Immune Genetic Algorithms for Optimization of Task Priorities and FlexRay Frame Identifiers

Soheil Samii[1], Yanfei Yin[1,2], Zebo Peng[1], Petru Eles[1], Yuanping Zhang[2]
[1]Department of Computer and Information Science
Linköping University, Sweden
[2]School of Computer and Communication
Lanzhou University of Technology, China

***Abstract*—FlexRay is an automotive communication protocol that combines the comprehensive time-triggered paradigm with an adaptive phase that is more suitable for event-based communication. We study optimization of average response times by assigning priorities and frame identifiers to tasks and messages. Our optimization approach is based on immune genetic algorithms, where in addition to the crossover and mutation operators, we use a vaccination operator that results in considerable improvements in optimization time and quality.**

I. Introduction and Related Work

The FlexRay [2] communication protocol is major communication component in many modern automotive embedded systems. The development of the protocol was initiated partly by the need to integrate traditional time-triggered protocols (e.g., TTP [6]) with adaptive, event-driven protocols like CAN [1], which has been used for decades in the automotive applications domain. FlexRay supports communication in two periodic phases: a *static* phase and a *dynamic* phase. The static phase is based on a TDMA scheme that provides deterministic and comprehensive timing behavior. The dynamic phase is based on a flexible TDMA scheme and is typically used to implement communication for applications with relaxed timing constraints (e.g., embedded control applications)—although recent research results [10], [4] enable designers to implement hard real-time communication also in the dynamic phase.

An important task in the design process is to decide parameters for execution and communication (e.g., task and message priorities, or start times for task executions and message transmissions). Pop et al. [9] proposed a solution to the problem of assigning frame identifiers to messages in the dynamic phase of FlexRay such that strict timing constraints are satisfied. Other design methods for systems with the TTP or CAN protocols have been reported by Pop et al. [8], [7] and Harbour et al. [3]. An important objective for many embedded applications is the minimization of average delays, especially for embedded control applications [14]. The contribution of this paper is a method for priority assignment and bus-access optimization for FlexRay-based embedded systems, where the objective is to minimize the average end-to-end delays. We propose

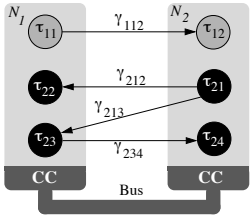

Figure 1. Platform and application example

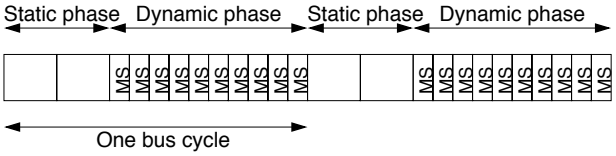

Figure 2. Bus cycle

an optimization method based on traditional genetic algorithms [5] and a vaccination operator [12], [13], resulting in so-called immune genetic algorithms.

II. System Model

A. Platform Model

The execution platform consists of a set of computation nodes $\mathbf{N}$, indexed by $\mathcal{I}_{\mathbf{N}}$, that are connected by a communication controller to a bus. Figure 1 shows a system with two nodes (N_1 and N_2) connected by communication controllers (denoted CC in the figure) to a bus. The figure also shows a set of application tasks; we shall discuss this in Section II-B. The communication in the system is conducted according to the FlexRay protocol. The bus cycle is periodic and is composed of two phases: the static phase and the dynamic phase. Each of the two phases is of fixed and given length. The static phase is divided into static slots of equal lengths. Each static slot is assigned one node; thus, a node has a set of static slots (possibly no slots assigned) and can send messages in the static phase only in those slots. Figure 2 shows an example of a FlexRay bus cycle with two static slots in the static phase.

The dynamic phase is divided into minislots. The bus cycle in Figure 2 has 10 equal-length minislots (denoted MS) in the dynamic phase. The communication in the dynamic phase is conducted in dynamic slots (as opposed to static slots of fixed size in the static phase). The slots in the dynamic phase are of varying size depending on whether messages are sent in certain minislots and the

number of minislots that are needed to send a particular message. The maximum number of dynamic slots n_{DYN} is given as a part of the FlexRay configuration. Each dynamic slot is assigned one node. Thus, given is a function $\mathsf{dyn} : \{1, \ldots, n_{\text{DYN}}\} \longrightarrow \mathbf{N}$ that assigns a computation node to each dynamic slot. Let us also introduce the function $\mathsf{dyn}^* : \mathbf{N} \longrightarrow 2^{\{1,\ldots,n_{\text{DYN}}\}}$ that gives the set of dynamic slots assigned to a node—thus, $\mathsf{dyn}^*(N_d) = \{\delta \in \{1, \ldots, n_{\text{DYN}}\} : \mathsf{dyn}(\delta) = N_d\}$.

B. Application Model

On the execution platform runs a set of applications $\mathbf{\Lambda}$, indexed by the set $\mathcal{I}_{\mathbf{\Lambda}}$. An application $\Lambda_i \in \mathbf{\Lambda}$ $(i \in \mathcal{I}_{\mathbf{\Lambda}})$ is modeled as a directed acyclic graph $(\mathbf{T}_i, \mathbf{\Gamma}_i)$, where the nodes $\mathbf{T}_i$, indexed by $\mathcal{I}_i$, represent computation tasks and the edges $\mathbf{\Gamma}_i \subset \mathbf{T}_i \times \mathbf{T}_i$ represent data dependencies between the tasks. The set $\mathbf{T}_{\mathbf{\Lambda}} = \bigcup_{i \in \mathcal{I}_{\mathbf{\Lambda}}} \mathbf{T}_i$ represents all application tasks. Let us also introduce the set $\mathcal{I}_{\text{tasks}} = \cup_{i \in \mathcal{I}_{\mathbf{\Lambda}}} (\cup_{j \in \mathcal{I}_i} \{(i, j)\})$ that is used to index all application tasks in the system. In the system depicted in Figure 1, we have two applications, Λ_1 and Λ_2. Application Λ_1 consists of two tasks, $\mathbf{T}_1 = \{\tau_{11}, \tau_{12}\}$, depicted with grey circles. The second application consists of four tasks depicted with black circles. The tasks have data dependencies that are indicated by the arrows in the figure. For example, task τ_{12} receives a message $\gamma_{112} = (\tau_{11}, \tau_{12})$ from task τ_{11}.

