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

Abstract

Software-to-hardware allocation plays an important role in the development of resource-constrained automotive embedded systems that are required to meet timing, reliability and power requirements. This paper proposes an Integer Linear Programming optimization approach for the allocation of fault-tolerant embedded software applications that are developed using the AUTOSAR standard. The allocation takes into account the timing and reliability requirements of the multirate software applications and the heterogeneity of their execution platforms. The optimization objective is to minimize the total power consumption of the applications that are distributed over multiple computing units. The proposed approach is evaluated using a range of different software applications from the automotive domain, which are generated using the real-world automotive benchmark. The evaluation results indicate that our proposed allocation approach is effective while meeting the timing, reliability, and power requirements of the considered automotive software applications.

1. Introduction

The software-to-hardware allocation is a very important step during the development of automotive embedded systems. Basically, it allows the designer to explore system-level solutions that meet functional and extra-functional software requirements together with resource availability on the execution platform. Software allocation is a well-researched area in the domain of embedded systems, including in hardware/software co-design [1], platform-based system design [2] and the Y-chart design approach [3]. It is a type of bin-packing problem, and therefore finding an optimal solution, in the general case, is NP-hard [4]. The methods to solve such problems can be exact [5], which means solutions are guaranteed to be optimal, or heuristic, which deliver near-optimal solutions [6][7]. Exact methods such as Integer Linear Programming (ILP) [8] have been used widely in several resource optimization problems. In contrast to heuristic methods, ILP returns optimal solutions faster for relatively small problems [9]. However, many problems in real-time systems are nonlinear by nature [10], e.g., response time of cause-effect actions, system reliability, etc. To benefit from linear optimization techniques, non-linear functions are approximated using *Linearization* - a widely-used technique in the optimization of non-linear problems.

Preprint submitted to Elsevier *February 14, 2019*

In case of fault-tolerance with replication [11], the search space to find the optimal allocation is increased due to the replicas. The search space becomes even larger if we assume that the real-time system executes over different sampling rates, known as *multirate* [12], in which case the feasible (timed) paths that pass through the different sampling points (or activation patterns) increase exponentially with the number of activation patterns increase . Furthermore, due to the different sampling rates that result in oversampling and undersampling effects, the timing analysis of signals propagation is complex [13]. Existing methods of software allocation lack exact results for the timing analysis of multirate systems.

In this paper, we propose an allocation scheme based on ILP for relatively small- and medium-sized fault-tolerant distributed applications, with the number of allocatable components not exceeding 15, operating-system tasks less than 100, and cause-effect chains in the range of 30 to 60. These parameters are deducted from the real-world automotive benchmark [14], and from previous experience in developing automotive systems and experiments. The applications are distributed over heterogeneous computing units that share a single network. The allocation aims for minimizing the total power consumption of the system while meeting timing and reliability requirements. Our proposed solution targets the automotive domain, in particular systems that conform to the AUTomotive Open System ARchitecture (AUTOSAR) standard. In comparison to related work [9][15][5], we consider a fault-tolerant and multirate system model. Furthermore, we follow a highly integrated approach in the allocation process, which includes response-time analysis (RTA), and utilization bound checking (UB), as well as bounding the level of fault tolerance via the imposed reliability requirement on the application. The main contributions of our work are: i) an ILP model for the allocation of a fault-tolerant multirate application on heterogeneous nodes with the objective of minimizing the total power consumption, and ii) an approach for reducing overhead of replications and cause-effect chains on the allocation of such applications.

Our approach is evaluated on synthetic automotive applications that are generated according to the real-world automotive benchmark proposed by Kramer et al. [14]. In the evaluation, we show the performance of our proposed approach in terms of allocation time and resource efficiency with respect to the size of applications. The tool and the synthetic applications used in this experiment are publicly available from BitBucket[1].

The rest of the paper is organized as follows. Section 2 provides a brief overview of AUTOSAR-based software development, emphasizing the role of software allocation. Section 3 describes the system model, and Section **??** describes the extra-functional models including the timing, reliability, and power consumption models. Section 4.7 presents the proposed allocation scheme, and in Section 6, we provide an evaluation of the proposed approach using the automotive benchmark. In Section 8, we compare to related work. Finally,

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

we conclude the paper in Section 9, and outline the possible future work.

2. AUTOSAR

The AUTomotive Open System ARchitecture (AUTOSAR) partnership has defined the open standard AUTOSAR for automotive software architecture that enables manufacturers, suppliers, and tool developers to adopt shared development specifications, while allowing sufficient space for competitiveness. The specifications state standards and development methodologies on how to manage the growing complexity of Electronic/Electrical (E/E) systems, which take into account the flexibility of software development, portability of software applications, dependability, efficiency, etc., of automotive solutions. The conceptual separation of software applications from their infrastructure (or execution platform) is an important attribute of AUTOSAR and is realized through different functional abstractions [16].

2.1. Software Application

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

2.2. Timing and Reliability of Applications

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

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

3. System Model

The system model shown in Figure 1 is captured as 5-tuple $\langle A, N, \mapsto, Req, Pro \rangle$, where $A = \{A_k : k = 1, \ldots, n_A\}$ denotes software applications, $N = \{n_i : i = 1, \ldots, n_N\}$ heterogenous computation (or computing nodes) $\mapsto: A \to N$ maps the software applications to at least a single computing node, and Req, Pro are assignment functions, respectively defines the requirements of the applications and resource provisions of the computing nodes. The function $Req : A \to (\mathrm{RL}, \mathrm{EE}, \mathrm{CL}) : \mathbb{R}^+ \times [0, 1] \times \mathbb{I}^+$ a, assigns the reliability, end-to-end timing and criticality level of applications. The function $Pro : N \to (\mathrm{HZ}, \mathrm{PW}, \mathrm{FR}) : \mathbb{R}^+ \times \mathbb{R}^+ \times \mathbb{R}^+$ assigns, respectively the processor speed, power consumption and failure rate of the computing nodes.

The end-to-end timing requirements are the timing constraints over the end-to-end functional behaviors of the applications. These requirements are often specified on the *cause-effect chains* consisting of software components and (potentially network messages) within the application. The reliability requirement is the expected reliability goal of an application which is discussed further in Subsection 4.3, and the criticality level signifies the importance of an application over other applications that have lower criticality levels, thus prioritizing the application during resource contention. The criticality levels are defined systematically, e.g., following the hazard analysis according to the ISO 26262 standard for functional safety in road vehicles [18].

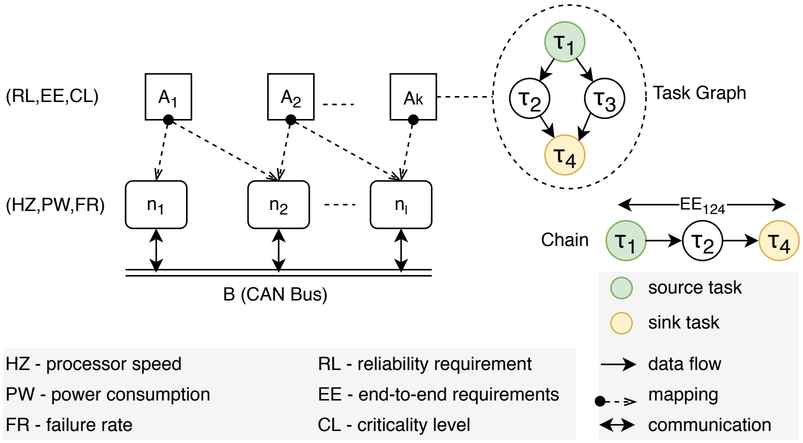


Figure 1: System Model.

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

*Note: the total elements in a set S is denoted by n_S, e.g., n_A denotes the number of applications in the set S, essentially it refers to its cardinality.

