Energy-Efficient Transaction Scheduling in Data Systems

by

Mustafa Korkmaz

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2021

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Alexandra Fedorova Associate Professor,Electrical and Computer Engineering, University of British Columbia
Supervisor:	Ken Salem Professor, Cheriton School of Computer Science, University of Waterloo
Internal Members:	Martin Karsten Associate Professor, Cheriton School of Computer Science, University of Waterloo
	Semih Salihoglu Assistant Professor, Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Wojciech Golab Associate Professor, Electrical and Computer Engineering, University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis consists of material all of which I authored or co-authored. Some portions of the this theses are based on the peer-reviewed joint work with Dr. Ken Salem, Dr. Martin Karsten and Dr. Semih Salihoglu, in which I am the first author and the primary contributor [105].

Abstract

Natural short term fluctuations in the load of transactional data systems present an opportunity for power savings. For example, a system handling 1000 requests per second on average can expect more than 1000 requests in some seconds, fewer in others. By quickly adjusting processing capacity to match such fluctuations, power consumption can be reduced. Many systems do this already, using dynamic voltage and frequency scaling (DVFS) to reduce processor performance and power consumption when the load is low. DVFS is typically controlled by frequency governors in the operating system or by the processor itself. The work presented in this dissertation shows that transactional data systems can manage DVFS more effectively than the underlying operating system. This is because data systems have more information about the workload, and more control over that workload, than is available to the operating system.

Our goal is to minimize power consumption while ensuring that transaction requests meet specified latency targets. We present energy-efficient scheduling algorithms and systems that manage CPU power consumption and performance within data systems. These algorithms are workload-aware and can accommodate concurrent workloads with different characteristics and latency budgets. The first technique we present is called POLARIS. It directly manages processor DVFS and controls database transaction scheduling. We show that POLARIS can simultaneously reduce power consumption and reduce missed latency targets, relative to operating-system-based DVFS governors. Second, we present PLASM, an energy-efficient scheduler that generalizes POLARIS to support multi-core, multi-processor systems. PLASM controls the distribution of requests to the processors, and it employs POLARIS to manage power consumption locally at each core. We show that PLASM can save power and reduce missed latency targets compared to generic routing techniques such as round-robin.

Acknowledgements

I would like to express my sincere gratitude to my advisor Dr. Ken Salem for his continuous guidance and invaluable kindness throughout many years of this journey. Without his patience and support, this dissertation would not have been possible. Working with him has been an honour and a privilege.

I would like to thank Dr. Martin Karsten and Dr. Semih Salihoglu for their contribution to the research projects composed in this thesis.

I would also like to thank the other members of my committee as well: Dr. Alexandra Fedorova and Dr. Wojciech Golab; for taking the time to read and review this thesis.

I feel grateful and lucky to have my family and owe thanks to my parents *Müberra* and *Nabi*, and my brothers *Ömer Faruk* and *Tolga Ertuğrul* for their support and encouragement throughout my life.

Finally, I would like to express special thanks to my wife *Zeynep* for her support, understanding and love. I feel blessed to walk through this life with her, and of course with our lovely daughter and the joy of my life, *Elif*...

Dedication

To my amazing parents and brothers, and my loved wife and daughter...

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Servers do not always run at maximum capacity. For example, Twitter and Google have reported average server utilization of only 20% [51] and 30% [23], respectively. A major reason for this is that server capacity needs to be scaled to accommodate peak workloads, but actual workloads fluctuate and can be bursty. For example, e-commerce servers are busiest during holidays and many production servers are more lightly loaded at night than during the day. When the load is not at its peak, the energy consumption of underutilized servers can be reduced. One option is to shut down some servers to save power, leaving fewer servers to handle the reduced workload.

Workload fluctuations occur on many time scales. In addition to diurnal patterns and longer term seasonal trends, loads also exhibit shorter-term fluctuations, on time scales of seconds or less [16]. These short-term fluctuations are caused by natural variations in the arrival rates of work, as well as variations in the service times of individual requests. Thus, a system that is handling 1000 requests per second on average may handle only 500 requests in some seconds, and 1500 requests in other seconds.

Like longer-term fluctuations, short term fluctuations also present an opportunity for reducing server power consumption. Modern CPUs can quickly increase and decrease their execution speed to adapt to these fluctuations. At lower speeds, CPUs consume less power. This is the opportunity we seek to exploit in this thesis.

Techniques designed to address longer-term workload fluctuations are generally unable to respond to short term load fluctuations. That is because they rely on relatively heavyweight mechanisms (such as powering servers down or migrating processes) or on mechanisms such as feedback control that take time to measure load and gradually adjust

power consumption. On the other hand, server CPU speed and, thus, power consumption can be adjusted quickly.

Modern server CPUs' performance and power consumption can be adjusted using dynamic voltage and frequency scaling (DVFS), which is supported by many server processors. DVFS allows a processor's voltage and frequency, and hence power consumption, to be adjusted on the fly. On modern processors, these adjustments can be made very quickly, e.g., on sub-microsecond time scales. This is fast enough to allow server power and performance to be adjusted on the time scale of individual server requests, even for systems with request latencies in the millisecond range.

DVFS must be managed. That is, something must control the scaling and decide whether and when to adjust voltage and frequency. Currently, DVFS is commonly managed by low-level governors implemented in an operating system (OS) or directly in hardware. Such governors typically base their decisions on low-level metrics, such as processor utilization, that are directly available to the OS. One advantage of these governors is that they are generic. Since they rely only on low-level metrics, they can be applied to save power across a broad range of applications. However, they may miss application-specific power saving opportunities.

1.1 Latency Critical Data Systems

Latency-critical data-intensive applications are common in data centers. Examples of such workloads include online transaction processing (OLTP) workloads in relational database systems [77], search queries [125, 50, 168], and key-value stores [144, 136]. In latency-critical workloads, units of work are short and well-defined. Furthermore, they often have implicit or explicit latency targets. In this work we focus on in-memory transactional database systems. We consider database systems that support multiple concurrent transactional workloads, each of which may have distinct characteristics and different latency targets. For example, a workload associated with high priority customers may have a lower latency target than a workload associated with regular customers.

Our central premise is that, for latency-critical data systems, DVFS can be managed more effectively by the data systems than by the underlying operating system. The data system has two main advantages when managing DVFS. First, the data system is aware of the units of work, such as queries and transactions. It may also have valuable information about these units of work, such as priorities, service level objectives, or the nature of the work itself. A data system can use this information to make better DVFS decisions.

For example, a data system can slow down the CPU when executing a small transaction. Second, the data system can also directly control its units of work. For example, it can reorder requests, or reject low-value requests when the load is high, or route requests among multiple CPU cores.

Our objective is to use DVFS to minimize server power consumption while ensuring that all workloads' latency targets are met. We show that by exploiting knowledge about transactions and the ability to manage the transaction execution, database systems can do a better job of both reducing both power consumption and hitting latency targets than OS-based DVFS managers.

1.2 Thesis Organisation and Research Contribution

In the remainder of this thesis, we first present background information in Chapter 2. In Chapter 3, we present an algorithm called POLARIS for DVFS-aware scheduling of tasks with latency targets in single processor CPUs. POLARIS chooses the order of task execution and controls execution speed to minimize power consumption while avoiding missed latency targets. POLARIS runs inside the database system, not in the underlying operating system. We show, empirically, that it is significantly more effective than OS-based DVFS governors.

In Chapter 4, we generalize the single-processor problem and present an energy-efficient scheduling algorithm called PLASM that is designed for multi-processor, multi-core CPUs. PLASM uses POLARIS for power and priority management at each individual CPU core, and a routing algorithm called FLARE for assigning requests to cores. In the chapter, we discuss various scheduling problems specific to multi-processor environments. We also empirically show that FLARE is significantly more effective than generic routing techniques.

In Chapter 5, we discuss execution time estimation in POLARIS and PLASM, which is an essential part of both algorithms. We also explain and empirically show how execution time estimation is used for trading power and performance. Finally, Chapter 6, presents related work on improving the energy efficiency of various types of software systems, and Chapter 7, summarizes our conclusions and offers some directions for future work.

Chapter 2

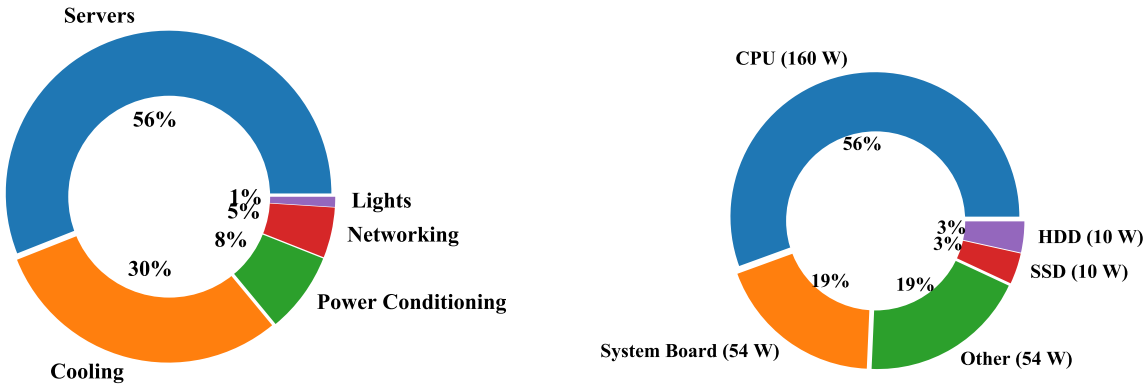
Background

Energy efficiency is a significant concern in data centers [24]. Recent studies show that data centers are responsible for 1% [128] of worldwide electricity consumption, and that power constraints can limit data center scale. Therefore there is an ongoing effort to improve energy efficiency, with many frontiers. Some of these efforts are holistic. For example, Google has been working on increasing overall power usage effectiveness for its data centers [2]. Microsoft is exploring the use of underwater data centers to cut cooling costs [145].

Other efforts focus on individual factors that contribute to overall power consumption. These factors include the power consumed by servers, storage, networking and infrastructure, and cooling [74, 152]. When we break down overall data center power consumption [140], servers account for a significant portion, as shown in Figure 2.1(a). Furthermore, servers are the primary generators of heat. Therefore, servers are indirectly responsible for the power used for cooling.

Figure 2.1(b) shows a breakdown of the power consumption of a modern server [163]. CPUs are largest consumers of power and they dominate overall power consumption, especially when the server utilization is high. Thus, CPUs play a major role in overall data center power consumption.

In the remainder of the chapter, we provide an overview of CPU power consumption and introduce some concepts that will be used in the remainder of the thesis. Section 2.1 explains the factors that contribute to CPU power dissipation, and Section 2.2 describes how power consumption can be managed and controlled.



(a) Data center components

(b) Server components

Figure 2.1: Power Consumption Break down

2.1 CPU Power Dissipation

Multiple factors, such as supply voltage, leakage, and switching, affect CPU power consumption [172]. However, a simple general model [32] for CPU power consumption (P) is given by

$$P = CV^2f + P_s \tag{2.1}$$

where C is a constant capacitance value, V is voltage, f is execution frequency, P_s represents static power dissipation. In this model, the term CV^2f represents the dynamic power dissipation.

CPUs' static power dissipation overhead has improved over recent years. For example, a detailed study on server power consumption breakdown in 2010 [163] reports that static power consumption accounts for 30% of total CPU power. However, a more recent study in 2018 [102] shows that this ratio has dropped all the way down to 8%, yet total CPU power consumption is still the dominant factor in server power consumption.

2.1.1 Dynamic Voltage and Frequency Scaling

Modern processors (and memory [47]) support dynamic voltage and frequency scaling (DVFS), which allows execution frequency and voltage (f and V in Equation 2.1) to be controlled [79]. With this capability, processors can operate at different power and performance levels.

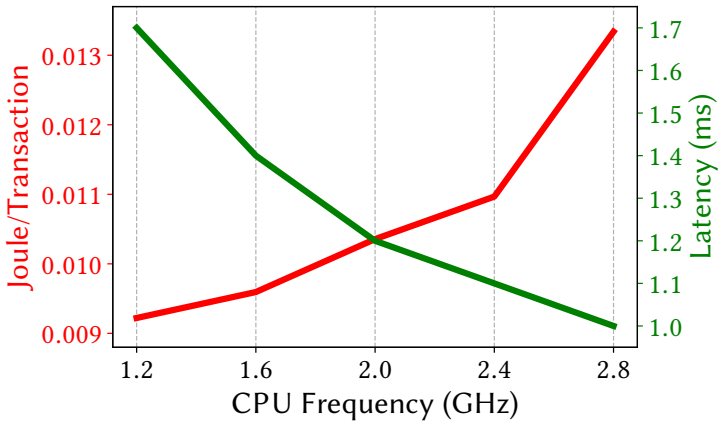


Figure 2.2: Energy per transaction and transaction latency under at different frequencies in Intel E5-2640 v3

In practice, V and f are not varied independently. Increasing f requires corresponding increases in V. Thus, the formula for dynamic power dissipation (P_d) is often simplified as $P_d \propto f^{\alpha}$, where α is typically in the range $1 < \alpha \leq 3$ for server-grade processors [32, 167, 66]. Because $\alpha > 1$, the dynamic power-frequency relation in CPUs is convex. This means that the higher the frequency, the more dynamic power the processor uses per unit of work that it performs. That is, if $f_{slow} < f_{fast}$ and w represents a unit of work, then $\frac{w}{f_{slow}}(f_{slow})^{\alpha} < \frac{w}{f_{fast}}(f_{fast})^{\alpha}$. Thus, running a job as slowly as possible results in the least dynamic power consumption.

We experimented and measured power consumption under different CPU frequency levels to validate the convex relation between dynamic power and frequency, using the server, Shore-MT system, and TPC-C workload that will be described in more detail in Chapter 3. In each experiment, we set all of the server's CPU cores to run a fixed frequency. Shore-MT workers on each CPU core executed TPC-C NewOrder transactions in a tight loop, as quickly as possible. After a warm up period, we allowed the system to run for a fixed amount of time, and measured the average CPU power consumption, the total number of transactions completed, and the average latency per transaction. From these measurements, we computed the average energy consumption per NewOrder transaction.

Figure 2.2 summarizes our results, showing energy per transaction and transaction latency at each CPU frequency. As expected, the CPU consumes more energy per transaction when we use a higher frequency. As the static system power is the approximately the same across different frequency levels, we attribute the extra energy to the extra dynamic

power consumption due to higher frequency. Our observation is consistent with the convex relation in the CPU power and speed model. As expected, latency decreases as CPU frequency increases, but not in proportion to the frequency increase. For example, doubling frequency from 1.2GHz to 2.4GHz reduced transaction latency, but does not quite cut it in half.

2.1.2 Power Gating

Modern server-grade CPUs can also adjust static power consumption by using power gating [120, 116]. Power gating allows CPUs to save additional power by cutting the supply voltage to certain parts of the CPU when they are not in use. This reduces static power consumption. Power gating may also have some impact on performance, because there is normally some delay involved in waking up and restoring power to gated components.

2.2 ACPI

The Advanced Configuration and Power Interface (ACPI) is a cross-platform industry standard that defines a power management interface [165]. For CPUs, ACPI defines two different types of states: C-states (processor power states) and P-states (device and processor performance states). P-states are used to abstract DVFS, while C-states abstract power gating [169, 114].

2.2.1 C-States

ACPI C-states define the different power states of a processor [90], which are normally implemented by gating [172] different parts of the processor [89, 102]. There are two types of C-states; core C-states and package C-states. The C-states are numbered starting from zero, such as *C0*, *C1*, and so on. The standard permits different types of CPUs to implement different subsets of the possible C-states defined by ACPI. However, *C0* must be implemented in all CPUs.

Core C-States

C0 represents the active state, in which the CPU core is fully active and powered. The remaining C-states represent idle states in which parts of the core are power gated. Higher-

numbered C-states represent deeper idle states with greater power reductions from gating. For example, in *C1*, the processor clock is stopped via clock gating, but L1 and L2 caches are still coherent. In *C3*, L1 and L2 are flushed to the last level cache. In *C6*, all of the core's execution context is flushed to the last level cache, and the whole CPU core is power gated. The deeper the C-state, the greater the power savings. However, deeper C-states also require more time to return the core to *C0* [97].

Package C-States

In addition to core-level C-states, there are package-level C-states that represent power gating in other parts of the processor besides the execution cores. To prevent any confusion, we identify package C-states as *PCx*. In each package, the core with the lowest C-state determines the package C-state. Thus, if even a single core is active (in state C0), its package cannot go into an idle C-state.

2.2.2 P-States

ACPI P-states are used to represent the performance states of an active processor and are relevant only when a core is in *C0* (active) C-state. Each P-state represents an operating voltage and frequency pair. Different types of processors can define different numbers of P-states. For example, the AMD FX 6300 offers five distinct P-states for each core, as shown in Figure 2.3. P-states are numbered starting from zero, such as *P0*, *P1*, and so on. The lower the P-state number, the higher the frequency.

P-State	Frequency	Voltage
P0	3.5 GHz	1.4 V
P1	3.0 GHz	1.225 V
P2	2.5 GHz	1.125 V
P3	2.0 GHz	1.025 V
P4	1.4 GHz	0.9 V

Figure 2.3: P-states of AMD FX 6300

2.2.3 Power Control

Typically, there is no direct mechanism for software control of processor C-states, although software may offer control hints. For example, the x86 instruction *mwait* accepts a target

C-state hint and initiates idle wait. The hint recommends a particular C-state for the idle wait, but the actual C-state(s) that are used are determined by the processor [17].

Unlike C-states, P-states can be directly controlled by software. There are several mechanisms for doing so. OS-level power manager interfaces allow applications or system administrators to control P-States. For example, Windows 10 offers several pre-defined power plans such as *Balanced*, *Power Saving*, and *Performance*. In Linux, the `cpufreq` kernel module [31] provides a variety of power governors. `cpufreq` governors can be categorized into two groups; static and dynamic governors.

`cpufreq`'s static governors allow users to set a specific CPU frequency. For example, the "performance" governor sets the CPU speed to the peak speed, and the "userspace" governor allows user-level applications to select any of the available P-states. On the other hand, `cpufreq`'s dynamic governors adjust CPU speed on the fly according to the CPU utilization. For example, the "ondemand" governor sets a higher P-state when the utilization is high and vice versa.

For x86 processors, all of the P-state management mechanisms ultimately rely on Model Specific Registers (MSRs) [3, 91] to control P-states. MSRs contain CPU-specific information which can be read and written by software, and which can be used to control some aspects of the processor, including core P-states.

Power management can also be implemented directly in the hardware. For example, recent Intel CPUs are equipped with a mechanism called RAPL (Running Average Power Limit) [71]. Given a power consumption target and a time window, the RAPL mechanism adjusts execution frequency to keep the running average power consumption of the processor (over the specified time window) at or below the specified target level. RAPL allows finer-grained control of power and execution frequency than P-states. However, it is specific to Intel CPUs. In this thesis, we use ACPI P-states as, to the best of our knowledge, all the modern server-grade server CPUs support P-states.

Chapter 3

Single-Processor Energy Aware Transaction Scheduler

3.1 Overview

As described in Chapter 1, latency-critical systems' workloads fluctuate over time, even on time scales of seconds or less. Operating system frequency governors can reduce CPU power consumption by using DVFS to adjust execution speed in response to these fluctuations, increasing speed when the CPU is heavily utilized, and reducing it when utilization is low. For systems that handle latency-critical workloads, such as transactional database systems, we argued that it should be possible to managed DVFS more effectively in the database system, which has more information about the latency critical workload that it is supporting.

In this chapter, we test this premise by developing an energy-aware scheduling algorithm for transactional database systems, and comparing it to operating system based governors. Like OS governors, our algorithm controls DVFS to take advantage of short-term workload fluctuations. However, unlike OS governors, it takes advantage of database-system-specific workload information to guide its decisions.

This chapter makes the following technical contributions:

- We present an on-line workload-aware scheduling and frequency scaling algorithm called *POLARIS* (POwer and Latency Aware Request Scheduling). POLARIS controls both transaction execution order and processor frequency to minimize CPU

power consumption while observing per-workload latency targets. Many modern in-memory transaction data processing systems, like VoltDB [159] and Silo [164], are architected to execute each transaction from start to finish in a single thread on a single processor core. POLARIS is a non-preemptive scheduler because non-preemptive scheduling is a good fit for such systems.

- We provide a competitive analysis of POLARIS against YDS [182], a well-known optimal offline preemptive algorithm, as well as a YDS-based on-line preemptive algorithm (OA [182]). This analysis provides insight into aspects of POLARIS's behaviour, such as the impact of non-preemptiveness and the importance of transaction scheduling.

- We present a prototype implementation of POLARIS within the Shore-MT storage manager [95]. Section 3.5 describes some of the practical issues that we had to address in doing so. We use the prototype to perform an empirical evaluation of POLARIS under a variety of workloads and load conditions, using in-kernel dynamic DVFS governors as baselines. Our results show that POLARIS produces greater power savings, fewer missed transaction deadlines, or both. We also show how POLARIS' effectiveness is affected by two key factors: (1) the average load on the system, and (2) scheduling slack, i.e., the looseness of the transactions' deadlines. Although POLARIS dominates the baselines under almost all conditions, its benefits are greatest when the average load is neither very high nor very low. Not surprisingly, greater scheduling slack increases the advantage of deadline-aware schedulers, like POLARIS, over deadline-blind operating system alternatives.

3.2 POLARIS

Servers in data centers host a broad spectrum of applications [61], including collaboration and business solutions, database and analytics tools, video streaming and social networking [22, 132]. Data systems of various kinds are among the most commonly used data center applications. Database management systems, object storage systems, key-value stores, batch processing and stream processing systems are all counted as data systems, and they are primarily responsible for the storage and processing of data [88, 139, 125].

Latency-critical workloads are common in data systems. Latency critical workloads consist of short requests with tight latency objectives, often on the scale of a few seconds or less. Depending on the workload, latency objectives may be fine-grained (e.g, a deadline for each unit of work) or may be expressed at the level of an entire workload. For example, some

systems have Service Level Agreements(SLA) that require the most requests in a workload achieve latencies below a specified target [50]. Missing latency targets can have financial implications for service providers [73, 54], and can affect end-user satisfaction [149, 43].

POLARIS is designed to manage DVFS for latency critical data systems' workloads. For the purposes of the presentation in this section, we assume a server with a single single-core processor that supports DVFS. To manage multiple processors, or processors with multiple frequency-scalable cores, we can use multiple instances of POLARIS. In Section 3.5, we describe the POLARIS prototype architecture that uses this approach to manage a multi-processor, multi-core server.

A server accepts transaction execution requests, each of which is associated with a *latency target*. POLARIS's objective is to minimize CPU power consumption while ensuring that each request is completed within its latency target.

POLARIS is *workload-aware*. In addition to its latency target, each request is assumed to be tagged with a *workload type* to indicate which workload it is part of. Workloads are important to POLARIS, since POLARIS creates an execution time prediction model *per workload type*, as we describe in section 3.3.

Many data systems provide sophisticated *workload managers* [131, 87, 138, 162, 80, 141] that allow incoming requests to be assigned to workloads based on the properties of the request. For example, these properties might include the name of the user, application, or function that generated the request, the complexity or estimated cost of the request, the connection over which the request arrived, and so on. Some workload managers track workloads' performance or resource consumption, allow priorities or performance targets to be associated with individual workloads, and take action or make recommendations when targets are missed. For example, IBM DB2 Workload Manager [86] can monitor workloads' performance, and can adjust priorities and resource allocations or take other user-specified actions when targets are missed. POLARIS assumes that incoming requests are assigned to workloads by such a mechanism. However, POLARIS itself is agnostic with regards to how this assignment is defined. That is, it neither defines nor depends on specific policies for workload assignment.

POLARIS's primary objective is to ensure that transactions meet their workloads' latency targets. However, because transaction execution speed is limited by the processor's highest-frequency P-state and because there are no constraints on the the arrival of transactions or on transaction deadlines, it may not be possible for POLARIS (or any scheduling algorithm) to ensure that all transactions meet their deadlines. In such cases, POLARIS will run the processor at the highest frequency, which will have the effect of completing late transactions as quickly as possible.

Notation	Meaning
Q	transaction request queue
t_0	currently running transaction
e_0	running time (so far) of t_0
$\mathcal{W}$	set of workloads
$L(c)$	latency target of workload $c \in \mathcal{W}$
$c(t)$	workload of transaction t, $c(t) \in \mathcal{W}$
$a(t)$	arrival time of transaction t
$d(t)$	deadline of transaction t, $d(t) = a(t) + L(c(t))$
$\mathcal{F}$	set of possible processor frequencies
$\hat{\mu}(c, f)$	estimated execution time of workload c transaction at frequency f
$\hat{q}(t, f)$	estimated queuing time of t at frequency f

Figure 3.1: Summary of Notation

3.2.1 The POLARIS Algorithm

The arrival of a new transaction request or the completion of a request triggers the execution of POLARIS. In each of these situations, POLARIS chooses a frequency for the processor, based on the set of transactions that are running or waiting to run. It assumes that there is a fixed set of voltage and frequency configurations in which the the processor can run, corresponding to the processor's available P-States. Higher frequencies allow the processor to execute transactions faster, but they also consume more power. Figure 3.1 summarizes notation that we use to describe transactions and processor frequencies.

Figure 3.2 shows the POLARIS frequency selection procedure, SetProcessorFreq, which runs each time a transaction request arrives or is completed. SetProcessorFreq chooses the smallest available processor frequency such that all transactions, including the running transaction and all waiting transactions, will finish running before their deadlines if the processor were to run at that frequency.

The frequency selection algorithm relies on a transaction execution time model, which predicts the execution time of a transaction of a given workload at a given processor frequency. We use $\hat{\mu}(c, f)$ (in Figure 3.2) to represent the predicted execution time of a workload c transaction at frequency f. (We discuss how POLARIS predicts execution time in Section 3.3.) In Figure 3.2, $\hat{q}(t, f)$ represents the total estimated queueing time for transaction $t \in \mathcal{Q}$, assuming that the processor runs at frequency f. This is defined as

State: Q: queue of waiting transactions
State: t_0: currently running transaction
State: e_0: run time (so far) of t_0
State: t_{now}: current time

function SetProcessorFreq( )
    ▷ find minimum freq for current transaction
    for each f_{new} in $\mathcal{F}$, in increasing order **do**
        if $t_{now} + \hat{\mu}(c(t_0), f_{new}) - e_0 \leq d(t_0)$ **then break**
        end if
    end for
    ▷ ensure all queued transactions finish in time
    for each t in Q, in EDF order **do**
        if $t_{now} + \hat{q}(t, f_{new}) + \hat{\mu}(c(t), f_{new}) \leq d(t)$ **then continue**
        end if
        ▷ f_{new} is not fast enough for t
        ▷ find the lowest higher frequency that is
        for each $f \in \mathcal{F} | f > f_{new}$, in increasing order **do**
            $f_{new} \leftarrow f$
            if $t_{now} + \hat{q}(t, f) + \hat{\mu}(c(t), f) \leq d(t)$ **then break**
            end if
        end for
        ▷ no further checking once we need highest freq
        if f_{new} = maximum frequency in $\mathcal{F}$ **then**
            set processor frequency to f_{new}
            return
        end if
    end for
    set processor frequency to f_{new}
    return
end function

Figure 3.2: POLARIS Processor Frequency Selection

follows:

$$\hat{q}(t,f) = \hat{\mu}(c(t_0),f) - e_0 + \sum_{t' \in Q | d(t') < d(t)} \hat{\mu}(c(t'),f)$$

That is, t must wait for the currently running transaction's remaining execution time, and must also wait for all queued transactions with deadlines earlier than t's.

POLARIS also controls transaction execution order. Transaction requests that arrive while the processor is busy running another transaction are queued in order of their workloads' deadlines by POLARIS. When the running transaction finishes, POLARIS dispatches the next transaction (the one with the earliest deadline) from the queue. As we note in Section 3.1, each transaction, once dispatched runs to completion. In Section 3.4, we relate POLARIS to YDS, a well known, *optimal* offline frequency scaling and scheduling algorithm. YDS achieves optimality by identifying batches of so-called "critical" transactions and executing them in earliest deadline first (EDF) order, since this may allow YDS to run the batch at a lower frequency than would be possible if transactions ran in arrival order. POLARIS executes transactions in EDF order for the same reason.

3.3 Execution Time Estimation

POLARIS requires estimates of the execution time $\hat{\mu}(c,f)$ for transactions of each workload $c \in \mathcal{W}$ at each possible processor execution frequency $f \in \mathcal{F}$. There is a substantial body of work on estimating execution times of data system queries [60, 53, 6, 177]. This work varies in the amount of workload complexity it assumes, the amount of workload information that is required, and in the use of black-box vs. white-box modeling. For POLARIS, our focus is on transactional workloads with many short units of work, rather than complex SQL queries. However, even in this relatively simple setting, accurate prediction of individual transactions' execution times is challenging, since factors such as resource contention and data contention can affect execution. Besides, workload characteristics can change over time.

For POLARIS, we have taken a simple, conservative, dynamic, black-box statistical approach to estimate execution time. Specifically, for each combination of workload c and frequency f in $\mathcal{W} \times \mathcal{F}$, POLARIS tracks the pth percentile of measured execution times over a sliding window of the S most recent transactions from workload c that run at frequency f. The current tracked value is used as $\hat{\mu}(c,f)$ in POLARIS's SetProcessorFreq algorithm (Figure 3.2).

To track these percentiles, we adapted an algorithm of Härdle and Steiger [76] for tracking a running median to instead tracking the pth percentile of the observed execution time distribution. For all of the experiments reported in this section, we use $S = 1000$ and experimented with percentiles in the range $95 \leq p \leq 99$.

This approach has several advantages in our setting. First, it is fast, which is important because we do not want to squander POLARIS's power savings on estimation overhead. Second, it requires little space: a few kilobytes per element of $\mathcal{W} \times \mathcal{F}$. We expect both $\mathcal{W}$ and $\mathcal{F}$ to be small; both are less than ten in our experiments. Third, it can adapt to changing workloads and system conditions, because of the sliding window. Finally, it requires no information about each transaction, other than its workload label.

This approach is conservative because we are using tail latencies to predict the execution time of every transaction. For most of the experiments presented in this chapter, we have used $p = 95$. This is important because POLARIS's primary objective is to meet transaction latency targets. For example, Figure 3.3 illustrates the mean and 95th percentile latencies for the individual transactions in our TPC-C workload and also, in the last row, the latencies for the overall combined workload. In this example, the tail latencies are 2.5 to 4.8 times larger than the means. The use of lower values than $p = 95$ will make POLARIS save power more aggressively, but also increases the risk of missed latency targets.

In Chapter 5, we revisit time estimation, which is needed by both POLARIS and the transaction routing algorithm we introduce in Chapter 4. In Chapter 5, we describe an alternative to the quantile-based estimator introduced here. Both estimators allow POLARIS to function effectively. However, the estimator described in Chapter 5 guarantees that the execution time estimates for each workload class decrease monotonically with frequency. This property, which helps to ensure that POLARIS can fully consider the use of all available P-states, is not shared by the quantile-based estimator presented here, since it estimates quantiles independently for each frequency.

3.4 Theory & Competitive Ratio Analysis

In this section we analyze the performance of POLARIS through a competitive analysis against two existing algorithms YDS [182] (Section 3.4.2) and OA [21, 182] (Section 3.4.3).

We have two objectives in this section. The first is to provide a theoretical justification for why POLARIS is an effective algorithm. The second is to establish a connection between the behaviors of POLARIS and OA under certain settings. We provide our analysis under

Request Type	Execution Time (μs) @2.8 GHz Mean	@2.8 GHz $\mathbf{P_{95}}$	@1.2 GHz Mean	@1.2 GHz $\mathbf{P_{95}}$
New Order (45%)	2059	5414	4772	12048
Payment (47%)	301	859	733	2388
Order Status (4%)	250	1682	809	3453
Stock Level (4%)	3435	5106	8062	11495
Combined Workload	1560	4465	3941	13525

Figure 3.3: TPC-C mean and 95th percentile ($\mathbf{P_{95}}$) transaction execution times at maximum and minimum CPU frequency. Percentages indicate the transaction mix in the workload.

the standard theoretical model [15, 21, 182] in which algorithms can scale the speed of the CPU to arbitrarily high levels and thus execute every transaction before its deadline. Therefore we focus only on the energy consumption of algorithms and not their success rates. We review this standard model in Section 3.4.1.

Broadly, energy aware scheduling algorithms can be classified into four categories along two dimensions as shown in Figure 3.4: (1) preemptive vs non-preemptive; and (2) offline vs online algorithms. Offline preemptive algorithms are the most computationally powerful algorithms. YDS [182] is the optimal offline preemptive algorithm and therefore consumes the lowest possible energy among all scheduling algorithms. In contrast, online non-preemptive algorithms, such as POLARIS, are the most computationally constrained ones.

The natural algorithm to compare POLARIS against would be the optimal offline non-preemptive algorithm, which we refer to as OPT_{np}. However, computing the optimal offline non-preemptive schedule is NP-hard [15], and an explicit description of OPT_{np} is not known. Instead, we provide a competitive ratio of POLARIS against YDS, which also implies a competitive ratio against OPT_{np}. As we show in Sections 3.4.4 and 3.4.5, we get a competitive ratio of POLARIS against YDS indirectly through a competitive analysis against OA, which is an online preemptive algorithm. In doing so we also meet our second objective of establishing the connection between POLARIS and OA.

