Workload-Aware CPU Performance Scaling for Transactional Database Systems

Mustafa Korkmaz, Martin Karsten, Kenneth Salem, Semih Salihoglu
University of Waterloo
{mkorkmaz,mkarsten,kmsalem,semih.salihoglu}@uwaterloo.ca

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. In this paper, we show that transactional database systems can manage DVFS more effectively than the underlying operating system. This is because the database system has more information about the workload, and more control over that workload, than is available to the operating system. We present a technique called POLARIS for reducing the power consumption of transactional database systems. POLARIS directly manages processor DVFS and controls database transaction scheduling. Its goal is to minimize power consumption while ensuring the transactions are completed within a specified latency target. POLARIS is workload-aware, and can accommodate concurrent workloads with different characteristics and latency budgets. We show that POLARIS can simultaneously reduce power consumption and reduce missed latency targets, relative to operating-system-based DVFS governors.

ACM Reference Format:
Mustafa Korkmaz, Martin Karsten, Kenneth Salem, Semih Salihoglu. 2018. Workload-Aware CPU Performance Scaling for Transactional Database Systems. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3183713.3196901

1 INTRODUCTION

Servers do not always run at maximum capacity, because workloads fluctuate and can be bursty [13, 15, 20]. Thus, when load is not at its peak, server resources are underutilized [11, 29]. Workload fluctuations occur on many time scales. They may exhibit regular seasonal, weekly, or diurnal patterns. Workloads can also fluctuate on timescales of hours or minutes, perhaps due to external events, news cycles, social effects, and other factors. A variety of techniques have been proposed for improving the power proportionality of servers, or clusters of servers, in the face of such fluctuations. We review some of these techniques in Section 7.

Server loads also fluctuate at shorter, such as second or subsecond, scales. These 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 *on average* 1000 requests per second may see only 500 requests in some seconds, and 1500 requests in other seconds. These short term fluctuations present an opportunity for reducing server power consumption by quickly adjusting the capacity (and power consumption) of the server to match these fluctuations. This is the opportunity we seek to exploit in this paper. Techniques designed to address longer-term workload fluctuations are generally unable to respond to such short term changes. This 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 that adjust power consumption gradually.

The performance and power consumption of a server 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, such as in-memory transaction processing systems or key-value storage systems.

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) and/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 used to save power across a broad range of applications. However, for specific applications, they may leave substantial opportunities on the table.

Our central premise is that, for database servers, DVFS can be managed more effectively by the database management system (DBMS). The DBMS has two main advantages when managing DVFS. First, the DBMS is aware of database units of work, such as queries and transactions. It may have valuable information about these units of work, such as priorities, service level objectives, or the nature of the work itself. A DBMS can use this information to make better DVFS decisions. For example, a DBMS can slow down the CPU when executing a request with a low amount of work. Second, the

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196901

DBMS can also *control* its units of work. For example, it can reorder requests, or reject low value requests when load is high.

In this paper, we focus on the managing DVFS in transactional in-memory database systems. We consider systems that support multiple concurrent transactional *workloads*, each of which may have distinct characteristics, and each of which may have a different request latency target. For example, a workload associated with high priority customers may have a lower latency target than a workload of regular customers. 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 transaction execution, we can reduce *both* power consumption and the number of missed deadlines. For example, for TPC-C, POLARIS is able to cut power consumption by up to 40 Watts, while missing fewer deadlines, compared to running the CPUs in performance mode, i.e., at high frequency.

This paper 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 processing systems, like VoltDB [48] and Silo [51], 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 [58], a well-known optimal offline preemptive algorithm, as well as a YDS based on-line preemptive algorithm (OA [58]). This analysis provides insight into aspects of POLARIS's behavior, 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 [28]. Section 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.

2 BACKGROUND: DVFS

DVFS and related power-management mechanisms are standardized as the Advanced Configuration and Power Interface (ACPI) [52]. ACPI is an architecture-independent framework that defines discrete power states for CPUs and that allows software to control the power state of the underlying hardware. ACPI defines *P-States*, which represent a different voltage and frequency operating points. P_0 represents the P-State with the highest voltage and frequency, and hence the highest performance and the highest power consumption. Additional P-States, $P_1, \ldots, P_n$, represent successively lower voltage and frequency states, and hence greater tradeoffs of performance for power reductions. The exact operating point associated with each P-State varies from processor to processor. POLARIS works by choosing a P-State for the processor.

ACPI also defines *C-States*, which represent the processor's idle states. Although POLARIS does not directly manage C-states, we give a brief overview here. C-State C_0 is the processor's normal non-idle operating state, in which the CPU is active and executing instructions. In C_0 state, a CPU is running in one of the P-States. Additional C-States, $C_1, \ldots, C_m$, represent idle states, in which the CPU is not executing instructions. Higher-numbered C-States represent "deeper" idle states, in which more parts of the CPU are shut down. Normally, the deeper the idle state, the lower the power consumption of the idle processor, but the longer it takes the processor to return to the normal operating state (C_0) when there is work to do. Since C-States are idle states, C-State transitions are normally managed by the CPU itself.

