LUND UNIVERSITY

Control-Quality Optimization for Distributed Embedded Systems with Adaptive Fault Tolerance

Samii, Soheil; Bordoloi, Unmesh D.; Eles, Petru; Peng, Zebo; Cervin, Anton

Published in:
24th Euromicro Conference on Real-Time Systems (ECRTS), 2012

DOI:
10.1109/ECRTS.2012.40

2012

Link to publication

Citation for published version (APA):
Samii, S., Bordoloi, U. D., Eles, P., Peng, Z., & Cervin, A. (2012). Control-Quality Optimization for Distributed Embedded Systems with Adaptive Fault Tolerance. In *24th Euromicro Conference on Real-Time Systems (ECRTS), 2012* (pp. 68-77). IEEE - Institute of Electrical and Electronics Engineers Inc.. https://doi.org/10.1109/ECRTS.2012.40

Total number of authors:
5

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Control-Quality Optimization for Distributed Embedded Systems with Adaptive Fault Tolerance

Soheil Samii[1,2], Unmesh D. Bordoloi[2], Petru Eles[2], Zebo Peng[2], Anton Cervin[3]
[1]EIS by Semcon AB, Sweden
[2]Department of Computer and Information Science, Linköping University, Sweden
[3]Department of Automatic Control, Lund University, Sweden

***Abstract*—In this paper, we propose a design framework for distributed embedded control systems that ensures reliable execution and high quality of control even if some computation nodes fail. When a node fails, the configuration of the underlying distributed system changes and the system must adapt to this new situation by activating tasks at operational nodes. The task mapping as well as schedules and control laws that are customized for the new configuration influence the control quality and must, therefore, be optimized. The number of possible configurations due to faults is exponential in the number of nodes in the system. This design-space complexity leads to unaffordable design time and large memory requirements to store information related to mappings, schedules, and controllers. We demonstrate that it is sufficient to synthesize solutions for a small number of base and minimal configurations to achieve fault tolerance with an inherent minimum level of control quality. We also propose an algorithm to further improve control quality with a priority-based search of the set of configurations and trade-offs between task migration and replication.**

I. Introduction and Related Work

Today, many control applications are implemented on distributed embedded systems comprising multiple computation nodes that communicate over a bus. Automotive and avionic applications are two of the most prominent examples of such systems. The aggressive shrinking of transistor sizes and the environmental factors of such systems make embedded electronic devices increasingly prone to faults [1], [2]. Faults can occur due to aging, wear out, design defects, or manufacturing defects [3].

Analysis and synthesis of embedded control applications under the influence of transient and intermittent faults have been presented in literature recently [4], [5], [6]. Permanent faults, however, sustain for much longer time intervals than transient or intermittent faults, or—in the worst case—for the remaining lifetime of the system. In case a computation node fails due to a permanent fault, the *configuration* (i.e., the set of operational computation nodes) of the underlying platform changes, meaning that applications that are controlled by tasks running on this node will potentially become unstable. To avoid such situations, these tasks that were running on failed nodes must now be activated and executed on other nodes. This is achieved by implementing appropriate mechanisms for fault detection, adaptation, and reconfiguration [2], [3]. It is thus important to construct systems that are resilient to node failures and, in addition, provide as high control quality as possible with the available computation and communication resources during operation. Related approaches for embedded systems design and permanent faults, relying on task remapping by replication and migration, have been proposed recently [7], [8]. These methods are restricted to a predefined set of fault scenarios to be handled at runtime with no requirements on system performance.

To adapt to a new configuration, the system must switch to a *solution* that is customized for this new situation. A solution for a configuration comprises a mapping of tasks to the computation nodes of that configuration, a schedule (or priorities) for execution and communication, and controller parameters. Unfortunately, the total number of configurations (i.e., fault scenarios) to be considered at design time is exponential in the number of computation nodes of the platform. This complexity leads to unaffordable design time to synthesize solutions for all configurations of the system. In addition, the amount of required memory needed to store information related to all solutions is excessive and unaffordable. Towards solving these problems, we propose a framework with two main design steps.

First, we demonstrate that it is sufficient to generate solutions for a set of *base* and *minimal* configurations, which are typically few in number (Sections V and VI). These solutions can be used to operate the system in any fault scenario given by the fault-tolerance requirements, which we derive directly based on the inherent design constraints for distributed embedded control systems. The first design step comprises identification of base and minimal configurations, as well as synthesis of solutions for those configurations. This results in an implementation that satisfies the fault-tolerance requirements and, in addition, provides a certain level of control quality.

The second design step (Section IX) explores the design space judiciously to improve control quality in an incremental manner, relative to the minimum quality level provided by the first step. The generated design solutions in this second optimization step are realized based on task replication and migration as reconfiguration mechanisms. An integer linear program is solved at design time to find optimal trade-offs between migration time and memory usage of the platform.

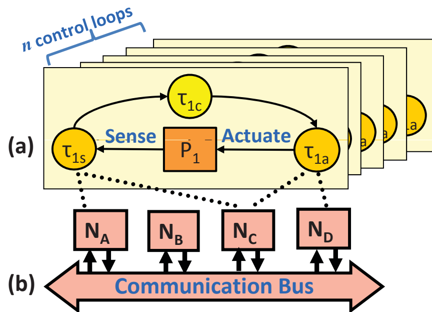

Figure 1. (a) A set of feedback-control applications running on a (b) distributed execution platform. Several periodic tasks execute on the computation nodes to read sensors, compute control signals, and write to actuators.

II. System Model

A. Feedback-Control Applications

We consider given a set of plants $\mathbf{P}$, indexed by $\mathcal{I}_\mathbf{P}$, where each plant $P_i \in \mathbf{P}$ ($i \in \mathcal{I}_\mathbf{P}$) is modeled as a continuous-time linear system [9]. Specifically, the dynamical behavior of a plant P_i is given by a system of linear differential equations

$$\dot{\boldsymbol{x}}_i(t) = A_i\boldsymbol{x}_i(t) + B_i\boldsymbol{u}_i(t) + \boldsymbol{v}_i(t), \qquad (1)$$

where the vector functions of time $\boldsymbol{x}_i$ and $\boldsymbol{u}_i$ are the plant state and controlled input, respectively. The vector $\boldsymbol{v}_i$ models plant disturbance as a white-noise stochastic process. The matrices A_i and B_i model how the plant state evolves in time depending on the current plant state and provided control input, respectively. The measured plant outputs are modeled as

$$\boldsymbol{y}_i(t) = C_i\boldsymbol{x}_i(t) + \boldsymbol{e}_i(t), \qquad (2)$$

where $\boldsymbol{e}_i$ is an additive measurement noise. The continuous-time output $\boldsymbol{y}_i$ is measured and sampled periodically and is an input to the computation and update of the control signal $\boldsymbol{u}_i$. The control law that describes the mapping from $\boldsymbol{y}_i$ to $\boldsymbol{u}_i$ is a design parameter. The C_i matrix, which often is diagonal, thus indicates those plant states that can be measured by available physical sensors of the platform (C_i is the identity matrix if all plant states can be measured). The control signal is actuated at discrete time-instants and is held constant between two updates by a hold circuit in the actuator [9].

As an example, let us consider a set of two inverted pendulums $\mathbf{P} = \{P_1, P_2\}$. Each pendulum P_i ($i \in \mathcal{I}_\mathbf{P} = \{1, 2\}$) is modeled according to Equations 1 and 2, with $A_i = \begin{bmatrix} 0 & 1 \\ g/l_i & 0 \end{bmatrix}$, $B_i = \begin{bmatrix} 0 & g/l_i \end{bmatrix}^\intercal$, and $C_i = \begin{bmatrix} 1 & 0 \end{bmatrix}$, where $g \approx 9.81$ m/s^2 and l_i are the gravitational constant and length of pendulum P_i, respectively ($l_1 = 0.2$ m and $l_2 = 0.1$ m). The two states are the pendulum position and speed, respectively. The inverted pendulum model appears often in literature as an example of control problems for unstable processes.