Finally we note that several online non-preemptive algorithms have been developed in literature for variants of the speed-scaling problem. Examples include maximizing the throughput [13] or minimizing the total response time [10] of transactions under a fixed energy budget. However, no prior work studies the problem of minimizing energy con-

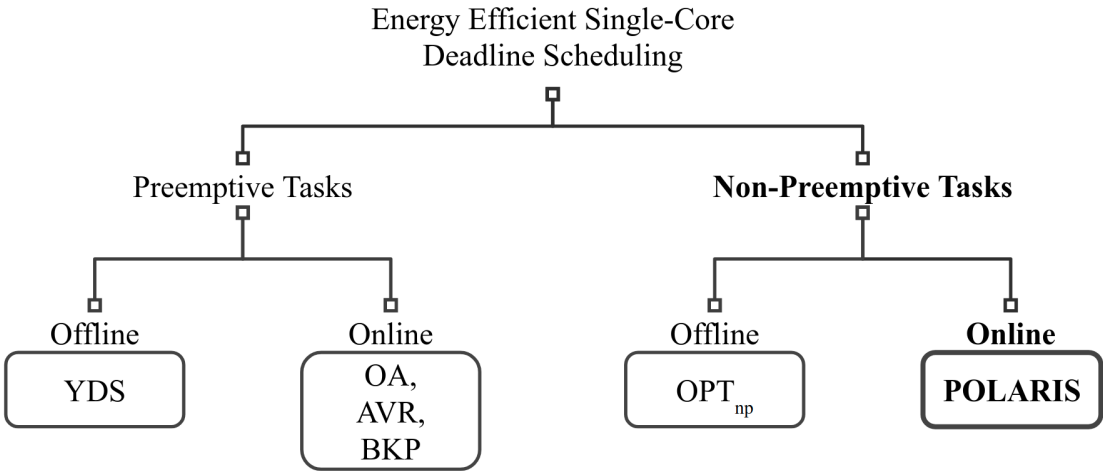


Figure 3.4: Energy aware scheduling algorithms.

sumption as we do in this section. We refer the reader to references [7] and [62] for a survey of these algorithms.

3.4.1 Standard Model

In the standard model, a problem instance P consists of n transactions, where each transaction t arrives with an arrival time $a(t)$, a deadline $d(t)$, and a load $w(t)$. $w(t)$ represents the amount of work that a transaction must perform, which is assumed to be known accurately. Algorithms can scale the speed of the processor to arbitrarily high levels. When the processor is running at frequency (speed) f, a transaction t executes in $w(t)/f$ time. The power consumption of the processor is assumed to be f^α, where $\alpha > 1$ is a constant [32], which guarantees that the power-speed function is convex. We observe that in this model algorithms, including POLARIS, are *idealized* and can execute every transaction before its deadline, i.e., achieve 100% success rate. This is because (a) they know transactions' loads accurately; and (b) can pick arbitrarily high speeds to finish any transaction on time.

3.4.2 Yao-Demers-Schenker (YDS)

YDS is the optimal offline preemptive algorithm. Given a problem instance P, let an *interval* be the time window between the arrival time $a(t_i)$ of some transaction t_i and the (later) deadline $d(t_j)$ of a possibly different transaction t_j in P. Define the *density*

of a given interval I to be $\Sigma_k w(t_k)/|I|$, where the summation is over all transactions t_k such that $[a(t_k), d(t_k))$ is within I. Given P, YDS iteratively performs the following step until there are no transactions left in the problem. It finds an interval with the maximum density, which is called the *critical interval*. Let CI be the first critical interval YDS finds. The algorithm schedules the speed of the processor during CI to the density of CI and schedules execution of the transactions in CI in EDF order. Then, the algorithm removes CI and the set of transactions in CI from P, constructs a *reduced* problem P', and repeats the previous step on P'. P' is the same as P except any transaction whose arrival and deadline intersects with CI is shortened exactly by the time it overlaps with CI.

In its final schedule, YDS potentially preempts a transaction t whenever transaction t has an arrival time and a deadline that spans a critical interval CI that the algorithm has picked at some step. That is, YDS might run part of t before the start of the CI, preempt t when CI starts, and resume executing t after CI.

3.4.3 Online Preemptive Algorithms

OA is an online preemptive algorithm based on YDS [182]. Each time a new transaction arrives, OA uses YDS to choose a schedule. Suppose that a new transaction arrives at time τ. OA runs YDS on a problem instance consisting of the following transactions:

- The newly arrived transaction, t_{new}.
- The currently running transaction, t_r, with its load $w(t_r)$ taken to be the *remaining* load of t_r, and with its arrival time taken to be τ.
- Any other transactions waiting in the system, with their arrival times adjusted to be τ.

We make an important observation here. Note that in the problem instance constructed by OA, all transactions have the same arrival time τ. Thus, if there are k transactions in the system, there are exactly k intervals from which YDS chooses the first critical interval. The first includes just the transaction with the earliest deadline, the second includes the transactions with the two earliest deadlines, and so on. Furthermore, the first critical interval will include the transaction with the earliest deadline, since it is part of all of the possible intervals. Since YDS schedules transactions in EDF order, this first transaction must be either t_r or t_{new}. Thus, if $d(t_{new}) < d(t_r)$, OA will *preempt* t_r and start running t_{new}. If, on the other hand, $d(t_r) < d(t_{new})$, t_r will continue running after t_{new}'s arrival, and t_{new} will run later.

In addition to OA, Yao et al. propose another online heuristic for the preemptive problem, called Average Rate (AVR)[182]. AVR sets CPU speed to the sum of the densities of the transactions that are not yet competed: $\Sigma_i(w(t_i)/(d(t_i)-a(t_i)))$. Yao et al show that AVR's competitive ratio against YDS is $2^{\alpha-1}\alpha^\alpha$. In another work, Bansal et al. showed that OA is α^α competitive against YDS [21]. Bansal et al. also propose another online algorithm, called BKP [21]. Like OA, BKP uses interval densities. However, it increases speed by a factor of e. BKP's competitive ratio against YDS is $2\left(\frac{\alpha}{\alpha-1}\right)^\alpha e^\alpha$. This is better than OA's for large values of α. However, α is typically small in server-grade CPUs (e.g, $1 < \alpha \leq 3$), and OA gives a better competitive ratio in that case.

3.4.4 OA vs. POLARIS

Next, we compare the behavior of OA with that of (idealized) POLARIS . We start by comparing the algorithms under the scenario in which a newly arriving transaction has a later deadline than the currently running transaction.

Lemma 3.4.1. *Suppose that both POLARIS and OA have the same queue at a point in time, with k total transactions, one running (t_r) and $k-1$ waiting, with the exact same loads. Suppose a new transaction t_{new} arrives, and that $d(t_r) \leq d(t_{new})$. Until the arrival of the next transaction, POLARIS and OA will execute transactions in the same order, and with the same processor frequency.*

Proof. First, we consider execution order. By definition, POLARIS will finish running t_r and then run the remaining transactions in earliest-deadline-first (EDF) order. Since t_r has the earliest deadline, this amounts to running all transactions in (EDF) order. OA identifies a critical interval, schedules the transactions in that interval in EDF order, reduces the problem instance by removing the critical interval and its transactions, and repeats on the reduced instance. However, because all transactions have the same arrival time, all transactions in the first critical interval chosen by OA will have deadlines earlier than all remaining transactions. Since the resulting reduced problem instances all have the same structure as the original instance, each successive critical interval's transactions' deadlines will be later than those of previously selected intervals, and earlier than those of subsequently selected intervals. Thus, by scheduling each critical interval in EDF order, OA will execute all transactions in EDF order, like POLARIS.

Second, we consider processor speed. Let CI_i represent the ith critical interval chosen by OA. Let P_1 represent the original problem instance considered by OA, and let P_i represent the reduced problem instance under which $CI_i (i > 1)$ is chosen. Since both algorithms

agree on EDF execution order, we show by induction on the number of transactions that POLARIS and OA agree on the processor speed used to execute each transaction.

Base Case: Consider the transaction with the earliest deadline in the original, non-reduced problem instance, P_1. OA will run this transaction first, using frequency $den(CI_1)$. Now consider POLARIS. When t_{new} arrives, POLARIS will use SetProcessorFreq (Figure 3.2) to set the processor frequency. SetProcessorFreq iterates over the transactions present in the system, including t_r and t_{new}. After iterating over all $k+1$ transactions in the system, the selected frequency will be

$$\max_{1 \leq j \leq k+1} den(I_j)$$

where I_j represents the interval consisting of the j earliest-deadline transactions. Thus, after considering all $k+1$ transactions, the frequency chosen by POLARIS will correspond to that required by the interval with the highest density, i.e., the frequency of the critical interval. Thus, POLARIS , will set the processor speed to $den(CI_1)$, the same speed chosen by OA. Since POLARIS only adjusts processor speed when transactions arrive or finish, it will remain at $den(CI_1)$ until the transaction completes.

Inductive Step: Suppose that the nth transaction is finishing execution under POLARIS, and that POLARIS has run it and all preceeding transactions at the same frequencies that were chosen by OA. Consider the $n+1$st transaction. There are two cases:

Case 1: Suppose that the nth and $n+1$st transactions belong to the same critical interval under OA. Suppose it is the mth critical interval, which implies that both transactions ran at speed $den(CI_m)$ under OA. By our inductive hypothesis, the nth transaction also ran at speed $den(CI_m)$ under POLARIS. When the nth transaction completes, POLARIS will run SetProcessorFreq. The set of transactions over which it runs will be exactly those in P_m, minus those transactions in CI_m that have already finished executing, including the nth transaction. When POLARIS runs SetProcessorFreq, the highest density interval it finds will be CI_m, but shortened to account for transactions from that interval that have already finished. The density it finds for this interval will be exactly $den(CI_m)$, since the work of the already-completed transactions in CI_m was done at rate $den(CI_m)$. Thus, POLARIS chooses $den(CI_m)$ as the execution frequency for transaction $n+1$.

Case 2: Suppose than the nth transaction belongs to CI_m and the $n+1$st belongs to CI_{m+1}. In this case, when transaction n finishes and POLARIS runs SetProcessorFreq, the set of transactions remaining at the processor is exactly those in P_{m+1}. Furthermore, transaction $n+1$ has the earliest deadline of all transactions in P_{m+1}. Thus, by the same argument used in the base case, both OA and POLARIS choose $den(CI_{m+1})$ as the processor speed for transaction $n+1$. □

Next, we consider the situation in which the newly arriving transaction t_{new} has an earlier deadline than the running transaction t_r. In such a situation, OA will *preempt* t_r and start running t_{new}. POLARIS , which is non-preemptive, cannot do this. Instead, POLARIS will continue to run t_r, but will increase the speed of the processor to ensure that both t_{new} and t_r finish by t_{new}'s deadline. This is captured by the following lemma:

Lemma 3.4.2. *Suppose that both POLARIS and OA have the same queue at a point in time, with k total transactions, one running (t_r) and the rest waiting, with the exact same loads. Suppose a new transaction t_{new} arrives,and that $d(t_{new}) < d(t_r)$. Until the arrival of the next transaction, POLARIS will execute transactions in the same order, and with the same processor frequency, as OA would have if $d(t_r)$ were decreased to $d(t_{new})$.*

Proof. The proof is similar to that of Lemma 3.4.1. In the modified problem instance in which the deadline of t_r is reduced, no other transactions have deadlines earlier than t_r and t_{new}. Thus, there are two possibilities for CI_1, the first critical interval chosen by OA. Either it includes only t_r and t_{new}, or it includes t_r, t_{new}, and some additional transactions. In the former case, $den(CI_1) = (w(t_r) + w(t_{new}))/d(t_{new})$. In the latter case, it is higher.

Now consider POLARIS. When t_{new} arrives, POLARIS keeps executing t_r since it is non-preemptive. However, it runs SetProcessorFreq to adjust the processor frequency. Because of the definition of $\hat{q}(t, f)$, the miminum frequency identified for each transaction includes the (remaining) time for t_r, even if t_r has a later deadline. Thus, SetProcessorFreq will identify frequency $(w(t_r) + w(t_{new}))/d(t_{new})$ when it checks t_{new}, and will set this frequency if CI_1 includes just t_r and t_{new}. If CI_1 includes more transactions, SetProcessorFreq will find $den(CI_1)$ when it checks the last transaction in CI_1. □

3.4.5 Competitive Ratio of POLARIS

We next prove POLARIS' competitive ratio against OA and YDS both on *arbitrary* and *agreeable* instances. Arbitrary problem instances are those in which transactions can have arbitrary loads, arrival times, and deadlines. Agreeable instances are those in which transactions have arbitrary loads but their arrival times and deadlines are such that for any pair of transactions t_i and t_j if $a(t_i) \leq a(t_j)$ then $d(t_i) \leq d(t_j)$. Intuitively, agreeable problem instances capture workloads in which sudden short deadline transactions do not occur. Throughout the rest of the section, $Pow[POLARIS(P)]$ and $Pow[YDS(P)]$ denote the power consumed by POLARIS and YDS on a problem instance P, respectively.

We next make a simple observation about POLARIS' competitive ratio on agreeable problem instances.

Theorem 3.4.3. *Under agreeable problem instances* $Pow[POLARIS(P)] \leq \alpha^\alpha Pow[YDS(P)]$. *Therefore POLARIS has* α^α *competitive ratio against YDS and therefore* OPT_{np}.

Proof. Recall from Section 3.4.4 that the only difference in the behaviors of OA and POLARIS is when a new transaction with the earliest deadline in the queue arrives. Since this never happens in agreeable instances, POLARIS behaves the same as OA, which has a competitive ratio of α^α with respect to YDS [21]. □

Next we analyze POLARIS' competitiveness on arbitrary problem instances. In the rest of this section, given an arbitrary problem instance P, we let w_{max} and w_{min} be the maximum and minimum loads of any transaction in P. Let $c = (1 + \frac{w_{max}}{w_{min}})$. Given a problem instance $P = t_1, ..., t_n$, let $P' = t'_1, ..., t'_n$ be the problem instance in which each t_i and t'_i have the same arrival times and deadlines, but $w(t'_i) = c \times w(t_i)$. Essentially P' is the problem instance where we keep the same transactions as P but increase their loads by a factor of c. Our analysis consists of two steps.

Theorem 3.4.4. $Pow[POLARIS(P)] \leq \alpha^\alpha Pow[YDS(P')]$

Proof. Our proof is an extension of the proof used by Bansal et al. to show that OA has an α^α competitive ratio against YDS [21], and is provided in Appendix A. □

We next show that YDS on P' consumes exactly c^α times the power it does on P.

Theorem 3.4.5. $Pow[YDS(P')] = c^\alpha Pow[YDS(P)]$.

Proof. Since the load of each transaction increases by a factor of c, YDS on P' will find exactly the same set of critical intervals, but with c times larger densities. Therefore, YDS' processor speed on P' will be a factor c faster than on P at any moment. Let $s(t)$ be the processor speed of YDS on P. Since $\int_t (cs(t))^\alpha = (c^\alpha) \int_t s(t)^\alpha$, YDS will consume exactly c^α more energy on P' than P. □

The next corollary is immediate from Theorems 3.4.4 and 3.4.5.

Corollary 3.4.6. *POLARIS has a* $(c\alpha)^\alpha$ *competitive ratio against YDS and therefore* OPT_{np}.

3.4.6 Discussion of Competitive Ratio Analysis

The competitive ratio in Corollary 3.4.6 has two components: α^α and c^α. Recall that (idealized) POLARIS has two disadvantages against YDS. First, it does not know the future, and second it cannot preempt transactions. Recall that the OA algorithm, which does not know the future but can preempt transactions, has α^α competitive ratio [21]. Thus, one interpretation is that the α^α component captures POLARIS' disadvantage of not knowing the future. In contrast, the c^α component captures POLARIS' disadvantage of not being able to preempt. For an example of this disadvantage, consider two transactions t_1 and t_2. t_1 has load w_{max} and arrives at time 0 and has a very late deadline. t_2 has a load w_{min} and arrives after an infinitesimally small time after 0, and has a very short deadline. POLARIS will start t_1 will receive t_2 and will finish both t_1 and t_2 by the deadline of t_2. Instead YDS would execute t_2 first and then t_1. By appropriate choices of the deadlines for t_1 and t_2, POLARIS will perform c^α worse than YDS.

3.5 POLARIS Prototype

To test POLARIS, we implemented it in Shore-MT [95]. Shore-MT is a multi-threaded data storage manager which is designed for multiprocessors. Shore-Kits [1] provides a front-end driver for Shore-MT. It includes implementations of several database management systems benchmarks, including TPC-C and TPC-E. For the remainder of this chapter, we refer to the combination of Shore-Kits and Shore-MT as Shore-MT.

Shore-MT has multiple worker threads, each with an associated request queue. Each request corresponds to a transaction of a particular type, e.g., NewOrder in the TPC-C workload. Each worker sequentially executes requests from its queue, using the storage manager to access data.

There are also request handling (RH) threads that handle incoming requests from clients and routes them to worker queues. To simplify our experimental setup, we do not drive the Shore-MT server using remote clients. Instead, a request handler simulates a set of remote clients by generating randomized requests and then handling them as if they had arrived over the network from remote clients.

Our test server's multi-core CPUs allow CPU frequency to be controlled separately for each core. In our prototype, we fix the number of workers to match the number of cores in our server and pin each worker to a single core. We run a separate POLARIS instance for each core, which manages the request queue of that core's worker and controls the core's execution frequency.

POLARIS requires action when two types of events occur: the arrival of a new transaction request, and completion of a request. In our prototype, RH threads handle the POLARIS's request arrival action. When a new request arrives, one of the RH threads enqueues the request to one worker queue and then runs the POLARIS SetProcessorFreq algorithm (Figure 3.2) to adjust the execution frequency of that worker's core. We modified Shore-MT's request queues so that requests are queued in EDF order, as required by POLARIS. The worker threads handle POLARIS's request completion action. On completion of a request, workers pull the earliest-deadline request from their queues and run SetProcessorFreq to set their core's frequency before executing the dequeued request.

POLARIS's overhead depends on the length of the request queue. The longer the queue, the higher the overhead. At high load, when queues are longest, we measured its execution time at about 10 microseconds, which is one or two orders of magnitude less than the mean execution times, at peak frequency, of the transactions in our TPC-C workload.

The POLARIS SetProcessorFreq function requires some means of actually adjusting a core's P-State. As noted in Chapter 2, there are several mechanisms for doing so in which all of the alternatives ultimately rely on MSRs for x86 processors. Since POLARIS adjusts execution frequencies frequently (potentially on each transaction request arrival or completion), the RH and worker threads in our prototype modify the MSRs directly via the MSR driver, which is much faster [169].

3.6 Evaluation

We use our prototype to conduct an empirical evaluation of POLARIS. The primary goal of our evaluation is to compare POLARIS against low-level, OS frequency governors. We want to determine whether the extra information available to POLARIS leads to greater power savings than can be achieved with the OS baselines. We test POLARIS under a variety of load conditions. In addition, we test POLARIS's ability to differentiate among concurrent workloads with different latency targets.

3.6.1 Methodology

In our experiments, we use Shore-Kits' TPC-C and TPC-E implementations. For both benchmarks, Shore-MT's buffer pool is configured to be large enough to hold the entire database. For each experimental run, we choose a method for controlling core frequencies (POLARIS, or one of the baselines), and then run the benchmark workload against

our Shore-MT prototype. Each run consists of three phases: (1) a *warmup* phase, during which each worker executes 30,000 transactions, (2) a short *training* phase for warming up POLARIS' execution time estimators by filling the initial sliding window for each frequency level and workload type combination, and (3) the *test* phase, during which power consumption and system performance are measured.

The training phase is used only so that we can test POLARIS in a state in which its estimation model has been fully initialized. In practice, the execution time estimates for all workloads at all frequencies can be initialized to zero. This will cause POLARIS to gradually explore and initialize its estimators for unexplored frequencies, from lowest to highest, as it encounters load conditions under which the already-explored frequencies are not fast enough to handle the load. POLARIS performance may suffer as it initializes these estimators, but this is a transient effect, and the number of estimators is relatively small ($\mathcal{W} \times \mathcal{F}$).

We change Shore-Kits request generation from a closed-loop design to an open-loop design, so that we can specify a mean offered load (transaction requests per second) for the system for each experiment. Request interarrival delays are chosen randomly from a uniform distribution with the mean determined by the target request rate, a minimum of zero, and a maximum of twice the mean. Thus, the actual instantaneous request rate fluctuates randomly around the target. We run experiments at three target load levels: high, medium, and low. High load is 90% of the peak throughput for our test system, which is about 21250 transactions per second for TPC-C, and 14900 transactions per second for TPC-E. The medium and low loads correspond to 60% and 30% of the peak throughput, respectively.

In addition to these "steady" loads, we use World Cup site access traces [16] to generate TPC-C workloads with time-varying target request rates. To do this, we vary the target request arrival rate between 30% and 90% of the peak TPC-C throughput for our server to match the observed normalized fluctuations in the World Cup trace. The target rate is adjusted once per second.

For each experiment, transactions are assigned to one or more workloads, each with an associated latency target. We use the notion of *slack* to provide a uniform way of describing the tightness of the latency targets. We define slack as the ratio between a workload's latency target and the mean execution time of the workload's transactions, at the highest processor frequency. For example, for a TPC-C New Order transaction, which has an average execution time of 2059 μs (recall Figure 3.3) at the highest frequency level, a slack of 20 indicates latency target of 41180 μs. We experiment with slack values ranging from 10 to 100 to illustrate the effect of the tightness of latency targets on the algorithm's

behavior.

For each run, we measure the average power consumed by the server during the test phase. To measure server power draw, we used a Watts up? PRO [82] wall socket power meter, which has a rated ±1.5% accuracy. We measure the power consumption in one-second intervals (the finest granularity of the power meter) and average those over the test duration. We also measure the power consumption of the CPUs (alone), as reported through the RAPL MSRs. However, we use the *whole server power*, as reported by the Watts up? meter, as our primary power metric.

In addition to the power metric, we also measure performance during the test phase. In each of our experiments, the mean system throughput is fixed and controlled by our open-loop request generator. Thus, we are primarily interested in transaction latency. Specifically, we measure the percentage of transactions that do not finish execution before their deadline, which we refer to as the *failure rate*.

We run experiments with POLARIS and with several operating system baselines:

Dynamic Kernel Governors: In these tests, Shore-MT uses its default transaction scheduling and did not control core frequencies. Instead, we use the Linux `cpufreq` dynamic governors to manage core frequencies. We experiment with two dynamic governors: *Conservative* and *OnDemand*. The former favors performance over power savings, while the latter adjusts core frequencies more aggressively to save power.

Static Frequencies: In these tests, Shore-MT uses its default transaction scheduling and does not control core frequencies. Instead, we use MSRs to set all cores to run at a fixed frequency.

In our experiments, we use a server with two Intel® Xeon® E5-2640 v3 processors with 128 GB memory using Ubuntu 14.04 with kernel version 4.2.8, where the `cpufreq` driver is loaded by default. For the experiments with in-DBMS power scheduling algorithms and those with the static frequencies, we disable the CPU ACPI software control in the BIOS configuration to prevent the `cpufreq` driver from interfering with power control. For the experiments using the dynamic kernel governors, we enable ACPI software control in the BIOS. To reduce non-uniform memory access (NUMA) effects and get more homogeneous memory access patterns, we enable memory interleaving in the BIOS.

Each E5-2640 CPU has 8 physical and 16 logical cores (hyper-threads), thus our system has a total of 16 physical (32 logical) cores. Each physical core's power level can be set separately. The CPU has 15 frequency levels from 1.2 GHz to 2.6 GHz with 0.1 GHz steps,

plus 2.8 GHz. In our experiments, we chose five of the frequency levels, 1.2, 1.6, 2.0, 2.4 and 2.8 GHz, as the possible target frequency levels for POLARIS.

For all of our experiments, our Shore-MT prototype is configured to use two Request Handler (RH) threads and sixteen worker threads. We pin each worker thread to a one logical core (hyperthread) in one of the 16 physical cores. The RH threads are free to run on any of the remaining logical cores, as determined by the kernel's thread scheduler. Each RH thread distributes requests to the workers round robin, regardless of the requests' workload types. For TPC-C, we set the database scale factor to 48 and for TPC-E, we use a database with 1000 customers and set the benchmark's working days and scale factor parameters to 300 and 500, respectively, which are given as their default values in the TPC-E specification [45]. We use Shore-MT's default staged group commit configuration, under which log I/O is forced at least once per 100 transactions, and we observe multiple log flushes throughout the test phase of the experiments.

3.6.2 Results: Medium Load

We consider both TPC-C and TPC-E workloads. We begin with TPC-C, and present TPC-E in Section 3.6.2.

For TPC-C, we define four workloads for POLARIS, one corresponding to each of the four TPC-C transactions implemented by Shore-Kits. For each transaction, the target latency is set to *slack* times the mean execution time (at high frequency) for that transaction's workload type. These mean execution times ranged from about 0.25 milliseconds for Order Status transactions to about 3.4 milliseconds for Stock Level as we show in Figure 3.3. Thus, when the slack is 50 (for example), the latency target for the Order Status transaction workload is set to about $0.25 * 50 = 12.5$ milliseconds, and the target for Stock Level transactions is $3.4 * 50 = 170$ milliseconds. We vary *slack* in the range from 10 to 100.

TPC-C Medium Load

Figure 3.5 shows the results of this experiment, as a function of slack. In addition to POLARIS, we report the results for the two Linux dynamic governors (OnDemand and Conservative), as well as the results for two highest static frequency governors.

In this test, running all cores at the highest frequency (2.8 GHz) causes the server to consume about 170 watts of power. When slack is tight, about 15% of transactions exceed

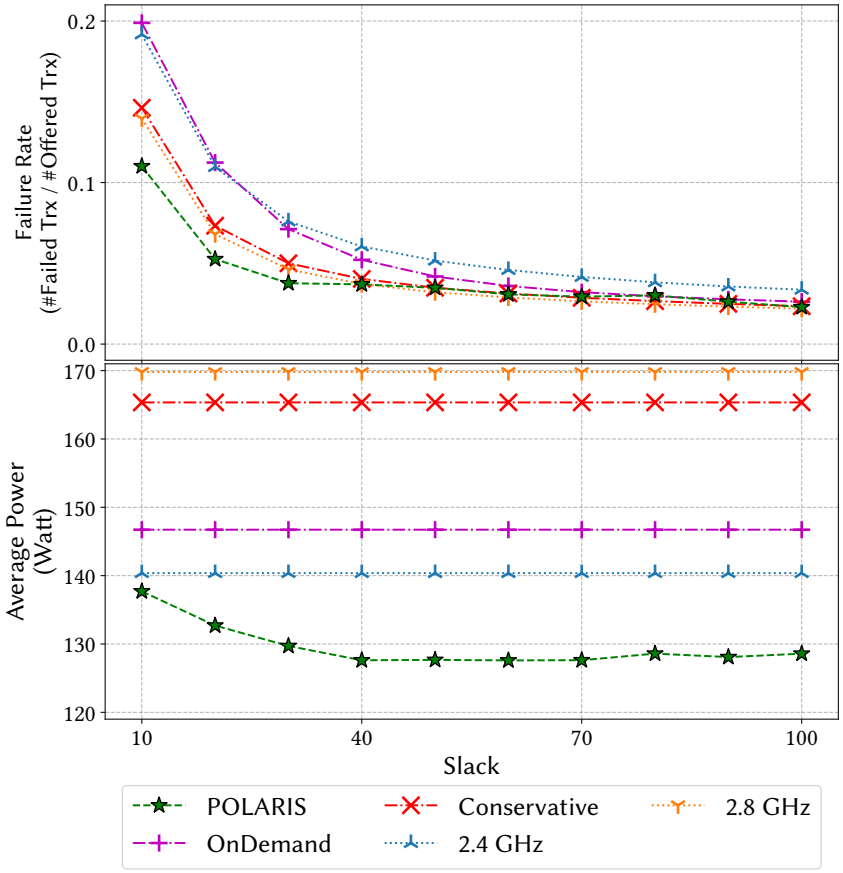


Figure 3.5: Performance and power of different power management schemes under medium load, as functions of slack (S).

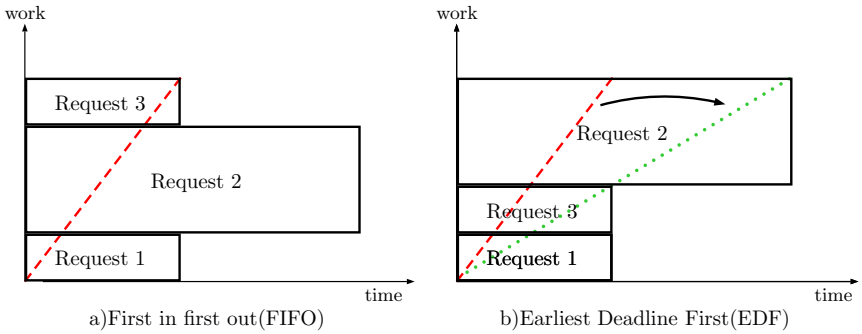


Figure 3.6: An example illustrating the impact of FIFO vs EDF scheduling on frequency selection.

their latency targets. Moving to a lower static frequency (2.4 GHz) results in almost 30 watts of power savings, but at the expense of more missed latency targets.

In this setting, the Linux Conservative governor's behavior is similar to that of the static, high-frequency governor. Indeed, the Conservative governor rarely lowers frequency below 2.8 GHz in these experiments. The Linux OnDemand governor reduces core frequencies more aggressively. This produces power savings, but at the expense of more missed latency targets when slack is tight.

POLARIS performs better because it is deadline-aware. With tight slack, POLARIS lowers power consumption by about 40 watts relative to consumption at peak frequency - about 15 watts more than the OnDemand governor. These power savings do *not* come at the expense of missed latency targets. Indeed, when slack is tight, POLARIS misses *fewer* latency targets than the high-frequency (2.8 GHz) static governor. This is because POLARIS is able to re-order transactions and run them in EDF order, which the static governors cannot do.

Figure 3.6 shows a three-transaction example which illustrates the impact of request reordering. Each rectangle represents a transaction request. Rectangle height indicates the amount of work required to complete the request, and width indicates the request deadline. We assume that the large transaction arrives second. The scenario on the left shows FIFO ordering, and the slope of the dashed line represents the execution frequency chosen by a deadline-aware algorithm, like POLARIS. The scenario on the right shows EDF execution of the same transactions, and the reduced execution frequency that results.

As slack increases, POLARIS produces *greater* power savings, since it reduce processor frequencies to take advantage of the extra slack. The baselines are unaware of slack, and hence are insensitive to it. In loose-slack settings (slack greater than 50), POLARIS reduces

total server power by about 40 watts relative to peak frequency, almost twice the reduction achieved by the OnDemand governor.

TPC-E Medium Load

For the TPC-E medium load experiment, we define ten POLARIS workloads, each corresponding to one TPC-E request type. Mean execution times for requests range from 0.06 to 2.3 milliseconds at peak frequency. We use slack to assign a latency target for each workload, as for TPC-C.

Figure 3.7 shows the results of the TPC-E experiment, which are similar to those for TPC-C. POLARIS reduces power consumption by about 40 watts relative to peak frequency execution. As for TPC-C, the power savings are greater with greater slack, although the effect is not as strong. The operating system's OnDemand governor does better (relative to POLARIS) than it did for TPC-C, but it still consumes more power and misses more transaction deadlines than POLARIS.

One difference between the TPC-E and TPC-C results is that, for very tight slack, POLARIS's rate of missed latency targets is higher than that of the Conservative governer. However, this is achieved at the cost of about 35 watts.

3.6.3 Results: Effect of Load

To investigate the effects of system load on POLARIS, we repeat our medium-load TPC-C experiment under low and high load conditions. Low load means an average request arrival rate of 30% of the systems peak sustainable load, while high load corresponds to 90% of peak.

TPC-C Low Load

Figure 3.8 shows the results of this experiment under low load. POLARIS results in power savings of about 40 watts, relative to execution at peak frequency. This is similar to the savings that were achieved at medium load. In this setting, the Conservative governor is able to achieve the same power savings as POLARIS, but it does so at the expense of significantly higher rates of missed latency targets when slack is tight. The OnDemand governor has in-between performance, and is dominated by POLARIS.

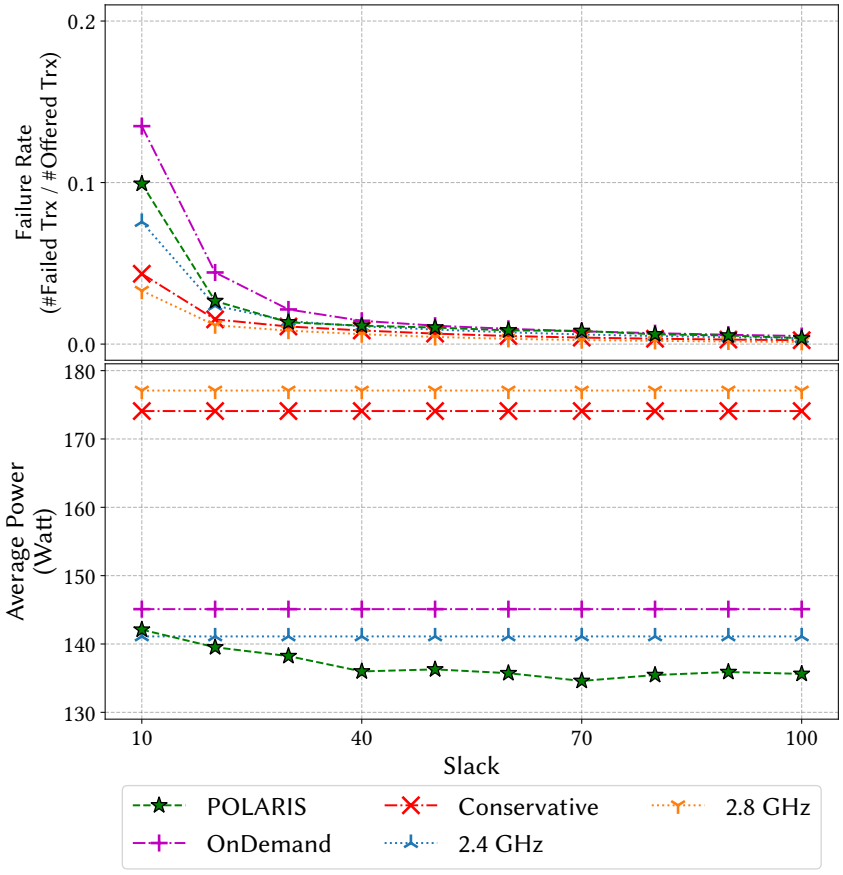


Figure 3.7: TPC-E performance and power of different power management schemes under medium load, as functions of slack (S).

