Memory Power Consumption in Main-Memory Database Systems

by

Alexey Karyakin

A thesis

presented to the University of Waterloo

in fulfillment of the

thesis requirement for the degree of

Doctor of Philosophy

in

Computer Science

Waterloo, Ontario, Canada, 2022

Examining Committee Membership

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

External Examiner:	Pinar Tözun Professor, IT University of Copenhagen
Supervisor:	Kenneth Salem Professor, David R. Cheriton School of Computer Science, University of Waterloo
Internal Member:	Martin Karsten Professor, David R. Cheriton School of Computer Science, University of Waterloo
Internal Member:	Ali Mashtizadeh Assistant Professor, David R. Cheriton School of Computer Science, University of Waterloo
Internal-External Member:	Seyed Majid Zahedi Assistant Professor, Dept. of Electrical and Computer Engineering, University of Waterloo

Author's Declaration

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

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

Statement of Contibutions

I am the sole author of this thesis, with the exception of parts of Chapters 2, 3, and 4.

Chapters 2 and 3 contain material that was published previously [45] with Kenneth Salem as a co-author.

Chapter 4 contains material that was published previously [46] with Kenneth Salem as a co-author.

Abstract

In main-memory database systems, memory can consume a substantial amount of power, comparable to that of the processors. However, existing memory power-saving mechanisms are much less effective than processor power management. Unless the system is almost idle, memory power consumption will be high.

The reason for poor memory power proportionality is that the bulk of memory power consumption is attributable to background power, which is determined by memory power state residency. The memory workload in existing systems is evenly distributed over the memory modules and also in time, which precludes the occurrence of long idle intervals. As a result, deep low-power states, which could significantly reduce background power consumption, are rarely entered.

In this work, we aim to reduce the memory power consumption of main-memory database systems. We start by investigating and explaining the patterns of memory power consumption, under various workloads. We then propose two techniques, implemented at the database system level, that skew memory traffic, creating long periods of idleness in a subset of memory modules. This allows those modules to enter low-power states, reducing overall memory power consumption. We prototyped these techniques in DimmStore, an experimental database system.

The first technique is rate-aware data placement, which places data on memory modules according to its access frequency. The background power in the unused or least-used modules is reduced, without affecting background power in the most-used modules. Rate-aware placement saves power and has little performance impact. Under a TPC-C workload, rate-aware placement resulted in memory power savings up to 44%, with a maximum throughput reduction of 10%.

The second technique is memory access gating, which targets background power in less-frequently accessed memory modules by inserting periodic idle intervals. Memory gating reduces power consumption of memory modules for which rate-aware placement alone does not create sufficient idleness to reduce power consumption. With gating, memory accesses to these modules become concentrated outside of the idle intervals, creating the opportunity for low-power state use. However, because it delays memory accesses, memory gating impacts performance. Higher memory power savings and lower performance impact occur in workloads with lower memory access rates. Thus, in the YCSB workload with a medium transaction rate, memory gating reduced memory power by 26%, adding 0.25 ms (30%) of transaction latency, compared to DimmStore without gating. In the more memory intensive TPC-C workload and low to medium transaction rate, gating can save 5% of memory power, adding 1.5 ms (60%) of transaction latency, compared to DimmStore without gating.

Acknowledgements

I am lucky to have Dr. Ken Salem as my adviser during my years at the University. His teaching, inspiration, and guidance shaped my approach to research and professional work. His critical feedback has helped me to improve my research, which ultimately made this thesis possible.

I thank professors, fellow students, and other members of the Database Systems Group for my experience and valuable insights that I received during meetings and discussions.

I thank the members of my committee Pinar Tözun, Martin Karsten, Ali Mashtizadeh, and Seyed Majid Zahedi for reading, providing feedback, and evaluating my work.

I thank my Mom and Dad for giving me an opportunity to make my own choices while showing me the right example.

Table of Contents

List of Figures

Chapter 1

Introduction

Main memory is a significant consumer of energy in database servers. In general computing servers, memory is considered to the second-largest power consumer after the processors, responsible for up to 40% of total server's power consumption [54]. Typical estimations of memory power consumption may not represent server configurations that maximize the amount of memory. It has been projected that increasing memory density and fully populating memory slots may cause memory power consumption to exceed that of the processors' [12].

