The Information Processing Factory: A Paradigm for Life Cycle Management of Dependable Systems

Special Session Paper

Eberle A. Rambo
Thawra Kadeed
Rolf Ernst
{rambo,kadeed,ernst}@ida.ing.tu-bs.de
TU Braunschweig
Braunschweig, Germany

Minjun Seo
Fadi Kurdahi
{minjun.seo,kurdahi}@uci.edu
UC Irvine
Irvine, USA

Bryan Donyanavard
Caio Batista de Melo
Biswadip Maity
Kasra Moazzemi
Kenneth Stewart
Saehanseul Yi
Amir M. Rahmani
Nikil Dutt
{bdonyana,cbatista,maityb,moazzemi}@uci.edu
{kennetms,saehansy,amirr1,dutt}@uci.edu
UC Irvine
Irvine, USA

Florian Maurer
Nguyen Anh Vu Doan
Anmol Surhonne
Thomas Wild
Andreas Herkersdorf
{flo.maurer,anhvu.doan,anmol.surhonne}@tum.de
{thomas.wild,herkersdorf}@tum.de
TU Munich
Munich, Germany

ABSTRACT

The number and complexity of embedded system platforms used in mixed-criticality applications are rapidly growing. They run large and evolving applications on heterogeneous multi- or many-core processing platforms requiring dependable operation and long lifetime. Examples include automated and autonomous driving, smart buildings, industry 4.0, and personal medical devices. The Information Processing Factory (IPF) applies principles inspired by factory management to master the complexity of future, highly-integrated embedded systems and to provide continuous operation and optimization at runtime. A general objective is to identify a sweet spot between a maximum of autonomy among IPF constituent components and a minimum of centralized control in order to ensure guaranteed service even under strict safety and availability requirements. This paper addresses the challenges of IPF and how to tackle them with a set of techniques: self-diagnosis for early detection of degradation and imminent failures combined with unsupervised platform self-adaptation to meet performance and safety targets.

CODES/ISSS '19 Companion , October 13–18, 2019, New York, NY, USA

ACM ISBN 978-1-4503-6923-7/19/10.
https://doi.org/10.1145/3349567.3357391

CCS CONCEPTS

• **Computer systems organization → Embedded and cyber-physical systems**; **Dependable and fault-tolerant systems and networks**; *Real-time systems.*

KEYWORDS

self-awareness, mixed-criticality, dependability

ACM Reference Format:
Eberle A. Rambo, Thawra Kadeed, Rolf Ernst, Minjun Seo, Fadi Kurdahi, Bryan Donyanavard, Caio Batista de Melo, Biswadip Maity, Kasra Moazzemi, Kenneth Stewart, Saehanseul Yi, Amir M. Rahmani, Nikil Dutt, Florian Maurer, Nguyen Anh Vu Doan, Anmol Surhonne, Thomas Wild, and Andreas Herkersdorf. 2019. The Information Processing Factory: A Paradigm for Life Cycle Management of Dependable Systems: Special Session Paper. In *International Conference on Hardware/Software Codesign and System Synthesis Companion (CODES/ISSS '19 Companion), October 13–18, 2019, New York, NY, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3349567.3357391

1 INTRODUCTION

In the last decade, embedded system platforms have grown considerably in number and complexity, and they continue to grow. They run large and evolving applications on heterogeneous multi- or many-core processing platforms. Examples include automated and autonomous driving, smart buildings, industry 4.0, and personal medical devices. Such systems are required to provide dependable operation for the user while dealing with a large number of internal and external variabilities, threats, and uncertainties in their lifetimes.

The "Information Processing Factory" (IPF) was introduced in ESWEEK 2016 [2] as a paradigm to master such complex dependable systems. The IPF paradigm applies principles inspired by factory management to the continuous operation and optimization of highly-integrated embedded systems. A general objective is to identify a sweet spot between a maximum of autonomy among IPF constituent components and a minimum of centralized control in order to ensure guaranteed service even under strict safety and availability requirements. Emphasis is on intensive self-diagnosis for early detection of degradation and imminent failures combined with unsupervised platform self-adaptation to meet performance and safety targets.

An abstract concept at that time, IPF has been further elaborated and became the foundation of a US-German research initiative to investigate detailed solutions and applications. The initiative developed into a research cluster that is jointly funded by the National Science Foundation (NSF) and the German Research Foundation (DFG). The cluster exploits a variety of technologies including proactive reconfiguration to mitigate the risk of failures, self-optimization using hardware-based learning classifier tables [20], and chip-level operation with flexible boundaries between safety-critical and best-effort zones. With complementary purpose, the different technologies operate concurrently at different time granularities, all guided by a self-aware planning component.

Since its conception [2], IPF has been developed on multiple fronts whose advances are presented in this paper. A novelty of the self-aware IPF paradigm is the holistic approach integrating self-optimization, self-diagnosis, and self-organization techniques in synergy to meet performance targets and safety requirements. Thanks to its self-awareness and high degree of adaptability, IPF is an attractive solution for the life cycle management of future dependable systems, which require monitoring, identifying, and handling different internal and external variabilities, threats, and uncertainties that occur with different frequencies.

1.1 IPF's hierarchical organization

To master the increased complexity of managing such a self-aware system, IPF is hierarchically organized in five layers [12]. Figure 1 shows the organization. The *production line* (layer 1) contains the mixed-critical workload, with tasks of different criticality levels, and the system resources where the workload executes. The remainder of this paper refers to the two representative levels: best-effort (BE) and safety-critical (SC). The workload executes within the infrastructure and the container-based execution model of the *process support* (layer 2). This layer provides basic execution support, such as operating system (OS), real-time operating system (RTOS), and the runtime environment (RTE). The *supervisory process control* (layer 3) is responsible for monitoring the resources and optimizing the workload execution. The entities in this layer act locally and autonomously within boundaries specified by the layers above. That is the case of LCT and TAL, which perform local self-optimization and self-diagnosis and are introduced in Sections 2 and 4, respectively. The *manufacturing execution control* (layer 4) is responsible for enforcing safe system configurations by globally monitoring, assessing risks, as well as controlling the layers below. The layer contains two entities, the system controller (SC) and the

