# Run-Time Enforcement of Non-Functional Application Requirements in Heterogeneous Many-Core Systems

Jürgen Teich Behnaz Pourmohseni Oliver Keszocze Jan Spieck Stefan Wildermann

Hardware-Software-Co-Design, Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU)
juergen.teich@fau.de behnaz.pourmohseni@fau.de oliver.keszoecze@fau.de jan.spieck stefan.wildermann@fau.de

**Abstract—For many embedded applications, non-functional requirements such as safety, reliability, and execution time must be guaranteed in tight bounds on a given multi-core platform. Here, jitter in non-functional program execution qualities is caused either by outer influences such as faults injected by the environment, but can be induced also from the system management software itself, including thread-to-core mapping, scheduling and power management. A second huge source of variability typically stems from data-dependent workloads. In this paper, we classify and present techniques to enforce non-functional execution properties on multi-core platforms. Based on a static design space exploration and analysis of influences of variability of non-functional properties, enforcement strategies are generated to guide the execution of periodically executed applications in given requirement corridors. Using the case study of a complex image streaming application, we show that by controlling DVFS settings of cores proactively, not only tight execution times, but also reliability requirements may be enforced dynamically while trying to minimize energy consumption.**

## I. Introduction

In a broad range of embedded systems, e.g., in real-time and safety-critical domains, applications require guarantees (rather than a best-effort behavior) w.r.t. non-functional properties of their execution such as timing and reliability. Delivering the required guarantees is, therefore, of utmost importance for the successful introduction of multi-/many-core architectures in the embedded domains of computing. In a many-core context, existing analysis tools either impose an immense computational complexity or deliver worst-case guarantees that suffer from a massive over-/under-approximation for the vast majority of execution scenarios (due to the inherent uncertainty of these scenarios) and, hence, are of no practical interest. Noteworthy, a major source of this uncertainty originates from the interferences among concurrent applications.

In view of abundant computational and storage resources becoming available, new programming paradigms such as *invasive computing* [1] have proven to be effective in alleviating these interferences by means of spatial isolation among applications. Here, hybrid (static analysis/dynamic mapping) approaches, e.g., [2]–[5], enable a static generation of different mappings for each application on system resources in form of mapping classes rather than individual mappings. For each concrete mapping within such a class, safe bounds on the non-functional execution properties, e.g., latency, may be asserted, see, e.g., [6]. The statically generated and analyzed sets of optimal mapping classes are then provided to the run-time system which checks the availability of such constellations of resources under the current system workload, and, if enough resources are available, finally launches the application [6].

Although spatial isolation among applications significantly reduces the aforementioned uncertainties, a considerable degree of them remain unaffected. This might be unacceptable, e.g., for safety-critical applications. But also, real-world applications from the domain of streaming often exhibit a large jitter in the latency and throughput (in spite of inter-application resource isolation) which is not tolerable, e.g., in case of camera-based medical surgery. This variation mainly stems from two sources of uncertainty that cannot be eliminated or restricted through resource isolation. The *execution state uncertainty* is a result of a combination of environmental (e.g., temperature) and internal (e.g., cache status) influences while the *input uncertainty* results from the application's input as, e.g., in image processing the scene may greatly influence the workload per image.

In the presence of execution state and input uncertainties, *application-specific run-time techniques* can offer a practical approach to confine the non-functional properties of execution to acceptable bounds and to prevent the violation of requirements. Such techniques dynamically adjust a given set of control knobs, e.g., voltage/frequency settings, in reaction to observed (or predicted) changes in the input and/or environment states to steer the non-functional properties of execution within the desired range. For instance, in the context of reactive systems, [7] presents the concept of enforcement of safety properties using automata. For the enforcement of non-functional properties such as latency or power, techniques based on run-time monitoring and control theory have been investigated to minimize energy under timing constraints [8]. While control-oriented approaches often cannot avoid temporal violations, [9] proposes an approach to minimize energy consumption under hard timing constraints by selecting a suitable multiprocessor voltage/frequency setting. We refer to this emerging class of application-specific run-time techniques as *Run-Time Requirement Enforcement (RRE).*

This paper presents the fundamentals, definitions, and taxonomy of RRE in the context of many-core systems. We exemplify the practice of RRE techniques and present a discussion on their advantages, drawbacks, and challenges in a case study on the enforcement of timing and at the same time reliability requirements for a distributed real-time image processing application.

978-1-7281-4123-7/20$31.00 © 2020 IEEE

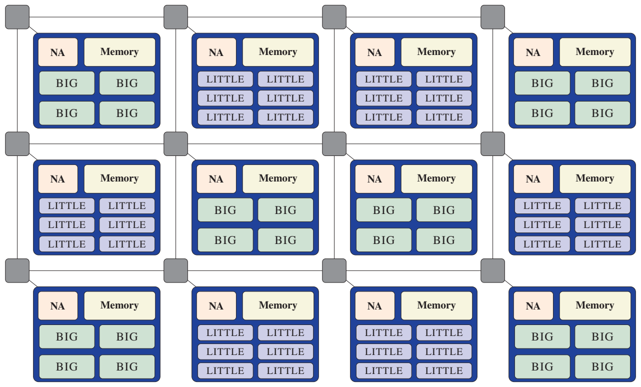


Fig. 1: Example heterogeneous $3 \times 4$ NoC-based many-core architecture.

## II. Preliminaries and Definitions

### A. System Model

A many-core *architecture* is typically organized as a set of so-called compute tiles which are often interconnected by a Network-on-Chip (NoC) for scalability, see, e.g., Fig. 1. Each compute tile itself is typically organized as a multi-core or a processor array and comprises a set of processing cores, peripherals such as memories, and a network adapter which are interconnected via one or more buses. An *application* to be executed on the architecture is typically composed of a set of processing tasks with known data dependencies, provided as a task graph. In case of periodic applications, actor-based models of computation and languages such as ActorX10 [10] may be used for parallel programming of MPSoCs. On top of a program specification, each application or just individual actors may be assigned one or a set of *requirements* on specific non-functional properties of its execution, e.g., execution time, throughput, or power consumption. In the following, a *mapping* of an application on a given architecture corresponds to a binding of its tasks to platform cores, a routing of the data exchanged between communicating tasks, an allocation of the required processing, communication, and storage resources, and a scheduling of tasks and communications on the allocated resources. Alternatively to concrete mappings, a set of constraints that reflect a constellation of required resources and, hence, correspond to several concrete deployments of the application on the architecture may be characterized at design time through techniques of design space exploration [2], [4], [6].

