Towards Dynamic Green-Sizing for Database Servers

Mustafa Korkmaz
University of Waterloo
mkorkmaz@uwaterloo.ca

Alexey Karyakin
University of Waterloo
a2karyak@uwaterloo.ca

Martin Karsten
University of Waterloo
mkarsten@uwaterloo.ca

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

ABSTRACT

This paper presents two techniques for reducing the power consumed by database servers. Both techniques are intended primarily for transactional workloads on servers with memory-resident databases. The first technique is database-managed dynamic processor voltage and frequency scaling (DVFS). We show that a DBMS can exploit its knowledge of the workload and performance constraints to obtain power savings that are more than twice as large as the power savings achieved when DVFS is managed by the operating system. The second technique is rank-aware memory allocation, the goal of which is to power memory that the database system needs and avoid powering memory it does not need. We present experiments that show rank-aware allocation allows unneeded memory to move to low-power states, reducing memory power consumption.

1. INTRODUCTION

Data centers consume a lot of power. Koomey [20] estimated in 2010 that US data center energy consumption is in the range of 70 - 90 billion kWH/year, or approximately 2% of total US electricity use. Energy also represents a significant share of the cost of operating data centers [8]. Since many applications rely on a backend database, database management systems of various kinds are ubiquitous in data centers. The goal of the work described in this paper is to improve database systems' energy efficiency.

Idleness is the enemy of energy efficiency in software systems, including database systems [32]. Servers are not power proportional, meaning power consumption does not increase in proportion to load. In particular, idle servers consume a substantial fraction of power consumed at peak load. For example, Tsirogiannis et al., in 2010, reported idle power consumption was more than 50% of peak power consumption for a database server [32]. Because of on-going intensive efforts to improve the power efficiencies of servers and their components, this situation is improving. Nonetheless, servers are still far from being power proportional.

In Proc. Int'l Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS), Kohala Coast, Hawai'i, USA. Aug 2015.

To maximize energy efficiency, computing resources should not be lightly utilized. They should be busy, or they should be off. Unfortunately, this is difficult to achieve. Workloads fluctuate, so a server that is busy one minute may not be the next. Systems are typically overprovisioned to ensure acceptable performance during periods of high load.

One way to address this problem is to dynamically adjust the capacity of the server in response to fluctuations in load. The goal is to avoid the negative effects of overprovisioning by ensuring that the server remains as fully utilized as possible as the load fluctuates. In this paper, we explore this approach in the context of memory-intensive database systems, which keep most active data in memory. In particular, we explore two techniques for dynamic adjustment of server capacity in database systems. The first uses dynamic voltage and frequency scaling (DVFS) to adjust processor speeds. The second allows the database system to dynamically adjust memory size (and power consumption) to reflect the characteristics of the workload. Since our goal is to improve energy efficiency while limiting the impact on performance, we refer to these techniques collectively as *green-sizing*.

This paper makes the following contributions:

DVFS for Database Systems: DVFS is widely implemented, and mainstream operating system kernels now routinely adjust processor frequencies based on load. In our work, we consider management of DVFS in a database system, with the goal of quickly adjusting processor capacity to reflect short term load fluctuations. We show that by managing DVFS in the database system, where we have more workload information, DVFS can be more effective than it is when managed by the kernel. Our approach assumes that the DBMS has latency targets for database operations. It manages DVFS to ensure that these targets are met, while maximizing energy efficiency. This technique can be viewed as a kind of fine-grained, dynamic server capacity management: we reduce the capacity of the server (by slowing down the cores) when the load is light, and increase it when the load is higher.

Memory Sizing for Database Systems: Because of uncertainty about the size of the database or the size of its working set (e.g., due to fluctuations over time), database servers are often conservatively overprovisioned with memory. This suggests that it may be possible to reduce energy consumption by dynamically adjusting the amount of memory used by the database system, and reducing or eliminating power consumption by the rest of memory. However, applying this idea in practice is not as simple as it sounds, because it requires a means of aligning memory allocation in the database system to the underlying memory system, so that memory power consumption can also be correlated with the amount of memory the database system uses. We describe some of the challenges associated with doing this, and we present the results of some experiments that characterize the reductions in memory energy consumption that can be achieved by applying this approach to database systems.

Our work on these topics is preliminary, and hence we have focused in this paper on characterizing the potential benefits of both approaches. In addition, we discuss some of the challenges and hazards that will need to be addressed if these techniques are to be used in practice.

2. BACKGROUND AND RELATED WORK

In this section, we provide an overview of energy consumption in processors and memory, as well as a brief survey of techniques for managing energy consumption.

2.1 Processor Power Consumption

The dynamic power consumed by a processor is proportional to its operating frequency and to the square of its voltage. Thus, it is possible to reduce a processor's dynamic power consumption by running it at a lower frequency. In addition to the lower frequency's direct effect on power consumption, running the processor at lower frequency also allows voltage to be reduced, resulting in further power savings. This technique, which implements a tradeoff between performance and power, is known as *dynamic voltage and frequency scaling (DVFS)* when voltages and frequencies are adjusted on-the-fly. Själander, Martonosi and Kaxiras [30] provide an excellent overview of DVFS.

The Advanced Configuration and Power Interface (ACPI) standard codifies voltage and frequency scaling by defining a set of discrete processor states, $P_0, P_1, P_2, \ldots$, where P_0 represents the processor's highest voltage and frequency (and hence highest performance), and each successive $P_i (i > 0)$ represents a lower frequency and voltage setting. The number of processor states, and the specific frequency associated with each state, is processor-specific.

Processors, or individual processor cores, may also spend some of their time idle, not executing instructions. The ACPI standard also defines a set of processor states (C states) corresponding to different degrees of idleness. Deeper idle states generally require lower power. However, the deeper the idle state, the longer it takes to come out of that state and return to normal instruction execution. Ideally, idle cores would require little power even in shallow idle states, but in reality, idle power can be substantial.

2.2 Managing CPU Energy Consumption

As noted by Själander et al. [30], the most common form of DVFS is commercial implementations such as Intel's SpeedStep and AMD's Cool'n'Quiet, controlled at the operating system level. These allow voltage and frequency to be controlled by the kernel, through control of processor P-states. The granularity of control ranges from the entire processor to individual cores, depending on the processor.

DVFS is supported in mainstream operating systems. In particular, the Linux kernel implements a variety of DVFS governors, which have different performance and power management objectives. These include static governors, which fix the processors in a specific P-state, and dynamic governors which change P-state according to CPU utilization.

Scaling down frequency and voltage reduces power consumption at the expense of reduced performance. This tradeoff is workload-specific. Ideal settings for DVFS are those in which loss of performance is relatively small. For example, for heavily memory-bound workloads, DVFS may be able to reduce processor frequency with little impact on performance. Another setting is situations in which reduced performance can be tolerated, e.g., a server with request latency bounds or deadlines. In such settings, there is no penalty for slower execution as long as deadlines are not missed. This is the scenario we focus on in the current paper.

There has been some previous work on management of DVFS in database systems. Tu et al. [33] describe feedback control approaches to energy management, and present preliminary work towards a controller that controls both DVFS and query optimizer to maintain a target request throughput. Xu et al. [38] also use feedback control to manage DVFS for database workloads, based on a throughput target. The goal is to minimize power consumption as workload characteristics change, e.g., the workload becomes more I/O intensive. However, since the controller uses a throughput target, this approach may be difficult to apply in the settings we consider, in which the workload intensity fluctuates. In such settings, we expect throughput to vary along with the offered load. Psaroudakis et al. [27] demonstrate that voltage and frequency scaling and power-aware scheduling can affect the energy efficiency of parallelizable database query operations, such as aggregations and scans.

