13. Optimized Allocation of Fault-tolerant Embedded Software with End-to-end Timing Constraints

Nesredin Mahmud, Guillermo Rodriguez-Navas, Hamid Faragardi, Saad Mubeen and Cristina Seceleanu. *Mälardalen Real-time Research Center Technical Report (MRTC)*. Submitted to Elsevier Journal of Systems Architecture (JSA), 2019.

Abstract: It is desirable to optimize power consumption of distributed safety-critical software that realize fault tolerance and maximize reliability as a result, to support the increasing complexity of software functionality in safety-critical embedded systems. Likewise, safety-critical applications that are required to meet end-to-end timing constraints may require additional computing resources. In this paper, we propose a scalable software-to-hardware allocation based on hybrid particle-swarm optimization with hill-climbing and differential algorithms to efficiently map software components to a network of heterogeneous computing nodes while meeting the timing and reliability constraints. The approach assumes fixed-priority preemptive scheduling, and delay analysis that value freshness of data, which is typical in control software applications.

Our proposed solution is evaluated on a range of software applications, which are synthesized from a real-world automotive AUTOSAR benchmark. The evaluation makes comparative analysis of the different algorithms, and a solution based on integer-linear programming, which is an exact method. The results show that the hybrid with the hill-climbing algorithms return very close solutions to the exact method and outperformed the hybrid with the differential algorithm, though consumes more time. The hybrid with the stochastic hill-climbing algorithm scales better and its optimality can be deemed acceptable.

13.1 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 electronic control nodes (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 includes the optimization of power and energy, which has been researched at different levels, such as electronic circuit design [15, 37], dynamic power and energy management [58], software/hardware partitioning [13, 25, 61]. In this paper, we propose power-efficient allocation of distributed software applications on heterogeneous computing nodes, which are ECUs with different power-consumption specifications, processor speed and failure rates.

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 [59], platform-based system design [47] and the Y-chart design [30] approaches. It is a type of job-shop problem with constraints, and therefore finding an optimal solution, in the general case, is NP-hard [23]. 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 [45]. 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 [35], we demonstrate the limitation of integer-linear programming (ILP) [9] using exact method by the CPLEX solver. Similarly, the scalability issues of exact methods on software allocation is indicated in several works [45]. In contrast, heuristic methods device a working technique to solve practical problems, which are usually large-scale, non-linear, without guarantee of optimality

[19, 10]. 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" [41].

Metaheuristics has found wide applications in many domains, e.g., cellular networks, cloud computing, software design, etc [1]. Many of the existing meta-heuristic 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 works, such as by Qin-Ma et al. [28] on maximizing reliability of distributed computing systems using honybee algorithm, and maximizing reliability of distributed systems using hill-climbing particle-swarm optimization by Yin et al. [62]. In this work, we apply differential evolution and hybrid particle-swarm optimization algorithms on 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 nodes realize end-to-end functionality, also known as *cause-effect* chains. The chains are triggered by different sampling rates, also known as *multirate* [56]. The propagation of signals over multirate chains result in undersampling/oversampling effects, which makes end-to-end timing analysis difficult [38]. In order to maximize software applications reliability and meet their reliability goals, we implement fault tolerance. The contributions of our work are: (i) an allocation mechanism to maximize reliability of a distributed safety-critical software via replication of software components, which are mapped to different computing nodes, (ii) a hybrid particle optimization to efficiently map a fault-tolerant safety-critical software on a network of heterogeneous computing nodes, considering exact timing and reliability models, with respect minimization of power consumption, (iii) an approximation algorithm to minimize the overhead of replication on the calculation of end-to-end age delay, (iv) performance analysis of various meta-heuristic algorithms such as differential evolution, particle-swarm optimization and the latter’s hybrids with differential-evolution, hill-climbing and stochastic hill-climbing algorithms, and an exact method based on integer-linear programming.

Our approach is evaluated on synthetic automotive applications that are generated according to the real-world automotive benchmark proposed by Kramer et al. [32]. In the evaluation, we show comparative performance of the various optimization algorithms in terms 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 13.2 provides a brief overview of AUTOSAR, emphasizing on end-to-end timing and reliability modeling, and software allocation, Section 13.3 describes the AUTOSAR system model, including timing analysis, reliability and power-consumption assumptions. Formulation of the software allocation problems is presented in Section 13.4, 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 13.5. The evaluation of our proposed methods are demonstrated in Section 13.6 using the automotive benchmark. Our work is compared to related works in Section 13.7. Finally, we conclude the paper in Section 13.8, and outline the possible future work.

13.2 AUTOSAR

The AUTomotive Open System ARchitecture (AUTOSAR) partnership has defined the open standard AUTOSAR for automotive software architecture that enables manufacturers, suppliers, and tool developers to adopt shared development specifications, while allowing sufficient space for competitiveness. The specifications state standards and development methodologies on how to manage the growing complexity of Electronic/Electrical (E/E) systems, which take into account the flexibility of software development, portability of software applications, dependability, efficiency, etc., of automotive solutions. 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 [40].

13.2.1 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

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

level. The runnables are mapped to tasks, and subsequently are scheduled by the AUTOSAR operating system [6]. 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 [40].

13.2.2 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 [5] 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.

13.3 System Model

Figure 13.1 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 $\langle \mathscr{N}, \mathscr{B} \rangle$, where $\mathscr{N} = \{n_h : h = 1,\ldots,n_{\mathscr{N}}\}$ are the computing node, $\mathscr{B}$ is the shared network bus. Each application has user-defined requirement $(\mathsf{RL},\mathsf{EE},\mathsf{CL})$, where the tuple elements respectively refer to the reliability goals (or requirement), end-to-end timing requirements and criticality levels. And each computing node has provisions (or capabilities) $(\mathsf{HZ},\mathsf{PW},\mathsf{FR})$, where the tuple elements respectively refer to the processor speed, power-consumption specifications and failure rates.

Notations

For easy reading, we introduce the main notations used throught the paper as shown in Table 13.1.

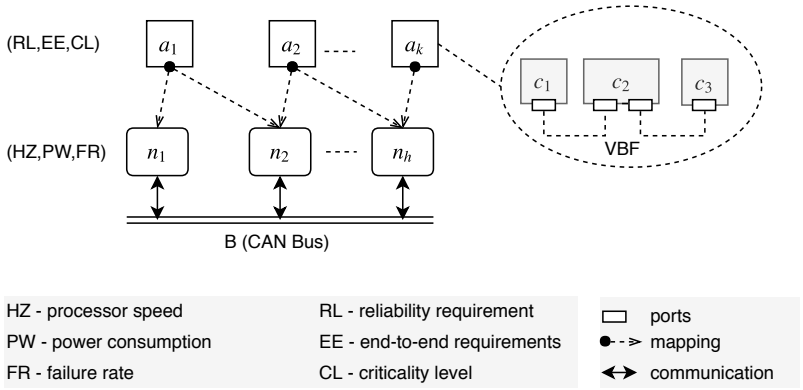


Figure 13.1: System model.

13.3.1 Software Applications

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

Definition 1 (AUTOSAR Software Application[2]) *We represent the AUTOSAR software application 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* $i \neq j$*. 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 node* $n_h \in N$*.*

A path in the graph $\Gamma_l = r_i \rightarrow, ..., \rightarrow, r_j$ represents a *cause-effect* chain, where $r_i, r_j \in \mathscr{V}(a_k)$ are source and sink of the chain, e.g., in the application a_1 in Figure 13.2, $r_1 \rightarrow r_3 \rightarrow r_5$ 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 to actuate the 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 the chain. The set

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
$\mathscr{V}(a_k) \in R_i, \mathscr{E}(a_k)$	nodes and links of a_k, respectively
$\Gamma = \{\Gamma_i : i = 1,\ldots,n_\Gamma\}$	paths of a_k and denote end-to-end chains
• Related to software components	
$C = \{c_i : i = 1,\ldots,n_C\}$	software-component types used in $a \in A$**
$Q_i = \{q_{i,j} : j = 1,\ldots,n_{Q_i}\}$	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 13.1: *The Main Notations Used Throughout 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 cardinality of the set.*
*** Throughout the paper the superscript is A_k is removed, e.g., from C^{A_k} to improve readability. However, it is used whenever it is needed, e.g., to calculate the total power consumption of all applications.*

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

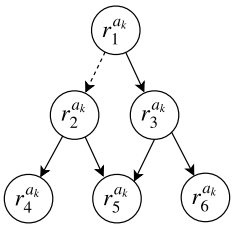


Figure 13.2: Software application example, modeled as DAG of runnables.

