Chapter 6
HARPA RT

Dimitrios Rodopoulos, Nikolaos Zompakis, Michail Noltsis, Francky Catthoor, and Dimitrios Soudris

6.1 HARPA RT

6.1.1 Primitives

As technology nodes approach deca-nanometer dimensions, a variety of reliability violations threaten the binary correctness of processor execution. Computer architects typically enhance their designs with mechanisms that correct such errors, possibly at the cost of extra clock cycles. However, this clock cycle overhead increases the performance vulnerability factor (PVF) of the processor. The goal is to minimize the aforementioned cost using dynamic voltage and frequency scaling (DVFS), which is typically featured in the state-of-the-art processors. Towards this direction, the concepts of deadline vulnerability factor (DVF), cycle noise, and slack are introduced. A closed-loop implementation is proposed, which configures the clock frequency based on the observed slack value. Thus, the temporal overhead of error mitigation is absorbed and performance dependability is improved. We evaluate the transient and steady-state behavior of our approach, proving its responsiveness within less than 1 ms for a wide variety of error rates.

Before presenting the details of the HARPA RT, we will establish a common theoretical background, introducing all key terms. Then, the two components of the HARPA RT with System Scenarios are presented: (1) a PID Controller, which ensures stable control of the performance dependability scheme and (2) a System

D. Rodopoulos (✉) · F. Catthoor
Imec, Leuven, Belgium
e-mail: drodo@microlab.ntua.gr; catthoor@imec.be

N. Zompakis · M. Noltsis · D. Soudris
MicroLab-ECE-NTUA, Athens, Greece
e-mail: nzompakis@microlab.ntua.gr; mnoltsis@microlab.ntua.gr; dsoudris@microlab.ntua.gr

© Springer International Publishing AG, part of Springer Nature 2019

W. Fornaciari, D. Soudris (eds.), *Harnessing Performance Variability in Embedded and High-performance Many/Multi-core Platforms*,
https://doi.org/10.1007/978-3-319-91962-1_6

Scenario component, which detects the applicable performance dependability target at all times. The latter is very important, since it indirectly provides information to the HARPA RT about the extent of performance variability that needs to be mitigated.

Error mitigation inflates the number of clock cycles that is required to complete portions of the instruction stream (corresponding to TNs), which pass through the processor. In order to guarantee dependable performance, this overhead should be quantified and absorbed. We formulate this problem using the following definitions. From prior art, we repeat that the concept of the PVF is the per unit increase in the number of clock cycles required for a specific instruction stream, as a result of temporal error mitigation overheads [2].

Definition 1 Cycle budget (N) is the number of clock cycles (cc) of relevant computation assigned to each TN. This can be determined at design time or at runtime according to the following equation, based on a specification for the cycles per instruction (CPI) of the processor, the number of instructions (L) in a TN, and a user-defined PVF tolerance ($\text{PVF}_{\text{limit}}$) :

$$N = (1 + \text{PVF}_{\text{limit}}) \times \text{CPI} \times L \tag{6.1}$$

Definition 2 Cycle noise (x) is the sum of clock cycles (cc) that are inevitably wasted by mechanisms like ECC, shadow latches, etc. In general, it comes across as a result of error mitigation, which consequently leads to PVF degradation.

Definition 3 Frequency multiplier (x) is a positive real number by which the default clock frequency (f) is multiplied, reflecting possible DVFS configurations.

Definition 4 Deadline vulnerability factor (DVF) is the per unit difference between the real execution time (T_{real} with an m_{real} frequency multiplier), including cycle noise, and a reference execution time (T_{ref} with a frequency multiplier m_{ref}), where no cycle noise occurs:

$$\text{DVF} = 1 - \frac{T_{\text{ref}}}{T_{\text{real}}} = 1 - \frac{\frac{N}{m_{\text{ref}} \times f}}{\frac{N+x}{m_{\text{real}} \times f}} = 1 - \frac{m_{\text{real}} N}{m_{\text{ref}}(N + x)} \tag{6.2}$$

Let us assume that a stream of TNs is executed by a processor. For each TN, a frequency multiplier can be selected ($m[n]$). A cycle noise value ($x[n]$) is also associated with each TN. A recursive formulation of DVF at the end of each TN is possible. An expression is derivable from the DVF definition, by expressing $T_{\text{real}}[n]$ using $T_{\text{real}}[n-1]$. In that sense, we can write:

$$\text{DVF} = 1 - \frac{T_{\text{ref}}[n]}{T_{\text{real}}[n]} = 1 - \frac{T_{\text{ref}}[n]}{T_{\text{real}}[n-1] + \frac{N[n]+x[n]}{m[n]f}} = 1 - \frac{T_{\text{ref}}[n]}{\frac{T\text{ref}[n-1]}{1-\text{DVF}[n-1]} + \frac{N[n]+x[n]}{m[n]f}} \tag{6.3}$$

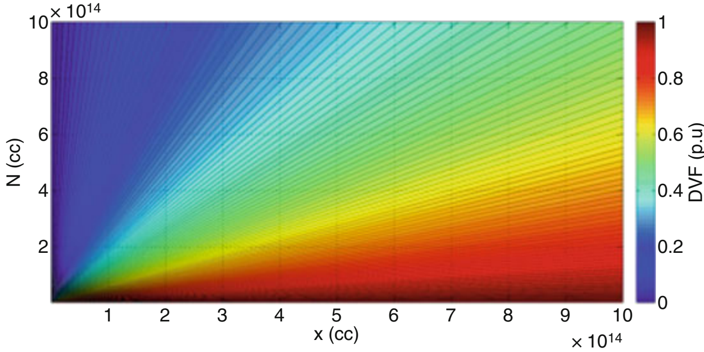

Fig. 6.1 Deadline vulnerability factor (DVF) converges to zero, by definition, when $x << N$

Understandably, this formulation is complicated and has increased computational requirements to derive (many math operations). Even if compatible to the state-of-the-art formulation (e.g., PVF), its complexity is prohibitive for a run-time engine that is supposed to configure a platform and guarantee performance dependability within roughly 1 ms. As a result, we need a recursive metric that requires few math operations, thus enabling hardware integration of a DVF degradation monitor.

It is also very interesting to observe the long-term trends of DVF according to the above definition, assuming that no DVFS-like countermeasure has yet been engaged. In Fig. 6.1, we present various values of DVF for a sweep of N and x and identify certain regions of interest:

- When $x >> N$, DVF converges to 1, reflecting processing entirely dominated by cycle noise.
- Another region is where $x \approx N$, where we observe intermediate values for DVF.
- The upper-left side of Fig. 6.1 corresponds to $x << N$ and reflects realistic cases, since cycle noise is not expected to exceed cycle budget. We see that as N increases (e.g., when useful processing follows a cycle noise burst), DVF converges to zero. It is to be expected that, as long as no cycle noise occurs, DVF will asymptotically converge to 0. Thus, deadline vulnerability is restricted to a significant portion of the execution, right after cycle noise has manifested itself.