For each plant P_i, we have a control application $\Lambda_i = (\mathbf{T}_i, \mathbf{\Gamma}_i)$ that implements a feedback-control loop. Application Λ_i is modeled as a directed acyclic graph in which the vertices $\mathbf{T}_i$ represent computation tasks and the edges $\mathbf{\Gamma}_i \subseteq \mathbf{T}_i \times \mathbf{T}_i$ represent messages and data dependencies between tasks. Let us denote the set of control applications by $\mathbf{\Lambda}$ and index it with the index set $\mathcal{I}_\mathbf{P}$ of $\mathbf{P}$. Thus, for each $i \in \mathcal{I}_\mathbf{P}$, the pair Λ_i and P_i form a closed-loop control system. We also introduce the set of all tasks in the system as

$$\mathbf{T}_\mathbf{\Lambda} = \bigcup_{i \in \mathcal{I}_\mathbf{P}} \mathbf{T}_i.$$

Tasks are released for execution periodically. The period of each task is a design parameter and is decided mainly based on the dynamics of the control plant, the available computation and communication bandwidth, and trade-offs with the period of other applications. Figure 1(a) shows a set of control loops comprising n plants $\mathbf{P}$ with index set $\mathcal{I}_\mathbf{P} = \{1, \ldots, n\}$ and, for each plant P_i, a control application Λ_i with three tasks $\mathbf{T}_i = \{\tau_{is}, \tau_{ic}, \tau_{ia}\}$. The edges indicate the data dependencies between tasks, as well as input–output interfaces to sensors and actuators.

B. Distributed Platform

The distributed execution platform, on which the control applications run, comprises a set of computation nodes $\mathbf{N}$, indexed by $\mathcal{I}_\mathbf{N}$, which are connected to a bus. For the platform in Figure 1(b), we have $\mathbf{N} = \{N_A, N_B, N_C, N_D\}$ ($\mathcal{I}_\mathbf{N} = \{A, B, C, D\}$). We consider given a function $\Pi : \mathbf{T}_\mathbf{\Lambda} \longrightarrow 2^\mathbf{N}$ that, for each task $\tau \in \mathbf{T}_\mathbf{\Lambda}$ in the system, gives the set of computation nodes $\Pi(\tau) \subseteq \mathbf{N}$ that task τ can be mapped to. For example, tasks that read sensors or write to actuators can only be mapped to computation nodes that provide input–output interfaces to the needed sensors and actuators. Also, some tasks may require specialized instructions or hardware accelerators that are not available on all nodes. The function Π thus models mapping constraints. In Figure 1, tasks τ_{1s} and τ_{1a} may be mapped to the nodes indicated by the dotted line. Thus, we have $\Pi(\tau_{1s}) = \{N_A, N_C\}$ and $\Pi(\tau_{1a}) = \{N_C, N_D\}$. We consider that task τ_{1c} can be mapped to any of the four nodes in the platform (i.e., $\Pi(\tau_{1c}) = \mathbf{N}$); we have omitted the many dashed lines for task τ_{1c} to obtain a clear illustration. For each task $\tau \in \mathbf{T}_\mathbf{\Lambda}$ and each computation node $N \in \Pi(\tau)$, we consider that the best-case and worst-case execution times are given when task τ executes on node N.

At any moment in time, the system has a set of computation nodes $\mathbf{X} \subseteq \mathbf{N}$ that are available. The remaining nodes $\mathbf{N} \setminus \mathbf{X}$ have failed and are not available for computation. We shall refer to $\mathbf{X}$ as a *configuration* of the distributed platform. The complete set of configurations is the power set $\mathcal{X} = 2^\mathbf{N}$ of $\mathbf{N}$ and is a partially ordered set under the subset relation. The partial order of configurations is shown in Figure 2 as a Hasse diagram of configurations for our example with four computation nodes in Figure 1 (note that we have excluded the empty configuration $\emptyset$ because it is of no interest to consider the scenario where all nodes have failed). For example, the configuration $\{N_A, N_B, N_C\}$ indicates that N_D has

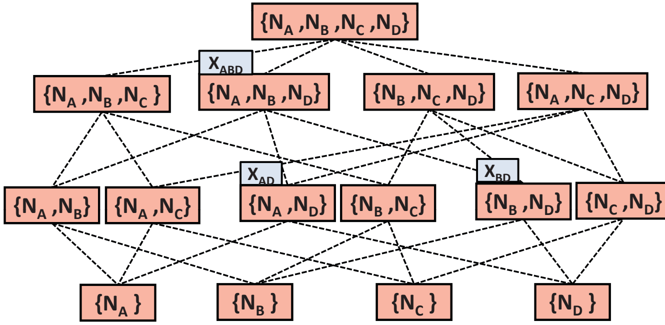

Figure 2. Hasse diagram of configurations for a system with four nodes. The set of possible configurations due to faults is partially ordered under the subset relation.

failed and only the other three nodes are operational and available for computation. For the communication, we consider that the communication protocol of the system ensures fault-tolerance for messages by different means of redundancy [10], [2].

The platform has appropriate mechanisms for fault detection. The failure of a node must be detected and all remaining operational nodes must know about such failures [2]. In addition, when a node has been repaired it is detected by the other nodes in the system. This allows each operational node to know about the current configuration. Adaptation due to failed or repaired nodes involves switching schedules and control algorithms that are optimized for the available resources in the new configuration (Section IX). This information is stored in the nodes of the platform [11], [2]. Another phase during system reconfiguration is task migration [8] that takes place when tasks running on failed nodes must be activated at other nodes in the system. The system has the ability to migrate tasks to other nodes in the platform [8]. Each node stores information regarding those tasks that it must migrate on the bus when the system is adapting to a new configuration. This information is generated at design time (Section IX-B).

III. Control Quality and Design Optimization

A. Metric for Control Quality

Considering one of the controlled plants P_i in isolation, the goal is to control the plant states in the presence of the plant disturbance $\boldsymbol{v}_i$ and measurement error $\boldsymbol{e}_i$. We use a quadratic control cost [9] as a quality and performance metric for control applications. This includes a cost for the error in the plant state and the cost of changing the control signals (the latter cost can be related to the amount of energy spent by the actuators). Specifically, the quality of a controller for plant P_i is given by the quadratic cost

$$J_i = \lim_{T\to\infty} \frac{1}{T} \mathrm{E} \left\{ \int_0^T \begin{bmatrix} \boldsymbol{x}_i \\ \boldsymbol{u}_i \end{bmatrix}^{\mathsf{T}} Q_i \begin{bmatrix} \boldsymbol{x}_i \\ \boldsymbol{u}_i \end{bmatrix} dt \right\}. \quad (3)$$

A small cost indicates high control quality, and vice versa. The weight matrix Q_i is used to model weights of individual components of the state and input vectors, as well as to indicate the importance relative to other control applications in the system. An infinite cost indicates an unstable closed-loop system.

The cost in Equation 3 is a common quality metric for control systems [9]. It is a function of the sampling period of the controller, the control law, and the characteristics of the delay between sampling and actuation. This delay is complex and is induced by the schedule and mapping of the tasks on the distributed platform [12], [13]. We use the Jitterbug toolbox [12] to compute the control cost J_i by providing as inputs the controller and the characteristics of the sampling–actuation delay.

B. Task Mapping for Configurations

Let us define the *mapping* of the task set $\mathbf{T}_{\mathbf{\Lambda}}$ onto a configuration $\mathbf{X}$ as a function $\mathsf{map}_{\mathbf{X}} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{X}$. For each task $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$, $\mathsf{map}_{\mathbf{X}}(\tau)$ is the computation node that executes task τ when the system configuration is $\mathbf{X}$. It is required that the mapping constraints are considered, meaning that $\mathsf{map}_{\mathbf{X}}(\tau) \in \Pi(\tau)$ for each $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$. The mapping affects the delay characteristics indirectly through task and message scheduling, thus affecting the control quality of the running applications. It is thus of great importance to optimize task mapping, schedules, and control laws to obtain a customized solution with high control quality in a given configuration.