### B. *-Predictability

Non-functional requirements of applications, e.g., real-time constraints, can often be expressed in form of intervals according to the definition for the predictability of a non-functional property from [11]:

*Definition 1 (*-predictability):* Let $o$ denote a non-functional property of a program (implementation) $p$ and the uncertainty of its input (space) given by $I$ and environment by $Q$. The predictability (marker) of objective $o$ for program $p$ is defined by the interval

$$o(p,Q,I) = [\mathit{inf}_o(p,Q,I), \ldots, \mathit{sup}_o(p,Q,I)] \qquad (1)$$

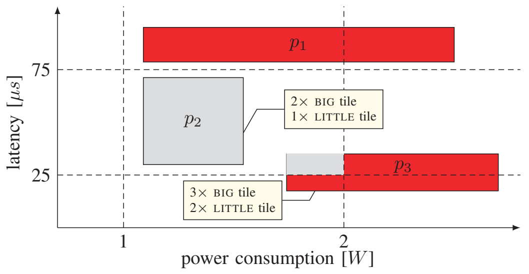


Fig. 2: Example of an application program $p$ with a latency requirement $[25\,\mu s, 75\,\mu s]$ and a power requirement $[1\,W, 2\,W]$. Shown are three program implementations. $p_1$ does not satisfy the latency requirement for any possible execution. $p_2$ satisfies the two requirements for any possible variation in input $i \in I$ and state $q \in Q$. Finally, $p_3$ satisfies the requirements for some executions. Here, RRE techniques might be used to keep $p_3$ within the acceptable region.

where $\mathit{inf}_o$ and $\mathit{sup}_o$ denote the infimum and supremum of property $o$, respectively, under variation of state $q \in Q$ and input $i \in I$.

Figure 2 exemplifies Definition 1 for three implementations $p_1$, $p_2$, and $p_3$ of an application with two requirements in terms of latency and power consumption[1]. The rectangle associated with each implementation $p_i$ confines the observable latency and power range for $p_i$ under the variation of input $i \in I$ and state $q \in Q$. As illustrated, $p_1$ never satisfies the latency requirement under any input/state and, thus, is of no interest. Contrarily, $p_2$ satisfies both requirements in all input/state scenarios which—although offering desirable qualities—is achieved through, e.g., an over-reservation of resources or a persistently maximized core voltage/frequency which is often not affordable and/or practical. Contrarily to $p_1$ and $p_2$, $p_3$ exhibits an attractive case: Under certain input/state scenarios it satisfies latency and power requirements while under others, the acceptable region is left.

In real-life use cases, the observable predictability intervals are often too coarse, so that a large share of viable implementations (like $p_3$) do not satisfy the given requirements under all input/state scenarios. For such partially satisfactory implementations, run-time techniques can be employed to render them consistently satisfactory by regularly monitoring (or predicting) the on-line input/state scenario and either acting pro-actively to avoid any violation of a set of given requirements, e.g., by adjusting the voltage/frequency settings of cores prior to program execution, or in reaction to any observed violation. The purpose of such Run-Time Requirement Enforcement (RRE) techniques is, therefore, to enforce that the desired latency and power corridor is never (or only occasionally) violated.

[1]Note that a lower bound on latency makes sense in many applications that communicate result data to other applications or systems. Here, either buffer limitations would cause overflows in case the producer would be faster than the consumer. Alternatively, data might get lost if the producer overwrites not yet consumed data. Similarly, a minimal lower bound is the default in the case of reliability requirements. There, the lower bound could indicate a minimal expected lifetime. Finally, even lower power bounds can be found in the area of high-performance computing. In fact, the energy bill of a supercomputer increases by the amount of not consumed power but reserved by the provider.

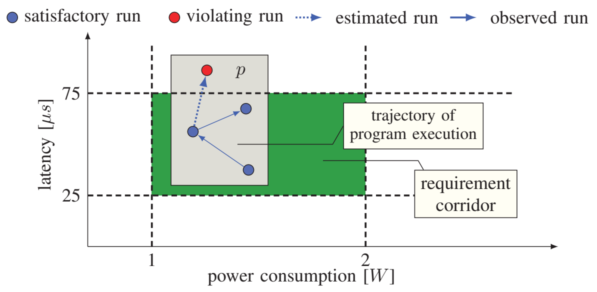


Fig. 3: Example of Run-Time Requirement Enforcement (RRE).

## C. Run-Time Requirement Enforcement

To satisfy a set of given requirements, the observable predictability intervals of the partially satisfactory implementations must be obviously tuned to stay within a requirement corridor spanned by a lower bound $LB_o$ and an upper bound $UB_o$ of an objective $o$. In general, this can be achieved by techniques such as *restricting* the input space $I$ or using *approximate computing* [11]. Alternatively, *isolation* techniques that reduce the state space $Q$ may be applied such as the use of simpler cores, resource reservation protocols, or using invasive computing [1]. In the latter approach, an application program invades a set of processing and communication resources prior to execution. Through inter-application isolation, composability is established which is essential for an independent analysis of individual applications [12]–[14]. In the context of this paper, we define RRE as follows.

*Definition 2 (Run-Time Requirement Enforcer):* A Run-Time Requirement Enforcer (RRE) of a requirement $r_o(p) = [LB_o, UB_o]$ of a program $p$ is a control technique to steer $o$ within the corridor spanned by a lower bound $LB_o$ and an upper bound $UB_o$ for each execution of $p$.