Lo et al. [25] propose an external feedback control approach called *iso-latency* to manage DVFS for what they call *on-line, data-intensive (OLDI)* workloads, such as search. As is the case for our work, there is an explicit application quality-of-service (QoS) target, and DVFS is managed to minimize energy consumption while ensuring that the QoS target is met. We address a similar problem, but our approach uses feed-forward control implemented within a database server. In addition, our technique is intended to exploit shorter-term workload fluctuations. In contrast, Lo et al. focus on longer term (diurnal) fluctuations.

Tsirogiannis, Harizopoulos and Shah [32] take a dim view of DVFS (and of energy optimization in DBMS in general), arguing that the gains to be had in database systems are small. However, in their setting, power optimizations are considered beneficial only if they improve the system's energy-delay product (EDP). This is a very demanding standard. For example, saving a factor of two in energy consumption is only considered beneficial if the corresponding reduction in performance is less than a factor of two. In contrast, for iso-latency and our LAPS technique, performance reductions are of no concern as long as the explicit QoS target is met.

2.3 Memory Power Consumption

Memory power consumption consists of several components, such as array power, I/O power, register power, and termination power [10]. Some power is consumed even when memory is not in use, while other consumption depends on usage. For the purpose of understanding and modeling power consumption, all memory power is usually split into two parts: operation power and background power. Operation power is modeled as proportional to the frequency of operations performed on the memory devices, with a fixed weight (energy) associated with each operation type. In contrast, background power is consumed regardless of how many operations the device performs. Background power consumption differs when the memory device is in different states. Although the number of possible states, including substates, is large (e.g., 8 in [10]), usually a smaller number of states is used for modeling. The STANDBY (or ACTIVE) state is the only state in which the device is ready to accept commands, and its power consumption is the highest. The POWER-DOWN state deactivates input and output buffers and is has lower power consumption and the fastest exit latency (on the order of a few DRAM clock cycles). The SELF-REFRESH state is the deepest low-power state, during which the clock signal is disabled and most of the interface circuits are turned off. In this state the device performs refresh using an internal counter. The exit latency of the SELF-REFRESH state is approximately 500 DRAM clock cycles.

Controlling memory power states is the responsibility of the memory controller (MC). The algorithms used by the MC are poorly documented; however, at least some memory controllers use timer-based algorithms [3]. Timer-based algorithms switch memory into increasingly deeper low-power states after configured periods of inactivity elapse since the last access. When the MC supports multiple power states, deeper states are associated with larger timeout values to compensate for increased exit latencies from those states.

2.4 Managing Memory Energy Consumption

Modern systems include tools for measuring and controlling DRAM power consumption. In some Intel processors, RAPL (Running Average Power Limiting) [11] technology is used for both CPU and DRAM to estimate and, potentially, limit total system power consumption. In RAPL, DRAM power estimation is based on a linear analytical model which takes operation counters and power state durations as input. The model coefficients are calibrated by sensing current from the DRAM voltage regulator (VR) while running a set of synthetic memory test patterns during boot time. Using the VR sensor for direct measurements is deemed problematic because of its non-uniform accuracy and the design complexities associated with high frequency sampling and the need to transfer samples to the CPU outside of the test setting.

Some memory controllers implement counters that can be used for performance optimization and memory power estimation. Intel Xeon E5/E7 processors support a set of counters [5] in each memory channel, which can be programmed to count certain events or measure the duration of certain states. The counters relevant for power measurement include the number of row and column access operations, refreshes, number of clock cycles during which the internal clock is enabled, or the rank is in SELF-REFRESH state.

DRAM power consumption has received substantial attention in recent years but most of the work has focused on general computing systems. Maximizing low power state residencies is the goal of operating system scheduler-based power state management in [13]. The OS scheduler tracks rank use for each application process and switches the rank power state accordingly on a process context switch. Application processes are opaque to the scheduler and rank power management cannot be applied within a single process. Achieving the same goal by classifying memory regions as *hot* and *cold* and migrating them into separate ranks is discussed in [17]. Hotness is estimated by counting memory accesses in a simulated environment with no live evaluation. A technique for dynamic page migration between hot and cold ranks is analyzed in more detail in [35] and evaluated by simulation. Recent work [39] is similar to ours in proposing rank-aware memory allocation to save power in datacenters, but does not focus on a particular application class, such as databases. Managing power consumption by memory DVFS is proposed and analyzed in [10]. A similar approach in [11] limits memory power consumption by throttling memory operations by the memory controller.

Memory power management is less studied for database applications. Controlling the buffer pool is the obvious target as it has the largest memory footprint. [9] proposes a system that heuristically chooses one of two fixed buffer pool sizes to adapt to a changing workload. [19] proposes a power-aware buffer pool management system for real-time databases on flash memory where reads and writes have different energy costs. By dynamically adjusting the dirty page ratio in the buffer pool, the system minimizes energy use while meeting the response time requirements.

2.5 Server-Level Energy Management

In addition to energy management at the level of system components (processors, memory), it is also possible to manage energy consumption more holistically, at the server level. In systems with multiple servers, one general approach is to consolidate the system's work on as few servers as possible, allowing unneeded servers to be powered down. Servers can be dynamically (de)provisioned to accommodate time-varying workloads [28, 29, 21, 14, 24]. Orgerie et al. [26] provide a recent survey of these techniques for managing energy efficiency in distributed systems. Another approach is to use multiple types of servers, with different power/performance characteristics, and then direct work to the most appropriate type of server [22, 29]. While all of these techniques demonstrate improvements in energy efficiency, they are largely complementary to the work described in this paper, which focuses on single-server techniques.

Another approach, specific to DBMS, is energy-aware query optimization. An energy-aware optimizer characterizes both performance and energy consumption of candidate plans, and chooses a plan based on some combination of performance and energy efficiency objectives [36, 37, 23].

3. DBMS-MANAGED DVFS

Support for DVFS is widely implemented in server operating systems, and DBMS running in such systems can benefit from it. No DBMS modification is required. However, a DBMS may be able to obtain more benefit from DVFS if DVFS is managed directly by the DBMS, rather than the kernel. There are two reasons for this:

1. The DBMS can align DVFS with database objectives, which the kernel is unaware of. In particular, DBMS can use database workloads' quality-of-service (QoS) objectives to guide DVFS.

2. The DBMS has more detailed knowledge of its workload than the kernel does, and can use it to guide DVFS. The DBMS understands the logical structure of the workload (e.g., kernel and transaction boundaries)

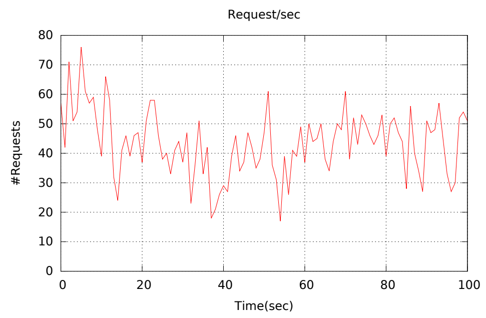

Figure 1: Request Arrival Rates from the World Cup 98 Request Trace

and has knowledge of both query execution plans and the internal state of the database execution engine.

Our goal in this work is to characterize the opportunity for DBMS-managed DVFS. We have focused on transactional workloads in which transactions have a *latency objective*. When the DBMS manages DVFS, its objective is to maximize the energy efficiency of transaction execution while ensuring that the latency objective is met. In our evaluation of DBMS-managed DVFS, the baselines include both transaction execution without DVFS, and transaction execution with kernel-managed DVFS.

