The Hipster Approach for Improving Cloud System Efficiency

RAJIV NISHTALA, Universitat Politècnica de Catalunya & Barcelona Supercomputing Center
PAUL CARPENTER, Barcelona Supercomputing Center
VINICIUS PETRUCCI, Federal University of Bahia
XAVIER MARTORELL, Universitat Politècnica de Catalunya & Barcelona Supercomputing Center

In 2013, U.S. data centers accounted for 2.2% of the country's total electricity consumption, a figure that is projected to increase rapidly over the next decade. Many important data center workloads in cloud computing are interactive, and they demand strict levels of quality-of-service (QoS) to meet user expectations, making it challenging to optimize power consumption along with increasing performance demands.

This paper introduces Hipster, a technique that combines heuristics and reinforcement learning to improve resource efficiency in cloud systems. Hipster explores heterogeneous multi-cores and dynamic voltage and frequency scaling (DVFS) for reducing energy consumption while managing the QoS of the latency-critical workloads. To improve data center utilization and make best usage of the available resources, Hipster can dynamically assign remaining cores to batch workloads without violating the QoS constraints for the latency-critical workloads. We perform experiments using a 64-bit ARM big.LITTLE platform, and show that, compared to prior work, Hipster improves the QoS guarantee for Web-Search from 80% to 96%, and for Memcached from 92% to 99%, while reducing the energy consumption by up to 18%. Hipster is also effective in learning and adapting automatically to specific requirements of new incoming workloads just enough to meet the QoS and optimize resource consumption.

CCS Concepts: • **Computer systems organization → Cloud computing**; • **Software and its engineering** → Scheduling;

Additional Key Words and Phrases: data center, cloud computing, warehouse-scale computer, energy efficient computing, resource efficiency, interference, scheduling, QoS, latency-critical applications, performance isolation.

ACM Reference format:
Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. 2017. The Hipster Approach for Improving Cloud System Efficiency. *ACM Trans. Comput. Syst.* 35, 3, Article 8 (December 2017), 25 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

This work has been partially supported by the European Union FP7 program through the Mont-Blanc-3 (FP7-ICT-671697) and EUROSERVER (FP7-ICT-610456) projects, by the Ministerio de Economia y Competitividad under contract Computación de Altas Prestaciones VII (TIN2015-65316-P), and the Departament de Innovació, Universitats i Empresa de la Generalitat de Catalunya, under project MPEXPAR: Models de Programació i Entorns d Execució Paral·lels (2014-SGR-1051).

Prior Publication: Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci and Xavier Martorell. Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads. In *Proceedings of the 23rd High Performance and Computer Architecture (HPCA 2017).*

In this work, we extend our previous work in several ways. First, we present an analysis of the size of the reward lookup table and an optimization for the table to improve the scalability of our reinforcement learning mechanism. Second, we demonstrate Hipster's capability to adapt to changes in the latency-critical application at runtime and still satisfy QoS guarantees of the new incoming applications. Lastly, we present a deployment methodology for setting up new applications managed by Hipster's runtime system.

Author's addresses: Rajiv Nishtala and Xavier Martorell, Universitat Politècnica de Catalunya and Barcelona Supercomputing Center; Paul Carpenter, Barcelona Supercomputing Center; Vincius Petrucci, Federal University of Bahia, Salvador, Brazil. emails:{rajiv.nishtala, paul.carpenter, xavier.martorell}@bsc.es; email: vpetrucci@ufba.br .

DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

In 2013, U.S. data centers consumed 91×10^9 kW h, which corresponds to 2.2% of the country's total electricity consumption and about 100 million metric tons of carbon pollution per year [16, 24, 70]. Energy efficiency is, in fact, a major issue across the whole computing spectrum, and modern systems have been exploring alternative heterogeneous processors [11, 12, 28, 29, 45, 52, 68] and Dynamic Voltage and Frequency Scaling (DVFS) [26, 42, 59, 69] to trade-off performance and energy consumption.

Many important cloud workloads are latency-critical, and they require strict levels of quality-of-service (QoS) to meet user expectations [36, 40, 41]. A web-search, for example, must complete within a fraction of a second [5], otherwise users are likely to give up and leave. A previous study [22] has shown that marginal QoS delays (of hundreds of milliseconds) can greatly impact user experience and advertising revenue. In particular, it is important to meet the QoS tail latency, such as the 95th or 99th percentile of the request latency distribution [15].

Recent works [36, 39–41, 47, 56] have shown that traditional power management practices and CPU utilization measures are unsuitable to drive task management for cloud workloads. This is because prior schemes (like OS-level DVFS) work well to deliver long-term performance for batch workloads, but they can severely hurt the QoS of latency-critical data center workloads.

As noticed in prior work [17, 56], cloud workload management is very challenging in heterogeneous server systems because an application can experience different behavior in QoS and resource efficiency depending on specific resource allocation decisions. This requires careful resource characterization of the running workloads to optimize resource usage. In many data centers, there also is a wish to run both latency-critical and batch workloads. Running both latency-critical and batch workloads, in this way, increases cluster utilization during periods of low demand, reducing the operational cost and the total energy consumption.

In this paper, we introduce **Hipster**, a scheme that manages heterogeneous multi-core allocation and DVFS settings for latency-critical workloads given QoS constraints, while optimizing system resource consumption. In shared data centers, Hipster allows collocation of latency-critical and batch workloads to maximize the data center utilization. In such scenarios, the resources allocated to latency-critical workloads are just enough to meet the QoS target, and the remaining resources are allocated to throttle the batch workloads.

The major contributions of our work are:

(1) We present **Hipster**, a hybrid management solution combining heuristic techniques and reinforcement learning to make resource-efficient allocation decisions, specifically deciding the best mapping of latency-critical and batch workloads on heterogeneous muti-cores and their DVFS settings.

(2) Hipster is presented in two variants: **HipsterIn** and **HipsterCo**. HipsterIn (for interactive workloads) is targeted towards allocating resources to latency-critical workloads so that the system power consumption is minimized, whereas HipsterCo (for collocated workloads) enables running both latency-critical and batch workloads for improved server utilization. Both variants ensure that QoS is met for the latency-critical workloads.

(3) We carried out real measurement experiments on a 64-bit big-LITTLE (ARM Juno R1) platform along with back-end services such as Web-Search and Memcached. The request generator for the applications follows a diurnal load pattern typical of production data centers.

(4) We evaluate Hipster against the only other heterogeneous platform-aware state-of-the-art scheme [56] that dynamically allocates heterogeneous cores to latency-critical workloads. Our results show that HipsterIn outperforms prior work, in energy consumption reduction by 13%, while achieving up to 99% QoS guarantees for the latency-critical workloads. In addition, our results for HipsterCo show that it improves performance by 2.3× compared to a static/conservative policy running batch workloads, while meeting QoS targets for the latency-critical workloads.

(5) We demonstrate the ability of Hipster to adapt dynamically to new incoming workloads (not known in advance) at runtime. Despite changes in the external workload pattern, Hipster delivers up to 98% QoS guarantees for the latency-critical workloads. We open source Hipster with a deployment methodology that can be used to manage other cloud workloads.

2 MOTIVATION

Previous studies [6, 19, 56, 57] have shown that data centers like Google and Facebook operate at peak load only for a small fraction of the day, where the power consumption and load are proportional. The number of queries per second during a web search, a typical load at Google data center, varies between about 5% and 80% of maximum capacity [31, 40, 47]. Similarly, Facebook consistently sees diurnal load variations between 10% and 95% of maximum capacity, across multiple server clusters [4, 8]. The periods of low server utilization provide opportunity to reduce data center energy consumption [17, 18, 41, 56].

For instance, Figure 1 shows a Web-Search application running on two big cores of an ARM big.LITTLE architecture (details on experimental setup are given in section 4.1). The diurnal load variation pattern was obtained from [31, 40, 47]. As can be seen in the figure, although load drops dramatically, power consumption is always at 60% or above. For this reason, both academia and industry are working towards better energy proportionality; i.e. that the system's power consumption is proportional to utilization.

There are also opportunities to improve energy efficiency using heterogeneous servers combined with DVFS [62]. Heterogeneous servers can minimize power consumption at low load by deploying small cores, and can provide maximum performance using big cores to meet the QoS target for latency-critical workloads [11, 35, 56].

2.1 Mixing Different Core Types with DVFS

Figure 2 shows the energy efficiency in RPS/Watt (Requests Per Second per Watt) and QPS/Watt (Queries Per Second per Watt) when using a state-of-the-art baseline policy [56]. We explore a heterogeneous architecture mixing different cores types and DVFS (HetCMP) running Memcached (Figure 2a) and Web-Search (Figure 2b) at different load levels. For each policy, among the configurations where the QoS is met at each load level, the configuration with the least power consumption is selected. The table at the bottom of each subfigure shows the configuration selected by HetCMP and our baseline policy. In the configurations of the embedded table, *B* and *S* represent big and small cores, respectively. The configuration space for HetCMP consists of core-mappings (big and small cores) and DVFS combinations for a heterogeneous platform, whereas the baseline policy consists exclusively of either big or small cores at the highest DVFS. The configurations available for the baseline policy are therefore a subset of HetCMP. Experimental details are given in Section 4.

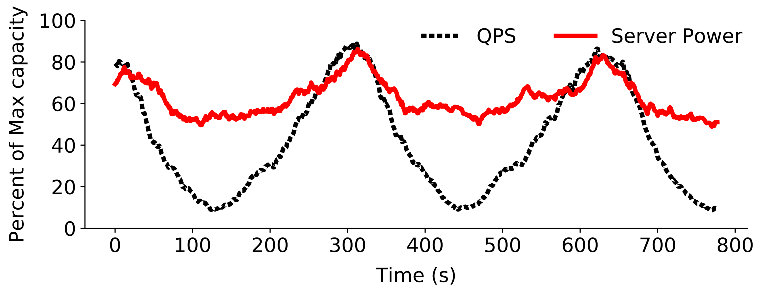
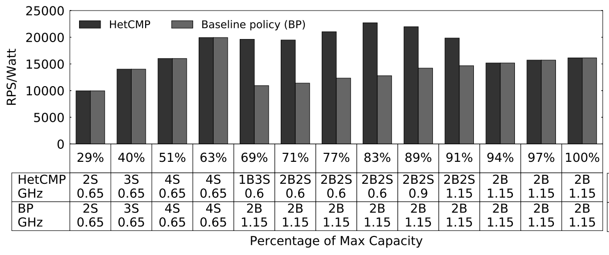
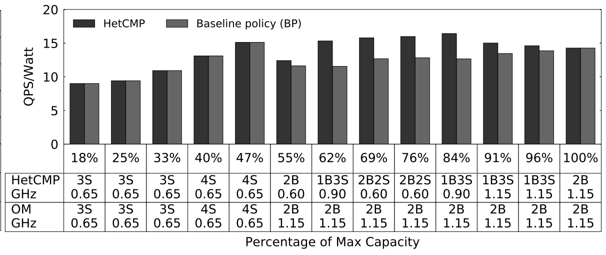


Fig. 1. Power drawn for a diurnal load [31, 40, 47] for Web-Search running on two big cores of the ARM Juno R1 (64-bit big.LITTLE) platform. The workload is given by queries per second (QPS).



