2017 IEEE International Symposium on High Performance Computer Architecture

Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads

Rajiv Nishtala*†, Paul Carpenter*, Vinicius Petrucci‡, Xavier Martorell*†

* Barcelona Supercomputing Center, Barcelona, Spain
{*rajiv.nishtala, paul.carpenter, xavier.martorell*}*@bsc.es*

† Universitat Politècnica de Catalunya, Barcelona, Spain

‡ Federal University of Bahia, Salvador, Brazil
{*vpetrucci*}*@ufba.br*

***Abstract*—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 workloads are interactive, and they demand strict levels of quality-of-service (QoS) to meet user expectations, making it challenging to reduce power consumption due to increasing performance demands.**

This paper introduces Hipster, a technique that combines heuristics and reinforcement learning to manage latency-critical workloads. Hipster's goal is to improve resource efficiency in data centers while respecting the QoS of the latency-critical workloads. Hipster achieves its goal by exploring heterogeneous multi-cores and dynamic voltage and frequency scaling (DVFS). 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%.

I. Introduction

In 2013, U.S. data centers consumed 91 billion kilowatt-hours, which corresponds to 2.2% of the country's total electricity consumption and about 100 million metric tons of carbon pollution per year [1]–[3]. Energy efficiency is, in fact, a major issue across the whole computing spectrum, and modern systems have been exploring alternative heterogeneous processors [4]–[10] and Dynamic Voltage and Frequency Scaling (DVFS) [11]–[14] 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 [15]–[17]. A web-search, for example, must complete within a fraction of a second [18], otherwise users are likely to give up and leave. A previous study [19] 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 [20].

Recent works [15]–[17], [21]–[23] have shown that traditional power management practices and CPU utilization measures are unsuitable to drive task management for data center 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 [21], [24], workload management can be 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 operational cost and 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 minimizing system power consumption. In addition, Hipster allows collocation of latency-critical and batch workloads in shared data centers 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**, which is 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 cores (big and small) and their DVFS setting.

(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

2378-203X/17 $31.00 © 2017 IEEE
DOI 10.1109/HPCA.2017.13

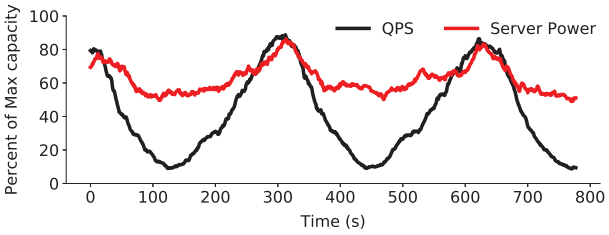


Fig. 1: Power drawn for a diurnal load [15], [22], [25] for Web-Search running on two big cores of the ARM Juno R1 (64-bit big.LITTLE) platform.

latency-critical and batch workloads for improved server utilization. Both variants ensure that QoS for latency-critical workloads is met.

(3) We carried out real measurement experiments using a 64-bit big-LITTLE (ARM Juno R1) platform along with back-end services such as Web-Search and Memcached. The request generator for each of the back-end services 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 [21] 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 latency-critical workloads.

II. Motivation

Typical web applications experience large variations in user traffic over time. Figure 1, for example, shows how the number of Web-Search queries per second, a typical load at Google data center, varies between about 5% and 80% of maximum capacity [15], [22], [25]. Similarly, Facebook consistently sees diurnal load variations between 10% and 95% of maximum capacity, across multiple server clusters [26], [27].

The periods of low server utilization provide opportunity to reduce data center energy consumption [16], [21], [24]. As seen in Figure 1, 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. 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 [5], [21], [28].

