Dynamic Colocation Policies with Reinforcement Learning

YUHAO LI, Duke University
DAN SUN, Duke University
BENJAMIN C. LEE, Duke University

We draw on reinforcement learning frameworks to design and implement an adaptive controller for managing resource contention. During run-time, the controller observes the dynamic system conditions and optimizes control policies that satisfy latency targets yet improve server utilization. We evaluate a physical prototype that guarantees 95th percentile latencies for a search engine and improves server utilization by 50% for varied batch workloads in machine learning.

CCS Concepts: • **Information systems applications** → **data centers**; • **Computer systems organization** → *Self-organizing autonomic computing*; Cloud computing; • **Reinforcement learning** → Sequential decision making.

Additional Key Words and Phrases: resource contention, adaptive control, machine learning

1 INTRODUCTION

Most datacenter servers are under-utilized. Average server utilization is 30-50% in warehouse-scale datacenters and much less in typical datacenters [7, 10]. We can explain low utilization, in part, by the prevalence of interactive workloads. These workloads often compute alone, avoiding shared resource contention and meeting performance targets, but cannot fully utilize the server.

In this paper, we study strategies for increasing server utilization by colocating interactive services with batched machine learning workloads. Machine learning has emerged as, not only a driving force that enhances the user experience, but also an important consumer of server resources in datacenters [15]. Machine learning workloads comprise a new breed of batch workloads with three unique characteristics and challenges.

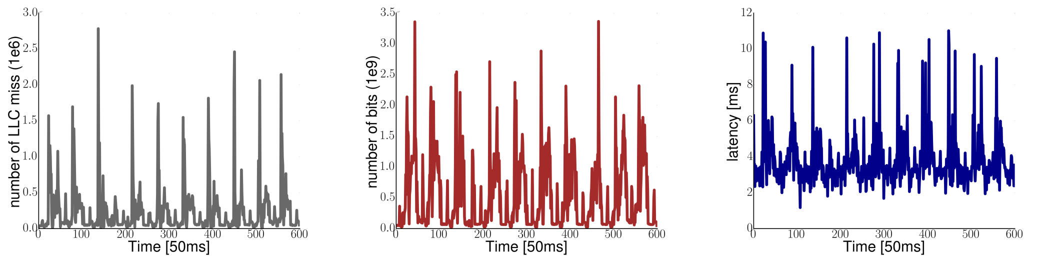
First, machine learning algorithms are iterative with diverse, recurring computational phases. Colocation managers should monitor the batch workload to determine when and which resources should be allocated. This approach departs from prior work that focuses on monitoring and managing the interactive workload [20, 24, 25, 31, 45]. Indeed, we find that neglecting the batch dynamics leads to conservative colocation policies and poor server utilization.

Second, machine learning requires fine-grained, responsive colocation management. Computational phases span seconds and, when the workload transitions from phase to phase, the colocation manager should respond quickly. These short time scales differ from prior studies which manage pairwise colocations that persist until workloads complete [13, 23].

Finally, machine learning applications compute on diverse datasets, which determine memory intensity and shared resource contention. This diversity precludes offline profiling, popular in prior

Authors' addresses: Yuhao Li, yuhao.li@duke.edu, Duke University, 213 Hudson Hall, Durham, North Carolina, 27708; Dan Sun, Duke University, 213 Hudson Hall, Durham, North Carolina, dan.sun@duke.edu; Benjamin C. Lee, Duke University, 210 Hudson Hall, Box 90291, Durham, North Carolina, 27708, benjamin.c.lee@duke.edu.

XXXX-XXXX/2019/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn



(a) Last-level cache misses for linear regression (b) Memory transfers for linear regression (c) 95th-percentile latency for search latency

Fig. 1. Colocation dynamics as varying memory activity for batch workload affects tail latency for the interactive workload. Search engine and linear regression are colocated. Data collected every 50ms.

studies [13, 23, 28], because profiling every machine learning library for every dataset is intractable. Colocation managers should analyze dynamics within the batch workload online, updating models and policies as the server runs.

We address these challenges with reinforcement learning, which analyzes data to make a sequence of optimized decisions [37]. We use reinforcement learning to build a lightweight, online controller. The controller modulates batch computation to balance service targets and server utilization. It uses Q-learning to optimize its control policy as the server computes. The optimized policy assesses system conditions and dynamically scales the voltage and frequency for cores running machine learning workloads.

We show that reinforcement learning can respond quickly to severe, short-term variability in machine learning workloads that jeopardize interactive service targets. The colocation controller learns policies that balance short- and long-term objectives, meeting interactive performance targets and increasing server utilization. Moreover, the controller requires no offline profiling and learns to adapt its control policies as resource demands from batch workloads evolve. In summary, we make the following contributions:

- We characterize machine learning codes and identify colocation challenges that arise from their diverse computational phases and variable resource demands.
- We use reinforcement learning to determine, during run-time, optimal policies for setting the batch workload's processor core frequencies.
- We implement a controller that balances service quality and server utilization. It learns policies online and adapts quickly as workload demands evolve.
- We prototype the controller and find that it enforces a target on 95th-percentile latency, improves server utilization by up to 70%, and reduces datacenter energy consumption by 50% through colocation.

2 UNDERSTANDING BATCH DYNAMICS

We study datacenter servers that colocate interactive and batch machine learning workloads. As these workloads share server resources, software mechanisms, such as Linux cgroup [1] and Docker containers [29], manage the allocation of cores, memory capacity and network I/Os to each process. However, these software mechanisms are unable to regulate the competition for hardware-managed resources such as last-level cache capacity and main memory bandwidth. In this paper, we modulate

Algorithm	**Type**	**Data Operation**
Linear Regression	Regression	data sampling, gradient calculation, gradient aggregation
Support Vector Machine	Classification	data sampling, gradient calculation, gradient aggregation
Decision Tree	Classification/Regression	tree split (sort, reduction)
Gradient Boost Tree	Classification	sampling, tree split(sort, reduction)
Naive Bayesian	Classification	filtering, extracting features, data aggregation
Correlation	Statistics	map, data aggregation

Table 1. Basic data operators in machine learning workloads implemented with spark mllib

batch computation to manage contention in the shared memory subsystem and avoid interference with the interactive service.

Variability in Batch Workloads. Maintaining both service quality and server utilization is difficult because of workload dynamics. Researchers have characterized interactive workloads and their variations in service time, queue length, and query loads [20]. Such variability lengthens the tail distribution for query latency. However, there has been no corresponding study of workload dynamics for batch workloads. In this paper, we find that distributed machine learning workloads exhibit diverse, recurring phases that cause contention in the last-level cache and memory channel to vary across time.

We survey machine learning algorithms used in production machine learning pipelines, as shown in Table 1. It includes two regression algorithms, three classification algorithms, and one statistical computation. We find these algorithms are built upon basic data operations, which repeatedly operate on the input data and thus produce rich variations in machine analytics' resource demand.

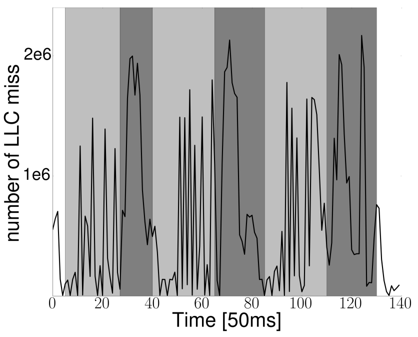
Figure 1 illustrates variations in batch machine learning and its effect on interactive service quality. We colocate Spark linear regression, a representative machine learning workload, with a search engine. Regression's cache misses and memory transfers exhibit phase behavior with periodic spikes in activity. Variations in the regression workload produce variations in search query latency. The search engine fails to meet its target 95th-percentile latency when memory activity spikes and the engine satisfies its target otherwise.

For this example, previously proposed colocation mechanisms are reactive and ineffective. They rely only on the interactive workload's profile to throttle batch computation when interactive service quality is poor [20, 24, 25, 45]. Because these mechanisms lack knowledge of the batch workload's phases and periodic demands for memory, they would conservatively halt batch computation when interactive latency spikes. Server utilization would suffer.

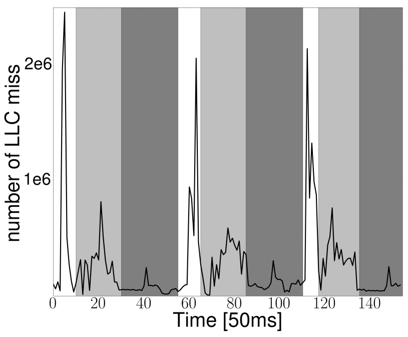
Phase Behaviors in Machine Learning. We can understand oscillations between high and low memory activity by examining the structure of machine learning algorithms. Many algorithms rely on iterative training, which repeatedly updates model parameters until predictive accuracy for the input data reaches some target.

