Database-Managed CPU Performance Scaling for Improved Energy Efficiency

[Technical Report CS-2017-3, Cheriton School of Computer Science, University of Waterloo. June 2017]

Mustafa Korkmaz
University of Waterloo
mkorkmaz@uwaterloo.ca

Martin Karsten
University of Waterloo
mkarsten@uwaterloo.ca

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

Semih Salihoglu
University of Waterloo
semih.salihoglu@uwaterloo.ca

ABSTRACT

Dynamic voltage and frequency scaling (DVFS) is a technique for adjusting the speed and power consumption of processors, allowing performance to be traded for reduced power consumption. Since CPUs are typically the largest consumers of power in modern servers, DVFS can have a significant impact on overall server power consumption. Modern operating systems include DVFS governors, which interact with the processor to manage performance and power consumption according to some system-level policy.

In this paper, we argue that for database servers, DVFS can be managed more effectively by the database management system. We present a power-aware database request scheduling algorithm called POLARIS. Unlike operating system governors, POLARIS is aware of database units of work and database performance targets, and can achieve a better power/performance tradeoff by exploiting this knowledge. We implemented POLARIS in SHORE-MT, and we show that it can improve *both* power consumption and performance relative to operating system baselines.

1. INTRODUCTION

Shehabi et al. [40] report that, in 2014, data centers in the US consumed 70 billion kilowatt-hours (kWh) of energy. This represents 1.8% of total US electricity consumption, and consumption grew 4% from 2010 to 2014. Power consumption is also an important issue in data center design, and is directly or indirectly responsible for a significant portion of the total cost of data center ownership (TCO) [16, 21]. A recent survey by Dayarathna et al. [13] indicates that servers and the cooling infrastructure (required to remove server-generated heat) are the largest power consumers in a typical data center. Since CPUs are responsible for much of the servers' power consumption [25, 34] and are also the primary heat generators [15], power-efficient use of CPUs can have a positive environmental impact and can help to reduce the TCO of data centers.

Servers do not always run at maximum capacity, because workloads fluctuate and can be bursty [12, 14, 18]. To accommodate such fluctuations, data centers are generally overprovisioned [19]. Thus, during non-peak times, which are common, data center resources are not fully utilized [10, 25]. To better match power consumption to the workload, most server processors support *dynamic voltage and frequency scaling (DVFS)*. DVFS allows the CPU's voltage and frequency, and hence power consumption, to be adjusted on the fly. This permits a performance/power tradeoff. During periods of low load, when peak performance is not critical, CPU voltage and frequency can be scaled down to save power.

In this paper, we consider the problem of CPU power management for database servers, i.e., servers that are dedicated to running a database management system (DBMS). We focus specifically on systems designed to support transactional workloads against in-memory databases. These systems handle workloads that have relatively short units of work, for which there are latency targets.

Our central premise is that DVFS can be managed more effectively by the DBMS than by the operating system. Modern operating systems include power governors that manage DVFS based on system-perceived load. Unlike the operating system, the DBMS is aware of database units of work, such as queries and transactions. Because of this, it can take database-level performance requirements into account when managing DVFS. Operating system power governors, in contrast, are limited to kernel-level metrics, such as CPU utilization, and generic objectives, such as avoiding utilizations that are too low. In addition, the DBMS has the ability to control how database units of work are scheduled to execution threads. Operating systems are unaware of this mapping, and are thus unable to control it. In this paper, we show that, because of these flexibilities, database-managed DVFS can provide *both* better power efficiency and better performance than operating-system-based alternatives.

The specific problem we address in this paper is *power-aware transaction scheduling* in database systems. Given a set of transactions, each with an associated deadline (latency target), the problem is to choose both the execution order of the transactions and the processor frequency

at which they will execute. The objective is to minimize power consumption while ensuring that as many transactions as possible complete before their deadline. In Section 4 of this paper, we present an on-line power-aware scheduling algorithm for non-preemptive requests, called POLARIS (POwer and Latency Aware Request Scheduling). We focus on non-preemptive scheduling because many modern in-memory transaction processing systems, like VoltDB [43] and Silo [45], are architected to minimize blocking and to execute each transaction from start to finish in a single thread on a single processor core. Non-preemptive scheduling is a good fit for such systems. We have implemented POLARIS within the Shore-MT storage manager [24]. Section 6 describes some of the practical issues that we had to address in doing so.

We have analyzed POLARIS' behavior and performance in two ways. First, in Section 5, we provide a competitive analysis of POLARIS against two existing algorithms for power-aware scheduling. We compare it against both an existing on-line preemptive algorithm (OA [52]) and an optimal off-line preemptive algorithm (YDS [52]). Second, in Section 7, we present an empirical evaluation of our Shore-MT implementation, using transactional (TPC-C) workloads. Our evaluation compares POLARIS against two Linux DVFS governors, and also against two in-DBMS alternatives from the literature. Our results show that POLARIS dominates the alternatives, producing greater power savings, fewer missed transaction deadlines, or both. We also show how POLARIS' effectiveness is affected by two key factors: (1) *load on the system*, and (2) *scheduling slack*, i.e., the looseness of the transactions' deadlines. Although POLARIS dominates the baselines under all conditions, its benefits are greatest when the 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 frequency scaling alternatives in the Linux kernel.

2. BACKGROUND

DVFS and related power-management mechanisms are standardized under the Advanced Configuration and Power Interface (ACPI) [46]. 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.

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 processors 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 [23]. 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 operating system based governors, RAPL is unaware of DBMS-level workload information, such as transaction deadlines.

3. RELATED WORK

In this section, we discuss related work in three broad categories: cluster level energy efficiency, single server energy efficiency for generic applications, and single server DBMS specific energy efficiency. POLARIS falls into the third category.

3.1 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 [29, 31]. Another is to focus on energy-efficient ways of virtual machine placement across the cluster [30, 49]. Facebook controls server power consumption to prevent data center power overloads [48]. PEGASUS [33] is a system for controlling DVFS across many similar servers. It targets scenarios in which individual servers cannot be shut down when load is low. These techniques typically operate at much longer time scales (e.g., minutes or hours) than POLARIS, which schedules and manages DVFS for individual requests on a single server. 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.

3.2 Server-Level Energy Efficiency

Another group of studies targets single server energy efficiency, focusing on software and/or hardware-level opportunities that do not target a specific type of application. Spiliopoulos et al. [41] propose an operating system power governor which uses memory stalls as an input and tries to optimize CPU energy efficiency accordingly. Sen and Wood [39] propose an operating system governor that predicts the system power/performance pareto optimality frontier and keeps the power/performance at this frontier. Al-

though these techniques may help for a variety of applications, they do not use DBMS-specific information that POLARIS utilizes.

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 [32, 36] or because processors consume a lot of energy to recover from deep sleep states [25, 38]. Therefore, some work encourages deeper C-States by extending sleep periods [5, 35]. In contrast, we focus only on P-states in this work.

3.3 Server-Level Energy Efficiency in DBMSs

Several studies describe techniques for improving energy efficiency through query optimization and query operator configuration. Tsirogiannis et al. [44] 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. In the same direction, Psaroudakis et al. [37] 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. [50] and Lang et al. [28] 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. In contrast to these efforts, POLARIS exploits deadline information to reduce power consumption.