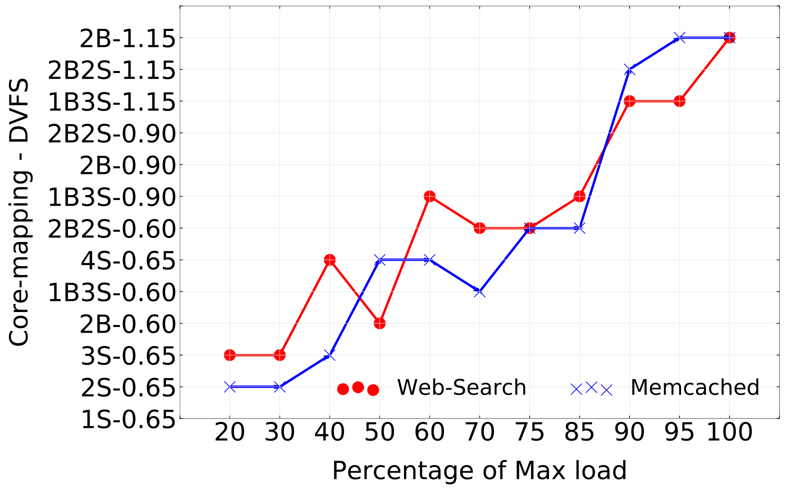
	29%	40%	51%	63%	69%	71%	77%	83%	89%	91%	94%	97%	100%
HetCMP GHz	2S 0.65	3S 0.65	4S 0.65	4S 0.65	1B3S 0.6	2B2S 0.6	2B2S 0.6	2B2S 0.6	2B2S 0.9	2B2S 1.15	2B 1.15	2B 1.15	2B 1.15
BP GHz	2S 0.65	3S 0.65	4S 0.65	4S 0.65	2B 1.15	2B 1.15	2B 1.15	2B 1.15	2B 1.15	2B 1.15	2B 1.15	2B 1.15	2B 1.15

(a) Memcached workload



	18%	25%	33%	40%	47%	55%	62%	69%	76%	84%	91%	96%	100%
HetCMP GHz	3S 0.65	3S 0.65	3S 0.65	4S 0.65	4S 0.65	2B 0.60	1B3S 0.90	2B2S 0.60	2B2S 0.60	1B3S 0.90	1B3S 1.15	1B3S 1.15	2B 1.15
OM GHz	3S 0.65	3S 0.65	3S 0.65	4S 0.65	4S 0.65	2B 1.15	2B 1.15	2B 1.15	2B 1.15	2B 1.15	2B 1.15	2B 1.15	2B 1.15

(b) Web-Search workload



(c) State Machine

Fig. 2. Throughput per watt of Memcached (2a) and Web-Search (2b) with baseline policy (BP) [56] and heterogeneous platforms with DVFS (HetCMP) at different load levels along with their respective state machines (2c)

Figure 2 raises two main concerns with current state-of-the-art heuristic algorithm [56]. First, Figure 2a demonstrates that in periods of low load (less than 60% of max capacity for Memcached), exclusive use of low performance cores at lower DVFS ensures QoS is met while reducing static-power, thus making it an excellent option to use for periods of low load. As the load increases, HetCMP transitions from low performance cores to a best combination of small and big cores at a given DVFS (for instance, 2 big and 2 small cores – 2B2S at 0.9 GHz at 89% load) to deliver the required latency. On the other hand, the baseline policy transitions directly from low performance cores to high performance cores at highest DVFS to deliver the required latency, thereby increasing energy consumption by 27.74% (mean). In periods of very high load (more than 90% for Memcached), exclusive use of high performance cores at higher DVFS ensures QoS is met with better energy proportionality. Similar results were observed for Web-Search (Figure 2b) with up to 25% (mean) energy savings. The state-machine configuration for Memcached and Web-Search are represented by the blue and red line, respectively, in Figure 2c.

In summary, we show that small and big cores are an attractive option for periods of low load and very high load, respectively, while meeting QoS targets at a much lower cost. On the other hand, for intermediate loads, which are generally experienced by data centers during the day [66], harnessing HetCMP provides the opportunity for higher performance at a lower cost.

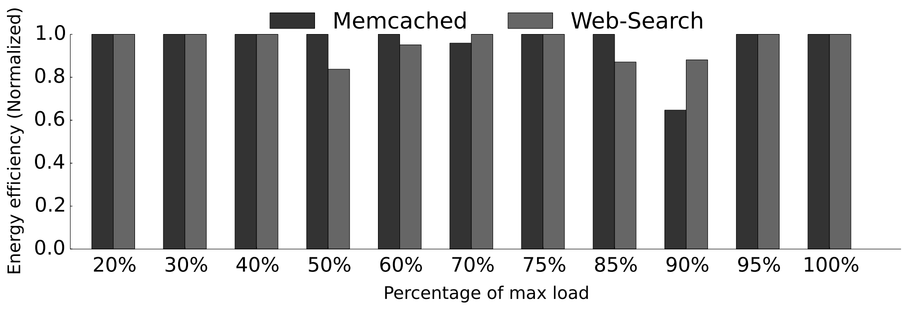


Fig. 3. Energy efficiency at various load levels for Memcached while meeting QoS, using the state-machine of Web-Search normalized to the state-machine of Memcached (*lower is worse*); converse for Web-Search.

2.2 Exploring Individual Workload Particularities

We observe that prior work [56] relies on a general single heuristic to allocate exclusively big or small cores to workloads. By allowing an arbitrary allocation mix of big and small cores with DVFS, this kind of heuristic can be sub-optimal across diverse applications and architectures (evaluation details in Section 4); that is, a single state-machine management (as in prior work) may fail to precisely satisfy the QoS targets given distinct workload characteristics of diverse applications. To illustrate this point, Figure 2c shows two distinct/unique state transition mappings that are optimal (throughput per watt) at different load capacities for Memcached and Web-Search.

Figure 3 shows the energy efficiency that would be neglected (ensuring QoS is met at different load levels) when using the state-machine built for Web-Search but used for the Memcached workload, normalized to the energy efficiency using the state-machine built exclusively for Memcached; and vice-versa. Figure 3 demonstrates that different latency-critical applications benefit from different state-transition mappings and show improvement in energy efficiency up to 35% for Memcached (at 90% load) and up to 19% for Web-Search (at 50% load). For instance, at low loads and at very high loads both applications use exclusively small cores (low static power) or big cores (high static power), respectively. However, for intermediate loads, the configurations in the state-transition for Web-Search are not present in Memcached and vice-versa, thus providing minimal to no energy optimization.

In practical scenarios, each workload has a time-varying load [38] and a QoS target that needs to be met. As shown in Figure 2 and 3, there exists a unique configuration for each load that optimizes energy efficiency. Moreover, the time-varying load presented in two forms: sudden load spikes [15] or gradual load changes [31, 47]. Both these forms present a challenge for a heuristic based approach as it jumps across multiple configurations to meet the QoS target, thereby leading to QoS violations due to rampant core oscillations. Also, the authors of Rubik [36] note that core-transitions are far more costly relative to DVFS changes.

We argue that there is a need for an application-tailored learning approach that can best exploit the energy efficiency benefits of heterogeneous architectures and DVFS features, and can deal with sudden/gradual load changes across different levels. This is precisely what **Hipster** delivers.

3 THE HIPSTER APPROACH

In this section, we introduce **Hipster**, a hybrid reinforcement learning (RL) approach coupled with a feedback controller that dynamically allocates workloads to heterogeneous cores while selecting optimized DVFS settings. We propose a variant, called HipsterIn, that is optimized for latency-critical workloads running solo in the system, adjusting the system configuration to reduce energy consumption. The HipsterCo variant, which supports collocation of latency-critical and batch workloads, and focuses on maximizing the throughput of the batch workloads. Both variants of Hipster always ensure that the QoS requirements are met for the latency-critical workloads.

3.1 Machine Learning to Design Efficient Systems

We observe that Reinforcement Learning (RL) can be a better approach to design efficient systems in contrast to simpler techniques like regression techniques for the following reasons:

(1) In practical settings, RL can provide substantial advantage over regression techniques by eliminating the need to train the model with large amounts of offline labeled-data.

(2) Given the dynamic nature of applications in modern data centers, RL can adapt/tune the model at runtime for different applications and servers, thereby making it a convenient black-box method to deploy in production data centers. This has the potential to dynamically learn an applications' behavior and improve the QoS guarantee while reducing energy consumption.

(3) While regression models may have lower overhead than RL approaches, we show in our experiments that Hipster's overhead is still <0.2% (see Section 3.10); this is similar to the overhead incurred by prior work [41, 56] without using RL techniques.

3.2 Hipster Reinforcement Learning

The RL problem solved by Hipster is formulated as a Markov Decision Process (MDP) [58]. In an MDP, a decision-making process must learn the best course of action to maximize its total reward over time. At each discrete instant, n, the process can observe its current *"state"*, w_n, and it must choose an *"action"* c_n from a finite set of alternatives. Depending on the chosen action and current state (but nothing else), there is an unknown probability distribution controlling which state, w_{n+1}, it enters next and the reward, λ_n, that it receives. The problem is to maximize the total discounted reward, given by $\sum_{n=0}^{\infty} \gamma^n \lambda_n$, where γ is the discounting factor. The discounting factor γ should be positive and (slightly) less than one, in order to reflect a moderate preference for rewards in the near future.

The hybrid task management problem solved by Hipster is translated to an MDP as follows. The state w_n indicates the current load on the latency-critical application, measured during the (prior) time interval t_{n-1} to t_n. Hipster quantizes the load into buckets. Specifically, the latency-critical application provides a measurement of the percentage load during the time interval, in terms of requests per second, queries per second, or similar. The action, c_n, chosen by Hipster depending on the state, determines the configuration to be used in the (next) time interval, t_n to t_{n+1}; i.e. the combination of cores and DVFS settings allocated to the latency-critical application. These settings are used for the upcoming interval, at the end of which, at time t_{n+1}, the reward λ_n is determined depending on the level of QoS relative to the target, given a metric of optimization: either the system power consumption (HipsterIn) or the throughput of the batch workloads (HipsterCo). A precise definition of the calculation of the reward is given in Section 3.7.

RL is a type of unsupervised machine learning with a focus on online learning [49]. It solves an MDP by maintaining a table of values, $R(w, c)$, indexed on the possible states $w \in W$ and possible actions $c \in C$. The entry $R(w, c)$ estimates the total discounted reward that will be received, starting from state w, if the decision-making process starts by choosing next action c. Assuming that the **lookup table**, $R(w, c)$ has close to correct values, then, if the current state is w_n, the best action c_n is the one that gives the maximum total discounted reward; i.e. $c_n = \arg\max_c R(w_n, c)$. The process chooses this value of c_n, then it updates $R(w_n, c_n)$ using a particular formula based on the old and new states, w_n and w_{n+1}, and the reward λ_n.[1] A classic problem in RL is known as the *exploitation–exploration dilemma*, which captures the need not only to exploit the best solution identified so far, but also to fully explore alternatives, which may or may not be better than the current ones stored in the table.

Hipster uses a hybrid RL approach [65], which combines reinforcement learning with a heuristic, to be used while the algorithm is still learning optimized resource allocation decisions. For Hipster, the heuristic improves QoS at the beginning of the execution and it is also re-used after a change in the characteristics of the problem, e.g. the mix of batch workloads. A hybrid RL [65] has the potential to outperform pure RL schemes [33, 64] that only deal with the exploitation–exploration dilemma (e.g. Q-learning), for several reasons:

(1) During the learning phase, online unsupervised learning without a heuristic generates random decisions, which would produce unacceptable QoS violations.

[1]The update of $R(w_n, c_n)$ is on line 18 of Algorithm 1.

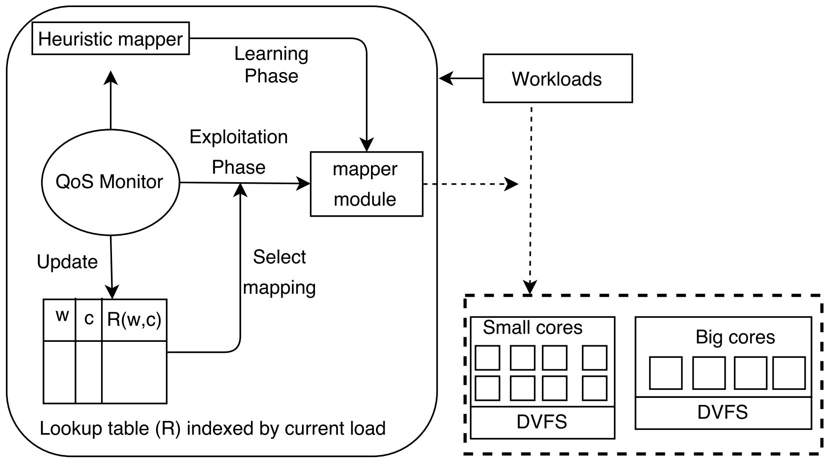


Fig. 4. High-level view of Hipster runtime system

(2) As the complexity of the problem increases, in terms of workloads, number of cores, DVFS settings, and so on, it may take longer to learn the table *R*. In contrast, a hybrid RL can quickly find acceptable solutions even during the learning phase.

(3) The exploration feature of many RL approaches is necessary to capture a global maximum, but it may cause extra QoS violations. Using a heuristic in the learning phase can reduce the need to explore configurations that clearly violate QoS.

3.3 Hipster Runtime System

Figure 4 shows a high-level view of the Hipster runtime system, including QoS Monitor, Heuristic Mapper (used in the learning phase), Exploitation Phase, and Mapper Module (used in both learning and exploitation phase). Given a QoS target, an incoming load, and a metric to optimize for, Hipster learns the most adequate core configuration and DVFS settings by managing a lookup table that is used to map the workloads to the available hardware resources.

3.4 QoS Monitor

The QoS Monitor is responsible for periodically collecting the performance statistics from the latency-critical and batch workloads. For the latency-critical workload, Hipster gathers the appropriate application-level QoS metrics such as throughput (RPS or QPS) and latency (query tail latency). It also reads the current load on the latency-critical workload and quantises this value into discrete buckets between 0 and $T - 1$, for (some) small value T. HipsterCo uses a profiling tool to measure the throughput of the batch workloads, using per core hardware performance counters, such as CPU utilization, cache-misses and IPS.

