Chapter 9
Run-Time Enforcement of Non-functional Program Properties on MPSoCs

Jürgen Teich, Pouya Mahmoody, Behnaz Pourmohseni, Sascha Roloff, Wolfgang Schröder-Preikschat, and Stefan Wildermann

9.1 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 characteristics 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* [22] have proved effective in alleviating these interferences by means of spatial isolation among applications. Here, hybrid (static analysis/dynamic mapping) approaches, e.g., [11, 19, 20, 24], 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 hold, see, e.g., [25]. 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

J. Teich (✉) · P. Mahmoody · B. Pourmohseni · S. Roloff · W. Schröder-Preikschat · S. Wildermann
Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany
e-mail: juergen.teich@fau.de

J.-J. Chen (ed.), *A Journey of Embedded and Cyber-Physical Systems*,
https://doi.org/10.1007/978-3-030-47487-4_9

application [25]. Such a hybrid approach has been implemented within the language InvadeX10, a library-based extension of the X10 programming language. In this extension, the so-called *requirements* [23] on non-functional execution properties, e.g., latency, may be annotated to individual applications or program segments thereof.

Although spatial isolation among applications significantly reduces the aforementioned uncertainties, a considerable degree of them remain unaffected which 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 intolerable or annoying variation mainly stems from two sources of uncertainty that cannot be eliminated or restricted through resource isolation:

- **Execution State Uncertainty.** This source of uncertainty originates either from the environment (termed exogenous), e.g., ambient temperature, or from within the computing system itself (termed endogenous), e.g., cache states or the voltage/frequency modifications applied by the power manager. While the vast majority of exogenous sources of uncertainty cannot be avoided or controlled, endogenous sources of uncertainty may be eliminated, e.g., by flushing caches before execution or by pinning the voltage/frequency of each core to a desired fixed level.
- **Input Uncertainty.** This source of uncertainty originates from the application's input(s). For instance, in image processing, the content of a scene may greatly influence the amount of workload to be processed 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 within acceptable bounds or 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. Examples of such approaches include the enforcement of safety properties using automata [13] or the satisfaction of timing constraints (while minimizing energy) using control-theory oriented approaches [9]. 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 different classes of RRE techniques and present a discussion on their advantages, drawbacks, and challenges in a case study on the enforcement of timing requirements for a distributed real-time image processing application.

9.2 Preliminaries and Definitions

9.2.1 System Model

A many-core *architecture* is typically organized as a set of so-called storage, I/O, and compute tiles which are interconnected by a Network-on-Chip (NoC) for scalability, see, e.g., Fig. 9.1. Memory and I/O tiles enable mass storage and off-chip access, respectively. Each compute tile 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 [17] may be used for parallel programming of MPSoCs. Each application may be augmented with one or a set of *requirements* on specific non-functional properties of its execution, e.g., execution time, throughput, or power corridors. 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

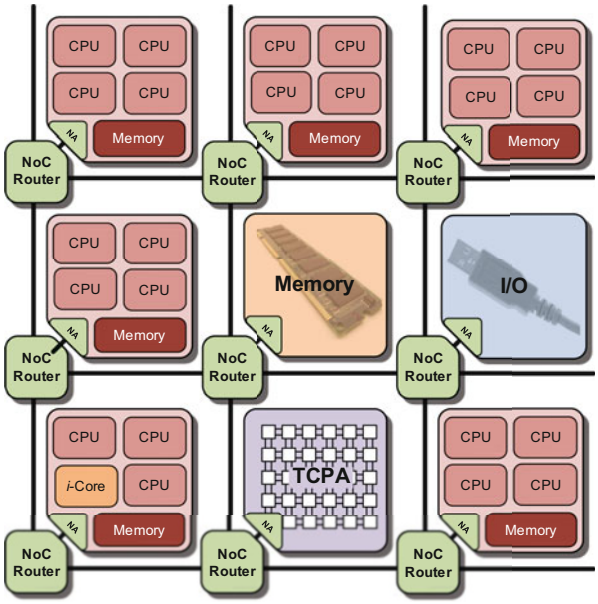

Fig. 9.1 A heterogeneous invasive MPSoC architecture

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 [19, 24, 25].

9.2.2 *-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 [23]:

Definition 9.1 (*-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

$$o(p, Q, I) = [inf_o(p, Q, I), \ldots, sup_o(p, Q, I)] \tag{9.1}$$

where inf_o and sup_o denote the infimum and supremum of property o, respectively, under variation of state $q \in Q$ and input $i \in I$.

Figure 9.2 exemplifies Definition 9.1 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 the requirements (with an affordable resource demand), while under other scenarios the acceptable latency-power region is surpassed.

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

[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.

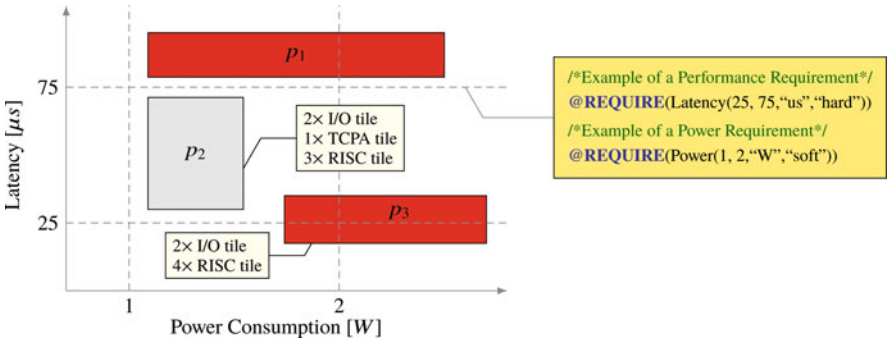

Fig. 9.2 Example of a program p with a latency requirement and a power requirement given each by an interval (corridor). Shown are three program implementations. p_1 does not satisfy the latency requirement for any possible execution. p_2 satisfies the two requirements for any possible variation in input $i \in I$ and state $q \in Q$. Finally, p_3 may satisfy the two requirements, but obviously not for all observable executions. Here, run-time requirement enforcement techniques might be applicable to control the resources of the platform based on run-time monitoring to stay within the requirement corridors

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 online input/state scenario and either acting proactively 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 techniques is, therefore, to enforce that the desired latency and power corridor are never (or only occasionally) violated. We refer to these application-specific run-time techniques as Run-Time Requirement Enforcement (RRE) in the following.

9.3 Run-Time Requirement Enforcement

To satisfy a set of given requirements, the observable predictability intervals of the partially satisfactory implementations must be obviously reduced. In general, this can be achieved by techniques such as *restricting* the input space I or using *approximate computing* [23]. 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* [22]. 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 [1, 8, 12].