PAT [51], like POLARIS, uses DVFS to save power under user-defined performance bounds. PAT uses a feedback control loop mechanism, like PEGASUS [33]. While this kind of approach can be effective, it operates on a longer time scale than POLARIS, which adjusts frequencies based on the deadlines of individual transaction requests. In addition, since POLARIS is a scheduling algorithm, it is capable of reordering transactions in addition to adjusting processor frequency.

Closest to POLARIS are our previous algorithm, LAPS [27] and Kasture et. al.'s Rubik [26] algorithm. Like POLARIS, both LAPS and Rubik adjust processor frequency for each request in a request queue. Unlike POLARIS, LAPS does not reorder transactions and adjusts processor frequency only on request completion (see Section 4). Rubik is similar to LAPS, but adjusts processor frequency on both request arrival and completion. In our empirical evaluation of POLARIS, we use LAPS and a Rubik-like algorithm, called LAPS-Arrival, as baselines.

4. POLARIS

In this section, we present POLARIS. For simplicity, we assume a system consisting of a single processor with a single core. In Section 6, we describe how we use POLARIS in a system with multiple cores. In our setup, one or more clients submit transaction execution requests to the processor. There are different types, or classes, of transactions. Each transaction also has an associated, client-selected soft execution deadline. These deadlines represent latency targets: the submitting client expects the requested transaction to finish executing before the specified deadline.

notation	meaning
Q	transaction request queue
t_0	currently running transaction
e_0	running time (so far) of t_0
$\mathcal{C}$	set of possible transaction classes
$c(t)$	class (type) of transaction t, $c(t) \in \mathcal{C}$
$a(t)$	arrival time of transaction t
$d(t)$	deadline of transaction t
$\mathcal{F}$	set of possible processor frequencies
$\hat{\mu}(c, f)$	estimated execution time of class c transaction at frequency f
$\hat{q}(t, f)$	estimated queuing time of t at frequency f

Figure 1: Summary of Notation

There is a fixed set of frequencies at which the processor can run. 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.

The goals of the POLARIS algorithm are, first, to ensure that transactions finish execution before their deadlines and, second, to minimize the energy consumed by the processor. Because 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.

POLARIS is free to control two aspects of transaction execution. First, it can control the order in which queued transactions are executed. Second, it can control the processor's frequency. Next, we describe each of these aspects of POLARIS.

Transaction Execution Order:
POLARIS runs queued transactions in earliest deadline first (EDF) order. When a transaction request arrives, it is placed in a request queue unless the processor is idle, in which case the new request runs immediately. When a transaction finishes executing, the queued transaction request with the earliest deadline is dequeued and the requested transaction starts running immediately. Transactions are executed non-preemptively. That is, each transaction, once started, runs until it is finished.

Processor Frequency Selection:
POLARIS considers adjusting processor execution frequency at two points: when a new request arrives, and when a transaction finishes executing. In each case, POLARIS uses the procedure shown in Figure 2 to choose a new execution frequency. This procedure chooses the smallest processor frequency such that all transactions, including the running transaction and all queued transactions in Q, will finish running before their deadlines if run at that frequency.

The POLARIS frequency selection algorithm relies on a transaction execution time model, which predicts the execution time of a transaction of a given class at a given processor frequency. We use $\hat{\mu}(c, f)$ (in Figure 2) to represent the predicted execution time of a class c transaction at frequency f. For now, we assume that the model is given, and that it is accurate. A realistic implementation of POLARIS must have a way of obtaining such a model, and must account

State: Q: queue of waiting transactions
State: $\hat{\mu}(c, f)$: execution time of class $c \in \mathcal{C}$ at freq $f \in \mathcal{F}$
State: t_0: currently running transaction
State: e_0: run time (so far) of t_0

```
 1: function SetProcessorFreq( )
 2:     ▷ find mininum freq for current transaction
 3:     for each f_new in F, in increasing order do
 4:         if μ̂(t_0, f_new) − e_0 ≤ d(t_i) then
 5:             break
 6:         end if
 7:     end for
 8:     ▷ ensure all queued transactions finish in time
 9:     for each t in Q, in EDF order do
10:         if q̂(t, f_new) + μ̂(t, f_new) ≤ d(t) then
11:             continue
12:         end if
13:         ▷ f_new is not fast enough for t
14:         ▷ find the lowest higher frequency that is
15:         for each f ∈ F | f > f_new, in increasing order do
16:             f_new ← f
17:             if q̂(t, f) + μ̂(t, f) ≤ d(t) then
18:                 break
19:             end if
20:         end for
21:         ▷ no further checking once we need highest freq
22:         if f_new = maximum frequency in F then
23:             set processor frequency to f_new
24:             return
25:         end if
26:     end for
27:     set processor frequency to f_new
28:     return
29: end function
```

Figure 2: POLARIS Processor Frequency Selection

for any inaccuracy in its predictions. We defer discussion of these issues to Section 6.

In Figure 2, $\hat{q}(t, f)$ represents the total estimated queueing time for transaction $t \in \mathcal{Q}$, assuming that the processor runs at frequency f. This is defined as follows:

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

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.

5. POLARIS ANALYSIS

In this section we analyze the performance of POLARIS through a competitive analysis against two existing algorithms YDS [52] (Section 5.2) and OA [8, 52] (Section 5.3). 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, 8, 52] 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 5.1.

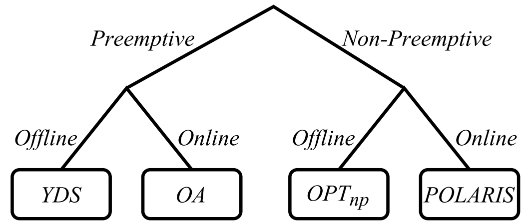

Figure 3: Energy aware scheduling algorithms.

Broadly, energy aware scheduling algorithms can be classified into four categories along two dimensions as shown in Figure 3: (1) preemptive vs non-preemptive; and (2) offline vs online algorithms. Intuitively, offline preemptive algorithms are the most computationally powerful algorithms. The YDS algorithm [52] is the optimal, i.e., least energy consuming, 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 5.4 and 5.5, 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 algorithms that maximize the throughput [6] or minimize 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. Several preemptive algorithms other than YDS and OA have also been proposed for various speed-scaling problems. We refer the reader to references [3] and [17] for a survey of these algorithms.

5.1 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 workload $w(t)$. The workload of a transaction represents the amount of work that it must perform, which is assumed to be known accurately. Algorithms can scale the speed of the processor to arbitrarily high levels, and the time required to execute a transaction t is assumed to be $w(t)/f$, where f the processor frequency (speed). The power consumption of the processor is assumed to be f^{α}, where $\alpha > 1$ is a constant [11]. The assumption that $\alpha > 1$ guarantees the convexity of the power-speed function, i.e., the faster the speed, the more power the processor uses per unit of work that it performs. We observe that under 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' workloads accurately; and (b) can pick

arbitrarily high speeds to finish any transaction on time.

5.2 Yao-Demers-Schenker (YDS)

The optimal offline preemptive algorithm YDS works as follows. 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 such that $[a(t), d(t))$ 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 new *reduced* problem instance P', and repeats the previous step on P'. P' is the same as P except that the interval CI is removed from the timeline. Specifically, 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 the resume executing t after CI.