3.5 Learning and exploitation phases

The data collected by the QoS monitor is used to make the thread-to-core mapping decisions. In the learning phase, Hipster uses a feedback control loop based on heuristics to map the latency-critical workload to resources. Following the intuition from Section 2, when load is low, the mapper executes the latency-critical workload on small cores at lower DVFS states, and when load is high, it uses a combination of big and small cores at higher DVFS. Hipster also begins populating the lookup table so that each entry approximates the corresponding total discounted reward. Specifically, Hipster uses the reward mechanism (Section 3.7) to prefer core configurations that minimize system energy consumption or maximize batch workload throughput, while ensuring as well as possible that at least 95 % QoS guarantee is achieved [32].

In the exploitation phase, Hipster uses the lookup table to select the core mapping and DVFS settings, based on the load. It also continues to update the values in the lookup table, in order to continue to improve the mapping decisions. At runtime, Hipster determines when to dynamically switch between the learning and exploitation phases, based on a prefixed time quantum. At deployment stage, we ensure that the bucket size for each workload gives at least 95 % QoS guarantee [38] with minimal energy consumption.

3.6 Heuristic Mapper (Learning Phase)

The heuristic mapper is a state machine with a feedback control loop based on the ideas of Octopus-Man [56]. The current state identifies the core configuration: the DVFS settings and number and type of cores to use for the latency-critical workload.[2] The choice of available states depends on the platform; i.e. the total number and types of cores, and the DVFS settings. There is a predefined ordering of the states, approximately from highest to lowest power efficiency. This ordering is determined by measuring the power and performance of each state using a CPU-intensive microbenchmark consisting of several mathematical operations without memory accesses.

Whenever QoS is close to being violated, the state machine transitions into the next-higher power state. The QoS is quantified using the currently measured tail latency at the 95th or 99th percentile, denoted QoS_{curr}. The target tail latency is denoted by QoS_{target}. The state machine transitions to the next-higher state whenever the time interval ends in the so-called *danger zone* defined by:

$$QoS_{\mathrm{curr}} > QoS_{\mathrm{target}} \times QoS_{\mathrm{D}}$$

where QoS_{D} is a parameter between 0 and 1 that defines the size of the danger zone. Whether such a state transition improves or degrades performance and whether it actually increases or decreases power depends on the characteristics of the platform and the particular workloads. The state machine may have to make several consecutive state transitions until the QoS is met.

In contrast, whenever the QoS is far from being violated, the state machine transitions into the next-lower power state. This happens whenever the time interval ends in the so-called *safe zone* defined by:

$$QoS_{\mathrm{curr}} < QoS_{\mathrm{target}} \times QoS_{\mathrm{S}}$$

where QoS_{S} is a parameter between 0 and QoS_{D} that defines the size of the safe zone. The values of QoS_{D} and QoS_{S} are determined to avoid oscillations between adjacent states, empirically computed as in prior work [32, 56].

We observe that the heuristic proposed by Octopus-Man [56] is attractive because of its simplicity but it can be sub-optimal (see Section 2, Figure 2c) because there is no common static ordering of configuration states that works for all workloads. Moreover, in practice, the state machine may respond slowly to rapid changes in load. Nevertheless, we found that such a state machine heuristic is powerful when used to accelerate the learning phase of the RL algorithm. The heuristic enables exploring viable core configurations to quickly populate reasonable reward values into the lookup table.

3.7 Reward Calculation

During both the learning and exploitation phases, the values in the lookup table are dependent on the reward, calculated as defined in Algorithm 1. The reward calculation is invoked after each monitoring interval, and its definition is determined empirically (more details in Section 3.9). The reward λ_n has three parts: QoS Reward, Stochastic Reward, and either Power Reward (for HipsterIn) or Throughput Reward (for HipsterCo).

3.7.1 ***QoS Reward.***

The ratio of the measured QoS to the QoS target is known as QoS_{reward}. If this value is less than one, then the QoS target has been met, and it quantifies how quick the response was as the **QoS earliness**. In this case, line 7 or 9 applies a positive reward that prefers configurations that approach the QoS target, which acts as a heuristic to reduce energy consumption or improve batch workload throughput. If QoS_{reward} is greater than one, then the QoS target has *not* been met, and it determines how intense the violation was as the **QoS tardiness**. In this case, line 11 applies a negative QoS reward.

3.7.2 ***Stochastic Reward.***

When the QoS is below the target, as defined in Section 3.6, but still over the danger zone, then a stochastic penalty is applied (line 9 of Algorithm 1). The stochastic penalty offers the possibility to continue to explore the configuration, but with a smaller probability. In future, other external influences for the latency-critical workload like noise, contention on shared resources, pending queue lengths, etc., may cause a QoS violation.

[2]State machines and Markov Decision Processes use "state" with different meanings. In Section 3.6 (only), "state" refers to the core configuration, elsewhere it refers to the load.

Algorithm 1 Reward mechanism

```
                                                   ▷ Determine reward $\lambda_n$ based on interval $t_n \dots t_{n+1}$
 1: Let $QoS_{\text{target}}$ be the target QoS of the interactive workload.
 2: $QoS_{\text{curr}} = QoSMonitorLatency$
 3: $Power = QoSMonitorPower$
 4: $QoS_{\text{reward}} = QoS_{\text{curr}}/QoS_{\text{target}}$
 5: $Power_{\text{reward}} = TDP/Power$                    ▷ TDP (thermal design power)
 6: if $QoS_{\text{curr}} < QoS_{\text{target}} \times QoS_{\text{D}}$ then
 7:     $\lambda_n = QoS_{\text{reward}} + 1$
 8: else if $QoS_{\text{curr}} < QoS_{\text{target}}$ then
 9:     $\lambda_n = QoS_{\text{reward}} + 1 - Random(0,1)$
10: else
11:     $\lambda_n = -QoS_{\text{reward}} - 1$
12: if there exist batch jobs then
13:     $Throughput_{\text{reward}} = \frac{B_{\text{IPS}}+S_{\text{IPS}}}{maxIPS(B)+maxIPS(S)}$
14:     $\lambda_n = \lambda_n + Throughput_{\text{reward}}$
15: else
16:     $\lambda_n = \lambda_n + Power_{\text{reward}}$
17: if $\nexists R(w_n, c_n)$ then $R(w_n, c_n) = 0$
18: $R(w_n, c_n) = R(w_n, c_n) + \alpha\left(\lambda_n + \gamma \max_{d \in C} R(w_{n+1}, d) - R(w_n, c_n)\right)$
```

3.7.3 ***Throughput Reward (HipsterCo).*** We define $Throughput_{\text{reward}}$ to capture the benefits of co-locating batch jobs, which is calculated in line 13 to 14 of Algorithm 1 in relationship with the total throughput of the batch workloads. Since HipsterCo does not require modifications to the batch workloads, it is only possible to measure their throughput in a generic way using performance counters. Specifically, the throughput is quantified in terms of IPS. The parameters B_{IPS} and S_{IPS} measure the total IPS of the big and small clusters running batch workloads, respectively. The denominator is constant given by the sum of $maxIPS(B)$ and $maxIPS(S)$, which measure the maximum IPS, at highest DVFS, for the big and small cores respectively. More details are given in Section 4.1.

3.7.4 ***Power Reward (HipsterIn).*** The ratio of the thermal design power (TDP) to the measured system power consumption is known as $Power_{\text{reward}}$ as shown in line 5 and applied in line 16. A smaller value of this term means that the system power consumption was lower, and it increases the reward.

3.7.5 ***Reward Update Function.*** Once the reward λ_n has been calculated, line 18 updates the value of $R(w_n, c_n)$ in the lookup table, and this is done in the same way during both the learning and exploitation phases. This update is done using two scalar parameters, both between zero and one: the learning rate, α, and the discounting factor, γ.

The α coefficient in line 18 of Algorithm 1 is the **learning factor**, which controls the rate at which the values in the lookup table $R(w, c)$ are updated. A large value of α close to one means that the algorithm learns quickly, favoring recent experience, but increasing the susceptibility to noise. In contrast, a small value of α means that the algorithm learns slowly. In our experiments we used $\alpha = 0.6$

The γ coefficient in line 18 of Algorithm 1 is the **discounting factor**, which quantifies the preference for short-term rewards [63]. Setting $\gamma = 0$ means that the algorithm only relies on immediate short-term rewards. To allow a balance between short-term and future rewards, we set $\gamma = 0.9$ (empirically determined). In other words, this methodology allows the optimization problem to also take into account future rewards.

Algorithm 2 Exploitation Phase

```
 1: Let $X$ be threshold on QoS guarantee to re-enter learning phase
 2: Let $w_n$ be observed load for interval $t_{n-1} \dots t_n$
 3: Let $c_n$ be configuration for interval $t_n \dots t_{n+1}$
 4: Let $n = 0$
 5: repeat
                                        ▷ At time $t_n$, choose configuration for $t_n$ to $t_{n+1}$
 6:     Let $c_n = \max_{d \in C} R(w_n, d)$
 7:     if there exist batch jobs then
 8:         Allocate remaining cores to batch jobs
 9:         if latency-critical jobs on a single core type then
10:             Set highest DVFS for other core type
11:     else
12:         Set lowest DVFS for remaining cores
13:     Sleep until $t_{n+1}$                                  ▷ Run for interval $t_n$ to $t_{n+1}$
14:     Let $w_{n+1}$ be the quantised load from the latency-critical workload
15:     Call Algorithm 1                            ▷ Algorithm 1 updates $R(w_n, c_n)$
16:     $n = n + 1$
17:     if $QoSGuarantee \leq X$ then Learning phase
18: until Terminated
```

3.8 Exploitation Phase

The exploitation phase of Hipster is defined by Algorithm 2. Line 6 determines the configuration, c_n, with the highest estimated total discounted reward. Lines 7 to 12 apply the configuration by mapping the workloads to the available resources, as described below, depending on the specific variant of Hipster (HipsterIn or HipsterCo). Line 13 runs the workload for the next time interval, and line 15 calls Algorithm 1 to update the lookup table, based on the metrics obtained by the QoS Monitor during the time interval. Line 17 re-enters the learning phase when necessary. The mapping of workloads to computational resources is performed in accordance to the following reward mechanisms.

3.8.1 Reward Mechanism for HipsterIn. To minimize power consumption while meeting the QoS target for latency-critical workloads, the configuration with the highest reward is selected and then DVFS setting for the remaining cores is set to the lowest value (Lines 11 to 12 of Algorithm 2).

3.8.2 Reward Mechanism for HipsterCo. Corroborating the findings of prior work [41], we observe that collocating both latency-critical and batch workloads degrades QoS at higher loads due to shared resource contention. If the reward mechanism is not aware of such collocations, it may make decisions that violate QoS for the latency-critical workload and/or reduce the throughput of the batch workloads. To mitigate this issue, we introduce the following mechanisms. First, to maximize the throughput of the throughput-oriented workloads while meeting QoS targets, all of the remaining cores are allocated to the batch workloads (lines 7 to 8 in Algorithm 2). Second, in case the latency-critical job is allocated exclusively to a given core type, the other core type is set to the highest DVFS to accelerate the batch workloads (lines 9 to 10 in Algorithm 2). For instance, on a two-socket/cluster system with two cores per socket/cluster, if the latency-critical workload is running on two small cores, the big cores are allocated to the batch workloads at the highest DVFS.

3.9 Load Responsiveness

To ensure that QoS is met for latency-critical workloads, Hipster must quickly respond to fluctuations in load and latency, either due to changes in core mapping, DVFS or any external influence. Therefore, the responsiveness to load in Hipster is determined by (a) the computation latency in migrating cores and setting DVFS, (b) the reaction time of QoS between migrating an application from current mapping to future mapping, and (c) the granularity of monitoring for the latency-critical workload's QoS.

In Hipster, we determine the sampling interval as a sum of the monitoring interval for the latency-critical application, and the overhead to switch the core mapping and DVFS. As the default monitoring interval for Memcached and Web-Search is one second, the latency required for changes in core mapping and DVFS are negligible [12, 37, 43].

3.10 Hipster Implementation

Hipster is implemented in user space, and it uses minimal hardware support exposed by Linux. The hardware-dependent components of Hipster are QoS Monitor and Mapper Module, together with a lookup table, as shown in Figure 4.

QoS Monitor. Hipster uses a separate process to read the power measurements using native energy meters, at the sampling interval of the application. In addition to measuring energy, the QoS Monitor also gathers runtime statistics for the query/request latency of the latency-critical workload, using a logfile interface. In the case of HipsterCo, the batch workload aggregate IPS per core are measured using the performance monitoring tool, perf [55], specifically using the *perf_event* instructions [13, 14]. Alternatives to perf include the profiling tools [61] supported by Docker and Kubernetes [7].

