3

Feedback-Based Admission Control Heuristics

Applying feedback mechanisms to monitor the capacity of computing resources and quality-of-service (QoS) levels can guarantee a bounded time response, stability, bounded overshoot even if the exact knowledge of a system workload and service capacity is not available a priori [2]. Thus, in case of a careful fine-tuning of parameters, they can be successfully applied even to systems with real-time constraints (see the Related Work section). It was verified that this approach helps to find a trade-off between multiple objectives of a workflow management system, e.g., minimal slacks and maximum core utilisation [53].

The feedback-control dynamic resource allocation heuristics impose some requirements on the target system, which should guarantee that the appropriate input data is available and that the generated output can be used to perform the proper resource allocation. Usually to perform a resource allocation decision we can rely on various metrics, provided by the monitoring infrastructure tools and services, such as utilization and the time latency between input and output timestamps [81]. The system should also guarantee an appropriate level of responsiveness to the decisions made by the heuristics, as well as update the values of the metrics used as inputs in the algorithm frequently enough for the particular application. The platform should support scheduling on distributed-memory infrastructure resources. It is important to provide the heuristic algorithm with realistic data about system workload, service capacity, worst-case execution time and average end-to-end response time [84].

The task mapping process presented in this chapter is comprised of the resource allocation and task scheduling. The technique proposed in this chapter assumes the presence of a common task queue, which is used by the global dispatcher. The resource allocation process is executed on a particular processing unit. Its role is to send the processes to be executed to other processing units, putting them into the task queue of a particular core. The process dispatching, i.e., selecting the actual process to run, is also a part of the scheduling algorithm and is carried out locally on each core. It is

assumed that task scheduling is performed in a non-preemptive early deadline first (EDF) or first-in-first-out (FIFO) based manner.

Later in this chapter we propose an algorithm to map firm real-time tasks into multi-core systems dynamically, using dynamic voltage and frequency scaling (DVFS) to decrease energy dissipation in cores. According to simulation results, the proposed method leads to more than 55% of dynamic energy reduction.

3.1 System Model and Problem Formulation

3.1.1 Platform Model

The controlling process of dynamic behaviour of a target system can be performed in two ways: feed-forward and feed-back, presented in Figure 3.1. Although the closed-loop scheme includes larger number of functional blocks and requires measuring output values, it requires less accurate model of the target system and is also more resistant to disturbances [7]. A closed-loop system is characterised with a feedback loop, which carries values of *measured output* (*y(t)*, aka *controller value*). These values are subtracted from their desired value (r, *reference signal*, *setpoint*). The result of this operation forms *error* ($e(t)$) signal, which is used to compute *control input* ($u(t)$). This value is sent back to the target system.

A proportional-integral-derivative (PID) controller is particularly often used in various industrial control system, recently including computing systems [57].

The PID controller in the time-domain form is described in the following way:

$$u(t) = k_p e(t) + k_i \int_0^t e(\tau)d\tau + k_d \frac{d}{dt} e(t). \tag{3.1}$$

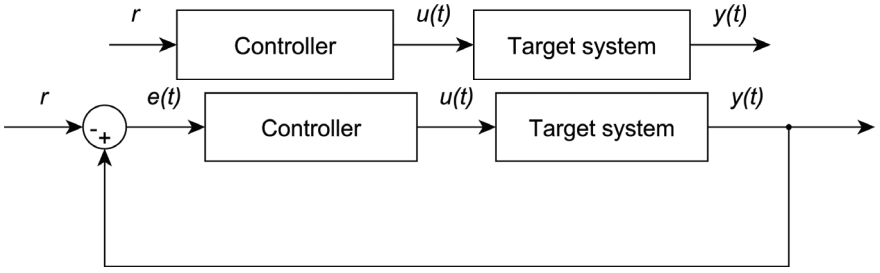

Figure 3.1 Block diagrams of control system architectures: feed-forward (*above*) and feedback (*below*).

The determination of proportional (k_p), integral (k_i) and derivative (k_d) constant components of PID controller is known as PID controller tuning.

The PID controller is often presented in an equivalent form in the frequency domain, where function (3.1) of time t is presented as a function of complex frequency s using the Laplace transform, leading to

$$K(s) = k_p + \frac{k_i}{s} + k_d s. \tag{3.2}$$

A PID controller is often described using other constant parameters: k – so called proportional gain, T_i – integral time constant and T_d – derivative time constant

$$K(s) = k\left(1 + \frac{1}{sT_i} + sT_d\right). \tag{3.3}$$

Since increasing the value of parameter k_d enhances noise, the derivative component is often omitted in numerous practical applications [57]. It is also not used in the work described in this chapter despite its positive influence on stability or speed.

In Figure 3.8, a general view of the proposed architecture is presented.

3.1.2 Application Model

We consider a workflow of a particular structure. There is no dependencies between tasks and the deadline of each task computation is set as a sum of its computation time multiplied by an arbitrary value and arrival time. There is only one priority of task; tasks cannot be preempted during their execution. During simulation we measure cluster core utilisation, which is the percentage of cores in the clusters executing tasks in particular simulation time t.

3.2 Distributed Feedback Control Real-Time Allocation

After releasing task t_i, the role of the dispatcher is to decide which of the clusters C_j, $j = 1, \ldots, m$, is to execute the task. This decision can be made using various metrics, we decide to apply a choice of the cluster whose cores are currently the most idle. If more than one core satisfies the chosen condition, one of them is chosen randomly. For the comparison purpose we also allowed the dispatcher to choose the target cluster C_j in the round-robin manner. The task t_i is then placed in the j-th queue.

Each j-th cluster includes one admission control block, AC_j. Its role is to decide whether a task t_i, read from the j-th input queue, should be executed by the cluster. The first condition of admittance is that the deadline of t_i, D_i, is not lower than the sum of its computation time, C_i and the current simulation