** For readability, we prefer to use $S_i^{(k)}$ in place of $S_i^{(A_k)}$.

*** IFor other uses of the iterators, they are defined in the context.

3.1. Software Applications

The software application represents an independent and self-contained user-defined software functionality, e.g., x-by-wire, electronic throttle control, flight control, etc. The application is assumed to be developed using the principles of model-based development and component-based software engineering [19][20].

Definition 1 (Software Application Model). It is modeled as a set of communicating software components. Note that a software component is a design-time concept, representing the lowest-level hierarchical element in software architecture of the application. The set of software components in the application is modeled as a *directed acyclic vertex-weighted* graph $\langle V_\tau, L_\tau, w \rangle$ of periodic task nodes V_τ, where $a_{ij} \in L_\tau$ refers to the data-flow link from node τ_i to node τ_j and $i \neq j$. The function $w : V_\tau \rightarrow (e_{n_i}, \mathrm{D}, \mathrm{P})$ assigns computation cost to each task node, where e_m is worst-case execution tim (WCET) on the computing node n_i, D deadline and P period.

Multiple applications can be executed on the same computating node(s) and can share the on-board network, e.g., the CAN bus. Since the applications can have different criticality requirements, the computing nodes and the network should provide an isolation mechanism in order to avoid interference of lower-criticality applications on higher-criticality applications. The isolation of different criticality applications on the same execution platforms is important in the so-called mixed-criticality systems [21], which can be already found in the avionics domain and also trending in other domains such as the automotive domain where safety-critical applications, such as x-by-wire and electronic throttle control systems, are required to be consolidated with the infotainment system on the same ECUs [**?**].

3.2. Scheduling Software Applications

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

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

3.2.1. Scheduling Tasks and Messages

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

$$\forall \tau_1 \in V(g_\tau^{(A_i)})\tau_2 \in V(g_\tau^{(A_j)})\ [cri(A_i) > cri(A_j) \implies Pri(\tau_1) > Pri(\tau_2)],$$

where cri, pri are predicates which detemine the critiality and priority of tasks τ_1, τ_2, respectively; $V(g_\tau^{(A_i)}), V(g_\tau^{(A_j)})$ are the set of tasks which implement the applications A_i, A_j, respectively.

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

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

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

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

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

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

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

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

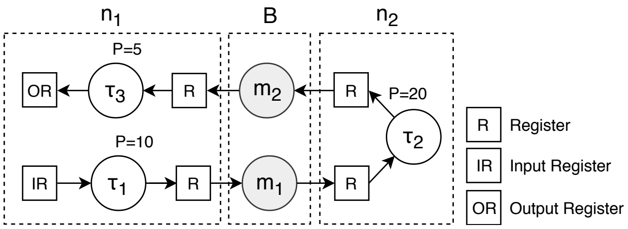


Figure 2: A Cause-effect Chain, mapped on nodes n_1 and n_2.

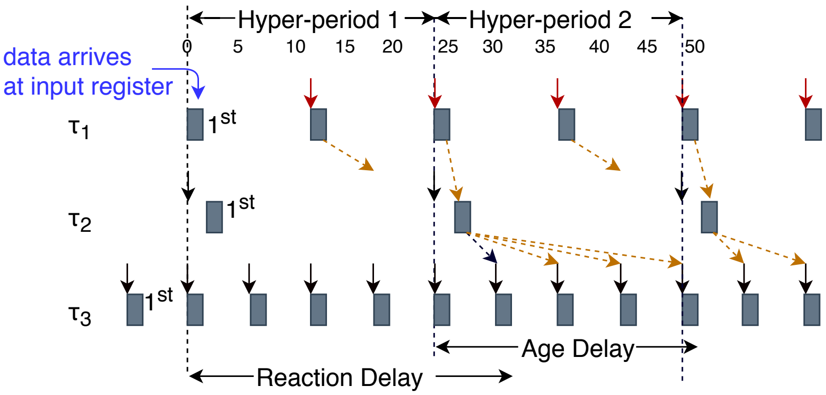


Figure 3: Reaction and Age Delays of the Cause-effect Chain, Shown in Figure 2.

3.2.2. Scheduling Cause-effect Chains

The software application can be considered as a set of *cause-effect chains* $\Gamma^{(k)} = \{\Gamma_i^{(k)} : i = 1, \ldots, n_\Gamma\}$, which are directed paths in the graph, annotated by end-to-end timing constraints. They represent sequences of actions triggered usually by external events (or causal actions or stimuli) and produce corresponding effects (or responses), e.g., pressing a rotary-wheel to activate a cruise control system, pressing a brake pedal to slow down a car, etc. The end-to-end requirements put upper-bounds on the duration of the stimuli-response elapse time. An example of a cause-effect chain is shown in Figure 2, which consists of three independently clocked tasks τ_1, τ_2, τ_3, and messages m_1, m_2. It uses single-register buffers for communication, which is a common practice in control systems design, e.g., automotive software applications [26].

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

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

$$\Delta_s(\Gamma) = \alpha_{\tau_j} - \alpha_{\tau_i} + R_{\tau_j} \qquad \text{single node} \qquad (5)$$

$$\Delta_m(\Gamma) = \sum_{q \in subch(\Gamma)} \Delta_s(q) + \sum_{m \in msg(\Gamma)} R_m, \qquad \text{muliple nodes} \qquad (6)$$

where $\tau_j = sink(\Gamma)$, $\tau_i = source(\Gamma)$, α_τ returns the activation time of τ.

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

3.3. AUTOSAR Runnables

The AUTOSAR standard introduced the notion of *Runnables* to facilitate early analysis, that is at the VBF level, and to support interoperability of automotive applications across different execution platforms. Basically, runnables are schedulable pieces of codes similar to tasks. In this work, we assume

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

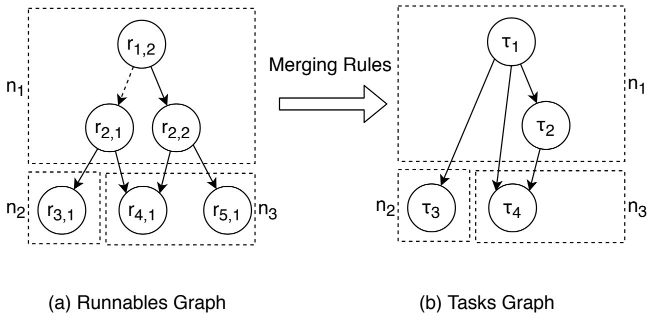
Table 1: Runnables Timing Specifications.

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

Table 2: Tasks-Runnables Mappings.

periodically activated runnables with support for multiple worst-case executions that correspond to the different computation processor types. Unlike tasks, runnables' functional and extra-functional properties, e.g., timing, memory requirements, are part of the AUTOSAR software component specification. Therefore, the software application model defined in Definition 1 is extended to accommodate the notion of runnables, using the following simplified formal definition.

Definition 2 (AUTOSAR Software Application Model). It is modeled as directed acyclic vertex-weighted graph $g_r = \langle V_r, L_r, w_r, v\rangle$ of runnable nodes V_r, where $a_{ij} \in L_r$ represents either a triggering or data-flow link from the runnable r_i to runnable r_j and $i \neq j$. The cost $w_r = w$ defines the timing model of the runnable nodes, which is equivalently defined as the cost of the task nodes explained in Definition 1.



(a) Runnables Graph

(b) Tasks Graph

Figure 4: Example of a Software Application, Modeled as Directed Acyclic Graph, where $\dashrightarrow$, $\dashrightarrow$ denote triggering link, data-flow link, respectively.

