Learning Energy-Aware Transaction Scheduling in Database Systems

by

Udhav Sethi

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

Author's Declaration

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 execution frequency. 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 thesis, 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 results show that governors learned using our technique are competitive with state-of-the-art methods, and are able to adapt to a variety of workload conditions. We also show that our method has an added advantage - it allows flexibility in tuning frequency governance to balance a power-performance trade-off. Finally, we discuss the challenges associated with using RL in this setting due to the overheads of using a learned frequency governor.

Acknowledgements

I would like to express my sincere gratitude to my advisor, Prof. Ken Salem, whose continuous support and guidance made this thesis possible. His encouragement and insightful feedback steered me through this research. Working with him has been a privilege.

I would like to thank my committee members, Prof. Semih Salihoglu and Prof. Jimmy Lin, for taking the time to read and review this thesis.

I am grateful to my friend Harman for his assistance and stimulating discussion sessions that kept me motivated through the process. I would also like to acknowledge my friends Shubham, Tarpit, and Anup, who provided happy distractions to rest my mind outside of my work.

Finally, I would like to thank my family: my parents and my sister Pratula, for their unconditional love and support and for always believing in me.

Dedication

This thesis is dedicated to my loving parents...

Table of Contents

List of Figures

List of Tables

Chapter 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, the processor's frequency, and hence its 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 [36]. 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 P-States (or 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 [40, 41, 21, 30, 15, 17, 16] has focused on managing DVFS above the operating system, e.g, in the DBMS, where units of work and their performance requirements are understood.

In this thesis, 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 [16] 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 thesis 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 the power and performance overheads of using a learning approach for frequency governance.

Chapter 2

Background

In this chapter, we provide an overview of the energy-aware scheduling problem that we will be considering in this thesis. We discuss the problem setting in Section 2.1, followed by a description of the scheduling objective in Section 2.2.

2.1 Online Transaction Processing

We consider a database server running a DBMS. Short On-Line Transaction Processing (OLTP) style requests arrive at the DBMS from clients. There is a fixed number of transaction types, and clients submit requests to execute transactions of different types. Each transaction request arrives with a client-specified quality-of-service requirement, in the form of a soft execution deadline.

Internally, the DBMS routes requests to a set of workers, each with an associated transaction request queue. The worker's assigned requests are queued up in its request queue, from where they are picked up for execution 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. An example OLTP system with four workers is illustrated in Figure 2.1.

2.2 DVFS and Energy Aware Scheduling

There exists a power/performance tradeoff in the CPU, which can be controlled using DVFS. The DBMS workers can use DVFS to adjust CPU frequencies quickly - these ad-

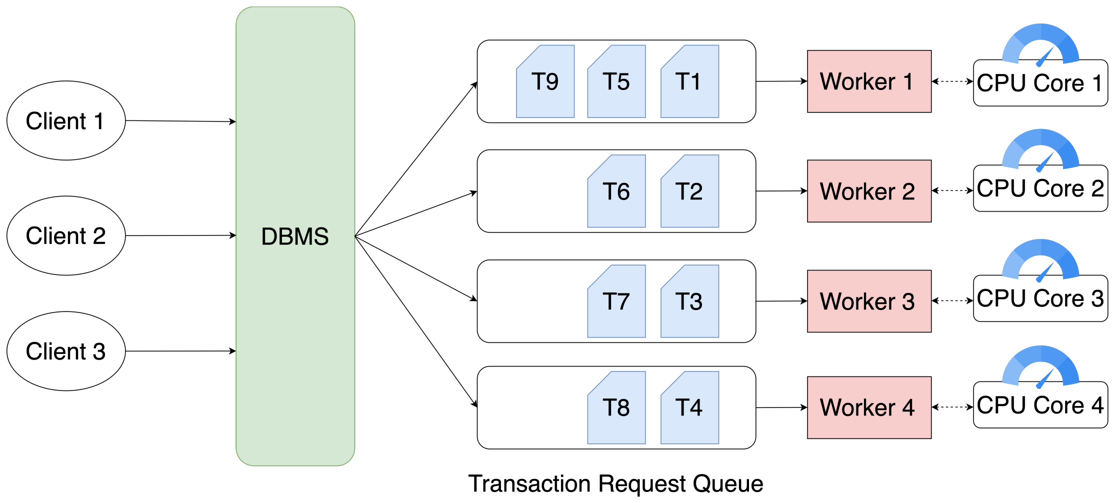


Figure 2.1: OLTP system

justments can be made at the time scale of individual transaction requests without introducing significant overheads. As expected, request latencies are lower at higher execution frequencies, and vice versa.

Dynamic power consumption in CPUs is typically modelled as proportional to f^{α}, where f is the execution frequency of the CPU, and $1 < \alpha \leq 3$ [4, 9, 38]. Since $\alpha > 1$, CPU power-performance relationship is convex. This means that higher the frequency of execution, higher the power consumed by the processor per unit of work that it completes.

Energy-aware scheduling aims to leverage the transaction latency budget to fulfil a two-fold objective - maximizing power savings while ensuring that the transaction latency targets are met. 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 use DVFS to dynamically adjust the execution frequencies of the server's CPU cores.

In this thesis, we focus on the problem of determining 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.

Frequency Governance Problem We define the frequency governance problem as follows: Given a sequence of transaction requests arriving online, each with an associated transaction type and an execution deadline, the objective is to adjust the processor frequency over time in order to reduce energy consumption while avoiding missed transaction deadlines.

Chapter 3

Related Work

3.1 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 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) [42], 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 [16] is a 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 [15] and PEGASUS [21] 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. It assumes a single-class workload, which means that all requests represent comparable units of work and thus have similar execution times.

Rubik predicts the execution time distributions of the queued transactions, and uses these estimates to set the execution frequency so that transaction latency targets are met. PEGASUS is intended to manage a cluster of servers running a single-class workload. It uses a control-theoretic approach to adjust CPU execution speeds across the cluster as the offered load fluctuates. Both Rubik and PEGASUS use a single transaction response time as reference, and therefore depend on the assumption that all requests are of the same type. As with POLARIS, our proposed method can accommodate any number of request types.

3.2 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 [43, 19, 10], join query optimization [25, 18, 24], and datacenter congestion control [13, 34]. RL has also been extensively applied to resource management on distributed compute clusters. DeepRM [22] uses RL for resource scheduling in an on-line non-preemptive setting. It considers jobs with multi-dimensional resource profiles, and assumes that the resource demand of each job is known upon arrival. The scheduling objective in DeepRM is to minimize the average job slowdown. Decima [23] uses a similar approach to learn workload-specific scheduling policies for DAG-structured jobs to minimize completion latencies. A number of other methods address the resource allocation inside jobs' computation graphs [27, 26, 8, 1, 28]. Liu et al [20] propose an RL-based hierarchical framework to address the overall resource allocation and power management problem in cloud computing systems.

RL-based approaches have also been proposed for on-line energy efficiency optimization through DVFS management. Some key differentiators of our proposed approach compared to these methods are summarized in Table 3.1. 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 [37, 12]. For instance, Islam et al [37] present an approach that manages CPU frequencies by switching between three real-time DVFS techniques: cycle-conserving (CC), look-ahead (LA) [2], and dynamic reclaiming algorithm (DRA) [29]. These techniques differ in the strategies they employ to save power. For example, the CC algorithm initially starts with a high CPU frequency and gradually decreases it when tasks complete their execution. In contrast, the LA algorithm starts with a lower frequency and tries to defer as much work as possible until the deadline. Another body of work targets learning to govern frequency directly, rather than choosing among existing techniques. Learned frequency governors have been proposed for both single core [32], and multi-core settings [31, 5, 39, 35]. These governors operate at the os-level and consider workloads consisting of either a single application, or a series of applications with different performance requirements. For example, Shafik et al [31] consider applications consisting of a series of periodic tasks, such as the MiBench benchmark [11].

Some approaches make their decisions without taking advantage of application-level workload information. They use low-level metrics for workload characterization, such as memory accesses, the number of CPU cycles executed, or cache misses suffered. Others characterize the workload using units of work in the application. Tian et al [35] consider a workload consisting of single-class application tasks with a user-defined performance requirement per instance. Islam et al [37] assume a periodic real-time task model, in which a task is repeatedly executed at regular intervals, and is assigned a deadline equal to its period. In this work, we characterize the workload based on transactions in an OLTP system. We assume that tasks belong to different classes (types), and each task has an associated soft deadline. The tasks are not periodic and can randomly arrive at any point in time.

The various studies formalize the objective of energy-efficiency optimization in different ways. Wang et al [39] define the objective as maximizing throughput per energy consumption, where throughput is measured in terms of the number of busy CPU cycles per unit time. Shen et al [32] measure performance by the deviation from the maximum frequency, and energy consumption by the deviation from the energy-optimal frequency. In some cases, the objective is defined as minimizing power consumption for a given performance constraint. Performance constraints are often specified using quality-of-service requirements of the running application. Tian et al [35], for example, measure performance in terms of the average execution time per application iteration. With the hard real-time periodic task model [37, 12], the objective is to save power without missing any deadlines. In our case, it is possible for the system to be overwhelmed with arrivals, due to which tasks may fail to meet their deadlines. Accordingly, our objective is to balance the trade-off between energy consumption and task failures.

The various studies also differ in their choice of interval between DVFS control decisions. Some scale CPU frequencies at fixed intervals, with manually configured period lengths, generally in the order of several milliseconds [32, 5]. Others trigger the DVFS controller after finishing a specific unit input data-block [31, 39, 35]. Islam et al [37] invoke DVFS on multiple events: a task's release, completion, preemption, or dispatch. In this work, DVFS decisions are made up to twice for every application task.