Definition 9.2 (Run-Time Requirement Enforcer (RRE)) A Run-Time Requirement Enforcer (RRE) of a requirement $r_o(p) = [LB_o, UB_o]$ of a program p is a

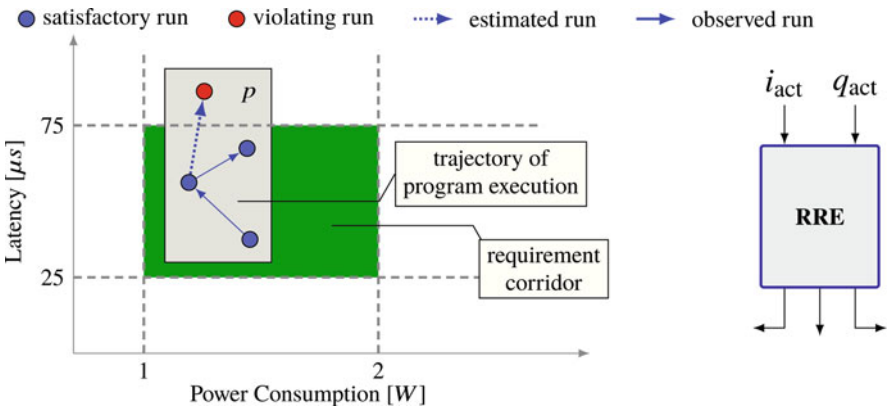

Fig. 9.3 Example of Run-Time Requirement Enforcement (RRE)

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 9.3 exemplifies Definition 9.2 for a latency and a power requirement corridor of an implementation p of an application. An RRE is also depicted whose task is to confine the observable predictability interval of p within the corridor specified by the latency and power requirements. Given the actual (current) input $i_{\text{act}} \in I$ and state $q_{\text{act}} \in Q$, the RRE in this case proactively estimates the expected latency L_{est} and power consumption P_{est} based on which it takes actions (outgoing arcs of the RRE) 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 [14].

9.4 Taxonomy of Run-Time Requirement Enforcers

According to [23], 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, a RRE can be classified as either a *loose* or a *strict* enforcement technique as follows:

Definition 9.3 (Loose/Strict RRE) A Run-Time Requirement Enforcer (RRE) of a requirement $r_o(p) = [LB_o, UB_o]$ of a program p is called 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.

Independent from the above definition, an RRE can be classified as a *centralized* or a *distributed* enforcement technique:

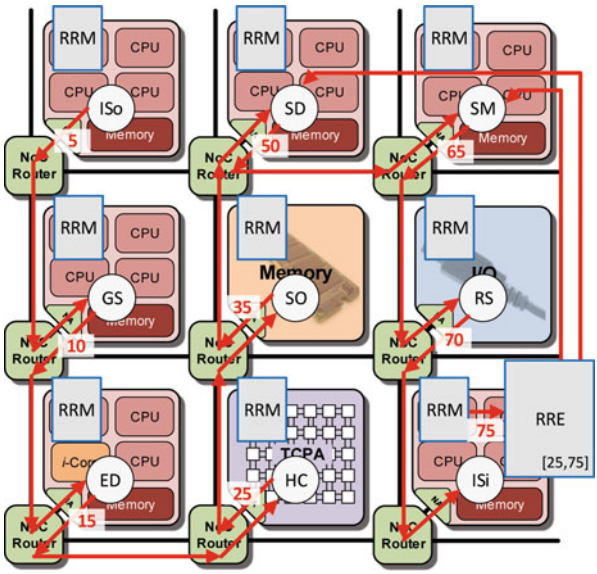

Fig. 9.4 Centralized RRE

Definition 9.4 (Centralized/Distributed RRE) A Run-Time Requirement Enforcer (RRE) of a requirement $r_o(p) = [LB_o, UB_o]$ of a program p is called centralized if a single enforcer instance is used to enforce the requirement. It is called distributed in case multiple enforcers jointly enforce the requirement.

Figure 9.4 illustrates an example of a centralized RRE of a latency requirement for an object detection streaming application from the area of robot vision illustrated in Fig. 9.6. Here, the execution time of the 9 tasks (actors) of the application is monitored by a local so-called *Run-Time Requirement Monitor (RRM)* instantiated on each of the invaded tiles. A centralized RRE instance is also instantiated which, in the example, receives the monitored timing information of the last actor in the chain, i.e., Image Sink (ISi), from the RRM on the respective tile based on which it conducts enforcement decisions. During the execution of the application, each RRM derives the time elapsed for the execution of its local actor(s) for the current image frame and creates a time stamp that is sent together with the processed frame to the subsequent actor. Thus, each actor in the chain is provided with the information about the time already elapsed for the processing of the frame by the previous actors based on which it determines the slack available for the remainder of the processing. Once the last actor in the chain has completed its processing, the local RRM computes and sends a completion time stamp to the centralized RRE. In case of a soft latency requirement, a loose RRE would react to any latency corridor violation by adjusting the voltage/frequency of the tiles that host the time-critical actors, i.e., SIFT Description (SD) and SIFT Matching (SM), with the goal to steer the latency of the chain back into the corridor for subsequent image frames.

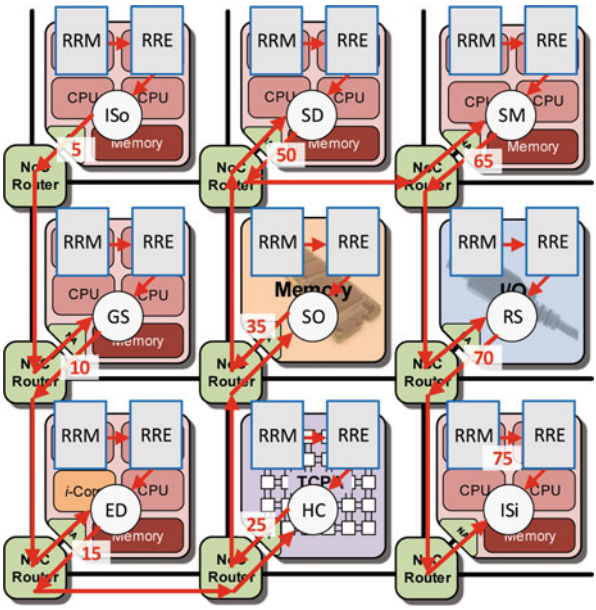

Fig. 9.5 Distributed RRE

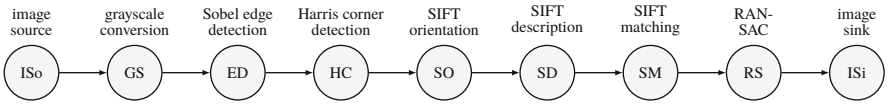

Fig. 9.6 Object detection streaming application

Figure 9.5 illustrates an example of a distributed RRE for the object detection application. Here, the overall time requirement per image frame could be partitioned into sub-corridors (or interval budgets) which are assigned to the invaded tiles. Also, in addition to the RRMs, a local RRE is instantiated per tile to enforce its assigned sub-corridor locally. Evidently, distributed RRE benefits from a simpler realization and scalability in comparison to centralized RRE. Nonetheless, centralized enforcement could better use global information to optimize secondary goals such as energy consumption, as we will show in Sect. 9.5.