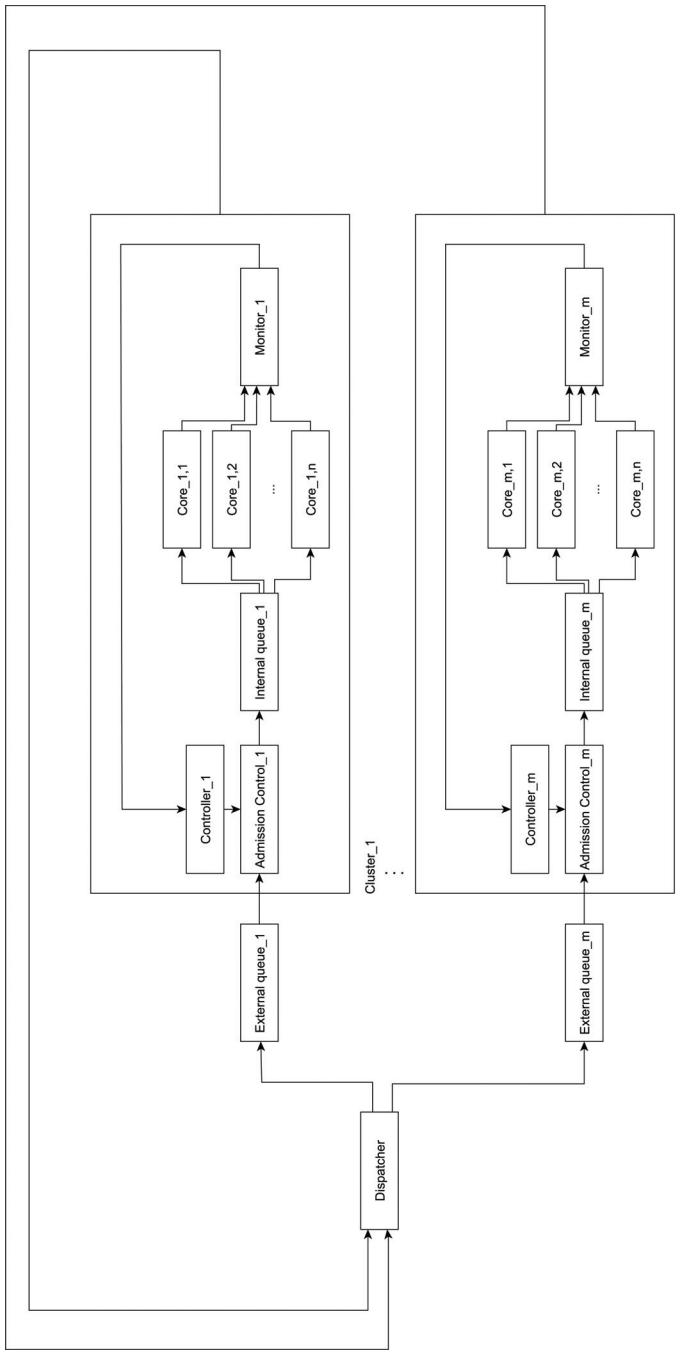

Figure 3.2 Distributed feedback control real-time allocation architecture.

time t. Then the input value from the controller, $u_j(t)$, is tested. If this value is positive, the task is admitted, otherwise it is rejected. Admitted tasks are placed in the internal cluster queue. This queue is planned to be rather short to minimise the delay between decision about admittance and the execution of the task, and to keep the timeliness of the lateness input.

To control the admittance in each cluster, we use discrete-time controllers in two variants. The first of them is a PI (i.e., a PID controller without the derivative component) whose controlled value is an average lateness of a (parameterisable) number of previous tasks computed by the cluster cores, where lateness is defined as the difference between a task response time and its deadline. If a lateness is negative, the task has been finished before its deadline, and positive otherwise. The current value of lateness is compared with the setpoint, r, and an error $e_j(t)$ is computed. It is provided as an input to a controller, which computes admittance allowance value $u_j(t)$. The second variant includes a P controller (i.e., a PID controller with the proportional component only) whose output value $u_j(t)$ depends on the difference between the current core utilisation and the setpoint. The output value of $u_j(t)$ is sent to AC_j, where it is used to perform a task admittance decision. In both situations, as long as value of control input $u_j(t)$ is positive, the task is allowed to be submitted to cores, otherwise it is rejected. The admitted tasks are placed in the queue.

An idle core $Core_{j,k}$, $j = 1, \ldots, m$, $k = 1, \ldots, n$, fetches a task t_i from the j-th core queue and then executes it in a non-preemptive manner. After execution, the lateness of the i-th task, $L_i = D_i - t$, is computed. Each core also informs the observer whether it is occupied or idle.

The role of observer $Monitor_j$ is to compute two metrics based on the performance of all cores in the j-th cluster. The first metric is core utilisation and the second metric is an average lateness of the previous q tasks computed by the cores in the j-th cluster. These data are provided to the j-th controller and the dispatcher.

3.3 Experimental Results

3.3.1 Controller Tuning

In order to check the efficiency of the proposed feedback-based admission control and real-time task allocation, we developed a simulation model using SystemC language. We firstly configured it to operate in the open-loop manner.

In Figure 3.3 we present the maximum task lateness in the open-loop system consisted of three clusters, each including three cores. In every situation, at 5,000 ns a number of tasks, ranging from 5 to 500, each requiring execution time equal to 50,000 ns, has been generated. Then we looked at the

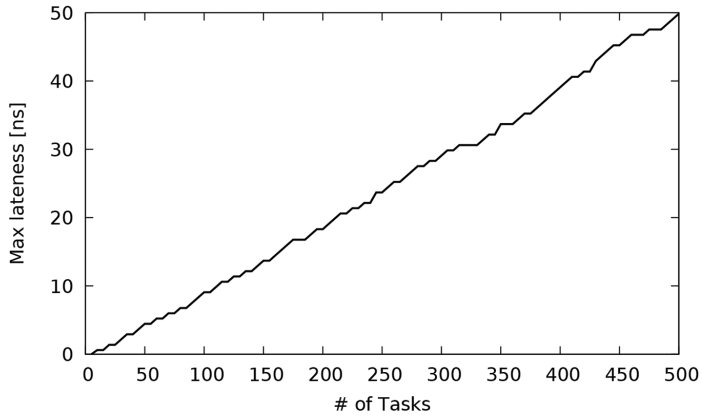

Figure 3.3 Maximum normalised task lateness (with execution time equal to 50,000 ns) in step responses for a number of tasks (3 clusters, 3 cores in each).

maximal task lateness, where each lateness has been normalized by dividing it with the deadline.

In order to tune the controller, we analysed the step-input maximum normalised task lateness response in the open-loop system. As an input we have used a burst release of 500 tasks (with execution time equal to 50,000 ns) at 5,000 ns. The system was comprised of 3 clusters, each including 3 computing cores. The obtained result confirms the accumulating (or integrating) nature of the process, which can be described by the following model [64]:

$$F(s) = \frac{V}{s} e^{-s\tau}, \tag{3.4}$$

where τ is the dead time, i.e., the delay between changing input and the observable output reaction, and V is the velocity gain, which is the slope of the asymptote of the process output.

In such kind of processes, to choose proper values of PI controller components, AMIGO (Approximate M-constrained Integral Gain Optimisation) tuning formulas can be applied [7]. According to these formulas, the parameters k and T_i are equal:

$$k = \frac{0.35}{V\tau}, \tag{3.5}$$

$$T_i = 13.35\tau. \tag{3.6}$$

Both these parameters can be determined using the step output illustrated in Figure 3.4: $k = 0.2741$ and $T_i = 1108$.

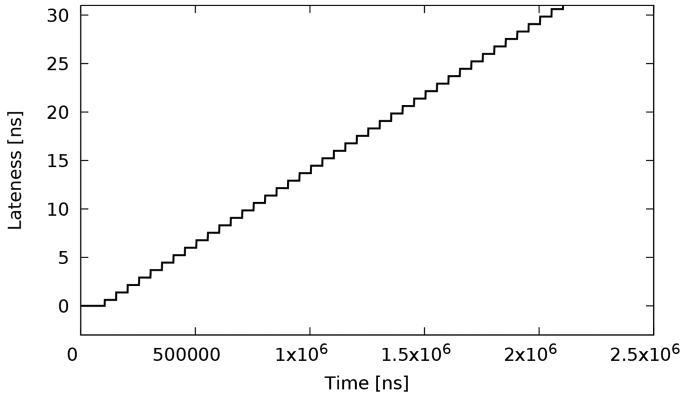

Figure 3.4 Maximum normalised task lateness step response for 500 tasks (with execution time equal to 50,000 ns) released at 5,000 ns (3 clusters, 3 cores in each).

