Keep It Slow and in Time: Online DVFS with Hard Real-Time Workloads

Kai Lampka and Björn Forsberg
Department of Information Technology, Uppsala University, Sweden
Email: kai.lampka@it.uu.se, bjfo5755@student.uu.se

***Abstract*—To handle hot spots or power shortages, modern multicore processors are equipped with a supervisory dynamic thermal and power management (DTPM) system. When necessary, the DTPM system autonomously adapts the capacity of the cooling system or throttles the speed of core-local clocks via dynamic voltage and frequency scaling (DVFS) techniques. Opposed to best-effort scenarios, online DVFS with real-time workloads also needs to consider completion times of computations. Whereas execution times can be bounded adequately with worst-case estimates, arrival times of computation requests are potentially unknown. A deadline for completing a computation can easily be missed, if workloads suddenly peak and past clock speed assignments have built-up a non-negligible backlog of computations. To overcome this problem, we introduce an online DVFS management scheme which is history-aware. It operates a core at higher speed levels only if the future workload has the potential to result in timing violations, if not anticipated by rising clock speed assignments. We present an implementation of the scheme running on the Gem5 hardware simulator.**

I. Introduction

Motivation: Advances in electronics will bring about cost-effective multicore micro-controllers where the densities of transistors may give rise to hot spots or dark silicon, if executing at high clock speeds. Hot spots on a processor stem from the circumstance that not all components of a processor can be sufficiently cooled whenever the processor executes beyond its thermal design power.

To manage such behaviour, the processors of today are already equipped with dynamic thermal and power management (DTPM) systems. Through dynamic voltage and frequency scaling (DVFS), it can be ensured that a processor's heat production or power consumption is within the accepted limits. However, the performance loss due to autonomous throttling of clock speeds appears sporadic and is very difficult to predict.

For this reason, we argue that cores running hard real-time workloads, i. e., applications the execution of which must adhere to set deadlines, call for a mechanism to explicitly control their execution speeds. This mechanism needs to find clock speed assignments which on the one hand are as low as possible, but on the other hand guarantee that timing violations, i. e., deadline misses, are strictly ruled out.

Contribution: Most work in the direction of DVFS and hard real-time workloads rely on deterministic modelling of arrival times of computation demands, e. g., periodic real-time event models. When the workload express bursty, irregular or imprecise patterns, these approaches can either not be applied or make severe overapproximations about occurrences of computation demands.

For addressing workloads with a high degree of non-determinism in the arrival times of computation demands, this paper

1) introduces a scheme for computing safe processor speed assignments, where we invent the concept of a **worst-case ready queue (WCRQ)** (Sec. IV)
2) presents an implementation of the scheme on top of the HW-simulator Gem5 [1] (Sec. V)
3) evaluates the scheme by carrying out a simulation study with randomly generated traces of computation demands. The complete setup can be freely obtained from the authors.

To the best of our knowledge, this paper is the first one to *combine trace-less workload monitoring of hard real-time workloads with online DVFS processor management.* This is motivated by the desire to reduce the number of interventions of a processor's DTPM system at run-time, but not to sacrifice the timing correctness of the deployed hard real-time applications.

II. Related Work

When it comes to energy minimisation or avoidance of hot spots several strategies exist, e. g., [18] proposes task migration. Temporarily unused part of the processor infrastructure can be switched off to save energy or improve cooling efficiency. In contrast, this paper considers DVFS to reduce the energy consumption or heat emission of a multicore processor. This is particularly of use if the penalty of process migration is too high or target cores might be automatically powered down by the DTPM system.

In the past, a wide range of DVFS techniques have been proposed. The authors of [3] present a power management algorithm and simulation framework. The voltage/frequency level is derived from the power and performance sensors of the processor. Power modes are chosen in a greedy fashion, namely such that the number of executed instructions is maximized (computation throughput), but the periodic, per-core, power budgets hold.

The authors of [6] argue that the readings of processor sensors is not sufficient to characterize the workload of a core. Instead, the authors propose to decompose workloads into sequences of power modes by estimating a workload function from processor state readings. This leads to better

978-3-9815370-6-2/DATE16/

energy reductions and ensures that performance requirements are still met, e. g., computation throughput.