9.4.1 Enforcement Automata (EA)

Although arbitrary algorithmic behavior can be envisioned for enforcement, in the following we focus on automata-based enforcement techniques, as they are simpler to generate and ideal for application of formal verification techniques for proof of correctness due to their strong formal semantics. Formal proofs are necessary

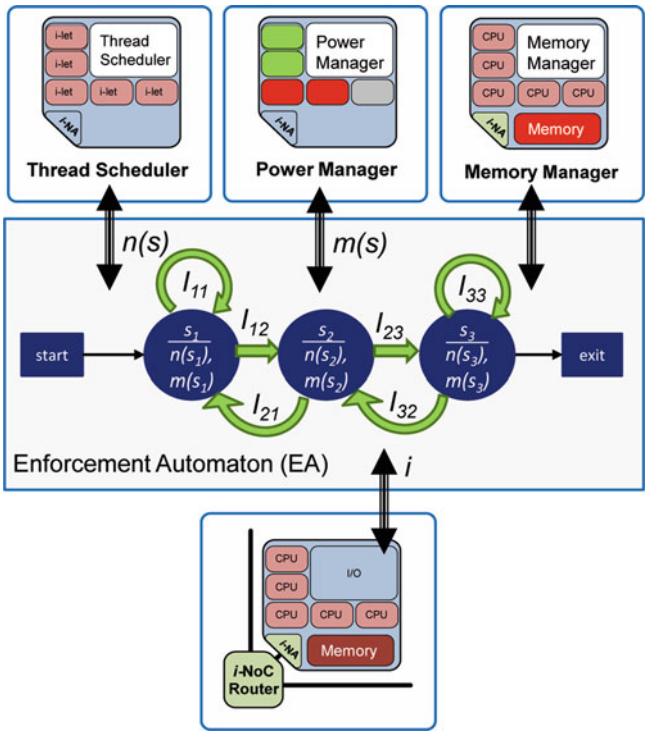

Fig. 9.7 Example of an Enforcement Automaton (EA). Depending on the input i of a program p and a current state s, the automaton takes a state transition to enforce a requirement. In the example, in state $s \in S$, the EA outputs how many cores $n(s)$ shall be powered on and in which power mode $m(s)$ (voltage, frequency) p shall be executed

particularly for enforcement of hard requirements. Figure 9.7 illustrates an example of an enforcement automaton (EA) of type Moore in which the input is a measure of the current workload i of a periodically executed program (segment or task) p, e.g., an image processing actor or kernel. In each state $s \in S$, the EA produces a vector of two outputs: the number $n(s)$ of cores to be powered on for executing the current job and the power mode $m(s)$ to be applied to the active core(s). As illustrated, the RRE acts as an interface between the application and the system software of the tile. Although in the examples provided in this paper, only the power management facility (voltage/frequency settings) and the degree of parallelism are controlled by RREs, they could in general control or restrict other system software components as well, e.g., the thread scheduler or the memory managing unit, for the enforcement of the given requirements.

9.4.2 i-lets and e-lets

Whereas for centralized enforcement, we assume that only one enforcer is instantiated per application program p, each task/actor of a distributedly mapped application program will be assigned its own local enforcer. In our implementation, an enforcer is implemented as a preferential thread called *e-let* in the following whereas application threads spawned for each task execution are called *i-lets*. Note that even if both are considered logically equivalent in terms of executable threads at the level of operating system, there is a notable difference between both: Even if i-lets present the application code for which requirements need to be guaranteed, they are usually not preferred by the operating system over other threads of their kind of the same application program. Whereas e-lets are always considered the preferred execution entities of the application program, they dominate the i-lets also included in this program. In addition, according to the principle of least privilege, e-lets have the capability to overrule or restrict the behavior of system-level software components including schedulers as well as cache, memory, and power managers in order to be able to enforce the properties required of their assigned i-lets. Another major difference between application i-lets and e-lets is the way they are executed. Whereas i-lets are created and executed upon each activation, e-lets are created only once at the time where an application program invades a tile of cores. They remain active not only for one iteration, but until the whole application retreats from all occupied resources. e-lets, in particular, state transitions in case the behavior is described by an EA, are triggered by incoming events, very similar to data-driven execution. In case of the following robot vision application, e.g., a state transition is triggered each time a new frame is arriving from a neighbor tile. In normal execution, the i-lets of an activated application task start after the EA has transited from the actual to the next state and have run-to-completion semantics. Whereas, e-lets may alternatively be triggered by asynchronous events, e.g., an exception from a temperature monitor.

Since an e-let is to be provided with special system privileges, including in particular the capability for immediate and low-latency response in bounded time to system events and operational state changes, it is implemented as a *kernel-level thread*. As a first-class object of the operating-system kernel, such a thread makes it possible to establish and maintain a semantic relationship between system-level and user-level code. The same applies to an i-let, but without granting the associated kernel-level thread any special privileges.

An ensemble of kernel-level threads with and without special capabilities for the control of system behavior depending on the particular requirements of an application program is managed by the kernel in the shape of a *squad*. A squad is a special unit within a *team* (a non-empty set of processes sharing a common address space and common computing resources [4]) of related kernel-level threads. This unit consists of two types of threads: on the one hand, those that make up the actual *lead* of the application program and on the other hand at least one *aide* who assists the lead threads as system mediator. From the kernel's point of view, the

aide has all the capabilities to assure the lead the required system behavior in a controlled manner. In addition to be able to override or modify certain operating-system decisions, the aide is able to instantly respond to system events. In such a setting, an e-let is mapped to an aide, while the i-lets for which certain properties are enforced appear as lead.

9.5 Case Study

In this section, we present examples of enforcement techniques for strict vs. loose as well as distributed vs. centralized enforcement of timing requirements for the case study of the previously introduced object detection application depicted in Fig. 9.6. The application consists of a chain of 9 tasks (actors) processing each input image in succession: an image source (ISo) actor to read in input images periodically at a constant rate, a gray-scale (GS) conversion actor, a Sobel edge detection (ED) actor and a Harris corner (HC) detection actor to determine, respectively, edges and corners in an image, a SIFT orientation (SO) actor to achieve invariance to image rotation, a SIFT description (SD) actor to extract the *features* in an image, a SIFT matching (SM) actor to detect objects in the image based on a previously trained set of object features, and a RANSAC (RS) actor to insert the detected objects into the image which is finally sent out by an image sink (ISi) actor. As platform, we consider a NoC-based 3×3 many-core architecture as depicted in Fig. 9.1 and map the application's actors on the architecture as illustrated in Figs. 9.4 and 9.5. All evaluations presented in this section are carried out using InvadeSIM [16, 18], a high-level functional simulation framework for multi-/many-core architectures and supporting resource-aware programming.

9.5.1 Enforcement Problem Description

In the following, we assume that each image frame of the given time-critical application must be processed within a latency upper bound $UB_L = 115\,\text{ms}$. Table 9.1 provides the average, standard deviation, and overall contribution of each actor's latency when processing a sequence of 9 149 images stemming from different sources of video streams when each actor is processed in isolation on a single core and running constantly at maximum frequency. As can be seen in Table 9.1, the SD and SM actors exhibit the highest degree of input-dependent variation in execution time and also the highest contribution to the overall latency. The remaining actors, on the other hand, do not exhibit a comparable execution time jitter and/or a comparable contribution to the overall application latency across the input space.