The usage of the core utilisation in a cluster as a controlled value is a bit more tricky due to its non-linearity. Because of the obvious saturation at 100 per cent (see Figure 3.5) in the case of step response, to compute parameters of a controller we limit the considered operating region to the proportional range before the saturation, which ranges from 1 to 4 ns. Its maximum slope tangent can be described by linear formula $y = 0.33\,x - 0.33$. According to classic Ziegler-Nichols method [64], the k parameter of the P controller can be computed as

$$k = \frac{1}{\lambda}, \tag{3.7}$$

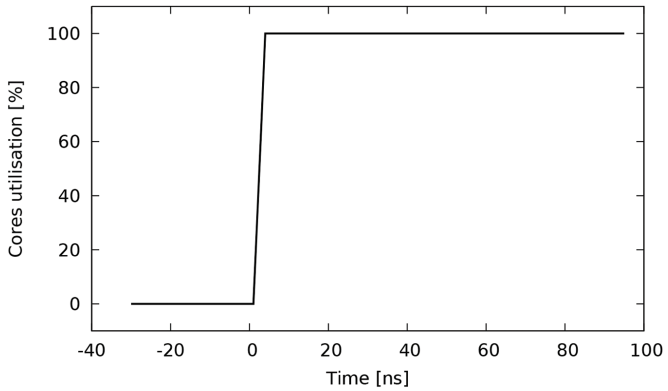

Figure 3.5 Cores utilisation step response for 500 tasks (with execution time equal to 50,000 ns) released at 0 ns (1 cluster with 3 cores).

where λ is the absolute value of the y-coordinate of the intersection of the max slope tangent with the OX axis. In our case $\lambda = 0.33$ and, consequently, $k = 3$.

3.3.2 Stress Tests

The workload used in our introductory experiment consists of 900 independent tasks, one released every 5,000 ns, whose computation time equal to 50,000 ns and deadline is set to the sum of computation time multiplied by 1.2 and the task release time. In Table 3.1, the number of rejected tasks, tasks executed before and after their deadlines in various controlling environment settings is presented.

The two first rows present the result obtained in the open-loop systems. The choice of the external queue has only slight influence on the number of tasks executed before their deadlines. In these situations task can be rejected by the admission control only if the task slack (computed as $D_i - C_i - t$) is negative.

Applying a closed-loop approach improves the system performance significantly, but the proper choice of the measured output is also essential. In case of the core utilisation not a single task finishes after its deadline, as the tasks are submitted to the queue only when there is at least one idle core, that can start executing the task instantly. The lateness is less correlated with the real temporal availability of computational power, so as many as 116 tasks

Table 3.1 Number of rejected tasks, tasks executed before and after their deadlines in various controlling environment configurations for a periodic task workload simulation scenario (3 clusters, 3 cores in each): configuration parameters (*above*) and obtained results (*below*)

Config. No.	Architecture	Queue	Controller Value	Controller	Allocation
1	Open-loop	FIFO	–	–	min core util.
2	Open-loop	EDF	–	–	min core util.
3	Closed-loop	Both	core utilisation	P	min. CPU util.
4	Closed-loop	Both	lateness	PI	min. CPU util.
5	Closed-loop	Both	core utilisation	P	RR
6	Closed-loop	Both	lateness	PI	RR

Config. No.	Tasks before Deadline	Tasks after Deadline	Tasks Rejected
1	149	661	90
2	154	655	91
3	738	0	162
4	614	116	170
5	675	0	225
6	607	127	166

have been sent to the queue despite not a single core was capable of computing the task before its deadline.

To asses the improvement of the core-utilisation-based allocation, we performed simulations where the tasks are allocated to clusters in a round-robin manner. In both P- and PI-based architectures we obtained worse results by 8.5 and 1.14 per cent, respectively. Importantly, the higher improvement has been observed in the architecture leading to the overall better results.

The clusters' cores utilisation for this architecture during the first 500 ns is presented in Figure 3.6. Except for the initialisation (and finalisation, not shown in the figure) there is no time when any of the clusters has less then 66 per cent of the core utilisation. After computing the average core utilisation during the whole simulation time we get 90.12%, 90.20%, and 90.16% for the first, second and the third core, respectively. The tasks have been sent by the dispatcher to the first cluster 296 times and to the 2nd and the 3rd core respectively 292 and 313 times, which can be viewed as a quite even distribution.

The control signal (generated by a controller, sent to the admission control) for the first 500 ns of the simulation is shown in Figure 3.7. The positive value of this signal means that at least one core from the given cluster has finished the previous task computation and is being idle. In this situation the next task should be submitted to the cluster as soon as possible.

Similar results have been observed in other workloads of a periodic nature with uniform (or nearly uniform) execution time.

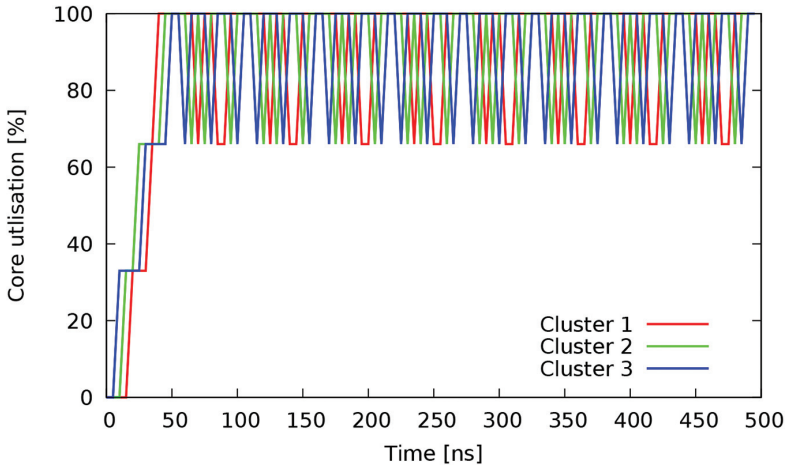

Figure 3.6 Core utilisation measured during the first 500 ns of the simulation.

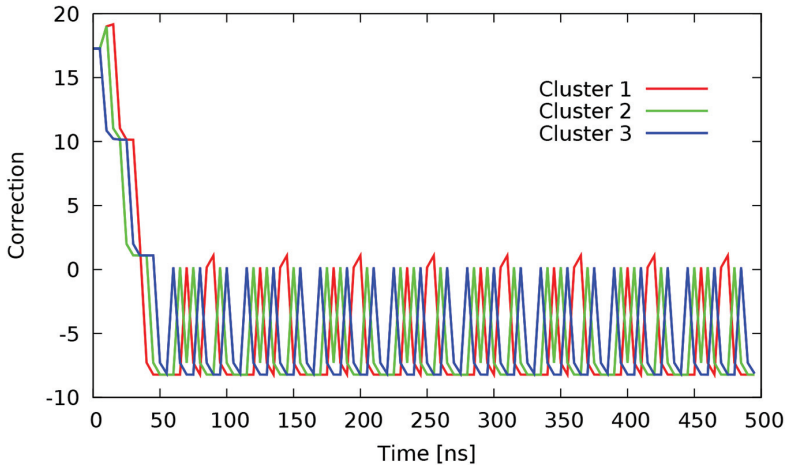

Figure 3.7 Control signal observed during the first 500 ns of the simulation.