For example, Figure 2b presents phases that arise from iterative training in `regression`. In each iteration, `regression` performs three stages of computation. Data sampling suffers the highest cache miss rates because it accesses data and memory randomly. Gradient calculation also suffers high cache miss rates because it must multiply large matrices. Gradient aggregation is relatively compute-intensive and generates little contentious behavior in the memory system.

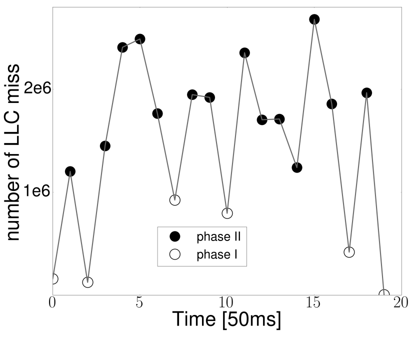
In another example, Figure 2b presents phases from the iterative construction of a `gradient boosting tree`. Data sampling determines threshold candidates for splitting the dataset. Next, the algorithm recursively splits the data and constructs the decision tree. As the recursion progresses and the algorithm approaches the tree's leaf nodes, the amount of data split decreases and the cache miss rate falls.



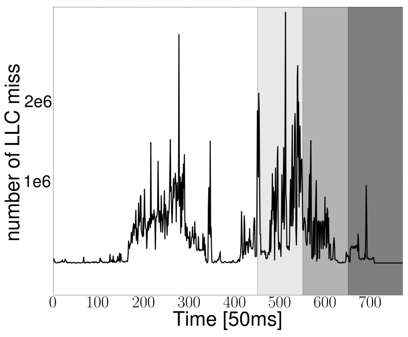
(a) **Gradient Boosting Tree.** Phases: (I) data preprocessing; (II) data sampling to determine split candidates; (III) recursively choose best split for each sub partition.



(b) **Linear Regression.** Phases: (I) data sampling; (II) gradient calculation; (III) gradient aggregation.



(c) **Correlation.** Phases: (I) calculating coefficients (map operation); (II) aggregating feature values.



(d) **Naive Bayesian.** Phases: (I) data preprocessing; (II) filtering; (III) extracting features; (IV) aggregating values.

Fig. 2. Computational phases for representative machine learning workloads. Data collected every 50 ms.

Some workloads must compute a large number of parameters. This computation cannot be parallelized within the server and the same operation must be repeated for different values in the dataset. Figure 2c visualizes dynamics for `Correlation`, which determines the Pearson correlation between each feature of the input and output variable. The workload aggregates feature values and then calculates the correlation coefficient. Although calculating Pearson for one feature is short and simple, the dataset may possess a large number of features (e.g., over 1 million in kdda'10 [4]) and computation may be serialized for subsets of the features.

For completeness, Figure 2d presents `Naive-Bayesian`, an application that does not operate on data repeatedly. `Naive-Bayesian` performs parallel tasks for each class of input. Because the dataset has two classes, the algorithm uses two threads to train in parallel.

Implications for Management. The rich variations in behavior across computational phases have several implications for colocation management. First, the manager should determine the batch workload's current phase, from hardware performance counters, and respond accordingly. Second, because only some phases and spikes in memory activity cause an interactive workload to

Data Size	Duration	Peak LLC Miss Count
1GB	0.52s	1.0e6
512MB	0.30s	1.8e6
256MB	0.17s	0.8e6
128MB	0.08s	0.3e6
64MB	<0.05s	0.2e6

Table 2. **Linear Regression.** Dataset size and its effect on experimental duration, last-level cache misses. Results for Kdda2010 dataset [4].

Dataset	Feature Count	Operation Duration	Peak LLC Miss Count
phishing	68	0.72s	1.60e6
madelon	500	2.60s	1.50e6
gisette	5,000	5.01s	1.60e6
rcv1	47,236	41.00s	1.65e6
kdda2010	1,163,024	>300.00s	1.70e6

Table 3. **Gradient Boosting Tree.** Number of dataset features and its effect on experimental duration, last-level cache misses. Results for [4].

violate its service targets, the framework should only throttle the batch workload during the most contentious phases.

Third, the manager should learn an effective throttling strategy online as the server computes. Online analysis is imperative because offline profiling is intractable. Machine learning workloads' behavior is highly sensitive to the dataset size and the number of features. An offline profile for a particular dataset cannot generalize and collect offline profiles for every representative dataset is impossible. For example, Tables 2– 3 show the impact of dataset size and the number of features on experimental duration and the number of last-level cache misses.

In this paper, we propose reinforcement learning for an online colocation controller that manages resource contention. This controller learns and adapts to phases in batch workloads. It also predicts and safeguards the service quality of interactive workloads. Our approach departs from related work, which focuses on the interactive workload's tail latency alone and neglects batch workload dynamics [14, 28].

3 CONTROLLER DESIGN

3.1 Control Mechanisms

We consider a colocation scenario in which interactive services are assigned to a set of cores and the rest are allocated to batch machine learning. We aim to control batch workloads at fine time granularities to respond to rapid changes in analytics phases and interactive services' query loads. For fine-grained control, we use per-core dynamic voltage and frequency scaling (DVFS) to control resource contention and enforce service quality[1]. DVFS can respond in microseconds for mission-critical workloads [12, 17, 20, 45]. In this project, we propose to use DVFS as a throttling mechanism to control the behavior of batch analytic applications.

3.2 Reinforcement Learning

Managing service quality can be viewed as a sequential decision-making problem, illustrated in Figure 3. At each step in the sequence, the controller monitors system conditions that may produce long tail latencies. Measurements may include software-level queuing statistics, such as distributions on service time and queue length, and hardware-level performance counters. These

[1]our method can be generalized to coarse-grained control mechanisms,such as core reallocation, with minimal effort

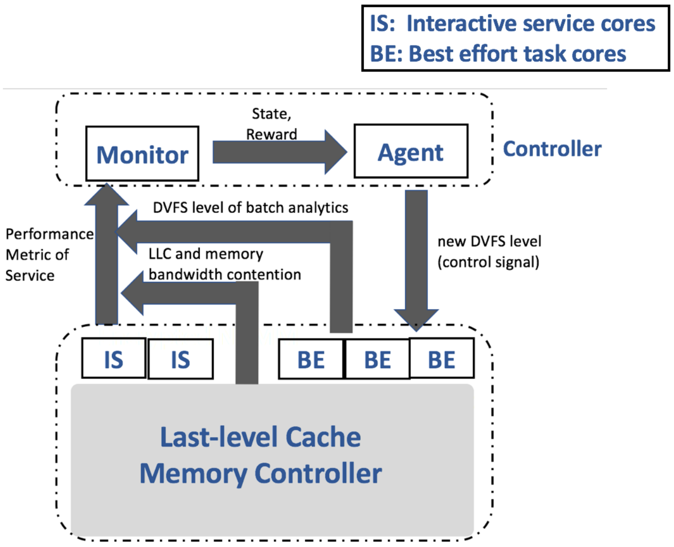

IS: Interactive service cores
BE: Best effort task cores
State, Reward
Monitor
Agent
Controller
DVFS level of batch analytics
Performance Metric of Service
LLC and memory bandwidth contention
new DVFS level (control signal)
IS
IS
BE
BE
BE
Last-level Cache
Memory Controller

Fig. 3. Enforce service quality as a decision-making process: At each step in the sequence, the controller monitors system conditions affecting tail latencies and the performance of interactive services. These information are translated into state description and reward signal which guide the controller to make decisions on best effort cores' new DVFS level.

measured features define a state, which maps to a decision—the voltage and frequency setting for the batch workload's processor cores. The decision changes the system state and tail latency.

The controller defines its reward signal based on tail latency. In each step of the sequence, its goal is maximizing the reward it will receive in subsequent steps. Pursuing this goal is particularly difficult when system conditions vary on short timescales. The controller models system variations and uncertainty with stochastic models such as Markov decision processes (MDP). Traditional approaches that use stochastic models would take two separate, offline steps—one to model the system and another to derive control policies.

In contrast, reinforcement learning is dynamic and online. Reinforcement learning constructs an optimal, decision-making policy using trial and error in a stochastic environment. It is a powerful framework for solving problems when environmental features that define the state space follow unknown probability distributions. We provide an overview of reinforcement learning and then detail its application to enforcing service quality.

Reinforcement learning places an agent in a stochastic environment. At every time step t, it perceives the state of the environment s_t and issues an action a_t that yields a reward r_t. The decision process is defined by a five-tuple where

- S is a finite set of states that describe the environment.
- A is a finite set of actions that an agent may perform. The action at time t affects the state at time $t + 1$.
- $\mathrm{P_a(s, s') = Pr(s_{t+1} = s' | s_t = s, a_t = a)}$ is the probability that action a taken at time t changes the state to s'. The framework of Reinforcement Learning assumes that $P_a(s, s')$ depends only on the current state s_t and the current action a_t, and is conditionally independent of previous states and actions.
- $\mathrm{R(s, a)}$ is the reward received after action a is taken at state s.