The main utility of DVFS is that it allows performance to be traded for reductions in power consumption, allowing the performance of the processor to be adjusted to fit the needs of its time-varying workload. Workloads vary on many time scales. For example, many workloads experience diurnal fluctuations as a result of human behavior. Shorter, transient fluctuations are also common. For example, Figure 1 shows request arrival rates over a 100 second interval from publicly available traces from the 1998 World Cup web site. While the technique that we have explored can adjust processor speed in response to low frequency oscillations (e.g., diurnal), our primary interest is in determining the ability of database-managed DVFS to respond to natural short term fluctuations in the workload, down to time scales comparable to transaction execution times. Modern processors can switch P-states quickly, with latencies on the order of microseconds even under user space control [15]. Thus, it should be possible to react quickly, even to transient load fluctuations. DVFS controllers at the operating system kernel level already do so.

The remainder of Section 3 is structured as follows. We begin by briefly describing transaction execution in Shore-MT, which we have used as our experimental platform. Next, present our technique for managing DVFS, and show how it is implemented in Shore-MT. We then present the results of some experiments comparing our Shore-MT-based DVFS to our baselines. We conclude with some discussion of additional issues that will need to be addressed to make DBMS-managed DVFS practical.

3.1 Shore-MT

We used Shore-MT [18] together with ShoreKits [6] as an experimental platform for our work. Shore-MT is a multi-threaded storage manager optimized for multi-core architectures. The ShoreKits [6] OLTP benchmark suite provides

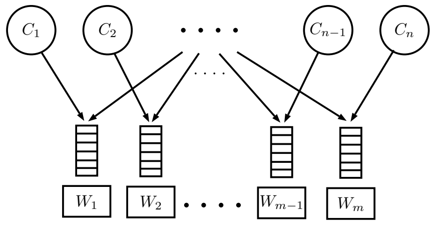

Figure 2: Clients, Workers, and Request Queues

```
Input: $l_{max}$: max allowed request latency
Input: $n_{max}$: transaction queue length threshold
Input: $Q$: transaction request queue
Input: $\mathcal{F} : f_1 \ldots f_k$: allowed frequencies (low to high)
Output: $f_{new} \in \mathcal{F}$: selected frequency
 1: procedure LAPS($l_{max}, n_{max}, Q, \mathcal{F}$)
 2:   $f_{new} \leftarrow f_1$
 3:   $s[f_i] \leftarrow 0$                                ▷ for each $1 \le i \le k$
 4:   if $|Q| > n_{max}$ then return $f_k$
 5:   for each transaction $t$ in $Q$ do          ▷ older to younger
 6:     $wait \leftarrow$ now $- t.arrival$               ▷ wait time of $t$
 7:     for $f \leftarrow f_{new}$ to $f_k$ do
 8:       /* cumulative service time thru $t$, at freq $f$ */
 9:       $s[f] \leftarrow s[f]+$ ESTIMATESERVICETIME($f$)
10:       if $(wait + s[f] > l_{max}) \wedge (f < f_k)$ then
11:         continue                          ▷ try higher frequency
12:       else
13:         $f_{new} \leftarrow f$
14:         break                         ▷ check next transaction
15: return $f_{new}$
```

Figure 3: The LAPS Algorithm. Each frequency in $\mathcal{F}$ corresponds to a distinct P-state.

hard-coded implementations of the TPC-C benchmark operations, designed to be run against the Shore-MT interface. ShoreKits also provides TPC-C clients, which can be used to generate a workload for the ShoreKits/Shore-MT TPC-C server. In the remainder of this paper, we'll refer to the combination of ShoreKits and Shore-MT as Shore-MT.

Shore-MT provides a set of worker threads, which are responsible for TPC-C request processing. Each worker has its own separate request queue. Workers take requests from their queues and execute them to completion, one request at a time. Client threads generate transaction requests and add them to worker queues. This is illustrated in Figure 2. Client and worker threads are all part of a single Shore-MT process.

For our experiments, we configured Shore-MT with one worker per core, and pinned each worker to its core. Each client generates requests for a single worker, with an equal number of clients per worker. All of the client threads are pinned to a single core to localize client interference to a single core.

3.2 DVFS in Shore-MT

To add the ability to manage DVFS to Shore-MT, we modified Shore-MT's workers to control DVFS. Each worker manages DVFS for the core to which it is pinned. DVFS for each core is managed independently of the other cores, based on the request queue for that core's worker.

Our goal is to manage core speed and power consumption in response to short-term fluctuations in request load. To achieve this, we took a simple approach. Each worker checks its request queue before it starts executing a new transaction, and potentially adjusts the P-state of its core. This P-state remains in effect throughout the execution of that transaction, until the worker prepares to execute its next transaction.

To choose a P-state, worker uses the *latency-aware P-state selection (LAPS)* algorithm shown in Figure 3. The LAPS algorithm selects the *lowest-frequency P-state such that all requests in the worker's request queue can be expected to complete within an application specified request latency bound*, which is an input to the LAPS algorithm.

To choose a P-state, LAPS tracks the arrival time of each request, so that it can determine the amount of waiting time that request has already experienced. In addition, it must estimate the remaining waiting time for the request, as well as the request's execution time. Since workers consume requests from their queues in FIFO order, the remaining waiting time for each queued request is the sum of the execution times of the requests ahead of it in the queue. Thus, LAPS requires a model that it can use to estimate request execution times. Since execution times depend on the P-state, which LAPS is selecting, the model must provide execution time estimates that are conditioned on P-state. In Figure 3, this model is encapsulated by the ESTIMATESERVICETIME function, at line 9. The LAPS algorithm traverses the queue and checks request response time under different frequency levels, tracking the minimum frequency that will allow it to avoid SLO (latency bound) violations for all of the requests it has checked so far. LAPS also incorporates an early-out mechanism (line 4 in Figure 3) which simply chooses the highest-frequency P-state in case the request queue length exceeds a configurable threshold (n_{max}). This bounds the overhead of the frequency selection. For all of the experiments reported here, n_{max} was set to ten and k, the number of P-states supported by our test CPU, is five.

Since our focus is on evaluating the potential of database-managed DVFS, we took a simple approach to the problem of estimating request execution times. We assume that the workload includes a fixed set of transaction types, and that the execution time of a request depends primarily on its type and on the core's P-state when the request is executed. We modified Shore-MT to monitor request execution times, and to track a running average execution time for each combination of request type and P-state. The ESTIMATESERVICETIME function uses these averages as its service time predictions. To simplify the presentation, request types are not shown in Figure 3, which is written is as if there is only a single type.

After using the LAPS algorithm to choose a P-state, the Shore-MT worker sets its core to the selected P-state and then executes the request to completion. On our Linux test platform, there are two different mechanisms that the worker could use to set the P-state. One possibility to make use of Linux's "UserSpace" DVFS governor, which allows application programs to set P-states. An alternative is to directly manipulate the processor's Machine Specific Registers (MSR). Linux exposes these registers through its filesystem interface. Our initial experiments with these two mechanisms showed that changing the P-state by directly manipulating the MSR was much faster(tens of microseconds) than doing so via the UserSpace governor, which is consistent with latencies reported by others [15, 34]. Since our DVFS approach approach may change the P-state frequently, we used MSR manipulation to do so.

3.3 DVFS Evaluation

We experimented with the LAPS algorithm, with the goal of characterizing how effectively it is able to reduce energy consumption, and how effectively it is able to meet its specified latency targets. We compared LAPS against the "On-Demand" and "Conservative" and dynamic scaling governors in the Linux kernel. In addition, we compared against configurations in which DVFS is not used at all.

3.3.1 Methodology

Our experiments used a 12-warehouse TPC-C database. Shore-MT was configured with 12 GB of buffer pool memory, which is sufficient hold the entire database. We configured twelve Shore-MT clients. To minimize data contention, each client was configured to issue requests for a different warehouse. To simplify the task of estimating request execution times, clients issued only one type of request, namely TPC-C *NEW ORDER* transactions.