For a given configuration $\mathbf{X} \in \mathcal{X}$, we use our design tool [13], [14] for integrated control, scheduling, and mapping of distributed embedded systems. The synthesized design solution comprises a task mapping $\mathsf{map}_{\mathbf{X}} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{X}$, an execution and communication schedule (task and message priorities if the scheduling policy of the platform is based on fixed priorities), and controllers (periods and control laws). The optimization objective is to minimize the overall control cost

$$J^{\mathbf{X}} = \sum_{i\in\mathcal{I}_{\mathbf{P}}} J_i, \quad (4)$$

which indicates maximization of the total control quality of the system, under the consideration that only the nodes in $\mathbf{X}$ are operational, and that the other nodes $\mathbf{N} \setminus \mathbf{X}$ in the system have failed.

We denote with $\mathsf{mem}_d^{\mathbf{X}}$ the amount of memory[1] required on node N_d ($d \in \mathcal{I}_{\mathbf{N}}$) to store information related to the mapping, schedule, periods, and control laws that are optimized for configuration $\mathbf{X}$. This memory consumption is given as an output of our design tool and is considered when formulating the memory constraints in Section IX-B.

IV. Classification of Configurations

A. Example of Configurations

Let us consider our example in Figure 1. Task τ_{1s} reads sensors and τ_{1a} writes to actuators. Task τ_{1c} does not perform input–output operations and can be executed

[1]We model the memory consumption as an integer representing the number of units of physical memory that are required to store the design solution.

on any node in the platform. Sensors can be read by nodes N_A and N_C, whereas actuation can be performed by nodes N_C and N_D. The mapping constraints for the tasks are thus given by $\Pi(\tau_{1s}) = \{N_A, N_C\}$, $\Pi(\tau_{1c}) = \mathbf{N}$, and $\Pi(\tau_{1a}) = \{N_C, N_D\}$. The same mapping constraints and discussion hold for the other control applications Λ_i ($i = 2, \ldots, n$). Thus, in the remainder of this example, we shall restrict the discussion to application Λ_1.

First, let us consider the initial scenario in which all computation nodes are operational and are executing one or several tasks each. The system is thus in configuration $\mathbf{N} = \{N_A, N_B, N_C, N_D\}$ (see Figure 2) and we assume that the actuator task τ_{1a} executes on node N_C in this configuration. Consider now that N_C fails and the system reaches configuration $\mathbf{X}_{ABD} = \{N_A, N_B, N_D\}$. Task τ_{1a} must now execute on N_D in this new configuration, because actuation can only be performed by nodes N_C and N_D. According to the mapping constraints given by Π, there exists a possible mapping for each task in configuration $\mathbf{X}_{ABD}$, because $\mathbf{X}_{ABD} \cap \Pi(\tau) \neq \emptyset$ for each task $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$. We refer to such configurations as *feasible* configurations. In feasible configuration, there is at least one node that can execute a given task (without violation of the imposed mapping constraints).

If the system is in configuration $\mathbf{X}_{ABD}$ and node N_A fails, a new configuration $\mathbf{X}_{BD} = \{N_B, N_D\}$ is reached. Because task τ_{1s} cannot be mapped to any node in the new configuration, we say that $\mathbf{X}_{BD}$ is an *infeasible* configuration (i.e., we have $\Pi(\tau_{1s}) \cap \mathbf{X}_{BD} = \emptyset$). If, on the other hand, node N_B fails in configuration $\mathbf{X}_{ABD}$, the system reaches configuration $\mathbf{X}_{AD} = \{N_A, N_D\}$. In this configuration, tasks τ_{1s} and τ_{1a} must execute on N_A and N_D, respectively. Task τ_{1c} may run on either N_A or N_D. Thus, $\mathbf{X}_{AD}$ is a feasible configuration, because it is possible to map each task to a node that is both operational and allowed according to the given mapping restrictions. We observe that if either of the nodes in $\mathbf{X}_{AD}$ fails, the system reaches an infeasible configuration. We shall refer to configurations like $\mathbf{X}_{AD}$ as *base* configurations. Note that any configuration that is a superset of the base configuration $\mathbf{X}_{AD}$ is a feasible configuration. By considering the mapping constraints, we observe that the only other base configuration in this example is $\{N_C\}$ (node N_C may execute any task). The set of base configurations for our example system is thus

$$\mathcal{X}^{\text{base}} = \{\{N_A, N_D\}, \{N_C\}\}.$$

Let us consider that design solutions are generated for the two base configurations in $\mathcal{X}^{\text{base}}$. Considering Figure 2 again, we note that the mapping for base configuration $\{N_A, N_D\}$, including the produced schedule, task periods, and control laws, can be used to operate the system in the feasible configurations $\{N_A, N_B, N_C, N_D\}$, $\{N_A, N_B, N_D\}$, and $\{N_A, N_C, N_D\}$. This is done by merely using the two nodes in the base configuration (i.e., N_A and N_D), even though more nodes are operational in the mentioned feasible configurations. Similarly, base configuration $\{N_C\}$ covers another subset of the feasible configurations. By studying Figure 2 again, note that the two base configurations cover all feasible configurations together (there is a path to any feasible configuration, starting from a base configuration).

By generating a mapping (as well as customized schedules, periods, and control laws) for each base configuration, and considering that tasks are stored in the memory of the corresponding computation nodes to realize the base configuration mappings, the system can tolerate any sequence of node failures that lead the system to any feasible configuration. Thus, a necessary and sufficient step in the design phase (in terms of fault tolerance) is to identify the set of base configurations and to generate design solutions for them. It can be the case that the computation capacity is insufficient in some base configurations, because of the small number of operational nodes. We shall discuss this issue in Section VI.

B. Formal Definitions

We consider that the mapping constraint $\Pi : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow 2^{\mathbf{N}}$ is given, meaning that $\Pi(\tau)$ defines the set of computation nodes that task $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$ may execute on. A configuration $\mathbf{X} \in \mathcal{X}$ is defined as a *feasible* configuration if $\mathbf{X} \cap \Pi(\tau) \neq \emptyset$ for each task $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$. The set of feasible configurations is denoted $\mathcal{X}^{\text{feas}}$.

For an infeasible configuration $\mathbf{X} \in \mathcal{X} \setminus \mathcal{X}^{\text{feas}}$, there exists at least one task that due to the given mapping constraints cannot execute on any computation node in $\mathbf{X}$ (i.e., $\mathbf{X} \cap \Pi(\tau) = \emptyset$ for some $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$).

A *base configuration* $\mathbf{X}$ is a feasible configuration for which the failure of any computation node $N \in \mathbf{X}$ results in an infeasible configuration $\mathbf{X} \setminus \{N\}$. The set of base configurations is thus defined as

$$\mathcal{X}^{\text{base}} = \{\mathbf{X} \in \mathcal{X}^{\text{feas}} : \mathbf{X} \setminus \{N\} \notin \mathcal{X}^{\text{feas}} \text{ for each } N \in \mathbf{X}\}.$$

The set of configurations $\mathcal{X} = 2^{\mathbf{N}}$ is thus partitioned into disjoint sets of feasible and infeasible configurations. Some of the feasible configurations form a set of base configurations, which represents the boundary between the set of feasible and infeasible configurations.

In the next section, we shall discuss an approach to identify the set of base configurations. In the ideal case, solutions for base configurations are synthesized, enabling the system to operate in any feasible configuration. If not all base configurations allow for acceptable solutions to be synthesized, we construct solutions for a set of *minimal* configurations in Section VI to cover as many feasible configurations as possible. Such situations occur if the computation capacity is too restricted to achieve stability in certain base configurations.