- $\gamma \in [0, 1]$ is the discount factor, which represents how much less the agent values reward at $t + 1$ compared to reward at time t.

A policy π determines the action an agent takes in a given state. The optimal policy maximizes the expected long-term cumulative reward $E\left(\Sigma_{t=0}^{\infty}\gamma^t R(s_t, a_t)\right)$.

This measure of long-term reward aligns with our system goals. We wish to improve server utilization while satisfying the 95th-percentile latency. Temporarily violating tail latency may reduce rewards now, but permit sustained rewards in the future as a short-term burst in the batch workload's resource intensity subsides.

Q-Learning. We optimize the policy with Q-learning, which avoids the difficulty of analyzing transition probabilities in a Markov decision process. Q-Learning is an attractive technique for policy optimization, especially for systems management, because it is model-free and light-weight. The model-free approach means we need not explicitly model the relationship between contention and latency penalties. The light-weight approach to updating models means the agent can respond quickly to changes in system conditions. For these reasons, Q-learning has already found application in server power management and memory controller design [19, 22].

The Q-value $Q^{\pi}(s, a)$ for policy π is based on expected future value when taking a particular action in a particular state.

$$Q^{\pi}(s, a) = E_{\pi}\left\{\Sigma_{k=0}^{\inf}\gamma^k r_{t+k+1}|s_t, a_t\right\}$$

Deriving the optimal policy is equivalent to learning the maximum Q-value for each state-action pair. In state s, the best action a is the one that maximizes the Q-value.

The learning procedure updates Q-values as the controller visits states, takes actions, and observes rewards. The update procedure accounts for the current reward r, the discounted Q-value under the best action a', and the previous Q-value. The learning rate of the algorithm is dictated by α.

$$Q(s, a) \leftarrow Q(s, a) + \alpha(r + \gamma \max_{a'} Q(s', a') - Q(s, a)) \quad (1)$$

The Q-table tracks Q-values for all state-action pairs. Tabular Q-learning initializes all Q-table entries with the same value. During run time, the agent repeatedly and greedily selects the action that maximizes Q-value in its current state. The agent updates the Q-value for state-action (s, a) with Equation 1 when it experiences the transition $(s, a) \rightarrow (s', r)$.

3.3 Learning for Colocation Management

We formulate the contention management problem as a sequential decision-making problem, illustrated in Figure 3. We model the problem as an MDP and solve it with Q-learning. We carefully design the action space, state space, and reward signal to satisfy Markovian properties in the Reinforcement Learning framework.

Action Space. We define actions for the control agent to be the available core frequencies in the system. The optimal action depends on the current state of the system and workload.

State Space. We design a state space with five features. We choose features that characterize short-term uncertainty in the workload and system. We restrict the space to five features to constrain state size and Q-table storage overheads. The features include software statistics that describe the interactive workload's performance, hardware counters that describe the system's resource contention, and a single value that describes the voltage and frequency setting for the batch workload's cores.

We use two software statistics, maximum queue length and maximum service time, to characterize uncertainty and variation in the interactive workload. The maximum queue length measures the largest number of pending requests, which is measured when each request enters the system, in

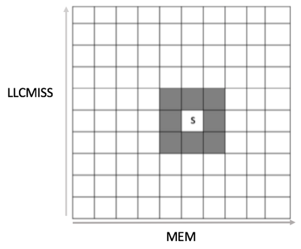


Fig. 4. The Q-value for state s is computed by averaging Q-values for adjacent states in a simplified two-dimensional state space.

the past time interval. The maximum service time measures the longest time required to execute a request in the past time interval. The control agent learns the most effective core frequency for the batch workload for given queue lengths and service times by trying different settings and observing the interactive workload's tail latency.

We also characterize resource contention with two hardware performance counters, batch cores' last-level cache misses and server-wide memory bandwidth usage. We measure cache misses to assess the competition for cache lines between co-runners. We count memory transactions to characterize the contention for shared memory bandwidth. These profiles account for system conditions and phases that vary across time within batch workloads. Finally, we include the current clock frequency for the batch workload's cores. Together, hardware counters and core frequency describe server resources that have been allocated to the batch workload.

Features in our state space are either continuous or unbounded, but Q-tables require discrete representations. First, we bound queue length, service time, memory throughputs with cut-off values such that any value greater than the cut-off is assumed to cause performance violations in the interactive workload. Second, we divide each continuous domain into twenty bins with equal ranges. The discretized space includes $O(10^5)$ states.

Reward Function. We define the reward as a function of interactive latency and system utilization. The function must encode the control agent's two optimization objectives. First, the agent seeks to minimize the difference between measured and target tail latency for the interactive workload. Second, the agent seeks to maximize task throughput for the batch workload. We realize these objectives with the following function.

$$R = \begin{cases} = \text{ Batch Frequency} & \text{Latency} \leq \text{Target} \\ = -2 \cdot \text{ Max Frequency} & \text{Latency} > \text{Target} \end{cases}$$

The reward function balances competing objectives for the interactive and batch workloads. The control agent is incentivized to meet interactive latency targets with a positive reward when the target is met and a negative reward otherwise. The agent is incentivized to sustain high system utilization with a reward that is proportional to the batch workload's processor core frequency.

Penalties for violating latency targets are large and set at twice the core's maximum frequency. The magnitude of this penalty gives the agent flexibility to balance long-term reward and short-term penalty. For example, the agent may increase the batch workload's core frequency, which leads to an immediate penalty but sustained rewards in subsequent management intervals. Such trade-offs are encoded in the Q-value for each state-action pair.

3.4 Improving the Learning Mechanism

Although Q-learning is guaranteed to converge in theory, two difficulties arise from tabular Q-learning in practice. First, the agent may rarely observe some states in the table and suffers from randomly chosen actions in those states. Updates to these states are not only rare, but also slow. The agent updates the Q-value for the state-action pair (s, a) only after it has taken the action a at state s. There is a one-step delay between updating the Q-value and using the new Q-value. In practice, the agent may observe some states only once and needs a mechanism to improve on actions taken in rarely visited states by drawing on Q-values from other states.

Second, baseline Q-learning uses ϵ-greedy algorithms to avoid local minima. This algorithm selects a random action with probability ϵ and follows a greedy strategy with probability $1 - \epsilon$. In practice, choosing a good value of ϵ is difficult, depends on hand-tuned values for a specific control problem, and is hard to justify. Moreover, a fixed ϵ causes the agent to react slowly to changes in system conditions. We propose remedies to address these challenges—neighborhood value approximation and adaptive exploration.

Neighborhood Approximation. One way to avoid the one-delay between updating and utilizing the Q-value is to exploit similarities between adjacent states in the Q-table. Because nearby states are similar, we can approximate an unvisited state's Q-values with the average of neighboring Q-values [9], as shown in figure 4. Approximation extends Q-values of visited states to rarely visited states, avoiding the poor outcomes from choosing actions randomly. We use this approximation to choose an action when a state has been visited fewer than k times and use the state's own Q-value otherwise. In our implementation, we set k to be 4 and the neighborhood radius to be 2.

Adaptive Exploration. Balancing exploration and exploitation is one of the most challenging parts in online reinforcement learning. Excessive exploration prevents the controller algorithm from pursuing short-term rewards, causing the system to violate latency targets frequently. Exploration is as likely to choose a poor action as it is to discover a good one because the ϵ-greedy algorithm selects an action uniformly at random from the space. On the other hand, exploiting a developed policy may prevent the agent from discovering greater long-term rewards. The controller may stay with a sub-optimal policy that often throttles the batch workload.

We propose adaptive exploration with a mechanism that combines Softmax Exploration [6] and Value-Difference Based Exploration [39]. Softmax Exploration uses the Boltzamann distribution function to determine the probability that an agent selects action a in state s. T is the temperature factor in the Boltzman distribution. As $T \Rightarrow 0$, the agent explores less.

$$P(s, a) = \frac{\exp\left(Q(s, a)/T\right)}{\sum_{i=1}^{m} \exp\left(Q(s, a_i)/T\right)}$$

Value Difference Based Exploration (VDBE) extends the original ϵ-greedy algorithm by assigning independent $\epsilon[s]$ for each state and adjusting $\epsilon[s]$ over time. The desired outcome is that the agent aggressively explores when it knows little about the environment and follows the developed policy when it understands the environment. Knowledge of the environment is reflected by the difference δ between existing and updated Q-values. According to this intuition, we adapt $\epsilon[s]$ with the following rule whenever we update the Q-value for (s, a).

$$\epsilon_{t+1}[s] = \delta \cdot \frac{1 - \exp\left(-|\alpha \cdot \delta|\right)}{1 + \exp\left(-|\alpha \cdot \delta|\right)} + (1 - \delta) \cdot \epsilon_t[s]$$

3.5 Deploying the Algorithm Online

Algorithm 1 summarizes our control algorithm. The algorithm has two tunable parameters: the discount factor γ and the learning rate α. First, the discount factor reflects how much less the agent values expected future rewards compared with the current reward. A small discount (i.e., large γ) motivates agents to increment the batch workload's core frequencies to achieve higher task throughput in subsequent management intervals. In contrast, a large discount (i.e., small γ) says that rewards and interactive performance now is worth much more than rewards and batch throughput later. Second, the learning rate represents the agent's expectations for changes in the environment. In our system, $\gamma = 0.9$ and $\alpha = 0.5$.

