DimmStore: Memory Power Optimization for Database Systems

Alexey Karyakin
Cheriton School of Computer Science
University of Waterloo
alexey.karyakin@uwaterloo.ca

Kenneth Salem
Cheriton School of Computer Science
University of Waterloo
ken.salem@uwaterloo.ca

ABSTRACT

Memory can consume a substantial amount of power in database servers, yet memory power has received considerably less attention than CPU power. Memory power consumption is also highly non-proportional. Thus, memory power becomes even more significant in the common case in which a database server is either not completely busy or not completely full. In this paper, we study the application of two memory power optimization techniques - rank-aware allocation and rate-based layout - to database systems. By concentrating memory load, rather than spreading it out evenly, these techniques create and exploit memory idleness to achieve power savings. We have implemented these techniques in a prototype database system called DimmStore. DimmStore is part of a memory power testbed which includes customized hardware with direct power measurement capabilities, allowing us to measure the techniques' effectiveness. We use the testbed to empirically characterize the power saving opportunities provided by these techniques, as well as their performance impact, under YCSB and TPC-C workloads. Under simple YCSB workloads, power savings ranged up to 50%, depending on load and space utilization, with little performance impact. Savings were smaller, but still significant, for TPC-C, which has more complex data locality characteristics.

PVLDB Reference Format:
Alexey Karyakin and Kenneth Salem. DimmStore: Memory Power Optimization for
Database Systems. *PVLDB*, 12(11): 1499-1512, 2019.
DOI: https://doi.org/10.14778/3342263.33422629

1. INTRODUCTION

A 2014 report [28] from the Lawrence Berkeley National Laboratory estimates the annual power consumption of US data centers to be about 70 billion kWh, and predicted growth to 73 billion kWh by 2020. Much of this power is consumed directly by servers, and CPUs and memory are responsible for most of that. As major power consumers, CPUs have long been power optimization targets. Modern CPUs include power saving features such as voltage and frequency scaling and core- and package-level low-power idle states, and a substantial amount of work has been devoted to techniques for exploiting these features [21, 22, 23].

Memory power consumption, in contrast, has received much less attention. This has made sense, because memory was typically assumed to consume less server power than CPUs. However, servers are being equipped with increasing amounts of memory to support data-intensive in-memory computing, and this affects servers' power consumption profiles. As an example, one recent study [3] used a manufacturer's server configuration tool to estimate CPU and memory power consumption of system configurations with varying amounts of memory, up to 6TB. For a loaded four processor server, estimated memory power consumption *exceeded* CPU power consumption for memory sizes above 3TB. Of course, this is a single example, but it serves as a useful reminder that memory power grows with memory capacity.

As memories become larger and more power hungry, how can we design database systems to reduce memory power consumption? How much memory power can be saved by doing so? In this paper, we present DimmStore, which represents a first step towards answering these questions. DimmStore is an in-memory transactional database system, based on H-Store [19], that is designed specifically to reduce memory power consumption.

DimmStore works by taking advantage of the power characteristics of modern memory modules (DIMMs). Like CPUs, DIMMs implement power saving features that allow them to reduce power consumption during periods of light load. The reductions can be substantial (Section 2). However, these features are not easy to exploit. For example, a recent study [20] of several database systems demonstrated that they had high memory power consumption even when memory utilization was low, indicating that the DIMM power saving features were not being exploited. DimmStore, in contrast, is specifically designed to exploit them. It uses *rank aware memory allocation* and *rate-based data placement* to deliberately skew memory access rates across the available DIMMs. This creates idleness on the least-loaded DIMMs, reducing overall memory power consumption.

This paper makes the several research contributions. First, through the design of DimmStore, we show how an in-memory transactional database system can be designed for memory power optimization by exploiting the characteristics of modern DIMMs. To our knowledge, DimmStore is the first attempt to build a database system designed to

Proceedings of the VLDB Endowment, Vol. 12, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3342263.33422629

exploit DIMM power states. Second, we observe that the rank-aware memory allocation and layout techniques used by DimmStore are poorly supported by operating systems. We describe how we work around these restrictions on our Linux-based server so that we can test and evaluate DimmStore. Our work-arounds are not suitable for production deployments. However, they illustrate the issues that must be addressed in order to enable DimmStore-style power optimizations in database systems. Finally, we present an evaluation of the power and performance characteristics of DimmStore, in relation to a non-optimized baseline. Our testbed includes custom direct memory power measurement capabilities. Our results, using YCSB and TPC-C workloads, show memory power reductions of nearly 50% in the best cases, with relatively minor impact on system throughput and latency. More importantly, measurements show how these savings are affected by load and memory utilization, and analyze DIMM behavior to show how the savings arise.

2. BACKGROUND

Modern servers commonly employ a Non Uniform Memory Architecture (NUMA). A NUMA system consists of a number of nodes, each including a processor and directly attached *local memory.* Access to memory on other NUMA nodes (*remote memory*) is possible by communicating with the corresponding remote node's processor via an inter-processor link.

Each processor accesses its local memory through one or more *memory controllers*, which are often integrated into the processor itself. Each controller, in turn, communicates with its DRAM through an interface called a *channel.* There may be multiple channels on a controller to increase total memory bandwidth and capacity. A memory channel is shared by a number of logical memory units called *ranks.* On each channel, the memory controller communicates to one rank at a time. For power management purposes, each channel includes Clock Enable (CKE) signals, which the controller can use to enable individual memory ranks to enter low power states, which will be described below. A memory rank may be implemented as a set of DRAM devices or as a logical partition within a single set of high-density devices. Physically, a number of ranks are placed on a memory module (DIMM), which is inserted into one of the memory slots on the server's motherboard.

To increase usable memory bandwidth, memory controllers are often configured to *interleave* memory. This means that memory addresses that are close together (in the physical address space) are distributed across multiple channels and ranks. Interleaving increases the likelihood that multiple channels and ranks can be used for a single memory transfer, reducing the duration of the transfer.

To support memory power management, DIMMs allow their memory ranks to be placed in different *power states.* In our work, we consider three of these states:

Stand By : Stand By is the normal operating state, and the only state in which memory operations (reads and writes) are possible. In the Stand By state, all interface circuitry is enabled and clocked, and therefore the power consumption is highest.

Power Down : If a memory rank is idle, the memory controller can put it into the Power Down state by deactivating its CKE signal. In the Power Down state, parts of the interface are disabled, reducing its memory power consumption. Before the rank can be used again, the controller must first put it back into the Stand By state, which introduces a small *exit latency.*

State	Power, W	Exit latency, ns
Stand By	~ 1.5	0
Power Down	0.9	~ 5
Self Refresh	0.3	~ 500

Figure 1: DDR4 Memory States

Self Refresh : In the deeper Self Refresh state, the rank's interface is completely disabled, and the memory's contents are refreshed autonomously. A rank in the Self Refresh state has the lowest power consumption. However, transitioning from the Self Refresh to the Stand By state requires restarting and synchronizing the memory interface, which makes its exit latency longer than that of the Power Down state.

The power and latency characteristics of these power states are summarized in Figure 1.

Memory controllers are responsible for driving power state transitions for the memory ranks on their channels. Because of the power/performance tradeoff that power state transitions present, there is no single best state switching strategy. In our testbed's server, the memory controllers implement a timeout-based switching policy. A memory rank is transitioned into a low power state once there have been no operations on that rank for a configured amount of time.

The total power consumed by a memory rank consists of two components. The first is *background power*, which is determined solely by the rank's power state (Figure 1). In addition, while a rank is in the Stand By power state, additional *operation power* is consumed by each operation (such as a read or write) performed on that rank. Operation power increases in proportion to the frequency of memory operations performed by the rank. In contrast, background power is also affected by the workload's burstiness, the power policies of the memory controller, and other factors. Decreasing the load on a memory rank may lead to decreases in background power consumption (because of creation or extension of idle periods) in addition to workload-proportional decreases in operational power.

3. DimmStore

DIMM power states present an opportunity for memory power optimization, but how can we actually exploit this opportunity in a database system? How much memory power can actually be saved?

To answer these questions, we developed DimmStore. DimmStore is an extension of H-Store [19], an in-memory database system that targets transactional workloads. DimmStore, like H-Store, logically partitions the database, and gives a single worker thread responsibility for each partition. Single-partition transactions are handled sequentially by a worker. Cross-partition transactions involve multiple coordinated workers.