Like other modern operating systems, Linux utilizes ACPI through a variety of kernel modules. Among these is the generic CPU power control module `cpufreq`, which supports a wide variety of CPU architectures. The `cpufreq` driver provides a number of *power governors* in two groups. The first group consists of static governors, which can be used to set a constant P-State for the processor. The other group includes dynamic governors, which monitor CPU utilization and adjust the processor's P-State in response to utilization changes. The `cpufreq` driver subsystem is exposed through the Linux sysfs filesystem. Through that interface, a system administrator or a privileged user-level application can select a governor, and can adjust governor parameters.

It is also possible for DVFS to be managed directly by the hardware. One example of this is Intel's Running Average Power Limit (RAPL) mechanism [27]. RAPL allows user-level applications to monitor CPU power consumption. In addition, given a specified power consumption limit, RAPL can dynamically adjust processor voltage and frequency levels to keep the CPU's power consumption within the specified upper bound. Like OS-based governors, RAPL is unaware of DBMS-level workload information, such as transaction deadlines.

3 POLARIS

POLARIS is designed to manage DVFS for a transactional database server. 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 5, we describe the POLARIS prototype architecture that uses this approach to manage a multi-processor, multi-core server.

POLARIS is *workload-aware*. A server accepts transaction execution requests, and each request is assumed to be tagged with a

workload identifier to indicate which workload it is part of. Each workload known to POLARIS is associated with a *latency target*. POLARIS's objective is to minimize CPU power consumption while ensuring that each request is completed within its workload's latency target.

Workloads are important to POLARIS, since different workloads can have different latency targets, e.g., one workload with a tight latency target for priority customers' transactions, and a different workload with a looser target for others. POLARIS also tracks and estimates request execution times on a per-workload basis. Many database systems provide sophisticated *workload managers* [23, 26, 40, 42, 43, 49] 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 [25] 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 there are no constraints on 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.

3.1 The POLARIS Algorithm

Execution of the POLARIS algorithm is triggered by the arrival of a new transaction request, or the completion of a request. 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 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 1 summarizes notation that we use to describe transactions and processor frequencies.

Figure 2 shows the POLARIS frequency selection procedure, SetProcessorFreq, which runs each time a transaction request arrives or is completed. SetProcessorFreq chooses the smallest 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

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 1: Summary of Notation

of a given workload at a given processor frequency. We use $\hat{\mu}(c, f)$ (in Figure 2) to represent the predicted execution time of a workload c transaction at frequency f. We discuss how POLARIS predicts execution time in Section 3.2.

In Figure 2, $\hat{q}(t, f)$ represents the total estimated queueing time for transaction $t \in Q$, assuming that the processor runs at frequency f. This is defined as 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 1, each transaction, once dispatched runs to completion. In Section 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.

3.2 Execution Time Estimation

POLARIS requires estimates of the execution time $\hat{\mu}(c, f)$ for transactions of each $c \in \mathcal{W}$ at each $f \in \mathcal{F}$. There is a substantial body of work on estimating execution times of database queries [2, 16, 18, 21]. This work varies in the assumed complexity of the workload, 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

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 **:**
    if $t_{now} + \hat{\mu}(c(t_0), f_{new}) - e_0 \leq d(t_0)$ **: break**
  ▷ ensure all queued transactions finish in time
  for each t in Q, in EDF order **:**
    if $t_{now} + \hat{q}(t, f_{new}) + \hat{\mu}(c(t), f_{new}) \leq d(t)$ **: continue**
    ▷ 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 **:**
      $f_{new} \leftarrow f$
      if $t_{now} + \hat{q}(t, f) + \hat{\mu}(c(t), f) \leq d(t)$ **: break**
    ▷ no further checking once we need highest freq
    if f_{new} = maximum frequency in $\mathcal{F}$ **:**
      set processor frequency to f_{new}
      return
  set processor frequency to f_{new}
  return

Figure 2: POLARIS Processor Frequency Selection

and data contention can affect execution. In addition, workloads 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 2).

To track these percentiles, we adapted an algorithm of Härdle and Steiger [22] 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 paper, 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 paper, we have used $p = 95$. This is important because POLARIS's primary objective is to meet transaction latency targets. For example, Figure 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

Request Type	Execution Time (μs) @2.8 GHz Mean	@2.8 GHz P_{95}	@1.2 GHz Mean	@1.2 GHz 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: TPC-C mean and 95th percentile (P_{95}) transaction execution times at maximum and minimum CPU frequency. Percentages indicate the transaction mix in the workload.

are 2.5 to 4.8 times larger than the means. The use of lower values of $p = 95$ will make POLARIS save power more aggressively, but also increases the risk of missed latency targets.

4 POLARIS ANALYSIS

In this section we analyze the performance of POLARIS through a competitive analysis against two existing algorithms YDS [58] (Section 4.2) and OA [10, 58] (Section 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 the standard theoretical model [7, 10, 58] 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 4.1.

Broadly, energy aware scheduling algorithms can be classified into four categories along two dimensions as shown in Figure 4: (1) preemptive vs non-preemptive; and (2) offline vs online algorithms. Offline preemptive algorithms are the most computationally powerful algorithms. YDS [58] 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 [7], 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 4.4 and 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 [6] or minimizing the total response time [4] of transactions under a fixed energy budget. However, no prior work studies the problem of minimizing energy consumption as we do in this section. We refer the reader to references [3] and [19] for a survey of these algorithms.