An application $\Lambda_i \in \mathbf{\Lambda}$ has a release period h_i. At time $(q-1)h_i$, a job of each task in the application is released for execution. Job q of task τ_{ij} is denoted $\tau_{ij}^{(q)}$ and is released at time $(q-1)h_i$. For a message $\gamma_{ijk} = (\tau_{ij}, \tau_{ik}) \in \mathbf{\Gamma}_i$, the message instance produced by job $\tau_{ij}^{(q)}$ is denoted $\gamma_{ijk}^{(q)}$. An edge $\gamma_{ijk} = (\tau_{ij}, \tau_{ik}) \in \mathbf{\Gamma}_i$ indicates that the earliest start time of a job $\tau_{ik}^{(q)}$ is when $\tau_{ij}^{(q)}$ has completed its execution and the produced data (i.e., $\gamma_{ijk}^{(q)}$) has been communicated to $\tau_{ik}^{(q)}$. We also define the *hyperperiod* $h_{\mathbf{\Lambda}}$ as the least common multiple of the periods $\{h_i : i \in \mathcal{I}_{\mathbf{\Lambda}}\}$. The time difference between the completion and release of a job is called its *response time*. Thus, if job $\tau_{ij}^{(q)}$ finishes at time t, then its response time is defined as $r_{ij}^{(q)} = t - (q-1)h_i$. The response time is a key consideration in the design phase of real-time systems. The worst-case response time of a task is one of the most important characterizations of hard real-time applications. In other application domains (e.g., for control systems), average response times are more important parameters of the system performance.

C. Mapping and Execution Times

Each task is mapped to a node. The mapping is given by a function $\mathsf{map} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{N}$. Let us also introduce the function $\mathsf{map}^* : \mathbf{N} \longrightarrow 2^{\mathbf{T}_{\mathbf{\Lambda}}}$ that, given a node N_d, gives the set of tasks that are mapped to N_d—thus, $\mathsf{map}^*(N_d) = \{\tau_{ij} \in \bigcup_{i \in \mathcal{I}_{\mathbf{\Lambda}}} \mathbf{T}_i : \mathsf{map}(\tau_{ij}) = N_d\}$. We remind that Figure 1 also shows the mapping of our example system. A message between tasks mapped to different nodes is sent on the bus; thus, the set of messages that are communicated on the bus is $\mathbf{\Gamma}_{\text{bus}} = \{\gamma_{ijk} = (\tau_{ij}, \tau_{ik}) \in \mathbf{\Gamma}_i : \mathsf{map}(\tau_{ij}) \neq \mathsf{map}(\tau_{ik}), i \in \mathcal{I}_{\mathbf{\Lambda}}\}$. For a message instance $\gamma_{ijk}^{(q)}$, we denote with c_{ijk} the communication time when there are no conflicts on the bus. Given a mapping of the tasks to the nodes, for each task, we have a specification of possible execution times. The execution time of task τ_{ij} is modeled as a stochastic variable c_{ij} with probability function $\xi_{c_{ij}}$, and it is bounded by given best-case and worst-case execution times that are denoted c_{ij}^{bc} and c_{ij}^{wc}, respectively.

D. Task and Message Scheduling

The tasks are scheduled on each node according to fixed priorities. When a job is released for execution, it is assigned the priority of the task that released it. At any point in time, the job with highest priority executes. An executing job can be preempted by a higher-priority job.

Let us now consider message transmission in the dynamic phase of the FlexRay bus cycle (see Figure 2). The dynamic phase has a given length and consists of a given number of minislots; the length of a minislot is thus also given. Given is also the maximum number of dynamic slots n_{DYN} within one dynamic phase. These parameters for the bus configuration were discussed in Section II-A. There, we also introduced the dynamic-slot mapping to computation nodes as a function dyn. In Figure 3, we show the communication subsystem of the example in Figure 1. We can see that the first node has two dynamic slots assigned (slots 1 and 3), whereas the second node can send messages only in the second dynamic slot. The total number of dynamic slots is three in this example. A node has in its communication controller a message queue for each assigned dynamic slot. The message queues are depicted with white rectangles in Figure 3. When a task terminates and produces a message to a task on another computation node, the message is transferred to the designated queue. The communication controller transmits the message on the bus when the corresponding dynamic slot of the queue starts.

Each message $\gamma_{ijk} = (\tau_{ij}, \tau_{ik}) \in \mathbf{\Gamma}_{\text{bus}}$ on the bus is assigned a frame identifier f_{ijk}. The message is sent in the dynamic slot indicated by the assigned frame identifier. The frame identifier must be assigned a value that corresponds to a dynamic slot available to the sending computation node—thus, $f_{ijk} \in \mathsf{dyn}^*(\mathsf{map}(\tau_{ij}))$. In the example in Figure 3, we have four messages. Message γ_{112} is assigned frame identifier 1 and is thus sent in the first dynamic slot. Messages that are assigned the same frame identifier might be readyfor transmission at the same time during runtime. Those messages are ordered

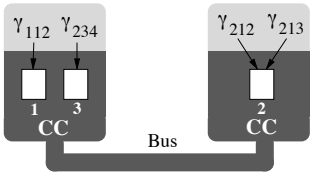

Figure 3. Dynamic-slot assignment

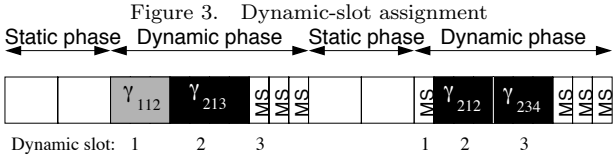

Figure 4. Communication in dynamic slots

by priorities. Each message is given a fixed priority, which decides the position of the message in the message queue in the communication controller. Messages γ_{212} and γ_{213} have been assigned the same frame identifier. We consider that γ_{213} has the higher priority.

III. FlexRay Communication Example

Let us consider again the system depicted in Figure 1. The bus is configured like in Figure 2 with a static phase comprising two static slots (not used in this example) and a dynamic phase comprising 10 minislots. The maximum number of dynamic slots is three ($n_{\text{DYN}} = 3$). The dynamic slots have been assigned to the two nodes as shown in Figure 3. The figure shows the messages $\gamma_{112}, \gamma_{212}, \gamma_{213}, \gamma_{234}$ and with arrows their corresponding frame identifiers, which indicate the dynamic slot of each message. Initially, only task τ_{11} can be executed on node N_1 and only τ_{21} can be executed on node N_2. The other tasks are waiting for messages. Let us assume that both tasks execute and finish during the static phase. When the dynamic phase starts, messages γ_{112} (to task τ_{12}), γ_{212} (to task τ_{22}), and γ_{213} (to task τ_{23}) are in the corresponding message queues in the communication controllers and are ready to be transmitted. Message γ_{112} has been assigned frame identifier 1 and is therefore sent in the first dynamic phase. This is indicated in the bus schedule in Figure 4. The length of the message is three minislots, which means that, in this bus cycle, the first dynamic slot comprises three minislots.