The above work elaborates DVFS techniques in the setting of best-effort applications, i. e., in settings where the throughput needs to be maximized, but no strict deadlines need to hold. The authors of [17] took hard real-time systems to DVFS-capable architectures. Many have since followed, but like the original, most of the work guarantees timing correctness only for workloads with a high degree of determinism in the arrival pattern of the computation demands. E.g., the authors of [7] derive an energy optimal sequence of power modes by formulating a Mixed Integer Linear Programme which requires known arrival times of computation demands.

Non-determinism in the arrival pattern of computation demands calls for online schemes which do not perform optimal w. r. t. energy consumption or heat production, but, perform reasonably well and still guarantee absence of timing violations. Examples of such online DVFS schemes in the context of hard real-time workloads are [5] and [10].

The authors of [5] introduce a threshold speed. Whenever the online speed computation hit the threshold speed, [5] proposed to set the device to the maximum speed instead. By statically setting the threshold speed sufficiently low enough, the authors rule out deadline misses. A method for computing the threshold speed was presented in [12]. We advance over this line of work as the presented scheme is aware of the arrival history of the workload.

Maxjagine et al. [10] monitor the arrival of the workload and compare is to their upper bounding function provided at design time. This requires recording of actual arrival times and requests the use of the resulting arrival trace in a series of min- and max-plus (de)convolution operations. On the positive side, this scheme is history-aware and therefore less pessimistic when computing speed assignments for queue jobs. As drawback, the scheme suffers from the following shortcomings: (a) once significant behaviour, i.e., a burst of computation demands drops out of the buffer of arrival time recordings, forecasting of future computation demands becomes extremely pessimistic. (b) Min and Max-plus operations impose a significant computational overhead when deriving the history-aware bounding function.

These shortcoming can be overcome by trace-less run-time monitoring techniques [9], [4], [11]. However, instead of explicit recording of the arrival times of computation demands and the use of complex min- and max-plus algebra operations, these methods track task activations w. r. t. predefined (complex) arrival models. E.g., Lampka et al. [9] uses a series of linear, i. e., leaky bucket, traffic shapers. This work builds upon this idea.

The aforementioned work either relies on explicit event recordings or strictly periodic workload arrivals to bound future workload arrivals. The trace-less run-time monitoring techniques [9], [4], [11] overcame this obstacle, however, the different authors did not propose algorithms for the online computation of processor speed assignments. To the best of our knowledge, it is left to this paper to combine trace-less monitoring of hard real-time workloads with DVFS and propose the required algorithms.

III. System Model

A. Processor and DVFS Model

This paper considers a set of K abstract processing units or cores. Each of them executes a fixed set of applications. Let m be a voltage/frequency level and let the processing speed of a core be measured in processing cycles available per unit of time at this frequency, e. g., ($\frac{cycle^m}{\mu sec}$). The supply bound function $sbf_m(\Delta)$ defines the minimum number of processing cycles which are available during any time interval of length Δ and with the core running at clock speed s^m.

We normalize all execution times to the maximum clock speed $\frac{cycle^{max}}{\textit{unit of time}}$ of the processor such that the maximum clock speed level s^{max} is set to 1. Any other speed level s^m is defined by $s^m = (\frac{cycle^m}{cycle^{max}})$. It follows that any clock speed specific supply bound function (SBF) is defined as $sbf_m(\Delta) = s^m \cdot \Delta$.

The reason for normalizing with regard to the maximum clock speed is as follows: The execution time of the CPU-bound parts of an application are proportional to the clock frequency, however, the memory bound parts are not. The reason for this is that, if $s^l < s^h$, the memory latency (in cycles) at speed s^l is lower than at speed s^h, since the off-chip memory operates at a constant frequency. Therefore, the maximum number of clock cycles used during the execution of a task is when it is executed at the maximum speed [8].