Method	Workload Characterization	Objective	DVFS trigger period
Shen et al [32]	clock frequency, temperature, instructions per second (IPS), CPU intensiveness	minimize performance deviation from maximum frequency, minimize energy deviation from energy-optimal frequency	every 20 ms
Shafik et al [31]	CPU cycles' count	minimize energy consumption, minimize deviation from average performance requirement per task	after each (periodic) application task
Chen et al [5]	Instruction Per Cycle (IPC), Million L2-cache-misses Per Kilo-Instructions (MPKI), current power, current frequency level	minimize energy budget overshoot, maximize throughput per over-the-budget energy	every 500μs to 10ms
Islam et al [37]	periodic real-time task model	minimize energy, meet hard deadlines	after every task release, completion, preemption, or dispatch
Wang et al [39]	core throughput (busy-cycle-count per unit time), CPU intensiveness	maximize throughput per energy consumption	after each application task/iteration
Huang et al [12]	periodic real-time task model	minimize energy, meet hard deadlines	least common multiple of task periods
Tian et al [35]	single-class workload model	minimize energy consumption, minimize deviation from average performance requirement per application task	after each application task/iteration
Proposed	multi-class workload model with probability distribution over classes	minimize energy consumption, minimize transaction failures	after each task arrival or completion

Table 3.1: Related work summary - Learning to Schedule

Chapter 4

Design

In this chapter, we describe our design for an energy-aware scheduling approach using reinforcement learning (RL). We outline the general RL framework in Section 4.1 and describe how to fit the frequency governance problem into this framework in Section 4.2. We then describe our training mechanism in Section 4.3. Table 4.1 summarizes some of the notation we use in this chapter to describe our design.

4.1 Reinforcement Learning

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 4.1.

The agent uses a policy π to pick its actions. Through its interaction with the environment, the agent alters the policy with the goal of maximizing the (expected) cumulative *discounted* reward that it will receive from the environment over the long run:

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

where $0 \leq \gamma < 1$ is the discount rate, a parameter that determines the present value of future rewards.

Notation	Description
π	agent's policy
θ	policy parameters
M	transactions in state
N	number of frequencies supported by processor
$\mathcal{T}$	set of transaction types
γ	reward discount
δ	failure reward weight
T	timesteps/episode
η	REINFORCE step size
f_t	processor frequency in timestep t
L_t	length of timestep t
F	maximum frequency from available processor frequencies
E_{avg}	average energy consumed per transaction when executing at F

Table 4.1: Summary of notation

4.2 RL Formulation - Frequency Governance

We formulate frequency governance as an RL problem. The various elements of our RL formulation are illustrated in Figure 4.2 and formalized as follows.

4.2.1 Environment

In our problem setting, 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 execution frequencies. The clients generate transaction execution requests, submit them to the database system, and await responses. Each transaction has a type (from among a set of possible transaction types $\mathcal{T}$) and a latency target. The submitted transactions are queued in a transaction queue. The database system's worker picks transactions for execution in EDF (Earliest Deadline First) order. It runs a single transaction at a time in a non-preemptive fashion.

We define environmental state transitions to occur as a result of either of two events - arrival of a new transaction request in the queue, or completion of the currently running transaction. These events divide time into a series of steps, which may differ in length. In

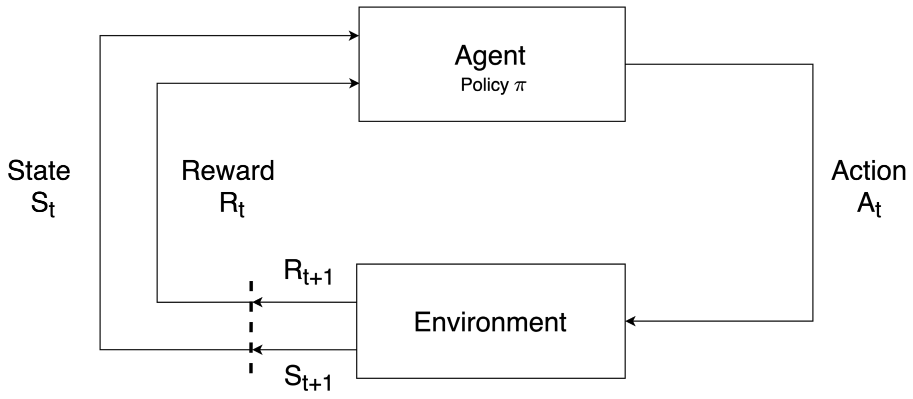


Figure 4.1: Agent-environment interaction in reinforcement learning

Figure 4.3, 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 4.3, 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.

4.2.2 Agent

The agent is responsible for observing the state of the system and making a decision on the processor's execution frequency. It uses a policy to make these decisions. In this work, we consider so-called deep reinforcement learning. We represent the agent's policy π 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 the system state and a is one of the possible actions. The agent picks an action by randomly selecting one according to the probability distribution defined by the policy.

4.2.3 State Space

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. Since the agent requires a fixed-size state representation as input for its policy network, it captures only the M transactions with the earliest deadlines from the current transaction queue.

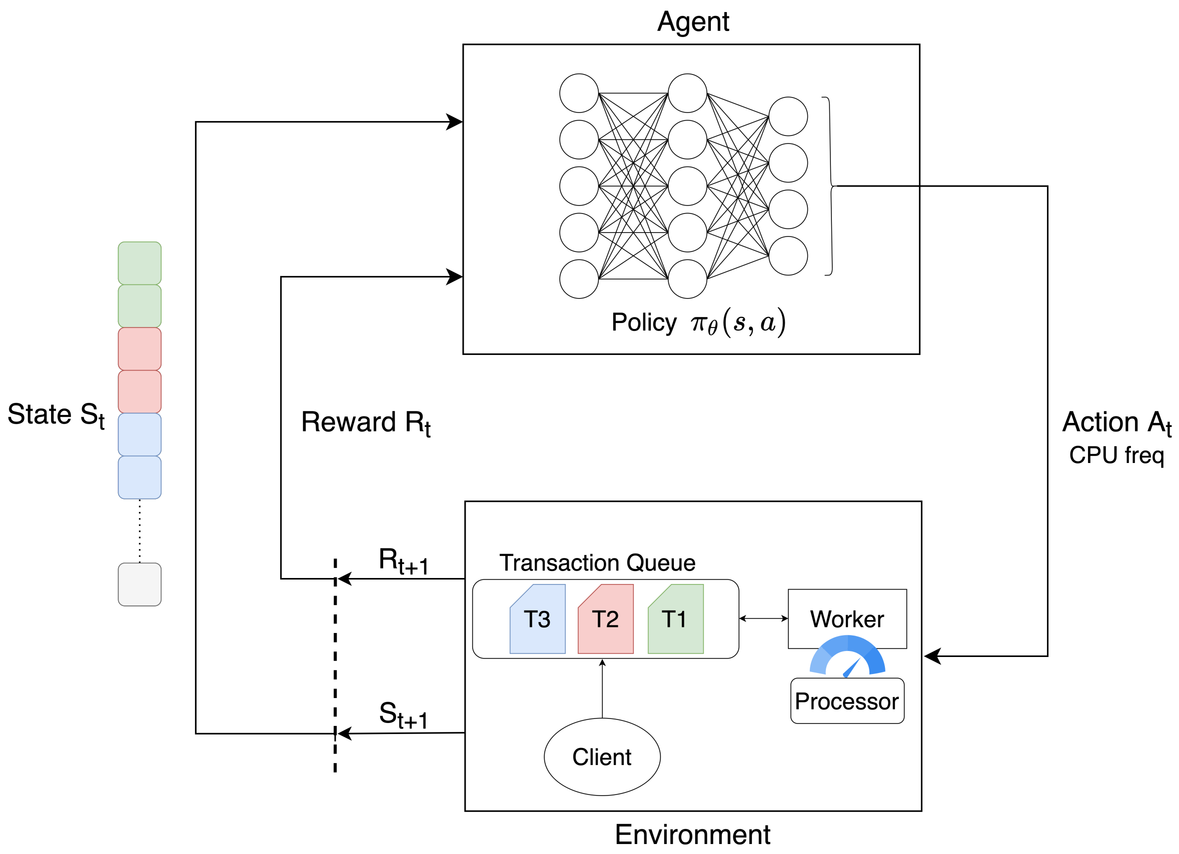


Figure 4.2: RL Formulation of frequency governance problem

Specifically, for each of these M transactions, the state includes the following:

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

The agent captures the earliest-deadline transactions because transactions are executed by the database system in EDF order. Thus, the transactions captured in the state represent the queued transactions that will be executed soonest.

In addition to information about the M earliest deadline transactions, 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.

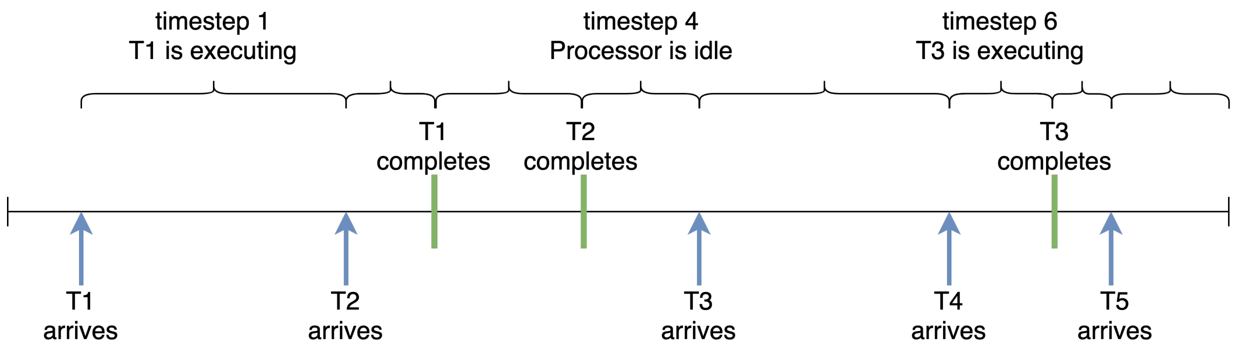


Figure 4.3: Time divided into variable length timesteps

4.2.4 Action Space

At any point in time, the agent chooses an execution frequency from amongst the frequencies at which the processor can run. In this setting, the action space is defined by a set of N possible target CPU frequency levels. It is given by $\{1, 2, ..., N\}$ where $A = i$ means "run the CPU at frequency level i".

4.2.5 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 $R_{failure}$, which capture these two goals respectively.

The total reward assigned to the agent for an interval t, R^t, depends on whether the currently running 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 total discounted reward, which corresponds to minimizing the penalties.

Next, we describe how R^t_{energy} and $R^t_{failure}$ are determined. The R_{energy} component of the reward function represents the energy consumed by the processor during timestep t.

We set

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

where f_t is the CPU frequency chosen by the agent for timestep t, and L_t is the length of the timestep t and α is a constant. As discussed in Section 2.1, f_t^{α} models the frequency-dependent dynamic power consumption of CPUs, with $1 < \alpha \leq 3$ (In our formulation, we set $\alpha = 2$). Multiplying by the length of the interval then models the energy consumed during the interval. The agent is assigned a higher energy penalty for picking a higher CPU frequency since it leads to greater energy consumption as compared to lower frequencies.