V. Identification of Base Configurations

We shall present an algorithm that constructs the set of base configurations $\mathcal{X}^{\text{base}}$ directly based on the mapping constraint $\Pi : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow 2^{\mathbf{N}}$. Without loss of generality, we shall assume that the function Π is injective (i.e., $\Pi(\tau_i) \neq \Pi(\tau_j)$ for $\tau_i \neq \tau_j$). If this is not the case, then,

for the purpose of finding the set of base configurations, it is an equivalent problem to study an injective function $\Pi' : \mathbf{T}'_{\mathbf{\Lambda}} \longrightarrow 2^{\mathbf{N}}$ as a mapping constraint, where $\mathbf{T}'_{\mathbf{\Lambda}} \subset \mathbf{T}_{\mathbf{\Lambda}}$. Further in that case, it is required that, for each $\tau \in \mathbf{T}_{\mathbf{\Lambda}} \setminus \mathbf{T}'_{\mathbf{\Lambda}}$, there exists exactly one $\tau' \in \mathbf{T}'_{\mathbf{\Lambda}}$ for which $\Pi(\tau) = \Pi'(\tau')$. Finally, in the following discussion, $\mathbf{T}'_{\mathbf{\Lambda}}$ and Π' replace $\mathbf{T}_{\mathbf{\Lambda}}$ and Π, respectively.

We construct the set of base configurations starting from the tasks that have the most restrictive mapping constraints. Towards this, let us consider a bijection

$$\sigma : \{1, \ldots, |\mathbf{T}_{\mathbf{\Lambda}}|\} \longrightarrow \mathbf{T}_{\mathbf{\Lambda}},$$

where $|\Pi(\sigma(k))| \leqslant |\Pi(\sigma(k+1))|$ for $1 \leqslant k < |\mathbf{T}_{\mathbf{\Lambda}}|$. This order of the tasks is considered during the construction of the set of base configurations $\mathcal{X}^{\text{base}}$. The construction is based on a function

$$\mathsf{construct} : \{1, \ldots, |\mathbf{T}_{\mathbf{\Lambda}}|\} \longrightarrow 2^{\mathcal{X}},$$

where $\mathsf{construct}(k)$ returns a set of configurations that include the base configurations of the system when considering the mapping constraints for only tasks $\sigma(1), \ldots, \sigma(k)$. We shall give a recursive definition of the function $\mathsf{construct}$. For the base case, we define

$$\mathsf{construct}(1) = \bigcup_{N \in \Pi(\sigma(1))} \{N\}.$$

Before we define $\mathsf{construct}(k)$ for $1 < k \leqslant |\mathbf{T}_{\mathbf{\Lambda}}|$, let us define a function

$$\mathsf{feasible} : \mathcal{X} \times \{1, \ldots, |\mathbf{T}_{\mathbf{\Lambda}}|\} \longrightarrow 2^{\mathcal{X}}$$

as

$$\mathsf{feasible}(\mathbf{X}, k) = \{\mathbf{X}\} \quad (5)$$

if $\mathbf{X} \cap \Pi(\sigma(k)) \neq \emptyset$ (i.e., configuration $\mathbf{X}$ already includes an allowed computation node for task $\sigma(k)$) and

$$\mathsf{feasible}(\mathbf{X}, k) = \bigcup_{N \in \Pi(\sigma(k))} \mathbf{X} \cup \{N\} \quad (6)$$

otherwise. If $\mathbf{X}$ contains a computation node that task $\sigma(k)$ can execute on, then $\mathsf{feasible}(\mathbf{X}, k)$ does not add additional nodes to $\mathbf{X}$ (Equation 5). If not, however, then $\mathsf{feasible}(\mathbf{X}, k)$ extends $\mathbf{X}$ in several directions given by the set of nodes $\Pi(\sigma(k))$ that task $\sigma(k)$ may execute on (Equation 6). Now, we define recursively

$$\mathsf{construct}(k) = \bigcup_{\mathbf{X} \in \mathsf{construct}(k-1)} \mathsf{feasible}(\mathbf{X}, k)$$

for $1 < k \leqslant |\mathbf{T}_{\mathbf{\Lambda}}|$. The set $\mathsf{construct}(k)$ thus comprises configurations for which it is possible to execute the tasks $\{\sigma(1), \ldots, \sigma(k)\}$ according to the mapping constraints induced by Π.

We know by construction that $\mathcal{X}^{\text{base}} \subseteq \mathsf{construct}(|\mathbf{T}_{\mathbf{\Lambda}}|)$. We also know that $\mathsf{construct}(|\mathbf{T}_{\mathbf{\Lambda}}|)$ does not contain infeasible configurations. A pruning of the set $\mathsf{construct}(|\mathbf{T}_{\mathbf{\Lambda}}|)$ must be performed to identify feasible configurations $\mathsf{construct}(|\mathbf{T}_{\mathbf{\Lambda}}|) \setminus \mathcal{X}^{\text{base}}$ that are not base configurations.

VI. Minimal Configurations

By definition, it is not possible to operate the system in infeasible configurations, because at least one task cannot be executed in such situations. In this section, we shall discuss the synthesis of mandatory solutions that are required to achieve system operation in feasible configurations. The first approach is to synthesize solutions for each base configuration of the system. It can be the case, however, that no solution can be found for some base configurations; the control cost in Equation 4 is infinite in such cases, indicating that at least one control loop is unstable. If a solution cannot be found for a certain configuration, this means that the computation capacity of the system is insufficient in that configuration. In such cases, we shall progressively synthesize solutions for configurations with additional computation nodes.

We first synthesize a solution for each base configuration $\mathbf{X} \in \mathcal{X}^{\text{base}}$. If a solution could be found—the control cost $J^{\mathbf{X}}$ is finite—then that solution can be used to operate the system in any feasible configuration $\mathbf{X}' \in \mathcal{X}^{\text{feas}}$ for which $\mathbf{X} \subseteq \mathbf{X}'$. If a solution cannot be found for base configuration $\mathbf{X}$, we proceed by synthesizing solutions for configurations with one additional computation node. This process is repeated as long as solutions cannot be found. Let us now outline such an approach.

During the construction of solutions to configurations, we shall maintain two sets $\mathcal{X}^{\min}$ and $\mathcal{X}^{*}$ with initial values $\mathcal{X}^{\min} = \emptyset$ and $\mathcal{X}^{*} = \mathcal{X}^{\text{base}}$. The set $\mathcal{X}^{\min}$ shall contain the configurations that have been synthesized successfully: Their design solutions have finite control cost and stability is guaranteed. The set $\mathcal{X}^{*}$ contains configurations that are yet to be synthesized. The following steps are repeated as long as $\mathcal{X}^{*} \neq \emptyset$.

1) Select any configuration $\mathbf{X} \in \mathcal{X}^{*}$.
2) Synthesize a solution for $\mathbf{X}$. This results in the control cost $J^{\mathbf{X}}$.
3) If $J^{\mathbf{X}} < \infty$, update $\mathcal{X}^{\min}$ according to
$$\mathcal{X}^{\min} \longleftarrow \mathcal{X}^{\min} \cup \{\mathbf{X}\}, \quad (7)$$
otherwise update $\mathcal{X}^{*}$ as
$$\mathcal{X}^{*} \longleftarrow \mathcal{X}^{*} \cup \bigcup_{N \in \mathbf{N} \setminus \mathbf{X}} (\mathbf{X} \cup \{N\}). \quad (8)$$
4) Remove $\mathbf{X}$ from $\mathcal{X}^{*}$ by the update
$$\mathcal{X}^{*} \longleftarrow \mathcal{X}^{*} \setminus \{\mathbf{X}\}.$$
5) If $\mathcal{X}^{*} \neq \emptyset$, go back to Step 1.

