Twig: Multi-Agent Task Management for Colocated Latency-Critical Cloud Services

Rajiv Nishtala*, Vinicius Petrucci†, Paul Carpenter‡, Magnus Själander*

* Norwegian University of Science and Technology, Norway
{*rajiv.nishtala, magnus.sjalander*}*@ntnu.no*
‡ Barcelona Supercomputing Center, Spain
paul.carpenter@bsc.es
† Federal University Bahia, Brazil & University of Pittsburgh, USA
vpetrucci@pitt.edu

***Abstract*—Many of the important services running on data centres are latency-critical, time-varying, and demand strict user satisfaction. Stringent tail-latency targets for colocated services and increasing system complexity make it challenging to reduce the power consumption of data centres. Data centres typically sacrifice server efficiency to maintain tail-latency targets resulting in an increased total cost of ownership.**

This paper introduces Twig, a scalable quality-of-service (QoS) aware task manager for latency-critical services co-located on a server system. Twig successfully leverages deep reinforcement learning to characterise tail latency using hardware performance counters and to drive energy-efficient task management decisions in data centres. We evaluate Twig on a typical data centre server managing four widely used latency-critical services. Our results show that Twig outperforms prior works in reducing energy usage by up to 38% while achieving up to 99% QoS guarantee for latency-critical services.

I. Introduction

Large-scale online data-intensive services are increasingly pervasive across data centres and require consistently low response times to attract and retain users [1, 2]. These data centres increasingly colocate numerous services on the same node to improve cost efficiency [3]. Unfortunately, this causes shared resource contention among co-scheduled cloud services [4], leading to unpredictable performance degradation [5]. This is especially problematic for latency-critical (LC) cloud services as a marginal increase in service delay can greatly impact the user experience [6]. Ensuring consistent quality-of-service (QoS) is a challenging problem, especially in tandem with improved energy efficiency [7].

Managing shared-resource contention is a well-studied but still an open problem [5, 8–11]. Prior work has addressed this problem in two ways, by (a) disallowing resource sharing for LC services in periods of high load to avoid interference [12–19] or (b) disallowing colocation of services even if they are unlikely to interfere with each other [20–22]. Blindly applying either solution preserves the QoS of the LC service but results in low server-efficiency.

Heuristic-based techniques have been proposed to perform energy-efficient, task-mapping decisions for different resource allocations, including number of cores and dynamic voltage frequency scaling (DVFS) settings [3, 12, 13, 21, 23]. Nevertheless, heuristic parameters are highly specialised for the specific architecture/service, making these techniques difficult to adapt and generalise to new platforms. Reinforcement-learning (RL) techniques have been explored for enhanced QoS and resource-efficient task management [15] to improve adaptivity and generalisation. Still, previous techniques fail to scale on large server systems as they use a table mapping from specific states to actions, which grows exponentially in space and complexity.

A desirable solution would be to manage the resource allocation while being scalable and agnostic to the running service. Most processors include hardware-assisted performance monitoring counters (PMCs) that can be used to track several types of hardware events, such as instructions retired and cache misses at multiple levels. Such PMC data can be explored as general indicators to help understand service characteristics and to design an agnostic task-management solution. Nevertheless, it is far from trivial to leverage PMC data in real systems. Prior work has shown that conventional IPC-based (instructions per cycle) task-management mechanisms cannot be used for LC services as there is no clear relationship between IPC and tail latency [12].

In this paper, we empirically demonstrate that there exists a complex relationship between certain PMCs and the tail latency of LC services. Based on this insight we introduce *Twig*, an action-branching learning agent that is capable of learning this relationship, resulting in a scalable and energy-efficient task-management solution for colocated LC services. Twig assumes no prior knowledge of the service making it a quick *drop-in replacement* for existing task-management solutions. Twig exploits recent ideas in deep Q-learning algorithms and advances over state-of-the-art by leveraging PMCs instead of service-centric metrics to allocate resources while significantly reducing the required memory space.

We present Twig in two variants: *Twig-S* and *Twig-C*, which are targeted toward single and colocated LC services, respectively. Both variants aim at maximising energy efficiency while meeting the QoS target of the LC service(s). Twig-S is evaluated against Hipster [15] and Heracles [12], two state-of-the-art task managers for single services. Twig-S outperforms Hipster and Heracles in reducing energy usage on average by 11% and 38%, respectively while achieving up to 99.2% QoS guarantees. Twig-C is evaluated against PARTIES [3], the only other state-of-the-art task manager for colocated services. The results show that Twig-C outperforms prior work in reducing energy usage on average by 28% while achieving up to 98.9% QoS guarantees.

This paper makes the following main contributions:

1) We demonstrate that there exists a relationship between tail latency and a set of PMCs that can be explored to build a service-agnostic cloud task manager (Section II-A).
2) We introduce an extension to an action-branching dueling Q-network (BDQ) capable of coordinating multiple agents in a shared environment (Section III-A).
3) We present the design of **Twig**, an RL-based task-management solution that can dynamically coordinate and assign the best core mapping and DVFS settings for colocated LC services (Section III-B).
4) We demonstrate Twig's ability to dynamically adapt to new cloud services at runtime. Despite changes in the service and batch-workload mix, Twig delivers up to 99% QoS guarantee for the allocated LC services. We also demonstrate that Twig quickly learns to efficiently manage new LC services without prior knowledge of the service characteristics or system platform (Section V).

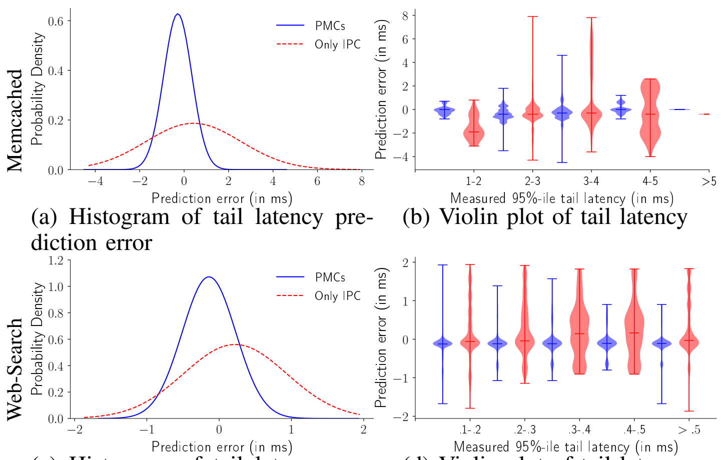
II. Motivation and Background

The problem of determining the best resource allocation (e.g., core allocation and DVFS setting) for each LC service over time requires solving two important challenges: (1) How to best characterise the LC service behaviour with minimal intervention at development or deployment stage? (2) How to design a task manager that can best partition the limited resources given the joint behaviour of multiple LC services interacting on a shared platform?

To address the first challenge, we propose using PMCs as a generic, non-invasive and scalable method to characterise the tail-latency behaviour of LC services running on a particular platform (Section II-A). Designing an energy-efficient task manager requires significant amount of exploration in the resource-allocation configurations, thus simple heuristics may fail to deliver the best resource-allocation decisions over time. We address this second challenge by exploring recent advances in deep reinforcement learning (Section II-B) that can enable a runtime system to automatically learn how to best allocate the resources to multiple LC services.

A. Characterising Tail Latency

When designing a task manager, it is important to be able to precisely and quickly determine a service's behaviour on a given server platform. To this end, we perform experiments to understand if tail latency can be estimated using deep RL as a function of multiple PMCs and using only IPC (Section III-B1 elaborates on system monitoring). Figure 1 shows the main result from our experiments. We ran two well-known cloud services: Memcached (top, figures (a) and (b)) and Web-Search (bottom, figures (c) and (d)) with the maximum number of cores available on the system and at the highest DVFS setting while varying the incoming load. The figures to the left show the prediction error as a probability density function and the figures to the right show a violin plot of the prediction error as a function of the measured tail latency. Each plot was generated with 30 000 samples.



(a) Histogram of tail latency prediction error
(b) Violin plot of tail latency
(c) Histogram of tail latency prediction error
(d) Violin plot of tail latency

Fig. 1: Exploring the relationship between multiple PMCs and tail latency for Memcached (top) and Web-Search (bottom). The left column shows the prediction error as a probability density function, and the right column shows a violin graph of the prediction error as a function of the measured tail latency.

We observe (Figure 1a) that the mean error in estimating the tail latency for Memcached using multiple PMCs is -0.286 ms with a standard deviation of 0.63 ms, while when using only IPC the mean error is 0.45 ms with a standard deviation of 2.13 ms. Similarly, Figure 1c shows that the mean error in estimating the tail latency for Web-Search using multiple PMCs is -0.132 ms with a standard deviation of 0.37 ms, while when using only IPC the mean error is 0.24 ms with a standard deviation of 0.72 ms. Note that the probability of zero prediction error increases by a factor of atleast 1.91$\times$ (3.36$\times$ best case) when transitioning to multiple PMCs from using only IPC as an input variable.

