A Study of Contributing Factors to Power Aware Vertical Scaling of Deadline Constrained Applications

1st Pradyumna Kaushik
Binghamton University, SUNY
New York, USA
pkaushi1@binghamton.edu

2st Srinidhi Raghavendra
Binghamton University, SUNY
New York, USA
sraghav2@binghamton.edu

3rd Madhusudhan Govindaraju
Binghamton University, SUNY
New York, USA
mgovinda@binghamton.edu
0000-0002-7953-8000

***Abstract*—The adoption of virtualization technologies in datacenters has increased dramatically in the past decade. Clouds have pivoted from being just an infrastructure rental to offering platforms and solutions, made possible by having several layers of abstraction, providing internal and external users the ability to focus on core business logic. Efficient resource management has in turn become salient in ensuring operational efficiency. In this work, we study key factors that can influence vertical scaling decisions, propose a policy to vertically scale deadline constrained applications and surface our findings from experimentation. We observe that (a) the duration for which an application is profiled has an almost cyclic influence on the accuracy of behavior predictions and is inversely proportional to the time spent consuming backlog, (b) the duration for which an application is scaled can help achieve up to a 9.6% and 4.2% reduction in the 75^{th} and 95^{th} percentile of power usage respectively, (c) reducing the tolerance towards accrual of backlog influences the application execution time and can reduce the number of SLA violations by 50% or 100% at times and (d) increasing the time to deadline offers power saving opportunities and can help achieve a 9.3% improvement in the 75^{th} percentile of power usage.**

***Index Terms*—power-aware, vertical scaling, deadline constrained applications, containers**

I. Introduction

Resource management comes in various forms: (a) scheduling solutions that alter the combinations of co-scheduled applications, (b) horizontal scaling solutions that change the number of compute instances (servers, virtual machines etc) running the workloads, (c) vertical scaling solutions to compress or relax resource constraints on individual applications in order to respond to resource demand variations and (d) regulation of raw resources supplied to infrastructure and applications by power capping, hardware throttling and other such measures. Depending on the demand and the desired target, a combination of these techniques can be employed. In this work, we study vertical scaling of *deadline constrained* applications and more specifically, analyze a few key factors that can influence scaling decisions. Vertical scaling offers fragmentation reduction opportunities by facilitating adjustment of an application's resource allocation either by freeing up resources to be used by another or aligning with an increase in resource demand. A key challenge with vertical scaling is the uncertainty of the impact on performance, especially when tightening resource constraints (commonly termed *scaling down*). In addition, scheduling constraints such as deadlines add complexity to vertical scaling and hence is the focus of our research work.

In this work, we focus on vertical scaling of deadline constrained applications and study some key factors that influence the scaling decisions. We study the:

- effect of varying duration for which an application is profiled, which we term *profiling window*, on the accuracy with which we can estimate the behavior of that application under the influence of resource constraints.
- effect of varying a configurable knob representing the tolerance towards the amount of accrued backlog, resulting from a performance impact, on the number of SLA violations. We term this knob *backlog tolerance*.
- impact of extending the duration for which an application can be vertically scaled prior to be being profiled again, termed *scaling window*, on the induced power draw of the host CPU.
- impact of loosening the deadline constraints on the power saving opportunities that arise from it.

We make the following contributions.

- Discuss four contributing factors to power aware vertical scaling of deadline constrained applications: *time to deadline*, *profiling window*, *scaling window* and *backlog tolerance*.
- Propose a vertical scaling policy that efficiently controls the resource constraints on an application by taking into account these contributing factors in order to make smart power and performance trade-offs.
- Introduce *Accordion*, a power aware vertical scaler that has been designed for this project.
- Quantify the impact of the contributing factors on SLA violations, application behavior prediction, and power savings.

II. Contributing Factors

This section discusses contributing factors to vertical scaling of deadline constrained batch applications.

Table I lists some terms used hence forth.

TABLE I: Terminologies

Term	Description
Time to deadline (ttd)	Duration from application execution start time to the time it is expected to finish execution
profiling window (pw)	time duration for which application is profiled
cron cycle (cc)	duration between two successive executions of the vertical scaler periodic job.
metrics cycle (mc)	time duration between two successive fetches of metrics from Prometheus
Instruction Backlog (IB)	number of instructions that were expected to retire in addition to those that already did

A. Deadline Constraints

Batch applications are typically run during off-peak hours and have associated deadline constraints. Deadline sensitive scheduling algorithms are commonly used to schedule such applications. Application to compute node mapping is performed with the inference that the application will complete execution prior to reaching its deadline. Note that execution speed up of applications is out of scope of this paper. With eT as the execution time (unknown to us) and ttd as the time to deadline, it is assumed that $eT < ttd$. Otherwise, either (a) the scheduling algorithm has excessively delayed launching the application or (b) the deadline constraints are inaccurate.

In order to mitigate the risk of violating an application's deadline constraints, prior knowledge can help exploit variations in computational intensity to trade performance for power savings. However, if limited or no insight is available on the type of applications being run, then it is complex to make such trade-offs between performance and power consumption.

We consider applications to be black boxes in that no prior information is available other than their resource requirements and deadline constraints. Deadline constraints from the point of view of the vertical scaler are expressed as the time duration from the start of execution to the time by when the application is required to have completed execution. Note that this does not factor in the time that an application spent waiting for the scheduler to assign it to a particular compute node. Timely dispatching of applications by the scheduler can be achieved by employing deadline based scheduling algorithms and is out of scope of this paper.

B. Application Behavior

Applications that demonstrate high computational intensity are more susceptible to suffrage caused by tightening of resource limits. In addition, applications can showcase different characteristics through the course of their execution. For instance, an application might use 80% of 6 cores in the first 10 minutes and then use 40% of 4 cores in the remaining time. Such an application might therefore be less tolerant towards a certain amount of resource squeeze in the first 10 minutes than it would be in the latter portion of its execution period.