For lower clock speeds, one may obtain a decreased worst-case bound on the number of processing cycles to be consumed by a task τ_i. When lowering the clock speed by a factor of $k \leq 1$, the experienced prolongation of the WCET C of a task is, because of the lower number of cycles experienced while waiting for memory, less than $\frac{1}{k}$. For this reason, the assumed $sbf_m(\Delta) = s^m \cdot \Delta$ provides truly a lower bound on the available processing capacity with the core running at speed $s^m \times cycle^{max}$ per unit of time. This way, we guarantee that the worst-case execution times are conservative, i. e., are truly bounded by $\frac{1}{s^m} \times C$.

B. Workload model

It is assumed that on a core under speed management there are N sporadic real-time applications, termed as hard real-time tasks. The number of task activations per interval of time is bounded from above by a subadditive function known as arrival curve [16]. In the following we define the different ingredients of the assumed workload model.

1) Real-time task model: This paper considers infinite sequences of executions of hard real-time tasks. When a task τ_i is activated, we say that a job $j_{i,k}$ has been released, i. e., this job is the k'th invocation of the application modelled as task τ_i. In case the processing of a released job $j_{i,k}$ has not been completed, we say that the job (or task) is pending.

A task τ_i is defined by a 4-tuple $(C_i, D_i, B_i, \alpha_i)$, its components have the following meaning.

a) Normalized worst-case execution time: C_i represents the worst-case execution time (WCET) normed w. r. t. s^{max}.

b) Real-time requirements D_i and B_i: D_i is a task's relative deadline, i. e., it is required that upon any activation of τ_i at time t the execution of the respective instance has been completely processed before time $d_{i,k} = t + D_i$. $d_{i,k}$ is labeled absolute deadline of the k'th job released by task τ_i.

B_i is the bound on pending task activations of task τ_i, it specifies the maximum size of the input buffer allocated for task τ_i. For preventing buffer overflows, it is required that at any time t at most B_i jobs of task τ_i are pending.

c) Task activation bound α_i: For covering a wide range of job release patterns, this work assumes that the job releases of a task τ_i are specified by a task-specific release bound function α_i, more details follow below.

2) Modelling of task activation patterns: Activation pattern modelling should feature non-determinism in the activation timing, as the concrete points in time where jobs are released are potentially unknown. Addressing this issue, different models have been proposed in the literature. Modelling of task activations ranges from strictly periodic job releases and sporadic task activations with minimum inter-arrival distance, over periodic job releases with jitter and minimal release distances (PJD-model) to the bounding curves of Real-time Calculus [16]. PJD, as well as RTC-based activation pattern modelling, feature synchronous job releases of a task, whereas the periodic and sporadic activation pattern only allow a single job release per task and instant of time.

This work exploits the view inherent to RTC as it is sufficiently expressive and leaves room for underspecification, i. e., abstracts over exact task activation times. Sufficiently expressive means that standard activation patterns can be transformed into this model, e. g., the conversion of PJD to RTC arrival curves is discussed in [13].

Using RTC arrival curves, an activation of task τ_i is bounded from above by a subadditive curve $\alpha_i : \mathbb{R}^+ \to \mathbb{N}_0$ for which the following relation holds:

$$s, t \in \mathbb{R}_+ : 0 \leq s \leq t : \ \alpha_i(t-s) \geq R_i(t-s)$$

with $R_i(t-s)$ as the cumulative counting function which reports the number of activations of task τ_i in the interval $[s, t]$. For $s > t$, we define $\alpha_i(t-s) = R_i(t-s) = 0$.

3) Core-local demand bound function: The demand bound function (DBF) is the largest cumulative processing time required by all pending jobs of a task which have both release times and deadlines within a contiguous interval of Δ time units [2].

From each task-specific activation bound α_i, we derive the task-specific DBF as $dbf_i(\Delta) = (max(\alpha_i(\Delta - D_i), \alpha_i(\Delta) - B_i) \times C_i$, where $\alpha_i(\Delta - D_i)$ guards the reaching of all deadlines and $\alpha_i(\Delta) - B_i$ reflects that there might be input buffer constraints which limits the number of pending task activations.

By summing up the task-specific demand bounds and adding the residual worst case execution times c_i of the jobs within the scheduler's ready queue RQ, we can produce the core-local demand-bound function as $\mathcal{G}dbf(\Delta) = \sum_{\tau_i \in \tau} dbf_i(\Delta) + \sum_{i \in RQ(\Delta)} c_i$. Please note, as the worst-case execution time for a task is normed to the maximum clock speed level, the above demand bound functions are also normed to the maximum clock speed level.