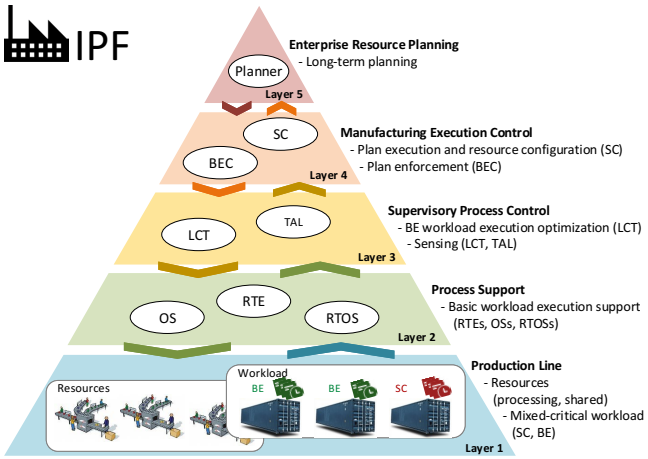


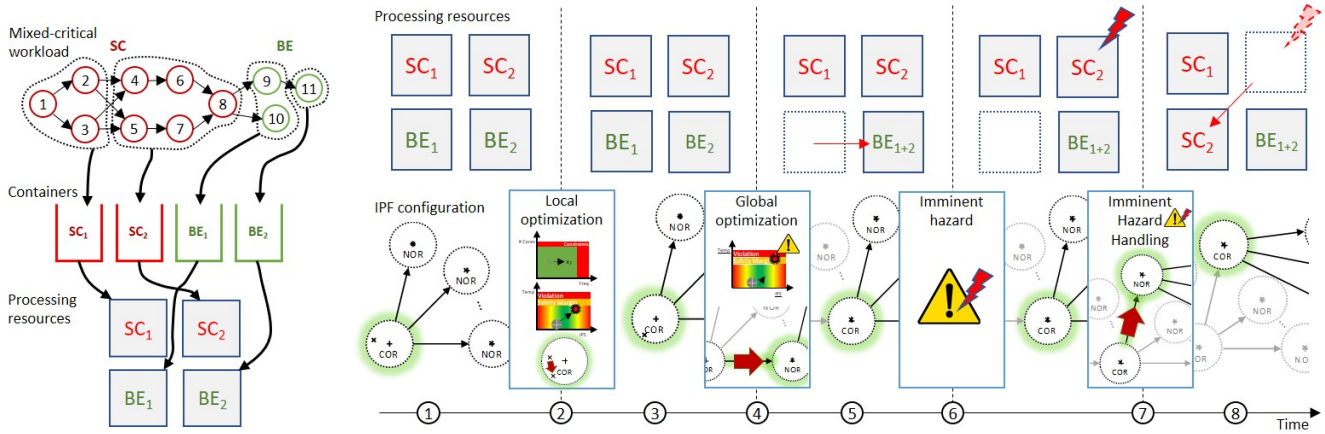
Figure 1: IPF's five-layer hierarchical organization [12].

best-effort controller (BEC), which manage the execution of the different criticalities. The BEC manages the best-effort workload execution under the supervision of the system controller, which also manages the safety-critical workload execution. That takes place under the guidance of the top layer, the *enterprise resource planning* (layer 5), which is responsible for the planning in IPF. It plans future proactive and reactive actions, taking into account the operating conditions of the system, assessing risks and impacts of short-term factors such as error rates, energy consumption and workload variations. It further considers long-term factors such as aging, energy constraints, and changes in the workload, quality-of-service (QoS) goals, and non-functional constraints. For a detailed description of the model, the interested reader can refer to [12].

1.2 An illustrative example

As a self-aware, mixed-critical factory, the IPF system comprises different features. It verifies the execution of safety-critical workload at runtime and it proactively acts upon critical conditions for the safety-critical workload, called imminent hazards. It also optimizes both locally and globally the best-effort workload execution. These features are illustrated in Figure 2 through an example of a healthcare pacemaker application executed on an IPF tile-based many-core system.

Figure 2a illustrates the initial mapping of the pacemaker application (adapted from [17]) to the system. The mixed-critical workload, represented as a directed acyclic graph (DAG), consists of eight safety-critical tasks (red nodes) and three best-effort tasks (green nodes). The arrows represent data and control dependencies between tasks. The workload is mapped to best-effort and safety-critical containers, which contain basic execution support. The set of processing resources to which the BE containers and SC containers are mapped form the BE zone and SC zone, respectively. The containers are mapped to four tiles (processing resources). The configuration of the IPF system is done by means of operating regions (ORs) and operating points (OPs) [12]. The mappings and the configuration ranges of the containers and resources form an OR. An OP is a specific configuration for the containers and resources



(a) Mapping to an IPF system

(b) Operation of an IPF system in time

Figure 2: Example of a healthcare pacemaker application with a mixed-critical workload mapped to an IPF instance (a). The operation of that IPF system in time and its configuration by means of operating regions (ORs) and operating points (OPs) (b).

within an OR. There might be multiple OPs within an OR, depending on the configuration ranges of the OR. The interested reader can refer to [12] for detailed definitions.

A possible execution sequence of that IPF system is illustrated in Figure 2b. The system starts in an initial configuration ① within the range of the current operating region (COR), which specifies the system configuration with bounds for local changes. Local changes in the configuration of containers and resources move the OP within the COR. At time ②, workload variations are detected in one of the resources (BE_2) by layer-3 entities, who identify opportunities for local self-optimization. The local self-optimization is then carried out in ③ by changing the configuration of the resource (executing workload of container BE_2) according to a given goal, represented by a change of the system's OP within the same COR. Later at time ④, a more significant workload variation is detected by the layer-4 global monitoring of the system. The global self-optimization and self-organization are carried out at ⑤ by moving the best-effort workload (containers BE_1 and BE_2) into a common processing resource (combined container BE_{1+2}) and powering-off one of the resources to save energy and reduce aging. That is represented by the transition from the COR to a new operating region, called next operating region (NOR). Later on ⑥, an imminent hazard caused by an impending permanent fault due to aging processes is detected by IPF's layer-3 self-diagnosis. It represents an increased risk to the system. Upon detection, IPF's self-organization takes a proactive measure to mitigate the increased risk to the safety-critical functions in the workload ⑦. The proactive action is carried out by migrating the affected safety-critical workload (SC_2) to a resource with reduced risk, represented by a transition from the COR to an NOR ⑧.

The remainder of this paper discusses each of the four aforementioned features of the IPF paradigm. The local self-optimization with hardware-based learning classifiers is discussed in Section 2. The global self-optimization is discussed in Section 3. The self-diagnosis with runtime verification is presented in Section 4. The self-organization with proactive handling of imminent hazards for safety-critical applications is presented in Section 5. Finally, Section 6 concludes the paper.

2 HARDWARE-BASED LEARNING CLASSIFIERS FOR MIXED-CRITICAL ENVIRONMENTS

Best-effort and approximate applications are controlled and optimized in IPF with a hierarchical, system-wide management structure. This section describes a method for the optimization of the best-effort tasks, which complies with the safety-critical workload management performed by the system controller and the constraints imposed by the planner in consideration of the requirements of the safety-critical workload. Exploiting maximum control flexibility to achieve the given performance objectives with a minimum amount of occupied resources (e.g., w.r.t. power budget and compute elements) while fully adhering to the specified constraints (e.g., in terms of local power dissipation or temperature) is the factory inspired paradigm that is applied here.