Next, each graph in Figures 1b and 1d is a violin plot showing the prediction error for a given tail latency range. The horizontal bar in the middle shows the median of the prediction error for that tail latency range. The width of the violin represents probability density of the error. When using multiple PMCs, it is clear that within each latency bucket the median prediction error and the highest probability density is around zero. However, when using only IPC, the probability density is spread across the error range indicating that IPC alone is insufficient to estimate the tail latency.

Hence, feeding PMCs to a learning agent using RL is a promising approach to design a service-agnostic task manager.

B. Reinforcement Learning

Q-learning is the most widely deployed RL algorithm, and applications based on this technique have recently achieved spectacular results [24]. A Q-learning agent learns to maximise its total reward in a dynamic environment [25], which is modelled as a Markov decision problem (MDP). Given the current state, $s \in S$ imposed by the environment, the learning agent must choose an action $a \in A$. Given the state s and action a, the environment then transitions to a new state, $s' \in S$ according to a probability distribution function, $P : S \times A \times S \to \mathbb{R}$. The agent also receives a reward according to the reward function, $R : S \times A \times S \to \mathbb{R}$. This procedure is iterated for some length of time, during which the agent must build a policy to maximise the total discounted reward.

A Q-function (Q) is used to learn the transition probabilities, and can be represented as a table of states and actions, with each entry $Q(s, a)$ representing the estimated total discounted reward when starting in state s and taking action a. The goal of the Q-function is to learn a policy that theoretically selects the optimal action for each state [26]. The state, which is a continuous value, is quantised into discrete buckets (b). The action can be a combination of numerous dimensions (D), where each action dimension $d \in D$ can have a discrete number of actions ($n \in N$).

Q-learning based agents have been applied for task management of LC services (as in Hipster [15]) by specifying the state as the load measured in requests per second (RPS), and the action as a combination of tunable hardware knobs. The straightforward way would be to translate this approach to our problem by quantising each PMC as part of the state, and action as a combination of tunable parameters. However, this requires maintaining a 2D-array of states and actions, with a total of $b \times D^N$ elements. The total size of the array grows quickly, leading to a *combinatorial explosion*, which considerably increases the learning time.

1) Deep Q-Network: A potential solution to standard Q-learning would be to replace the 2D-array that stores the quality of individual state–action pairs with a non-linear function approximator that approximates the quality of state–action pairs. This provides two main advantages: (1) it reduces the memory space for storing the state-action space and (2) it eliminates the need to explicitly traverse through each state–action pair to understand the quality of an action in that state.

A well-known non-linear function approximator that can estimate the Q-function is a deep Q-network (DQN) [24]. In a typical DQN architecture, the final layer is a soft-max function, which gives a probability distribution, and requires the network to select a single action, i.e., the action with the highest Q-value.

A desirable task manager for colocated LC services should effectively manage several action dimensions (e.g., number of cores, DVFS settings, etc.), and deploying vanilla DQNs means that a single instance requires combinations of actions, leading to an action-space combinatorial explosion.

2) Branching Dueling Q-network: One way to tackle the combinatorial explosion is by maintaining multiple DQNs, one for each action dimension [27]. However, this results in a lack of coordination between the different actions since each action is selected independently without considering the global outcome of all actions, as demonstrated by Tavakoli et al. [28]. As a solution, recent works propose action-branching architectures to overcome the combinatorial state explosion of a single learning agent.

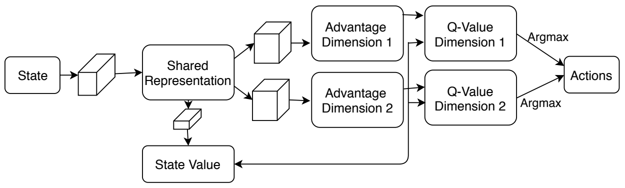


Fig. 2: BDQ architecture by Tavakoli et al. [28].

An *action-branching architecture* enables each action dimension to have a non-uniform number of discrete action outputs that can be selected. At runtime, when a decision is requested, the agent will select one action per branch, while maintaining a shared representation of the state space. An action branching architecture is based on a *branching dueling Q-network* (BDQ) and a Dueling DQN [27]. For more information, we refer the reader to prior works [27, 28].

BDQ Architecture. Figure 2 shows the BDQ architecture. As in DQN, the neural network is responsible for estimating the Q-value, which specifies the total discounted reward obtained by first selecting a certain action given a certain state. In the BDQ architecture, the Q-value is split into the advantage dimension and the state value. The advantage dimension determines how much better it is to take a certain action relative to other actions. The state value is independent of the action, and it represents to what extent the state is a good one in general. Separating the state value and advantage dimension enables the agent to determine whether to make an action or stick with the current action. For example, imagine driving a car on a straight road. It is only important to change the direction (action) if there is an obstacle in front of you. The BDQ architecture includes a *shared representation* of the state, the state value, the advantage dimension for each action (branch), and the combined Q-value dimension resulting in the actions. The shared state module has a representation of the input and helps with the coordination among branches. Each branch represents an action and has a discrete action space, potentially with different sizes. The state value generated using the shared representation is fed to each branch to determine the Q-value, and thus the final action taken by the agent.

III. Twig

This section introduces Twig, a deep RL-based solution for task management of colocated LC services. Twig leverages a novel multi-agent BDQ architecture that takes PMCs as input to build an function approximator for the tail latency of running LC services to deliver best mapping decisions. The design goal of Twig is to maximise the energy efficiency subject to meeting the QoS target of the LC services.

A. Multi-Agent BDQ Architecture

A coordinated management of multiple tunable hardware knobs for a single LC service can be efficiently done using a classic BDQ architecture. However, coordinating the resource management across several LC services requires multiple agents interacting with each other to solve a single objective function, i.e., to meet the QoS constraints of all LC services while minimising the energy usage.

We introduce a novel multi-agent BDQ architecture consisting of state agents that derive a state value for each of the learning agents. Multiple agents acting simultaneously would otherwise affect the learning process of each other. For the example shown in Figure 3, which is capable of managing two LC services, we include two state agents to obtain the state values and the shared representation. The state value of each learning agent is added individually to the advantage dimension to determine the Q-value dimension for that agent. The common advantage dimension for each branch across numerous learning agents enables shorter learning periods [27]. The loss is computed as the mean squared error across each branch per agent. Since all Q-values have to pass through the advantage dimension during the backpropagation, we rescale the combined gradient prior to entering the deepest layer in the advantage dimension by one over the number of learning agents. Similarly, we rescale the combined gradient for the shared representation by one over the number of dimensions.

B. The Twig Task Manager

Figure 3 shows a high-level view of Twig. Its three main components are: (1) *System monitor*, which is responsible for gathering PMCs; (2) *Learning agent*, which is responsible for learning the best task management decisions based on available resources; (3) *Mapper module*, which is responsible for allocating tasks to cores and setting the DVFS state.

1) System Monitor:

The system monitor is responsible for periodically gathering the PMCs at a *per-thread* level using a profiling tool to measure the activity of each LC service. For each service, we sum the PMCs across all its threads. To reduce the noise over time, a weighted sum for each aggregated counter is computed over the last η time steps.

We reduce the number of required PMCs through a systematic approach to maximise the correlation with the tail latency, while minimising redundant counters. We run each LC service for $1000\,\text{s}$ at each DVFS and core combination while gathering all available PMCs at a fixed sampling interval (1-second). The Pearson correlation is used to build a correlation matrix between PMCs and tail latency [29]. Thereafter, the number of principal components is chosen such that there is at least a 95% co-variance. Finally, principal component analysis [30] is performed to determine the most vital and distinct PMCs that capture the tail latency. This is similar to the methodology of Malik et al. [31]. The selected PMCs are *feature scaled* to have values in the range $[0, 1]$. The data is scaled using max-value normalisation with non-zero centralisation. Feature scaling enables the neural network to capture the importance of each state variable equally.

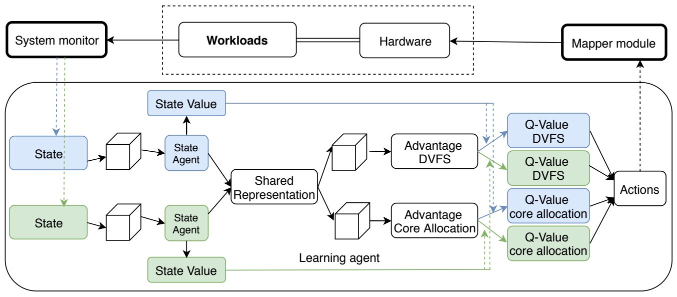


Fig. 3: Twig runtime with multi-agent BDQ. The individual agents are represented in green and blue.