According to the AUTOSAR specification [28], runnables are mapped to tasks, and the tasks execute the runnables respecting their timing specifications. In the mapping process, one or more runnables can be merged to optimize the runtime execution by reducing the number of schedulable tasks. Thereore, through the mappings, eventually runnables graphs are refined by tasks graphs as shown in Figure4 (b). In this work, the following rules are applied in order to merge any runnables a, b, that is the link $(a, b) \in L_r$ merges to a task node $v \in V(g_\tau)$, if the following rules satisfy:

(i) the runnables are co-hosted in the same computing node, that is

$$a \mapsto m \land b \mapsto m$$

(ii) activation periods of the runnables are the same, i.e., $a.P = b.P$

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

3.4. Reliability of Software Applications

Redundancy is the most common way to increase the reliability of a system. In this context, *software application reliability* refers to the probability that a software application functions correctly by the time t, or within the time interval $[0, t]$ [**?**]. Redundancy can be implemented according to different schemes, such as hot stand-by, cold stand-by, etc [29], in this work, we consider the hot-standby scheme, where replicated components maintain the same state. However, only the *primary* replicas act on the environment, e.g., activating an actuator. The primary software component is the one in operation and is indicated by $q_{i,1}^{(k)}$, the secondary software component, which is in the stand-by, by $q_{i,2}^{(k)}$, etc., for a software application A_k. Note: the software components are replicated unless the reliability requirements of applications are satisfied.

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

Under this assumptions, the reliability of a software application is equivalent to the reliability of the execuiton platform such as the computating nodes and the communication bus, if any, on wich the application is deployed. The reliability of a computing node or the bus is calculated using $e^{\lambda t}$, where λ is an exponentially distributed failure-rate. However, the reliability calcultion of the execution platform that services a software application is not trivial in the case with replication, e.g., the series-parallel reliability approach cannot be applied in the general case, due to the *functional* interdependency created between computing nodes as the result. To demonstrate the functional interdependency, let us assume a software application A_k, having component configurations with

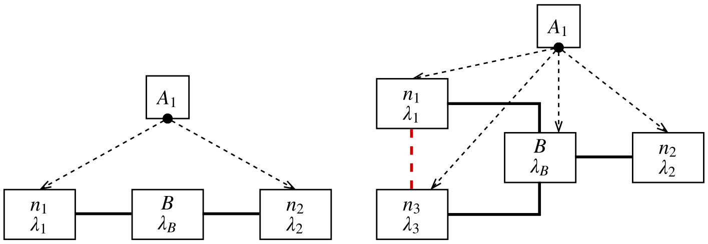
n_1	n_2	n_3
$q_{1,1}$	$q_{2,1}$	
$q_{3,1}$		

(a) Without Replication.

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

(b) With Replication.

Figure 5: Deployments of the Software Application A_1.



(a) Without replication.

(b) With replication.

Figure 6: Reliability Block Diagrams of the Software Application.

and without replication as shown Table 5b and 5a, respectively, where $q_{i,j}^{(A_k)}$ is the j^{th} software-component replica of software-component type $c_i^{(A_k)} \in C_i^{(A_k)}$. Note: for readability of the example, we remove the superscript k.

The reliability of the software application without replication forms a series path, indicated by the reliability block diagram (RBD) of Figure 6a, hence is computed as products of the reliability of n_1, n_2 and B. However, with replication, two computing nodes can form as series and parallel to service the sofware application, e.g., due to $q_{1,1}$ and $q_{2,2}$ or $q_{3,2}$, n_1 and n_3 make series, and due to $q_{3,1}$ and $q_{3,2}$, the nodes make parallel, to relialize partional functionality of the application. In this case, the series-parallel diagram depicted in Figure 6b does not accurately capture the reliaiblity calculation of the application with replication. Note: the red-dashed line between n_1 and n_3 indicates the possibility of the computing nodes becoming series. To address this problem, we use the exact reliability calculation technique based on the enumeration of the different failure states of the computing nodes, that is the different failure states of the execution platform is enumerated exaustively, and subsequently, the total probability the software application functions is computed, which is discussed in great length in Subsection 4.3.

4. Define the Problem of Mapping the Software Applications

The software applications are deployed on the execution platform in such a way that the user-defined requirements of the software applications such as the reliability requirements, end-to-end timing requirements, and criticality of the applications are satisfied. It is conducted by mapping the software-components replicas (or components in short) effectively on the computing nodes and the network bus, that is that is $\bigcup Q_i^{(A_k)} \mapsto M'$, so that the total power consumption $Power(\mathbf{x})$ of the applications is minimized. The latter is achieved by selecting nodes that consume lower power $M' \subseteq M$, provided that the reliability of each application $Reliability_a(\mathbf{x})$, the timing $ResponseTime_\tau(\mathbf{x})$, $Delay_\gamma(\mathbf{x})$, and placement constraints fullfill respective software applications requirements.It is a type of Job Shop scheduling with constraints, as such, it is a discrete optimization problem.

The solution to the allocation problem is represented by a vector-matrix $\mathbf{x} = \{\mathbf{x}^{(A_k)} : k = 1, ..., N_a\}$ as shown in Equation (7), where $\mathbf{x}^{(k)}$ is a matrix of size $n_C \times K$, and $x_{ij}^{(k)} = h \in \{1, \ldots, n_N\}$ denotes the mapping of the software-component replica $q_{i,j}^{(k)}$ to the computating node n_h.

$$\mathbf{x}^{(k)} = \begin{bmatrix} x_{11}^{(k)} & x_{12}^{(k)} & \cdots & x_{1K}^{(k)} \\ x_{21}^{(k)} & x_{22}^{(k)} & \cdots & x_{2K}^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N_c1}^{(k)} & x_{N_c2}^{(k)} & \cdots & x_{N_cK}^{(k)} \end{bmatrix} \tag{7}$$

The formulation of the power consumption and timing depends on the tasks graphs $\bigcup g_\tau^{(A_k)}(\mathbf{x})$, which are a function of the mapping $\mathbf{x}$. However, if the tasks graphs are apriori to the software allocation problem, the formulation of a task graph is not necessary. Note: the latter is true if no runnables from different components map to a single task, in this case, the tasks graphs are independent of $\mathbf{x}$. The task graph formulation is shown in the next subsection.

4.1. Task Graph Formulation

The task graph formulation implies the generation of a task graph, with the cost of the tasks nodes and the links updated for the mapping $\mathbf{x}$. The tasks nodes are updated with the timing information and the mapping to computing nodes, likewise, the links with the mapping on the bus (1 if mapped on the bus, 0 otherwise). The formulation has two steps: i) for a mapping $\mathbf{x}$, we update the runnables with nodes information to which they are mapped by traversing elements of the mapping $\mathbf{x}$, $x_{ij}^{(k)}$, in linear-time complexity $O(n)$, where $n = |\mathbf{x}| * n_r$, and n_r is the runnables per software component using Equation (8); ii) by applying the merging rules, we traverse the runnables graphs in the system and subsequently generate the tasks graphs using Equation (9).

$$\forall k, ij\ \forall r \in V(g^{(A_k)})\ node_r = n_h \qquad \text{, where } h = x_{ij}^{(k)} \tag{8}$$

$$\forall k\ g_r^{(A_k)}(\mathbf{x}) \xrightarrow{\text{Eqn. (13);Merging Rules}} g_\tau^{(A_k)}(\mathbf{x}) \tag{9}$$

where $V(g^{(A_k)})$ returns the runnables nodes of the graph $g^{(A_k)}$.

4.2. Total Power Consumption

There are several models (or techniques) to estimate the power consumption of a computing node. In this work, we use a technique based on processor load (or utilization) to estimate the average power consumption of a computating node. Specifically, we use the linear polynomial model proposed by Fan et al. [30], which is shown in (10). The model states that the power consumption of a node is directly proportional to its load, and is inductively formulated from experimental results:

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

where u is the utilization of a computating node, p_{idle} and p_{busy}, respectively refer to the power consumption measured at minimum and maximum processor loads. Such measurements can be obtained by running performance benchmark suits, e.g., MiBench [31], AutoBench [32], etc.