The impact on the execution time as a result of scaling down an application is determined by the behavior of the application during that phase of execution and is therefore important to realize in order to exert limits on the extent of scaling. C. Delimitrou et al. [1] mention instructions per second (IPS) as a metric that defines performance of single node (single threaded or multi-threaded) batch applications. R. Brondolin et al. [2] also used number of instructions retired (IR) within two sampling cycles as a metric to define performance. In this work, we consider the average instruction retirement rate (IRR) over a window of time, called *profiling window (pw)* (explained in table I), to represent the behavior of an application until the next profiling window. Instruction Retirement Rate is a measure of throughput indicating the number of instructions completing execution within a second. Equation (1) shows how application behavior is computed. The factor pw/mc in equation (1) gives the number of metric data points collected in a profiling window with *mc* denoting the time interval between two successive metrics fetches (shown in table I). IRR_i in equation 1 is calculated by dividing the number of instructions retired between two successive metrics collection events by the number of seconds separating them. Number of instructions retired is surfaced by cAdvisor [3] which is exposed by Prometheus [4] (explained later in Section III). In this work, it is assumed that an application can demonstrate its true behavior when allocated the requested amount of resources. Therefore, profiling of an application is preceded by a scale up of its current resource allocation until it matches the requested amounts.

$$appBehav = (\sum_{\mathrm{i=0}}^{\mathrm{i=pw/mc}} IRR_{\mathrm{i}})/(pw/mc) \quad (1)$$

Assumption: Allocating more than the requested amount of resources does not provide any gains in performance.

1) ***Profiling Window****:* The duration for which an application is profiled, called *profiling window*, can influence the accuracy of scaling decisions, thereby affecting the time that an application spends consuming any instruction backlog. Short profiling duration will suffice if the application under study demonstrates constant behavior. On the other hand, a longer profiling duration would be required for an application behaving erratically (random fluctuations of compute usage, for example). To avoid resource constraints influencing behavior, an application is allocated all the requested resources while being profiled. Profiling Window configured as the number of cron cycles to be utilized by the vertical scaler in a single profiling event.

2) ***Scaling Window****:* Given that applications can demonstrate varied behavior over time, it is important to repeatedly capture this behavior to aid in estimating the performance impact of vertical scaling. *Scaling Window (sw)* represents the time separation (configured as the number of cron cycles) between two successive profiling events. So, *sw=2* indicates

that the time duration between two profiling events is *2*cc*, where *cc* is the time duration between two cron job cycles of the vertical scaler (shown in Table I).

C. Backlog

The deadline constraints of applications impose restrictions on the extent to which resource allocations can be adjusted. However, viewing the application as a black box complicates estimation of the amount of pending work. With application behavior quantified using average instruction retirement rate over a window of time (described in section II-B), we measure backlog as the additional number of instructions that would have otherwise executed if it were not for the adjustment of the application's resource allocation.

The number of instructions that are expected to retire between two consecutive cron job execution events, with the requested resources allocated, is termed I_{exp} and is calculated using profiled application behavior as shown in equation (2). When an application is scaled down, it is possible that it retires lesser instructions than it would have otherwise. Backlog (IB) is defined as the additional number of instructions that would have otherwise been retired if the application was not subject to vertical scaling. Algorithm 1 shows how instruction backlog is accrued. The schedule that Accordion's vertical scaler follows can be different from the schedule on which metrics are fetched. With mc as the interval between two successive metric fetches, $IRRValues$ convey the instruction retirement rate during mc. $IRRValues$ are used to determine the portion of the historic backlog that is consumed. If some of the time was spent consuming any historic backlog, then it implies that the application has accrued new backlog and I_{exp}, application behavior, is used to determine the new backlog.

The time it would take to execute any backlog (IB_{eT}) is calculated using equation (3).

Figure 1 illustrates how instruction backlog is accrued and how application behavior is profiled. The x axis represents time where $\mathtt{T}_x$ represents an arbitrary point in the application's execution period, and cc represents the time interval between two successive cron job cycles (refer table I).

- At $\mathtt{T}_x$, the application has been profiled and the behavior (red solid line) is understood. The application is scaled down (say) (shown by the solid green line) affecting its behavior as seen by a decrease in the instruction retirement rate (shown by the solid yellow line). This impact on behavior causes an increase in the instruction backlog (shown by the solid pink line).
- At $\mathtt{T}_{x+cc}$ some amount of backlog has been accrued and it is now time to profile the application again (say). To profile the application the resource allocation is adjusted such that the application is now allocated all the requested resources. The intent of profiling is to realize new behavior of the application and therefore the first step in that process is to zero-out any pending instruction backlog. Profiling begins only after all pending instruction backlog has been consumed. This is seen by the wait from $\mathtt{T}_{x+cc}$ to $\mathtt{T}_{x+2cc}$.
- Assuming profiling window (pw) is one cycle, $\mathtt{T}_{x+2cc}$ to $\mathtt{T}_{x+3cc}$ is spent in profiling. With the newly witnessed behavior of the application (shown by the solid red line after $\mathtt{T}_{x+3cc}$), vertical scaling commences once again.

$$I_{exp} = appBehav * cc \tag{2}$$

Algorithm 1 Accrue Instruction Backlog

```
IRRValues[..] ← IRR data points
curB ← current instruction backlog
consumedB ← consumed backlog
totRetI ← total number of instructions executed
pendB ← backlog pending consumption
retI ← number of instructions retired
function AccrueBacklog(IRRValues, cc, mc)
    pendB = curB
    for irr in IRRValues[..] do
        retI = irr * mc
        totRetI += retI
        if pendB > 0 then
            c = min(retI, pendB)
            consumedB += c
            pendB -= c
        end if
    end for
    curB -= consumedB
    if scaledDown then
        curB += max(0, I_exp - (totRetI - consumedB))
    end if
end function
```

$$IB_{eT} = IB \div appBehav \tag{3}$$

*1) **Backlog Tolerance**:* At any given point during an application's execution, the remaining time to deadline is going to be utilized to not only complete any remaining work but also to consume any accrued backlog. It is impractical to expect an application to speed up its execution in order to complete before the deadline. Therefore, the more the accrued backlog, the lesser the chances of the application having time available to complete remaining work. The acceptable portion of the remaining time to deadline that any backlog can take to execute, is defined as $Backlog\ Tolerance$ (IB_{tol}). Backlog Tolerance provisions for (a) throttling the accrual of instruction backlog and (b) regulating the extent to which an application's resource constraints can be tightened.