In Fig. 6.2, we see a pseudo-transient evolution of DVF, assuming impulse cycle noise occurrences. Again, we observe that DVF asymptotically converges to zero, immediately after an impulse of cycle noise has been injected. This makes DVF very suboptimal for on-the-fly performance variability depiction, since it always requires calculations from the initial point of execution. The situation is made even worse, given the math operation requirements of DVF. Hence, a new term is needed to provide at all times the proximity of the system to a deadline violation.

Definition 5 Rate Divergence Slack (s) expresses the divergence of the processor from a default clock budget that is assigned to a TN ($N[n]$) assuming the default

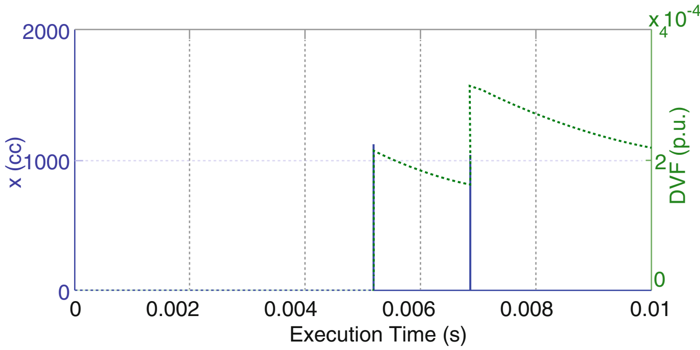

Fig. 6.2 Transient view of DVF, for impulse injections of cycle noise

Table 6.1 Operations required for performance variability indicators

	Divisions	Multiplications	Additions	Subtractions
Deadline vulnerability factor	3	1	2	2
Rate divergence slack	0	0	1	2

(m_{ref}) and current (m_{real}) frequency multipliers. It is given, by the following Equation, whereas for the sake of brevity, we will refer to this term simply as slack for the rest of the current document:

$$s[n] = \frac{N[n]}{m_{\text{ref}}} - \frac{N[n] + x[n]}{m_{\text{ref}}} + s[n-1] \tag{6.4}$$

It appears that, by definition (see Table 6.1), slack is a simpler metric when compared to DVF, in terms of required math operations. Considering the slack evaluation, we have reduced the cost significantly because the variable multiplications and divisions are simplified and only a few shifts and add/subs are left. If we are looking at a HW or firmware implementation of the HARPA RT, it is very important to consider such overheads.

It is also important to show that *iff slack is equal to zero, then DVF is also equal to zero* [9]. For the sake of brevity, we do not elaborate on the proof of the above material equivalence. However, given the above recursive definitions, the proof can be easily derived using mathematical induction and assuming that $s[0] = \text{DVF}[0] = 0$, which can be expected at the beginning of a TN series execution. Having provided an illustration of the proof, we can conclude that aiming at $s[n] = 0$ is equivalent with aiming at $\text{DVF}[n] = 0$, which is that ultimate goal of mitigating performance variability. That way, it is consistent to design the HARPA RT in such a way that processor slack is forced to converge to zero.

6.1.2 Performance Variability Model

In this subsection, we will illustrate how performance variability is perceived in the context of the HARPA RT. Cycle noise is the key term for the defined model, given that it quantifies the intervention of error correcting mechanisms with the processor performance. We define the rate at which cycle noise occurs and its peak amplitude. If a detected and corrected error [4] occurs, the correction mechanisms add cycle noise (x) to the TN cycle budget (N). Hence, the rate of cycle noise is described by an MTTF estimation [4]. The probability of cycle noise occurrence after wall-clock time is given by the following equation, which is equivalent to the (MTTF,1) distribution [8]:

$$P = 1 - \exp\left(\frac{-\Delta T}{\text{MTTF}}\right) \tag{6.5}$$

In a simulation environment, a random number r can be created and compared to P. If we assume that cycle noise is injected at that point of the execution and that it is equal to, as shown in Fig. 6.3. The parameters μ and σ depend on the PVF impact of error correction and reflect the amplitude of cycle noise. As a result, in a simulation environment, we can assume the injection of cycle noise bursts, which generally follow the above model. Through the rate and amplitude parameters, one can adjust the aggressiveness of performance variability. Also, such a model is linkable to reliability models, if one assumes the error rates of architectural components, as well as the error correction coverage that is applied to the processor.

The aforementioned model definition further defines the interface of the HARPA RT with the greater HARPA workpackage on performance variability modeling. In that sense, the outcome should be compatible to the above formulation. That way, in order to verify the HARPA RT in a prototype or a simulation environment (regardless of the underlying scenario methodology), the injected cycle noise should

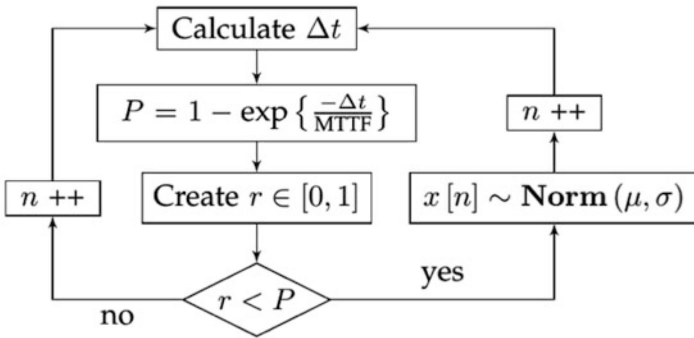

Fig. 6.3 Modeling of cycle noise during TN execution

come about as a result of processor variability that is corrected by specific circuitry, which in turn adds temporal overheads (causing the PVF to increase).

6.1.3 HARPA RT System Scenarios

The HARPA RT utilizes a PID controller to mitigate the performance degradation effects due to RAS interventions. The PID controller operates on cycle budgets that a scenario detector provides. Thus, the definition of the system scenarios is critical for the optimal responses of the HARPA RT. Figure 6.4 depicts the functionality of the proposed PID controller. Table 6.2 explains the variables of the proposed instantiation.

The proposed control scheme (for the case of constant cycle budget) is shown in Fig. 6.4. The idea is to create an error signal ($e[n]$) that indicates the currently experienced slack and adjust the frequency of the processor accordingly. This error signal is amplified in traditional discrete-time PID fashion, by adjusting the respective gains, according to the following equation:

$$\hat{m}[n] = \underbrace{e[n]k_p}_{\text{proportional component}} + \underbrace{\hat{m}_i[n-1] + e[n]k_i}_{\text{integral component}} + \underbrace{(e[n] - e[n-1])k_d}_{\text{differential component}} \quad (6.6)$$