The second dynamic slot starts at the end of the third minislot. As we can see in Figure 3, both γ_{212} and γ_{213} have been assigned frame identifier 2. Message γ_{213} has higher priority than γ_{212} and is therefore the message that is transmitted in the second dynamic slot (as indicated in Figure 4). The length of the message is four minislots. When this message has been received by N_1, task τ_{23} can start its execution. If message γ_{212} is assigned a higher priority, it will affect the temporal characteristics of γ_{213}, which is the message that will be sent in the second dynamic phase. The third dynamic slot starts at the end of minislot 7. The only message that is assigned frame identifier 3 is γ_{234} on node N_1.

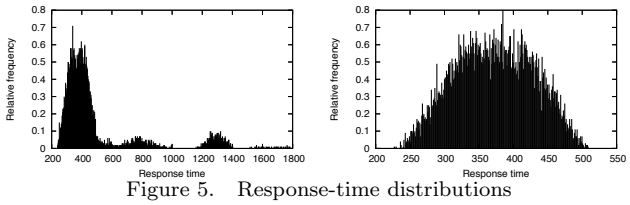

Figure 5. Response-time distributions

However, because τ_{23} has not yet finished its execution, the message is not ready to be sent. In this first bus cycle, nothing is sent in the third dynamic slot (the slot comprises one minislot). The length of a dynamic slot is one minislot if no message is transmitted, or the length of the message if a message is transmitted.

In the second bus cycle, we assume that τ_{23} finishes its execution during the static phase. At the start of the second dynamic phase, messages γ_{212} and γ_{234} (from task τ_{23}) are ready for transmission. The first dynamic phase is one minislot, because no message with frame identifier 1 is available. Message γ_{212} is sent in the second dynamic phase and occupies 3 minislots. Message γ_{234} is transmitted in the third dynamic slot. The waiting tasks will execute upon reception of messages. Note that a message will be transmitted only if there are enough minislots left until the end of the dynamic phase. If there are not enough minislots, the message transmission is delayed to the next bus cycle.

Let us end this section by highlighting that the temporal behavior of the system is affected by the task priorities and the frame identifiers. We have conducted two simulations of a system with 35 tasks and considered the response time of one task. Each simulation is performed for a certain assignment of priorities and frame identifiers. Figure 5 shows two histograms of response times, corresponding to the two different assignments of priorities and frame identifiers, respectively. On the horizontal axis, we depict the response time, whereas on the vertical axis, we show the number of times a certain response time occurred relative to the total number of response times observed during simulation. Each graph represents the response-time distribution of a task. The average response time for the left graph is 477, whereas for the right graph it is 369. The priorities and frame identifiers corresponding to the right graph lead to much better temporal behavior and average response time in particular, compared to the priorities and frame identifiers corresponding to the left graph.

IV. Problem Formulation

The inputs to our design-optimization problem are

- a set of applications $\mathbf{\Lambda}$, indexed by $\mathcal{I}_{\mathbf{\Lambda}}$,
- a release period h_i for each application Λ_i ($i \in \mathcal{I}_{\mathbf{\Lambda}}$),
- a set of computation nodes $\mathbf{N}$ connected to a bus,
- a mapping function $\mathsf{map} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{N}$ of the whole task set,

- the length of the static and dynamic phase of the FlexRay communication cycle,
- the number of minislots in the dynamic phase,
- the number of dynamic slots n_{DYN} and the dynamic-slot assignment $\mathsf{dyn} : \{1, \ldots, n_{\text{DYN}}\} \longrightarrow \mathbf{N}$ to the computation nodes, and
- execution-time distributions of the tasks and communication times of messages.

The outputs are a fixed priority for each task, and a frame identifier and priority for each message.

Before we discuss the optimization objective, let us define what constitutes a solution. A solution consists of three parts: (1) the task priorities, (2) the frame identifiers for the messages, and (3) the message priorities. We represent a solution as a sequence $\boldsymbol{\psi} = (\boldsymbol{\rho}, \boldsymbol{f}, \boldsymbol{\omega})$ consisting of the three parts. Let $\sigma_{\mathbf{T}} : \{1, \ldots, |\mathbf{T}_{\mathbf{\Lambda}}|\} \longrightarrow \mathbf{T}_{\mathbf{\Lambda}}$ be any bijection, giving an ordering of the tasks. The first part $\boldsymbol{\rho}$ (the task priorities) is a tuple

$$(\rho_1, \ldots, \rho_{|\mathbf{T}_{\mathbf{\Lambda}}|}) \in \underbrace{\mathbb{N} \times \ldots \times \mathbb{N}}_{|\mathbf{T}_{\mathbf{\Lambda}}| \text{ times}} = \mathbb{N}^{|\mathbf{T}_{\mathbf{\Lambda}}|},$$

where each ρ_k is the priority of task $\sigma_{\mathbf{T}}(k)$. We require unique task priorities—that is, $\rho_k \neq \rho_l$ for $k \neq l$. The set of allowed priority assignments is

$$\mathbf{P} = \left\{ (\rho_1, \ldots, \rho_{|\mathbf{T}_{\mathbf{\Lambda}}|}) \in \mathbb{N}^{|\mathbf{T}_{\mathbf{\Lambda}}|} : \rho_k \neq \rho_l, k \neq l \right\}.$$

Let us now continue with the remaining two parts and introduce $\sigma_{\mathbf{\Gamma}} : \{1, \ldots, |\mathbf{\Gamma}_{\text{bus}}|\} \longrightarrow \mathbf{\Gamma}_{\text{bus}}$ as any bijection. The second part $\boldsymbol{f}$ (the frame identifiers) assigns a frame identifier to each message and is a tuple $\boldsymbol{f} = (f_1, \ldots, f_{|\mathbf{\Gamma}_{\text{bus}}|})$, where f_k is the frame identifier of message $\sigma_{\mathbf{\Gamma}}(k)$. The frame identifiers must be assigned according to the dynamic-slot assignments to computation nodes. This means that

$$\boldsymbol{f} \in \mathbf{F} = \{ (f_1, \ldots, f_{|\mathbf{\Gamma}_{\text{bus}}|}) \in \mathbb{N}^{|\mathbf{\Gamma}_{\text{bus}}|} : f_k \in \mathsf{dyn}^*(\mathsf{map}(\tau_{ij})), (\tau_{ij}, \tau) = \sigma_{\mathbf{\Gamma}}(k) \}.$$

The third part of the solution comprises the message priorities. They are used to prioritize among messages with the same frame identifier. The message priorities are given by a tuple $\boldsymbol{\omega} = (\omega_1, \ldots, \omega_{|\mathbf{\Gamma}_{\text{bus}}|})$, where ω_k is the priority of message $\sigma_{\mathbf{\Gamma}}(k)$. The set of allowed message-priority assignments, from which $\boldsymbol{\omega}$ can be chosen, is

$$\mathbf{\Omega} = \left\{ (\omega_1, \ldots, \omega_{|\mathbf{\Gamma}_{\text{bus}}|}) \in \mathbb{N}^{|\mathbf{\Gamma}_{\text{bus}}|} : \omega_k \neq \omega_l, k \neq l \right\}.$$

Having defined the solution space, our optimization objective is to find a solution $\boldsymbol{\psi} = (\boldsymbol{\rho}, \boldsymbol{f}, \boldsymbol{\omega}) \in \mathbf{P} \times \mathbf{F} \times \mathbf{\Omega}$ that minimizes a cost C. This cost is defined as a weighted sum of average response times

$$C = \sum_{(i,j) \in \mathcal{I}_{\text{tasks}}} w_{ij} \overline{R}_{ij}, \tag{1}$$

where $\overline{R}_{ij}$ denotes the average response time of task τ_{ij}. The weights $w_{ij} \geqslant 0$ are given by the designer. We are interested to minimize the delays of selected tasks in the system. Many of the weights are usually zero.