III. Accordion

Accordion is a vertical scaling framework, we developed for this project, that can vertically scale docker containers using a defined policy and set configuration. Accordion uses Task-Ranker [5] to pull container metrics from Prometheus [4]. For this work, Task-Ranker is being used for its natural side effect: the ability to periodically fetch, parse and broadcast

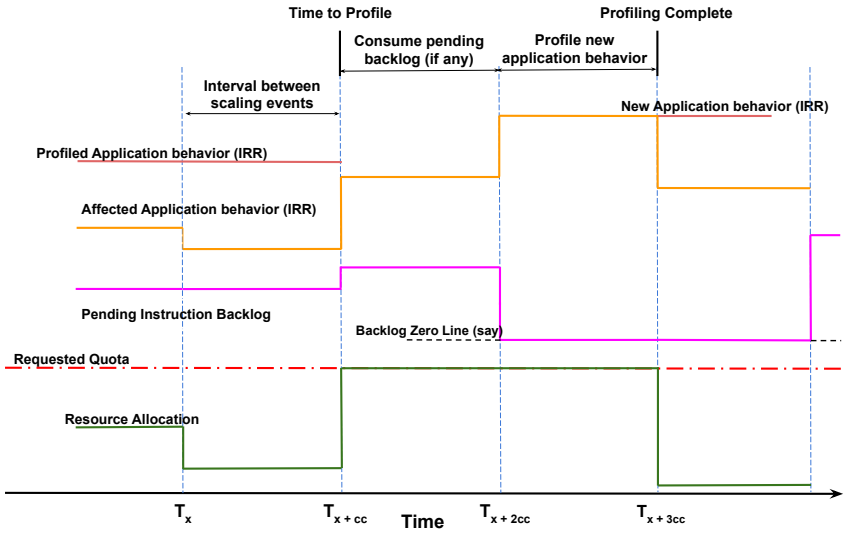


Fig. 1: Illustration of Instruction Backlog Accrual.
Note that this is synthetic data but captures the instruction backlog accrual process.

metrics from Prometheus. Figure 2 illustrates the architecture of Accordion and shows the various components and their interactions. These are described below.

- **Metrics Collector** - Metrics Collector is a receiver of fetched and parsed Prometheus metrics. This box in the diagram encapsulates the interactions that Accordion has with Task-Ranker. The Metrics Collector attaches itself as a receiver with Task-Ranker. Task-Ranker is configured to periodically query Prometheus metrics and broadcast them to the registered callbacks. The collected metrics are then stored in a metric data store that exposes APIs for CRUD operations.
- **Metrics Analyzer** - Metrics Analyzer can be used to derive additional metrics by analyzing the collected metrics in the data store.
- **Resource Recommender** - The Resource Recommender is responsible for analyzing metrics and recommending an acceptable range for resource quota adjustments. The metrics analyzed are listed below.
  - Application behavior provides insight into the work being done by the application and thereby facilitates the estimation of impact that resource adjustments have on the completion time.
  - Current accrued backlog.
  - Remaining time to deadline.
  - Backlog Tolerance.

  Given that memory is a hard constraint and exceeding memory limits results in the respective containers getting killed, the resource recommendations made are on the CPU dimension. Algorithm 2 outlines the steps taken to recommend a range of possible cpu adjustments for a given application. Note that vertical scaling along the memory dimension of deadline constrained applications is out of scope of this paper and would need to also factor in the possibility of the application OOMing [6] and employ checkpoint and restart [7], [8] to preserve execution state or factor in the delay should an execution restart occur.
  For each running task, the recommender checks (a) if the task can be scaled down relative to the requested resources, (b) if it is time to start profiling the task to witness any change in behavior, or (c) if the task is currently being profiled and whether to continue profiling for another cycle. The actions executed for each of the checks are mentioned below.

  a Realize new application behavior by analyzing the monitored metrics (see Section II-B). Given the deadline constraints, it is important to have a view of the current backlog in order to assess the impact of vertical scaling. Using the instruction retirement rate of the task monitored in the past cron cycle (see Table I), some or all of the backlog is consumed and new backlog, if any, is accrued (see Section II-C). Immediately after realizing new behavior scaling recommendation is kept simple to max=requested CPU and min=requested CPU - 2 CPUs. This is a result of having minimal information on the impact of scaling on the application with the newly witnessed behavior. At other times, scaling recommendations are made such that any CPU allocation within the recommendation range is not expected to result in a backlog that is going to apply more pressure on the remaining time to deadline than defined by IB_{tol} (see Section II-C1). Impact on task behavior is estimated using historical data. We run linear regression on historical data to estimate the instruction retirement rate

(application behavior) as a result of adjusting the CPU allocation (appBehav$_{est}$). Estimated application behavior is then used to gauge accrual of backlog (IB$_{eT}$) using equation (3). If Backlog Tolerance is not breached by potentially accruing IB$_{eT}$ then the corresponding CPU adjustment is included in the recommended range.

b If it is time to profile then the requested resources are what is recommended. This sets up the task to be scaled up all the way prior to it being profiled.

c If we are currently in the process of profiling a task, then we update the current backlog, make no scaling recommendation and maintain the current resource allocation for another cron cycle.

- **Resource Updater** - The basic assumption that the resource updater makes is that any of the recommendations can be selected. Instead of randomly selecting a recommendation, the Resource Updater selects a recommendation based on the elapsed portion of the total time to deadline. With *cpuMin* being the minimum recommendation and *cpuMax* being the maximum recommendation, the range `[cpuMin, cpuMax]` is divided into four sub-ranges of equal size and discretized into five values. Four sub-ranges were chosen to simplify the heuristic to converge at a CPU allocation by selecting one allocation amongst five. The resource adjustment is stricter if lesser time has elapsed and vice versa. For instance, if cpuMin=2 and cpuMax=7, the discretized values representing possible adjustments are [2, 3.25, 4.5, 5.75, 7]. If the application has elapsed more than 80% of its total available time, it would be scaled to 7 cpus. If the application has elapsed more than 60% but less than 80% of its total available time, it would be scaled to 5.75 cpus, and so on.

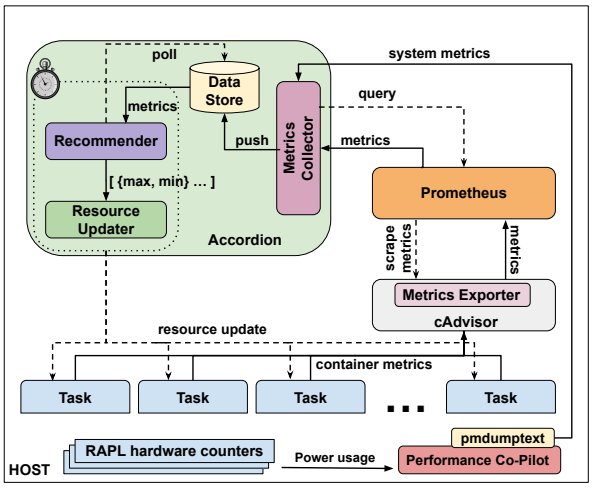


Fig. 2: Accordion Architecture

Algorithm 2 CPU Max Min Recommendations

```
mAnalyzer ← metric analyzer
CPU_req = requested cpu
for app in runningApps do
    if app.profiling.Finished() then
      appBehav = mAnalyzer.behavior(IRR_prof[..])
      recommend(CPU_req, CPU_req - 2)
      stop_profiling()
    else if app.profiling.in_progress() then
      profiling.continue()
    else
      accrue_backlog()
        if time_to_profile() then
          recommend(CPU_req, CPU_req)
        else
          recommend(find_max_min)
        end if
    end if
end for
function FIND_MAX_MIN
    IB_eT ← Time to execute backlog
    IB_tol ← Backlog Tolerance
    TTD_pend ← Remaining time to deadline
    for CPU_new in [scale_limit_min, CPU_req] do
        appBehav_est ← estimate_app_behavior(CPU_new)
        if IB_eT(appBehav_est) < IB_tol * TTD_pend then
          update_max_min(CPU_adj)
        end if
    end for
      return max, min
end function
```

IV. Experimental Setup

To avoid intrinsic bias in data, we conducted our experiments on two different servers. The processor architecture of them has been listed below.