H-Store was designed to support transactional workloads, with many small, short operations which typically access data from a single partition. Since DimmStore is based

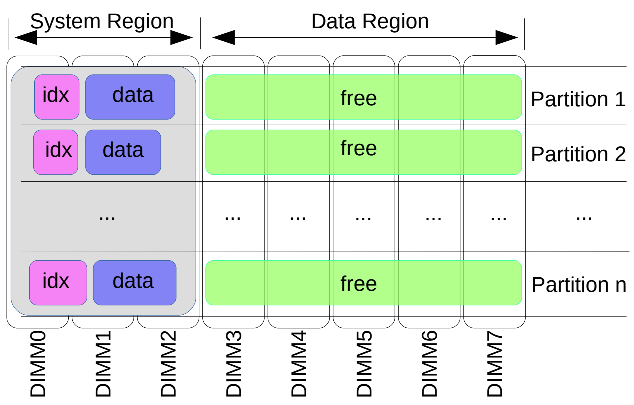

Figure 2: DimmStore With a Small Database

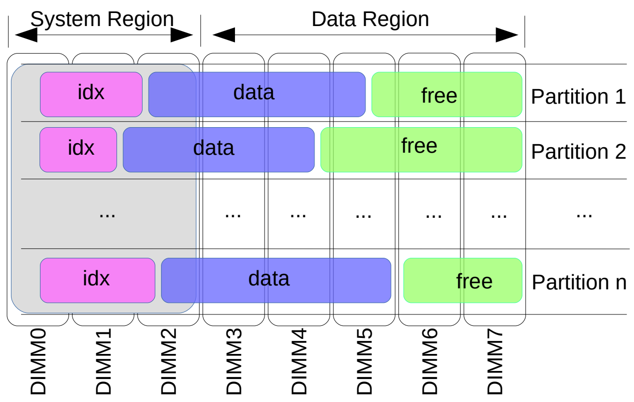

Figure 3: DimmStore After Spilling to the Data Region

on H-Store, our focus in this paper is also on transactional workloads. Similar memory power optimization opportunities may exist in database systems targeting analytical or hybrid workloads, but we do not consider them here.

In typical server configurations, memory load is distributed more-or-less evenly across the DIMMs (see Section 4.1). As a result, all DIMMs are busy and there is little opportunity for memory controllers to move DIMMs into low power states. DimmStore's power-saving strategy is to *unbalance* the memory load, shifting it away from some DIMMs and concentrating it on others. This creates idleness on the least-loaded DIMMs, and provides opportunities for them to enter low-power states.

To shift load, DimmStore controls memory allocation and data placement. The virtual address space in which DimmStore runs is divided into two *regions*, which we refer to as the *system region* and the *data region*. The formation of DimmStore's regions is *rank-aware*. This means that the virtual memory in the system region is backed by physical memory located on a subset of the available memory DIMMs. These are called the *system DIMMs*. The data region is backed by physical memory located on the remaining DIMMs, called the *data DIMMs*. In Section 4.2, we describe how this rank-aware memory partitioning is accomplished.

If possible, DimmStore uses *only* memory from the system region. This is illustrated by the DimmStore configuration shown in Figure 2. This has the effect of concentrating all memory accesses on the system DIMMs, leaving the data DIMMs completely idle and allowing them to sink into the deepest low-power state. This can save considerable power, as we show in Section 5. However, this situation is possible only if the entire database fits within the system region. When the database does not fit, DimmStore allocates memory from the data region and *spills* part of the database into it. DimmStore spills only as much data as it must to relieve memory pressure in the system region, and it places that data on as few of the data DIMMs as possible, as illustrated in Figure 3. Furthermore, it tries to spill only infrequently accessed (cold) data. The overall goal is to use as few of the data DIMMs as possible, and to access those that are used as infrequently as possible, to encourage the data DIMMs to spend as much time as possible in low power states.

In the remainder of this section, we present a more detailed description of memory power optimization in DimmStore. DimmStore's memory management requires support from the underlying operating system, since the operating system controls the mapping of DimmStore's virtual address space into physical memory. In Section 4.2, we describe the operating system support that is required, and how we implemented it in our testbed.

3.1 DimmStore's System Region

As we have described, DimmStore's power optimization strategy is to squeeze as much of the memory workload as possible onto the DIMMs that back the system region, so that power can be saved in the data region. DimmStore maintains separate memory allocators for the system and data regions. Whenever it requires memory, DimmStore must choose which region to allocate it in. DimmStore *always* chooses to allocate in the system region, except when it is spilling database tuples to the data region, as described in Section 3.3. This means that all of DimmStore's internal data structures, including all of its database indexes, are allocated in the system region. All newly-inserted database tuples are also located in the system region, although they may eventually spill out. Memory allocation in the system region is rank-unaware, i.e., DimmStore does not control which of the system region DIMMs a new memory allocation will map to.

The size of the system region is an important DimmStore parameter. It must be a multiple of the capacity of a single DIMM. In the examples shown in Figures 2 and 3, the system region occupies three of the server's eight DIMMs. DimmStore saves memory power by creating idleness in the data region DIMMs. If the system region is too large, then the number of data region DIMMs will be small, and this will limit the memory power savings that DimmStore can achieve. If the system region is too small, then DimmStore may be forced to spill hot data to the data region. This will reduce data region DIMM idleness and limit power savings.

In our current DimmStore implementation, the size of the system region is fixed at system boot time. An improved implementation would allow the system region to grow and shrink dynamically according to the characteristics of the workload and the database. This is feasible, but we have left this extension to future work. Our primary goal in this paper is to characterize the power savings potential of DimmStore's memory power optimizations, assuming that the system region is properly sized.

3.2 DimmStore's Data Region

If space becomes tight in the system region, DimmStore workers can spill database tuples into the data region, as will be described in Section 3.3. The available capacity of the data region is sliced and distributed among DimmStore's worker threads. Each worker uses its slice to spill tuples from the logical database partition that it is responsible for.

Each worker's slice is distributed across all of the data region DIMMs. When workers spill tuples into the data region, they fill their slices one DIMM at a time, in a common predefined order, as illustrated in Figure 3. The objective of this layout strategy is to leave some DIMMs completely or mostly unused in situations in which the data region is not completely filled.

To manage memory in this way, DimmStore's data region memory allocator must be rank-aware, i.e., it must understand how to allocate memory on a specific data region DIMM. We describe how this is accomplished in Section 4.2.

3.3 Tuple Eviction

When the system region is under space pressure, DimmStore spills database tuples to the data region. It evicts (spills) cold tuples, and only as many as needed to relieve the space pressure. The goal is to keep the data region DIMMs as lightly loaded as possible, while minimizing the performance and power overheads associated with eviction.

DimmStore adapts H-Store's anti-caching [10] mechanism to implement tuple eviction. As originally conceived, H-Store's anti-cache was tuple repository located on secondary storage. H-Store evicted cold tuples to the anti-cache when main memory was full. Anti-caching allowed H-Store to handle databases that would not fit into memory, while maintaining performance close to that of a fully in-memory system. In DimmStore, the data region serves as the anti-cache. The goal of DimmStore's anti-cache is to keep memory power consumption close to what can be achieved when the database fits entirely in the system region, and the data region is fully idle.

In DimmStore, tuple eviction is controlled independently in each logical database partition, and is implemented by the partition's worker thread. Each worker is given a capacity threshold, which depends on the size of the system region and the number of partitions. Every t_{evict} milliseconds, each worker checks the total system region size of the data and indexes in its partition. If the total exceeds the capacity threshold, the worker pauses transaction execution and normally evicts N_{evict} bytes worth of tuples from the system region to the data region, although this amount may increase if memory pressure does not abate. Normal transaction processing stalls in the worker's partition until eviction is complete. The two eviction parameters (t_{evict} and N_{evict}) control a tradeoff between eviction stalls (which can impact performance) and the maximum rate with which tuples can be evicted.

DimmStore workers use per-partition LRU lists to identify cold tuples to evict. A partition's LRU list includes all of that partition's unevicted tuples. When eviction is required, the worker evicts N_{evict} bytes worth of LRU tuples and removes them from the list. To evict a tuple, the worker must allocate space in the data region, move the tuple, deallocate space in the system region, and update database indexes to reflect the new tuple location.

The original implementation of anti-caching in H-Store used a global memory monitoring thread and per-table LRU lists. DimmStore uses per-partition monitoring, implemented directly in the worker threads, to reduce the overhead of monitoring and eviction. H-Store's original per-table LRU list required an additional policy to determine how much to evict from each table, but also provided the administrative flexibility of completely avoiding monitoring tables that are known to be hot. DimmStore uses multi-table LRU because it is simpler, but it could easily be modified to use per-table LRU lists in each partition.