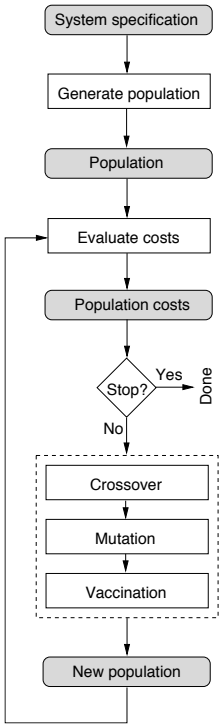

Figure 6. Flowchart of the otimization approach

V. Optimization Approach

We propose an immune genetic algorithm [12], [13] for the assignment of priorities and frame identifiers to the application tasks and messages. A flowchart of our approach is depicted in Figure 6. We start by generating a given number of solutions randomly, also called members. These members comprise the initial population $\mathbf{\Psi}_1 \subset \mathbf{P} \times \mathbf{F} \times \mathbf{\Omega}$. We proceed by evaluating the cost of each member in the population. Then, we check for a convergence criterion to determine whether we should proceed or stop the optimization. We shall discuss this stopping condition later in this section. After this step, if it is decided to continue the optimization, we create a new generation and go back to the evaluation step.

Let us now consider any iteration $k > 0$ in the optimization process. The population is denoted $\mathbf{\Psi}_k$ and contains the members to be evaluated. We compute the cost of each member by simulation. Given a member $\boldsymbol{\psi} = (\boldsymbol{\rho}, \boldsymbol{f}, \boldsymbol{\omega}) \in \mathbf{\Psi}_k$ (priorities and frame identifiers), we use our simulation environment for distributed real-time systems [11] to obtain an approximation of the average response times, which constitute our optimization objective. In addition to the given priorities and frame identifiers, the simulation environment considers the execution-time distributions of the tasks and the given communication times of messages. During simulation, we compute the average response times periodically with the period $h_{\mathbf{\Lambda}}$. Let $\mathbf{R}_{ij}^{(k)}$ denote the set of response times of task τ_{ij} that occur in the time interval $[0, kh_{\mathbf{\Lambda}}[$ during

simulation ($k > 0$). At times $qh_{\mathbf{\Lambda}}$ during simulation, we compute the average delays

$$\overline{R}_{ij}^{(q)} = \sum_{r \in \mathbf{R}_{ij}^{(q)}} r \Bigg/ \left|\mathbf{R}_{ij}^{(q)}\right|$$

for each task τ_{ij} with $w_{ij} \neq 0$. The simulation is terminated at the first simulated time-instant $q' h_{\mathbf{\Lambda}}$ ($q' > 1$) in which the condition

$$\frac{\left|\overline{R}_{ij}^{(q')} - \overline{R}_{ij}^{(q'-1)}\right|}{\overline{R}_{ij}^{(q'-1)}} < \zeta^{\text{sim}}$$

is satisfied for all tasks τ_{ij} with $w_{ij} \neq 0$. The parameter ζ^{sim} is given as an input; we have experimentally tuned this parameter to $\zeta^{\text{sim}} = 0.05$, which means that the simulation is stopped when the average response time of each task has changed with less than 5 percent. After the simulated time $q' h_{\mathbf{\Lambda}}$, we have the average response time as $\overline{R}_{ij} = \overline{R}_{ij}^{(q')}$. We compute then the cost $C_{\boldsymbol{\psi}}$ of solution $\boldsymbol{\psi}$ according to Equation 1.

We keep track off the best solution found up to any point in the optimization process. After the evaluation step in an iteration, we find the solution with the smallest cost in the population. If this cost is smaller than the cost of the best solution found so far, then this best solution is updated. The optimization terminates if the best solution is unchanged for five consecutive iterations. If this last condition is not satisfied, we generate a population $\mathbf{\Psi}_{k+1}$ to be evaluated in the next iteration. In the remainder of this section, we shall discuss the generation of the next population; this generation step (the dashed box in Figure 6) is the most important part of the whole optimization approach. The first two steps in the dashed box are crossover and mutation (Sections V-A and V-B). The third and last step is the application of a vaccination operator (Section V-C).

A. Crossover

After the evaluation of the current population $\mathbf{\Psi}_k$ (simulation of each member $\boldsymbol{\psi} \in \mathbf{\Psi}_k$), we identify the two solutions $\boldsymbol{\psi}^{\text{first}}$ and $\boldsymbol{\psi}^{\text{second}}$ with the smallest cost and second-smallest cost, respectively. These two members shall survive to the next generation; that is, $\boldsymbol{\psi}^{\text{first}}, \boldsymbol{\psi}^{\text{second}} \in \mathbf{\Psi}_{k+1}$. It remains to generate the other $|\mathbf{\Psi}_k| - 2$ members (we require $|\mathbf{\Psi}_k|$ to be even). Thus, after the identification of the two best members in $\mathbf{\Psi}_k$, the next step is to perform crossover. A crossover operation is initiated by choosing two distinct parent members $\boldsymbol{\psi}^{(1)}, \boldsymbol{\psi}^{(2)} \in \mathbf{\Psi}_k$. Then, with probability ζ^{co}, we generate two offsprings $\boldsymbol{\psi}'^{(1)}$ and $\boldsymbol{\psi}'^{(2)}$ from the parents $\boldsymbol{\psi}^{(1)}$ and $\boldsymbol{\psi}^{(2)}$. All operations on the members are done individually on the three parts of the solution.