3.3.3 Random Workloads

In our next experiment, summarised in Table 3.2, we analysed 30 randomly bursty workloads, generated according to the method described in [22], including from 827 to 962 tasks of diverse execution time, ranging from 1 to 2,67,582 ns. Three target system configurations have been checked: open-loop with EDF and closed-loop with CPU utilisation as the controller value, where the allocation is performed using the minimal core utilisation metric and in the round-robin way. Once again, the closed-loop approach leads to better results but, in comparison with the periodic-task scenario in the

Table 3.2 Total number of rejected tasks, tasks executed before and after their deadlines in various controlling environment configurations for 30 random bursty task workload simulation scenarios (3 clusters, 3 cores in each): configuration parameters (*above*) and obtained results (*below*)

Config. No.	Architecture	Queue	Controller Value	Controller	Allocation
1	Open-loop	EDF	–	–	min. core util.
2	Closed-loop	Both	core utilisation	P	min. core util.
3	Closed-loop	Both	core utilisation	P	RR

Config. No.	Tasks before Deadline	Tasks after Deadline	Tasks Rejected
1	10603	1752	14059
2	12296	753	13365
3	11946	675	13793

experiment described above, the improvement, equal to about 16%, is slightly less impressing. Similarly, the difference between allocating task under the minimal core utilisation criteria and round-robin is rather slight and equals 3 per cent. It is worth stressing, however, that the tasks in the analysed workloads are characterised with very diverse time of computations, but despite this variance they are not differentiated by our model. Consequently, one task can occupy a core for longer time, not allowing other (submitted a bit later) tasks to be executed on this core because of the lack of preemption.

3.4 Dynamic Voltage Frequency Scaling

Dynamic Voltage Frequency Scaling (DVFS) is a power saving technique, omnipresent in CMOS circuits, benefiting from the fact that their dynamic (or switching) power P is proportional to the the square of core supply voltage V, and its clock frequency f, i.e., $P \propto fV^2$. Since any reduction of core voltage requires an adequate decrease of the clock frequency, some trade-off between energy savings and computation performance is achieved. Some guidance in real-time systems stems from the fact that there is usually no additional benefits from faster task execution as long as it is before the deadline. Moreover, for typical workloads the required peak computational performance is usually much higher than the average [106]. Thus sustaining a lower voltage/frequency for most of time and increasing it only when required by a workload growth, in a way it risks missing some deadlines, seems to be a sensible strategy. To perform a proper voltage scaling decision, it is possible to rely on various metrics, provided by the monitoring infrastructure tools and services, such as utilization and time latency between input and output timestamps [81]. In multiprocessor domain, the cores can operate on different voltage at a given instant, so allocating a task to the most suitable core starts to be a more sophisticated task even in case of homogeneous cores, since assigning a task to a core with lower voltage can lead to missing the deadline that would be met in case of a different decision. The term voltage scheduling has been introduced to refer to scheduling policies using DVFS facility to improve energy efficiency.

In Multiprocessor Systems on Chips (MPSoCs) a task can be mapped to a core either statically or dynamically, just before its execution, which is particularly beneficial in case of workloads not known a priori [123]. In DVFS-based systems, the problem of dynamic task mapping is even more difficult, since not only resource utilisation and application structure have to be analysed, but also the present voltage level of each processor needs to be considered. Modern operating systems, including both Windows and Linux (2.6 Kernels and above) support dynamic frequency scaling for systems

with Intel (SpeedStep technology) and AMD (PowerNow! or Cool'n'Quiet technology) processors. Frequency levels in these chips are not continuously available, but a limited number of discrete voltage/frequency levels is offered. They follow the Advanced Configuration and Power Interface (ACPI) open standard, defining such processor states as C0 (operating state), C1 (halt), C2 (stop-clock), and C3 (sleep). In C states with higher numbers less energy is consumed, but returning to the normal operating state imposes more latency. In some device families additional C-states have been introduced, such as C6 in Intel Xeon when an idle core is power gated and its leakage is almost entirely reduced to zero [52]. While core is in the C0 state, it operates with one of several power-performance states, known as P-States. In P0, a core works with the highest frequency and voltage level, and subsequent P-States offer less performance but also require less energy. The most recent ACPI specification can be found at Unified Extensible Firmware Interface Forum[1].

In operating systems, frequency scaling depends on an applied governor. In case of Linux, the *ondemand* governor switches frequency to the highest value instantly in case of high load, whereas the *conservative* governor increases frequency gradually [77]. These policies aim to keep processor utilization close to 90%, progressively decreasing or increasing frequency using heuristics [102]. This approach may, however, negatively impact applications with timing constraints. To overcome this limitation, a *custom* governor can be developed and applied. These governors can then operate on per-core and per-chip basis, taking into account utilisation of other machines in a cluster, etc. A valuable comparison between per-core and per-chip DVFS is presented in [68], where per-core DVFS is shown to offer even more than 20% energy savings in comparison with the conventional chip-wide DVFS with off-chip regulators. However, per-core DVFS is rarely implemented and, for example, all active cores in contemporary Intel i7 processors must operate with the same frequency in the steady state, whereas AMD processors allows their cores work with different frequencies, but one voltage value, appropriate to the core with the highest frequency, is to be provided to all the cores [52].

In the remainng part of this chapter, we propose a custom governor algorithm for per-chip DVFS. The algorithm performs dynamic resource allocation and assumes the presence of a common task queue, which is used by a global dispatcher. The resource allocation process is executed on a particular processing unit, whose role is to send the processes to be executed to other processing units, putting them into the task queue of that core. The task dispatching, i.e., selecting the actual task to run, is also a part of the mapping algorithm and is carried out locally on each processor. It is assumed that task

[1] http://www.uefi.org

scheduling is performed in a non-preemptive first-in-first-out (FIFO) based manner for simplicity, but another scheduler can be used instead.

3.5 Applying Controllers to Steer DVFS

In Figure 3.8, a general view of the proposed architecture is presented, where dashed lines are used for steering P-States. We consider workflows of a particular structure. All tasks are assumed to be firm real-time, so certain number of missing deadlines is allowed, but the task executed after its deadline is invaluable to the user. There are no dependencies between tasks and all tasks have equal priorities. Further, tasks cannot be preempted during their execution.

After releasing task T_i, the role of the dispatcher is to decide which of the processors $Processor_j, j = 1, \ldots, m$, is to execute the task. This decision can be made using various metrics. We measure processor core utilisation, which is the percentage of busy cores in the processors executing tasks in particular simulation time t, and choose the processor whose cores are currently the least utilized. If more than one processor have the same lowest utilisation, one of them is chosen randomly. The task T_i is then placed in the j-th external queue.