In a factory, a department or manufacturing group is given a target production rate objective as well as firm constraints on the utilities to use, such as electricity, source materials, and monetary production cost per unit. The department or group has a maximum of freedom on how to accomplish these targets under the constraints. At the same time, the produced goods of a given group are not as critical as other critical goods for the success of the factory as a whole. However, assuming that a given performance target can be achieved with different amounts of resources occupied, finding an approach to satisfy the objective target with the minimum amount of resources (i.e., having identified a Pareto point) frees up resources and utilities that may be used to optimize the critical production processes. Likewise in IPF, a configurable operating point needs to be identified within the tunable solution space (e.g., frequency and amount of occupied cores) of system parameters in order to accomplish the performance objectives, while respecting

constraints defined so as to avoid violation of the safety-critical requirements. This results in a lower local temperature and/or power dissipation and creates an extra margin for the critical applications. In IPF, this is achieved by local self-optimization using Learning Classifier Tables (LCTs).

In the following, we will describe the basic structure and properties of the learning classifier table based reinforcement machine learning engine. Then, the envisaged strategy to optimize the solution space exploration will be presented.

The presented LCTs and their envisaged operating mode provide the ability of local self-optimization for the best-effort workload in the mixed-critical IPF system, as illustrated by steeps ① and ② in the example of Figure 2b. Providing local self-optimization with LCTs is important to enable dependable systems operation as the continuous learning ability will allow to meet required performance while reacting to changes, e.g., in workload, software, and environment.

2.1 Learning Classifier Table (LCT)

In IPF, LCTs are used as reinforcement learning based controllers to meet objective targets for best-effort tasks within the given constraints.

This is accomplished by exploiting the rule-based structure of the LCT learning engine, whereas a rule consists in a combination of condition, action and fitness. Therefore, an LCT observes how a system behaves after a given action has been applied. This observation results in a grade representing the effect of this action with respect to an objective function and is used to update the current fitness of this specific rule. LCTs are a subset of Wilson's ZCS [18], which is a type of learning classifier system (LCS), and were proposed by Zeppenfeld et al. in the ASoC project [20].

Figure 3 represents the overall structure and periodically repeating operation steps of an LCS:

(1) By comparing the sensor values to the conditions of all rules in the population [P], a match set [M] is generated. Conditions can contain wildcards to match several sensor values.
(2) A roulette-wheel selection algorithm decides on an action based on the fitnesses of the rules in [M]. All rules in [M] with the same action as the one selected by the roulette wheel are saved as the action set [A].
(3) The selected action is forwarded to the actuators which apply it to the system. Further, the action set is saved in $[A]_{t-1}$.
(4) After applying the action, its effect is measured by sensors, and based on the effectiveness towards achieving the objective, some reward is given by the credit assignment component. The fitnesses of the rules in the saved action set $[A]_{t-1}$ are updated based on the reward according to a modified version of Q-learning proposed by Wilson [18].
(5) Finally, the updated rules are forwarded to the population [P] for the next evaluation.

The roulette-wheel selection algorithm allows the LCTs to escape from local optimums by trying out actions which result in less optimal configurations than already explored ones. Compared to LCSs, the in hardware implemented LCTs do not currently have the possibility to generate new rules through, e.g., a genetic algorithm. This is currently under investigation.

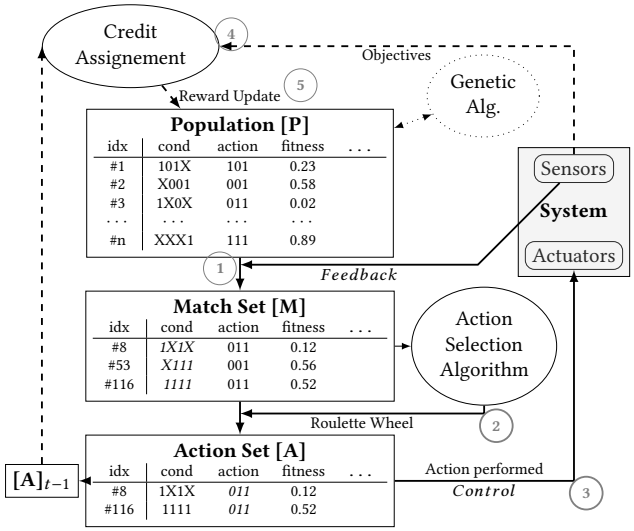


Figure 3: Overview of the LCT logic (dashed lines correspond to the fitness update path; dotted entities are part of LCSs, but not LCTs).

2.2 Optimization strategy

The main optimization challenge in a system management problem lies in the fact that the optimization is performed for metrics that cannot directly be influenced by changing different tunable parameters. In our case, this means that LCTs act on the solution space whereas the actions have an impact on the objective space (e.g., instructions per second, temperature, utilization). Moreover, the metrics from the objective space are used to grade the actions applied on the solution space. This behavior is represented in Figure 4, where a configuration (c1) results in a state (s1), which cannot be exactly determined ahead due to uncertainties, such as surrounding temperature and software complexity. This state (s1) is used to decide for an action (a1) which results in a new configuration (c2) and a state (s2). Based on the new state (s2), the action (a1) for the condition (s1) in the LCT is graded.

2.2.1 Boundaries. In best-effort only environments these spaces are only constrained by the hardware's limits such as maximum frequency and amount of cores for the solution space as well as maximal temperature for the objective space.

In a mixed-critical system, however, there is a bidirectional impact between best-effort and safety-critical workloads [12]. To rule out unpredictable impact on the execution of safety-critical workload, the enterprise resource planning (layer 5 in Figure 1) specifies additional constraints on the solution as well as on the objective space of the best-effort containers by limited CORs. These constraints are enforced by the BEC of the manufacturing execution control (see Section 1.1 and Figure 5). Additionally, the application and the operating system might specify constraints or at least a reference (see mark in Figure 4) to optimize the configuration for the best-effort objective.

To extend their applicability to mixed-critical platforms, LCTs must handle such additional constraints (see invalid zones in Figure 4).