The $R_{failure}$ component of the reward function promotes the completion of transactions before their deadlines. $R_{failure}$ for a timestep t is defined as follows:

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

Here, f_t is the CPU frequency chosen by the agent for timestep t, F is the maximum frequency from among the supported frequencies in the action space, E_{avg} is the average energy consumed per transaction when executing at F, 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:

1. δE_{avg} expresses the penalty for failure as the average energy consumption of a transaction at F multiplied by a scaling factor (δ). As the value of δ increases, the magnitude of $R_{failure}$ increases, and transaction failures become more expensive, which makes it less appealing for the agent to miss deadlines in favour of saving 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. This is explained as follows: Suppose that the failure rate (percentage of transactions that miss their deadlines) when the processor runs at F all the time is λ. An $X\%$ additive increase in failure rate (i.e., changing failure rate from λ to $\lambda + X$), would require at least an $X * \delta$ percent decrease in the energy reward, in order to reduce the overall penalty. For example, if λ=0.02 (2% failure rate) and δ=5, then a failure rate of 5% (a 3% additive increase, so X=0.03) will demand a 15% decrease in energy consumption as compensation.

2. $\frac{F}{f_t}$ ensures that the failure penalty is inversely proportional to the CPU frequency in the timestep where the transaction failure occurs. The effect of this term is to make transaction failure more expensive the slower the processor is running when the transaction fails. This encourages the agent to shift towards higher frequencies when faced with greater loads and failure rates.

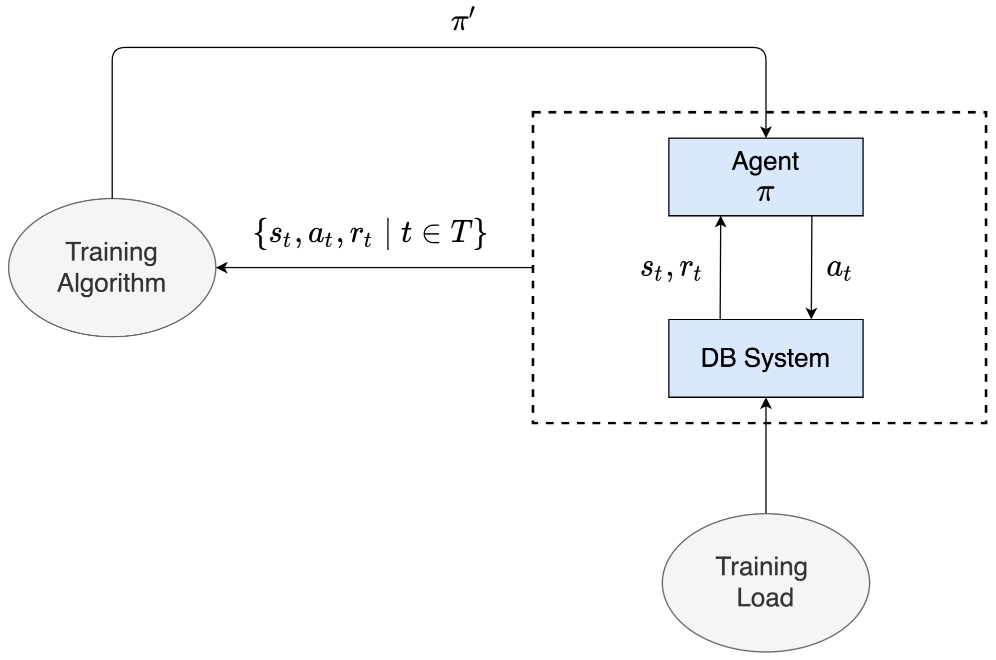


Figure 4.4: Training Loop

4.3 Training

We train the agent's policy using the training loop illustrated in Figure 4.4. 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 an 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 [33], 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 the return 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 [33]:

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

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

Chapter 5

Transaction Simulation

Our method is intended to control the speed of transaction execution in a real OLTP system. However, to make it possible to quickly explore a variety of RL problem formulations and generate ample data for training, we use a simple discrete event simulation of the transaction execution system, rather than running actual transactions against a database.

Our simulator simulates transaction execution with a single processor and a transaction request queue. It consists of three main components - a workload generator, a worker, and a scheduler. It is illustrated in Figure 5.1.

5.1 Generator

The generator is responsible for generating a transaction workload in the simulation. It simulates clients that submit requests to the database for processing. The generator loop wakes up, generates a transaction, pushes it to the request queue, and sleeps for some time interval *think time*. This loop continues to run for the entire duration of the simulation. The generated transactions await execution in the request queue, from where they are picked up one at a time by the worker.

The think time between two requests is exponentially distributed around a mean value, determined by the generator's *utilization* parameter. The utilization can be configured to control the rate of transaction generation, and thus the average load on the system. For example, setting utilization = 0.4 would result in a transaction arrival rate of 40% of the maximum arrival rate that the simulated processor can sustain at maximum frequency.

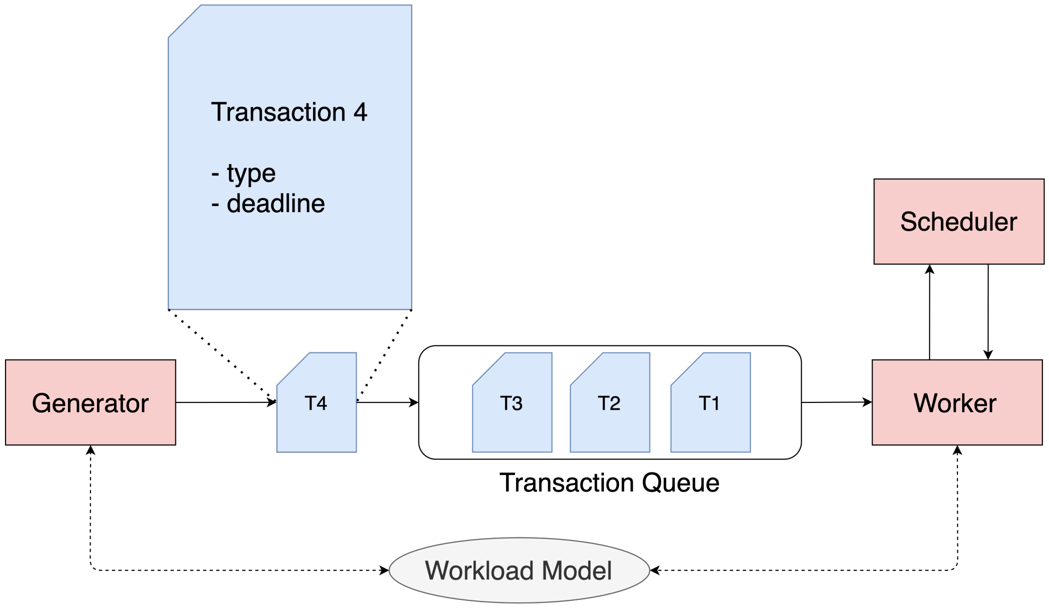


Figure 5.1: Transaction Execution Simulator

Each generated transaction has a type and a latency target associated with it. The type of a transaction describes its “size”, and so is an indication of its expected execution time at a given CPU frequency. The latency target for a transaction is determined using its type and the notion of *slack*, a measure of the tightness of transaction deadlines. We define slack as the ratio of a transaction’s latency target to its mean execution time at maximum frequency. Each transaction is assigned a deadline by adding its latency target to its generation timestamp. The generator generates a mix of transaction types, using a probability distribution over the types. This probability distribution is given by a *workload model*.

Workload Model The workload model serves two purposes - first, it specifies the likelihood of each type of transaction (used by the generator), and second, it gives the execution time of each type of transaction at each CPU frequency (used by the worker). For example, a simple workload model may specify two transaction types, each with a generation probability of 50%, and with associated execution times at the various frequencies supported by the CPU.

5.2 Worker

The worker simulates a DBMS worker and "executes" the transaction requests generated by the generator. If the request queue is not empty, it picks the transaction with the earliest deadline, and runs it to completion in a sequence of *bursts*. Each burst corresponds to one timestep as described in Section 4.2.1. A burst begins when the chosen transaction is initiated and ends with either its completion, or when a new request arrives in the queue. If execution of the running transaction is interrupted by an arrival, it is resumed in the following burst. If the burst ends with completion of the running transaction, a new transaction is chosen from the queue in EDF order, which then begins execution in a new sequence of bursts. However, if the queue is empty, there is nothing for the worker to run and so it waits idly until a new request arrives in the queue.

The worker contacts the scheduler at the beginning of each burst to determine the processor frequency to be used during the burst. For this purpose, it passes a representation of the current state of the system to the scheduler. It uses the transaction queue to construct this state representation as described in Section 4.2.3. This construction also involves some preprocessing steps: the transaction type, which is a categorical feature, is encoded as a one-hot numeric array, the remaining time until deadline is scaled by a fixed constant, and the backlog is scaled and clipped at a maximum value.

On receiving the scheduler's choice of execution frequency, the worker determines the execution time for the chosen transaction using the workload model described in Section 5.1. If a previously interrupted transaction is being resumed, the output of the workload model is scaled to compute the remaining execution time.

As the worker runs, it produces output after every burst. The output indicates whether a transaction completed in the burst, and if so whether the completed transaction finished within its deadline. The worker also reports the energy consumed by the simulated processor during the burst using the energy model in Section 2.1, with $\alpha = 2$. Energy consumption is proportional to $L_t f_t^2$, where L_t is the length of the burst, and f_t is the processor's frequency during the burst. For simplicity, we consider the energy consumed by the simulated processor to be 0 when the worker is idle.

5.3 Scheduler

The scheduler is a framework which can be used to plug in various algorithms for frequency governance. The frequency governors used in our experiments are described in Chapter 6. The scheduler supports both baseline governors and governors learned using our method.

When using a learned governor, the scheduler infers the execution frequency using the queue representation provided by the worker. It makes these inferences using a policy neural network as discussed in Section 4.2.2. While training a learned governor, the parameters of the policy model are updated at the end of each epoch. These parameters are saved as “checkpoints” at regular intervals of training iterations. During evaluation, the scheduler loads weights from a saved checkpoint and uses them for inference.