Mapper Module. The workloads are mapped to cores using the Linux sched_setaffinity call and DVFS is controlled using acpi-cpufreq. In addition, Hipster suspends and resumes the batch workloads using the relevant OS signals (SIGSTOP and SIGCONT in Linux).

Runtime overhead. Hipster has a simple algorithm, requiring few control flow statements and main memory accesses, so its runtime overhead is negligible. We measured the execution time overhead (implemented in Python and including I/O) to be <2 ms, so triggering Hipster every second, as in our experiments, incurs an overhead <0.2%.

Lookup table. Each iteration of the RL algorithm accesses and modifies entries in the lookup table. To ensure these operations take negligible time, the computational complexity to access the table should be at most a few instructions. The lookup table in Hipster was implemented using a Python dictionary, using open addressing to resolve hash collisions, thereby having a computational complexity of $O(1)$ irrespective of the operation [54].

We observe that the lookup table size can grow quickly (see Figure 5) with the number of big/small cores, DVFS settings, and the number of load buckets. Let us say there are B big cores and S small cores available, and B_f and S_f available DVFS settings for big and small cores, respectively, then the total number of actions in the table would be $C(B+B_f,\ B_f) \times C(S+S_f,\ S_f)$, where C(x, y) refers to the number of different ways to choose actions (y) for out of x distinct elements. This grows in the order of $O(B^{B_f} \times S^{S_f})$.

Figure 5 shows the unoptimized (or naïve) lookup table size in kB when the number of big cores is equal to number of small cores ($B = S$) with 10 DVFS states for big and small cores ($B_f = S_f = 10$) with 100 load buckets (w). For instance, 64 cores of each type leads to 40,960,000 combinations (in contrast to 56 configurations on ARM Juno), which gives a table size of approximately 327 MB for a whole node.

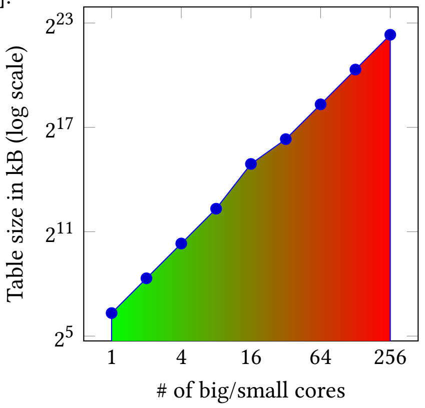


Fig. 5. Lookup table size (log scale) with same number of small and big cores, using 10 different DVFS settings, and bucket size of 100.

To reduce the size of the lookup table in Hipster, we store the configuration and reward only for the configurations which have been explored so far. The lookup table is still implemented using a Python dictionary as the complexity is $O(1)$ irrespective of the operation. Furthermore, in the exploitation phase, we ensure that each load bucket (w) contains *only* the top K (in our case, $K = 5$) configurations with highest discounted maximum rewards.

4 EVALUATION

4.1 Cloud Benchmarks

We evaluate Hipster using two latency-critical applications, **Memcached** and **Web-Search**, which have distinct characteristics and impact on shared resources [41]. **Memcached** [25] is an open source implementation of an in-memory key-value store for data caching used in many services from Twitter, Facebook and Google [4, 51]. The backend of **Web-Search** [6, 35] is an instance of Elasticsearch [21], an open source implementation of a search engine used by many companies including Netflix and Facebook.

The load generator (Faban) for Memcached and Web-Search is adapted from CloudSuite 3.0 [25]. It is configured to model diurnal load changes (Figure 1), simulating a period of 36 hours [47]; each hour in the original workload corresponds to one minute in our experiments. For the batch workloads [44, 71], we use the programs from the SPEC CPU 2006 suite [30].

For Memcached, we define the tail latency to be the 95th percentile request latency, with a target of 10 ms; for Web-Search, we define it to be the 90th percentile query latency, with a target of 500 ms. The QoS targets are based on the specification of CloudSuite [25]. Table 1 lists the two latency-critical applications, their configurations, maximum loads, and target tail latency in milliseconds. For each latency-critical workload, the maximum load is chosen so that the platform is able to meet the tail latency when running on the big cores at maximum DVFS state.

We specify the QoS targets according to the capacity and characteristics of our target platform. We first ran memcached, without Hipster, pinning it to the two big cores at the highest DVFS setting (2B0S-1.15 GHz). We observed that the minimum achievable tail latency target, for a 95% QoS guarantee, was 3 ms. This justifies using the 10 ms target from the CloudSuite specification.

4.2 Heterogeneous Platform

We perform the evaluation experiments on an ARM Juno R1 developer board [2] with Linux (kernel 4.3). The Juno board is a 64-bit ARMv8 big.LITTLE architecture with two high-performance out-of-order Cortex-A57 (big) cores and four low-power in-order Cortex-A53 (small) cores. The cores are integrated on a single chip with off-chip 8 GB DRAM. The two big cores form a cluster with a shared 2 MB L2 cache, and the four small cores form another cluster with a shared 1 MB L2 cache. The big cores are capable of DVFS from 0.6 GHz up to 1.15 GHz, whereas the small cores are fixed at 0.65 GHz. The architecture is schematically shown in Figure 6. A cache coherent interconnect (CoreLink CCI-400) provides full cache coherency among the heterogeneous cores, allowing a shared memory application to run on both clusters simultaneously. The workload generator runs on another machine: an AppliedMicro X-Gene2 64-bit ARMv8-A [48] with eight cores at 2.4 GHz and 128 GB DRAM.

4.2.1 Power efficiency. The power consumption of the Juno platform is obtained by reading specific hardware registers [1]. These registers report separately the power consumed by the big cluster, small cluster, and the rest of the system (Juno's *sys* register [3]). The power consumption of the Mali GPU is also available, but it is negligible because the GPU is disabled in all our experiments. Performance is quantified using the IPS measured by hardware performance counters.

Table 2 summarizes the power and performance characteristics of the big cluster (2 cores) and the small cluster (4 cores). We characterize the heterogeneous platform by running a compute-intensive microbenchmark (same as used in Section 3.6) consisting of mathematical operations without memory accesses. We report the system power consumption as the sum of the big and small clusters and the rest of the system (including memory controllers, etc). For the per-cluster comparison, we run the microbenchmark on all the cores in the cluster (two big cores vs four small cores). For a per-core comparison, we run the microbenchmark either on a single big core or on a single small core.

Considering system power, a single big core is 52% more power-efficient than a single small core, in terms of IPS per watt (Table 2). However, taking into account all cores in a cluster, and assuming that all cores can be fully utilized, a small cluster is 25% more power-efficient than a big cluster. This discrepancy is due to the rest of the system, outside the core clusters, which consumes about the same power as a big core at full utilization (0.76 W). If we subtract the power of the rest of the system, a single small core is 2.3× more power-efficient than a big core. We notice that small cores are attractive for throughput-oriented workloads, because of improved power efficiency. Big cores are, however, still needed for latency-critical workloads with tight QoS constraints, as a result of computationally-intensive single-threaded requests.

4.3 HipsterIn Results

This section evaluates the effectiveness of HipsterIn, as a policy for managing a single interactive workload. The objective is to minimize system energy consumption while satisfying QoS.

For HipsterIn, we set the learning phase to be 500 seconds, except when quantifying the learning time, where we set it to 200 seconds. In addition, we quantify HipsterIn with changes in application at runtime, where we run each application for 1500 seconds. In deploying Octopus-Man for comparisons, we first performed a sweep on the danger and safe thresholds, and picked the combination of thresholds with the highest QoS guarantee.

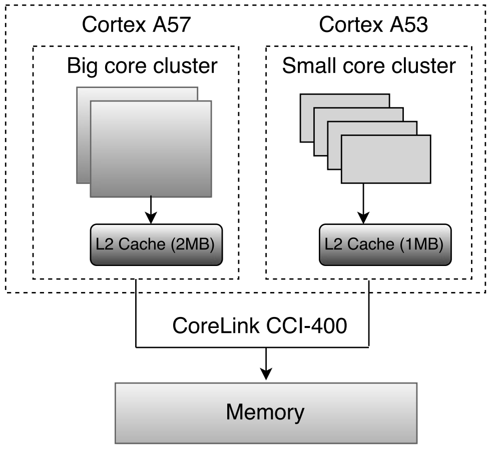


Fig. 6. Heterogeneous processor platform (ARM Juno R1)

App	Workload Configuration	Max. Load	Target Tail latency (ms)
Memcached	Twitter caching server of 1.3 GB	36 000 RPS	10 (95%ile)
Web-Search	English Wikipedia Zipfian distribution	44 QPS | think-time of 2sec[23]	500 (90%ile)

Table 1. Workload configurations, maximum load while meeting the target tail latency with two big cores for latency-critical applications.

	Power (Watts)		*Perf.(IPS* $\times 10^6$*)*	
Core type (GHz)	All cores	One core	All cores	One core
Big A57 (1.15)	2.30	1.62	4,260	2,138
Small A53 (0.65)	1.43	0.95	3,298	826

Table 2. Power and performance characterization on Juno platform using microbenchmark from section 4.1