5.3 Optimal Available (OA)

OA is an online preemptive algorithm based on YDS [52]. Each time a new transaction arrives, OA uses YDS to choose a schedule for all transactions currently in the system, including the new transaction, the currently running transaction (if any), and any other transactions that are waiting to run. It then runs transactions using that schedule until a new transaction arrives, at which point it again reschedules all transactions in the system using YDS. Bansal et al showed that OA is α^α competitive against YDS [8].

Suppose that a new transaction arrives in the system at time τ. OA schedules the transactions in the system by running YDS on a problem instance consisting of the following transactions:

- The newly arrived transaction, t_{new}.
- The currently running transaction, t_r, with its workload $w(t_r)$ taken to be the *remaining* workload 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_0 or t_{new}. Thus, if $d(t_{new}) < d(t_0)$, OA will *preempt* t_0 and start running t_{new}. If, on the other hand, $d(t_0) < d(t_{new})$, t_0 will continue running after t_{new}'s arrival, and t_{new} will run later.

5.4 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 5.1. *Suppose that both POLARIS and OA have the same queue at a point in time, with k total transactions, one running t_r and the rest waiting, with the exact same workloads. Suppose a new transaction t_{new} arrives, and that $d(t_r) \leq d(t_{new})$. Until the arrival of the next transaction, POLARIS and OA will execute transactions in the same order, and with the same processor frequency.*

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

Second, we consider processor speed. Let CI_i represent the ith critical interval chosen by OA. Let P_1 represent the original problem instance considered by OA, and let P_i represent the reduced problem instance under which $CI_i (i > 1)$ is chosen. Since both algorithms agree on EDF execution order, we show by induction on the number of transactions that POLARIS and OA agree on the processor speed used to execute each transaction.

Base Case: Consider the transaction with the earliest deadline in the original, non-reduced problem instance, P_1. OA will run this transaction first, using frequency $den(CI_1)$. Now consider POLARIS. When t_{new} arrives, POLARIS will use SETPROCESSORFREQ (Figure 2) to set the processor frequency. SETPROCESSORFREQ iterates over the transactions present in the system, including t_r and t_{new}. After iterating over all $k+1$ transactions in the system, the selected frequency will be

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

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

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

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

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

Next, we consider the situation in which the newly arriving transaction t_{new} has an earlier deadline than the running transaction t_r. In such a situation, OA will *preempt* t_r and start running t_{new}. This is the most power-efficient way to execute the current transactions. POLARIS , which is non-premptive, 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 5.2. *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 workloads. Suppose a new transaction t_{new} arrives,and that $d(t_{new}) < d(t_r)$. Until the arrival of the next transaction, POLARIS will execute transactions in the same order, and with the same processor frequency, as OA would have if $d(t_r)$ were decreased to $d(t_{new})$.*

PROOF. The proof is similar to that of Lemma 5.1, so we provide a sketch. 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 SETPROCESSORFREQUENCY 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. □

In general, POLARIS may execute t_{new} and t_r at higher frequency than would OA (on the unmodified instance), but all other waiting transactions will run at a frequency no higher than they would have run at under OA.

5.5 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 workloads, arrival times, and deadlines. Agreeable instances are those in which transactions have arbitrary workloads 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 $d(t_i) \leq d(t_j)$. That is there is no transaction t_i that is "squeezed" between the arrival time and deadline of another transaction 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 5.3. *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 5.3 that the only difference in the behaviors of OA and POLARIS is when a new transaction with the earliest deadline in the queue arrives. Note that in agreeable instances this never happens. Therefore, in agreeable instances, POLARIS behaves exactly the same as OA. Since OA has a competitive ratio of α^{α} with respect to YDS [8], so does POLARIS. □

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 workloads 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 workloads by a factor of c.

Our analysis consists of two steps. First, we establish a relationship between the power consumed by POLARIS on instance P and the power consumed by YDS on instance P'.

THEOREM 5.4. $Pow[POlARIS(P)] \leq \alpha^{\alpha} Pow[YDS(P')]$

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 [8]. The proof can be found in the Appendix. □

We next show that YDS on P' consumes exactly c^α times the power it does on P, proving that $Pow[POLARIS(P)] \leq (c\alpha)^\alpha Pow[YDS(P)]$. This shows that POLARIS has a $(c\alpha)^\alpha$ competitive ratio against YDS and therefore OPT_{np}.

THEOREM 5.5. $Pow[YDS(P')] = c^\alpha Pow[YDS(P)]$.

PROOF. Since we are increasing the workload of each transaction by a factor of c, YDS on P' will find exactly the same set of critical intervals, except the density of each critical interval will be a factor of c larger. This implies that at any time YDS' processor speed on P' will be a factor c faster than on P. 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. $\square$

The next corollary is immediate from Theorems 5.4 and 5.5.

COROLLARY 5.6. *POLARIS has a $(c\alpha)^\alpha$ competitive ratio against YDS and therefore OPT_{np}.*

5.6 Discussion

The competitive ratio in Corollary 5.6 has two components: α^α and c^α. One interpretation of this result is 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 [8]. Thus, one interpretation we can give is that the α^α component captures POLARIS' disadvantage of not knowing the future. In contrast, the c^α component captures the POLARIS' disadvantage of not being able to preempt. Although we do not know if this competitive analysis is tight, it is easy to construct problem instances in which POLARIS' power consumption is either α^α or c^α worse than YDS. Bansal et. al. have given an agreeable problem instance in which OA performs α^α worse than YDS [8]. Since POLARIS performs exactly the same as OA on agreeable instances, POLARIS will also perform α^α worse than YDS on this instance. For an informal instance capturing the second component, consider a two transaction workload where t_1 has workload w_{max} and arrives at time 0 and has a very late deadline. t_2 has a workload w_{min} and arrives after an infinitesimally small amount of time after 0, and has a very short deadline. This is the worst situation for POLARIS, where POLARIS will start t_1 but after immediately seeing t_2, because it cannot preempt t_1, will try to finish both t_1 and t_2 by the deadline of t_2. Whereas 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 on this instance.

6. POLARIS IMPLEMENTATION

We implemented POLARIS, along with several baseline algorithms, in Shore-MT [24]. In this section, we present an overview of the implementation and also describe some of the practical problems that it addresses.

6.1 Shore-MT

Shore-MT is a multi-threaded data storage manager which is designed for multiprocessors. Shore-Kits [1] provides a front-end driver for Shore-MT. It includes implementations of several database benchmarks, including TPC-C. For the remainder of the paper, we refer to the combination of Shore-Kits and Shore-MT as Shore-MT.

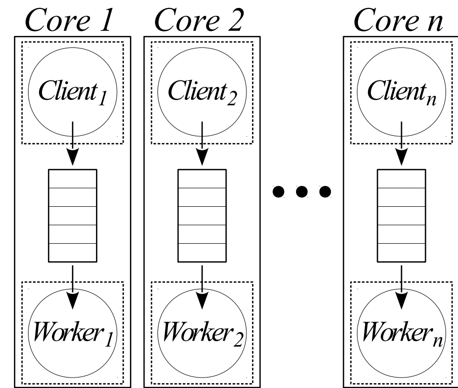

Figure 4: Clients, workers and request queues in the POLARIS Shore-MT prototype. Each client/worker pair is pinned to a seperate core.