Table 9.1 Average, standard deviation, and overall contribution to the overall latency of each actor of the object detection application in Fig. 9.6 when processing a test sequence of 9 149 images and executed each in isolation on a single core running constantly at maximum frequency according to Table 9.2

Latency index	Actor						
	GS	ED	HC	SO	SD	SM	RS
Average [ms]	0,21	0,18	1,50	1,79	146,86	21,02	0,01
Std. deviation [ms]	0,09	0,08	0,64	0,80	106,15	15,04	0,03
Overall contribution	0,1 %	0,1 %	0,9 %	1,0 %	85,6 %	12,3 %	0,0 %

In the following, we present examples of RRE techniques using Dynamic Voltage and Frequency Scaling (DVFS) [3, 10, 21, 26] to enforce the global latency upper bound $UB_L = 115\,\text{ms}$ for the given application. Due to the small variation and overall latency contribution of all except the actors SD and SM according to Table 9.1, we dedicate a time budget of 20 ms to the other actors altogether, assuming that their cumulative latency per input image does not exceed this budget. This translates into a latency upper bound of $UB_L = 95\,\text{ms}$ for the SD and SM actors. For the demonstration of distributed RRE techniques, we further decide to split this bound into two individual latency upper bounds, namely $UB_L = 80\,\text{ms}$ for SD and $UB_L = 15\,\text{ms}$ for SM. Next, we present examples of loose vs. strict as well as distributed vs. centralized enforcement. As a merit of profit, we also investigate the potential energy savings of each RRE strategy in addition to evaluating its capability in enforcing the latency requirement(s).

According to Fig. 9.7, the following RRE techniques implemented as enforcement automata are privileged to adjust two control knobs prior to processing an image frame: (a) the degree of execution parallelism per actor that is adjusted by setting the number n of active cores that process the workload of each actor and (b) the power (voltage/frequency) mode m of the core(s) allocated for each actor adjusted through DVFS (for active cores) and power gating (for inactive cores). To this end, each RRE decides on a per input image basis how to distribute the workload of each actor being enforced between one and four cores available per tile according to the mappings shown in Figs. 9.4 and 9.5. At the same time, it sets the power mode of the cores of each tile to either a power-gated mode (with $f = 0$ and $V_{\text{DD}} = 0$) or 20 possible DVFS configurations (with a frequency step size of 0.2 GHz and a maximum frequency of 4 GHz) summarized in Table 9.2. For both actors under enforcement, SD and SM, we analyzed the major source of latency variation according to Table 9.1 (single core, constant maximal frequency) as stemming from the variability in the number i of *features* in each image to be processed. Therefore, this number is used as a direct indicator of the input workload to the following RRE strategies.

Table 9.2 Voltage/frequency (DVFS) modes of each core

mode m	$f(m)$ [GHz]	$V_{DD}(m)$ [V]	mode m	$f(m)$ [GHz]	$V_{DD}(m)$ [V]	mode m	$f(m)$ [GHz]	$V_{DD}(m)$ [V]	mode m	$f(m)$ [GHz]	$V_{DD}(m)$ [V]
1	0.2	0.5	6	1.2	0.91	11	2.2	1.26	16	3.2	1.58
2	0.4	0.6	7	1.4	0.98	12	2.4	1.32	17	3.4	1.65
3	0.6	0.69	8	1.6	1.05	13	2.6	1.39	18	3.6	1.71
4	0.8	0.77	9	1.8	1.12	14	2.8	1.45	19	3.8	1.78
5	1.0	0.84	10	2.0	1.19	15	3.0	1.52	20	4.0	1.84

9.5.2 Power, Latency, and Energy Model

Our investigation of enforcement strategies involves the evaluation of power consumption, execution latency, and energy demand per actor under enforcement. To evaluate the power consumption $P(m)$ of a core in power mode m, we use Eq. (9.2) in which the first summand represents the dynamic power contribution calculated based on the effective switching capacitance C_{eff} and the supply voltage $V_{DD}(m)$ and operating frequency $f(m)$ of the core in power mode m. The second summand describes the static power consumption calculated as the product of leakage current I_{leak} and supply voltage $V_{DD}(m)$.

$$P(m) = C_{\text{eff}} \cdot V_{DD}(m)^2 \cdot f(m) + I_{\text{leak}} \cdot V_{DD}(m) \tag{9.2}$$

For the construction of proper enforcement automata, we need to know the relation between the number i of input features and the execution latency L of each actor to be enforced 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). In the following, $L(1, 1, m_{\max})$ is determined by simulatively determining the execution latency of each actor per image for a representative set of 9 149 test images that fully covers the considered input space. Subsequently, the latency per feature of an actor is determined for each image by dividing its latency by the number of features i in that image. Figure 9.8 illustrates the distribution (left) and the cumulative distribution (right) of the per-feature latency for the SD actor. Based on the obtained distribution, we then determine $L(1, 1, m_{\max})$ according to the *strictness* of the latency requirement which specifies the minimum rate $s \in [0, 1]$ of requirement satisfaction that must be achieved, specified by the user. In case of (a) strict enforcement, a strictness of $s = 1$ is considered, and hence, the maximum observed per-feature latency among all images is used as $L(1, 1, m_{\max})$. For (b) loose enforcement, i.e., when $s < 1$, $L(1, 1, m_{\max})$ is set to the lowest per-feature latency among all images such that for $s \cdot 100\,\%$ of images, the latency per feature is lower than or equal to the selected $L(1, 1, m_{\max})$. In Fig. 9.8 (right), this calculation corresponds to finding the lowest x-coordinate with a cumulative density of s. Having $L(1, 1, m_{\max})$ determined, the following Eq. (9.3) is then used to determine the actor latency $L(i, n, m)$ based on

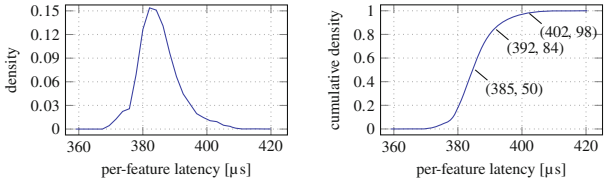

Fig. 9.8 Distribution (left) and cumulative distribution (right) of observed per-feature latency of the SD actor for a test sequence of 9 149 input images with number i of features to be processed varying between 0 and 5 513. To the right, the value of $L(1, 1, m_{\max})$ is marked for requirement strictness values of $s = 0.5$, 0.84, and 0.98

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. (9.3), $e(n)$ denotes the parallel efficiency in dependence of the number of cores n employed for the computation with $e(n) = 1$ in the best case. In our experiments, we consider $e(n) = 1$.

$$L(i, n, m) = L(1, 1, m_{\max}) \cdot \left\lceil \frac{i}{n \cdot e(n)} \right\rceil \cdot \frac{f(m_{\max})}{f(m)} \tag{9.3}$$