To control the admittance in the j-th processor, we use a discrete-time PI controller (i.e., a discrete-time PID controller without the derivative component) whose output value $u_j(t)$ depends on the difference between the current core utilisation and the setpoint. The output value of $u_j(t)$ is sent

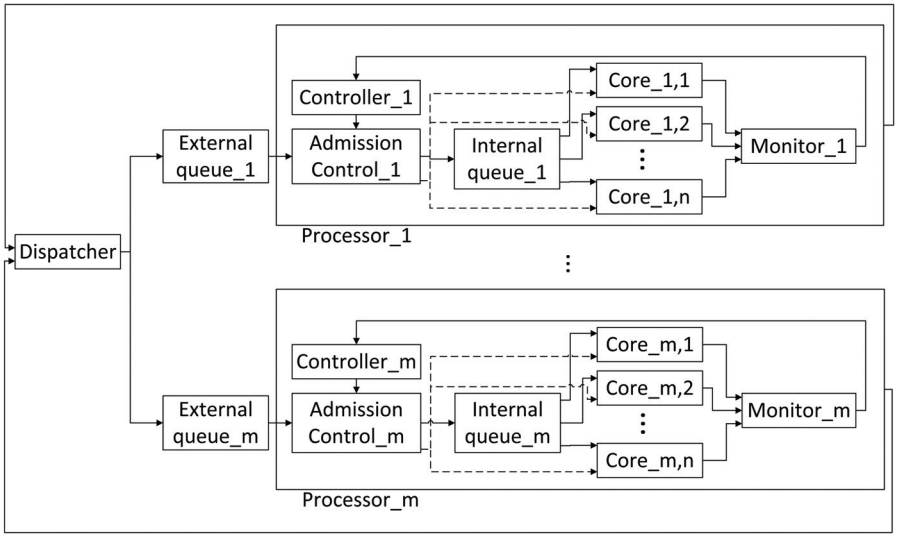

Figure 3.8 Distributed feedback control real-time allocation with DVFS architecture.

to admission control block AC_j, where it is used to perform a task admittance decision.

The role of block AC_j is to decide whether a task T_i, fetched from the j-th external input queue, should be executed by the processor. The first condition of admittance is that the deadline of T_i, D_i, is not lower than the sum of its worst-case computation time, C_i and the current simulation time t. Then the output controller value, $u_j(t)$, is checked and it influences the decision of the task rejection or admission as described in the next paragraph. The admitted tasks are placed in the internal processor queue. This queue shall be rather short to minimise the delay between decision about admittance and the execution of the task, and to keep the timeliness of the lateness input.

The additional role of block AC_j is to scale the voltage of the cores. The controller output value, $u_j(t)$, is tested against two threshold values $+\Upsilon$ and $-\Upsilon$. If $u_j(t) > +\Upsilon$, the processing cores are more utilised than the setpoint r for relatively long period (depending on the I-Window length and k_i value) and thus increasing the frequency (and voltage) of the set of cores is desirable. On the other hand, if $u_j(t) < -\Upsilon$, the processing cores are too idle for relatively long period and it is recommended to decrease the frequency (and voltage) of the cores to conserve energy. It is important to select the value of Υ wisely, taking into account that $u_j(t)$ depends on the current error value (multiplied by k_p) and on the sum of the previous errors (multiplied by k_i) and the length of I-Window used during this sum calculation. After choosing these three values, it is possible to assign an appropriate value to this threshold. Identification of these values and the threshold is performed in Section 3.3.

Since in any core transferring between various voltage levels is penalised both in terms of switching time and energy [52], some mechanism preventing too frequent transitions is needed. In our case, we decided to use threshold Γ, which determines the minimal time between two consecutive voltage level alterations. Each P-State change request issued earlier than Γ is ignored. This value should be determined by taking into account the hardware parameters as a trade-off between the system flexibility (lower parameter value) and efficiency (higher parameter value), which is presented in Section 3.3.

The proposed admission control algorithm is composed in two parts, described respectively by lines 1–28 and 29–36 in Figure 3.9, which are executed concurrently. The first part consists of the following steps.

Step 1. Invocation and initialization (lines 1–3, 27): The block functionality is executed in an infinite loop (line 1), activated every time interval Δt (line 27). The current P-State is set to the lowest value (i.e., the highest performance – line 2), and the time of the previous P-State change, γ, is set to 0 (line 3).

Step 2. Task fetching and schedulability analysis (lines 4–5): The tasks input FIFO queue is checked if empty (line 4) and a task T_i is fetched (line 5).

Inputs: Task T_i (from Task queue)
Controller output value u
Admission controller invocation periods Δt
Outputs: Task executing or rejection decision
New P-State of Cores
Constants: P_{max} - maximal P-State available in processor
Υ - threshold value of cumulated error from controller
Γ - minimal time elapsed between P-State change
Variables: P - current P-State
γ - time of the previous P-State change

```
1:  while (true) do
2:      P = 0
3:      γ = 0
4:      while (task queue is not empty) do
5:          Fetch T_i
6:          if (P = 0 and u < 0)
            or (u < 0 and current_time ≤ γ + Γ) then
7:              if  P > 0 and current_time > γ + Γ then
8:                  P = P − 1
9:                  γ = current_time
10:                 Clear I-Window in Controller
11:             end if
12:             Reject task T_i
13:         else
14:             if u < −Υ and P > 0
                and current_time > γ + Γ then
15:                 P = P − 1
16:                 γ = current_time
17:                 Clear I-Window in Controller
18:             else if  u > +Υ and P < P_max
                and current_time > γ + Γ then
19:                 P = P + 1
20:                 γ = current_time
21:                 Clear I-Window in Controller
22:             end if
23:             Admit task T_i
24:             Send T_i to FIFO
25:         end if
26:     end while
27:     Wait Δt
28: end while

29: while (true) do
30:     if (task queue is empty for Γ and P < P_max) then
31:         P = P + 1
32:         Clear I-Window in Controller
33:         γ = current_time
34:     end if
35:     Wait Δt
36: end while
```

Figure 3.9 Pseudo-code of the proposed admission controller functionality.

Step 3. Task conditional rejection (lines 6–12): If the output value of controller (u) is negative and the cores operate with the highest performance

(P-State set to P0) or the cores operate below the highest performance and the previous change of P-State (at time γ) was done early enough (determined by condition $current\_time > \gamma + \Gamma$), task T_i should be rejected (line 6). Moreover, if P-State is different from P0 and there was no recent change of P-State (line 7), P-State is decreased (line 8) and variable keeping the previous P-State change time, γ, is set accordingly (line 9). The buffer storing the previous error values of the controller, I-Window, used by the integral component of the PID controller, is cleared (line 10), since the previous errors have been obtained in a different P-State and thus should not influence future admittance decisions.

Step 4. Task conditional admittance (lines 14–25): If the controller output value is below threshold $-\Upsilon$, the processor performance is not the highest possible and the previous change of P-State was done early enough (line 14), P-State is decreased (line 15) and the $current\_time$ is substituted to γ (line 16). Similarly, provided the controller output value is above threshold $+\Upsilon$, the processor performance is not the lowest possible (P-State is different from P_{max}, the highest P-State available in the processor) and the previous change of P-State was done early enough (line 18), the processor P-State is increased (line 19) and the current time is assigned to γ (line 20). Task T_i is sent to the FIFO queue (line 24).

The second part of the algorithm consists of the following steps.

Step 1. Invocation (lines 29, 35): The block functionality is executed in an infinite loop (line 29), activated every time interval Δt (line 35).

Step 2. P-State conditional increase (lines 30–33): If no new tasks have been fetched from the task queue for time Γ and the processor performance is not the lowest possible (P-State is different from P_{max}), the processor P-State is increased (line 31) and the current time is assigned to γ (line 33).