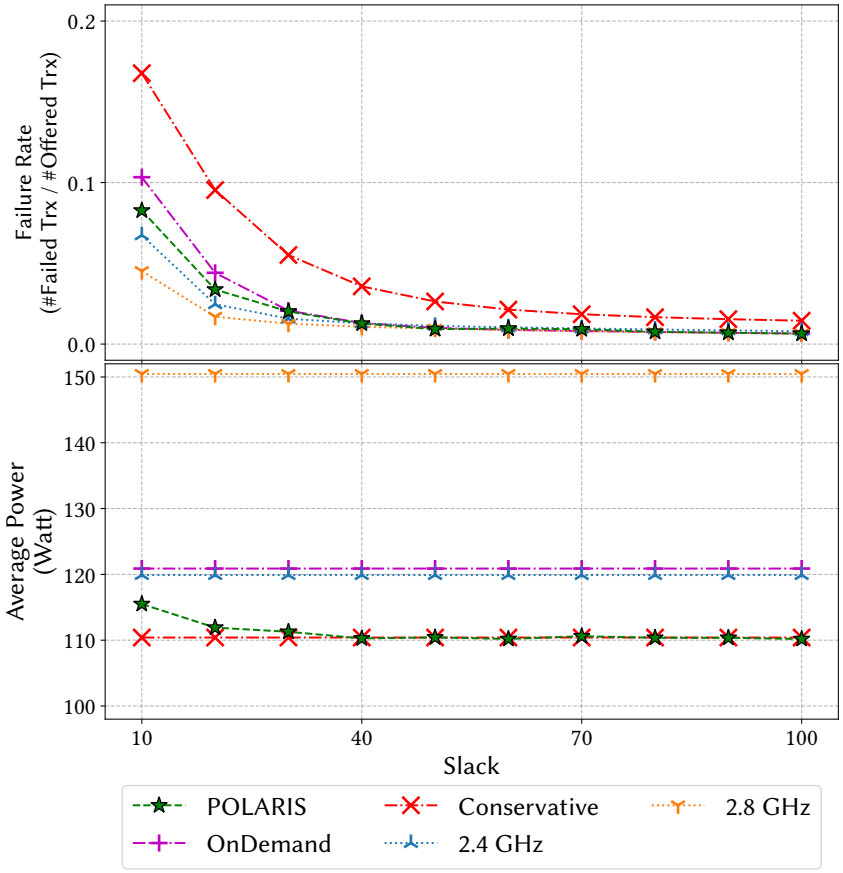


Figure 3.8: Performance and power of different power management schemes under low load, as functions of slack (S).

A comparison of the medium and low load experiments (Figures 3.5 and 3.8) shows that the two baseline dynamic governors switch roles in these two settings. At medium load, the OnDemand governor results in lower power consumption but more missed latency targets than Conservative, which rarely leaves the highest frequency. However, at lower load, it is the Conservative governor that results in greater power savings but more missed latency targets. This illustrates the challenges of relying on low-level metrics, like processor utilization, to achieve latency targets. POLARIS, in contrast, has stable behavior in both settings.

TPC-C High Load

Finally, Figure 3.9 shows the results of the high-load experiments. This is a challenging setting for both POLARIS and the baselines, as there is little opportunity for power optimization under such an intense workload.

All of the methods, including POLARIS, have higher rates of missed latency targets, especially when those targets are tight. This is simply because there are periods when requests come in too fast for the system to handle, even at peak frequency. Under high load, both POLARIS and the OnDemand governor are able to reduce power only by about 10 watts (relative to peak frequency), although POLARIS does so with fewer missed latency targets.

As we note in Chapter 1, real systems may experience both longer term and shorter term load fluctuations. Our results with low, medium, and high load experiments suggest that POLARIS can function effectively as load fluctuates over the longer terms. When load is in the low or medium range, which is common, POLARIS can reduce power substantially without compromising latency targets. During windows of peak load there is little opportunity for power savings, but POLARIS performs at least as well as running the processors at peak frequency in that setting.

3.6.4 Results: Time-Varying Load

In our previous experiments, we test with workloads that exhibit random fluctuations around a steady average request rate. In our next experiment, we consider a workload in which the average request rate fluctuates to match the request trace of a real application. We use a World Cup trace [16] to generate the request rate fluctuations.

Specifically, we vary the target TPC-C request rate in the range from 6400 transactions per second to 19440 requests per second. (These rates correspond to our steady "low" and

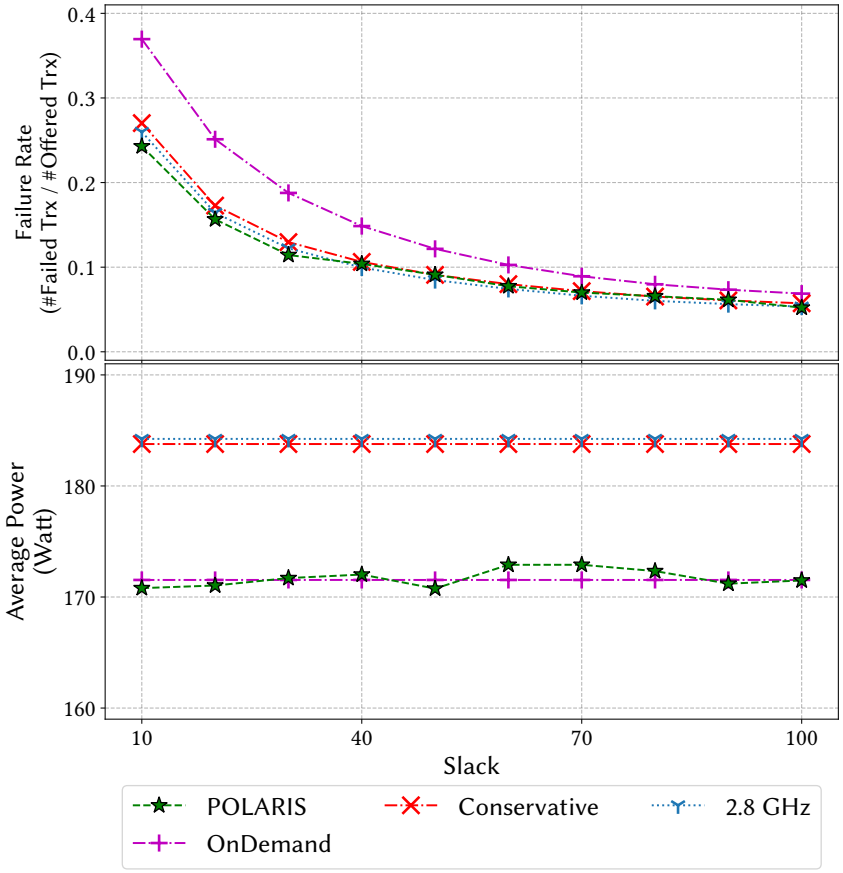


Figure 3.9: Performance and power of different power management schemes under high load, as functions of slack (S).

"high" workload levels.) We set a new target rate every second, according to the (normalized) request rate from the World Cup trace. Otherwise, the experimental configuration is identical to the configuration we used for the steady load TPC-C experiments.

Figure 3.12(a) illustrates the normalized request rate we generated, as well as the power consumption of POLARIS and the Conservative and OnDemand baselines. Power consumption is normalized to the minimum and maximum consumption (of any algorithm) observed during our experiments, so that the reported values are comparable across algorithms. Figure 3.12(b) summarizes the average power consumption and failure rate (percentage of transactions that missed latency targets) over the entire experiment. As is the case in the steady load experiments, POLARIS results in both lower power consumption and fewer missed latency targets than either of the operating system benchmarks. All of the algorithms adjust power consumption in response to the load changes, but POLARIS's adjustments tend to be sharper and deeper.

Energy Efficiency

So far, we have used two metrics (power and transaction failure rate) to evaluate the behavior of POLARIS and the baselines. Figure 3.13 summarizes and restates these results using a single, combined efficiency metric that reflects both power consumption and transaction latencies. This metric is the mean number of *successful* transactions completed per joule of energy consumed. Under this metric, the energy consumed by transactions that miss their latency targets is considered to have been wasted.

Figure 3.13 shows that POLARIS is at least as energy efficient as all of the baselines at all load levels and all slack levels. The figure shows that energy efficiency is higher at higher load levels. This is a consequence of the fact that processors are not power proportional, and it has been noted by other researchers [23]. It can also be observed in SPECpower_ssj2008 server benchmark results [158]. The figure also shows that POLARIS' efficiency advantage over the baselines is greatest at medium loads, and at tight (low) slack levels. Medium loads allow POLARIS to utilize the entire dynamic power range of the processor. At low loads, both POLARIS and the baselines are limited in their ability to improve efficiency by the processor's lower bound on execution speed. The maximum speed of the processor is similarly limiting when load is high.

CPU Utilization

Another way to think of processor frequency scaling is as a kind of fast, fine-grained capacity provisioning mechanism. Increasing the processor frequency increases its capacity to do

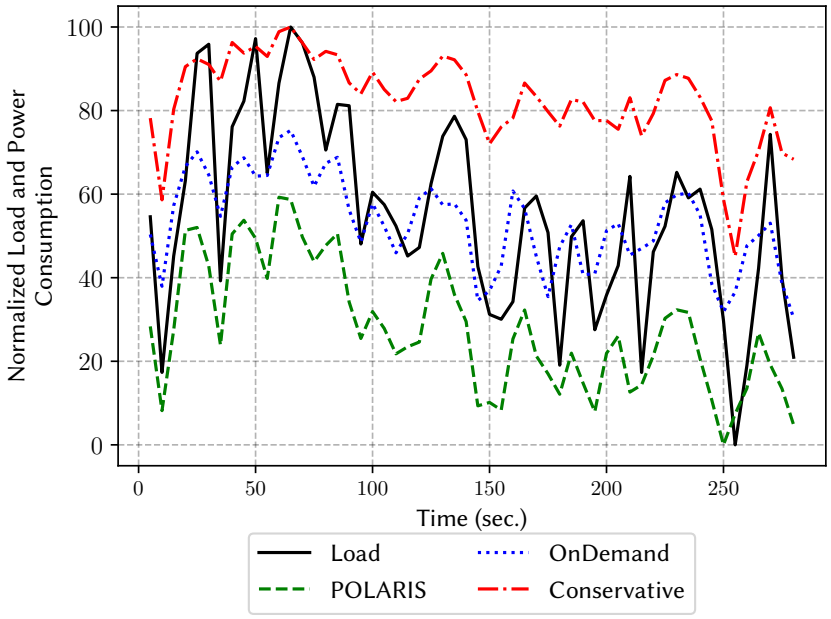


Figure 3.10: World Cup Trace Timeline for Normalized Load Level and power consumptions, bins of 5 seconds.

Baseline	Avg. Power (Watt)	Failure Rate
Conservative	168.9	0.09
OnDemand	152.9	0.13
POLARIS	139	0.07

Figure 3.11: Average Power consumption and failure rate of baselines in World Cup Trace

Figure 3.12: World Cup Trace - Normalized.

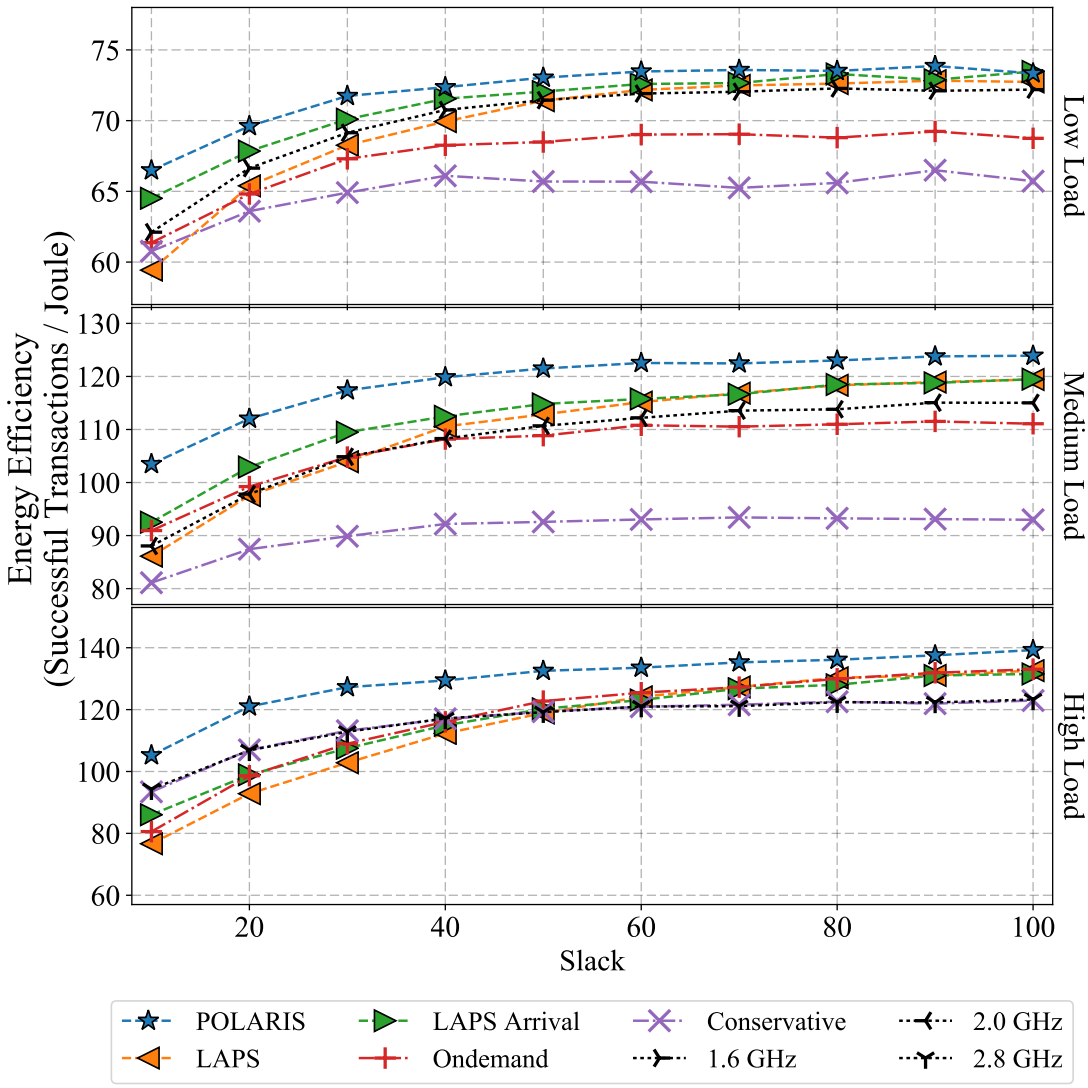


Figure 3.13: Successful transactions per joule, as a function of load and slack.

work, at a cost of increased power consumption. Reducing frequency reduces capacity. From this perspective, the role of frequency scaling algorithms, like POLARIS and the baselines, is to reduce processor capacity as much as possible without causing transactions to miss their latency targets.

By measuring CPU utilization, we can quantify algorithms' success at adjusting processor capacity. Ideally, with a perfect frequency scaling technique and processor with a wide range of possible frequencies, we would see CPU utilization approaching 100%. Figure 3.14 reports the actual CPU utilization we observed for POLARIS and the baseline algorithms at all load and slack levels.

At low load, the utilization of the processor is less than 60% even at the lowest processor frequency (1.2 GHz). Higher utilizations (and greater power savings) would require the ability to reduce execution frequency below 1.2 GHz. With sufficient slack, POLARIS and the in-DBMS baselines are able to approach this limit, indicating that they are achieving the maximum possible capacity (and hence power) reductions on this processor. In contrast, the in-kernel baselines have lower utilizations. At high load, the figure shows that all of the frequency scaling algorithms have little room to maneuver, as processor utilization is barely below 80% even at peak frequency. Medium load, however, allows plenty of room for capacity adjustment, with processor utilization varying from about 50% at the highest frequency to almost 100% at the lowest. All of the in-DBMS algorithms, including POLARIS, are much more effective than the in-kernel baselines at driving down frequency and increasing utilization.

3.6.5 Results: Workload Differentiation

In this experiment, we focus on how POLARIS and the baselines react when the are multiple similar workloads with different latency targets. For this purpose, we define two TPC-C workloads, each consisting of all four types of TPC-C transactions, in the standard proportions. Requests for each workload are generated at half of our medium TPC-C workload rate, so that the total load (on average) is equivalent to our TPC-C medium load. For one workload, which we refer to as *gold*, we set a latency target of 7.5 milliseconds. For the other, which we refer to as *silver*, we set a latency target of 37.5 milliseconds. We track the failure rate (late transactions) separately for the gold and silver workloads.

Figure 3.15 shows the failure rate for each workload, under POLARIS, the Linux dynamic governors, and the high frequency static governor. Each failure rate is plotted against the total power consumption for that run, as we cannot separately attribute power to individual workloads.

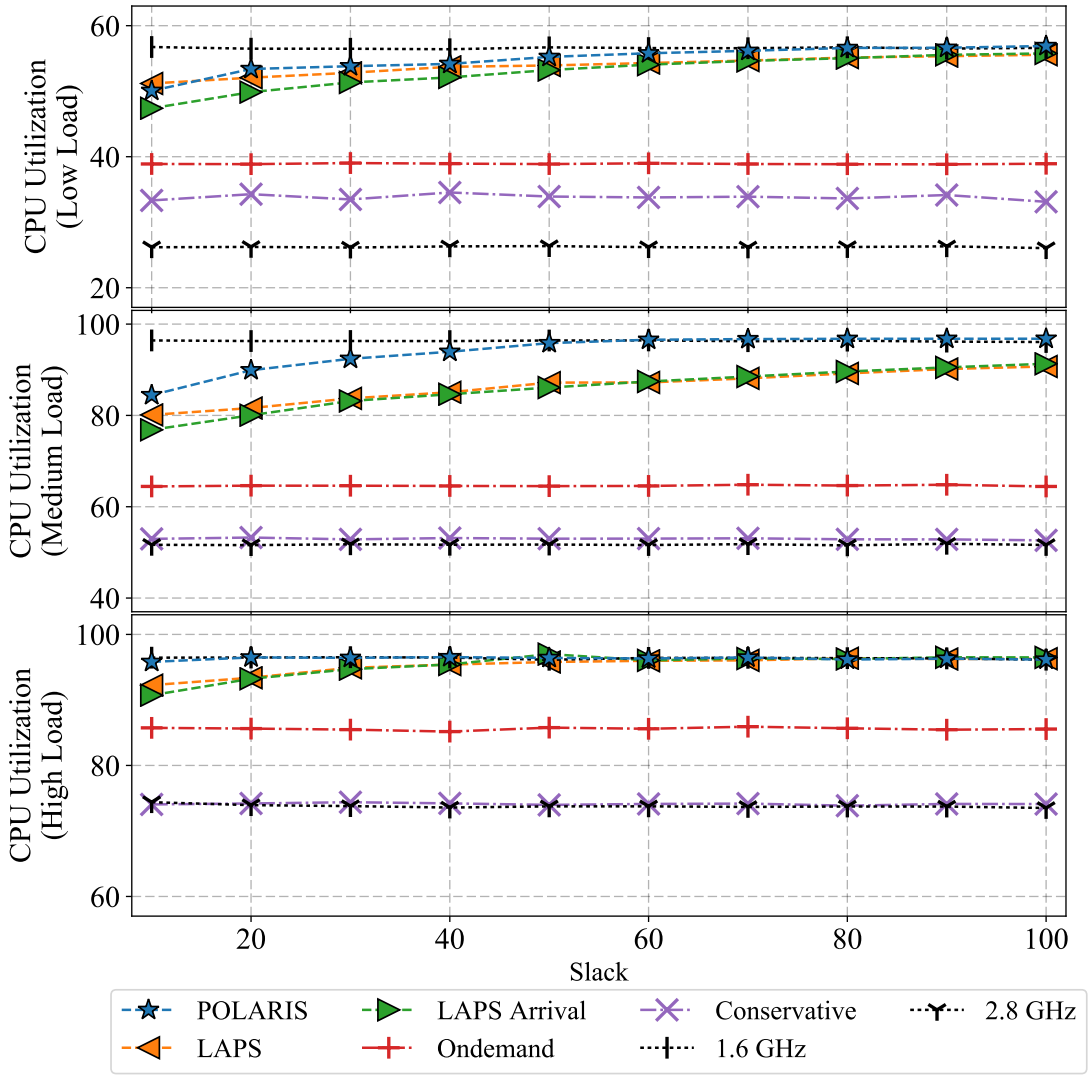


Figure 3.14: CPU Utilization under various load levels

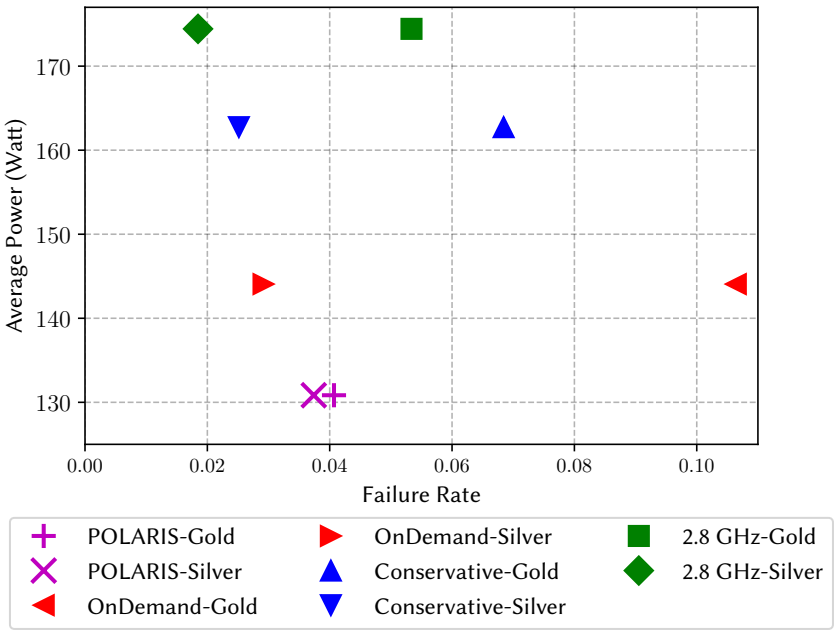


Figure 3.15: Per-Workload Performance for Gold and Silver TPC-C Workloads

Non-POLARIS managers have a large gap between the failure rates of gold and the silver, as they are not able to take SLA into account. Thus, gold requests fail more because of their tighter latency target. POLARIS, because it is deadline aware, produces similar failure rates for both workloads. Gold transactions are much less likely to miss their latency targets, while silver transactions are slightly more likely.

3.6.6 POLARIS Component Analysis

In our final experiment, we evaluate the importance of different aspects of POLARIS by comparing it to two variants. The first, POLARIS-FIFO, is identical to POLARIS but runs transactions in FIFO order, rather than EDF. The second, POLARIS-FIFO-NOARRIVE, runs transactions in FIFO order and adjusts frequency only on transaction completion, not on arrival. Figure 3.16 shows the power and performance of POLARIS and the variants for TPC-C under medium load.

The results show that both EDF and frequency adjustment on arrival are important for achieving latency targets when slack is tight. The latter allows POLARIS to react quickly to the arrival of new transactions by increasing frequency when necessary. They also show that EDF contributes to power savings, because it allows POLARIS to meet latency targets with lower frequencies.

3.7 Conclusion

In this chapter, we have presented a workload-aware frequency scaling and scheduling technique for latency-critical data systems, and related it to other well-known off-line and on-line algorithms.

Unlike operating system power governors, POLARIS is aware of per-transaction latency targets and takes advantage of them to keep processor execution frequency, and hence power consumption, as low as possible. On our server, POLARIS was able to reduce power consumption substantially, with no increase in missed transaction deadlines. Operating system governors, in contrast, either save little power or save power at the expense of missed deadlines. Through comparison of several variations of POLARIS, we showed that it is necessary for POLARIS to control transaction execution order *and* processor frequency to achieve this performance.

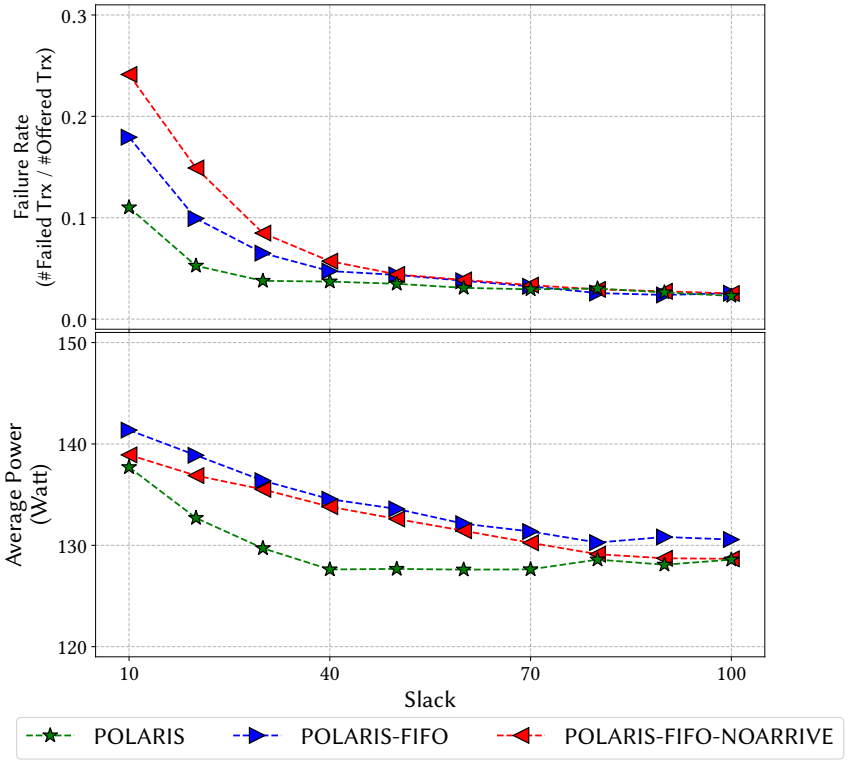


Figure 3.16: Performance POLARIS and Variants

Chapter 4

Multi-Processor Energy Aware Scheduling

4.1 Overview

In Chapter 3, our focus is on single processor routing, and we use POLARIS at each core to simultaneously control the transaction execution order and the core's execution frequency. In this chapter, we consider the same problem in a more general setting. We focus on multi-processor energy-aware scheduling in modern servers that employ multiple multi-core processors [152].

As in Chapter 3, our target is an in-memory database system that works on a server with multiple homogeneous processors that can operate at different frequencies. The system accepts sporadic requests with arbitrary deadlines. Each request belongs to a workload type, and each workload has a latency target for its transactions. The system executes requests non-preemptively, and no migration is allowed across the processors [48]. Our problem is to decide which CPU core will execute each transaction, and in which order and at what speed they will be executed. The primary goal is to meet transactions' latency targets. The secondary goal is to use as little energy as possible to execute the transactions.

In this multi-processor, multi-core setting, a scheduler must address several new problems in addition to those we considered in Chapter 3. Specifically, it must consider which and how many cores to use, and it must consider how to distribute transaction executions across those cores. In Chapter 3, we assumed simple round-robin (RR) distribution of transactions across all of the cores. Here, we consider energy-aware strategies for alloca-

tion (which and how many cores to use) and transaction routing. We provide overviews of these two problems in Sections 4.3 and 4.4.

In Section 4.5, we then present an on-line workload-aware scheduling and frequency scaling algorithm called *PLASM* (POwer and Latency Aware Request Scheduling in Multi-processor CPUs). PLASM controls request execution order, processor speed and routing of requests to processors to minimize CPU power consumption while observing per-workload latency targets.

We evaluate a prototype implementation of PLASM under a variety of load conditions. Our results (Section 4.6) show that PLASM produces greater power savings and fewer missed transaction deadlines than POLARIS with RR routing. We also show how PLASM's effectiveness is affected by two key factors: (1) the average load on the system, and (2) scheduling slack, i.e., the looseness of the transactions' deadlines.

4.2 Related Work

There is a body of existing work related to energy-efficient multi-processor scheduling. Many variations of this problem are known to be NP-hard [38], including the one we focus on in this chapter.

Much of the existing work focuses on offline settings in which all of the requests are known in advance. Albers et al.[11] present offline scheduling solutions for the preemptive execution model. They show that RR is an optimal offline algorithm for unit size requests with agreeable deadlines, which means that requests with earlier arrival times have an earlier deadlines. For requests of arbitrary size, the authors present an algorithm called Earliest Deadline and List scheduling (EDL). EDL first orders the request according to the Earliest Deadline First (EDF) policy and then assigns the requests to the least loaded worker. They show EDL's approximation ratio for the case where all the requests have a common arrival time with arbitrary deadlines. They also provide an approximation ratio for CRR (Classified Round Robin) for arbitrary size requests with agreeable deadlines. CRR classifies requests according to their density, which corresponds to a request's work divided by its relative deadline. After that, CRR uses a separate RR order for each class to distribute requests across the processors. Similar to CRR, Bell et al. [27] proposes another grouped RR technique called DCRR, which prevents adversarial cases in the original CRR algorithm [11] by grouping the request according both density and size. In a variation of the classical problem where preempted requests can migrate across the processors, Albers et al. [8] show that the optimal schedule can be computed in polynomial time. The CRR

and EDL algorithms assume that each individual processor uses the YDS algorithm [182] to schedule the requests that have been routed to it. As described in Chapter 3, YDS offers an optimal offline solution for single processor preemptive scheduling.

For offline *non-preemptive* multiprocessor scheduling where each processor can have a different convex speed to power relation(heterogeneous), Cohen-Addad et al. [42] present offline algorithms for both arbitrary and unit-size tasks. The algorithm divides all the jobs into sets where no two jobs in a set's life span (arrival to deadline) intersect to transform the problem into preemptive heterogeneous multiprocessor scheduling. They then use a randomized routing algorithm [20] to solve the transformed problem.

Some existing research focuses on online multi-processor scheduling. Albers et al.'s study [11] provides competitive ratio analyses of online versions of RR (RR-ON) for unit size requests with agreeable deadlines and CRR (CRR-ON) for requests with arbitrary size and agreeable deadlines. Greiner et al. [65] give approximation bounds for any preemptive single-processor energy-efficient scheduling algorithms in a multi-processor setting, assuming random routing. CRR-ON and RR-ON use AVR [182] and BKP [21], respectively, for scheduling requests at the individual processors. AVR and BKP are described in Section 3.4.3.

In our experimental evaluation of PLASM (Section 4.6), we have used both a grouped RR approach similar to the idea behind CRR-ON and an on-line version of EDL (Lowest Load First) as baselines.

4.3 Allocation

In this section, we present the allocation problem for energy-aware multi-processor, multi-core task scheduling. This is the problem of deciding how many and which cores should be used to execute tasks, allowing the unused cores to idle.

Allocation strategies affect both power consumption and performance. On the power front, using a subset of cores while idling the others is one way to trade static and dynamic power consumption. When cores or entire processors are idled, they can go into sleep states and consume less power by reducing their static power draw. However, the workload must then be handled by remaining active cores. Those cores may have to run at higher speeds to meet the performance requirements of the workload, resulting higher dynamic power consumption.

On the performance front, using fewer processors may cause more missed deadlines. Since active processors and cores must handle higher loads when others are idled, they

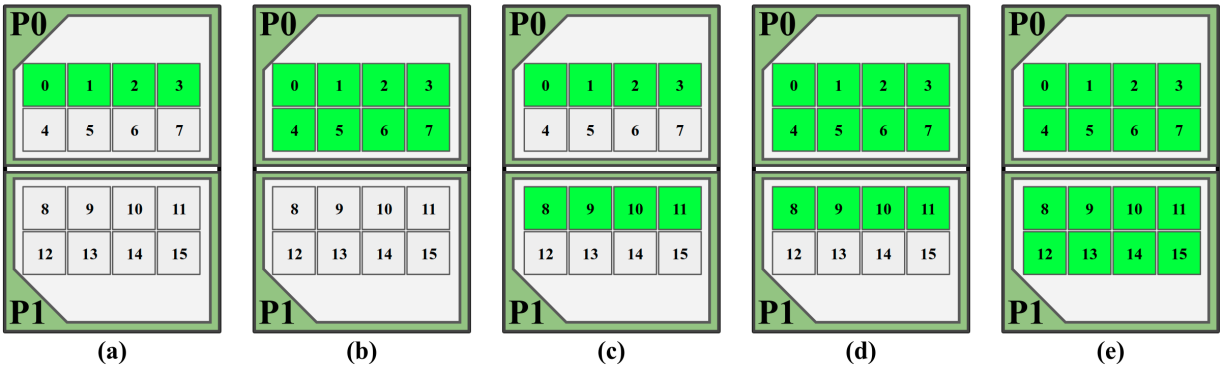


Figure 4.1: Five different allocation strategies in a multi-processor, multi-socket CPU. P0 and P1 are packages in sockets and each numbered square represent a separate core. The cores highlighted with green are the ones selected for allocation. The strategies from (a) to (e) are named as A4-0, A8-0, A4-4, A8-4, A8-8, respectively.

have less performance headroom. As a result, they are more sensitive to load fluctuations as they have less potential for handling extra work. More requests may miss their latency targets as a consequence.