3.4 Cold Tuple Access

In H-Store, any attempt to access an anti-cached tuple results in that tuple being *unevicted* from the anti-cache in secondary storage and returned to main memory. Since DimmStore's anti-cache is located in memory, it has more flexibility. Like H-Store, DimmStore can unevict cold tuples on access. Alternatively, DimmStore can access cold tuples directly in the data region, without first unevicting them.

Tuple uneviction is less expensive in DimmStore than it is in H-Store, because H-Store must read a block of tuples from secondary storage to retrieve the tuple. However, uneviction in DimmStore is still significantly more expensive than accessing the tuple directly. Uneviction is essentially the reverse of eviction. Like eviction, it requires memory allocation and deallocation, a memory-to-memory tuple copy, and index updates.

To avoid these overheads, DimmStore prefers to access cold tuples directly in the data region, without unevicting them. For cold evicted tuples that are rarely accessed, this is a good strategy. However, workloads can change, and tuples that had been cold may become warm. If a cold evicted tuple becomes warm, uneviction is preferable to frequent, on-going tuple accesses in the data region, which is supposed to remain cold.

DimmStore manages this dilemma using a simple randomized approach. Each time an evicted tuple is accessed, DimmStore unevicts the tuple with probability $p_{unevict}$, which is a system parameter. Otherwise, it simply accesses the tuple in place in the data region, without uneviction. This approach does not require any tracking of access recency or frequency for evicted tuples. It also has the desired property that cold tuples that become warm will, with high probability, eventually be unevicted.

4. SYSTEM SUPPORT FOR DimmStore

DimmStore requires that its two memory regions be placed on separate DIMMs. Within its data region, DimmStore also needs to be able to fill the underlying DIMMs one at a time as it spills out cold tuples. These capabilities require support from the operating system for *rank aware* memory allocation, i.e, the ability to allocate memory on specific DIMMs. Unfortunately, although rank-aware memory allocation has been explored in a variety of research settings [16, 17, 18, 32], we are not aware of any production operating system that supports rank-aware allocation.

In this section, we describe how we worked around this deficiency to allow DimmStore to run on our Linux-based testbed server. Our workarounds are not suitable for production use, but they do allow us to run DimmStore, and hence to gauge the power savings that could be achieved in production if suitable kernel support were available. The design of kernel support for rank-aware allocation is beyond the scope of our current work. However, we expect that an API similar to those currently provided by Linux (and other systems) for NUMA-aware memory allocation could be used. In addition, the workarounds described in this section provide some insight into the technical issues that would need to be addressed by a kernel implementation of such an API.

Applications (like DimmStore) request allocations of virtual memory from the kernel. In response, the kernel allocates physical memory to back the virtual memory, and establishes a mapping from virtual to physical addresses. Rank-aware allocation involves going one step further, because it is necessary to understand and manage the mapping from physical memory to the underlying DIMMs. In the remainder of this section, we first discuss physical-to-DIMM mapping, and then describe how we supported DimmStore's rank-aware allocation needs in Linux.

4.1 Physical Memory Mapping

The mapping from physical addresses to DIMMs is controlled by low-level configuration settings in the system BIOS. A common configuration is to *interleave* physical memory across the DIMMs, or across the DIMMs attached to a single memory controller in a multi-socket NUMA system. Memory interleaving stripes *each* page of physical memory across the DIMMs at a fine granularity. As a result, each page in an application's virtual address space will also be striped across all DIMMs. Memory interleaving is a performance optimization that can parallelize sequential memory accesses. However, it is incompatible with rank-aware memory allocation, which seeks to map virtual memory allocations to specific DIMMs. Thus, as a first step, we disable memory interleaving on our testbed system through BIOS settings.

Once interleaving has been disabled, the next challenge is to discover the (non-interleaved) mapping from physical memory addresses to DIMMs, a process we refer to as *DIMM mapping*. There is no existing mechanism that we are aware of that can reliably report this information to software. However, there are several indirect ways to infer the mapping. We used a method that takes advantage of the RAPL performance counters[1] in our server's Intel processors. In this method, the system is booted with a minimum amount of memory allocated for the kernel. We then run a program that sequentially probes physical memory addresses while monitoring RAPL counters. Depending on the version of the Intel platform, different RAPL counters are available, some offering per-channel or per-rank resolution. As the probing program probes a memory location, the counter associated with that location's physical memory channel or rank will be incremented, which is detected by the probing program. In our system, with an Intel E5 v3 processor and single DIMM populated in each channel, we used the per-channel CAS_COUNT event, which reports the number of reads and writes on a channel. Using this method we can build a complete map from physical addresses to DIMMs. By applying this method to our testbed server, we learned that most of the DIMMs are laid out sequentially in the physical address space according to their hardware numbering on the motherboard, with the exception of the very first DIMM, which stores two discontiguous physical address ranges.

[1]Intel processors estimate power consumption using models driven by hardware-maintained counts of events, such as memory accesses. These counts are accessible to software.

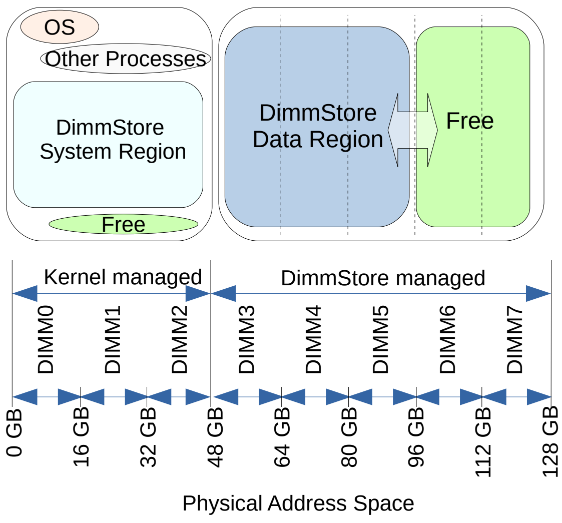

Figure 4: Physical Memory Management in the DimmStore Testbed

4.2 Rank-Aware Allocation

Once the physical-to-DIMM mapping is known, we use kernel boot parameters to restrict the physical memory available to the kernel to a subset of the DIMMs. We refer to this as *kernel-managed* memory. All memory allocations performed by the kernel occur within the kernel-managed memory. The physical memory on the remaining DIMMs is visible to the kernel, but is unmanaged. We limit the kernel-managed memory in Linux by setting the `mem` and `memmap` kernel parameters. The `mem` parameter sets the initial limit on the available memory at the beginning of the physical address space. The `memmap` parameters are used to add physical memory regions to the memory available to the kernel.

DimmStore's system region is mapped to kernel-managed memory, as shown in Figure 4. As was noted in Section 3.1, DimmStore uses separate memory allocators for its two regions. The system region memory allocator obtains memory from the kernel in the usual way, and the kernel satisfies these requests using kernel-managed physical memory. Hence, the entire system region will be confined to the kernel-managed DIMMs. Any other processes running on the server also obtain virtual memory from the kernel in the usual way, and hence they, too, will be confined to the kernel-managed DIMMs.

The physical memory that is not managed by the kernel is managed directly by DimmStore, and forms its data region. To take control of the unmanaged memory, DimmStore uses Linux's `/dev/mem` special device, which represents all of physical memory (including the unmanaged memory) as a file. DimmStore's rank-aware data region memory allocator uses Linux `mmap` calls to allocate virtual memory that is backed by the unmanaged parts of `/dev/mem`. We provide the data region allocator with the complete physical-to-DIMM mapping so that it can allocate memory on specific DIMMs by targeting specific parts of `/dev/mem`. For obvious security reasons, `/dev/mem` is only usable by privileged processes in Linux. Therefore, absent any operating system support for rank-aware allocation, DimmStore must run as a privileged process for the purposes of our experiments.

5. EVALUATION

In this section, we present an empirical study of memory power optimization, using DimmStore. Our goal is to answer two questions. First, how effective are the power optimization techniques presented in Section 3 at reducing memory power consumption? Second, do these techniques have a significant impact on performance? We consider two transactional workloads. The first is the Yahoo! Cloud Serving Benchmark (YCSB) [8], which has simple and easily controllable data access patterns. The second is TPC-C [1], which exhibits more complex and dynamic patterns.

5.1 Evaluation Platform

All experiments were performed using our testbed server, which has two 8-core Intel Xeon E5-2640 v3 processors working at nominal 2.6 GHz. Each CPU socket is provided with four memory channels and two DDR4 DIMM slots per channel. We populated only half of the DIMM slots to leave room for our memory power measurement apparatus. As a result, each channel has a single 16 GB DDR-4 DIMM, and the server overall has 8 DIMMs, for a total of 128 GB of memory. The number of DIMMs in the system and data regions varies between experiments, and is specified in the relevant sections below.