We first generate crossover points $\kappa_{\boldsymbol{\rho}} \in \{1, \ldots, |\mathbf{T}_{\mathbf{\Lambda}}|\}$ and $\kappa_{\boldsymbol{f}}, \kappa_{\boldsymbol{\omega}} \in \{1, \ldots, |\mathbf{\Gamma}_{\text{bus}}|\}$ randomly. Let us denote the first offspring with $\boldsymbol{\psi}'^{(1)} = (\boldsymbol{\rho}'^{(1)}, \boldsymbol{f}'^{(1)}, \boldsymbol{\omega}'^{(1)})$ and consider each of the three parts $\boldsymbol{\rho}'^{(1)}$, $\boldsymbol{f}'^{(1)}$, and $\boldsymbol{\omega}'^{(1)}$ individually. The sequence $\boldsymbol{\rho}'^{(1)}$ begins with the elements $\rho_1^{(1)}, \ldots, \rho_{\kappa_{\boldsymbol{\rho}}}^{(1)}$ (the first $\kappa_{\boldsymbol{\rho}}$ priorities in $\boldsymbol{\rho}^{(1)}$). We shall now describe how the remaining $|\mathbf{T}_{\mathbf{\Lambda}}| - \kappa_{\boldsymbol{\rho}}$ elements are chosen from $\boldsymbol{\rho}^{(2)}$. For each $q \in \{1, \ldots, |\mathbf{T}_{\mathbf{\Lambda}}|\}$, we remove $\rho_q^{(2)}$ from $\boldsymbol{\rho}^{(2)}$ if $\rho_q^{(2)} \in \{\rho_1^{(1)}, \ldots, \rho_{\kappa_{\boldsymbol{\rho}}}^{(1)}\}$. Exactly $\kappa_{\boldsymbol{\rho}}$ priorities are removed from $\boldsymbol{\rho}^{(2)}$. Let us denote the sequence of priorities that remain after the removal with $\rho_1^*, \ldots, \rho_{|\mathbf{\Lambda}_{\mathbf{T}}| - \kappa_{\boldsymbol{\rho}}}^*$. These are all different from the priorities $\rho_1^{(1)}, \ldots, \rho_{\kappa_{\boldsymbol{\rho}}}^{(1)}$. Finally, we obtain

$$\boldsymbol{\rho}'^{(1)} = \left(\rho_1^{(1)}, \ldots, \rho_{\kappa_{\boldsymbol{\rho}}}^{(1)}, \rho_1^*, \ldots, \rho_{|\mathbf{\Lambda}_{\mathbf{T}}| - \kappa_{\boldsymbol{\rho}}}^*\right).$$

In this way, we guarantee that the new priority tuple $\boldsymbol{\rho}'^{(1)}$ assigns a unique priority to each task. The crossover of the frame identifiers gives

$$\boldsymbol{f}'^{(1)} = \left(f_1^{(1)}, \ldots, f_{\kappa_{\boldsymbol{f}}}^{(1)}, f_{\kappa_{\boldsymbol{f}}+1}^{(2)}, \ldots, f_{|\mathbf{\Gamma}_{\text{bus}}|}^{(2)}\right).$$

The message priorities $\boldsymbol{\omega}'^{(1)}$ are generated with the crossover point $\kappa_{\boldsymbol{\omega}}$ and the parents $\boldsymbol{\omega}^{(1)}$ and $\boldsymbol{\omega}^{(2)}$ in the same way as the crossover of the task priorities. The second offspring $\boldsymbol{\psi}'^{(2)} = (\boldsymbol{\rho}'^{(2)}, \boldsymbol{f}'^{(2)}, \boldsymbol{\omega}'^{(2)})$ is given by the same construction as already described for the first offspring, but by considering $\boldsymbol{\psi}^{(2)}$ to the left of each crossover point and $\boldsymbol{\psi}^{(1)}$ to the right.

With probability $1 - \zeta^{\text{co}}$, we do not generate the offsprings but instead let $\boldsymbol{\psi}^{(1)}$ and $\boldsymbol{\psi}^{(2)}$ survive to the next generation. We repeat this crossover step $(|\mathbf{\Psi}_k| - 2)/2$ times. The parent members are chosen based on the so-called "roulette-wheel selection" scheme, which means that members are chosen with probabilities induced by their costs. In this selection scheme, a member with a smaller cost than another member has a larger probability to be chosen. The crossover process gives us a set of members $\mathbf{\Psi}_k^{\text{co}}$, where $|\mathbf{\Psi}_k^{\text{co}}| = |\mathbf{\Psi}_k| - 2$. Some of the members in $\mathbf{\Psi}_k^{\text{co}}$ are taken from $\mathbf{\Psi}_k$ and some are new offspring members as a result of the crossover step. The parameter ζ^{co} is called the *crossover rate* and decides the portion of $\mathbf{\Psi}_k^{\text{co}}$ that are offsprings from the crossover operation and the portion that contains members from $\mathbf{\Psi}_k$ that survive into $\mathbf{\Psi}_k^{\text{co}}$.

B. Mutation

After the crossover step, which combines solutions into new solutions, we apply the mutation operator on the population $\mathbf{\Psi}_k^{\text{co}}$. Mutation is performed on each member $\boldsymbol{\psi} \in \mathbf{\Psi}_k^{\text{co}}$ with probability ζ^{mut}, called the *mutation rate*. Mutation on a member $\boldsymbol{\psi} = (\boldsymbol{\rho}, \boldsymbol{f}, \boldsymbol{\omega}) \in \mathbf{\Psi}_k^{\text{co}}$ generates a new member $\boldsymbol{\psi}' = (\boldsymbol{\rho}', \boldsymbol{f}', \boldsymbol{\omega}')$ to replace $\boldsymbol{\psi}$. We first generate two distinct positions $i, j \in \{1, \ldots, |\mathbf{T}_{\mathbf{\Lambda}}|\}$, $i < j$, randomly. These two positions are swapped, obtaining the mutated task priorities $\boldsymbol{\rho}' =$

$$\left(\rho_1, \ldots, \rho_{i-1}, \rho_j, \rho_{i+1}, \ldots, \rho_{j-1}, \rho_i, \rho_{j+1}, \ldots, \rho_{|\mathbf{T}_{\mathbf{\Lambda}}|}\right).$$

Second, mutation is applied on the frame identifiers $\boldsymbol{f}$. A position $q \in \{1, \ldots, |\mathbf{\Gamma}_{\text{bus}}|\}$ is generated randomly. Then, the frame identifier pointed out by this position

is modified to a frame identifier generated randomly from the set of dynamic slots available to the node that sends the message. Let $(\tau_{ij}, \tau) = \sigma_{\mathbf{\Gamma}}(q)$ be the message. The new frame identifier f'_q is chosen randomly from the set of dynamic slots $\mathsf{dyn}^*(\mathsf{map}(\tau_{ij}))$. Thus, we have $\boldsymbol{f}' = \left(f_1, \ldots, f_{q-1}, f'_q, f_{q+1}, \ldots, f_{|\mathbf{\Gamma}_{\text{bus}}|}\right)$.