Note that Eq. (9.3) is a latency model specific to the SD and SM actors of our running application where $L(1, 1, m_{\max})$ must be determined individually for each actor to be enforced. Moreover, Eq. (9.3) could be alternatively replaced with an elaborate many-core timing analysis, e.g., those from [2, 5–7, 15, 25], to derive tight worst-case latencies that support a variety of different resource arbitration policies and resource sharing schemes. Based on the power consumption and latency models in Eqs. (9.2) and (9.3), 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 derived using Eq. (9.4).

$$E(i, n, m) = L(i, n, m) \cdot P(m) \cdot n \tag{9.4}$$

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. (9.5) which is derived from Eq. (9.3), considering $L(i, n, m) \leq UB_L$.

$$i_{\max}(UB_L, n, m) = \left\lfloor n \cdot e(n) \cdot \left\lfloor \frac{UB_L}{L(1, 1, m_{\max})} \cdot \frac{f(m)}{f(m_{\max})} \right\rfloor \right\rfloor \tag{9.5}$$

For example, with Eq. (9.5), we may compute $i_{\max}(80, 4, 20)$ for the SD actor which is the highest number i of features of an input image for which a latency upper bound of $UB_L = 80$ ms can be enforced with a strictness of $s \in [0, 1]$. For instance, for loose enforcement with $s = 0.5$, we obtain $i_{\max}(80, 4, 20) = 828$, and for

strict enforcement where $s = 1$, the maximum enforceable workload decreases to $i_{\max}(80, 4, 20) = 760$ features.

9.5.3 Energy-Minimized Timing Enforcement

According to Fig. 9.7, RRE may involve to set, modify, or impose restrictions on typically OS-related techniques such as thread scheduling or memory management. In the following examples, we exemplify enforcement strategies for latency enforcement of individual actor executions or complete applications by varying the number $n \in [1, 4]$ of cores (parallelism) and the power mode $m \in [1, 20]$ configuration for each actor execution. As, in general, multiple ways and settings for n and m might be feasible to enforce a requirement, the question becomes which requirement-adhering constellation the enforcer selects at run-time. Often, this freedom of choice may be exploited by optimizing one or more (secondary) objectives in addition to satisfying the given requirement. In the following, we consider energy demand as an objective to be minimized.[2] Given a latency requirement UB_L and the RRE decision space of $n \in [1, 4]$ and $m \in [1, 20]$, design space exploration can be conducted per actor (or a set of actors) to derive, e.g., in our running example for the SD actor, the maximum number $i_{\max}$ of features that can be processed under each choice of (n, m) while respecting the latency requirement. Taking the SD actor with a latency upper bound $UB_L = 80$ ms as an example, Fig. 9.9 illustrates the maximum workload $i_{\max}$ and the respective energy demand for each of the 80 possible (n, m) configurations

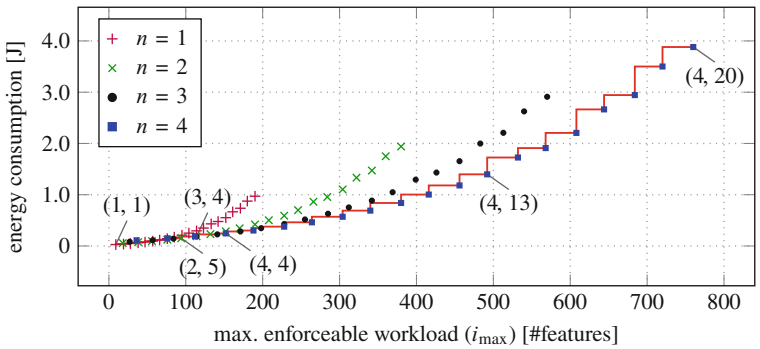

Fig. 9.9 Maximum enforceable workload $i_{\max}$ and energy demand of the SD actor under the variation of the number $n \in [1, 4]$ of active cores and their power mode $m \in [1, 20]$ for a hard ($s = 1$) latency bound of $UB_L = 80$ ms. Pareto-optimal (n, m) configurations are connected by a red line. For an exemplary subset of them, the Pareto-optimal configuration (n, m) is also annotated

[2] Other objectives for choice of settings could be to activate the least number n of cores for increasing aspects of long-term reliability.

derived using Eqs. (9.5) and (9.4), respectively, in case of strict enforcement ($s = 1$). The red line designates Pareto-optimal (n, m) configurations.

Based on such a design space exploration and the Pareto front of (n, m) configurations derived thereby, an energy-minimizing enforcement automaton may be systematically constructed in which prior to each execution of the SD actor, the enforcement automaton selects a state (Pareto-optimal (n, m) configuration) that is energy-minimal while satisfying the latency requirement in case input i is enforceable, thus if $i \leq i_{\max}(UB_L, n, m)$. For the example in Fig. 9.9, the enforcement automaton has 31 states, each corresponding to one of the 31 Pareto-optimal (n, m) configurations and the maximum enforceable workload $i_{\max}$ associated with that configuration. Here, the state selection is steered solely by the number i of features in the image to be processed by the SD actor.[3] For instance, for images with $i \leq 9$ features, $n = 1$ and $m = 1$ minimizes the energy demand of the SD actor without violating the given latency requirement $UB_L = 80$ ms. For input images with $142 \leq i \leq 152$ features, an energy-minimal and requirement-adhering execution can be realized only if $n = 4$ cores are used for SD in parallel and power mode $m = 4$. Finally, a strict enforcement becomes impossible if $i > 760$, even using the configuration with the highest compute power, i.e., $n = 4$ and $m = 20$. For non-enforceable inputs, the enforcer needs to either throw an exception, stop processing (drop) the image, or process only as much as the latency bound allows to be processed. In Sect. 9.5.6, we propose a number of exception handling techniques under the topic of range extension. Before that, we first present techniques for distributed enforcement where each actor is individually enforced. Subsequently, we present also an example of centralized enforcement in which a more global view of the system state can be obtained by a centralized RRE instance that can take decisions affecting multiple actors and resources.

9.5.4 Distributed Enforcement

Figure 9.10 shows the resulting automatically generated energy-minimizing enforcement automata for a distributed enforcement strategy of the two individual actors SD and SM with latency upper bounds 80 ms and 15 ms, respectively. The EAs for selecting the energy-minimizing (n, m) configurations obtained through the previously presented design space exploration are implemented as lightweight lookup tables for each actor. At run-time, once an image is ready to be processed, the number i of features in it becomes known. Prior to processing an image, the RRE (e-let) retrieves the energy-minimizing (n, m) configuration corresponding to

[3] Note that in this example, the RRE could also be represented by a function table rather than an FSM, as the selection of state is only dependent on the input. More general cases such as restricting the allowed settings in each state to allow only step-wise increase or decrease of DVFS modes can be constructed.