Algorithm 1 Twig's Deep Q-learning

```
1  Let $n = 0$
2  Let $K$ be the number of services
3  Let prediction function $Q$ with weights $\beta$
4  Let $\vec{s} = \{s_1, \ldots, s_K\}$ be the observed state for interval $t_{n-1} \ldots t_n$
5  Let $\vec{a} = \{a_1, \ldots, a_K\}$ be the action (core mapping + DVFS) for interval
   $t_n \ldots t_{n+1}$
6  **repeat**
                                   ▷ *At time $t_n$, choose action for $t_n$ to $t_{n+1}$*
7    With probability $\epsilon$ select a random action
8    Else, select $\vec{a} = \arg\max_{\vec{a}'} Q(\vec{s}, \vec{a}'; \beta)$
9    Sleep until $t_{n+1}$                  ▷ *Run for interval $t_n$ to $t_{n+1}$*
10   **for** $k \in K$ **do**
11     Observe new state $s_k$ of each service $k$
12     Compute the reward $r_k$ based on Equation 1.
13   Update weights $\beta$ by computing the back propagation.
14   $n = n + 1$
15 **until** Terminated
```

2) Learning Agent:

The learning agent, which is based on the multi-agent BDQ architecture, learns the "optimal" decisions over time by interacting with the environment using the *exploration–exploitation dilemma* [25]. In the exploration–exploitation dilemma, the agent not only captures the need to exploit the "optimal" solution found so far but also explores actions that may or may not be better. The probability to explore rather than exploit is captured by epsilon (ϵ). While having a fixed yet small ϵ is the dominant approach in pure RL settings, it becomes infeasible in large action spaces. Twig instead uses epsilon annealing, which transitions from an exploratory policy to an exploitative policy over time for efficient exploration of the discrete action domain [32]. The agent's interactions with the environment at each timestep are driven by gathering the state variables, generating actions that are either deterministic or random. The agent then receives a reward in the following timestep, determins how well the agent did in the previous timestep.

Twig solves the task management problem by translating it to a Markov decision problem (MDP) that is then solved by a multi-agent BDQ. The pseudo-code for the multi-agent BDQ is shown in Algorithm 1. We instantiate a single multi-agent BDQ for all LC services executing on the server. Let K be the total number of agents/services. The algorithm starts by initialising the prediction (Q) with weights β (line 3). Next, we observe the state s_k as represented by the PMCs for each service within the time interval t_{n-1} to t_n, and the initial action a_k, i.e., the mapping configuration taken for time interval t_n to t_{n+1} (lines 4–5). Thereafter, at each time step, we determine the mapping configuration for each service either stochastically (line 7) or deterministically (line 8), with the $\arg\max$ implemented using the multi-agent BDQ network.[1] This configuration is allocated to each service for the time interval t_n to t_{n+1}, after which the next state (s_k) is observed (i.e., the PMCs gathered for the time interval t_n to t_{n+1}) and the reward is computed for each service (lines 11–12). The loss incurred due to the current prediction is then computed and the weights of the neural network are updated through back propagation (line 13).

Reward Function. The Twig reward mechanism determines the mapping decisions based on PMCs and is invoked periodically at each monitoring interval. The reward is computed per service ($r_k,\ k \in K$) and it aims at minimising the power consumption subject to meeting the QoS target, and is expressed as follows:

$$r_k = \begin{cases} QoS_{\text{rew.}} + \theta \times Power_{\text{rew.}} & QoS \leq QoS_{\text{target}} \\ \max\left(-QoS^{\phi}_{\text{rew.}}, \varphi\right) & QoS > QoS_{\text{target}} \end{cases} \quad (1)$$

QoS Reward. The ratio of the measured QoS to the QoS target is known as the $QoS_{\text{rew.}}$. If this value is less than or equal to 1, then the QoS target has been met, and quantifies how quick the response was. If this value is greater than 1, then the QoS target has been violated, and therefore we severely penalise the learning agent. As a precursor to ensure that the negative reward is bounded, we cap it to a prefixed value (φ).

Intuitively, the only reward that the learning agent should receive is the power reward (if QoS is met) and a large negative value (if QoS is not met). The part of the reward related to QoS is a heuristic to encourage the algorithm to choose configurations that just meet QoS, which are likely to minimise power consumption (if QoS is met) and tries to reduce the latency in finding an acceptable solution (if QoS is not met).

Power Reward. With multiple LC services running in the system, it is essential for Twig to know the power consumed per service to provide a precise reward for each agent. The *power reward* ($Power_{\text{rew.}}$) is given by the ratio of the maximum measured power consumption to the estimated power consumption of the particular service. A larger value for this term implies that the service's power consumption is lower, and a higher value is added for power savings. The maximum system power consumption is given by running a stress microbenchmark that has no memory accesses. The parameter θ controls the balance between meeting the QoS and reducing the power consumption.

3) Mapper Module: The mapper module has three key roles. (1) Receive resource allocation request of each service from the learning agent. (2) Ensure that services are mapped to cores and set the DVFS state. The remaining cores, if any, are set to the lowest DVFS state to conserve power. (3) Prioritise the order of the cores for each service to improve cache locality. For example, if two services (sv-1 and sv-2) are running on the same CPU with a total of 16 cores and they request three cores at 1.6 GHz and four cores at 1.8 GHz, respectively. Then, the mapper module allocates cores 0, 2 and 4 for sv-1 and cores 10, 12, 14, and 16 for sv-2.

[1]For ease of reading, we simplify the algorithm, but as in [28], there are two networks with the same initial weights that are updated periodically.

TABLE I: State variables that are part of the MDP formulation. Each variable is summed across all LC service threads. Boldfaced counter has the highest importance.

#	Counter name	Range	Importance
1.	UNHALTED˙CORE˙CYCLES	$[0, 1]$	10
2.	INSTRUCTION˙RETIRED	$[0, 1]$	6
3.	PERF˙COUNT˙HW˙CPU˙CYCLES	$[0, 1]$	9
4.	UNHALTED˙REFERENCE˙CYCLES	$[0, 1]$	11
5.	UOPS˙RETIRED	$[0, 1]$	7
6.	BRANCH˙INSTRUCTIONS˙RETIRED	$[0, 1]$	3
7.	MISPREDICTED˙BRANCH˙RETIRED	$[0, 1]$	8
8.	PERF˙COUNT˙HW˙BRANCH˙MISSES	$[0, 1]$	**1**
9.	LLC˙MISSES	$[0, 1]$	2
10.	PERF˙COUNT˙HW˙CACHE˙L1D	$[0, 1]$	4
11.	PERF˙COUNT˙HW˙CACHE˙L1I	$[0, 1]$	5

IV. Twig Implementation

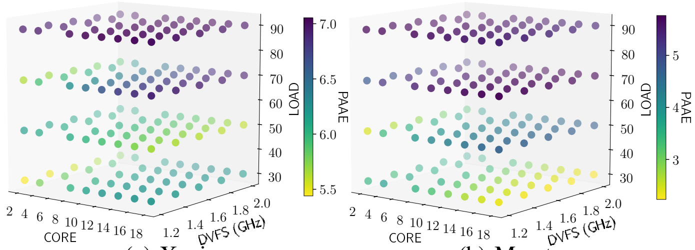
Twig is implemented in user space, and it only uses hardware support exposed by the Linux kernel. The hardware dependent components are PMCs measurement and selection, power and QoS measurements, and the mapper module.

PMCs Measurement and Selection. The PMCs are measured using the performance monitoring tool perfmon (libpfm 4.10.0) [33]. Table I shows the PMCs [34] that were selected for the evaluation of Twig by following the process outlined in Section III-B1. The 4^{th} column shows the importance value of each counter. The boldfaced value has the highest importance. We used $\eta = 5$ (Section III-B1) as empirically it yielded the best results. The maximum value for counters 1–5 was obtained by running a CPU-intensive microbenchmark consisting of several mathematical operations without memory accesses; counters 6–8 was obtained by running a microbenchmark that generates numerous branch misses by aggregating elements from an unsorted vector of data to check if they are greater than a certain value; counters 9–11 were obtained by running the stream benchmark [35].