- *Class A* server - 10 cores, 20 thread Intel Xeon E5-2650 v3 @ 2.30GHz and 128 GB RAM
- *Class B* server - 8 core, 16 thread Intel Xeon E5-2640 v3 @ 2.60GHz and 128 GB RAM

Each server runs 64-bit Linux 4.7.10 and shares an NFS server. For scheduling our workloads, we implemented a simple task launcher into Accordion. The task launcher uses docker API to execute tasks described using JSON files. Each of our benchmarks are run inside docker containers [9] to ensure consistency of runtime environments. We use Docker 17.12.0-ce as the container technology. Container specific metrics are queried from Prometheus [4] which scrapes them from cAdvisor [3]. Performance Co-Pilot [10] was used to collect system metrics and power readings from RAPL [11] hardware counters. For all of our experiments conducted in this work, Accordion's cron cycles were 60 seconds apart. Below is a list of applications and the parameters under study and how each of them were configured.

TABLE II: Parameter value range used in experiments

Parameter	Description	Values
ttdeT	total time to deadline as a factor of baseline execution time	[2x, 3x, 4x, 5x]
profiling window (pw)	time duration for each profiling period	[1,2,3,4]
backlog tolerance (IB_{tol})	% remaining time that can be used for backlog execution	[0.25, 0.5, 0.75, 1.0]
scaling window (sw)	duration (as number of cron cycles) between two successive profiling events	[1,3,5,7]

Applications

- *avifenc* [12] - This is a test of the AOMedia libavif library testing the encoding of a JPEG image to AV1 Image Format (AVIF)
- *svt-av1* [13] - This is a benchmark of the SVT-AV1 open-source video encoder/decoder.

Parameters Under Study

- **Profiling Window**: For simplicity, an Accordion cron cycle is either used to vertically scale an application or to profile it. Profiling Window is therefore configured as the number of cron cycles to be consumed each time for profiling.
- **Total time to deadline**: Total time to deadline is expressed in seconds. For ease of analysis, total time to deadline is represented as a factor of baseline execution time as *ttdeT*. For example, ttdeT=2 implies that the total time to deadline is twice the baseline execution time.
- **Backlog Tolerance**: Backlog Tolerance is configured as a double representing the portion of remaining time that can be used to consume any accrued instruction backlog.
- **Scaling Window**: Scaling Window is configured as the number of cron cycles separating two profiling events.

We conducted experiments for all combinations of parameter values shown in Table II to study the influence of each of them on peak power consumption or application performance and ultimately on vertical scaling decisions.

V. Analyses

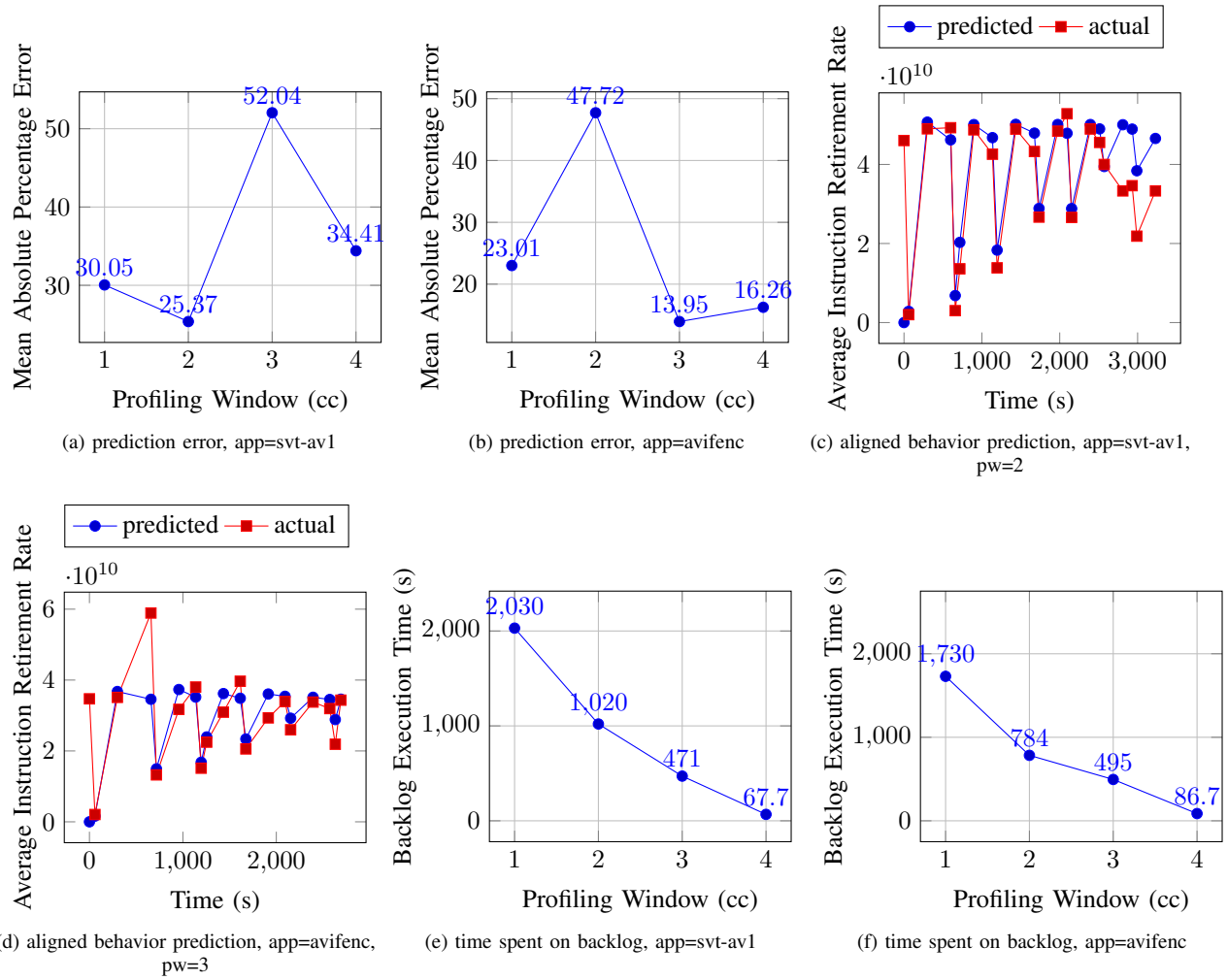
To study the influence of a single parameter in table II, the other three parameters are fixed to a value in the specified ranges. Due to space constraints, we have shown a subset of trends that capture the intended behavior.