4.3.1 Empirical Analysis of Processor Allocation Strategies

In this section, we evaluate the power and performance impact of allocation through some experiments. We aim to determine first whether idling cores or entire processors results in net power savings. That is, do the static power savings obtained by idling cores make up for the higher dynamic power consumption of the active cores? Second, we want to characterize the performance risk associated with idling some cores. Are transactions more likely to miss their latency targets?

In the experiments, we configure our test system to use different allocation strategies, as shown in Figure 4.1. We use five different allocation strategies for a two-socket server with 8 cores per CPU. These strategies allocate from 4 to 16 cores. Each allocation strategy represents a distinct point at the trade-off of static and dynamic power consumption and performance. We name the allocation strategies as A4-0, A8-0, A4-4, A8-4, and A8-8. The first number in each name indicates the number of allocated cores on package 1, and the second indicates the number of allocated cores on package 2.

In each experiment, we apply a fixed workload to the system under each allocation

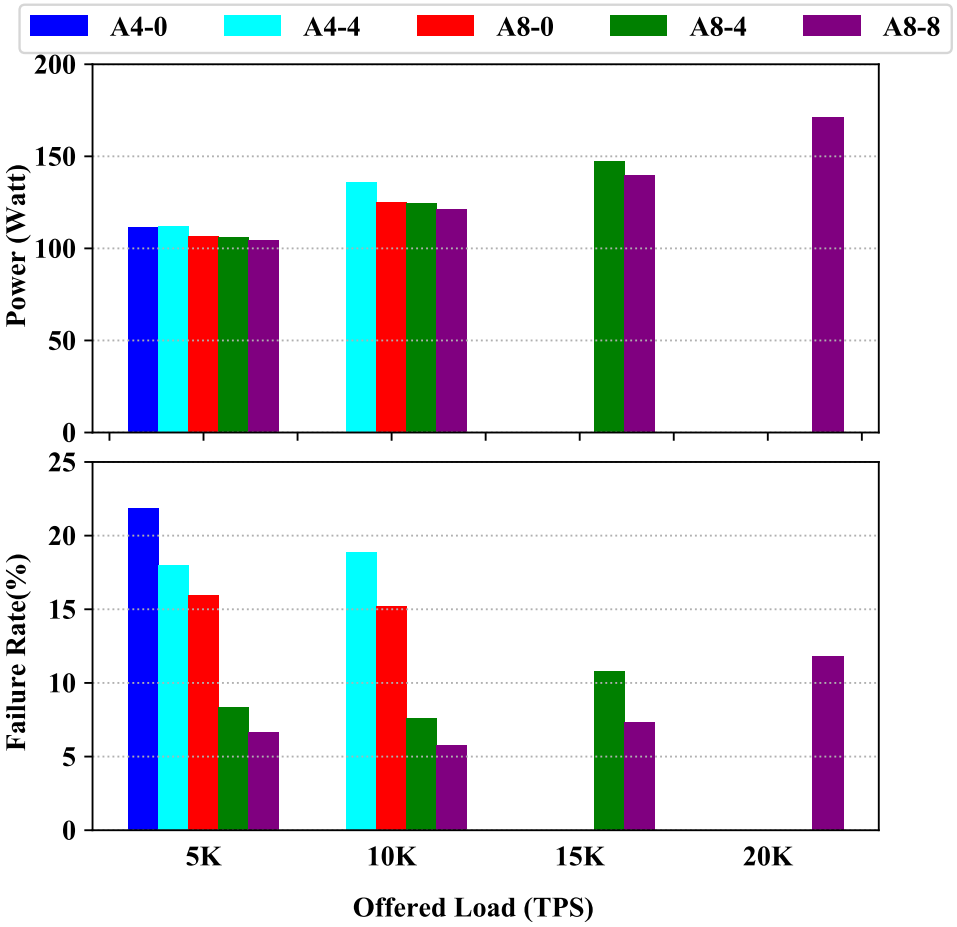


Figure 4.2: Power and failure rate of different allocations strategies

configuration and measure both power consumption and failure rate, i.e, the percentage of transactions that miss their deadlines. We repeat the experiment for several different fixed load levels.

These experiments used the same test system and TPC-C workload that is used in Section 3.6. The system uses round robin (RR) routing to distribute transactions to the allocated cores, and uses POLARIS to control the execution order and execution speed at each core. We experimented with TPC-C load levels ranging from 5000 transaction/second to 20000 transactions/second, in steps of with 5000 transactions/second. Transaction deadlines are set using a slack multiplier of 10.

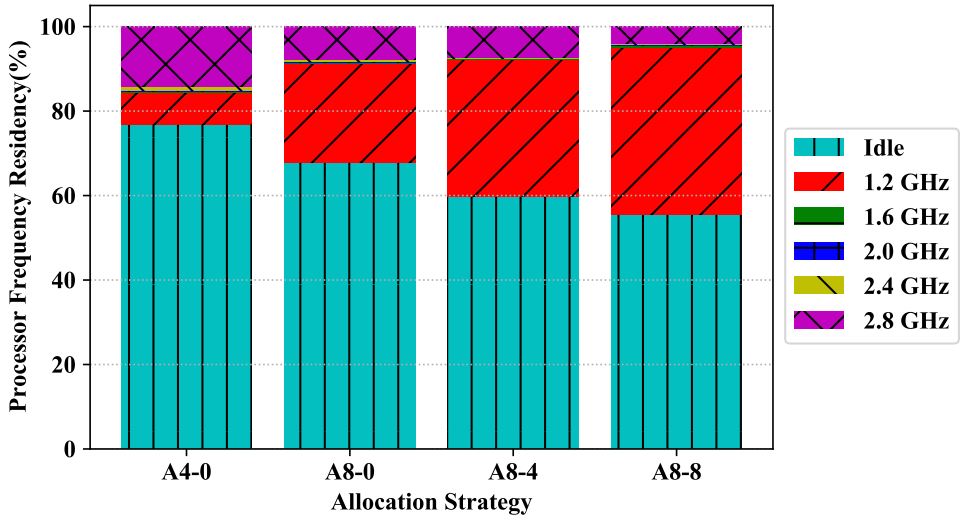


Figure 4.3: Dynamic and static power consumption residencies of different allocation strategies at 5000 TPS and slack multiplier 10. Failure rate and power consumption of each strategy is shown on top of the bars.

Figure 4.2 shows the power consumption and failure rate (percentage of transactions that miss their deadlines) for the five different allocation strategies as a function of offered load. For each load level, we consider only those allocations that are able to handle the load with a failure rate of less than 25%.

Our results show that the best allocation strategy is to use all of the available cores, at least on our test system. At all load levels, using more processors (A8-8) results in fewer failed requests. Thus, it is the safest option. In addition, A8-8 results in the lowest power consumption at every load level, although the differences in power consumption among the different allocations are small. Although Figure 4.2 only shows results for a slack level of 10, we saw similar results in experiments with looser slack, although differences in failure rates among the allocation strategies were smaller.

At lower loads, we expected to see that allocations A4-0 and A8-0, which direct all work to just one of the two processors in our test system, would save power by completely idling one processor. In fact, this is not the case. A8-0 does have slightly lower power consumption than A4-4 because the former idles a processor while the latter does not. However, both have higher power consumption (and more missed deadlines) than A8-8.

To better understand why allocating all the cores results in the lowest overall power consumption, we look more closely at processor frequency and idleness residency at the

experiments we present in this section. We calculate the residency using the data from within the database system by recording the times of frequency and idleness transition times throughout the experiments. Figure 4.3 shows CPU frequency and idleness residency of the different allocation strategies. As A4-4's residency is similar to A8-0, we do not depict it in the figure. The residency values are based on the sum of all sixteen cores' residency values, counting unallocated processors (for example, cores 4-15 in A4-0) as idle. The results show that the restricted allocation strategies such as A4-0 result in higher overall idleness, as expected, which means lower static power consumption. However, the results also show that such restricted allocations spend more time at higher frequency levels, which causes higher dynamic power consumption. These two effects, lower static power and higher dynamic power consumption counterbalance each other. As a result, different allocation strategies' power consumption levels are similar.

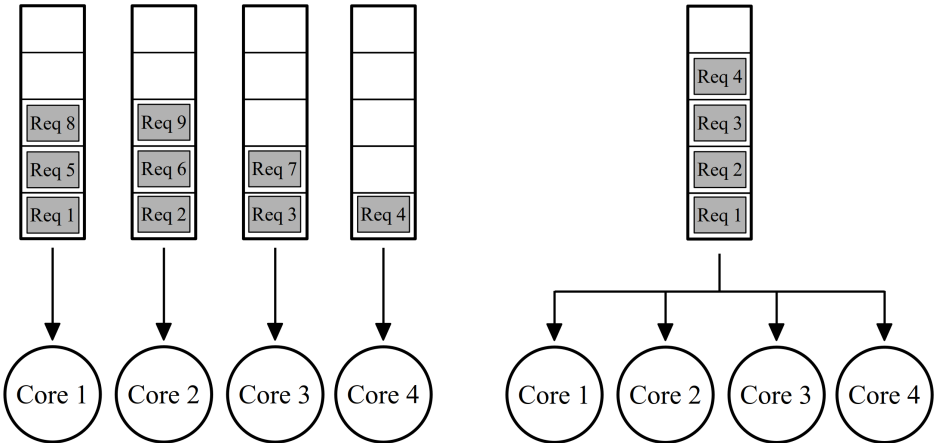
4.4 Routing

In this section we describe the routing problem, which is the problem of determining which core, on which processor, should be used to execute each transaction. Routing is closely related to the way that wait queues are formed [48]. Broadly, there are two classes of routing techniques in use: partitioned and global [64]. In partitioned routing, there is a separate work queue for each transaction executor (core). When transaction request arrives, they are assigned to one of the per-core work queues. In global routing, arriving requests are placed in a single centralized work queue which is shared by all cores. Each core pulls work from the centralized queue when it needs new work to do. Each approach has advantages, and there is no clear winner among them [18]. Empirical evaluations are often used distinguish their performance [26].

There are also hybrid scheduling solutions [35] that lie between the partitioned and global extremes. For example, a scheduler can have multiple queues, each shared by a different group of cores. However, in this section, we focus on the "pure" global and partitioned alternatives.

Partitioned routing is illustrated in Figure 4.4(a). In *static* partitioned routing, requests are never moved once they have been placed in a queue. In *dynamic* partitioned routing, requests can migrate across the different queues while they are waiting to be executed. Dynamic routing is more flexible, but it adds additional complexities to the routing problem, such as potential concurrency problems due to inter-queue transfers.

Partitioned routing offers several advantages. One of them is that the per-worker queue scheme mitigates the problem of contention on the waiting queues. Each queue has one



(a) Partitioned Routing (b) Global Routing

Figure 4.4: Routing Schemes

subscriber (the worker thread) and one publisher (central request handler). One other advantage is that unexpectedly long running transactions affect only requests in the local queue. Finally, since requests are routed without any delay and are accumulated in per-core queues, each core has information about its upcoming workload. This allows cores to take advantage of well-studied single-processor scheduling algorithms to manage their queues. Thus, having separate per-core queues enables a simple end-to-end solution to the multi-processor energy aware scheduling problem: combine a request router with an existing single-server energy-aware scheduling algorithm such as POLARIS.

Global routing is illustrated in Figure 4.4(b). In the global setting, the routing and request ordering problems are coupled because of the single queue. Requests are prioritized in the global queue according to the scheduling objective, and processors pull highest priority request request when they need work. Thus, there is no explicit routing policy.

One major advantage of global routing is that it provides automatic load-balancing across the processors, as any transaction can be executed by any worker. Thus, scheduling is work-conserving; that is, no processor stays idle as long as there are waiting requests in the queue. A disadvantage of global routing is contention for the work queue, which is shared by the request executors at all cores.

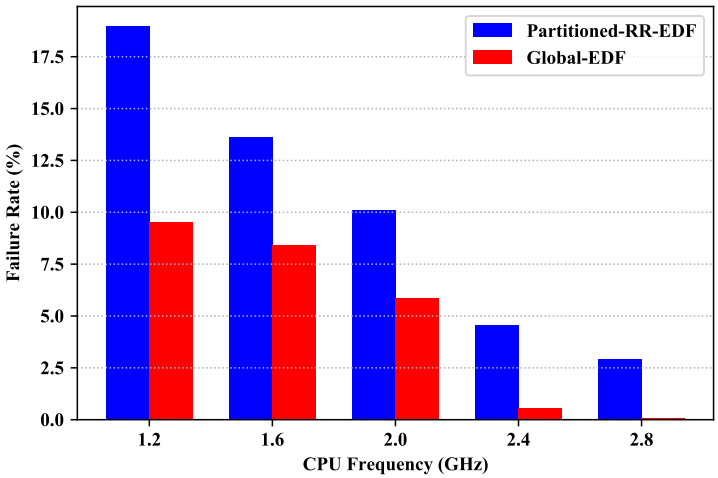


Figure 4.5: Partitioned routing with EDF order and global routing with EDF order, under different constant CPU speeds. Offered load is 9000 transactions per second.

4.4.1 Does Routing Matter?

In Chapter 3, we used simple partitioned round robin (RR) routing. Here, we would like to understand whether a different choice might have a significant effect on the overall performance of the scheduler. To shed some light on this question, we ran some preliminary experiments comparing two simple schedulers, one based on partitioned RR routing and one based on global routing. Both schedulers prioritize transactions in EDF order, and both schedulers execute transactions at the same fixed core frequency. Since all transactions run at the same frequency, we do not expect to see any difference in power consumption between these alternatives. However, we may see differences in performance, measured as the percentage of transactions that fail to hit their latency targets.

For these experiments, we use the Shore-MT system and server described in Chapter 3. We used the TPC-C workload at two distinct loads: a relatively low load of 9000 transactions/second, and a high load of 16000 transactions/second. Transaction deadlines were set using a slack multiplier of 10. All available cores were allocated, and we ran experiments using five different fixed core frequencies. In each experiment, we measured the percentage of transactions that failed (missed their deadline), as well as power consumption. However, we report only the failure rates, since power consumption at each combination of execution frequency and load level is approximately the same for the two alternatives.

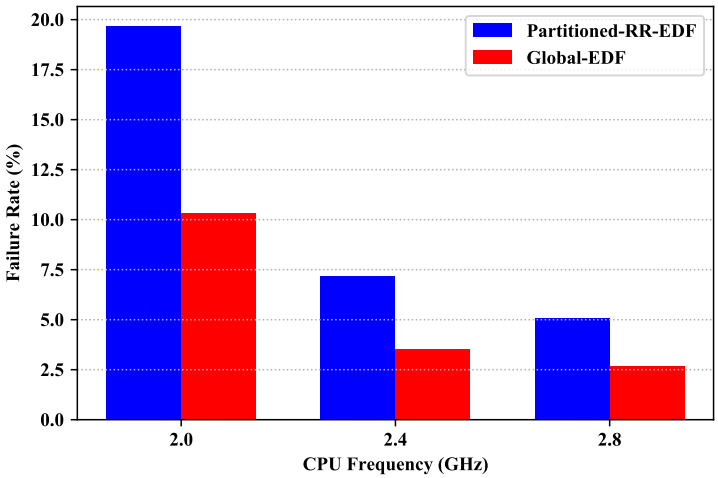


Figure 4.6: Partitioned routing with EDF order and global routing with EDF order, under different constant CPU speeds. Offered load is 16000 transactions per second.

Figures 4.5 and 4.6 show the failure rate of the two schedulers at each of the two load levels. At both load levels, and at all execution frequencies, the global scheduler results in much lower failure rates than partitioned RR. The absolute performance gap is higher at lower frequencies, since the cores have less capacity to overcome load imbalances causes by the RR routing.

These results suggest that it is worth investigating routing strategies other than partitioned RR. Our scheduling objective is to reduce energy consumption while avoiding missed latency targets. The results do not directly show whether reduced energy consumption is possible through better routing, but they do show that there is at least a substantial opportunity for improvement on the latency front.

4.5 PLASM

This section presents our multi-processor energy-efficient scheduling algorithm PLASM (Power and Latency Aware Scheduling in Multi-processor CPUs). PLASM works online and schedules requests non-preemptively. It addresses the allocation problem by using all of the available processors in the system, for the reasons described in Section 4.3.

PLASM uses partitioned routing. This allows it to take advantage of the POLARIS

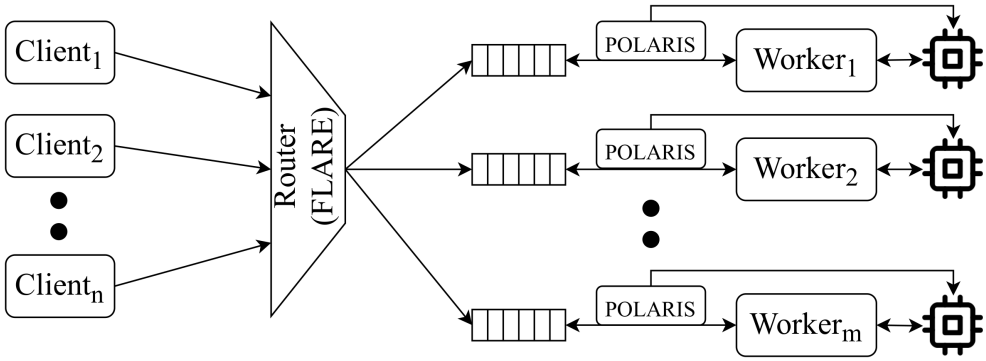


Figure 4.7: PLASM System architecture

algorithm from Chapter 3 to manage the single-server work queues at each core. In Section 4.4.1, we showed that partitioned RR routing results in higher-than-necessary transaction failure rates. For this reason, PLASM uses a custom workload- and energy-aware routing strategy FLARE, which we present here.

Figure 4.7 gives an overview of the PLASM design. PLASM schedules requests across multiple workers, where each worker is pinned to a separate processor core. Each worker has a separate request queue, and schedules its requests using POLARIS. Requests submitted by clients are sent to a centralized router. The router uses FLARE to choose a worker to handle the request, and immediately adds the request to that worker's queue. There is no queueing at the global (router) level.

4.5.1 An Ideal Router

The off-line version of our multi-processor scheduling problem is NP-hard, and existing on-line energy-aware scheduling techniques use heuristic routing strategies (like RR), as described in Section 4.2. PLASM's routing strategy, FLARE, is also heuristic. To motivate FLARE's design, we begin by describing an idealized but impractical greedy routing strategy, which we call IdealGreedy. FLARE is fast, lightweight router that is intended to approximate IdealGreedy's behavior.

Figure 4.8 presents the IdealGreedy routing algorithm. The *simulatePOLARIS* function simulates the execution of the transactions in a worker's request queue, and reports the total amount of energy that will be consumed, and the number of transactions that will miss their deadlines. *simulatePOLARIS*'s simulation assumes that the transactions are executed as determined by POLARIS.

State: H: set of workers
State: p_{dest}: destination worker

function IDEALGREEDY(Request r)
    $p_{dest} \leftarrow 0$
    $missed_{low} \leftarrow \text{inf}$
    $energy_{low} \leftarrow \text{inf}$
    for each p in H **do**

        ▷ Get the current state of the waiting requests
        $Q_{before} \leftarrow Q(p)$
        ▷ Simulate POLARIS on the current state of the queue
        $(missed_{before}, energy_{before}) \leftarrow$ **simulatePOLARIS**(Q_{before})

        ▷ Insert the new request to current queue
        $Q_{after} \leftarrow$ **insertEDFOrder**(Q_{before}, r)
        ▷ Simulate POLARIS with the new request
        $(missed_{after}, energy_{after}) \leftarrow$ **simulatePOLARIS**(Q_{after})

        ▷ Calculate extra missed deadlines and energy with the new request
        $extraMissed \leftarrow missed_{after} - missed_{before}$
        $extraEnergy \leftarrow energy_{after} - energy_{before}$

        if $extraMissed < missed_{low}$ **then**
            $missed_{low} \leftarrow extraMissed$
            $energy_{low} \leftarrow extraEnergy$
            $p_{dest} \leftarrow p$
        else if $extraMissed = missed_{low} \wedge extraEnergy < energy_{low}$ **then**
            $energy_{low} \leftarrow extraMissed$
            $p_{dest} \leftarrow p$
        end if

    end for
    return p_{dest}
end function

Figure 4.8: IdealGreedy Processor Router Algorithm

To route a transaction, IdealGreedy performs a series of "what if" analyses. For each worker, it determines how many new transaction failures and how much additional power consumption would result if it were to route the transaction to that worker. It does so by comparing the results of $simulatePOLARIS$ with and without the new transaction in the worker's queue (lines 6-16 in Figure 4.8). With the results of the what-if analyses in hand, IdealGreedy routes the new transaction to the worker that will result in the fewest additional missed deadlines, since its primary goal is to meet latency targets. When there are multiple such workers, IdealGreedy chooses the one that will result in the lowest additional power consumption (lines 17-24 in Figure 4.8).

Using its what-if analyses, IdealGreedy greedily makes a locally optimal decision to route each new transaction. However, there are at least two problems with this approach in practice. First, the simulations themselves are imperfect, as they are must be based on estimates of transaction execution times rather than actual execution times, as discussed in Chapter 5. Second, IdealGreedy is very expensive, since it runs two simulations per worker on every transaction arrival. These simulations would introduce latency and would also consume power, working against the goals PLASM is trying to achieve. FLARE, which we present in the next section, is intended to approximate IdealGreedy's decisions, but at much lower cost.

4.5.2 FLARE

FLARE (Frequency and Load Aware Routing) is a light-weight energy-efficient routing algorithm. Like IdealGreedy, it greedily routes transaction so as to minimize missed deadlines and keep energy consumption low. However, instead of running POLARIS simulations to choose a routing target, FLARE bases its routing decisions on summary information maintained by each worker.

FLARE expects each worker to maintain two pieces of information. Each worker updates its information each time it runs the POLARIS scheduling algorithm on its local queue. The first piece of information is the current execution frequency of the worker's core, which is adjusted by POLARIS each time a new transaction request arrives in the worker's queue or is completed by the worker. The second piece of information is *load level* of the worker's queue. Load level characterizes the total amount of work that has been assigned to the worker. Load level is quantified as the total estimated processing time required to complete all of the requests in the worker's queue, including the currently running request. Since processing time depends on the worker's core's execution frequency, FLARE arbitrarily uses estimated processing time at peak frequency as its canonical load

Notation	Meaning
H	set of workers
$Q(p)$	wait queue of processor p
$s(p)$	processor frequency of worker p, $p \in P$
$l(p)$	load level of worker p, $p \in P$
$\mathcal{F}$	set of possible processor frequencies
$\hat{\mu}(w, f)$	estimated execution time of workload w transaction at frequency f

Figure 4.9: Summary of Notation in FLARE Algorithm

metric. Each worker piggybacks maintenance of its current load level on its POLARIS executions.

Figure 4.9 summarizes the notation we use to describe FLARE, and Figure 4.10 shows the FLARE algorithm. As shown in Figure 4.10, FLARE chooses the worker(s) with the lowest execution frequency (line 6). If there is a tie, FLARE breaks it in favour of the worker(s) with lowest load level (line 10). If there are multiple such workers with the same speed and load level, FLARE will pick the worker running on the lowest-numbered core. FLARE's complexity is $O(M)$ where M is the number of workers/CPU cores.

Why Execution Frequency?

The primary objective of the IdealGreedy algorithm is route transactions such that the number of additional missed transaction deadlines is minimized. FLARE uses execution frequency as its primary routing criterion to try to achieve the same goal. When a new transaction is assigned to a worker's queue, the POLARIS algorithm will increase the frequency of the worker's core as much as necessary to avoid missing transaction deadlines. FLARE routes to a worker with the lowest current execution frequency because those workers have the most *frequency headroom* that POLARIS can use to accommodate a new request without creating new deadline misses.

When the POLARIS algorithm runs on a worker's queue, it plans the execution of all requests that are currently in the queue, choosing execution frequencies for all of them (under the assumption that new requests will arrive). A property of POLARIS's plans is that the execution frequencies for later transactions in the queue are never greater than those of earlier transactions, as explained in Section 3.4.4. Hence, the current frequency represents a lower bound on the worker's *frequency headroom* throughout POLARIS's execution plan. This is illustrated in Figure 4.11 which shows the requests in a wait queue and how POLARIS would order them choose execution frequencies. In the figure, each rectangle represents a request. Rectangle height indicates the amount of work required to

State: H: set of workers
State: f_{low}: lowest frequency speed of workers (so far)
State: l_{low}: lowest load of a worker p where $s(p) = f_{low}$ (so far)
State: p_{dest}: destination worker

```
1: function FLARE( )
2:     p_dest ← 0
3:     f_low ← inf
4:     l_low ← inf
5:     for each p in H, in order of increasing ID do
6:         if s(p) < f_low then
7:             f_low ← s(p)
8:             l_low ← l(p)
9:             p_dest getsp
10:        else if s(p) = f_low ∧ l(p) < l_low then
11:            l_low ← l(p)
12:            p_dest ← p
13:        end if
14:    end for
15:    return p_dest
16: end function
```

Figure 4.10: FLARE Processor Router Algorithm

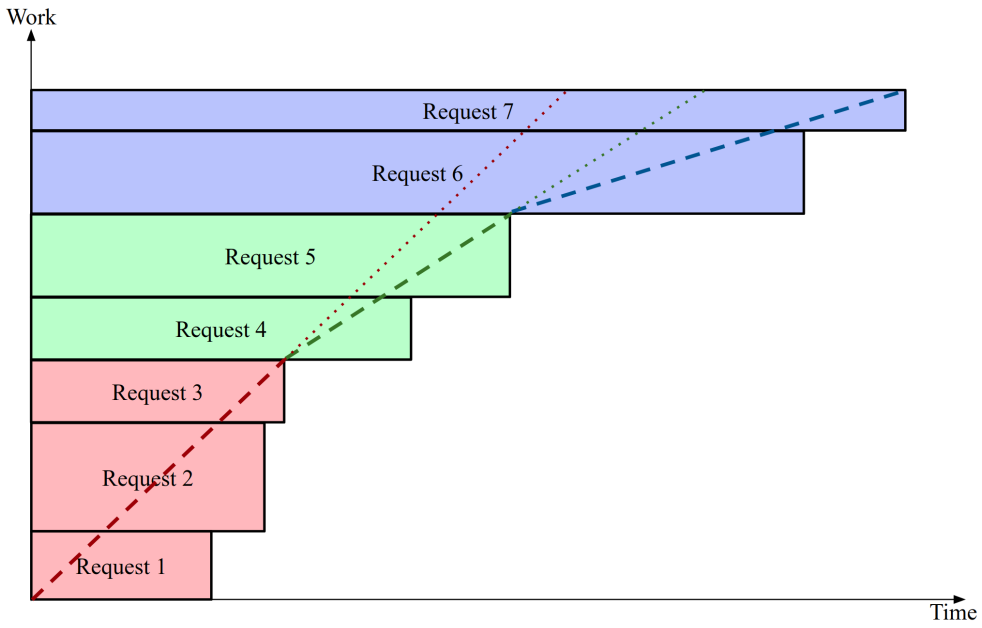


Figure 4.11: Load intervals in a POLARIS controlled processor. The first interval (red requests) is the critical interval. Frequency in the subsequent intervals (red, green and blue, respectively) are monotonically decreasing. The load of this processor is the total work (Requests 1 to 7) waiting to be executed.

complete the request, and the width indicates the request deadline. Dashed lines show the execution rate, i.e., the execution frequency. Different colours represent different intervals, with each interval consisting of requests to be executed with the same frequency.

Why Low Load?

The secondary goal of the IdealGreedy algorithm is to minimize energy consumption. If there are multiple workers with the same execution frequency, FLARE uses load level as a proxy for the likely impact of a new request on a worker's energy consumption. The energy consumed by a worker depends on the execution frequencies chosen by POLARIS. Those, in turn, depend on both the load level in the worker's queue, and on the deadlines of the queued transaction requests. Thus, it is difficult to accurately predict the impact of a new transaction on a worker's energy consumption without simulating the execution of the queued transactions, as IdealGreedy does. In practice, however, since POLARIS adjusts execution frequency in discrete steps, it can typically accommodate some amount of extra load before it is forced to increase frequency to avoid missing deadlines. Thus, absent

detailed information about the deadlines of the individual requests in the worker's queue, FLARE uses the worker's load levels as a proxy for the extra room they have available to accommodate an extra transaction.

To give an intuition for this choice, Figure 4.12 shows two processor queues with different load levels, and shows how POLARIS plans power consumption with the arrival of a new request in each case. In both queues, the first three requests are the same and are scheduled to run at the same frequency. In the lower load case (a), there are four requests and two frequency intervals, shown in red and blue. When the new request arrives, it is inserted into the second to last position according to the EDF order. This insertion causes the last transaction to run at a slightly higher frequency (dotted blue line versus the dashed purple line). The first frequency interval does not change.

The new request's has a much greater impact in the higher load case (b). In that case, there are five requests with three frequency intervals (red, green and blue). When the new request arrives, it is inserted into the second last position according to the EDF order, and the last request's frequency is slightly increased, much as in the low load case (a). However, in the high load case, the arrival of the new request also forces an increase in the execution frequency of the first four requests in the queue, which will result in a greater increase in overall power consumption.

Why the Lowest-Numbered Core?

If there are multiple processors with the same speed and load, then FLARE breaks the tie ultimately in favour of the worker with the lowest core ID. This final tie-breaker is mostly arbitrary. However, since cores are numbered one processor at a time (in Linux), this rule will have the effect of routing the available work to the processor(s) with the lowest-numbered cores, leaving the remaining processors idle. This may allow processors with no active cores to use lower package C-states, potentially resulting in some power savings.

Figure 4.13 shows how FLARE works when load is light and there are idle cores. In Figure 4.13 (a), CPUs 0 and 1 each have idle (grey) cores. When a request arrives, all idle cores have the same lowest speed (zero), and all have zero load. Therefore FLARE breaks the tie using core IDs and hence routes the request to CPU 0 (Figure 4.13 (b)). As a consequence, CPU 1 remains fully idle, and may take advantage of package C-states.

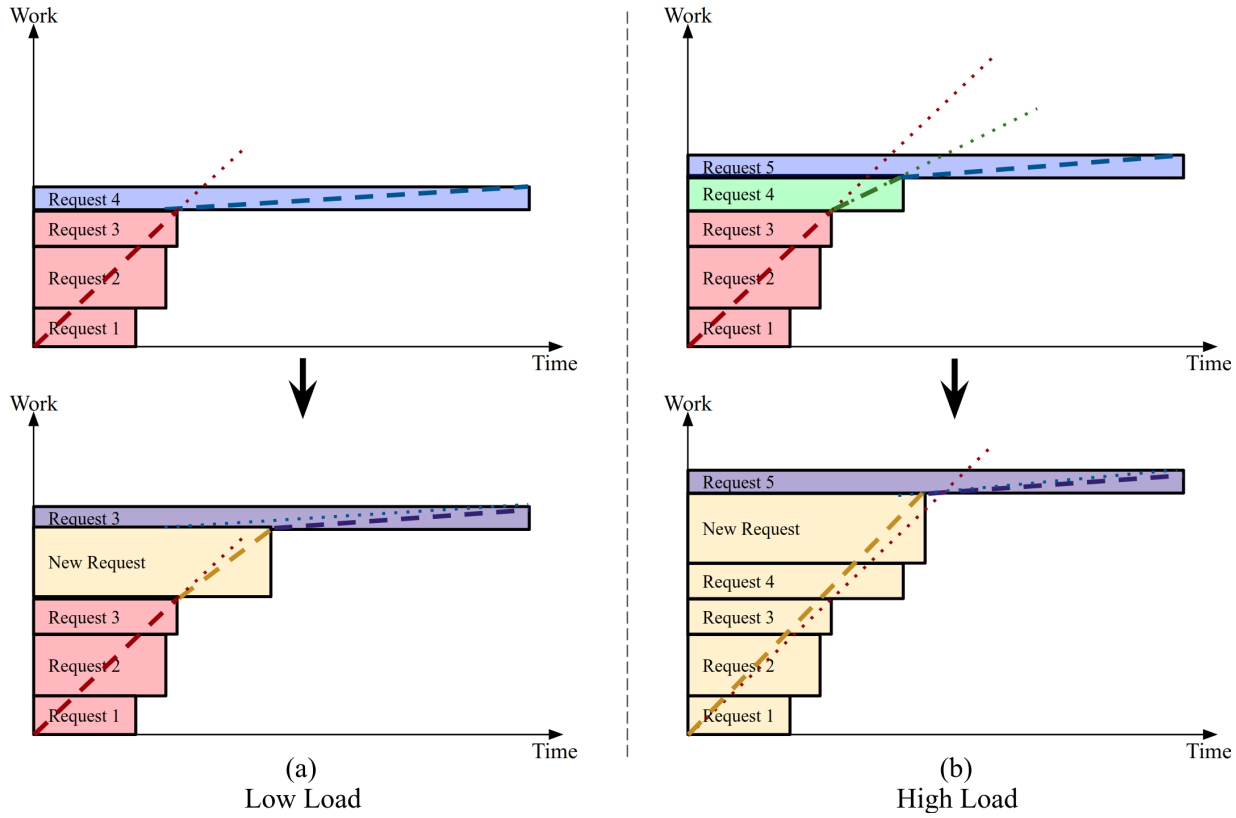


Figure 4.12: How POLARIS plans CPU speed with a new request. The first and second row show before and after the state after the new request of two different queues. The first column (a) and the second column (b) show a low and high load, respectively. Different colors show critical sections and dashed lines show speed required to run request in during different sections.