In the first two steps, configurations can be chosen for synthesis in any order from $\mathcal{X}^{*}$. In Step 3, we observe that the set $\mathcal{X}^{*}$ becomes smaller as long as solutions can be synthesized with finite control cost (Equation 7). If a solution for a certain configuration cannot be synthesized (i.e., the synthesis framework returns an infinite control cost, indicating an unstable control system), we consider configurations with one additional computation node to increase the possibility to find solutions (Equation 8). In the last two steps, the synthesized configuration is removed from the set $\mathcal{X}^{*}$ and the synthesis is repeated for another configuration in $\mathcal{X}^{*}$.

The configurations for which solutions could be synthesized form a set of *minimal* feasible configurations. The set of minimal configurations $\mathcal{X}^{\min}$ is thus defined

by Steps 1–5. A configuration $\mathbf{X} \in \mathcal{X}^{\min}$ is minimal in the sense that it is either a base configuration or it is a feasible configuration with minimal number of nodes that include the nodes in a base configuration that could not be synthesized due to insufficient computation capacity of the platform. For each minimal configuration $\mathbf{X} \in \mathcal{X}^{\min}$, we consider that each node $N \in \mathbf{X}$ stores all tasks $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$ for which $\mathsf{map}_{\mathbf{X}}(\tau) = N$; that is, we consider that tasks are stored permanently on nodes to realize mappings for minimal configurations. Further, we consider that all information (e.g., periods, control laws, and schedules) that is needed to switch to solutions for minimal configurations at runtime is stored in the memory of computation nodes.

The set of feasible configurations for which the system is operational with our solution is

$$\mathcal{X}^{\text{oper}} = \bigcup_{\mathbf{X} \in \mathcal{X}^{\min}} \{\mathbf{X}' \in \mathcal{X}^{\text{feas}} : \mathbf{X} \subseteq \mathbf{X}'\} \tag{9}$$

and it includes the minimal configurations $\mathcal{X}^{\min}$, as well as feasible configurations that are covered by a minimal configuration. The system is not able to operate in the feasible configurations $\mathcal{X}^{\text{feas}} \setminus \mathcal{X}^{\text{oper}}$—this set represents the border between base and minimal configurations—because of insufficient computation capacity of the platform. A direct consequence of the imposed mapping constraints is that the system cannot operate when it is in any infeasible configuration in $\mathcal{X} \setminus \mathcal{X}^{\text{feas}}$. Infeasible configurations, as well as feasible configurations not covered by minimal configurations, are identified by our approach. To tolerate particular fault scenarios that lead the system to configurations in

$$\left(\mathcal{X} \setminus \mathcal{X}^{\text{feas}}\right) \cup \left(\mathcal{X}^{\text{feas}} \setminus \mathcal{X}^{\text{oper}}\right),$$

the problem of insufficient computation capacity has to be solved by considering complementary fault-tolerance techniques (e.g., hardware replication). The system remains operational in all other configurations $\mathcal{X}^{\text{oper}}$ by using the solutions generated for minimal configurations. As a special case, we have $\mathcal{X}^{\min} = \mathcal{X}^{\text{base}}$ if solutions to all base configurations could be synthesized. In that case, we have $\mathcal{X}^{\text{oper}} = \mathcal{X}^{\text{feas}}$, meaning that the system is operational in all feasible configurations.

VII. Motivational Example for Optimization

The synthesis of a set of minimal configurations $\mathcal{X}^{\min}$ in the previous section results in a solution that covers all fault scenarios that lead the system to a configuration in $\mathcal{X}^{\text{oper}}$ (Equation 9). The synthesis of minimal configurations provides not only fault tolerance for the configurations $\mathcal{X}^{\text{oper}}$ but also stability and a certain level of control quality. We shall in this section illustrate and motivate additional improvements in control quality.

A. Improved Solutions for Feasible Configurations

Let us resume our example in Section IV-A by considering synthesis of additional configurations than the minimal configurations. We have considered three control applications for three inverted pendulums (i.e., $n = 3$ in Figure 1). Let us consider the set of base configurations $\mathcal{X}^{\text{base}} = \{\{N_A, N_D\}, \{N_C\}\}$. Considering that solutions for the two base configurations have been synthesized, and that these solutions have finite control costs, we note that the set of minimal configurations is $\mathcal{X}^{\min} = \mathcal{X}^{\text{base}}$. We thus have $\mathcal{X}^{\text{oper}} = \mathcal{X}^{\text{feas}}$, meaning that the system can operate in any feasible configuration with the solutions for minimal configurations. Let us also consider that a customized solution (mapping, schedule, and controllers) has been synthesized for the configuration in which all nodes are operational. This solution exploits the full computation capacity of the platform to achieve as high control quality as possible. Note that all feasible configurations are handled with solutions for the two base configurations.

Table I
Control costs for several configurations.

Configuration $\mathbf{X}$	Control cost $J^{\mathbf{X}}$
$\{N_A, N_D\}$	5.2
$\{N_C\}$	7.4
$\{N_A, N_B, N_C, N_D\}$	3.1
$\{N_A, N_B, N_C\}$	4.3

Table II
Mapping for two configurations and three applications.

Configuration	N_A	N_B	N_C	N_D
$\{N_A, N_B, N_C\}$	$\tau_{1s}, \tau_{2s}, \tau_{3s}$	$\tau_{1c}, \tau_{2c}, \tau_{3c}$	$\tau_{1a}, \tau_{2a}, \tau_{3a}$	–
$\{N_C\}$	–	–	$\tau_{1s}, \tau_{1c}, \tau_{1a}, \tau_{2s}, \tau_{2c}, \tau_{2a}, \tau_{3s}, \tau_{3c}, \tau_{3a}$	–

We shall now improve control quality by additional synthesis of configurations. Towards this, we have synthesized solutions for the two minimal configurations, as well as configuration $\{N_A, N_B, N_C\}$. Table I shows the obtained control costs defined by Equation 4. Considering that a solution for $\{N_A, N_B, N_C\}$ would not have been generated, then in that configuration the system can only run with the solution for the minimal configuration $\{N_C\}$ with a cost of 7.4. By generating a customized solution, however, we can achieve a better control quality in that configuration according to the obtained cost 4.3—a cost improvement of 3.1 (the control quality is improved by 41.4 percent). By synthesizing additional feasible configurations, we can obtain additional control-quality improvements—however, at the expense of the total synthesis time of all solutions.

B. Mapping Realization

Once a solution for a configuration—not a minimal configuration—has been synthesized, it must be verified whether it is possible for the system to adapt to this solution at runtime. Thus, for the additional mapping of configuration $\{N_A, N_B, N_C\}$ in our example, we must check whether the mapping can be realized if the system is in configuration $\{N_A, N_B, N_C, N_D\}$ and node N_D fails. In Table II, we show the mapping for this configuration, as well as the mapping of its corresponding minimal configuration $\{N_C\}$. For the minimal configurations, we consider that the tasks are stored on the corresponding

computation nodes. For example, the tasks in the column for N_C, corresponding to the minimal configuration, are stored on node N_C. Let us consider the mapping of the tasks to the configuration $\{N_A, N_B, N_C\}$. We note that all tasks that are needed to realize the mapping for node N_C are already stored on that node. Nodes N_A and N_B, however, do not store the tasks that are needed to realize the mapping for configuration $\{N_A, N_B, N_C\}$. When switching to the solution for this configuration—from configuration $\{N_A, N_B, N_C, N_D\}$—the tasks for nodes N_A and N_B need to be migrated[2] from node N_C.

Because any feasible configuration in $\mathcal{X}^{\text{oper}}$ is covered by a minimal configuration, which realizes its mapping by storing tasks in memory of the operational nodes, there is always at least one operational node that stores a certain task for a given feasible configuration. The migration time cannot exceed specified bounds, in order to guarantee stability. Hence, for our example, if the migration time for tasks τ_{1s}, τ_{2s}, τ_{3s}, τ_{1c}, τ_{2c}, and τ_{3c} satisfies the specified bound, the system can realize the solution for configuration $\{N_A, N_B, N_C\}$ at runtime.