Algorithm 1 Q-Learning for controlling the batch workload's core frequency

```
1: **Initialize** Q-table with $Q(s,a) = \frac{-max\_penalty}{1-\gamma}$
2: **Initialize** $\epsilon[s]$ to be 0.1 for all state $s$
3: **Initialize** BatchDVFS to be lowest
4: **while** True **do**
5:     sleep(50 ms)
6:     (LLCMiss,Mem) ← Get_Batch_Stage()
7:     (maxService,maxQueue,maxLatency) ←
8:    Get_Req_Statistics()
9:     $s_t \leftarrow$ (maxService,maxQueue,LLCMiss,Mem,BatchDVFS)
10:    $r_t \leftarrow$ reward(maxLatency, BatchDVFS)
11:    **if** rand() < $\epsilon[s_t]$ **then**
12:        $a_t \leftarrow$ Softmax($s_t$)
13:    **else**
14:        $a_t \leftarrow$ SelectwihMaxQ($s_t$)
15:    **end if**
16:    BatchDVFS ← $a_{t+1}$
17:    $Q(s_{t-1}, a_{t-1}) \leftarrow Q(s_{t-1}, a_{t-1}) + \alpha \cdot (\gamma \cdot (r_t +$
18:    $\max_a Q(s_t, a) - Q(s_{t-1}, a_{t-1})))$
19:    $\epsilon[s_t] \leftarrow$ VDBE($\epsilon[s_t]$)
20: **end while**
21: **return**
```

4 CONTROLLER IMPLEMENTATION

We implemented the controller with over 1K lines of python and C++ codes. The controller runs as a daemon process. It periodically polls queue statistics from the interactive workload and reads hardware counters from the system. It supplies these measurements to the Q-learning agent, which returns an action from the Q-table. The controller uses this action to update the frequency setting on the batch workload's processor cores.

Figure 5 presents the controller architecture. The controller and interactive workloads use inter-process communication and share a memory page [3]. The page contains a list of query service times, query latencies, and queue lengths at the time of query arrivals. The list is updated by the interactive workload whenever a query completes. The list is read and flushed by the controller periodically. The flush ensures that information read by the controller is independent in time, which guarantees the Markovian assumptions of the Q-learning algorithm. Additionally, the controller integrates performance counter monitors and uses them to read hardware counter values. Finally, the controller relies on Q-learning and Q-tables to update and issue actions.

Experience Replay. The controller implements experience replay to accelerate training [8], increasing the learning algorithm's performance by using its measured data more effectively. Figure

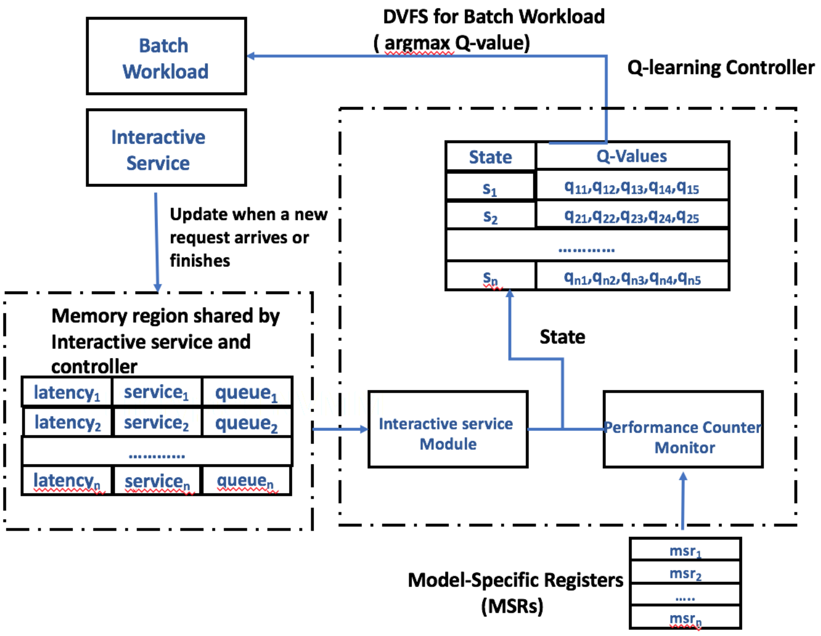


Fig. 5. The Architecture of Q-learning controller and its interaction with the interactive service and the batch workload. It communicates with the interactive service via a shared memory region. It outputs a DVFS level for the batch workload at each step.

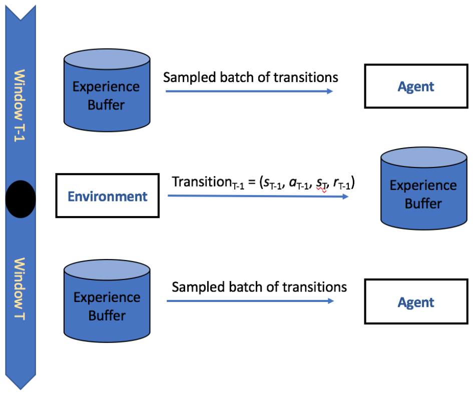

Window T-1
Window T
Experience Buffer
Sampled batch of transitions
Agent
Environment
Transition_{T-1} = (s_{T-1}, a_{T-1}, s_T, r_{T-1})
Experience Buffer
Experience Buffer
Sampled batch of transitions
Agent

Fig. 6. Online Training of the Q-learning Agent.Online, the experience buffer is updated with a state-action transition at the beginning of a time interval and trained with a batch of recorded transitions sampled from experience buffer during each time interval.

6 shows the training timeline. At the beginning of time window t, the controller meaures the state description s_t, generates a transition $(s_{t-1}, a_{t-1}) \rightarrow (s_t, r_t)$, updates the Q-table with the transition, and updates an experience list that holds the past n state-action-state transitions. The controller obtains an action from the Q-table and sets the batch workload's core frequency.

Workload	Memory Bandwidth (GB/s)
Correlation (CR)	25.05
Decision Tree (DT)	21.03
Gradient Boost Tree (GBT)	21.06
Support Vector Machine (SVM)	14.59
Linear Regression (LR)	14.66
Bayesian (BY)	23.44

Table 4. Distributed machine learning workloads (Spark) and their demands for memory bandwidth

During the time window t, the controller waits for the next round of software statistics. As it waits, it randomly samples a batch of transitions from the experience list and uses them to train the Q-table. Experience replay strengthens the agent's perception of the environment by repeating previously observed transitions. At the end of this window and the beginning of the next, the controller will receive new information and perform another transition.

Control Interval. We implement a control interval of 50ms. The control frequency represents a trade-off between hardware and software profiling. With a longer control period, the interactive workload observes a larger population of queries, which mitigates variations in query length and complexity. On the other hand, with a shorter control period, the hardware counters provide a more accurate snapshot of microarchitectural activity and contention.

5 EXPERIMENTAL METHODS

Server Configuration. We evaluate our controller on a single-socket server with an eight-core E5-2620-v4 chip multiprocessor and the Linux 3.13.0 kernel. The multiprocessor has a 2MB last-level cache and the server has 16GB memory. The nominal processor frequency is 2.1GHz and provides nine frequency steps. We use four of these frequency settings with a step size of 0.3GHz. We control each core's frequency setting with the `cpufreq` governor [2]. In addition to DVFS, the controller can temporarily stop batch computation and we denote this action as "0 GHz".

Interactive Workloads. We colocate interactive and batch workloads. We use the `Xapian` search engine and `masstree` in-memory, key-value store [21], setting the 95th-percentile latency target as the interactive workload's performance under maximum sustainable query load. We determine this load by increasing query arrival rates and measuring the inflection point at which latency increases rapidly. On two cores, the search engine sustains 600 queries per second with a 95th-percentile latency of 5.8ms. The key-value store sustains 6,000 queries per second with a 95th-percentile latency of 1.5ms.