Our testbed server includes custom instrumentation for memory power measurement. We directly measure the power consumed by each individual DIMM using a current-sensing DIMM riser card, providing analog current readings separately on the V_{dd} and V_{pp} power buses. These readings are captured by a 16-channel data acquisition system. Using these real-time current measurements, we calculate per-DIMM power consumption assuming nominal voltages (V_{dd} = 1.2V and V_{pp} = 2.5V), as per the DDR4 specification [2].

It is worth noting that the total absolute memory power consumption in our testbed server is not high: normally in the range of 5-15 watts. Thus, absolute power savings are not high either. This is a limitation of the test server, which has only 16 DIMM slots, only half of which are populated. Eight is the maximum number of DIMMs that our measurement infrastructure will allow us to measure simultaneously. Thus, while most of the figures in the paper present absolute power numbers, our discussion will focus primarily on the power consumption DimmStore *relative* to that of the H-Store baseline, as this is a quantity that can be extrapolated to larger systems.

Param	Values	Default	Notes
s	0.5-1.2	0.95	Zipf skew param
DB size	10GB-100GB	60 GB	YCSB table size
trans rate	22.5-180 Ktps	90 Ktps	offered load
Sys size	32 GB	32 GB	Sys Region Size
t_{evict}	1 ms	1 ms	Eviction interval
N_{evict}	64 KB	64 KB	Eviction volume
$p_{unevict}$	$\frac{1}{64}$	$\frac{1}{64}$	Uneviction prob.

Figure 5: YCSB Experiment Parameters

In addition to these direct power measurements, we use RAPL counters to measure the number of memory read and write operations in each memory channel, and hence on each DIMM. We also use RAPL counters to measure memory power state residencies, i.e., the amount of time each DIMM spends in each memory power state. These counters are provided by the integrated memory controller in our Xeon processors. Finally, we measured application-level performance statistics, such as transaction response times, in DimmStore.

5.2 YCSB Experiments

Our first set of experiments uses YCSB workloads, which have relatively simple and controllable skewed data access patterns. We used the existing YCSB benchmark implementation from H-Store.

The YCSB database consists of a single table and a single index on the integer primary key. The size of the tuples is approximately 1000 bytes. We used a read/write mix with the ratio of 80% READ_RECORD to 20% UPDATE_RECORD transactions. Each YCSB transaction chooses a single primary key value, and either reads the corresponding record or reads and then updates the record, depending on the transaction type. Keys are selected independently and randomly, according to a Zipf distribution with skew parameter s.

In each experimental run, transactions are generated at a fixed rate, which we control. We ran experiments at eight settings of offered load, up to 180 Ktps, which is about 80% of the peak load sustainable by the baseline H-Store system. The size of DimmStore's system region was set to two DIMMs (32 GB) for all YCSB experiments. Figure 5 summarizes the other YCSB workload and DimmStore parameter settings.

Each experimental run consists of three phases: database loading, warmup, and measurement. We ignore measurements collected during the loading and warmup phases. The warmup and measurement phases are each 5 minutes long at the peak load we tested. For lower loads, we scale both intervals up so that the same amount of work is performed at every load level during each phase.

5.2.1 *Effects of Power Optimizations*

We begin with an experiment that is intended to illustrate *how* the memory power optimization techniques implemented in our testbed affect memory usage and power consumption. For this experiment, we fix the database size at 60GB, and use the YCSB workload at 90 Ktps. We compare per-DIMM memory access rates and power consumption under DimmStore with those of the baseline H-Store system. Later in this section, we look at what happens to power consumption and performance as the load and database size are varied.

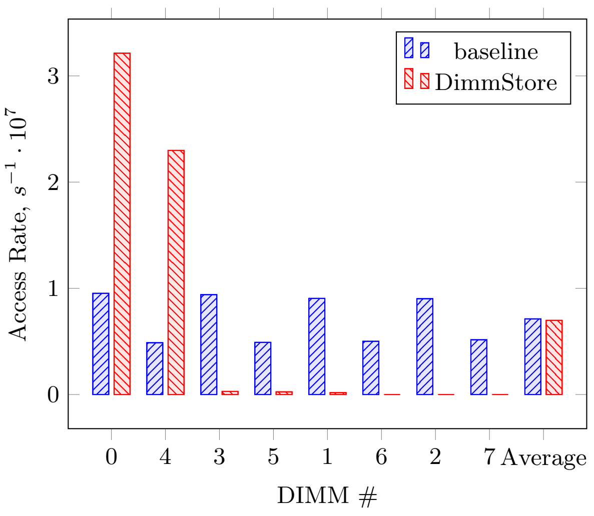

Figure 6: YCSB: Individual DIMM access rate, 60 GB database, 90 Ktps. For DimmStore, the system region consists of DIMMs 0 and 4, with the others making up the data region.

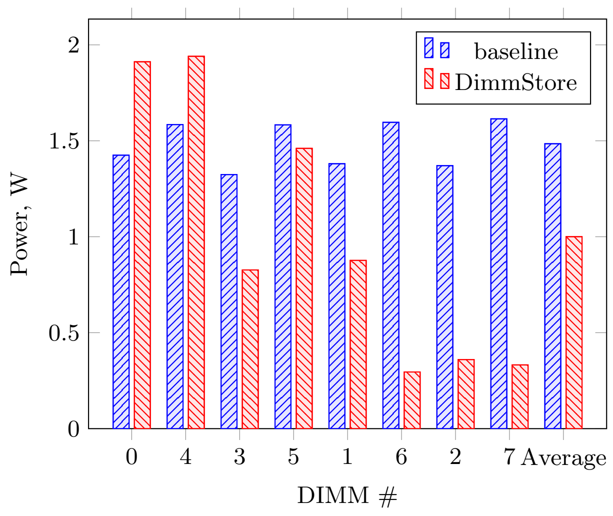

Figure 7: YCSB: Individual DIMM power consumption, 60 GB database, 90 Ktps load. For DimmStore, the system region consists of DIMMs 0 and 4, with the others making up the data region.

Figure 6 shows total memory access rates (read and writes combined) per DIMM for DimmStore and H-Store, as well as the average per-DIMM access rate across all DIMMs. This figure illustrates two key properties of the memory power optimizations in DimmStore. First, the average per-DIMM memory access rate in DimmStore is very close to that of the baseline. This indicates that the memory overhead of DimmStore's anti-cache, including tracking frequently accessed tuples and migration of tuples between the system and data regions, is very low for this workload. Second, DimmStore shifts memory accesses away from the data region, and into the system region (DIMMs 0 and 4), resulting in a very skewed load distribution across the DIMMs. In contrast, the baseline system, which uses memory interleaving, is not rank-aware, and does not attempt to separate hot and cold data, spreads the memory workload more evenly across the DIMMs.

Does the skewed access distribution created by DimmStore actually reduce memory power consumption? Figure 7 shows measured power consumption per DIMM for DimmStore and for the baseline H-Store system. Although both systems are handling approximately the same memory load, the average power consumption per DIMM is about 30% lower in DimmStore. DIMMs in the system region (DIMMs 0 and 4) consume more power in DimmStore than the corresponding DIMMs in the baseline system, due to the shifted workload. However, that is more than offset by power savings in DimmStore's data region DIMMs.

In this experiment, the server's memory capacity is not fully utilized. In its data region, DimmStore is rank-aware, and uses as few DIMMs as possible to store data. Thus, in this experiment, DIMMs 2, 6, and 7 are essentially empty, allowing them to sink into low-power states. DIMMs 1, 3, and 5 contain data, but it is cold data. Power consumption in DIMMs 1 and 3 is higher than that of the empty DIMMs, but still substantially lower than power consumption in the baseline. DIMM 5 also contains cold data but consumes more power than DIMMs 1 and 3, for reasons we discuss next.

The memory load shifting performed by DimmStore creates longer idle periods on the less-loaded DIMMs. If idle periods are long enough, those DIMMs can shift into low-power states, which reduces background power consumption. These background power savings are the reason for the net memory power savings in DimmStore. Figures 8 and 9 illustrate this effect. Figure 8 shows the memory power state residencies for each DIMM for the baseline system. All DIMMs spend at least half of their time in the full-power Stand By state, and almost never enter the very low power Self Refresh state. We can also observe that the memory controller on our server's second socket (which controls DIMMs 4-7) makes much less use of low power states than the controller on the other socket, although both sockets' DIMMs experience similar loads. We are uncertain of the reason for this, but it affects both DimmStore and the baseline.