We modified Shore-MT's TPC-C clients to convert them from a closed-loop design to an open-loop design, so that we could more precisely control the load offered to the server by the clients. Unlike closed-loop clients, open-loop clients do not wait for previously issued requests to finish before issuing subsequent requests. Instead, each client generates a request, waits for a configurable think time, and then generates the next request. Thus, the expected request rate per client is simply the inverse of the expected think time, regardless of system load. In all experiments we report, think times were uniformly randomly distributed, with a mean think time that we configured to control the total request load on the system. We also experimented with think times determined by a bounded Pareto distribution, which can generate self-similar request loads that are bursty over a wide range of time scales. However, the results were similar to the results for the uniformly distributed think times, at least for the Pareto parameters we tested, so we have reported only the uniform results here.

For each experiment, there is warm-up phase at the beginning followed by a short training phase. The training phase is used to initialize the LAPS service time estimates by running transactions in different P-states. For the purpose of reporting our results, we ignore transaction performance and system power measurements made during the warm-up and training phases.

All of our experiments ran on a server with a single 6-core AMD FX-6300 Bulldozer processor and 16 GB of DDR3 memory, running Ubuntu 14.04 Linux, kernel 3.13. This processor includes three modules, each with a pair of cores sharing an L2 cache. We chose this CPU for our experiments because it is able to perform DVFS independently on each module unlike the Intel CPU used for the experiments described in Section 4.3. Note that our LAPS algorithm is designed to choose a P-state independently for each core. However, on this platform, the effect is that both cores in a module will run at the faster of the two per-core P-states chosen by LAPS. It is becoming more common for processors to provide finer-grained (per core) control of P-states

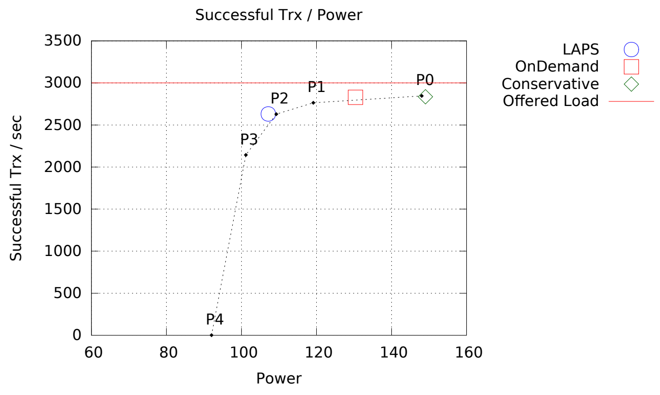

Figure 4: Performance vs. Power Under Medium Load

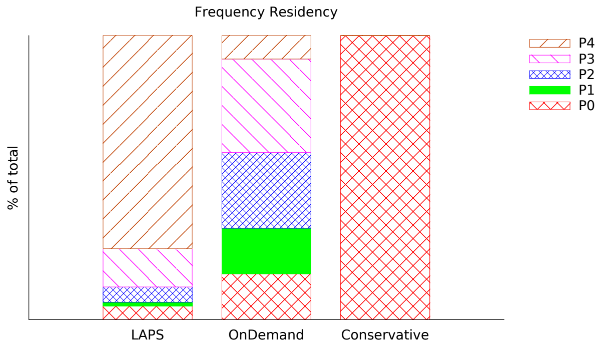

Figure 5: P-State Residency at Medium Load

and C-states, but we did not have such a processor available for these experiments.

During each experiment, we measured transaction response times so that we could determine the percentage of transactions that met the workload latency objective. Our primary performance metric is *successful transaction throughput*, which measures transaction throughput counting only those transactions with response times at or below the latency objective.

We used a Watts up? PRO ES power meter [7] to measure the power consumption of the entire server at one second intervals. This meter has a rated accuracy of $\pm1.5\%$ in our experiments' power range. The power reported for each of our experiments is the average of these one-second power readings over the entire measurement interval. We also used Machine Specific Registers (MSRs) on the processor to capture the processor's own per-core power measurements, also at one-second granularity. Unless otherwise indicated, all the power measurements we report are total system power measurements (from the power meter) rather than processor-only MSR measurements.

3.3.2 Experimental Results

Before running our primary experiments, we ran some calibration experiments determine the request loads under which we would test the system. To do this, we first determined the maximum load the system could sustain. We fixed all of the cores to the highest frequency P-state and then ran a series of experiments with higher and higher offered loads. We found that the transaction failure rate, i.e., the percentage of transactions with response times that exceeded the latency threshold, increased dramatically when the offered load exceeded 4800 transactions per second. Based on this measurement, we defined high, medium and low load scenarios for our primary experiments, with offered loads of 4350, 3000, and 2000 requests per second, respectively.

For all of the experiments reported here, we set the transaction latency objective to ten times the measured average service time of the transactions in the highest-frequency P-state. We also ran tests with a more relaxed latency objective of fifty times the mean service time. However, the results were (somewhat surprisingly) not qualitatively different from those with the tighter objective, and we have not presented them here.

Figure 4 shows our results for the *medium* workload, i.e., an offered load of 3000 transactions per second. Our server's processor supports five P-states, with frequencies ranging from 1.4 GHz to 3.5 GHz. The dashed line in Figure 4 shows the performance/power tradeoff we obtain for this workload by running it in each of the of the five P-states. In addition, the figure shows the performance and power we measured when running the workload with LAPS, and with the Linux OnDemand and Conservative dynamic frequency governors. Figure 5 shows P-state residencies under the medium workload for LAPS and the two kernel dynamic governors. These residencies describe the fraction of time the cores spend in each P-state under each dynamic governor.

For our workload, the Conservative governor provides the best performance, but consumes about 150 watts. The Conservative governor adjusts the P-state gradually, based on the CPU utilization. Under our workload, it rarely leaves the highest-frequency, highest-power P-state, as can be seen from Figure 5. In contrast, the OnDemand governor is more aggressive about power savings, and will move quickly among P-states in response to load changes. As a result, it reduces power consumption to about 130 watts with very little loss in performance.

LAPS, which has knowledge of transaction latencies and the latency objective, is able to reduce power consumption to less than 110 watts, about twice the reduction achieved by OnDemand. As shown in Figure 5, cores spend much more time in the lowest frequency state under LAPS than under OnDemand. Since LAPS is aware of latency slack, it is able to determine when execution at low frequency will be fast enough. OnDemand understands only that there is work to do, not how much time is available for doing it.

LAPS is also slightly less successful than OnDemand at ensuring that transactions hit their latency target, resulting a slight drop in successful transaction throughput. In theory, the transaction throughput under LAPS should be no worse than it is at the highest-frequency P-state, since LAPS should choose that P-state if necessary to ensure that transactions meet their deadlines. In practice, LAPS sometimes fails to achieve this because of mispredictons of transaction execution times.