If the time required to migrate the required tasks at runtime exceeds the given bounds, then the solution for the minimal configuration $\{N_C\}$ is used at runtime with control cost 7.4. In that case, the operational nodes N_A and N_B are not utilized. Alternatively, more memory can be used to store additional tasks on nodes N_A and N_B, in order to realize the mapping at runtime without or with reduced task migration. In this way, we avoid the excessive amount of migration time and we can realize the mapping, although at the cost of larger required memory space to achieve the better control cost of 4.3 in configuration $\{N_A, N_B, N_C\}$. We shall in the remainder of this paper study the trade-off among control quality, memory cost, and synthesis time.

VIII. Problem Formulation

Given is a distributed platform with computation nodes $\mathbf{N}$, a set of plants $\mathbf{P}$, and their control applications $\mathbf{\Lambda}$. We consider that a task mapping $\mathsf{map}_{\mathbf{X}} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{X}$, as well as corresponding schedules and controllers, have been generated for each minimal configuration $\mathbf{X} \in \mathcal{X}^{\text{min}}$ as discussed in Section VI. We consider that tasks are stored permanently on appropriate computation nodes to realize the task mappings for the minimal configurations (i.e., no task migration is needed at runtime to adapt to solutions for minimal configurations). Thus, to realize the mappings for minimal configurations, each task $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$ is stored on nodes

$$\bigcup_{\mathbf{X} \in \mathcal{X}^{\text{min}}} \{\mathsf{map}_{\mathbf{X}}(\tau)\}.$$

[2]During task migration, the program state does not need to be transferred (because of the feedback mechanism of control applications, the state is automatically restored when task migration has completed).

The set of tasks that are stored on node $N_d \in \mathbf{N}$ is

$$\mathbf{T}^{(d)} = \bigcup_{\mathbf{X} \in \mathcal{X}^{\text{min}}} \{\tau \in \mathbf{T}_{\mathbf{\Lambda}} : \mathsf{map}_{\mathbf{X}}(\tau) = N_d\}. \qquad (10)$$

In addition, the inputs specific to the optimization step discussed in this section are

- the time $\mu(\tau)$ required to migrate task τ from a node to any other node in the platform;
- the maximum amount of migration time μ_i^{max} for plant P_i (this constraint is based on the maximum amount of time that a plant P_i can stay in open loop without leading to instability [15] or degradation of control quality below a specified threshold, as well as the actual time to detect faults [2], [3]);
- the memory space $\mathsf{mem}_d(\tau)$ required to store task $\tau \in \mathbf{T}_{\mathbf{\Lambda}}$ on node N_d ($d \in \mathcal{I}_{\mathbf{N}}$);
- the additional available memory $\mathsf{mem}_d^{\text{max}}$ of each node N_d in the platform (note that this does not include the memory consumed for the minimal configurations, as these are mandatory to implement and sufficient dedicated memory is assumed to be provided); and
- the failure probability $p(N)$ per time unit for each node $N \in \mathbf{N}$.

The failure probability $p(N)$ depends on the mean time to failure (MTTF) of the computation node. The MTTF is decided by the technology of the production process, the ambient temperature of the components, and voltage or physical shocks that the components may suffer in the operational environment of the system [3].

The decision variables of the optimization problem are a subset of configurations $\mathcal{X}^{\text{impl}} \subseteq \mathcal{X}^{\text{oper}} \setminus \mathcal{X}^{\text{min}}$ and a mapping $\mathsf{map}_{\mathbf{X}}$, schedule, and controllers for each $\mathbf{X} \in \mathcal{X}^{\text{impl}}$. Thus, in addition to the minimal configurations, we generate mappings for the other feasible configurations $\mathcal{X}^{\text{impl}}$. We require that $\mathbf{N} \in \mathcal{X}^{\text{impl}}$, which means that it is mandatory to generate solutions for the case when all nodes in the system are operational.

Let us now define the cost that characterizes the overall control quality of the system in any feasible configuration based on the solutions (mappings, schedules, and controllers) for the selected set of configurations. We shall associate a cost $J^{\mathbf{X}}$ for each feasible configuration $\mathbf{X} \in \mathcal{X}^{\text{oper}}$. If $\mathbf{X} \in \mathcal{X}^{\text{min}} \cup \mathcal{X}^{\text{impl}}$, a customized mapping for that configuration has been generated with a cost $J^{\mathbf{X}}$ given by Equation 4. If $\mathbf{X} \notin \mathcal{X}^{\text{min}} \cup \mathcal{X}^{\text{impl}}$ and $\mathbf{X} \in \mathcal{X}^{\text{oper}}$, then at runtime the system uses the mapping of a configuration $\mathbf{X}'$ for which $\mathbf{X}' \in \mathcal{X}^{\text{min}} \cup \mathcal{X}^{\text{impl}}$ and $\mathbf{X}' \subset \mathbf{X}$. It is guaranteed that such a configuration $\mathbf{X}'$ can be found in the set of minimal configurations $\mathcal{X}^{\text{min}}$ (Equation 9). If such a configuration is also included in $\mathcal{X}^{\text{impl}}$, then the control quality is better than in the corresponding minimal configuration because of better utilization of the operational computation nodes. Thus, for the case $\mathbf{X} \in \mathcal{X}^{\text{oper}} \setminus (\mathcal{X}^{\text{min}} \cup \mathcal{X}^{\text{impl}})$, the cost of the

feasible configuration $\mathbf{X}$ is

$$J^{\mathbf{X}} = \min_{\substack{\mathbf{X}' \in \mathcal{X}^{\min} \cup \mathcal{X}^{\text{impl}} \\ \mathbf{X}' \subset \mathbf{X}}} J^{\mathbf{X}'}, \tag{11}$$

which means that the best functionally correct solution—in terms of control quality—is used to operate the system in configuration $\mathbf{X}$. The cost to minimize when selecting the set of additional feasible configurations $\mathcal{X}^{\text{impl}} \subseteq \mathcal{X}^{\text{oper}} \setminus \mathcal{X}^{\min} \setminus \{\mathbf{N}\}$ to synthesize is defined

$$J = \sum_{\mathbf{X} \in \mathcal{X}^{\text{oper}} \setminus \mathcal{X}^{\min} \setminus \{\mathbf{N}\}} p^{\mathbf{X}} J^{\mathbf{X}}, \tag{12}$$

where $p^{\mathbf{X}}$ is the probability of node failures that lead the system to configuration $\mathbf{X}$ (we shall discuss the computation of this probability in Equation 13). Towards this, we shall consider the given failure probability $p(N)$ of each computation node $N \in \mathbf{N}$.

The cost in Equation 12 characterizes the control quality of the system as a function of the additional feasible configurations for which solutions have been synthesized. If solutions are available only for the set of minimal configurations, the system tolerates all node failures that lead the system to a configuration in $\mathcal{X}^{\text{oper}}$—however, at a large cost J in Equation 12. This is because other feasible configurations operate at runtime with solutions of minimal configurations. In those situations, not all operational computation nodes are utilized, at the cost of reduced overall control quality. By synthesizing solutions for additional feasible configurations in $\mathcal{X}^{\text{oper}} \setminus \mathcal{X}^{\min} \setminus \{\mathbf{N}\}$, the cost in Equation 12 is reduced (i.e., the overall control quality is improved) due to the cost reduction in the terms related to the selected set of configurations.

IX. Optimization Approach

Figure 3 shows an overview of our proposed design approach. The first component, which we discussed in Sections V and VI, is the identification of base configurations and synthesis of minimal configurations (labeled as "fault-tolerant design" in the figure). The second component (labeled as "optimization") is the topic of this section and is our proposed solution to the problem in Section VIII. The selection and synthesis of additional feasible configurations is described in Section IX-A. For each synthesized feasible configuration, it must be checked whether the solution can be realized with regard to the memory consumption in the platform and the amount of task migration required at runtime. Memory and migration trade-offs, as well as memory-space and migration-time constraints, are presented in Section IX-B.