3.6 Experimental Results

In order to check the efficiency of the proposed feedback-based admission control and real-time task allocation, we developed a simulation model using SystemC language.

3.6.1 Controller Tuning

Firstly, the controller constant components k_p, k_i and k_d have to be tuned by analysing the corresponding open-loop system response to a bursty workload. Then random workloads of various weight have been tested to observe the system behaviour under different conditions and to find the most beneficial operating region.

To tune the parameters of the controller, the task slack growth after applying a step-input in the open-loop system (i.e., without any feedback) has been analysed, as mentioned earlier in this chapter. This is a typical way in control-theory-based approaches [7]. The workload used for this case consists of 500 independent tasks. They are split into five groups. Tasks belonging to one group are released every 5 ms each. After this bursty activity, during the following 500 ms no task is released. Then the tasks of the next group are released at the same pace. This process is repeated until all tasks from all groups are released. The computation time of each task is equal to 50 ms and its deadline is set to the sum of computation time multiplied by an arbitrary constant (equal to 1.5) and the task release time. This constant has been introduced to provide some flexibility in task scheduling; otherwise, all tasks would be required to start execution at their release time to meet the timing constraints.

The obtained results have confirmed the accumulating (integrating) nature of the process, and thus the accumulating process version of Approximate M-constraint Integral Gain Optimization (AMIGO) tuning formulas have been applied to choose the proper values of the PID controller components [7]. As a reference point, we executed a simulation without DVFS on a system comprised of one processor with three cores. As many as 140 tasks have been executed before the deadline, no task missed its deadline, and 360 tasks have been rejected by the dispatcher.

To use the DVFS features efficiently, it is crucial to find an appropriate value of threshold Υ. It should be large enough not to switch a core voltage too frequently – the switching should be performed not only due to the high value of $u(t)$ generated by the proportional component, but also with the relatively large value of the integral component, meaning that the error has been large for a longer interval. Simulations results for selected values of Υ are presented in Table 3.3. Form this table it follows that too high values of Υ result in keeping the current frequency too long at the beginning of a busy period, decreasing the performance of the system significantly (see the number of tasks executed before the deadline for $\Upsilon \in (30, 60)$). Particularly, keeping the lowest frequency too long results in executing some tasks after their deadlines. At some point ($\Upsilon = 70$ in the considered case), the threshold is too high for the given idle period and it does not manage to perform any voltage scaling before the next busy period. For further experiments, based on the above observations, we have chosen $\Upsilon = 10$ as a trade-off between performance and flexibility of the voltage switching.

In order to determine the threshold Γ of the task number that has to be processed by the dispatcher between subsequent alterations of the core voltage, we performed a series of simulation, where the threshold ranged from 25 ms to 400 ms. The highest number of tasks executed before their deadlines is

Table 3.3 Total number of tasks executed before and after their deadlines and rejected tasks with various Υ threshold in the introductory experiment (1 processor with 3 cores)

Threshold Υ	Tasks before Deadline	Tasks after Deadline	Tasks Rejected
5	120	0	380
10	120	0	380
20	120	0	380
30	64	84	352
40	52	92	356
50	52	92	356
60	52	92	356
70	140	0	360
∞	140	0	360

observed with threshold $\Gamma \in \{25 \text{ ms}, 50 \text{ ms}, 100 \text{ ms}\}$. The threshold set to 350 ms or above leads to the behaviour not differentiated from the simulation without DVFS. Two values $\Gamma = 50$ ms and $\Gamma = 300$ ms have been used in the further experiments.

To estimate the energy used by a processor, ACPI data for Pentium M processor (with Intel SpeedStep Technology) has been used, but with slight modification the proposed technique can be applied to any processor with ACPI implemented[2]. In Pentium M, there are six levels of allowed frequency and voltage pairs, known as P-States. In P-State P0, a core works with 1.6 GHz and 1.484 V, whereas for P5 – 600 MHz and 0.956 V, which uses 24.5 W and 6 W, respectively.

3.6.2 Random Workloads

Having selected all the required constants, the efficiency of the system has been checked against 11 sets of 10 random workloads, whose release and execution time probability distributions are based on the grid workload of an engineering design department of a large aircraft manufacturer, as described in [22]. Each workload is comprised of 100 tasks, including a random number (between 1 and 20) of independent jobs. The execution time of every job is selected randomly between 1 and 99 ms. All jobs of a task are released at the same instant, and the release time of the next task is selected randomly between $r_i + range\_min \cdot C_i$ and $r_i + range\_max \cdot C_i$, where C_i is the total worst case computation time of the current tasks T_i released at r_i, and $range\_min, range\_max \in (0, 1), range\_min < range\_max$. These values are inversely proportional to the workload weight.

[2]For example AMD Family 16h Series Processors ACPI parameters are provided in AMD Family 16h Models 00h – 0Fh Processor Power and Thermal Data Sheet, AMD, 2013.

We have measured the numbers of tasks computed before their deadlines and the number of tasks rejected by the admission controller block in a 3-processor system with 3 processing cores and 2-processor system with 4 processing cores each for systems with DVFS and without DVFS (i.e., with $\Gamma = \infty$) and presented it in Figures 3.10 and 3.11, respectively.

The number of tasks admitted with $\Gamma = 300$ ms is, in total, 26% higher than with $\Gamma = 50$ ms. The reason for this is that in case of $\Gamma = 50$ ms, P-States are changed more often and thus it is more likely to have a processor with a lower frequency and voltage level while a task is fetched, and since decrease of P-States is performed gradually (lines 8, 15, 19 and 31 in the algorithm in Figure 3.9), tasks are attempted to be executed with lower processor

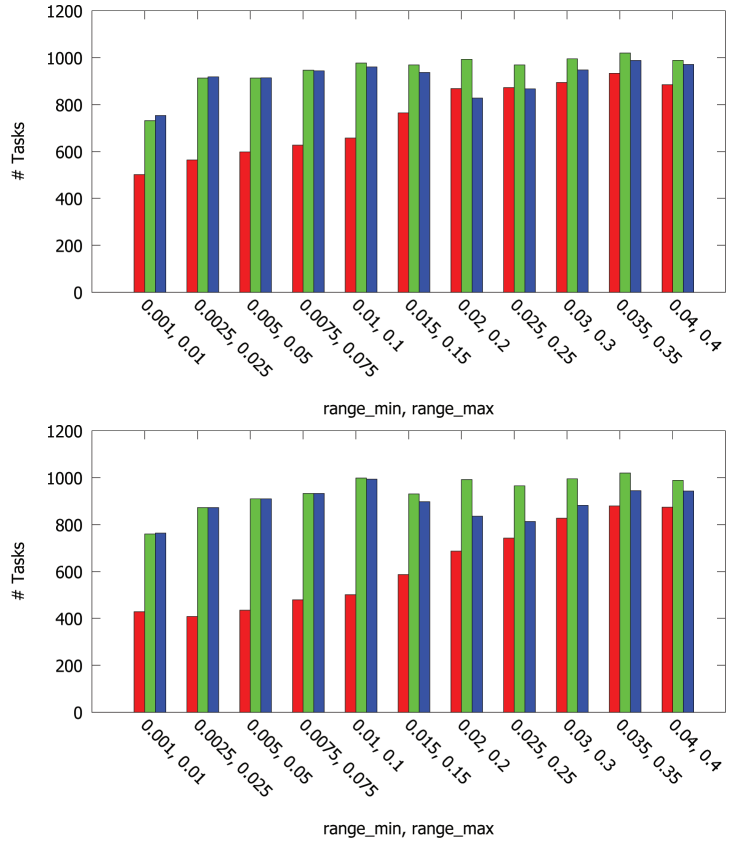

Figure 3.10 Tasks executed before their deadline in random workload scenarios for DVFS with $\Gamma = 50$ ms (red), $\Gamma = 300$ ms (green) and $\Gamma = \infty$ (blue) – three processors with three cores (*top*) and two processors with four cores (*bottom*) systems.