Figures 6 and 7 show power and performance under our high and low loads, respectively. Under high load, the dynamic governors have little room to manoeuvre. The system spends almost all of its time in the highest-frequency P-state under both Linux governors. LAPS is able to identify some

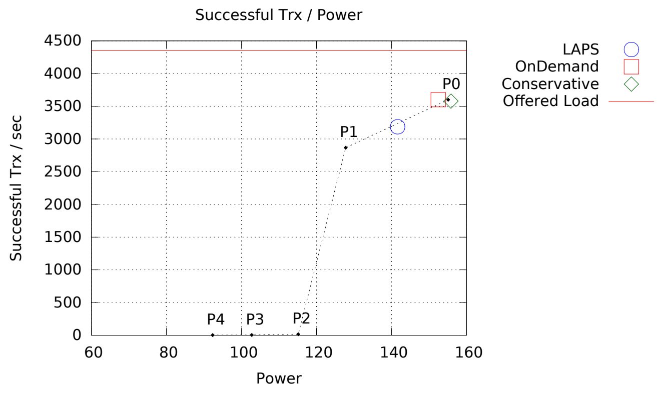

Figure 6: Performance vs. Power Under High Load

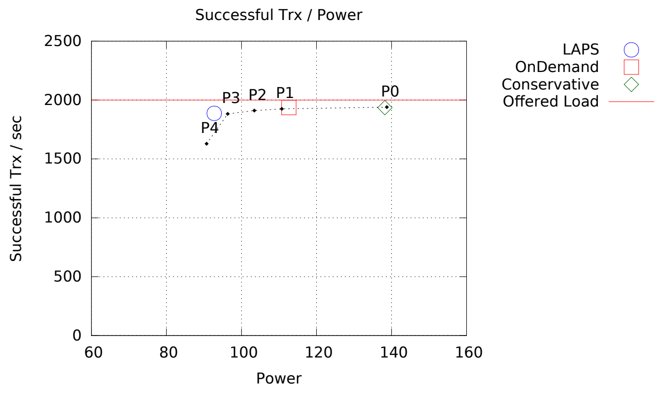

Figure 7: Performance vs. Power Under Low Load

opportunities to reduce core frequencies, and manages to reduce system power consumption by about 15 watts. However, as was the case under medium load, this power savings comes at the cost of a higher number of transactions failing to hit the latency target, because of mispredictions. The misprediction problem is somewhat worse at high load because longer request queues can magnify prediction errors.

LAPS provides its most substantial benefit in the low load case (Figure 7), where it reduces power consumption to slightly over 90 watts, almost with minimal impact on performance. Mispredictions are less consequential in this setting because there is usually a substantial amount of latency slack. The OnDemand governor also reduces power consumption, but not as effectively as LAPS. Analysis of P-state residencies at low load (not shown) indicate under LAPS, the cores are in the lowest-frequency P-state almost the entire time. Under OnDemand, cores are in the lowest-frequency state only about 30% of the time, while the Conservative governor leaves the cores in the highest-power P-state more than half of the time, even at this load.

3.4 Discussion

Our results suggest that DBMS-managed DVFS can be quite effective at reducing power consumption by exploiting latency slack to allow cores to remain in lower-power P-states. However, our approach was very simple, and our evaluation considered a highly-constrained setting. Our results also suggest that two issues in particular will be important if DBMS-managed DVFS is to be effective more generally. These issues are prediction of request execution times, and the brittleness of the LAPS algorithm.

No technique for managing DVFS can eliminate all violations of the latency bound, because load spikes can simply overwhelm the system, even at peak power. However, with LAPS there is also a risk that latency bounds will be exceeded if the scheduler underestimates request execution times. Our simple approach to execution time estimation used mean observed performance. Clearly, this will underestimate actual performance for a significant fraction of requests. We can address this problem by making more conservative predictions based on past performance. However, a related issue is that the LAPS algorithm itself is brittle, since it will always choose the lowest core frequency that it can get away with. Even small prediction errors can thus cause requests to miss their latency targets.

To address these problems, we will need a means of quantifying the potential error in service time estimates, as well as a simple and robust technique for managing the performance risk associated with setting a lower-power P-state. These are the subjects of our on-going work.

4. MEMORY POWER OPTIMIZATION

Along with processors, memory is another significant consumer of energy in computing systems. Studies have indicated that memory can be responsible for more than 25% of total system power [16]; however, this number strongly depends on the installed memory size and other hardware configuration parameters.

4.1 Power Optimization Opportunities

We can imagine two broad scenarios for memory power optimization in database systems:

In-Memory Database Systems
: In-memory database systems have a simplified architecture in which the buffer pool is eliminated, removing the corresponding management overhead and enabling faster transactions. The entire database must be memory-resident all of the time. If the database size cannot be accurately predicted, the database administrator must conservatively configure the system for the largest possible database size, resulting in memory overprovisioning. Virtual memory swapping, implemented by the operating system, can be used as a last resort; however, it is much less efficient than the DBMS buffer pool and causes a larger performance drop.

Non-Memory-Resident Databases
: Disk-based systems use a buffer pool to hold the most frequently used database pages. In such systems, it is typical that the buffer pool is the largest memory consumer, and its size can be relatively easily adjusted. Those properties make the buffer pool a natural candidate to consider when looking at memory power efficiency. A larger buffer pool size normally corresponds to better performance. However, skewed access patterns lead to diminishing performance gains as more memory is added to the buffer pool. Some recently proposed systems that employ "anti-caching" [12] or database-assisted paging [31] can also be classified into

this group as they operate with data larger that RAM and expose the memory/performance trade-off.

A common feature of these two cases is that the amount of provisioned memory is larger than the amount that is *actually* needed. Therefore, the idea of the memory power optimization in both cases is to avoid paying energy costs for the excess memory. We will explain our approach to tie memory energy consumption to the needed amount of memory in the next section. We expect the first case to be straightforward as some of the memory is simply not needed by the DBMS. In the second case, all memory is technically used, and reducing memory capacity may increase I/O rate and impact performance. However, due to diminishing performance returns from larger buffer pools, it may be possible to reduce the amount of memory with little impact on performance. When this is the case, the database system is *elastic* with regard to the amount of memory provisioned. In this work, we focus on the scenario in which the database fully fits in memory. We plan to explore the memory-performance trade-off in the future.

4.2 General Approach

To attack memory's energy consumption, we first need to understand how the memory energy consumption depends on memory use. As it was described in Section 2.3, the total power consumed by memory consists of operational and background power. Operational power includes the energy cost of memory access operations (such as page open, precharge, read, write) as requested by the processor and I/O devices. We assume that the frequency of those operations is determined by the workload and cannot be changed by controlling how much memory is used. In fact, the problem of reducing the number of memory accesses is one aspect of general performance optimization and can be achieved by making better use of CPU caches or designing more efficient algorithms. Therefore, we do not expect to obtain energy savings due to the operational power.

On the other hand, the amount of background power consumed is a complex function of workload, the total amount of memory installed, and memory allocation and power management policies. We focus on the reducing background to achieve power savings. Our goal is to keep memory ranks in their lowest power state as long as possible.

4.2.1 Rank-Aware Memory Allocation

The granularity of memory power management imposed by the current hardware architecture is a rank, which typically has a size of 4-16 GB in modern systems. We view memory granularity not as an absolute amount but as a portion of the total memory size. Thus, the increment of memory allocation is $\frac{1}{N}$ of the total capacity, where N is the number of installed memory ranks. In practice, the number of supported DIMM slots is not less than 8 even in smaller servers, which corresponds to 8 or 16 memory ranks. Therefore, such a system can support at least 8 power/performance levels. The number of slots in larger systems is higher so the granularity will pose even less of a problem.

Suppose that a database system that requires 32 GB of memory is running on a server with 64 GB of memory, in eight 8 GB ranks. If the 32 GB of memory actually used by the DBMS is allocated across all eight ranks, it will not be possible to save background power since all eight ranks will be in use. Thus, to be able to connect memory energy consumption to memory use, *it must be possible for the DBMS to control how the memory it is using is allocated to the system's memory ranks.* With this capability, the DBMS could allocate the 32 GB it requires on half of the available ranks, leaving the remaining ranks idle. Thus, the database system must know the physical memory configuration, and it must be able to allocate virtual memory from a particular rank. Since such *rank-aware memory allocation* is not supported by general-purpose operating systems such as Linux, we require a new mechanism to support rank-aware allocation. It includes the following components:

- Enumeration of physical memory ranks and determination of their physical start addresses, sizes, and NUMA nodes.
- Communication of the physical rank information to the application process (the DBMS).
- Mapping physical memory ranks to the application virtual memory space.
- Modification of the kernel to be able to use memory allocated in this way for I/O buffers.