Batch Workloads. As shown in Table 4, we use Spark for distributed machine learning on kdda'10 datasets [5]. As we observed in Section 2 and Figure 2, these workloads exhibit rich, diverse phase behaviors in the memory subsystem. `Linear Regression` (LR) and `SVM` employ stochastic gradient descent, which exhibits three recurring phases that sample data, calculate gradients, and aggregate gradients. `Gradient Boosting Tree` (GBT) and `Decision Tree` (DT) determine how best to construct a tree, which exhibits two recurring phases that sample data and recursively evaluate split points. Similarly, `Naive Bayesian`(BY) and `Correlation`(CR) exhibit several distinct phases. Recurring memory activity interferes with the interactive workload and tests the controller's ability to learn and adapt online.

Colocation. We allocate two physical cores to the interactive workload and the remainder to batch workloads. We turn off hyper-threading on the cores that run interactive workloads to avoid intra-core resource contention. Moreover, we assign interactive workloads to the highest priority

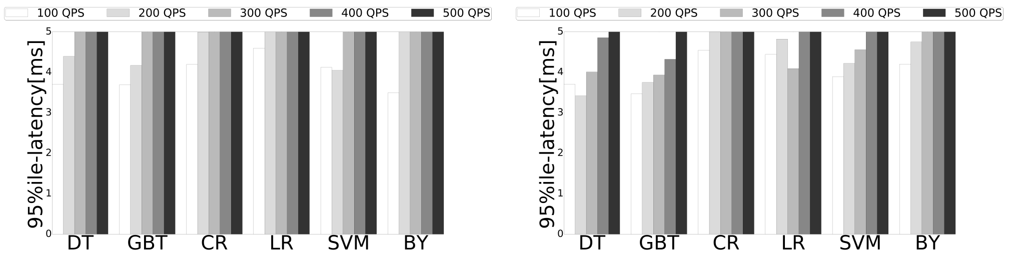


Fig. 7. Search latency with and without the reinforcement learning controller (5.8ms target)

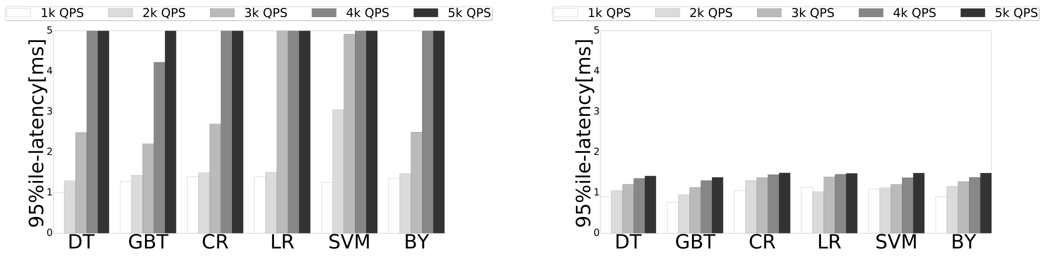


Fig. 8. Key-value store latency with and without the reinforcement learning controller (1.5ms target)

to avoid performance variations due to the operating system scheduler. Finally, we configure Spark jobs with sufficient thread-level parallelism to fully utilize the allocated cores.

6 EVALUATION

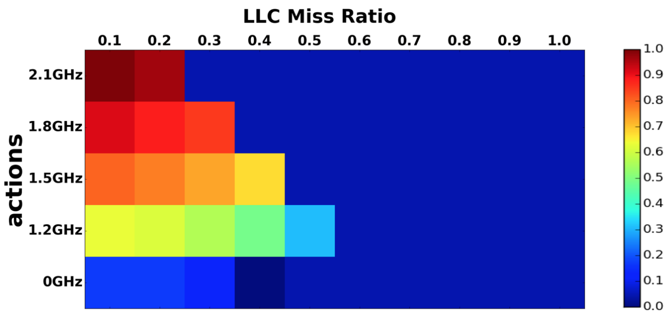
Figures 7–8 show how reinforcement learning can colocate batch and interactive workloads while ensuring the latter's service quality. The batch workloads, running at nominal frequencies and without controlling resource contention, cause the interactive workloads to violate service targets even at low query loads. In contrast, when the controller applies its policy from reinforcement learning, batch workloads adapt their core frequencies and interactive workloads satisfy their service obligations at every load level. Indeed, the controller successfully eliminates service violations even as interactive query load approaches the maximum load that can be sustained by the server.

6.1 Understanding Learned Policies

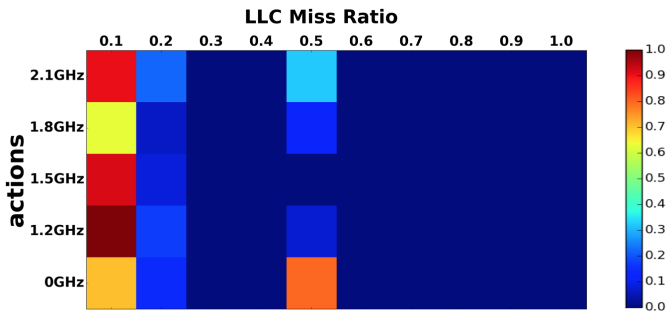
Figures 9–10 visualize reinforcement learning's results, the Q-tables. The Q-table estimates the reward from acting in a particular state. Visualizing the table provides insight into the policies learned by the controller. In the figures, grids illustrate the state (x-axis), the action (y-axis), and the corresponding reward (colormap). In each grid, we consider a particular dimension in the state space and assess the reward from an action averaged across the other dimensions. This approach projects a multi-dimensional state space into a single state to aid the visualization.

Figure 9 illustrates how the control policy responds to LLC miss ratios reported by the batch workload. In general, as the miss ratio increases, the batch workload is more likely to interfere with the interactive workload, which in turn penalizes expected rewards. To mitigate such penalties, the controller throttles batch computation and favors lower core frequencies.

The controller tailors this general policy to the specific behaviors from each batch workload. For example, compare and contrast policies when colocating search with `correlation` and with `regression`. The controller learns a more conservative policy for `correlation`. When the batch



(a) Reinforcement learning policy for search-regression colocation



(b) Reinforcement learning policy for search-correlation colocation

Fig. 9. Visualization of reinforcement learning policy for representative colocations, search-regression (top) and search-correlation (bottom). Grids show average, normalized reward for actions given varied LLC miss ratios reported by batch workload. Higher rewards indicate more likely actions in a given state.

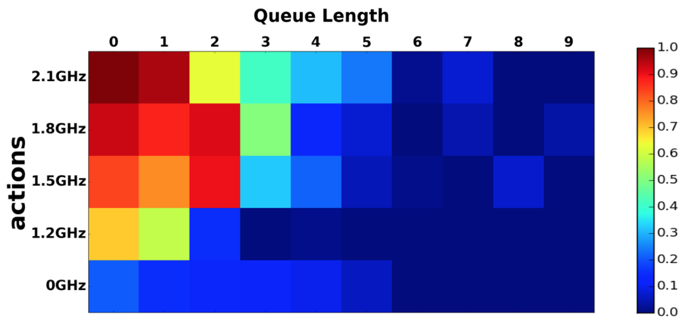
workload reports an LLC miss ratio of 0.1, the optimal frequency is 2.1GHz for `regression` but only 1.2GHz for `correlation`. The controller arrives at this policy by observing that `correlation`'s phases are relatively short and its memory activity is relatively contentious even when the reported LLC miss ratio is low.

Similarly, Figure 10 illustrates how the control policy responds to queue lengths reported by the interactive workload. Longer queues increase waiting time for computation and degrade service quality, which in turn penalizes expected rewards. The controller mitigates such penalties by lowering the core frequency for batch workloads as queue occupancy increases for the interactive workload.

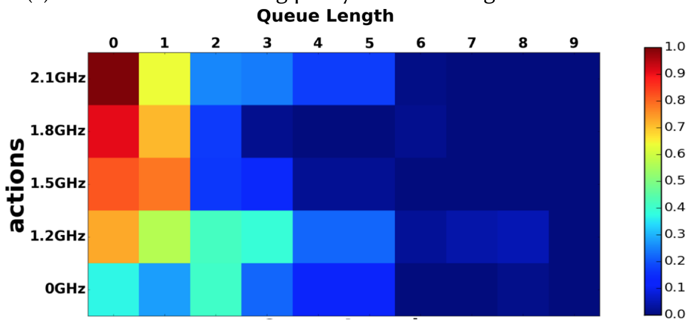
Again, the controller tailors policies to specific service requirements. The key-value store's target for tail latency is more stringent than the search engine's. And its latency is more sensitive to the number of pending queries. In response, the controller learns a more conservative policy that throttles the batch workload more often when colocated with the key-value store. For example, the optimal core frequency for the batch workload is 2.1GHz when the search engine's queue occupancy is one but is only 1.8GHz when the key-value store's queue occupancy is one.

