Learned Frequency Governance for Database Transactions (Regular Paper)

Udhav Sethi
University of Waterloo
udhav.sethi@uwaterloo.ca

Kenneth Salem
University of Waterloo
ken.salem@uwaterloo.ca

ABSTRACT

Servers are typically sized to accommodate peak loads, but in practice they remain under-utilized for much of the time. During periods of low load, there is an opportunity to save power by quickly adjusting processor performance to match the load. Many systems do this by using Dynamic Voltage and Frequency Scaling (DVFS) to adjust the processor's rate of execution. In transactional database systems, workload-aware approaches running in the DBMS have proved to be able to manage DVFS more effectively than the underlying operating system, as they have more information about the workload and more control over the workload. In this work, we ask whether databases can learn to manage DVFS effectively by observing the effects of DVFS on their workload. We present an approach that uses reinforcement learning (RL) to learn in-DBMS frequency governors. Our initial results indicate that learned governors are competitive with state-of-the-art methods. We also show that our approach allows frequency governance to be tuned to balance a power-performance tradeoff. However, RL is not a panacea. We also discuss two of the main challenges we identified: the overhead of the learned governor and the difficulty of training the governor for heavy loads.

1 INTRODUCTION

Dynamic CPU voltage and frequency scaling (DVFS) is an effective and widely available tool for reducing server energy consumption. DVFS allows the execution frequency of a CPU, or of an individual CPU core, to be varied over time. This enables a tradeoff between performance and power consumption. During periods of lower load, frequency, and hence performance and power consumption, can be reduced. Conversely, frequency (and power consumption) can be increased to make the full capacity of the processor available when loads are high.

DVFS must be managed. The Advanced Configuration and Power Interface (ACPI) standard defines so-called P-states, which represent distinct DVFS operating points. By switching the processor between different P-States, software can control the power-performance tradeoff offered by DVFS. Often, this control is provided by power governors in the server's operating system, which monitor operating metrics like system utilization and use them to choose P-States for the system's CPU cores. For example, Linux offers a variety of governors which differ, for instance, in how quickly they react to changes in system load. Server administrators can choose from available governors depending on their performance and power objectives.

For database management systems (DBMS), it is possible to simply allow the operating system's power governor to manage CPU frequencies in the underlying server. However, operating system governors have no visibility into DBMS-level abstractions and requirements. In particular, operating system governors do not understand DBMS-level units of work, such as queries or transactions. Thus, recent work [13–15, 19, 28, 37, 38] has focused on managing DVFS above the operating system, e.g, in the DBMS, where units of work and their performance requirements are understood.

This paper describes our on-going work, in which we ask whether database systems can learn to govern DVFS effectively. Our focus is on latency-critical transaction processing systems. Each arriving transaction request has an associated latency target, and the system's job is to complete each transaction ahead of its target, while minimizing the amount of power it consumes. Using DVFS, the system can reduce its execution frequency, slowing transactions down and saving power. However, it should not slow them so much that they fail to meet their latency targets.

One advantage of this learning approach is its flexibility. It can be applied in a variety of different types of systems, and algorithmic objectives are easily parameterized. Our on-line setting is also very conducive to reinforcement learning, since it offers a steady stream of transaction executions which can be used for training. Our primary question is whether a frequency governor learned with RL will be competitive with state-of-the-art in-DBMS frequency governors. In particular, we focus on a recent energy-aware transaction scheduler called POLARIS [14] as our baseline. POLARIS has been shown to manage DVFS much more effectively than operating system governors. However, it hard codes a specific optimization objective and specifics about the DBMS, e.g, that transactions are executed non-preemptively.

This paper offers three research contributions. First, we show how to formulate the DVFS frequency governance problem for transaction workloads as a reinforcement learning problem. Second, we evaluate the effectiveness of learned DVFS governors with respect to POLARIS and other baselines, and show that learned frequency governors are indeed competitive. Finally, we describe some of the challenges we have encountered in applying reinforcement learning in this setting. These include training challenges that arise from the operating dynamics of the underlying database system, as well as the cost and overhead of using a learned policy.

2 ENERGY-AWARE SCHEDULING

We begin with a description of the energy-aware scheduling problem that we will be considering in this paper. We assume a server running a DBMS which is subjected to a transactional workload. There is a fixed number of transaction types, and clients submit requests to execute transactions of different types. Internally, the DBMS routes requests to a set of workers, each of which executes its assigned requests in some order. There is one worker for each of the server's CPU cores, and each worker can control the execution frequency of its core. Each core offers a fixed set of possible execution frequencies (P-states) for the worker to choose from.

Each transaction request arrives with a quality-of-service requirement, in the form of a soft execution deadline. The goal of each worker is to minimize energy consumption of its CPU core, while ensuring that transactions meet their latency targets.

The DBMS has several tools at its disposal for meeting this objective. It can control how transaction requests are routed to workers, it can control the order in which requests are executed, and it can dynamically adjust the execution frequencies of the server's CPU cores. In this work, we focus on how each worker chooses the execution frequency for its core, which we call the *frequency governance* problem. We assume that the DBMS uses round-robin routing to distribute transaction requests to workers, and we assume that each worker executes its assigned transactions in earliest-deadline-first (EDF) order. POLARIS, the primary baseline algorithm we compare against, makes similar assumptions.

2.1 Related Work: Energy-Aware Scheduling

Energy-aware scheduling is a well-studied problem. A variety of on-line and off-line algorithms have been targeted at single and multiple processor processor settings. Theoretical algorithms often assume that the amount of work per transaction is known exactly, and that processors can be set to arbitrary speeds, with no upper or lower bounds. Perhaps best-known of these is Yao-Demers-Schenker (YDS) [39], which is an optimal off-line algorithm for energy aware scheduling on uniprocessors. OA (Online Available) [3] is an on-line algorithm based on YDS.

POLARIS [14] is recent algorithm that takes inspiration from YDS, but is designed to operate in a more realistic setting. It assumes that transaction execution times are not known in advance, and that the processor has only a limited range of frequencies available. POLARIS runs at each worker, where it controls the order of transaction execution and governs speed of the worker's core. Like OA and YDS, POLARIS runs transactions in EDF order. POLARIS considers frequency adjustments when transactions finish execution and when new transactions arrive. It uses dynamically adjusted estimates of transaction execution times at different frequencies to estimate lowest frequency at which it can ensure that all currently assigned transactions will meet their deadlines.

Rubik [13] and PEGASUS [19] are two other approaches to power governance for latency sensitive workloads. Rubik is similar to POLARIS in that it can adjust frequency on the time scale of individual transactions. Rubik assumes a monolithic workload and sets execution frequency based on an estimate of the execution time distributions of the queued transactions. PEGASUS is intended to manage a cluster of servers running a monolithic workload. It uses a control-theoretic approach to adjust CPU execution speeds across the cluster as the offered load fluctuates.