Subsequently, the power consumption of a node m for a given mapping $\mathbf{x}$ is formulated as follows: i) first using Equations (11-13), by calculating first the node's utilization $U_m(\mathbf{x})$ using Equation (12). The node's utilization is computed from the set of components allocated to it (which are $\forall_{ij} x_{ij} = m$) using Equation (12). And the utilization of a component on the node m is computed from its constituent tasks T_c using Equation (13).

$$nodPow(\mathbf{x}) = f_p(nodUti(\mathbf{x}, m)) \tag{11}$$

$$nodUtil(\mathbf{x}, m) = \sum_k \sum_i \sum_j comUtil(m, c) | x_{ij}^{(k)} = m \quad \text{, where } c = C_i^{(A_k)} \tag{12}$$

$$comUtil(m, c) = \sum_{r \in c.R} \frac{r.e_m}{r.\tau.P}, \tag{13}$$

where $c.R$ is set of runnables in component c, $r.e_m$ and $r.\tau.P$ are the execution of runnable r on node m and its period, respectively.

The applications requirements are modeled as constraints that need to be satisfied in the allocation problem. The constraints formulations are shown in the following subsections, respectively for reliability, timing and other design constraints such as related to runnables-to-tasks merging and replication.

4.3. Software-Applications Reliability Constraints

The applications reliability constraints ensure the mapping $\mathbf{x}$ satisfies the user-defined reliability requirements, that is $\forall k \; Reliability_{A_k}(\mathbf{x}) \leq RelReq_{A_k}$. The reliability is computed from the execution framework that is provided to run the application, which consists of computation nodes $M^{(k)}$ and the shared CAN bus B, as shown in Equation (14). The nodes host the components $C^{(k)}$ and are determined by searching the mapping $\mathbf{x}^{(k)}$ in polynomial time using Equation (15).

$$Reliability_{A_i}(\mathbf{x}) = Reliability_{A_i}(M^{(A_k)}) * Reliability(B) \tag{14}$$

$$M^{(A_k)} = \{e | e \in M \land \forall ij (e = m_h)\}, \text{ where } h = x_{ij}^{(k)} \tag{15}$$

The reliability of the nodes, $Reliability_{A_i}(M^{(A_k)})$ with respect to the application

A_k is calculated using the *state-enumeration* technique [33] as shown in Equations (16). According to the technique, the reliability is basically the total probability that the application A_k *functions* under mutually exclusive and uniformly distributed failure-events of the nodes, represented by $\mathcal{F}^{(A_k)}(M^{(A_k)}) = \{\mathbf{0}, \mathbf{1}\}^{M^{(A_k)}}$, where $\mathbf{0}$ means node fails, $\mathbf{1}$ means node is operational. An event $\omega \in \mathcal{F}_{A_k}(M^{(A_k)})$ represents the states of nodes by a sequence of 0-1 variables $(b_1, ..., b_n)$, where $\omega.b_i$ denote the state of node $m_i \in M^{(A_k)}$.

$$Reliability_{A_i}(M^{(A_k)}) = \sum_{s \in \mathcal{F}^{(A_k)} | g(A_i, s) = 1} probability(s) \tag{16}$$

The fact that an application functions $g(A_k, s)$ is defined via its inverse, which is *software application failure*, deductively as follows:

Definition 3 (Software Application Failure). The application A_k fails in the event $\omega \in \mathcal{F}^{(A_k)}(M^{(A_k)})$ if there exists a component type $c_i^{(k)}$ where all of its replica $Q_i^{(k)}$ *fail*, otherwise, the application functions, as shown in Equation (17). A component replica $q_{ij}^{(A_k)} \in Q_i^{(A_k)}$ fails if the node that host it m_h fails, where $h = x_{ij}^{(k)}$.

$$g(a, \omega) = \begin{cases} \mathbf{0} & \text{if } \exists i \forall j \ (m_h \in M_\omega^{(A_k)}) \\ \mathbf{1} & \text{otherwise} \end{cases} \tag{17}$$

$$M_\omega^{(A_k)} = \{e | e \in M^{(A_k)} \wedge (\omega.b_e = 1)\}, \tag{18}$$

where $M_\omega^{(k)}$ denote the nodes that function at the event ω, and can be found by searching the nodes $M^{(k)}$ with state $\omega.b_e = 1$ in constant time , as show in Equation (19).

The probability that a nodes-failure event ω occurs is computed as a product of the probabilities of its consitutuent elements $\omega.b$ as shown in Equation (**??**).

$$\prod_{(m,b) \in (M,\omega)} \lambda_m * (1 - b) + (1 - \lambda_m) * b \tag{19}$$

where λ_m is the failure-rate of node m.

4.4. Timing constraints

The timing constraints ensure that the applications are schedulable on the execution platform, that is the tasks and the cause-effect chains in each application meet their timing requirements. The schedulability of each task is checked using the worst-case response-time anlaysis presented in Subsection 3.2.1, and for the cause-effect chains using the end-to-end data-propagation delay analysis shown in Subsection 3.2.2.

4.5. Tasks Timing constraints

The tasks timing constraints states that the worst-case response time of each task in the system meets its respective deadlines for a mapping $\mathbf{x}$, that is $\forall k \forall \tau \in V(g_r(\mathbf{x}))$ $ResponseTime(\tau) \leq Deadline(\tau)$, where $V(g_r(\mathbf{x})_i^{(A_k)})$ is the nodes in the tasks graphs. To compute the worst-case response time of the tasks, first we partition the tasks per node, that is tasks mapped to the same node grouped, represented by the T_{n_h}, by traversing the tasks graphs using Equation (20). The complexity of this equation, considering an adjucency matrix tasks graphs representation is linear-time $O(N_a * n^{(A_k)})$, where $n^{(k)}$ is the sum of the order and the size of graph $g_\tau^{(A_k)}(\mathbf{x})$.

$$T_{m_h} = \{e \in V(g_\tau^{(A_k)}(\mathbf{x})) | e.node == m_h, \} \quad \text{where } h = x_{ij}^{(k)} \text{ for all } h = 1, ..., n_N \tag{20}$$

Then, we calculate the response time of each task $\tau \in T_m$ by invoking the response-time analyais formula, and construct the tasks timing constraints as shown in Equation (21).

$$\forall \tau \in T_m \; ResponseTime((\mathbf{x}, \tau) \leq Deadline(\tau) \tag{21}$$

4.6. Cause-effect Chains Timing constraints

For a mapping $\mathbf{x}$, the age delays of cause-effect chains should meet thier respective end-to-end requirements, that is $\forall k \forall ij \; AgeDelay_\Gamma(\mathbf{x}) \leq E2eReq_\Gamma^{(A_k)}$, where $\Gamma \in \Gamma_i^{(A_k)}$. To calculate the age delays, first we identify the messages scheduled by the CAN bus for the mapping $\mathbf{x}$ using Equation (22).

$$M = \{e | \forall (a, b) \in g_\tau(\mathbf{x}) \forall n \in N \; (a \mapsto n \wedge b \mapsto n = false) \implies createMsg(e)\}, \tag{22}$$

where $Period(e) = Period(a)$, that is the message inherits the period of its predecessor (or sender) task. Accordingly, we update only the chains that communicate over the shared CAN bus to incorporate the messages, that is $\Gamma_i^{(A_k)} = \{\tau_1, e^*\}$, where τ_1, τ_2 are $Source(\Gamma_i^{(A_k)}) and Sink(\Gamma_i^{(A_k)})$, respectively, and $e \in V(g_\tau(\mathbf{x})) \cup M$. Then, the cause-effect timing constrains are formulated over the updated list of chains $\Gamma_i^{(A_k)}$ using Equation (23).