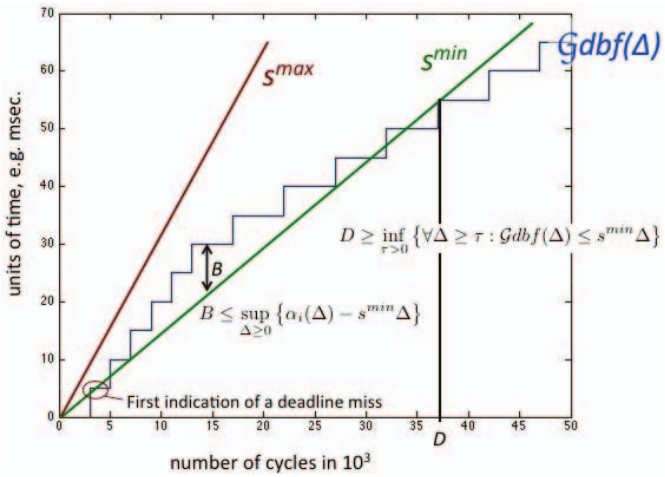


Fig. 1. Demand bound function and speed bounds

The memory space B required for storing backlogged jobs needs to be large enough to hold at least $\sum_i B_i$ jobs. To statically bound this number at design time, we request that for the minimal device speed s^{min} the following relation holds: $B \leq \sup_{\Delta \geq 0} \{\alpha_i(\Delta) - s^{min}\Delta\}$

Satisfaction of this relation does not prevent timing violations. With $\mathcal{G}dbf(\Delta) > s^{min}\Delta$, it might occur that under the worst-case scenario the processing of a job cannot be completed prior to its deadline. Henceforth, we assume that s^{min} is chosen such that beyond some threshold value $\tau > 0 : \forall \Delta \geq \tau : \mathcal{G}dbf(\Delta) \leq s^{min}\Delta$ holds. The following relation must be satisfied:

$$D \geq \inf_{\tau > 0} \{\forall \Delta \geq \tau : \mathcal{G}dbf(\Delta) \leq s^{min}\Delta\} \qquad (1)$$

The threshold value which satisfies the above relation is denoted $horizon$ in the following. It resembles what is known as busy period [14]. Threshold value $horizon$ is used by us as input parameter to the algorithm for filling the worst-case ready queue with virtual jobs. This parameter provides an upper bound on the time window for which we need to consider job releases such that timing violations are strictly ruled out.

Example: As an example, assume a PJD task activation pattern with period $P = 5$, jitter $J = 16$ and minimum distance $D = 2$. This implicitly defines a bounding curve α. Fig. 1 shows the DBF which we derive from α by assuming a worst-case execution time of $C_i = 250cycles$ and a deadline of $D_i = 3$ units of time. In addition, we depict the supply function when running the processor at the minimum speed s^{min} or at the maximum speed s^{max}. On the basis of the curves dbf and the speed-dependent supply functions, we compute the bound on the buffer space B and the bound D on the busy period, where the later gives parameter $horizon$

Algorithm 1 Maintaining the Worst-case Ready Queue

```
1:  procedure COMPUTEWCRQ(horizon)
2:      Require: Time elapsed since last renewal point T
3:      /* Loop over all task sets */
4:      for i = 1 to N do
5:          repeat x = 0, sum = 0, s = undef
6:              /* number of newly expiring jobs at k × D_i */
7:              new = F_i((x + 1) × D_i) − sum
8:              d = now + (x + 1) × D_i
9:              for j = |WCRQ| to new + |WCRQ| do
10:                 WCRQ ∪= {(i, C_i, d, s, v, sum++)}
11:             end for
12:             x = x + 1
13:         until x × D_i ≥ horizon
14:     end for
15: end procedure
```

to be used as input parameter to our scheme discussed below.