We do not have an automated mechanism to enumerate ranks and determine their addresses and sizes. For our experiments in this paper, we manually reconstructed this information from various sources such as the hardware description and Linux kernel boot logs. In the future, it may be possible to retrieve rank addresses from the SMBIOS/DMI tables, or by reading the MC configuration.

Once we have enumerated the ranks, we classify each rank as either *managed* or *unmanaged*. The unmanaged ranks serve as a source of conventional memory for the OS and applications, while the managed ranks are used for memory power optimization. Usually, a minimal number of ranks (one per NUMA zone) is necessary for the OS and other applications to function.

To be able to use the managed ranks, we modified the Linux kernel to reserve a preconfigured range of physical address space at boot time. The DBMS is provided with the list of managed ranks, including their physical address boundaries. Knowing these boundaries, the DBMS can map the managed ranks into its virtual address space using the kernel's "/dev/mem" device.

Normally, an OS uses memory for its file cache when serving file I/O. The file cache can use a substantial amount of memory and can cause a significant number of memory accesses due to data copying. However, the file cache is redundant for a database system which implements its own buffer pool. Therefore, we configured the database system to use direct I/O and modified the kernel managed memory regions can be used as targets for direct I/O.

4.2.2 Memory Power State Control

Once the DBMS has decided that it does not need to use a managed memory rank, we would like it to be able to put that rank into a low power state to save energy. However, in existing systems it is typically impossible for software to directly control the power state of a memory rank. Instead, power state management is handled by the memory controller, and only a few configuration options are available for adjustment at boot time. Therefore, to cause a power state transition, the DBMS simply avoids using an unneeded memory rank completely, and relies on the MC to move that rank into a lower power state when it observes the rank is idle. This reliance on the MC's power management heuristics may impose several limitations on the power saving potential. First, the state transition may be unnecessarily delayed due to the idle timeout implemented by the MC. Second, the MC may not make use of deeper low-power states at all, since without rank-aware memory allocation most applications would rarely exhibit access patterns that would allow such deeper power states. However, we have relied on this mechanism for our experiments, as it is the only one available in our test system.

4.2.3 Memory Interleaving

Memory interleaving is an operational mode of a MC in which adjacent memory blocks are spread over multiple physical memory devices. Interleaving is possible on the bank, rank, and channel level. Since physical memory devices can operate in parallel, interleaving substantially improves memory throughput for larger sequential transfers. Because of potentially better performance and no apparent drawbacks, many systems enable the most aggressive interleaving by default.

Unfortunately, interleaving is clearly incompatible with rank-aware memory allocation, as it distributes the physical memory address space across the ranks at a very fine granularity. Therefore, disabling interleaving is a precondition for rank-aware memory allocation. Since disabling interleaving reduces memory throughput, at least for sequential memory access patterns, overall system performance may be affected. However, the performance impact of interleaving is strongly application-dependent. For many database workloads, which are naturally concurrent, we hypothesize that disabling interleaving may not have a significant impact on performance. In our experiments, which focus on transactional workloads, we have included a baseline configuration with interleaving enabled, so that we can characterize experimentally the performance impact of disabling it. We have not yet evaluated the performance effect of interleaving on other workloads.

4.3 Evaluation

All of our memory experiments used Shore-MT with open-loop clients, as described in Section 3.3. Our memory tests used the full, unmodified TPC-C transaction mix implemented by ShoreKits.

Since the TPC-C benchmark is update-intensive and the database grows as new orders are being added, we chose the initial database size carefully to accommodate new rows without exceeding the configured memory size. We also limit the test duration to 1 minute so that database growth is less than 10% for all runs. Since the database is entirely in memory during all experiments, 1 minute is sufficient for the system to reach the steady state. The offered load generated by the open-loop clients in our tests is 178000 tpmC.

Our memory experiments ran on a dual-socket Super Micro server featuring two Intel Xeon E5-2620 v2 (IvyBridge) processors, each with 4 channels of memory. Each memory channel was populated with one single-rank 8 GB Samsung DDR3L DIMM (1.35 V, part number M393B1G70QH0-YK0), operating at 1600 MHz. One DIMM on each processor was left for general use by the OS and applications, while the other three DIMMs per processor were managed for power management. A 200 GB Intel DC S3700 SSD was used to store the database. ShoreMT transaction logs were redirected to a tmpfs RAM-disk allocated in the unmanaged memory area. A custom Linux kernel was used based on the Ubuntu Linux kernel version 3.13.11.

4.3.1 Estimating Memory Energy Consumption

Estimating the energy consumption of DRAM is challenging. In a standard system, the DRAM power rails are integrated in the motherboard and are not easily accessible by a meter. The total system power consumption can be measured but it is difficult to know how power is distributed between multiple components. Moreover, it was difficult to reliably estimate memory power changes based on changes in the total system power, since memory is a relatively small power consumer in our test system, which has only 64GB of memory. A common way of estimating power consumption is modelling. In Intel processors, the RAPL mechanism [11] estimates DRAM energy consumption using event counters implemented in the memory controller. However, the RAPL power model and its calibration algorithm are not documented. Moreover, RAPL's memory power estimates for a previous generation of the Xeon processors are known to be inaccurate, and Intel advises against using them for measurements [4].

For these reasons, we estimate DRAM power consumption using an analytical model, similar to other models that have been described in the literature [11]. The model estimates power as a function of the numbers of occurrences of three types of memory operations (ACTIVATE, READ, WRITE), and of the time spent in three different power states (STANDBY, POWER-DOWN, SELF-REFRESH). A real memory device exposes a larger number of finer-grained power states [10]. However, the differences in power consumption between them are small, and they are more difficult to control or measure.

The memory power model is shown in Equation 1. In this equation, A, R, and W represent the numbers of ACTIVATE, READ, and WRITE operations. T_{st}, T_{pd}, and T_{sr} represent the time spent in STANDBY, POWER-DOWN, and SELF-REFRESH power states, and $T_{st} + T_{pd} + T_{sr}$ is equal to the total time of the experiment. The model uses six coefficients, which represent either the energy consumed per operation or the energy consumed per unit time in a given power state. Figure 8 lists these coefficients and gives the coefficient values we used for our experiments.

$$E = E_A A + E_R R + E_W W + T_{st} W_{st} + T_{pd} W_{pd} + T_{sr} W_{sr} \quad (1)$$

The memory power model coefficient values are derived from the nominal currents specified in the DIMM datasheet [1], which are specified for the test patterns from [2]. The power coefficients (W_{st}, W_{pd}, W_{sr}) are obtained by multiplying IDD3N (Active Standby Current), IDD3P (Active Power-Down Current), and IDD6 (Self-Refresh current), respectively, by the nominal voltage (1.35 V). The energy coefficients (E_A, E_R, and E_W) are obtained from the values of IDD0 (Operating One Bank Active-Precharge Current), IDD4R (Operating Burst Read Current), and IDD4W (Operating Burst Write Current) by subtracting the corresponding background currents, converting the difference to power, and dividing the final value by the number of operations per

Parameter	Abbreviation	Value	Units
Standby power	W_{st}	1490	mW
Power Down power	W_{pd}	1148	mW
Self Refresh power	W_{sr}	405	mW
Activate energy	E_A	19.1	nJ
Read energy	E_R	7.0	nJ
Write energy	E_W	7.6	nJ

Figure 8: Power model parameters for 8 GB Samsung DDR3L R-DIMM

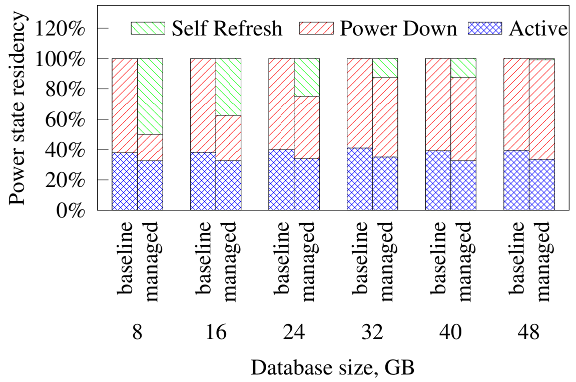

Figure 9: DRAM power states residency

second. For E_A, the background current is assumed to be an average of IDD3N and IDD2Q (Precharge Quiet Standby Current) because the module alternates between ACTIVE and PRECHARGE states during that test pattern. For E_R and E_W, the background current is IDD3N. When converting average power in test patterns to per-operation energies, reads and writes are assumed to take 4 DRAM clock cycles, while the ACTIVE-PRECHARGE cycle to take 39 DRAM clock cycles (nRC).