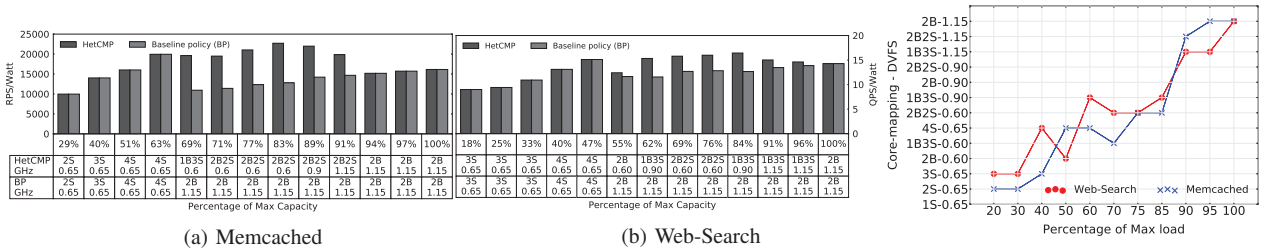
[Mixing different core types with DVFS]: Figure 2 shows the energy efficiency in RPS/Watt (Requests Per Second per Watt) and QPS/Watt (Queries Per Second per Watt) when using a state-of-the-art baseline policy [21]. We explore a heterogeneous architecture mixing different cores types and DVFS (HetCMP) running Memcached (Figure 2a) and Web-Search (Figure 2b) at different load levels. 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. For each policy, among the configurations where the QoS is met at each load level, the configuration with the least power consumption is selected. Experimental details in Section IV.

Figure 2 raises two main concerns with current state-of-the-art heuristic algorithm [21]. First, Figure 2a demonstrates 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. Similarly results were observed for Web-Search (Figure 2b) with up to 25% (mean) energy savings.

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 [29], harnessing HetCMP provides the opportunity for higher performance at a lower cost.

[Exploring workload particularities]: We note that prior work [21] relies on a 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 IV); 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



(a) Memcached (b) Web-Search (c) State Machine

Fig. 2: Throughput per watt of Memcached (2a) and Web-Search (2b) with baseline policy (BP) [21] and heterogeneous platforms with DVFS (HetCMP) at different load levels along with their respective state machines (2c)

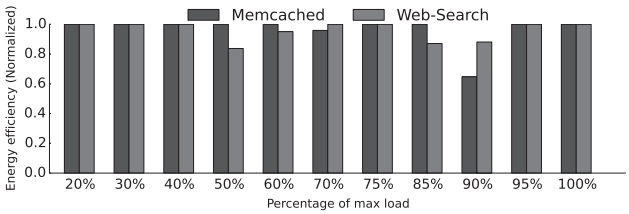


Fig. 3: Energy efficiency at various load levels for Memcached while meeting QoS, using the state-machine of Web-Search normalized to the state-machine of Memcached (*lower is worse*); converse for Web-Search.

efficiency using the state-machine built exclusively for Memcached; and vice-versa. The state-machine configuration for Memcached and Web-Search are represented in blue and red line, respectively, in Figure 2c. 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 [30] 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 [20] or gradual load changes [22], [25]. 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, Kasture et al [17] note that core-transitions are far more costly – relative to DVFS changes.

There is a need for application agnostic learning approach that can 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.

III. Hipster

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 workload.

A. Hipster Reinforcement Learning

The RL problem solved by Hipster is formulated as a Markov Decision Process (MDP). 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 workload, 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, which is 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 III-D.

RL is a type of unsupervised machine learning with a focus on online learning [31]. 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 largest 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.

Hipster uses a hybrid RL approach [32], which combines reinforcement learning with a heuristic, to be used while the algorithm is still learning the optimal behavior. 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 [32] has the potential to outperform pure RL schemes [33], [34] 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 an unacceptable number of QoS violations.

(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 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.

B. Hipster Design

Figure 4 shows a high-level view of Hipster. Hipster includes a QoS Monitor, a Learning Phase and an 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.

[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 quantizes 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 instructions per second (IPS).

[1]The update of $R(w_n, c_n)$ is on line 16 of Algorithm 1.

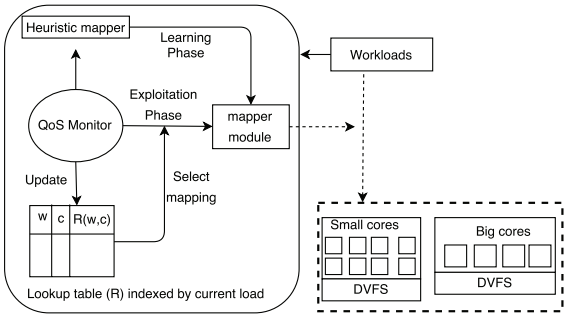


Fig. 4: High-level view of Hipster runtime system

[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 II, 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 III-D) 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 [35].

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 [30] with minimal energy consumption.