IV. Online Handling of Real-time Tasks

A. Fine-grained vs. coarse-grained DVFS

We assume that the system operates at a fixed, small set of speed levels S. This limitation takes its motivation from the following: (a) the authors of [15] report that (fine-grained) DVFS is effective on older platforms, but on newer architectures, DVFS can even increase energy usage. (b) Dealing only with few speed levels limits power and switching overheads and allows one to include them as fixed constants.

B. Run-time monitoring of task activations

We track a task's activations w. r. t. a linearized overapproximation of its pre-defined bound α_i. Each linear piece is hereby guarded by its own so called *dynamic counter*. Essentially a dynamic counter works like a leaky bucket traffic shaper, where the fill-level of a set of these are used by us for actually bounding future task activations. Further details are omitted here for brevity and can be found in [9].

C. Computing safe speed assignments

In the following we give the algorithm for assigning the speeds of the real-time workload on a single core. Instead of a ready queue, we maintain a worst-case ready queue from which we compute safe speed assignments for the task activations to be processed.

1) The Worst Case Ready Queue (WCRQ): The worst-case ready queue is the queue of actual and potential task activations. Each entry is a tuple $j = (id, c, d, s, type, max)$ where id is a task identifier, c is the residual worst-case execution time, d is the absolute deadline, and s is the requested execution speed. In addition to this, we add a flag $type$ to mark entries as being either *real* or *virtual*, and we track the maximum number max of potential job releases up to the release under consideration. With type real, it is meant that the entry refers to a pending job, whereas type virtual refers to a potential job release. *By adding the released jobs to the $WCRQ$, the scheduler's ready queue RQ becomes a subset of the $WCRQ$ and can be dropped.*

The algorithm for computing the entries of the WCRQ is given in Algo. 1. Function $F_i(\Delta)$ gives an upper bound on the number of job releases to be expected up until time Δ, for further explanation, see [9]. The FOR-loop (line 9-11) expands entries with multiplicity larger than one, which simplifies the speed computations which we illustrate below.

2) Computing speed assignments for tasks: Algo. 2 determines speed assignments for the real jobs within the $WCRQ$. The algorithm takes the most pessimistic point of view by considering the highest possible interference from jobs potentially preempting the job under consideration, here for job e at position i in the $WCRQ$ (line 3). The interference is considered by looking at the (virtual) jobs in front of job e in the $WCRQ$.

Lines 7-12 find the lowest speed in S, at which job e can finish before its deadline. This is done by calculating how much computation time is available up to the deadline $d_{e.id}$ and subtracting the time reserved $CTime$ by real and virtual jobs already in the queue.

Once a speed level is found which allows e to complete within the available time slot, the loop is left (line 10), and the time needed to complete e is reserved in $CTime$ on line 13. As the job now has reserved the time needed for it to complete, no further action is necessary and the algorithm moves on to consider the next job (line 15).

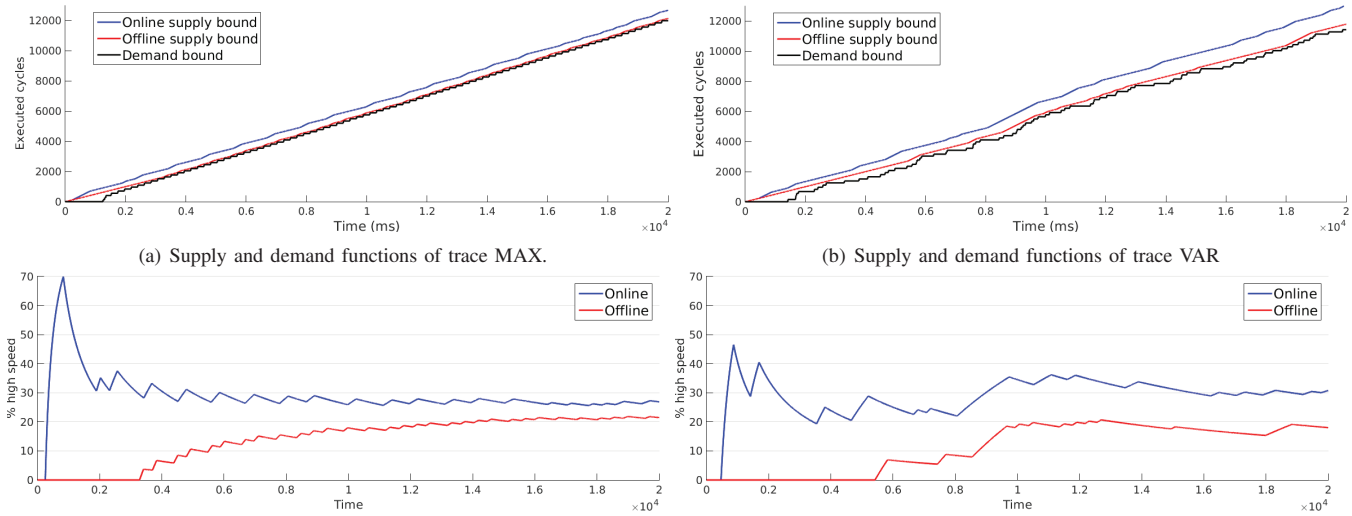
If no speed level within S is high enough to give e sufficient time to complete, the speed assignment for e defaults to s^{max} (line 4). The algorithm then iterates backwards through the WCRQ and speeds up earlier jobs (lines 18-34). The speed assignments of earlier jobs are increased one job at a time to s^{max}, until the resulting free time slot is large enough for e to finish before its deadline.