Servers' power consumption is an important area of study that has produced practically significant mechanisms for saving energy. However, most of these mechanism target processors' power consumption. Operating systems include *power governors* to adjust processor power states, frequency, and voltage in response to changing operating conditions. In contrast, memory power optimization is less studied, and is not well supported by existing systems.

Main-memory database management systems (DBMS) represent an important use-case for optimizing memory power consumption. The use of main-memory DBMSes is expanding because they offer better performance than traditional disk-based systems. Main-memory DBMSes store the full database in main memory, thus avoiding direct and indirect costs associated with accessing external storage. A main-memory DBMS must have enough memory to accommodate the future data growth and potential variation in memory demand. Exceeding the amount of available memory may cause the system to crash or significantly degrade its performance. Therefore, it is normal for a DBMS to operate using a portion of its available memory.

However, partial memory utilization will not necessarily reduce a system's memory power consumption. As it will be shown in Chapter 2, memory power consumption in existing systems is not sensitive to the memory utilization. Similarly, memory power consumption is *non-proportional* with regard to the load i.e., transaction rate. As a result, memory power consumption is near its maximum over much of the utilization range.

Current memory technology uses *power states* as a primary mechanism to reduce power consumption when memory is lightly used. A memory module must be in the highest-powered state to fulfill requests, but can sink into one of several *low-power states* when idle. Memory power consumption in the deepest low-power state is several times lower than in the highest-powered state. We refer to the part of memory power consumption that depends on the power state as *background power*. Background power is not directly tied to average utilization and load. Reducing load will introduce idle intervals between accesses but due to the latency associated with power state transitions these intervals may be too short to allow the memory to enter a low power state. The significance of background power and low usage of low-power states are the main reasons for the lack of memory power proportionality with regard to utilization and load in existing systems.

1.1 An Approach to Saving Memory Power

The goal of this work is to study and propose mechanisms to reduce memory power consumption in main-memory DBMSes, focusing on memory power states as the primary power reduction mechanism. The focus on a narrowly defined class of applications, main memory DBMSes, allows us to develop techniques that benefit from knowing the memory organization and access patterns. To maximize the use of low-power states, a power-efficient DBMS must control data placement in physical memory and shape the flow of memory accesses to individual memory modules. Due to complexity of the software, managing all memory accesses in an arbitrary system may not be feasible, but a DBMS may identify memory regions with known access patterns and physically separate them. If such regions constitute a large portion of all memory, focusing on these region may reduce average power consumption.

1.2 Thesis Organization and Research Contributions

The thesis is structured as follows. In Chapter 2, we present the necessary background on memory architecture and how it affects memory power consumption. We also define a

simple memory power model, based on power state residencies, that can be used to explain memory power consumption as a function of workload characteristics.

In Chapter 3, we focus on understanding memory power consumption in main-memory DBMSes. We experimentally investigate memory power consumption under transactional and analytical database workloads. We measure actual memory power consumption and demonstrate that it is independent of the database size, and that it is non-proportional with respect to database load. By measuring power state residencies and using the power model to break memory power consumption down to components, we explain the effects of workload characteristics on measured power. We show that background power makes up the bulk of total power consumption. We also show that low power states are rarely used even when the database is small and load is low. The reason for this is that memory accesses tend to evenly spread over all memory and most of the intervals between accesses are too short for low-power state transitions.

The findings from Chapter 3 inform our approach for designing a memory power-efficient system. As the memory access patterns in existing systems do not allow for significant use of low-power states, we target background power through the use of software-based techniques described in the Chapters 4 and 5.

In Chapter 4, we describe a design of DimmStore, a power efficient main-memory DBMS, which uses the techniques of *rank-aware memory allocation* and *rate-based placement*. To reduce background power, DimmStore divides the database into frequently and infrequently used regions and places different regions on different DIMMs. The DIMMs that store the region with a low access rate see significantly longer idle intervals between memory accesses, which allows them to increase their low-power state residency and reduce background power. If the database size is smaller than the amount of memory available, some DIMMs will always stay in the lowest power state. Concentrating most of the memory load in a few frequently accessed DIMMs does not significantly increase power consumption of this region because the baseline low power state residency was low anyway. The DimmStore prototype was evaluated using the YCSB and TPC-C transactional workloads. Compared to the baseline, DimmStore reduces memory power consumption by up to 44%, with peak throughput degradation below 10%.