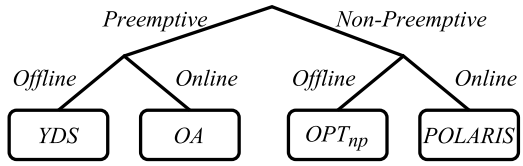


Figure 4: Energy aware scheduling algorithms.

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 [12], 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.

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.

4.3 Optimal Available (OA)

OA is an online preemptive algorithm based on YDS [58]. 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. Bansal et al. showed that OA is α^α competitive against YDS [10].

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 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.*

The proof of this lemma is provided in Appendix A. The proof argues inductively that a) by the observation we made at the end of Section 4.3, both POLARIS and OA will execute transactions in EDF order; and b) will have identical processor speeds at any moment.

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 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 $d(t_{new})$.*

The proof is similar to that of Lemma 4.1, and a sketch of it is given in Appendix B

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 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 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 [10]. □

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 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 [10], and is provided in Appendix C. □

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

Theorem 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 4.4 and 4.5.

Corollary 4.6. *POLARIS has a* $(c\alpha)^\alpha$ *competitive ratio against YDS and therefore* OPT_{np}.

4.6 Discussion

The competitive ratio in Corollary 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 [10]. 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.

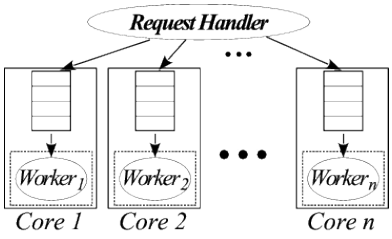


Figure 5: Architecture of the POLARIS Shore-MT Prototype

5 POLARIS PROTOTYPE

To test POLARIS, we implemented it in Shore-MT [28]. Shore-MT is a multi-threaded data storage manager which is designed for multiprocessors. Shore-Kits [17] provides a front-end driver for Shore-MT. It includes implementations of several database benchmarks, including TPC-C and TPC-E. For the remainder of the paper, 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 which 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. This architecture is illustrated in Figure 5.

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. For each core, we run a separate instance of POLARIS, 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: arrival of a new transaction request, and completion of a request. In our prototype, POLARIS's request arrival action is handled by the RH threads. 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 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. POLARIS's request completion action is handled by the worker threads. 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. There are several mechanisms for doing so. For x86 processors, all of the alternatives ultimately rely on Model Specific Registers (MSRs) [1, 27]. MSRs contains CPU related information which can be read and/or written by software, and which can be used to control some aspects of the processor, including core frequencies.

One common way to change CPU frequency on Linux systems is to use the `cpufreq` driver's `userspace` governor. Application code can specify a core frequency in a special `sysfs` system file, and the `userspace` governor then uses the `cpufreq` driver to set core frequency as specified. The driver, in turn, controls frequency using the MSRs. This interface is relatively simple to use, but we found that it introduces substantial latency, as was previously observed by Wamhoff et al. [53]. 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.

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.

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 request 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 [9] 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) 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 [24] 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 static and dynamic OS baselines. For the dynamic OS baselines, 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. For the static OS baselines, we use MSRs to set all cores to run at a fixed frequency. Under both the static and dynamic baselines, Shore-MT uses its default transaction scheduler and does not attempt to adjust core frequencies.

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 (hyper-thread) 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 transaction type or workload. 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 [14]. We use Shore-MT's default staged group commit configuration, under which log I/O is forced at least once per 100 transactions.

6.2 Results: Medium Load

For our initial experiment, we focus on the medium load scenario, meaning that the overall request rate is about 60% of the server's maximum capacity. We consider both TPC-C and TPC-E workloads. We begin with TPC-C, and present TPC-E in Section 6.2.1. 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 workload, the target latency is set to *slack* times the mean execution time (at high frequency) for that workload's transaction 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. 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.

Figure 6 shows the results of this experiment, as a function of slack. In addition to POLARIS, we report the results for the two

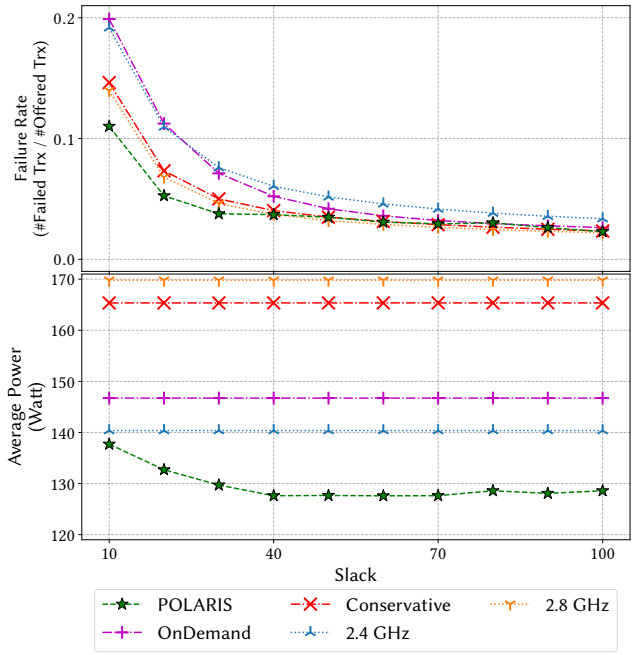


Figure 6: Performance and power of different power management schemes under medium load.