Shore-MT has multiple worker threads, each with an associated request queue. In the TPC-C implementation, each request corresponds to a TPC-C transaction of a particular type, such as NewOrder. Each worker sequentially executes requests from its queue, using the storage manager to access data. There are also client threads. In our prototype setting, these clients play dual roles. First, they simulate actual remote database system clients by generating sequences of transaction requests. Second, they play the role of a server-side connection listener by enqueuing those transaction requests into the worker queues.

The POLARIS algorithm is designed to control the processor execution speed for a single transaction request queue, with requests executed non-premptively. This is the execution model followed by each individual Shore-MT worker thread. The multi-core CPUs in our test servers allow CPU frequency to be controlled separately for each core. Thus, we fixed the number of workers to match the number of cores in our server, and pinned each worker to a single core. We also fixed the number of clients to match the number of cores, and each client submits requests to a single worker's queue. Each worker/client pair then uses POLARIS to control the execution frequency of its assigned core. Figure 4 gives a high-level view of the way we place Shore-MT threads in multi-core CPUs.

POLARIS requires action when two types of events occur: arrival of a new transaction request, and completion of a request. In our prototype, arrival work is handled by the client threads, in their role as server-side request receivers. When a new request arrives, the client enqueues the request and then runs the POLARIS SETPROCESSORFREQ algorithm (Figure 2) to set the execution frequency of its own core. We modified Shore-MT's request queues so that requests are queued in EDF order, as required by POLARIS. Completion work 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 the their core's frequency before executing the dequeued request.

6.2 Controlling CPU Frequency

There are different mechanisms for controlling CPU core frequencies. For x86 processors, all of the alternatives ultimately rely on Model Specific Registers (MSRs) [2, 23]. 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 possible and 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 introduced substantial latency, as was previously observed by Wamhoff et al. [47]. Since POLARIS adjusts execution frequencies on a short time scale (potentially on each transaction request arrival or completion), clients and workers in our prototype modify the MSRs directly via the MSR driver, which is much faster.

6.3 Execution Time Estimation

POLARIS requires estimates of the execution time ($\hat{\mu}(c, f)$) for transactions of each class c at each possible processor execution frequency f. In general, accurate estimation of request execution times is not easy [20]. However, as estimation was not our primary focus, we took a relatively simple approach in our prototype. Specifically, POLARIS tracks moving means and standard deviations of measured transaction execution times for each combination of transaction type and frequency. It then predicts execution times using

$$\hat{\mu}(c, f) = \alpha\mu_{meas}(c, f) + \beta\sigma_{meas}(c, f)$$

where $\mu_{meas}(c, f)$ and $\sigma_{meas}(c, f)$ are the measured mean and standard deviation and α and β are parameters that control how conservative POLARIS' estimates are.

Because POLARIS adjusts processor frequency every time a transaction arrives or finishes execution, it has some built-in robustness against estimation errors. For example, if POLARIS underestimates the execution time of a transaction, it has the opportunity to increase processor frequency when that transaction finishes, if that is necessary to ensure that the remaining transactions in the queue will finish in time. Nonetheless, underestimating $\hat{\mu}(c, f)$ can cause transactions to miss their deadlines. Overestimates, in contrast, may result in POLARIS choosing unnecessarily high processor frequencies, which increases power consumption. We experimented with a variety of values of α and β for our TPC-C test workload. For all of the experiments reported in the this paper, we used $\alpha = 2$ and $\beta = 0$.

7. EVALUATION

Next, we present an empirical evaluation of POLARIS. The primary goal of our evaluation is to compare POLARIS against operating-system-based governors, which serve as our baselines. In addition, we also compare POLARIS against two in-DBMS baselines to provide some insight into POLARIS' performance.

7.1 Methodology

All of our experiments use the Shore-Kits TPC-C benchmark implementation, with scale factor 16. 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 20000 transactions, (2) a short *training* phase (20000 transactions per worker) for warming up POLARIS' execution time estimation model, and (3) the *test* phase, during which power consumption and system performance are measured. We changed the TPC-C implementation in Shore-Kits from a closed-loop design to an open-loop design, so that we can set a fixed offered load (transaction requests per second) for the system for each experiment. We ran experiments at three load levels: high, medium, and low. High load is 90% of the estimated peak throughput for our test system, which is about 28000 transactions per second. The medium and low loads correspond to 60% and 30% of the peak throughput, respectively.

Request Type	**Execution Time (μs) @2.8 GHz Mean**	**StDev**	**Execution Time (μs) @1.2 GHz Mean**	**StDev**
New Order (45%)	782	964	1698	1802
Payment (47%)	152	465	333	827
Order Status (4%)	222	492	463	968
Stock Level (4%)	2679	753	5790	1384

Figure 5: TPC-C transaction execution times at maximum and minimum CPU frequency. Percentages indicate the transaction type mix in the workload.

Figure 5 illustrates the characteristics of our TPC-C workload, as measured by the execution time estimator in our prototype. The figure shows execution time of each transaction type at the highest (2.8 GHz) and lowest (1.2 GHz) CPU frequencies for our server. The workload is not simple. Execution times of different transaction types vary by an order of magnitude, and there is considerable execution time variance among transactions of the same class. The figure also illustrates the range of control offered by DVFS on our server: moving from the lowest frequency to the highest reduces execution times by about a factor of two.

We modified the TPC-C request generator to associate a deadline with each request. Deadlines for transactions of class c (e.g, NewOrder) are set to be S times the mean execution time of class c transactions at the highest processor frequency, as shown in Figure 5. The deadline slack, S, is an experimental parameter which controls the tightness of the deadlines. We experiment with slack in the range from $S = 10$ to $S = 100$. Note that since mean transaction execution times are low, deadlines are tight, even at the highest slack we tested. For example, NewOrder transactions have a mean execution time of about 780 microseconds at the highest frequency (Figure 5). Thus, the deadlines from NewOrder transactions range from 7.8 - 78 milliseconds as S varies from 10 to 100.

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 [22] wall socket power meter, which has a rated ±1.5% accuracy. We measured the power consumption in one-second intervals (the finest

granularity of the power meter) and averaged those over the test duration. We also measured 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, system throughput (load) 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 ran experiments with POLARIS and with several in-DBMS and in-kernel baselines:

Static Frequencies: In these tests, Shore-MT used its default transaction scheduling and did not control core frequencies. Instead, we used the Linux `cpufreq` static governors to set all cores to run at a fixed frequency.

Dynamic Kernel Governors: In these tests, Shore-MT used its default transaction scheduling and did not control core frequencies. We used the Linux `cpufreq` dynamic governors to manage core frequencies. We experimented with two dynamic governors: *conservative* and *On-Demand*. The former favors performance over power savings, while the latter adjusts core frequencies more aggressively to save power.

In-DBMS Dynamic Governors: In addition to POLARIS, we considered two other in-DBMS power governors. The first is *LAPS* [27], which we proposed in earlier, preliminary work. Like POLARIS, LAPS sets core frequencies to meet transaction deadlines, but unlike POLARIS it uses FIFO scheduling. The second, *LAPS Arrival*, is a variant of LAPS that is similar to Rubik [26]. Unlike LAPS, which adjusts processor frequencies only on transaction completion, LAPS Arrival adjusts frequencies both on transaction completion and arrival. Comparing POLARIS to Rubik helps to isolate the effect of transaction scheduling on overall performance, while comparing LAPS and LAPS Arrival helps to isolate the effect of adjusting core frequencies on transaction arrival.