In Chapter 5, we propose a second technique called *memory access gating*, which is complimentary to rate-based placement and provides additional power savings. In DimmStore, memory access gating is applied to the less frequently accessed database region, where the access rate is already reduced as a result of rate-based placement. With gating, longer idle intervals are created during workload by temporary prohibiting (gating) access to that region's DIMMs. Memory accesses which would have occurred during these gated

intervals are temporarily delayed. The longer idle intervals introduced by gating cause an increased use of low power states and reduce background power.

Memory gating is an intrusive technique that may negatively impact the performance of the system. We evaluate the effect of memory access gating with the YCSB and TPC-C workloads by measuring memory power consumption and transaction latency. Higher memory power savings and lower performance impact occur in workloads with lower memory access rate. Thus, in the YCSB workload and medium transaction rate, memory gating reduced memory power by 26%, adding 0.25 ms (30%) of transaction latency, compared to DimmStore without gating. In the more memory intensive TPC-C workload and low to medium transaction rate, gating can save 5% of memory power, adding 1.5 ms (60%) of transaction latency, compared to DimmStore without gating.

Chapter 2

Background

In this chapter, we provide necessary background on Dynamic Random Access Memory (DRAM) and its power characteristics. We describe memory organization in servers and memory architecture, focusing on power states.

In a simple random-access experiment with an instrumented server, we demonstrate the use of low-power states by the memory controller. The power state policy in our server uses an idle timer to control when to enter a low-power state. We estimate the values of timeouts for the Self Refresh and Power Down states.

Later, we introduce a simple power model used to break down power consumption into components, such as active and background power.

Finally, we describe the memory power measurement system, used in the experiments, and reference existing work related to methods of memory power estimation.

2.1 System Memory Architecture

Servers are typically built using a Non-Uniform Memory Architecture (NUMA), consisting of several NUMA nodes. Each NUMA node is a processor connected to local memory. Nodes can communicate to each other and can access other nodes' memory via inter-processor data links.

Modern processors have memory controllers built in. A typical server processor may have several (up to 4 or more) independent memory controllers.3 Therefore, the processor packages expose memory interfaces, which are used to directly connect the built-in controllers to the processor's memory.

The memory interfaces are defined by JEDEC standards for DRAM, such as DDR3 [6] and DDR4 [7]. The standardisation of memory and its interfaces allows for interoperability between processors and memory made by different manufacturers. It also means that the behaviour and power consumption patterns of different models of DRAM are similar.

The design of the memory interface in a NUMA node is largely motivated by the desire to achieve high performance and configuration flexibility. Each NUMA node has several (up to 24) slots to install memory modules (Dual-Inline Memory Module, DIMM). The DIMM data bus is 64 bits wide and the signals are bidirectional, changing the information flow direction dynamically for reading and writing.

To support high bandwidth, server memory interfaces work at the clock rates between 800 and 1600 MHz and use Double Data Rate signalling, sending a bit at both rising and falling edges of the clock signal, effectively achieving the data rate of up to 3200 mbits per second per data line. Achieving such high data rates in the current design of the memory interface is challenging, and, unfortunately, causes the interface to be power hungry. The relatively long signal distance between the CPU and memory modules, needed to accommodate several DIMM slots, makes it behave as a *transmission line*. To avoid signal reflections, the data lines are electrically terminated in the DRAM device and in the controller, causing substantial amount of power to be dissipated in the termination.

Another aspect of the memory interface design that increases its power consumption is related to ensuring accurate signal timing at a high data rate. For example, at a rate of 1866 mbits per second, which is in the lower range of the available rates in a modern server, the bit duration is only 536 *picoseconds*. To allow for necessary timing margins, the signal has to be sampled in the middle of the bit data window with an accuracy of about 100 picoseconds. To achieve such timing accuracy when the signal propagation times are affected by even small differences in the length of the wires, the variations in signal propagation due to changes in temperature or supply voltage, DDR memory utilizes a *Delay Lock Loop* circuit (DLL) in each of the DRAM devices, as well in the controller. The DLL adjusts the timing of the output data signals to precisely align them with the edge of the input clock received from the controller. DLL uses a control loop that needs a relatively long time, an order of 0.5 µs, to match the signal delay to the input clock edge. DLL also consumes significant amount of power, which is a price to pay for the high data rate in modern server memory interfaces. Historically, memory interface power consumption increases as data rates become higher and is projected to increase further with newer generations of memory [48].

