{40} n_{20}$

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.

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 [6] 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 4.4, optimizes the calculation of the cause-effect chain delays in the presence 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.

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.

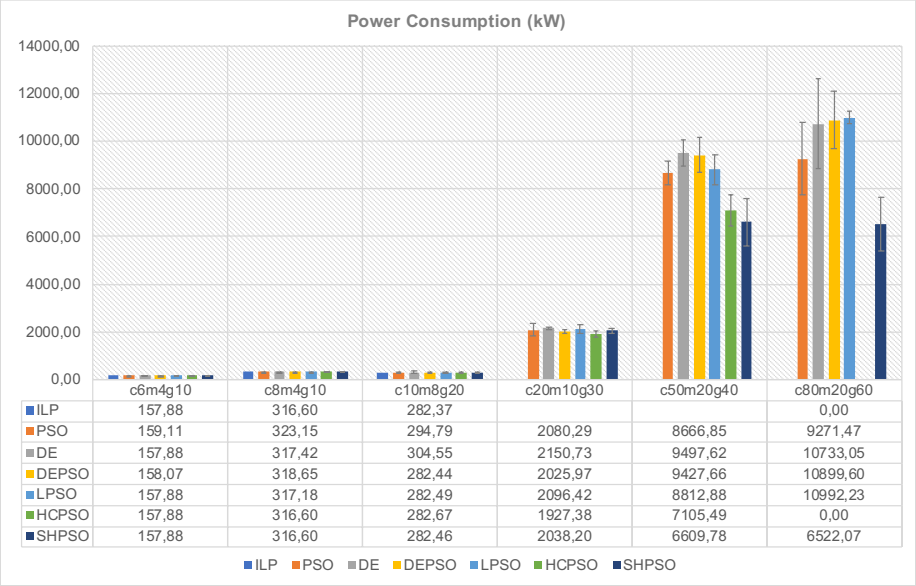
Solution Quality. In the 1^{st} optimization problem $c_6g_{10}n_4$, the ILP, DE, LPSO, HCPSO, SHPSO returned the optimal power consumption, which is 227KW, but DEPSO and PSO returned near optimal solutions with $> 99\%$ quality measures. In the 2^{nd} problem $c_8g_{20}n_6$, similar results are obtained 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, and rest performed worse. In the last three optimization problems $c_{20}g_{30}n_{10}$, $c_{50}g_{40}n_{20}$ and $c_{80}g_{60}n_{20}$, ILPdid not return solutions due to extremly large computation time, hence terminated manually. However, the hybrid algorithms based on hill-climbing such as HCPSO and SHPSO 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 SHPSO 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 13 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 dimenstional problems possible. However, 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

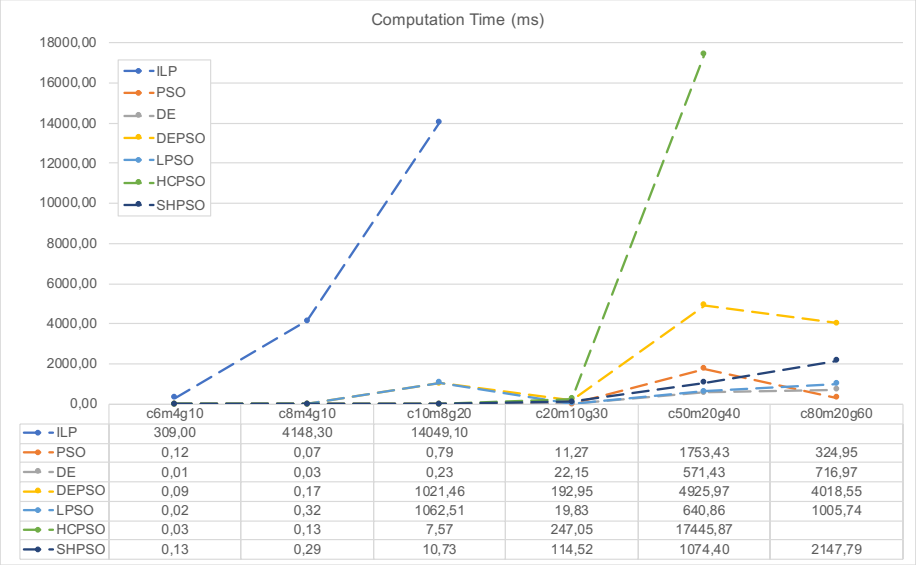
Problem	Algorithm	Fitness		Time (ms)		Quality
		Mean	SD	Mean	SD	
$c_6g_{10}n_4$	**ILP**	227.88	0	309	57.74	100.00
	PSO	229.11	2.38	0.12	0.34	99.46
	DE	227.88	0	0.01	0	100.00
	DEPSO	228.07	0.31	0.09	0.01	99.92
	LPSO	227.88	0	0.02	0.02	100.00
	HCPSO	227.88	0	0.03	0	100.00
	SHPSO	227.88	0	0.13	0.03	100.00
$c_8g_{20}n_6$	**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
	SHPSO	2466535.41	89380.36	2147.79	357.58	100.00

Table 11: Fitness and Allocation Time of the ILPand 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 12: (Near) Optimal Power Consumption of the Different Software Allocation Problems.