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 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 of about 25 watts, 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 over 30 watts relative to consumption at peak frequency. These power savings do *not* come at the expense of missed latency targets. Indeed, when slack is tight, POLARIS cuts the missed deadline rate almost in half relative to OnDemand. Perhaps surprisingly, fewer transactions miss their deadlines under POLARIS than under 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. When slack is high, POLARIS and the baselines have similar failure rates. Because of load fluctuations and occasional execution time outliers, a small percentage of transactions miss their deadlines regardless of how processor speed is controlled. However, POLARIS is able to take advantage of the higher slack to slow transactions down and reduce power consumption, which

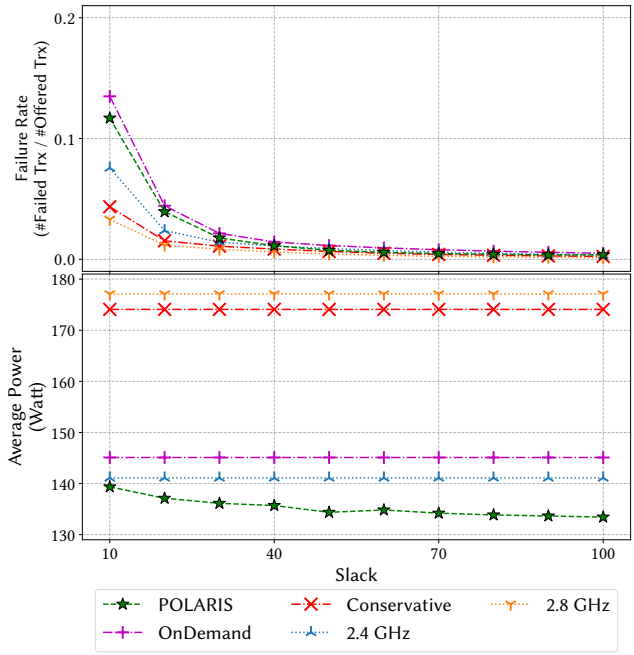


Figure 7: TPC-E performance and power of different power management schemes under medium load.

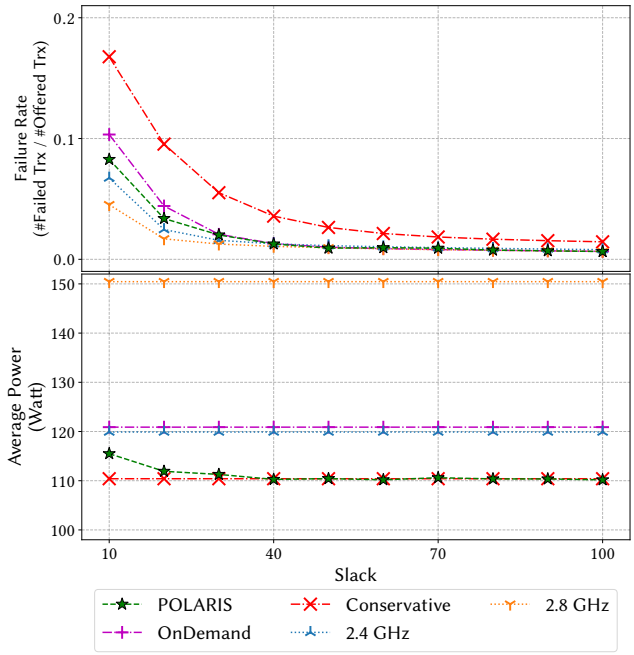


Figure 8: Performance and power of different power management schemes under low load.

the baselines cannot do. With increasing slack, POLARIS' power savings (relative to 2.8 GHz) climb to more than 40 watts, with no increase in late transactions. 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.

6.2.1 Medium Load, TPC-E. 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. We use the same estimator parameter settings as for TPC-C.

Figure 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 was the case for TPC-C, the power savings are greater with greater slack, although the effect is not as strong. The 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 governor. However, this is achieved at the cost of about 35 watts.

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 and high load mean an average request arrival rate of 30% and 90% of the systems peak sustainable load, respectively.

Figure 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.

A comparison of the medium and low load experiments (Figures 6 and 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.