6.2 Increasing Utilization and Efficiency

The ultimate goal of workload colocation increasing a server's utilization, thereby amortizing its fixed power costs over more computation. We measure reinforcement learning's effect on utilization even as it ensures service quality for the interactive workload. Server utilization is

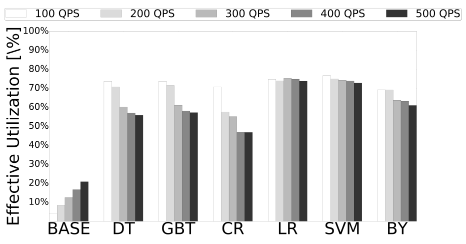


(a) Reinforcement learning policy for search-regression colocation

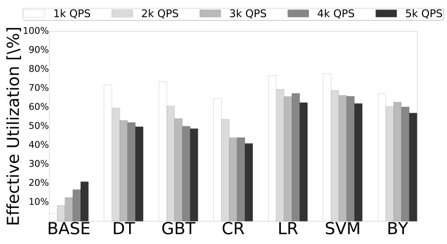


(b) Reinforcement learning policy for memcached-regression colocation

Fig. 10. Visualization of reinforcement learning policy for representative colocations, search-regression (top) and key value-regression (bottom). Grids show average, normalized reward for actions given queue lengths reported by interactive workload. Higher rewards indicate more likely actions in a given state.



(a) Server utilization for search service



(b) Server utilization for memcached service

Fig. 11. Colocations with reinforcement learning increase server utilization, a weighted average of interactive and batch workloads' utilization.

often reported relative to its peak but determining the normalizing factor can be complicated for colocated workloads. We assess utilization in three steps.

First, we normalize interactive throughput by the server's maximum sustainable throughput. If the search engine's two cores support up to 600 queries per second without violating service targets, a search engine that completes 300 queries per second leads to 50% utilization of its cores.

Second, we normalize batch throughput by the workload's task completion rate when running alone at nominal frequencies. If the Spark run-time reports 20 tasks per second under nominal

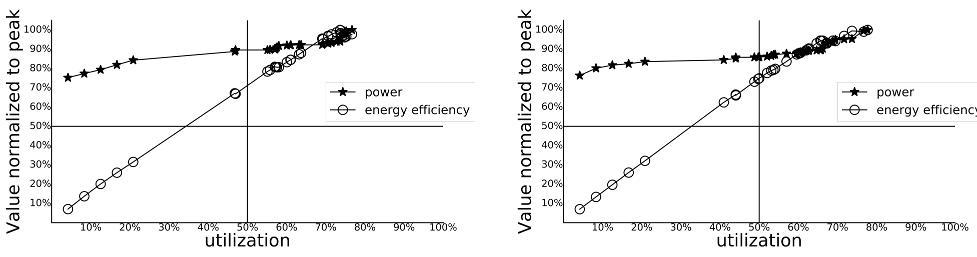


Fig. 12. Colocations with reinforcement learning increase server efficiency by amortizing fixed power costs over more (batch) computation. Power measured for interactive(left: search; right: memcached), batch workloads in Figure 11.

operation, the batch computation that completes 15 tasks per second leads to 75% utilization of its cores. Note that tasks count towards batch throughput only when the interactive workload meets its performance target.

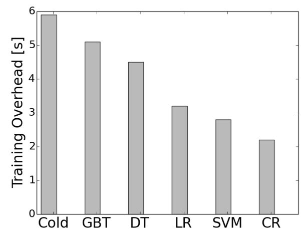
Finally, we calculate the average of interaction and batch workloads' throughputs weighted by their share of the server's cores. When the system assigns two and six cores to interactive and batch workloads, respectively, throughputs are weighted by 0.25 and 0.75. In our system, the interactive workloads, the search engine and the key-value store, cannot use more than two cores because additional thread parallelism induces contention for shared data structures such as the search engine's index table and the key-value store's in-memory database.

Figure 11 presents server utilization when the search engine and key-value store are colocated with batch machine learning workloads. In the baseline system, the lack of contention control causes any colocation to violate latency targets and utilization depends only on the interactive workload's query arrival rate. With reinforcement learning, contention control permits colocations that do not compromise interactive latencies, increasing utilization from 40% to 80% of the server's peak capacity. The controller achieves similar utilization levels for batch workloads with similar workload phases (e.g., `regression` and `svm`). Batch workloads that do not achieve high utilization, such as `correlation`, exhibit large oscillations in memory intensity that cause the controller to learn conservative policies and throttle the batch workload's processor cores. The controller is similarly conservative when colocating batch workloads with the key-value store, which is sensitive to memory system contention.

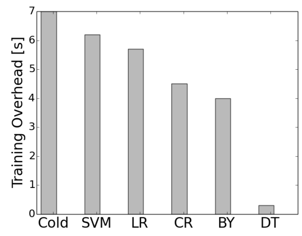
Figure 12 illustrates the server's energy disproportionality and the effect on efficiency, presenting data for every experiment in Figure 11. We define efficiency as utilization divided by power, measured relative to the server's peak power. We find that idle power, at 30W, is a dominant share of the total because we must turn off processor cores' C-states to guarantee the interactive service's performance. Under these conditions, normalized efficiency is (approximately) inversely proportional to batch throughput. As throughput increases, servers' large fixed power costs are amortized over more work and dissipated for shorter durations as jobs complete faster. Thus, by permitting colocation, reinforcement learning increases efficiency by up to 50%.

6.3 Measuring Controller Responsiveness

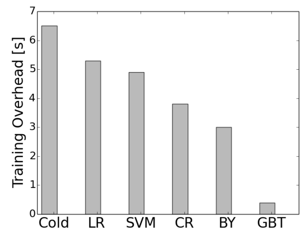
Reinforcement learning requires time to accumulate experience and optimize policies. This online training time determines the controller's responsiveness to a dynamic system. The interactive service's query load may change. Batch workloads may depart or arrive. As the system evolves, the controller must collect data and update its Q-tables.



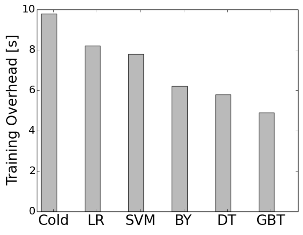
(a) Naive Bayesian



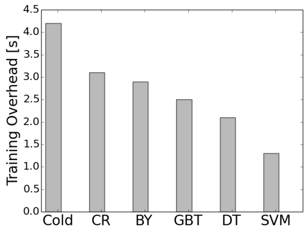
(b) Gradient Boosting Tree



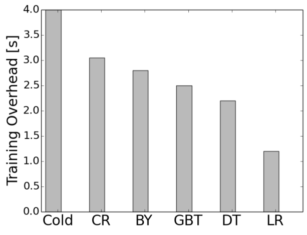
(c) Decision Tree



(d) Correlation



(e) Linear Regression



(f) Support Vector Machine

Fig. 13. Online overhead for adapting to the arrival of batch applications. The interactive service used in this set of experiments is search. x-axis in each figure represents the original batch application in the system. The "cold" data represents the coldstart scenario in which no batch application is in the system before the arrival of new batch application.

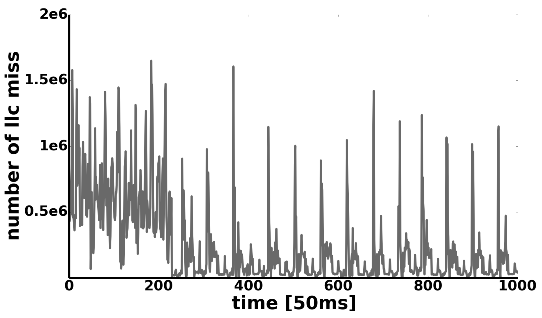
We assess training overheads by measuring how much time the controller needs to stabilize tail latencies and meet performance targets. In our definition, a time series of tail latency is stable if 99 % of the series is below the target tail latency. We measure two types of training overhead. First, reinforcement learning incurs cold-start overheads when the server is turned on and the controller construct a new Q-table. Second, learning incurs transition overheads when a batch workload completes, a different one begins, and the controller updates a previously trained Q-table.

Figure 13 illustrates how quickly reinforcement learning responds when the batch workload changes. Transition overheads are less than cold-start overheads because the controller benefits from learned policies and a complete Q-table even if they were learned for different, prior workloads. Qtables for different Batch workloads often share common characteristics as we saw in Figure 9. By exploiting these similarities, the controller spends less time transitioning between workloads.