A. Profiling Window (pw)

Figure 3 shows the influence that profiling window has on (a) the error in prediction (mean absolute percentage error) of application behavior (3a and 3b) and (b) the time that an application spends consuming its backlog (3e and 3f).
From figures 3a and 3b we can see that the influence of profiling window on prediction error shows an interesting trend where it is almost cyclic. This can be attributed to the alignment of the profiling window with fluctuations in application behavior. A positive take away from the experiment is that there exists a profiling window where the prediction error is minimum. However, it is not the same for all applications and is dependent on behavior consistency. For application svt-av1 a profiling window of 1 cron cycle (cc) results in a prediction error of 30.05% while increasing the window to 2 cc improves the prediction error by 4.7%. Higher values of profiling window ($3cc$ and $4cc$) experience worse prediction errors of 52.04% and 34.41% respectively. For application avifenc, a similar cyclic pattern is observed where the prediction error varies from 23.01% to 47.72% to 13.95% and then to 16.26% for profiling windows $1cc$, $2cc$, $3cc$ and $4cc$ respectively. This cyclic nature can be used to develop a policy where the profiling window is dynamically adjusted until the prediction error is within acceptable ranges. Such a policy is subject of future work. Figures 3c and 3d show the predicted behavior and actual behavior when the prediction error is small for applications $svt - av1$ and $avifenc$.
From figures 3e and 3f we can see that the profiling window can dictate the amount of time spent by an application to consume its backlog. A larger profiling window translates to the application spending more time with a higher resource allocation, thereby reducing the net time spent in consuming any backlog. On the other hand, a smaller profiling window results in the application spending a greater portion of its execution time in a scaled state (allocated resources $<$ requested resources). This leads to the application consuming its backlog at a slower rate.

B. Backlog Tolerance (IB_{tol})

Figure 4 illustrates the influence of backlog tolerance (IB_{tol}) on application execution times and SLA violations. In this context, an SLA violation is said to occur when the execution time exceeds the specified deadline constraints.
Figures 4a and 4c show the number of SLA violations for different levels of IB_{tol}. Lower IB_{tol} imposes a restriction on the extent to which an application is scaled down therefore allowing instructions to be retired at a faster rate, eventually reducing the number of SLA violations. This can be observed in figure 4a where the number of SLA violations goes from 0 to 2 when IB_{tol} is increased from 0.25 to 1, respectively, for application svt-av1. Additionally, for application avifenc, in figure 4c, we see that the number of SLA violations goes from 2 to 4 when IB_{tol} is increased from 0.25 to 1 respectively.
Figures 4b and 4d show the impact of varying IB_{tol} on application execution times. Higher IB_{tol} allows Accordion to be more aggressive when vertically scaling thereby leading to a larger backlog. This leads to more time spent to consume this backlog prior to profiling (duration between T_{x+sc} and T_{x+2c} in figure 1). For application svt-av1, in figure 4b, we can see that an increase in IB_{tol} can affect the 95^{th} percentile of execution time by up to 10.3% and average execution time by up to 4.5%. For application avifenc, in figure 4d, we can see that an increase in IB_{tol} can affect the 95^{th} percentile of



(a) prediction error, app=svt-av1

(b) prediction error, app=avifenc

(c) aligned behavior prediction, app=svt-av1, pw=2

(d) aligned behavior prediction, app=avifenc, pw=3

(e) time spent on backlog, app=svt-av1

(f) time spent on backlog, app=avifenc

Fig. 3: Influence of Profiling Window on the prediction accuracy of application behavior (figures 3a and 3b) and time spent executing backlog (figures 3e and 3f). Figures 3c and 3d show the predicted and actual behavior for applications svt-av1 and avifenc respectively corresponding to the runs with least prediction error. These plots correspond to experiments with ttdeT=3x, sw=5 and bklTol=0.5.

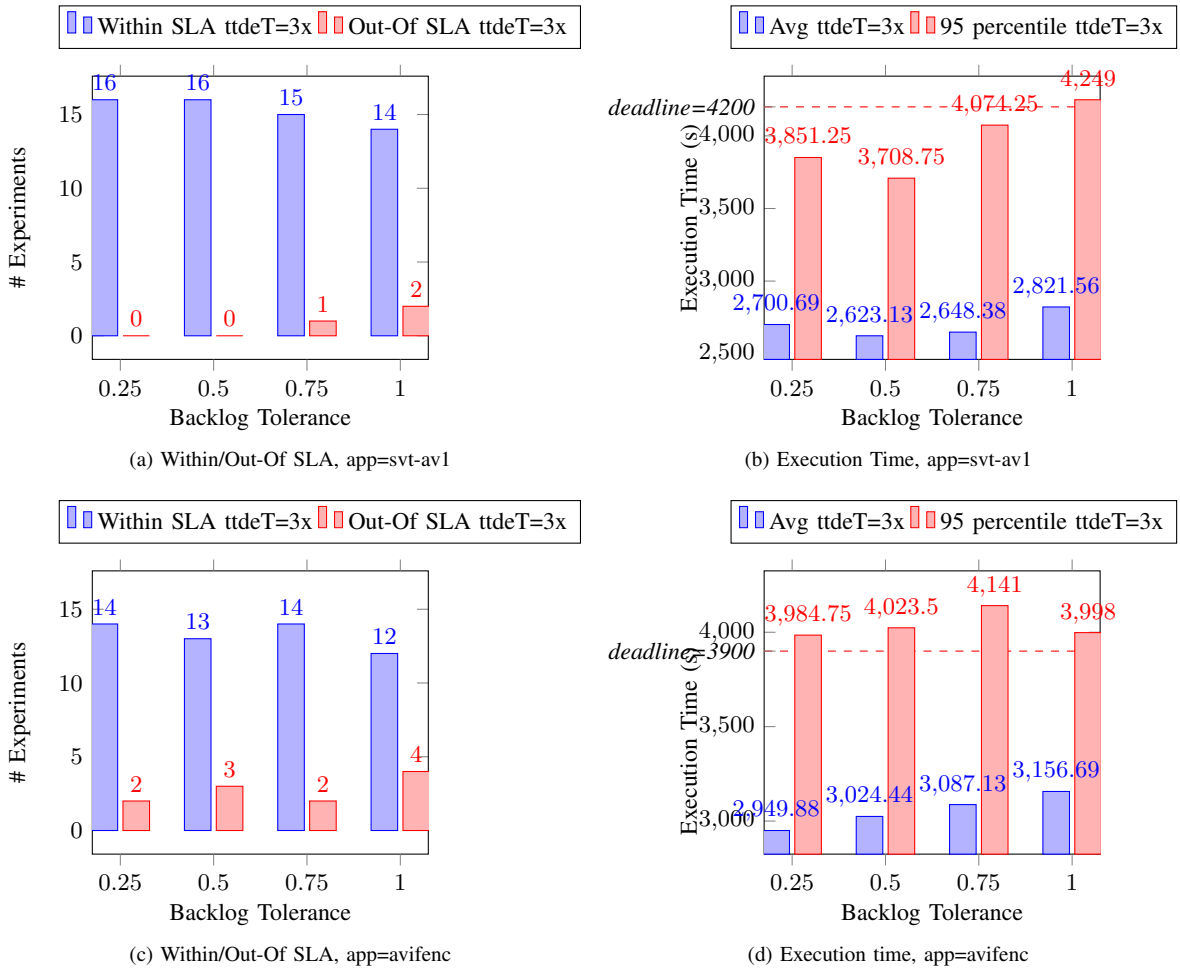
execution time by up to 3.9% and average execution time by up to 7%.