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 13.2: Timing specifications of the runnables from Figure 13.2

13.3.2 Scheduling Software Applications

We assume software applications share the execution platform such as the computing nodes and the on-board network bus, following the mixed-criticality design [55]. Thus, software applications with different software criticality should be isolated in order to prevent interference of lower-criticality applications on the higher-critical applications. For example the brake-by-wire system realizes a safety-criticality functionality, and is distributed over multiple computing nodes. Another application with infotainment functionality also shares the nodes. The mixed-criticality design ensures that 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 [55]. In this work, we consider the *partitioned criticality (PC)* (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 monotonic assignment [7]. In contrast to other techniques, CAPA is easy to implement and does not require runtime monitoring, e.g., using servers [2, 3, 27], though not efficient[3]. Thus, the software applications are schedulable according to CAPA if the runnables, messages, and chains in the applications are schedulable, that is, they meet their corresponding deadlines.

Next, we explain the mapping rules applied in this work.

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

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

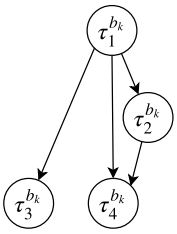


Figure 13.3: A Software application modeled as Directed Acyclic Graph.

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 13.4: Tasks timing specifications after merging.

this work, we merge the runnables $a, b \in \mathscr{V}(a_k)$ into the task $v \in \mathscr{V}(b_k)$ if the following two conditions are satisfied: (i) $(a,b) \in \mathscr{E}(a_k)$ 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 schedulable, 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.

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 criticality, the higher the priority that 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 determine the criticality and priority of tasks τ_1, τ_2, respectively; $\mathscr{V}(b_i) and \mathscr{V}(b_j)$ return the tasks of b_i and a_j, respectively.

The tasks are scheduled using the *fixed-priority preemptive scheduling policy* (FPPS) [49], and the schedulability analysis is conducted via the classical response-time analysis (RTA) as shown by Equation (13.1) [7]. The task τ is schedulable if the response time of a task δ_τ is less than or equal to its deadline DL, i.e., tasks a $\delta_\tau \leq \mathsf{DL}_\tau$. The

$$R_\tau = C_\tau + \sum_{j \in hp(\tau)} \left\lceil \frac{R_\tau}{P_j} \right\rceil C_j, \tag{13.1}$$

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

The messages in the CAN bus are scheduled using the 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 achieved 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 schedulability of messages is checked using the classical response-time analysis of the CAN network using Equation (13.2), as presented by Rob Davis et. al [14]. The worst-case response time of a message is computed as the sum of its *jitter* (that is, the time taken by the sender task to queue for transmission) 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 the message to be transmitted) c_m over the network.

$$R_m = J_m + w_m + c_m \tag{13.2}$$

$$w_m = B_m + \sum_{\forall k \in hp(m)} \left\lceil \frac{w_m + \tau_{bit}}{P_k} \right\rceil c_k \tag{13.3}$$

$$B_m = \max_{\forall k \in lp(m)} (c_k), \tag{13.4}$$

Note: we assume no jitter, therefore, the interference formula is reduced as shown in Equation (13.3), 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 (13.4); $hp(m)$ finds the set of higher-priority messages, which interfere with the transmission of the message m.

Scheduling Cause-effect Chains

The chain consists of independently clocked tasks, which results in undersampling/oversampling effects. As a result, a data that propagates across the chain is characterized by having various delays, which are discussed in detail by Feiertag et al. [22] in the context of single-register buffer communication, which is a common practice in control systems design, e.g., automotive software applications [8]. In this work, we demonstrate the two widely used semantics in the automotive domain: *age* delay and *reaction* delay, 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 (response) at the output

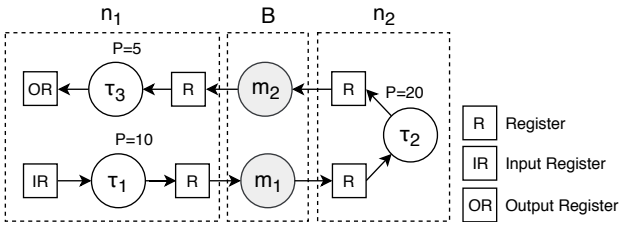


Figure 13.5: A Cause-effect chain, mapped on nodes n_1 and n_2.

register. 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.

Consider the chain $\tau_1 \rightarrow \tau_2 \rightarrow \tau_4$ in the task graph from Figure 13.2. Assume the chain is mapped to the computing nodes n_1 and n_2 as illustrated in Figure 13.5, and that the tasks in the chain communicate using the single-register buffers. The tasks τ_1 and τ_4 execute on node n_1, whereas, task τ_2 executes on node n_2. The input data can arrive at any time in the input register IR.

The execution behavior of the chain is illustrated in Figure 13.6 over two hyper-periods. Note that, 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-line 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. The timing constraint corresponding to the age delay 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. [39].

Thus, if the chain is mapped to a single computing node, the age delay Δ^{sub} is computed using Equation (13.5), that is by taking the difference between the activation of the sink and the source tasks, plus the response-time of the sink task. Otherwise, if it is mapped to multiple nodes and the bus, the delay Δ is computed compositionally by identifying the subchains using Equation (13.6). 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

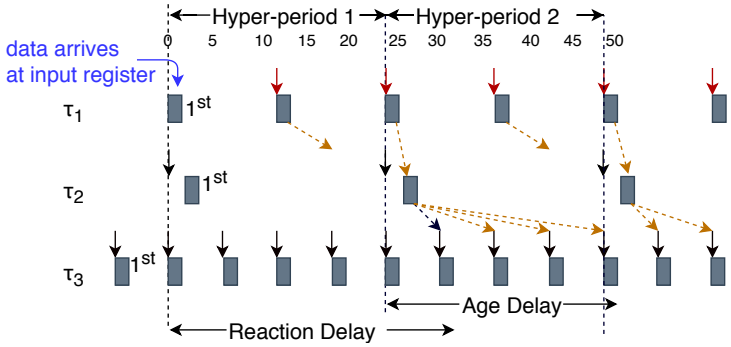


Figure 13.6: Reaction and age delays in the cause-effect chain, shown in Figure 13.5.

into consideration.

$$\Delta^{sub}(\gamma) = \alpha(sin(\gamma)) - \alpha(src(\gamma)) + \delta(sink(\gamma)) \qquad \text{single node} \qquad (13.5)$$

$$\Delta(\gamma) = \sum_{i \in I_\gamma} \Delta^{sub}(i) + \sum_{j \in J_\gamma} [\delta^{msg}(j) + P_{suc(j)}], \qquad \text{multiple nodes} \qquad (13.6)$$

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

13.3.3 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 $[0,t]$ [24]. Redundancy is the most common way of 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. [16], 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 to issue an input. In the software application A_k, the primary software component is denoted as $q_{i,1}$, whereas the secondary software component, which is in the stand-by is identified, by $q_{i,2}$.

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.

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.*

Table 13.3: *Allocation of the software components.*

(ii) Hot stand-by redundancy (also known as Primary/backup) is used for detection and replacement of failed components.

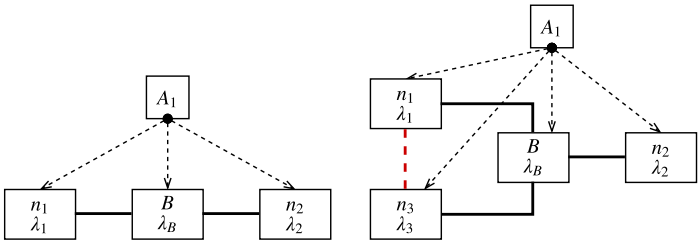
(iii) Software components need to be replicated only if the application's reliability requirement is not met without replication, otherwise they are not replicated.

(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 the 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, it 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 reliability of a computing node (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* inter-dependencies created between computing nodes as the result of replication and allocation. To illustrate the complexity, let us assume a software application A_k, having component configurations without and with replication as shown in Table 13.3a and 13.3b, respectively, where $q_{i,j}$ is the j^{th} software component replica of software component type $c_i \in C_i$. Note: the superscript k is not used for sake of readability.

The reliability of the software application without replication forms a series path, indicated by the reliability block diagram (RBD) of Figure 13.7a. Hence, it is computed as a product of the reliability of n_1, n_2 and B. However, with replication, two computing nodes can form 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 making 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 13.7b does not accurately capture the reliability calculation of the application with replication. Note that, the red-line arrows between



(a) Without replication. (b) With replication.

Figure 13.7: Reliability Block Diagrams (RBD) of the software application.

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 detail in Subsection 13.4.5.