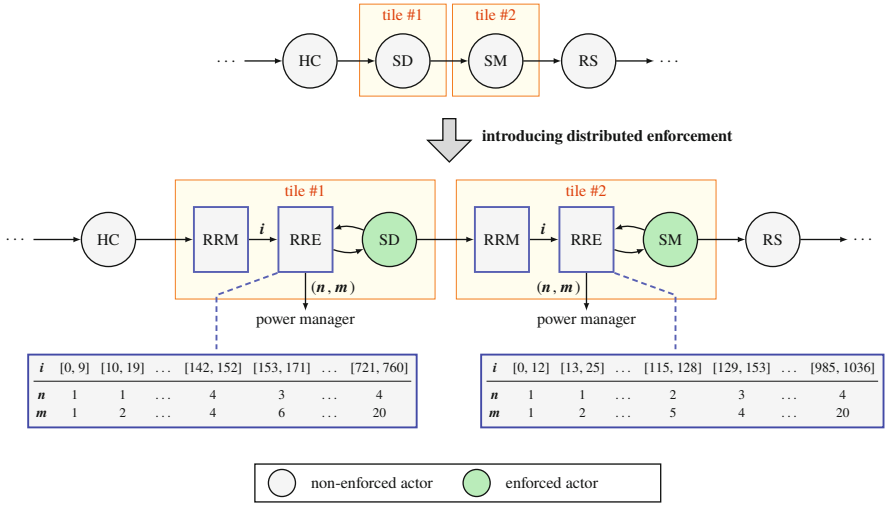

Fig. 9.10 Implementation of distributed RRE using pre-explored energy-optimal parallelism degree and DVFS settings (n, m) for SD and SM actors with hard ($s = 1$) latency upper bounds of $UB_L = 80$ ms and $UB_L = 15$ ms, respectively

i features from the table and instructs the power manager to use these settings. As shown in Fig. 9.10, the integration of enforcers may be achieved at the level of actor graphs as a model transformation by inserting the enforcer as an actor in front of each actor to be enforced, such that for each image to be processed, the energy-minimizing (n, m) configuration is set prior to execution of the image, and the configuration stays constant over the duration of processing this image. Employing the above enforcement strategy, the run-time manager is not compelled to run the enforced actors constantly with the maximum number $n = 4$ of cores and in the highest power mode $m = 20$ to guarantee the satisfaction of latency constraints in the presence of input variations, unless $i \geq 721$ for SD or $i \geq 985$ for SM. Also note that the given latency bounds cannot be strictly enforced for a feature count $i > 760$ for SD and $i > 1\,036$ for SM. Thus, the maximum workload that can be strictly enforced by both actors is limited to $i = 760$ features.

The histograms of observable latencies of the SD and SM actors (a) without enforcement ($n = 4$ and $m = 20$) and (b) with enforcement considering hard ($s = 1$) latency upper bounds of $UB_L = 80$ ms and 15 ms for SD and SM, respectively, are illustrated in Fig. 9.11. As shown in the plots, the RREs choose a power mode that maximizes energy savings while satisfying the given latency upper bound of each actor under enforcement. For a variety of requirement strictness levels, Table 9.3 finally presents the average dynamic energy consumption and the achieved dynamic energy savings of the SD and SM actors compared to the non-enforced scenario with $n = 4$ and $m = 20$. As can be seen, in case of *loose enforcement*, i.e., a strictness $s < 1$, the RRE achieves between 38.3 % and 41.2 % dynamic energy savings per enforced actor (respectively, between 39.3 % and 40.8 % collectively

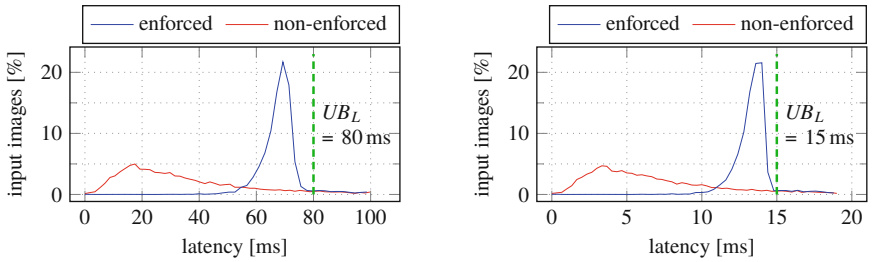

Fig. 9.11 Latency distribution for the SD (left) and SM (right) actors. The enforced case corresponds to the energy-minimized enforcement for hard ($s = 1$) latency bounds of $UB_L = 80$ ms for SD and $UB_L = 15$ ms for SM. The non-enforced case corresponds to a fixed setting of $n = 4$ and $m = 20$ per actor

for the two actors) while satisfying latency upper bounds of 80 ms and 15 ms for the SD and SM actors, respectively. In case of *strict enforcement* which corresponds to a requirement satisfaction rate of $s = 1$, the RRE still is able to achieve dynamic energy savings of 37.6% for SD and 37.2% for SM (respectively, 37.6% collectively for the two actors) while guaranteeing that the given latency upper bound for each actor will never be violated. Evidently, this guarantee holds only for enforceable input images, i.e., those with $i \leq 760$ features for the SD actor and $i \leq 1\,036$ for SM (see the RRE tables in Fig. 9.10). In Sect. 9.5.6, we discuss approaches that can be employed to enable the enforcement of latency requirements for inputs which are not enforceable merely using the given RRE control knobs. Finally, when analyzing the overall energy consumption of all actors per input frame, we obtain an overall dynamic energy reduction of 33.8 % in case of strict enforcement ($s = 1$) and between 35.4 % and 36.8 % in case of loose enforcement ($s < 1$) for the whole application, even though only two out of 9 actors are enforced. Noteworthy, the additional execution time and energy consumption of the RREs themselves can be neglected as these are implemented by simple table lookups.

9.5.5 *Centralized Enforcement*

In this section, we consider the combined enforcement of the SD and SM actors using centralized enforcement. As depicted in Fig. 9.12, a single instance of an RRE is now enforcing the overall hard ($s = 1$) latency upper bound of $UB_L = 80 + 15 = 95$ ms for both SD and SM actors collectively. Similar to the distributed case, the energy-minimizing (n, m) configurations which are required for the construction of the RRE are obtained through a previously presented design space exploration, but now considering a unified latency upper bound $UB_L = 95$ ms for the execution of both SD and SM actors. Note that considering a compound

Table 9.3 Average dynamic energy consumption and savings per image for the SD and SM actors through distributed enforcement in dependence of requirement strictness defined as the minimum acceptable requirement satisfaction rate

Requirement strictness		SD actor ($UB_L = 80$ ms)			SM actor ($UB_L = 15$ ms)			Energy savings	
norm. dist. index	min. sat. rate (s)	$L(1, 1, m_{\max})$ [μs]	Avg. energy [mJ]	Energy savings	$L(1, 1, m_{\max})$ [μs]	Avg. energy [mJ]	Energy savings	SD+SM	Overall
Median	50 %	385,4	131,1	41,2 %	55,4	28,9	38,9 %	40,8 %	36,8 %
avg+1·σ	84,1 %	392,0	132,9	40,4 %	55,6	29,0	38,7 %	40,1 %	36,1 %
avg+2·σ	97,7 %	402,3	135,0	39,5 %	55,9	29,1	38,3 %	39,3 %	35,4 %
Maximum	100 %	420,7	139,2	37,6 %	57,8	29,6	37,2 %	37,6 %	33,8 %
Without enforcement		–	223,2	(ref)	–	47,2	(ref)	(ref)	(ref)

The energy consumption without enforcement ($n = 4$, $m = 20$) serves as a baseline