2.2 Related Work: Learning to Schedule

In recent years, reinforcement learning (RL) has been used to tackle a variety of problems in computer systems such as database configuration tuning [8, 17, 40], join query optimization [16, 22, 23], and datacenter congestion control [11, 32]. RL has also been extensively applied to resource management on distributed compute clusters. DeepRM [20] uses RL for resource scheduling in an on-line non-preemptive setting. It considers jobs with multi-dimensional resource profiles, and learns to schedule them in order to optimize for average job slowdown. Decima [21] uses a similar approach to learn workload-specific scheduling policies for DAG-structured jobs to minimize completion latencies. Other methods address the resource allocation inside jobs' computation graphs [1, 7, 24–26]. Liu et al [18] propose an RL-based hierarchical framework to address the overall resource allocation and power management problem in cloud computing systems.

A number of RL-based approaches have also been proposed for energy efficiency optimization through DVFS management. Some methods use RL to learn mechanisms that can choose the most appropriate DVFS management scheme from a set of existing techniques and switch between them in various situations [10, 34, 35]. For example, Islam et al [34] present an approach that manages DVFS by choosing between cycle-conserving (CC), look-ahead (LA) [2], and dynamic reclaiming algorithm (DRA) [27], while using EDF for scheduling priority. Other methods devise ways to learn frequency governors directly, both for single core [30], and multi-core systems [4, 9, 29, 33, 36]. These algorithms operate in an on-line setting and share a common objective of minimizing energy consumption under some given performance constraint. Tian et al [33], for example, define the objective as saving power while meeting user-defined performance requirements in terms of the execution time of one iteration of the running application. Some of these methods scale CPU frequencies at fixed intervals, with manually configured period lengths, generally in the order of several milliseconds [9, 29, 30]. Others trigger the DVFS controller after finishing a specific unit input data-block [33, 36]. Also, most of these approaches generally make their decisions without an extensive knowledge of the workload properties. They use low-level metrics for workload characterization, such as request generation rate, the number of CPU cycles executed, or cache misses suffered. In this work, we assume that requests belong to different classes (types), and the only workload

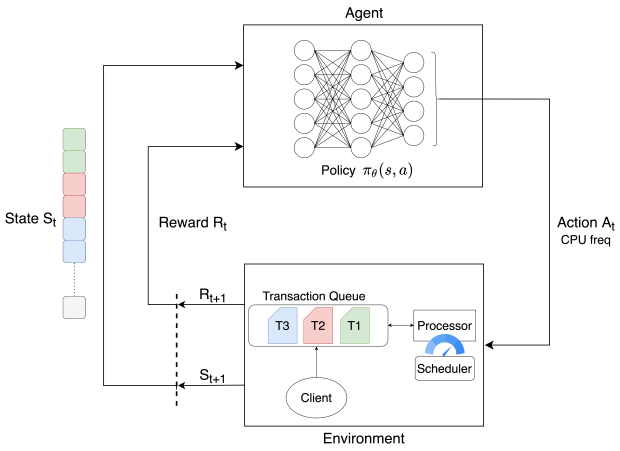


Figure 1: RL Formulation

characteristic other than the class that is available to the learner is the time to the request's deadline.

3 FREQUENCY GOVERNANCE AS A REINFORCEMENT LEARNING PROBLEM

In reinforcement learning, an agent learns to behave in an environment by performing actions and observing the results, in order to maximize some notion of a cumulative reward. At each timestep t, the agent observes a state S_t and on that basis takes an action A_t. Following the action, the environment transitions to a new state S_{t+1} and the agent receives feedback in the form of a numerical reward R_{t+1}. The agent-environment interaction in RL is illustrated in Figure 1.

The agent uses a policy π to pick its actions. In this work, we consider so-called deep reinforcement learning. In deep reinforcement learning, the policy π is represented as a deep neural network, with parameters θ, that, given a state, defines a probability distribution over the space of possible actions. That is, $\pi_\theta(s,a) \rightarrow [0,1]$, where s is an environmental state and a is an action. Through its interaction with the environment, the agent alters the policy parameters (θ) with the goal of maximizing the cumulative discounted reward that it will receive from the environment over the long run.

3.1 Environment, States, and Actions

In our case, the environment consists of a database system and a set of clients. For simplicity, we assume that the database system runs a single worker using a single processor that supports multiple P-states. The clients generate transaction execution requests, submit them to the database system, and await responses. Each transaction has a type and a latency target. The database system's worker runs transactions non-preemptively in EDF (Earliest Deadline First) order.

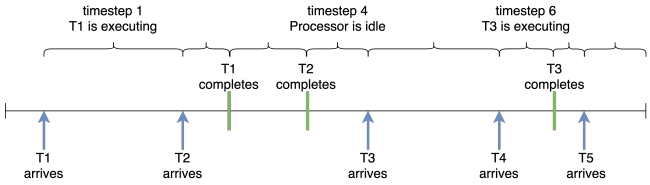


Figure 2: Time divided into variable length timesteps

We define environmental state transitions to occur as a result of either of two events: the arrival of a new transaction request, or the completion of the currently running transaction. Thus, these events divide time into a series of steps, which may differ in length. In Figure 2, for example, transaction arrivals and completions define a total of eight time steps. During each step, either a single transaction will be executing or the processor will be idle. If a transaction is executing, other transaction(s) may be queued and awaiting execution. For example, in Figure 2, the processor is idle during the fourth time step. Transaction T_3 is executing during the fifth and sixth time steps, and transaction T_4 is queued and awaiting execution during the sixth time step.

We define the state of the environment to include information about the currently running transaction as well as any transactions that are queued for execution. Specifically, for the running transaction and each of the first M (in EDF order) queued transactions, the state includes

- the transaction type
- the remaining time until the transaction's deadline

In addition, the state includes a backlog count, which represents the number of excess transactions in the queue. If the queue length is M or less, the backlog count is zero. Otherwise, the backlog count is the actual queue length minus M. The state representation does not include any information about the types and deadlines of the excess transactions represented by the backlog count. Representing the state in this way results in a fixed-size state representation, regardless of the number of queued transactions.

In our setting, the action space is defined by the set of P-states (frequencies) at which the processor can run. Thus, for each possible environment state, the policy π_θ defines a probability distribution over the available P-states. On each state transition, the agent selects the P-state to be used for the next time step by randomly selecting one according to the probability distribution defined by the policy.