Figure 3 exemplifies Definition 2 for a latency and a power requirement of a program $p$ implementing an application. Here, the task of an RRE is to confine the observable predictability interval of $p$ within the corridor as specified by the latency and power requirements. For a given input and environmental state, the RRE in this case pro-actively estimates the expected latency $L_{\text{est}}$ and power consumption $P_{\text{est}}$. Based on these estimates, it takes actions with the goal to avoid any violation of the requirements. Examples of RRE actions include adjusting the voltage/frequency of the cores or awaking reserved cores that are currently in a sleep state for power reduction, or even changing the mapping of some tasks to other cores [15].

## D. Taxonomy of Run-Time Requirement Enforcers

According to [11], each requirement of an application can be either *soft* or *hard*. In case of a soft requirement, occasional violations are still considered acceptable. In this context, an RRE of program $p$ can be classified as *strict* if it can be formally proven that no concrete execution of $p$ will leave the given corridor at run time. It is called *loose*, if one or multiple consecutive violations of $o$ are tolerable.

Furthermore, an RRE can be classified as a *centralized* enforcement technique if a single enforcer instance is used to enforce the requirement. It is called *distributed* in case multiple enforcers jointly enforce the requirement.

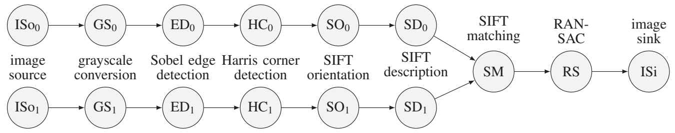


Fig. 4: Stitching streaming application for two input images.

Each invaded tile is equipped with a *Run-Time Requirement Monitor (RRM)* that provides the RRE with the necessary run-time information to enforce a set of given requirements. The run-time information is kept locally on the tile in case of distributed RRE or sent to a single RRE instance in case of the centralized RRE technique.

## III. Case Study

In this section, we present a case study to explain the concepts of RRE for a simultaneous enforcement of timing and reliability requirements for a stitching streaming application as depicted in Fig. 4. The application consists of two 6-actor chains, each processing one input image in succession through an image source (ISo$_i$) actor to read in input images periodically at a constant rate, a gray-scale conversion (GS$_i$) actor, Sobel edge detection (ED$_i$) and Harris corner detection (HC$_i$) actors to determine respectively edges and corners in an image, a SIFT orientation (SO$_i$) actor to achieve invariance to image rotation, and a SIFT description (SD$_i$) actor to extract the features in an image. The output of the SD actors form the two chains is then provided to a SIFT matching (SM) actor to detect common features of both images, and a RANSAC (RS) actor to calculate the transformation between both images based on the matched features. The image is finally sent out by an image sink (ISi) actor. As platform, we consider a heterogeneous NoC-based $3 \times 4$ many-core architecture composed of two types of compute tiles as depicted in Fig. 1. Here, each compute tile of type LITTLE comprises 6 low-power cores while tiles of type BIG are composed of 4 high-performance cores.

## A. Invasive Programming

*Invasive computing* [1] has been shown to enable resource-awareness for parallel programs on multi-core targets. Here, an initial *claim* is requested from the operating system, containing a set of processing resources, memory and communication resources that the application can exclusively use for its parallel execution. Claim construction is done by issuing a call to *invade* with a set of constraints that describe the required claim. After that, *infect* is used to start the actual application code on the allocated *claim*. Once the execution on all cores finishes, the *claim constraints* can be altered by calling *invade* or *retreat* to, e.g., expand or shrink the application's *claim*.

Programming support for these has been developed in the form of InvadeX10, a library-based extension of X10, a modern parallel programming language to program scalable multi-core systems. An invasive program written in InvadeX10 basically looks like this:

```
val claim = Claim.invade(constraints);
claim.infect(code);
claim.retreat();
```

```
public static def main(args:Rail[String]) {
  // Declare global requirement on reliability
  @REQUIRE("ag", new PFH(0.001, 0.0000001))
  val ag = new ActorGraph("ag");

  // Declare actors
  ag.addActor(new SourceActor("ISo_0"));
  ag.addActor(new GrayScaleConvActor("GS_0"));
  ...

  // Declare requirements on latency and power
  // for SD actor
  @REQUIRE("SD_0", new Latency(0, 100, "ms", "hard"))
  @REQUIRE("SD_0", new Power(0, 1, "W", "soft"))
  ag.addActor(new SIFTDescriptionActor("SD_0"));
  ...
}
```

Fig. 5: Example of actor graph generation and execution in ActorX10 as well as annotations regarding requirements on objectives.

An example of a portion of an invasive program written in ActorX10 [10] for the case study is provided in Fig. 5. Besides generation of the actors and the actor graph, requirements on objectives can be specified as annotations. In the example, requirements on probability of failures per hour (PFH) and power consumption are specified. Moreover, an upper latency bound of 100 ms is specified for the SD actor. Ideally, an RRE would automatically be generated to ensure the satisfaction of the requirements.

## B. Enforcement Problem Description

In our case study, we consider the distributed and simultaneous enforcement of timing and reliability requirements for the SIFT description (SD) actors, namely, $SD_0$ and $SD_1$, as these have been pre-characterized to provide the highest contribution to the overall processing latency per frame but also having a high degree of input-dependent execution time variation. Here, each periodic execution of each SD actor (corresponding to processing one input image) shall be completed within a strict latency upper bound of $UB_L = 100\,\text{ms}$. Moreover, subject to an observed Soft Error Rate (SER), each iteration of each SD actor can be performed either non-redundantly or in a Triple Modular Redundant (TMR) fashion to enhance reliability. For the enforcement of the given requirements, the RRE responsible for each actor is privileged to adjust two control knobs of the respective tile prior to the enforced actor processing an input image: a) the degree of execution parallelism adjusted by setting the number $n$ of active cores used for processing the input image and b) the power (voltage/frequency) mode $m$ of the cores adjusted through Dynamic Voltage and Frequency Scaling (DVFS) (for active cores) and power gating (for inactive cores) [16]–[19]. To this end, each RRE decides on a per-image basis how the workload of its enforced actor must be distributed on 1–6 cores in case of execution on a LITTLE tile or 1–4 cores in case of a BIG tile. At the same time, it sets the power mode of the cores to either a power-gated mode ($V_{DD} = 0$) for inactive cores or one of the DVFS configurations given in Fig. 6 (left) for the two core types.