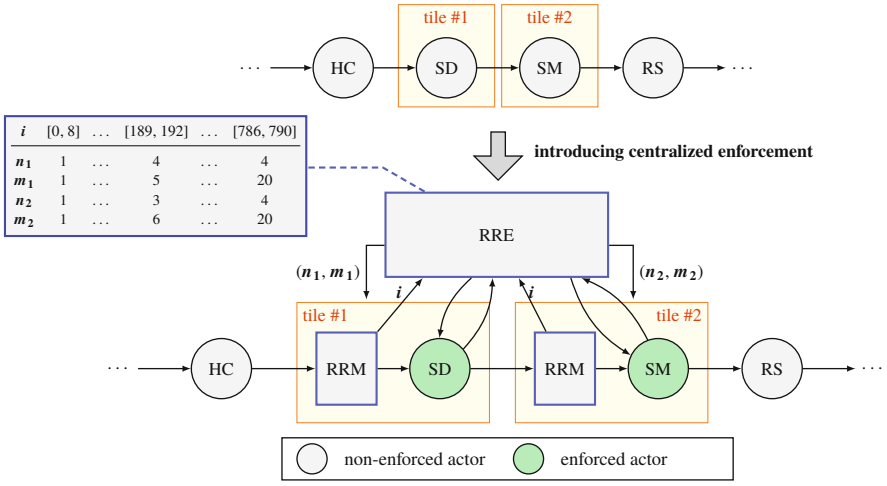

Fig. 9.12 Implementation of a centralized RRE using pre-explored energy-optimal parallelism degree and DVFS settings (n, m) for SD and SM actors with a hard $(s = 1)$ latency upper bound of $UB_L = 95$ ms for both actors collectively

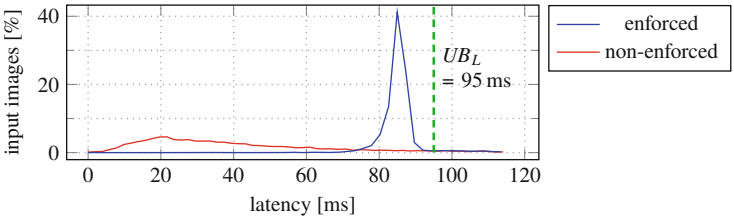

Fig. 9.13 Latency distribution of the SD and SM actors when collectively enforced for a hard $(s = 1)$ compound latency bound of $UB_L = 95$ ms. The enforced scenario is realized using energy-minimized enforcement, and the non-enforced scenario corresponds to a fixed configuration of $n = 4$ and $m = 20$ per actor

latency bound for both actors enables enforcing this bound for images with up to $i = 790$ features.

The histogram of observable collective latency of the SD and SM actors (a) without enforcement ($n = 4$ and $m = 20$ for both actors) and (b) with enforcement considering a hard latency upper bound, i.e., for a requirement strictness of $s = 1$, is illustrated in Fig. 9.13. As shown in the plots, the RRE assigns the number n of active cores and their power mode m for each actor under enforcement to maximize energy savings while satisfying the given latency upper bound of $UB_L = 95$ ms collective for both actors. For a variety of requirement strictness levels, Table 9.4 finally presents the average dynamic energy consumption and the achieved dynamic energy savings of the two enforced actors using centralized enforcement compared to the non-enforced scenario with $n = 4$ and $m = 20$. As can be seen, the RRE achieves in case of *loose enforcement*, i.e., strictness $s < 1$, between 39.7 %

Table 9.4 Average dynamic energy consumption and savings per image for the SD and SM actors through centralized enforcement for a compound latency bound of $UB_L = 95\,\text{ms}$ in dependence of requirement strictness defined as the minimum acceptable requirement satisfaction rate

Requirement strictness		SD actor			SM actor			Energy savings	
norm. dist. index	min. sat. rate (s)	$L(1,1,m_{\max})$ [μs]	Avg. energy [mJ]	Energy savings	$L(1,1,m_{\max})$ [μs]	Avg. energy [mJ]	Energy savings	SD+SM	Overall
median	50 %	385,4	129,9	41,8 %	55,4	26,7	43,6 %	42,1 %	37,9 %
avg+1·σ	84,1 %	392,0	131,3	41,2 %	55,6	27,0	42,9 %	41,5 %	37,3 %
avg+2·σ	97,7 %	402,3	135,3	39,4 %	55,9	27,9	40,9 %	39,7 %	35,7 %
maximum	100 %	420,7	137,2	38,5 %	57,8	28,2	40,2 %	38,8 %	35,0 %
Without enforcement		–	223,2	(ref)	–	47,2	(ref)	(ref)	(ref)

The energy consumption without enforcement ($n = 4$, $m = 20$) serves as a baseline

and 42.1 % dynamic energy savings collectively for the two enforced actors while satisfying a compound latency upper bound of 95 ms. In case of *strict enforcement* ($s = 1$), the RRE still is able to achieve dynamic energy savings of 38.8% collectively for the two actors while guaranteeing that the given latency upper bound $UB_L = 95$ ms will never be violated. Finally, when analyzing the overall energy consumption of all actors per input frame, we obtain a dynamic energy reduction of 35 % in case of strict enforcement ($s = 1$) and between 35.7 % and 37.9 % in case of loose enforcement ($s < 1$), even though only two out of 9 actors are enforced. In summary, compared to distributed enforcement, the centralized scheme is able to even save slightly more dynamic energy while enforcing a higher workload.

9.5.6 *Lower Latency Bound Enforcement and Range Extenders*

In certain cases, a latency requirement may introduce—in addition to an upper bound UB_L—also a lower bound, LB_L, thus, demanding the enforcement of a latency corridor. Such a lower latency bound could be enforced by means of, e.g., a simple timer (counter) that measures the time elapsed from the beginning of the current execution of the actor(s) under enforcement. The transmission of the produced result(s) to the next actor(s) could then be simply delayed to the time the timer indicates that the time interval of LB_L has passed, see Fig. 9.14. More

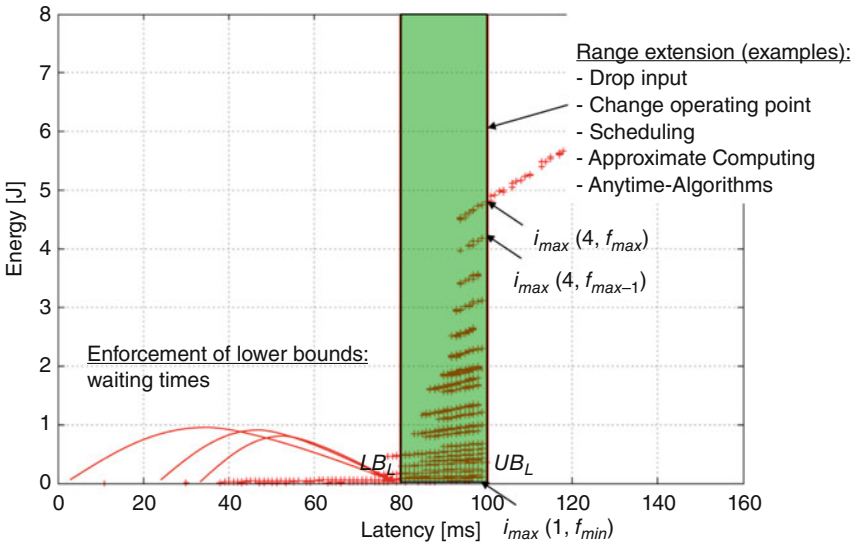

Fig. 9.14 Examples of range extenders and enforcement of lower latency bounds LB_L and thus latency corridors