Power Model/Measurements. A straightforward approach would be to collect the power measurements per core and aggregate them across the allocated cores [36]. This is not possible on current architectures as they *only* provide power measurements at a per socket level [37]. For this reason, we build a first-order model to enable each agent to distinguish the power consumed for the actions it requested. We estimate the power consumed per service using a simple polynomial model based on three metrics: load (as a percentage of the max load), number of cores and DVFS state.

$$Power_{app} = \kappa \times load + \sigma \times num_{cores} + \omega^2 \times DVFS \quad (2)$$

To build the model, we do extensive profiling of two services at three load levels (20%, 50% and 80% of the maximum load), core (alternate number of cores) and DVFS states (alternate DVFS states), and measure the dynamic power consumed every second for the cores allocated. The unused cores are disabled using CPU hot-plugging. We define



(a) Xapian (b) Masstree

Fig. 4: Percentage Absolute Average Error (PAAE) when estimating power consumption per service.

dynamic power as the difference between the current power consumption and power consumption when idle. The obtained model has a mean squared error of 2.91 mW and R^2 of 0.92. The model was built by performing a random grid search with 5-fold cross validation across the possible parameter space to obtain the model co-efficients [30].

Figure 4 shows the percentage absolute average error (PAAE) in estimating the power consumed at each load level, number of cores and DVFS state for Xapian and Masstree (from Tailbench [38]). As shown, the mean PAAE across services is 5.46% (7% max). As a first-order approximation, the error rate is sufficiently low for the learning agent to understand the cost of requesting for a specified number of cores at a given DVFS state.

The power model is used *only* as part of the reward function during training. The evaluation results in Section V report the true power consumption measured using the running average power limit (RAPL) [37] register, accessible via the MSR register [39]. The RAPL register is polled at the same polling interval as the LC service.

QoS Measurements. As a proof-of-concept, the tail latency is measured using a log-file interface from the LC services, at a fixed polling interval. An alternative to the log-file would be to gather the end-to-end latency via the network interface card (NIC), and compute the latency distribution [40]. In a production system, end-to-end latency can be obtained by taking advantage of features of advanced NICs and low-latency networking stacks.

Mapper Module. The services are mapped to cores using the Linux sched_setaffinity system call. We select the userspace govenor via the acpi-cpufreq module and then control DVFS according to our mapping algorithm.

Resource Arbitration. Within a multi-agent learning system, resource request conflicts are inevitable. When the number of cores requested by both agents exceeds the numbers of available cores, we determine how many overlapping cores are requested and select the highest DVFS state among the requested DVFS states for the overlapping cores. These cores are timeshared by both the services. The remaining cores are set to the DVFS state requested by the learning agent. For example, consider a socket with 10 cores hosting two services (sv-1 and sv-2), and if sv-1 needs 8 cores at 1.2 GHz, and sv-2 needs 5 cores at 2 GHz. Then, we set the first 5 cores to 1.2 GHz, and the remaining cores to 2 GHz.

Neural Network Parameters. We determine through experimental analysis [41, 42] that the following hyper-parameters have yielded the best energy efficiency while improving the QoS guarantee. We use the Adam optimiser [43] with a learning rate of 0.0025. We set the minibatch size to 64 and the discount factor to 0.99. The target network was updated every 150 time steps. The epsilon annealing starts at 1 and drops to 0.1 over a period of 10 000 s and drops to 0.01 in 25 000 s. We used the rectified non-linearity (ReLU) [44] for all hidden layers and linear activation for output layers. The network has two hidden layers with 512 and 256 units in the shared module and a single hidden layer per branch with 128 units. We add a dropout layer [45] after each fully connected layer with the probability rate set to 50% (default in tensorflow) to prevent over-fitting. We used the prioritised experience replay [46] with a buffer size of 10^6 and pr_α=0.6, and linear annealing of pr_{β_0}=0.4 to 1 over 10^{-8} steps. For the remainder of the paper, we refer to the first 10 000 s as the learning phase. The parameters θ, ϕ, and φ of the reward function were determined empirically and set to 0.5, 3, and –100, respectively, which yielded the best energy efficiency while improving the QoS guarantee. The source code is released under General Public License (GPL) v3 and it is available for download [47].

Transfer Learning. To reduce the learning time for a different problem in the same domain, we use a state-of-the-art technique called transfer learning [48]. Transfer learning works by removing the last layer of a trained network (the most dominant layer) as it is specialised to a given problem and then re-initialising the last layer with random weights to retrain for a short interval. This enables the multi-agent BDQ to learn new problems quickly.

V. Evaluation

Hardware Resources. We perform the evaluation of Twig on the NTNU EPIC compute cluster [49]. Each node runs Linux kernel 3.10 and contains two Intel Xeon E5-2695v4 sockets that together comprise 36 cores and dual NVidia Tesla P100 GPUs. Each core is capable of frequency scaling from 1.20 GHz to 2.00 GHz with steps of 0.1 GHz. The server contains 128 GB of DDR4-2400 GHz RAM. Hyperthreading was disabled as in most production servers.

Benchmarks. We evaluate Twig using four widely deployed LC services from Tailbench suite [38]. The LC services are Masstree [50], Xapian [51], Moses [52], and Img-dnn [53]. We use the default dataset provided by Tailbench. As prescribed by Tailbench, we use *loopback configuration* for the experiments. In this setup, client and servers are launched on different sockets on the same node. This allows us to accurately report the request processing times from the server side while not experiencing unpredictable network interference. Yet, this methodology captures majority of the overhead introduced by the network stack [38].

We specify the QoS targets and maximum incoming load according to the capacity and characteristics of our platform. We run each service consecutively by increasing the incoming

TABLE II: Services from TailBench [38].

Services	Masstree	Xapian	Moses	Img-dnn
Max load (RPS)	2,400	1,000	2,800	1,100
Target QoS (ms)	1.39	3.71	6.04	5.07

load step by step until the latency increases exponentially. We perform this experiment without any external interference while pinning the server application to all cores on a socket running at the highest DVFS setting. Table II summarises the maximum load, and the 99^{th} percentile target latency.

Overhead. The overhead of triggering Twig every second, as in our experiments, incurs an overhead of $<5\,\%$ in the worst case, as shown in Table III. The gradient descent computation (includes I/O) was implemented in Python using TensorFlow [54]. The computational complexities of changing cores and DVFS states are in the order of microseconds [12] and nanoseconds [55], respectively. A large percent of the core allocation/DVFS overhead is due to the sysfs call. Once Twig has seen sufficient experiences, we recommend pure exploitation i.e, dropping gradient descent computation, to reduce the overhead to under $< 1\%$.

Evaluation Metrics. The metrics for LC services are: *QoS guarantee* and *QoS tardiness*. QoS guarantee is defined as the percentage of measured QoS samples that met the QoS. QoS tardiness is defined as the ratio of measured QoS to the QoS target, and it determines how intense the violation was. A QoS violation has occurred if the QoS tardiness is above 1.

A. Baseline Comparisons

We compare Twig-S with static baseline and single LC service task managers: Hipster [15] and Heracles [12]. Similarly, we compare Twig-C with static baseline and multiple LC service task manager: PARTIES [3]. Each experiment begins by setting all cores to 2 GHz, and then launching the services. In case of Twig-S, we allocate the client and server on sockets zero and one, respectively, and refer to this as static baseline. Similarly, in case of Twig-C, we allocate all clients on one socket, and all the servers on the other socket. We implemented PARTIES and Heracles based on available documentation as they are not available as open source.

Hipster is a hybrid RL algorithm that combines heuristics with RL to determine mapping decisions based on the current load of the service. The heuristic explored by Hipster is a state-machine based algorithm that orders the mapping configuration (cores and DVFS) in increasing order of power efficiency [13]. A transition between states occurs when the tail latency is too close or too far away to/from the target. The current load is quantised into multiple buckets as part of the state. The action from the learning agent is a mapping configuration for the LC service. As recommended by the authors of Hipster [15], we set Hipster's bucket size using an exhaustive sweep, with jumps of 10%, to find the best trade-off between energy usage and QoS. The other parameters were set as indicated in the Hipster publication. We therefore set the learning rate to 0.6, the discount factor to 0.9, the bucket size to 4% and the learning phase to 7500 seconds.

TABLE III: Overhead of Twig

1	Gradient descent computation on GPU/CPU	25 ms/48 ms
2	Gather and pre-process PMCs	2 ms
	PMCs datasize per service	352 B/s
3	Core allocation & DVFS change	7 ms
	Total overhead with GPU/CPU	34 ms/57 ms

Heracles is a feedback controller that aims to maximise the system throughput subject to meeting the QoS of the LC service. Heracles maintains three levels of the feedback controllers: main, core and memory and power controller. The main controller is polled every 15 s and allocates all resources to the LC service for a period of 5 min, if the LC service either violates the tail latency or if the load is higher than 85%. The core and memory controller is polled every 2 s, and is responsible for allocating cores and memory resources to the service. If the tail latency equals or exceeds 80% of the QoS target or if the measured memory bandwidth has increased, then the LC service is allocated an additional core. Otherwise, a core is de-allocated from the LC service. In addition to core allocation, Heracles explores Intel cache allocation technology (CAT) [56], but we were unable to experiment with this technology in our production servers. The power controller is polled every 2 s, and is responsible for decreasing the DVFS setting when the current power is at 90% of TDP.