Finally, Figure 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

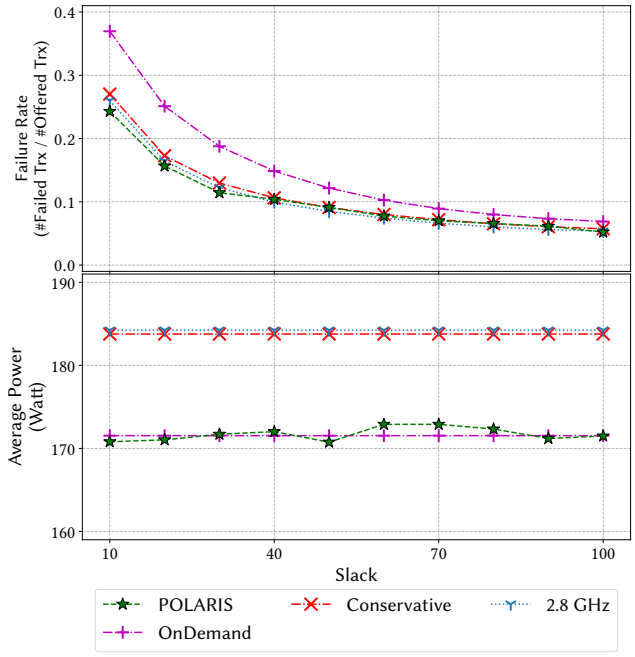
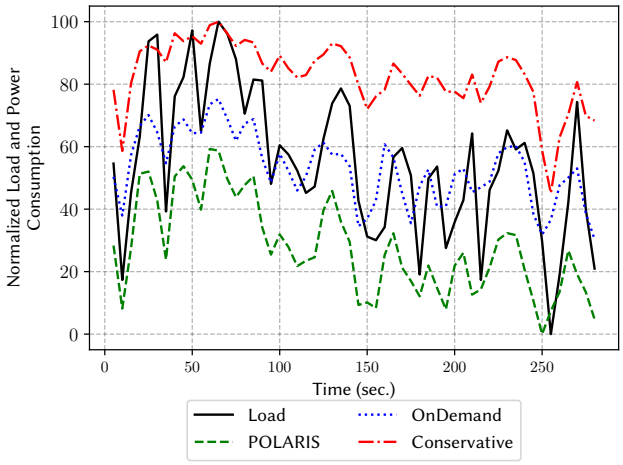


Figure 9: Performance and power of different power management schemes under high load.



(a) World Cup Trace Timeline for Normalized Load Level and power consumption, bins of 5 seconds.

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

(b) Average Power consumption and failure rate of baselines.

Figure 10: World Cup Trace - Normalized.

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 noted in Section 1, real systems may experience both long term and short term load fluctuations. Our results with low, medium, and high load experiments suggest that POLARIS can function effectively as load fluctuates over longer time frames. 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.

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 [9] 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 "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 10(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 10(b) summarizes the average power consumption and failure rate (percentage of transactions that missed latency targets) over the entire experiment. As was the case in the steady load experiments, POLARIS results in both lower power consumption and fewer missed latency targets than either of the OS baselines. All of the algorithms adjust power consumption in response to the load changes, but POLARIS's adjustments tend to be sharper and deeper.

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 11 shows the failure rate for each workload, under POLARIS, the Linux dynamic governors, and the high frequency static

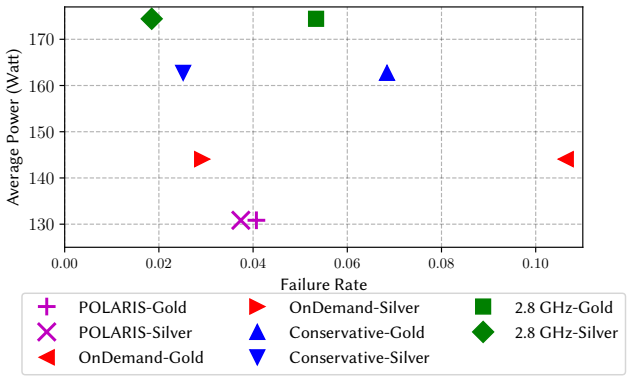


Figure 11: Per-Workload Performance for Gold and Silver TPC-C Workloads

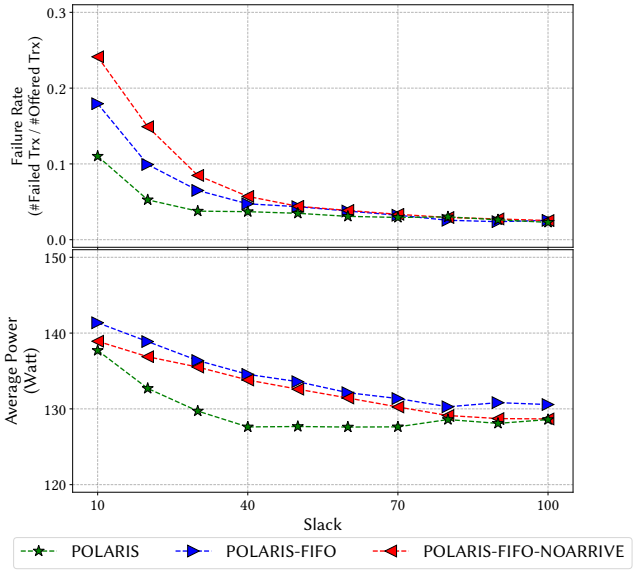


Figure 12: Performance POLARIS and Variants

governor. Each failure rate is plotted against the total power consumption for that run, as we cannot separately attribute power to individual 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.

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 12 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. This is why POLARIS-FIFO has fewer missed transactions than POLARIS-FIFO-NOARRIVE, at the cost of some additional power consumption. The results also show that EDF contributes to power savings, because it allows POLARIS to meet latency targets with lower frequencies.

7 RELATED WORK

In this section, we discuss related work in several broad categories. We first consider cluster-level techniques that are designed to operate across multiple servers, and then single-server techniques. Finally, we consider techniques that have been specifically targeted at database systems.