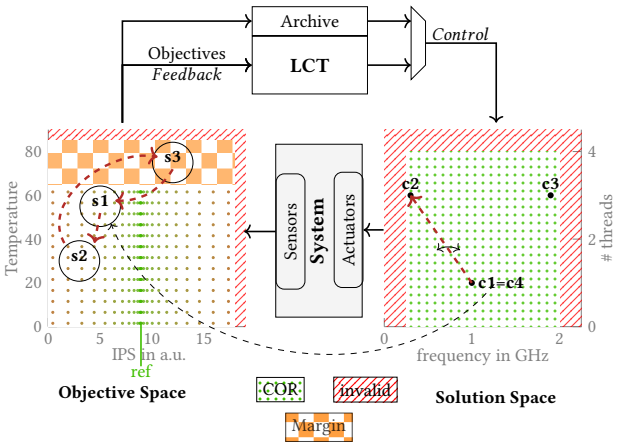


Figure 4: Impact of safety-critical boundaries on the LCTs' behavior

2.2.2 Approach. Applying LCTs on this new optimization problem containing additional constraints requires to tackle two issues. First, the actions of the LCTs need to be limited in order to stay within the defined solution space. Second, the LCTs have to be prevented from applying actions which violate the objective space limits.

Restraining the actions to prevent violations of the solution space constraints can be easily achieved. Indeed, it only requires checking whether violations can occur before applying a given action and, in such a case, limiting it to the closest feasible value allowed.

Enforcing the constraints in the objective space is a more complex challenge. For example, in case no precaution is taken, consecutive small steps could lead to violations, due to the stochastic behavior of the roulette wheel selection algorithm presented in Section 2.1. Also using a deterministic action selection algorithm can equally cause a violation of constraints for a yet unlearned rule set.

Violating a constraint is of course not permitted within IPF, as this would trigger a change to a new OR, as mentioned in Section 1.2. To prevent violating any constraints in the objective space, we introduce margin zones in front of them (see Figure 4). These margins allow us to recognize the risk of violation in the next LCT cycle. To ensure that this risk is really recognized, the margin needs to be larger than the step size a single action of the current rule set can have on the objective space. This is ensured by either having margin zones that are large enough or by limiting the dynamic range of the actions being allowed in a rule set.

In case of an imminent constraint violation due to an action taken by the LCT, it is important to bring the subsystem back to a safe operation point. State-of-the-art methodologies for solving this issue make use of a backup policy that is able to lead the subsystem from approaching or violating a constraint to a known safe operating point [4]. We utilize an archive containing the best configuration experienced by this subsystem so far, for the currently executed workload. Doing so, the applied "emergency configuration" fits the current system, requiring only a low amount of memory and a few comparators in hardware to be implemented. Further, this approach overcomes the probabilistic behavior of the roulette wheel selection algorithm as well as the issue of non learned rule sets/configurations since, by design, a valid configuration for the current setup must exist in the archive, due to the plan from layer 5 which is enforced at least when transitioning to the COR. An example situation is illustrated in Figure 4. There, in case a configuration (c3) results in a state (s3) which is in the safety margin, the archive applies the best so far experienced configuration (c1). After successfully returning to a valid configuration and its corresponding state, the learning is regularly continued. A rule resulting in the margin zone gets the lowest possible reward to prevent or minimize further application.

3 MANAGING BEST EFFORT WORKLOADS

Hierarchical management of the best-effort workload in IPF contributes to dependable system operation. The hierarchical management provides self-adaptivity and guides self-optimization within the BE zone in conjunction with the system controller. As described, LCTs provide self-optimization of dynamic workloads within a COR. However, LCTs themselves require supervision to: (1) ensure they are achieving application goals, (2) specify their operating configuration bounds, and (3) enforce system constraints. These responsibilities belong to the manufacturing execution control (layer 4), specifically the BEC.

Figure 5 shows the hierarchical relationship between BECs and LCTs. The application specifies goals with respect to the best-effort portion of its workload, and provides the BE goals to the BEC. The BEC is responsible for accomplishing the BE goals, but does not directly control the underlying system. Instead, the BEC provides *control parameters* to the LCTs in order to indirectly control the subsystem(s). For example, a workload may specify a target value for an application-specific QoS metric. The BEC is responsible for determining the distribution of QoS "budget" among subsystems, and in turn providing LCTs with an appropriate objective function.

Provided an objective function, LCTs are designed to search the subsystem's configuration/solution space for optimal combinations of knob settings. The OR defines the valid ranges of knob settings, and it is possible that the allowable range within an OR is only a subset of the possible range (as depicted in step ② of Figure 2b). The system controller (SC, Figure 1) specifies the COR, and informs the BEC of the valid knob settings. The BEC is then responsible for enforcing the allowable range of knob settings. For example, the SC may determine that, for a given workload, cores must cap their

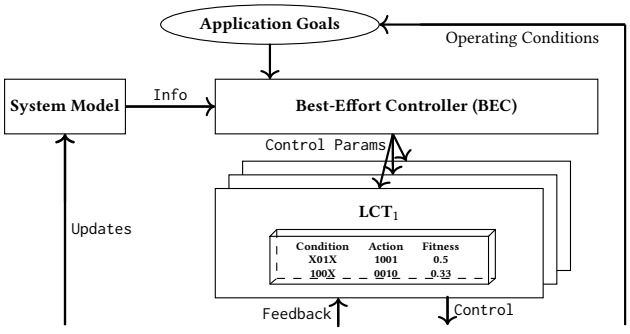


Figure 5: Hierarchical relationship between the Best-Effort Controller (BEC) and LCTs.

operating frequency at 50% of the maximum in order to prevent thermal events. The BEC will then provide control parameters to the LCTs such that the LCTs only allow actions that result in core frequencies under 50%.

It is possible that, due to operating conditions or workload variability, LCTs' self-optimization may explore valid configurations of subsystems that violate system constraints specified by the SC (depicted in step ④ of Figure 2b). The BEC is responsible for sensing the system state and enforcing system constraints provided by the SC. That is, if the BEC observes LCTs violating system constraints, the BEC must update the control parameters (e.g., objective function and margins) in order to honor the constraints. For example, a phasic change in BE workload could cause a valid operating point (OP) to violate a specified power budget. In this case, the BEC should take immediate action to update the LCT objective function, corresponding margins, and the active ruleset to prioritize lowering the power over achieving target QoS. This is how BECs provide self-adaptivity. Additionally, it may be necessary to take more significant precautions if the same system constraint is continuously violated. If this is the case, the BEC can trigger an *event*, causing the system controller to change the OR.

In addition to managing BE containers, the BEC is also required to respond to SC requests during self-organization in order to enable the enhanced dependability provided by the entire IPF. Self-organization is described further in Section 5.2.

3.1 Enabling self-adaptivity through reflection

BECs use principles of computational self-awareness to make runtime decisions when managing the best-effort zone of an IPF system. Reflection is a key property of self-awareness. Reflection enables decisions to be made based on both *past* observations, as well as predictions made from past observations. Reflection and prediction require a self-model of the subsystem(s) under control, as well as models of other policies that may impact the decision-making process. Predictions consider *future* actions, or events that may occur before the next system evaluation and (re)configuration, enabling "what-if" exploration of alternatives. The main goal of the prediction model is to estimate system behavior based on potential actuation decisions.