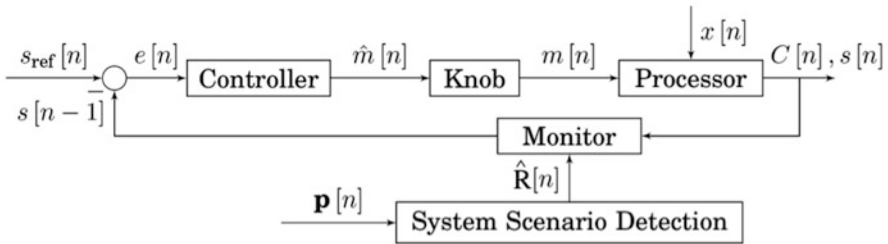

Fig. 6.4 Block diagram of the proposed performance dependability scheme

Table 6.2 Variables of the proposed HARPA-RT instantiation

$s_{\text{ref}}[n]$	Target slack specification (0 unless else specified)	$x[n]$	Timing noise interfering with RTS
$s[n-1]$	Slack at the end of the previous RTS	$C[n]$	Actual budget of current RTS
$e[n]$	Error signal $s_{\text{ref}}[n-1] - s[n-1]$	$s[n]$	Slack at the end of the current RTS
$\hat{m}[n]$	Proposed frequency multiplier	$\hat{R}[n]$	Default timing budget for current RTS
$m[n]$	Applied frequency multiplier	$p[n]$	System RTS parameters

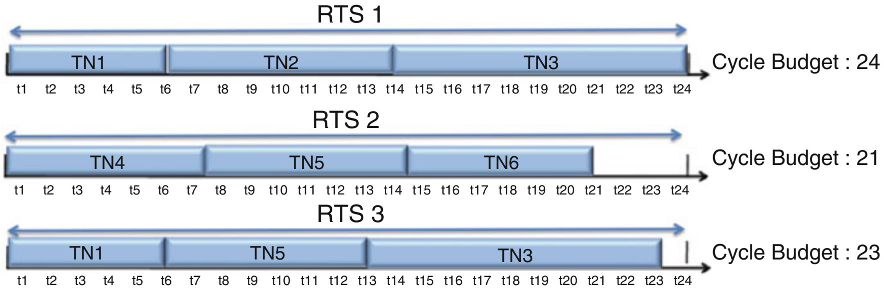

Fig. 6.5 Thread nodes sequences into run-time situations (RTSs)

The PID controller reacts at each RTS consisting of thread nodes (TNs). The entire program code is divided up into TNs. Each TN is a sequence of instructions with deterministic behavior (e.g., the body of a loop) that is executed in an atomic way. Run-time situation (RTS) is a term that is assigned to a TN or a sequence of TNs (see Fig. 6.5), in terms of the TNs timing (latency) or energy budget. Resolution of this label assignment is based on parameters of the TN, which are called RTS parameters. Due to the existence of multiple TNs in applications that are executed with several combinations (see Fig. 6.5), the RTS with multiple TNs allows the PID controller to respond to a more coarse granularity.

For dependable performance to be realized, each RTS needs to respect a timing deadline as indicted by its default cycle budget and the applicable reference frequency (m_{ref}). Due to the variety of RTSs that appear in real applications, it is rather suboptimal to store their respective cycle budgets, let alone detect them at runtime. In other words, the number of RTSs that are encountered in the instruction stream of a processor is quite complex to handle. Apart from storing the cycle budgets per RTS, it is additionally complex to detect the RTSs. The system scenario approach [1] overcomes this issue by clustering RTSs into cumulatively representative cases called system scenarios (or simply, scenarios). Assigning a cycle budget that is disproportionately high to an RTS may lead to degradation of the system performance, due to the relaxation of the timing deadline that is assigned. Conversely, an underestimated cycle budget for an RTS may create a false impression of negative slack, thus pushing the PID controller to an over-boosting that will increase energy consumption. System scenarios aim [10] to balance the above trade-off by clustering RTSs to scenarios ensuring the worst-case RTS for each scenario without losing significantly in terms of accuracy (achieved precision around 90%) Assuming a workload that is composed of A different RTSs, each with cycle budget Ni, where $i = 0, 1, A$, the goal of this methodology is to narrow down to B scenarios, each with cycle budget estimators (or simply, scenarios) $\hat{N}_j$ where $j = 0, 1, B$ and $A >> B$. The reduced number of scenarios to be considered (in comparison to RTSs) facilitates their storage in the system and their less complex utilization at runtime. Additionally, clustering from cycle budgets N_i to estimators $\hat{N}_j$ should be such that no cycle budget is excessively under- or overestimated.

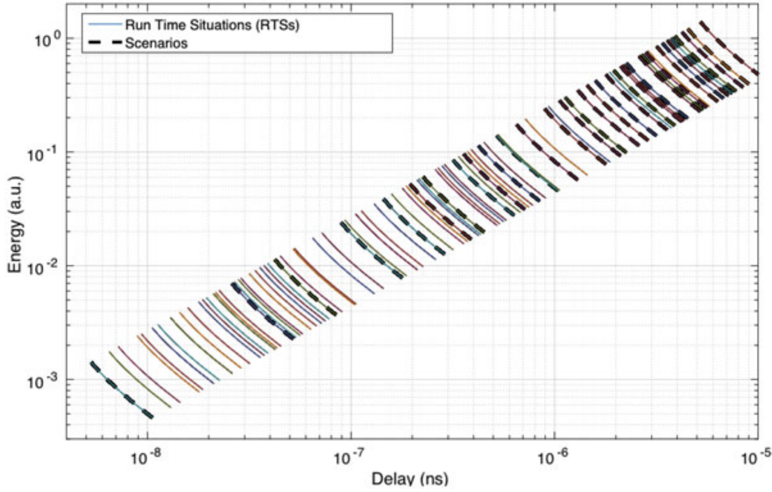

Fig. 6.6 RTS clustering to scenarios on a representative spectrum sensing application —Energy is calculated using the square law for dynamic power, i.e., $P_{\text{dyn}} \infty f V_{dd}^2$ and numbers are normalized to the smallest RTS at $V_{dd} = 1.2\,\text{V}$ and $f = 2.588\,\text{GHz}$

To achieve this, the difference between N_i and $\hat{N}_j$ has to be minimized for every clustering. We illustrate the above concepts with an example: Fig. 6.6 illustrates an example of a significant number of RTSs of a real spectrum sensing application that have been clustered to scenarios. Each RTS represents a certain cycle budget. Given a number of OPP points on any processor that supports DVFS, we can draw an energy vs. delay line, similar to the solid lines of Fig. 6.6. For these very lines to be produced, we are using the OPP points of a well-documented processor [6] which have also been used in the previous work [7]. The multitude of solid lines in Fig. 6.6 corresponds to a spectrum sensing application and already represents the complexity of handling TNs using the RTS abstraction. Clearly, this amount of information needs to be abstracted further. System scenarios can replace the many RTSs with a more manageable set of scenarios, to be used as cycle budget estimators. The idea is to bin RTSs, based on similarity of their cycle budgets as they are projected on the multidimensional Pareto metric space (e.g., Energy Delay). In that sense, RTSs that require a similar number of cycles to execute are binned to the same scenario. The scenario is uniquely identified with its cycle budget estimator $\hat{N}_k$, which is the minimum cycle budget of the RTSs that have been binned therein. By extension, TNs that belong to similar RTSs are also falling under the same scenario when perceived at runtime. More specifically, in Fig. 6.6, we see an artist's illustration of the clustering process. The assigned cycle budget is that of the fastest RTS in the scenario. As we move away from the origin of the axis, new scenarios are defined and certain RTSs are mapped to them. The assigned cycle budget is that of the fastest RTS in the scenario. Eventually, the wealth of RTSs is produced by all possible combinations of application configuration such as accuracy settings and

input size. The system scenario methodology reduces this huge set of RTSs to a more manageable set of scenarios. Given that the fastest RTS is dictating the budget estimator of each scenario, it is clear that the complexity reduction detailed above is introducing an accuracy overhead, which is typically called clustering overhead. There lies a trade-off between the accuracy and the number of the derived scenarios.