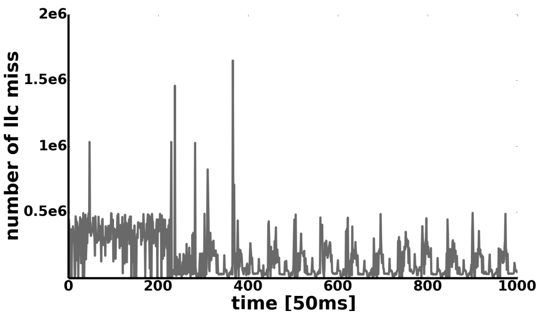
Transition overheads decrease with workload similarity. `gradient boosting tree` and `decision tree` perform similar computation to construct their trees, which leads to negligible transition overheads in Figures 13b–13c. `linear regression` and `SVM` both require gradient descent, which leads to low transition overheads in Figures 13e–13f.

Figure 14 indicates that reinforcement learning responds quickly to new batch workloads despite transition overheads. The controller can re-stabilize the interactive workload's latency shortly after the batch workload changes from `gradient boosting tree` to `linear regression` at 200s in the time series. The transition requires only tens of online profiling intervals.

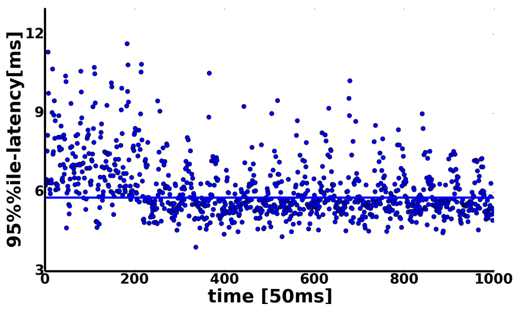
Similarly, Figure 15 shows that reinforcement learning responds quickly to variations in interactive query load. When the load peaks for the first time at the 40s, the search engine violates the targeted latency for five seconds as the controller explores new states with greater queue lengths.



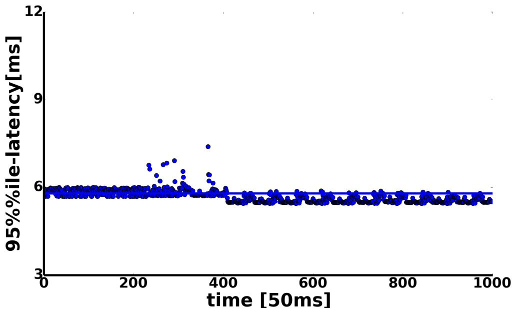
(a) LLC misses for batch cores without learning



(b) LLC misses for batch cores with learning

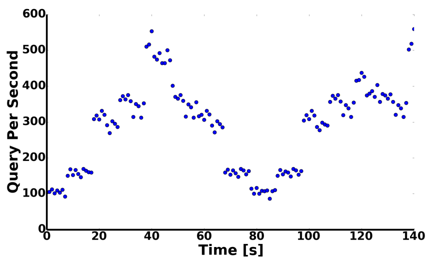


(c) Search latency without learning

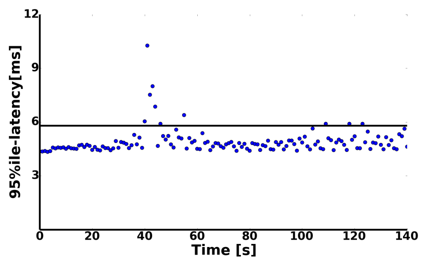


(d) Search latency with learning

Fig. 14. Reinforcement learning responds to new batch workloads while interactive service runs at peak query loads. At 200s, `gradient boosting tree` is ends and `linear regression` begins.



(a) Search query load.



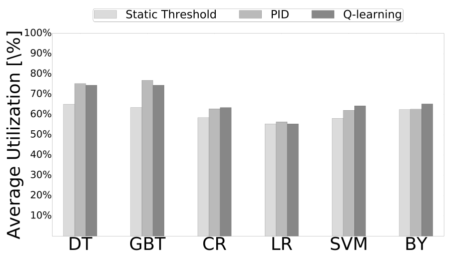
(b) Search tail latency.

Fig. 15. Reinforcement learning learns from peak at 40s, responds to peak at 140s.

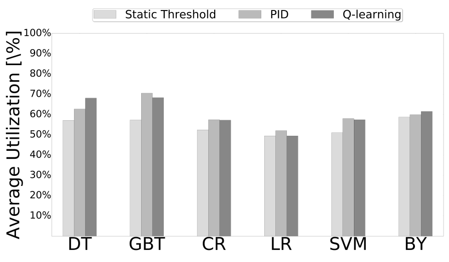
When the load peaks again at 140s, the search engine meets its performance targets as the controller implements decisions with newly learned policies and updated Q-tables.

6.4 Comparing with Feedback Control

We compare reinforcement learning against popular closed-loop methods. First, we consider static control thresholds [24, 25]. Algorithm 2 sets two thresholds, one on slack and another on query load, to control the batch workload's core frequency. Slack measures the difference between target and



(a) Result for search service



(b) Result for memcached service

Fig. 16. Comparisons between reinforcement learning and closed-loop methods [16, 24, 25, 34]. Data shown for search engine (left) and key-value store (right).

measured latency for the interactive workload. If no slack exists, batch workloads are halted. If slack is modest and less than some threshold T, batch workloads compute at lower core frequencies. If slack is abundant, but interactive query load is high and exceeds some threshold Q, batch workloads continue at current frequencies. Otherwise, batch computation proceeds at higher frequencies. We optimize parameters T and Q for each colocated batch workload to ensure that 95th-percentile latency targets are met.

Algorithm 2 StatThreshold controls batch core's frequency

```
 1: while True do
 2:     sleep(50ms)
 3:     (InstantQPS,TailLatency) ← Get_Req_Statistics()
 4:     slack ← (Target - TailLatency)/Target
 5:     if slack < 0 then
 6:         action ← StopBatch
 7:     else if slack < $T$ then
 8:         action ← DecreaseFreq
 9:     else if InstantQPS > $Q$ then
10:         action ← KeepFreq
11:     else
12:         action ← IncreaseFreq
13:     end if
14: end while
15: return
```

Second, we consider PID control to set batch workloads' core frequencies [16, 34]. Error $e(t)$ at time t is the difference between target and measured latency for the interactive workload. In each time interval, the controller computes the proportional error $p = e(t)$, the net error $i = \sum_{i=0}^{t} e(i)$, and the derivative error $d = e_t - e_{i-1}$. The weighted sum of these terms $u = k_p \cdot p + k_i \cdot i + k_d \cdot d$ is used to correct the output in next interval by increasing, decreasing, or maintaining the batch workload's core frequency.

Figure 16 compares server utilization from reinforcement learning and previously proposed closed-loop methods. The figure reports utilization averaged across five interactive query load levels. **StatThreshold** achieves less utilization than the other two methods because it is being over-conservative. On the other hand, **PID** achieves the comparable result but it requires excessive offline profiling. Determining weight parameters k_* is challenging. The root locus method requires building a transfer function between input signals (actions) and the output signal (tail latency) [16]. Doing so for every combination of interactive and batch workloads is prohibitively expensive.

Control with reinforcement learning is typically 10-30% higher than control with static thresholds. First, static thresholds pause batch computation whenever interactive performance targets are violated. Because machine learning codes can oscillate between high and moderate memory activity, simply pausing batch computation will over-throttle. Second, mechanisms with static thresholds gradually increase frequency after resuming computation, finding a suitable decision and under-utilizing the server during this time. Finally, static thresholds lack adaptivity and cannot respond to variations in workload behavior.

7 RELATED WORK

Managing Service Quality. Many studies predict the interactive workload's quality-of-service on a shared chip multiprocessor, with different degrees of offline profiling and heuristics. In contrast, Q-learning develops policies online.

Bubble up and Bubble flux [28, 41] microbenchmark batch workloads' contributions to memory system contention and predict latency with a calibration model. SMiTe [44] profiles a workload's sensitivity to shared core resources and predict tail latency with heuristics. Dirigent [45] monitors execution and compares progress with offline profiles to estimate completion time. Paragon and Quasar estimate interference with recommendation algorithms [13, 14]. These frameworks classify each application by its resource demands, sensitivity to resource contention, and recommend the best co-runner. Q-clouds and DeepDive [30, 32] address resource allocation and interference in virtual machines to ensure QoS.

Sparrow [33] schedules fine-grained tasks that complete in hundreds of milliseconds. It supports strict priority-based resource allocation policies but fails to consider resource contention between colocated tasks.

Isolating Resources in Multiprocessors. Recent microarchitectural advances motivate significant work in shared cache partitioning. Effective partitions can support varied latency, throughput, and fairness goals [11, 20, 30, 35, 36]. CPI^2 [43] develops a CPI-based software mechanism that identifies the application that causes severe contention and throttles it if necessary. Heracles [25] combines hardware and software mechanism that manage processor, memory, cache, and network usage to mitigate the interference from batch application. It uses threshold-based feedback control to dynamically adjust the resource allocation. But static thresholds may be conservative and leave throughput unexploited.