Figures 9 shows the corresponding memory power state residencies for DimmStore. DIMMs 2, 6, and 7, which are empty, spend all of their time in Self Refresh state, reducing power consumption to about 0.3W per DIMM. Out of three DIMMs that do contain data, DIMMs 1 and 3 spend about 80% of their time in the Power Down state and about 10% in the Self Refresh state, and little time in the full power Stand By state. This is because these DIMMs hold only cold data. Thus, using both rank-aware allocation and access-rate-based layout, DimmStore is able to reduce background memory power consumption throughout the data region.

5.2.2 *Effects of Database Size*

Next, we show how memory power consumption is affected by the database size (Figure 10). With the largest (100GB) database, when memory is fully utilized, DimmStore saves roughly 11% of memory power, relative to the baseline system. DimmStore's power savings come from concentrating cold tuples in the data region, so that data region DIMMs have low access rates and reduced background power consumption.

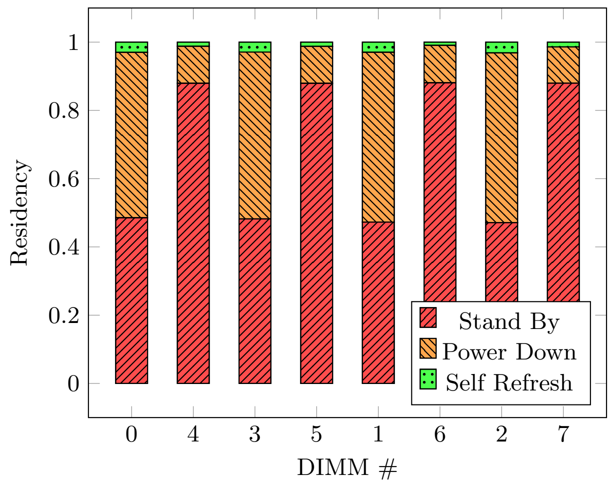

Figure 8: YCSB: Average Power State residency in the Baseline, 60 GB database, 90 Ktps load

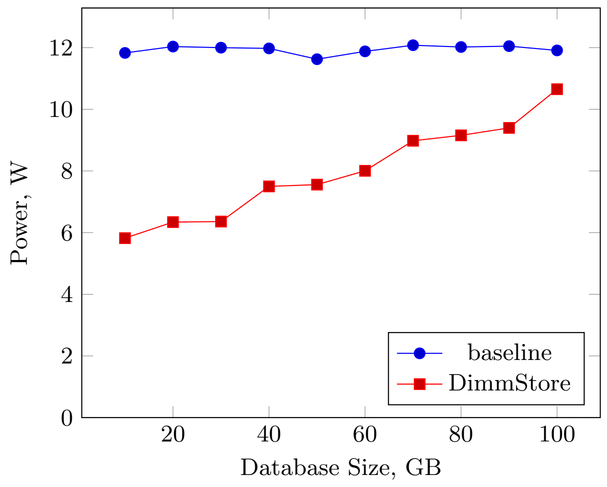

Figure 10: YCSB: Memory power consumption by database size, 90 Ktps load.

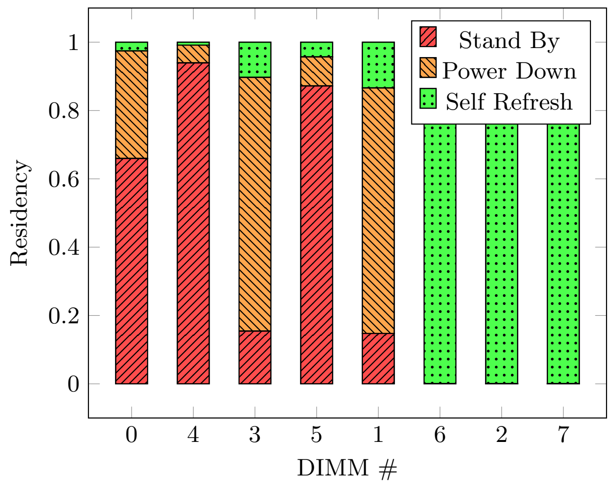

Figure 9: YCSB: Average Power State residency in DimmStore, 60 GB database, 90 Ktps load

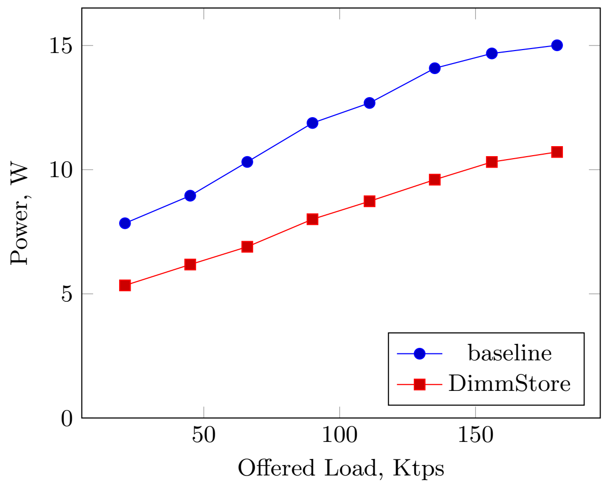

Figure 11: YCSB: Memory power consumption by load, 60 GB database

In experiments with smaller databases, memory accesses are "funnelled" to a smaller number of tuples while the total transaction rate stays the same. The baseline cannot take advantage of this, because the tuples are spread across all DIMMs. Hence, memory power consumption is insensitive to database size. In DimmStore, smaller database reduce memory power consumption. At the smallest database size we tested (10GB), memory power consumption in DimmStore was about half of that in the baseline. Smaller databases lead to reduced power consumption in DimmStore because of rank-aware allocation, which leaves some DIMMs completely unused when their space is not needed.

5.2.3 *Effects of Workload Intensity*

To study the effects of YCSB workload intensity, we fixed the database size at 60GB and varied the transaction request rate. Figure 11 shows total memory power consumption as a function of the request rate. Both the baseline and DimmStore show nearly linear increases in power consumption with increasing loads. However, DimmStore consumes roughly 30% less power across all load levels. Active memory power grows in proportion to memory access rate and is partially responsible for the power consumption increase in both systems. However, the contribution of active power to total memory power consumption is small, even at high transaction loads. The primary reason that power increases with load is background power. To explain this, we show the average DRAM power state residencies for all DIMMs, for baseline and DimmStore, in Figures 12 and 13, respectively. Figure 12 shows that increasing load increases time spent in the full-power Stand By state, largely at the expense of the Power Down state. For DimmStore, Figure 13 shows a similar increase in Stand By state residency, but at the expense of both Self Refresh and Power Down states combined.

5.2.4 *Effects of Access Skew*

We ran experiments in which the workload skew was varied, for a fixed database size (60 GB) and offered load (90 Ktps). Workloads with higher skew have more power saving potential because tuple accesses are more concentrated

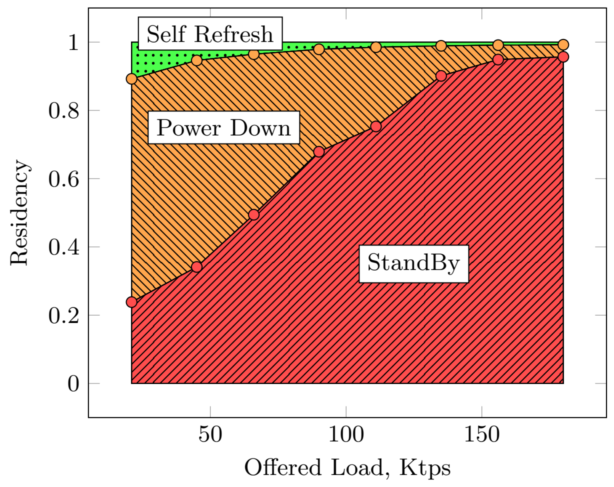

Figure 12: YCSB: Average Power State Residency in the Baseline by Load, 60 GB database

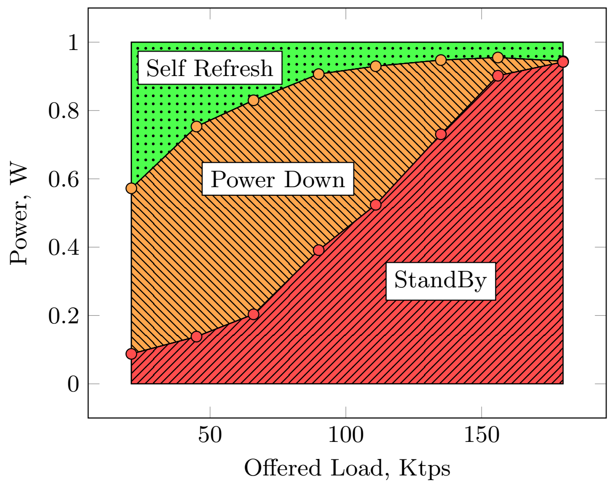

Figure 13: YCSB: Average Power State Residency of Non-Empty Data Region DIMMs in DimmStore by Load, 60 GB database.

towards the hot side of the distribution. As shown in Figure 14, this has only a small impact on power consumption. The effect is not large because access rates in the data region are already quite low at the default skew level.