13.4 Software-to-Hardware Allocation Problem

The software applications are allocated to the execution platform by mapping the software-component replicas (or software components) $Q_i^{A_k}$ to the computing nodes $\mathscr{N}$. The software-to-hardware allocation problem is to find the mapping $\mathbf{x} : Q_i^{A_k} \rightarrow \mathscr{N}$ 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 computing node n_h, where $h = x_{ij}^k$, of the application A_k.

13.4.1 Total Power Consumption

The total power consumption of the applications $\mathscr{P}_{total}(\mathbf{x})$ is computed as the sum of the power consumption of the computing nodes to which the applications are allocated. The power consumption of a node is computed according to the linear model proposed by Fan et al. [18] as shown by Equation (13.7), which is directly proportional to its load (or utilization) and is inductively formulated from experimental results.

$$\mathscr{P}(u) = P_{idle} + (P_{busy} - P_{idle}) * u, \tag{13.7}$$

where u is the utilization of a computing node, 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 [26], AutoBench [17], etc.

The utilization of a computing node 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 (13.8), where T_i refers to the tasks implementing the component type $c_i^{A_k}$.

$$T_{n_h} = \{T_i^{A_k} | \forall kij \; h = x_{ij}^k\} \text{ forall } h = 1, ..., n_N \tag{13.8}$$

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

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

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

$$\mathscr{P}_{total}(\mathbf{x}) = \sum_{h=1}^{n_N} \mathscr{P}(u_h(\mathbf{x})) \text{ for all } h = 1, .., n_N, \tag{13.9}$$

13.4.2 Tasks and Messages Timing Constraints

The tasks timing constraints ensure that the tasks in the distributed system meet their respective deadlines, that is $\forall \tau \in T_{n_h} \; ResponseTime_\tau(\mathbf{x}) \leq \mathsf{DL}_\tau$ for all $h = 1, ..., n_N$, following Equation (13.8). 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 determined by traversing the edges of the tasks graphs. If the edge relates tasks located on different nodes, 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 $\mathscr{R}_\tau^{b_k} = \{(\tau, n) \in \tau \times N_\tau^{A_k}\}$ is the set of the replicas of type $\tau \in \mathscr{V}(b_k)$, now assume $(t1, t2) \in \mathscr{E}(b_k)$ is an edge in the task graph and $\mathscr{R}_{t2}^{b_k}, \mathscr{R}_{t2}^{b_k}$ are the replicas of type $t1$ and $t2$, respectively. The set of subedges between $t1$ and $t2$ is $\mathscr{R}_{t2}^{b_k} \times \mathscr{R}_{t2}^{b_k}$, 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 is determined as,

$$M = \{m_{i,j} | \forall (t1, t2) \in \mathsf{EE}(b_k(\mathbf{x})) \forall (i, j) \in \mathsf{RL}_{t1}^{b_k} \times \mathsf{RL}_{t2}^{b_k} \; n_i \neq n_j\}, \tag{13.10}$$

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, $n_i, n_j \in \mathscr{N}$.

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

13.4.3 End-to-end Timing Constraints

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

The delay calculation of a chain Γ is multiplicity Γ^* due to replication. Consider the chain $\Gamma = (\tau_1,...,\tau_l)$. The set of chains with replication is a cartesian product of the tasks replicas (or the tasks nodes mapping to computing nodes according to $\mathbf{x}$ in the chain, that is $\Gamma^*(\mathbf{x}) = \mathscr{R}_{\tau_1}^{b_k} \times,...,\times \mathscr{R}_{\tau_l}^{b_k}$, where l is the chain length. Assume that we want to calculate the age 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 in the chain. The subchains I are subsets of the chain γ where the communication between the sender and receiver tasks of the chain use a network bus. That is, if t_i is the sender task, and its receiver task t_{i+1} is mapped to a different node, 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,

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

according to the age-delay formula shown in Equation (13.6), 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 for a mapping $\mathbf{x}$ is:

$$\forall \gamma \in \Gamma^*(\mathbf{x})^{A_k}\ \Delta_\gamma^{A_k}(\mathbf{x}) \leq \mathsf{EE}_\gamma^{A_k}, \tag{13.11}$$

Example 1 (Delay Calculation) *Consider the chain $\Gamma = \tau_1 \rightarrow \tau_2 \rightarrow \tau_4$ from Figure 13.3 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 13.3 (b), i.e., with replication. Thus, the nodes to which τ_1 and τ_2 are mapped are $\mathscr{R}_{\tau_l}^{b_k} = \mathscr{R}_{\tau_2}^{b_k} = \{n_1, n_2\}$, and τ_4 to $\mathscr{R}_{\tau_4}^{b_k} = \{n_2, n_3\}$, by infering the mappings of respective components. Table 13.4 illustrates how to compute the chains, considering replication of degree 2, which is $\Gamma^* = \mathscr{R}_{\tau_1}^{b_k} \times \mathscr{R}_{\tau_2}^{b_k} \times \mathscr{R}_{\tau_4}^{b_k}$, and also how to compute the subchains and messages of each chain $\gamma \in \Gamma^*$. The delays of the*

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

Table 13.4: *Chains with replication of degree 2 for the chain* $\Gamma = \tau_1 \rightarrow \tau_2 \rightarrow \tau_4$*, its subchains I and messages J.*

subchains is computed according to the age-delay semantics demonstrated in Subsection 13.3.2.

13.4.4 Approximation of Age 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 13.3.2. For these reasons, the age delay computation is sometimes prohibitively expensive considering the meta-heuristic algorithms, which compute large-space candidate solutions over thousands of iterations.

Therefore, we propose an approximation algorithm to efficiently compute the delays. In the case that the chain is mapped to a single node, the delay δ^{sub*} is calculated as the sum of the response time of each task in the chain, i.e.g, $\delta^{sub*} = \delta$. However, if the chain is mapped to multiple nodes, the delay is computed compositionally as explained in Equation (13.6) of Subsection 13.3.2.

$$\Delta^{approx} = \sum_{i \in subch(\gamma)} \delta^{sub*}(i) + \sum_{j \in J_\gamma(\mathbf{x})} \left[\delta_j^{msg}(\mathbf{x}) + P_{suc(j)}(\mathbf{x}) \right], \tag{13.12}$$

where $subch()$ computes subchains of the the chain γ in the case that chain is mapped to multiple nodes. The predicate $suc(j)$ determines the receiver task of the message m_j.

13.4.5 Software-Applications Reliability Constraints

The applications reliability constraints ensure the mapping $\mathbf{x}$ satisfies the user-defined reliability requirements, that is $Reliability_{A_k}(\mathbf{x}) \leq \mathsf{RL}_{A_k}$, for all $k = 1...n_A$. The reliability of each application is computed over t period of time from the computing nodes 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 nodes 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 nodes and the network bus as shown in Equation (13.13). Note that, if application does not use the shared bus, then $Reliability_B = 1$. Equation (13.14) finds the nodes N^{A_k} that the application $a_k$uses by traversing the partition $\mathbf{x}$ in linear time.

$$Reliability_{a_k}(\mathbf{x}) = Reliability_{\mathscr{N}^{A_k}}(\mathbf{x}) * Reliability_{\mathscr{B}} \tag{13.13}$$

$$\mathscr{N}^{A_k} = \{e \in \mathscr{N} | \forall ij\ e = m_h\}, \text{ where } h = x_{ij}^k \tag{13.14}$$

We assume applications are mutually exclusive, i.e., 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 nodes is $Reliability_N(\mathbf{x}) = e^{-\lambda_N(\mathbf{x})t}$, where $\lambda_N(\mathbf{x})$ is the failure rate of an N-node system over the partition $\mathbf{x}$. The system failure-rate is computed using the state enumeration as shown in [34], which is an exact technique to calculate reliability, as opposed to using series-parallel technique motivated in Subsection 13.3.3. 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 node. Assume the Boolean variable $s_h \to \Omega$, 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 nodes $N = \{n_1, .., n_{n_N}\}$, the states of the nodes (or configuration) is indicated by the N-cardinality set $S = \{s_1, ..., s_{n_\mathscr{N}}\}$.

- $\xi = \Omega^S$ are elementary events that correspond to the possible configurations of the nodes N, therefore, the events are mutually exclusive. Consider $N = \{n_1, n_2, n_3\}$, Table 13.5 shows the possible configurations ξ. Assume the configuration $s \in \xi = \{0,1,0\}$, it shows n_1 and n_3 fail as indicated by $s_1 = 0, s_3 = 0$, respectively, and n_2 operates as indicated by $s_2 = 1$.