C. Heuristic Mapper (Learning Phase)

The heuristic mapper is a state machine with a feedback control loop. 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

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

Algorithm 1 Reward mechanism

▷ *Determine reward λ_n based on interval $t_n \dots t_{n+1}$*
1 Let QoS_{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 $\quad\lambda_n = QoS_{\text{reward}} + 1$
8 **else if** $QoS_{\text{curr}} < QoS_{\text{target}}$ **then**
9 $\quad\lambda_n = QoS_{\text{reward}} + 1 - Random(0,1)$
10 **else**
11 $\quad\lambda_n = -QoS_{\text{reward}} - 1$
12 **if** there exist batch jobs **then**
13 $\quad\lambda_n = \lambda_n + \frac{B_{\text{IPS}}+S_{\text{IPS}}}{maxIPS(B)+maxIPS(S)}$
14 **else**
15 $\quad\lambda_n = \lambda_n + Power_{\text{reward}}$
16 $R(w_n, c_n) = R(w_n, c_n) + \alpha\Big(\lambda_n + \gamma \max_{d \in C} R(w_{n+1}, d) - R(w_n, c_n)\Big)$

performance of each state using a stress microbenchmark consisting of 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: $QoS_{\text{curr}} > QoS_{\text{target}} \times QoS_{\text{D}}$, 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: $QoS_{\text{curr}} < QoS_{\text{target}} \times QoS_{\text{S}}$, 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. In particular, QoS_{D} is empirically computed as in prior work [21], [35].

The heuristic proposed by Octopus-Man is attractive because of its simplicity but it can be sub-optimal (see Section II, 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 suitable to accelerate the learning phase of the RL algorithm by exploring viable core configurations to quickly populate reasonable values into the lookup table.

D. Reward Calculation

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

[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 10 applies a negative QoS reward.

[Stochastic Reward]: When the QoS is below the target, as defined in Section III-C, 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.

[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 15. A smaller value of this term means that the system power consumption was lower, and it increases the reward.

[Throughput Reward (HipsterCo)]: Lines 12 to 13 of Algorithm 1 calculate the Throughput Reward, which is approximately proportional to 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 IV-A.

Once the reward λ_n has been calculated, line 16 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 controlled using two scalar parameters, both between zero and one: the discounting factor, γ, and the learning rate, α.

[Discounting Factor, γ]: The γ coefficient in line 6 of Algorithm 1 is the discounting factor, which quantifies the preference for short-term rewards [36]. 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.

[Learning Factor, α]: The α coefficient in line 16 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

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

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 $\alpha = 0.6$

E. Exploitation Phase

The exploitation phase of Hipster is defined by Algorithm 2. Line 7 determines the configuration, c_n, with the highest estimated total discounted reward. Lines 8 to 13 apply the configuration by mapping the workloads to the resources, as described below, depending on the specific variant of Hipster (HipsterIn or HipsterCo). Line 14 runs the workload for the next time interval, and line 16 calls Algorithm 1 to update the lookup table, based on the metrics obtained by the QoS Monitor during the time interval. Line 18 re-enters the learning phase when necessary. The mapping of workloads to resources is as follows:

[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 12 to 13 of Algorithm 2).

[Reward Mechanism for HipsterCo]: Corroborating the findings of prior work [16], we observed that collocating both latency-critical and batch workloads degrades QoS at higher loads due to shared resource contention. If the reward mechanism were 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. As a precursor to this condition, 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 8 to 9 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 10 to 13 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.

F. Responsiveness and Stability

To ensure that QoS is met for latency-critical workloads, the scheduling policy must quickly respond to fluctuations in load and latency, either due to changes in core mapping, DVFS or any external influence. Therefore, the responsiveness and stability of 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.

The computational latency for changes in core mapping and DVFS are negligible [9], [37], [38]. The default monitoring interval for Memcached and Web-Search is one second. Based on the aforementioned overheads, 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.

G. Hipster Implementation

Hipster is implemented in user space, and it uses minimal hardware support exposed by Linux. It consists of the 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 [39], specifically using the *perf_event* instructions [40], [41]. Alternatives to perf include the profiling tools [42] supported by Docker and Kubernetes [43].

[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).

[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. Thus, in the prototype implementation of Hipster, the lookup table was implemented using a Python dictionary, using open addressing to resolve hash collisions, thereby having a computational complexity of $\mathcal{O}(1)$ irrespective of the operation [44].

[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 $<2\,\mathrm{ms}$, so triggering Hipster every second, as in our experiments, incurs an overhead $<0.2\%$.