In our experiments, we use a server with a 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, we disabled the CPU ACPI software control in the BIOS configuration to prevent the `cpufreq` driver from interfering with power control. For the experiments using the static and dynamic kernel governors, we enabled ACPI software control in the BIOS. To reduce non-uniform memory access (NUMA) effects and get more homogeneous memory access patterns, we enabled 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

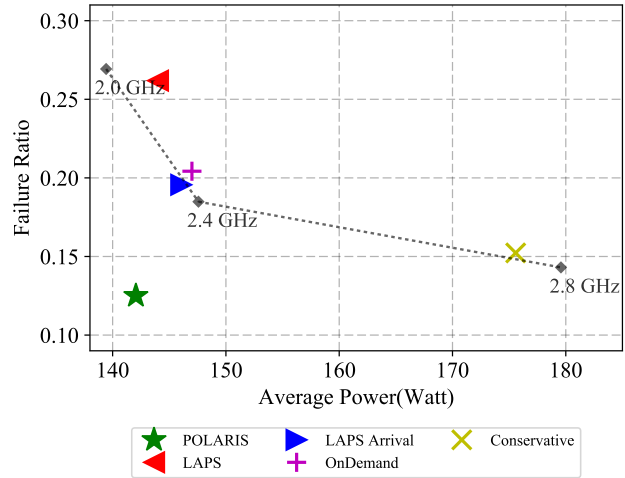

Figure 6: Performance and power of different power management schemes with tight deadlines ($S = 10$) under medium load

for POLARIS and the two other dynamic power management algorithms, LAPS and LAPS Arrival.

For all of our experiments, our Shore-MT prototype is configured to use sixteen client threads and sixteen worker threads. Each client generates requests for a single worker, and we pinned each client/worker pair to a single physical core, with the client on one hyper-thread and the worker on the other. To minimize database contention in these experiments, each client generates requests for a separate TPC-C warehouse.

7.2 Experimental Results

Figure 6 summarizes our results for our default scenario: medium load, and deadline slack $S = 10$. The figure shows the average power consumption and failure rate for POLARIS as well as the in-DBMS and in-kernel dynamic baselines. For reference, we also show the power consumption of the three highest static frequencies (2.0-2.8 GHz). The two lowest static frequencies result in very high failure rates at this load level, and are not shown. The dashed line connecting the static frequency points represents the Pareto frontier for power management techniques that control processor frequency without changing the default FIFO transaction execution order.

With $S = 10$ at this load level, almost 15% of transactions miss their deadlines even if the processor remains at the highest frequency (2.8 GHz). The kernel's Conservative power governor leaves the cores at the highest frequency most of the time, and hence its behavior is very close to that of the high frequency static governor. In contrast, the more aggressive OnDemand governor reduces power consumption by almost 35 watts relative to 2.8 GHz. However, it does this at the expense of the transaction failure rate, which rises by about one third, to more than 20%. The kernel governors are unaware of transaction deadlines and transaction request queues, and base their frequency decisions on CPU utilization alone.

POLARIS results in even greater power savings than OnDemand (close to 40 watts relative to 2.8 GHz), with no

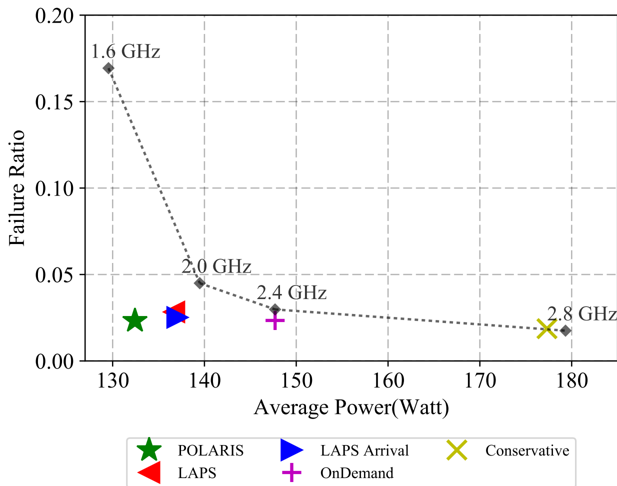

Figure 7: Performance and power of different power management schemes with loose deadlines ($S = 100$) under medium load

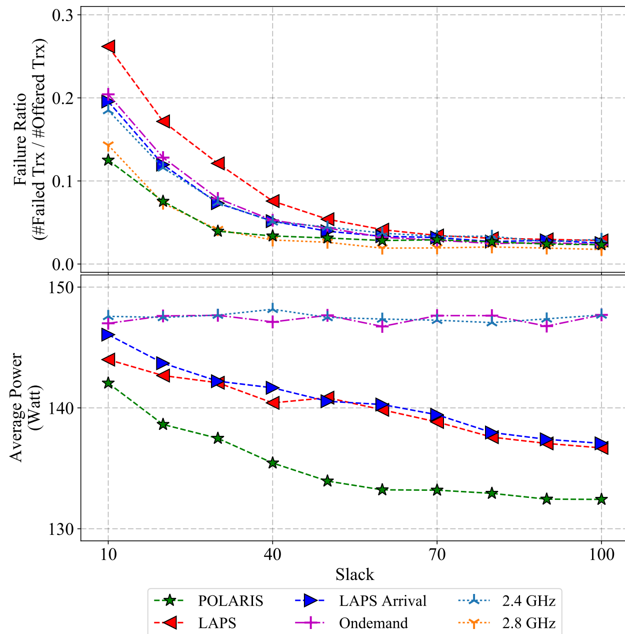

Figure 8: Performance and power of different power management schemes under medium load, as functions of slack (S).

increase in the transaction failure rate. Indeed, the transaction failure rate under POLARIS is actually somewhat *lower* than that achieved by the 2.8 GHz static governor. This is possible because POLARIS uses EDF transaction scheduling, in addition to setting the processor execution frequency. Under this medium workload, the difference in power consumption between the highest and lowest static frequencies is approximately 60 watts. This range represents the maximum opportunity for power savings from frequency scaling under this workload. POLARIS is able to capture about 2/3 of this opportunity, with no increase in the transaction failure rate.

Figure 6 also shows that POLARIS results in slightly lower power consumption and much lower transaction failure rates than the two in-DBMS baselines. LAPS results in more missed deadlines than LAPS Arrival because it adjusts execution frequency only when transactions commit, while LAPS Arrival adjusts on both completion and arrival of transactions. Interestingly, LAPS' inability to adjust execution frequency upwards when new transactions arrive means that it consumes slightly less power than LAPS Arrival. However, the price (in missed transaction deadlines) for this improvement is very high. POLARIS' abilty to re-order transactions by deadline results in an even larger drop in the failure rate relative LAPS Arrival, with no corresponding increase in power consumption.

7.2.1 Effect of Deadline Slack

Next, we consider the effect of deadline slack. We repeated the medium-load experiment, but this time increased slack by a factor of ten, to $S = 100$. Figure 7 shows the power consumption and failure rates for POLARIS and the baselines under this higher-slack scenario.