3.2 Rewards

The reward is crafted to signal the agent to realize the two-fold objective of minimizing energy consumption while ensuring that the transactions meet their specified deadlines. Accordingly, the reward consists of two components, R_{energy} and $R_{failure}$. The total reward assigned to the agent for an interval t, R^t, depends on whether the currently running

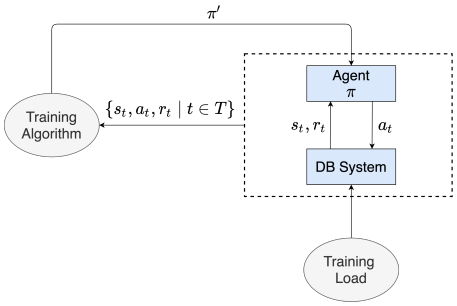


Figure 3: Training Loop

transaction fails (finishes after its deadline) during t:

$$R^t = \begin{cases} -R^t_{failure} & \text{if a deadline is missed in t} \\ -R^t_{energy} & \text{otherwise} \end{cases}$$

In the first case, $-R^t_{failure}$ represents a penalty for missing a transaction deadline. In the second case, $-R^t_{energy}$ represents a penalty for the energy consumed by the processor during time step t. The learning process seeks to maximize the (expected) total discounted reward:

$$\mathbb{E}\left[\sum_t \gamma R^t\right]$$

where $\gamma \in (0, 1]$ is the discount factor, a learning parameter. Maximizing the discounted reward corresponds to minimizing the penalties.

The energy penalty, R^t_{energy}, represents the energy consumed by the processor during timestep t. We set

$$R^t_{energy} = f_t^{\alpha} len_t$$

where f_t is the CPU frequency chosen by the agent for timestep t, α is a constant, and len_t is the length of the timestep t. Here, f_t^{α} models the frequency-dependent dynamic power consumption of CPUs. Multiplying by the length of the interval then models the energy consumed during the interval.

$R_{failure}$ is the penalty for missing a transaction's execution deadline. We define it as follows

$$R^t_{failure} = \delta E_{avg} \frac{f_M}{f_t}$$

Here, f_t is the CPU frequency chosen by the agent for timestep t, f_M is the maximum frequency from among the P-states in the action space, E_{avg} is the average energy consumed per transaction at f_M, and δ is a tunable parameter which controls the importance of avoiding deadline misses, relative to the importance of saving energy.

The formulation of $R_{failure}$ has two parts, each serving a distinct purpose. The first part, δE_{avg}, expresses the penalty for failure as the average energy consumption of a transaction multiplied by a scaling factor (δ). $R_{failure}$ increases with δ, making it less appealing for the agent to miss deadlines in order to save energy.

δ can be interpreted as the percentage power savings the agent should achieve as a trade-off for an additive 1% increase in the failure rate. Suppose that the failure rate (percentage of transactions that miss their deadlines) when the processor runs at f_M all the time is 5%. A 1% additive increase in failure rate (i.e., from 5% to 6%) would require at least δ% decrease in the energy penalty in order to reduce the overall penalty.

The second part, $\frac{f_M}{f_t}$, ensures that the failure penalty is inversely proportional to the CPU frequency in the timestep where the transaction failure occurs. This makes it more expensive to fail at lower frequencies than at higher frequencies. We describe the motivation for this term in more detail in Section 6.1.

4 TRAINING

We train the agent's policy using the training "loop" illustrated in Figure 3. In each iteration of the loop, we apply some training workload to the database system and use the current policy π to choose the CPU frequency at each step. We record the state (s_t), chosen action (a_t), and reward (r_t) at each time step t. This continues until the system has run for a fixed number of (non-idle) time steps. At the end of the iteration, we use the collected information and a training algorithm to adjust the policy parameters (θ). Then we repeat, using the adjusted model π' in the next iteration. Each iteration is called an *episode*.

For training, we use a policy gradient method, a type of RL technique in which the policy is optimized directly with respect to the expected long term discounted cumulative reward. This is done by performing gradient ascent on the policy parameters. According to the policy gradient theorem, for any differentiable policy $\pi_\theta(s, a)$, the policy gradient is given by:

$$\nabla_\theta J(\theta) = \mathbb{E}[\nabla_\theta \log \pi_\theta(s, a)\, Q^{\pi_\theta}(s, a)]$$

Here, $Q^{\pi_\theta}(s, a)$ is the expected discounted cumulative reward which will result from picking action a in state s, and subsequently following policy π_θ. We use a Monte Carlo method, in which each training episode is treated as a random sample that is used to produce v_t, an unbiased estimate of $Q^{\pi_\theta}(s_t, a_t)$:

$$v_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...$$

We then use v_t to update the policy parameters using the REINFORCE policy gradient algorithm [31]:

$$\theta \leftarrow \theta + \eta \sum_{t=1}^{T} v_t \nabla_\theta \log \pi_\theta(s_t, a_t)$$

where T is the number of timesteps in an episode, and η is a meta-parameter called the step size. Intuitively, $\nabla_\theta \log \pi_\theta(s_t, a_t)$ provides a direction in the parameter space to increase the probability of choosing action a_t at state s_t, while v_t estimates how good (or bad) it is to move in this direction. The net effect of these updates over all of the episode's time steps

Transaction Type	Mean Execution Time (μs)					Latency Target (μs)
	1.2 GHz	1.6 GHz	2.0 GHz	2.4 GHz	2.8 GHz	
New Order (45%)	4772	4094	3415	2737	2059	20590
Payment(47%)	733	625	517	409	301	3010
Order Status (4%)	809	669	529	390	250	3900
Stock Level (4%)	8062	6905	5748	4592	3435	34350

Table 1: Transaction execution times and latency targets at various CPU frequencies. Percentages indicate the transaction mix in the workload.

is to increase the probability of picking actions that lead to better returns.

5 EVALUATION

We conduct an evaluation of our reinforcement learning method to determine whether it can learn policies that can manage the processor frequency effectively. In this section, we consider whether learned policies are competitive with the state-of-the-art POLARIS frequency governor for transaction processing systems. We also consider how quickly the policies can be learned. Later, in Section 6, we switch our focus to the difficulties we encountered in using reinforcement learning for frequency governance.

5.1 Transaction Simulation

To make it possible to quickly explore a variety of RL problem formulations and identify major challenges with the approach, we used a simple discrete event simulation of the transaction execution system, rather than running actual transactions against a database. The simulator consists of a workload generator that generates and enqueues transaction requests, and a simulated worker which executes queued requests sequentially in EDF order.

The transaction generator generates requests for the four types of TPC-C [5] transactions shown in Table 1, with generation probabilities shown in the table. For example, 45% of transaction requests are for New Order transactions. After generating a transaction request, the generator waits for an exponentially distributed think time before generating the next request. The mean of the exponential distribution is determined by the generator's *utilization* parameter. It can be set so that the average request arrival rate will be 20%, 40%, 60%, or 80% of the maximum arrival rate that can be sustained by the simulated processor at maximum frequency for the transaction mix given in Table 1. Each generated transaction is assigned a type-dependent latency target, as shown in the table's Latency Target column.