$$\forall \gamma \in \Gamma_i^{(A_k)} \; Delay_\gamma(\mathbf{x}) \leq E2eReq_\gamma \tag{23}$$

4.7. Software Allocation Optimization

The software allocation is defined as a single-objective optimization problem. The objective function $Power(\mathbf{x})$ is a cost function which minimizes the total power consumption of the software applications as deployed in the heterogenuous computing nodes, where $\mathbf{x}$ is the decision variable (or solution) of the optimization. The cost function is formulated in Equation 24, with inequality constraints shown by Equation (25, 26,27). The constraints ensure the solution meet the reliability requirements, the tasks deadlines, and the chains end-to-end requirements. Furthermore, the overlapping constrain shown in Equation (28) ensure that replicas are not allocated to the same computing nodes.

$$\min_{\mathbf{x}\in X} \; Power(\mathbf{x}) \qquad \text{subjected to:} \tag{24}$$

$$Reliability_{A_k}(\mathbf{x}) \leq RelReq_{A_k} \qquad \text{forall } k = 1, ..., n_{A_k} \tag{25}$$

$$\forall \tau \in T_{m_h} \; ResponseTime_{\tau}(\mathbf{x}) \leq Deadline_{\tau} \qquad \text{forall } h = 1, ..., n_M \tag{26}$$

$$\forall \gamma \in \Gamma^{(A_k)} \; Delay_{\gamma}(\mathbf{x}) \leq E2eReq_{\gamma} \qquad \text{forall } k = 1, ..., n_A \tag{27}$$

$$\forall k \forall ij \; x_{ij}^{(k)} \neq x_{ij}^{(k')}, \qquad \text{where } k \neq k' = 1, ..., n_{rep} \tag{28}$$

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

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

5. Solution using Hybrid Particle Swarm Optimization (PSO)

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

Metaheuristics does not guarantee optimal solutions, nevertheless, the solutions can be good enough (or acceptable) in practice. Thus, although the power consumption of the applications may not be optimal, the solution can be deemed acceptable. PSO has been applied to solve a wide range of problems, including a task allocation problem [34], and DE is shown to scale well for problems with high dimensions. In fact, PSO and DE are used together for improved performance in several optimization problems [**?**], likewise, PSO is used with local search techniques such as Hill climbing to intensify the search [**?**]. Finally, we evaluate the different meta-heuristic methods based on solution quality and computation time for different software allocation problems.

A meta-heuristic algorithm comprises two major parts: solution representation and an objective function. The solution representation shows the data structure that is used to represent each point in the problem space, and it has a significant impact on the performance of the meta-heuristic algorithm. The fitness function is used to evaluate the quality of candidate solutions based on their fitness to

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

Figure 7: Binary (0-1) Representation.

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

Figure 8: Integer Representation.

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

meet the problem objectives. A solution that delivers lower power consumption and violates less constraints is indicated by a lower fitness value. In the following, we describe the solution representation and the objective function proposed in our solution framework to run the meta-heuristic algorithms.

5.1. Solution Representation

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

In this work, consider the integer representation due to efficient encoding (much fewer variables), and it is computationally more efficient, considering the various operations in the optimization problem, e.g., the binary representation shown in Figure 9 uses a pair of metrices to represent the primary and secondary replicas using 18 binary variables versus only 6 variables used in the case of integer representation shown in Figure 8.

5.2. Fitness Function

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

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

Consequently, the original constrained optimization problem is transformed into unconstrained optimization problem, by extending the objective function $P(\mathbf{x})$ with the constrains, represented by a set of *penalty functions* $\{\phi_{reliability}(\mathbf{x}), \phi_{deadline}(\mathbf{x}), \phi_{e2e}(\mathbf{x}), \phi_{rep}(\mathbf{x})\}$. The first penalty function corresponds to the reliability constraint which returns 0 if the reliability constrain is not violated, otherwise returns a positive number denoting how far the reliability constraint is violated. The further violation, the higher value of the penalty function. Similarly, the other penalty functions correspond to the deadline, the end-to-end timing requirement, and the replication constraints, respectively. Indeed, the penalty functions penalize the candidate solution by increasing its fitness (for our minimization problem), thus discriminating the solution. Section 5.3 explains how our solution framework formulates the penalty functions.

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

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

where β_1 to β_4 are penalty coefficients used to tune the weight of the penalty functions with regard to the range of the objective function. In Section 5.4, the proper value of the penalty coefficients is discussed in more details.

5.3. Penalty Function

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

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

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

$$\phi_{rep}(\mathbf{x}) = \sum_{\forall c_i} \sum_{\forall n_j} \sum_{k=1}^{n_{rep}-1} \sum_{k'=k+1}^{n_{rep}} x_{ij}^{(k)} x_{ij}^{(k')} \tag{33}$$

5.4. Penalty Coefficients

5.5. Metaheuristic Algorithms

5.5.1. 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 [35]. Since then, it is extended in order to address various metaheuristic optimization challenges, such as intensification, diversification, convergence analysis, local optima, parameter tuning, computation time, etc. It is successfully applied on several complex real-world problems, e.g., diagnosis and classification of diseases, efficient engineering designs, tuning control design parameters, scheduling problems, etc [36].

In PSO, the population (or swarm) $PN = \{p_1, p_2, \ldots, p_N\}$ is a collection of particles $p_i \in PN$, organized according to a certain population topology [37]. A particle has a position $\mathbf{x}$ and a velocity $\mathbf{v}$, which denote current location and direction of the particle's motion, and current momentum, respectively. It is a memory-based technique, that is, it remembers the best performance of every particle as well as the best performance of the swarm $\mathbf{z}$ in order to plan for the next move of the particles, where $\mathbf{y}, \mathbf{z}$ are position vectors and have the same dimensions as $\mathbf{x}$. The velocity of a particle is the resultant vector of its current velocity and the particles attraction vectors $(\mathbf{y}-\mathbf{x}), (\mathbf{z}-\mathbf{x})$, respectively, known as *cognitive* and *social* components of the particle's velocity formula, as shown in Equation 34. The attraction vectors impose force of attraction on the particle to move closer to their respective components. Thus, the next position of a particle is the resultant of its current position and its next velocity as shown in Equation (35).

$$\mathbf{v} \leftarrow \omega\mathbf{v} + c_1 Rand() \circ (\mathbf{z} - \mathbf{x}) + c_2 Rand() \circ (\mathbf{z} - \mathbf{x}) \tag{34}$$

$$\mathbf{x} \leftarrow \mathbf{x} + \mathbf{v} \tag{35}$$

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

Although PSO was originally proposed for continuous problem, it is applied to discrete problems successfully as well. In the latter case, the solutions are represented by *0-1* integer variables [38] or integer-linear by approximation to the nearest integer values [39], which is the representation employed adopted in our problem as it is compact, hence fewer decision variables. Accordingly, after the new position (or candidate solution) is determined, following Equations 34 and 35, the solution is discretized by rounding off the its elements to the nearest integer values, that is $\mathbf{x} \leftarrow [\mathbf{x}]$.

5.5.2. Differential Evolution

Similar to PSO, Differential Evolution (DE) is a population-based metaheuristic technique for the global optimization which includes non-linear and non-differentiable problems. It was initially proposed by Storn and Price in 19995 [40], since then it has improved with regard to the different operators of DE such as mutation and crossover, and variants over population topology and hybridization [41]. It is a parallel search technique, therefore, is ideal for computationally intensive problems, and employs mutation and crossover operators that allow the search to skip local minima as opposed to PSO.