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

nodes 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 13.5: *Example of the application reliability calculation using state enumeration over 10-year 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 13.3, the variable* $s_{c_i} \in \{0,1\}$ *indicates if the replicas of type* c_i *fails or functions, respectively.*

- $f : \xi \rightarrow \{0,1\}$ determines the status of the application 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 in Equation (13.15). The component replica* $q_i, j \in Q_i$ *of type* c_i *fails if* n_h *fails, that is* $s_h = 0$*.*

$$f_s(\boldsymbol{x}) = \begin{cases} 0 & \text{if } \exists i\ c_i | \forall j\ s_h = 0 \\ 1 & \text{otherwise} \end{cases} \qquad \text{where } h = x_{ij} \tag{13.15}$$

Thus, the failure rate of the N-node 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 13.5 over a 10-year (or 87600h) operational lifetime. The reliability of the nodes 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* $Reliability_N * Reliability_B = 0.99649339932$*.*

13.4.6 Software Allocation Optimization

The software allocation is defined as a single-objective optimization problem. The objective function $\mathscr{P}(\mathbf{x})$ is a cost function which minimizes the total power consumption of the software applications as deployed in the heterogeneous computing nodes, where $\mathbf{x}$ is the decision variable (or solution) of the optimization. The cost function is formulated in Equation 13.16, with inequality constraints shown by Equation (13.17, 13.18,13.19). The constraints ensure the solution meet the reliability requirements, the tasks deadlines, and the end-to-end requirements of the chains.

$$\min_{\mathbf{x}\in X} \mathscr{P}_{total}(\mathbf{x}) \qquad \text{subjected to:} \qquad (13.16)$$

$$Reliability_{A_k}(\mathbf{x}) \leq \mathsf{RL}_{A_k} \qquad \text{forall } k = 1,...,n_A \qquad (13.17)$$

$$\forall i \in T_{m_h}\ ResponseTime_{\tau_i}^{A_i}(\mathbf{x}) \leq \mathsf{DL}_{\tau_i}^{A_i} \qquad \text{forall } h = 1,...,n_N \qquad (13.18)$$

$$\forall \gamma \in \Gamma^{*A_k}\ Delay_{\gamma}^{A_k}(\mathbf{x}) \leq \mathsf{EE}_{\gamma}^{A_k} \qquad \text{forall } k = 1,...,n_A \qquad (13.19)$$

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.

13.5 Solution using Hybrid Particle Swarm Optimization (PSO)

When ILP is used to optimize the problem discussed in Section 13.3, the CPLEX solver returned optimal solutions to problems in the *small* and *medium* range applications, where the small problem refers to a software application with software components less than 10 and chains less than 30. Whereas, the medium problem refers to applications with components less than 15 and chains less than 40. The specifications are stipulated from the real automotive benchmark proposed by Kramel et al. [32]. However, the ILP approach, as also shown for similar problems, suffers from the scalability problem, that does not return solutions for large-scale software allocation problems. In this section, we propose multiple meta-heuristic algorithms 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, although the power consumption of the applications may not be optimal, the solution can be deemed acceptable. In fact, PSO and DE are used together for improved performance in several optimization problems [48, 36], likewise, PSO is used with local search techniques such as Hill climbing to intensify the search [62].

Finally, we evaluate the different meta-heuristic methods based on solution quality and computation time for different software allocation problems.

13.5.1 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 [29]. Since then, it is extended in order to address various metaheuristic optimization challenges, such as intensification, diversification, convergence analysis, local optima, parameter tuning and computation time [48]. It is successfully applied on several complex real-world problems, e.g., diagnosis and classification of diseases, efficient engineering designs, tuning control design parameters and scheduling problems [42].

In PSO, the population (or swarm) $\mathscr{PN} = \{p_1, p_2, \ldots, p_N\}$ is a collection of particles, organized according to a certain population topology [33]. A particle has a position $\mathbf{p}$ and a velocity $\mathbf{v}$. PSO is memory-based in the sense that it remembers the best position of a particle, identified by $\mathbf{p}_{bst}$. 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 13.20. 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's best position. In fact, the next velocity of the particle is the resultant of the attraction components and the current velocity. Thus, the next position of the particle is the resultant of its current position and its next velocity as shown by Equation (13.21).

$$\mathbf{v} \leftarrow \omega\mathbf{v} + c_1 Rand() \circ (\mathbf{p}_{bst} - \mathbf{p}) + c_2 Rand() \circ (\mathbf{z} - \mathbf{p}) \tag{13.20}$$

$$\mathbf{p} \leftarrow \mathbf{p} + \mathbf{v}, \tag{13.21}$$

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

13.5.2 Solution Representation

The software allocation is a discrete problem, as such, the solutions are discrete values. The PSO was originally proposed for continuous problem, nevertheless, it is applied to discrete problems successfully as well, e.g.,

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

Figure 13.8: Binary (0-1) representation.

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

Figure 13.9: Integer representation.

Figure 13.10: Solution representations for components $\{c_1, c_2, c_3\}$ Mapped to computing nodes $\{n_1, n_2, n_3\}$ based on Table 13.3b.

to the sales man problem [11]. There are two commonly used solution representations of PSO for discrete problems: the binary (0-1) and integer representations, which are demonstrated in Figure 13.10 using the example provided in Figure 13.3. In the latter, the variable indicate the computing-node 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 13.9, and it is computationally more efficient considering our problem. Following the integer-representation approach, the solution is discretized by approximating 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 (13.22) is the solution of a_k, and $x^k_{ij} = h \in \{1, \ldots, n_N\}$ denotes the mapping of the software-component replica $q^k_{i,j}$ to the computing node n_h.

$$\mathbf{x}^k = \begin{bmatrix} x^k_{11} & x^k_{12} & \cdots & x^k_{1K} \\ x^k_{21} & x^k_{22} & \cdots & x^k_{2K} \\ \vdots & \vdots & \ddots & \vdots \\ x^k_{N_c 1} & x^k_{N_c 2} & \cdots & x^k_{N_c K} \end{bmatrix} \tag{13.22}$$

13.5.3 Fitness Function

The fitness function $f : \mathbf{x} \rightarrow \mathbb{R}$ is a type of objective function that summarizes the contributions of the decision variables via real numbers. The fitness value is used to compare feasible solutions, i.e., the higher the fitness the better it is (for minimization problem as in our case, a lower fitness value is 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 [53, 19]. Thus, it combines the objective function (i.e., the power-

consumption minimization) with the reliability and timing constraints into a single function (i.e., fitness function) by using penalty functions.

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 $\mathbf{x}_2$ 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 $\mathbf{x}_2$ 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 in comparison with the case in which the heuristic algorithm only knows that they are both infeasible. solution.

Consequently, the original constrained optimization problem is transformed into unconstrained optimization problem by extending the objective function $\mathscr{P}_{total}(\mathbf{x})$ with the constrains, which are represented by a set of *penalty* functions $\{\phi_{rel}(\mathbf{x}), \phi_{ddl}(\mathbf{x}),\ \phi_{e2e}(\mathbf{x})\}$. The first penalty function (Equation (13.23) 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 the value of the penalty function. Similarly, the $\phi_{ddl}(\mathbf{x})$ Equation (13.24) and $\phi_{e2e}(\mathbf{x})$ Equation (13.25) penalty functions returns the violation due to missing deadlines and missing end-to-end requirements, respectively, that is 0 no violations otherwise return the magnitude of the violations.

$$\phi_{rel}(\mathbf{x}) = \sum_{k=1}^{n_A} \max\{0, Reliability_{A_k}(\mathbf{x}) - \mathsf{RL}_{A_k}\} \tag{13.23}$$

$$\phi_{ddl}(\mathbf{x}) = \sum_{\forall \tau \in T_{m_h}} \max\{0, ResponseTime_{\tau}(\mathbf{x}) - \mathsf{DL}_{\tau}\} \tag{13.24}$$

$$\phi_{e2e}(\mathbf{x}) = \sum_{\forall \gamma \in \Gamma^{A_k}} \max\{0, Delay_{\gamma}(\mathbf{x}) - \mathsf{EE}_{\gamma}\} \tag{13.25}$$

Thus, the fitness function can be written as follows.

$$\min_{\mathbf{x} \in X} f(\mathbf{x}) = \mathscr{P}_{total}(\mathbf{x}) + \beta_1 \phi_{rel}(\mathbf{x}) + \beta_2 \phi_{ddl}(\mathbf{x}) + \beta_3 \phi_{e2e}(\mathbf{x}), \tag{13.26}$$

where β_1, β_2 and β_3 are penalty coefficients used to tune the weights of the penalty functions regarding the range of the objective function. In Section 13.5.4, the proper values of the penalty coefficients are discussed in more details.

13.5.4 Penalty Coefficients

To calculate the values of the penalty coefficients β_1, β_2 and β_3, we use the analytical approach similar to the one proposed in [21], 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 $\mathscr{P}(\mathbf{x})$ 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 [19]. 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 [53].