Third, mutation is applied to the message priorities (similar to mutation on task priorities). Two distinct positions $i, j \in \{1, \ldots, |\mathbf{\Gamma}_{\text{bus}}|\}$, $i < j$, are generated randomly. These two positions are swapped in $\boldsymbol{\omega}$, thus obtaining $\boldsymbol{\omega}'$. After the mutation step, we have a new population $\mathbf{\Psi}_k^{\text{mut}}$, which contains members from $\mathbf{\Psi}_k^{\text{co}}$ that are either preserved or mutated (the distribution depends on the chosen mutation rate ζ^{mut}).

C. Vaccination

A vaccine is an operator that enforces a certain improving characteristic into a solution. Typically, a vaccine changes one or very few positions of a member based on intuitions or indications of what improves an existing solution. Such indications are based on offline analysis of the problem at hand, or based on the results obtained during the optimization process. Vaccination can achieve improvements of genetic algorithms, both in terms of quality of the final solution and in terms of optimization time (faster convergence).

Let us define a vaccine as a function $v : \mathbf{P} \times \mathbf{F} \times \mathbf{\Omega} \longrightarrow \mathbf{P} \times \mathbf{F} \times \mathbf{\Omega}$ that enforces some characteristic into a member. Vaccination of a member $\boldsymbol{\psi} \in \mathbf{\Psi}_k^{\text{mut}}$ generates a new member $v(\boldsymbol{\psi})$ to be included in the next population $\mathbf{\Psi}_{k+1}$. Vaccines can be combined together to form a new vaccine. For example, the composition $v_1 \circ v_2$ of two vaccines is another vaccine. When applied to a member $\boldsymbol{\psi} \in \mathbf{\Psi}_k^{\text{mut}}$, it produces a vaccinated member $(v_1 \circ v_2)(\boldsymbol{\psi}) = v_1(v_2(\boldsymbol{\psi}))$. Let us now describe how vaccines are created during the optimization process. Then, we describe how we select members for vaccination.

1) Vaccine Creation: In each iteration, after the evaluation of the members in the current population $\mathbf{\Psi}_k$, we identify the member $\boldsymbol{\psi}_k^{\text{best}} \in \mathbf{\Psi}_k$ with smallest cost among all members. To avoid memory explosion, we consider only the best members of the last 100 iterations, which means that old solutions are replaced with new solutions. At any iteration $k > 0$, we have a set

$$\mathbf{\Psi}_k^{\text{best}} = \begin{cases} \{\boldsymbol{\psi}_1^{\text{best}}, \ldots, \boldsymbol{\psi}_k^{\text{best}}\} & \text{if } k \leqslant 100 \\ \{\boldsymbol{\psi}_{k-99}^{\text{best}}, \ldots, \boldsymbol{\psi}_k^{\text{best}}\} & \text{if } k > 100 \end{cases}$$

of the best members available. Table I shows an the contents of the set $\mathbf{\Psi}_5^{\text{best}}$ with the best solutions found during the first five iterations of an optimization process. Each row in the table shows a solution and its value at each position. There are four tasks and three messages. The task priorities are displayed first, followed by the frame identifiers and priorities of the messages. The best solution of the fourth iteration assigns priority 3 to the second task and frame identifier 3 to the first message.

Table I
BEST OBTAINED SOLUTIONS

Iteration	ρ_1	ρ_2	ρ_3	ρ_4	f_1	f_2	f_3	ω_1	ω_2	ω_3
1	3	1	2	4	3	2	3	3	1	2
2	1	4	2	3	3	3	3	1	3	2
3	2	4	1	3	1	1	1	2	1	3
4	1	3	2	4	3	1	3	1	2	3
5	3	4	2	1	3	4	2	2	1	1
Dom. value	–	4	2	–	3	1	3	–	1	–
Dominance	–	0.6	0.8	–	0.8	0.4	0.6	–	0.6	–

At each iteration $k > 0$, we create vaccines based on $\mathbf{\Psi}_k^{\text{best}}$. For each position in the members in $\mathbf{\Psi}_k^{\text{best}}$, we are interested in the value that occurs most frequently. This value is called the *dominating value* of that position. In general, for the task priorities and for each position $q \in \{1, \ldots, |\mathbf{T}_{\mathbf{\Lambda}}|\}$, we build the sequence $(\rho_q)_{\boldsymbol{\psi}=(\boldsymbol{\rho},\boldsymbol{f},\boldsymbol{\omega}) \in \mathbf{\Psi}_k^{\text{best}}}$ and denote with ρ_q^{dom} the value that occurs most frequent in that sequence. Each column in Table I represents such a sequence. The value ρ_q^{dom} is the dominating value of position q of the task priorities. Similarly, for each position in $q \in \{1, \ldots, |\mathbf{\Gamma}_{\text{bus}}|\}$, we have the dominating values of the frame identifier and message priority as f_q^{dom} and ω_q^{dom}, respectively. In Table I, the second position ρ_2 of the task priorities has the dominating value 4, as indicated on the first of the last two row in the table. The dominating values are also given for the frame identifiers and message priorities.

After identifying the dominating values of each position, we count the number of solutions in $\mathbf{\Psi}_k^{\text{best}}$ that have this value in the position we are considering. Let us introduce the notation $\eta_q^{\boldsymbol{\rho}}$ for the relative occurrence frequency of the value ρ_q^{dom} in position ρ_q (i.e., the number of times the dominating value occurs in the sequence divided by $\left|\mathbf{\Psi}_k^{\text{best}}\right|$). We also refer to $\eta_q^{\boldsymbol{\rho}}$ as the *dominance factor*. Similarly, for each position in $q \in \{1, \ldots, |\mathbf{\Gamma}_{\text{bus}}|\}$, we have the dominance factors $\eta_q^{\boldsymbol{f}}$ and $\eta_q^{\boldsymbol{\omega}}$. The last row in the Table I shows the dominance factor of each dominating value. For example, the dominance factor is 0.6 in position ρ_2, which has the dominating value 4.

Having extracted the dominating values and their dominance factors from $\mathbf{\Psi}_k^{\text{best}}$, we proceed by creating the vaccines to be used in the next step. We require a *dominance threshold* $\eta^{\min}$ as a parameter ($0 \leqslant \eta^{\min} \leqslant 1$). This is used to determine the dominating values to use as vaccines. We shall create vaccines based on those dominating values with dominance factor larger than or equal to the given $\eta^{\min}$. Considering the task priorities, for each position $q \in \{1, \ldots, |\mathbf{T}_{\mathbf{\Lambda}}|\}$, if $\eta_q^{\boldsymbol{\rho}} \geqslant \eta^{\min}$, we create a vaccine $v_q^{\boldsymbol{\rho}}$ that, when applied to a member $\boldsymbol{\psi} = (\boldsymbol{\rho}, \boldsymbol{f}, \boldsymbol{\omega}) \in \mathbf{\Psi}_k^{\text{mut}}$, swaps the values of position q and the position in $\boldsymbol{\rho}$ that has the value ρ_q^{dom}. In this way, task $\sigma_{\mathbf{T}}(q)$ will be forced to have the priority ρ_q^{dom}. For the frame identifiers, if for some position $q \in \{1, \ldots, |\mathbf{\Gamma}_{\text{bus}}|\}$ the dominance factor is above the