4.3.2 *Measuring Memory Activity*

The input to the power model is the counts for operations and residency times for the various power states. Those values were collected using the uncore performance monitoring counters of the Intel Xeon processor [5]. Each memory channel has four programmable counters. Each of those counters is a 48-bit register that can collect either occurrences of an event or the number of DRAM clock cycles while a certain condition exists. The following counters were collected: ACT_COUNT, CAS_COUNT, POWER_CKE_CYCLES, and POWER_SELF_REFRESH. Since the number of required counters exceeded four, sampling was used.

4.4 Estimating Potential Energy Savings for Memory-Resident Databases

Our test system has 48GB of "managed" memory that can be used for rank-aware memory allocation. To estimate the potential impact of rank-aware memory allocation for memory-resident databases, we ran a series of experiments with database sizes (including space for growth) ranging from 8 GB to 48 GB. In each experiment, the DBMS uses rank-aware allocation to allocate and use only enough memory to hold the database. Memory interleaving is disabled when rank-aware allocation is being used. The baseline in this experiment is the system in the default configuration: 4-way channel interleaving enabled and all available memory (48 GB) used for the buffer pool.

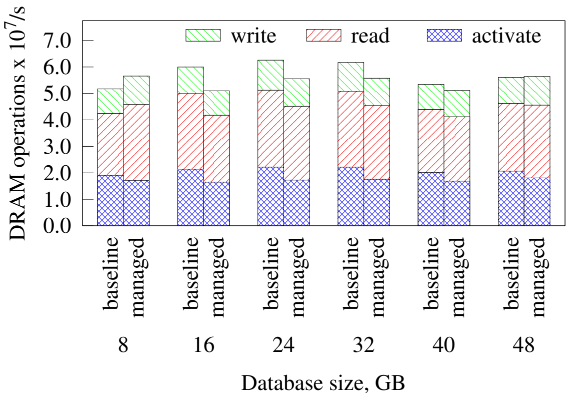

Figure 10: DRAM operation counts

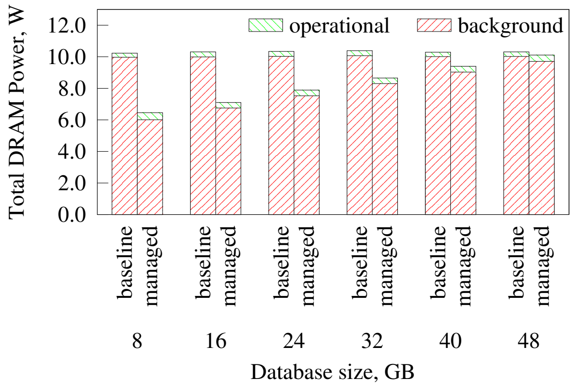

Figure 11: DRAM power consumption (modeled)

To evaluate the effectiveness of the proposed memory management method, we first report raw operation counters and power state residencies. Then, we feed that data to our power model (1) to estimate the power consumption of the memory subsystem. Finally, we measure transaction response times to see how transaction processing performance is affected by memory working in non-interleaving mode.

Memory power state residencies are shown in Figure 9. In the baseline case, with interleaving, memory is never put into the SELF-REFRESH state. In the managed case, the SELF-REFRESH residency increases with the number of ranks deactivated, reaching 50% with the smallest database. Increased SELF-REFRESH residency is the main factor driving savings in memory power from rank-aware allocation.

Memory operation frequencies are shown in Figure 10. As expected, there is no significant change in the number of operations from configuration to configuration. All configurations see the same offered TPC-C transaction load, and all have sufficient memory to hold the entire database.

The average memory power consumption estimated by the model is shown in Figure 11. Operation power constitutes a small percentage of the total in all cases, which indicates

Transaction	**Response time increase, %**		
	min	max	mean
New Order	1.1	5.2	2.8
Payment	0.9	3.6	2.2
Order Status	1.2	5.4	2.8
Stock Level	0.9	6.9	3.8

Figure 12: Response Time Increase Due to Rank-Aware Allocation

that the workload does not use memory heavily. When rank-aware memory allocation is used, the background memory power decreases as more ranks become idle. For the smallest database, the use of rank-aware allocation reduces background power consumption from just under 10 W to about 6 W The average saving is 0.8 W per idled rank.

Finally, we consider the performance impact caused by tight rank-aware allocation and the lack of memory interleaving. Figure 12 shows the increase in response time for each TPC-C transaction type when rank-aware allocation is used, relative to its response time with interleaving. We ran experiments using several different database sizes, with 8 runs at each database size. Figure 12 shows the minimum, mean, and maximum increases in response time over all of these runs. We observed response time degradation for all database sizes, although the impact was small. Thus, the average slowdown is 3-4% with maximum up to 7%.

4.5 Discussion

Our experimental results show that rank-aware memory allocation can be used to idle memory in excess of what is needed by the DBMS, and our memory power model predicts that idling memory in this way will reduce memory power consumption by up to 30%. Total memory power consumption in our test system is relatively low (just over 10 W in the baseline configuration). However, many systems have substantially more memory than our test server, and correspondingly higher memory consumption.

Our experiments also identify several challenges that must be overcome if rank-aware allocation is to be used. One challenge is accounting for the complex relationship between the amount of non-idle memory and DBMS performance. Idling a DIMM reduces power consumption, but also reduces the total memory bandwidth available to the system. It will be important to determine whether or not this will have a significant effect on the performance of a target workload. A second challenge is that memory power reductions are not proportional to the amount of memory that is idled. Estimated DRAM power consumption with three active ranks (one managed, two unmanaged) was about 35% lower than DRAM power consumption with all eight ranks active.

The degree of control of memory power states is ultimately determined by the hardware platform. With the exception of small embedded platforms, we could not find any generally available system that would expose power state control of memory modules to software. At the same time, the power control algorithms in memory controllers are rather conservative. We believe one reason for this is that the potential for power savings due to automatic power state management is negligible when applications are not aware of physical memory configuration and its power characteristics. However, current DRAM technology provides more opportunities for power savings, such as the SELF-REFRESH state or even completely disabling refresh for unused memory regions. We believe that the demand from applications for better control of memory power will stimulate system builders to implement better control mechanisms and make them available to applications.

5. CONCLUSIONS

In this paper, we have presented two techniques for power-optimization in database systems. Database-managed DVFS allows the DBMS to manage DVFS on the server's processors. This allows DBMS to exploit its knowledge of request latency targets, trading latency slack for reduced power consumption. Rank-aware memory allocation allows memory power consumption to be scaled according to memory capacity that is required by the DBMS, rather than the total memory capacity of the server.