A fine grain clustering decreases the gap between the budget estimators of the same scenario but increases the number of the scenarios and vice versa. Assigning the minimum cycle budget from the RTSs that belong to a scenario cluster forces the rest included RTSs due to their distance from the worst case to be more restricted, compared to their actual timing deadlines. This trend is proportional to the clustering overhead that leads the PID controller to act more aggressively, boosting the operation frequency. This effect is not necessarily undesirable for the purpose of our design. The mission of the PID controller is to absorb the interfering cycle noise of the RAS interventions. The extra frequency boosting operates preventively to the extra delays imposed by cycle noise. The clustering overhead acts as an offset that permits to partially eliminate the introducing noise without increasing slack. This permits less PID controller interventions preventing the repeated frequency DVFS switches. As a result, the inevitable clustering overhead is not undesirable, provided that it does not force the PID controller to "worst-case behavior" (i.e., forcing it to suggest the maximum operating frequency). Without the loss of generality in our claims, a simple linear binning scheme, which results in the cycle budget estimator, is shown in Fig. 6.6. Prior art already features many ways to cluster scenarios at design time and detect them at runtime from the RTSs encountered during TN execution (Fig. 6.14).

6.1.4 Different Classes of Scenarios

The PID controller as refereed previously absorbs the interfering cycles of the RAS interventions. The controller adjusts the processor frequency, actuating suitable DVFS schemes. The incoming RTSs are valued at a certain cycle budget that is used as reference for the expected operation cycles. Cycle noise being superimposed on cycle budget introduces negative slack that pushes the PID controller to a new DVFS boosting. The open issue is how the PID controller can act proactively. Due to the RAS intervention randomness, the deployment of prediction mechanisms that fully forecast the interfering noise at runtime is neither realistic nor cost effective. The key point is how to encapsulate in the reference cycle budget the gradual and partly

predictable performance of the platform. The goal is to provide a low-cost run-time cycle budget mechanism that will provide opportunities for less slack without eliminating it.

The existing slack reclaiming mechanism exploits a system scenario detector. The system scenarios provide the default cycle budget of the running RTSs. System Scenarios are clusters of RTSs with similar cycle budgets as shown previously in Fig. 6.6. The hierarchy and the structure of the RTSs in the System Scenarios is the key for an efficient PID implementation. The effects of hardware degradation increase RAS interventions so a static definition of system scenarios and an equally static cycle noise formulation is not expected to yield dependable performance in an efficient way. The dynamic adjustment of the system scenario concept to the new requirements is an intermediate step towards the ultimate introduction of dynamic scenarios.

Outlining the system scenario primitives, RTSs are discrete executions of TN sequences, with a default cycle budget while the most restricted deadline into a scenario represents the fixed cycle budget for the entire RTSs. Due to variability of the RTSs into the same scenario, overestimated constraints push the PID controller to a frequency multiplier higher from what is required, thus partially mitigating noise cycles and paying a price at energy consumption. A noise shift higher than a limit creates negative slack and the PID controller is called to force to zero with a frequency increase. On condition that the RTS positions into scenarios can be updated at runtime, the scenario detector can modify the RTS hierarchy by re-clustering the existing scenario distribution. This process provides a run-time adjustment of the RTS cycle budget estimations [1]. This leads the PID controller to act partially proactively (see Fig. 6.7a) considering RAS events that are gradually becoming a norm on the target processor. To ensure a fully proactive operation [5] in a complete dynamic scenario approach, a predictor has to provide a runtime trend of the incoming noise. Redistributing the predicted slack across a future scheduling period in a globally optimized way as pictured in Fig. 6.7b we will achieve an even better trade-off between delay and energy. In the context of the current chapter, we focus on the scenario adaptivity through re-clustering defining the conditions of the

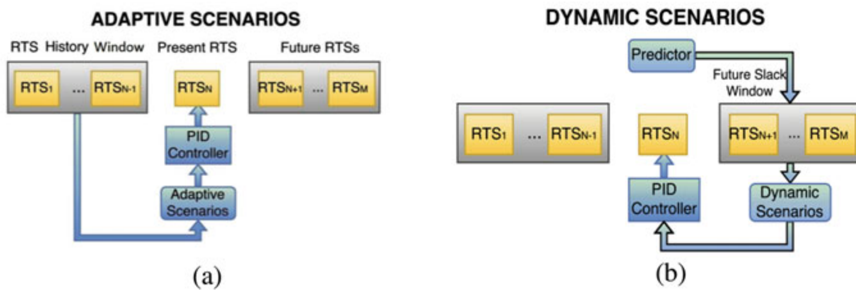

Fig. 6.7 Flow charts adaptive vs dynamic scenarios. (**a**) Adaptive scenarios. (**b**) Dynamic scenarios

scenario modifications at runtime. The incorporation of a predictor is not part of the current work and represents an individual challenging issue that needs more elaborate research that will be part of the future work.

6.1.5 *Adaptive Scenarios*

In this subsection, we explain how an adaptive re-clustering mechanism is built from scratch. The objective is to keep the operation cost of such a mechanism low, adapting at runtime modifications that keep our system updated. The key issue is the splitting of the processes that take place at design time and at runtime. The concern is to build a light consuming flexible mechanism that reacts quickly ensuring the respecting deadlines. Thus, the heavy computation parts are pushed to the design time, keeping at run-time decisions with short response time. This presupposes a design time analysis that builds scenario metadata without requiring repeating the whole calculations at runtime.