7.1 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 [35, 36]. Another is to focus on energy-efficient virtual machine placement across the cluster [33, 56]. Facebook controls server power consumption to prevent data center power overloads [54]. These techniques typically operate at much longer time scales (e.g., minutes or hours) than POLARIS, 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 is complementary to some of these techniques. For example, it can be used to manage DVFS on servers that are not shut down by a cluster-level manager.

7.2 Server-Level Energy Efficiency

Another body of work targets single server energy efficiency, like POLARIS. Spiliopoulos et al. [47] propose an operating system power governor which uses memory stalls as an input and tries to optimize CPU energy efficiency accordingly. Sen and Wood [46] 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 6, these do not take advantage of application-level workload information.

PAT [55] and PEGASUS [37] apply feedback control to manage processor DVFS. PAT uses a control mechanism to maintain a target system throughput 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. Unlike POLARIS, which operates at the timescale of individual transactions and tries to ensure that every transaction meets its deadline, PEGASUS tries to ensure that mean transaction latency over a sliding time window (say, 30 or

60 seconds) is less than the specified target. Thus, it takes time for PEGASUS to observe system state and adjust to fluctuations. PEGASUS is intended to react to changes over longer time scales (minutes, hours, days). PEGASUS is also not easily generalized to handle multiple concurrent workloads, as a single control setting applies to all transactions in a time window.

There are a few techniques designed to respond to very short term load fluctuations, like POLARIS. Both LAPS [32] and Rubik [31] manage DVFS on the time scale of individual transaction arrivals. 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, offline, 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. LAPS adjusts frequency only on transaction completion, and does not define a specific technique for estimating transaction execution times. In Section 6.6, we considered two variants of POLARIS that are similar to Rubik (POLARIS-FIFO) and LAPS (POLARIS-FIFO-NOARRIVE). These variants execute transactions and set processor frequency like Rubik and LAPS. However, both variants use POLARIS' execution time estimation technique.

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 [37, 38] or because processors consume a lot of energy to recover from deep sleep states [29, 45]. Therefore, some work encourages deeper C-States by extending sleep periods [5, 39]. In contrast, we focus only on P-states in this work.

There is also work on DRAM energy efficiency. Appuswamy et al. [8] show that large-memory servers can consume a substantial amount of power and illustrate the potential for memory power saving. Karyakin et al. [30] demonstrate that memory power consumption in-memory database systems is not proportional to system load, and is also not proportional to database size. They point to DRAM low-power states as the key to memory power savings.

7.3 Energy Efficiency in DBMSs

Several studies describe techniques for improving energy efficiency through query optimization and query operator configuration. Tsirogiannis et al. [50] 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 under enough load and parallelism, while different CPU core frequencies 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. [44] 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. [57] and Lang et al. [34] explore possibilities of energy aware query optimization in relational DBMSs. For this, they propose a cost function having both performance and power as the objective. They show that DBMSs can execute queries according to specific power/performance requirements. Mühlbauer et al.[41] show that heterogeneity-aware parallel query plans can save power while having better performance in emerging heterogeneous multi-core CPUs. These techniques are complementary to POLARIS.

8 CONCLUSION & FUTURE WORK

In this paper, we have presented a workload-aware frequency scaling and scheduling technique for transactional database systems, and related it to other well-known offline and online 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.

Our target for POLARIS is in-memory transaction processing systems. However, POLARIS is potentially applicable to a broader class of systems. The most important requirement for POLARIS is that units of work (transactions) are executed non-preemptively. In addition, POLARIS will work best when the execution times of units of work are relatively short. Other kinds of systems, such as search engines and key-value databases, have these properties, and may also be good targets for POLARIS. Another interesting direction to explore is the use of POLARIS-like algorithms in virtualized environments, such as clouds. Here, key issues include how the physical processors are shared by virtual machines, and whether to implement POLARIS-like functionality as part of the virtualization layer, or within each virtual machine.

One challenge faced by POLARIS is the need to estimate transaction execution times at different processor frequencies. The power savings achieved by POLARIS in our experiments were achieved using a conservative estimator that can track time-varying workloads. Thus, perfect estimation is not necessary for in-DBMS workload-aware frequency scaling to be effective. However, better estimates would allow POLARIS to reduce processor frequency more aggressively, and further reduce power.

On multi-socket, multi-core servers, there may be opportunities for reducing power consumption beyond what is achievable by POLARIS. POLARIS controls transaction execution order and speed independently on each core, and request handling threads do round-robin assignment of requests to workers. By controlling how transactions are distributed to workers, we can obtain additional power savings by allowing some workers (and their cores) to idle and move into low-power C-states. The same technique can be applied at the package level. We are currently exploring extensions to POLARIS in this direction.