A. Exploration of the Set of Configurations

Our optimization heuristic aims to minimize the cost in Equation 12 and is based on a priority-based search of the Hasse diagram of configurations. The priorities are computed iteratively as a step of the optimization process based on probabilities for the system to reach the different configurations. The heuristic belongs to the class of anytime algorithms, meaning that it can be stopped at any point in time and return a feasible solution. This is due to that minimal configurations already have been synthesized and fault tolerance is achieved. The overall quality of the system is improved as more optimization time is invested.

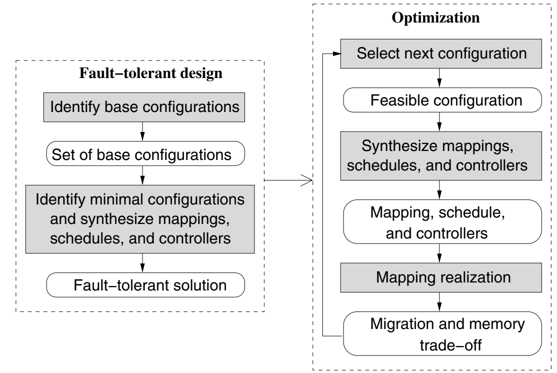

Figure 3. Overview of the design framework. In the first step, we synthesize solutions for base and minimal configurations to achieve fault tolerance and stability (with some inherent level of control quality). In the second step, the system is further optimized for additional configurations.

Initially, as a mandatory step, we synthesize a mapping for the configuration $\mathbf{N}$, in order to support the execution of the control system for the case when all computation nodes are operational. During the exploration process, a priority queue with configurations is maintained. Whenever a mapping $\mathsf{map}_{\mathbf{X}} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{X}$ has been synthesized for a certain feasible configuration $\mathbf{X} \in \mathcal{X}^{\text{oper}}$ (note that $\mathbf{N}$ is the first synthesized configuration), each feasible configuration $\mathbf{X}' \subset \mathbf{X}$ with $|\mathbf{X}'| = |\mathbf{X}| - 1$ is added to the priority queue with priority equal to the probability

$$p^{\mathbf{X}'} = p^{\mathbf{X}} p(N), \tag{13}$$

where $\{N\} = \mathbf{X} \setminus \mathbf{X}'$. For the initial configuration $\mathbf{N}$, we consider $p^{\mathbf{N}} = 1$.

Subsequently, for configuration $\mathbf{X}$, we check whether it is possible to realize the generated mapping $\mathsf{map}_{\mathbf{X}} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{X}$ at runtime with task migration and the available additional memory to store tasks. This step is described in detail in the next subsection (Section IX-B). If this step succeeds, it means that the mapping can be realized at runtime and we thus add $\mathbf{X}$ to the set $\mathcal{X}^{\text{impl}}$ (this set is initially empty). Further in that case, for each node N_d, the set of tasks $\mathbf{T}^{(d)}$ stored on N_d and the amount of additional consumed memory mem_d are updated. The set of tasks $\mathbf{T}^{(d)}$ that are stored on node N_d is initialized according to Equation 10. If the mapping realization does not succeed, the generated solution for configuration $\mathbf{X}$ is excluded. This means that a solution for a minimal configuration must be used at runtime to operate the system in the feasible configuration $\mathbf{X}$. Independently of whether the mapping realization of $\mathbf{X}$ succeeds, the exploration continues by generating a solution for the next configuration in the maintained priority queue of configurations. The explo-

ration terminates when the additional memory space on all computation nodes has been consumed, or when a specified design time has passed (e.g., the designer stops the exploration process).

B. Mapping Realization

For each configuration $\mathbf{X} \in \mathcal{X}^{\text{oper}} \setminus \mathcal{X}^{\min}$ that is considered in the exploration process, a mapping $\mathsf{map}_{\mathbf{X}} : \mathbf{T}_{\mathbf{\Lambda}} \longrightarrow \mathbf{X}$ is constructed (along with customized schedules and controllers). We shall now focus on whether and how this mapping can be realized at runtime in case the system reaches configuration $\mathbf{X}$. We first check whether there is sufficient memory to store information related to the solution (mapping, schedules, and controllers) for the configuration. The required memory for this information is denoted $\mathsf{mem}_d^{\mathbf{X}}$ and is an output of the mapping and synthesis step for configuration $\mathbf{X}$ (Section III-B). Let us denote with mem_d the amount of additional memory that is already consumed on N_d for other configurations in $\mathcal{X}^{\text{impl}} \subset \mathcal{X}^{\text{oper}} \setminus \mathcal{X}^{\min}$. If

$$\mathsf{mem}_d + \mathsf{mem}_d^{\mathbf{X}} > \mathsf{mem}_d^{\max}$$

for some $d \in \mathcal{I}_{\mathbf{N}}$, it means that the information related to the mapping, schedules, and controllers for configuration $\mathbf{X}$ cannot be stored on the computation platform. For such cases, we declare that the mapping $\mathsf{map}_{\mathbf{X}}$ cannot be realized (we remind that solutions for minimal configurations, however, can be used to operate the system in configuration $\mathbf{X}$).

If the solution for $\mathbf{X}$ can be stored within the given memory limit, we check whether migration of tasks that are needed to realize the mapping can be done within the maximum allowed migration time

$$\mu^{\max} = \min_{i \in \mathcal{I}_{\mathbf{P}}} \mu_i^{\max}.$$

If the migration-time constraint cannot be met, we aim to reduce the migration time below the threshold $\mu^{\max}$ by storing tasks in memory. The main idea is to store as few tasks as possible to satisfy the migration-time constraint. Towards this, let us consider the set of tasks

$$\mathbf{\Psi}_d(\mathbf{X}) = \left\{ \tau \in \mathbf{T}_{\mathbf{\Lambda}} \setminus \mathbf{T}^{(d)} : \mathsf{map}_{\mathbf{X}}(\tau) = N_d \right\}$$

that need to be migrated to node N_d at runtime in order to realize the mapping $\mathsf{map}_{\mathbf{X}}$, given that $\mathbf{T}^{(d)}$ is the set of tasks that are already stored on node N_d. The objective is to find a set of tasks $\mathbf{S}_d \subseteq \mathbf{\Psi}_d(\mathbf{X})$ to store on each node $N_d \in \mathbf{N}$ such that the memory consumption is minimized and the maximum allowed migration time is considered. We formulate this problem as an integer linear program (ILP) by introducing a binary variable b_d^{τ} for each node $N_d \in \mathbf{N}$ and each task $\tau \in \mathbf{\Psi}_d(\mathbf{X})$. Task $\tau \in \mathbf{\Psi}_d(\mathbf{X})$ is stored on N_d if $b_d^{\tau} = 1$, and migrated if $b_d^{\tau} = 0$. The memory constraint is thus formulated as

$$\mathsf{mem}_d + \mathsf{mem}_d^{\mathbf{X}} + \sum_{\tau \in \mathbf{\Psi}_d(\mathbf{X})} b_d^{\tau} \mathsf{mem}_d(\tau) \leqslant \mathsf{mem}_d^{\max}, \quad (14)$$

which models that the memory consumption $\mathsf{mem}_d^{\mathbf{X}}$ of the solution together with the memory needed to store the selected tasks do not exceed the memory limitations. The migration-time constraint is formulated similarly as

$$\sum_{d \in \mathcal{I}_{\mathbf{N}}} \left(\sum_{\tau \in \mathbf{\Psi}_d(\mathbf{X})} (1 - b_d^{\tau}) \mu(\tau) \right) \leqslant \mu^{\max}. \quad (15)$$

The memory cost to minimize in the selection of $\mathbf{S}_d \subseteq \mathbf{\Psi}_d(\mathbf{X})$ is given by

$$\sum_{d \in \mathcal{I}_{\mathbf{N}}} \left(\sum_{\tau \in \mathbf{\Psi}_d(\mathbf{X})} b_d^{\tau} \mathsf{mem}_d(\tau) \right). \quad (16)$$

If a solution to the ILP formulation cannot be found, then the mapping cannot be realized. If a solution is found, we have $\mathbf{S}_d = \{\tau \in \mathbf{\Psi}_d(\mathbf{X}) : b_d^{\tau} = 1\}$. In that case, we update the set $\mathbf{T}^{(d)}$ to $\mathbf{T}^{(d)} \cup \mathbf{S}_d$ and the memory consumption mem_d to $\mathsf{mem}_d + \mathsf{mem}_d^{\mathbf{X}} + \sum_{\tau \in \mathbf{S}_d} \mathsf{mem}_d(\tau)$. Even for large systems, the ILP given by Equations 16, 14, and 15 can be solved optimally and efficiently with modern solvers. We have used the `eplex` library for ILP in ECL^{i}PS^e [16], and it incurred negligible time overhead—less than one second—in our experiments.