Due to having multiple memory slots, modern servers allow the end user to change the amount of memory in a server to accommodate the application requirements. Electrical

and signal quality issues limit the number of memory modules that can be attached to each interface. To enable larger amounts of memory to be installed, processors designers started to implement multiple memory interfaces in a physical processor, called *memory channels*. In a typical modern processor, there are between one and four memory channels, the upper limit being due to the processor pin count constraints. With multiple memory channels, the processor can increase its available memory bandwidth by transferring data in parallel over multiple channels.

Multiple memory channels allow multiple threads to access memory in parallel, maximizing total memory bandwidth. However, applications cannot always evenly distribute memory workload between multiple threads. To increase the available bandwidth even for memory accesses from a single thread, memory controllers map consecutive blocks in the physical address space to different channels. This address mapping strategy is called *memory interleaving*. The unit of memory interleaving is usually a block of one or a small number of processor cache lines. Therefore, memory interleaving can effectively parallelize memory access even for sequential data transfers larger than a few hundreds of bytes.

2.2 Memory Microarchitecture and Power States

Internal DRAM architecture is important for understanding DRAM function and, therefore, power behaviour.

The basic architecture of modern DRAM devices has not fundamentally changed since the 3rd generation DRAM circa 1973 [2]. A DRAM device consists of a two-dimensional cell array, where bits are stored as electrical charges, and control and interface circuits, which direct the device's operations and allow memory to communicate data to the outside world. The two-dimensional array is formed by *rows* and *columns*, which dictates a two-step access cycle. First, the row address is received from address lines and the corresponding row is copied to the *row buffer*. Copying is achieved by *sensing* the charges in the array cells by row sense amplifiers. This operation is called row activation (ACTIVATE) and it is strobed by the RAS signal. Once the bits of a selected row are in the row buffer, a particular word in the buffer can be read or written using the column address, strobed by the CAS signal (READ/WRITE).

Sensing a one-transistor DRAM cell is a destructive operation as the small cell charge is consumed in the process. Therefore, another operation, called PRECHARGE, is required to write back the contents of the row buffer, potentially modified by writes, to the array. Even in the absence of row accesses, the cell charge dissipates due to leakage. To ensure

data preservation, a separate REFRESH operation is periodically performed on each row in the array.

The cell array and row buffers are *asynchronous* circuits and their energy consumption occurs due to moving the change between the cells and buffers during ACTIVATE, PRECHARGE, READ/WRITE, and REFRESH operations. The energy used by the ACTIVATE, PRECHARGE, and READ/WRITE operations can be attributed to reads and writes, and the energy for the REFRESH operations is consumed in order to preserve memory content.

In contrast, the control and interface circuits are *synchronous*, as they accept and internally distribute clock signals. The presence of an active clock signal causes the circuits to consume energy even when they do not perform any useful work. This component of power consumption is proportional to the clock frequency. To reduce power consumption of the control and interface circuits, some of them can be temporarily disabled by removing the clock signal. In those cases, the DRAM device is said to enter one of the *low-power states*.

Memory power is classified into two broad categories: *active* and *background* [4], [19], [81]. Power consumed due to memory performing ACTIVATE, READ/WRITE, and PRECHARGE operations is considered active power. Active power is directly proportional to the rate of these operations received by DRAM from the memory controller. This rate is determined by the memory bandwidth demand of the running programs, i.e. the workload.

Power consumption determined by the current power state of the DRAM is considered background power. Average background power over a time interval depends on the composition of the power states during this interval, i.e. memory power state *residencies*. Power state residency is a fraction of time the DIMM spends in that state during a measurement interval. Power consumed due to DRAM refresh is also included in the background power category because the rate of refresh is always constant, so its average power is constant as well.

DRAM behaviour can be described by a state machine [7], where each state is associated with distinct background power consumption. Due to DRAM device complexity, the number of distinct states is large. For example, the "simplified state diagram" in [7] shows more than 20 states. DRAM manufacturers specify typical consumption in many of the states defined by the DDR4 standard. Using this data and knowing state residencies, it is possible to calculate the total background power consumption, using architectural DRAM power models (see Section 2.6.1). Architectural models simulate execution of a workload and produce a detailed execution trace of each device in the system. Using architectural

State	Power, W	Exit latency, ns	DDR4 States
StandBy	1.5	-	Active StandBy and Precharge StandBy
Power Down	0.9	~ 50	Active Power Down and Precharge Power Down
Self Refresh	0.35	~ 500	Self Refresh

Figure 2.1: Power states reported by performance counters

models is time consuming and requires a highly detailed description of each component in the simulated system.