In this direction, the involved TNs of the targeted application are analyzed at design time. The targeted application is split into TNs based on specific criteria [3]. The combination of TN execution creates the exploration space of the involved RTSs. To identify the operation cost of each RTS, we analyze the several trade-offs for the available DVFS knobs. This information is critical to define which knobs deal with the application requirements. The depicture of the RTS on a Pareto space provides a good representation of the RTSs behavior and a comparison of the interested design points presents the cost impact of each RTS. This comparison creates a hierarchy of the RTSs based on their position in the metric space (see Fig. 6.8). Based on this hierarchy, cluster domains define the enclosed RTSs into scenarios. Thus, a splitting of the metric space into domains is a first approach that provides an abstraction view of the operation cost hiding the details of the individual RTSs. The split point is defined as the upper bound (see in Fig. 6.8 RTS1 red

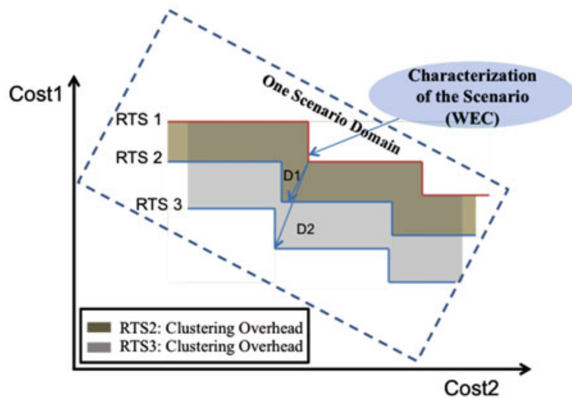

Fig. 6.8 RTS Pareto curve hierarchy into scenarios

curve) of each scenario that distinguishes it from the other scenarios. Two criteria specify these points. The first criterion is that the chosen upper bound has to have the minimum distance from the included RTS Pareto curves (see in Fig. 6.8 D1, D2). The second criterion that cannot be visualized is the clustering of the suitable combinations of RTSs that minimizes during the operation of switching between the scenarios. Thus, in the first case an algorithm has to solve a trivial distance minimization problem and a second algorithm solves a probability distribution issue. Both algorithms tend to be competitive in respect to the targeted improvements. More precisely, the first algorithm aims to reduce the clustering overhead by increasing the switching overhead and vice versa. The achievement of the minimum total overhead is a balance between them, and more details can be found in [10]. Both algorithms can be highly computation consuming due to the involved number of RTSs and the derived scenarios so they take place at design time and provide the necessary information for the run-time stages. In particular, the final clustering is based on providing information (metadata) about the cycle budget of each scenario based on the most time-consuming RTS that will be exploited at runtime.

At runtime, a re-clustering mechanism identifies the running RTS and the correlated slack. If the same RTSs present cycle noise repeatedly, this means that these RTSs require a regrouping adjustment. For this reason, we use a sliding window that traces the history of the RTS running. The re-clustering starts by priority from RTSs with the most significant slacks at more frequent times. The RTS slacks are watched on fixed time windows. The result of the extra cycle noise is to increase the overhead of the effected RTS pushing it in a more costly position at the metric space (delay-energy). The possible transitions of an RTS due to the added cycle noise are outlined in Fig. 6.9a–e. RTS1 represents the shifted-RTS. For each repositioning, an individual re-clustering decision (each decision will be described at the following subsection) is taken to equalize the performance degradation. The scenario metadata that were initialized at design time are updated after a re-clustering. Apart from the scenario budget, information is included about the distance from the neighboring scenarios. This information permits the rapid recognition of the optimal re-clustering decision updating only the scenario data that are changed. Each re-clustering decision is taken based on the new position of the RTS at the metric space after a reviewing of the updated data, ensuring that the RTS deadlines will be respected with the minimum cost. The re-clustering is a parallel process that does not interrupt the operation of the PID controller. In any case, the re-clustering time can be easily kept a fraction of the RTS operation time, due to the simplicity of the algorithm to find the new shifted RTS position without reordering the whole RTSs by scratch. At the end of each re-clustering, the PID controller has updated information about the suitable DVFS scheme that minimizes the generated slack paying a price in energy. A trade-off between the produced slack and energy exists that will be explored in the last section with experimental results.

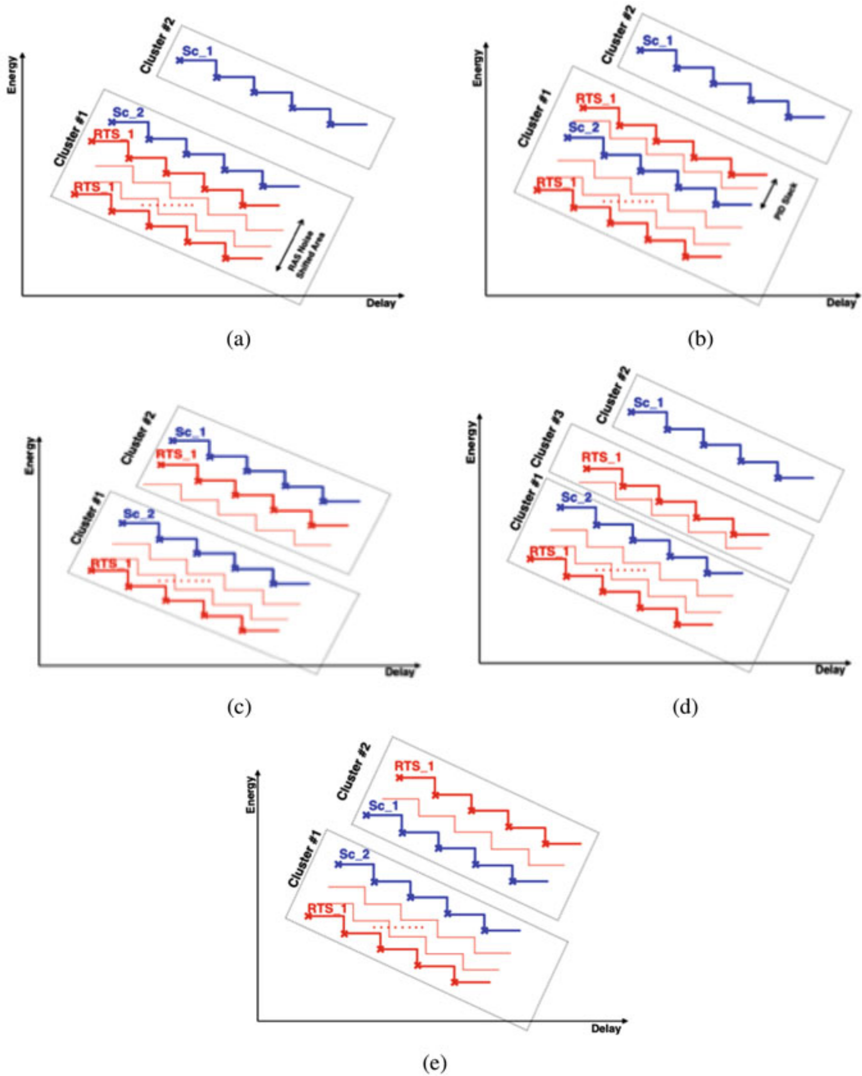

Fig. 6.9 RTS-Shifting cases due to RAS interventions. (**a**) Shifted-RTS1 remains at the scenario bounds. (**b**) Shifted-RTS1 remains at the scenario changing the bounds. (**c**) ShiftedRTS1 is merged by another neighboring scenario. (**d**) ShiftedRTS1 is creates an individual scenario. (**e**) ShiftedRTS1is merged by another scenario changing the new scenario bound

6.1.5.1 Adaptive Scenarios Instantiated for the Cycle Noise Reduction in the PID Controller

Concentrating on the re-clustering decisions, the key point, as referred previously, is to distinguish transient from (quasi-)permanent cycle noise events. For example, a very brief cycle noise burst that is never reoccurring should not be triggering the re-clustering phase. On the other hand, when cycle noise bursts appear to have a more predictable behavior, re-clustering RTSs may be required in order to increase the efficiency of our PID-controlled performance dependability scheme. The hardware degradation effects occur during the operation life and cannot be predicted in advance. Thus, an updated trace history of the budget and noise per RTS is needed to consider their fluctuation with time. Identifying a new trend in cycle noise manifestation is important for the changes/reformulation of the system scenarios through RTS re-clustering.