For the SD actors under enforcement, we analyzed the major source of latency variation to lie in the variability

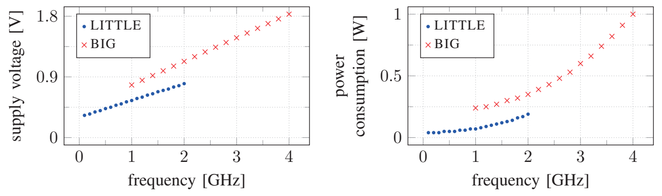


Fig. 6: Supply voltage (left) and power consumption (right) for the BIG and the LITTLE core types versus their operating frequency.

of the number $i$ of *features* to be processed for each input image. Thus, in our case study, we use this number as an indicator of the actor's workload. As a merit of profit, we investigate the energy savings achievable through an enforced execution of the actors in addition to the satisfaction of their timing and reliability requirements. Moreover, besides enforcement schemes invading just a single tile of type LITTLE or BIG per actor, we also investigate an enforcement scheme called COMBINED where each enforced actor is decided to be executed on either an invaded LITTLE or an invaded BIG tile. For the latter scheme, each input of an enforced actor is sent to two invaded tiles[2]. The RRE on an invaded tile then also decides on which tile type the current instance of the enforced actor is executed, subject to the current workload and the demanded redundancy scheme while the other tile is power-gated. After processing the input image, the activated actor instance sends the output image to the subsequent SM actor. In the following, we exploit sampling-based techniques to determine the relationship between latency, power consumption, and energy demand of actors to be enforced in order to characterize suitable enforcement decisions.

## C. Power, Latency, and Energy Modeling

To evaluate the power consumption $P(m)$ of a core in power mode $m$, we use Eq. (2) in which the first summand represents the dynamic power contribution based on the effective switching capacitance $C_{\text{eff}}$, supply voltage $V_{\text{DD}}(m)$, and operating frequency $f(m)$ of the core in power mode $m$. The second summand describes the static power consumption based on leakage current $I_{\text{leak}}$ and supply voltage $V_{\text{DD}}(m)$. The power consumption of each core type is also given in Fig. 6 (right).

$$P(m) = C_{\text{eff}} \cdot V_{\text{DD}}(m)^2 \cdot f(m) + I_{\text{leak}} \cdot V_{\text{DD}}(m) \qquad (2)$$

For the construction of proper enforcement strategies for the SD actors, we need to know the relation between the number $i$ of input features and the execution latency $L$ in dependence of the number $n$ of cores and power mode $m$. Let $L(1, 1, m_{\max})$ denote the latency for processing one feature on one core in power mode $m_{\max}$ (highest voltage and frequency). First, the execution latency of each actor is determined by simulating a total of 1 224 input image pairs (to be stitched) as a representative set of the considered input space. Subsequently, the latency $L(1, 1, m_{\max})$ per feature of the SD actor is

[2]Instead of statically invading one BIG and one LITTLE tile for the SD actor, also real-time task migration may be applied, see, e.g., [15]. Moreover, applications with no requirements (best-effort workload) may be executed at times where the enforced actor does not use one tile.

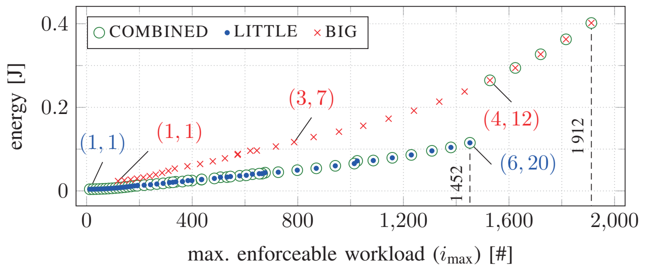


Fig. 7: Statically characterized Pareto-optimal $(n, m)$ configurations in the space of energy consumption and maximum enforceable workload $i_{\max}$ for an energy-minimizing enforcement of the SD actors given a latency upper bound of $UB_L = 100$ ms. The front is given for three cases of using a) a BIG tile, b) a LITTLE tile, and c) a COMBINED enforcement scheme which can switch between both tiles.

determined for each image by dividing its latency by the number of features $i$ in that image. As we target latency as a hard requirement, the maximum observed per-feature latency among all images is used as $L(1, 1, m_{\max})$ which is equal to $0.21\,\mu$s on a BIG core and $0.41\,\mu$s on a LITTLE core for a clock frequency $f(m_{\max})$.

The following Eq. (3) is then used to determine an upper bound on the actor latency $L(i, n, m)$ based on the number of features $i$ to be processed within an image, the number of cores $n$ employed, and the power mode $m$ selected by an RRE scheme. In Eq. (3), $e(n)$ denotes the parallel efficiency in dependence of the number of cores $n$ employed. In our experiments, we consider the best case of $e(n) = 1$.

$$L(i, n, m) = L(1, 1, m_{\max}) \cdot \left\lceil \frac{i}{n \cdot e(n)} \right\rceil \cdot \frac{f(m_{\max})}{f(m)} \qquad (3)$$

Note that Eq. (3) is a latency model specific to the SD actors of our running application where $L(1, 1, m_{\max})$ must be determined individually for each actor to be enforced. Moreover, Eq. (3) could be alternatively replaced with an elaborate many-core timing analysis, e.g., those from [6], [20]–[24], to derive tight worst-case latencies that support a variety of different resource arbitration policies and resource sharing schemes.