5.2.5 *Performance*

The memory power optimizations implemented in our testbed may introduce some performance degradation. At the architectural level, concentrating memory load on a small number of DIMMs may introduce contention for those DIMMs. At the application level, DimmStore itself incurs costs to maintain the LRU list for identification of cold data, and for evicting and unevicting tuples from the data region. However, for the YCSB workload, we observed little performance impact from these optimizations.

We measured the peak throughput sustainable by DimmStore and the baseline by offering each system a very high load while placing the benchmark client in the blocking mode. In the blocking mode, the client senses backpressure

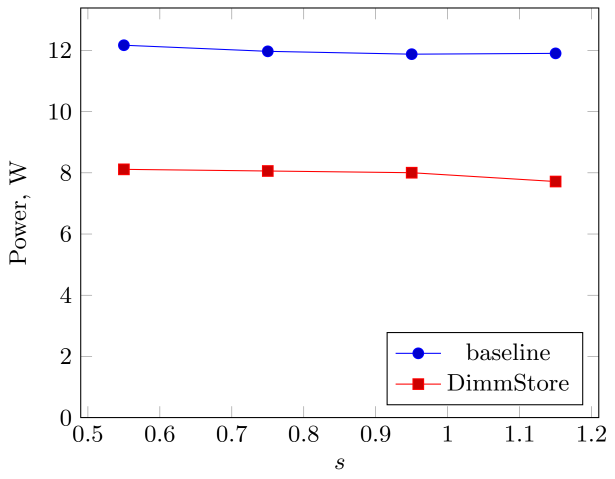

Figure 14: YCSB: Memory power consumption by access skew, 60 GB database, 90 Ktps load

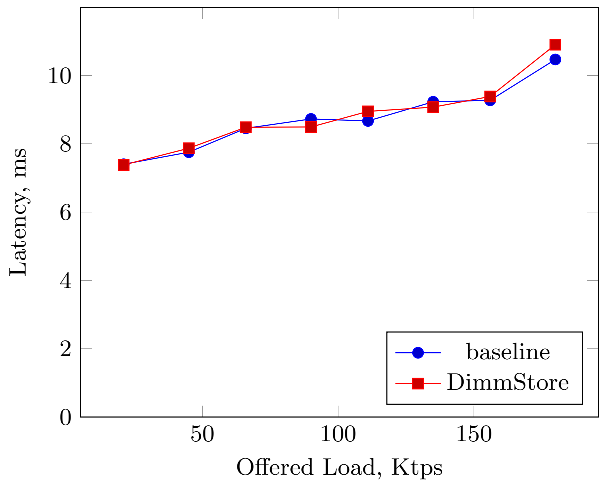

Figure 15: YCSB: Average transaction latency by load, 60 GB database

in the transaction queue, and throttles the load queue when backpressure is detected. We measured a peak sustainable throughput of 224 Ktps for the baseline H-Store system and 219 Ktps for DimmStore (using a 60 GB database), a degradation of about 2%. We also measured transaction latency at a range of off-peak loads. Figure 15 shows mean transaction latency as a function of load. Latencies in the two systems are very similar under this workload.

5.2.6 *CPU Power Consumption*

The overhead of detecting hot data and evicting and unevicting tuples translates to additional power consumed by the CPU. To estimate the additional CPU power consumption, we collected CPU power reports from the RAPL counters. For the YCSB workload, the overhead and resulting additional CPU power consumption are small. On average, over a set of 16 YCSB experiments with varying loads and database sizes, we observed that CPU power in DimmStore was less than 1% higher than baseline CPU power, with a worst case increase of 2.7%.

5.3 TPC-C Experiments

TPC-C is a widely used transactional benchmark that simulates an order-processing system. TPC-C exhibits more complex, time-varying memory access patterns than YCSB. We repeated our YCSB experiments using TPC-C. In particular, we compared the memory power consumption and transaction performance of DimmStore and the H-Store baseline across various database sizes and load levels.

5.3.1 *Methodology*

We used the TPC-C implementation from H-Store, with several modifications. First, since DimmStore requires that all indexes reside within the system region, we removed redundant indexes and dropped foreign key constraints. Second, we switched most indexes from hash to B-Tree, since the latter are more space efficient. As a result of these changes, the index-to-data ratio decreased from above 40% to about 22%, allowing us to test with larger databases. Finally, we disabled out-of-line data storage for large attributes, so that entire tuples are stored together. In TPC-C, only two columns were affected: S_DATA in the STOCK table (size 64) and C_DATA in the CUSTOMER table (size 500). The change did not increase the effective database footprint because these columns are assigned values of the maximum size.

For each experimental run, we choose a database scale factor, load the database, and then run the TPC-C workload. The scale factor in TPC-C, which is measured in "warehouses", determines the initial size of the database. We experiment with scale factors from 100 to 900 warehouses. Each 100 warehouses translates to about 10 GB of data. To leave some head room for the client processes and background tasks, we configured both DimmStore and the H-Store baseline to use 12 database partitions and 12 workers, which use 12 of the 16 cores available on our testbed server. In DimmStore, the size of the system region is set to three DIMMs (48 GB) for all experiments. The remaining DimmStore configuration parameters were set as for YCSB, as shown in Figure 5.

Each experimental run has loading, warmup, and measurement phases, as for YCSB. The warmup and measurement phases lasted 2.5 and 5 minutes, respectively, at the highest offered load level. During each run, the TPC-C database grows. We extended the warmup and measurement phases when testing below peak loads so that the actual database size (after growth) was approximately the same during the measurement period, regardless of the load. We performed runs with offered loads up to about 65 Ktps, which is about 90% of the peak load sustainable by the baseline H-Store system.

5.3.2 *Effects of Database Size*

In our first set of experiments, we fixed a medium offered load level (36 Ktps) and compared the memory power consumption of DimmStore and the baseline as the initial database size is varied. Figure 16 shows the result of these experiments. The results here are similar to what we observed for YCSB (Figure 10). Memory power consumption in the baseline is insensitive to the database size, while DimmStore is able to translate smaller databases into memory power reductions. The maximum power savings we observed was about 43%, for the smallest database.

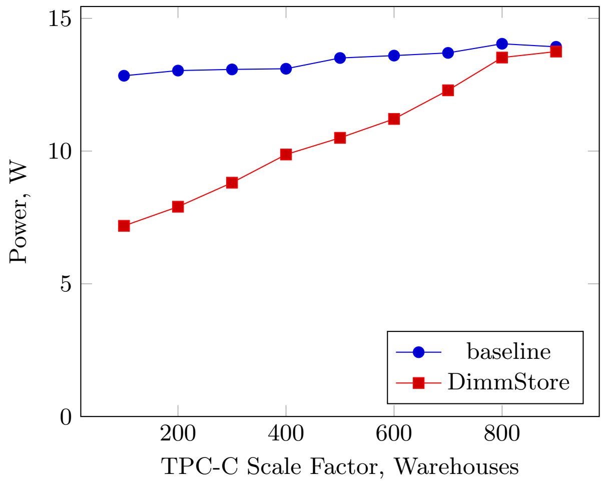

Figure 16: TPC-C: Memory power consumption by database scale factor, 36 Ktps load

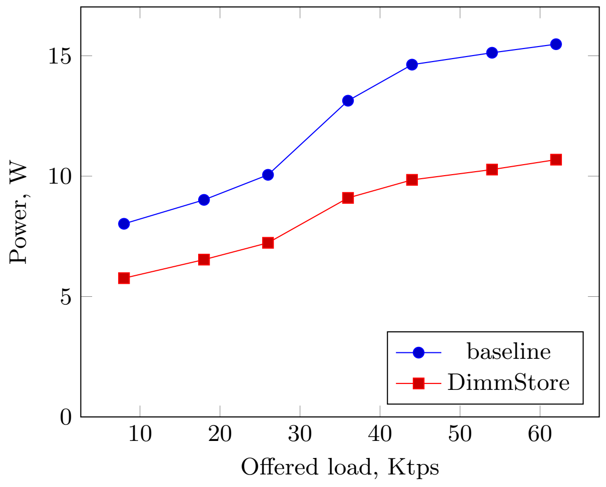

Figure 17: TPC-C: Memory power consumption by load, 350 warehouse DB

5.3.3 *Effects of Workload Intensity*

In our second set of experiments, we fixed the initial database scale factor at 350 warehouses, and varied the offered load. Figure 17 shows memory power consumption as function of the offered load. As was the case for YCSB (Figure 11), the memory power gap between DimmStore and the baseline is maintained across the load spectrum.