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

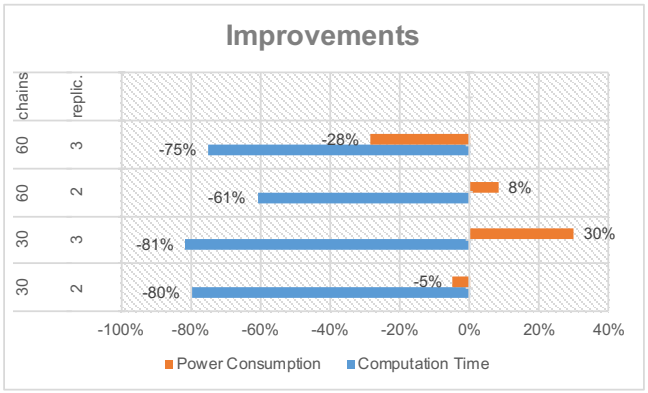


Figure 14: Effect of Approximate Algorithm over Delay Calculations with Replication.

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 DEand 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, HCPSOand SHPSO, however, for the rest it is not uniform.

6.2.2. Analysis of Experiment 2

Table 14 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 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 [48][12][13][49]. Interleaving real-time constraints into the problem adds more complexity to reliability-aware task allocation in distributed systems [50]. As opposed to [51][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, [52] 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 [53] 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) [54][55], and ii) task consolidation to minimize the number of used computing and communication units [56], 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 [57][58]. 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) [59] which considers computation, communication and memory resources, and ii) [60] 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] 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-`

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

[8] 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.

[9] 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.

[10] I. H. Osman, G. Laporte, Metaheuristics: A bibliography, Annals of Operations Researchdoi:10.1007/bf02125421.

[11] Handbook of Metaheuristics, 2006. arXiv:0102188v1, doi:10.1007/b101874.

[12] 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.

[13] 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.

[14] 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

[15] 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.

[16] 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.

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

[18] 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

[19] 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.

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

[21] 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/

[22] 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.

[23] 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.

[24] 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`

[25] AUTOSAR, Specification of Operating System AUTOSAR Release 4.2.2, Tech. rep., AUTOSAR (2018).
URL `https://www.autosar.org/fileadmin/user_upload/standards/classic/4-2/AUTOSAR_SWS_OS.pdf`

[26] L. Sha, T. Abdelzaher, K. E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, A. K. Mok, Real time scheduling theory: A historical perspective (2004). doi:10.1023/B:TIME.0000045315.61234.1e.

[27] R. I. Davis, A. Burns, R. J. Bril, J. J. Lukkien, Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised (2007). doi:10.1007/s11241-007-9012-7.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

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

[35] 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.

[36] S. Sengupta, S. Basak, R. Alan, P. Ii, Particle Swarm Optimization: A survey of historical and recent developments with hybridization perspectives, arXiv:1804.05319doi:10.20944/preprints201809.0007.v1.

[37] S. Mirjalili, Particle swarm optimisation, in: Studies in Computational Intelligence, 2019. arXiv:0510023v1, doi:10.1007/978-3-319-93025-1_2.

[38] 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.

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

[40] 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.

[41] 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.

[42] E. G. Talbi, Metaheuristics: From Design to Implementation, 2009. doi:10.1002/9780470496916.

[43] H. R. Faragardi, M. Vahabi, H. Fotouhi, T. Nolte, T. Fahringer, An efficient placement of sinks and SDN controller nodes for optimizing the design cost of industrial IoT systems, in: Software - Practice and Experience, 2018. doi:10.1002/spe.2593.

[44] D. P. Rini, S. M. Shamsuddin, Particle Swarm Optimization: Technique, System and Challenges, International Journal of Applied Information Systemsdoi:10.5120/ijais-3651.

[45] S. Sengupta, S. Basak, R. Alan, P. Ii, Particle Swarm Optimization: A survey of historical and recent developments with hybridization perspectives, arXiv:1804.05319doi:10.20944/preprints201809.0007.v1.

[46] R. Storn, K. Price, Differential Evolution - A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces, Journal of Global Optimizationdoi:10.1023/A:1008202821328.

[47] 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

[48] 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.

[49] 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.

[50] H. R. Faragardi, R. Shojaee, M. A. Keshtkar, H. Tabani, Optimal ocation 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.

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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).

[59] 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.

[60] 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.