Consider the best-effort controller shown in Figure 6 that includes a task mapping policy, and supervises LCTs responsible for dynamic voltage and frequency scaling (DVFS). At the finest time granularity, we have the operating system scheduler, whose goal is to select a task to execute on a given core. A new decision must be made whenever a new task is created, a task completes, a task's quantum expires, an interrupt is raised, etc. Such decisions are made on the order of microseconds. At a coarser time granularity we have the DVFS policy deployed by LCTs, which execute periodically (10 milliseconds) to analyze the system load and select an appropriate operating frequency. At the coarsest time granularity (100 milliseconds), the task mapping policy runs periodically to define a new task-to-core assignment. Migrating a task from one core to another has significantly more overhead than changing the CPU frequency in a typical heterogeneous multiprocessor [19]. The BEC *coordinates* among different policies through policy models, regardless of varying time granularity.

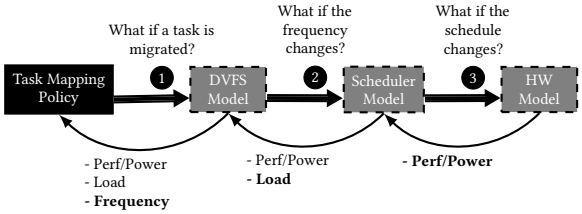


Figure 6: Example of a task mapping policy that queries models of OS policies for DVFS and scheduling.

In order to make an informed task mapping decision, for instance, the policy must consider the effects of its decision on the behavior of the underlying DVFS and scheduling policies. Furthermore, the invocation period of actuations dictates how complex the decision making logic can be. For instance, a scheduling decision must be made in the sub-microsecond range in order not to disrupt the system. Task-to-core mapping, on the other hand, is done comparatively infrequently, and affects the system performance over a long timespan. Therefore, the overhead of using complex models to make such decisions can be mitigated by the potential benefits of an informed decision.

The components within the reflective system model interact in a hierarchy defined by the dependencies of the actuations performed in the system. For instance, Figure 6 illustrates the scenario for our example. *Workload models* assume that each core can run multiple tasks and there is no formal or explicit dependency between threads. Before the task mapping policy decides to migrate a task, it (1) queries the reflective model asking: *what will be the performance of task A if it is migrated?* (2) The LCT's model predicts the resulting core frequency provided the hypothetical task mapping. (3) This information is passed on to the performance/power model which predicts the task performance. *Architecture models* define the architectural characteristics of the target platform, including instruction-set architecture (ISA), number of cores, core types, etc. Finally, the predicted metrics are used by the policy to make the decision, which is passed to the actuator through the actuation interface.

4 RUNTIME VERIFICATION AND INFERENCE

Within IPF, the techniques in this section are relevant for a long-term dependable system because short-term and long-term system monitoring/verification/inference is a prerequisite for system state identification/recovery as shown in Figure 1 (layer 3).

Runtime verification has been widely used to monitor the execution of systems. At the system development stage, verification and debug consume about 75 % of the effort, and despite the effort, hazard-resulting failures at runtime cannot be ruled out. A software method, i.e., instrumentation, has the advantage of not requiring any extra hardware, but it has too much overhead in situations where short response time is critical. Thus, in a safety-critical system, it is ideal for performing non-intrusive runtime verification in hardware. Modern microprocessors have a trace interface that can be used for non-intrusive runtime verification with an independent runtime verification hardware unit [16]. That is, the system loads

a programmable hardware unit with properties through software and monitors that are satisfied at runtime, detecting errors caused by permanent faults.

Runtime verification provides a reactive way to detect an error immediately, but detection of imminent hazards in advance through it is difficult or almost infeasible. In order to cope with imminent hazards, a method that predicts the future through learning is required. An inference engine (IE) predicts imminent hazards based on trained data using machine learning (ML) techniques such as recurrent neural networks (RNN) or temporal convolutional neural networks (CNN). In the case of safety-critical tasks, ML-based prediction can be used because it is essential to respond even with some false positives proactively.

In this section, we introduce the Trace Abstraction Layer (TAL) for runtime verification and inference consisting of: (a) a design-time methodology for developing software and constructing verification properties for runtime verification to handle errors, (b) a design-time methodology for training the inference engine to predict imminent hazards. Figure 7c presents TAL, consisting of several functionally/logically separated components which are: (1) compatibility layer, (2) filtering layer, and (3) verification/inference layer for permanent fault/imminent hazards detection. Section 4.1 discusses the need for TAL and the role and relationship of each layer. In Section 4.2, we will look at the runtime verification that is currently being used in IPF, and look at the inference engine which is the future direction in Section 5.1.

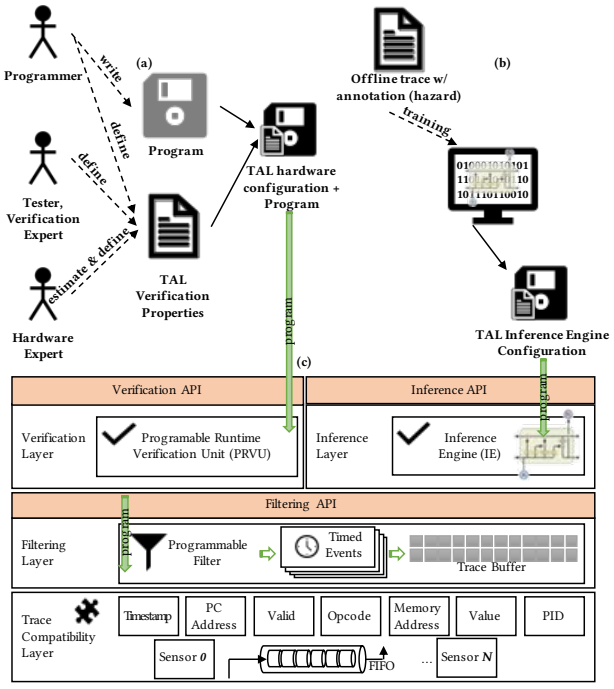


Figure 7: Overview of a design-time methodology for runtime verification (a), a design-time methodology for inference engine (b), and the Trace Abstraction Layer (TAL) hierarchy (c).

4.1 Trace Abstraction Layer (TAL)

TAL's well-defined and independent layers shown in Figure 7c make it possible to be used in a variety of ways to meet goals and circumstances.