Chapter 6

Evaluation

We conduct an evaluation of our reinforcement learning method in order to address the following questions:

- Can our method learn policies that can manage the power/performance trade-off effectively?
- Can our method achieve power savings comparable to those achieved by state-of-the-art POLARIS?
- Can the learned policies perform well under a variety of load conditions, as would be expected in a real database system?
- How quickly can the policies be learned?
- Are there any advantages of using learned frequency governors?
- What are the performance and power overheads of using deep RL for scheduling?

6.1 Methodology

In each experiment, we pick a specific frequency governor to test at a specific workload. We allow the generator to generate a series of transactions at at a specified utilization, which determines the request arrival rate. The transactions are executed by the worker at CPU frequencies chosen by the frequency governor under test. The governor chooses

Transaction Type	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 6.1: Transaction execution times at various CPU frequencies and assigned latency targets. Percentages indicate the transaction mix in the workload.

from among the five execution frequencies supported by the simulated processor: 1.2 GHz, 1.6 GHz, 2.0 GHz, 2.4 GHz, and 2.8 GHz. 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. An experiment ends when the system has run for a fixed number of (non-idle) time steps. Each experiment is repeated 100 times and the metrics of interest are recorded for each run. For each of these metrics, we report the mean value across all runs, along with a 95% confidence interval (shown in the form of error bars on the result plots).

Workloads The simulated workloads comprise of a mix of four transaction types from the TPC-C OLTP benchmark [6], with an associated probability distribution: New Order (45%), Payment (47%), Order Status (4%), and Stock Level (4%). In our initial experiments, we assume that transaction execution times are fixed and depend only on the transaction type and processor frequency. That is, there is no variability in execution time among transactions of the same type unless execution frequency changes. Table 6.1 shows the runtimes used by our transaction simulator for each of the four TPC-C transaction types at each of the five available processor frequencies. These correspond to the mean TPC-C transaction execution times measured by Korkmaz et al [16] against an in-memory database in Shore-MT [14]. We assign each transaction with a type-dependent latency target, which is determined using a slack of 10 relative to its execution time at maximum frequency (see Latency Target column in Table 6.1). For example, for a Stock Level transaction, which has an average execution time of 3435 μs at the highest frequency level, the latency target is set to 34350 μs (Note that the parameters shown in Table 6.1 are known to the simulator, but are are not part of the state representation used for learning).

We conduct our evaluation using both static and dynamic workloads. Static workloads maintain a fixed transaction arrival rate during the entire experimental run. We consider four different static workloads at 20%, 40%, 60% and 80% utilization. Dynamic workloads include fluctuations in the arrival rate during the course of each experimental run. To generate a dynamic workload, we varied the arrival rate through eight *phases* of equal length during each experimental run. In each phase, the arrival rate is fixed at one of the four static utilizations, but successive phases use different utilizations. We used a "ramp up, ramp down" pattern, with successive phases at 20%, 40%, 60%, 80%, 80%, 60%, 40%, 20% utilization.

Frequency Governors 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. All of the governors execute transactions in EDF order.

1. Non-learned baseline governors
   (a) fixed-frequency governors
      i. f_{min}: runs all transactions at the lowest processor frequency (1.2 GHz)
      ii. f_{max}: runs all transactions at the highest processor frequency (2.8 GHz)
   (b) POLARIS (implemented in our simulator)
2. Learned governors
   (a) m20, m40, m60, m80: use policy models trained with static workloads at 20%, 40%, 60% and 80% utilization respectively.
   (b) mcross: uses a policy model trained with the dynamic workload.

Metrics We evaluate the frequency governors on two key metrics - energy consumption per transaction (total energy consumed per number of completed transactions) and failure rate (number of transactions that missed their deadlines per number of completed transactions). The two metrics respectively capture the performance of our model in minimizing power consumption and achieving transaction latency targets. The reported energy values are calibrated to match the power measurements for TPC-C workloads in Shore-MT as reported by Korkmaz et al [16].

Parameter	Value	Description
M	10	transactions in state
N	5	number of frequencies supported by processor
γ	0.99	reward discount
δ	3	failure reward weight
T	5000	timesteps/episode
η	0.0001	REINFORCE step size
	8000	number of training epochs
	10	scheduling slack

Table 6.2: Simulation and Learning Parameters

Training of Learned Frequency Governors Each of the learned frequency governors need to be trained before they can be used for evaluation. We built the policy neural network used in the learned governors using 5 fully connected hidden layers with 256, 128, 128, 64, and 64 neurons respectively. We trained the governors using the mechanism described in Section 4.3. The values of other simulation and learning parameters are as shown in Table 6.2 unless stated otherwise.

6.2 Results

6.2.1 Fixed execution times and Fixed Load

To evaluate whether our method can learn policies that can manage the processor frequency effectively, we begin by considering the frequency governance problem in a simple setting. Specifically, we make two simplifying assumptions, both of which will be relaxed in subsequent experiments:

1. We assume that there is no variability in execution time, i.e., all transactions of a given type and running at the same frequency will finish in the same amount of time. Specifically, we use a workload model that determines the execution time of a transaction of a given type at a given frequency using the values shown in Table 6.1.

2. We test our learned policy models at the same load intensity at which they are trained. For example, when testing at 40% utilization, we use the m40 learned model, which is trained at 40% utilization.

We compare the performance of the learned policies with POLARIS. We also run 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.

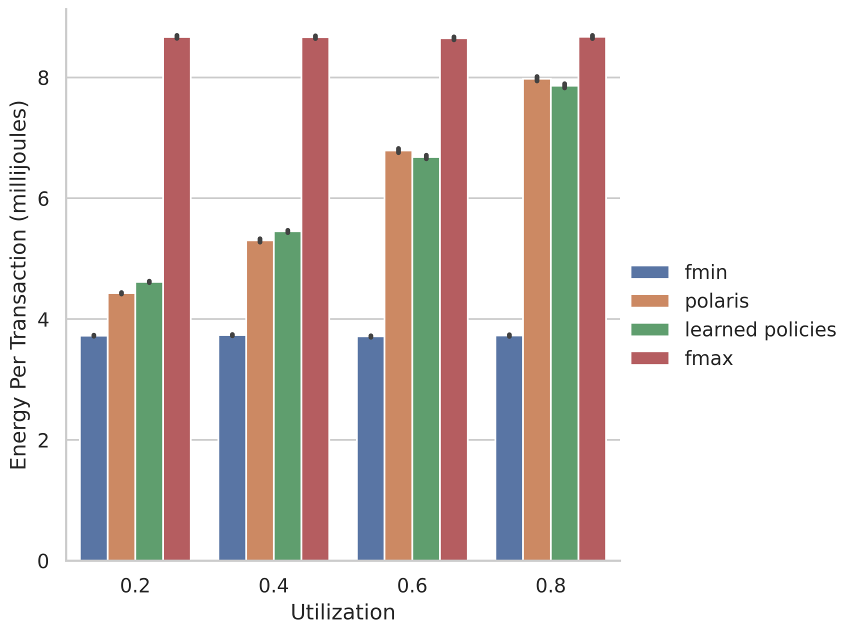
Figure 6.1 shows the top-level results from this experiment. Using f_{max} for scheduling in this setting causes the simulated processor to consume about 9 millijoules of energy per completed transaction. f_{max} achieves the lowest failure rates compared to other governors, with less than 4% transactions missing their latency targets at high utilization. In comparison, f_{min} saves approximately 5 millijoules of energy per transaction, but at the expense of substantially higher failure rates, approaching almost 100% at higher utilization levels.

The policies learned using our method appear to behave in accordance with the utilization at which they are trained. At low utilization, the learned policy offers energy savings of about 4 millijoules per transaction compared to f_{max}, but these savings do not come at the expense of a considerable increase in failure rate. At high utilization, the learned policy consumes much more energy and offers savings of only about 1 millijoule per transaction compared to f_{max}, and suffers a 5% additive increase in failures as a trade-off. Comparing our method to POLARIS, we find that in this simple setting, the learned policies offer failure rates and energy consumption similar to what POLARIS achieves, at all utilizations.

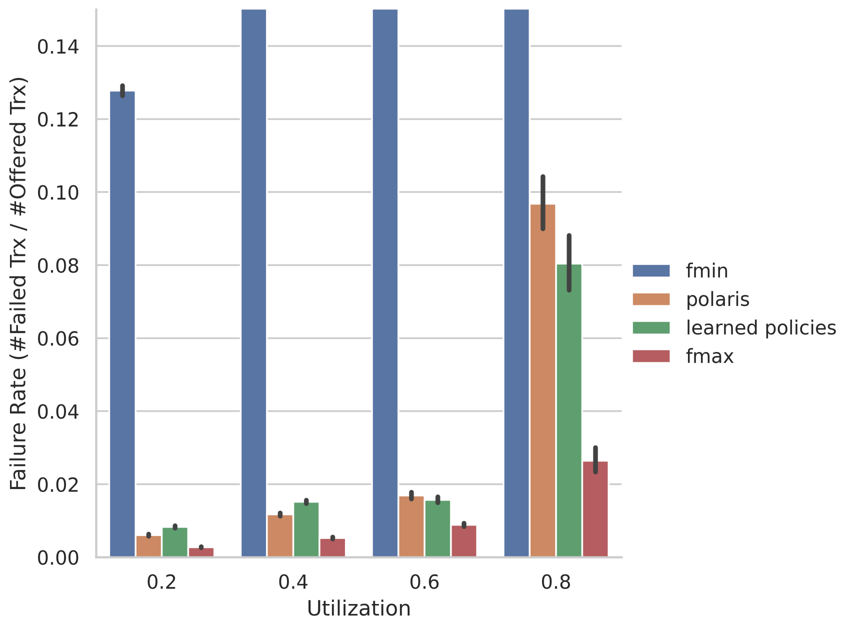
6.2.2 Randomized execution times and Fixed Load

In practice, transactions of the same type and running at the same frequency may not necessarily execute in the same amount of time. Execution times could vary due to factors such as transaction parameter values and contention with other transactions for access to the underlying database. To evaluate whether our learned policies can accommodate this variability, we relax one of the two assumptions made in the previous experiment by introducing randomness in transaction runtimes. Specifically, we switch to a different workload model, which determines the simulated execution time of a transaction of a given type at a given frequency randomly, using an exponential distribution with a mean given by the measured values in Table 6.1. For example, the execution time of Payment transactions at 2.0 GHz is determined using an exponential distribution with a mean of 517 μs.