The energy $E(i, n, m)$ required by the actor for processing an image with $i$ features using $n$ cores running in power mode $m$ is then derived using Eq. (4).

$$E(i, n, m) = L(i, n, m) \cdot P(m) \cdot n \qquad (4)$$

Finally, the maximum number of features that can be processed within a given latency bound $UB_L$ using $n$ active cores running in power mode $m$ can be determined using Eq. (5) which is derived from Eq. (3), considering $L(i, n, m) \leq UB_L$.

$$i_{\max}(UB_L, n, m) = \left\lfloor n \cdot e(n) \cdot \left\lfloor \frac{UB_L}{L(1,1,m_{\max})} \cdot \frac{f(m)}{f(m_{\max})} \right\rfloor \right\rfloor \qquad (5)$$

For example, $i_{\max}(100, 4, 20)$ denotes the highest number $i$ of features for which a latency upper bound of $UB_L = 100$ ms for the SD actor can be enforced using $n = 4$ cores and power mode $m = 20$. Note that Eq. (5) must be developed separately for each core type, e.g., BIG and LITTLE in our example.

## D. Energy-Minimized Timing Enforcement

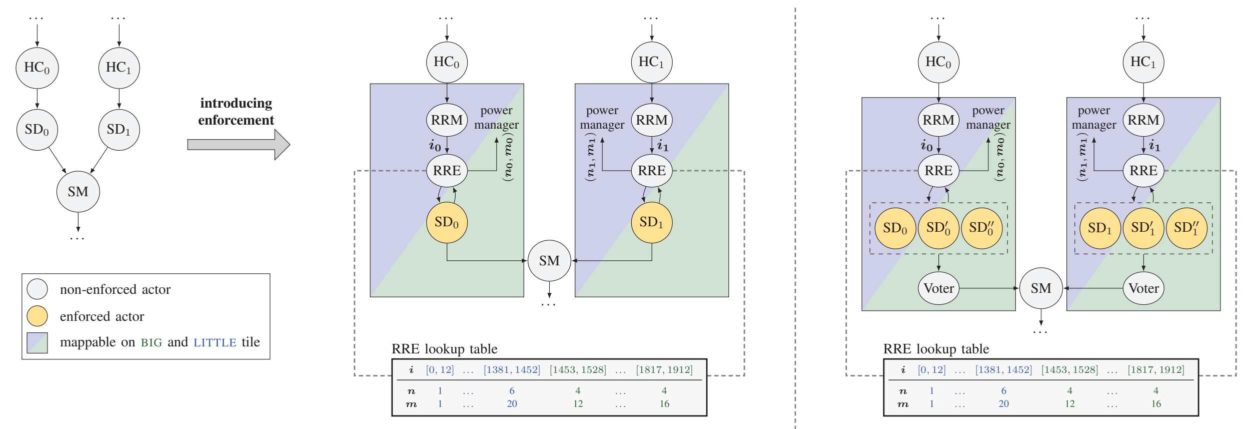
In general, requirement enforcement may involve to set, modify, or impose restrictions on typically OS-related techniques such as thread scheduling, or memory management. In our case study, the enforcement is realized by varying the number $n$ of active cores (parallelism) and their power mode $m$ for each execution iteration of each enforced SD actor. Since in general, multiple settings for $n$ and $m$ might enforce the given requirements, the question becomes which requirement-adhering $(n, m)$ configuration to select at run time. Often, this freedom of choice may be exploited by optimizing one or more (secondary) objectives in addition to satisfying the given requirements. In the following, we consider energy consumption as a secondary objective to be minimized.

Given the latency upper bound of $UB_L = 100$ ms and the RRE decision space of $n \in [1, 4]$ and $m \in [1, 16]$ in case of a BIG tile or $n \in [1, 6]$ and $m \in [1, 20]$ in case of a LITTLE tile according to Fig. 6, design space exploration can be conducted per enforced actor to derive, e.g., in our running example for the SD actors, the maximum number $i_{\max}$ of features that can be processed under each choice of $(n, m)$ while respecting the given latency bound. Form the 64 (or 120) possible $(n, m)$ configurations for a BIG (or LITTLE) tile, the Pareto-optimal configurations in the space of maximum enforceable workload $i_{\max}$ (to be maximized) and the respective energy demand (to be minimized) can then be identified and retained to construct the RREs. Figure 7 illustrates the distribution of the Pareto-optimal $(n, m)$ configurations in the space of maximum enforceable workload $i_{\max}$ and energy, derived using Eq. (5) and Eq. (4), respectively, for the two tile types LITTLE and BIG. Accordingly, the Pareto-optimal configurations for a COMBINED enforcement scheme using both tile types (one tile of each type) are outlined with a green circle in Fig. 7.

Based on such a design space exploration and the Pareto front of $(n, m)$ configurations derived thereby, an energy-minimizing enforcement scheme may be systematically constructed in which prior to each execution of the SD actor, the RRE selects the energy-minimal $(n, m)$ configuration that satisfies the latency requirement if it is enforceable for the current input workload $i$. According to Fig. 7, on a BIG tile, the latency bound of $UB_L = 100$ ms is enforceable for input images with up to $i = 1,912$ features. In case of a LITTLE tile, the maximum enforceable workload is restricted to $i = 1,452$ features. Finally, in case of a COMBINED enforcement scheme, the LITTLE tile can be used for input images with $i \leq 1,452$ features while the BIG tile is used for images with $i > 1,452$ features. In all three schemes, the RRE selects the energy-minimal $(n, m)$ configuration solely based on the number $i$ of features in the image to be processed by the SD actor. Note that for non-enforceable sizes of input, the enforcer needs to either throw an exception, drop the image, or process only as much as the latency bound allows to be processed.