IV. Evaluation

A. Experimental Methodology

[Benchmarks]: We evaluate Hipster using two latency-critical applications, **Memcached** and **Web-Search**, which have distinct characteristics and impact on shared resources [16]. **Memcached** [45] is an open source implementation of an in-memory key-value store for data caching used in many services from Twitter, Facebook and Google [27], [46]. The backend of **Web-Search** [28], [47] is an instance of Elasticsearch [48], 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 [45]. It is configured to model diurnal load changes (Figure 1), simulating a period of 36 hours [22]; each hour in the original workload corresponds to one minute in our experiments. For the batch workloads [49], [50], we use the programs from the SPEC CPU 2006 suite [51].

[Tail latency]: For Memcached, we define the tail latency to be the 95th percentile request latency, with a target of 10 ms [15], [45]; for Web-Search, we define it to be the 90th percentile query latency, with a target of 500 ms [45]. Table I 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.

[Hardware resources]: We perform the evaluation experiments on an ARM Juno R1 developer board [52] 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. 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 [53] with eight cores at 2.4 GHz and 128 GB DRAM.

[Power efficiency]: The power consumption of the Juno platform is obtained by reading specific hardware registers [54]. These registers report separately the power consumed by the big cluster, small cluster, and the rest of the system (Juno's *sys* register [55]). 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.

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 [45]	500 (90%ile)

TABLE I: 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 II: Power and performance characterization on Juno platform.

Table II 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 III-C) 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.

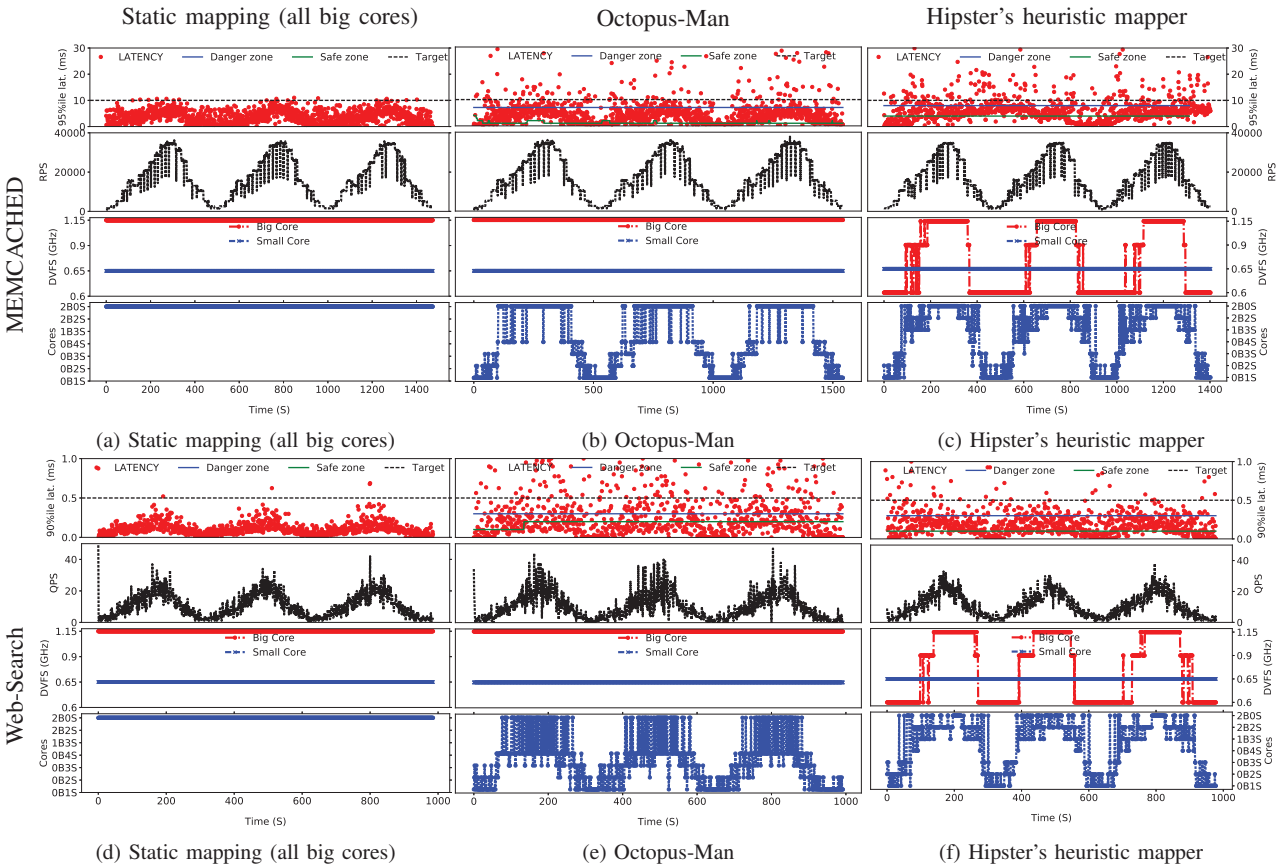
Considering system power, a single big core is 52% more power-efficient than a single small core, in terms of IPS per watt (Table II). However, taking into account all cores in a cluster, and assuming that all cores can be fully utilized, a small cluster is 25% more power-efficient than a big cluster. 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.