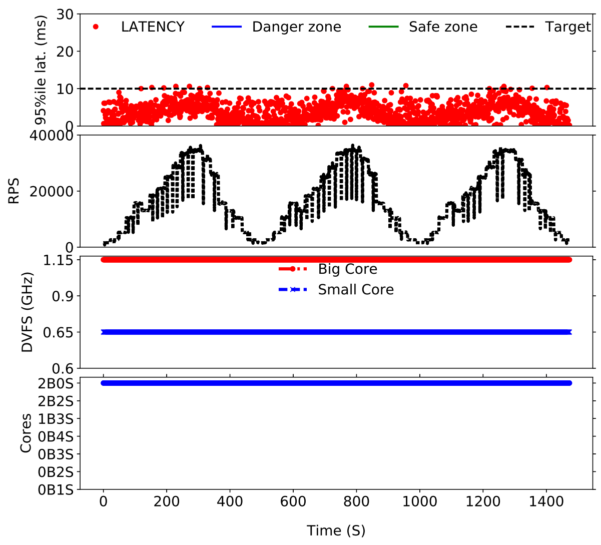


Fig. 7. Static mapping (all big cores) on Memcached

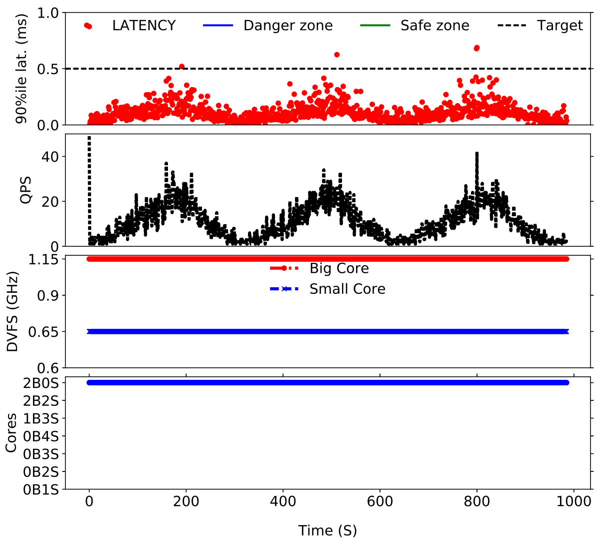
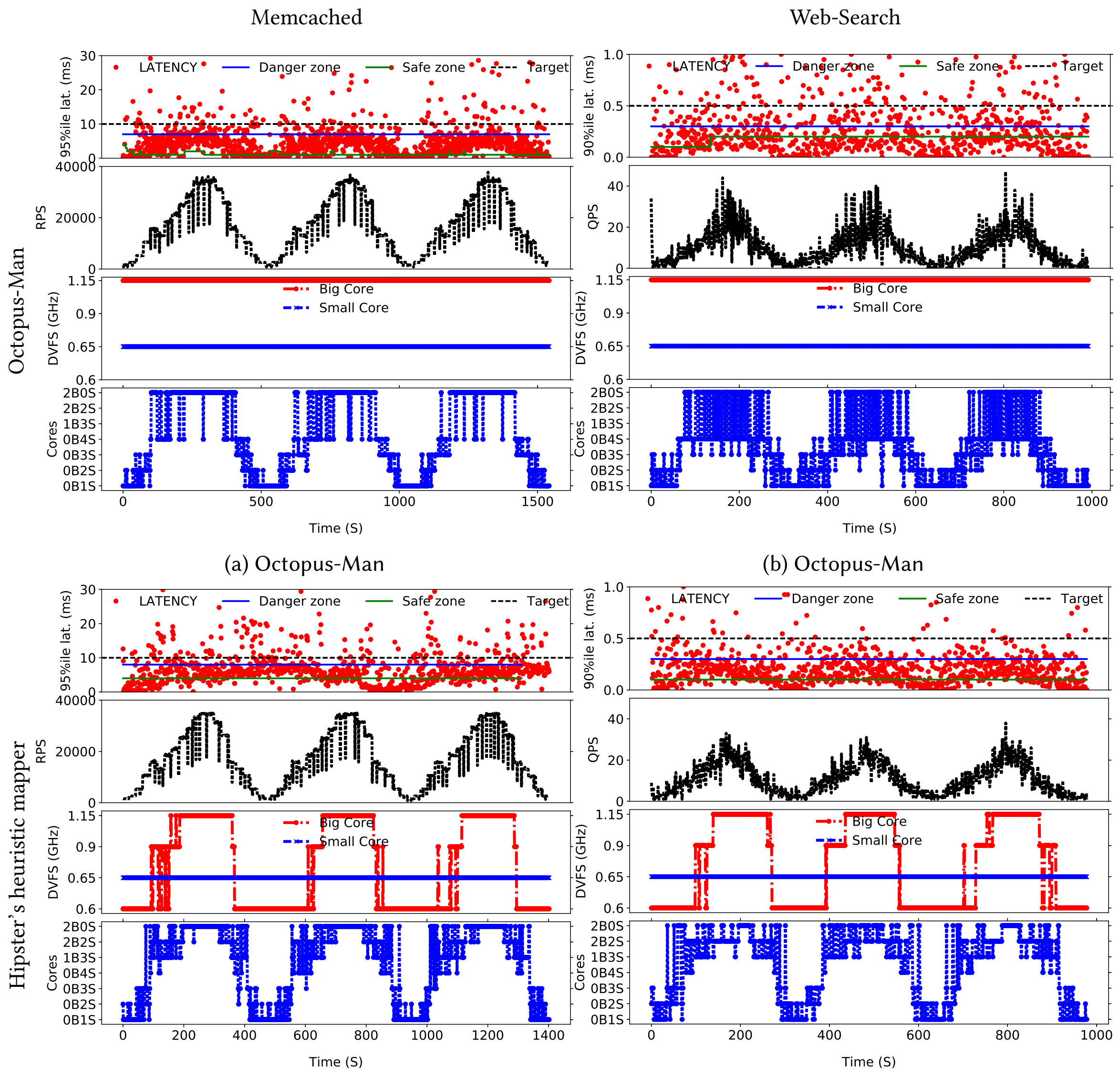


Fig. 8. Static mapping (all big cores) on Web-Search

4.3.1 ***Hipster's Heuristic Policy (interactive only)*.** We first evaluate the effectiveness of Hipster's heuristic policy alone, for mapping interactive workloads. Figure 7 and 8 show the results for both Memcached and Web-Search with static mapping, for which the interactive threads are mapped to the two big cores at highest DVFS of 1.15 GHz. Figure 9 shows the results for both workloads: Memcached (left-hand column, subfigures (a) and (c)) and Web-search (right-hand column, subfigures (b) and (d)). The columns, from left to right, correspond to Octopus-Man ((c) and (d)) and Hipster's heuristic policy ((e) and (f)). For each subfigure, from top to bottom, the first plot presents the tail latency (QoS), with the target marked with a dashed line. The second plot shows the achieved throughput in RPS (requests per second). The third plot presents the DVFS of the big and small cores, and the fourth plot represents the choice of core mapping.



(a) Octopus-Man

(b) Octopus-Man

(c) Hipster's heuristic mapper

(d) Hipster's heuristic mapper

Fig. 9. Comparison of Hipster's heuristic policy (bottom) with Octopus-Man (top). Results are shown for Memcached (left-hand column) and Web-Search (right-hand column) on ARM Juno R1.

Comparing the DVFS and core configuration subplots, we observe that Hipster's heuristic policy is successfully exploring the DVFS settings available on the Juno platform (third plots), and it is exploring all configurations including those that use both big and small cores at the same time (bottom plots). In contrast, Octopus-Man does not adjust the DVFS settings and it uses either the big or small cores, but not both at once.

Both Octopus-Man and Hipster's heuristic policy frequently oscillate between consecutive core configurations. In the case of Octopus-Man, there are clear oscillations between two big cores and four small cores, for example between the 600th and 800th seconds of Memcached. Using two big cores satisfies QoS but since it is within the safe zone, Octopus-Man switches to four small cores, which enters the danger zone, generating an alert provoking a return to two big cores. Such oscillations between cores in different clusters leads to severe QoS degradation of up to 20%. As expected, the static mapping (all big cores) has the least number of violations.

In summary, although Hipster's heuristic policy alone improves over Octopus-Man by exploiting a wider search space, it still suffers from an unacceptable number of QoS violations.

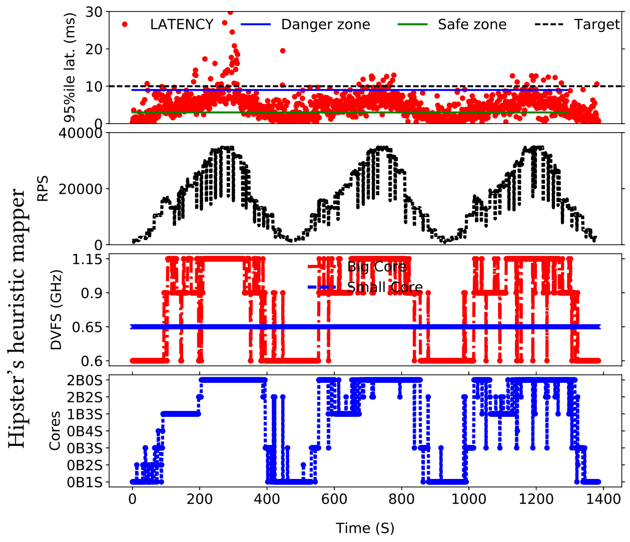


Fig. 10. HipsterIn on Memcached

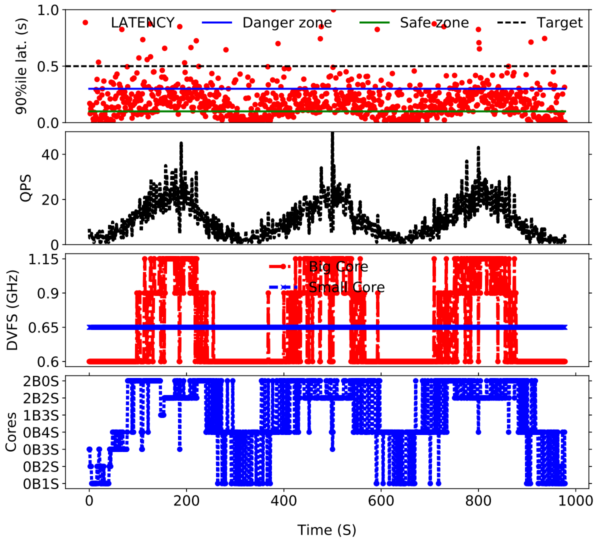


Fig. 11. HipsterIn on Web-Search

4.3.2 ***HipsterIn: Memcached Results.*** Figure 10 shows the results using HipsterIn for Memcached. After completing the learning phase, the oscillatory effect between core mappings is greatly reduced (by 8.3%), and overall the QoS guarantee is improved by 24% compared with the learning phase. HipsterIn performs well because it moves directly to the appropriate core configuration for a given load that satisfies QoS. In addition to switching between a combination of different cores, HipsterIn also explores more fine-grained DVFS adaptations, which has lower overheads (of microseconds) compared with migrations between cores (order of milliseconds) [36].

4.3.3 ***HipsterIn: Web-Search Results.*** Figure 11 shows the results using HipsterIn for Web-Search. In contrast to the heuristic policies (Octopus-Man and Hipster's heuristic), during the exploitation phase, HipsterIn monitors the QoS and dynamically adjusts the core mapping and DVFS settings to adapt to load fluctuations. Both Hipster's heuristic and Octopus-Man perform aggressive changes to core mappings to reduce energy, leading to a negative impact on QoS. On the other hand, HipsterIn shows a more balanced behavior, performing 4.7× fewer task migrations than Octopus-Man for Web-Search, while improving QoS up to 16% and reducing energy consumption by 13.5%.

4.3.4 ***HipsterIn: Swapping applications.*** We also evaluate the effectiveness of HipsterIn to adapt to changes in application at runtime and deliver real-time performance guarantees even for the new incoming application.

Web-Search to Memcached. Figure 12 shows the results using HipsterIn when swapping from Web-Search (represented in red) to Memcached (represented in cyan) after 1500 s of execution. "NA" in the last two subplots on y-axis represents the period when Memcached or Web-Search are running exclusively. HipsterIn shows a more balanced behavior, performing 22% fewer violations than learning phase, set at 500-seconds, when running Web-Search. Thereafter, when swapping Web-Search with Memcached observe that HipsterIn has a far more QoS violations but quickly adapts to changes in application by learning the lookup table suitable for Memcached.

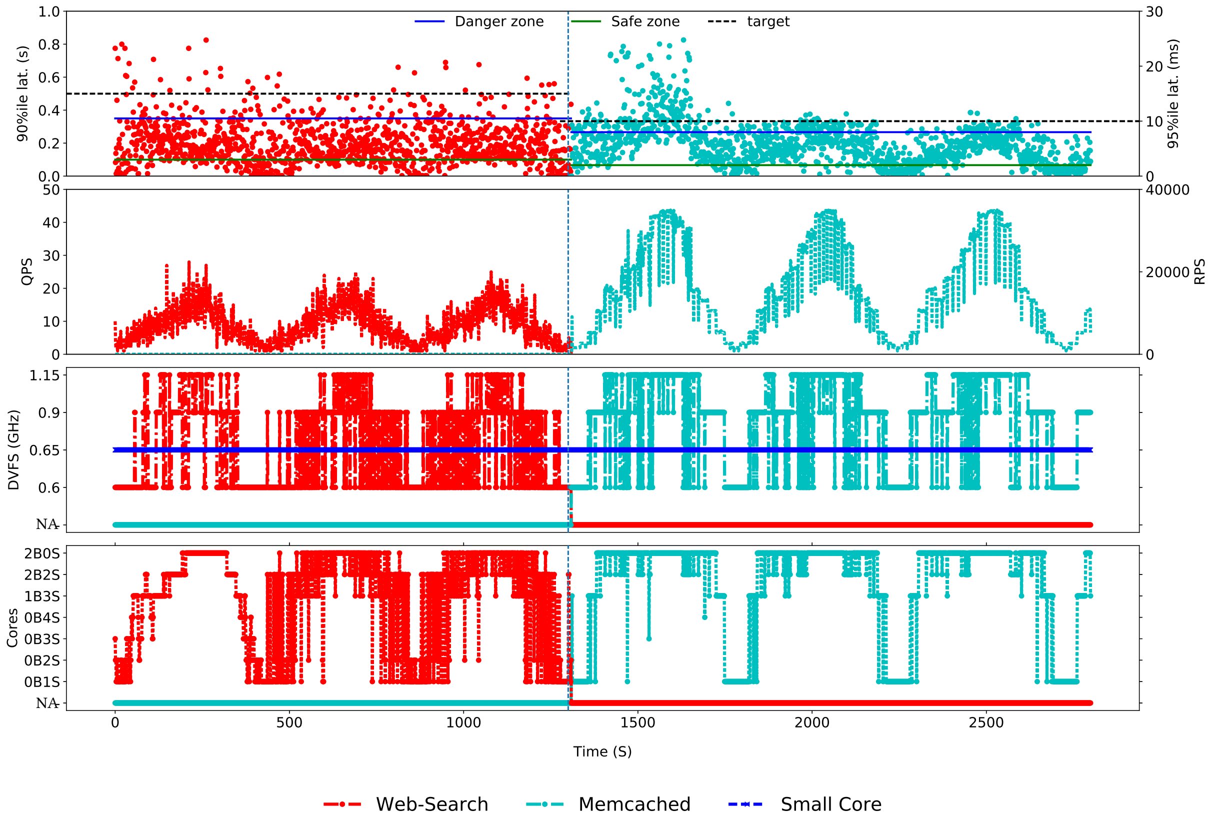


Fig. 12. HipsterIn on Web-Search swapping to Memcached

Memcached to Web-Search. Figure 13 shows the results using HipsterIn when swapping from Memcached (represented in cyan) to Web-Search (represented in red) after 1500 s of execution. "NA" in the last two subplots on y-axis represents the period when Web-Search or Memcached are running exclusively. When application is swapped at runtime from Memcached to Web-Search, HipsterIn dynamically updates the lookup table to adapt to changes in application behavior. For instance, observe at 1500 s, HipsterIn has large number of violations (by 7.1%) but after 2000 s the number of violations drop as HipsterIn adapts the lookup table and the overall QoS guarantee is improved by 18%. The QoS guarantee of HipsterIn for Web-Search is 95%, which is similar to QoS guarantee when running Web-Search from the start (Table 3)

*4.3.5 **HipsterIn Summary**.*

Table 3 summarizes the QoS guarantee, QoS tardiness and energy reduction for Memcached and Web-Search for different policies: Static (all big cores), Static (all small cores), Hipster's heuristic mapper, Octopus-Man and HipsterIn. We compare the energy consumption of each mapping schema against Static (all big cores). We quantify the QoS behavior at each sampling interval by assessing the measured QoS using two metrics: QoS guarantee and QoS tardiness.[3] The QoS Guarantee is the percentage of samples for which the measured QoS did not violate the target (100%-QoS violations%). The QoS tardiness in the table is the average (mean) of the QoS tardiness, including only the samples that violated the QoS target.