X. Experimental Results

To evaluate our proposed design framework, we constructed a set of test cases with inverted pendulums, ball and beam processes, DC servos, and harmonic oscillators [9]. The test cases vary in size between 5 and 9 computation nodes with 4 to 6 control applications. All experiments were performed on a PC with a quad-core CPU at 2.2 GHz, 8 GB of RAM, and running Linux.

As a baseline of comparison, we considered a straightforward design approach for which we synthesize solutions for all minimal configurations and the initial configuration $\mathbf{N}$. This constitutes the mandatory set of solutions to achieve fault tolerance in any feasible configuration, as well as an optimized solution for the case when all nodes are operational. We computed a cost $J^{\min}$ according to Equation 12, considering that solutions have been synthesized for minimal configurations and the initial configuration, and that all other feasible configurations run with the corresponding minimal configuration with the minimum level of control quality given by Equation 11. The cost $J^{\min}$ indicates the overall control quality of the fault-tolerant control system with only the mandatory solutions synthesized.

Subsequently, we used our optimization heuristic to select and synthesize additional configurations. For each feasible configuration that is synthesized, individual cost terms in Equation 12 are decreased (control quality is improved compared to what is provided by minimal configurations). The optimization phase was conducted for varying amounts of design time. For each additional configuration that was synthesized, the total cost in Equation 12 was updated. We are interested in the control-cost improvement $\left(J^{\min} - J\right)/J^{\min}$ relative to the control cost $J^{\min}$ that is obtained when only considering the mandatory configurations.

Figure 4 shows the design time on the horizontal axis and the corresponding relative improvement on the

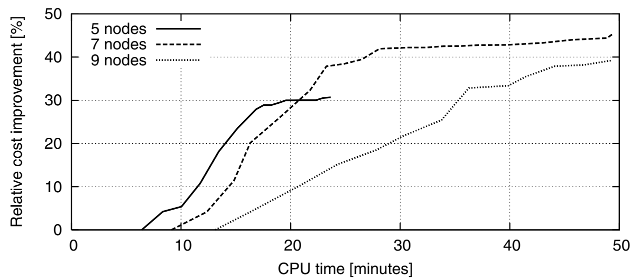

Figure 4. Relative cost improvements and runtimes of the proposed design approach. The time offset on the horizontal axis shows the synthesis time for the mandatory configurations.

vertical axis. The design time corresponding to the case of zero improvement refers to the mandatory design step to identify and synthesize solutions for minimal configurations, which requires around 10 minutes of design time. After this step, we have a solution that covers all fault scenarios and provides a certain minimum level of control quality in any feasible configuration. Any additional design time that is invested leads to improved control quality compared to the already synthesized solution. For example, we can achieve an improvement of around 30 percent already after 20 minutes for systems with 5 and 7 computation nodes. We did not run the heuristic for the case of 5 nodes for more than 23 minutes, because at that time it has already synthesized all feasible configurations. For the other cases, the problem size is too large to afford an exhaustive exploration of all configurations. Note that the quality improvement is relatively small at large design times. For those cases, the heuristic typically evaluates and optimizes control quality for configurations with many failed nodes. These quality improvements do not contribute significantly to the overall quality (Equation 12), because the probability of many nodes failing is very small (Equation 13). Our heuristic allows the designer to stop the optimization process when the improvement at each step is no longer considered significant. For the case of 7 computation nodes, the optimization process can be stopped after 30 minutes.

XI. Conclusions

We proposed a design framework for distributed embedded control applications with support for execution even if some computation nodes in the system fail. We presented an algorithm to identify base configurations and construct mappings for minimal configurations of the distributed system to achieve fault-tolerant operation. To improve the overall control quality relative to the minimum level of quality provided by the minimal configurations, we construct additional design solutions. Task replication and migration are mechanisms that are used to implement adaptation and reconfiguration of the system in case nodes fail at runtime. The alternative to our software-based fault-tolerance approach is hardware replication, which can be very costly in some application domains; for example, the design of many applications in the automotive domain is highly cost constrained.

References

[1] S. Y. Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, 2005.

[2] H. Kopetz, *Real-Time Systems—Design Principles for Distributed Embedded Applications*, 2nd ed. Springer, 2011.

[3] I. Koren and C. M. Krishna, *Fault-Tolerant Systems*. Morgan Kaufmann, 2007.

[4] L. Schenato, B. Sinopoli, M. Franceschetti, K. Poolla, and S. S. Sastry, "Foundations of control and estimation over lossy networks," *Proceedings of the IEEE*, vol. 95, no. 1, pp. 163–187, 2007.

[5] S. Sundaram, J. Chang, K. K. Venkatasubramanian, C. Enyioha, I. Lee, and G. J. Pappas, "Reputation-based networked control with data-corrupting channels," in *International Conference on Hybrid Systems: Computation and Control*, 2011, pp. 291–300.

[6] R. Majumdar, E. Render, and P. Tabuada, "Robust discrete synthesis against unspecified disturbances," in *International Conference on Hybrid Systems: Computation and Control*, 2011, pp. 211–220.

[7] C. Pinello, L. P. Carloni, and A. L. Sangiovanni-Vincentelli, "Fault-tolerant distributed deployment of embedded control software," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 5, pp. 906–919, 2008.

[8] C. Lee, H. Kim, H. Park, S. Kim, H. Oh, and S. Ha, "A task remapping technique for reliable multi-core embedded systems," in *International Conference on Hardware/Software Codesign and System Synthesis*, 2010, pp. 307–316.

[9] K. J. Åström and B. Wittenmark, *Computer-Controlled Systems*, 3rd ed. Prentice Hall, 1997.

[10] N. Navet, Y. Song, F. Simonot-Lion, and C. Wilwert, "Trends in automotive communication systems," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1204–1223, 2005.

[11] D. Srivastava and P. Narasimhan, "Architectural support for mode-driven fault tolerance in distributed applications," in *Workshop on Architecting Dependable Systems*, 2005.

[12] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén, "How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime," *IEEE Control Systems Magazine*, vol. 23, no. 3, pp. 16–30, 2003.

[13] S. Samii, A. Cervin, P. Eles, and Z. Peng, "Integrated scheduling and synthesis of control applications on distributed embedded systems," in *Design, Automation and Test in Europe Conference*, 2009, pp. 57–62.

[14] A. Aminifar, S. Samii, P. Eles, and Z. Peng, "Control-quality driven task mapping for distributed embedded control systems," in *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, 2011, pp. 133–142.

[15] P. Tabuada, "Event-triggered real-time scheduling of stabilizing control tasks," *IEEE Transactions on Automatic Control*, vol. 52, no. 9, pp. 1680–1685, 2007.

[16] K. R. Apt and M. G. Wallace, *Constraint Logic Programming using ECL^iPS^e*. Cambridge University Press, 2007.