In every generation, the population undergoes mutation, crossover, and selection according to the formulas shown in Equation , (37), and (38), respectively. A mutant vector v is created from randomly selected elements $\{a, b, c\} \in PN$ according the mutation operation shown in Equation (36), that is by adding the base matrix to the weighted difference matrix $F \circ (b - c)$, where F controls the amplification of the $(\mathbf{b} - \mathbf{c})$ variation.

$$\mathbf{v} \leftarrow \mathbf{a} + F \circ (\mathbf{b} - \mathbf{c}) \tag{36}$$

$$u_{ik} \leftarrow \begin{cases} v_{ik} & \text{if } U(0,1) \leq CF \text{ and } h = (i * K + k) \\ x_{ik} & \text{if } U(0,1) > CF \text{ and } h \neq (i * K + j) \end{cases} \tag{37}$$

$$\mathbf{x} \leftarrow \begin{cases} \mathbf{u} & \text{if } f(\mathbf{u}) < f(\mathbf{x}) \text{ functions} \\ \mathbf{x} & \text{otherwise} \end{cases} \tag{38}$$

5.5.3. Hybrid Particle Swarm Optimization

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

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

5.5.4. Differential Evolution PSO (DEPSO)

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

5.5.5. Hill-climbing PSO

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

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

6. Evaluation

In this section, we evaluate the proposed approach using synthesized automotive applications that conform to the automotive benchmark proposed by Kramel et al. [14]. The number of runnables, timing specification and activation patterns within the cause-effect chains are also selected according to the benchmark. In order to show the scalability of our approach and to assess the scope of its applicability in practice, in some cases, we use higher specifications standard than what is indicated in the benchmark, e.g., the maximum number of activation patterns is extended from three to four.

In the rest of the section, we describe the setup and method of evaluation, followed by discussion of the evaluation results.

6.1. Setup

The evaluation setup consists of three hardware platforms with different computing capacities, i.e., processing speed and memory size as shown in Table 3. The evaluation on different platforms can be used as performance indicator and also to identify performance bottlenecks in the model. HP EliteBook and Lenovo 20378 are personal computers with core-i5 and core-i7 processors, respectively, whereas PowerEdge is a workstation with much higher processing and memory specification than the personal computers.

Hardware Model	Pro. Model	#Pro.	#Core	Cache	RAM
[1]HP EliteBook	[3]Core i5, 2.2GHz	1	2	3M	8G
Lenovo 20378	[4]Core i7, 2.6GHz	1	4	6M	16G
[2]PowerEdge	[5]Xeon(R), 2.4GHz	24	6	15M	256G

Table 3: Summary of the Hardware Specifications.
[1]HP EliteBook 840 G2, [2]PowerEdge R730 Rack Server
[3]Intel® Core™ i7-4720HQ @ 2.60GHz, [4]Intel® Core™ i7-4720HQ @ 2.60GHz [5]Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz

6.1.1. Software Application Benchmark

In order to evaluate the proposed approach on different ranges of applications, we automatically generate synthetic examples that comply with the automotive benchmark [14]. The examples denote simple to complex automotive functions such as reverse-parking assistance system and engine control system. For simplicity, we identify three classes of applications with different sizes and complexity, shown as tuple (c, r, t, g), respectively for the number of software components, runnables, tasks, and cause-effect chains. Table 4 shows the range of values used in the applications for evaluation.

Parameter	Spec.-I	spec.-II	spec.-III
components, c	≤ 10	≤ 15	> 15
runnables, r	≤ 50	≤ 100	> 100
tasks, t	≤ 30	≤ 60	> 60
cause-effect chains, g	≤ 30	≤ 40	> 60
activation-pattern	$[2, 3, 4]$		
share of activation-patterns	$[0.7, 0.2, 0.1]$		

Table 4: Specification of the Applications for Evaluation.

6.1.2. Platform Benchmark

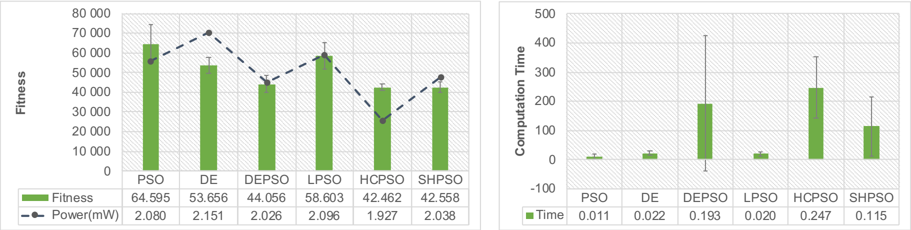
The specification of nodes can be obtained from simulation, vendor product specification and experience. Figure 5 shows the range of values that are used in the nodes' specification. In the experiment, the values are randomly generated while respecting the benchmark.

6.2. Result

We conduct three experiments that assess the proposed approach for scalability in terms of *allocation time*. The allocation time is defined as the time required

Parameter	Range
nodes	$4 - 10$
power consumption (Watt), p	$10 - 200$
failure-rate (/Mhr), λ	$10^4 - 10^{-2}$
speed factor, hz	$0.0 - 1.0$

Table 5: Range of Values for the Specification of Nodes.



(a) Fitness and Power Consumption.

(b) Computation Time

Figure 10: Allocation of Software Application ($N_c = 20, N_r = 200, N_\Gamma = 30$) on N_m number of Heterogeneous Nodes using Different Optimization Methods.

to prepare and solve the allocation problem using the proposed ILP method. Furthermore, we show resource efficiency in terms of saving nodes (i.e., using smaller number of nodes). The experiments consist of: i) varying the size of applications in order to observe the effect of increasing components, runnables and tasks in the system, ii) varying the complexity of applications in order to observe the effect of cause-effect chains in the system, and iii) varying the replications. The experiments are discussed in detail in the next subsection.

A preliminary analysis of the evaluation indicates a successful termination of the allocation for Spec-I&II of the applications. Whereas for Spec-III, the allocation problem is intractable. In fact, it took days on the PowerEdge machine and many times it did not terminate successfully. Therefore, the experimental results that are shown in this paper are conducted on the Spec-I&II classes.

6.2.1. ILP Evaluation

Varying the Size of Application. This refers to increasing the number of software components, as well as runnables, tasks, and cause-effect chains in the system. Figure 11 shows the effect of increasing the size of an application from ($c4, r40, t19, g30$) to ($c10, r100, t57, g60$) on the allocation time and the number of nodes utilized. The applications are allocated to a pool of 8 heterogeneous nodes sharing a single network and with specifications shown in Table **??**. The specifications are generated randomly, with uniform distribution, within the scope of Table 5.

The CPLEX solver returns an optimal solution for the application ($c8, r80, t44, g30$) within 6.06 sec. The allocation time increased sharply to 30.3 sec and 129.4 sec respectively for 9 and 10 components. Even if not indicated in this chart, the

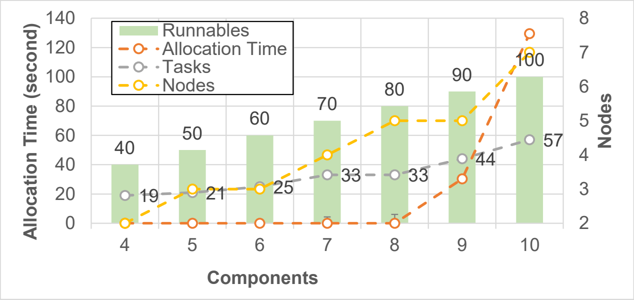
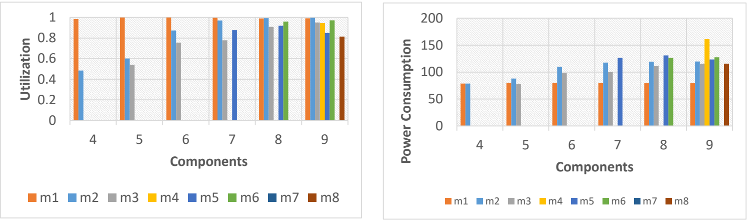


Figure 11: Effect of Varying the Application Size on the Allocation Time and Number of Utilized Nodes.