The RAS interventions change the RTS hierarchy into scenarios by removing from the default cycle budgets the noise cycles and shifting the position of the RTSs in the scenarios. Thus, more restricted deadlines trigger more aggressive DVFS schemes that absolve the extra cycles. As the noise is impossible to be accurately defined at runtime, the negative slack is identified as noise. Examining the re-clustering decisions that have to be considered at runtime, we outline the following cases:

1. In the first case, despite the added cycle noise, the scenario budget is restricted enough and no slack exists, hence there is no need for scenario change. This case is visualized by cost perspective in Fig. 6.9a. The slack absence is visible from the fact that the added noise does not exceed the distance between the RTS and the scenario budget, exploiting the clustering overhead as an offset. Thus, clustering overhead can be utilized as a proxy of cycle noise tolerance. A high clustering overhead permits wide RTS shifting without slack but with negative effects at the usage of overestimated DVFS schemes. The next four cases cover situations where the negative slack cannot be avoided.
2. In the second case, the noise pushes an RTS to a more restricted budget than the entire scenario budget. The new RTS position forces the revision of the existing scenario budget (see Fig. 6.9b). The new budget is decreased per equal cycles to negative slack.
3. The differentiation at the third case is that another existing scenario merges the shifted RTS (see Fig. 6.9c). The new position of the RTS fits better to another existing scenario so an updated list changes the distribution of RTS into scenario.
4. The fourth case is when the shifted RTS represents a new distinct scenario (see Fig. 6.9d). Two reasons lead to such a decision: (1) none of the existing scenario cycle budgets cover the new RTS constraint so a new worst-case scenario is needed or (2) the creation of a different scenario is more efficient from clustering overhead perspective, than to be merged by an existing scenario due to the distance between the scenarios.

5. The fifth case is a combination of the second and the third case. The RTS is merged to another existing scenario changing its upper bound (see Fig. 6.9e).

6.1.6 Experimental Results

Having applied static system scenarios and adaptive system scenarios to the RTSs of a representative application [9] (starting up with roughly 30 scenarios), we submit this workload to a simple RTS-level simulator [7], to compare the efficiency of the new concept with the existing system scenario approach. The targeted application is a real spectrum sensing application that consists of 25 TNs that constitute an exploration space of about 800 RTSs (see Fig. 6.10). For the sake of completeness, we repeat the basic functionality of the simulator here. As illustrated in Fig. 6.4, the simulator operates at RTS granularity. For each RTS, the scenario clustering provides the cycle budget estimator $\hat{R_n}$.

Additionally, cycle noise is injected based on the values of Λ and that represent the rate and amplitude of RAS events, respectively. We assume that cycle noise follows a bivariate Gaussian distribution with mean Λ and M values (λ and μ) and sigmas ($\sigma\{\Lambda\}$,$\sigma\{M\}$). The timing and power models of the simulator create traces from which DVF and total energy can be deduced. Given the definition of cycle budget estimators and their derivation, DVF expresses the per unit increase in execution time required for an instruction stream, following the temporal overhead of invoked RAS mechanisms and is formulated by Eq. (6.7). Respectively, PVF expresses the per unit increase in number of clock cycles required for a specific workload, as a result of RAS technique invocation and is formulated by Eq. (6.8). It is noteworthy that all overheads of DVFS actuation are included in both the timing and power models [7]. Finally, at the end of each RTS, the PID controller selects the frequency multiplier of the next iteration, as detailed in Fig. 6.4.

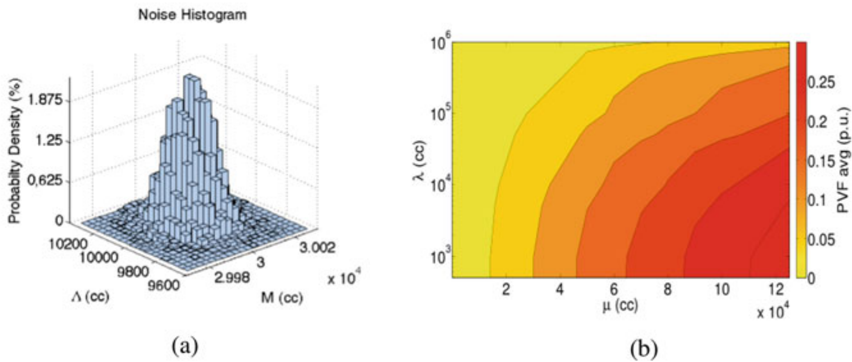

Fig. 6.10 Statistical representations of the simulation conditions. (**a**) Histogram of the cycle noise injected in one RTS steam. (**b**) PVF average values of 100 identical iterations

$$\mathrm{DVF}[n] = 1 - \frac{\sum_{i=0}^{n} \frac{N_i}{m_{\mathrm{ref}}}}{\sum_{i=0}^{n} \frac{N_i + x_i}{m_i}} \tag{6.7}$$

$$\mathrm{PVF}[n] = 1 - \frac{\sum_{i=0}^{n} N_i}{\sum_{i=0}^{n} N_i + x_i} \tag{6.8}$$

The simulation results presented herein correspond to a set of random streams of RTSs, drawn from the scenarios of Fig. 6.6. The simulated workload assumes equally probable RTSs. Cycle noise is injected by sweeping μ and λ and assuming $\sigma\{M\} = 50$ and $\sigma\{\Lambda\} = 10$. The noise histogram for one stream of RTSs is shown in Fig. 6.10a. In fact, each measurement corresponds to a stream of 105 RTSs and, when enabled, PID controller gains are $k_p = 1.5 \times 10^{-7}$, $k_i = 5 \times 10^{-8}$, and $k_d = 4 \times 10^{-8}$. For each combination of μ and λ, we present the average over 100 random measurements to provide insight into the statistical variability of our simulation results.