[Algorithm configuration]: In deploying Octopus-Man, we first performed a sweep on the danger and safe thresholds, and picked the combination of thresholds with the highest QoS guarantee. For HipsterIn, we set the learning phase to be 500 seconds, except when quantifying the learning time, where we set it to 200 seconds.

B. 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.

*1) **Hipster's Heuristic Policy (interactive only)**:* We first evaluate the effectiveness of Hipster's heuristic policy alone, for mapping interactive workloads. Figure 5 shows the results for both workloads: Memcached (top row, subfigures (a), (b) and (c)) and Web-search (bottom row, subfigures (d), (e) and



(a) Static mapping (all big cores) (b) Octopus-Man (c) Hipster's heuristic mapper

(d) Static mapping (all big cores) (e) Octopus-Man (f) Hipster's heuristic mapper

Fig. 5: Comparison of Hipster's heuristic policy (right-hand column) with static mapping (left) and Octopus-Man (centre). Results are shown for Memcached (top) and Web-Search (bottom) on ARM Juno R1.

(f)). The columns, from left to right, correspond to static mapping, for which the interactive threads are mapped to the two big cores at highest DVFS of 1.15 GHz ((a) and (d)), Octopus-Man ((b) and (e)) and Hipster's heuristic policy ((c) 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.

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 600^{th} and 800^{th} seconds. 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.

*2) **HipsterIn: Memcached Results**:* Figure 6 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) [17].

*3) **HipsterIn: Web-Search Results**:* Figure 7 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

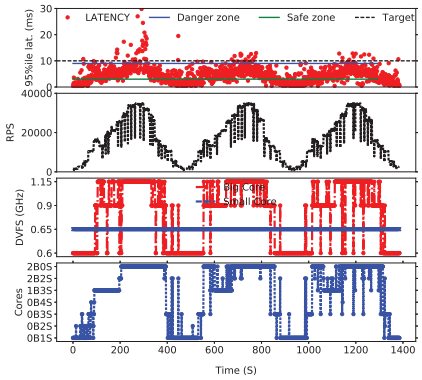


Fig. 6: HipsterIn on Memcached

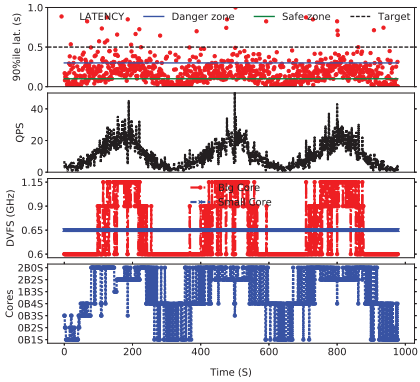


Fig. 7: HipsterIn on Web-Search

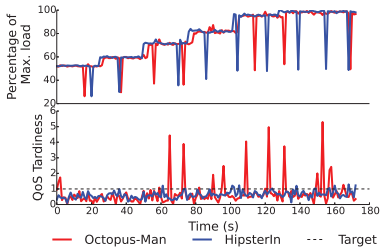


Fig. 8: Percentage of Max. load, and Tail latency (QoS Tardiness) running Memcached with HipsterIn and Octopus-Man.

	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 III: HipsterIn: summary of QoS guarantees, tardiness and energy savings for Memcached and Web-Search.

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) **HipsterIn Summary**:* Table III 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 III, 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 III-D.