In this work, we measure state residency using performance counters in the memory controller of Intel Xeon processors [8]. This way, we can obtain actual state residencies in a real system without the effort associated with modelling. However, states reported by performance counters are coarse-grained. Each state residency, obtained from performance counters, corresponds to a mix of state residencies as per the DDR4 specification [7].

The short list of memory states used in this work, with their approximate power consumption and exit latencies, is shown in Figure 2.1. During the StandBy state, all control and interface circuits are enabled and clocked, and DRAM can immediately receive and execute commands from the controller. The power consumption in the StandBy state is the highest.

During the Power Down state, the output buffers are disabled, which reduces DRAM power consumption. However, they can be enabled after a short ($\sim$ 50 ns) delay. In the Self Refresh state, all of the interface circuits are disabled with the exception of the Clock Enable signal receiver, which is necessary to bring it out of the Self Refresh state. In particular, the Delay Locked Loop (DLL) circuit, used to ensure integrity of high-speed signals between the DRAM and controller and characterized by relatively high power consumption, is stopped. Restarting the DLL requires a significant time ($\sim$ 500 ns). In the Self Refresh state, the DRAM does not receive REFRESH operations from the controller, but performs array refresh using an internal counter.

The memory controller tries to minimize memory power consumption by switching DIMMs into a lower-power state when they are idle. Since the use of lower-power states may result in some performance degradation due to exit latencies, the controller has to balance energy consumption and performance. A particular algorithm used by the controller to switch power states based on the workload is called *power management policy*.

A simple and widely used power management policy is based on the concept of idle timer. For each power state, the controller implements a timer which is reset on every

memory access and counts during the idle time. When the timer reaches a certain threshold (timeout), specific to each power state, the controller initiates the state transition. The timeout values are lower for "shallow" power states, which have lower exit latency but less energy advantage.

2.3 Power State Policy Experiment

For the Intel Xeon processors used in this work, the power management policy is not documented. To get an initial understanding of how power states are used, we set up a simple experiment. In this experiment, one processor core was generating single cache-line memory accesses to one of the DIMMs, with a precisely controlled interval between accesses. To capture possible NUMA effects, we repeated the experiment once for each processor in the system, with the workload generator thread bound to a core in that processor. We also repeated the experiment targeting one of the DIMMs in each of the processor. We will refer to memory accesses from a thread running in one processor to memory attached to this processor as *local* and to memory attached to the other processor as *remote*. To minimize the effect of processor caches, the addresses were generated at random over the whole DIMM. We repeated this experiment for multiple values of the inter-access interval. During each run, we measured DIMM's power state residencies, as reported by RAPL counters. The resulting graph that captures these metrics as functions of the access interval is shown in Figure 2.2 for local accesses and Figure 2.3 for remote accesses. We noticed that the results differ when accessing the same DIMM from the locally-attached processor compared to accesses from the other processor.

There are a number of observations that can be made using this graph. First, the Self Refresh state is never entered when the inter-access interval is shorter than a threshold, which is approximately 200 µs in experiments with remote accesses and 1000 µs for experiments with local accesses. Since Self Refresh is the state with the lowest power consumption, a power-efficient system must maximize the number of intervals longer than these thresholds.

Second, once any of the two low-power states is entered, their state residency increases with the increase in the inter-access interval. For the Power Down state, its residency increases until the Self Refresh state starts to take over. The Power Down state was observed only in remote accesses.

Although the actual algorithm used by the memory controller to control power state transitions is unknown, these observations are consistent with a timer-based policy [34].