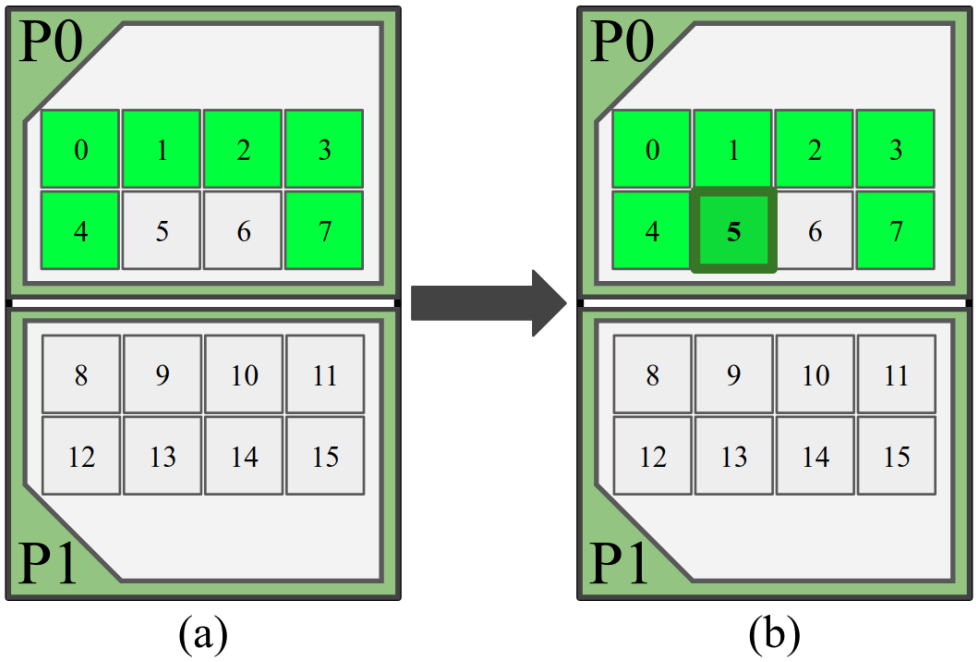


Figure 4.13: States of a 2 socket CPU with 8 cores each before (a) and after (b) a new request is arrived. Green cores are busy and gray cores are idle.

4.6 Evaluation

We implemented PLASM in Shore-MT, adding FLARE routing to the existing POLARIS implementation that is described in Section 3.5. Using this implementation, we aim to compare PLASM against several multi-processor scheduling baselines, which are explained in Section 4.6.2. We want to determine whether PLASM results in better power consumption and performance than can be achieved using the baselines.

4.6.1 Methodology

For our PLASM evaluation, we use a methodology very similar to the one explained in Section 3.6.1. We use Shore-MT TPC-C benchmark implementation and a database scale factor of 48. Shore-MT's buffer pool is configured to accommodate the entire database in memory.

In each experimental run, we choose PLASM or one of the scheduling baselines and run the benchmark workload against our Shore-MT prototype. There are three phases at each run. First, there is a warmup phase during which each worker executes 30,000 requests. Second, there is a short training phase for collecting data for the execution time estimation model tht is used by POLARIS and FLARE. Finally, there is a test phase, during which we measure power consumption and transaction performance.

We changed Shore-MT's request generation from a closed-loop design to an open-loop design so that we can target a specific offered load for each experimental run. The request interarrival times are randomly chosen from a uniform distribution where the minimum is zero and the maximum is twice the mean interarrival time that results in the target load. We used three target load levels in our experiments, referred to as low, medium and high. These correspond to request rates of 15000, 19000 and 23000 transactions/second, respectively. For reference, the maximum capacity of our test server is about 11000 transactions/second (TPS) if the processors run at their lowest speed all the time, and is about 24000 TPS if they run at peak frequency all of the time. These limits are illustrated in Figure B.2 in Appendix B.

We use the notion of slack to provide a uniform way to control the deadline tightness of requests in our TPC-C workload. The slack multiplier represents the ratio between a request's relative deadline and its mean execution time under the peak speed level. For example, for a TPC-C New Order transaction, which has an average execution time of 2059 μs (recall Figure 3.3) at the highest frequency level, a slack of 20 indicates a latency target of 41180μs (20×2059). In our experiments, we use slack values ranging from 10 to 100 to explore the effect of different latency tightness on different scheduling baselines' power consumption and performance.

For each run, we measure the average power consumed by the server during the test phase. To measure server power draw, we used a Watts up? PRO [82] wall socket power meter, which has a rated $\pm 1.5\%$ accuracy. We measure the power consumption in one-second intervals (the finest granularity of the power meter) and average those over the test duration. In addition to the power metric, we also measure performance during the test phase. In each of our experiments, the mean system throughput is fixed and controlled by our open-loop request generator. Thus, we are primarily interested in transaction latency. Specifically, we measure the failure rate, which is defined as the percentage of transactions that do not finish execution before their deadline.

We also measure CPU frequency and idleness residencies, i.e., the percentage of time that each core spends in each P-state, and in idle states (C-states other than C0). There are several ways to measure these residencies. MSRs can be used, but these only expose

the instantaneous frequency level. Thus, they must be sampled repeatedly to determine frequency residencies. Instead, since our Shore-MT prototype directly controls the frequencies of all processor cores, we collected the residency data from within Shore-MT by logging the times at which it triggers frequency changes and idlings at each core.

In our experiments, we use a server with two Intel® Xeon® E5-2640 v3 processors with 128 GB memory, running Ubuntu 14.04 with kernel version 4.17. For the experiments we disable the CPU ACPI software control in the BIOS configuration to prevent the `cpufreq` driver from interfering with power control. To reduce non-uniform memory access (NUMA) effects and get more homogeneous memory access patterns, we enable memory interleaving in the BIOS. Also, Shore-MT's log flusher is configured to use a flushing interval long enough to ensure that flushing does not occur during the test phase of an experimental run.

Each E5-2640 CPU has eight physical and sixteen logical cores (hyper-threads); thus, our system has a total of 16 physical (32 logical) cores. Each physical core's power level can be set separately. The CPU has 15 frequency levels from 1.2 GHz to 2.6 GHz with 0.1 GHz steps, plus 2.8 GHz. In our experiments, we chose five of the frequency levels, 1.2, 1.6, 2.0, 2.4 and 2.8 GHz, as the possible target frequency levels for POLARIS.

For all of our experiments, our Shore-MT prototype is configured to use two Request Handler (RH) threads and sixteen worker threads. We pin one worker thread to a logical core (hyperthread) in each of the 16 physical cores. In most of our experiments, except those using a global routing baseline, each worker thread has a transaction request queue which it manages using POLARIS or a baseline scheduler.

The RH threads are responsible for receiving incoming requests from clients and routing those requests to workers. Routing is done using FLARE or a baseline routing policy, depending on the experiment. RH threads are free to run on any of the remaining logical cores in the first socket, as determined by the kernel's thread scheduler.

To simplify our experiments, we do not drive the Shore-MT server using remote clients. Instead, each RH thread simulates a set of remote clients by generating randomized requests and then handling them as if they had arrived over the network from remote clients.

4.6.2 Baselines

In our experiments, we compared PLASM against the following baseline multi-processor scheduling algorithms:

- **POLARIS/RR:** POLARIS/RR is the partitioned scheduling technique that we used in Chapter 3. It uses our energy-aware single processor scheduling algorithm, POLARIS, to manage the work queue at each worker, and round robin routing to distribute requests to the workers. We showed in Chapter 3 that POLARIS/RR results better performance and energy efficiency than OS-based frequency governors. Here, we want to determine whether we can further improve on POLARIS/RR by using energy aware routing (FLARE).

- **PerformanceBaseline:** PerformanceBaseline represents a multi-processor scheduler whose sole objective is to minimize failure rate, regardless of the power consumption. In PerformanceBaseline, all the processor cores are set to peak frequency. Global routing is used, with a single centralized work queue. Requests in the global queue are prioritized in EDF order. We are primarily interested in the failure rate achieved by PerformanceBaseline, since its power consumption will be quite high. Since PLASM's primary goal is to avoid missed deadlines, we hope that PLASM's failure rate will be similar to that of the PerformanceBaseline - though with lower power consumption. We discuss the PerformanceBaseline in more detail in Appendix B.1.

- **EnergyBaseline:** EnergyBaseline represents a multi-processor scheduler whose sole objective is to minimize power consumption, regardless of the failure rate, while accommodating the offered load. Like the PerformanceBaseline, the EnergyBaseline uses global routing, with centralized EDF-prioritized work queue. An ideal EnergyBaseline would fix the frequencies of all cores at the lowest frequency that can accommodate the offered transaction request rate. In practice, however, we are limited in the frequencies that we can choose since our test server's processors support a limited set of P-States. Thus, the power consumption we report for the EnergyBaseline is determined by interpolating between measured power consumptions obtained using P-States with frequencies near the workload-specific minimum frequency. The details of this interpolation are described in Appendix B.2.

  Since the EnergyBaseline is unconcerned with transaction latencies, transaction failure rates are very high, near 100%. That is, most transactions miss their deadlines. For this reason, we report only power consumption for this baseline, not failure rates.

- **GRR:** GRR (Grouped Round Robin) works based on the idea behind the grouped RR routers CRR [11] and DCRR [27] that we describe in Section 4.2. CRR classifies requests according to their work density, and requests of each class are routed according to a separate round-robin order. The benefit of CRR is that it avoids the adversarial cases in which high-density requests are queued up together on some processors while other processors have lower-density requests. In our experiments, we

apply the same slack multiplier to all workload types; therefore, densities are equal. This means CRR would have a single group and effectively make the same routing decisions as RR. Instead, our GRR baseline classifies the requests according to their workload type, and routes each type separately using RR. This is similar to DCRR, which groups requests according to their size when densities are equal. Like PLASM and the POLARIS/RR baseline, the GRR baseline uses POLARIS to managed the work queues at each core.

4.6.3 Results: Medium Load

Figure 4.14 shows the failure rate and power consumption of PLASM and the scheduling baselines under the medium load(19000 TPS), as a function of slack.

These results show that PLASM results in fewer missed deadlines than POLARIS/RR across all the slack levels while consuming less power. Unlike RR, PLASM's FLARE router steers new requests away from workers that have less frequency headroom, making deadline misses less likely. When slack is loose, this makes little difference, since all schedulers have very low failure rates. However, at tighter slacks, FLARE cut failure rates almost in half, compared to RR. Furthermore, except at the tightest slack level (10), PLASM's failure rate is almost identical to that of PerformanceBaseline, which uses centralized routing and runs at peak speed all of the time. This suggests that there is little room for any scheduling and routing algorithm to improve on PLASM, except perhaps when the slack is very tight.

GRR resulted in slightly lower failure rates than POLARIS/RR's, but not as low as FLARE's. GRR is designed to prevent adversarial cases that can result in deadline misses when plain RR routing is used. For example, in an arrival order of requests where RR ends up routing all the large requests to one of the processors and smalls to the others, GRR can distribute the load more evenly. However, such extreme adversarial cases are not likely in our workload.

On the power consumption front, PLASM reduces server power consumption by 10W-15W relative to both POLARIS/RR and GRR. In Figure 4.14, the power gap between the PerformanceBaseline (>180W) and the EnergyBaseline (∼150 W) represents the power saving opportunity for this workload. POLARIS/RR and GRR capture about one half to two-thirds of this opportunity, depending on the slack. PLASM captures almost all of it. Its power consumption is as low as EnergyBaseline's at most of the slack levels and is only slightly higher in others. Since EnergyBaseline represents the minimum energy required to handle this request rate, this suggests that no other routing and scheduling algorithm can significantly improve on PLASM's power consumption, at least for this workload.

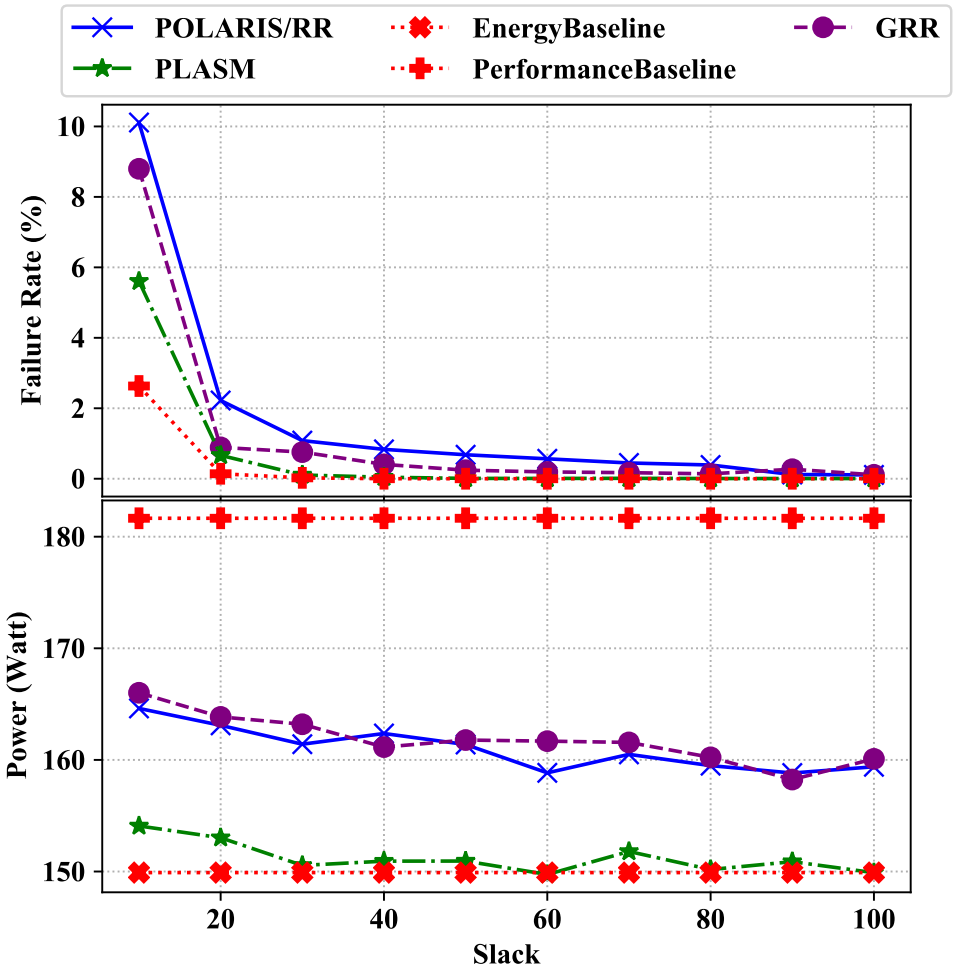


Figure 4.14: Failure Rate and Power of different multi-processor schedulers under medium load, as functions of slack (S)

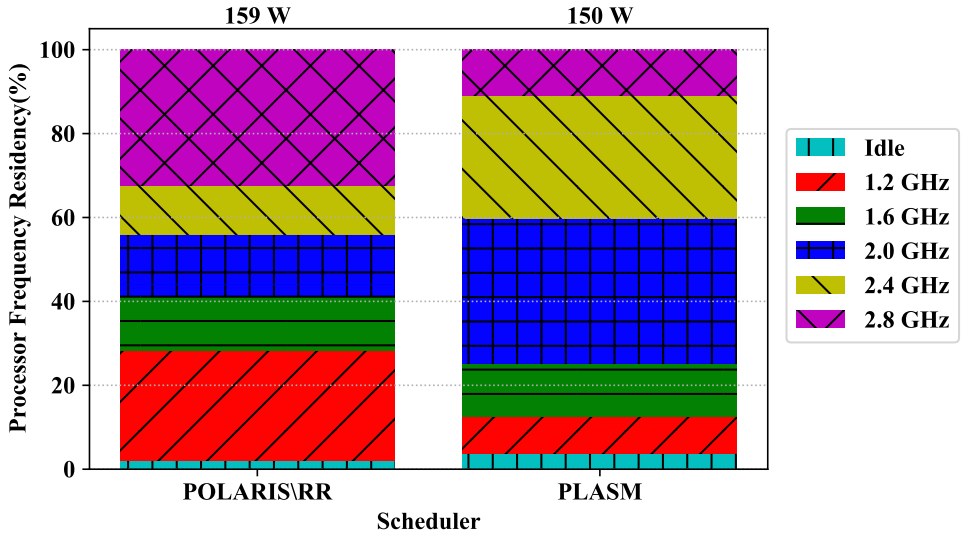


Figure 4.15: PLASM and POLARIS/RR Processor Frequency Residency, under medium load and the slack multiplier is 60.

To understand why PLASM saves power relative to POLARIS/RR, we considered the frequency residency distributions of the two schedulers. Figure 4.15 shows these distributions for the medium load with a slack multiplier 60.

Even though both schedulers use POLARIS at each core to regulate frequency, different routing techniques result in quite visibly different frequency residencies. Under PLASM, workers spends most of their time at two consecutive intermediate frequency levels, 2.0 and 2.4 GHz. In contrast, workers under POLARIS/RR spend most of their time either at the lowest speed (1.2 GHz) or highest speed (2.8 GHz) and relatively less time at intermediate frequencies. Since RR does not understand workers' speeds and load states, it may route requests to relatively busy workers, forcing them to to increase frequency even though other workers are relatively idle. In contrast, FLARE deliberately routes requests to worker that it believes are least likely to require a frequency increase, which tends to keep worker frequencies away from the extremes.

POLARIS/RR's frequency residency distribution results in higher power consumption because of the convex relationship between frequency and power. It is better to run a steady "just right" frequency than to swing between too fast and too slow. Figure 4.16 illustrates this choice, showing different ways to complete a request (represented by the rectangle) within its deadline. At one end, S_1 uses a single steady frequency level throughout the

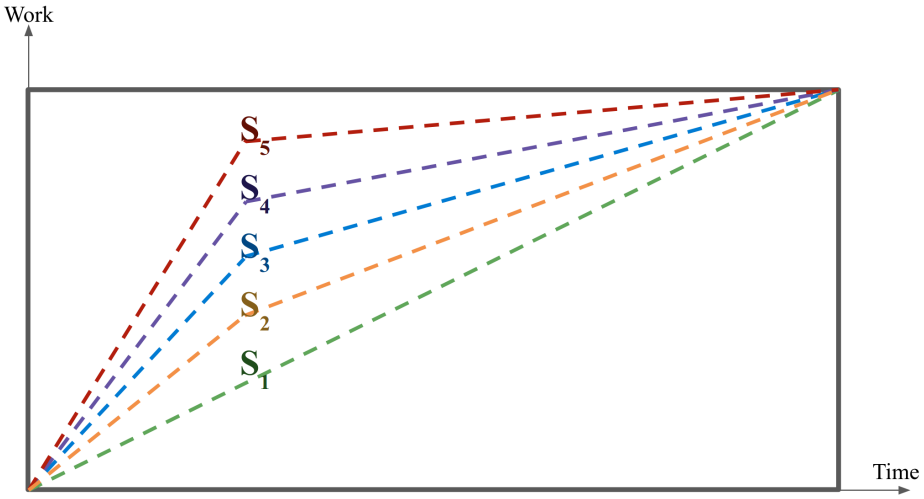


Figure 4.16: Different frequency combinations to execute a unit work. The rectangle represents a request, similar to the representation in Figure 4.11. The height is amount of the requests work and the length is the time between the request's arrival and deadline. The dashed lines represent execution rate (work/time).

execution, whereas the other options have two parts: first a higher frequency, then a lower one. Because of the convex relation, S_1 consumes the least power, whereas S_5 consumes the most as the gap between high and low parts is the greatest[94].

4.6.4 Results: Effect of Load

To investigate the effects of system load on PLASM, we repeat our medium load experiment under low and high load conditions. Here, we offer 23000 TPS and 15000 TPS loads to generate high and low load, respectively.

High Load

Figure 4.17 shows the power consumption and failure rate of PLASM and the baselines under high load, as a function of slack. In high load, the power gap between EnergyBaseline and PerformanceBaseline is narrow, indicating that there is little power saving opportunity. PLASM is able to capture the available opportunity when there is sufficient slack, but can

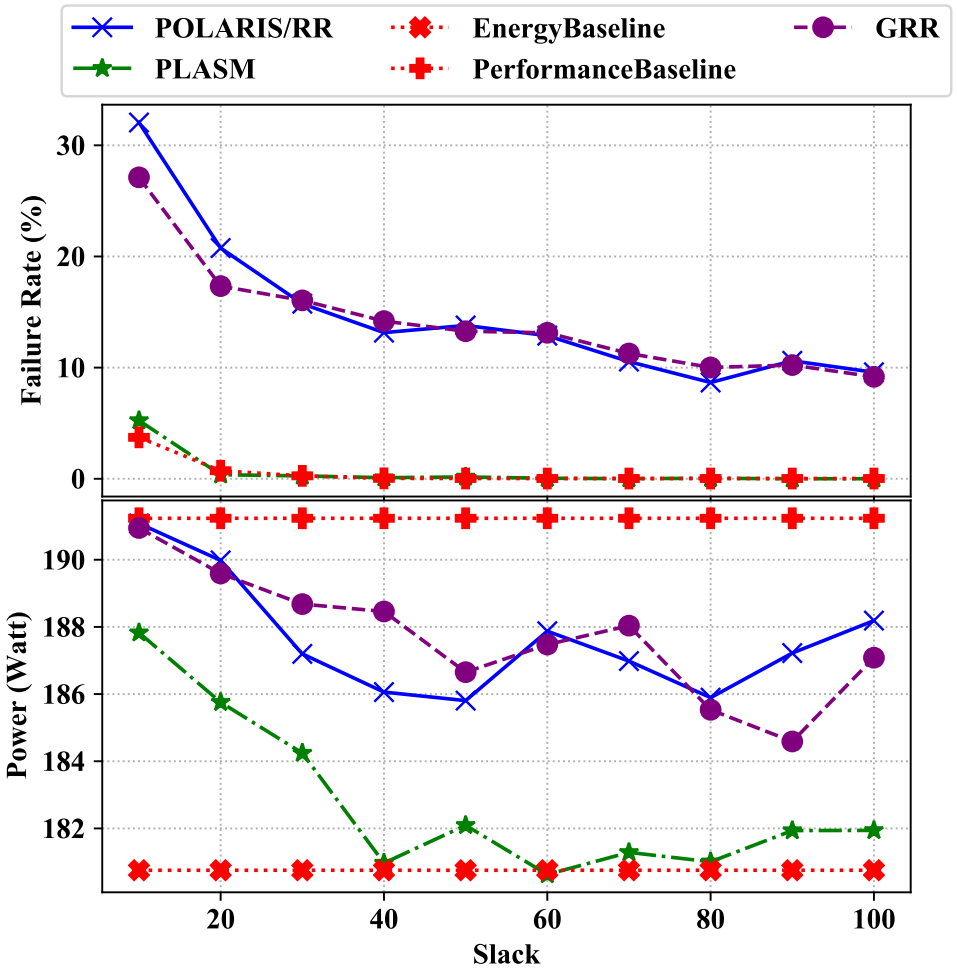


Figure 4.17: Failure Rate and Power of different multi-processor schedulers under high load, as functions of slack (S)

only partially do so at tighter slacks. POLARIS and GRR are able to capture about half of the opportunity when slack is high.

Although high load does not offer substantial power saving opportunity, our results show that PLASM results in much lower rates of missed transaction deadlines than both POLARIS/RR and GRR, at all slack levels. This is because FLARE steers incoming requests to workers that have the most frequency or load headroom to accommodate them without missing deadlines. PLASM's failure rate is almost identical to that of the centralized high-frequency PerformanceBaseline.

Low Load

Finally, Figure 4.18 shows the failure rate and power consumption of scheduling baselines under the low load. At this load, the power saving opportunity is more than 40W. PLASM results in slightly lower power than POLARIS/RR and GRR, but all three algorithms are able to capture most of the available opportunity. All three algorithms are also able to achieve near-zero failure rates when slack is loose. When slack is tight, PLASM's headroom-aware routing results in about half of the failure rate achieved by POLARIS/RR and GRR, though not as low as the PerformanceBaseline. There is almost no difference between POLARIS/RR and GRR in the low-load setting, since request queues are very short.

4.6.5 Results: FLARE Component Analysis

FLARE's routing decisions depends on two factors: the current speed and the current load of each worker. In our final experiments, our goal is to test whether POLARIS needs to take both of these factors into account. To test this, we compared FLARE against two variants; one that considers only the frequency and one that considers only the load.

The first variant is called SSF (Slowest Speed First). The SSF router picks the processor with the slowest speed, and if there is a tie, it breaks in favour of a random processor. The second variant is called LLF (Least Load First). The LLF router picks the processor with the lowest load, and if there is a tie, it breaks in favour of a random processor. As it is noted in Section 4.2, the energy-efficient multi-processor scheduling algorithm EDL [11] uses the least-load-first criteria and works similar to LLF, except it is designed for an offline setting.

Figure 4.19 shows the power consumption and performance of PLASM and the two variants for TPC-C under a medium load (19000 TPS) as a function of slack.

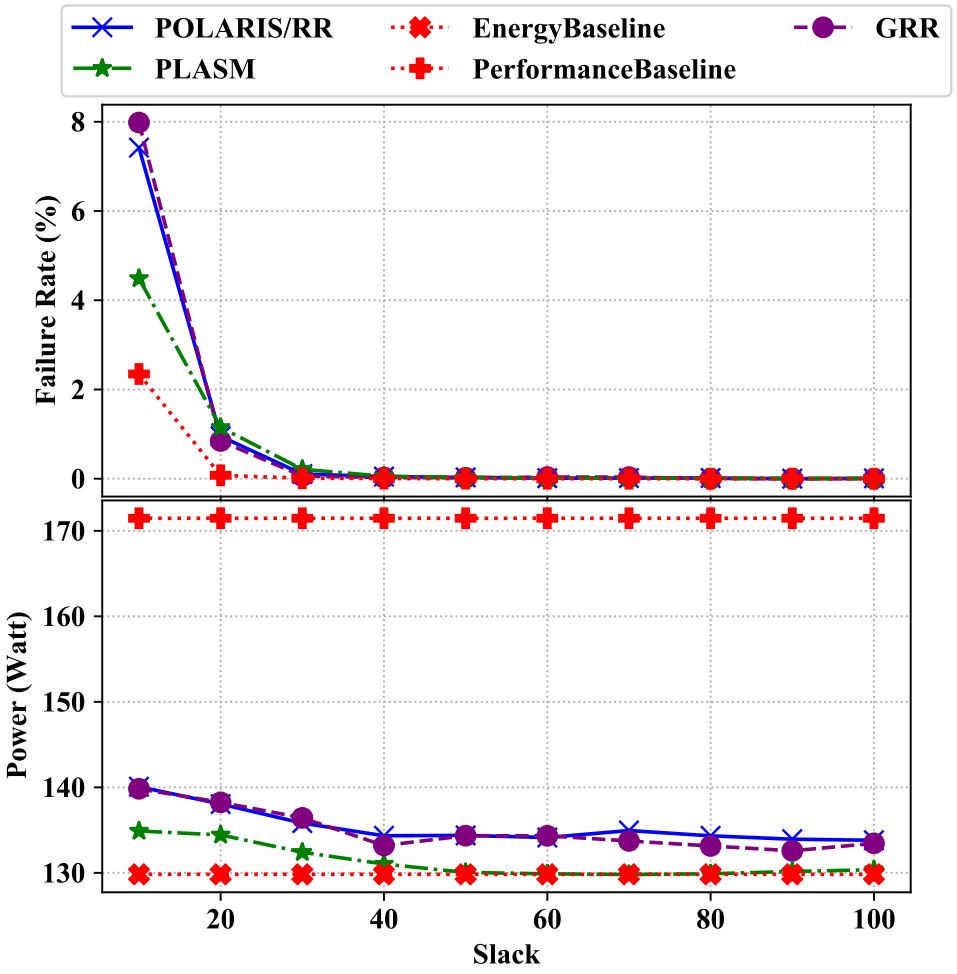


Figure 4.18: Failure Rate and Power of different multi-processor schedulers under low load, as functions of slack (S)

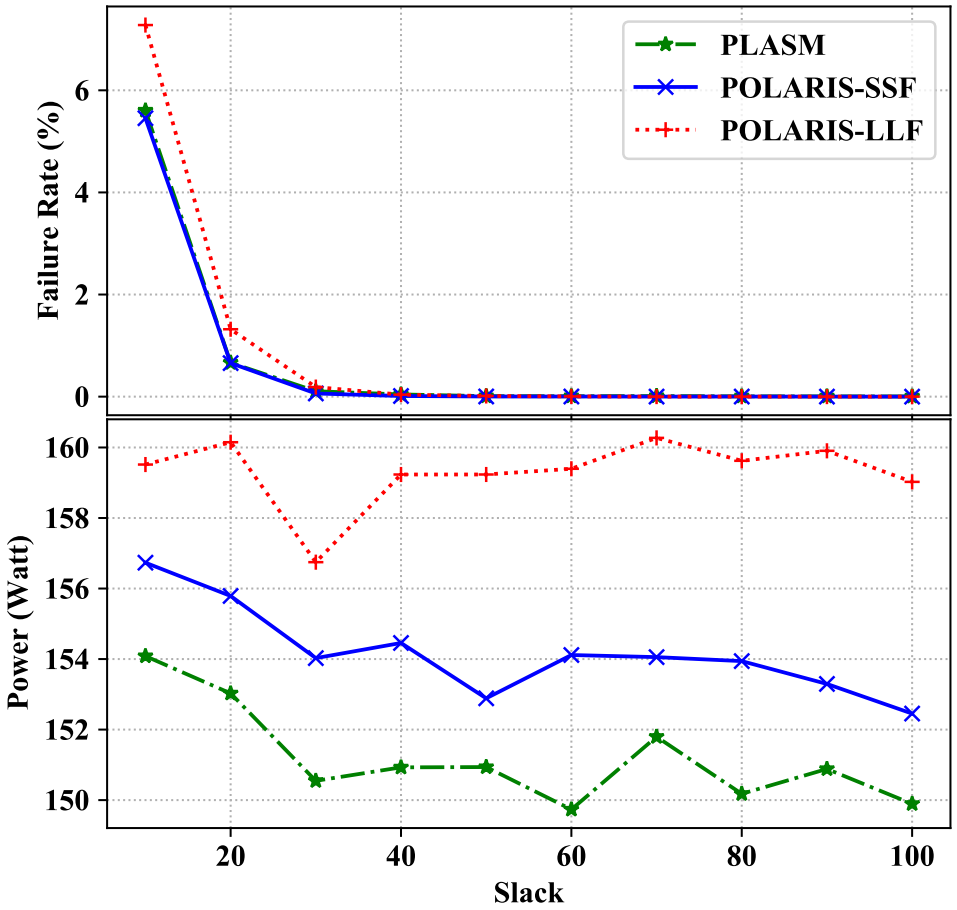


Figure 4.19: Power and performance of POLARIS and its variants under the medium load

When we compare PLASM with POLARIS-LLF, both schedulers perform similarly, except PLASM achieves a slightly better failure rate at tight deadline levels. However, POLARIS-LLF consumes more power than PLASM across all the load and slack levels. FLARE uses frequency as its primary factor for routing and uses load for breaking ties, whereas LLF uses load. When the least loaded worker runs at the lowest speed level, these FLARE and LLF will make the same routing decisions. However, if the least loaded worker is running at higher frequency than other workers - which may occur depending on the deadlines of requests in the workers' queues - LLF will route to the higher frequency worker while FLARE will route elsewhere. This can result in missed deadlines, when slack is tight. However, the more significant effect is increased power consumption.

PLASM and SSF perform similarly, but PLASM consumes slightly less power across all slack levels. These results indicate that PLASM's load-based tie-breaker has little effect on failure rates, but does result in some additional power savings.

We also compared PLASM with POLARIS-SSF and POLARIS-LLF in low and high loads where we offer 15000 and 23000 TPS, respectively. We observe similar relative results in both high and low load.

4.7 Conclusion

In this chapter, we have explored multi-processor specific energy-efficient scheduling problems in latency-critical systems, discussed the solution space and presented an energy-efficient scheduling algorithm for latency-critical systems running on multi-processor CPUs.

PLASM uses a FLARE, which, unlike the generic routers, uses per-processor information. FLARE is a lightweight mechanism and approximates an expensive greedy router that simulates POLARIS at each processor for each routing decision. On our server, PLASM saved both power consumption and failed request comparing to other routing baselines that work with POLARIS, including POLARIS with RR. More importantly, in almost all load and slack levels, PLASM performs very close to ideal minimum power consumption and maximum performance baselines. By comparing FLARE variations, we showed that both CPU speed and load level are essential for FLARE to achieve this energy efficiency.

Chapter 5

Execution Time Estimation

5.1 Overview

Typical execution time estimation techniques predict how much time it takes to run a given instance of a program in an execution environment such as a data center server. These techniques may use some features about the program instance and the environment for estimation. In general, their objective is to minimize estimation error [134, 177].

Execution time estimation is an essential part of POLARIS, as described in Chapter 3. It is also important for PLASM, since PLASM relies on POLARIS to manage each core's work queues, and PLASM uses POLARIS estimates to characterize the load on each worker. POLARIS imposes some particular requirements on its estimates. First, it requires an estimator that can predict program execution times at the different CPU frequencies under which the program might be run. Furthermore, as we describe in Section 5.2, these estimates are not independent. For a given program, the execution time estimates for higher frequencies should be lower than estimates for lower frequencies. Second, POLARIS needs conservative estimates. That is, underestimation is a bigger problem than overestimation.