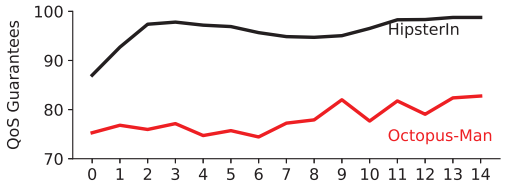


Fig. 9: QoS Guarantees of HipsterIn and Octopus-Man. Each data point represents the QoS guarantees over 100 s intervals.

*5) **HipsterIn Analysis**:*

[Rapid adaptation to load changes]: Hipster can respond to rapid changes in load by directly mapping to a configuration that satisfies the QoS. Figure 8 shows how HipsterIn (during the exploitation phase) and Octopus-Man respond to changes in load. From top to bottom, we express the input load in terms of the percentage of maximum load, where it increases from 50% to 100% over a period of 175 seconds for Memcached. In the second graph, we express the 95^{th} percentile tail latency as QoS Tardiness. A QoS violation has occurred if the QoS Tardiness is above 1, otherwise QoS is satisfied. We find that Octopus-Man violates QoS due to aggressive core mappings 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 III). In practice, to best optimize for energy efficiency, and to improve QoS,

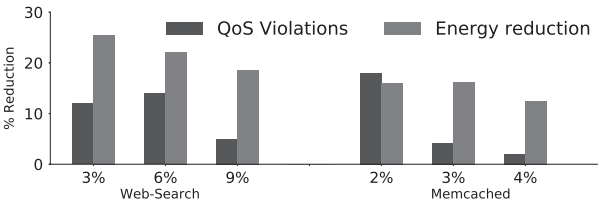


Fig. 10: Impact of bucket size on HipsterIn QoS guarantees, and energy savings, normalized to static (all big cores) on Web-Search and Memcached.

HipsterIn needs a short learning phase. Figure 9 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 10 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 bucket. Therefore, in tuning Hipster, we empirically determine the bucket size to maximize energy savings subject to at least 98% QoS guarantee.

C. 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 11 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 11, 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 IV-B3), so it achieves superior QoS guarantees.

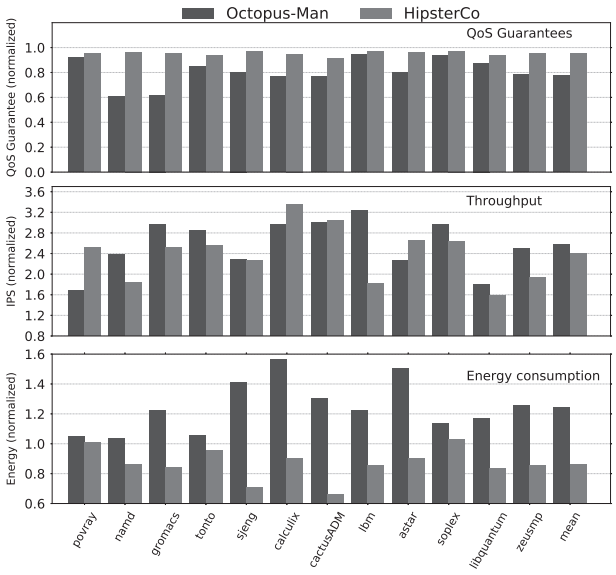


Fig. 11: QoS guarantee (top), Throughput improvement (middle) and Energy consumption (bottom) when Web-Search is collocated with batch workloads. The results are normalized to static all big cores.

As shown in the middle plot of Figure 11, for all benchmarks, Hipster and Octopus-Man deliver much higher throughput compared to static mapping, with an average of $2.3\times$ and $2.6\times$ 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 $2.6\times$ more powerful than small cores). For *Calculix*, a compute-bound application, HipsterCo achieves the highest throughput improvement over static of $3.35\times$, and for *libquantum*, a memory-bound program, the least improvement is still $1.6\times$.

As shown in the bottom plot of Figure 11, 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 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.

V. Related Work

Bubble-flux and Bubble-Up [56], [57] 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 [58] identifies and manages performance interference between VM systems collocated on the same system. Q-Clouds [59] develop a feedback based mechanism to tune resource assignment to avoid negative interference to collocated VMs. CPI2 [60] enables race-to-finish for low-priority workloads to not have a deadlock with high priority services.