## E. Reliability Enforcement

In our case study, also a reliability requirement needs to be enforced according to Fig. 5. Depending of the observed SER, this soft requirement can be respected by switching from a non-redundant execution of the enforced SD actors to a



RRE lookup table

| $i$ | [0, 12] | … | [1381, 1452] | [1453, 1528] | … | [1817, 1912] |
|---|---|---|---|---|---|---|
| $n$ | 1 | … | 6 | 4 | … | 4 |
| $m$ | 1 | … | 20 | 12 | … | 16 |

a) non-replicated variant

RRE lookup table

| $i$ | [0, 12] | … | [1381, 1452] | [1453, 1528] | … | [1817, 1912] |
|---|---|---|---|---|---|---|
| $n$ | 1 | … | 6 | 4 | … | 4 |
| $m$ | 1 | … | 20 | 12 | … | 16 |

b) TMR variant

Fig. 8: Implementation of distributed RRE: Depending on the input $i$, the pre-explored energy-optimal parallelism degree and the DVFS settings $(n, m)$ are determined for each of the two SD actors as well as whether the actor instance is executed on a BIG or a LITTLE tile. Moreover, depending on the observed SER, it is processed in either a non-replicated (a)) or a TMR fashion (b)).

TMR execution scheme (see a pseudo-code of the enforcer in Fig. 9). In view of an RRE constructed as detailed before, a TMR execution of an SD requires to compute each of the $i$ feature threads three times and perform majority voting on the results. This is considered by replicating the workload from $i$ to $i' = 3 \times i$ features to be processed within the given latency upper bound. Thus, it suffices for the RRE to select the energy-minimal $(n, m)$ configuration corresponding to a maximum workload of $i' = 3 \times i$ if a TMR execution scheme is needed to enforce the reliability requirement. Evidently, under a TMR scheme, the latency upper bound $UB_L = 100$ ms is enforceable for input images with up to $i = 484$ features for an execution on a LITTLE tile while it can be enforced for images with up to $i = 637$ features using a BIG tile.

## F. Implementation

To enable such an enforced execution of the SD actors, we adapt the application by inserting an RRE actor before each instance of each SD actor, thus, conducting a model transformation at the level of actor graphs.

For the exemplary COMBINED enforcement scheme, Fig. 8 shows the resulting deployment of the two enforced SD actors, namely, SD$_0$ and SD$_1$, of the stitching application. Each enforced actor SD$_x$ (instantiated on a BIG and/or a LITTLE tile) is preceded by a RRM to extract the number $i_x$ of features in the current input image and a RRE actor which activates the optimal $(n, m)$ configuration based on $i_x$ before the processing of the SD$_x$ actor begins. To that end, the statically computed Pareto-optimal $(n, m)$ configurations to be used by each RRE are stored in a lightweight lookup table on the respective tile. Moreover, the RRE decides between a non-redundant execution of the enforced actor (see, Fig. 8a) or a TMR execution scheme (see, Fig. 8b), subject to the observed SER at run time. The pseudo-code of this combined timing and reliability enforcer is shown in Fig. 9.

Note that, in case two tiles (one of type BIG and one of type LITTLE) are invaded to host an SD actor, each input image of that actor is processed on only one of the two tiles: At run time, the output of each HC actor is sent to both tiles that host the instances of the subsequent SD actor. Then, the local RRE

```
// Code of RRE for the SD Actor
static class SD_RRE_Actor extends Actor {
  ...
  protected def task () {
    // Features are available at the input port
    // of the actor
    val inToken = inPort.read();
    var i = inToken.features.size();

    // If SER is bigger than serThreshold, TMR is used,
    // which increases workload by factor 3
    useTMR = false;
    if (SER > serThreshold) {
     useTMR = true;
     i *= 3;
    }

    // Look up results from DSE
    val targetTileType = getBigOrLittle(i);
    val numCores = getCores(i);
    val powerMode = getPowerMode(i);

    // Set power mode via OS interface if the RRE is
    // on the correct tile (big or little)
    if (getMyTile() == targetTileType) {
      osPowerManager.setMode(numCores, powerMode);
      ...
    }
  }
}
```

Fig. 9: RRE code of SD actor.

on each tile determines based on the number of features $i$ to be processed a) whether the image is to be processed on its tile and, if so, b) sets the optimal $(n, m)$ configuration to be used, see Fig. 9. After the settings have been adapted by the power manager, the processing of the SD actor is initiated (method *infect* in invasive computing [1]). The cores on the tile that is not used for processing the current image are power-gated. Once the processing of the active actor instance is complete, the output is sent to the SM actor.

In the RRE tables shown in Fig. 8, the columns displayed in blue denote the range of workload $i$ and the corresponding optimal $(n, m)$ configurations for which the LITTLE tile will be active. The entries given in green denote the range of workload and the corresponding optimal $(n, m)$ configuration for which the BIG tile is active. For instance, for input images

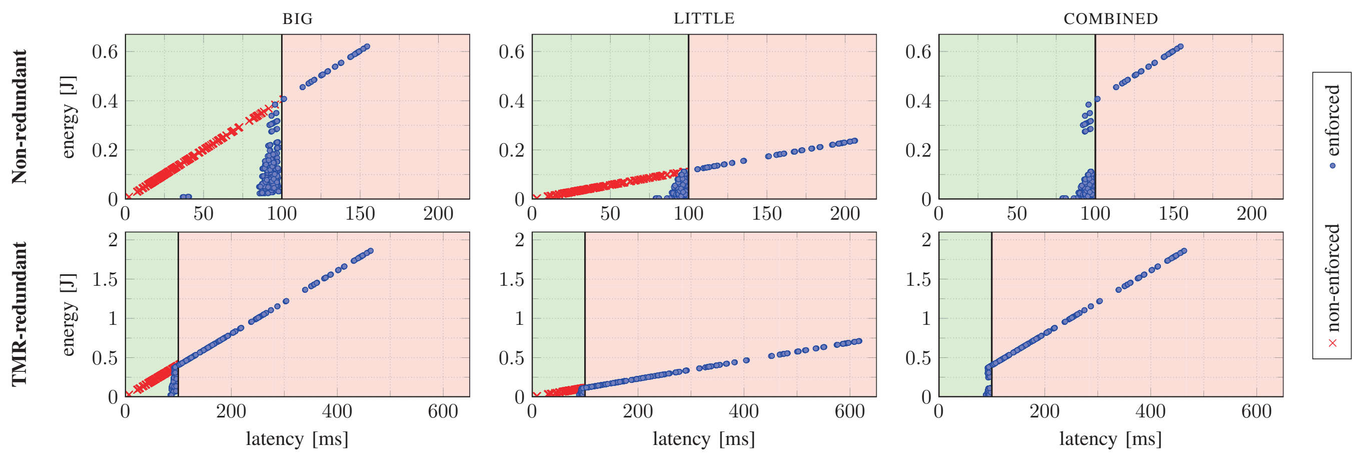


Fig. 10: Distribution of the energy consumption and the latency of the $SD_0$ actor for processing the 1 224 input images without enforcement (red plots) and with enforcement (blue plots) for non-replicated (top row) and TMR-redundant (bottom row) schemes when executed on a BIG tile (left column), a LITTLE tile (middle column), or under the COMBINED enforcement scheme (right column).