Switching directly to the maximum speed is motivated by simplicity. At the cost of increased computational effort, the speed levels of the previous jobs can be more finely tuned, although this would increase the worst-case execution time of the scheduler. On the other hand, switching directly to maximum should increase the idle times of a core and thereby lead to better power savings or faster cool down of the core.

In the speed computations, Algo. 2 assumes the earliest release of each task instance. However, the modelling of the task activation pattern includes non-determinism w. r. t. the actual release time of a job. As job deadlines are specified relative to the release time, any delay in the release of a job may result in a change of the execution order. A delayed release may actually yield that the job processed up until now, according to EDF, has to wait until the newly released job has been processed. With job e executing at speed s^l this might not give enough computation time for the pushed back job to complete. For conservatively taking this into account, we cumulate the computing time of jobs the execution of which can potentially delay the currently executing job. We do this by tracking the variable $e.max$ of the respective entry in the $WCRQ$ (line 28-31 of Algo. 2).

V. Empirical Evaluation

The presented scheme has been implemented in C and is executed (bare-metal) on a simulated ARM system using



(a) Supply and demand functions of trace MAX.

(b) Supply and demand functions of trace VAR

(c) Percentage of accumulated time of which the scheduler executes the *MAX* trace at the higher speed.

(d) Percentage of accumulated time of which the scheduler executes the *VAR* trace at the higher speed.

Fig. 2. On- and Offline computed speed assignments.

Algorithm 2 Computing a safe speed-up time

```
1:  procedure SpeedAssignments(WCRQ)
2:      /* Requires that S is ordered LO to HI */
3:      for CTIME = now(), i = 1 to |WCRQ| do
4:          e = peek(WCRQ, i) ;    e.s = s^max
5:          /* find lowest speed in S which meets deadline */
6:          found_safe_speed = false
7:          for each s^cur in S do
8:              if 1/s^cur × c_e.id ≤ d_e.id − CTime then
9:                  e.s = s^cur ;    found_safe_speed = true
10:                 break
11:             end if
12:         end for
13:         CTime += 1/e.s × c_e.id
14:         if found_safe_speed then
15:             continue
16:         end if
17:         /* job will miss deadline with any available speed */
18:         for V = ∅, j = i − 1 to 1 do
19:             n = peek(WCRQ, j)
20:             /* meet deadline by speeding up earlier jobs*/
21:             if CTime > e.d then
22:                 CTime −= c_n.id × (1/n.s − 1) ;    n.s = s^max
23:             end if
24:             if e.type == v ∨ n.type == r ∨ n ∈ V then
25:                 continue
26:             end if
27:             /* check if slow is safe for virtual job n */
28:             if n.max × c_n.id + c_e.id × (1/e.s > D_e.id) then
29:                 CTime −= c_e.id × (1/e.s − 1) ;    e.s = s^max
30:                 break
31:             end if
32:             V ∪= {n.id}
33:         end for
34:     end for
35: end procedure
```

Gem5, an open-source licensed and extensively configurable hardware simulator.