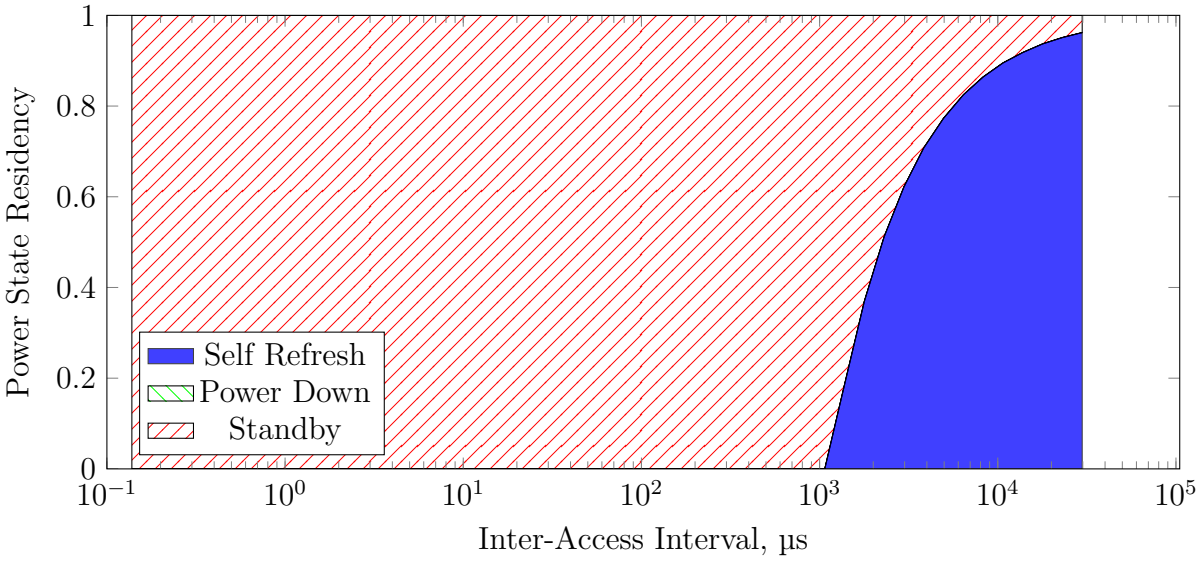

Figure 2.2: Power state residencies vs. inter-access interval, local memory accesses

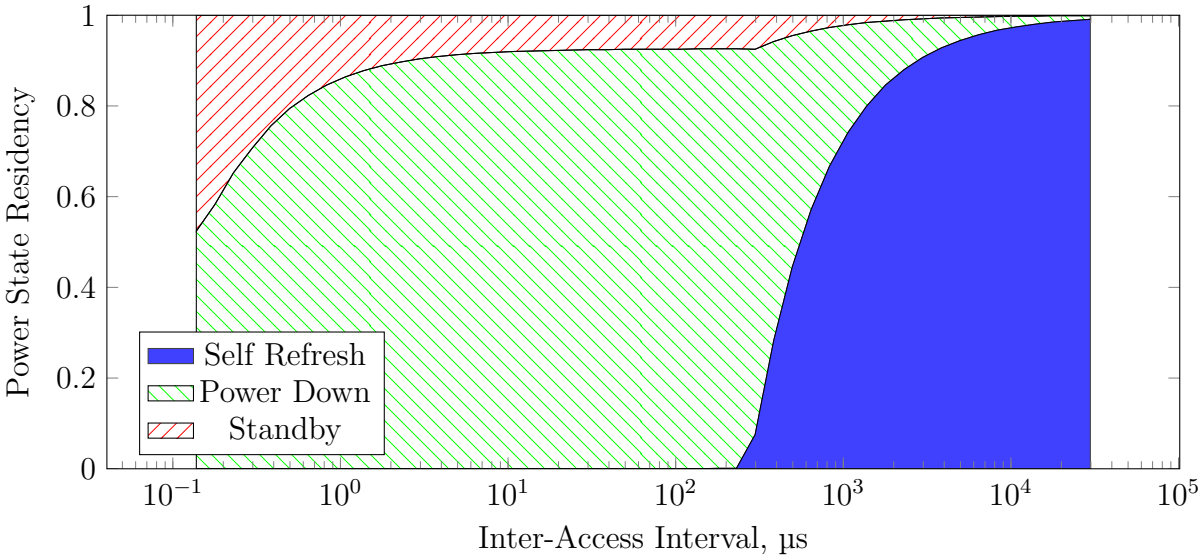

Figure 2.3: Power state residencies vs. inter-access interval, remote memory accesses

In a timer-based policy, the memory controller implements a counter (timer) which measures the time elapsed since the last access. Once this time exceeds a constant threshold (timeout), the controller triggers the transition to the low power state. In case of multiple states, each state has its own idle threshold and deeper power states take precedence. The power state timeout value can be estimated by the shortest inter-access interval when this power state starts being used.