PARTIES is a feedback controller that aims to improve the system throughput, subject to meeting the QoS of LC services. PARTIES controls one resource at a time periodically (every 2 s) in the following order: core count, Intel CAT (not used in our experiments), DVFS, and memory allocation. PARTIES' controller begins by randomly selecting one of the resources and identifying those services that are closest/furthest to/from the target. If the tail latency equals or exceeds 95% of the target, it increases one of the control resources, otherwise it starts reclaiming resources from the service with the highest slack. In the reclaiming process, it reduces one resource at a time ensuring QoS is not violated. If the QoS is violated as a consequence of reducing that particular resource, it reverts the adjustment and adjusts another resource next time.

To ensure a fair comparison with the learning algorithms, we determine the energy usage, QoS guarantee and QoS Tardiness after the first 10 000 s, allowing Twig and Hipster to gain sufficient experiences. We summarise the results for Twig-C and PARTIES over the last 600 s, as PARTIES has a sampling interval of 2 s. For the remainder of the experiments, we summarise over the last 300 s.

B. Twig Evaluation

This section evaluates Twig in two variants: Twig-S, when LC services are running solo and Twig-C, when LC services are colocated. For Twig-S, we inject each service with either 20% (low), 50% (mid) or 80% (high) of the maximum load. Similarly, for Twig-C, we run all N-combinations of services, for a total of ${}^{N}C_2$ combinations, at low, mid and high load. Finally, both variants are evaluated with a diurnal load variations which are common in data centres [57]. The objective of both variants is to maximise the energy efficiency subject to meeting the QoS target of the LC services.

1) Twig-S: Single LC service: **Fixed Load**. Figure 5 shows the average QoS guarantee (top) and normalised energy usage (bottom) for each service over low, mid and high incoming loads with Twig-S, Hipster, Heracles and static mapping. All results are normalised to static mapping. The graph shows that all task managers deliver similar QoS guarantee, while Twig-S reduces energy usage by 11.8% and 38% (on avg.) across all services than Hipster and Heracles, respectively. There are a few reasons for this:

- Heracles migrates across cores based on two metrics: tail latency and memory bandwidth utilisation. In addition, it only reduces the DVFS state once the power consumption reaches the TDP. This causes Heracles to allocate more cores than required despite having a large slack in QoS.
- Heracles allocates all cores to the LC service for a period of 5 min, if it incurs a short-term violation even in periods of low load and as a result it has high energy usage.
- Twig-S incurs 2.3× fewer migrations compared to Hipster, as it reduces harmful oscillatory behaviour by understanding the impact of varying the number of core and DVFS settings individually.

To better understand why Twig-S reduces energy usage, we look at the specific resource allocations for Heracles, Hipster and Twig-S for Masstree at 50% of the maximum load in Figure 6. Masstree is extremely sensitive to memory bandwidth interference, although it does not use much in itself [3]. Heracles, for instance, oscillates between 12 or 13 cores at 2 GHz to increase memory bandwidth and to maintain the tail latency at 85% of the QoS target. Hipster, on the other hand, uses just 6 cores at 2 GHz for the majority of the time and has a QoS guarantee of 80.67%. The drop in QoS guarantee is a result of Hipster not considering detailed service related features (e.g., in this case, memory bandwidth) as part of its input state. Twig-S, on the other hand, understands the tail latency sensitiveness of each service to core and DVFS changes through multiple PMCs, and allocates mapping decisions to meet the QoS target. The violations observed ($<$ 4%) are a result of the random exploration which diminishes over time (see Section III-B2).

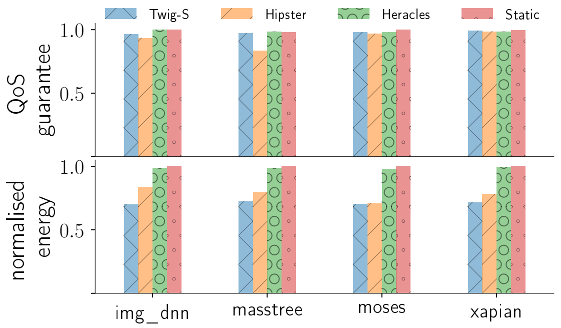


Fig. 5: Heracles, Hipster and Static mapping when executing over a fixed load of 20%, 50% and 80% for Twig-S. The top and bottom graph show the QoS guarantee and energy usage normalised, respectively.

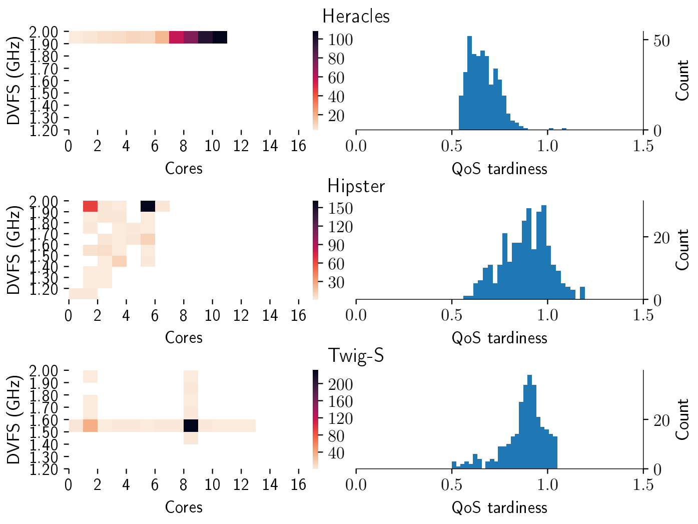


Fig. 6: Core mapping decisions (left) and histogram of QoS tardiness (right) with Heracles (top), Hipster (middle) and Twig-S (bottom) for masstree at 50% of maximum load. The colourmap represents the time distribution of core allocation over a period of 300 seconds.

Learning Time Complexity. Figure 7 shows the QoS guarantee over time for Masstree with Hipster and Twig-S. For Twig-S, we anneal the epsilon to 0.1 in 5000 s and for Hipster, the learning phase ends at 5000 s. Each data point in the graph refers to a period of 500 s. As can be seen, Hipster has a better QoS guarantee than Twig-S for the first 5000 s. This is because Hipster has the prior understanding of the power efficiency for each possible action dimension combination, which becomes infeasible in large scale servers as it requires extensive and exhaustive prior knowledge. As a result, to improve the QoS guarantee and reduce energy usage, Hipster needs to explore each action multiple times to improve its confidence in that action. Contrary to Hipster, Twig-S does not require *any* prior knowledge of the server system and learns the impact of each action dimension individually, thereby, delivering an improved QoS guarantee (more than 80%), faster than Hipster.

Memory Complexity Impact. Scaling a task manager to handle several actions requires frugal memory usage. In this context, we demonstrate the memory usage of Hipster and Twig-S for a server with three action dimensions ($D = 3$) and each dimension containing 30 discrete actions ($N = 30$, e.g., 30 DVFS settings, 30 cores, and 30 different cache allocation schemes). For Hipster, the state metric, RPS, is quantised into buckets of 4% resulting in 25 buckets (b). This results in a total of 25 $\times$ 3^{30} array entries with a memory usage in the order of TBs. For Twig-S, 11 state variables are used. When using Twig-S, there is a fixed memory complexity for understanding the raw state space and, thereafter, increases linearly with the number of dimensions in each action. With three action dimensions and the number of actions per dimension set to 30, the memory complexity is under 5 MB. Despite Twig-S

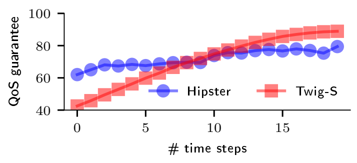


Fig. 7: Learning time complexity for Hipster and Twig-S. Each data point represents the average over 500 s.

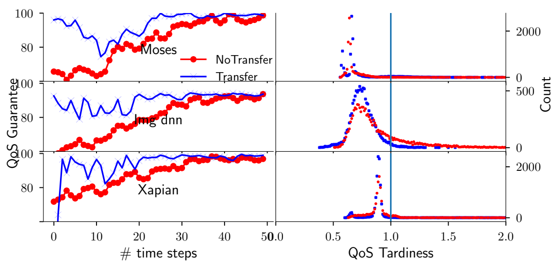


Fig. 8: Comparing performance of Twig-S with and without Transfer learning.