A. Experimental setup

The platform configured in the simulator is a RealView Versatile Express EMM board, which is one of the predefined ARM-based board configurations shipped with Gem5. In addition to the predefined hardware, we add a new device, called the SchedHelper. This device is tasked with reading events, i. e., task activations, from a trace file and makes task activations available to the scheduler via a set of memory mapped registers. The scheduler implements the proposed scheme, executes on top of the simulated hardware, and is invoked each time a previous job finishes or a new job arrives, as these are the only points under EDF where a context switch may occur. Consequently, this is also the granularity at which the speed assignments are done. The clock speed of the processor is controlled by the scheduler via the Gem5 DVFS interface.

For evaluating the proposed scheme, we implemented an online and offline scheduler within Gem5. While the online scheduler implements the proposed scheme, the offline scheduler operates on an ideal (optimal) WCRQ. This is implemented through the use of two PEEK registers provided by the SchedHelper device, which allow the offline scheduler to look ahead into the future (as it is defined by the trace file) and code the virtual jobs with their exact deadline, and the real jobs with their actual execution time. The on- and offline schemes both then execute the speed assignment algorithm (Alg. 2) on the WCRQ. This means that the offline results are the optimal results as calculated under perfect knowledge.

B. Experiments

We evaluate the system with respect to a single task, whose activations follow a PJD-model, with $p = 220$, $j = 388$, $d = 48$. The task has a relative deadline $D = 1250$, worst-case execution time $C = 150$ and the jobs have random actual execution times in the interval $[125, 150]$. All values are given in milliseconds. In addition to this the system is evaluated on a set of two speeds $S = \{\frac{1}{2}s^{max}, s^{max}\}$.

We generated and evaluated two different traces of task activations, referred to as *MAX* and *VAR*. Both traces are 20

seconds long. In the *MAX* trace, job releases strictly adhere to the upper bound provided by the arrival curve which models the PJD pattern defined above, i. e., these traces show a burst in the beginning, followed by strictly periodic job releases. The second trace, *VAR* specifies job releases that are more irregular, but still within the bounds derived from the PJD model above.

C. Results

The speed assignments for the *MAX* and *VAR* traces are presented in Figures 2(a) and 2(b) respectively.

The percentage of time the processor is executing at the higher speed for the *MAX* and *VAR* traces is presented in Figures 2(c) and 2(d) respectively.

The assigned speeds produce a piecewise linear two-speed supply curve which in the offline case minimizes the area between the supply and demand curve. In the online case, the area is minimized w. r. t. the worst-case demand bound function at each point in time, which, as it is calculated online from the dynamic counters, is not depicted here. Since the speed assignments are limited to two speeds, none of which are the *zero* speed, the system will produce clock cycles at all points, even when there is no job to execute. This is visible in the *VAR* trace, in which the offline supply curve does not tightly follow the demand curve, as with the more periodic (i. e., smooth load) *MAX* trace.

In both traces, the speed assignments by the offline scheduler outperforms the online scheduler. This is expected, as the offline scheduler can verify that all jobs meet their deadline by looking into the future events of the trace file. The online scheduler can only determine this once the leaky bucket style workload monitor indicates that the peak workload has passed.

The offline speed assignments stabilize at around 20% of the execution being done at the higher speed for both the periodic *MAX* trace and the bursty *VAR* trace. The online scheduler produces results comparable to this on the *MAX* trace, although it assigns the higher speed more often. On the more bursty *VAR* trace, it assigns the higher speed about 50% more often than the offline scheduler, which is considerably more, but must still be considered acceptable due to the unknown arrival times.

VI. Conclusions

We propose an history-aware, online DVFS scheme which operates the processor at higher clock frequencies only if the future worst-case real-time workload potentially produces timing violations if not compensated by a higher device speed.

The proposed monitoring scheme ensures that the online scheduler does not forget the bursty behaviour of the workload history. This yields clearly an advantage of the presented scheme over the DVFS schemes of [5], [10].