We observe that POLARIS and all baselines, even those that are not deadline-aware, have lower failure rates in this case. This is simply because transaction deadlines have been relaxed. Power consumption under the in-kernel dynamic governors is unaffected by the higher slack (compare to Figure 6), since they are unaware of transaction deadlines. However, both POLARIS and the in-DBMS baselines produce greater power savings when slack is higher. Both LAPS and LAPS Arrival reduce power consumption by more than 40 watts relative to 2.8 GHz, while POLARIS' reduction approaches 50 watts.

Figure 8 summarizes the effect of slack by showing power consumption and transaction failure rate as a functions of slack, for the in-kernel OnDemand governor and the three in-DBMS governors. (We do not show the kernel's Conservative governor, which behaves like the OnDemand governor but with significantly higher power consumption.) While all techniques have low failure rates at high slack, POLARIS results in lower failure rates at low slack because of its extra transaction scheduling flexibility. The figures also show that the power "gap" between the in-DBMS and kernel governors increases with slack.

Interestingly, POLARIS maintains a power advantage over the other in-DBMS governors even at high slack. This indicates that the *combination* of deadline-aware scheduling and deadline-aware frequency scaling can result in greater power savings than deadline-aware frequency scaling alone. Figure 9 shows a three-transaction example which illustrates how this can happen. Each rectangle represents a transaction request. Rectangle height indicates the amount of work required to complete the request, and width indicates the request deadline. We assume that the large transaction arrives second. The scenario on the left shows FIFO ordering, and the slope of the dashed line represents the execution frequency chosen by a deadline-aware algorithm. The scenario on the right shows earliest deadline execution of the same transactions, and (reduced) execution frequency that results.

7.2.2 Effect of Load

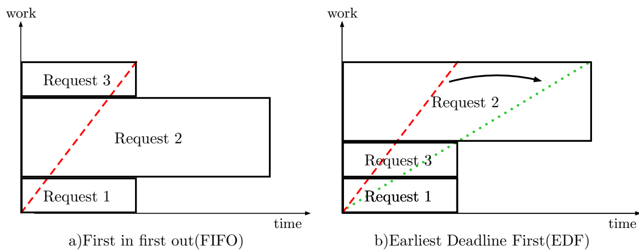

Figure 9: An example illustrating the impact of FIFO vs EDF scheduling on frequency selection.

So far, we have considered only the medium load scenario. Figures 10 and 11 show power consumption and performance under high and low load, respectively, as a function of slack. In both figures, we show only one of the two in-kernel baselines (OnDemand), as it is the most competitive with the in-DBMS techniques in its power/performance tradeoff. In addition, we show only a few of the fixed-frequency baselines in each figure to reduce clutter.

The key observation from these figures is that POLARIS is almost never worse than any of the baselines in terms of power consumption and transaction performance. The only exception is at high load with low slack, where the in-kernel OnDemand governor results in marginally lower power consumption than POLARIS - though at the cost of a substantially higher failure rate.

Comparison of Figures 10 and 11 with Figure 8 show the same general trends in all cases. The power savings achieved by all of the in-DBMS techniques improves with increasing slack. In addition, POLARIS' advantage over the in-DBMS baselines in terms of failure rate is greater when slack is tight, since POLARIS has the flexibility to re-order transactions. At high load, POLARIS reduces power by 20-25 watts relative to the 2.8 GHz static governor, which is the only baseline with failure rates comparable to POLARIS'. At low load, the differences in power consumption among the various dynamic governors are relatively small - less than 10 watts, even with high slack.

Although POLARIS dominates the baselines at almost all load and slack levels we tested, our results show that the sweet spot for power savings is at medium load. This is good, since production systems are often provisioned to operate normally at with utilizations that are neither too high nor too low.

7.2.3 Energy Efficiency

So far, we have used two metrics (power and transaction failure rate) to evaluate the behavior of POLARIS and the baselines. Figure 12 summarizes and restates these results using a single, combined efficiency metric that reflects both power consumption and transaction latencies. This metric is the mean number of *successful* transactions completed per joule of energy consumed. Under this metric, the energy consumed by transactions that miss their latency targets is considered to have been wasted.

Figure 12 shows that POLARIS is at least as energy efficient as all of the baselines at all load levels and all slack levels. The figure shows that energy efficiency is higher at higher load levels. This is a consequence of the fact that processors are not power proportional, and it has been noted by other researchers [9]. It can also be observed in

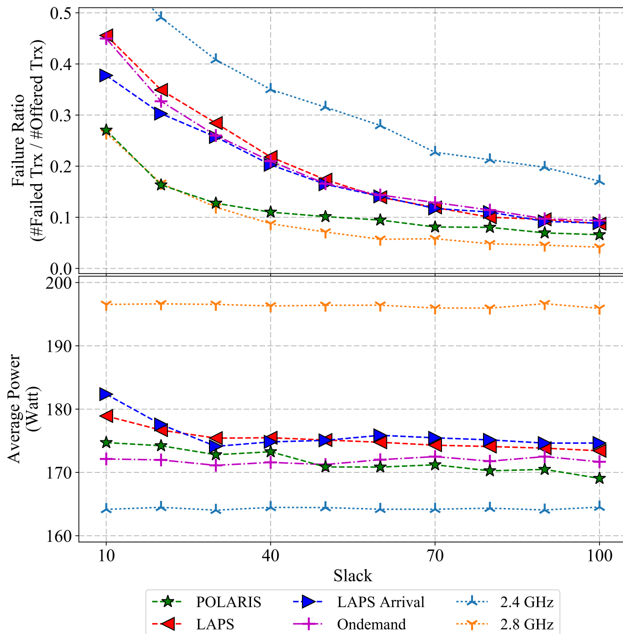

Figure 10: Performance and power of different power management schemes under high load, as functions of slack (S).

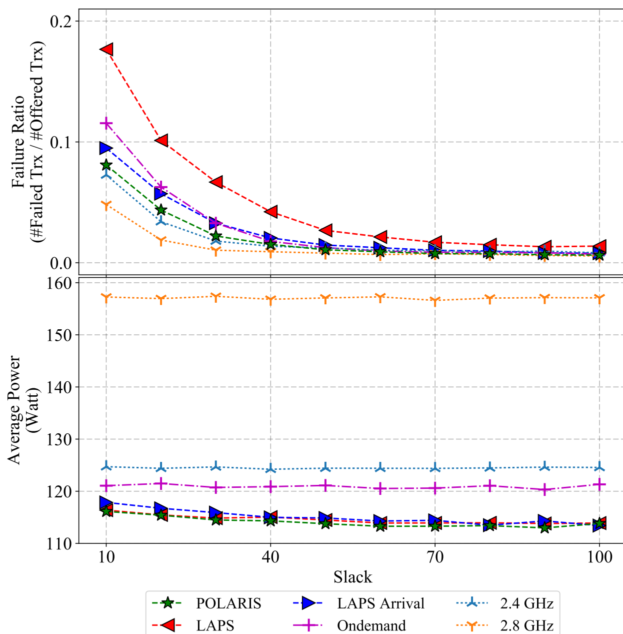

Figure 11: Performance and power of different power management schemes under low load, as functions of slack (S).