A noticeable outcome in figures 4b and 4d is a large difference between the 95^{th} percentile of execution time and average execution time. For application svt-av1, 95^{th} of execution time is on average 47% more than the average execution time. For application avifenc, 95^{th} percentile of execution time can be on average 32.2% more than the average execution time. This can possibly be attributed to misalignment of vertical scaling and application behavior, warrants a separate study and is the subject of future work.

In this work, only information on the total backlog accrued in a scaling cycle is used when recommending resource adjustments. A limitation with this is that it does not monitor and factor in the rate at which backlog is accrued. A higher backlog accrual rate can indicate that the application is tending towards generating a larger load and this information can be utilized to loosen the resource limits. On the other hand, lower backlog accrual rate can indicate that the application is tending to do lesser work and this information can be employed to tighten the resource limits. This is subject of future work.

C. Scaling Window (sw)

Figure 5 illustrates the influence that scaling window has on power savings under different scenarios. Scenario A (figures 5a and 5c) pertains to having a large profiling window (pw=4) and a low backlog tolerance (bklTol=0.25). Scenario B (figures 5b and 5d) corresponds to having a small profiling window and a high backlog tolerance. Overall, it is evident from the figures that scaling window (sw) can help govern the power envelope and thereby facilitate power savings if need be. For *svt-av1*, we see that varying sw can result in a 9.6% and 4.1% reduction in the 75^{th} and 95^{th} percentiles of power consumption respectively. For *avifenc*, we see that varying sw



(a) Within/Out-Of SLA, app=svt-av1

(b) Execution Time, app=svt-av1

(c) Within/Out-Of SLA, app=avifenc

(d) Execution time, app=avifenc

Fig. 4: Figures 4a and 4c show the impact of backlog tolerance on SLA violations and 4b and 4d show the impact of backlog tolerance on execution time. The *deadline* marker represents the seconds to deadline given ttdeT=3x. It can be observed that backlog tolerance is correlated to execution time and proves viable in mitigating SLA violations.

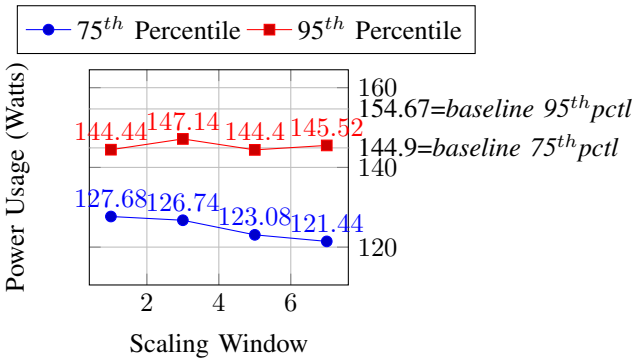
can result in a 6.3% and 2% reduction in the 75^{th} and 95^{th} percentiles of power consumption respectively. Interestingly, scenario B does not produce the best results for *avifenc*. This can be attributed to the rate of backlog accrual not being factored into performance sensitivity and is subject of future work.

D. Total Time to Deadline as a Factor of Baseline Execution Time (ttdeT)

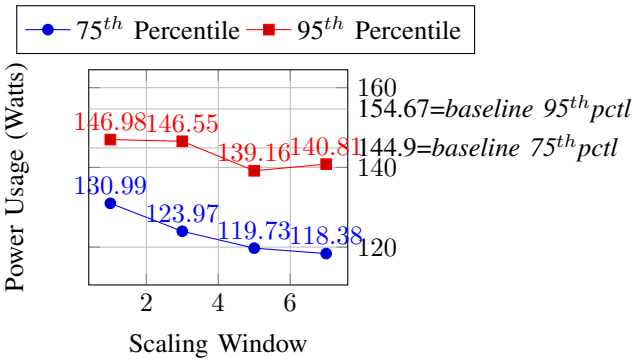
Figures 6a and 7a show the 75^{th} and 95^{th} percentile of power usage averaged across multiple experiments for different values of *ttdeT*. Figures 6b and 7b show the best results for power usage for different values of *ttdeT*. Firstly, it is apparent that increasing the time to deadline does provide power saving opportunities (applications have some buffer to suffer performance impact) which can be seen by a 2% and 3.2% reduction in the average and best result, respectively, in 75^{th} percentile of power usage for *svt-av1*. For *avifenc*, we can see a 2.2% and 9.3% reduction in the average and best result, respective, in 75^{th} percentile of power usage. Secondly, a larger *ttdeT* value does not guarantee a proportional gain in power usage. Lastly, we can see that beyond a point, the power usage can actually increase. This can be attributed to two reasons: (a) more backlog accrued due to longer deadlines leads to more time spent at the beginning of each profiling event in just consuming that backlog and (b) the rate of backlog accrual not being considered to better judge the tolerance of an application.