In this Chapter, our goal is to define an execution time estimator that produces execution time estimation for a given workload type and a frequency level. Similar to the model in Section 3.3, the execution time estimator uses observed execution times for each frequency level and workload type combinations to model the underlying system.

In the remainder of this chapter, we first (Section 5.2) describe the desired estimator characteristics for POLARIS in more detail. After surveying some existing execution time estimation techniques in Section 5.3, we introduce (Section 5.4) a variety of estimation

techniques that can be used with POLARIS, and discuss their strengths and weaknesses with respect to POLARIS's requirements. Finally, in Section 5.5, we present an empirical analysis of the impact of estimation on the performance of POLARIS and provide an empirical look at the properties of the various POLARIS estimators.

5.2 Estimator Properties

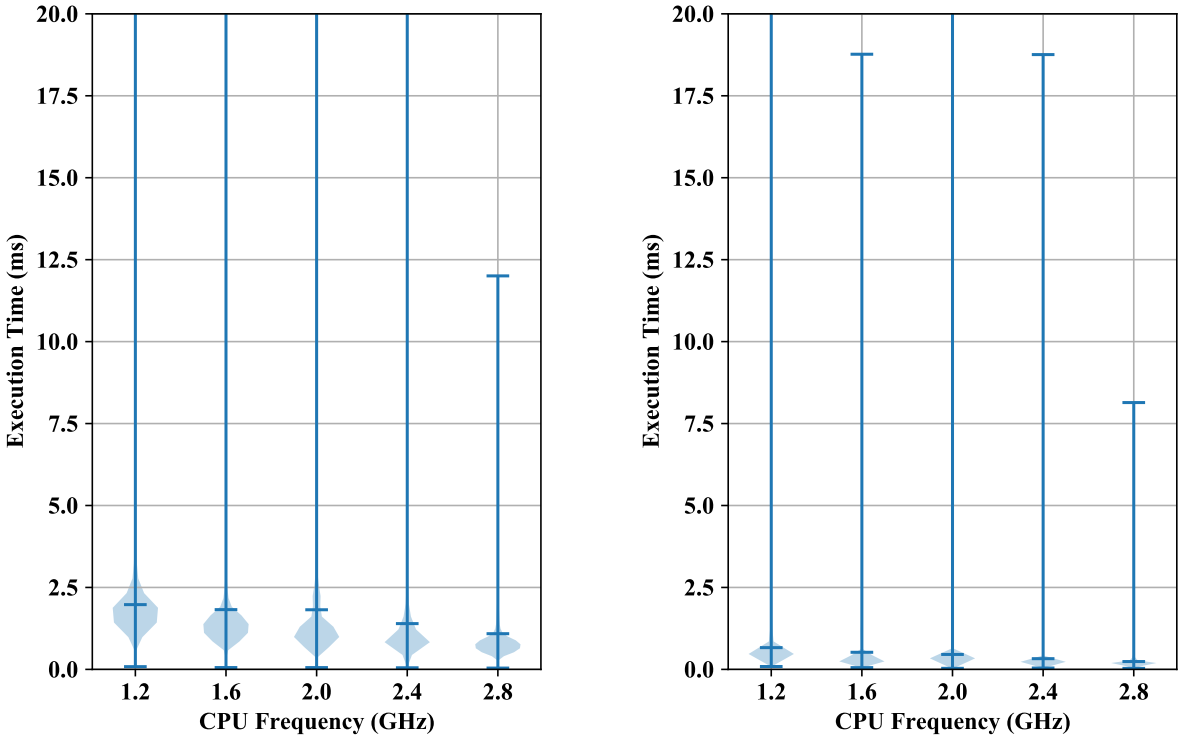
POLARIS relies on an estimator that predicts the execution time of a single transaction from a specified workload class, at a specified execution frequency. In this section, we describe the desired properties of this estimator in more detail.

5.2.1 Conservative

The first property of a POLARIS estimator is that it should be conservative. That is, it should prefer overestimation to underestimation. Estimation errors are inevitable, and error margins can be significant when there is a high execution time variance, as is common in data intensive systems [50, 99, 123, 144]. For example, Figures 5.1(a) and (b) show the execution time distributions of TPC-C New Order and Payment transactions at different frequencies. Execution times for each type of transaction vary significantly. The underlying cause of these variations is contention for resources, including hardware resources, software resources, and data. Previous work [77] has shown that the lifetime of a request may contain a significant overhead caused by contention in shared resources such as buffer pool and lock manager, and contention for shared resources is a well-known source of variability and tail latency in latency-critical systems [50].

In POLARIS, underestimation can lead to missed transaction deadlines. This is because underestimation may cause POLARIS to choose a processor frequency that is too low, resulting in one or more missed transaction deadlines. Perhaps surprisingly, underestimates may also result in increased power consumption. When a transaction's actual execution time is longer than POLARIS' estimate, POLARIS may be forced to compensate for that by increasing the execution frequencies for later transactions. Because of the convex relationship between frequency and dynamic power consumption, such slow-then-fast executions result in greater power consumption than steady execution an intermediate frequency.

In contrast to this, overestimates can lead to increased power consumption in POLARIS, but they do not cause transactions to miss deadlines. Since POLARIS's primary objective



(a) New Order

(b) Payment

Figure 5.1: TPC-C transaction execution time distribution under various CPU frequency levels. Each distribution is represented by a violin plot. Horizontal bars in each plot are used to show the minimum, maximum, and mean. Some distribution is truncated that maximum execution times were as high as 46 ms for New Order and 38 ms for Payment.

is to avoid missing deadlines, its estimates should be conservative.

5.2.2 Tunable

We have argued that POLARIS should prefer overestimation to underestimation, but at what point does overestimation become excessive? Extreme overestimation will cause POLARIS to choose high-frequency execution all of the time, squandering opportunities for power savings while doing little to reduce the likelihood of transaction deadline misses. Thus, the degree of conservativeness of the POLARIS estimator controls a trade-off between deadline misses and power savings. Ideally, the estimator should be conservative enough

to eliminate most of the risk of deadline misses while still allowing POLARIS to choose power-saving frequencies.

Since this tradeoff may depend on the workload and the execution environment, the conservativeness of POLARIS's estimator should be tunable. A tunable estimator allows POLARIS itself to be tuned to balance potential power savings against the risk of missed transaction deadlines.

5.2.3 Frequency Monotonic

The POLARIS algorithm assumes that increasing the execution frequency will not increase the execution time of any request, an assumption that generally holds in practice. Since POLARIS uses an estimator to predict the execution times of individual tasks, it is important that that estimator preserve this property. Thus, if f_1 and f_2 are execution frequencies ($f_1 < f_2$) and $\hat{x}_{f_1}$ and $\hat{x}_{f_2}$ are execution time estimates for those frequencies (for a given workload class), then it should be the case that $\hat{x}_{f_1} \geq \hat{x}_{f_2}$. We refer to this property as *frequency monotonicty*.

If POLARIS uses an estimator that is not frequency monotonic, it may fail to make effective use of some of the CPU frequencies available to it. That is, if $\hat{x}_{f_1} < \hat{x}_{f_2}$, POLARIS will never choose to set the processor frequency to f_2. If f_1 is inadequate, POLARIS will jump to some $f_3 > f_2$ instead of choosing f_2, since f_2 appears to offer no benefit. As a result, the system may end up consuming more power than is actually necessary to handle the given load.

5.2.4 Lightweight

Finally, a POLARIS estimator needs to be lightweight, due to the practical challenges of scheduling in latency-aware data systems. The POLARIS algorithm needs to obtain execution time estimates each time it runs, which means each time a request arrives or is completed. Since latency-critical tasks are short, we need to ensure that estimation overhead is not significant. Similarly, estimation consumes power, which works against the objectives of POLARIS and FLARE. Therefore the estimation mechanism must be lightweight, to minimize its impact on performance and power consumption.

5.3 Related Work

Execution time estimation is a common problem. In this section, we give a brief overview of work in this area.

One body of related work focuses on generic performance estimation techniques which can be applied to a wide range of applications. Some of this work explicitly considers DVFS and the impact of CPU frequency on execution time [56, 100, 146, 160]. One approach is to profile application execution at a particular frequency and then use the profile to estimate execution time for the same application at different frequencies. Profiling is used to break down total execution time into frequency-sensitive and frequency-insensitive parts, such as delays due to memory access. The estimator can then linearly scale the CPU frequency sensitive portion to estimate execution times at different frequencies.

Other techniques rely on extracting information from application source code that can be used to guide estimation. Brandolese et al. [30] define a set of elementary components in an arbitrary C source code called atoms, and they analyze source code to find the count of these atoms. Finally, they estimate execution time with a white-box approach using the atom counts and unit atom costs. Huang et al. [85] use a black-box model for source-code-based estimation. They propose machine learning techniques for identifying features, such as loop and branch counts, the most strongly affects execution time, and then they use these features in a polynomial regression to make execution time estimates.

These techniques are complementary to the approaches we discuss in this chapter, in that they identify features of programs or program executions that provide useful input for execution time estimation. Such features could be included in the estimators used by POLARIS, allowing it to refine its estimates and perhaps allowing some of the observed execution time variance to be explained.

Execution time estimation has also been widely studied in the context of database management systems, to address a variety of problems including resource sizing [170], progress monitoring [176] and request scheduling [68, 105]. In the database setting, the units of work are typically database queries rather than arbitrary programs. Execution time estimators can take advantage of knowledge of query structure and database characteristics to help estimate execution times.

Wu et al. [177] use PostgreSQL's query cost model for execution time estimation. The cost model describes the cost of query operations in terms of primitive costs, such as page accesses and normalized CPU time units. To estimate execution times, the authors calibrate primitive cost units for the underlying hardware. Similarly, in a separate work [176] focusing on parallel query execution, the authors uses a similar approach to model the

execution time of subqueries, and then model parallel subquery interaction to estimate overall query execution time. Wu et al [178] further extended this approach to incorporate an explicit notion of uncertainty into their cost-based approach. Instead of reporting point estimates, this allows them to report a range of possible execution times, with an associated probability distribution.. This approach could potentially be adapted to produce the conservative execution time estimates needed by POLARIS, by choosing estimates near the tail of the reported distribution.

Cost estimates from database query optimizers can also be used as input features for black-box query execution time models. Akdere et al. [6] use support vector machine for query-level estimation and linear regression for operator level estimation. For both approaches, they rely on features exposed by the query optimizer. Ganapathi et al. [60] also rely on query optimizer features for estimation. They first identify critical features using canonical correlation analysis before mapping features to query performance.

Duggan et al. [53] use linear regression to estimate execution time as a function of buffer access latencies in I/O-bound systems. They also model buffer access latency in the presence of concurrent queries to estimate performance of parallel queries. Gupta et al. [70] use both historical data and system load information to perform execution time classification of database queries. Their approach uses a decision tree to classify queries, and associates an execution time range with each leaf classification.

A relatively small body of works focuses on estimation in latency-critical data systems. Mozafari et al. [134] build a white box model tailored for MySQL, for predicting the maximum system throughput. They analyze three different factors in database transactions: I/O, CPU and concurrency, and identify a bottleneck resource to estimate maximum throughput. For that they use related events such as log writes, dirty pages writebacks, and I/O due to cache misses. The results show that their system-specific white-box approach can successfully estimate the throughput for I/O-bound, CPU-bound, and lock-bound workloads. In a separate work [135], the authors consider a linear regression technique that uses a separate linear-regression model per workload type (e.g., I/O-bound workloads), instead of using a single global linear regression model.

Rubik [99] models the amount of work per request with a probability distribution, and uses convolution to determine a probability distribution over the total amount of work required for a sequential queue of requests. For better accuracy, the Rubik's estimator splits work into memory and CPU parts, and assumes that only the execution time for the CPU part will scale with CPU frequency. Since Rubik works directly with full probability distributions, it can easily make conservative estimates of the amount of time that will be required to execute individual requests or all of the requests in the queue. However, Rubik

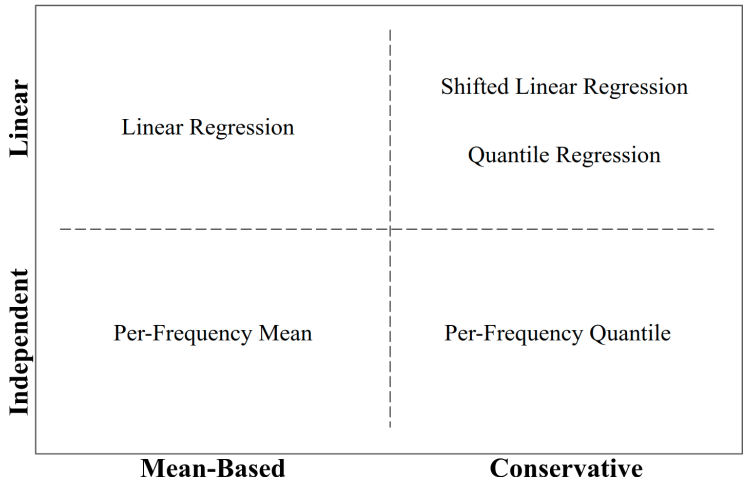


Figure 5.2: Categorization of the execution time estimation baselines presented in this chapter.

assumes that there is only a single type of request, which simplifies the task of determining the probably distribution for the total execution time of all of the requests in a queue.

5.4 POLARIS Estimators

In this section, we describe a space of possible estimators that can be used with POLARIS. We describe specific estimators within this space, and consider them in the light of the estimator properties that we define in Section 5.2.

Figure 5.2 shows the space of estimators that we consider. We first categorize them as either independent or linear. Independent estimators use a separate model for each frequency level and do not enforce any constraints across the estimates at different frequency levels. On the other hand, linear estimators use a single model parameterized by frequency, allowing them to enforce a linear (or other) relationship between estimates at different frequencies.

The second dimension classifies estimators as either conservative or mean-based. As the name suggests, mean-based estimators estimate mean execution times. Mean-based estimators are widely used [53, 134, 176, 177]. Conservative estimators, on the other hand, estimate other statistics - such as quantiles - that describe the tail of the execution time distribution.

Notation	Meaning
$\mathcal{F}$	set of possible processor frequencies
f	a specific processor frequency from $\mathcal{F}$
X_f	set of execution time observations of under frequency f
X	set of execution time observations under all frequency levels $f \in \mathcal{F}$
x_f^i	i^{th} observation in X_f
$\hat{x}_f$	execution time estimation for frequency f
P_i	i^{th} percentile

Figure 5.3: Summary of Notation

All of the estimators that we consider are black-box estimators that use actual execution time observations to construct estimates. We collect the observations before running the actual tests during a separate training phase. In the following, we assume that we have a set X of measurements of execution times of requests from the workload class for which we are trying to construct an estimator. X includes measurements taken at all possible CPU frequencies, and we use $X_f \subset X$ to represent the set of measurements taken at frequency f. Figure 5.3 summarizes the notation that we use to describe the estimators.

5.4.1 Per-Frequency Mean

The simplest approach to estimation in our space is to separately and independently estimate the mean execution time at each possible execution frequency in $\mathcal{F}$. We can get an unbiased estimate of the mean by taking the mean of our observations at each frequency:

$$\hat{x}_f = \frac{\sum_{x_f^i \in X_f} x_f^i}{|X_f|}, f \in \mathcal{F} \tag{5.1}$$

Although this approach is simple, it is neither conservative nor tunable, and we will show later that using estimates of the the mean can lead to large numbers of missed deadlines, since many requests may have execution times well above the mean.

Since this approach generates a separate estimator for each frequency, estimates produced by this approach are also not guaranteed to be frequency monotonic. In practice, however, we do expect mean execution times to decrease with increasing frequency. Modulo sample error, estimated execution times should also decrease with increasing frequency under this approach.

We used this simple technique for estimation in our preliminary energy-efficiency scheduling mechanism called LAPS [106].

5.4.2 Per-Frequency Percentile

The second type of estimator we consider also constructs a separate independent execution time estimate for each execution frequency. However, instead of estimating the mean, this approach estimates a specified quantile or percentile in the execution time distribution based on the observations at each frequency (X_f). The selected quantile or percentile is a tunable parameter of the estimator. For example, a P_{90} estimator estimates an execution time (for a particular frequency) such that 90% of requests should have execution times at or below the estimated value.

There are many techniques for estimating the quantiles or percentiles of a distribution from a set of observations [107]. By choosing high percentiles or quantiles, this approach generates conservative estimates, e.g., with a P_{90} estimator, actual execution times are much more likely to be below the estimate than above it. Furthermore, these estimators are easily tunable, as they are parameterized by a target percentile or quantile. In our evaluation of POLARIS in Chapter 3, we used per-frequency quantile estimates because of these properties.

One challenge with per-frequency quantile estimation is that it does not guarantee frequency monotonicity. In contrast with per-frequency means, which are likely to be frequency monotonic in practice, we regularly observed per-frequency quantile estimates that are non-monotonic, particularly for conservative quantiles that are out on the tail of the distribution. We show examples of non-monotonic per-frequency quantile estimates later in this chapter.

5.4.3 Linear Regression

The two per-frequency estimators we have presented so far have the common shortcoming that they do not guarantee frequency monotonicity. We can avoid this problem by moving away from constructing a separate estimator for each frequency. Instead, we can construct a single estimator that estimates execution time as a function of frequency. This allows us to constrain the relationship between estimates at different frequencies.

A widely used technique in this category is linear regression, which generates an estimate of the form

$$\hat{x}_f = a + bf, f \in \mathcal{F} \tag{5.2}$$

The intercept (a) and slope (b) of the regression line are chosen to minimize the mean squared error of the line with respect to the available execution time observations at all frequencies (all observations in X).

Linear regression is will suited to estimating the effect of frequency on execution time, since that effect is expected to be linear [56, 100, 146]. Its primary drawback is that, like per-frequency estimates of the mean, linear regression estimates are not conservative.

Another potential challenge with linear regression is that it does not guarantee frequency monotonicity. Specifically, it does not guarantee that the slope of the regression line (b) will be negative, so that higher frequencies lead to lower execution time estimates. This problem can be fixed by modifying the regression to include an explicit constraint on the sign of the slope b. However, we have found that this problem is unlikely to occur in practice. Higher frequencies do result in faster executions, so linear regression over a sufficiently large set of observations is likely to produce a regression line with a negative slope. We will return to this issue in Section 5.5.

5.4.4 Shifted Linear Regression

One approach for obtaining estimates that are both conservative and monotonic is to start with a linear regression, and then shift the regression line up to make the estimates more conservative.

Suppose that linear regression finds the relationship

$$\hat{x}_f = a + bf \tag{5.3}$$

for some intercept a and slope b. In shifted linear regression, we shift the line by replacing a with a new intercept a_τ, while keeping the slope (b) constant. Here, $0 \leq \tau \leq 1$, is a parameter that controls the conservativeness of the new estimator.

Suppose that $P_\tau(f)$ represents the per-frequency τ-percentile estimate for frequency f. To shift the regression line, we choose the smallest a_τ such that

$$a_\tau + bf \geq P_\tau(f) \tag{5.4}$$

for all frequencies $f \in \mathcal{F}$. That is, we lift the linear regression line until the execution time estimate at every frequency is at least as high as the per-frequency τ-percentile estimate.

Shifted linear regression estimates are frequency monotonic as long as the underlying linear regression is frequency monotonic. They are also conservative and tunable, through the choice of τ.

5.4.5 Quantile Regression

Another way to obtain a linear estimator that is conservative is to use linear quantile regression [103]. Linear quantile regression estimates conditional quantiles (or percentiles) of the execution time distribution. Linear quantile regression is similar to linear regression, except that overestimation and underestimation errors are treated asymmetrically. For example, in the quantile regression described by Koenker [103], for P_{90} linear quantile estimate, underestimation errors are weighted with 0.9, whereas underestimations are weighted with 0.1. Frequency level is the independent variable in quantile regression, as it is for the linear regression and shifted linear regression estimators.

Unlike linear regression, linear quantile regression can produce estimates that are conservative (by choosing high quantiles) and tunable. However, like linear regression, linear quantile estimates are not guaranteed to be frequency monotonic. That is, execution time estimates for higher frequencies are not guaranteed to be lower than estimates at lower frequencies. For linear regression, this is not a problem in practice. However, it is a more significant problem for linear quantile regression, particularly for quantiles far out on the tail of execution time distribution, because outlying quantiles are not as well-behaved as the mean. In Section 5.5, we will show that linear quantile estimation can result in practice in execution time estimates that are not frequency monotonic.

5.5 Evaluation

In this section, we present an empirical comparison of the various types of estimators described in Section 5.4. Our primary goal is to understand the impact of conservativeness: how important is it to have a conservative estimator, and how conservative should it be? Our secondary goal is to gain some insight into the estimates themselves. In particular, we would like to understand how independent per-frequency estimators differ from linear estimators in practice.

5.5.1 Impact of Conservative Estimation

In our first set of experiments, we consider the impact of two different estimators on the performance (failure rate) and power consumption of PLASM. The two estimators are shifted linear regression and linear regression, one of which is conservative, the other is not.

In our evaluation, we use the same experimental setup and methodology described in Section 4.6, running the TPC-C benchmark using Shore-MT with PLASM scheduling requests and managing CPU speed. We run experiments at three different load levels, and with varying deadline slacks.

Figure 5.4 shows failure rate and power consumption achieved by PLASM using the two different estimators, under high load (23000 TPS). Our first observation is that having a conservative estimator is important for keeping failure rates low. With non-conservative estimates obtained from linear regression, failure rates are much higher, and the "failure gap" grows as slack gets tighter. When slack is tight, the POLARIS scheduler has less flexibility and is more sensitive to estimation errors. A single unexpectedly slow transaction can result not only in the slow transaction missing its deadline, but in other queued transactions missing their deadlines as well.

The impact of conservativeness on power consumption is less clear. As we showed earlier, power savings are not very significant at high load because the opportunity gap is small. We expected to see that PLASM would consume less power with the linear regression estimates than with the conservative estimates, and this is true when slack is tight. When slack is loose, however, conservative estimates result in both lower failure rates *and* slightly lower power consumption. When slack is loose, POLARIS tries to take advantage of this flexibility to run transactions slowly and save power. This allows request queues to build up. If a transaction takes much longer to run than POLARIS estimated, POLARIS has to react by switching to a higher frequency to ensure that the remaining transactions in the queue do not miss their deadlines. These slow-then-fast frequency patterns lead to higher power consumption. In contrast, with conservative estimates, POLARIS tends to start with a higher frequency. It is less likely to underestimate execution times in the first place, and if an underestimation does occur it is less likely to force POLARIS to increase frequency to accommodate the remaining transactions in the queue. Thus, conservative estimates allow POLARIS to maintain a steady frequency and avoid slow-then-fast cycles.

Figure 5.5 shows the results of similar experiments under medium and low loads. The results at these load levels are qualitatively similar to the high-load results. There is still a "failure gap". It is not as large in absolute terms as the gap at high-load. Nonetheless, failure rates with linear regression estimates can be two or three times as high as those with the conservative estimator when slack is tight. Differences in power consumption are small.

In summary, conservative estimators result in lower failure rates, especially at higher loads. Conservative estimation sometimes results in higher power consumption, particularly when load is high and slack is tight, but differences in power consumption are small.

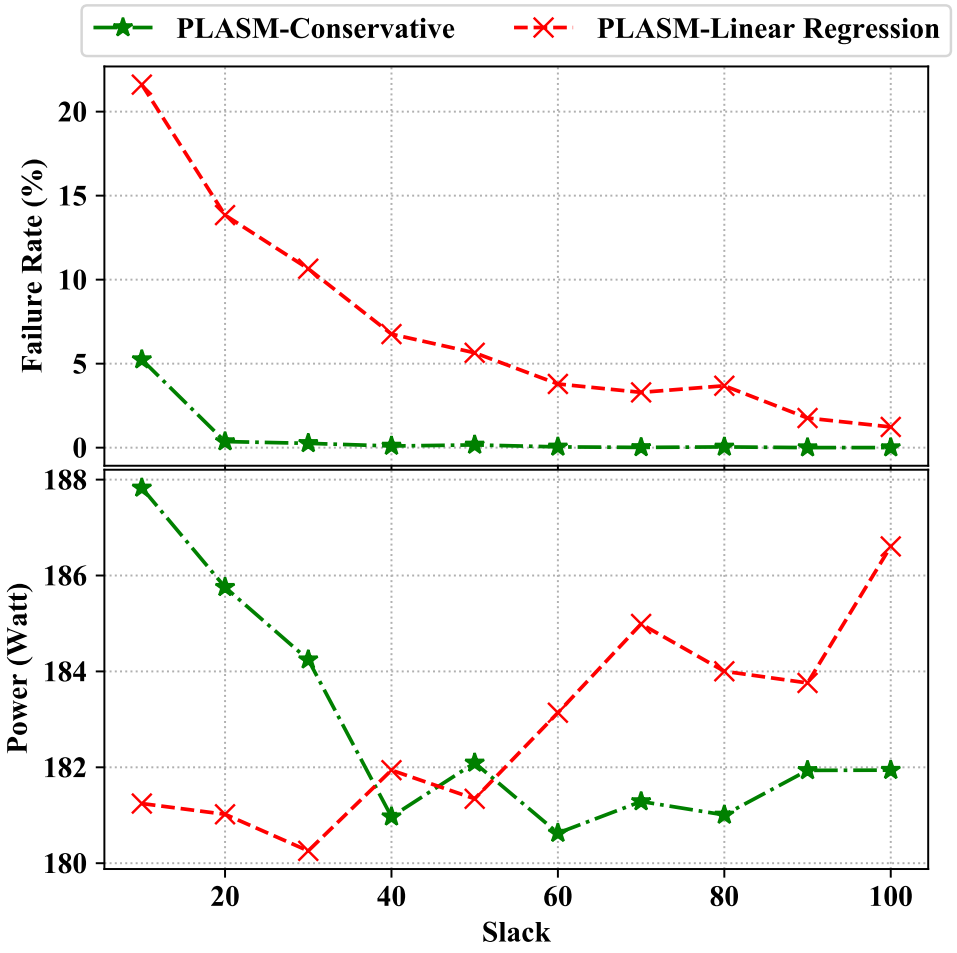
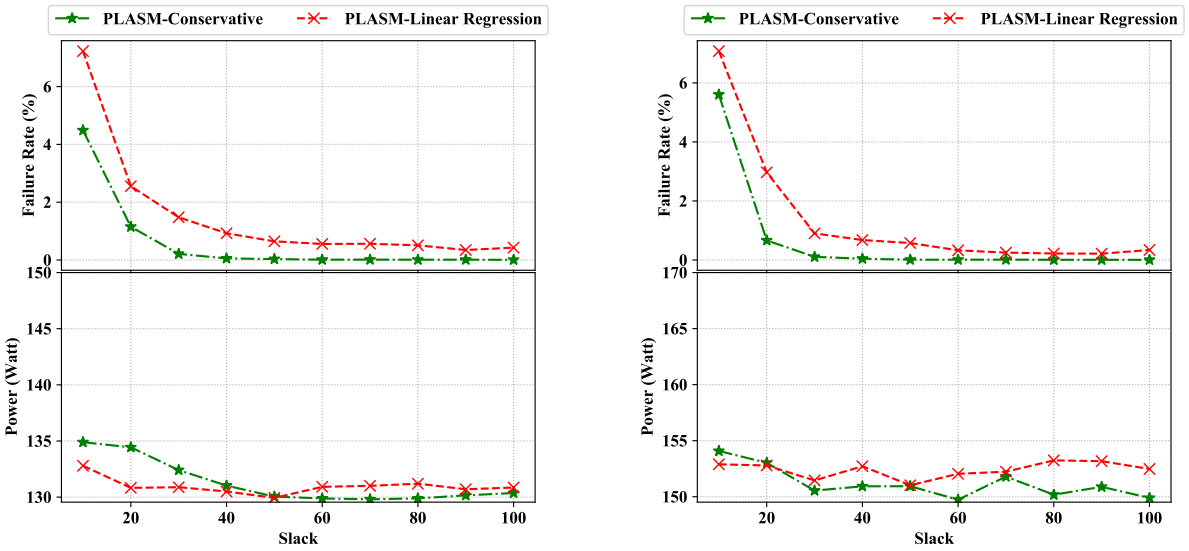


Figure 5.4: PLASM with two different estimators, at High Load (23000 TPS)



(a) Low Load (15000 TPS)

(b) Medium Load (19000 TPS)

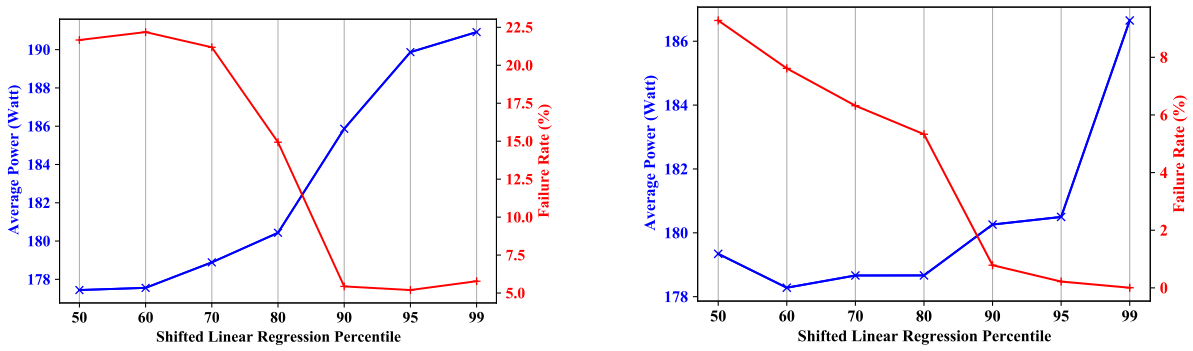
Figure 5.5: PLASM with estimators with different conservativeness

5.5.2 How Conservative?

In our previous experiments, we showed that a conservative estimator reduces failure rates without introducing a substantial power penalty. Here, we consider the question of how conservative an estimator needs to be to achieve these benefits.

To answer this question, we ran experiments with PLASM using a shifted linear regression estimator with varying degrees of "shift", ranging from P_{60} to P_{99}. Again, we use the same experimental setup and methodology described in Section 4.6. We run experiments at two slack settings (10 and 40) and at high (23000 TPS) and low (15000 TPS) loads.

Figure 5.6 shows failure rate and power consumption as a function of the conservativeness of the shifted linear regression model, at high load (23000 TPS). These results show that the conservativeness of the estimator controls a tradeoff between transaction failures and power consumption. For our workload, increasing the regression percentile (conservativeness) of the estimator results in substantial reductions in failure rates until the percentile reaches 90%, after which there is little additional benefit. One the power front, we see little change in power consumption for percentiles in the 50%-80% range, but power does increase when the regression percentile exceeds 80%. Thus, at least for our workload, there is a "sweet spot" in the range of 80%-90% in which PLASM achieves most



(a) Tight deadline (slack 10)

(b) Loose deadline (slack 40)

Figure 5.6: Power consumption and failure rate of PLASM under high load (23000 TPS) with shifted linear regression estimator using different percentiles

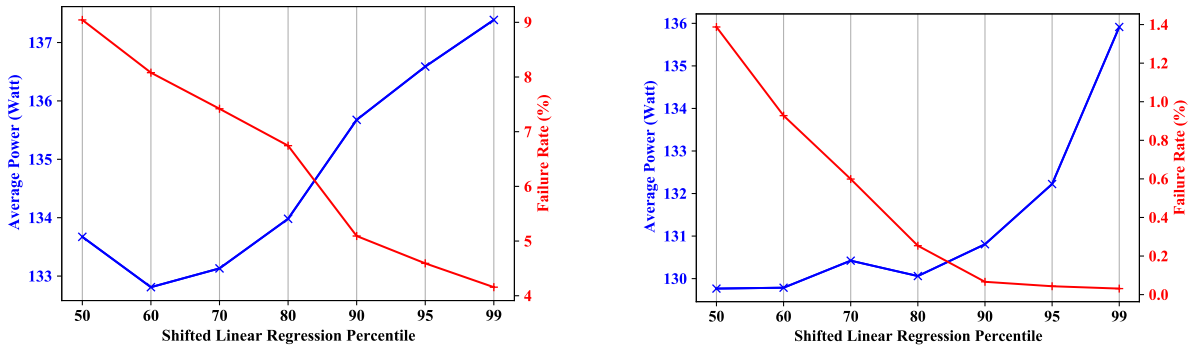
of the failure rate reduction without a significant power penalty.

Figure 5.7 shows the result of the same experiment run at low load. Absolute failure rates and power consumption are lower at low load, but we observe behavior that is similar to what we observed under high load. Specifically, using a regression percentile in the 80%-90% range achieves most of the available failure rate reduction with little power penalty.

5.5.3 Characterization of Estimates

Our previous experiments have focused on how the choice of estimator, and estimator parameters, affects the overall performance of the PLASM scheduler. In our next set of experiments, we hope to gain some insight into different types of estimators by directly observing and comparing the estimates they produce.

In these experiments, we use our PLASM prototype and the TPC-C workload to run only the training (estimator calibration) phase of our usual experimental methodology. That is, after warming up the system, we run TPC-C workload with an average of 320000 instances of each transaction type, including 64000 instances at each frequency level. and we record the transaction execution times. Using this execution time dataset, we then construct estimators using the five different techniques described in Section 5.4. For all of the conservative estimators, we used the P_{95} percentile. Finally, we compare the estimates generated by each technique.