with $i \in [1,381,\ 1,452]$ features, the processing of the SD actor will be performed on the LITTLE tile using $n=6$ cores running on power mode $m=20$ while the cores on the BIG tile are power-gated. For input images with $i \in [1,453,\ 1,528]$ features, the processing is performed on the BIG tile using $n=4$ cores running on power mode $m=12$ while the cores on the LITTLE tile are power-gated. In case a TMR scheme is required, each RRE first calculates the total number $i'=3\times i$ of features that must be processed for the given input image with $i$ features. The optimal $(n,m)$ configuration is then selected based on $i'$.

### G. Results Discussion

Figure 10 illustrates the distribution of energy consumption and latency for the exemplary $SD_0$ actor when processing 1 224 input images varying in number of features $i$ to be processed between $i = 43$ and $i = 3\,022$, provided for two cases of without enforcement (red crosses) and with enforcement (blue circles). The image test sequence has been chosen on purpose to include also images with $i > 1,912$ which cannot be enforced even on a BIG tile in a non-redundant mode of processing. The plots in the top row correspond to a non-redundant execution of $SD_0$ while those in the bottom row correspond to a TMR execution of $SD_0$. The results are provided for the three scenarios of executing $SD_0$ on a BIG tile (left column), a LITTLE tile (middle column), or under the COMBINED enforcement scheme (right column). Accordingly, Fig. 11 illustrates the histograms of observable latencies of the $SD_0$ actor without enforcement (red) and with enforcement (blue) for the redundancy and deployment schemes stated above. Without enforcement, the $SD_0$ actor is always executed using all cores available on the tile running on the maximum frequency. By employing the described enforcement strategy, the run-time manager is no longer compelled to constantly run the $SD_0$ actor with the maximum number $n$ of cores using the highest power mode $m$ and in a TMR scheme to guarantee the satisfaction of the given latency and reliability requirements in presence of input and SER variations. Instead, in an enforced execution of the $SD_0$ actor, the RRE chooses a $(n,m)$ configuration that minimizes energy consumption while still respecting the given latency upper bound, $UB_L = 100\,\text{ms}$.

Table I provides the average energy consumption of the $SD_0$ actor under different enforcement and redundancy scenarios, normalized to the maximum energy consumption which happens in the case of non-enforced TMR-redundant execution on a BIG tile. For all redundancy and enforcement schemes, the results verify the energy efficiency of the enforced executions compared to their non-enforced variants. Finally, Table II provides the proportion of timely executions of the $SD_0$ actor under different enforcement and redundancy scenarios, normalized to the ratio of timely executions for the case of non-enforced TMR-redundant execution on a BIG tile. A comparison among the three enforcement schemes, namely, BIG, LITTLE, and COMBINED, in terms of energy consumption (Table I) and proportion of timely executions (Table II) demonstrates that using a BIG tile offers the highest proportion of timely executions at the cost of the highest energy consumption. On the other hand, relying on a LITTLE core leads to the lowest energy consumption at the cost of a reduced proportion of timely executions. Alternatively, by switching between the two tile types, the COMBINED scheme offers the same proportion of timely executions as the BIG scheme, yet with a reduced energy consumption as given in Table I. Note that the additional execution time and energy consumption of the RREs themselves can be neglected as these are implemented by simple table lookups.

TABLE I: Average normalized energy dissipation.

| redundancy scheme | without enforcement | | with enforcement | | |
|---|---|---|---|---|---|
| | BIG | LITTLE | BIG | LITTLE | COMBINED |
| non-redundant | 0.33 | 0.13 | 0.28 | 0.11 | 0.20 |
| TMR-redundant | 1.0 | 0.38 | 0.95 | 0.37 | 0.88 |

TABLE II: Normalized ratio of timely executions.

| redundancy scheme | BIG | LITTLE | COMBINED |
|---|---|---|---|
| non-redundant | 1.5 | 1.4 | 1.5 |
| TMR-redundant | 1.0 | 0.7 | 1.0 |