A first conclusion of our experiments is that apart from the extreme cases of noise (high μ and low λ), the PID controller manages to enable dependable performance, especially considering the increment in PVF (see Fig. 6.10b) that cycle noise injection is causing. In addition, for the average values of Fig. 6.11a, b, we observe that $\mathrm{DVFDyn}_{\mathrm{scen}} \leq \mathrm{DVFSys}_{\mathrm{scen}}$. Except from an area ($\mu > 9 \times 10^{14}$)

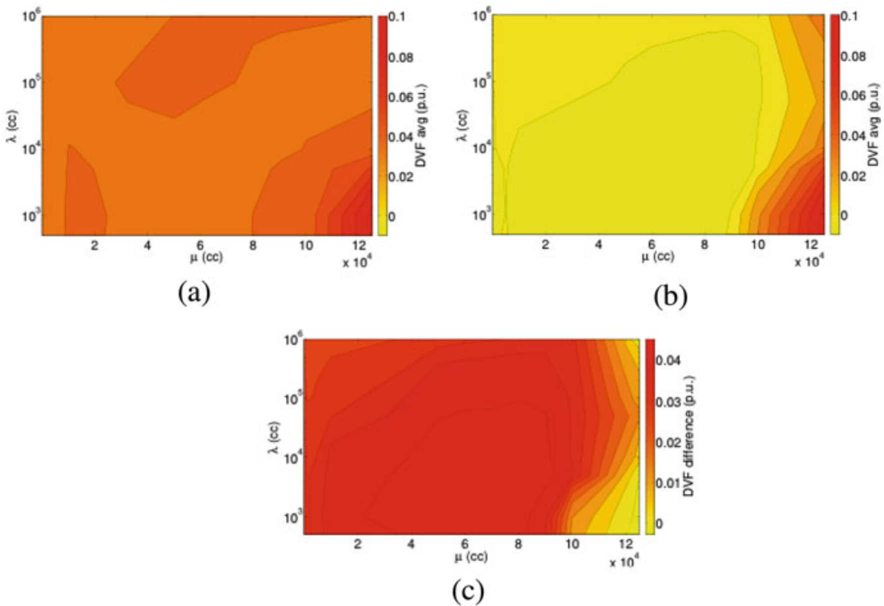

Fig. 6.11 DVF comparison system vs dynamic (adaptive) scenarios. (**a**) DVF system scenarios. (**b**) Dynamic (adaptive) scenarios. (**c**) DVF system scenarios less dynamic (adaptive) scenarios

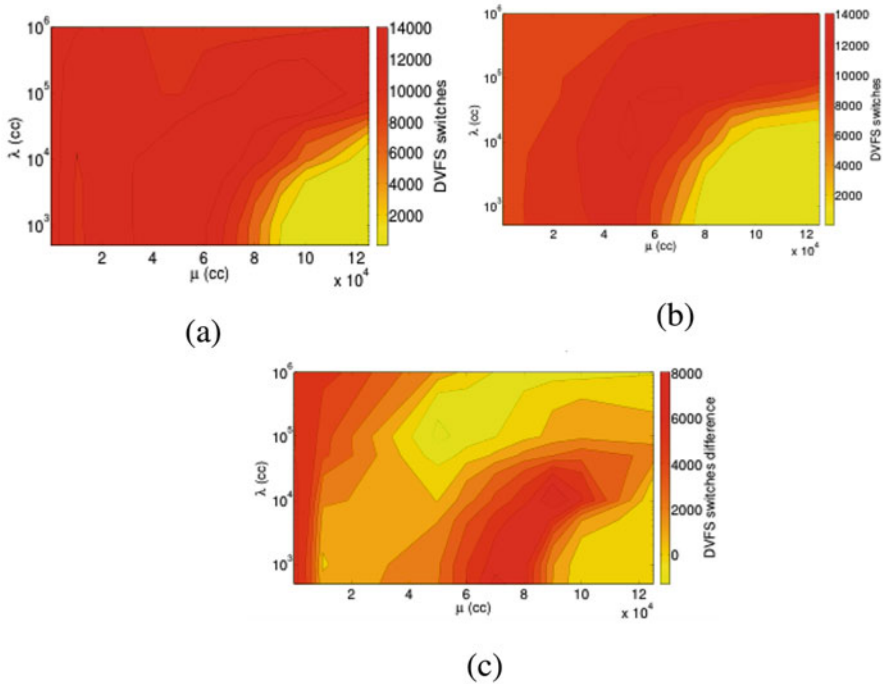

Fig. 6.12 DVFS switches comparison system vs dynamic (adaptive) scenario. (**a**) DVFS switches system scenarios. (**b**) DVFS switches dynamic (adaptive) scenarios. (**c**) DVFS switches comparison system less adaptive scenario

that the mitigation of the cycle noise is not possible due to the limitation at the frequency boosting, adaptive scenarios are evident more effective in terms of eliminating negative slack meeting the deadlines ($\text{DVFDyn}_{\text{scen}}$) compared to the system scenarios. Another important remark is the deviation in the total number of DVFS switches as shown in Fig. 6.11. It is clear that the adaptive scenarios for the same amount of cycle noise cause significantly less DVFS switches. The scenario adaptiveness seems to counteract the slack fluctuation so the PID controller does not need to overreact (Fig. 6.12).

Finally, total energy (see Fig. 6.13) is increasing as the injected cycle noise profile becomes more aggressive (high μ and low λ). This is to be expected, given that the PID controller needs to work overtime in these cases. The remarkable conclusion is that the deviation between the system and adaptive scenario energy consumption is minor (see Fig. 6.13a, b) and do not exceed the 3.5% at the worst cases (see Fig. 6.13c). This occurs due to the fact that the more energy consuming frequency boosting of the adaptive scenarios is partially equalized by the less DVFS switches (see Fig. 6.12c). Nevertheless, a fundamental trade-off exists between the achieved performance as expressed by the DVF and the energy impact. Figure 6.14 presents this trade-off between static and adaptive scenarios. However, this will also be the

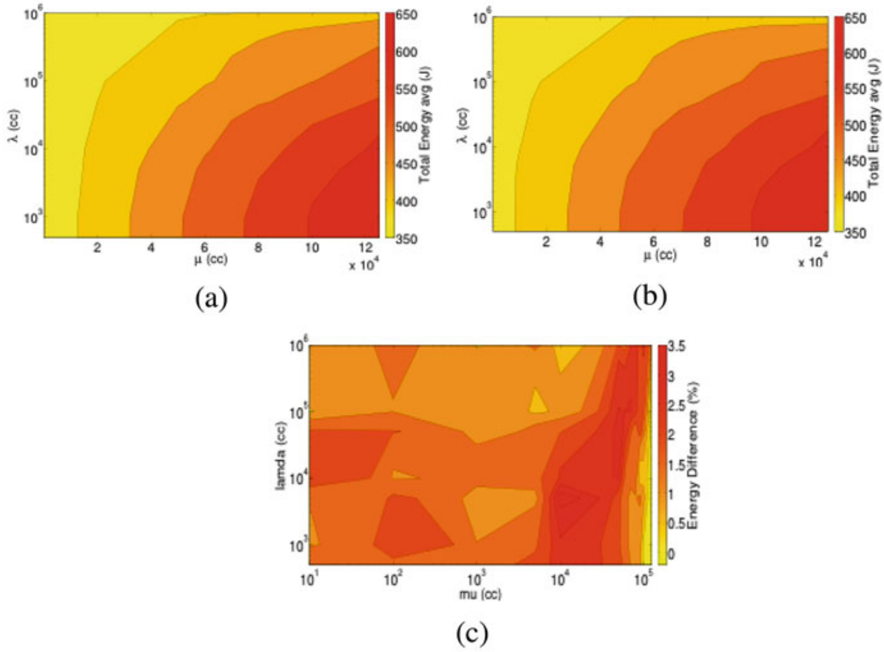

Fig. 6.13 Energy comparison system vs dynamic (adaptive) scenarios. (**a**) Energy system scenarios. (**b**) Energy adaptive scenarios. (**c**) Percentage energy deviation system less adaptive scenarios