The transaction execution simulation is based on measurements of TPC-C transactions running against an in-memory database in Shore-MT [12], as reported by Korkmaz et al [14]. Key parameters are summarized in the Mean Execution Time columns in Table 1. It shows the measured mean execution times for each of four TPC-C transaction types (in Shore-MT) at each of five processor frequencies, ranging from a minimum of 1.2 GHz to a maximum of 2.8 GHz. Our simulator

Parameter	Value	Description
M	10	transactions in state
δ	3	failure reward weight
γ	0.99	reward discount
η	0.0001	REINFORCE step size
T	5000	timesteps/episode
	8000	number of training epochs

Table 2: Simulation and Learning Parameters

simulates a processor with five available P-states, corresponding to the five execution frequencies shown in Table 1. The simulated execution time of a transaction of a given type at a given frequency is determined randomly, using an exponential distribution with a mean determined by the measured values in Table 1. For example, the execution time of Payment transactions at 2.0 GHz is determined using an exponential distribution with a mean of 517 μsec. This variability models the fact that execution times for transactions of the same type may vary due to factors such as transaction parameter values and contention with other transactions for access to the underlying database. (Note that the parameters shown in Table 1 are known to the simulator, but are are not part of the state representation used for learning.)

As the simulator runs, it produces output each time a new transaction request is generated and each time a transaction finishes execution, i.e., after every time step shown in Figure 2. The output indicates whether a transaction completed in the timestep, and if so whether the completed transaction finished within its deadline. The simulator also reports the energy consumed by the simulated processor during the previous timestep. Energy consumption is proportional to $L_t f_t^2$, where L_t is the length of the time step, and f_t is the processor's frequency during the time step, and is calibrated to match the power measurements for TPC-C workloads in Shore-MT as reported by Korkmaz et al [14]. Dynamic power consumption in CPUs is typically modeled as proportional to f^α, with $1 < \alpha < 3$.

5.2 Experimental Methodology

In each experiment, we pick a specific frequency governor to test. We allow the generator to generate a series of transactions, which are executed by the worker at CPU frequencies

chosen by the frequency governor under test. The worker consults the frequency governor and adjusts the processor speed each time a new transaction request is generated and each time a new transaction starts execution.

There are two outputs for each experimental run. The first is the measured *failure rate*, which is defined as the percentage of completed transactions that finished after their assigned deadline. The second output is the average amount of energy consumed by the processor per transaction execution. This is determined by summing the energy consumed across all of the time steps in the run, and dividing the total by the number of transactions that completed execution during the run. Each experiment is run for 100 iterations and the mean values of these two metrics are reported across all runs, along with a 95% confidence interval (shown in the form of error bars/bands on the result plots)

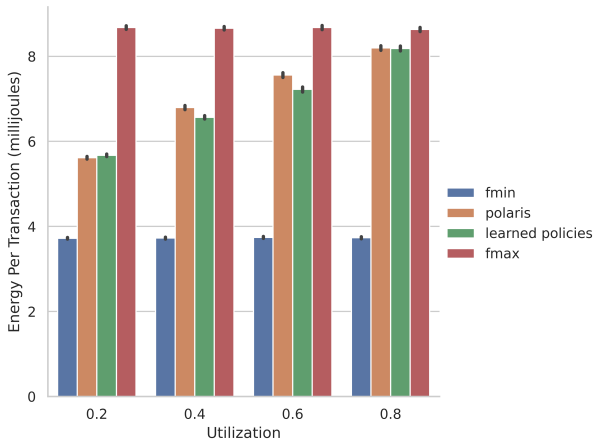
We experiment with a variety of frequency governors that use policies learned using our RL methodology. We also experiment with several non-learned baseline governors, including POLARIS [14] (which is implemented in our simulator) and two fixed-frequency baselines, which we refer to as f_{min} and f_{max}. The former always runs the processor at the lowest frequency (1.2 GHz), and the latter always runs at the highest frequency (2.8 GHz).

We built the policy neural network used in the learned frequency governors using 5 fully connected hidden layers with 256, 128, 128, 64, and 64 neurons respectively. The values of other simulation and learning parameters are as shown in Table 2 unless stated otherwise.

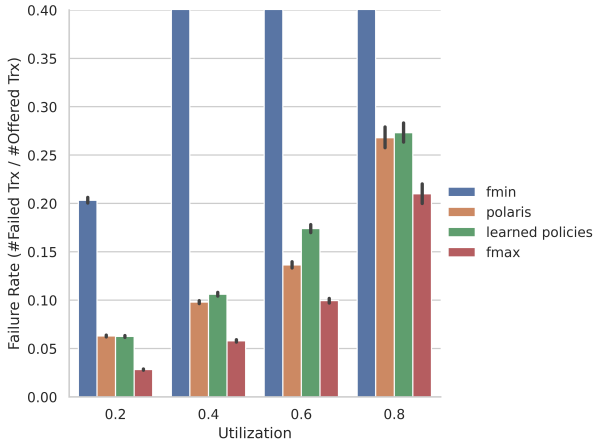
5.3 Performance of Learned Governors

In our first experiment, we compared learned governors against the state-of-the-art POLARIS baseline. We used training workloads at 20%, 40%, 60%, and 80% utilization to train four different governors, which we refer to as m20, m40, m60, and m80. We then ran experiments with test workloads generated at the same four utilizations. At each test utilization, we compare POLARIS to the learned governor trained *at the same utilization*. This is the best case for the learned governors, since training and testing workloads are of comparable intensity. We also ran each test workload with the f_{min} and f_{max} static governors to determine upper and lower bounds on energy consumption and transaction failure rates.

Figure 4 shows the top-level results from this experiment. At all utilizations, the learned governors offer failure rates and energy consumption similar to what POLARIS achieves. However, we also found that while POLARIS and the learned governors have similar performance, they do behave differently. Figure 5 shows the *frequency residency distributions* for POLARIS and the learned policies at each test workload level. The frequency residency distributions show how much time the processor spends at each frequency level during the test run. As the figure shows, under the learned policies the processor spends almost all of its non-idle time at either the maximum frequency or the minimum frequency. In contrast,



(a) Energy per Transaction



(b) Transaction Failure Rate. The failure rate for f_{min} approaches 1.0 at higher utilizations. The y-axis has been truncated at 0.4 to more clearly distinguish the other governors.

Figure 4: Performance of learned Governors and baselines. Training utilization matches test utilization.

POLARIS makes some use of the intermediate frequency levels.

Figure 6 illustrates the model training process for the m60 model. It shows the frequency residency distribution of the processor during each training episode. Training starts with an initial model that assigns similar probabilities to each possible frequency. As training progresses, the model quickly learns to favor higher frequencies. This provides a substantial reward payoff by driving down costly penalties due to failed transactions. Once avoidable failures have been eliminated, the model gradually "relaxes", choosing lower frequencies more often in a quest to improve rewards by reducing energy consumption, without re-introducing failures.