Application	Measure	PSO	LPSO	DE	DEPSO	HCPSO	SHPSO
App1	Mean						
	Std.Dev.						
App2							

solver, the solver returns an optimal solution within 45 min for components reaching 15 on the PowerEdge machine and OutOfMemory error on the Lenovo and HP machines. Figure 12a and Figure 12b show the utilization and power consumption on each node for the different application sizes. The optimal allocation, in the general case, favor nodes with higher processor speed and lower power consumption specifications.

Varying the Number of Cause-effect Chains. In order to observe the effect of increasing the cause-effect chains on the allocation time, we vary the number of chains in the application from 10 to 60, which is consistent with the benchmark [14]. The share of activation patterns also increases proportionally with the ratio 1 : [0.7, 0.2, 0.1], respectively, for two, three, and four activation patterns. For instance, out of 10 cause-effect chains, there are 7 chains (with two activation patterns), 3 chains (with three activation patterns), and 1 chain (with four activation patterns). The experiment is conducted on two cases of schedulability

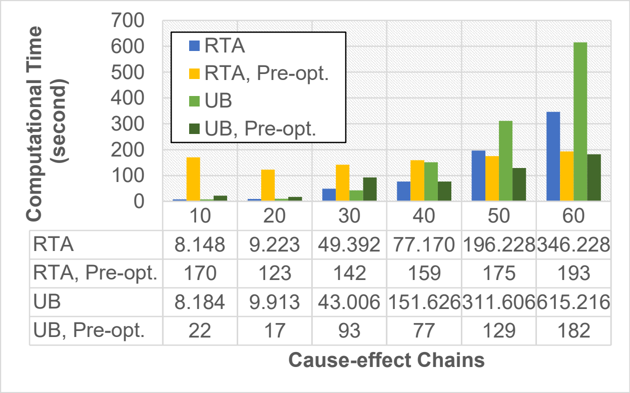


(a) Utilization of Nodes.

(b) Power Consumption of Nodes.

Figure 12: Allocation of Applications on Heterogeneous Nodes.

analysis, namely response time analysis (RTA) and utilization bound (UB), and their result is shown in Figure 13 for increasing number of chains.

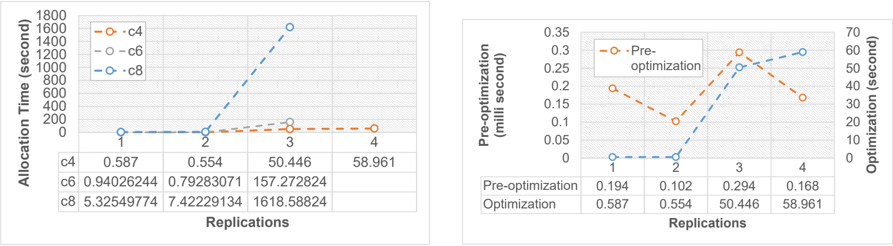


	10	20	30	40	50	60
RTA	8.148	9.223	49.392	77.170	196.228	346.228
RTA, Pre-opt.	170	123	142	159	175	193
UB	8.184	9.913	43.006	151.626	311.606	615.216
UB, Pre-opt.	22	17	93	77	129	182

Figure 13: Effect of Increasing Cause-effect Chains (under RTA and UB) on the Allocation Time.

The figures in the data table show an exponential growth of allocation time whenever the cause-effect chains are increased linearly for a specific application, in both cases of schedulability analysis. In the case of RTA, the overhead in the pre-optimization is higher than the overhead in the optimization for 50 or less chains. Whereas in the UB case, the computational time of pre-optimization is almost always less than the computational time of the optimization. The results are consistent with our expectation that the RTA computation, in the preparation of the timing assertions, is expensive, albeit provides schedulable tasks allocation based on the fixed-priority scheduling policy. In contrast, the UB computation time is relatively low; however, the search space gets larger due to more and more feasible tasks partitioning and chains fulfilling the timing constraints. As a result, the optimization time in the case of UB is usually higher as compared to the case of RTA. Therefore, for applications with chains not more than 40 and naive scheduling assumption using UB, the experiments favor the UB assumption. Whereas, the allocation with the RTA assumption should be selected for applications with exact scheduling requirements. Note, the scalability of this experiment should be seen in conjunction with the experiment discussed in the previous subsection.

Varying Replications. In this experiment, we evaluate the allocation time of the applications with the increase in the number of replications. For the applications specification shown in Figure 14, we vary the replications from 1 to 4. The allocation in all applications took not more than 10 sec for replications 1 and 2. For Spec-I with replication 3 and 4, the allocation time went up close to 1 min. For Spec-III with replication 3, the allocation time went up rapidly to 30 min, and took extremely large time for replication 4 which is also the case for Application-II.



	1	2	3	4
c4	0.587	0.554	50.446	58.961
c6	0.94026244	0.79283071	157.272824	
c8	5.32549774	7.42229134	1618.58824	

	1	2	3	4
Pre-optimization	0.194	0.102	0.294	0.168
Optimization	0.587	0.554	50.446	58.961

(a) Effect on Allocation Time.

(b) Effect on Pre-optimization.

Figure 14: Effect of Varying the Component Replications on the Allocation Time.

Figure 14b shows the effect of shared constraints on the overhead of replications during the pre-optimization and optimization phases of the allocation. In the pre-optimization phase, the allocation time was stable for the increased size of applications. This is due to the fixed constraints regardless of the replications. In contrast, the allocation time increases during optimization as the constraints are applied for the various combination of task partitions, cause-effect chains, and reliability states generated as the result of replications.

6.3. Metaheuristic Evaluation

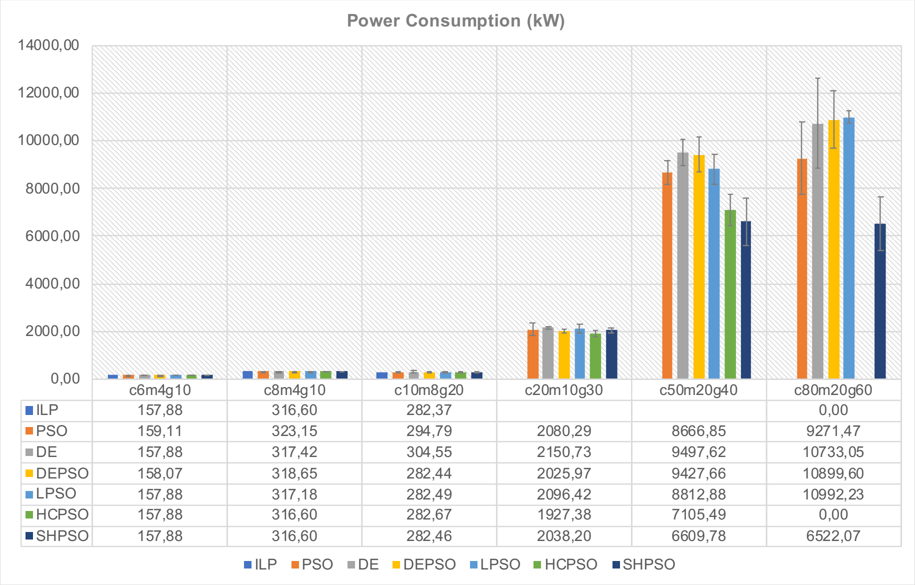
Clearly, the ILP evaluation shows that the ILP approach does not scale to large software allocation problems, that is, that allocation time increases exponentially as the size of the software application increases. In fact, the allocation does not return results for applications consisting more than 15 software components. In this subsection, we evaluate ILP in conjunction with the PSO, DE, and hybrid PSO techniques for different problem sizes.