The compatibility layer (CL) at the lowest layer ensures compatibility with different processors. Existing microprocessors provide different trace interfaces for various vendors. Because these interfaces are not compatible with each other and the bits, types, and numbers of signals are different from each other, there is the problem that the hardware using trace interface is also incompatible across platforms. Therefore, CL is a layer to ensure data compatibility between heterogeneous processors and different sensors. Different architectures need CL implementations that are tailored for each, and common implementations can be used for layers above CL.

Since tracing is a cycle-by-cycle method, too much information can be flooded into verification/inference hardware. Therefore, the filtering layer (FL) focuses on filtering this data and delivering only the necessary information with a timestamp to the next layer. Software-programmable filters through the Filtering API enable context-sensitive filtering for tracing.

The verification layer (VL) provides a variety of primitives for non-intrusive runtime verification. The inference layer (IL) supports the prediction of a hazard ahead of time. Instances of core modules, which are programmable runtime verification unit (PRVU) and inference engine (IE), of VL and IL, are introduced in Sections 4.2 and 5.1, respectively.

4.2 Runtime verification

In Figure 7a, programmers perform typical software development, providing basic verification properties with verification experts. Testers, verification experts, and hardware experts analyze the requirements and prepare verification properties. TAL verification property has the data required for filter configuration, properties for runtime verification, loaded into the hardware at the beginning of verification, and helps to perform runtime verification through hardware during system execution.

From the IPF point of view, runtime verification hardware realization has been discussed in various methods such as coarse-grained reconfigurable architecture (CGRA), runtime partial reconfiguration in field programmable gate array (FPGA), and automata processors (APs). The above methods are intended to perform RV using the limited hardware resources with maximum efficiency.

In IPF systems, the current generation of TAL uses AP or equivalent implementation as a verification hardware [15], accompanying a model transformation from the well-known timed automata (TA)-based requirement model to an AP-based model and uses it for verification. In the experiment, TAL with the pacemaker and collision avoidance examples, we achieved 100 % error detection rate using only 3 % of the resources of AP, providing the root cause of the errors. Clearly defined properties can perfectly detect all the errors caused by both logical/physical failures, but they have two problems: 1) It might be late to recover system problems as a reactive method. 2) There is no way to verify what is not in the requirements. In order to detect such anomalies, an inference-based detection process is needed.

5 PROACTIVE SELF-DIAGNOSIS AND SELF-ORGANIZATION FOR SAFETY-CRITICAL APPLICATIONS

IPF's proactive self-diagnosis and self-organization address the strict requirements of future safety-critical applications. Besides functional requirements, safety-critical applications have strict non-functional requirements that must be met at runtime, such as timing, integrity, availability, and reliability [3]. The two mechanisms cooperate in synergy to increase the system dependability while ensuring that those requirements are met despite the presence of imminent hazards.

Imminent hazards are unacceptably high risks of system failure [12]. As such, imminent hazards can have different causes. For high dependability, we focus on imminent hazards caused by the imminent failure of a processing element due to an impending permanent fault caused aging processes. Those faults are preceded by increasing error rates and intermittent faults [1]. Note that imminent hazards must be distinguished from latent faults and errors caused by unrelated transient and intermittent fault occurrences.

The self-diagnosis must detect imminent hazards, and must do so in time, so that they can be proactively handled by IPF's self-organization. The self-organization must then ensure that the system proactively acts in time by reorganizing the system and reducing the risk to an acceptable level. Both must finish before a hazard occurs and leads to the violation of any of the timing, integrity, availability, or reliability requirements.

Figure 8 plots the risk of a system failure in time to illustrate the proactive concept and compares it with a conventional, reactive one, and a static one that is neither proactive nor reactive. In time, the risk of an error increases, such as the occurrence of a permanent fault, up to a point where it becomes an imminent hazard. The imminent hazard is detected ①, and then handled ② with time to spare before the error finally occurs ③ (the time interval Ⓐ) and causes the hazard ④. In contrast, the reactive approach only detects the error ③ and must handle it within a very short time before the hazard and any violation of the non-functional requirements ④ (the time interval Ⓑ). The system in a static configuration fails and results in a hazard ④ since it does not tolerate the error.

The two proactive mechanisms operate in layer 3, supervisory process control, and layer 4, manufacturing execution control, of the IPF organization (cf. Figure 1), where a local and a global view of the system are maintained, respectively. The remainder of this section discusses the mechanisms and the particular challenges involved in achieving the above-mentioned guarantees.

5.1 Proactive self-diagnosis

The on-chip proactive self-diagnosis goes beyond conventional error detection: it detects hazards in advance before the error occurs and affects the system. Its proactive nature enables IPF to handle imminent hazards with more time and flexibility since the system execution and state have not been affected by the error yet. In comparison with conventional, reactive error detection approaches, the proactive self-diagnosis is a promising mechanism that enables a more lightweight proactive hazard handling with self-organization.

Error detection in mixed- and safety-critical computation usually relies on modular redundancy approaches applied in time or space, while error detection in communication heavily relies on information redundancy. The current state of the art includes the so-called cross-layer approaches, which combine error detection and handling techniques in different layers of the system stack for lower overhead and increased efficiency. An example is the redundant software execution with hardware-supported error detection [13], where the software can be protected with replication in space or time combined with efficient error detection in hardware. The approach can be additionally coupled with other error detection mechanisms to increase coverage and resilience while ensuring integrity [14]. Nonetheless, the industry still relies on costly hardware modular redundancy approaches, such as diverse dual modular redundancy (DMR) or triple modular redundancy (TMR) with lock-step execution [6]. Although effective, conventional approaches are highly inefficient, as they incur substantial overhead – e.g., more than 100% overhead in DMR, and severely restrict the reaction time for recovery, due to its reactive nature.

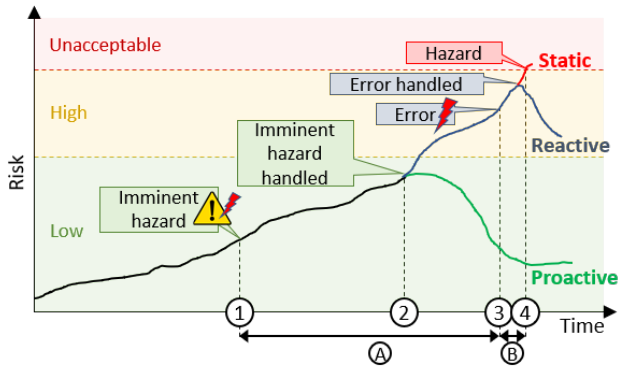


Figure 8: Comparison of proactive hazard detection and handling vs. reactive error detection and handling vs. static system (neither proactive nor reactive).