Both of these techniques show promise in the in-memory, OLTP setting we considered. Database-managed DVFS resulted in higher power savings than DVFS managed by the operating system, particularly at lower loads. Rank-aware memory allocation enabled estimated memory power savings of nearly 40% when the database size was much smaller than the amount of memory provisioned on the server. Such savings are not possible without rank-aware allocation.

Although both techniques are promising, our experiments also highlighted challenges that must be addressed before these techniques can be applied. DBMS-managed DVFS relies on request execution time estimates, and estimation errors can limit its effectiveness. Since it is probably impossible to eliminate estimation error, the challenge is to implement DBMS-managed DVFS in a way that is robust against such errors. Fully exploiting rank-aware memory allocation may require more effective tools for management of memory power states in software. Rank-aware allocation also affects the memory system's bandwidth. While the performance impacts of this were small for our TPC-C workload, in general it will be necessary to manage the power/performance tradeoff that rank-aware allocation introduces.

6. ACKNOWLEDGEMENTS

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

7. REFERENCES

[1] 240pin Registered DIMM based on 4Gb Q-die. Datasheet. `http://www.samsung.com/global/business/semiconductor/file/product/DS_DDR3_4Gb_Q_die_RDIMM_Rev11_135V-2.pdf`.

[2] 4Gb Q-die DDR3L SDRAM. Datasheet. `http://www.samsung.com/global/business/semiconductor/file/product/DS_4G_Q-die_DDR3_135V_Rev10-1.pdf`.

[3] Desktop 3rd Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/3rd-gen-core-desktop-vol-1-datasheet.pdf`.

[4] Intel Xeon Processor E5 Product Family Specification Update. `http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-family-spec-update.pdf`.

[5] Intel Xeon Processor E5 v2 and E7 v2 Product Families Uncore Performance Monitoring Reference Manual. `http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/3rd-gen-core-desktop-vol-1-datasheet.pdf`.

[6] Shore-Kits. `https://sites.google.com/site/shoremt/shore-kits`.

[7] Watts up? power meter. `https://www.wattsupmeters.com/secure/products.php?pn=0`.

[8] Cost of power in large-scale data centers. Perspectives (blog), Nov. 2008.

[9] C. S. Bae and T. Jamel. Energy-aware Memory Management through Database Buffer Control. In *Proc. Workshop on Energy-Efficient Design*, 2011.

[10] H. David, C. Fallin, et al. Memory power management via dynamic voltage/frequency scaling. In *Proc. ICAC*, pages 31–40, 2011.

[11] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le. RAPL: Memory Power Estimation and Capping. In *Proc. Int'l Symp. on Low Power Electronics and Design*, pages 189–194, 2010.

[12] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-caching: A new approach to database management system architecture. *Proc. VLDB Endow.*, 6(14):1942–1953, Sept. 2013.

[13] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Scheduler-based DRAM energy management. In *Proc. Design Automation Conf.*, pages 697–702, 2002.

[14] A. Gandhi, M. Harchol-Balter, et al. Distributed, Robust Auto-Scaling Policies for Power Management in Compute Intensive Server Farms. In *Proc. Open Cirrus Summit*, pages 1–5, 2011.

[15] D. Hillenbrand, Y. Furuyama, et al. Reconciling application power control and operating systems for optimal power and performance. In *Proc. Int'l Wkshp. on Reconfigurable and Communication-Centric Systems-on-Chip*, pages 1–8, 2013.

[16] U. Hoelzle and L. A. Barroso. *The Datacenter As a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan and Claypool Publishers, 1st edition, 2009.

[17] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller. Improving energy efficiency by making DRAM less randomly accessed. In *Proc. Int'l Symp. on Low Power Electronics and Design*, pages 393–398, 2005.

[18] R. Johnson, I. Pandis, et al. Shore-MT: A scalable storage manager for the multicore era. In *Proc. EDBT*, pages 24–35, 2009.

[19] W. Kang, S. H. Son, and J. A. Stankovic. Power-Aware Data Buffer Cache Management in Real-Time Embedded Databases. In *Proc. RTCSA*, pages 35–44, Aug. 2008.

[20] J. Koomey. Growth in data center electricity use 2005-2010. Analytics Press, 2011.

[21] D. Kusic, J. O. Kephart, J. E. Hanson, N. Kandasamy, and G. Jiang. Power and Performance Management of Virtualized Computing Environments Via Lookahead Control. In *Proc. ICAC*, pages 3–12, 2008.

[22] W. Lang, S. Harizopoulos, et al. Towards Energy-efficient Database Cluster Design. *Proc. VLDB Endow.*, 5(11):1684–1695, July 2012.

[23] W. Lang, R. Kandhan, et al. Rethinking Query Processing for Energy Efficiency: Slowing Down to Win the Race. *IEEE Data Eng. Bull.*, 34(1):12–23, 2011.

[24] W. Lang and J. M. Patel. Energy Management for MapReduce Clusters. *Proc. VLDB Endow.*, 3(1-2):129–139, Sept. 2010.

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

[26] A.-C. Orgerie, M. D. d. Assuncao, and L. Lefevre. A Survey on Techniques for Improving the Energy Efficiency of Large-scale Distributed Systems. *ACM Comput. Surv.*, 46(4):47:1–47:31, Mar. 2014.

[27] I. Psaroudakis, T. Kissinger, et al. Dynamic Fine-grained Scheduling for Energy-efficient Main-memory Queries. In *Proc. DaMoN*, pages 1:1–1:7, 2014.

[28] P. Ranganathan, P. Leech, D. Irwin, and J. Chase. Ensemble-level Power Management for Dense Blade Servers. In *Proc. ISCA*, pages 66–77, 2006.

[29] D. Schall and T. Hrder. Energy-proportional Query Execution Using a Cluster of Wimpy Nodes. In *Proc. DaMoN*, pages 1:1–1:6, 2013.

[30] M. Själander, M. Martonosi, and S. Kaxiras. Power-Efficient Computer Architectures: Recent Advances. *Synthesis Lectures on Computer Architecture*, 9(3):1–96, Dec. 2014.

[31] R. Stoica and A. Ailamaki. Enabling efficient os paging for main-memory oltp databases. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, pages 7:1–7:7, New York, NY, USA, 2013. ACM.

[32] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the energy efficiency of a database server. In *Proc. SIGMOD*, pages 231–242, 2010.

[33] Y.-C. Tu, X. Wang, et al. A System for Energy-efficient Data Management. *SIGMOD Rec.*, 43(1):21–26, May 2014.

[34] J.-T. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber, and D. Dice. The turbo diaries: Application-controlled frequency scaling explained. In *Proc. USENIX ATC*, pages 193–204, June 2014.

[35] D. Wu, B. He, X. Tang, J. Xu, and M. Guo. RAMZzz: Rank-aware Dram Power Management with Dynamic Migrations and Demotions. In *Proc. Int'l Conf. on High Performance Computing, Networking, Storage and Analysis*, pages 32:1–32:11, 2012.

[36] Z. Xu, Y.-C. Tu, and X. Wang. Exploring power-performance tradeoffs in database systems. In *Proc. ICDE*, pages 485–496, Mar. 2010.

[37] Z. Xu, Y.-C. Tu, and X. Wang. Dynamic Energy Estimation of Query Plans in Database Systems. In *Proc. ICDCS*, pages 83–92, July 2013.

[38] Z. Xu, X. Wang, and Y.-C. Tu. Power-Aware Throughput Control for Database Management Systems. In *Proc. ICAC*, pages 315–324, 2013.

[39] D. Zhang, M. Ehsan, M. Ferdman, and R. Sion. DIMMer: A case for turning off DIMMs in clouds. In *Proc. SoCC*, pages 1–8, 2014.