difficult and also diverse in the space of possible solutions, however, is the question of how to deal with non-enforceable inputs. In case of our running distributed object detection application, our test image sequences on purpose contained images with more features i than for which the given latency upper bound can be enforced with only $n = 4$ cores in highest power mode $m = 20$. In case of strict enforcement ($s = 1$) corresponding to hard real-time requirements, not even a single violation of a latency upper bound is tolerable. Hence, there must be techniques to avoid such violations per construction, if a non-enforceable input is observed. This is a matter of current research. We therefore briefly outline a few techniques how to deal with these cases: 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 (change of operating point [14]), see also Fig. 9.14.

9.6 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. It has been shown by introductory examples on latency enforcement of a distributed object detection application that enforcers may be generated through profiling and the creation of high priority system-level threads called *e-lets* that are formally described in behavior by an enforcement automaton each. These e-lets proactively control the system resources claimed by an application program in view of observed workload variation. First, based on the assignment of exclusive resources to periodic workload such as streaming applications, composability is created that is necessary to allow for a static and independent analysis of each application running on a given MPSoC platform. This enables us to statically analyze non-functional properties of applications or parts thereof and define RRE techniques to control requirements dynamically. For a distributed object detection application as an example, it has been shown that the variability of non-functional execution properties can be greatly reduced in dependence of the level of strictness that shall be fulfilled for each requirement. Moreover, it has been shown that RRE techniques can be either implemented in a centralized or distributed manner. In the future, we want to look at how

to decompose requirement corridors for distributed enforcement and study the control overheads of centralized enforcement. Finally, techniques for simultaneous enforcement of multiple non-functional requirements need to be investigated, as here, not only the input (workload) variation as considered in this seminal paper, but also the shared system state must be taken into account once multiple RREs are at work.

Acknowledgments This paper is dedicated to Peter Marwedel on behalf of his 70th birthday in recognition of his lifetime achievements in the area of design automation for embedded systems. Special thanks also to Zhai Ming for several experiments conducted for this paper. Finally, we would like to acknowledge the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation)—Project Number 146371743—TRR 89 Invasive Computing that funded our work.

References

1. B. Akesson, et al., Composability and predictability for independent application development, verification, and execution, in *Multiprocessor System-on-Chip* (Springer, Berlin, 2011), pp. 25–56
2. S. Altmeyer, et al., A generic and compositional framework for multicore response time analysis, in *Proceeding of RTNS* (ACM, New York, 2015), pp. 129–138
3. D. Angioletti, et al., A runtime resource management policy for OpenCL workloads on heterogeneous multicores, in *Proceeding of DATE* (IEEE/ACM, New York, 2019), pp. 1385–1390
4. D.R. Cheriton, et al., Thoth, a portable real-time operating system. Commun ACM **22**(2), 105–115 (1979)
5. R.I. Davis, et al., An extensible framework for multicore response time analysis. Real-Time Syst. **54**(3), 1–55 (2017)
6. G. Giannopoulou, et al., Timed model checking with abstractions: towards worst-case response time analysis in resource-sharing manycore systems, in *Proceedings of the International Conference Embedded Software* (ACM, New York, 2012), pp. 63–72
7. G. Giannopoulou, et al., Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources. Real-Time Syst. **52**(4), 399–449 (2016)
8. A. Hansson, et al., CoMPSoC: a template for composable and predictable multi-processor system on chips. ACM TODAES **14**(1), 2 (2009)
9. C. Imes, et al., POET: a portable approach to minimizing energy under soft real-time constraints, in *Proceeding of RTAS* (IEEE, Silver Spring, 2015), pp. 75–86
10. A. Kanduri, et al., Approximation-aware coordinated power/performance management for heterogeneous multi-cores, in *Proceeding of DAC* (IEEE/ACM, New York, 2018), pp. 1–6
11. P.N. Khanh, et al., Incorporating energy and throughput awareness in design space exploration and run-time mapping for heterogeneous MPSoCs, in *Proceeding of DSD* (IEEE, Silver Spring, 2013), pp. 513–521
12. H. Kopetz, *Real-time Systems: Design Principles for Distributed Embedded Applications*, 2 edn. (Springer, Berlin, 2011)
13. S. Pinisetty, et al., Runtime enforcement of reactive systems using synchronous enforcers, in *Proceeding of ACM SIGSOFT International SPIN Symposium Model Checking of Software* (2017), pp. 80–89
14. B. Pourmohseni, et al., Hard real-time application mapping reconfiguration for NoC-based many-core systems. Real-Time Syst. **55**(2), 1–37 (2019)
15. B. Pourmohseni, et al., Isolation-aware timing analysis and design space exploration for predictable and composable many-core systems, in *Proceeding of ECRTS* (2019)

16. S. Roloff, et al., Execution-driven parallel simulation of PGAS applications on heterogeneous tiled architectures, in *Proceeding of DAC* (IEEE/ACM, New York, 2015), pp. 1–6
17. S. Roloff, et al., ActorX10: an actor library for X10, in *Proceeding of ACM SIGPLAN Workshop on X10* (ACM, New York, 2016), pp. 24–29
18. S. Roloff, et al., *Modeling and Simulation of Invasive Applications and Architectures* (Springer, Berlin, 2019)
19. T. Schwarzer, et al., Symmetry-eliminating design space exploration for hybrid application mapping on many-core architectures. IEEE TCAD **37**(2), 297–310 (2018)
20. A.K. Singh, et al., Accelerating throughput-aware runtime mapping for heterogeneous MPSoCs. ACM TODAES **18**(1), 9:1–9:29 (2013)
21. A.K. Singh, et al., Energy optimization by exploiting execution slacks in streaming applications on multiprocessor systems, in *Proceeding of DAC* (IEEE/ACM, New York, 2013), p. 115
22. J. Teich, et al., *Invasive Computing: An Overview* (Springer, New York, 2011)
23. J. Teich, et al., Language and compilation of parallel programs for *-predictable MPSoC execution using invasive computing, in *Proceeding of MCSOC* (IEEE, Silver Spring, 2016)
24. A. Weichslgartner, et al., DAARM: design-time application analysis and run-time mapping for predictable execution in many-core systems, in *Proceeding of CODES+ISSS* (IEEE/ACM, New York, 2014), pp. 1–10
25. A. Weichslgartner, et al., *Invasive Computing for Mapping Parallel Programs to Many-Core Architectures* (Springer, Berlin, 2018)
26. Z. Zhu, et al., Energy minimization for multi-core platforms through DVFS and VR phase scaling with comprehensive convex model, in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2019)