ACKNOWLEDGMENTS

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

REFERENCES

[1] Advanced Micro Devices(AMD). 2017. Architecture Programmer's Manual: Volume 2: System Programming. 24593 (2017).
[2] M. Akdere et al. 2012. Learning-based Query Performance Modeling and Prediction. In *Proc. ICDE*. 390–401.
[3] S. Albers. 2011. Algorithms for dynamic speed scaling. In *STACS*, Vol. 9. 1–11.
[4] S. Albers and H. Fujiwara. 2007. Energy-efficient algorithms for flow time minimization. *ACM TALG* 3, 4 (2007).
[5] H. Amur et al. 2008. Idlepower: Application-aware management of processor idle states. In *Proc. Workshop on Managed Many-Core Systems*, Vol. 8.
[6] E. Angel et al. 2016. Throughput maximization in multiprocessor speed-scaling. *Theoretical Computer Science* 630 (2016), 1–12.
[7] A. Antoniadis and C. Huang. 2013. Non-preemptive speed scaling. *Journal of Scheduling* 16, 4 (2013), 385–394.
[8] R. Appuswamy et al. 2015. Scaling the memory power wall with dram-aware data management. In *Proc. DaMoN*.
[9] M. Arlitt and T. Jin. 2000. A workload characterization study of the 1998 world cup web site. *IEEE Network* 14, 3 (2000), 30–37.
[10] N. Bansal, T. Kimbrel, and K. Pruhs. 2007. Speed scaling to manage energy and temperature. *J. ACM* 54, 1 (2007).
[11] L. A. Barroso and U. Hölzle. 2007. The case for energy-proportional computing. *IEEE Computer* 40, 12 (2007).
[12] D. M. Brooks et al. 2000. Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *IEEE Micro* 20, 6 (2000), 26–44.
[13] Y. Chen, S. Alspaugh, and R. Katz. 2012. Interactive Analytical Processing in Big Data Systems: A Cross-industry Study of MapReduce Workloads. *Proc. VLDB Endow.* 5, 12 (2012), 1802–1813.
[14] Transaction Processing Performance Council. 2015. TPC BENCHMARK E-Standard Specification-Version 1.14.0. (2015).
[15] C. Delimitrou and C. Kozyrakis. 2014. Quasar: Resource-efficient and QoS-aware Cluster Management. *ACM SIGPLAN Not.* 49, 4 (2014), 127–144.
[16] J. Duggan, U. Cetintemel, O. Papaemmanouil, and E. Upfal. 2011. Performance Prediction for Concurrent Database Workloads. In *Proc. SIGMOD*. 337–348.
[17] EPFL. 2012. EPFL Official Shore-MT Page, Shore-Kits. https://sites.google.com/site/shoremt/shore-kits. (2012). Accessed: Feb. 2017.
[18] A. Ganapathi et al. 2009. Predicting Multiple Metrics for Queries: Better Decisions Enabled by Machine Learning. In *Proc. ICDE*. 592–603.
[19] M. E. T. Gerards, J. L. Hurink, and P. K. F. Hölzenspies. 2016. A survey of offline algorithms for energy minimization under deadline constraintsa. *Journal of Scheduling* 19, 1 (2016), 3–19.
[20] D. Gmach et al. 2007. Workload analysis and demand prediction of enterprise data center applications. In *IEEE IISWC*. 171–180.
[21] H. Hacigumus et al. 2013. Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable?. In *Proc. ICDE*. 1081–1092.
[22] W. Hardle and W. Steiger. 1995. Algorithm AS 296: Optimal median smoothing. *Journal of the Royal Statistical Society. Series C (Applied Statistics)* 44, 2 (1995), 258–264.
[23] Hewlett Packard Enterprise. 2017. HPE Global Workload Manager 7.6. https://support.hpe.com/hpsc/doc/public/display?sp4ts.oid=3725908. (2017). Accessed: Oct. 2017.
[24] J. M. Hirst et al. 2013. Watts Up? PRO AC Power Meter for automated energy recording: A product review. *Behavior Analysis in Practice* 6, 1 (2013), 82.
[25] IBM Corporation. 2013. 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: Jan. 2018.
[26] IBM Corporation. 2017. 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.
[27] Intel Corporation. 2016. *Intel 64 and IA-32 Architectures Software Developer's Manual.*
[28] R. Johnson et al. 2009. Shore-MT: a scalable storage manager for the multicore era. In *Proc. EDBT*. 24–35.
[29] S. Kanev, K. Hazelwood, G. Wei, and D. Brooks. 2014. Tradeoffs between power management and tail latency in warehouse-scale applications. In *IEEE IISWC*. IEEE, 31–40.
[30] A. Karyakin and K. Salem. 2017. An analysis of memory power consumption in database systems. In *Proc. DaMoN*. 2.
[31] H. Kasture et al. 2015. Rubik: Fast Analytical Power Management for Latency-critical Systems. In *IEEE/ACM MICRO (MICRO-48)*. 598–610.
[32] M. Korkmaz et al. 2015. Towards Dynamic Green-Sizing for Database Servers. In *ADMS@VLDB*. 25–36.
[33] Gregor V. L. et al. 2009. Power-aware scheduling of virtual machines in DVFS-enabled clusters, in. In *Proc. IEEE Int'l Conf. Cluster Computing*. 1–10.
[34] W. Lang, R. Kandhan, and J. M. Patel. 2011. Rethinking Query Processing for Energy Efficiency: Slowing Down to Win the Race. *IEEE Data Eng. Bull.* 34, 1 (2011), 12–23.
[35] W. Lang and J. M Patel. 2010. Energy management for mapreduce clusters. *Proc. of the VLDB Endow.* 3, 1-2 (2010), 129–139.
[36] J. Leverich and C. Kozyrakis. 2010. On the Energy (in)Efficiency of Hadoop Clusters. *ACM SIGOPS Oper. Syst. Rev.* 44, 1 (2010), 61–65.
[37] D. Lo et al. 2014. Towards Energy Proportionality for Large-scale Latency-critical Workloads *(IEEE ISCA)*. 301–312.