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

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

Let us assume that (i) $\mathscr{P}^{min} = 0$, (ii) $\mathscr{P}^{max}$ 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 infeasible solution. Hence,

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

The experimental evaluation discussed in the next section verifies this discussion. We observed that when $\beta_1 = 10^8 \mathscr{P}^{max}$, 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 $\beta_2 > 1 \times \mathscr{P}^{max}$ and $\beta_3 > 1 \times \mathscr{P}^{max}$. Note that, the minimum violation of ϕ_{ddl} and ϕ_{e2e} is one each.

13.5.5 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 [48]. In particular, it is shown to perform better in the tasks assignment problem, that is when hybridized with, e.g., the genetic algorithm [46], the hill-climbing [62], simulated annealing [64], differential evolution [54]. As compared to the hybridization with genetic, the hybridization with hill-climbing HCPSO is shown to perform better by Yin et al. [62] for the tasks allocation problem to maximize reliability of distributed systems.

In this work, we apply HCPSO to the problem at hand, and to tackle its stagnation when applied to large problems. Moreover, we hybridize PSO with the differential evolution technique, DEPSO, to improve diversification by applying the mutation and cross-over operators of the differential evolution. Algorithm 1 show the pseudocode of the hybrid PSO. Line 3 and 4 compute the personal best and the swarm best solutions, respectively. For each particle in the swarm, the velocity and position is computed in Lines 5-8. Lines 9-13 apply the hybridization based on the choice of the algorithm, i.e., DE, HCPSO and SHPSO intermittently, i.e., whenever the interval criterion condition is met.

```
   input : PSOparameters, DEparameters
   output: Software allocation solution sBest.x
 1 Particles P ← initPSO();
 2 while termination criteria do
 3 |   p_bst ←ComputePersonalBest(P);
 4 |   z ←ComputeSwarmBest(P);
 5 |   foreach p ∈ P do
 6 |   |   computeParticleVelocity(p) according to
   |   |    Equation (13.20);
 7 |   |   computeParticlePosition(p) according to
   |   |    Equation (13.21);
 8 |   end
 9 |   if interval criteria then
10 |   |   P ← optimizeUsingDE(P);
11 |   |   // P ← optimizeUsingHC(P)
12 |   |   // P ← optimizeUsingSHC(P)
13 |   end
14 end
```

Algorithm 1: Hybrid PSO pseudocode.

13.5.6 Differential Evolution

Similar to PSO, the differential evolution technique [51, 12] is a population-based meta-heuristic algorithm and employs a fixed set of particles (or agents)

to traverse the search space. Similar to the genetic algorithm, it uses mutation, crossover and selection operators unlike PSO. It creates each agent **x** a mutant **v** out of three other random agents from the population, **a,b,c**, and differential weight $F \in [0,2]$, as shown in Equation (13.27). The mutant undergoes the crossover process as indicated by Equation (13.28), where $CF \in [0,1]$ is the crossover probability, which creates solution **u**. If the solution **u** performs better than the agent, it is selected thus replaces the agent **x** as shown in Equation (13.29).

$$\mathbf{v} \leftarrow \mathbf{a} + F \circ (\mathbf{b} - \mathbf{c}) \tag{13.27}$$

$$\mathbf{u} \leftarrow crossOver(\mathbf{v}, \mathbf{x}, CF, F) \tag{13.28}$$

$$\mathbf{x} \leftarrow \begin{cases} \mathbf{u} & \text{if } f(\mathbf{u}) < f(\mathbf{x}) \text{ functions} \\ \mathbf{x} & \text{otherwise} \end{cases} \tag{13.29}$$

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

Stochastic Hill-climbing PSO

Hill-climbing is a popular local search algorithm 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 **x** is neighbor to **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, which subsequently replaces the current candidate solution if it performs better and continues until the 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 time complexity is high especially for high-dimensional problems. To offset this problem, we 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.

13.6 Evaluation

In this section, we evaluate our proposed hybrid PSO algorithms for the allocation of software applications to heterogeneous computing nodes, which conform to the system model presented in Section 13.3. The algorithms are evaluated against different specifications of automotive

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 13.6: *Specification of the applications for evaluation.*

software applications and execution platforms with regard to effectiveness, stability and scalability. The software-application specifications consist of the number of software components c, runnables r, tasks t and cause-effect chains g. The specifications are synthesized from the automotive benchmark proposed by Kramel et al. [32]. 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 electrical/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 application 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 13.6. The specification classes are useful to evaluate and discuss the effectiveness and scalability of the different optimization algorithms. The first specification class Spec-I encompasses 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 stretch 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 nodes. The values of these parameters are shown in Table 13.8.

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

Table 13.7: *Ranges of values for applications requirements.*

Parameter	Range
Nodes n_N	$[4,10]$
P_{idle} (Watt)	$[10,200]$
P_{busy} (Watt)	$[20,500]$
λ_n, λ_B (h^{-1})	$[10^{-8}, 10^{-6}]$
Hz	processor speed*

Table 13.8: *Ranges of values for execution platforms.*
*Note: * is reflected in the worst-case execution time.*

Applications Requirements Specifications

Table 13.7 shows the range of values used in our experiments 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 requirements are assumed as a function of length of the chain n_Γ, i.e., the longer the chain the higher the number. The reliability range of a typical safety-critical automotive application is usually given in higher degree of 9, for operation of over a long period of time, which implies almost no failure during the specified duration.

Evaluation Setup

The evaluation is conducted on a MacBook Pro laptop computer, 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.

13.6.1 Results

We conducted two experiments: i) the first experiment is designed to compare the performance such as the convergence time, computation time, optimality (or quality of solutions) and stability 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 due to the computation of end-to-end delays, and also to evaluate the effect of the approximation algorithm proposed in Subsection 13.4.4 to reduce the overhead.

Experiment 1

Based on the ranges specifications presented in Table 13.7, Table 13.6 and Table 13.8, we synthesized six optimization problems as shown in Table 13.10. The problems emulate the software allocation of safety-critical distributed automotive applications on heterogeneous computing nodes with

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	Penalty Function: $\beta_1 = \mathscr{P}^{max}$, $\beta_2 = \mathscr{P}^{max}, \beta_3 = 10^8\,\mathscr{P}^{max}$, where $\mathscr{P}^{max} = \sum_{i \in \mathscr{N}} P_{busy_i}$

Table 13.9: *Parameters settings of the metaheuristic optimization.*

Identifier	Components c	Runnables r	Chains g	Nodes n
$c_6 g_{10} n_4$	6	60	10	4
$c_8 g_{20} n_6$	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 13.10: *Specifications of optimization problems.*

respect to processor speed, failure rate and power consumption. The problems are identified by handlers of type $\langle c_i g_j n_i \rangle$ for readability, where the c, g, n variables denote number of components, cause-effect chains and computing nodes, respectively. The $c_6 g_{10} n_4$ and $c_8 g_{20} n_6$ problems correspond to Spec-I, thus represent the allocation of small size software applications. The $c_{10} g_{20} n_8$ problem is based on Spec-II, thus denote the allocation of medium size software applications, and the $c_{20} g_{30} n_{10}$, $c_{50} g_{40} n_{20}$ and $c_{80} g_{60} n_{20}$ are based on Spec-III, thus denote the allocation of large size software applications. The optimization problems are executed each $30\times$ using our ILP method proposed in [35] and the meta-heuristic algorithms presented in Section 13.5. The optimization parameters such as the penalty function coefficients and the meta-heuristic parameters control the optimization, and their settings are shown in Table 13.9. The optimization parameter values are obtained from literature as well as from our experimentation. Subsequently, we recorded the computation time, fitness values and power consumption delivered by each algorithm.

Experiment 2

Usually the replication exerts heavy computation over the calculations of the cause-effect delay due its combinatorial nature. The approximation technique,

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 13.11: *Specifications of chains g and degrees of replication d, used in Experiment 2.*

which is presented in Subsection 13.4.4 optimizes the calculations of the cause-effect chain delay 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 13.11. The degree of replication indicates the multiplicity of each component in the software applications.