(a) Tight deadline (slack 10)

(b) Loose deadline (slack 40)

Figure 5.7: Power consumption and failure rate of PLASM under low load (15000 TPS) with shifted linear regression estimator using different percentiles

Figure 5.8 shows the execution time estimates produced for the TPC-C Payment transaction type by all five estimators described in Section 5.4. In addition to the estimates themselves, the figure also includes a box plot at each frequency to illustrates the distribution of our execution time measurements, on which the estimates are based.

Our first observation is that both mean-based estimators (per-frequency mean and linear regression) produce very similar estimates. Although the per-frequency mean estimator does not guarantee frequency monotonic estimates, the estimates it produces are in fact frequency monotonic - we observed this to be the case for all TPC-C transaction types. This reflects the fact that the impact of a frequency increase on the execution time of any individual Payment transaction should be a decrease in execution time. Thus, over the entire sample of transactions at each frequency, we should expect a decrease in mean execution time as frequency increases.

For the conservative estimators, we cannot be so certain about frequency monotonicity. Both of the linear conservative estimators (shifted linear regression and quantile regression) produce frequency monotonic estimates for the Payment transaction, but the per-frequency quantile estimator does not - its estimate for 2 GHz is higher than its estimate for 1.6 GHz. This simply reflects the fact that estimates far out on the tail of the execution time distribution are not as well-behaved as the mean, and the per-frequency estimator has no constraints that force its estimates to be linear or frequency-monotonic.

Another observation is that the shifted linear estimator results in much more conservative estimates at high frequency than the linear quantile estimator. This is because the shifted linear estimator preserves the slope of the linear regression estimate, while the lin-

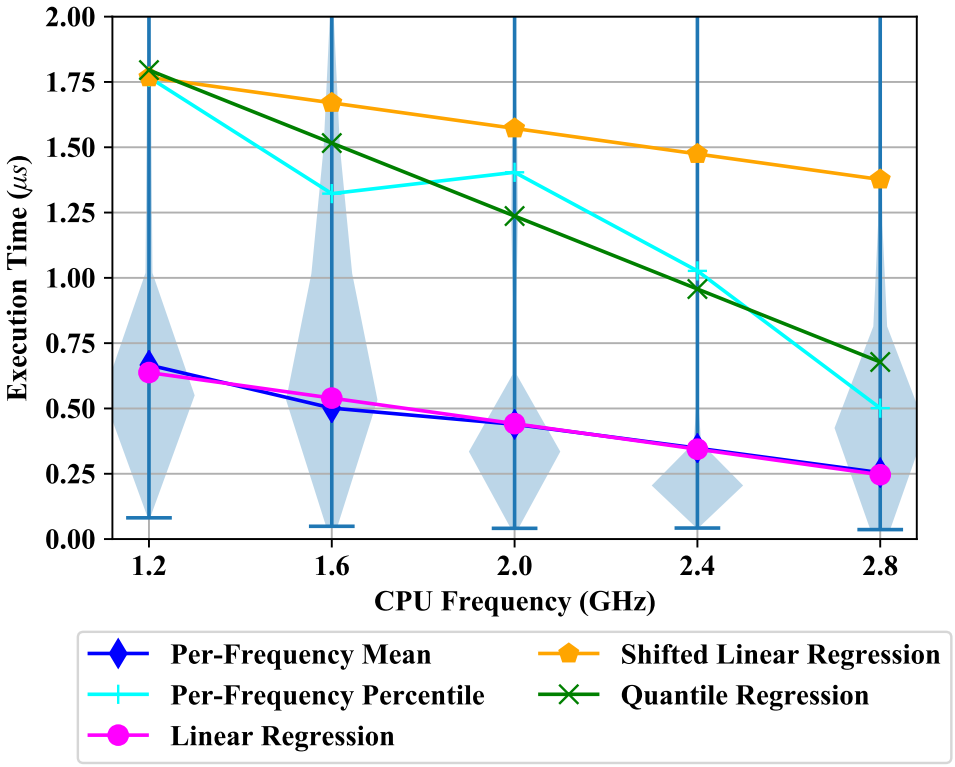


Figure 5.8: TPC-C Payment transaction Estimations

ear quantile estimator does not. For the Payment transaction, both the mean execution time and the execution time variance drop as frequency increases. Thus, the linear quantile estimates drop more quickly than the mean as frequency increases.

Finally, Figure 5.9 show the estimates for the TPC-C NewOrder transaction, which is larger and more complex than the Payment transaction. As was the case for the Payment transaction, the linear regression and per-frequency mean estimators produce almost identical estimates. Also, the per-frequency quantile estimates are once again non-monotonic with frequency, including a very substantial jump in the estimate as frequency increases from 1.2 GHz to 1.6 GHz. One significant difference between the NewOrder and Payment estimates is that the NewOrder estimates produced by the linear quantile estimator are not frequency monotonic either - higher frequencies result in higher execution time estimates.

These results illustrates that while we can expect frequency monotonicity when estimating mean execution time, we cannot expect it when making conservative estimates

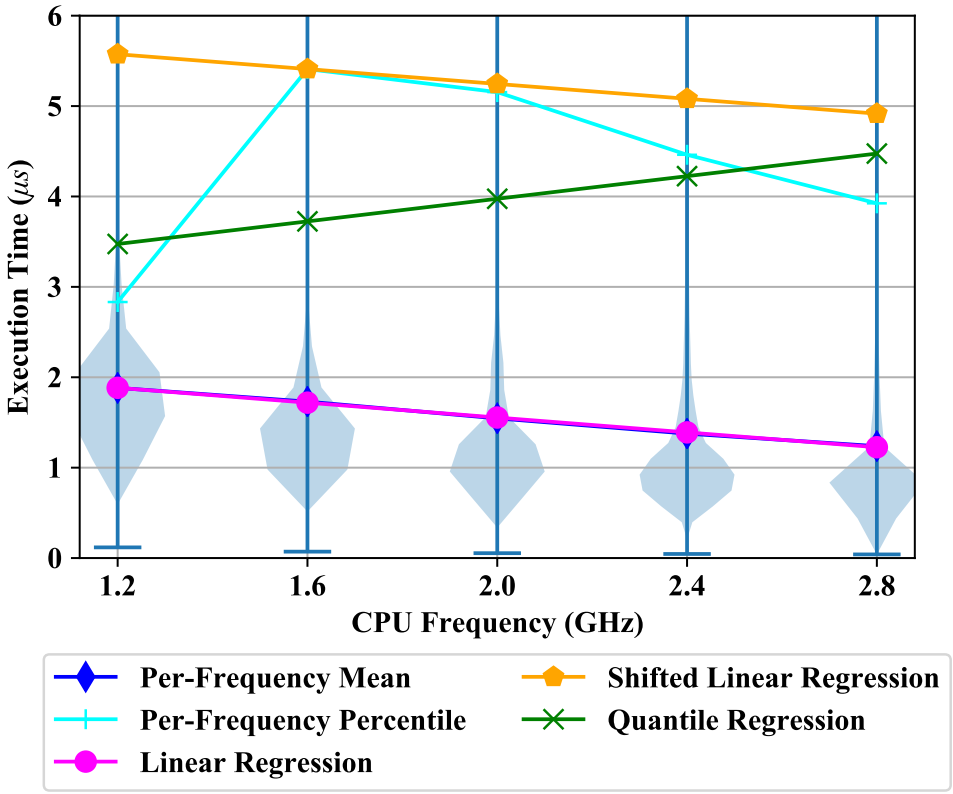


Figure 5.9: TPC-C New Order transaction Estimations

further out on the tail of the execution time distribution. Thus, in order to use linear quantile estimation with PLASM, it is important to explicitly constrain the resulting estimate to have a negative slope. Alternatively, shifted linear regression can be used, as it will produce conservative frequency monotonic estimates as long as the linear regression line is frequency monotonic.

5.6 Conclusion

In this chapter, we examine the execution time estimation component of PLASM. We discuss some properties that help an estimator work well with PLASM and presented a variety of estimation techniques that offer some or all of these properties. By comparing PLASM results using different estimators, we showed that conservative estimation is im-

portant - particularly for keeping transaction failure rates low. We also showed that it can be challenging to obtain estimates that are both conservative and frequency monotonic, since execution time quantiles are not as well behaved as the mean as execution frequency is varied.

Chapter 6

Related Work

Previous chapters presented some related work specific to those chapters. Section 3.4, presents theoretical work on single-processor energy-efficient scheduling. Sections 4.2 and 4.4, cover work on energy-efficient and general multi-processor scheduling. Finally, Section 5.3 presents work on execution time estimation techniques.

This chapter presents related work on energy-efficiency in software systems, in several broad categories. Section 6.1 presents techniques that are designed to operate across multiple servers, while Section 6.2 covers single-server techniques. Finally, in Section 6.3, we consider techniques that have been specifically targeted at database systems.

6.1 Cluster Level Energy Efficiency

Some approaches for improving data center energy efficiency operate at the scale of a cluster or data center as a whole. One technique is to shut down servers when they are idle [112, 118]. Another set of techniques focus on the energy-efficient virtual machine placement across the cluster [113, 179]. Facebook controls server power consumption to prevent data center power overloads [175].

These techniques typically operate at much longer time scales (e.g., minutes or hours) than POLARIS and PLASM, typically because the actions used to control power consumption, such as placing or migrating virtual machines, or powering servers up and down, are relatively time consuming. POLARIS and PLASM are complementary to some of these techniques. For example, they can be used to manage DVFS on servers that are not shut down by a cluster-level manager.

6.2 Server-Level Energy Efficiency

Another body of work targets single server energy efficiency, like POLARIS and PLASM. GreenRT [36] finds the slowest CPU speed that satisfies the deadlines of periodic tasks in soft real-time systems. Spiliopoulos et al. [156] propose an operating system power governor which uses memory stalls as an input and tries to optimize CPU energy efficiency accordingly. Sen and Wood [150] propose an operating system governor that predicts the system power/performance pareto optimality frontier and keeps the power/performance at this frontier. Like the Linux DVFS governors we have used as baselines in Section 3.6, these do not take advantage of application-level workload information. Weiser et al. [171] propose a generic energy-efficient scheduling algorithm at the operating system level to maximize the number of instructions per unit of energy. We refer to Zhuravlev et al. [184] for a comprehensive survey of energy-aware scheduling techniques at the operating system level.

PAT [181] and PEGASUS [123] apply feedback control to manage processor DVFS. PAT uses a control mechanism based on a simple system model to maintain a target system throughput as the I/O intensity of the workload fluctuates. However, similar to other feedback mechanisms, it focuses on history for power adjustment, and it does not understand the latency requirements of waiting requests. as the I/O intensity of the workload fluctuates. However, this may be difficult to apply in a system in which the intensity of the offered load is fluctuating. PEGASUS, like POLARIS, targets request latency in so-called on-line data intensive (OLDI) applications. PEGASUS assumes a homogeneous workload, with a target request latency, and it manages DVFS to try to barely meet this target, so as to maximize power savings. Thus, its broad objectives are similar to POLARIS's. However, because it uses feedback control over a large distributed system, which requires time to observe system state and adjust to fluctuations, it is intended to react to changes over longer time scales (minutes, hours, days), not in response to individual request completions and arrivals, like POLARIS. Unlike POLARIS, PEGASUS is unable to accommodate multiple concurrent workloads.

Rubik [99] manages DVFS on the time scale of individual transaction arrivals, like POLARIS. Rubik uses statistical models to try to predict the tail latency of the response times of all queued transactions, and uses DVFS to try to ensure that they hit latency targets. However, this approach does not extend to multiple workloads, since the response time prediction must be done periodically, off-line, and it assumes that all requests have identical service time distributions. Both techniques are limited to controlling DVFS, and do not reorder transactions like POLARIS.

Several studies explore the use of C- States for energy efficiency. These studies show that using C-states is challenging either because workloads are rarely idle enough to exploit sleep states [124, 130] or because processors consume a lot of energy to recover from deep sleep states [97, 148]. Therefore, some work encourages deeper C-States by extending sleep periods [12, 129]. In contrast, we focus only on P-states in this work.

6.3 Energy Efficiency in Database Management Systems

Several studies describe techniques for improving the energy efficiency of database management systems through query optimization and query operator configuration. Tsirogiannis et al. [163] investigate servers equipped with multi-core CPUs by studying power consumption characteristics of parallel operators and query plans using different numbers of cores with different placement schemes. Their findings suggest that using all of the available cores is the most power-efficient option for DBMSs if the system is fully loaded, while DVFS may allow further power/performance tradeoffs. Unfortunately, this does not provide guidance on how to improve energy efficiency in the common case of systems that are not 100% loaded. In the same direction, Psaroudakis et al. [143] take CPU frequency into account along with core selection. They show that different CPU frequency levels can be more energy efficient for execution of different relational operators. Both Xu et al. [180] and Lang et al. [111] explore possibilities of energy aware query optimization in relational DBMSs. For this, they propose a cost function where both performance and power contributes to the cost. They show that DBMSs can execute queries according to specific power/performance requirements. These techniques are largely complementary to POLARIS.

Chapter 7

Conclusions & Future Work

7.1 Conclusions

In this thesis, we provide a solution to the energy efficiency problem in servers hosting latency critical data systems. We presented energy-efficient algorithms and techniques to achieve the objective of minimizing power consumption while maintaining the desired quality of service defined by latency targets.

In Chapter 3, we presented a workload aware single-processor scheduling technique called POLARIS. POLARIS controls processor power as well as execution order. We established a competitive ratios for POLARIS's against an optimal offline non-preemptive energy aware scheduler, and also showed how the on-line non-preemptive nature of POLARIS affects its competitiveness. We prototyped POLARIS in a data system, and showed that it achieves both lower power consumption and fewer missed deadlines under various workload scenarios, relative to OS-based dynamic power governors.

Chapter 4 extended POLARIS to a more generalized case in which there are multiple parallel homogeneous processors. For the multi-processor version of the problem, we presented an energy-efficient multi-processor scheduling algorithm called PLASM. PLASM uses a routing mechanism called FLARE that distributes requests across the processors, each managed by POLARIS. FLARE is a light-weight router and uses summary data to make decisions. We implemented a prototype of PLASM, and showed that it consumes less power with fewer missed deadlines than POLARIS combined with a generic round robin routing. Our results also suggest that there is little room to improve on PLASM, except in settings where latency targets are very tight.

In Chapter 5, we discussed the execution time estimation problem, which is an essential component of both PLASM and FLARE. We identified desirable characteristics for estimators used by POLARIS and FLARE, and presented a simple regression-based estimator with the desired characteristics. We demonstrated empirically that POLARIS's estimates need to be conservative, and showed how conservativeness controls a latency/power scheduling tradeoff.

7.2 Future Work

There are many research directions that would extend the work presented in this thesis. POLARIS and FLARE are both online algorithms, and both make their decisions under the assumption that there will be no future requests in the system. We expect that speculating about the future load would improve energy efficiency. One essential step in this direction would be forecasting the future load [127] and dynamically integrating it to the online schedular.

Other hardware components besides the CPU cores, such as the uncore part of CPUs [102] and memory [98], contribute to overall CPU and server power consumption. The scheduling algorithms presented in this thesis, manage only CPU cores' power. Energy-efficiency could be improved through energy-aware management of those other components.

Another future extension is energy-efficiency in server clusters. In this thesis, we present the energy-efficiency first in single-processors and then in multi-processors. As a future direction, having energy-efficient resource management at the cluster level that possibly works with POLARIS or PLASM can potentially improve energy efficiency at the cluster level.

References

[1] Epfl Official Shore-MT Page, Shore-Kits. https://sites.google.com/site/shoremt/shore-kits. Accessed: Feb. 2017.

[2] Efficiency – data centers. https://www.google.com/about/datacenters/efficiency/, 2020.

[3] Advanced Micro Devices(AMD). Architecture programmer's manual: Volume 2: System programming. (24593), 2017.

[4] Mumtaz Ahmad, Ashraf Aboulnaga, Shivnath Babu, and Kamesh Munagala. Interaction-aware scheduling of report-generation workloads. *The VLDB Journal*, 20(4):589–615, 2011.

[5] Anastasia Ailamaki. The next 700 transaction processing engines. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1–2. ACM, 2017.

[6] M. Akdere et al. Learning-based query performance modeling and prediction. In *Proc. ICDE*, pages 390–401, 2012.

[7] Susanne Albers. Algorithms for dynamic speed scaling. In *Symposium on Theoretical Aspects of Computer Science (STACS2011)*, volume 9, pages 1–11, 2011.

[8] Susanne Albers, Antonios Antoniadis, and Gero Greiner. On multi-processor speed scaling with migration. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 279–288, 2011.

[9] Susanne Albers, Antonios Antoniadis, and Gero Greiner. On multi-processor speed scaling with migration. In *Proceedings of the twenty-third annual ACM symposium on Parallelism in algorithms and architectures*, pages 279–288, 2011.

[10] Susanne Albers and Hiroshi Fujiwara. Energy-efficient algorithms for flow time minimization. *ACM Transactions on Algorithms (TALG)*, 3(4), 2007.

[11] Susanne Albers, Fabian Müller, and Swen Schmelzer. Speed scaling on parallel processors. *Algorithmica*, 68(2):404–425, 2014.

[12] Hrishikesh Amur, Ripal Nathuji, Mrinmoy Ghosh, Karsten Schwan, and Hsien-Hsin S Lee. Idlepower: Application-aware management of processor idle states. In *Proceedings of the Workshop on Managed Many-Core Systems, MMCS*, volume 8, 2008.

[13] Eric Angel, Evripidis Bampis, Vincent Chau, and Nguyen Kim Thang. Throughput maximization in multiprocessor speed-scaling. *Theoretical Computer Science*, 630:1–12, 2016.

[14] Eric Angel, Evripidis Bampis, Fadi Kacem, and Dimitrios Letsios. Speed scaling on parallel processors with migration. In *European Conference on Parallel Processing*, pages 128–140. Springer, 2012.

[15] Antonios Antoniadis and Chien-Chung Huang. Non-preemptive speed scaling. *Journal of Scheduling*, 16(4):385–394, 2013.

[16] Martin Arlitt and Tai Jin. A workload characterization study of the 1998 world cup web site. *IEEE network*, 14(3):30–37, 2000.

[17] Hadi Asgharimoghaddam and Nam Sung Kim. Spinwise: A practical energy-efficient synchronization technique for cmps. *ACM SIGARCH Computer Architecture News*, 44(1):1–8, 2016.

[18] Theodore P Baker. Comparison of empirical success rates of global vs. partitioned fixed-priority and edf scheduling for hard real time. 2005.

[19] Theodore P. Baker. A comparison of global and partitioned edf schedulability tests for multiprocessors. Technical report, In International Conf. on Real-Time and Network Systems, 2005.

[20] Evripidis Bampis, Alexander Kononov, Dimitrios Letsios, Giorgio Lucarelli, and Maxim Sviridenko. Energy-efficient scheduling and routing via randomized rounding. *Journal of Scheduling*, 21(1):35–51, 2018.

[21] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM (JACM)*, 54(1):1–39, 2007.

[22] Thomas Barnett and Arielle Sumits. Cisco global cloud index 2015–2020. *Cisco Knowledge Network (CKN) Session*, 2016.

[23] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[24] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12):33–37, 2007.

[25] Sanjoy K Baruah. The non-preemptive scheduling of periodic tasks upon multiprocessors. *Real-Time Systems*, 32(1-2):9–20, 2006.

[26] Andrea Bastoni, Bjorn B Brandenburg, and James H Anderson. An empirical comparison of global, partitioned, and clustered multiprocessor edf schedulers. In *2010 31st IEEE Real-Time Systems Symposium*, pages 14–24. IEEE, 2010.

[27] Paul C Bell and Prudence WH Wong. Multiprocessor speed scaling for jobs with arbitrary sizes and deadlines. *Journal of Combinatorial Optimization*, 29(4):739–749, 2015.

[28] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. Improved schedulability analysis of edf on multiprocessor platforms. In *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, pages 209–218. IEEE, 2005.

[29] Stephen Boyd, Stephen P Boyd, and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[30] Carlo Brandolese, William Fornaciari, Fabio Salice, and Donatella Sciuto. Source-level execution time estimation of c programs. In *Proceedings of the ninth international symposium on Hardware/software codesign*, pages 98–103, 2001.

[31] Dominik Brodowski and Nico Golde. CPU frequency and voltage scaling code in the linux (tm) kernel. linux cpufreq. cpufreq governors, 2015.

[32] 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.

[33] Yang Cai and MC Kong. Nonpreemptive scheduling of periodic tasks in uni-and multiprocessor systems. *Algorithmica*, 15(6):572–599, 1996.

[34] J Calandrino and J Anderson. Quantum support for multiprocessor pfair scheduling in linux. In *Proc. of the 2nd Int'l Workshop on Operating System Platforms for Embedded Real-Time Applications*, 2006.

[35] John M Calandrino, James H Anderson, and Dan P Baumberger. A hybrid real-time scheduling approach for large-scale multicore platforms. In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pages 247–258. IEEE, 2007.

[36] Bo Chen, William Pak Tun Ma, Yan Tan, Alexandra Fedorova, and Greg Mori. Greenrt: a framework for the design of power-aware soft real-time applications.

[37] Jian-Jia Chen. Partitioned multiprocessor fixed-priority scheduling of sporadic real-time tasks. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 251–261. IEEE, 2016.

[38] Jian-Jia Chen, Heng-Ruey Hsu, Kai-Hsiang Chuang, Chia-Lin Yang, Ai-Chun Pang, and Tei-Wei Kuo. Multiprocessor energy-efficient scheduling with task migration considerations. In *Proceedings. 16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004.*, pages 101–108. IEEE, 2004.

[39] Tao Chen, Alexander Rucker, and G Edward Suh. Execution time prediction for energy-efficient hardware accelerators. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 457–469, 2015.

[40] Yanpei Chen, Sara Alspaugh, and Randy Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *arXiv preprint arXiv:1208.4174*, 2012.

[41] Hsiang-Yun Cheng, Jia Zhan, Jishen Zhao, Yuan Xie, Jack Sampson, and Mary Jane Irwin. Core vs. uncore: The heart of darkness. In *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2015.

[42] Vincent Cohen-Addad, Zhentao Li, Claire Mathieu, and Ioannis Milis. Energy-efficient algorithms for non-preemptive speed-scaling. In *International Workshop on Approximation and Online Algorithms*, pages 107–118. Springer, 2014.

[43] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. Pnuts: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008.

[44] Transaction Processing Performance Council. Tpc-energy specification, 2010.

[45] Transaction Processing Performance Council. Tpc benchmark e-standard specification-version 1.14.0. 2015.

[46] CPU-World. Amd fx-6300 specifications. https://www.cpu-world.com/CPUs/Bulldozer/AMD-FX-Series%20FX-6300.html. Accessed: Oct. 2020.

[47] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R Hanebutte, and Onur Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, pages 31–40, 2011.

[48] Robert I Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM computing surveys (CSUR)*, 43(4):1–44, 2011.

[49] Robert Ian Davis and Liliana Cucu-Grosjean. A survey of probabilistic schedulability analysis techniques for real-time systems. *LITES: Leibniz Transactions on Embedded Systems*, pages 1–53, 2019.

[50] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.

[51] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[52] Marios D Dikaiakos, Anne Rogers, and Kenneth Steiglitz. Fast: A functional algorithm simulation testbed. In *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 142–146. IEEE.

[53] Jennie Duggan, Ugur Cetintemel, Olga Papaemmanouil, and Eli Upfal. Performance prediction for concurrent database workloads. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 337–348, 2011.

[54] Kit Eaton. How one second could cost amazon $1.6 billion in sales. *Fast Company*, 14, 2012. Accessed: Oct. 2020.

[55] Jakob Engblom, Andreas Ermedahl, Mikael Sjödin, Jan Gustafsson, and Hans Hansson. Worst-case execution-time analysis for embedded real-time systems. *International Journal on Software Tools for Technology Transfer*, 4(4):437–455, 2003.

[56] Stijn Eyerman and Lieven Eeckhout. A counter architecture for online dvfs profitability estimation. *IEEE Transactions on Computers*, 59(11):1576–1583, 2010.

[57] Jason Flinn and Mahadev Satyanarayanan. *Energy-aware adaptation for mobile applications*, volume 33. ACM, 1999.

[58] Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

[59] Kenji Funaoka, Shinpei Kato, and Nobuyuki Yamasaki. Work-conserving optimal real-time scheduling on multiprocessors. In *2008 Euromicro Conference on Real-Time Systems*, pages 13–22. IEEE, 2008.

[60] Archana Ganapathi, Harumi Kuno, Umeshwar Dayal, Janet L Wiener, Armando Fox, Michael Jordan, and David Patterson. Predicting multiple metrics for queries: Better decisions enabled by machine learning. In *2009 IEEE 25th International Conference on Data Engineering*, pages 592–603. IEEE, 2009.

[61] Dennis Gannon, Roger Barga, and Neel Sundaresan. Cloud-native applications. *IEEE Cloud Computing*, 4(5):16–21, 2017.

[62] Marco E. T. Gerards, Johann L. Hurink, and Philip K. F. Hölzenspies. A survey of offline algorithms for energy minimization under deadline constraintsa. *Journal of Scheduling*, 19(1):3–19, 2016.

[63] Daniel Gmach, Jerry Rolia, Ludmila Cherkasova, and Alfons Kemper. Workload analysis and demand prediction of enterprise data center applications. In *2007 IEEE 10th International Symposium on Workload Characterization*, pages 171–180. IEEE, 2007.

[64] Giovani Gracioli, Antônio Augusto Fröhlich, Rodolfo Pellizzoni, and Sebastian Fischmeister. Implementation and evaluation of global and partitioned scheduling in a real-time os. *Real-Time Systems*, 49(6):669–714, 2013.

[65] Gero Greiner, Tim Nonner, and Alexander Souza. The bell is ringing in speed-scaled multiprocessor scheduling. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 11–18, 2009.

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

[67] Nan Guan, Wang Yi, Zonghua Gu, Qingxu Deng, and Ge Yu. New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In *2008 Real-Time Systems Symposium*, pages 137–146. IEEE, 2008.

[68] Shenoda Guirguis, Mohamed A Sharaf, Panos K Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. Adaptive scheduling of web transactions. In *2009 IEEE 25th International Conference on Data Engineering*, pages 357–368. IEEE, 2009.

[69] Shenoda Guirguis, Mohamed A Sharaf, Panos K Chrysanthis, Alexandros Labrinidis, and Kirk Pruhs. Adaptive scheduling of web transactions. In *2009 IEEE 25th International Conference on Data Engineering*, pages 357–368. IEEE, 2009.

[70] Chetan Gupta, Abhay Mehta, and Umeshwar Dayal. Pqr: Predicting query execution times for autonomous workload management. In *2008 International Conference on Autonomic Computing*, pages 13–22. IEEE, 2008.

[71] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the intel haswell processor. In *2015 IEEE international parallel and distributed processing symposium workshop*, pages 896–904. IEEE, 2015.

[72] Charles A Hall and W Weston Meyer. Optimal error bounds for cubic spline interpolation. *Journal of Approximation Theory*, 16(2):105–122, 1976.

[73] James Hamilton. The cost of latency. https://perspectives.mvdirona.com/2009/10/the-cost-of-latency/, 2009. Accessed: Oct. 2020.

[74] James Hamilton. Overall data center costs. https://perspectives.mvdirona.com/2010/09/overall-data-center-costs/, 2010. Accessed: Oct. 2020.

[75] Per Hammarlund, Alberto J Martinez, Atiq A Bajwa, David L Hill, Erik Hallnor, Hong Jiang, Martin Dixon, Michael Derr, Mikal Hunsaker, Rajesh Kumar, et al. Haswell: The fourth-generation intel core processor. *IEEE Micro*, 34(2):6–20, 2014.

[76] W. Hardle and W. Steiger. Algorithm as 296: Optimal median smoothing. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 44(2):258–264, 1995.

[77] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*, pages 409–439. 2018.

[78] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[79] Sebastian Herbert and Diana Marculescu. Analysis of dynamic voltage/frequency scaling in chip-multiprocessors. In *Proceedings of the 2007 international symposium on Low power electronics and design (ISLPED'07)*, pages 38–43. IEEE, 2007.

[80] Hewlett Packard Enterprise. Hpe global workload manager 7.6. https://support.hpe.com/hpsc/doc/public/display?sp4ts.oid=3725908, 2017. Accessed: Oct. 2017.

[81] Dominic Hillenbrand, Yuuki Furuyama, Akihiro Hayashi, Hiroki Mikami, Keiji Kimura, and Hironori Kasahara. Reconciling application power control and operating systems for optimal power and performance. In *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–8. IEEE, 2013.

[82] Jason M Hirst, Jonathan R Miller, Brent A Kaplan, and Derek D Reed. Watts up? pro ac power meter for automated energy recording: A product review. *Behavior Analysis in Practice*, 6(1):82, 2013.

[83] Dorit S Hochbaum and David B Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *Journal of the ACM (JACM)*, 34(1):144–162, 1987.

[84] Jiamin Huang, Barzan Mozafari, Grant Schoenebeck, and Thomas Wenisch. Identifying the major sources of variance in transaction latencies: Towards more predictable databases. *arXiv preprint arXiv:1602.01871*, 2016.

[85] Ling Huang, Jinzhu Jia, Bin Yu, Byung-Gon Chun, Petros Maniatis, and Mayur Naik. Predicting execution time of computer programs using sparse polynomial regression. In *Advances in neural information processing systems*, pages 883–891, 2010.

[86] IBM Corporation. Db2 workload management guide and reference. https://www.ibm.com/support/knowledgecenter/en/SSEPGG_10.1.0/com.ibm.db2.luw.admin.wlm.doc/com.ibm.db2.luw.admin.wlm.doc-gentopic1.html, 2013. Accessed: Dec. 2020.

[87] IBM Corporation. Db2 workload manager. https://www.ibm.com/support/knowledgecenter/en/SSEPGG_10.1.0/com.ibm.db2.luw.admin.wlm.doc/com.ibm.db2.luw.admin.wlm.doc-gentopic1.html, 2017. Accessed: Oct. 2017.

[88] Stratos Idreos, Lukas M Maas, and Mike S Kester. Evolutionary data systems. *arXiv preprint arXiv:1706.05714*, 2017.

[89] Thomas Ilsche, Marcus Hähnel, Robert Schöne, Mario Bielert, and Daniel Hackenberg. Powernightmares: The challenge of efficiently using sleep states on multi-core systems. In *European Conference on Parallel Processing*, pages 623–635. Springer, 2017.

[90] Intel. Intel® xeon® processor e5-1600/e5-2600/e5-4600 v2 product families, datasheet - volume one of two. Technical report, 2014. Accessed: Oct. 2020.

[91] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, September 2016.

[92] Yannis E Ioannidis. Query optimization. *ACM Computing Surveys (CSUR)*, 28(1):121–123, 1996.

[93] Canturk Isci, Alper Buyuktosunoglu, Chen-Yong Cher, Pradip Bose, and Margaret Martonosi. An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 347–358. IEEE, 2006.

[94] Tohru Ishihara and Hiroto Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Proceedings of the 1998 international symposium on Low power electronics and design*, pages 197–202, 1998.

[95] 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.

[96] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan PC Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[97] Svilen Kanev, Kim Hazelwood, Gu-Yeon Wei, and David Brooks. Tradeoffs between power management and tail latency in warehouse-scale applications. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 31–40. IEEE, 2014.

[98] Alexey Karyakin and Kenneth Salem. Dimmstore: memory power optimization for database systems. *Proceedings of the VLDB Endowment*, 12(11):1499–1512, 2019.

[99] 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.

[100] Georgios Keramidas, Vasileios Spiliopoulos, and Stefanos Kaxiras. Interval-based models for run-time dvfs orchestration in superscalar processors. In *Proceedings of the 7th ACM international conference on Computing frontiers*, pages 287–296, 2010.

[101] In Kee Kim, Jacob Steele, Yanjun Qi, and Marty Humphrey. Comprehensive elastic resource management to ensure predictable performance for scientific applications on public iaas clouds. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, pages 355–362. IEEE, 2014.

[102] Thomas Kissinger, Dirk Habich, and Wolfgang Lehner. Adaptive energy-control for in-memory database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 351–364, 2018.

[103] Roger Koenker and Gilbert Bassett Jr. Regression quantiles. *Econometrica: journal of the Econometric Society*, pages 33–50, 1978.

[104] Roger Koenker and Kevin F Hallock. Quantile regression. *Journal of economic perspectives*, 15(4):143–156, 2001.

[105] Mustafa Korkmaz, Martin Karsten, Kenneth Salem, and Semih Salihoglu. Workload-aware CPU performance scaling for transactional database systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 291–306, 2018.

[106] 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.

[107] Medhat Korna. Estimating percentiles of skewed data. Technical report, DAYTON UNIV OH RESEARCH INST, 1981.

[108] David Koufaty and Deborah T Marr. Hyperthreading technology in the netburst microarchitecture. *IEEE Micro*, 23(2):56–65, 2003.

[109] Pawan Kumar and Rakesh Kumar. Issues and challenges of load balancing techniques in cloud computing: A survey. *ACM Computing Surveys (CSUR)*, 51(6):1–35, 2019.

[110] Woo-Cheol Kwon and Taewhan Kim. Optimal voltage allocation techniques for dynamically variable voltage processors. *ACM Transactions on Embedded Computing Systems (TECS)*, 4(1):211–230, 2005.

[111] Willis Lang, Ramakrishnan Kandhan, and Jignesh M Patel. Rethinking query processing for energy efficiency: Slowing down to win the race. *IEEE Data Eng. Bull.*, 34(1):12–23, 2011.

[112] Willis Lang and Jignesh M Patel. Energy management for mapreduce clusters. *Proceedings of the VLDB Endowment*, 3(1-2):129–139, 2010.