Reinforcement Learning for Systems. The most significant application of reinforcement learning in computer architecture is a self-optimizing memory controller [19], which uses Q-learning to schedule memory transactions. In addition, reinforcement learning has also been applied to datacenter resource allocation [18, 26, 27, 38, 42].

Improving Energy Efficiency in Servers. Prior work reduces energy consumed by latency-critical workloads. Q-learning is also suitable for this setting since it can exploit and adapt to short-term variations in queueing delay. We could adapt our controller with slight modifications to the state and action space.

PEAGSUS [12] improves datacenter efficiency by dynamically adjusting server power caps in response to changes in load at coarse timescales. Rubik [20] and Timetrader [40] save energy by adjusting frequency at fine timescales in response to short-term variations in queueing delay. Adrenaline [17] adjusts the frequency on a per-query basis.

8 CONCLUSION

We present a reinforcement learning framework for dynamically optimizing resource management policies in shared servers. Servers can improve efficiency by colocating interactive and batch workloads. But batch computation, especially modern machine learning applications, can exhibit severe, short-term variations that jeopardize interactive latency targets. We show that reinforcement learning can respond quickly to these variations, satisfying latency targets while increasing server utilization. The controller requires no offline profiling and learns to adapt its control policies as resource demands from batch workloads evolve.

REFERENCES

[1] [n. d.]. Cgroups - The Linux Kernel Archives. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt. Accessed: 2017-11-17.

[2] [n. d.]. CPUFreq Governor - The Linux Kernel Archives. https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[3] [n. d.]. IPC:Shared Memory - Pages supplied by users. https://users.cs.cf.ac.uk/dave/C/node27.html. Accessed: 2017-11-20.

[4] [n. d.]. LIBSVM Data: Classification (Binary Class. https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/datasets/binary.html.

[5] [n. d.]. MLlib | Apache Spark. https://spark.apache.org/mllib/. Accessed: 2017-11-17.

[6] [n. d.].]Reinforcement Learning - Exploration vs Exploitation. http://home.deib.polimi.it/restelli/MyWebSite/pdf/rl5.pdf. Accessed: 2017-11-17.

[7] [n. d.]. The sorry state of server utilization and the impending post-hypervisor era. https://gigaom.com/2013/11/30/the-sorry-state-of-server-utilization-and-the-impending-post-hypervisor-era/.

[8] Sander Adam, Lucian Busoniu, and Robert Babuska. 2012. Experience replay for real-time reinforcement learning control. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)* 42, 2 (2012), 201–212.

[9] Martin Allen and Phil Fritzsche. 2011. Reinforcement learning with adaptive kanerva coding for xpilot game ai. In *2011 IEEE Congress of Evolutionary Computation (CEC)*. IEEE, 1521–1528.

[10] Luiz André Barroso and Urs Hölzle. 2007. The case for energy-proportional computing. *Computer* 40, 12 (2007).

[11] Sangyeun Cho and Lei Jin. 2006. Managing distributed, shared L2 caches through OS-level page allocation. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 455–468.

[12] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Sonal Patil, Mei-Hui Su, Karan Vahi, and Miron Livny. 2004. Pegasus: Mapping scientific workflows onto the grid. In *Grid Computing*. Springer, 131–140.

[13] Christina Delimitrou and Christos Kozyrakis. 2013. Paragon: QoS-aware scheduling for heterogeneous datacenters. In *ACM SIGPLAN Notices*, Vol. 48. ACM, 77–88.

[14] Christina Delimitrou and Christos Kozyrakis. 2014. Quasar: resource-efficient and QoS-aware cluster management. In *ACM SIGPLAN Notices*, Vol. 49. ACM, 127–144.

[15] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, et al. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on.* IEEE, 620–629.

[16] Joseph L Hellerstein, Yixin Diao, Sujay Parekh, and Dawn M Tilbury. 2004. *Feedback control of computing systems.* John Wiley & Sons.

[17] Chang-Hong Hsu, Yunqi Zhang, Michael A Laurenzano, David Meisner, Thomas Wenisch, Jason Mars, Lingjia Tang, and Ronald G Dreslinski. 2015. Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting. In *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on.* IEEE, 271–282.

[18] Markus C Huebscher and Julie A McCann. 2008. A survey of autonomic computing?degrees, models, and applications. *ACM Computing Surveys (CSUR)* 40, 3 (2008), 7.

[19] Engin Ipek, Onur Mutlu, José F Martínez, and Rich Caruana. 2008. Self-optimizing memory controllers: A reinforcement learning approach. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on.* IEEE, 39–50.

[20] 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.* ACM, 598–610.

[21] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *Workload Characterization (IISWC), 2016 IEEE International Symposium on.* IEEE, 1–10.

[22] Wei Liu, Ying Tan, and Qinru Qiu. 2010. Enhanced Q-learning algorithm for dynamic power management with performance constraint. In *Proceedings of the Conference on Design, Automation and Test in Europe.* European Design and Automation Association, 602–605.

[23] Qiuyun Llull, Songchun Fan, Seyed Majid Zahedi, and Benjamin C Lee. 2017. Cooper: Task colocation with cooperative games. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on.* IEEE, 421–432.

[24] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. In *ACM SIGARCH Computer Architecture News*, Vol. 42. IEEE Press, 301–312.

[25] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: improving resource efficiency at scale. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 450–462.

[26] Tania Lorido-Botran, Jose Miguel-Alonso, and Jose A Lozano. 2014. A review of auto-scaling techniques for elastic applications in cloud environments. *Journal of Grid Computing* 12, 4 (2014), 559–592.

[27] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. [n. d.]. Resource Management with Deep Reinforcement Learning.

[28] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. 2011. 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*. ACM, 248–259.

[29] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal* 2014, 239 (2014), 2.

[30] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems*. ACM, 237–250.

[31] Rajiv Nishtala, Paul Carpenter, Vinicius Petrucci, and Xavier Martorell. 2017. Hipster: Hybrid Task Manager for Latency-Critical Cloud Workloads. In *High Performance Computer Architecture (HPCA), 2017 IEEE International Symposium on*. IEEE, 409–420.

[32] Dejan Novakovic, Nedeljko Vasic, Stanko Novakovic, Dejan Kostic, and Ricardo Bianchini. 2013. Deepdive: Transparently identifying and managing performance interference in virtualized environments. In *Proceedings of the 2013 USENIX Annual Technical Conference*.

[33] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. 2013. Sparrow: distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 69–84.

[34] 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 *High Performance Computer Architecture (HPCA), 2015 IEEE 21st International Symposium on*. IEEE, 246–258.

[35] Moinuddin K Qureshi and Yale N Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*. IEEE, 423–432.

[36] Daniel Sanchez and Christos Kozyrakis. 2011. Vantage: scalable and efficient fine-grain cache partitioning. In *ACM SIGARCH Computer Architecture News*, Vol. 39. ACM, 57–68.

[37] Gerald Tesauro. 2007. Reinforcement learning in autonomic computing: A manifesto and case studies. *IEEE Internet Computing* 11, 1 (2007).

[38] Gerald Tesauro et al. 2005. Online resource allocation using decompositional reinforcement learning. In *AAAI*, Vol. 5. 886–891.

[39] Michel Tokic and Günther Palm. 2011. Value-difference based exploration: adaptive control between epsilon-greedy and softmax. *KI 2011: Advances in Artificial Intelligence* (2011), 335–346.

[40] Balajee Vamanan, Hamza Bin Sohail, Jahangir Hasan, and TN Vijaykumar. 2015. Timetrader: Exploiting latency tail to save datacenter energy for online search. In *Proceedings of the 48th International Symposium on Microarchitecture*. ACM, 585–597.

[41] Hailong Yang, Alex Breslow, Jason Mars, and Lingjia Tang. 2013. Bubble-flux: Precise online qos management for increased utilization in warehouse scale computers. In *ACM SIGARCH Computer Architecture News*, Vol. 41. ACM, 607–618.

[42] Jingling Yuan, Xing Jiang, Luo Zhong, and Hui Yu. 2012. Energy aware resource scheduling algorithm for data center using reinforcement learning. In *Intelligent Computation Technology and Automation (ICICTA), 2012 Fifth International Conference on*. IEEE, 435–438.

[43] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. 2013. CPI 2: CPU performance isolation for shared compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*. ACM, 379–391.

[44] Yunqi Zhang, Michael A Laurenzano, Jason Mars, and Lingjia Tang. 2014. Smite: Precise qos prediction on real-system smt processors to improve utilization in warehouse scale computers. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 406–418.

[45] Haishan Zhu and Mattan Erez. 2016. Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems. *ACM SIGPLAN Notices* 51, 4 (2016), 33–47.