2.4 Measuring Memory Power

For the experimental work in this thesis, I relied on direct measurement of the power consumed by the DIMMs in the test server, as described in this section. In addition, the measured power consumption was analyzed using a power model. Specifically, the model was used to estimate the background and active power components. This power model is presented in Section 2.5.

To accurately measure memory power consumption, I built a system for individual DIMM power measurement and installed it in the test server. The measurement system directly measures power consumption in each of the eight DIMMs installed in the server. The power readings are generated in real-time while the server is under load.

To obtain the value of power consumption in each DIMM, the system measures current in the V_{CC} and V_{PP} power rails on the DIMM with a 24-bit resolution. The maximum sampling rate is 78 thousand per second, however, a low rate of 612 samples per second was used during experiments. Due to the Sigma-Delta architecture of the Analog to Digital Converter (ADC) used, each collected sample is an integral of the analog value during the sampling interval.

To obtain power measurements, the collected values of current are multiplied by the nominal voltages in the DIMM power supplies, $V_{CC} = 1.2V$ and $V_{PP} = 2.5V$ [7]. Measuring the voltages would increase the power measurement accuracy, but would require doubling the number of measurement channels.

The measurement system consists of eight DIMM risers with current sensing capability, two 8-channel ADC boards, and a microcontroller board for ADC control and transferring collected data to the computer. For current sensing, each DIMM riser has current sense resistors installed in the V_{CC} and V_{PP} power rails. The differential voltage on the sense resistor is measured by a ADC board, using the Microchip MCP3914 simultaneous-sampling Sigma-Delta ADC [9].

Current measurement was calibrated by running a DC current through the sense resistors at two values of current and measuring the actual value of current with a digital multimeter. The two calibration points were used to obtain the end-to-end offset and gain of the current measurement system, which cancels the error due to the resistance of the traces on the DIMM risers, tolerance of the sense resistors, and linear error in the ADC. The calibration offset and gain were used in software during converting the raw data to the value of the current.

2.5 Memory Power Model

I used the memory power model presented in this section to explain measured memory power consumption as a function of the memory workload. Its explanatory ability relies on breaking down the measured total power consumption into several components, based on workload characteristics. I use this model mainly to determine the amount of active and background power. However, it is possible to further drill down to quantify the effects of individual workload characteristics on power.

The model estimates power components for one DIMM. The total memory consumption is the sum of the consumption of all DIMMs. As in other models ([4], [19], [25]), we model DIMM power as a linear function (weighed sum) of workload metrics. However, the set of used metrics differs from other models and is chosen based on available performance counters in the memory controller. The model was described previously [45] in a slightly different form.

The workload metrics used as model inputs are memory operation frequencies and power state residencies in the DIMM and in the individual memory ranks within the DIMM. Specifically, the model inputs include counts of the number of Activate/Precharge cycles, Read, and Write operations in the DIMM per second. The model inputs also include measured power state residencies, which represent the fraction of time during which the DIMM or rank is in the particular state, represented as a value in the range $[0-1]$. The workload metrics are listed in Figure 2.4.

The DIMM power is modeled as a sum of terms, each of which represents the contribution of a power state or type of memory operation to the total power consumption. The coefficients in each term are either operation energies or power in each of the states.

Total power is modelled as the sum of *Active* and *Background* power:

$$P = P_{bk} + P_{act} \tag{2.1}$$

Ref	Units	Description
T_{SR}	-	DIMM's Self Refresh state residency
T_{SB}	-	DIMM's StandBy state residency, at least one rank has active CKE
T^i_{CKE}	-	Active CKE duty cycle for rank i
N_{act}	s^{-1}	Frequency of Activate/Precharge cycles
N_r	s^{-1}	Frequency of column Read operations
N_w	s^{-1}	Frequency of column Write operations

Figure 2.4: Memory model workload metrics

Ref	Units	Value	Description
P_{SR}	W	0.36	Power in Self Refresh state
ΔP_{PD}	W	0.53	Additional power in Power Down state, over Self Refresh
ΔP_{SB}	W	0.67	Additional DIMM power in StandBy state, over Power Down
ΔP^i_{CKE}	W	0.098	Additional power when i-th rank has active CKE, over DIMM StandBy
E_{act}	nJ	5.97	Energy of Activate/Precharge cycle
E_r	nJ	6.63	Energy of column Read in active bank
E_w	nJ	8.74	Energy of column Write in active bank

Figure 2.5: Memory model coefficients

The active power is modelled as the sum of power consumed by Activate/Precharge, Read and Write operations:

$$P_{act} = N_{act}E_{act} + N_rE_r + N_wE_w \tag{2.2}$$

Here, the Ns are operation frequencies from Figure 2.4 and the Es coefficients are energy costs of the corresponding operations from Figure 2.5.