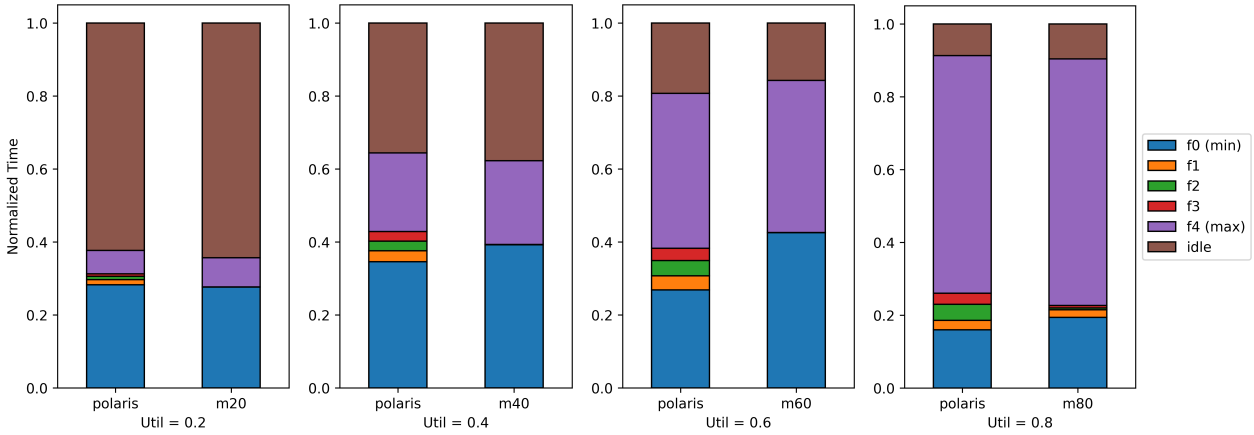


Figure 5: Processor frequency residency distribution during evaluation

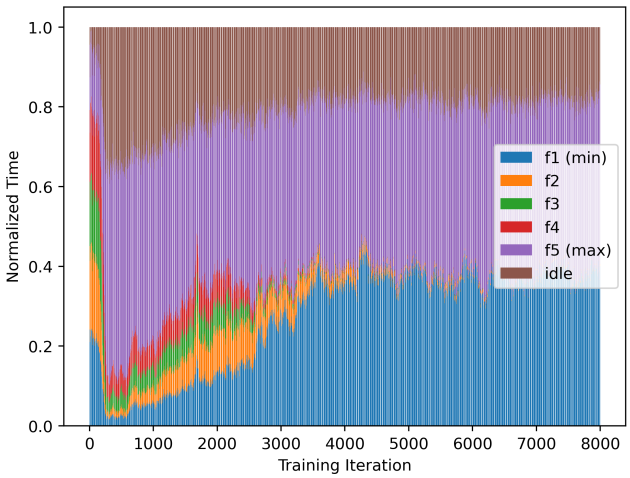


Figure 6: Processor frequency residency distribution during training at utilization 0.6

5.3.1 Training and Testing at Different Utilizations. In practice, it would be best to have a single learned governor that would perform well at all load levels. Otherwise, some additional control mechanism would be needed to switch among learned frequency governors as the load fluctuates. But, how should we train such a "load-universal" governor? In our next experiment, we evaluate how well models trained at one utilization perform when tested using a different utilization. We also consider a model learned using training loads in which the utilization varied during each epoch, starting low (20%), stepping up to 80%, and then stepping back down to 20% before the end of the epoch.

Figure 7 shows the results of this experiment. We found that the m20 and m80 models tended to result in lower failure rates and higher energy consumption than the models trained at intermediate utilizations. However, the differences were not that large - we expected that a model trained at low load would perform poorly when tested at high load, and vice versa. Overall, the model trained with fluctuating training loads, labeled "mcross" in Figure 7 seems to be a consistent middle-of-the-road performer.

5.3.2 Flexible Objective. One advantage of learned frequency governors is that it is easy to train them to different objectives. In contrast, POLARIS is hard-coded to try to avoid missing deadlines, regardless of the energy cost of doing so. In our RL formulation, the parameter δ acts as a knob which balances failure and energy penalties in the reward function.

We ran a simple experiment to illustrate the effect of δ. Using training workloads with 60% utilization, we trained models using δ values from ranging from 1 to 7 (the default is 3). We then tested each model using a workload at 60% utilization.

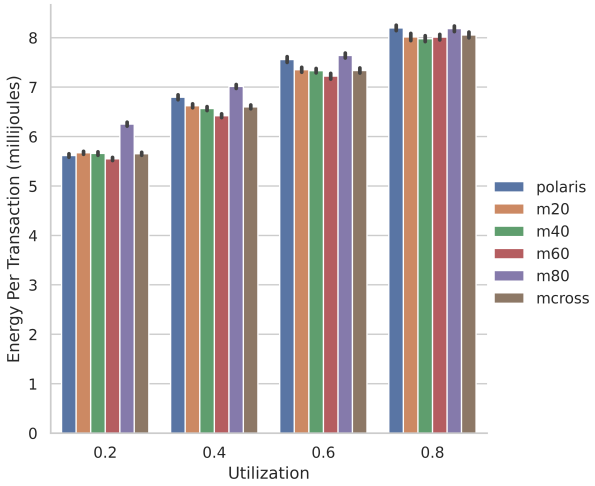
Figure 8 shows the energy consumption and failure rates that resulted from these models. The figure also shows the POLARIS baseline, for comparison. Low δ values reduce the penalty for failed transactions. Thus, we see higher failure rates than POLARIS, but also lower energy consumption. The situation is reversed at the $\delta = 7$. Setting $\delta = 5$ results in performance that closely matches POLARIS.

6 CHALLENGES

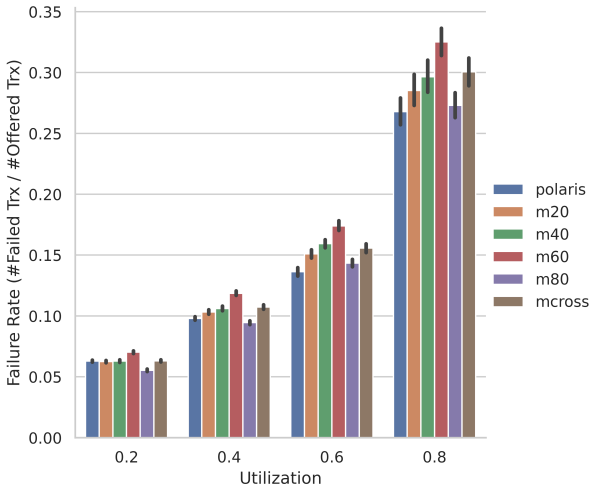
Our evaluation shows that it is possible to use reinforcement learning to train frequency governors that achieve performance comparable to that of a state-of-the-art non-learned governor. However, reinforcement learning is hardly a panacea, and there are some challenges in using this approach to control software systems. We discuss two of them in this section.