Integrity is paramount in safety-critical systems [7, 14]. Although the maximum hazard handling time with the proactive self-diagnosis (Ⓐ in Figure 8) is not as critically short as the maximum reaction time in reactive approaches (Ⓑ in Figure 8), integrity continues being a paramount system requirement. The system must be aware of all errors in it and cannot allow them to affect its service, which would result in the unacceptable risk of an uncontrolled failure. The system must detect all errors; it can handle or tolerate some; when it cannot handle or tolerate an error, it must indicate failure before it becomes a hazard [14]. Conventional, reactive techniques can ensure integrity by only allowing data to be consumed or an output to be made after the possibility of error has been ruled out [13]; or, less strictly, by detecting the error and reacting to it in time to contain its propagation [14]. In contrast, the proactive self-diagnosis detects imminent hazards before the error occurs and relies on IPF's self-organization to handle it before it propagates and becomes a hazard.

The self-diagnosis requires both a local and a global view of the system, and operates therefore in both layer 3, supervisory process control, and layer 4, manufacturing execution control, of IPF (cf. Figure 1). Once the risk of a hazard increases up to a point where

sample_id	timestamp	temp	power	load	freq	hazard
0	0.011	65	2.996	0.27	2000	0
1	0.021	64	1.958	0.40	2000	0
2	0.032	64	2.245	0.45	2000	0
3	0.042	65	3.113	0.90	2000	0
4	0.052	64	4.037	1	2000	0
5	0.063	66.5	4.6185	1	2000	0
6	0.073	68	4.644	1	2000	0
.....						.....
152	1.596	69	4.711	1	800	1
.....						.....

(a)

Trace data	5 ms periodic (Temperature, Power, Frequency, Load, Hazard (***tag***))
RNN	4 Long Short Term Memory-stateful layer
Training Data	Use core 0,1,2 data (2430 epochs)
Test Data (Accuracy)	Use core 3 (93%), 4 (99%) data

(b)

Table 1: Detailed trace data (a) and the result of the RNN-based inference engine for a given configuration (b).

a permanent fault is close, the imminent hazard is detected (⑥ in Figure 2b and ① in Figure 8) and reported to the system controller (layer 4) with a discrete event. The event triggers the handling of the imminent hazard by IPF's self-organization (⑦ in Figure 2b and ② in Figure 8).

An envisioned way to detect imminent hazards is by means of an inference engine (IE). The engine is coupled with TAL, as shown in Figure 7c. The trace data in which the microprocessor outputs information for each tick or various sensor data is treated as a time series data. An RNN with long short-term memory (LSTM), useful in predicting the time series data, is then employed to examine the obtained trace data. Figure 7b shows the offline training process. RNN LSTM-based weights are obtained using trace data at program execution and data consisting of normal/abnormal annotation (tag). The obtained weight is synthesized with the inference engine configuration and updated at runtime by the IE. Through this, IE can be used to predict hazards at runtime.

A preliminary experiment employed the approach to detect imminent hazards caused by thermal throttling due to a too high temperature of a core. Table 1a shows the extracted trace data with a 5 ms period on a 2 GHz ARM 8-core ODROID board. The hazard is tagged in the trace where the frequency throttling occurs, after its prediction. To use the trace of each time as input data, we construct a one-hot vector that assigns one entry to a unique row after performing normalization.

Considering the large trace size, a total of four LSTM layers is connected. That is shown in Figure 9, where x (trace one-hot vector) is input, y (0: No hazard, 1: Hazard) is output, and St is hidden state respectively. Trace data from cores 0, 1, and 2 are used for a training set and data from cores 3 and 4 for a test set. The result from the test set shows 93 % and 99 % accuracy to detect an imminent hazard 20 ms before the hazard. The preliminary experiment showed excellent performance in predicting hazards based on RNN LSTMs. Future work includes an efficient design of a hardware-based RNN LSTM implementation with high accuracy and low area overhead,

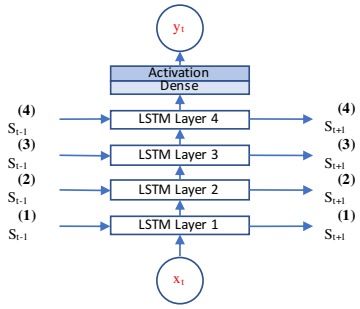


Figure 9: Multi-layer LSTM for the inference engine.

online learning interacting with runtime verification, and optimized network structure for trace data.

5.2 Proactive self-organization

In hard real-time and safety-critical domains, tackling the impact of hardware errors in time, i.e., before a failure occurs, is indispensable. The major limitation in conventional solutions, e.g., with modular redundancy, is that the mechanisms are developed to reactively handle errors. In addition to the short time available for error recovery, the system safety can also be jeopardized during the recovery process [5, 9]. As depicted in Figure 8, upon error occurrence, the reactive system must recover from the error in time, assuring temporal guarantees and functional safety. Any potential erratic behavior during the recovery process jeopardizes the whole system safety. As the chip becomes more complex, our aim is to address system early degradation – preventing the system from failing and being in a hazard. Therefore, proactively handling imminent hazards provides flexibility and reduces the vulnerability of the system.

To this end, IPF proactively handles an imminent hazard based on the hazard type. One potential hazard handling is by turning off the resource once it is idle to decrease temperature and mitigate aging [8, 10]. However, as IPF addresses early degradation, it mainly employs task migration as a protection tool. Upon detection, the imminent hazard event is handled through the system controller (cf. Figure 1, Layer 4). The system controller has the ultimate control of the platform so that when notified of the event by the TAL, it employs the task migration, transitioning the system to an appropriate new configuration, represented by a NOR, as depicted in steps ⑦ and ⑧ in Figure 2b. IPF's proactive approach has multiple advantages over a strictly reactive one. For example, it increases the *safety*, the *lifetime*, and the *flexibility* of the system, and decreases the *overhead* in comparison with reactive approaches.

While the system controller provides a proactive handling of an imminent hazard, and thus protects the application in advance from experiencing an error, it induces the following challenges during the system reconfiguration:

- *Temporal guarantees of the protected safety-critical tasks*
  The system controller interferes with a running system that is still safe, and performs reconfiguration – moving the system to a new OR. Thus, the protected safety-critical tasks,

during the reconfiguration, are no longer available, and their temporal guarantees are jeopardized.

- *Vulnerability to immediate errors*
  The system controller, during the reconfiguration process, induces vulnerability of the protected safety-critical tasks, caused by reducing the system redundancy. It is already a problem in terms of immediate errors that would then have higher impact on the system safety.
- *Functional safety*
  As the system controller interferes with a running system, it may induce system failure due to erratic behaviour during the reconfiguration.