13.6.2 Analysis of Experiment 1

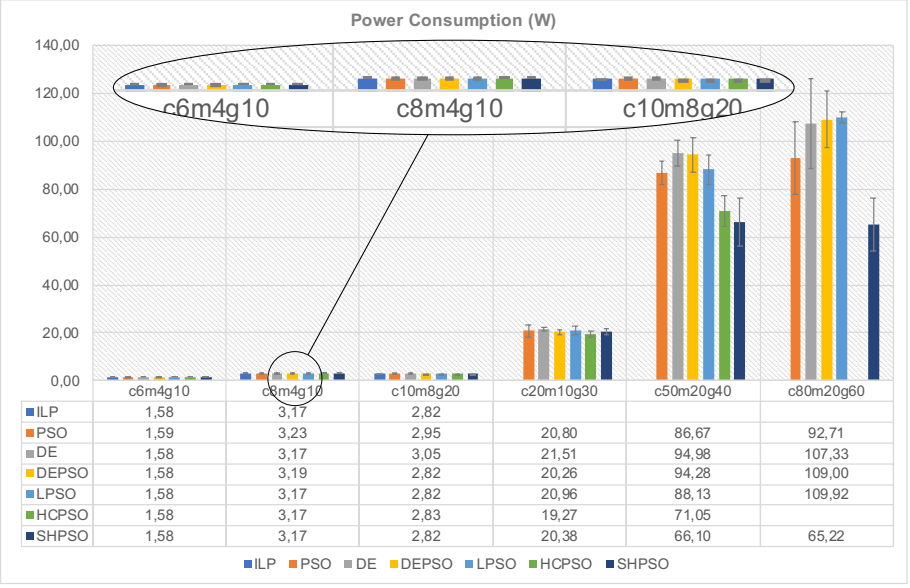
Table 13.12 shows a summary of the evaluation results from executing Experiment 1 such as the average and standard deviation of the computation times and fitness values, and the quality of solutions. The latter is calculated as a percentage of solution deviation from the best solution per optimization problem, which is indicated by the **boldface** type. In the first three optimization problems, the ILP method returns the best (or optimal) solution. The SHPSO algorithm returns the best solution for the $c_{20}g_{30}n_{10}$ and $c_{50}g_{40}n_{20}$ problems, and SHPSO for the last problem $c_{80}g_{60}n_{20}$. We analyze the results over three matrices: solution quality, computation time, and stability.

Solution Quality

In the $c_6g_{10}n_4$ optimization, all except DEPSO and PSO returned the optimal power consumption 227KW. DEPSO and PSO returned near optimal solutions, with $> 99\%$ quality measures (or optimality) as compared to ILP. In the $c_8g_{20}n_6$ optimization, ILP, HCPSO and SHPSOreturned optimal solutions. Whereas DE, LPSO and DEPSO performed worse than ILPby 1% but better than PSO by 2%. In the $c_{10}g_{20}n_8$ optimization, only ILPreturned optimal solution whereas the hybrid algorithms returned worse and PSOand DEreturned the worst. 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 extremely large computation time, as a result, it was terminated manually. However, the hybrid algorithms based on hill-climbing such as HCPSO and SHPSO performed well, which is followed by DEPSO in the

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

$c_{20}g_{30}n_{10}$ and $c_{80}g_{60}n_{20}$ optimization. HCPSO failed to return solution in the largest optimization problem $c_{80}g_{60}n_{20}$, however, the stochastic version of the hill-climbing algorithm SHPSO returned a near optimal solution.



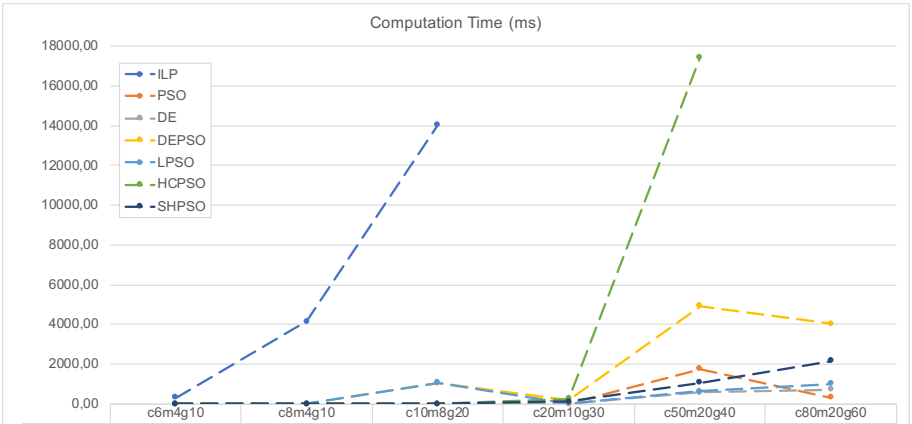
	c6m4g10	c8m4g10	c10m8g20	c20m10g30	c50m20g40	c80m20g60
ILP	1,58	3,17	2,82			
PSO	1,59	3,23	2,95	20,80	86,67	92,71
DE	1,58	3,17	3,05	21,51	94,98	107,33
DEPSO	1,58	3,19	2,82	20,26	94,28	109,00
LPSO	1,58	3,17	2,82	20,96	88,13	109,92
HCPSO	1,58	3,17	2,83	19,27	71,05	
SHPSO	1,58	3,17	2,82	20,38	66,10	65,22

Computation Time

In this context, the computation time is the elapsed time between the start of the optimization and the time at which the fitness value becomes steady, and evaluated for 5000 iterations (or generations). In a way, the computation time also indicates the convergence speed of the best solution. Figure 13.11 summarizes the computation time of the algorithms for the optimization problems listed in Table 13.11. The computation time of ILPis extremely higher than the rest, which is in the scale of milliseconds for the $c_6g_{10}n_4$ sample, and in seconds for the $c_8g_{20}n_6$ and $c_{10}g_{20}n_8$. In the case of the non-hybrid meta-heuristic algorithms, the computation time is in the scale of milliseconds in all of the optimization problems. The computation time of the hybrid PSO with the hill-climbing algorithm HCPSO got exponential in $c_{50}g_{20}n_{40}$ and returns no solution in the next largest problem. However, the stochastic version of the hill-climbing algorithm returned near optimal solution in $2sec$ while performing better than the hybrid PSO with the differential algorithm DEPSO, which returned in $\leq 6sec$.

Solutions Stability

The PSO and DE are characterized by random search that facilitates exploration of the search space. However, the randomness also introduces instability of the solutions, that is when the algorithms are executed for the same input, the solutions vary. Such variations are depends on the nature of



	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 13.11: Computation time of the various algorithms for solving different instances of the software allocation problem. **Color print should be used!**

the algorithms as well as on the optimization problems at hand. In this work, we use the standard deviation to measure the degree of stability, of the meta-heuristic algorithms. Figure 13.12 and Figure **??** show the deviation of each algorithms from the (near) optimal solutions for the different samples.

Regarding quality of the solutions, the results showed that HCPSO is more stable in the first three optimization problems, similarly DE and LPSO in the first problem. However, as the problem size increases to $g_{10}d_{20}n_8$, the HCPSO performed worse while PSO and others improve. Regarding computation time, the algorithms become less stable as the problem size increased uniformly for PSO, DE, HCPSO and SHPSO, however, it is not the case for rest of algorithms.

13.6.3 Analysis of Experiment 2

Figure 13.12 shows results of executing experiment 2, which shows improvements of the computation time by applying the approximation algorithm instead of the exact approach. In the case of the approximation, the delays are exhaustively calculated in the presence of replication. However, the quality of the solutions are degraded as expected due to the approximation. Specifically, the results show $61\% - 81\%$ computation time improvement over the exact method while facing quality degradation only for samples $g_{30}d_3$ and $g_{60}d_2$. 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

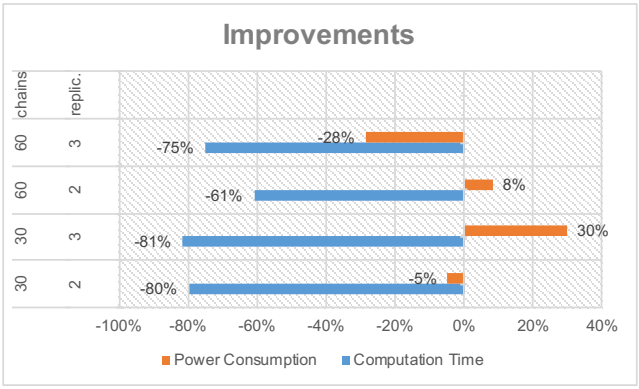


Figure 13.12: Effect of approximate algorithm over delay calculations with replication. **Color print should be used!**

iterations, the commutative effect of the algorithms can negatively impact the responsiveness to engineers. Thus, the improvements can be in trade-off with optimality of the solutions.

13.6.4 Discussion

The presented software allocation problem considers the complex AUTOSAR system model, which contains AUTOSAR software applications and a network of heterogeneous computing nodes with respect to power consumption, failure rate and processor speed. The software applications have end-to-end timing and reliability requirements, which are satisfied by effectively and efficiently mapping the software components of the applications to the computing nodes. We assume the worst-case response time analysis to check the schedulability of the tasks, and age delay analysis to check the schedulability of chains.