VI. Related Work

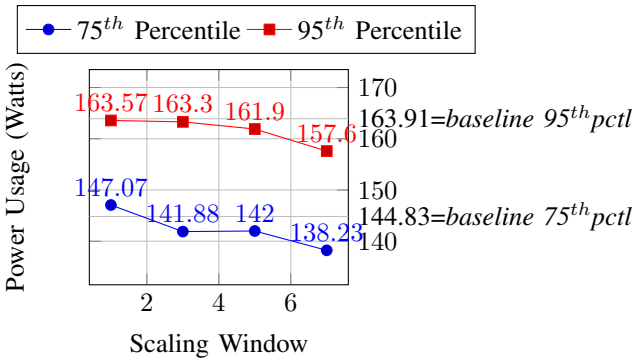
H. Mehta et al. [14] propose a method for power-aware container scheduling to minimize power cap violations on a server. Unlike [15] they perform container migrations and core reductions to achieve power-capping. They state that by doing this, the overall container performance is unaffected but only the container that violates the power-cap. J. Krzywda et al. [16] propose a method to reduce datacenter operational costs by redistributing the available power among applications hosted in the cluster. Three step approach - (a) the



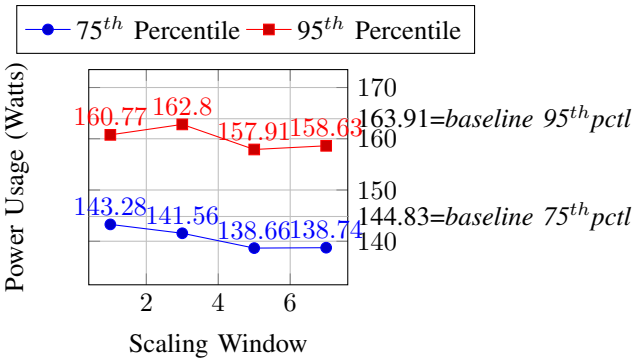
(a) app=svt-av1, ttdeT=3x, pw=4, bklTol=0.25



(b) app=svt-av1, ttdeT=3x, pw=2, bklTol=0.75

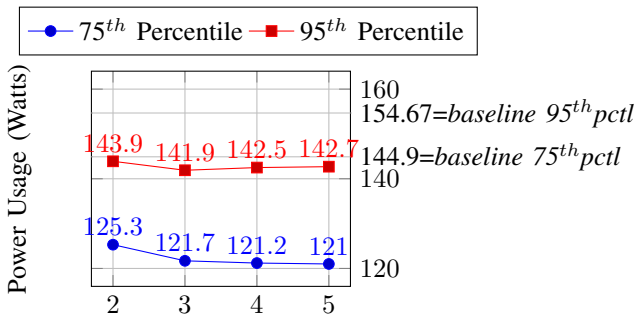


(c) app=avifenc, ttdeT=3x, pw=4, bklTol=0.25

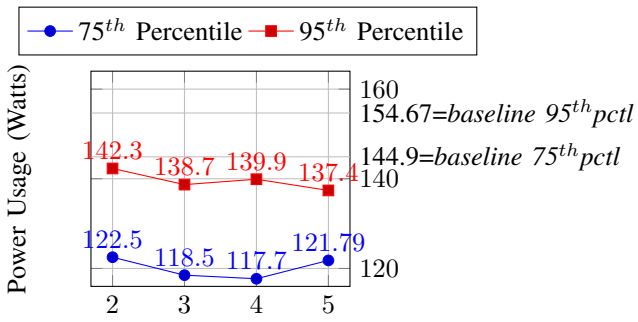


(d) app=avifenc, ttdeT=3x, pw=2, bklTol=0.75

Fig. 5: Influence of Scaling Window on power savings in two different scenarios: (5a, 5c) Large profiling window and low backlog tolerance and (5b, 5d) Small profiling window and high backlog tolerance. The baseline numbers show the power usage (in percentile) when the application is allocated all the requested resources and is not subject to vertical scaling.

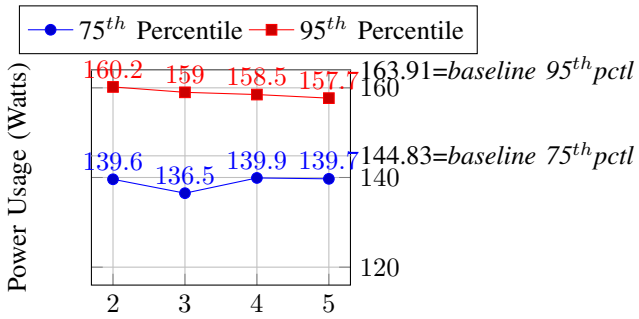


(a) Average Results

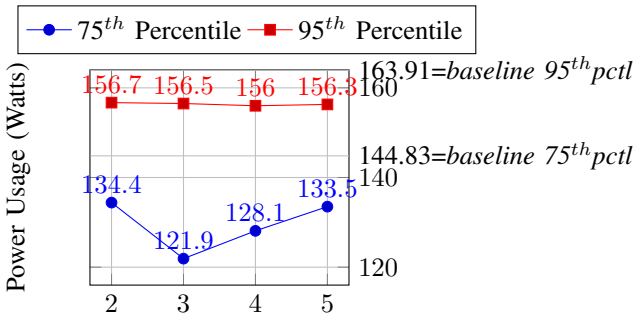


(b) Best Results

Fig. 6: Figure 6a shows the 75^{th} and 95^{th} percentile of power usage average across multiple experiments conducted with different values of $ttdeT$. Figure 6b show the best results of power usage across multiple experiments conducted with different values of $ttdeT$. These graphs are for app=svt-av1.



(a) Average Results



(b) Best Results

Fig. 7: Figure 7a shows the 75^{th} and 95^{th} percentile of power usage average across multiple experiments conducted with different values of $ttdeT$. Figure 7b show the best results of power usage across multiple experiments conducted with different values of $ttdeT$. These graphs are for app=avifenc.

cluster controller takes the cluster power budget and then allocates power to each server in order to minimize operation cost, (b) server controllers on each worker node enforce the power budgets (from the cluster controller) using RAPL and appropriately allocate the power to each application by adjusting cpu-quota and then provide future {power-budget, cost} offers back to the cluster controller. M. Arnaboldi et al. [17] proposed distributed Observe Decide Act (ODA) control loop able to introduce energy proportionality in a distributed and containerized infrastructure, while respecting SLAs (cpu resource requests from users who submit jobs to kubernetes). A. Asnaghi et al. [15] proposed a method to reduce performance impact of docker containers while maintaining a node within a power limit. They compare their results with RAPL (Running Average Power Limit [11]). The work proposed in this paper does not throttle/powercap nodes and instead just alters resource constraints of applications while still allowing the underlying CPU to operate unrestrained.