Previous trace-less monitoring schemes for bounding hard real-time workloads [9], [4], [11] did not target DVFS. Hence a comparison with our work is not possible. In order to execute the most critical assessment, we benchmarked our scheme with a scheduler under perfect knowledge. The presented evaluation of corner cases reflects the best and worst behaviour of the scheme w. r. t. the optimal, but unrealistic, behaviour under perfect knowledge.

For practical reasons, we only evaluated the system using 2 speed levels. An extension to larger sets of speed levels appears straight forward. Considering only few speed levels is justified as modern processor designs provide similar temperature and power characteristics for clock speeds beyond the thermal design power.

References

[1] The gem5 Simulator System. http://www.gem5.org/.

[2] S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *RTSS 1990*, pages 182–190. IEEE, 1990.

[3] R. A. Bergamaschi, G. Han, A. Buyuktosunoglu, H. D. Patel, I. Nair, G. Dittmann, G. Janssen, N. R. Dhanwada, Z. Hu, P. Bose, and J. A. Darringer. Exploring power management in multi-core systems. In *ASP-DAC 2008*, pages 708–713, 2008.

[4] F. Bodmann, N. Muehleis, and F. Slomka. Situation aware scheduling for energy-efficient real-time systems. In *8th Workshop Cyber-Physical Systems 2011*, 2011.

[5] J.-J. Chen and T.-W. Kuo. Procrastination determination for periodic real-time tasks in leakage-aware dynamic voltage scaling systems. In *ICCAD 2007*, pages 289–294, 2007.

[6] A. Das, A. Kumar, B. Veeravalli, R. Shafik, G. Merrett, and B. Al-Hashimi. Workload uncertainty characterization and adaptive frequency scaling for energy minimization of embedded systems. In *DATE 2015*, pages 43–48, San Jose, CA, USA, 2015. EDA Consortium.

[7] P. Huang, O. Moreira, K. Goossens, and A. Molnos. Throughput-constrained voltage and frequency scaling for real-time heterogeneous multiprocessors. In *SAC 2013*, pages 1517–1524, New York, NY, USA, 2013. ACM.

[8] G. Keramidas, V. Spiliopoulos, and S. Kaxiras. Interval-based models for run-time dvfs orchestration in superscalar processors. In *7th ACM Int. Conf. on Computing Frontiers*, CF '10, pages 287–296, New York, NY, USA, 2010. ACM.

[9] K. Lampka, K. Huang, and J.-J. Chen. Dynamic counters and the efficient and effective online power management of embedded real-time systems. In *CODES+ISSS 2011*, pages 267–276, Taipei, Taiwan, Oct 2011. ACM.

[10] A. Maxiaguine, S. Chakraborty, and L. Thiele. DVS for buffer-constrained architectures with predictable QoS-energy tradeoffs. In *the International Conference on Hardware-Software Codesign and System Synthesis*, pages 111–116, 2005.

[11] M. Neukirchner, T. Michaels, P. Axer, S. Quinton, and R. Ernst. Monitoring arbitrary activation patterns in real-time systems. In *RTSS 2012*, Dec 2012.

[12] S. Perathoner, J.-J. Chen, K. Lampka, N. Stoimenov, and L. Thiele. Combining optimistic and pessimistic dvs scheduling: An adaptive scheme and analysis. In *IEEE/ACM International Conference on Computer-Aided Design*, pages 131–138, San Jose, California, USA, 2010. IEEE.

[13] S. Perathoner, K. Lampka, and L. Thiele. Composing heterogeneous components for system-wide performance analysis. In *DATE 2011*, pages 1–6, Grenoble, France, Mar 2011. IEEE.

[14] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. *IEEE Trans. Comput.*, 44(1):73–91, Jan. 1995.

[15] E. L. Sueur and G. Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *In 2010 HotPower (HotPower?10*, 2010.

[16] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *Intl. Symposium on Circuits and Systems*, volume 4, pages 101–104, 2000.

[17] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *36th Annual Symposium on Foundations of Computer Science*, pages 374–382, 1995.

[18] G. Zeng, Y. Matsubara, H. Tomiyama, and H. Takada. Task migration for energy saving in real-time multiprocessor systems. In *High Performance Computing and Communications, 2014 IEEE 6th Intl Symposium Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Systems*, pages 685–692, Aug 2014.