threshold, we create a vaccine $v_q^{\boldsymbol{f}}$. When applied to a member $\boldsymbol{\psi} = (\boldsymbol{\rho}, \boldsymbol{f}, \boldsymbol{\omega}) \in \boldsymbol{\Psi}_k^{\text{mut}}$, the vaccine produces $v_q^{\boldsymbol{f}}(\boldsymbol{\psi}) = \left(\boldsymbol{\rho}, \left(f_1, \ldots, f_{q-1}, f_q^{\text{dom}}, f_{q+1}, \ldots, f_{|\boldsymbol{\Gamma}_{\text{bus}}|}\right), \boldsymbol{\omega}\right)$. For the message priorities, we produce vaccines $v_q^{\boldsymbol{\omega}}$ in the same way as the vaccines for the task priorities. Let us denote with $\mathbf{V}_k$ the set of vaccines generated from $\boldsymbol{\Psi}_k^{\text{best}}$. For each vaccine $v \in \mathbf{V}_k$, we denote with η_v the dominance factor of the corresponding position in the solution that the vaccine v modifies.

Let us consider our example in Table I again and assume that the dominance threshold is $\eta^{\min} = 0.7$. In this case, in iteration 5, only the positions ρ_3 and f_1 have dominance factors larger than 0.7 and thus two vaccines $v_3^{\boldsymbol{\rho}}$ and $v_1^{\boldsymbol{f}}$ are created. Vaccine $v_3^{\boldsymbol{\rho}}$ enforces the priority 2 in position ρ_3 (the priority of task $\sigma_{\mathbf{T}}(3)$) when applied to a member. Let us consider a member $\boldsymbol{\psi} = (\boldsymbol{\rho}, \boldsymbol{f}, \boldsymbol{\omega})$ given by row 3 in Table I. Then, after vaccination in position ρ_3, we obtain a new member $\boldsymbol{\psi}' = v_3^{\boldsymbol{\rho}}(\boldsymbol{\psi}) = ((\mathbf{1}, 4, \mathbf{2}, 3), \boldsymbol{f}, \boldsymbol{\omega})$. The bold values indicate the positions that have been changed to enforce priority 2 in position ρ_3. Vaccine $v_1^{\boldsymbol{f}}$ enforces the frame identifier 3 in position f_1, without modifying the rest of the solution. When applied to $\boldsymbol{\psi}'$, it produces the vaccinated member $v_1^{\boldsymbol{f}}(\boldsymbol{\psi}') = ((1, 4, 2, 3), (\mathbf{3}, 1, 1), \boldsymbol{\omega})$.

2) Vaccination of the Population: We shall now apply the vaccines in $\mathbf{V}_k$ on selected members from $\boldsymbol{\Psi}_k^{\text{mut}}$, which is the population generated with the crossover and mutation operators applied on $\boldsymbol{\Psi}_k$. The portion of $\boldsymbol{\Psi}_k^{\text{mut}}$ to be vaccinated is decided by the vaccination rate ζ^{vacc}. Each member $\boldsymbol{\psi} \in \boldsymbol{\Psi}_k^{\text{mut}}$ is chosen to be vaccinated with probability ζ^{vacc}. The vaccination of $\boldsymbol{\psi}$ is done in two steps: First, a set of vaccines are chosen from the set of available vaccines $\mathbf{V}_k$ in the current iteration, and second, the chosen vaccines are applied to generate the vaccinated member. We decide randomly a number of vaccines $p \in \{1, \ldots, |\mathbf{V}_k|\}$ to be chosen and applied to the solution $\boldsymbol{\psi}$. Then, we select p distinct vaccines $v_1, \ldots, v_p$ from $\mathbf{V}_k$ randomly. The probability of selecting a vaccine $v \in \mathbf{V}_k$ is larger the larger its corresponding dominance factor η_v is. Having selected the vaccines, we obtain the vaccinated member $(v_1 \circ v_2 \circ \ldots \circ v_p)(\boldsymbol{\psi})$. The members that were not selected for vaccination together with the vaccinated members constitute the vaccinated population $\boldsymbol{\Psi}_k^{\text{vacc}}$. We have now generated the next population $\boldsymbol{\Psi}_{k+1} = \boldsymbol{\Psi}_k^{\text{vacc}} \cup \{\boldsymbol{\psi}^{\text{first}}, \boldsymbol{\psi}^{\text{second}}\}$.

D. Tuning of Parameters

We have run the optimization for different values of the parameters (population size, crossover rate, mutation rate, vaccination rate, dominance threshold, and stopping condition). The population size is constant through the iterations and is chosen to be $|\boldsymbol{\Psi}_k| = 2\left(|\mathbf{T}_{\boldsymbol{\Lambda}}| + |\boldsymbol{\Gamma}_{\text{bus}}|\right)$. The crossover and mutation rates have been tuned experimentally to $\zeta^{\text{co}} = 0.65$ and $\zeta^{\text{mut}} = 0.1$,

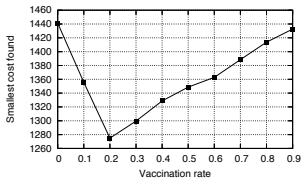

Figure 7. Tuning ζ^{vacc}

Figure 8. Tuning $\eta^{\min}$

respectively. Let us illustrate the tuning of the vaccination rate ζ^{vacc}. The dominance threshold was chosen to be $\eta^{\min} = 0.4$. Figure 7 illustrates the smallest cost (vertical axis) obtained with different values of ζ^{vacc} (horizontal axis). The figure shows that a pure genetic algorithm, achieved with $\zeta^{\text{vacc}} = 0$, results in a solution with high cost. Better solutions are found as the vaccination rate is increased. However, larger vaccination rates than 0.2 result in worse optimization.

We have also tuned the dominance threshold $\eta^{\min}$. Figure 8 shows the smallest cost (vertical axis) obtained with different values of $\eta^{\min}$ (horizontal axis), while keeping the other constant. The optimization does not find good solutions for small dominance thresholds. For these values, vaccines are created for many positions and thus the vaccination will be very similar to the mutation operator. The optimization finds better solutions as the dominance threshold increases to 0.4. As this value is increased further, less vaccines are created and therefore there are smaller number of vaccines available to choose during the vaccination of the population. Thus, for large dominance thresholds, vaccines are created for only a few positions and therefore the optimization quality decreases. We have chosen the dominance threshold as $\eta^{\min} = 0.4$ and the vaccination rate as $\zeta^{\text{vacc}} = 0.2$ for the extensive experiments reported in the next section.