As shown in Table 3, for Web-Search and Memcached, static (all small cores) cannot meet the required QoS. On the one hand, the heuristic policies reduce energy marginally, but violate QoS due to excessive core migrations. On the other hand, HipsterIn meets QoS at 99.4% and 96.4% for Memcached and Web-Search, while having energy savings of 14.3% and 17.8%, respectively.

[3]QoS Tardiness is $QoS_{\text{curr}}/QoS_{\text{target}}$, using the definitions from Section 3.7.

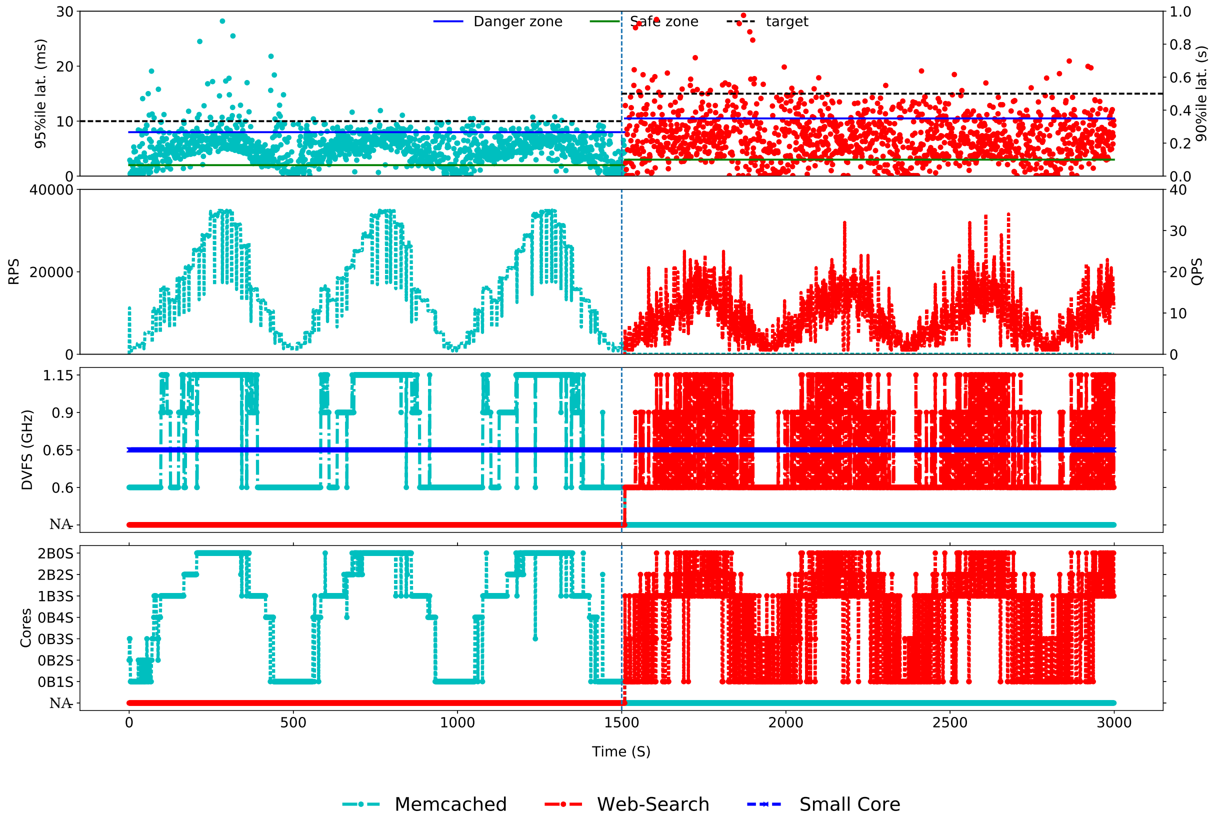


Fig. 13. HipsterIn on Memcached swapping to Web-Search

	QoS Guarantee		*QoS Tardiness*		*Energy Reduction*	
	Memcached	*Web − Search*	*Memcached*	*Web − Search*	*Memcached*	*Web − Search*
Static (all big cores)	99.5%	99.5%	1.1	1.3	-	-
Static (all small cores)	85.8%	78.4%	1.4	2.0	48.0%	31.0%
Hipster′s Heuristic	89.9%	95.3%	1.8	1.9	18.7%	13.6%
OctopusMan	92.0%	80.0%	2.2	2.1	17.2%	4.3%
HipsterIn	**99.4%**	**96.5%**	1.4	2.0	**14.3%**	**17.8%**

Table 3. HipsterIn: summary of QoS guarantees, tardiness and energy savings for Memcached (Mem) and Web-Search (Web).

4.3.6 ***HipsterIn Analysis.***

Rapid adaptation to load changes. We show that Hipster can respond to sudden or rapid changes in load by directly mapping to a configuration while satisfying the QoS. To demonstrate this, we perform an experiment by deploying Memcached and vary the load from 50% to 100% of the maximum load every 10 s and measure the 95th percentile tail latency, expressed as QoS Tardiness. A QoS violation has occurred if the QoS Tardiness is above 1, otherwise QoS is satisfied. Figure 14 shows the impact on QoS (bottom) upon sudden variations in load (top) for both, HipsterIn (during the exploitation phase) and

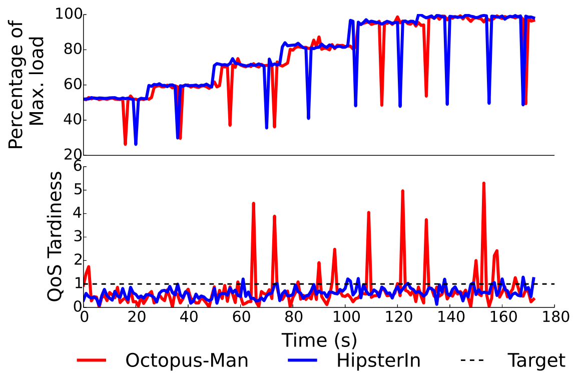


Fig. 14. Percentage of Max. load, and Tail latency (QoS Tardiness) running Memcached with HipsterIn and Octopus-Man.

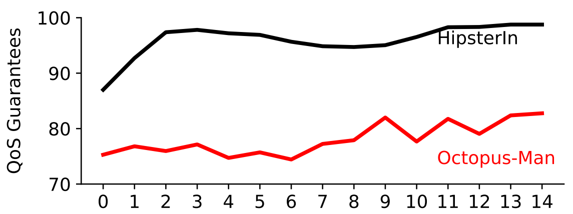


Fig. 15. QoS Guarantees of HipsterIn and Octopus-Man. Each data point represents the QoS guarantees over 100 s intervals.

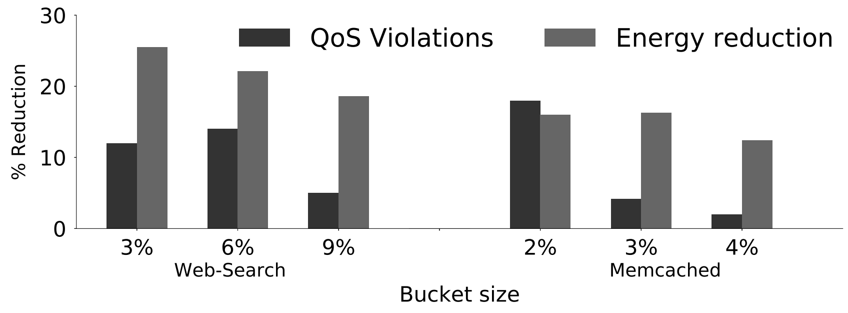


Fig. 16. Impact of bucket size on HipsterIn QoS guarantees, and energy savings, normalized to static (all big cores) on Web-Search and Memcached.

Octopus-Man. We find that Octopus-Man violates QoS due to aggressive core switching to minimize energy consumption. By contrast, HipsterIn achieves more stable tail latency even at higher load (80%). Note that, from 75% to 90% of the load, the QoS tardiness (extent of violation) experienced by HipsterIn is 3.7× (mean) lower than Octopus-Man.

Impact of learning time. HipsterIn aims to deliver the best balance between QoS guarantee and energy reduction compared to heuristic policies (Table 3). In practice, to best optimize for energy efficiency, and to improve QoS, HipsterIn needs a short learning phase. Figure 15 shows the QoS guarantee and energy distribution over 100 s intervals for Web-Search, for both HipsterIn and Octopus-Man. Each data point in the graph refers to a 100-second interval. The learning phase is set to 200 s. As can be seen, HipsterIn quickly learns during the heuristic phase, which improves QoS guarantees. On the other hand, for Octopus-Man, the QoS guarantees are consistently around the 80% mark, since it does not use past decisions and their associated effects to improve the future decisions.

Impact of bucket sizes. Figure 16 shows the impact on QoS and energy savings when varying the load bucket sizes in Hipster. The x-axis represents the bucket size, expressed as the percentage of maximum load. Each bar in the figure (y-axis) represents the QoS violations and energy reductions normalized to Static (all big cores). Using a large bucket size forces Hipster to use the same core configuration across a wide range of loads, whereas using a small bucket size allows fine-grained control. A small bucket size therefore improves the energy savings, but it tends to cause rapid changes in core configuration for small changes in load, and doing so incurs a larger number of QoS violations. On the other hand, larger bucket sizes provide better QoS guarantee but lower energy savings, because they categorize large variations in load into a single load

	Lookup Table	Memcached	Web-Search
QoS Guarantee (%)	Unoptimized	99.4	96.5
	Optimized	99.2	96.0
# of config./load bucket	Unoptimized	13	13
	Optimized	5	5
Table size	Unoptimized	1344	392
	Optimized	120 (*91% savings*)	35 (*91% savings*)

Table 4. Results of the table size optimization in Hipster

bucket. Therefore, in tuning Hipster, we empirically determine the bucket size to maximize energy savings subject to at least 98% QoS guarantee.

Optimizing the lookup table size. Table 4 summarizes the QoS guarantee, number of configurations per load bucket, and number of configurations for Memcached and Web-Search with unoptimized and optimized lookup tables. We quantify the reduction in lookup table size by assessing the number of configurations stored for each load bucket at the end of the execution. As shown in Table 4, the optimized lookup table saves 91 % of the memory footprint while providing similar QoS guarantee to unoptimized lookup table because, it only stores those configurations that have been explored in the learning phase and in the exploitation phase, the size of the table is further reduced by keeping only the top five configurations with the highest reward. This is possible, as the configurations explored at a given load level for a particular application are chosen from only a subset of the configurations in the learning phase instead of random solutions as in Q-Learning.

4.4 HipsterCo Results

This section evaluates the effectiveness of HipsterCo, as a policy for collocating a single latency-critical workload and a mix of batch workloads. The objective is to maximize the throughput of the batch workloads while satisfying QoS of the interactive workloads.

Figure 17 shows the QoS guarantee (top), throughput (middle) and energy consumption (bottom) for Web-Search collocated with batch workloads, managed by Octopus-Man and HipsterCo. All figures are normalized to a static mapping that allocates the latency-critical workload to the two big cores and the batch workloads to the four small cores. The number of running batch workloads is equal to the number of cores not utilized by Web-Search. We report the system throughput by aggregating the IPS of all batch programs.

As shown in the top plot of Figure 17, HipsterCo consistently delivers 94% QoS guarantees, whereas Octopus-Man has much lower QoS guarantees of 76%. This is because HipsterCo learns from the QoS behavior and performance history and is able to jump directly to a core mapping and DVFS state that satisfies QoS. As a result, it incurs fewer core migrations compared with Octopus-Man (see Section 4.3.3), so it achieves superior QoS guarantees.