[38] D.others Meisner. 2011. Power management of online data-intensive services. In *IEEE ISCA*. 319–330.
[39] D. Meisner, B. T. Gold, and T. F. Wenisch. 2009. PowerNap: Eliminating Server Idle Power. *ACM SIGARCH Comput. Archit. News* 37, 1 (2009), 205–216.
[40] Microsoft Corporation. 2017. Microsoft SQL SERVER resource governor. https://docs.microsoft.com/en-us/sql/relational-databases/resource-governor/resource-governor. (2017). Accessed: Oct. 2017.
[41] T. Mühlbauer et al. 2014. Heterogeneity-Conscious Parallel Query Execution: Getting a better mileage while driving faster!. In *Proc. DaMoN*. 2.
[42] Oracle Corporation. 2017. Managing Resources with Oracle Database Resource Manager. https://docs.oracle.com/database/121/ADMIN/dbrm.htm. (2017). Accessed: Oct. 2017.
[43] Pivotal Corporation. 2017. Greenplum Workload Manager. https://gpcc.docs.pivotal.io/220/gp-wlm/topics/gpwlm-docs.html. (2017). Accessed: Oct. 2017.
[44] I. Psaroudakis et al. 2014. Dynamic Fine-grained Scheduling for Energy-efficient Main-memory Queries *(Proc. DaMoN)*. 1–7.
[45] R. Schöne, D. Molka, and M. Werner. 2015. Wake-up latencies for processor idle states on current x86 processors. *Computer Science-Research and Development* 30, 2 (2015), 219–227.
[46] R. Sen and David A. Wood. 2017. Pareto Governors for Energy-Optimal Computing. *ACM TACO* (2017), 6:1–6:25.
[47] V. Spiliopoulos, S. Kaxiras, and G. Keramidas. 2011. Green governors: A framework for continuously adaptive dvfs. In *IEEE IGCC*. 1–8.
[48] M. Stonebraker and A. Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27.
[49] Teradata Corporation. 2017. Teradata Workload Manager. http://www.teradata.com/products-and-services/workload-management. (2017). Accessed: Oct. 2017.
[50] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. 2010. Analyzing the Energy Efficiency of a Database Server. In *Proc. SIGMOD*. 231–242.
[51] S. Tu et al. 2013. Speedy transactions in multicore in-memory databases. In *Proc. ACM SOSP*. 18–32.
[52] Unified EFI Inc. 2016. Advanced Configuration and Power Interface Specification. http://www.uefi.org/sites/default/files/resources/ACPI_6_1.pdf. (2016). Accessed: Feb. 2017.
[53] J. Wamhoff et al. 2014. The TURBO Diaries: Application-controlled Frequency Scaling Explained. In *Proc. USENIX ATC*. 193–204.
[54] Q. Wu et al. 2016. Dynamo: Facebook's data center-wide power management system. In *IEEE ISCA*. 469–480.
[55] Zichen X., Xiaorui W., and Yi cheng T. 2013. Power-Aware Throughput Control for Database Management Systems. In *Proc. ICAC*. 315–324.
[56] J. Xu and J. A. B. Fortes. 2010. Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments. In *PRoc. IEEE/ACM GreenCom & CPSCOM*. 179–188.
[57] Z. Xu, Y. C. Tu, and X. Wang. 2010. Exploring power-performance tradeoffs in database systems. In *Proc. ICDE*. 485–496.
[58] F. Yao, A. Demers, and S. Shenker. 1995. A Scheduling Model for Reduced CPU Energy. In *Symposium on Foundations of Computer Science*.

APPENDIX

A PROOF OF LEMMA 4.1

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 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

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

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 $n + 1$.

B PROOF SKETCH OF LEMMA 4.2

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.

C PROOF OF THEOREM 4.4

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 $a(t_j) \le t \le d(t_j)$. 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:

$$s_P(t)^\alpha + \frac{d\phi(t)}{dt} \le \alpha^\alpha s_Y^\alpha \quad (1)$$

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

$$Pow[POLARIS(P)] \le \alpha^\alpha Pow[YDS(P')].$$

.

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 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 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 [10] 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 [10] for OA.

We need to show that when no transactions arrive in the next dt time equation 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:

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

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:

$$\frac{d\phi(t)}{dt} = \alpha(s_{Pna}(t_0)^{\alpha-1}(-s_{Pna}(t_0)) - \alpha s_{Pna}(t_k)^{\alpha-1}(-s_Y(t_0)))$$

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

Substituting this into equation 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 [10] 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 [10]. 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 [10] 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 [10]. Second transactions added to POLARIS's queue and YDS's queue are different. The proof in reference [10] 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 [10]. Below we slightly simplify the proof in reference [10].

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 [10]. 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 [10] 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.