5.3.4 *Performance Effects*

Finally, we consider DimmStore's effect on TPC-C performance. We measured peak throughput of DimmStore and the baseline H-Store system with different initial database sizes, using the same methodology as in the YCSB experiments. Figure 18 shows the results of those experiments. The DimmStore's peak throughput is approximately 6% below H-Store's when the database is small enough to fit in the system region. As the database becomes larger, the additional overhead of data eviction and uneviction comes into play and throughput gap grows to about 10%.

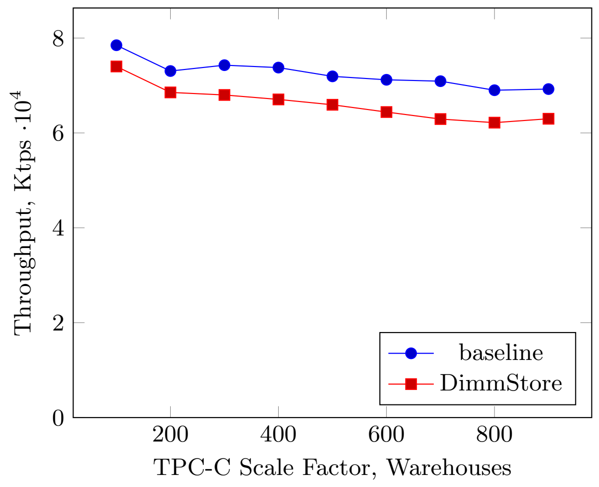

Figure 18: TPC-C: Peak throughput by database size

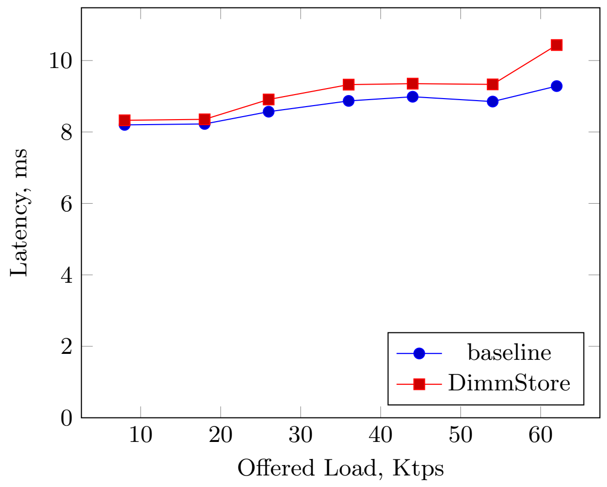

Figure 19: TPC-C: Average transaction latency by load, 350 warehouse DB

We also measured transaction latency for both systems at offered loads below peak. Figure 19 shows the mean transaction latency (averaged over all TPC-C transaction types) as a function of the offered load. At low to medium loads, DimmStore's memory optimizations have little to no effect on transaction latencies, but the gap was larger at the highest load levels. Since memory load is relatively light in these experiments, even at high transaction rates, we attribute this primarily to overheads within DimmStore, and not to memory contention. Overheads include maintenance of the LRU list and eviction and uneviction of tuples. While these overheads are not very significant at low load, their impact increases as load gets higher. In particular, tuple eviction in each database partition monopolizes that partition's worker for short periods of time, during which pending transaction work must queue. We believe the impact of these overhead can be reduced in DimmStore, e.g., by using lighter-weight access frequency estimation and by doing finer-grained tuple evictions, but we leave improvements of these mechanisms in DimmStore to future work.

5.3.5 *CPU Power Consumption*

As for YCSB, we used RAPL performance counters to measure CPU power. Due to the more complex workload, DimmStore's overhead is higher in TPC-C than in YCSB. Over all of our TPC-C experiments, with various offered loads and database sizes, we observed that DimmStore's CPU power consumption ranged from about 3% to 6% larger than the baseline's.

6. RELATED WORK

A small body of work has looked specifically at memory power optimization for database systems. Bae and Jamel [5] argue for DBMS-controlled memory management to achieve a good balance between memory power and performance. They use a heuristic algorithm to dynamically adjust the database buffer pool size according to changes in the workload. Unlike our work, this work targets disk-based DBMS. Appuswamy et al [3] predict that the significance of memory power consumption will grow. They study the effects of DRAM frequency scaling and power states on DBMS power efficiency and conclude that power states have a greater effect on memory power consumption. They also envision several directions towards memory power optimization in main-memory database systems, including the use of hot-cold data classification to direct data allocation on DRAM ranks. This is one of the power-saving techniques realized in DimmStore. In our previous work [20], we characterized memory power consumption in databases over a range of load and memory utilization settings. We found that memory power consumption was dominated by background power, and that memory power consumption was insensitive to both load and data size.

A larger body of work has considered memory power consumption across a wider variety of applications. These approaches are application-oblivious and typically work at the operating system, compiler, or hardware level. In most cases, memory power states are the primary mechanism for controlling power consumption, and it has been established that the lengths of memory access idle intervals are the key to using power states effectively. Delaluz et al [11] achieve memory power reductions using compiler-optimized access to memory locations residing in different memory ranks. In other work, Delaluz et al [12] rely on process scheduling to maximize the duration of idle intervals. Similarly, Jia et al [18] consider rank-aware allocation of memory to Linux thread groups.

Huang et al [17] observe that idle periods are very short in real workloads, which prohibits memory from entering deeper low power states. They introduce a data migration technique that migrate data between hot and cold memory ranks, guided by page-level access counts managed by the operating system. Wu et al [32] describe a related technique in which memory pages are mapped to hot or cold ranks according to the MQ algorithm, and the number of page migrations to realize this mapping is minimized. Finally, Huang et al [16] describe the concept of power-aware virtual memory, which minimizes the set of memory ranks used by each process. In contrast to these system-level approaches, we use the DBMS-specific information about data access to infer data hotness and manage migration. In addition, the units of data allocation and migration in DimmStore are logical rather than physical, allowing for more effective separation of hot and cold data.

Memory power optimization at the hardware level is also a popular research direction. Zheng et al [34] attribute the lack of power state effectiveness in reducing memory power to the large granularity at which power states are applied. They propose to partition a memory rank into multiple "mini-ranks" consisting of individual DRAM chips so that each "mini-rank" can sink into low power states independently. Malladi et al [24] argue that existing hardware support for memory power management is not well suited to handle access patterns with low idleness. They propose to redesign DRAM interfaces to make exiting from low power states much faster, which would reduce memory power consumption without explicit software optimization. Although we agree that the existing hardware mechanisms, e.g. deep low power states, are underused, we see opportunities to increase the amount of memory idleness through software. Another potential mechanism to control memory power is Dynamic Frequency Scaling (DFS), which is discussed by David et al [9], Deng et all [13], and Sharifi et al [27]. This approach is complementary to idleness-based techniques, such as those in DimmStore.

Looking at the problem more generally, Zhang et al [33] discuss essential features of a power-efficient system, such as non-interleaved memory access, use of DRAM low power states, powering down of DRAM modules and NUMA nodes, and migration of hot data to a subset of the modules. They argue that significant amounts of energy can be saved by taking advantage of these features. Tsirogannis et al [31] analyzed the impact of system configuration on the power efficiency of database systems. However, in that work the contribution of DRAM was assumed to be constant, due to the lack of tools or models to estimate its consumed power.

Our work relies on the idea of classifying data as hot or cold. Our system is based on H-Store's anti-cache [10] implementation. The anti-cache work considers several approaches to data classification, including LRU and timestamp based methods. Several other techniques have also been proposed. Stoica and Ailamaki [29] described an extension to VoltDB (also a derivative of H-Store) that offloads the processing of access logs to a dedicated thread or even a different machine to offset the performance effects on the transaction execution. Project Siberia [30], part of Microsoft Hekaton, identifies hot and cold data using a backward algorithm with low overhead. Similar problems arise in storage systems, in which it is also critical to detect frequently used items with low space and CPU overhead. For example, mechanisms based on hash functions [15] or bloom filters [6] have been shown to perform better than variants of LRU. Efficiency of hot tuple classification is not the focus of our current work, but we believe that the CPU overhead of DimmStore can be significantly reduced by utilizing a more sophisticated approach.