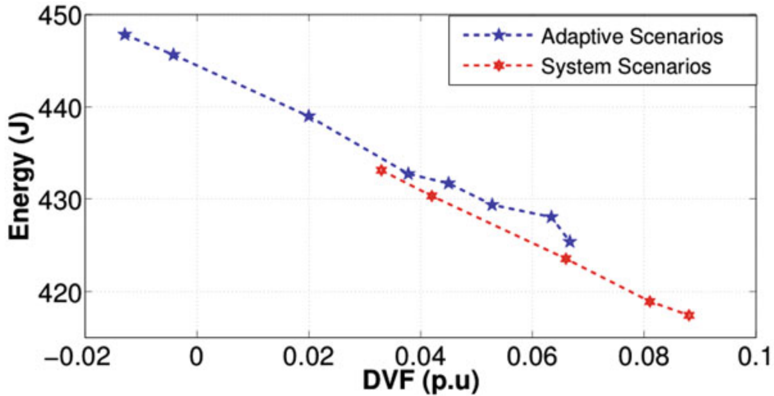

Fig. 6.14 Static—adaptive scenarios performance (DVF) vs energy trade-offs

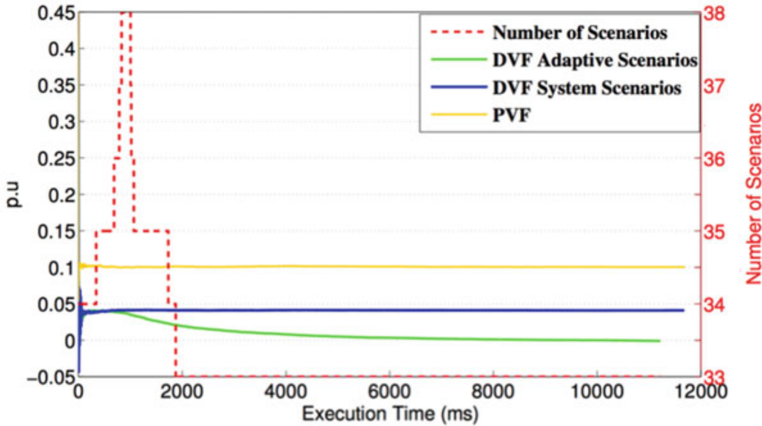

Fig. 6.15 Simulation instantiation (M: $\mu = 30{,}000\sigma = 50$, $L : \mu = 10{,}000\sigma = 100$) of dynamic (adaptive) scenarios applicability on the proposed DVFS scheme for dependable performance

case for all applications where enough DVFS switching options are available to trade-off spending more energy with more slack creation.

The Pareto curves of the two approaches (adaptive vs system scenarios) seem to be very close, presenting similar cost trade-offs (delay-energy). The adaptive scenario appears just a little worse trend (longer distance from the start of the metric axes) due to the extra implementation cost that they require. The gain is their ability to achieve better performance results (DVF close and under zero).

More precisely, Fig. 6.15 highlights how the adaptive scenarios clearly improve the performance dependability at a representative simulation instantiation (M: $\mu = 30{,}000\,\sigma = 50$, $L : \mu = 10{,}000\,\sigma = 10$). While system scenario DVF seems to balance to a static value (0.05), adaptive scenarios adjust DVF close to zero. For a transition period (0–2000 ms), the number of the adaptive scenarios is fluctuating to become to a stable state. During this time, the re-clustering process generates new scenarios increasing gradually their number up to 38 (1000 ms) and after this point some of the existing scenarios are merged to reach the 33 scenarios (2000 ms). Thus, the adaptive process creates and merges scenarios applying the aforementioned re-clustering decisions to find the optimum scenario hierarchy. When this is achieved (after 2000 ms), the scenarios remain stable and the DVF comes close to zero without applying an over-boosting policy (negative DVF) that will consume energy without reason. Thus, Dynamic (adaptive) scenarios seem to improve performance dependability without remarkable energy price.

6.1.7 Conclusion

In the context of the current chapter, we deployed the theoretical background and the operation specifications of the HARPA-RT engine. We presented both a system scenario-based PID mechanism and a partially proactive scenario configuration that adjust the responses of the controller at the presence of RAS interventions ensuring performance dependability. The key point in the adaptive approach is the re-clustering of the existing system scenarios with a flexible way, absorbing the cycle noise that is gradually becoming a norm during the observed application operation. The simulation results prove the efficiency of this approach, ensuring the system dependability constraints while achieving an energy gain up to 15% compared to a guardbanding implementation and exploiting in average 83.9% of the maximally available potential energy gain margins.

References

1. Gheorghita, S. V., Palkovic, M., Hamers, J., Vandecappelle, A., Mamagkakis, S., Basten, T., et al. (2009). System-scenario-based design of dynamic embedded systems. *ACM Transactions on Design Automation of Electronic Systems (TODAES), 14*(1), 3
2. Hardy, D., Sideris, I., Ladas, N., & Sazeides, Y. (2012). The performance vulnerability of architectural and non-architectural arrays to permanent faults. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* (pp. 48–59). New York: IEEE.
3. Ma, Z., Marchal, P., Scarpazza, D. P., Yang, P., Wong, C., Gómez, J. I., et al. (2007). *Systematic methodology for real-time cost-effective mapping of dynamic concurrent task-based systems on heterogenous platforms*. Berlin: Springer Science & Business Media.
4. Mukherjee, S. (2011). *Architecture design for soft errors*. San Francisco: Morgan Kaufmann.
5. Munaga, S., & Catthoor, F. (2013). Systematic design principles for cost-effective hard constraint management in dynamic nonlinear systems. In *Innovations and Approaches for Resilient and Adaptive Systems* (pp. 1–28). Hershey: IGI Global.
6. Park, S., Park, J., Shin, D., Wang, Y., Xie, Q., Pedram, M., et al. (2013). Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 32*(5), 695–708.
7. Rodopoulos, D., Catthoor, F., & Soudris, D. (2015). Tackling performance variability due to RAS mechanisms with PID-controlled DVFS. *IEEE Computer Architecture Letters, 14*(2), 156–159.
8. Xie, M., & Lai, C. D. (1996). Reliability analysis using an additive Weibull model with bathtub-shaped failure rate function. *Reliability Engineering & System Safety, 52*(1), 87–93.
9. Zompakis, N., Noltsis, M., Rodopoulos, D., Catthoor, F., & Soudris, D. (2017). Energy efficient adaptive approach for dependable performance in the presence of timing interference. In *Proceedings of the on Great Lakes Symposium on VLSI 2017* (pp. 209–214). New York: ACM.
10. Zompakis, N., Papanikolaou, A., Raghavan, P., Soudris, D., & Catthoor, F. (2013). Enabling efficient system configurations for dynamic wireless applications using system scenarios. *International Journal of Wireless Information Networks, 20*(2), 140–156.