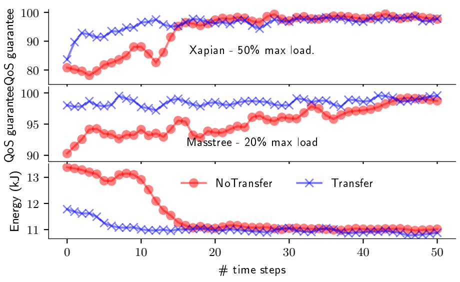


Fig. 9: Comparing performance of Twig-C with and without Transfer learning.

reducing the memory complexity significantly compared to Hipster, note that Twig-S delivers high QoS guarantees and minimises energy usage. This is because, Twig-S uses a function approximator to generalise the state-action importance instead of traversing through each state-action pair.

Transfer Learning. We use transfer learning with Twig-S to reduce the learning time for new, incoming services. To demonstrate the effectiveness of transfer learning, we learn the weights of the neural network with Masstree for 10 000 s, and then transfer them in consecutive experiments to Moses, img-dnn and Xapian. Each service is run at 50% of the max. load. Figure 8 compares QoS guarantee (left), and QoS tardiness for each service with and without transfer learning. Each data point in the graph refers to a period of 300 s. This graph shows two key points: (a) transfer learning reduces the learning time by 33.33% in contrast to learning from scratch while delivering high QoS guarantees. (b) transfer learning delivers similar QoS tardiness as learning from scratch, demonstrating that the configurations chosen aim to minimise the energy usage.

Varying Load. Figure 10 compares the performance of Twig-S, Heracles and Hipster when varying the load for img-dnn. When varying the load, we use a step-wise monotonic function, where the average load for the service is constant across two load changes, which can occur every 200 s, based on the change factor (set to 20%). The load starts at a minimum, and varies by multiplying with the change factor until it reaches a max. load; thereafter, the load is multiplied by the reciprocal of the change factor until it reaches the min. load. We report the results after the first 10 000 s.

From Figure 10, it is clear that Hipster fails to allocate the best mapping decisions at high load, and the reason for this is trivial. Hipster starts with a heuristic, and then transitions to RL approach after the learning period has ended. The learning period is set to 10 000 s (see Section V-A). Hipster uses the heuristic to determine the "optimal" mapping decisions for each load level. Given that there are 180 mapping decision (18 cores, and 10 DVFS states), Hipsters' heuristic spends significant time jumping between mapping decisions to determine "optimal" decision at each load-level. The transitions between mapping decisions does not have an effect on the QoS at low loads as there are few queued requests, and therefore this may impact the runtime overhead and not the QoS. Heracles, on the other hand, maintains 100% QoS guarantee by varying the core configuration with a fixed DVFS state but suffers from 2.3× higher migrations and 18% higher energy usage relative to Twig-S. Looking at Twig-S, it adjusts the number of cores and DVFS periodically to meet the QoS just about while have a QoS guarantee of 99.1%.

Hipster fails with varying loads because it uses a heuristic that is not able to adapt quickly enough to the incoming load.

2) Twig-C: Colocated LC services: **Fixed Load**. Figure 13 shows colocation of two LC service mixes with Twig-C, PARTIES and static. Within each graph, the top graph represents the QoS guarantee, and the bottom graph represents the energy usage normalised to static mapping. The x-axis refers to the load level normalised to the maximum load each service can operate at while meeting the QoS. The last bar refers to the average QoS guarantee and energy usage across all load combinations. Each bar in the graph refers to a task manager at a specific load level. In general, each service alone can meet the QoS at the highest load, but when colocated with another service it runs at a fraction of its maximum load (typically around 60%). To determine the maximum load each service can operate at when colocated with another service, we do an offline sweep of all service combinations in steps of 10% load increments. Our results show that Twig-C reduces energy usage over PARTIES by 28% on average. There are a few reasons for this:

- PARTIES does not deallocate resources from the service that is far away from the target until the workloads' closer to the target have met the QoS. Twig-C handles both services simultaneously.
- PARTIES ping-pongs across mapping decision as it does not anticipate if a service might violate the QoS, Twig-C is able to do this by understanding the pressure on individual computational components in relationship to the other service and thus having a stable mapping decision.
- PARTIES adjusts a single resource at a time, and has no collective module to understand the resource modification change on the other resources or services. Twig-C on the other hand, maintains a coordinated shared representation across services to deliver consistently high QoS while reducing energy usage.

We use the mapping distribution for pairs of services to understand why Twig-C reduces the energy usage. The colocation of Masstree and Moses is a particularly interesting

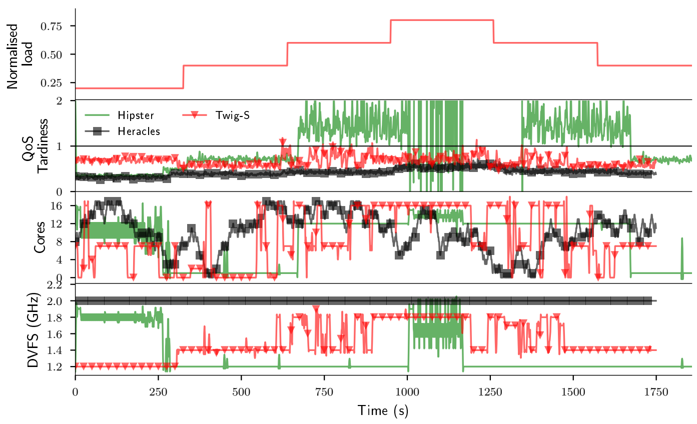


Fig. 10: Resource allocation with Twig-S, Hipster and Heracles with varying load for img-dnn.

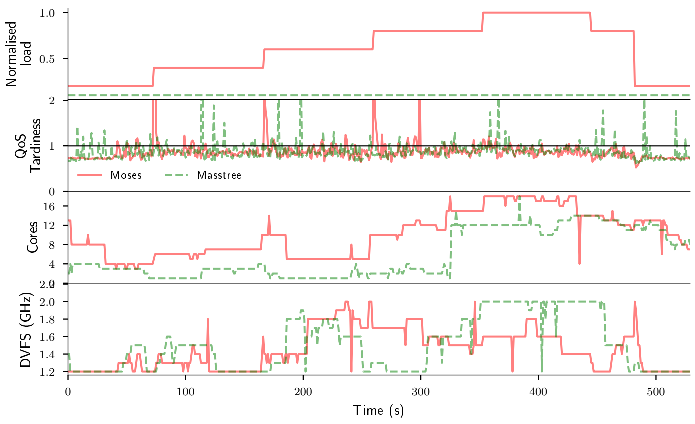


Fig. 11: Resource allocation with Twig-C when varying the load for Moses while Masstree has fixed load (20%).

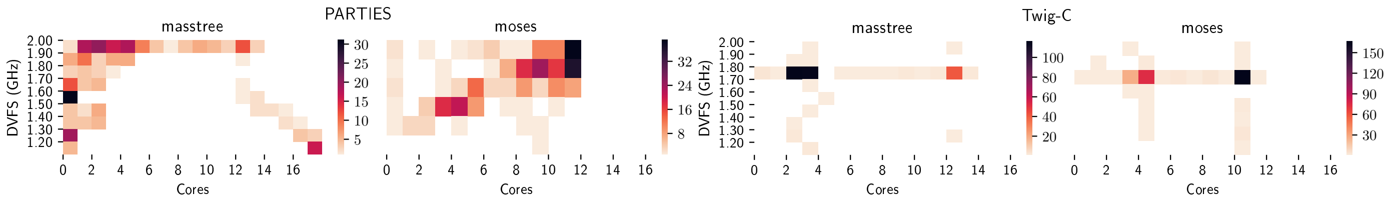


Fig. 12: Core mapping decisions with PARTIES and Twig-C for masstree and moses running at 20% and 80% of the max. load respectively. The colourmap represents the core allocation distribution over a period of 600 s.

combination. Figure 12 shows the mapping distribution for PARTIES (the first two columns from left) and Twig-C. Moses has a high demand for cache capacity and memory bandwidth [38], while Masstree is extremely sensitive to memory bandwidth interference. As can be seen in the graph, to maintain QoS, PARTIES makes minor changes to mapping decisions continuously based on how close it is from the target. Twig-C, on the other hand, uses numerous performance metrics to improve stability in mapping decisions. Consistently using fewer resources to meet the QoS, directly translates to more energy savings and more resources for other services.

Transfer Learning. We demonstrate the effectiveness of transfer learning with Twig-C by first learning with Moses and Masstree, and then swapping Moses with Xapian after the first 10 000 s. Moses and Xapian operate at 50% of the max. load, while Masstree operates at 20%. Figure 9 compares QoS guarantee (top two plots) and energy usage when running Xapian and Masstree with and without transfer learning. Each data point in the graph refers to a period of 300 s. This graph shows two key points: (a) With no transfer learning, Twig-C has a low QoS guarantee and high energy usage initially and this improves over time, as the learning agent transitions from an explorative to an exploitative policy. (b) With transfer learning, the learning agent adapts to service changes in under 10 time steps to deliver high QoS guarantees and low energy usage, which is similar to learning from scratch.