VI. Experimental Results

Let us consider the optimization of a system with 35 tasks. Figure 9 shows the progress of the optimization. On the horizontal axis, we show the iteration of the algorithm, and on the vertical axis, we show the cost of the best solution obtained so far in that iteration or in previous iterations. We show the progress of both the classical genetic algorithm (GA) and the immune genetic algorithm (IGA). We can see that the immune genetic algorithm finds a better solution and also terminates faster than the classical genetic algorithm.

To study the quality improvement that can be achieved by optimizing the priorities and frame identifiers, we defined as a straightforward approach based on a rate-monotonic assignment of priorities and frame identifiers. In this approach, application tasks with small periods are given high priorities. Similarly, messages that are sent from tasks with small periods are assigned small frame identifiers (a smaller frame identifier is in general better because it leads to smaller response times).

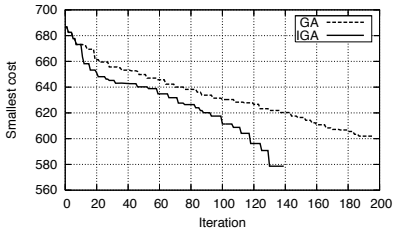

Figure 9. Optimization progress

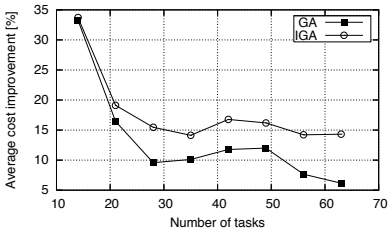

Figure 10. Achieved improvements

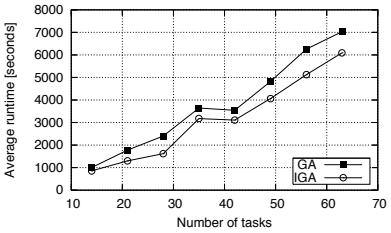

Figure 11. Optimization time

We have evaluated our optimization framework through experiments on 144 benchmarks that were generated with an in-house tool. We considered systems with 14 to 67 tasks and 2 to 9 nodes. We have run each benchmark with three approaches: the rate-monotonic approach (RM), the genetic algorithm (GA), and the immune genetic algorithm (IGA). For each of the two genetic algorithm-based approaches (GA and IGA), we computed the achieved cost-improvement relative to the RM approach. Thus, for each benchmark, we computed the relative cost improvements $\Delta C_{\mathrm{GA}} = (C_{\mathrm{RM}} - C_{\mathrm{GA}})/C_{\mathrm{RM}}$ and $\Delta C_{\mathrm{IGA}} = (C_{\mathrm{RM}} - C_{\mathrm{IGA}})/C_{\mathrm{RM}}$, where C_{RM}, C_{GA}, C_{IGA}, respectively, are the costs obtained with the RM, GA, and IGA approaches. The vertical axis in Figure 10 shows the average relative improvement, considering the benchmarks with the number of tasks given on the horizontal axis. For systems with small number of tasks, the optimization achieves a very large improvement, compared to the rate-monotonic assignment scheme. For larger number of tasks, the improvements are fairly constant. The genetic algorithm achieves a relative improvement around 10 percent. We can also see that this improvement decreases for benchmarks with large number of tasks (systems with more than 50 tasks). The vaccination step is important and leads to improvements of around 15 percent.

All experiments were run on a PC with CPU frequency 2.2 GHz, 8 Gb of RAM, and running Linux. In Figure 11, we show the average runtime in seconds as a function of the number of tasks in the benchmarks requested for. The complex optimization, involving assignments of priorities and frame identifiers, can be run with the immune genetic algorithm in less than 6000 seconds (corresponding roughly to 1.7 hours) to produce an efficient implementation for large systems of 67 tasks.

VII. Conclusions

FlexRay is a standard communication component in many modern automotive embedded systems. Task priorities and frame identifiers are important design parameters that affect the temporal behavior of the system. An efficient implementation can only be achieved by exploring and evaluating different assignments of priorities and frame identifiers. We proposed to use immune genetic algorithms to optimize average response times. In addition to the crossover and mutation operators in traditional genetic algorithms, we used a vaccination operator that leads to both faster and better optimization.

References

[1] R. Bosch GmbH. *CAN Specification Version 2.0.* 1991.

[2] FlexRay Consortium. *FlexRay Communications System. Protocol Specification Version 2.1.* 2005.

[3] J. J. Gutierrez Garcia and M. Gonzalez Harbour. Optimized priority assignment for tasks and messages in distributed real-time systems. In *Proceedings of the 3rd Workshop on Parallel and Distributed Real-Time Systems*, pp. 124–132, 1995.

[4] A. Hagiescu, U. D. Bordoloi, S. Chakraborty, P. Sampath, P. V. V. Ganesan, and S. Ramesh. Performance analysis of FlexRay-based ECU networks. In *Proceedings of the 44th Design Automation Conference*, pp. 284–289, 2007.

[5] J. H. Holland. *Adaptation in Natural and Artificial Systems.* University of Michigan Press, 1975.

[6] H. Kopetz. *Real-Time Systems—Design Principles for Distributed Embedded Applications.* Kluwer Academic, 1997.

[7] P. Pop, P. Eles, and Z. Peng. Schedulability analysis and optimization for the synthesis of multi-cluster distributed embedded systems. *IEE Computers and Digital Techniques Journal*, 150(5):303–312, 2003.

[8] P. Pop, P. Eles, Z. Peng, and T. Pop. Analysis and optimization of distributed real-time embedded systems. *ACM Transactions on Design Automation of Electronic Systems*, 11(3):593–625, 2006.

[9] T. Pop, P. Pop, P. Eles, and Z. Peng. Bus access optimisation for FlexRay-based distributed embedded systems. In *Proceedings of the Design, Automation and Test in Europe Conference*, pp. 51–62, 2007.

[10] T. Pop, P. Pop, P. Eles, Z. Peng, and A. Andrei. Timing analysis of the FlexRay communication protocol. *Real-Time Systems*, 39(1–3):205–235, 2008.

[11] S. Samii, S. Rafiliu, P. Eles, and Z. Peng. A simulation methodology for worst-case response time estimation of distributed real-time systems. In *Proceedings of the Design, Automation and Test in Europe Conference*, pp. 556–561, 2008.

[12] L. Wang and L. Jiao. The immune genetic algorithm and its convergence. In *Proceedings of ICSP*, pp. 1347–1350, 1998.

[13] L. Wang and L. Jiao. Immune evolutionary algorithms. In *Proceedings of ICSP*, pp. 1655–1662, 2000.

[14] K. E. Årzén, A. Cervin, J. Eker, and L. Sha. An introduction to control and scheduling co-design. In *Proceedings of the 39th IEEE Conference on Decision and Control*, pp. 4865–4870, 2000.