Power optimizations like DimmStore's have also been explored in the context of storage systems. Much as DIMMs can be put into low power states, magnetic disks can be spun down or completely shut off to save power. Power policies based solely on idle intervals in individual drives are not successful because these intervals are shorter than the break-even time [14]. Some strategies for extending idle periods in storage systems can be applied to main memory as well. For example, power-aware scheduling of I/O requests [7], temporary redirection of writes from powered down disks to active disks [25], and power-aware disk cache replacement policies [26], aim at extending disk idle intervals to save power. Similar ideas can be transferred to DimmStore to extend intervals between evictions and unevictions, further reducing the residual heat in the data region.

7. CONCLUSION AND DISCUSSION

In this paper, we explore opportunities for memory power optimization in database systems. We focus on two power optimization techniques: rank-aware allocation and access-rate-based data placement. They reduce memory power consumption by increasing memory idleness, allowing memory to spend more time in low power states. We implemented both techniques in DimmStore, which is based on H-Store.

For YCSB workloads, DimmStore reduces memory power consumption by up to 50% relative to H-Store, with larger savings for smaller databases. DimmStore produces power savings across all system load levels, and performance overhead is small. For TPC-C workloads, which have more complex memory access patterns, memory power savings still approached 50% for small databases. However, these gains disappeared at the largest database sizes.

Although DimmStore was able to deliver significant memory power savings, we were forced to rely on ad hoc techniques to work around a lack of operating support for rank-aware memory management. We identified some of the technical issues that will need to be addressed before the kernel can provide such support, e.g., the need to discover the mapping between physical addresses and DIMMs. DimmStore could also potentially benefit from additional hardware capabilities. For example, fine-grained, run-time control of interleaving would make it easier to dynamically size DimmStore's system and data memory regions. In addition, memory controllers could provide configurable scheduling policies that could be used to trade request latency for extended idle periods and increased power savings.

One issue that we have left for future work is sizing of DimmStore's memory regions. Ideally, it should be possible adjust these sizes dynamically in response to workload changes. For example, if memory pressure forces DimmStore to spill warm tuples to the data region, it would be better to increase the size of the system region instead, to ensure that the remainder of the data region remains cold.

Another related issue is the arrival of new non-volatile memories in DIMM form-factors, such as Intel's Optane DIMMs [4]. One interesting question is whether DimmStore's power optimization techniques will apply. We do not yet have access to detailed power specifications for these DIMMs. However, one relevant observation is that most of the background power consumption of our server's current DRAM DIMMs comes from their channel interface electronics, and not from refreshing the contents of the memory cells. As shown in Figure 1, refresh accounts for only about 20% of the DIMM's background power consumption. Since non-volatile DIMMs will also require channel interfaces, we expect them to have similar power-state profiles, and thus be amenable to the same techniques that DimmStore applies to DRAM DIMMs. Non-volatile DIMMs may expose additional power optimization opportunities, e.g., an imbalance between the active power consumption of reads and writes.

8. REFERENCES

[1] Transaction Processing Performance Council. TPC BENCHMARK C. Standard Specification. Revision 5.11, 2010.

[2] JESD79-4a. DDR4 SDRAM. JEDEC Standard., Nov. 2013.

[3] R. Appuswamy, M. Olma, and A. Ailamaki. Scaling the Memory Power Wall With DRAM-Aware Data Management. In *Proc. Int'l Workshop on Data Management on New Hardware*, pages 3:1–3:9, 2015.

[4] J. Arulraj and A. Pavlo. How to build a non-volatile memory database management system. In *Proc. SIGMOD*, pages 1753–1758, 2017.

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

[6] J. Chen, Y. Deng, and Z. Huang. HDCat: Effectively identifying hot data in large-scale I/O streams with enhanced temporal locality. In *Proc. Int'l Conf. on Algorithms and Architectures for Parallel Processing*, pages 120–133, 2015.

[7] J. Chou, J. Kim, and D. Rotem. Energy-aware scheduling in disk storage systems. In *Proc. ICDCS*, pages 423–433, 2011.

[8] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. SoCC*, pages 143–154, 2010.

[9] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu. Memory power management via dynamic voltage/frequency scaling. In *Proc. ICAC*, pages 31–40, 2011.

[10] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik. Anti-Caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013.

[11] V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-oriented compiler optimizations for partitioned memory architectures. In *Proc. Int'l Conf. on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 138–147, 2000.

[12] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Scheduler-based DRAM energy management. In *Proc. Design Automation Conference*, page 697, 2002.

[13] Q. Deng, D. Meisner, L. Ramos, T. F. Wenisch, and R. Bianchini. Memscale: Active low-power modes for main memory. *SIGPLAN Not.*, 47(4):225–238, Mar. 2011.

[14] S. Gurumurthi, J. Zhang, A. Sivasubramaniam, M. Kandemir, H. Franke, N. Vijaykrishnan, and M. J. Irwin. Interplay of energy and performance for disk arrays running transaction processing workloads. In *Proc. Int'l Symp. on Performance Analysis of Systems and Software*, pages 123–132, 2003.

[15] J.-W. Hsieh, T.-W. Kuo, and L.-P. Chang. Efficient identification of hot data for flash memory storage systems. *ACM Trans. Storage*, 2(1):22–40, Feb. 2006.

[16] H. Huang, P. Pillai, and K. G. Shin. Design and implementation of power-aware virtual memory. In *Proc. USENIX Annual Technical Conference*, pages 5–5, 2003.

[17] H. Huang, K. G. Shin, C. Lefurgy, and T. Keller. Improving energy efficiency by making dram less randomly accessed. In *Proc. ISPLED*, pages 393–398, 2005.

[18] G. Jia, X. Li, J. Wan, L. Shi, and C. Wang. Coordinate page allocation and thread group for improving main memory power efficiency. In *Proc. HotPower*, 2013.

[19] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *PVLDB*, 1(2):1496–1499, 2008.

[20] A. Karyakin and K. Salem. An analysis of memory power consumption in database systems. In *Proc. Int'l Workshop on Data Management on New Hardware*, pages 2:1–2:9, 2017.

[21] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proc. IEEE MICRO*, pages 598–610, 2015.

[22] M. Korkmaz, M. Karsten, K. Salem, and S. Salihoglu. Workload-aware cpu performance scaling for transactional database systems. In *Proc. SIGMOD*, pages 291–306, 2018.

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

[24] K. T. Malladi, I. Shaeffer, L. Gopalakrishnan, D. Lo, B. C. Lee, and M. Horowitz. Rethinking dram power modes for energy proportionality. In *Proc. IEEE MICRO*, pages 131–142, 2012.

[25] D. Narayanan, A. Donnelly, and A. Rowstron. Write off-loading: Practical power management for enterprise storage. *ACM Trans. Storage*, 4(3):10:1–10:23, 2008.

[26] Qingbo Zhu, F. M. David, C. F. Devaraj, Zhenmin Li, Yuanyuan Zhou, and Pei Cao. Reducing energy consumption of disk storage using power-aware cache management. In *Proc. HPCA*, pages 118–118, 2004.

[27] A. Sharifi, W. Ding, D. Guttman, H. Zhao, X. Tang, M. Kandemir, and C. Das. Demm: a dynamic energy-saving mechanism for multicore memories. In *Proc. MASCOTS*, pages 210–220, 2017.

[28] A. Shehabi, S. J. Smith, D. A. Sartor, R. E. Brown, M. Herrlin, J. G. Koomey, E. R. Masanet, N. Horner, I. L. Azevedo, and W. Lintner. United States data center energy usage report. Technical Report LBNL-1005775, Lawrence Berkeley National Laboratory, June 2016.

[29] R. Stoica and A. Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. *Proc. Int'l Workshop on Data Management on New Hardware*, 2013.

[30] R. Stoica, J. J. Levandoski, and P.-A. Larson. Identifying hot and cold data in main-memory databases. In *Proc. ICDE*, pages 26–37, 2013.

[31] D. Tsirogiannis, S. Harizopoulos, and M. A. Shah. Analyzing the Energy Efficiency of a Database Server. In *Proc. SIGMOD*, pages 231–242, 2010.

[32] D. Wu, B. He, X. Tang, J. Xu, and M. Guo. Ramzzz: Rank-aware dram power management with dynamic migrations and demotions. In *Proc. Supercomputing*, pages 32:1–32:11, 2012.

[33] D. Zhang, M. Ehsan, M. Ferdman, and R. Sion. Dimmer: A case for turning off dimms in clouds. In *Proc. SoCC*, pages 11:1–11:8, 2014.

[34] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-rank: Adaptive DRAM architecture for improving memory power efficiency. In *Proc. IEEE MICRO*, pages 210–221, 2008.