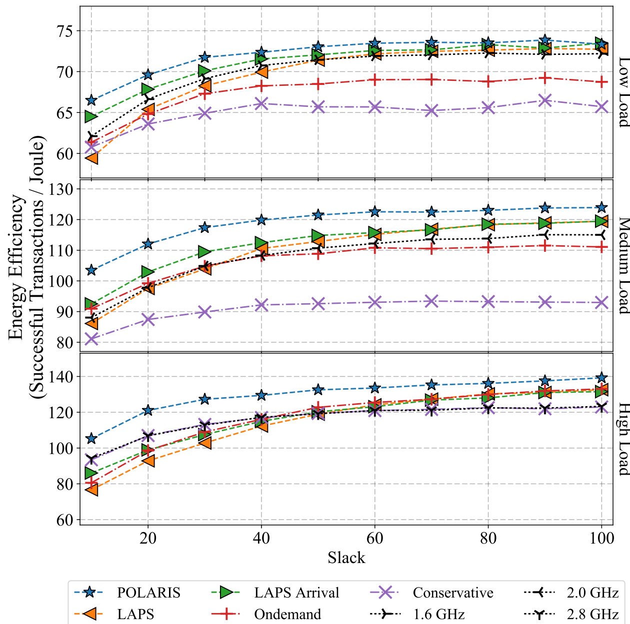

Figure 12: Successful transactions per joule, as a function of load and slack.

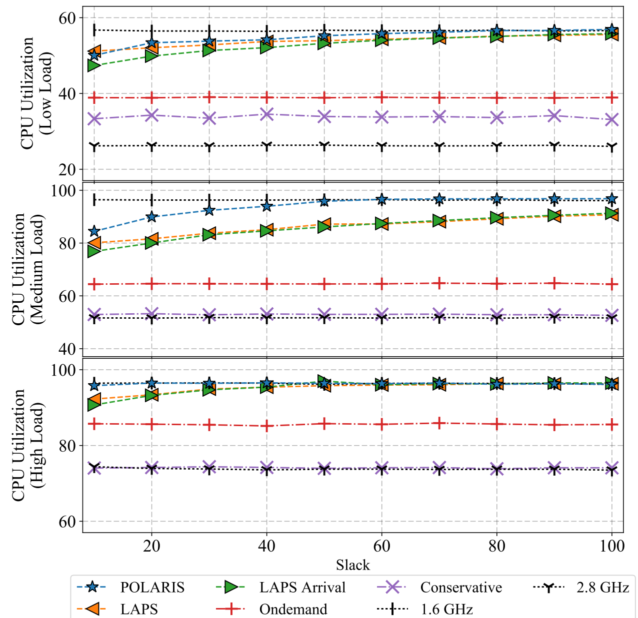

Figure 13: CPU Utilization under various load levels

SPECpower_ssj2008 server benchmark results [42]. The figure also shows that POLARIS' efficiency advantage over the baselines is greatest at medium loads, and at tight (low) slack levels. Medium loads allow POLARIS to utilize the entire dynamic power range of the processor. At low loads, both POLARIS and the baselines are limited in their ability to improve efficiency by the processor's lower bound on execution speed. The maximum speed of the processor is similarly limiting when load is high.

7.2.4 CPU Utilization

Another way to think of processor frequency scaling is as a kind of fast, fine-grained capacity provisioning mechanism. Increasing the processor frequency increases its capacity to do work, at a cost of increased power consumption. Reducing frequency reduces capacity. From this perspective, the role of frequency scaling algorithms, like POLARIS and the baselines, is to reduce processor capacity as much as possible without causing transactions to miss their latency targets.

By measuring CPU utilization, we can quantify algorithms' success at adjusting processor capacity. Ideally, with a perfect frequency scaling technique and processor with a wide range of possible frequencies, we would see CPU utilization approaching 100%. Figure 13 reports the actual CPU utilization we observed for POLARIS and the baseline algorithms at all load and slack levels. At low load, the utilization of the processor is less than 60% even at the lowest processor frequency (1.2 GHz). Higher utilizations (and greater power savings) would require the ability to reduce execution frequency below 1.2 GHz. With sufficient slack, POLARIS and the in-DBMS baselines are able to approach this limit, indicating that they are achieving the maximum possible capacity (and hence power) reductions on this processor. In contrast, the in-kernel baselines have lower utilizations. At high load, the figure shows that all of the frequency scaling algorithms have little room to maneuver, as processor utilization is barely below 80% even at peak frequency. Medium load, however, allows plenty of room for capacity adjustment, with processor utilization varying from about 50% at the highest frequency to almost 100% at the lowest. All of the in-DBMS algorithms, including POLARIS, are much more effective than the in-kernel baselines at driving down frequency and increasing utilization.

8. CONCLUSION

In this paper, we have presented a power-aware transaction scheduling technique for transactional database systems, and relate it to other well-known off-line and on-line algorithms. Unlike operating system power governors, POLARIS is aware of per-transaction latency targets and takes advantage of them to keep processor execution frequency, and hence power consumption, as low as possible. On our server, POLARIS was able to reduce power consumption by almost 50 watts 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 comparision with other in-DBMS baselines, we showed that it is necessary for POLARIS to control transaction execution order *and* processor frequency to achieve this performance.

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 despite noisy estimates. Thus, perfect estimation is not necessary for in-DBMS power-aware scheduling to be effective. However, better estimates would allow POLARIS to reduce processor frequency more aggressively, and further reduce power.

9. REFERENCES

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

[2] Advanced Micro Devices(AMD). Architecture programmers manual: Volume 2: System programming. (24593), 2017.

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

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

[5] H. Amur, R. Nathuji, M. Ghosh, K. Schwan, and H.-H. S. Lee. Idlepower: Application-aware management of processor idle states. In *Proceedings of the Workshop on Managed Many-Core Systems, MMCS*, volume 8, 2008.

[6] E. Angel, E. Bampis, V. Chau, and N. K. Thang. Throughput maximization in multiprocessor speed-scaling. *Theoretical Computer Science*, 630:1–12, 2016.

[7] A. Antoniadis and C.-C. Huang. Non-preemptive speed scaling. *Journal of Scheduling*, 16(4):385–394, 2013.

[8] N. Bansal, T. Kimbrel, and K. Pruhs. Speed scaling to manage energy and temperature. *Journal of the ACM (JACM)*, 54(1):3, 2007.

[9] L. A. Barroso, J. Clidaras, and U. Hölzle. The datacenter as a computer: An introduction to the design of warehouse-scale machines. *Synthesis lectures on computer architecture*, 8(3):1–154, 2013.

[10] L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12), 2007.

[11] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, M. Gupta, and P. W. Cook. Power-aware microarchitecture: design and modeling challenges for next-generation microprocessors. *IEEE Micro*, 20(6):26–44, 2000.

[12] Y. Chen, S. Alspaugh, and R. Katz. Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads. *Proc. VLDB Endow.*, 5(12):1802–1813, 2012.

[13] M. Dayarathna, Y. Wen, and R. Fan. Data center energy consumption modeling: A survey. *IEEE Communications Surveys Tutorials*, 18(1):732–794, 2016.

[14] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. *SIGPLAN Not.*, 49(4):127–144, 2014.

[15] K. Ebrahimi, G. F. Jones, and A. S. Fleischer. A review of data center cooling technology, operating conditions and the corresponding low-grade waste heat recovery opportunities. *Renewable and Sustainable Energy Reviews*, 31:622–638, 2014.