6.1 Stability During Training

The hallmark of reinforcement learning is that one starts with some initial policy and gradually refines it by exploring alternative actions and observing rewards. However, when RL is used to train a controller for a queueing system under

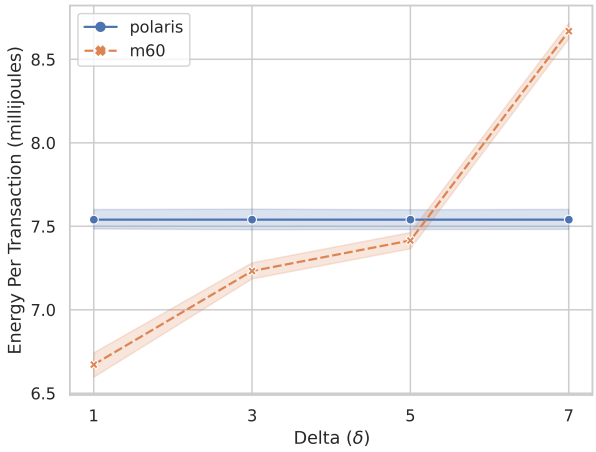


(a) Energy per Transaction

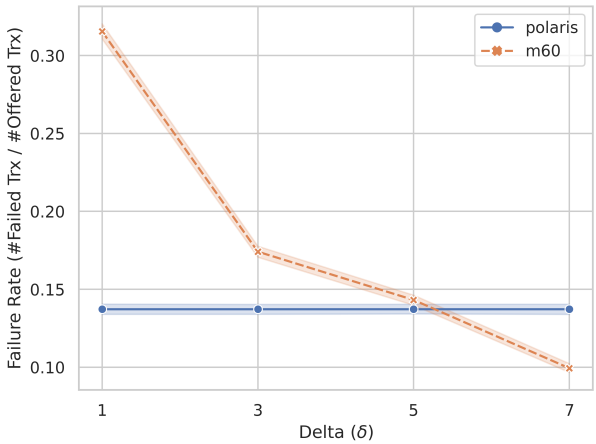


(b) Transaction Failure Rate

Figure 7: Cross-Utilization Performance of Learned Governors



(a) Energy per Transaction



(b) Transaction Failure Rate

Figure 8: Effect of Varying Reward Function

continuous load and the initial policy is poor, the system being controlled can become unstable. In our case, if the policy tends to choose execution frequencies that are too low to handle the training workload, then requests can quickly build up in the request queue, leaving the transaction system in a state in which no transactions can make their deadlines.

In our work, we used episodic training. This means that the policy is fixed during each training epoch, and is then adjusted for the next epoch. Thus, if the initial policy in an epoch is bad, the system becomes unstable during the epoch. Ideally, this situation would gradually improve during subsequent epochs as the training pushes the policy to choose higher frequencies. Indeed, Figure 6 showed an example of this behavior in the early training epochs. However, we observed that this sometimes does not happen. When the system is overloaded and the work queue is large enough, transactions miss their deadlines no matter which frequency the policy chooses. In this part of the state space, the objective function looks like a large nearly-flat plateau of unpalatable choices. This plateau slopes gradually towards a poor *local* minimum that is achieved by slowing the processor down to save energy, even though the system is overloaded. Indeed, we saw exactly this behavior during training at high loads.

We considered several approaches for solving this problem:

meta-governor: The first option we considered is to use multiple policies, with a meta-governor to switch among policies. Specifically, if the system becomes unstable, the meta-governor would use a policy that runs the CPU at maximum frequency all the time. Otherwise, the meta-governor would use the learned policy. The meta-governor switched between policies using a simple threshold on the queue length.

We were unhappy with this approach for several reasons. First, it adds complexity and an additional tuning parameter which is outside the scope of the learning process. Second, it complicates training of the learned policy. In particular, do we train the learned policy with the meta-governor in place, or without it? If the meta-governor is in place, does the reinforcement learning reinforce the action that the policy chose, or the action that the meta-governor ultimately chose? If the meta-governor is not in place during training, how do we avoid instability during training, or account for its effects? Pilot training experiments with a meta-governor resulted in oscillatory behavior during training, contributing to our decision not to pursue this further.

initial policy: The second option we considered was to control the initial policy that we start training from, in the hope of avoiding system instability. By default, our training started with a policy that assigned equal probability to all actions (frequencies). On average, this policy results in the system running about halfway between its minimum and maximum frequencies during the initial training epoch. Thus, any training workload with a request rate above half of the system's capacity (at peak frequency) would result in instability.

To correct for this, we could use an initial policy that is biased towards higher frequencies, so that the training workload would not produce instability. Essentially, the system would start with a "go fast" policy, and would learn to slow it down according to the training workload.

One drawback of this approach is that we have to decide how strongly to bias the initial policy. More bias reduces the likelihood that the training workload will result in instability, but will increase training time when the training workload is light. A more significant problem, though, is that this approach can reduce but not eliminate the problem, because the bias may not be sufficient. Furthermore, it does not increase our confidence that the governor will behave well in practice when the system is subjected to a high load higher than the training bias.

adjusted reward: A third approach, which we ultimately chose, is to adjust the reward function to eliminate the local minimum that the training would otherwise gravitate towards when the system becomes unstable.

Specifically, our reward function (Section 3.2) originally did not include the term $\frac{f_M}{f_t}$ in the failure penalty ($R_{failure}$). We introduced that term specifically to combat this problem. The effect of this term is to make transaction failure more expensive the slower processor is running when the transaction fails. When few transactions are failing, this term has little effect. However, when many transactions are failing, it encourages the training to reinforce the high frequency action, rather than the low-frequency energy-saving action. As the model evolves during training, this eventually pushes the system out of the "overload" part of the state space. Figure 9 illustrates the effect of this extra term in the reward function. For this figure, we ran an experiment in which we trained two models using a training workload at 80% utilization, which is high enough to result in overload. One model was trained with the reward function shown in Section 3.2. The second model was trained using a reward function without the term $\frac{f_M}{f_t}$ in $R_{failure}$.

The figure shows the energy per transaction and the transaction failure rate achieved by the two models after each training epoch. Both models start with a transaction failure rate near 100% because the initial policy model results in an unstable system at 80% load. Without the reward adjustment, the model reduces frequency to move to a local minimum. The failure rate stays near 100%, but energy per transaction drops. This model only ever sees the overloaded part of the state space during training, as it is never able to move the system out of overload. In contrast, with the new term the high frequency choice gets reinforced as long as the transaction failure rate is high, and the model eventually learns to bring the failure rate down.

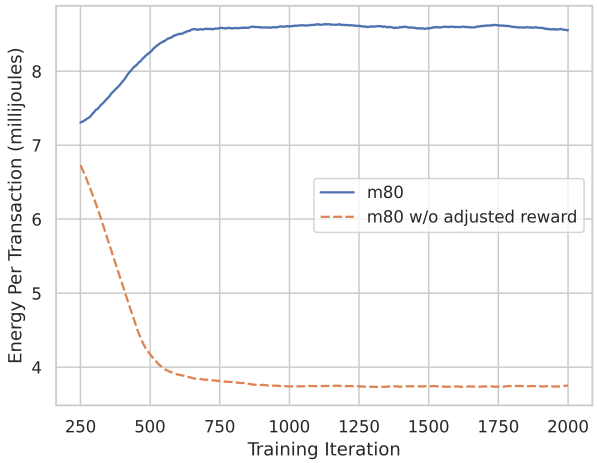
In our view, the major advantage of this approach is that it allows us to safely expose the model to the "overload" part of the state space during training. This should result in a frequency governor that can behave well even during periods of overload. Since overloading may occur in practice, this is preferable to trying to hide or prevent overloads during training.