Table 6 shows the fitness values and allocation time of the different algorithms for the increasing sizes of the software allocation problems. For no constraints violations, the fitness values denote the total power consumption, which is the case for the first three problems whereas it is not the case for the last three problems, thus the fitness values extremely increased as more violations are made. Furthermore, the algorithms highlighted in bold shows the benchmark for the corresponding specific problems.

The result shows that the ILP approach returned solutions for the first three problems, in which case it becomes the benchmark for the quality assessment of the metaheuristic algorithms. Whereas, for the last three problems, ILP does not return solution, which conforms the previous ILP evaluation results. The metaheuristic results are further discussed as follows, that is with regard to solution quality and computation time.

Solution Quality. For the first problem, which models the smallest software allocation over heterogeneous nodes, the algorithms DE, LPSO, HCPSO, SHPSO returned the optimal results besides ILP, with zero deviations. However, DEPSO performed relatively worse but better than PSO. The same results are repeated out of the second allocation problem for the algorithms HCPSO, and SHPSO, which are optimal. In contrast, DE and LPSO performed worse followed by

DEPSO, and PSO performed relatively the worst. For the third problem, no metaheuristic algorithms returned the optimal solutions on average. Furthermore, the stability of the solutions for the algorithms DE and PSO, as indicated by the standard deviations of the fitness values, are quite higher than the rest metaheuristic algorithms.



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

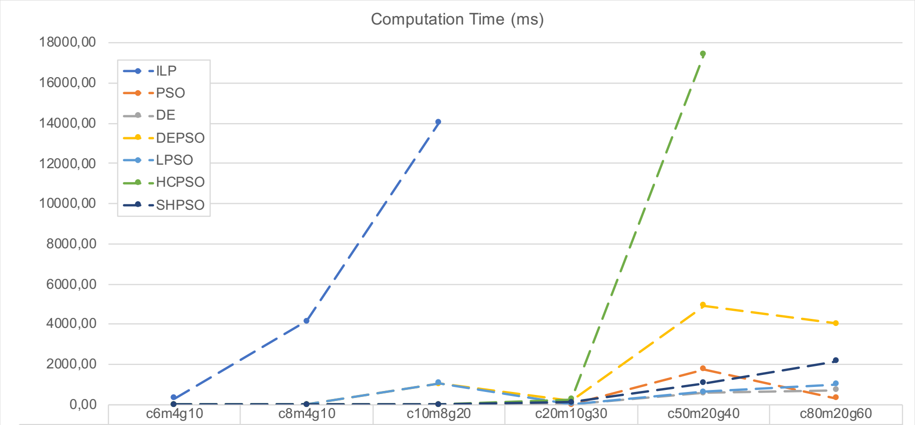
Figure 15: (Near) Optimal Power Consumption of the Different Software Allocation Problems.

As the size of the problem further increases, that is *c20m10g30* and *c20m10g30*, the hybrid PSO performed the far better, with HCPSO the best followed by SHPSO and DEPSO. However, as the size of the problem further increases, that is with *c80m20g60*, HCPSO did not return solutions at reasonable time. In contrast, its stochastic variant HCPSO performed the best followed by DEPSO, LPSO, and PSO, in order of their solutions quality, and DE performed the last.

Allocation Time. In the case of metaheuristic algorithms, the allocation time refers to the amount of time needed to return the near (optimal) result. It is calculated over a maximum of 5000 iterations (or generations) and maximum steady time of 5mins, that is, the period in which no changes are observed in the fitness values. Figure 16 shows the allocation time of the algorithms for the problems listed in Table 6. ILP rapidly increased to 14sec for the problem *c10m8g20* which became intractable for larger problems. Whereas HCPSO rapidly increased to 17sec for the problem *c50m20g40* and became intractable for larger problems. However, for the rest of the algorithms, the allocation time did not exceed 6secs for all of the allocation problems.

Problem	Algorithm	Fitness		Time (ms)		Quality
		Mean	SD	Mean	SD	
c6m4g10	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
c8m6g20	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
c10m8g20	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
c20m10g30	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
c50m20g60	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
c80m20g60	ILP	NA	NA	NA	NA	NA
	PSO	2692638.14	46015.42	324.95	103.66	91.60
	DE	2737416.39	23780.06	716.97	207.19	90.10
	DEPSO	2604249.6	46945.89	4018.55	12.37	94.71
	LPSO	2650992.23	35813.35	1005.74	375.25	93.04
	HCPSO	NA	NA	NA	NA	NA
	SHPSO	2466535.41	89380.36	2147.79	357.58	100.00

Table 6: Fitness and Allocation Time of the ILP and the Metaheuristic Techniques, for the Increasing Sizes of the Software Allocation Problem.



	c6m4g10	c8m4g10	c10m8g20	c20m10g30	c50m20g40	c80m20g60
ILP	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 16: Computation Time of the Various Algorithms for Solving Different Instances of the Software Allocation Problem.

7. Discussion

8. Related Work

In a heterogeneous distributed system where computing nodes and communications links could have various failure rates, a reliability-aware allocation of tasks to nodes, and using links with the lowest failure rates can noticeably improve the system reliability [42][43][34][44]. Interleaving real-time constraints into the problem adds more complexity to reliability-aware task allocation in distributed systems [45]. As opposed to [9][5], we assume that software applications are multirate, which increase the difficulty of software allocation due the complexity of their timing analysis, and increased search space as the result of increasing timed paths of cause-effect chains. Furthermore, we assume a fault-tolerant system model.

Although improving reliability of the system using a reliability-aware task allocation does not impose extra hardware/software cost, in reliability-based design approach, redundancy (or replication) of software or hardware components is frequently applied to improve reliability. In such systems not only optimal allocation of software components (or replicas) should be taken into account but also the cardinality of the replicas should be limited for improved efficiency while meeting the desired reliability requirement. The integration of these two approaches (i.e., reliability-aware task allocation and application redundancy) is a promising technique to deal with high criticality of the system to fulfill the required reliability. For example, [46] proposes a heuristic algorithm to maximize reliability of a distributed system using task replication while at the same time minimizing the makespan of the given task set. Furthermore, in systems with replication, it uses the Minimal Cut Sets method, which is an approximate algorithm, to calculate reliability of a system. In contrast, we apply an exact method based on state enumeration, which is applicable to the problem size assumed in this work.

In our problem, power consumption is the other criterion of the optimization problem. Several research work exist on improving power consumption in real-time distributed systems. The research work [47] shows a survey of different methods on energy-aware scheduling of real-time systems, which categorizes the study into two major groups: i) Dynamic Voltage Scaling (DVS) [48][49], and ii) task consolidation to minimize the number of used computing and communication units [50], which is the approach followed in our work.

In the context of automotive systems, there are few works considering the reliability of a distributed system subject to real-time requirements of the automotive applications [51][52]. There are also other works discussing the allocation of software components onto nodes of a distributed real-time systems that consider other types of constraints other than reliability, for example, i) [53] which considers computation, communication and memory resources, and ii) [15] which proposes a genetic algorithm for a multi-criteria allocation of software components onto heterogeneous nodes including CPUs, GPUs, and FPGAs. Our approach also considers a hetrogeneous platform, i.e., nodes with different power consumption, failure-rate, and processor speed. In this work, we consider only the processor time; however, it can easily be extended to take into account different types of memory consumption that the software applications require.

9. Conclusions and Future Work

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

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

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

Acknowledgement

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[38] J. Kennedy, R. Eberhart, A discrete binary version of the particle swarm algorithm, in: 1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation, Vol. 5, IEEE, pp. 4104–4108. doi:10.1109/ICSMC.1997.637339.
URL `http://ieeexplore.ieee.org/document/637339/`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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