[113] Gregor Von Laszewski, Lizhe Wang, Andrew J. Younge, and Xi He. Power-aware scheduling of virtual machines in dvfs-enabled clusters, in. In *Proc. IEEE Int'l Conf. Cluster Computing*, pages 1–10, 2009.

[114] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8, 2010.

[115] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, pages 442–450. IEEE, 2018.

[116] Jungseob Lee and Nam Sung Kim. Optimizing throughput of power-and thermal-constrained multicore processors using dvfs and per-core power-gating. In *2009 46th ACM/IEEE Design Automation Conference*, pages 47–50. IEEE, 2009.

[117] Scott T Leutenegger and Daniel Dias. A modeling study of the tpc-c benchmark. *ACM Sigmod Record*, 22(2):22–31, 1993.

[118] Jacob Leverich and Christos Kozyrakis. On the energy (in)efficiency of hadoop clusters. *SIGOPS Oper. Syst. Rev.*, 44(1):61–65, 2010.

[119] Jacob Leverich and Christos Kozyrakis. Reconciling high server utilization and sub-millisecond quality-of-service. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[120] Jacob Leverich, Matteo Monchiero, Vanish Talwar, Parthasarathy Ranganathan, and Christos Kozyrakis. Power management of datacenter workloads using per-core power gating. *IEEE Computer Architecture Letters*, 8(2):48–51, 2009.

[121] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[122] Zhen Liu and Don Towsley. Optimality of the round-robin routing policy. *Journal of applied probability*, pages 466–475, 1994.

[123] 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.

[124] David Lo, Liqun Cheng, Rama Govindaraju, Luiz André Barroso, and Christos Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 301–312. IEEE Press.

[125] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[126] José María López, José Luis Díaz, and Daniel F García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004.

[127] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*, pages 631–645, 2018.

[128] Eric Masanet, Arman Shehabi, Nuoa Lei, Sarah Smith, and Jonathan Koomey. Recalibrating global data center energy-use estimates. *Science*, 367(6481):984–986, 2020.

[129] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: Eliminating server idle power. *SIGARCH Comput. Archit. News*, 37(1):205–216, 2009.

[130] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. Power management of online data-intensive services. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 319–330. IEEE, 2011.

[131] Microsoft Corporation. Microsoft sql server resource governor. https://docs.microsoft.com/en-us/sql/relational-databases/resource-governor/resource-governor, 2017. Accessed: Oct. 2017.

[132] Asit K Mishra, Joseph L Hellerstein, Walfredo Cirne, and Chita R Das. Towards characterizing cloud backend workloads: insights from google compute clusters. *ACM SIGMETRICS Performance Evaluation Review*, 37(4):34–41, 2010.

[133] Bruce Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[134] Barzan Mozafari, Carlo Curino, Alekh Jindal, and Samuel Madden. Performance and resource modeling in highly-concurrent oltp workloads. In *Proceedings of the 2013 acm sigmod international conference on management of data*, pages 301–312, 2013.

[135] Barzan Mozafari, Carlo Curino, and Samuel Madden. Dbseer: Resource and performance prediction for building a next generation database cloud. In *CIDR*, 2013.

[136] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, pages 385–398, 2013.

[137] Graham R Nudd, Darren J Kerbyson, Efstathios Papaefstathiou, Stewart C Perry, John Stuart Harper, and Daniel V Wilcox. Pace—a toolset for the performance prediction of parallel and distributed systems. *The International Journal of High Performance Computing Applications*, 14(3):228–251, 2000.

[138] Oracle Corporation. Managing resources with oracle database resource manager. https://docs.oracle.com/database/121/ADMIN/dbrm.htm, 2017. Accessed: Oct. 2017.

[139] Pekka Pääkkönen and Daniel Pakkala. Reference architecture and classification of technologies, products and services for big data systems. *Big data research*, 2(4):166–186, 2015.

[140] Steven Pelley, David Meisner, Thomas F Wenisch, and James W VanGilder. Understanding and abstracting total data center power. In *Workshop on Energy-Efficient Design*, volume 11, pages 1–6, 2009.

[141] Pivotal Corporation. Greenplum workload manager. https://gpcc.docs.pivotal.io/220/gp-wlm/topics/gpwlm-docs.html, 2017. Accessed: Oct. 2017.

[142] Danica Porobic, Ippokratis Pandis, Miguel Branco, Pınar Tözün, and Anastasia Ailamaki. Oltp on hardware islands. *arXiv preprint arXiv:1208.0227*, 2012.

[143] 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.

[144] Waleed Reda, Marco Canini, Lalith Suresh, Dejan Kostić, and Sean Braithwaite. Rein: Taming tail latency in key-value stores via multiget scheduling. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 95–110, 2017.

[145] John Roach. Microsoft finds underwater datacenters are reliable, practical and use energy sustainably, 2020.

[146] Barry Rountree, David K Lowenthal, Martin Schulz, and Bronis R De Supinski. Practical performance prediction under dynamic voltage frequency scaling. In *2011 International Green Computing Conference and Workshops*, pages 1–8. IEEE, 2011.

[147] Barry Rountree, David K Lownenthal, Bronis R De Supinski, Martin Schulz, Vincent W Freeh, and Tyler Bletsch. Adagio: making dvs practical for complex hpc applications. In *Proceedings of the 23rd international conference on Supercomputing*, pages 460–469. ACM, 2009.

[148] Robert Schöne, Daniel Molka, and Michael Werner. Wake-up latencies for processor idle states on current x86 processors. *Computer Science-Research and Development*, 30(2):219–227, 2015.

[149] Eric Schurman and Jake Brutlag. The user and business impact of server delays, additional bytes, and http chunking in web search. In *Velocity Web Performance and Operations Conference*. oreilly, 2009.

[150] Rathijit Sen and David A. Wood. Pareto governors for energy-optimal computing. *ACM Trans. Archit. Code Optim.*, 14(1):6:1–6:25, 2017.

[151] Ilya Sharapov, Robert Kroeger, Guy Delamarter, Razvan Cheveresan, and Matthew Ramsay. A case study in top-down performance estimation for a large-scale parallel application. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 81–89, 2006.

[152] Arman Shehabi, Sarah Smith, Dale Sartor, Richard Brown, Magnus Herrlin, Jonathan Koomey, Eric Masanet, Nathaniel Horner, Inês Azevedo, and William Lintner. United states data center energy usage report. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2016.

[153] Rekha Singhal and Manoj Nambiar. Predicting sql query execution time for large data volume. In *Proceedings of the 20th International Database Engineering & Applications Symposium*, pages 378–385, 2016.

[154] Sukhdeep Sodhi, Jaspal Subhlok, and Qiang Xu. Performance prediction with skeletons. *Cluster Computing*, 11(2):151–165, 2008.

[155] Leonid B Sokolinsky. Survey of architectures of parallel database systems. *Programming and Computer Software*, 30(6):337–346, 2004.

[156] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. Green governors: A framework for continuously adaptive dvfs. In *Green Computing Conference and Workshops (IGCC), 2011 International*, pages 1–8. IEEE, 2011.

[157] Vaidyanathan Srinivasan, Gautham R Shenoy, Srivatsa Vaddagiri, Dipankar Sarma, and Venkatesh Pallipadi. Energy-aware task and interrupt management in linux. In *Ottawa Linux Symposium*, 2008.

[158] Standard Performance Evaluation Corporation(SPEC). Power and Performance Benchmark Methodology V2.1. https://www.spec.org/power/docs/SPEC-Power_and_Performance_Methodology.pdf, November 2012. Accessed: Feb. 2017.

[159] Michael Stonebraker and Ariel Weisberg. The voltdb main memory dbms. *IEEE Data Eng. Bull.*, 36(2):21–27, 2013.

[160] Bo Su, Joseph L. Greathouse, Junli Gu, Michael Boyer, Li Shen, and Zhiying Wang. Implementing a leading loads performance predictor on commodity processors. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, Philadelphia, PA, June 2014. USENIX Association.

[161] Bo Su, Junli Gu, Li Shen, Wei Huang, Joseph L Greathouse, and Zhiying Wang. Ppep: Online performance, power, and energy prediction framework and dvfs space exploration. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 445–457. IEEE, 2014.

[162] Teradata Corporation. Teradata workload manager. http://www.teradata.com/products-and-services/workload-management, 2017. Accessed: Oct. 2017.

[163] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 231–242, 2010.

[164] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[165] EFI Unified. Advanced configuration and power interface specification. https://uefi.org/sites/default/files/resources/ACPI_6_3_final_Jan30.pdf, 2016. Accessed: Oct. 2020.

[166] EFI Unified. Advanced configuration and power interface specification, 2016.

[167] 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.

[168] Ashish Vulimiri, Philip Brighten Godfrey, Radhika Mittal, Justine Sherry, Sylvia Ratnasamy, and Scott Shenker. Low latency via redundancy. In *Proceedings of the ninth ACM conference on Emerging networking experiments and technologies*, pages 283–294, 2013.

[169] Jons-Tobias Wamhoff, Stephan Diestelhorst, Christof Fetzer, Patrick Marlier, Pascal Felber, and Dave Dice. The {TURBO} diaries: Application-controlled frequency scaling explained. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 193–204, 2014.

[170] Ted J Wasserman, Patrick Martin, David B Skillicorn, and Haider Rizvi. Developing a characterization of business intelligence workloads for sizing new database systems. In *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP*, pages 7–13, 2004.

[171] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced CPU energy. In *Mobile Computing*, pages 449–471. Springer, 1994.

[172] Neil H. E. Weste and David Money Harris. *CMOS VLSI design: a circuits and systems perspective*. Pearson India, 2015.

[173] Michael Widenius, David Axmark, and Kaj Arno. *MySQL reference manual: documentation from the source*. " O'Reilly Media, Inc.", 2002.

[174] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[175] Qiang Wu, Qingyuan Deng, Lakshmi Ganesh, Chang-Hong Hsu, Yun Jin, Sanjeev Kumar, Bin Li, Justin Meza, and Yee Jiun Song. Dynamo: facebook's data center-wide power management system. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 469–480. IEEE, 2016.

[176] Wentao Wu, Yun Chi, Hakan Hacígümüş, and Jeffrey F Naughton. Towards predicting query execution time for concurrent and dynamic database workloads. *Proceedings of the VLDB Endowment*, 6(10):925–936, 2013.

[177] Wentao Wu, Yun Chi, Shenghuo Zhu, Junichi Tatemura, Hakan Hacigümüs, and Jeffrey F Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 1081–1092. IEEE, 2013.

[178] Wentao Wu, Xi Wu, Hakan Hacıgümüş, and Jeffrey F Naughton. Uncertainty aware query execution time prediction. *arXiv preprint arXiv:1408.6589*, 2014.

[179] Jing Xu and Jose A. B. Fortes. Multi-objective virtual machine placement in virtualized data center environments. In *Proceedings of the 2010 IEEE/ACM Int'L Conference on Green Computing and Communications & Int'L Conference on Cyber, Physical and Social Computing*, pages 179–188, 2010.

[180] Z. Xu, Y. C. Tu, and X. Wang. Exploring power-performance tradeoffs in database systems. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*, pages 485–496, 2010.

[181] 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.

[182] Frances Yao, Alan Demers, and Scott Shenker. A scheduling model for reduced CPU energy. In *Proceedings of IEEE 36th annual foundations of computer science*, pages 374–382. IEEE, 1995.

[183] Kai-Hau Yeung, Kam-Wa Suen, and Kin-Yeung Wong. Least load dispatching algorithm for parallel web server nodes. *IEE Proceedings-Communications*, 149(4):223–226, 2002.

[184] Sergey Zhuravlev, Juan Carlos Saez, Sergey Blagodurov, Alexandra Fedorova, and Manuel Prieto. Survey of energy-cognizant scheduling techniques. *IEEE Transactions on Parallel and Distributed Systems*, 24(7):1447–1464, 2012.

APPENDICES

Appendix A

In this appendix, we present the proof of Theorem 3.4.4. As noted in Section 3.4.5, this theorem shows the relation between the power consumption of POLARIS and YDS [182], in case two algorithms run over problem instances where each instance has the same set of requests with arbitrary arrival time and deadline. However, the request sizes in YDS' problem instance are increased by a factor of a particular value based on the ratio between the largest and smallest request size.

We assume w.l.o.g., that P and therefore P' are contiguous. In other words, for each time $t \in [0, d(t_n) = d(t'_n)]$ there is a transaction t_j, such that $a(t_j) \leq t \leq d(t_j)$. If the P and P' are not contiguous, we can break it into a finite number of contiguous parts and analyze POLARIS competitiveness in each part and get the same result. We let $s_P(t)$ and $s_Y(t)$ be the speed of POLARIS's and YDS's processors at time t when executing P and P', respectively. There are three types of events that will happen at any point of time. Either a new transaction arrives, POLARIS or YDS completes a transaction, or an infinitesimal dt amount of time elapses. We use the same potential function $\phi(t)$ as in reference (defined momentarily). We will show that:

(1) $\phi(t)$ is 0 at time t and at the end of the final transaction.

(2) $\phi(t)$ does not increase as a result of a task arrival or a completion of a task by POLARIS or YDS.

(3) At any time t between arrival events the following inequality holds:

$$s_P(t)^\alpha + \frac{d\phi(t)}{dt} \leq \alpha^\alpha s_Y^\alpha \tag{A.1}$$

Note that if these conditions hold, integrating equation A.1 between each arrival events and summing gives:

$$Pow[POLARIS(P)] \leq \alpha^{\alpha} Pow[YDS(P')].$$

.

We next define $\phi(t)$ and prove that all three conditions hold. Let $s_{Pna}(t)$ (for **P**OLARIS **n**o **a**rrival) denote the speed at which POLARIS would be executing if no new tasks were to arrive after the current time. By Lemma 3.4.1 we proved that when no tasks arrive POLARIS' behavior is identical to OA, which simply executes YDS on the transactions on its queue. Note POLARIS may have modified its queue to be T or T' in the latest arrival event prior to current time but after it finalizes its queue, it simply executes YDS on the transactions on its queue (recall Lemma 3.4.1). Throughout the proof we denote the current time always as t_0. Let $CI_1, ..., CI_k$ be POLARIS's current critical intervals (note that k will change over time) and let t_i be the end of critical interval CI_i. Let $w_P(t, t')$ and $w_Y(t, t')$ be the unfinished work that POLARIS and YDS have on their queue at t_0 with deadlines in interval $(t, t']$. Therefore, assuming that no new tasks arrive, at time t, where $t_i < t \leq t_{i+1}$, POLARIS has a planned speed $s_{Pna}(t) = den(CI_i) = \frac{w_P(t_i, t_{i+1})}{t_{i+1} - t_i}$. In particular note that $s_{Pna}(t_i)$ is the planned speed of POLARIS at time t_i when critical interval CI_i begins and the processor speed remains the same until CI_{i+1} begins.

We next make a simple observation about $s_{Pna}(t)$. Since POLARIS runs YDS on the transactions of its queue by considering their arrival times as the current time, the density of each critical interval is a non-increasing sequence. That is, when no new transactions arrive, POLARIS has a planned processor speed that decreases (or stays the same) over time, i.e. $s_{Pna}(t_i) \geq s_{Pna}(t_{i+1})$ for all i. We refer the reader to reference [21] for a formal proof of this observation (proved for OA).

The potential function we use is the following:

$$\phi(t) = \alpha \sum_{i \geq 0} s_{Pna}(t_i)^{\alpha - 1} (w_P(t_i, t_{i+1}) - \alpha w_Y(t_i, t_{i+1}))$$

We next show that claims (1), (3), and (2) are true, in that order.

Proof of claim (1): First observe that at time 0 and after the final transaction ends (call t_{max}), both algorithms have empty queues so all w_P and w_Y values are 0 so $\phi(0)$ and $\phi(t_{max})$ are 0, so claim (1) holds.

Proof of claim (3): This part of the analysis is identical to the analysis presented by Bansal et al [21] for OA.

We need to show that when no transactions arrive in the next dt time equation A.1 holds. Notice that when no transactions arrive in the next dt time, $s_{Pna}(t_i)$ remains fixed for each i and YDS executes at the constant speed of $s_Y(t_0)$. Therefore:

$$s_{Pna}(t_0)^\alpha - \alpha^\alpha s_Y(t_0)^\alpha + \frac{d}{d_t}(\phi(t)) \leq 0 \tag{A.2}$$

Let's first analyze how $\frac{d\phi(t)}{dt}$ changes in the next dt time. Notice that POLARIS will be working at one of the transactions in interval $(t_0, t_1]$ at speed $s_{Pna}(t_0)$, so $w_P(t_0, t_1)$ will decrease at rate s_{Pna} and other $w_P(t_i, t_{i+1})$ remain unchanged. YDS will be running one transaction t_{YDS} at speed $s_Y(t_0)$. W.l.o.g., let t_{YDS} be in interval $(t_k, t_{k+1}]$. So $w_Y(t_k, t_{k+1})$ will decrease at rate $s_Y(t_0)$ and all other $w_Y(t_i, t_{i+1})$ will remain the same. Therefore $\frac{d\phi(t)}{dt}$ is decreasing at a rate:

$$\begin{aligned}\frac{d\phi(t)}{dt} &= \alpha(s_{Pna}(t_0)^{\alpha-1}(-s_{Pna}(t_0)) - \alpha s_{Pna}(t_k)^{\alpha-1}(-s_Y(t_0))) \\ &= -\alpha s_{Pna}(t_0)^\alpha + \alpha^2 s_{Pna}(t_k)^{\alpha-1}(s_Y(t_0))\end{aligned}$$

Substituting this into equation A.2 and recalling the observation we made above that $s_{Pna}(t_i)$ are a decreasing sequence, gives us:

$$(1-\alpha)s_{Pna}(t_0)^\alpha + \alpha^2 s_{Pna}(t_0)^{\alpha-1} s_Y(t_0) - \alpha^\alpha \leq 0$$

Let $z = \frac{s_{Pna}(t_0)}{s_Y(t_0)}$. Note we assumed w.l.o.g. that P and P' are contiguous so both POLARIS and YDS will always be working on a transaction, so $z \geq 0$. Substituting z into the above equation gives us:

$$f(z) = (1-\alpha)z^\alpha + \alpha^2 z^{\alpha-1} - \alpha^\alpha \leq 0$$

By looking at the value $f(0)$, $f(\infty)$ and the derivative of f, one can show that $f(z)$ is indeed less than or equal to 0 for all $z \geq 0$. completing the proof. We refer the reader to reference [21] for the full derivation.

Proof of claim (2): We analyze the changes to $\phi(t)$, $s_P(t)$ and $s_Y(t)$ under two possible events:

Completion of a transaction by YDS and POLARIS: This part of the analysis is the same as the proof in reference [21]. Notice that the completion of a transaction by YDS has no effect on the $s_{Pna}(t_i)$, $w_P(t_i, t_{i+1})$, and $w_Y(t_i, t_{i+1})$ for all i, so does not increase $\phi(t)$. Similarly the completion of a transaction by POLARIS has no effect on $s_{Pna}(t_i)$, $w_P(t_i, t_{i+1})$, and $w_Y(t_i, t_{i+1})$, it merely shifts in the index in the summation of $\phi(t)$ by 1. This proves partially that claim (2) holds.

Arrival of a new transaction: Suppose a new transaction t_{new} arrives to POLARIS and t'_{new} arrives to YDS's queue. Recall that $cw(t_{new}) = w(t'_new)$. Suppose $t_i < d(t_{new}) \leq t_{i+1}$. Here our proof differs from the proof in reference [21] in two ways. First we need to consider two cases depending on whether t_{new} is the earliest deadline transaction or not. If t_{new} has the earliest deadline then, POLARIS' adds two transactions to its queue and removes one from its queue. This behavior does not occur in OA so does not need to be argued when comparing OA to YDS in reference [21]. Second transactions added to POLARIS's queue and YDS's queue are different. The proof in reference [21] needs to consider only arrival of same transactions.

We note that the case when t_{new} does not have the earliest deadline is similar to the argument in reference [21]. Below we slightly simplify the proof in reference [21].

t_{new} **does not have the earliest deadline:** Note that t_{new} may change POLARIS's critical intervals but we think of the changes to the critical intervals a sequence of smaller changes. Specifically, we view the arrival of t_{new} and t'_{new} initially as arrivals of new transactions $t_{new'}$ and $t'_{new'}$ with deadlines $d(t_{new})$ and workload of 0. We then increase $t_{new'}$'s and $t'_{new'}$'s workloads in steps by some amount $x \leq w(t_{new})$, where the increase of $t_{new'}$'s workload by x increases the density of one of POLARIS's critical interval CI_j from $\frac{w_P(t_j,t_{j+1})}{(t_{j+1}-t_j)}$ to $\frac{w_P(t_j,t_{j+1}+x)}{(t_{j+1}-t_j)}$ but does not change the structure of the critical intervals[1]. In addition, after we increase t'_{new}'s workload by x, optionally, one of two possible events occurs:

1. Interval CI_j splits into two critical intervals with the same increased density of CI_j..
2. Interval CI_j merges with one or more critical intervals with the same increased density of CI_j.

In each step we find the minimum amount of x that will result in this behavior, and recurse on the remaining workload of t_{new}. We argue that in each recursive step the potential function does not increase. Once $t_{new'}$'s workload becomes equal to $w(t_{new})$, we have a final step where we add a workload of $w(t'_{new}) - w(t_{new})$ to $t'_{new'}$ and again argue that this does not increase the potential function.

Recursive step: This analysis is the same as the recursive step from reference [21]. We start by noting that after the increase in the density of CI_j, the splitting or merging of critical intervals have no effect on $\phi(t)$ because it just increases or decreases the number

[1]Note that YDS's critical intervals are irrelevant for our analysis because $\phi(t)$ is defined in terms of POLARIS's critical intervals.

of indices in the summation but does not change the value of $\phi(t)$. So we only analyze increasing the density of CI_j by amount of x. In this case, $s_{Pna}(t_j)$ (or the density of CI_j) increases from $\frac{w_P(t_j,t_{j+1})}{(t_{j+1}-t_j)}$ to $\frac{(w_P(t_j,t_{j+1})+x))}{(t_{j+1}-t_j)}$. Thus the potential function changes as follows:

$$\alpha(\frac{(w_P(t_j,t_{j+1})+x)}{(t_{j+1}-t_j)})^{\alpha-1}((w_P(t_j,t_{j+1})+x)-\alpha(w_Y(t_j,t_{j+1})+x))-$$

$$\alpha(\frac{(w_P(t_j,t_{j+1}))}{(t_{j+1}-t_j)})^{\alpha-1}(w_P(t_j,t_{j+1})-\alpha(w_Y(t_j,t_{j+1})))$$

Let $q = w_P(t_j, t_{j+1})$, $\delta = x$ and $r = w_Y(t_j, t_{j+1})$ and rearranging the terms we get:

$$\frac{\alpha((q+\delta)^{\alpha-1}(q-\alpha r-(\alpha-1)\delta)-q^{\alpha-1}(q-\alpha r))}{(t_{j+1}-t_j)^{\alpha-1}}$$

which is nonpositive by Lemma 3.3 in reference [21] when $q, r, \delta \geq 0$ and $\alpha \geq 1$.

Final step: Note that at the end of the recursive step, we added only $w(t_{new})$ workload to $t'_{new'}$, so there is still a workload of $w(t'_{new}) - w(t_{new})$ to be added to $t'_{new'}$ to replicate the addition of t'_{new}. Note however that this can only decrease the potential function because increasing the weight of $t'_{new'}$ has no effect on the final s_{Pna} and $w_P(t_j, t_{j+1})$ values and will only increase the $w_Y(t_j, t_{j+1})$ value for the final critical interval CI_j (after the recursive steps) that $t_{new'}$ now falls into.

t_{new} **has the earliest deadline:** In this case POLARIS changes its queue by adding t_{new}, t'_{cur}, and removing t_{cur}. YDS changes its queue by only adding t'_{new}. Note that t_{new} and t'_{cur} can be seen as one transaction because they have the same deadline and their total weight is less than $w(t'_{new})$. That is because:

$$w(t_{new}) + w(t'_{cur}) \leq w(t_{new}) + w_{max} \leq w(t_{new}) + \frac{w_{max}w(t_{new})}{w_{min}}$$

$$\leq (1 + \frac{w_{max}}{w_{min}})w(t_{new}) = cw(t_{new}) = w(t'_{new})$$

Therefore by the same analysis we gave above we can argue that the addition of t_{new} and t'_{cur} to POLARIS's queue and t'_{new}'s to YDS's queue does not increase $\phi(t)$. We next need to argue that the removal of t_{cur} from POLARIS's queue also does not increase $\phi(t)$. The argument is similar to the argument we made when breaking the addition of t_{new} and t'_{new} in recursive steps. We can view the removal of a transaction in recursive steps in which we decrease the workload of t_{cur} by some amount of x that decreases the density of some critical interval CI_j by x. Optionally, after this decrease, CI_j can split into two critical

intervals with the same decreased density of CI_j or merge with one or more critical intervals with this same density. Note that the merging or splitting has no effect on the value of $\phi(t)$ because it just increases or decreases the number of indices in the summation but does not change the value of $\phi(t)$. These operations only change the indices in the summation of $\phi(t)$. Note also that decreasing the density of CI_j cannot increase $\phi(t)$ because it can only decrease $s_{Pna}(t_j)$, decrease $w_P(t_j, t_{j+1})$ and does not change the other $w_P(t_i, t_{i+1})$'s. Similarly it does not change any of $w_Y(t_i, t_{i+1})$ because we are not altering YDS's queue, completing the proof.

Appendix B

This Appendix provides details about the PerformanceBaseline and EnergyBaseline used in the PLASM experiments presented in Section 4.6.

B.1 PerformanceBaseline

The objective of the PerformanceBaseline is to ensure that transaction deadlines are met, regardless of power consumption. All CPU cores are set to run at peak frequency, and global routing is used for request distribution. The centralized global queue prioritizes the waiting requests according to EDF.

By using peak speed, the PerformanceBaseline avoids failures that result from running transactions too slowly, e.g., because of misprediction of the transaction execution time in a scheduling algorithm like POLARIS. By using a centralized global request queue, the PerformanceBaseline also avoids failures that can result in non-preemptive systems from short-after-long request arrival patterns. That is, if a small request arrives in an empty wait queue right after the worker starts running a large request, the small request may fail no matter how fast the CPU core is running because a non-preemptive must complete the long request before starting the short one. Such a short-after-long pattern is represented in Figure B.1. In Figure B.1 (a), the CPU runs at peak frequency(dashed line). When the small request arrives, as shown in Figure B.1 (b), even though the processor keeps running at peak speed(dashed line), it is not possible to complete both requests within their deadlines.

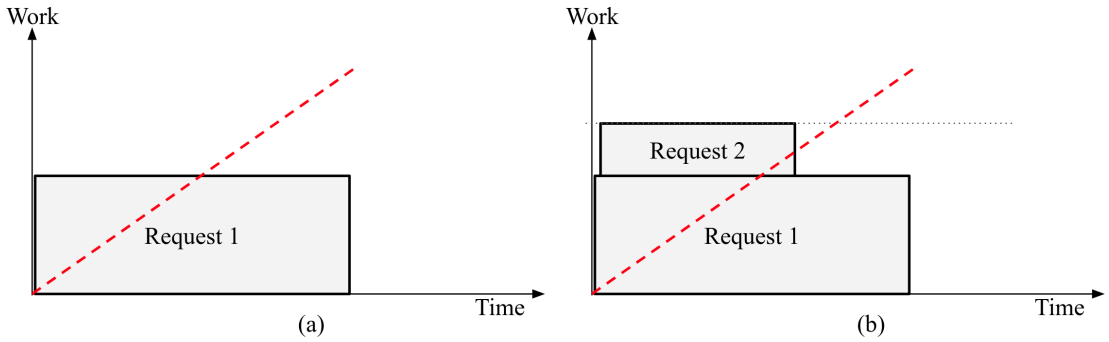


Figure B.1: Short request after running a long request. We use the representation in Figure 3.6. (a) shows request 1 arrives to an idle worker and the worker immediately executes it. POLARIS sets a speed level lower than the peak speed.(b) shows that, right after the execution of request 1 starts, a smaller request(request 2) arrives to the system. Because of the non-preemptive environment, Request 2 has to wait until Request 1 is completed. Therefore POLARIS increases to the peak speed, which is not sufficient to finish Request 2 within its deadline.

B.2 EnergyBaseline

The objective of the EnergyBaseline is to accommodate the offered transaction load using a little power as possible, without regard for transaction deadlines. Like the PerformanceBaseline, the EnergyBaseline uses a global request queue and the waiting requests are prioritized according to EDF. However, instead of setting the cores to run at their maximum frequency, the EnergyBaseline sets them to run at the lowest frequency that will accommodate the workload. The question we need to answer here is how to determine that frequency.

Figure B.2 shows the failure rates for scheduler with a global EDF-ordered request queue and all cores set to run a fixed frequency, as a function of offered load (TPC-C transaction request rate). The figure shows results for 5 different core frequency settings, covering the full range of frequencies available on our test system's processors. As shown in the figure, for each frequency, our test system saturates at a particular load level, and its failure rate gets close to 100%. For example, at 1.6 GHz, the system can sustain an offered load level up to 15000 TPS, whereas 2.4 GHz can handle a load level up to 21000 TPS.

Figure B.3 shows power consumption for the same experiments reported in Figure B.2.

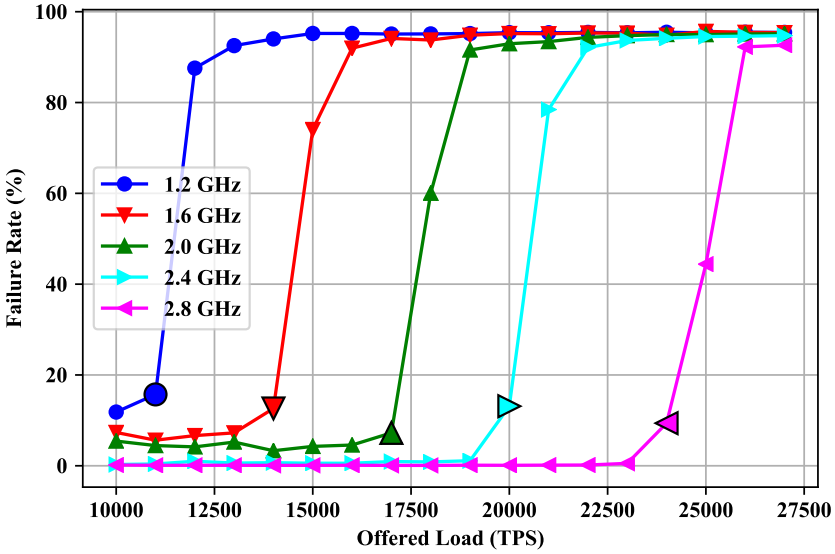


Figure B.2: Failure rate of EnergyBaseline

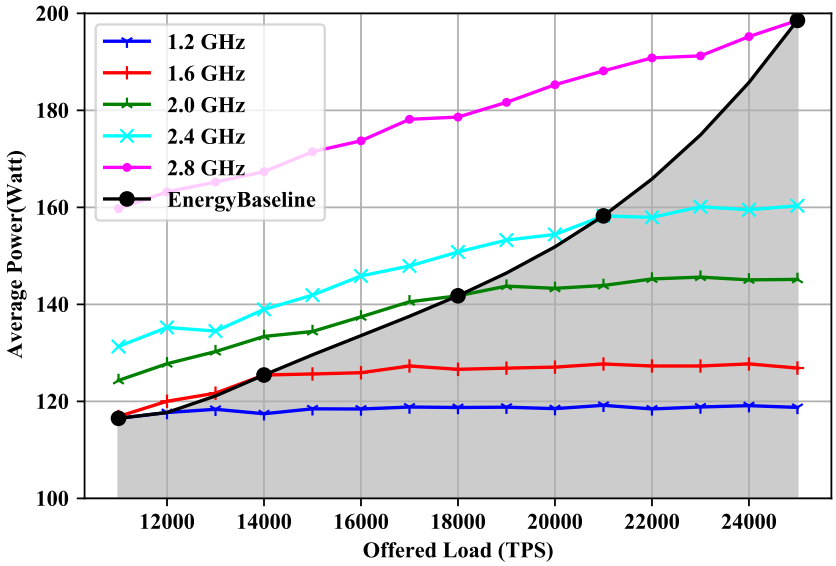


Figure B.3: EnergyBaseline power frontlines

At each frequency setting, power consumption increases with offered load until the system reaches saturation, and which point it levels off since system throughput stops increasing.

In Figure B.3, we marked the saturation point for each frequency using a black dot. For example, for 1.6GHz, the system saturates at about 14000 TPS, while consuming about 125 Watts. This represents the minimum possible power consumption for an offered load of 14000 TPS, since any decrease in frequency would leave the system unable to accommodate the offered load. The black line connecting these dots represents an estimate of the *power frontier* for this workload, representing the minimum power consumption at any offered load level. The power consumption that we report for the EnergyBaseline in Section 4.6 is this frontier power, for whatever load level is being used in a given experiment. We use cubic spline [72] interpolation to determine the power frontier between the measured points (black dots), which is suitable for the relation between frequency and power consumption, $P \propto f^{\alpha}$, where α is typically in the range $1 < \alpha \leq 3$ for server-grade processors.

On processors with a fixed set of available P-States, it should be possible to approximate the power frontier at any load level by switching processor frequency between two P-States. For example, to approximate minimum power while accommodating an offered load of 16000 TPS, we can switch the processor between 1.6 GHz and 2.0 GHz. The power-optimal way of doing such switching is to switch between at most two consecutive frequency levels [110, 94]. We did not implement this switching for our EnergyBaseline. Instead, we report spline-interpolated power values for load levels in between the measured points on the power frontier.