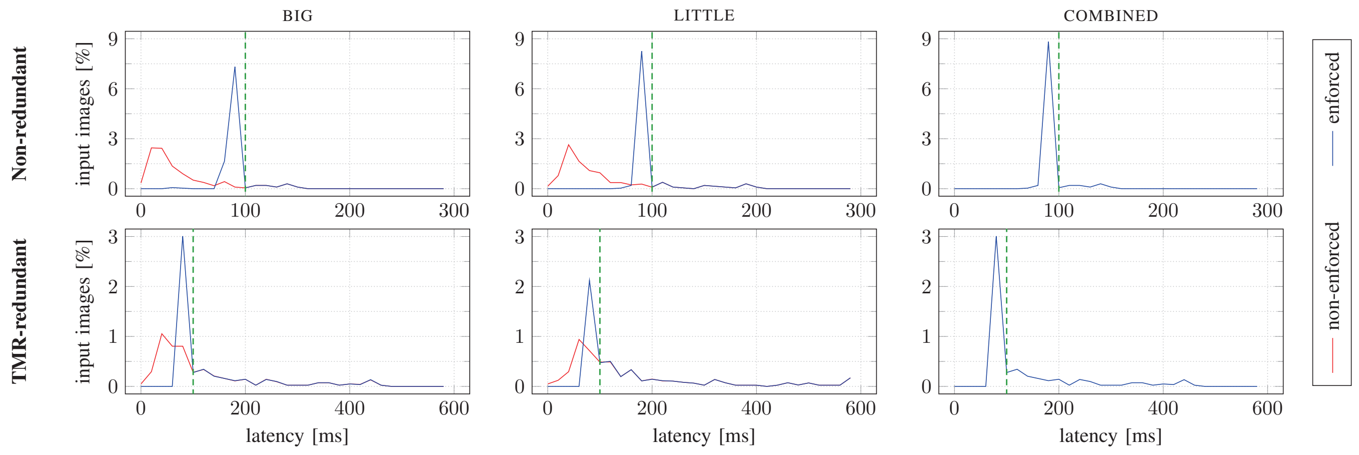


Fig. 11: Latency distribution for the $SD_0$ actor for processing the 1 224 input images without enforcement (red plots) and with enforcement (blue plots) for non-redundant (top row) and TMR-redundant (bottom row) schemes when executed on a BIG tile (left column), a LITTLE tile (middle column), or under the COMBINED enforcement scheme (right column).

## IV. Conclusions

In this paper,we presented a formalization, classification, and the practice of a class of run-time techniques subsumed under the term of Run-Time Requirement Enforcement (RRE) that make the system management software of an MPSoC platform become the advocate of a parallel application program instead of both acting independently with the goal to provide means for the satisfaction of given non-functional requirements of parallel program execution such as performance (latency, throughput), power or energy consumption, or reliability. The non-functional requirements can thereby be expressed by interval ranges and specified over the application program as a whole, e.g., when specified by an actor graph. Alternatively, requirements can be specified for individual actors/tasks or threads, or even segments thereof. The goal of RRE is to enforce the satisfaction of these requirements at run time. In this paper, we have shown an example of enforcers for timing and reliability requirements. For the cases where input might not be enforceable any more, techniques such as input omission (dropping), approximate computing to trade off processing speed with result accuracy (if applicable), revision of scheduling decisions, over-allocation of resources, or a dynamic reconfiguration between different mappings at run time might deliver solutions from case-to-case.

## Acknowledgement

This work is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – Project Number 146371743 - TRR 89 Invasive Computing.

## References

[1] J. Teich *et al.*, *Invasive Computing: An Overview*. Springer, 2011.
[2] T. Schwarzer *et al.*, "Symmetry-eliminating design space exploration for hybrid application mapping on many-core architectures," *IEEE TCAD*, vol. 37, no. 2, 2018.
[3] A. K. Singh *et al.*, "Accelerating throughput-aware runtime mapping for heterogeneous MPSoCs," *ACM TODAES*, vol. 18, no. 1, pp. 9:1–9:29, 2013.
[4] A. Weichslgartner *et al.*, "DAARM: Design-time application analysis and run-time mapping for predictable execution in many-core systems," in *Proc. CODES+ISSS*. IEEE/ACM, 2014, pp. 1–10.
[5] P. N. Khanh *et al.*, "Incorporating energy and throughput awareness in design space exploration and run-time mapping for heterogeneous MPSoCs," in *Proc. DSD*. IEEE, 2013, pp. 513–521.
[6] A. Weichslgartner *et al.*, *Invasive Computing for Mapping Parallel Programs to Many-Core Architectures*. Springer, 2018.
[7] S. Pinisetty *et al.*, "Runtime enforcement of reactive systems using synchronous enforcers," in *Proc. ACM SIGSOFT Int. SPIN Symp. Model Checking of Software*, 2017, pp. 80–89.
[8] C. Imes *et al.*, "POET: a portable approach to minimizing energy under soft real-time constraints," in *Proc. RTAS*, 2015, pp. 75–86.
[9] M. Damavandpeyma *et al.*, "Throughput-constrained DVFS for scenario-aware dataflow graphs," in *Proc. RTAS*, 2013.
[10] S. Roloff *et al.*, "ActorX10: an actor library for X10," in *Proc. ACM SIGPLAN Workshop on X10*. ACM, 2016, pp. 24–29.
[11] J. Teich *et al.*, "Language and compilation of parallel programs for *-predictable MPSoC execution using invasive computing," in *Proc. MCSOC*. IEEE, 2016.
[12] H. Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications*, 2nd ed. Springer, 2011.
[13] B. Akesson *et al.*, "Composability and predictability for independent application development, verification, and execution," in *Multiprocessor System-on-Chip*. Springer, 2011, pp. 25–56.
[14] A. Hansson *et al.*, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM TODAES*, vol. 14, no. 1, p. 2, 2009.
[15] B. Pourmohseni *et al.*, "Hard real-time application mapping reconfiguration for NoC-based many-core systems," *Real-Time Systems*, pp. 1–37, 2019.
[16] A. K. Singh *et al.*, "Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems," in *Proc. DAC*. IEEE/ACM, 2013, p. 115.
[17] A. Kanduri *et al.*, "Approximation-aware coordinated power/performance management for heterogeneous multi-cores," in *Proc. DAC*, 2018, pp. 1–6.
[18] D. Angioletti *et al.*, "A runtime resource management policy for OpenCL workloads on heterogeneous multicores," in *Proc. DATE*, 2019.
[19] Z. Zhu *et al.*, "Energy minimization for multi-core platforms through DVFS and VR phase scaling with comprehensive convex model," *IEEE TCAD*, 2019.
[20] B. Pourmohseni *et al.*, "Isolation-aware timing analysis and design space exploration for predictable and composable many-core systems," in *Proc. ECRTS*, 2019.
[21] S. Altmeyer *et al.*, "A generic and compositional framework for multicore response time analysis," in *Proc. RTNS*, 2015.
[22] R. I. Davis *et al.*, "An extensible framework for multicore response time analysis," *Real-Time Systems*, pp. 1–55, 2017.
[23] G. Giannopoulou *et al.*, "Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems," in *Proc. EMSOFT*. ACM, 2012, pp. 63–72.
[24] ——, "Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources," *Real-Time Systems*, vol. 52, no. 4, pp. 399–449, 2016.