As shown in the middle plot of Figure 17, for all benchmarks, Hipster and Octopus-Man deliver much higher throughput compared to static mapping, with an average of 2.3× and 2.6× improvement, respectively. Both task managers improve performance compared with the static mapping because they migrate the latency-critical workload to small cores during periods of low load, allowing the batch workloads to run on big cores (which can be 2.6× more powerful than small cores). For *Calculix*, a compute-bound application, HipsterCo achieves the highest throughput improvement over static of 3.35×, and for *libquantum*, a memory-bound program, the least improvement is still 1.6×.

As shown in the bottom plot of Figure 17, HipsterCo reduces the energy consumption to an average of 80% of static, whereas Octopus-Man increases energy to an average of 1.2 times that of static. This is because, as shown in Figure 2, Hipster explores a wider range of core configurations, including DVFS settings and mixing core types. In contrast, Octopus-Man only allows the latency-critical workload to occupy a single cluster and each cluster is set to the highest DVFS.

HipsterCo sometimes chooses a different performance–energy tradeoff than Octopus-Man. An example is *lbm*, a memory bound workload, for which HipsterCo delivers 40% the throughput of Octopus-Man, but 31% lower energy. There are two

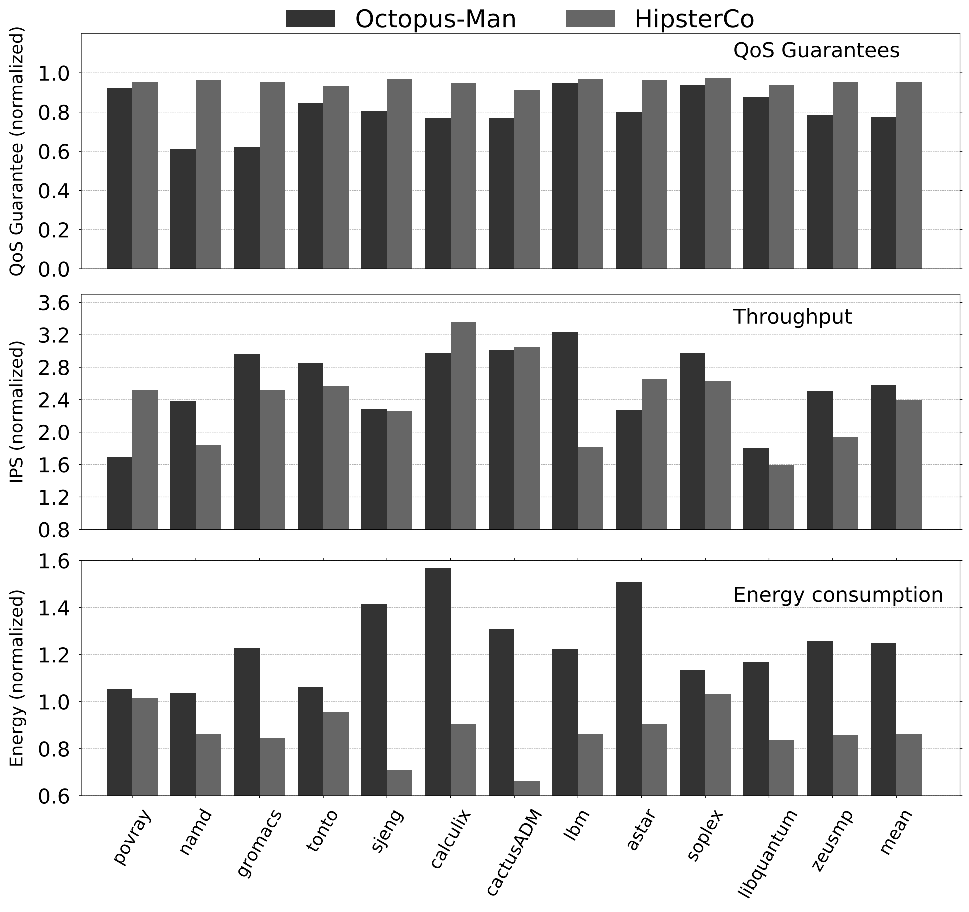


Fig. 17. QoS guarantee (top), Throughput improvement (middle) and Energy consumption (bottom) when Web-Search is collocated with batch workloads. The results are normalized to static all big cores.

main reasons for this. Firstly, when HipsterCo uses DVFS for the latency-critical workloads, this DVFS setting also applies to batch workloads running in the same cluster, reducing both batch throughput and system energy. Secondly, HipsterCo sometimes uses a larger number of cores at lower DVFS, leaving fewer resources available for the batch workloads. As a result, on average, HipsterCo marginally reduces performance (by 7%) but it delivers energy savings of 33%, both compared with Octopus-Man.

5 DEPLOYMENT METHODOLOGY FOR NEW WORKLOADS

Hipster is a user-level daemon written in Python that is designed to implement a hybrid reinforcement learning approach under Linux running on multicore machines. It is released under General Public License (GPL) v3 and its source is available for download.[4] To deploy Hipster, we perform a one-time platform profiling to capture the power and performance characteristics of the available hardware knobs (similar to Octopus-Man [56]). Next, the Hipster runtime is launched to manage a given workload on a physical machine.

[4]https://bitbucket.org/rnishtala/hipster.git

5.1 Platform Profiling

We profile the system by running a CPU-intensive stress microbenchmark (no memory access) at different core mappings and DVFS settings to gather performance (such as IPS, instructions per second) and power statistics.[5] The performance and power statistics can be obtained using performance monitoring tools (see section 3.10) and power meters (see section 4.1), respectively. Next, we sort and store the core configurations (the actions that will be used by Hipster) in the increasing order of the system's average power efficiency (Performance per Watt). The system power is obtained by running the microbenchmark on all cores at the highest DVFS setting.

To deploy a new latency-critical workload to be managed by Hipster, we quantify the maximum load (measured in terms of queries/requests per second or similar) the system can handle running on the most power-efficient configuration while satisfying the QoS target (measured in ms or s). This is a precursor to ensure the maximum load the system can handle without violating a given tail latency target.

5.2 Hipster Runtime

Hipster is invoked periodically, with the period determined by the sampling interval of the latency-critical workload. Hipster takes as input the *system power consumption*, *QoS target*, *load*, *maximum load* and *latency* (measured in ms or s) of the latency-critical workload and finally, the *performance* (measured in IPS) of the throughput-oriented workload (if any). The performance metrics such as QoS target, load and latency can be obtained either from the application directly (as in our case) or an external profiling tool such as Google Wide Profiler (GWP) [61].

To make the best use of Hipster, we suggest tuning the upper threshold (QoS_{D}) and lower threshold (QoS_{S}), learning factor (α), discounting factor (γ) empirically. Nevertheless, in the current implementation we predefine the upper threshold and lower threshold to 80% and 20% of the QoS target, respectively; and the learning factor and discounting factor are set to 0.6 and 0.9, respectively.

Our implementation assumes that there exists an ACPI (Advanced Configuration and Power Interface) governor [9] that allows external changes in DVFS state. The governor selected on our ARM processor is userspace. The deployment stage of Hipster needs permission to access Ring 0 or root as the ACPI module writes changes in DVFS state to the sysfs. We provide an example of Hipster's deployment method.

Hipster schedules the workloads using Hipster's heuristic algorithm and reinforcement learning mechanism. For a multicore server, we use Hipster to dynamically select the core mapping and DVFS state to "just meet" the QoS based on the load for an interactive workload and any remaining resources are allocated to throughput-oriented workloads (if any).

6 RELATED WORK

Bubble-flux and Bubble-Up [46, 72] detect at runtime the memory pressure and find the optimal collocation of processes to avoid negative interference within latency-critical workloads. They also have a mechanism to detect negative interference allocations via execution modulation. However such fall-back mechanism would not adhere to applications like Memcached, as modulations have to be done at a finer granularity. DeepDive [53] identifies and manages performance interference between VM systems collocated on the same system. Q-Clouds [50] develop a feedback based mechanism to tune resource assignment to avoid negative interference to collocated VMs. CPI2 [73] enables race-to-finish for low-priority workloads to not have a deadlock with high priority services.

Octopus-Man [56] was designed for big.LITTLE architectures to map workloads on big and small cores at highest DVFS state using a feedback controller in response to changes in measured latency. Heracles [41] uses a feedback controller that exploits collocation of latency-critical and batch workloads while increasing the resource efficiency of CPU, memory and network as long as QoS target is met. However, this work explores modern Intel architectures with cache allocation technology (CAT) and DRAM bandwidth monitor, which are available from Broadwell processors released after 2015. Pegasus [40] achieves high CPU energy proportionality for low latency workloads using fine-grained DVFS techniques. TimeTrader [66] and Rubik [36] exploit request queuing latency variation and apply any available slack from queuing delay to throughput-oriented workloads to improve energy efficiency. Quasar [19] use runtime classification to predict interference and collocate workloads to minimize interference.

Carvalha et al. [10] introduced an approach to allocate a subset of the unused resources for long-term availability SLOs. This methodology deploys time series forecasting under the premise that availability of resource usage patterns in short jobs.

[5]The stress microbenchmark is executed on each of the available cores simultaneously and is given by "stress_cpu.c" in the source code.

Nevertheless, this approach does not handle resource allocation effectively as short jobs do not have consistent resource usage patterns. Furthermore, the approach ignores the unused and underused resources caused by time-varying demands in data center environments.

Tarcil [20] and Firmament [27] uses information on the type of resources applications need in a sampling interval to deploy in distributed cluster which uses an analytically-derived sampling framework that provides high quality resources within a few milliseconds.

Violaine et al. [67] develop a dynamic resource allocation algorithm which considers each the architectural characteristics of the infrastructure such as performance, energy consumption, and their turn on/off latency. Using such information, the framework makes decisions of resource reallocation to ensure that there exist as little as possible QoS violations for latency-critical workloads while having potential energy gains.

KnightShift [68] introduces a server architecture that couples commercial available compute nodes to adapt the changes in system load and improve energy proportionality. Autoscale [60] is for load-balancing a single workload, whereas Hipster could be used for multi-tenant data centers (different workloads on different nodes). Also, Autoscale cannot exploit heterogeneity properly. In contrast, at low utilization, Hipster can use the small cores for the latency-critical workloads and leave the big cores for batch workloads.

Tesauro et al. [65] use an *offline model* based on heuristics for autonomous resource allocation, which may be limited to specific architectures or applications. Building a lookup table at runtime is important because applications have diverse power and performance characteristics which need to be learnt individually (as shown in Section 2). Prior work has previously introduced optimizations for lookup tables, which include building a priority queue for each load bucket, and eliminating configurations that seldom occur, and controlling the table size using function approximations as in [34, 66].

7 CONCLUSION

We propose Hipster, a hybrid scheme that combines heuristics and reinforcement learning to manage heterogeneous cores with DVFS control for improved energy efficiency. We show that Hipster performs well across workloads and interactively adapts the system by learning from the QoS/power/performance history to best map workloads to the heterogeneous cores and adjust their DVFS settings. When only latency-critical workloads are running in the system, Hipster reduces energy consumption by 13% in comparison to prior work. In addition, to improve resource efficiency in shared data centers by running both latency-critical and batch workloads on the same system, Hipster improves batch workload throughput by 2.3× compared to a static and conservative policy, while meeting the QoS targets for the latency-critical workloads.

REFERENCES

[1] ARM. 2016a. ARM Juno Power Registers. (December 2016). ARMRegistershttps://github.com/ARM-software/devlib/blob/master/src/readenergy/readenergy.c

[2] ARM. 2016b. ARM Juno R1. (December 2016). https://goo.gl/EcamOa

[3] ARM. 2016c. SYS_POW_SYS Register. (December 2016). https://goo.gl/fmTTQi

[4] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload analysis of a large-scale key-value store. In *SIGMETRICS*.

[5] Luiz Andre Barroso, Jimmy Clidaras, and Urs Hölzle. 2013. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition. *Synthesis Lectures on Computer Architecture* 8, 3 (7 2013), 1–154. DOI:http://dx.doi.org/10.2200/S00516ED2V01Y201306CAC024

[6] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. 2003. Web search for a planet: The Google cluster architecture. *IEEE Micro* 23, 2 (March 2003), 22–28. DOI:http://dx.doi.org/10.1109/MM.2003.1196112

[7] David Bernstein. 2014. Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing* 1, 3 (9 2014), 81–84. DOI:http://dx.doi.org/10.1109/MCC.2014.51

[8] Ozlem Bilgir, Margaret Martonosi, and Qiang Wu. 2011. Exploring the Potential of CMP Core Count Management on Data Center Energy Savings. In *3rd Workshop on Energy Efficient Design (WEED)*.