6.2 Overhead

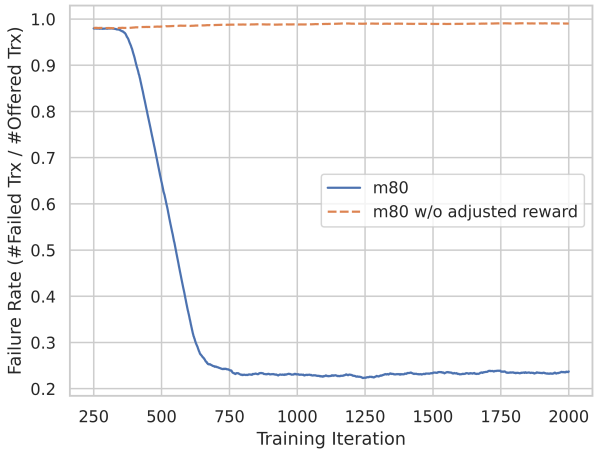
The second problem with use of deep RL for frequency governance is the overhead of using the learned model. Our technique trains a model that maps from the state of the transaction processing system to a probability distribution over the possible processor frequencies. To use the model, the transaction processing system must encode the current state, run the model to determine the probability distribution, and then choose a frequency according to the distribution. The model is used frequently: every time a new transaction requests arrives, and every time a transaction finishes execution.

We ran a simple experiment to measure the overhead of choosing a processor frequency. We used the agent to make 50000 frequency recommendations, and measured the total wall clock time required. We ran two variants of the experiment, one in which model inference occurs on our server's Intel(R) Xeon(R) Silver 4114 CPU, and a second in which inference is performed on an NVIDIA Tesla P40 24GB GPU. In each case, we report the time per recommendation, which is the total measured wall clock time divided by the number of recommendations generated.

Since model inference time depends on the size of the neural network, we repeated our experiment using several



(a) Energy per Transaction



(b) Transaction Failure Rate

Figure 9: System Instability During Training

Model	Time on CPU (μs)	Time on GPU (μs)
256-128-128-64-64	6472.12	710.20
256-128-64	881.08	593.99
128-64	581.98	511.68
64	381.24	460.40

Table 3: Time per Frequency Recommendation

different networks. Each evaluated network has an input layer of 51 neurons (determined by our state representation), an output layer of 5 neurons (corresponding to our five actions), and some fully connected hidden layers. The networks vary in the number of hidden layers and number of neurons in each hidden layer.

Table 3 summarizes the results of these experiments. In the table, the various networks we tested are denoted by the number and size of their hidden layers. For example, the model "256-128-64" has three hidden layers with 256, 128, and 64 neurons respectively.

Although this experiment measures the total time required to generate a frequency recommendation, including state encoding, model inference, and sampling an action from the resulting probability distribution, almost all of the time is attributable to model inference. To put the times reported in Table 3 into perspective, they can be compared to the TPC-C transaction execution times reported in Table 1, which are mostly in a range from about 0.5ms-5ms, depending on the transaction type and frequency. A single forward pass through the 256-128-64-64 network, which is what we used in all of the performance experiments reported earlier in the paper, takes longer on the CPU than most TPC-C transactions, and our RL formulation adjusts frequency up to twice for every transaction - once when it arrives, and again when it begins execution. The GPU can reduce the model time considerably, but it is still high. Worse, the goal of frequency governance is to reduce energy consumption, but given that frequency recommendations take at least as much time as the transactions themselves, we expect that they would also consume as much power as those transactions - even more so if the recommendations are made using the GPU. Thus, although the learned frequency governors are effective, the cost of using them must be reduced significantly before this approach is practical.

We are currently exploring ways of reducing this cost. Our models are implemented with PyTorch, and they are large, so it is possible that we can obtain some improvement with a more efficient implementation of a smaller model. However, Table 3 shows that even very small models are still relatively expensive, so it is unlikely that this will be enough to achieve the order-of-magnitude overhead reductions needed to make this approach practical.

We are considering two other strategies. One is memoization. We can cache model output and reuse it when we observe a similar input state, rather then rerunning the model. However, since the model input includes the remaining time to deadline for each of the M transactions tracked by the model, it is unlikely that we'll find exact state matches in a reasonably sized cache. Thus, the challenge is to determine when input states are similar enough that a cached value can be used to choose a frequency.

A second strategy is to replace the learned model with an approximation that is much cheaper to evaluate, at the cost of some loss of fidelity. For example, instead of using the learned network directly, we learn a decision tree from model input/output pairs, and then use the decision tree instead of the model to make on-line frequency decisions. A similar approach has been proposed as way to make deep network models more explainable [6]. Here, though, the motivation is to reduce the cost of using the model.

7 CONCLUSION

Task scheduling and frequency governance are good settings for reinforcement learning, as these systems make frequent decisions for which reward feedback can be collected. In this work, we demonstrate that reinforcement learning can learn to manage CPU frequency for a transaction processing system as effectively as a state-of-the-art algorithm. The RL approach has the advantage that the resulting governor can be tuned to balance request latencies and power savings.

Our work also shows that there are some challenges that need to be addressed before RL can be practical in this setting. The first challenge is how to train a governor or scheduler when the initial "starter" model can push the system into instability. The second and perhaps more significant problem is the power and performance overhead of using learned deep models when decisions need to be made quickly and frequently. These problems are the subjects of our current work.

REFERENCES

[1] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2018. Placeto: Efficient progressive device placement optimization. In *NIPS Machine Learning for Systems Workshop*.

[2] Hakan Aydin, Rami Melhem, Daniel Mossé, and Pedro Mejía-Alvarez. 2004. Power-aware scheduling for periodic real-time tasks. *IEEE Trans. Comput.* 53, 5 (2004), 584–600.

[3] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. 2007. Speed Scaling to Manage Energy and Temperature. *J. ACM* 54, 1, Article 3 (March 2007), 39 pages.

[4] Zhuo Chen and Diana Marculescu. 2015. Distributed reinforcement learning for power limited many-core system performance optimization. In *2015 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1521–1526.

[5] Transaction Processing Performance Council. 1990. TPC benchmark C standard specification.

[6] Nicholas Frosst and Geoffrey E. Hinton. 2017. Distilling a Neural Network Into a Soft Decision Tree. *CoRR* abs/1711.09784 (2017). arXiv:1711.09784 http://arxiv.org/abs/1711.09784

[7] Yuanxiang Gao, Li Chen, and Baochun Li. 2018. Spotlight: Optimizing device placement for training deep neural networks. In *International Conference on Machine Learning*. PMLR, 1676–1684.

[8] Yaniv Gur, Dongsheng Yang, Frederik Stalschus, and Berthold Reinwald. 2021. Adaptive Multi-Model Reinforcement Learning for Online Database Tuning.. In *EDBT*. 439–444.