The allocation problem is even more complex since we apply fault tolerance to maximize and subsequently meet the reliability requirements of the applications. The fault tolerance, which is realized by replicating software components, imposes heavy computation on the age delay of the chains. The reliability is computed using state enumeration, which is an exact method, as compared to series-parallel method due to functional inter-dependency of computing nodes.

Considering the complexity of the system model, the exact method based on the integer-linear programming is limited only to small and medium software allocation problems as shown in Experiment 1. Although, the meta-heuristic algorithms did not always return the optimal solutions, the results are actually very close to the results obtained from ILP, and the computation time is quite low as compared to the ILP results.

In terms of optimality, the stochastic hill-climbing performed best next to ILP which is attributed to the intensive local search of the algorithm. However, it felt short to return near optimal solutions in the largest optimization problem, which is not the case for its stochastic version. Thus, the hybrid PSO with the stochastic hill-climbing algorithm relatively more robust, effective and scales well as compared to the rest of the hybridization algorithms.

13.7 Related Work

In this section, we discuss related work on the software allocation in the context of resource consumption and improving reliability of distributed real-time systems. In a heterogeneous distributed system where computing nodes and communications links could have different failure rates, a reliability-aware allocation of tasks to nodes, and using nodes with the lowest failure rates can noticeably improve the system reliability [50][28][62][63]. Interleaving real-time constraints into the problem adds more complexity to reliability-aware task allocation in distributed systems [20].

In the AUTOSAR environment, many authors proposed mapping of runnables to nodes [20, 60, 57]. In contrast, but similar to [31, 43, 52], we propose mappings of software components, which give better scalability since multiple runnables that are co-hosted in a single component are always mapped to a single node, hence less decision variables in the optimization problem. Likewise, Svagor et al. [52] propose a genetic algorithm for a multi-criteria allocation of software components onto heterogeneous nodes that consist of CPUs, GPUs and FPGAs, however, the approach does not consider task level timing specifications. In contrast, our software components mapping strategy utilizes the timing specifications of tasks to improve accuracy of the allocation.

Different cost functions are considered in several software allocation problems, e.g., Assayad et al [4] propose a heuristic algorithm to maximize reliability of a distributed system using task replication while minimizing the makespan of the given taskset, likewise, Wenhao et al. [52] a Genetic algorithm to multiple metaheuristic algorithms to minimize a weighted value of the communication overhead and CPU utilization, Zheng et al [43] a heuristic algorithm (i.e., Clustering Algorithm based on Traffic) to minimize communication overhead of runnables while meeting timing constraints, Yin et al. [62] hybrid particle swarm optimization to maximize reliability. In this work, we minimize power consumption while meeting reliability and timing constraints with applications with different criticality. Some related work consider end-to-end timing analysis using the holistic response time analysis, however, if freshness of data is considered, the sort of delay semantics such as the age delay should give exact results. In contrast to the work by Assayad et al. [4], which use the Minimal Cut Sets method, i.e., an approximate

algorithm, to calculate reliability of a system, we apply an exact method based using state enumeration, albeit exhaustive. As opposed to [60][45], we assume that software applications are multi-rate, i.e., the tasks execute with different sampling rates, and triggering patters per chain consist of one more independently triggered tasks, which increase the difficulty of software allocation due the complexity of their timing analysis.

In contrast to other related work, we consider software component allocation by efficiently utilizing low level timing information both at tasks level and end-to-end chains of tasks. In the case of end-to-end delay calculation, we use a more accurate semantics, i.e., the age delay, which is more accurate especially if freshness of data is required, which is the case in many control applications, in communications that use single-buffer register schemes. To solve the software allocation problem, we propose several meta-heuristic algorithms such as differential evolution, particle swarm optimization and its hybrids with the former and with other local search algorithms such as hill-climbing and stochastic hill-climbing algorithms. The results from meta-heuristic algorithms, i.e., for small and medium optimization problems, are compared to results from the ILP approach, which is introduced in our previous work [35]. The evaluation of our proposed solutions are validated on specifications that are extrapolated from the engine management system automotive benchmark, which consist of realistic/industrial data from Bosch.

13.8 Conclusions and Future Work

We presented safety-critical software allocation on a network of heterogeneous computing nodes, with respect to failure rate, processor speed and power specification. The applications are developed according to the AUTOSAR standard and possess timing and reliability requirements. We assume worst-case response time analysis and age delay analysis to compute the schedulability of tasks and chains, respectively, which are exact but complex timing analysis techniques. Furthermore, the allocation considers maximization of reliability to meet the reliability requirements of the safety-critical applications via fault tolerance. The latter exerts computational overhead especially on the delay calculation, and requires more computational resources and consumes more power.

We proposed hybrid particle-swarm optimization algorithms to optimize the power total consumption of the distributed safety-critical software applications while meeting the requirements. The hybridization algorithms comprise differential evolution, hill climbing and stochastic hill climbing. For comparison, we also included differential evolution and local particle-swarm optimization algorithms. The result of the algorithms are compared to ILP results in the small and medium software allocations. In general, the

the hybridization with the hill-climbing algorithms performed better than the other meta-heuristic algorithms. The hybridization with the stochastic hill-climbing performed well in the largest optimization problem.

In future work, we plan to consider a complex power model that relates load to heat dissipation and the effect on reliability over a long run. The current work is limited to offline configuration, however, it can be extended to address the need for re-configurable distributed system, e.g., in the case of software evolution, system failures .

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.

13.9 References

[1] *Handbook of Metaheuristics*. 2006.

[2] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No.98CB36279)*, pages 4–13. IEEE Comput. Soc.

[3] Mohammad Ashjaei, Nima Khalilzad, Saad Mubeen, Moris Behnam, Ingo Sander, Luis Almeida, and Thomas Nolte. Designing end-to-end resource reservations in predictable distributed embedded systems. *Real-Time Systems*, 2017.

[4] Ismail Assayad, Alain Girault, and Hamoudi 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*, pages 347–356. IEEE, 2004.

[5] AUTOSAR. Specification of Timing Extensions. Technical report, AUTOSAR, 2017.

[6] AUTOSAR. Specification of Operating System AUTOSAR Release 4.2.2. Technical report, AUTOSAR, 2018.

[7] S. K. Baruah, A. Burns, and R. I. Davis. Response-time analysis for mixed criticality systems. In *Proceedings - Real-Time Systems Symposium*, 2011.

[8] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture*, 2017.

[9] Hax Bradley. *Applied Mathematical Programming*. Addison-Wesley, 1977.

[10] Alessio Bucaioni, Lorenzo Addazi, Antonio Cicchetti, Federico Ciccozzi, Romina Eramo, Saad Mubeen, and Mikael Sjodin. MoVES: A Model-driven Methodology for Vehicular Embedded Systems. *IEEE Access*, 6:6424–6445, 2018.

[11] M Clerc. Discrete particle swarm optimization, illustrated by the traveling salesman problem. In *New optimization techniques in engineering*, 2000.

[12] Swagatam Das, Sankha Subhra Mullick, and P.N. Suganthan. Recent advances in differential evolution – An updated survey. *Swarm and Evolutionary Computation*, 27:1–30, 4 2016.

[13] Bharat P. Dave, Ganesh Lakshminarayana, and Niraj K. Jha. COSYN. In *Proceedings of the 34th annual conference on Design automation conference - DAC '97*, pages 703–708, New York, New York, USA, 1997. ACM Press.

[14] Robert I. Davis, Alan Burns, Reinder J. Bril, and Johan J. Lukkien. Controller Area Network (CAN) schedulability analysis: Refuted, revisited and revised, 2007.

[15] Srinivas Devadas and Sharad Malik. A survey of optimization techniques targeting low power VLSI circuits. In *Proceedings of the 32nd ACM/IEEE conference on Design automation conference - DAC '95*, pages 242–247, New York, New York, USA, 1995. ACM Press.

[16] Elena Dubrova. *Fault-Tolerant Design*. Springer New York, New York, NY, 2013.

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

[18] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power Provisioning for a Warehouse-sized Computer. *ACM SIGARCH Computer Architecture News*, 35(2):13, 2007.

[19] Hamid Reza faragardi, Björn Lisper, Kristian Sandström, and Thomas Nolte. A Resource Efficient Framework to Run Automotive Embedded Software on Multi-core ECUs. *Journal of Systems and Software*, 139:64–83, 2018.

[20] Hamid Reza Faragardi, Reza Shojaee, Mohammad Amin Keshtkar, and Hamid Tabani. Optimal Task Allocation for Maximizing Reliability in Distributed Real-time Systems. In *Computer and Information Science (ICIS), 2013 IEEE/ACIS 12th International Conference On*, pages 513–519. IEEE, 2013.