Figure 6.2 shows the results for this experiment. While the energy measurements for the tested governors are similar to those of the experiment with fixed execution times, the failure rates are noticeably higher, likely due to the introduced variability in transaction execution times. Compared to f_{max}, the learned policies offer energy savings of 1-3 millijoules per transaction with comparable or slightly higher failure rates, depending on



(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 0.15 to more clearly distinguish the other governors.

Figure 6.1: Energy per transaction and transaction failure rates, with fixed (non-variable) transaction execution times. The learned policy tested at utilization u is trained at utilization u.

the test utilization. Overall, we observe that the learned policies are still able to achieve results comparable to POLARIS at all utilizations, indicating that our method is able to accommodate for the added randomization in transaction runtimes.

6.2.3 Training and Testing at Different Utilizations

Through the experiments described in Sections 6.2.1 and 6.2.2, we found that the learned governors show promise in their scheduling ability at the utilization where they are trained. This is the best case for the learned policies, since training and testing workloads are of comparable intensity. In practice, it would be best to have a single learned policy that would perform well at all load levels. Otherwise, some additional control mechanism would be needed to switch among learned governors as the load fluctuates. But, how should we train such a "load-universal" policy? 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 trained using the dynamic workload described in Section 6.1.

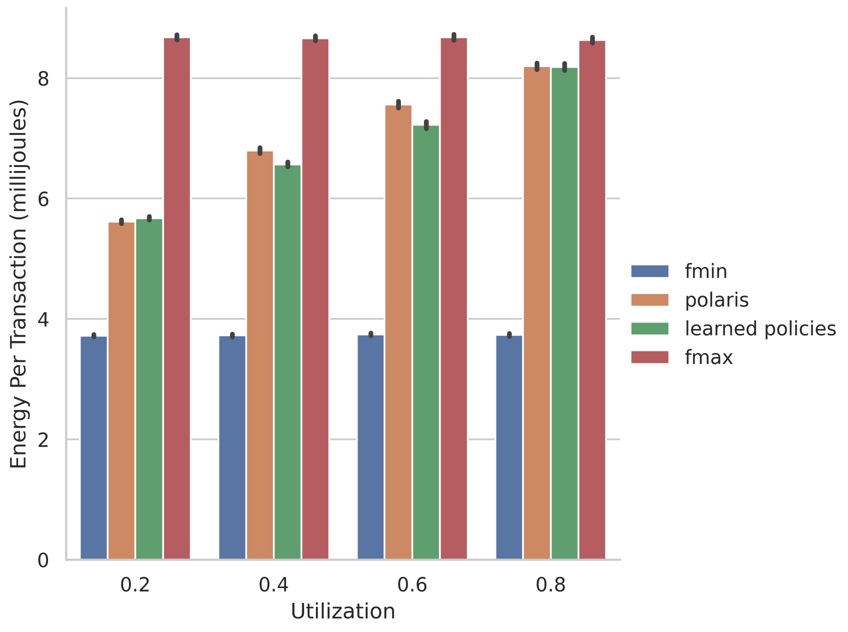
Figure 6.3 shows the results of this experiment. We find that the m80 model tends to result in lower failure rates and higher energy consumption than the models trained at other utilizations. However, the differences are not that large - we expected that a model trained at low load would perform poorly when tested at high load, and vice versa. Overall, the model trained with the dynamic workload, labeled "mcross" in Figure 6.3 seems to be a consistent middle-of-the-road performer.

6.2.4 Time-Varying Loads

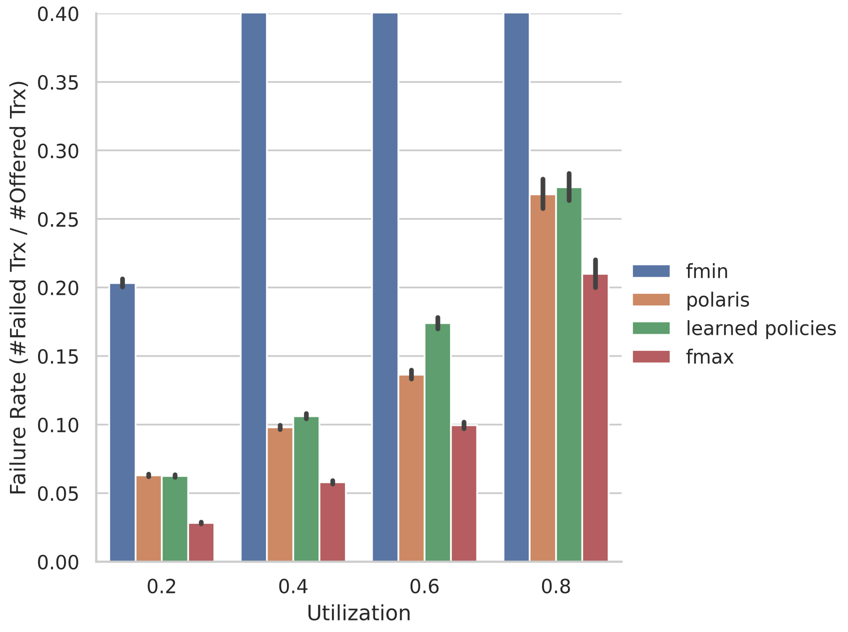
So far, we evaluated the learned frequency governors on static workloads, which have a steady transaction arrival rate with small fluctuations around a mean value. In this experiment, we assess the ability of our method in scheduling workloads in which the mean arrival rate varies significantly between low and high utilizations. We use the dynamic workload described in Section 6.1 for this purpose.

Figure 6.4 compares the average energy and failure rates offered by some of the learned policies and POLARIS on the dynamic workload. The policies deliver results which are competitive with POLARIS, indicating that they learn to handle not just workloads at various utilizations, but also "transition periods" of increasing or decreasing load.

To dig deeper into how a learned frequency governor responds to varying load, we monitor its behaviour *during* a test run on the dynamic workload. Figure 6.5a illustrates

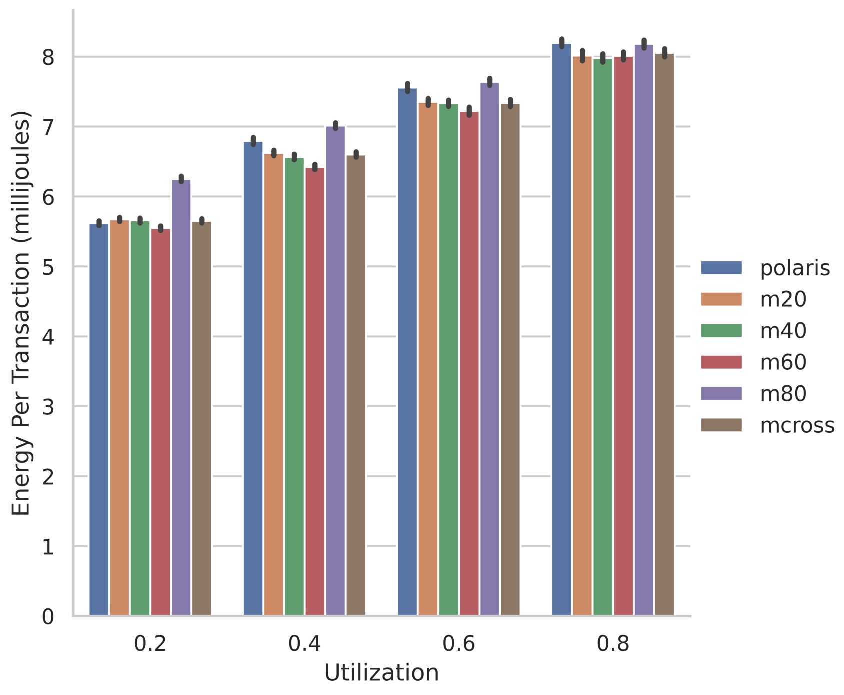


(a) Energy per transaction

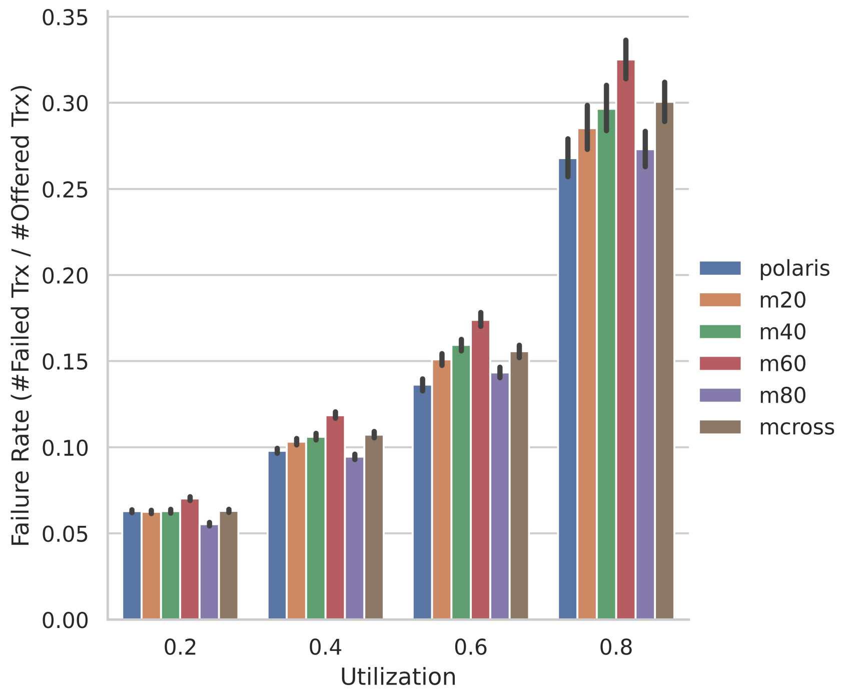


(b) Transaction failure rate. The failure rate for f_{min} approaches 1.0 at higher utilizations. The y-axis has been truncated at 0.4 to more clearly distinguish the other governors.

Figure 6.2: Energy per transaction and transaction failure rates, with randomized transaction execution times. The learned policy tested at utilization u is trained at utilization u.