[16] X. Fan, W.-D. Weber, and L. A. Barroso. Power provisioning for a warehouse-sized computer. *ACM SIGARCH Computer Architecture News*, 35(2):13–23, 2007.

[17] M. E. T. Gerards, J. L. Hurink, and P. K. F. Hlzenspies. A survey of offline algorithms for energy minimization under deadline constraintsa. *Journal of Scheduling*, 19(1):3–19, 2016.

[18] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper. Workload analysis and demand prediction of enterprise data center applications. In *IEEE 10th International Symposium on Workload Characterization (IISWC)*, pages 171–180, 2007.

[19] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel. The cost of a cloud: Research problems in data center networks. *SIGCOMM Comput. Commun. Rev.*, 39(1):68–73, 2008.

[20] H. Hacigumus, Y. Chi, W. Wu, S. Zhu, J. Tatemura, and J. F. Naughton. Predicting query execution time: Are optimizer cost models really unusable? In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 1081–1092, 2013.

[21] J. Hamilton. Cost of power in large-scale data centers. http://perspectives.mvdirona.com/2008/11/cost-of-power-in-large-scale-data-centers/, 2008. Accessed: Feb. 2017.

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

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

[24] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-mt: a scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, pages 24–35, 2009.

[25] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks. Tradeoffs between power management and tail latency in warehouse-scale applications. In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 31–40. IEEE, 2014.

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

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

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

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

[30] G. V. Laszewski, L. Wang, A. J. Younge, and X. He. Power-aware scheduling of virtual machines in dvfs-enabled clusters, in. In *Proc. IEEE Intl Conf. Cluster Computing*, pages 1–10, 2009.

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

[32] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, pages 301–312. IEEE Press.

[33] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis. Towards energy proportionality for large-scale latency-critical workloads. *ACM SIGARCH Computer Architecture News*, 42(3):301–312, 2014.

[34] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz. Towards energy-proportional datacenter memory with mobile dram. *SIGARCH Comput. Archit. News*, 40(3):37–48, 2012.

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

[36] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch. Power management of online data-intensive services. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 319–330. IEEE, 2011.

[37] I. Psaroudakis, T. Kissinger, D. Porobic, T. Ilsche, E. Liarou, P. Tzn, A. Ailamaki, and W. Lehner. Dynamic fine-grained scheduling for energy-efficient main-memory queries. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, pages 1:1–1:7. ACM, 2014.

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

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

[40] A. Shehabi, S. Smith, N. Horner, I. Azevedo, R. Brown, J. Koomey, E. Masanet, D. Sartor, M. Herrlin, and W. Lintner. United states data center energy usage report, 2016.

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

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

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

[44] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 231–242. ACM, 2010.

[45] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[46] Unified EFI Inc. Advanced configuration and power interface specification. `http://www.uefi.org/sites/default/files/resources/ACPI_6_1.pdf`, 2016. Accessed: Feb. 2017.

[47] J.-T. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber, and D. Dice. The TURBO diaries: Application-controlled frequency scaling explained. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, pages 193–204, 2014.

[48] Q. Wu, Q. Deng, L. Ganesh, C.-H. Hsu, Y. Jin, S. Kumar, B. Li, J. Meza, and Y. J. Song. Dynamo: facebook's data center-wide power management system. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 469–480. IEEE, 2016.

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

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

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

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

APPENDIX

Here we present the proof of Theorem 5.4. Instances P and P' are as defined in Section 5.5.

THEOREM 5.4 $Pow[POLARIS(P)] \leq \alpha^{\alpha} Pow[YDS(P')]$.

PROOF. We assume w.l.o.g., that P and therefore P' are contiguous. In other words, for each time $t \in [0, d(t_n) = d(t'_n)]$ there is a transaction t_j, such that $a(t_j) \leq t \leq d(t_j)$. If the P and P' are not contiguous, we can break it into a finite number of contiguous parts and analyze POLARIS competitiveness in each part and get the same result. We let $s_D(t)$ and $s_Y(t)$ the speed of POLARIS' and YDS' 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 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:

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

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

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

.

We next define $\phi(t)$ and prove that all three conditions hold. Let $s_{Pna}(t)$ (for **P**OLARIS **n**o **a**rrival) denote the speed at which POLARIS would be executing if no new tasks were to arrive after the current time. By Lemma 5.1 we proved that when no tasks arrive POLARIS 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 5.1). Throughout the proof we denote the current time always as t_0. Let $CI_1, ..., CI_k$ be POLARIS' current critical intervals (note that k will change over time) and let t_i be the end of critical interval CI_i. Let $w_D(t, t')$ 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 $s_{Pna}(t) = den(CI_i) = \frac{w_D(t_i, t_{i+1})}{t_{i+1} - t_i}$. 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 [8] for a formal proof of this observation (proved for OA).

The potential function we use is the following:

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

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_D 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 [8] 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:

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

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 $w_D(t_0, t_1)$ will decrease at rate s_{Pna} and other $w_D(t_i, t_{i+1})$ remain unchanged. YDS will be running one transaction t_{YDS} at speed $s_Y(t_0)$. W.l.o.g., let d_{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:

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

Let $z = \frac{s_{Pna}(t_0)}{s_Y(t_0)}$. Note we assumed w.l.o.g. that P and P' are contiguous 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 [8] for the full derivation.

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

i-Completion of a transaction by YDS and POLARIS: This part of the analysis is the same as the proof in reference [8] Notice that the completion of a transaction by YDS has no effect on the $s_{Pna}(t_i)$, $w_D(t_i, t_{i+1})$, and $w_Y(t_i, t_{i+1})$ for all i, so does not increase $\phi(t)$. Similarly the completion of an transaction by POLARIS have no affect on $s_{Pna}(t_i)$, $w_D(t_i, t_{i+1})$, 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.

ii-Arrival of a new transaction: Suppose a new transaction t_{new} arrives to POLARIS and t'_{new} arrives to YDS'

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 [8] 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 [8]. Second transactions added to POLARIS' queue and YDS' queue are different. The proof in reference [8] 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 [8]. Below we slightly simplify the proof in reference [8].

Case 1: t_{new} does not have the earliest deadline: Note that t_{new} may change the POLARIS' 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' critical interval CI_j from $\frac{w_D(t_j,t_{j+1})}{(t_{j+1}-t_j)}$ to $\frac{w_D(t_j,t_{j+1}+x)}{(t_{j+1}-t_j)}$ 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:

(a) Interval CI_j splits into two critical intervals with the same increased density of CI_j..

(b) 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 [8] . 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 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 $\frac{w_D(t_j,t_{j+1})}{(t_{j+1}-t_j)}$ to $\frac{(w_D(t_j,t_{j+1})+x))}{(t_{j+1}-t_j)}$. Thus the potential function changes as follows:

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

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

Let $q = w_D(t_j,t_{j+1})$, $\delta = x$ 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 [8] 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 $w_D(t_j,t_{j+1})$ 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.

Case 2: 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' queue and t'_{new}'s to YDS' queue does not increase $\phi(t)$. We next need to argue that the removal of t_{cur} from POLARIS' 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 $w_D(t_j,t_{j+1})$ and does not change the other $w_D(t_i,t_{i+1})$'s. Similarly it does not change any of $w_Y(t_i,t_{i+1})$ because we are not altering YDS' queue, completing the proof. □

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