Varying Load. Figure 11 compares the behaviour of Twig-C with dynamic load variations.[2] Specifically, we vary the load of Moses from 20% to 100% gradually, and set the load of Masstree. As can be seen, Twig-C directly jumps to the appropriate core configuration for the specified load that satisfies the QoS. In addition to switching to the desired number of cores, it explores finer DVFS adaptions are they are cheaper relative to core migrations. PARTIES, on the other hand, migrates across numerous combination of cores before selecting the desired mapping decision that satisfies the QoS. These gradual migrations negatively effect the QoS especially when there is a load spike.

[2]Inclusion of PARTIES renders plot illegible.

VI. Related Work

The dominant approach in major data centres is to allocate tasks to cores by first allocating the tasks to the least-loaded physical nodes, and then using a single-node resource manager to allocate these tasks to cores. Past work in this area falls in two categories. First, prior work [40, 58**?** –68] propose fine-grained resource partitioning solutions that aim at eliminating interference among LC and batch services. These techniques require extensive microarchitectural feature tuning prior to deployment in production clusters. Secondly, prior work [9, 14, 16, 21, 69, 70] detect interference among colocated services and adjust resource allocation dynamically or disallow colocation of LC services. While this approach works well to maintain QoS, it is imperative to both increase system throughput and improve energy efficiency to increase revenue.

As systems increase in complexity (hardware and services), observability (more PMCs), and controllability (DVFS settings and core counts), it gets increasingly more expensive and error-prone to develop custom heuristics [12, 13, 71]. In comparison

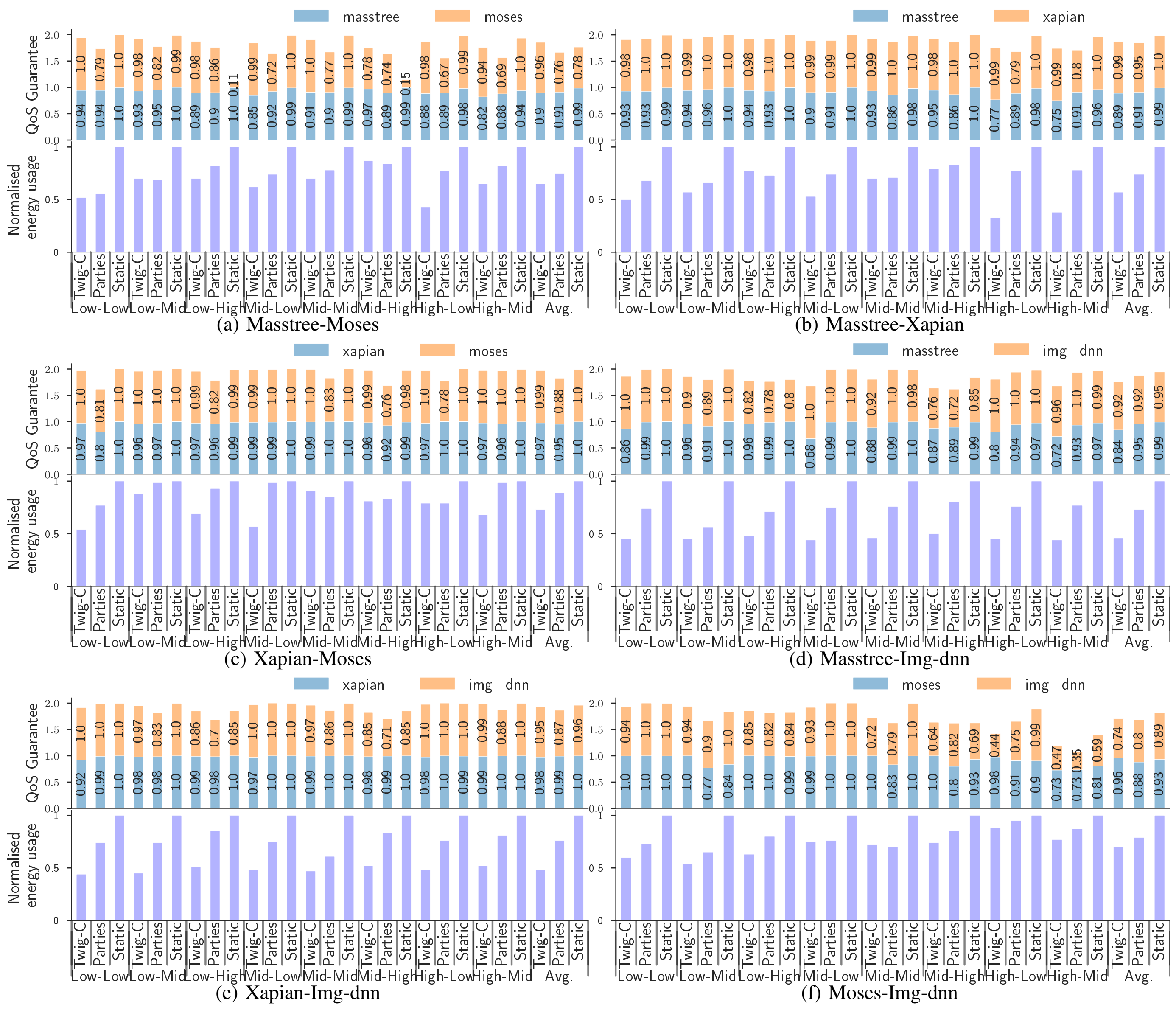


Fig. 13: Comparing the performance of Twig-C, PARTIES and Static mapping with a fixed load of 20% (low), 50% (mid) and 80% (high). Top and bottom graph show the QoS guarantee and energy usage normalised, respectively.

with Hipster [15, 72], Twig's use of a NN approximator for the state-space mapping means that: (1) Twig learns faster as it uses a NN instead of a Q-table, (2) Twig eliminates the need to explicitly traverse the state-action pairs to understand the quality of an action, (3) Twig reduces the memory usage by not storing the state-action space as a Q-table, (4) Twig understands the environment's state using a set of PMCs rather than a single metric, and (5) Twig can use transfer learning to quickly learn how to manage new services. Moreover, unlike other state-of-the-art approaches [12, 13, 40, 58], Twig's use of PMCs avoids the need for service-specific instrumentation.

VII. Conclusion

We propose Twig, a task-management solution based on deep reinforcement learning for energy-efficient resource management of colocated latency-critical services. Twig requires no service or system-specific information, and instead uses generic performance monitoring counters (PMCs) to manage the resource allocation.We demonstrate that Twig performs well across services and dynamically adapts the system by learning from the PMCs to improve the mapping of services to cores and adjust DVFS settings. Our results show that Twig reduces energy usage by up to 38% while achieving up to 99% QoS guarantees for latency-critical services.

Acknowledgement

This work was funded by the European Union under grant agreement No 754337 (EuroEXA), the Brazilian federal government under CNPq grant (Process n^{o} 430188/2018-8), and the Swedish Research Council under grant 2015-05159. The experiments were conducted on the NTNU EPIC computing infrastructure and support by NTNU's HPC group.

References

[1] J. Mars and L. Tang, "Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers," in *Proc. of the 40th Annual International Symposium on Computer Architecture*, ISCA '13, ACM, 2013.

[2] L. A. Barroso and U. Hoelzle, *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st ed., 2009.

[3] S. Chen, C. Delimitrou, and J. F. Martinez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in *Proc. of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[4] S. Zhuravlev, S. Blagodurov, and A. Fedorova, "Addressing shared resource contention in multicore processors via scheduling," *SIGPLAN Not.*, vol. 45, Mar. 2010.

[5] S. Blagodurov, S. Zhuravlev, M. Dashti, and A. Fedorova, "A case for numa-aware contention management on multicore systems," in *Proc. of the 2011 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'11, 2011.

[6] S. Eric and B. Jake, "The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search," *Velocity*, 2009.

[7] Y. Ding, N. Mishra, and H. Hoffmann, "Generative and multi-phase learning for computer systems optimization," in *Proc. of the 46th International Symposium on Computer Architecture*, ISCA '19, ACM, 2019.

[8] S. Blagodurov, S. Zhuravlev, A. Fedorova, and A. Kamali, "A case for NUMA-aware contention management on multicore systems," in *Proc. of the International Conference on Parallel Architectural and Compilation Techniques*, ACM Press, 2010.

[9] C. Delimitrou and C. Kozyrakis, "ibench: Quantifying interference for datacenter applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2013.