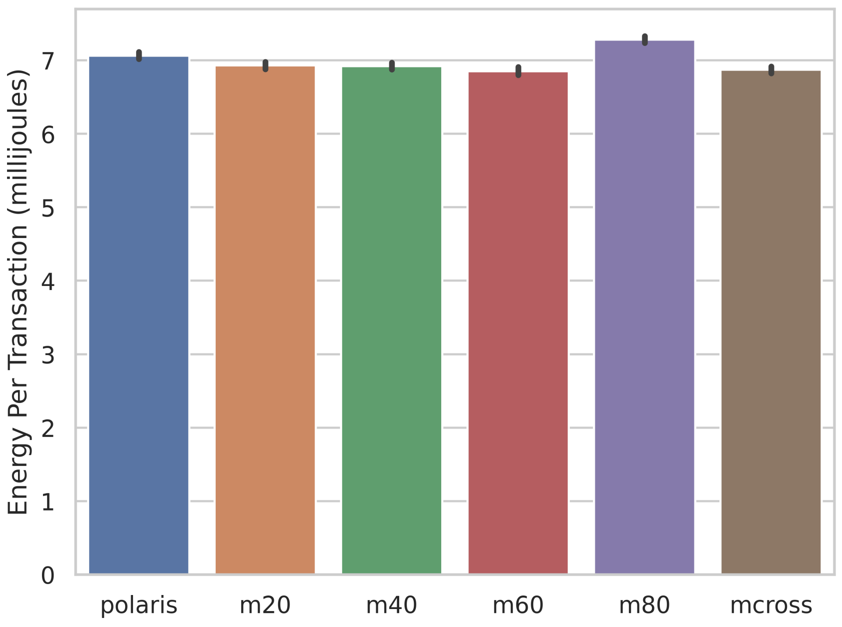


(a) Energy per transaction

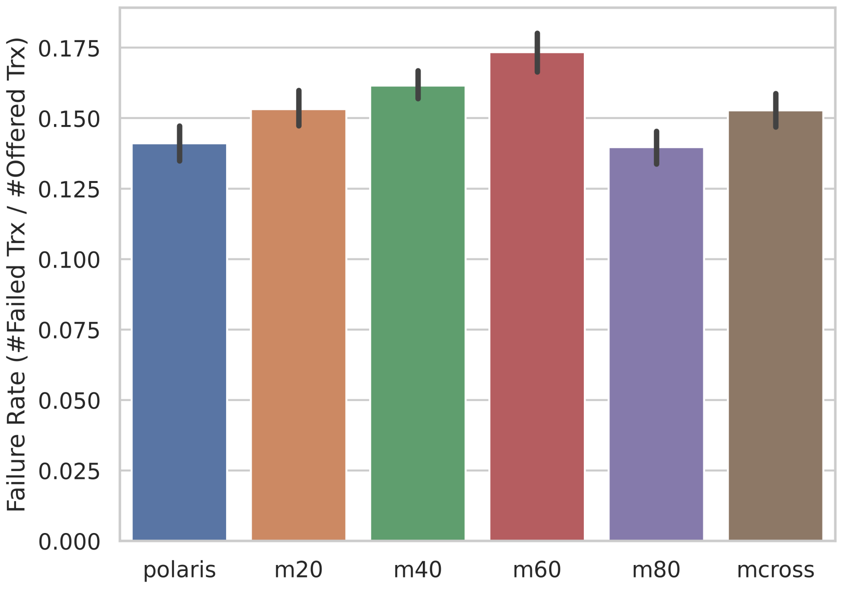


(b) Transaction failure rate

Figure 6.3: Energy per transaction and transaction failure rates, with randomized transaction execution times. Learned policies are tested at all utilizations.



(a) Energy per transaction



(b) Transaction failure rate

Figure 6.4: Energy per transaction and transaction failure rates with time-varying load

the variations in arrival rate during a dynamic workload run (with T=50000). Figures 6.5b and 6.5c respectively show the energy and failure rate offered by mcross as a function of time while scheduling the dynamic workload. We observe that as the request arrival rate increases or decreases, the energy consumed by the simulated processor follows the varying load. The policy uses periods of low utilization as opportunities to save energy, while burning more energy at high utilization in favour of meeting transaction latency targets. The figure also shows results for POLARIS, which exhibits similar behaviour in this setting. We notice that during periods of high utilization, both POLARIS and mcross observe a spike in transaction failure rate (with mcross suffering slightly higher number of failures than POLARIS). While this is non-ideal, we argue that some transaction failures are unavoidable even if the CPU is run at maximum frequency all the time. As load increases, the unavoidable failures become more probable, and hence the spike. We illustrate this effect by showing that a similar spike in failures is seen while using f_{max} for scheduling in this setting (see Figure 6.5c).

6.2.5 Scheduling Behaviour

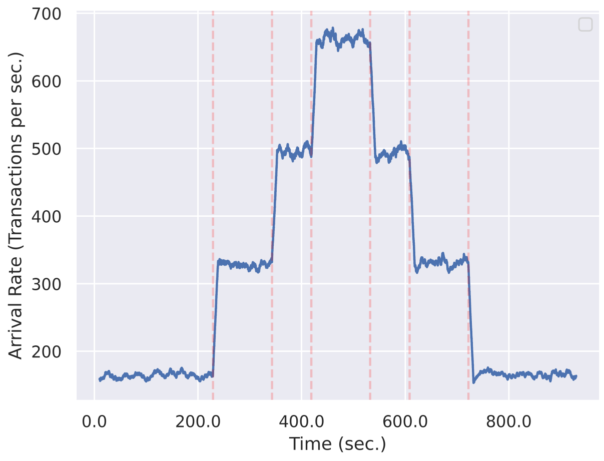
We found that while POLARIS and the learned governors have similar performance, they do differ in their scheduling behaviour. For each static workload utilization, we compared the *frequency residency distribution* of POLARIS with that of the learned policy trained at the same utilization (see Figure 6.6). 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, POLARIS makes some use of the intermediate frequency levels.

6.2.6 Training Efficiency

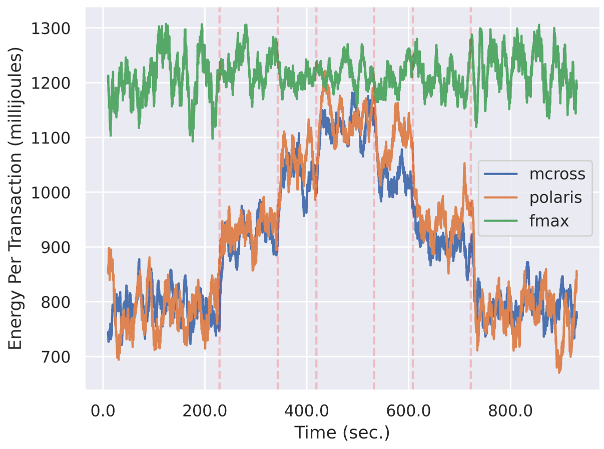
Length of a training episode

Learning to govern processor frequency in an online setting requires training the agent with a continuously running system. However, due to the bounded nature of training episodes, the system begins with a cold start in each episode. Due to this, the environment may behave differently during a short initial period in the episode.

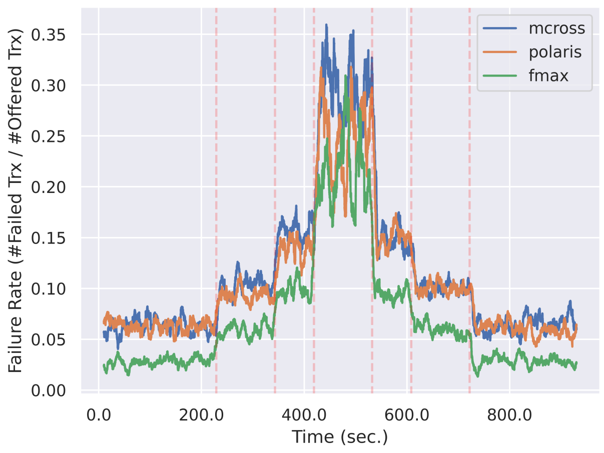
Initially, the transaction queue is empty and the CPU is idle. As transaction requests begin arriving, the environment observes a *transient state*, during which the queue starts



(a) Transaction arrival rate



(b) Energy per transaction



(c) Transaction failure rate

Figure 6.5: Variations in energy per transaction and transaction failure rate while scheduling a workload with time-varying transaction arrival rate. The dashed vertical lines illustrate points in time when system utilization ramps up/down. Reported values are averaged over a 10 second sliding window.

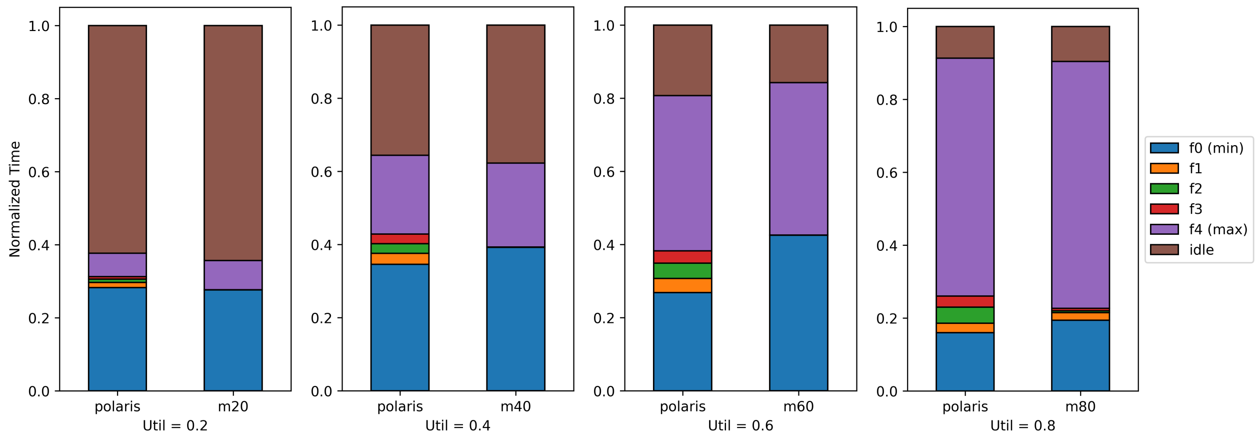


Figure 6.6: Processor frequency residency distribution during evaluation

building up, and the CPU starts picking and running transactions. As the episode progresses, eventually the queue stabilizes, and the system reaches a *stable state*. For effective training, it is important that the agent experiences more of the stable state and less of the transient state. That is, the episode length should be long enough that the system stabilizes early during the episode. This would be harder to attain in a real system, which may take longer to stabilize due to factors such as warming up of a large cache.