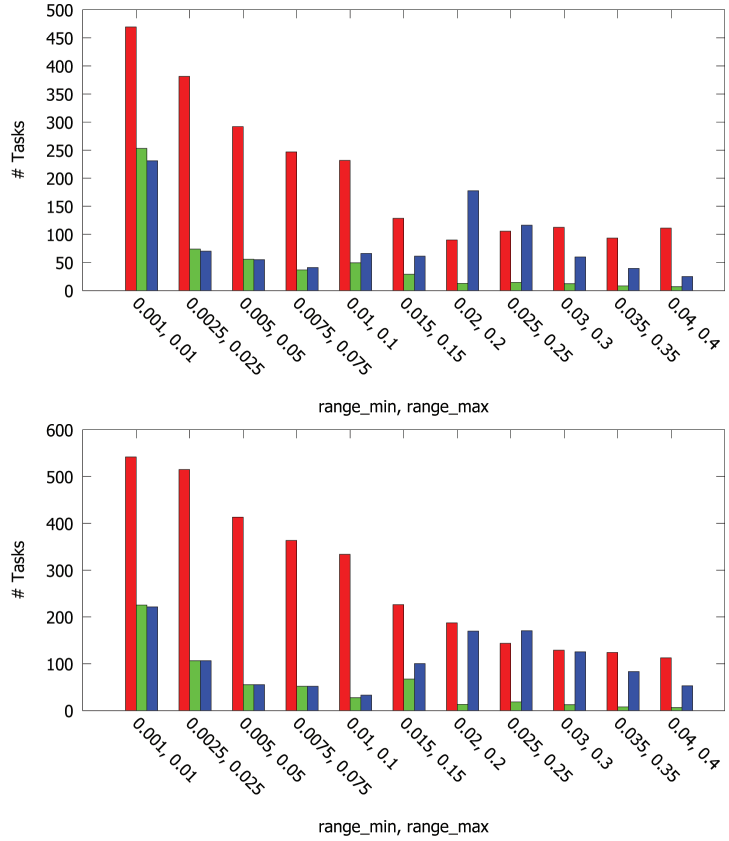

Figure 3.11 Tasks rejected in random workload scenarios for DVFS with $\Gamma = 50$ ms (red), $\Gamma = 300$ ms (green) and $\Gamma = \infty$ (blue) – three processors with three cores (*top*) and two processors with four cores (*bottom*) systems.

performance. It has been observed that this strategy leads to significant (about 39%) energy reduction. It may be, however, surprising, that the number of the executed tasks is higher with DVFS when $\Gamma = 300$ ms than in the system without DVFS for lighter workloads. This phenomena is innate to the proposed technique and can be explained using the pseudo-code in Figure 3.9. In a system without DVFS, each processor is always in its lowest P-State, P0. The admission controller has then no flexibility in decreasing the P-State while the Controller output is negative (checked in line 6) and then to clean the I-Window in the PID controller and, finally, admit the task (line 21).

Different size of the systems does not influence the relationship between obtained results. The number of tasks executed before deadlines for assorted weights is almost linearly dependent between the two considered architectures

(Pearson Correlation Coefficient $\rho = 0.96$; similarly for the number of rejected tasks $\rho = 0.97$, and for dissipated energy $\rho = 0.98$).

The dynamic energy dissipation, normalised with respect to the highest obtained value during the experiment, is presented in Figure 3.12. In general, almost 58% of the dissipated dynamic energy have been saved via the DVFS approach. For heavier loads from the random workloads, choosing a lower Γ value leads to significant energy reduction, whereas for lighter loads the result difference between the two chosen Γ values is almost negligible.

Looking at the normalized energy dissipation per task (Figure 3.13), computed for the system with three processors, it can be concluded that parameter $\Gamma = 50$ ms leads to more even energy per task usage in comparison with $\Gamma = 300$ ms, which is slightly more beneficial for lighter workloads only,

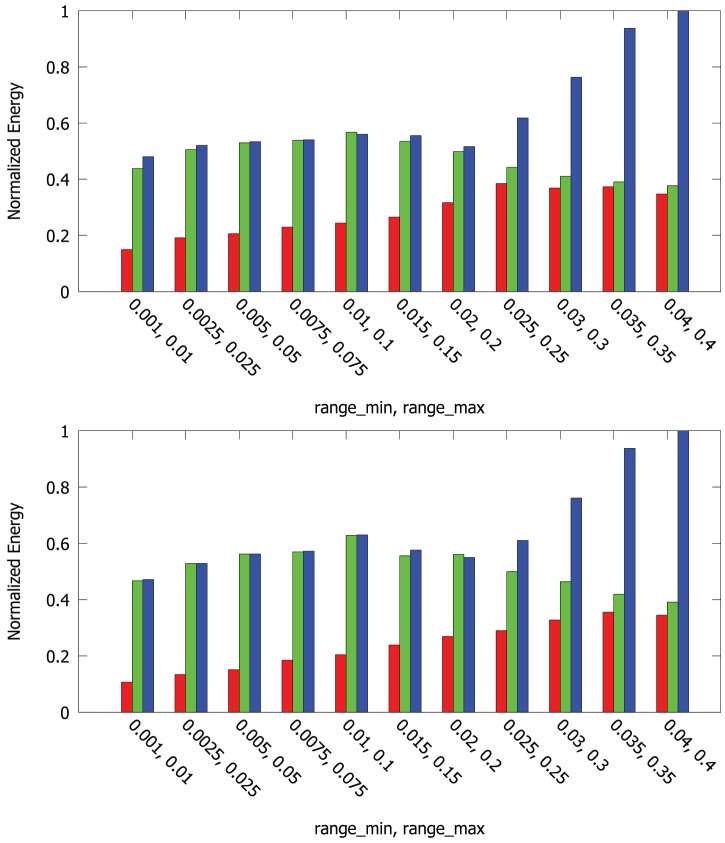

Figure 3.12 Normalized dynamic energy dissipation in random workload scenarios for DVFS with $\Gamma = 50$ ms (red), $\Gamma = 300$ ms (green) and $\Gamma = \infty$ (blue) – three processors with three cores (*top*) and two processors with four cores (*bottom*) systems.