Octopus-Man [21] 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 [16] 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 [15] achieves high CPU energy proportionality for low latency workloads using fine-grained DVFS techniques. TimeTrader [29] and Rubik [17] exploit request queuing latency variation and apply any available slack from queuing delay to throughput-oriented workloads to improve energy efficiency. Quasar [61] use runtime classification to predict interference and collocate workloads to minimize interference.

KnightShift [8] introduces a server architecture that couples commercial available compute nodes to adapt the changes in system load and improve energy proportionality. Autoscale [62] 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 [32] 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 II).

VI. 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.

Acknowledgment — This work has been partially supported by the European Union FP7 program through the Mont-Blanc-2 (FP7-ICT-610402) and EUROSERVER (FP7-ICT-610456) project, 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).

References

[1] P. Delforge and J. Whitney, "Data Center Efficiency Assessment," *Natural Resources Defense Council (NRDC)*, 2014.

[2] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song, "Dynamo: Facebook's Data Center-Wide Power Management System," *ISCA*, 2016.

[3] "Facebook is opening a new wind-powered data center in Texas, http://goo.gl/dKVnSB."

[4] J. Mars, L. Tang, and R. Hundt, "Heterogeneity in 'Homogeneous' Warehouse-Scale Computers: A Performance Opportunity," *IEEE Computer Architecture Letters*, vol. 10, no. 2, pp. 29–32, 2 2011.

[5] N. Chitlur, G. Srinivasa, S. Hahn, P. K. Gupta, D. Reddy, D. Koufaty, P. Brett, A. Prabhakaran, L. Zhao, N. Ijih, S. Subhaschandra, S. Grover, X. Jiang, and R. Iyer, "QuickIA: Exploring heterogeneous architectures on real prototypes," in *IEEE International Symposium on High-Performance Comp Architecture*. IEEE, 2 2012, pp. 1–8.

[6] M. Halpern, Y. Zhu, and V. J. Reddi, "Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction," in *HPCA*, 2016.

[7] M. Guevara, B. Lubin, and B. C. Lee, "Navigating heterogeneous processors with market mechanisms," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2 2013, pp. 95–106.

[8] D. Wong and M. Annavaram, "KnightShift: Scaling the Energy Proportionality Wall through Server-Level Heterogeneity," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 12 2012, pp. 119–130.

[9] J. Cong and B. Yuan, "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*. New York, New York, USA: ACM Press, 7 2012, p. 345.

[10] R. Nishtala, D. Mosse, and V. Petrucci, "Energy-aware thread co-location in heterogeneous multicore processors," in *EMSOFT*, 2013.

[11] W. Wonyoung Kim, M. S. Gupta, G.-Y. Wei, and D. Brooks, "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, 2 2008, pp. 123–134.

[12] W. Godycki, C. Torng, I. Bukreyev, A. Apsel, and C. Batten, "Enabling Realistic Fine-Grain Voltage Scaling with Reconfigurable Power Distribution Networks," in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 12 2014, pp. 381–393.

[13] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P. Y. Xiao, and D. Burger, "A reconfigurable fabric for accelerating large-scale datacenter services," in *ISCA*, 2014.

[14] D. Lo and C. Kozyrakis, "Dynamic management of TurboMode in modern multi-core chips," in *HPCA*, 2014.

[15] 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, no. 3, pp. 301–312, 10 2014.

[16] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles," *ACM SIGARCH Computer Architecture News*, 2015.

[17] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast Analytical Power Management for Latency-Critical Systems," in *MICRO-48*, 2015.

[18] L. A. Barroso, J. Clidaras, and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second edition," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 3, pp. 1–154, 7 2013.

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

[20] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, p. 74, 2 2013.

[21] V. Petrucci, M. A. Laurenzano, J. Doherty, Y. Zhang, D. Mosse, J. Mars, and L. Tang, "Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers," in *HPCA*, 2015.

[22] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Proceeding of the 38th annual international symposium on Computer architecture - ISCA '11*, vol. 39, no. 3. New York, New York, USA: ACM Press, 6 2011, p. 319.

[23] Y. Li, D. Wang, S. Ghose, J. Liu, S. Govindan, S. James, E. Peterson, J. Siegler, R. Ausavarungnirun, and O. Mutlu, "SizeCap: Efficiently handling power surges in fuel cell powered data centers," in *HPCA*, 2016.

[24] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware scheduling for heterogeneous datacenters," *ACM SIGARCH Computer Architecture News*, 2013.