We ran a simple experiment to determine an appropriate episode length in our simulated system. We used m80 to schedule a workload at 80% utilization, and monitored the length of the transaction queue during an episode with 5000 timesteps (see Figure 6.7). The figure shows that the queue length varies between 1 and 9 and the behaviour of the system remains roughly consistent throughout the episode. It appears that the initial transient phase in our simulator is quite short and the system reaches a stable state relatively quickly. We conclude that any reasonably long episode (with more than 1000 timesteps, for instance) would suffice for the agent to be able to observe the dynamics of the simulated system and to not be biased towards any temporary behaviour at the beginning of the episode.

Convergence behaviour

The number of training iterations required for our method to converge to a good scheduling policy was empirically determined. To understand the convergence behaviour of a learning policy, we consider the training process for the m60 model (illustrated in Figure 6.8).

Figures 6.8a and 6.8b respectively show the total reward assigned to the agent and

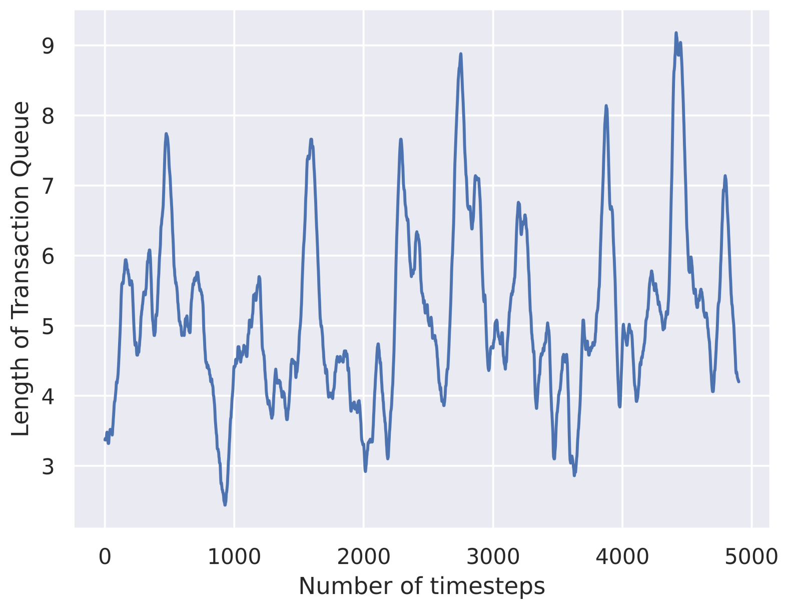
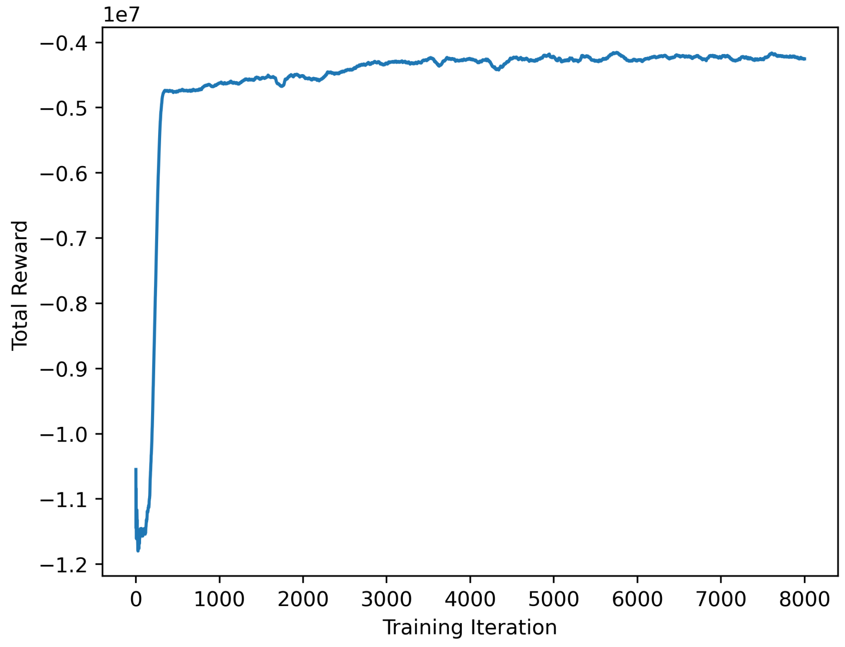


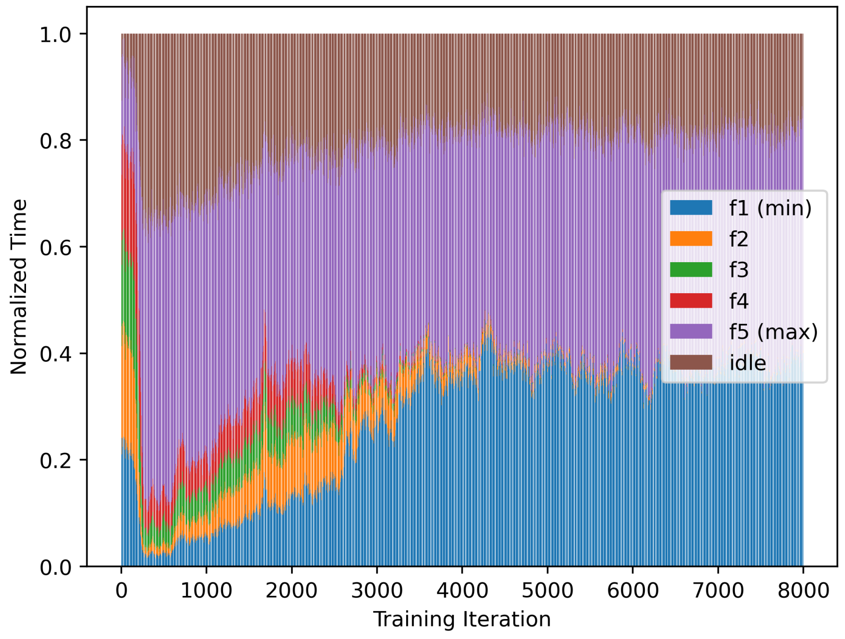
Figure 6.7: Length of transaction queue while scheduling with m80 at 80% utilization. Reported values are averaged over a sliding window of 100 timesteps.

the frequency residency distribution of the processor during successive training episodes. Training starts with an initial model that assigns similar probabilities to each possible frequency. As training progresses, the model quickly learns to favour higher frequencies. This provides a substantial reward payoff by driving down costly penalties due to failed transactions, and causes a sharp spike in the total reward. 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. This leads to small improvements in the total reward as training progresses.

We also found that the convergence of a policy model is load-dependent. Higher the intensity of the training workload, longer the model takes to converge to an optimal policy. Figure 6.9 illustrates this effect, by comparing the frequency residency distributions of the processor while training models m20 and m80. While the frequency residencies for m20 do not show significant change after 1000 epochs, those for m80 appear to be adjusting even after 8000 epochs.

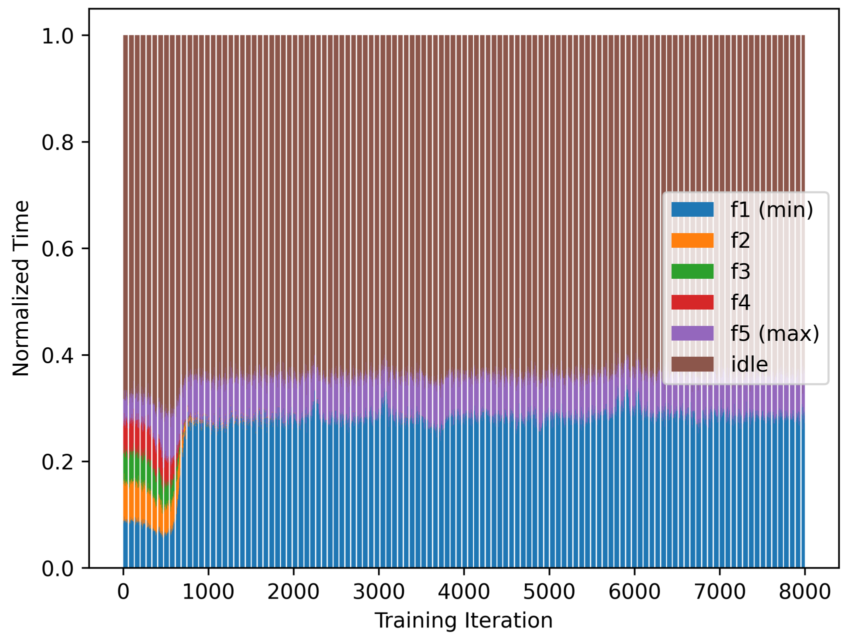


(a) Total Reward

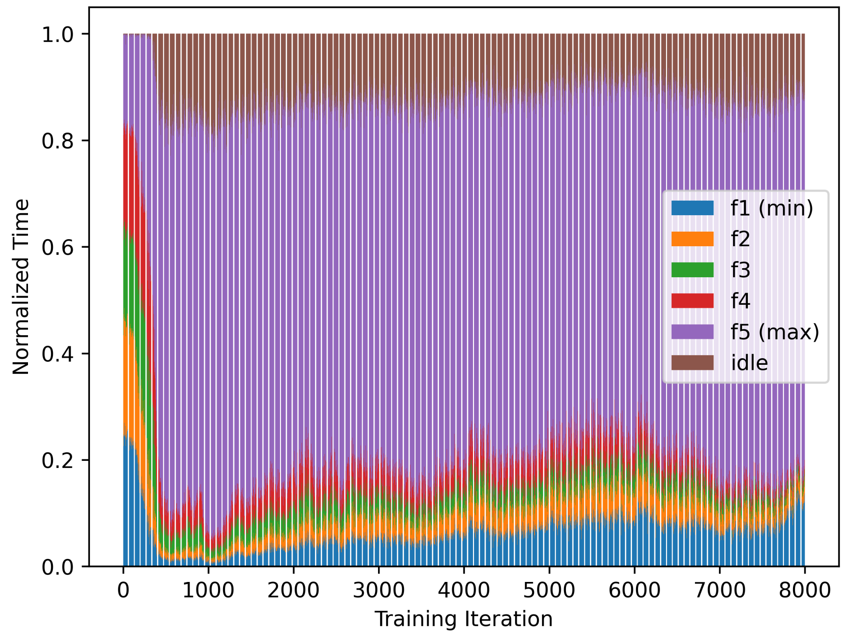


(b) Processor frequency residency distribution

Figure 6.8: Convergence behaviour during training at 60% utilization



(a) Processor frequency residency distribution during training at 20% utilization



(b) Processor frequency residency distribution during training at 80% utilization

Figure 6.9: Convergence behaviour at different workload intensities

6.2.7 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 ranging from 1 to 7 (the default is 3). We then tested each model using a workload at 60% utilization.