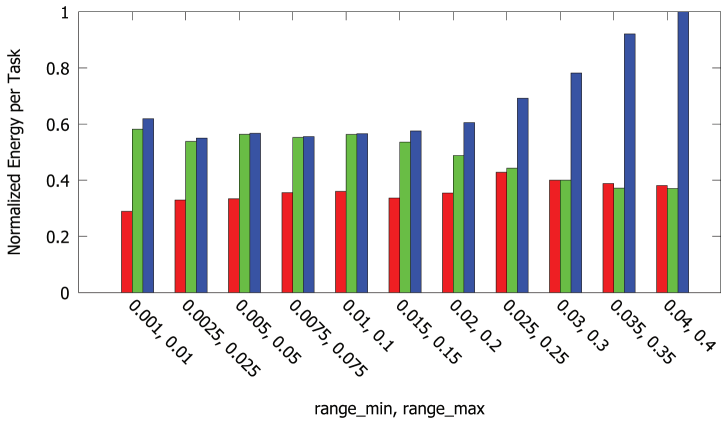

Figure 3.13 Normalized dynamic energy dissipation per task meeting its deadline in random workload scenarios for DVFS with $\Gamma = 50$ ms (red), $\Gamma = 300$ ms (green) and $\Gamma = \infty$ (blue).

but leads to similar energy per task value than in the system without DVFS (i.e., $\Gamma = \infty$) for heavier loads.

3.7 Related Work

Dynamic real-time scheduling (RTS) algorithms like EDF and rate monotonic support RTS characteristics (worst-case computation time, release rate, etc.), but they remain open-loop: once the schedule is built, it stays fixed [80]. Such algorithms perform well in meeting QoS levels in predictable systems. However, their performance degrade when dealing with unpredictable workloads which cannot be modelled accurately a priori [2]. In unpredictable systems, sophisticated schedulers like Spring depends on worst-case parameters which can result to resource under-utilisation based on pessimistic estimation of workloads [135].

It is desired to feed the system states back to the scheduler [26, 121], so it can be aware of sudden/unpredicted changes and act accordingly in order to meet the QoS levels. A system state can be defined as the system performance with respect to service capacity, QoS levels etc. Real-time systems analysis includes observing how tasks' RTS characteristics affect (1) meeting QoS levels e.g., high processor utilisation, and (2) compute resource availability. This can help adapting towards the varying system states by enforcing scheduling decisions [26]. This variance can be considered as system error which is the deviance from the system output and the desired one (QoS level(s)). Some related work like [26, 78] show the lack of adaptivity to varying system states in traditional RTS algorithms.

In the realm of control-theoretic RTS algorithms, there are adaptive approaches that are cost-effective for performance guarantees in systems with varying system states [26, 80]. For instance, Lu offers regulating the workload of a single-CPU RTS system via a PID-based admission control (PID-AC) algorithm to reduce the deadline miss ratio [80]. His algorithm guaranteed 95% CPU utilisation and 1% deadline miss ratio in comparison to an EDF algorithm with 100% and 52% respectively. PID-AC algorithms are plausible in some RTS systems where controlling tasks release rate is difficult, instead, the scheduler rejects specific tasks to meet QoS levels.

Also, in [81], Lu uses two PID controllers to meet two QoS levels; maximum CPU utilisation and minimum deadline miss ratio. The results confirm the findings of [80] in ensuring performance guarantees as opposed to basic EDF algorithm. The motivation behind Lu's 2-PID algorithm was to address the stability and transient response issues with the single PID algorithm due to PID control limitations handling multiple QoS levels. Our work, in this chapter, addresses minimising dependent tasks' latency, not deadline miss ratio, via a PID-based admission control algorithm.

Control-theory-based voltage level selection of unicore portable devices has been firstly proposed in [106]. Varma et al. in [147] choose Proportional-Integral-Derivative (PID) controllers to determine voltage of systems dealing with the workload not accurately known in advance and interpreted the meaning of the discrete PID equation terms with regards to dynamic workloads. They proposed a heuristic to predict the future system load based on the workload rate of change, leading to significant energy reduction. They also demonstrated that the design space is not particularly sensitive to changes in the PID controller parameters. However, the controller is used to predict the future workload and does not use any feedback information from the system about the processing core status.

In [162], a feedback-based approach to manage dynamic slack time for conserving energy in real-time systems has been proposed. A PID controller is used to predict a real execution time of a task, usually lower than its worst case execution time (WCET). Then each execution time slot for a task is split into two parts and the first part is executed with a lower voltage assuming the execution time predicted by the controller. If the task does not finish by this time, a core is switched to its highest voltage guaranteing that the task finishes its execution before its deadline.

In [153] a formal analytic approach for DVFS dedicated to multiple clock domain processors benefits from the fact that the frequency and voltage in each functional block or domain can be chosen independently. A multiple clock domain processor is modelled as a queue-domain network where queue occupancies linking two clock domains are treated as feedback signals.

A clock domain frequency is adapted to any workload changes by a proportional-integral (PI) controller.

The queue occupancy also drives PI controllers in [31]. In contrast to previous research, the limitation of using single-input queues only have been lessened and multiple processing stages have been allowed, but a pipelined configuration is still required. A realistic cycle-accurate, energy-aware, multiprocessor virtual platform is used for demonstrating the superiority of feedback techniques over the local DVFS policies during simulation of signal processing streaming applications with soft real-time guarantees. It is assumed that as long as the queues are not empty, a sufficient number of deadlines is met and no further analysis or simulation of deadline misses are provided.

From the literature survey it follows that there is no previous work on mapping the task dynamically to an MPSoC system and using DVFS together with control-theory based algorithm.

3.8 Summary

In this chapter, we have explored the possibility of applying feedback controlled values to dynamic task allocation and admission control for high-performance computing clusters executing real-time tasks. Two real-time metrics, task lateness and core utilisation, have been applied to perform admission control, whereas the former has been also used as a metric for dynamic task allocation. Two queue types (EDF and FIFO) have been used in the open-loop system. The P and PI controllers have been tuned using classic AMIGO and Ziegler-Nichols methods.

We have prepared simulation models in SystemC language and performed a number of experiments. In case of uniform periodic workload the closed-loop system has executed almost 5 times more tasks before their deadline in comparison with an adequate open-loop system. The queue type and controller value have slightly influenced the outcome. The metric-based dynamic allocation, in the best configuration, has been about 8.5 per cent better than the round-robin method.

In the case of bursty random workloads with large computation time variance, a closed-loop-based system has been about 16% better than the corresponding open-loop approach. However, to proper asses the difference between these systems in more accurate way, the differentiate between tasks of long and short execution time should be introduced.

We have also explored the possibility of applying feedback control values to dynamic task allocation and admission control for multi-core processors supporting DVFS while executing firm real-time tasks. Core utilisation has been applied as a run-time metric to perform admission control and dynamic

task allocation. The proposed governor algorithm has been tested with various parameter values and some guidance for tuning has been provided.

Even in case of relatively difficult bursty scenarios a significant power reduction has been obtained in exchange for executing lower number of tasks before their deadlines. It is a role of a system designer to choose proper parameter values to obtain a satisfiable trade-off between energy consumption and performance. The proposed approach leads to similar results in two considered systems of different sizes, thus it may be viewed as quite robust to different system configurations.

The minimal interval allowed between two consecutive switchings of P-States (threshold Γ) influences workloads of various weights in different, but predictable way. An adaptive choice of Γ value can be then viewed as a simple yet effective improvement of the proposed technique.