[21] Hamid Reza Faragardi, Maryam Vahabi, Hossein Fotouhi, Thomas Nolte, and Thomas 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.

[22] Nico Feiertag, Kai Richter, Johan Nordlander, and Jan 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.

[23] David Fernández-Baca. Allocating Modules to Processors in a Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1427–1436, 11 1989.

[24] A.L. Goel. Software Reliability Models: Assumptions, Limitations, and Applicability. *IEEE Transactions on Software Engineering*, SE-11(12):1411–1423, 12 1985.

[25] Michel Goraczko, Jie Liu, Dimitrios Lymberopoulos, Slobodan Matic, Bodhi Priyantha, and Feng Zhao. Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems. In *Proceedings of the 45th annual conference on Design automation - DAC '08*, page 191, New York, New York, USA, 2008. ACM Press.

[26] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Bbenchmark Suite. In *2001 IEEE International Workshop on Workload Characterization, WWC 2001*, pages 3–14, 2001.

[27] R. Inam, N. Mahmud, M. Behnam, T. Nolte, and M. Sjödin. The Multi-Resource Server for predictable execution on multi-core platforms. In *Real-Time Technology and Applications - Proceedings*, volume 2014-Octob, 2014.

[28] S Kartik and C Siva Ram Murthy. Task Allocation Algorithms for Maximizing Reliability of Distributed Computing Systems. *IEEE Transactions on computers*, 46(6):719–724, 1997.

[29] J. Kennedy, R. Eberhart, Carlos A Coello Coello, Gregorio Toscano Pulido, Maximino Salazar Lechuga, J. Kennedy, R. Eberhart, Carlos A Coello Coello, Gregorio Toscano Pulido, Maximino Salazar Lechuga, and From Scholarpedia. Particle swarm optimization. *Neural Networks, 1995. Proceedings., IEEE International Conference on*, 1995.

[30] Bart Kienhuis, Ed F Deprettere, Pieter van der Wolf, and Kees Vissers. A Methodology to Design Programmable Embedded Systems. In *Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation — SAMOS*, pages 18–37. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[31] Junsung Kim, Gaurav Bhatia, Ragunathan Raj Rajkumar, and Markus Jochim. An Autosar-compliant Automotive Platform for Mmeeting Reliability and Timing Constraints. Technical report, SAE Technical Paper, 2011.

[32] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real World Automotive Benchmarks for Free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2015.

[33] Qunfeng Liu, Wenhong Wei, Huaqiang Yuan, Zhi Hui Zhan, and Yun Li. Topology selection for particle swarm optimization. *Information Sciences*, 2016.

[34] Corinne Lucet and Jean-François Manouvrier. Exact Methods to Compute Network Reliability. In *Statistical and Probabilistic Models in Reliability*, pages 279–294. Birkhäuser Boston, Boston, MA, 1999.

[35] Nesredin Mahmud, Guillermo Rodriguez-Navas, Hamid Reza Faragardi, Saad Mubeen, and Cristina Seceleanu. Power-aware Allocation of Fault-tolerant Multi-rate AUTOSAR Applications. In *25th Asia-Pacific Software Engineering Conference*, 12 2018.

[36] Seyedali Mirjalili. Particle swarm optimisation. In *Studies in Computational Intelligence*. 2019.

[37] B. Moyer. Low-power design for embedded processors. *Proceedings of the IEEE*, 89(11):1576–1587, 2001.

[38] Saad Mubeen, Jukka Mäki-Turja, and Mikael 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):453–482, 2013.

[39] Saad Mubeen, Thomas Nolte, Mikael Sjödin, John Lundbäck, and Kurt-Lennart Lundbäck. Supporting timing analysis of vehicular embedded systems through the refinement of timing constraints. *Software & Systems Modeling*, 18(1):39–69, 2 2019.

[40] Nico Naumann. AUTOSAR Runtime Environment and Virtual Function Bus. *Hasso-Plattner-Institut, Tech. Rep*.

[41] Ibrahim H. Osman and Gilbert Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 2005.

[42] Riccardo Poli. An Analysis of Publications on Particle Swarm Optimisation Applications. *Journal of Artificial Evolution and Applications*, 2008.

[43] Zheng Ran, Hua Yan, Huimin Zhang, and Yun Li. Approximate optimal AUTOSAR software components deploying approach for automotive E/E system. *International Journal of Automotive Technology*, 18(6):1109–1119, 12 2017.

[44] Dian Palupi Rini and Siti Mariyam Shamsuddin. Particle Swarm Optimization: Technique, System and Challenges. *International Journal of Applied Information Systems*, 2011.

[45] Salah Eddine Saidi, Sylvain Cotard, Khaled Chaaban, and Kevin 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*, pages 1–8, New York, USA, 2015. ACM Press.

[46] Andreas Sailer, Stefan Schmidhuber, Michael Deubzer, Martin Alfranseder, Matthias Mucha, and Juergen Mottok. Optimizing the Task Allocation Step for Multi-Core Processors within AUTOSAR. In *2013 INTERNATIONAL CONFERENCE ON APPLIED ELECTRONICS (AE)*, 2013.

[47] Alberto Sangiovanni-Vincentelli, Luca Carloni, Fernando De Bernardinis, and Marco Sgroi. Benefits and Challenges for Platform-based Design. In *Proceedings of the 41st annual conference on Design automation - DAC '04*, page 409, New York, USA, 2004. ACM Press.

[48] Saptarshi Sengupta, Sanchita Basak, Richard Alan, and Peters Ii. Particle Swarm Optimization: A survey of historical and recent developments with hybridization perspectives. *arXiv:1804.05319*, 2018.

[49] Lui Sha, Tarek Abdelzaher, Karl Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K. Mok. Real time scheduling theory: A historical perspective, 2004.

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

[51] Rainer Storn and Kenneth Price. Differential Evolution - A Simple and Efficient Heuristic for Global Optimization over Continuous Spaces. *Journal of Global Optimization*, 1997.

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

[53] El Ghazali Talbi. *Metaheuristics: From Design to Implementation*. 2009.

[54] M Fatih Tasgetiren, Mehmet Sevkli, Yun-Chia Liang, and M Mutlu Yenisey. A particle swarm optimization and differential evolution algorithms for job shop scheduling problem. *International Journal of Operations Research*, 3(2):120–135, 2006.

[55] Steve Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings - Real-Time Systems Symposium*, 2007.

[56] Luc Vinet and Alexei Zhedanov. A "Missing" Family of Classical Orthogonal Polynomials. *Computers as Components*, page 528, 11 2010.

[57] Wenhao Wang, Sylvain Cotard, Fabrice Gravez, Yael Chambrin, and Benoit Miramond. Optimizing Application Distribution on Multi-Core Systems within AUTOSAR. 1 2016.

[58] Xi Wang, Imen Khemaissia, Mohamed Khalgui, ZhiWu Li, Olfa Mosbahi, and MengChu Zhou. Dynamic Low-power Reconfiguration of Real-time Systems with Periodic and Probabilistic Tasks. *IEEE Transactions on Automation Science and Engineering*, 12(1):258–271, 2015.

[59] W. Wolf. A Decade of Hardware/ Software Codesign. *Computer*, 36(4):38–43, 4 2003.

[60] Ernest Wozniak, Asma Mehiaoui, Chokri Mraidha, Sara Tucci-Piergiovanni, and Sébastien Gerard. An Optimization Approach for the Synthesis of AUTOSAR Architectures. In *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2013.

[61] Jigang Wu, Thambipillai Srikanthan, and Chengbin Yan. Algorithmic aspects for power-efficient hardware/software partitioning. *Mathematics and Computers in Simulation*, 79(4):1204–1215, 12 2008.

[62] Peng Yeng Yin, Shiuh Sheng Yu, Pei Pei Wang, and Yi Te Wang. Task allocation for maximizing reliability of a distributed system using hybrid particle swarm optimization. *Journal of Systems and Software*, 80(5):724–735, 2007.

[63] Longxin Zhang, Kenli Li, Yuming Xu, Jing Mei, Fan Zhang, and Keqin Li. Maximizing Reliability with Energy Conservation for Parallel Task Scheduling in a Heterogeneous Cluster. *Information Sciences*, 319:113–131, 2015.

[64] Fuqing Zhao, Yi Hong, Dongmei Yu, Yahong Yang, Qiuyu Zhang, and Huawei Yi. A hybrid algorithm based on particle swarm optimization and simulated annealing to holon task allocation for holonic manufacturing system. *International Journal of Advanced Manufacturing Technology*, 2007.