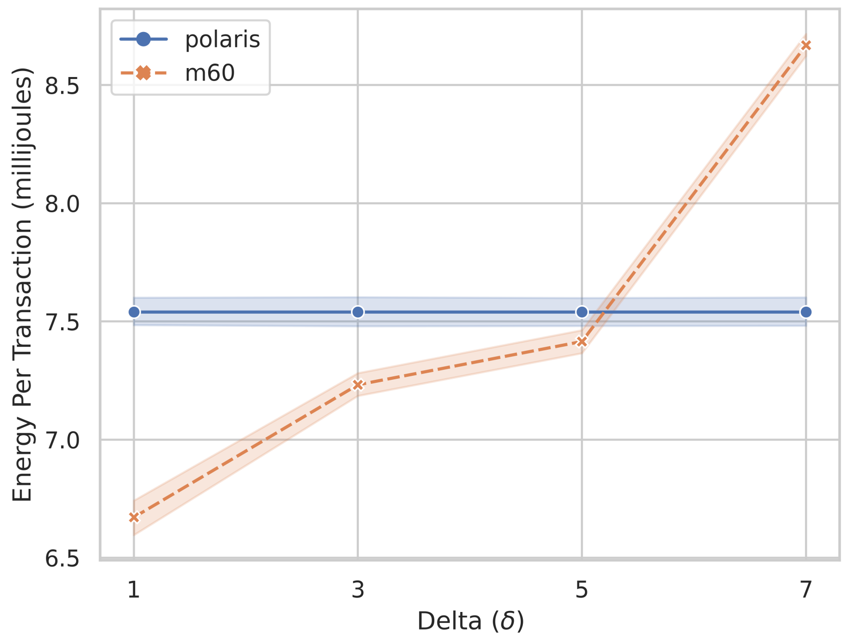
Figure 6.10 shows the energy consumption and failure rates that resulted from these models. The figure also shows the POLARIS baseline, for comparison. Low δ values reduce the penalty for failed transactions. Thus, we see higher failure rates than POLARIS, but also lower energy consumption. The situation is reversed at the $\delta = 7$. Setting $\delta = 5$ results in performance that closely matches POLARIS.

6.2.8 Overhead

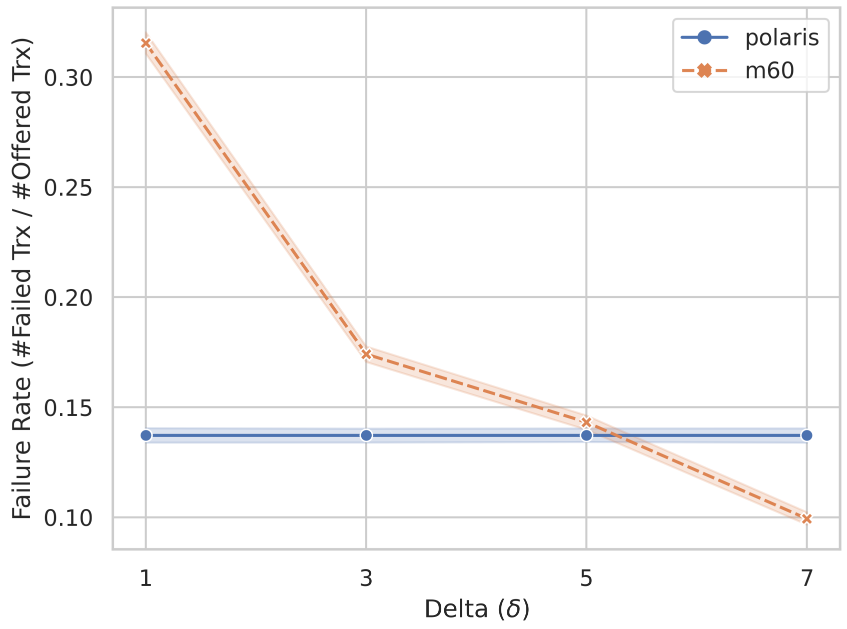
A problem with use of deep RL for frequency governance is the overhead of using the learned model. Our technique trains a model that maps from the state of the transaction processing system to a probability distribution over the possible processor frequencies. To use the model, the transaction processing 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 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.



(a) Energy per transaction



(b) Transaction failure rate

Figure 6.10: Effect of varying reward function

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 6.3: Time per frequency recommendation

Table 6.3 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 6.3 into perspective, they can be compared to the overhead introduced by POLARIS per recommendation, which is about 10 μs [16]. They can also be compared to the TPC-C transaction execution times reported in Table 6.1, which are mostly in a range from about 0.5ms to 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 thesis, 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 (except in the case of smaller models, in which the overhead of launching GPU kernels and data transfer to and from the GPU likely overshadows any improvements). But despite using a GPU, the model inference time is still high. Worse, the goal of energy-aware scheduling 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.

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 6.3 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.

There are other strategies that can be explored as ways of reducing this cost. 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 [7]. Here, though, the motivation is to reduce the cost of using the model.

Chapter 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 thesis, we demonstrate that RL-based governors can learn to manage CPU frequency for a transaction processing system as effectively as a state-of-the-art algorithm, and can adapt to both stable and time-varying workloads. The RL approach has the advantage that the resulting governor can be tuned to balance request latencies and power savings. We also show that there is significant power and performance overhead of using learned deep models when scheduling decisions need to be made quickly and frequently.

7.1 Future Work

In this thesis, we addressed the frequency governance problem, which is a sub-problem of energy-aware scheduling in transaction processing systems. While defining our problem setting, we assumed that the transaction execution priority is determined by EDF order. Extending the scheduling problem to also pick the transaction execution order could potentially further improve energy efficiency. Another future extension is generalizing the uniprocessor problem and designing RL-based energy-efficient scheduling algorithms for multi-processor CPUs.

The results presented in this thesis are based on experiments with a simulator, where the environment is a lot more controlled than a real system. This work can be extended to learn policies for scheduling workloads in a real database system. This would present additional challenges such as higher variability in transaction loads and execution times.

We also identified that the overheads of using learned frequency governors would likely outweigh their advantages when used in practice. As a future direction, strategies to reduce these overheads can be explored to make this approach practical for energy-aware scheduling.

References

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

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

[3] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Speed scaling to manage energy and temperature. *J. ACM*, 54(1), March 2007.

[4] David M Brooks, Pradip Bose, Stanley E Schuster, Hans Jacobson, Prabhakar N Kudva, Alper Buyuktosunoglu, John Wellman, Victor Zyuban, Manish Gupta, and Peter W Cook. Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

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

[6] Transaction Processing Performance Council. Tpc benchmark c standard specification, 1990.

[7] Nicholas Frosst and Geoffrey E. Hinton. Distilling a neural network into a soft decision tree. *CoRR*, abs/1711.09784, 2017.

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

[9] Ed Grochowski and Murali Annavaram. Energy per instruction trends in intel microprocessors. *Technology@ Intel Magazine*, 4(3):1–8, 2006.

[10] Yaniv Gur, Dongsheng Yang, Frederik Stalschus, and Berthold Reinwald. Adaptive multi-model reinforcement learning for online database tuning. In *EDBT*, pages 439–444, 2021.

[11] Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*, pages 3–14. IEEE, 2001.

[12] Hui Huang, Man Lin, and Qingchen Zhang. Double-q learning-based dvfs for multicore 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)*, pages 522–529. IEEE, 2017.

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

[14] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 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*, pages 24–35, 2009.

[15] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 598–610. ACM, 2015.

[16] Mustafa Korkmaz, Martin Karsten, Kenneth Salem, and Semih Salihoglu. Workload-aware cpu performance scaling for transactional database systems. In *Proc. ACM SIGMOD*, pages 291–306, 2018.

[17] Mustafa Korkmaz, Alexey Karyakin, Martin Karsten, and Kenneth Salem. Towards dynamic green-sizing for database servers. In *International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS@VLDB*, pages 25–36, 2015.

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

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

[20] Ning Liu, Zhe Li, Jielong Xu, Zhiyuan Xu, Sheng Lin, Qinru Qiu, Jian Tang, and Yanzhi Wang. 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)*, pages 372–382. IEEE, 2017.

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

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

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

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

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

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

[27] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device

placement optimization with reinforcement learning. In *International Conference on Machine Learning*, pages 2430–2439. PMLR, 2017.

[28] Xiang Ni, Jing Li, Mo Yu, Wang Zhou, and Kun-Lung Wu. Generalizable resource allocation in stream processing via deep reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pages 857–864, 2020.

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

[30] Iraklis Psaroudakis, Thomas Kissinger, Danica Porobic, Thomas Ilsche, Erietta Liarou, Pınar Tözün, Anastasia Ailamaki, and Wolfgang Lehner. Dynamic fine-grained scheduling for energy-efficient main-memory queries. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, pages 1:1–1:7. ACM, 2014.

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

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

[33] Richard S Sutton, David A McAllester, Satinder P Singh, Yishay Mansour, et al. Policy gradient methods for reinforcement learning with function approximation. In *NIPs*, volume 99, pages 1057–1063. Citeseer, 1999.

[34] Chen Tessler, Yuval Shpigelman, Gal Dalal, Amit Mandelbaum, Doron Haritan Kazakov, Benjamin Fuhrer, Gal Chechik, and Shie Mannor. Reinforcement learning for datacenter congestion control. *arXiv preprint arXiv:2102.09337*, 2021.

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

[36] Unified Extensible Firmware Interface UEFI. Advanced configuration and power interface specification. *ACPI. INFO, Roseville*, 2013.

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

[38] Rahul Urgaonkar, Ulas C Kozat, Ken Igarashi, and Michael J Neely. Dynamic resource allocation and power management in virtualized data centers. In *2010 IEEE Network Operations and Management Symposium-NOMS 2010*, pages 479–486. IEEE, 2010.

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

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

[41] Zichen Xu, Xiaorui Wang, and Yi cheng Tu. Power-aware throughput control for database management systems. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 315–324, 2013.

[42] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Symposium on Foundations fo Computer Science*, 1995.

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