F. Rossi et al. [18] proposed a policy that combines horizontal and vertical scaling using reinforcement learning to adapt to varying workloads. Similar to their work, we also acknowledge the variability of application behavior but we do not employ horizontal scaling policies and assume that applications can run within a single container.

Power Tolerance was introduced in [19] as an indicator of a task's sensitivity towards performance degradation due to resource contention. Similarly, tolerance has been used in our work as a regulator of power/performance trade-offs. Note however that the work proposed in this paper deals with vertical scaling of deadline constrained applications.

R. Brondolin et al. [20] propose PRESTO, a latency (request latency in microservices) aware power-capping orchestrator. PRESTO defines an Observe Decide Act (ODA) loop to manage power consumption and average latency of microservice-based workloads by considering all the network interactions between microservices in the cluster. Fan et al. [21] proposed a mechanism to better the energy-latency trade-off by putting nodes in deeper sleep states based on the variations in the workload and system-wide utilization. Wang et al [22] proposed a scheduling algorithm that reduces the energy consumption and limits the increase in execution times of tasks on homogeneous clusters. Our work, however, focuses on stretching execution as much as possible without violating deadline constraints to obtain power savings opportunities.

VII. Conclusion

We studied four contributing factors to vertical scaling of deadline constrained applications viz; *time to deadline*, *profiling window*, *scaling window* and *backlog tolerance*.

- *profiling window* demonstrates an almost cyclic influence on the accuracy of application behavior prediction. Increasing the profiling window can affect the absolute percentage error in application behavior prediction by up to 33.7% and 4.7% for applications *avifenc* and *svt-av1* respectively. In addition, A larger profiling window leads to a smaller amount of time spent consuming backlog.
- *scaling window* can help govern the power envelop and facilitate power savings. Increasing scaling window can achieve up to a 9.6% and 6.3% reduction in 75^{th} percentile, and up to a 4.1% and 2% reduction in 95^{th} percentile of power usage for applications *svt-av1* and *avifenc* respectively.
- *backlog tolerance* influences application execution time and therefore proves viable in checking SLA violations. Reducing backlog tolerance allows for a complete reduction in number of SLA violations for $svt - av1$ and a 50% reduction in number of SLA violations for *avifenc*.
- *time to deadline* is the direct derivation of the deadline constraint. Increasing the time to deadline does provide more power saving opportunities where we witness up to an additional 9.3% improvement in 75^{th} percentile of power usage when *ttdeT* is increased from 2x to 3x the execution time. Interestingly, arbitrarily increasing *ttdeT* does not guarantee a proportional gain in power usage.

REFERENCES

[1] C. Delimitrou, C. Kozyrakis, C. Delimitrou, C. Kozyrakis, C. Delimitrou, and C. Kozyrakis, "Quasar: resource-efficient and QoS-aware cluster management," *ACM SIGPLAN Notices*, vol. 49, no. 4, pp. 127–144, 2014.

[2] R. Brondolin, M. Arnaboldi, and M. D. Santambrogio, "Power consumption management under a low-level performance constraint in the Xen hypervisor," *ACM SIGBED Review*, vol. 17, no. 1, 7 2020.

[3] "cAdvisor," 2020. [Online]. Available: https://github.com/google/cadvisor

[4] "Prometheus," 2020. [Online]. Available: https://prometheus.io/

[5] Pradyumna Kaushik, "Task-Ranker," 2021. [Online]. Available: https://github.com/pradykaushik/task-ranker

[6] Wikipedia, "Out of Memory," 1 2022. [Online]. Available: https://en.wikipedia.org/wiki/Out_of_memory

[7] James Steven Plank, Micah Beck, and Gerry Kingsley, *Libckpt : transparent checkpointing under Unix*. Knoxville: University of Tennessee, Computer Science Dept., 1994.

[8] Wikipedia, "Application Checkpointing," 10 2021. [Online]. Available: https://en.wikipedia.org/wiki/Application_checkpointing

[9] Merkel and Dirk, "Docker: lightweight Linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[10] "Performance Co-Pilot," 2016. [Online]. Available: http://www.pcp.io/

[11] "Running Average Power Limit – RAPL," 2014. [Online]. Available: https://01.org/blogs/2014/running-average-power-limit--rapl

[12] OpenBenchmarking.org, "phoronix-test-suite benchmark avifenc." [Online]. Available: https://openbenchmarking.org/test/pts/avifenc-1.1.1

[13] ——, "phoronix-test-suite benchmark svt-av1." [Online]. Available: https://openbenchmarking.org/test/pts/svt-av1-2.4.0

[14] H. K. Mehta, P. Harvey, O. Rana, R. Buyya, and B. Varghese, "WattsApp: Power-Aware Container Scheduling," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 12 2020.

[15] A. Asnaghi, M. Ferroni, and M. Santambrogio, "DockerCap: A Software-Level Power Capping Orchestrator for Docker Containers," in *2016 IEEE Intl Conference on Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES)*. IEEE, 8 2016.

[16] J. Krzywda, A. Ali-Eldin, E. Wadbro, P.-O. Ostberg, and E. Elmroth, "Power Shepherd: Application Performance Aware Power Shifting," in *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 12 2019.

[17] M. Arnaboldi, R. Brondolin, and M. D. Santambrogio, "HyPPO: Hybrid Performance-Aware Power-Capping Orchestrator," in *2018 IEEE International Conference on Autonomic Computing (ICAC)*. IEEE, 9 2018.

[18] F. Rossi, M. Nardelli, and V. Cardellini, "Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 7 2019.

[19] P. Kaushik, S. Raghavendra, M. Govindaraju, and D. Tiwari, "Exploring the Potential of using Power as a First Class Parameter for Resource Allocation in Apache Mesos Managed Clouds," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 12 2020.

[20] R. Brondolin and M. D. Santambrogio, "PRESTO: a latency-aware power-capping orchestrator for cloud-native microservices," in *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. IEEE, 8 2020.

[21] Yao Fan, Wu Jingxin, Subramaniam Suresh, and Venkataramani Guru, "WASP: Workload Adaptive Energy-Latency Optimization in Server Farms Using Server Low-Power States," in *IEEE 10th International Conference on Cloud Computing (CLOUD)*. Honolulu, CA, USA: IEEE, 2017. [Online]. Available: https://ieeexplore.ieee.org/document/8030586

[22] Lizhe Wang, Gregor von Laszewski, Jay Dayal, and Fugang Wang, "Towards Energy Aware Scheduling for Precedence Constrained Parallel Tasks in a Cluster with DVFS," in *CCGRID '10 Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. Washington, DC: IEEE Computer Society, 2010. [Online]. Available: https://dl.acm.org/citation.cfm?id=1845153