To do so, the system controller must provide appropriate communication mechanisms and protocols with other local controllers in the system in order to fulfill the task migration from a failing tile to a healthy BE tile. Thus, when the self-diagnosis with TAL (layer 2) reports a detected imminent hazard to the system controller, the system controller guides and orchestrates the communication with the associated local controllers, the BEC, and the RTOS, in order to perform the task migration. All communication actions are planned in advance by the planner (layer 5), and provided to the system controller as plans. The system controller performs the plan by conveying the required actions to the local controllers. The BEC (layer 4) manages and controls the resources and the workload in the BE zone, based on the system controller indications to prepare a BE tile for migration whenever necessary. The RTOSes (layer 1) manage the workload in the SC zone and cooperate with the system controller to safely perform the migration without jeopardizing the temporal guarantees of the protected safety-critical functions. Thus, the system controller, during the migration process, must adhere to a strict requirement, which is mainly a deterministic critical path. The critical path, triggered by the proactive migration, must be upper bonded, providing system predictability and safety.

The system controller is a logical extension of an existing network-on-chip (NoC) controller, the resource manager (RM) [11]. The protocol-based synchronisation was already successfully applied for NoCs by the RM to improve performance while ensuring temporal guarantees of the safety-critical functions. Moreover, the concept of the RM has been extended by [8, 10] in order to perform safe NoC power management. The latter dynamically adjusts the NoC power dissipation while providing guaranteed service. The system controller extends the concept of the RM from the NoC level to the system level, where a suitable protocol and the migration supervision by the system controller are ongoing research.

6 OVERVIEW

This paper discussed the challenges of IPF, a paradigm for life cycle management of dependable systems, and how to tackle them. The IPF paradigm applies principles inspired by factory management to master the complexity of future, highly-integrated embedded systems and to provide continuous operation and optimization at runtime while ensuring guaranteed service even under strict safety and availability requirements. Achieving that requires a set of techniques that operate in synergy in the system under the guidance of a self-aware planning component. This paper discussed four techniques in IPF that make it a promising solution for highly dependable systems: from self-diagnosis for early detection of degradation and imminent failures to the unsupervised platform self-adaptation to meet performance and safety targets.

ACKNOWLEDGMENTS

We acknowledge financial support from the following: NSF Grant CCF-1704859; DFG Grants ER168/32-1 and HE4584/7-1.

REFERENCES

[1] Cristian Constantinescu. 2003. Trends and challenges in VLSI circuit reliability. *IEEE micro* 23, 4 (2003), 14–19.
[2] Nikil Dutt, Fadi J. Kurdahi, Rolf Ernst, and Andreas Herkersdorf. 2016. Conquering MPSoC complexity with principles of a self-aware information processing factory. In *Proceedings of the 11th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS) (CODES'16)*. ACM, Pittsburgh, Pennsylvania, 37.
[3] Rolf Ernst and Marco Di Natale. 2016. Mixed criticality systems - A history of misconceptions? *IEEE Design & Test* 33, 5 (2016), 65–74.
[4] Alexander Hans, Daniel Schneegaß, Anton Maximilian Schäfer, and Steffen Udluft. 2008. Safe exploration for reinforcement learning.. In *ESANN*. 143–148.
[5] Jörg Henkel, Lars Bauer, Joachim Becker, Oliver Bringmann, Uwe Brinkschulte, Samarjit Chakraborty, Michael Engel, Rolf Ernst, Hermann Härtig, Lars Hedrich, et al. 2011. Design and architectures for dependable embedded systems. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 69–78.
[6] Infineon. 2019. 32-bit TriCore™ AURIX™ – TC3xx. https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/. [Online].
[7] ISO 26262: 2018. ISO 26262: Road Vehicles – Functional Safety. International Standards Organization.
[8] Thawra Kadeed, Sebastian Tobuschat, Adam Kostrzewa, and Rolf Ernst. 2018. Safe and efficient power management of hard real-time networks-on-chip. *Integration* (2018).
[9] Amin Khajeh, Minyoung Kim, Nikil Dutt, Ahmed M Eltawil, and Fadi J Kurdahi. 2008. Cross-layer co-exploration of exploiting error resilience for video over wireless applications. In *2008 IEEE/ACM/IFIP Workshop on Embedded Systems for Real-Time Multimedia*. IEEE, 13–18.
[10] Adam Kostrzewa, Thawra Kadeed, Borislav Nikolić, and Rolf Ernst. 2018. Supporting Dynamic Voltage and Frequency Scaling in Networks-On-Chip for Hard Real-Time Systems. In *2018 IEEE 24th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*. IEEE, 125–135.
[11] Adam Kostrzewa, Sebastian Tobuschat, and Rolf Ernst. 2017. Self-aware network-on-chip control in real-time systems. *IEEE Design & Test* 35, 5 (2017), 19–27.
[12] Eberle A. Rambo, Bryan Donyanavard, Minjun Seo, Florian Maurer, Thawra Kadeed, Caio B. de Melo, Biswadip Maity, Anmol Surhonne, Andreas Herkersdorf, Fadi Kurdahi, Nikil Dutt, and Rolf Ernst. 2019. The Information Processing Factory: Organization, Terminology, and Definitions. arXiv:1907.01578
[13] Eberle A. Rambo and Rolf Ernst. 2017. Replica-Aware Co-Scheduling for Mixed-Criticality. In *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, Vol. 76. 20:1–20:20. https://doi.org/10.4230/LIPIcs.ECRTS.2017.20
[14] E. A. Rambo, Y. Shang, and R. Ernst. 2019. Providing Integrity in Real-Time Networks-on-Chip. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* (2019), 1–14. https://doi.org/10.1109/TVLSI.2019.2906471
[15] Minjun Seo and Fadi Kurdahi. 2019 forthcoming. Efficient Tracing Methodology Using Automata Processor. *ACM Transactions on Embedded Computing Systems (TECS)* (2019 forthcoming).
[16] Minjun Seo and Roman Lysecky. 2017. Hierarchical non-intrusive in-situ requirements monitoring for embedded systems. In *International Conference on Runtime Verification*. Springer, 259–276.
[17] Minjun Seo and Roman Lysecky. 2018. Non-Intrusive In-Situ Requirements Monitoring of Embedded System. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 23, 5 (2018), 58.
[18] Stewart W Wilson. 1994. ZCS: A zeroth level classifier system. *Evolutionary computation* 2, 1 (1994), 1–18.
[19] Kisoo Yu. 2012. big.LITTLE Switchers. In *2012 Korea Linux Forum*.
[20] Johannes Zeppenfeld, Abdelmajid Bouajila, Walter Stechele, and Andreas Herkersdorf. 2008. Learning Classifier Tables for Autonomic Systems on Chip. *GI Jahrestagung (2)* 134 (2008), 771–778.