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

[26] O. Bilgir, M. Martonosi, and Q. Wu, "Exploring the Potential of CMP Core Count Management on Data Center Energy Savings," in *3rd Workshop on Energy Efficient Design (WEED)*, 2011.

[27] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," in *SIGMETRICS*, 2012.

[28] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid, "Web search using mobile cores," in *ISCA*, 2010.

[29] B. Vamanan, H. B. Sohail, J. Hasan, and T. N. Vijaykumar, "TimeTrader: Exploiting Latency Tail to Save Datacenter Energy for Online Search," in *MICRO-48*, 2015.

[30] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the Tail," in *SOCC*, 2014.

[31] 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, no. 7540, pp. 529–533, 2 2015.

[32] G. Tesauro, N. Jong, R. Das, and M. Bennani, "A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation," in *2006 IEEE International Conference on Autonomic Computing*. IEEE, pp. 65–73.

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

[34] G. Tesauro, "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*, 1 2005, pp. 886–891.

[35] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu, "Dynamic Voltage Scaling in Multitier Web Servers with End-to-End Delay Control," *IEEE Transactions on Computers*, vol. 56, no. 4, pp. 444–458, 4 2007.

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

[37] J. Leverich, M. Monchiero, V. Talwar, P. Ranganathan, and C. Kozyrakis, "Power Management of Datacenter Workloads Using Per-Core Power Gating," *IEEE Computer Architecture Letters*, vol. 8, no. 2, pp. 48–51, 2 2009.

[38] N. Madan, A. Buyuktosunoglu, P. Bose, and M. Annavaram, "A case for guarded power gating for multi-core processors," in *HPCA*, 2011.

[39] "Perf: Linux profiling with performance counters,https://perf.wiki.kernel.org/."

[40] ARM Limited, "ARM ® Cortex ® -A53 MPCore Processor Technical Reference Manual."

[41] ——, "ARM ® Cortex ® -A57 MPCore Processor Revision: r1p0 Technical Reference Manual."

[42] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers," *IEEE Micro*, vol. 30, no. 4, pp. 65–79, 7 2010.

[43] D. Bernstein, "Containers and Cloud: From LXC to Docker to Kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 9 2014.

[44] R. Pattis, "Complexity of Python Operations, https://www.ics.uci.edu/˜pattis/ICS-33/lectures/complexitypython.txt."

[45] M. Ferdman, B. Falsafi, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, and A. Ailamaki, "Clearing the clouds," *ACM SIGARCH Computer Architecture News*, 2012.

[46] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *USENIX conference on Networked Systems Design and Implementation*, 2013.

[47] L. Barroso, J. Dean, and U. Holzle, "Web search for a planet: the google cluster architecture," in *IEEE Micro*, 2003.

[48] "Elasticsearch,https://github.com/elastic/elasticsearch."

[49] J. Mars and L. Tang, "Whare-map," *ACM SIGARCH Computer Architecture News*, 2013.

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

[51] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 9 2006.

[52] "ARM Juno R1,https://goo.gl/EcamOa."

[53] "Applied Micro XGene2,http://goo.gl/XA04r1."

[54] "ARM Juno Power Registers,https://github.com/ARM-software/devlib/blob/master/src/readenergy/readenergy.c."

[55] ARM, "SYS_POW_SYS Register, https://goo.gl/fmTTQi."

[56] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers," in *ISCA*, 2013.

[57] 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 *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11*. New York, New York, USA: ACM Press, 2011, p. 248.

[58] D. Novaković, N. Vasić, S. Novaković, D. Kostić, and R. Bianchini, "DeepDive: transparently identifying and managing performance interference in virtualized environments," in *USENIX conference on Annual Technical Conference*, 2013.

[59] R. Nathuji, A. Kansal, and A. Ghaffarkhah, "Q-clouds," in *Proceedings of the 5th European conference on Computer systems - EuroSys '10*. New York, New York, USA: ACM Press, 4 2010, p. 237.

[60] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI 2," in *Proceedings of the 8th ACM European Conference on Computer Systems - EuroSys '13*. New York, New York, USA: ACM Press, 4 2013, p. 379.

[61] C. Delimitrou and C. Kozyrakis, "Quasar: resource-efficient and QoS-aware cluster management," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 1, pp. 127–127, 4 2014.

[62] Qiang Wu, "Making Facebook's software infrastructure more energy efficient with Autoscale, goo.gl/vJi1kf."