[9] Dominik Brodowski. 2017. CPU frequency and voltage scaling code in the Linux kernel. (February 2017).

[10] Marcus Carvalho, Walfredo Cirne, Franciso Brasileiro, and John Wilkes. 2014. Long-term SLOs for reclaimed cloud computing resources. In *ACM Symposium on Cloud Computing (SoCC)*. Seattle, WA, USA, 20:1–20:13. http://dl.acm.org/citation.cfm?id=2670999

[11] Nagabhushan Chitlur, Ganapati Srinivasa, Scott Hahn, P K Gupta, Dheeraj Reddy, David Koufaty, Paul Brett, Abirami Prabhakaran, Li Zhao, Nelson Ijih, Suchit Subhaschandra, Sabina Grover, Xiaowei Jiang, and Ravi Iyer. 2012. QuickIA: Exploring heterogeneous architectures on real prototypes. In *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 1–8. DOI:http://dx.doi.org/10.1109/HPCA.2012.6169046

[12] Jason Cong and Bo Yuan. 2012. Energy-efficient scheduling on heterogeneous multi-core architectures. In *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design - ISLPED '12*. ACM Press, New York, New York, USA, 345. DOI:http://dx.doi.org/10.1145/2333660.2333737

[13] CortexA53, ARM. 2016. ARM ® Cortex ® -A53 MPCore Processor Technical Reference Manual. (December 2016).

[14] CortexA57, ARM. 2016. ARM ® Cortex ® -A57 MPCore Processor Revision: r1p0 Technical Reference Manual. (December 2016).

[15] Jeffrey Dean and Luiz Andre Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2 2013), 74. DOI:http://dx.doi.org/10.1145/2408776.2408794

[16] Pierre Delforge and Josh Whitney. 2014. Data Center Efficiency Assessment. *Natural Resources Defense Council (NRDC)* (2014).

[17] Christina Delimitrou and Christos Kozyrakis. 2013a. Paragon: QoS-aware Scheduling for Heterogeneous Datacenters. *SIGPLAN Not.* 48, 4 (March 2013), 77–88. DOI:http://dx.doi.org/10.1145/2499368.2451125

[18] Christina Delimitrou and Christos Kozyrakis. 2013b. QoS-Aware Scheduling in Heterogeneous Datacenters with Paragon. *ACM Trans. Comput. Syst.* 31, 4, Article 12 (Dec. 2013), 34 pages. DOI:http://dx.doi.org/10.1145/2556583

[19] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. *ACM SIGARCH Computer Architecture News* 42, 1 (4 2014), 127–127. DOI:http://dx.doi.org/10.1145/2654822.2541941

[20] Christina Delimitrou, Daniel Sanchez, and Christos Kozyrakis. 2015. Tarcil: Reconciling Scheduling Speed and Quality in Large Shared Clusters. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 97–110. DOI:http://dx.doi.org/10.1145/2806777.2806779

[21] Elasticsearch. 2016. Elasticsearch. (December 2016). https://github.com/elastic/elasticsearch

[22] Schurman Eric and Brutlag Jake. 2009. The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search. *Velocity* (2009).

[23] Faban. 2016. Faban. (December 2016). http://faban.org/

[24] Facebook. 2016. Facebook is opening a new wind-powered data center in Texas. (December 2016). http://goo.gl/dKVnSB

[25] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. 2012. Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware. *SIGPLAN Not.* 47, 4 (March 2012), 37–48. DOI:http://dx.doi.org/10.1145/2248487.2150982

[26] Waclaw Godycki, Christopher Torng, Ivan Bukreyev, Alyssa Apsel, and Christopher Batten. 2014. Enabling Realistic Fine-Grain Voltage Scaling with Reconfigurable Power Distribution Networks. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-47)*. IEEE Computer Society, Washington, DC, USA, 381–393. DOI:http://dx.doi.org/10.1109/MICRO.2014.52

[27] Ionel Gog, Malte Schwarzkopf, Adam Gleave, Robert N. M. Watson, and Steven Hand. 2016. Firmament: Fast, Centralized Cluster Scheduling at Scale. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, GA, 99–115. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gog

[28] Marisabel Guevara, Benjamin Lubin, and Benjamin C. Lee. 2013. Navigating heterogeneous processors with market mechanisms. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. 95–106. DOI:http://dx.doi.org/10.1109/HPCA.2013.6522310

[29] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. 2016. Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction. In *HPCA*.

[30] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News* 34, 4 (9 2006), 1–17. DOI:http://dx.doi.org/10.1145/1186736.1186737

[31] Urs Hoelzle and Luiz Andre Barroso. 2009. The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines. (5 2009). http://dl.acm.org/citation.cfm?id=1643608

[32] Tibor Horvath, Tarek Abdelzaher, Kevin Skadron, and Xue Liu. 2007. Dynamic Voltage Scaling in Multitier Web Servers with End-to-End Delay Control. *IEEE Trans. Comput.* 56, 4 (4 2007), 444–458. DOI:http://dx.doi.org/10.1109/TC.2007.1003

[33] IBM. 2007. IBM Research | Technical Paper Search | Model-Based and Model-Free Approaches to Autonomic Resource Allocation(Search Reports). (2 2007).

[34] Engin Ipek, Onur Mutlu, José F. Martínez, and Rich Caruana. 2008. Self-Optimizing Memory Controllers: A Reinforcement Learning Approach. In *International Symposium on Computer Architecture*.

[35] Vijay Janapa Reddi, Benjamin C. Lee, Trishul Chilimbi, and Kushagra Vaid. 2010. Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency. *SIGARCH Comput. Archit. News* 38, 3 (June 2010), 314–325. DOI:http://dx.doi.org/10.1145/1816038.1816002

[36] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. 2015. Rubik: Fast Analytical Power Management for Latency-critical Systems. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 598–610. DOI:http://dx.doi.org/10.1145/2830772.2830797

[37] Jacob Leverich, Matteo Monchiero, Vanish Talwar, Parthasarathy Ranganathan, and Christos Kozyrakis. 2009. Power Management of Datacenter Workloads Using Per-Core Power Gating. *IEEE Computer Architecture Letters* 8, 2 (2 2009), 48–51. DOI:http://dx.doi.org/10.1109/L-CA.2009.46

[38] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. 2014. Tales of the Tail. In *SOCC*.

[39] Yang Li, Di Wang, Saugata Ghose, Jie Liu, Sriram Govindan, Sean James, Eric Peterson, John Siegler, Rachata Ausavarungnirun, and Onur Mutlu. 2016. SizeCap: Efficiently handling power surges in fuel cell powered data centers. In *HPCA*.

[40] David Lo, Liqun Cheng, Rama Govindaraju, Luiz Andre Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. *ACM SIGARCH Computer Architecture News* 42, 3 (10 2014), 301–312. DOI:http://dx.doi.org/10.1145/2678373.2665718

[41] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles. *ACM SIGARCH Computer Architecture News* (2015).

[42] David Lo and Christos Kozyrakis. 2014. Dynamic management of TurboMode in modern multi-core chips. In *HPCA*.

[43] Niti Madan, Alper Buyuktosunoglu, Pradip Bose, and Murali Annavaram. 2011. A case for guarded power gating for multi-core processors. In *HPCA*.

[44] Jason Mars and Lingjia Tang. 2013. Whare-map: heterogeneity in "homogeneous" warehouse-scale computers. *ACM SIGARCH Computer Architecture News* (2013).

[45] Jason Mars, Lingjia Tang, and Robert Hundt. 2011a. Heterogeneity in 'Homogeneous' Warehouse-Scale Computers: A Performance Opportunity. *IEEE Computer Architecture Letters* 10, 2 (2 2011), 29–32. DOI:http://dx.doi.org/10.1109/L-CA.2011.14

[46] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011b. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11*. ACM Press, New York, New York, USA, 248. DOI:http://dx.doi.org/10.1145/2155620.2155650

[47] David Meisner, Christopher M. Sadler, Luiz AndrÃľ Barroso, Wolf-Dietrich Weber, and Thomas F. Wenisch. 2011. Power management of online data-intensive services. In *Proceeding of the 38th annual international symposium on Computer architecture - ISCA '11*, Vol. 39. ACM Press, New York, New York, USA, 319. DOI:http://dx.doi.org/10.1145/2000064.2000103

[48] Applied Micro. 2016. Applied Micro XGene 2. (December 2016). http://goo.gl/XA04r1

[49] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2 2015), 529–533. DOI:http://dx.doi.org/10.1038/nature14236

[50] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds. In *Proceedings of the 5th European conference on Computer systems - EuroSys '10*. ACM Press, New York, New York, USA, 237. DOI:http://dx.doi.org/10.1145/1755913.1755938

[51] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. 2013a. Scaling Memcache at Facebook. In *USENIX conference on Networked Systems Design and Implementation*.

[52] Rajiv Nishtala, Daniel Mosse, and Vinicius Petrucci. 2013b. Energy-aware thread co-location in heterogeneous multicore processors. In *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. 1–9. DOI:http://dx.doi.org/10.1109/EMSOFT.2013.6658599

[53] Dejan Novaković, Nedeljko Vasić, Stanko Novaković, Dejan Kostić, and Ricardo Bianchini. 2013. DeepDive: transparently identifying and managing performance interference in virtualized environments. In *USENIX conference on Annual Technical Conference*.

[54] Richard Pattis. 2016. Complexity of Python Operations. (December 2016). https://www.ics.uci.edu/

[55] Linux project Perf. 2016. Perf: Linux profiling with performance counters. (December 2016). https://perf.wiki.kernel.org/

[56] Vinicius Petrucci, Michael A. Laurenzano, John Doherty, Yunqi Zhang, Daniel Mosse, Jason Mars, and Lingjia Tang. 2015. Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers. In *HPCA*.

[57] George Prekas, Mia Primorac, Adam Belay, Christos Kozyrakis, and Edouard Bugnion. 2015. Energy Proportionality and Workload Consolidation for Latency-critical Applications. In *Proceedings of the Sixth ACM Symposium on Cloud Computing (SoCC '15)*. ACM, New York, NY, USA, 342–355. DOI:http://dx.doi.org/10.1145/2806777.2806848

[58] Martin L. Puterman. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming* (1st ed.). John Wiley & Sons, Inc., New York, NY, USA.

[59] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. 2014. A reconfigurable fabric for accelerating large-scale datacenter services. In *ISCA*.

[60] Wu Qiang. 2016. Making Facebook's software infrastructure more energy efficient with Autoscale. (December 2016). goo.gl/vJi1kf

[61] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. 2010. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro* 30, 4 (7 2010), 65–79. DOI:http://dx.doi.org/10.1109/MM.2010.68

[62] John Russell. 2017. ARM Waving: Attention, Deployments, and Development. (February 2017).

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

[64] Gerald Tesauro. 2005. Online Resource Allocation Using Decompositional Reinforcement Learning.. In *Proceedings, The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005, Pittsburgh, Pennsylvania, USA*. 886–891.

[65] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani. 2006. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *Proceedings of the 2006 IEEE International Conference on Autonomic Computing (ICAC '06)*. IEEE Computer Society, Washington, DC, USA, 65–73. DOI:http://dx.doi.org/10.1109/ICAC.2006.1662383

[66] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and T N Vijaykumar. 2015. TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search. In *MICRO-48*.

[67] Violaine Villebonnet, Georges Da Costa, Laurent Lefèvre, Jean-Marc Pierson, and Patricia Stolf. 2016. Energy Aware Dynamic Provisioning for Heterogeneous Data Centers. In *2016 28th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. 206–213. DOI:http://dx.doi.org/10.1109/SBAC-PAD.2016.34

[68] Daniel Wong and Murali Annavaram. 2012. KnightShift: Scaling the Energy Proportionality Wall through Server-Level Heterogeneity. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 119–130. DOI:http://dx.doi.org/10.1109/MICRO.2012.20

[69] Wonyoung Wonyoung Kim, Meeta S. Gupta, Gu-Yeon Wei, and David Brooks. 2008. System level analysis of fast, per-core DVFS using on-chip switching regulators. In *2008 IEEE 14th International Symposium on High Performance Computer Architecture*. IEEE, 123–134. DOI:http://dx.doi.org/10.1109/HPCA.2008.4658633

[70] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. 2016. Dynamo: Facebook's Data Center-Wide Power Management System. *ISCA* (2016).

[71] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013a. Bubble-flux. *ACM SIGARCH Computer Architecture News* (2013).

[72] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013b. Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers. In *ISCA*.

[73] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI 2. In *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*. ACM Press, New York, New York, USA, 379. DOI:http://dx.doi.org/10.1145/2465351.2465388

Received February 2017; revised August 2017; accepted September 2017