[9] Hui Huang, Man Lin, Laurence T Yang, and Qingchen Zhang. 2019. Autonomous power management with double-q reinforcement learning method. *IEEE Transactions on Industrial Informatics* 16, 3 (2019), 1938–1946.

[10] Hui Huang, Man Lin, and Qingchen Zhang. 2017. Double-Q learning-based DVFS for multi-core real-time systems. In *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 522–529.

[11] Nathan Jay, Noga Rotman, Brighten Godfrey, Michael Schapira, and Aviv Tamar. 2019. A deep reinforcement learning perspective on internet congestion control. In *International Conference on Machine Learning*. PMLR, 3050–3059.

[12] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*. 24–35.

[13] 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* (New York, NY, USA) *(MICRO-48)*. ACM, 598–610. https://doi.org/10.1145/2830772.2830797

[14] Mustafa Korkmaz, Martin Karsten, Kenneth Salem, and Semih Salihoglu. 2018. Workload-Aware CPU Performance Scaling for Transactional Database Systems. In *Proc. ACM SIGMOD*. 291–306.

[15] Mustafa Korkmaz, Alexey Karyakin, Martin Karsten, and Kenneth Salem. 2015. Towards Dynamic Green-Sizing for Database Servers. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS@VLDB*. 25–36.

[16] Sanjay Krishnan, Zongheng Yang, Ken Goldberg, Joseph Hellerstein, and Ion Stoica. 2018. Learning to optimize join queries with deep reinforcement learning. *arXiv preprint arXiv:1808.03196* (2018).

[17] Guoliang Li, Xuanhe Zhou, Shifu Li, and Bo Gao. 2019. Qtune: A query-aware database tuning system with deep reinforcement learning. *Proceedings of the VLDB Endowment* 12, 12 (2019), 2118–2130.

[18] Ning Liu, Zhe Li, Jielong Xu, Zhiyuan Xu, Sheng Lin, Qinru Qiu, Jian Tang, and Yanzhi Wang. 2017. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 372–382.

[19] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. 2014. Towards energy proportionality for large-scale latency-critical workloads. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 301–312.

[20] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM workshop on hot topics in networks*. 50–56.

[21] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2019. Learning scheduling algorithms for data processing clusters. In *Proceedings of the ACM Special Interest Group on Data Communication*. 270–288.

[22] Ryan Marcus, Parimarjan Negi, Hongzi Mao, Chi Zhang, Mohammad Alizadeh, Tim Kraska, Olga Papaemmanouil, and Nesime Tatbul. 2019. Neo: A learned query optimizer. *arXiv preprint arXiv:1904.03711* (2019).

[23] Ryan Marcus and Olga Papaemmanouil. 2018. Deep reinforcement learning for join order enumeration. In *Proceedings of the First International Workshop on Exploiting Artificial Intelligence Techniques for Data Management*. 1–4.

[24] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. 2018. A hierarchical model for device placement. In *International Conference on Learning Representations*.

[25] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. 2017. Device placement optimization with reinforcement learning. In *International Conference on Machine Learning*. PMLR, 2430–2439.

[26] Xiang Ni, Jing Li, Mo Yu, Wang Zhou, and Kun-Lung Wu. 2020. Generalizable Resource Allocation in Stream Processing via Deep Reinforcement Learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 857–864.

[27] Padmanabhan Pillai and Kang G Shin. 2001. Real-time dynamic voltage scaling for low-power embedded operating systems. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*. 89–102.

[28] Iraklis Psaroudakis, Thomas Kissinger, Danica Porobic, Thomas Ilsche, Erietta Liarou, Pınar Tözün, Anastasia Ailamaki, and Wolfgang Lehner. 2014. Dynamic Fine-grained Scheduling for Energy-efficient Main-memory Queries. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware* (New York, NY, USA) *(DaMoN '14)*. ACM, 1:1–1:7. https://doi.org/10.1145/2619228.2619229

[29] Rishad A Shafik, Sheng Yang, Anup Das, Luis A Maeda-Nunez, Geoff V Merrett, and Bashir M Al-Hashimi. 2015. Learning transfer-based adaptive energy minimization in embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 6 (2015), 877–890.

[30] Hao Shen, Ying Tan, Jun Lu, Qing Wu, and Qinru Qiu. 2013. Achieving autonomous power management using reinforcement learning. *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 18, 2 (2013), 1–32.

[31] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. 1999. Policy gradient methods for reinforcement

learning with function approximation.. In *NIPs*, Vol. 99. Citeseer, 1057–1063.

[32] Chen Tessler, Yuval Shpigelman, Gal Dalal, Amit Mandelbaum, Doron Haritan Kazakov, Benjamin Fuhrer, Gal Chechik, and Shie Mannor. 2021. Reinforcement Learning for Datacenter Congestion Control. *arXiv preprint arXiv:2102.09337* (2021).

[33] Zhongyuan Tian, Zhe Wang, Jiang Xu, Haoran Li, Peng Yang, and Rafael Kioji Vivas Maeda. 2018. Collaborative power management through knowledge sharing among multiple devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38, 7 (2018), 1203–1215.

[34] Fakhruddin Muhammad Mahbub ul Islam and Man Lin. 2015. Hybrid DVFS scheduling for real-time systems based on reinforcement learning. *IEEE Systems Journal* 11, 2 (2015), 931–940.

[35] Fakhruddin Muhammad Mahbub ul Islam, Man Lin, Laurence T Yang, and Kim-Kwang Raymond Choo. 2018. Task aware hybrid DVFS for multi-core real-time systems using machine learning. *Information Sciences* 433 (2018), 315–332.

[36] Zhe Wang, Zhongyuan Tian, Jiang Xu, Rafael KV Maeda, Haoran Li, Peng Yang, Zhehui Wang, Luan HK Duong, Zhifei Wang, and Xuanqi Chen. 2017. Modular reinforcement learning for self-adaptive energy efficiency optimization in multicore system. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 684–689.

[37] Zichen Xu, Yi-Cheng Tu, and Xiaorui Wang. 2012. PET: reducing database energy cost via query optimization. *Proceedings of the VLDB Endowment* 5, 12 (2012), 1954–1957.

[38] Zichen Xu, Xiaorui Wang, and Yi cheng Tu. 2013. Power-Aware Throughput Control for Database Management Systems. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*. 315–324.

[39] F. Yao, A. Demers, and S. Shenker. 1995. A Scheduling Model for Reduced CPU Energy. In *Symposium on Foundations fo Computer Science*.

[40] Ji Zhang, Yu Liu, Ke Zhou, Guoliang Li, Zhili Xiao, Bin Cheng, Jiashu Xing, Yangtao Wang, Tianheng Cheng, Li Liu, et al. 2019. An end-to-end automatic cloud database tuning system using deep reinforcement learning. In *Proceedings of the 2019 International Conference on Management of Data*. 415–432.