[10] R. Bitirgen, E. Ipek, and J. F. Martinez, "Coordinated management of multiple interacting resources in chip multiprocessors: A machine learning approach," in *Proc. of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, IEEE Computer Society, 2008.

[11] V. Ishakian, R. Sweha, J. Londoño, and A. Bestavros, "Colocation as a service: Strategic and operational services for cloud colocation," *2010 9th IEEE International Symposium on Network Computing and Applications*, 2010.

[12] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in *Proc. of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, ACM, 2015.

[13] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mossé, J. Mars, and L. Tang, "Octopus-man: Qos-driven task management for heterogeneous multicores in warehouse-scale computers," in *21st IEEE International Symposium on High Performance Computer Architecture, HPCA 2015*.

[14] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux," *ACM SIGARCH Computer Architecture News*, 2013.

[15] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell, "Hipster: Hybrid task manager for latency-critical cloud workloads," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017.

[16] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI 2," in *Proc. of the 8th ACM European Conference on Computer Systems - EuroSys '13*, ACM Press.

[17] C. Delimitrou and C. Kozyrakis, "QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon," *ACM Trans. Comput. Syst.*, vol. 31, Dec. 2013.

[18] C. Delimitrou and C. Kozyrakis, "Hcloud: Resource-efficient provisioning in shared cloud systems," *SIGPLAN Not.*, vol. 51, Mar. 2016.

[19] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot, "Stretch: Balancing QoS and Throughput for Colocated Server Workloads on SMT Cores," in *IEEE International Symposium on High Performance Computer Architecture (HPCA) 2019*.

[20] S. Blagodurov, D. Gmach, M. Arlitt, Y. Chen, C. Hyser, and A. Fedorova, "Maximizing server utilization while meeting critical slas via weight-based collocation management," in *2013 IFIP/IEEE International Symposium on Integrated Network Management (IM 2013)*, May 2013.

[21] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in *Proc. of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14.

[22] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-Clouds: Managing Performance Interference Effects for QoS-Aware Clouds," in *Proc. of the 5th European conference on Computer systems - EuroSys '10*, ACM Press, 4 2010.

[23] N. El-Sayed, A. Mukkara, P. Tsai, H. Kasture, X. Ma, and D. Sanchez, "KPart: A Hybrid Cache Partitioning-Sharing Technique for Commodity Multicores," in *Proc. of the International Symposium High-Performance Computer Architecture*, Feb 2018.

[24] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning," *CoRR*, vol. abs/1312.5602, 2013.

[25] Suton. R.S and A. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA: MIT Press, 1998.

[26] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstieß, and J. Schmidhuber, "PyBrain," *Journal of Machine Learning Research*, vol. 11, 2010.

[27] Z. Wang, N. de Freitas, and M. Lanctot, "Dueling network architectures for deep reinforcement learning," *CoRR*, vol. abs/1511.06581, 2015.

[28] A. Tavakoli, F. Pardo, and P. Kormushev, "Action branching architectures for deep reinforcement learning," *CoRR*, vol. abs/1711.08946, 2017.

[29] N. J. Cox, "Speaking stata: Correlation with confidence, or fisher's z revisited," *Stata Journal*, vol. 8, no. 3, 2008.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, 2011.

[31] M. Malik, H. Ghasemzadeh, T. Mohsenin, R. Cammarota, L. Zhao, A. Sasan, H. Homayoun, and S. Rafatirad, "Ecost: Energy-efficient co-locating and self-tuning mapreduce applications," in *Proc. of the 48th International Conference on Parallel Processing*, ICPP 2019, pp. 7:1–7:11, ACM, 2019.

[32] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, 2 2015.

[33] S. Eranian, "Perfmon2: improving performance monitoring on linux," July 2019.

[34] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4," January 2019.

[35] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, Dec. 1995.

[36] A. Guliani and M. M. Swift, "Per-Application Power Delivery," in *Proc. of the 14th EuroSys Conference 2019*, EuroSys, ACM, 2019.

[37] S. Pandruvada, "Running Average Power Limit (RAPL)." https://01.org/rapl-power-meter, July 2019.

[38] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in *2016 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2016.

[39] LLNL, "MSR Safe." github.com/LLNL/msr-safe, July 2019.

[40] C. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. F. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *Proc. of the International Symposium High-Performance Computer Architecture*, 2015.

[41] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl, "Algorithms for hyper-parameter optimization," in *Proc. of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, Curran Associates Inc., 2011.

[42] "HyperOpt." hyperopt.github.io/hyperopt/, July 2019.

[43] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[44] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. of the Fourteenth International Conference on Artificial Intelligence and Statistics*, vol. 15 of *Proc. of Machine Learning Research*, PMLR, 2011.

[45] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *Journal of Machine Learning Research*, vol. 15, 2014.

[46] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *CoRR*, vol. abs/1511.05952, 2015.

[47] Rajiv Nishtala. github.com/nishtala/TwigHPCA2020.git, December 2019.

[48] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2014.

[49] M. Själander, M. Jahre, G. Tufte, and N. Reissmann, "EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure," 2019.

[50] "Memcached." https://memcached.org/, July 2019.

[51] "Xapian." github.com/xapian/xapian, July 2019.

[52] P. Koehn, H. Hoang, A. Birch, C. Callison-Burch, M. Federico, N. Bertoldi, B. Cowan, W. Shen, C. Moran, R. Zens, C. Dyer, O. Bojar, A. Constantin, and E. Herbst, "Moses: Open source toolkit for statistical machine translation," in *Proc. of the 45th Annual Meeting of the ACL on Interactive Poster and Demonstration Sessions*, ACL '07, 2007.

[53] "A deep network handwriting classifier." github.com/xingdi-ericyuan/multi-layer-convnet, July 2019.

[54] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015.

[55] H. Kasture and D. Sanchez, "Ubik: efficient cache sharing with strict qos for latency-critical workloads," *ACM SIGARCH Computer Architecture News*, vol. 42, 4 2014.

[56] Nguyen, Khang T, "Software Enabling for Cache Allocation Technology in the Intel Xeon Processor E5 v4 Family." software.intel.com/en-us/articles/software-enabling-for-cache-allocation-technology, January 2019.

[57] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proc. of the International Symposium on Computer Architecture*, vol. 39, ACM Press, 6 2011.

[58] C. Hsu, Q. Deng, J. Mars, and L. Tang, "Smoothoperator: Reducing power fragmentation and improving power utilization in large-scale datacenters," in *Proc. of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2018.

[59] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar, "TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search," in *Proc. of the ACM/IEEE International Symposium on Microarchitecture*, 2015.

[60] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," *ACM SIGARCH Computer Architecture News*, vol. 42, 10 2014.

[61] P. Garefalakis, K. Karanasos, P. Pietzuch, A. Suresh, and S. Rao, "Medea: Scheduling of long running applications in shared production clusters," in *Proc. of the 13th EuroSys Conference*, EuroSys '18, ACM, 2018.

[62] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "DeepDive: transparently identifying and managing performance interference in virtualized environments," 6 2013.

[63] F. Romero and C. Delimitrou, "Mage: Online and interference-aware scheduling for multi-scale heterogeneous systems," in *Proc. of the International Conference on Parallel Architectural and Compilation Techniques*, ACM, 2018.

[64] D. Wong and M. Annavaram, "KnightShift: Scaling the Energy Proportionality Wall through Server-Level Heterogeneity," in *Proc. of the ACM/IEEE International Symposium on Microarchitecture*, IEEE, 2012.

[65] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," *SIGOPS Oper. Syst. Rev.*, vol. 50, Mar. 2016.

[66] H. Cook, M. Moreto, S. Bird, K. Dao, D. A. Patterson, and K. Asanovic, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proc. of the International Symposium on Computer Architecture*, ISCA '13, ACM, 2013.

[67] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *Proc. of the ACM/IEEE International Symposium on Microarchitecture*, ACM, 2015.

[68] H. Zhu, D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and M. Erez, "Kelp: Qos for accelerated machine learning systems," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 172–184, Feb 2019.

[69] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations," in *Proc. of the ACM/IEEE International Symposium on Microarchitecture*, ACM Press, 2011.

[70] C. Delimitrou and C. Kozyrakis, "Bolt: I know what you did last summer..in the cloud," *SIGARCH Comput. Archit. News*, Apr. 2017.

[71] R. Nishtala, D. Mossé, and V. Petrucci, "Energy-aware thread co-location in heterogeneous multicore processors," in *Proc. of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, IEEE Press, 2013.

[72] R. Nishtala, P. Carpenter, V. Petrucci, and X. Martorell, "The hipster approach for improving cloud system efficiency," *ACM Trans. Comput. Syst.*, vol. 35, Dec. 2017.