Background power, in turn, is modelled as a sum of four terms:

$$P_{bk} = P_{SR} + (1 - T_{SR})\Delta P_{PD} + T_{SB}\Delta P_{SB} + \sum_{i \in ranks} T^i_{CKE}\Delta P^i_{CKE} \tag{2.3}$$

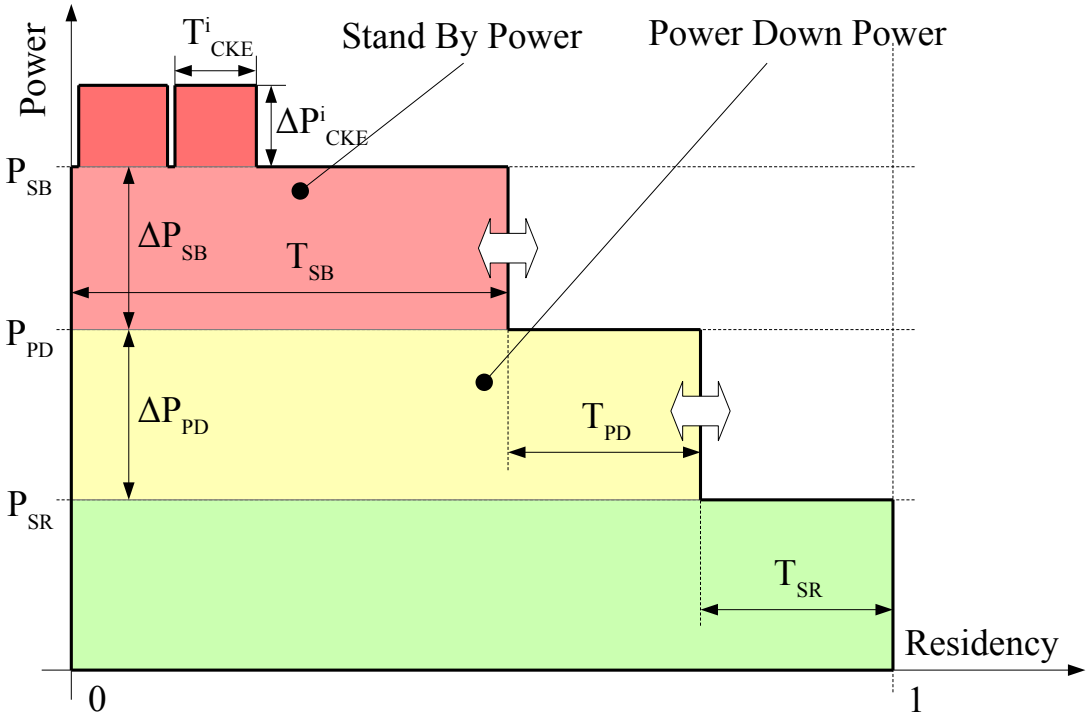

Figure 2.6: Background power model as a sum of per-state components

The background part of the power model has been reformulated compared to [45] to make is possible to estimate the contribution of individual states to the total background power consumption. The idea behind structuring the background power into state-dependent components is illustrated in Figure 2.6, where each term in (2.3) is shown in its own colour. The total power is the area under the whole coloured figure, bounded by a bold line.

The power model distinguishes between three power states: Self Refresh, Power Down, and StandBy. The Self Refresh state applies to the DIMM as a whole, including all the DRAM ranks and additional per-DIMM circuitry. In Figure 2.6, Self Refresh power P_{SR} is the lowest possible power consumption of the DIMM and always applies regardless of

the state the DIMM is in. Power components for other, higher-power states are defined as increments over the previous lower-power state.

When the DIMM is not in the Self Refresh state, each of its DRAM ranks can be either in the Power Down or StandBy state, independently from the other ranks on the same DIMM. The DIMM is considered to be in the StandBy state when at least one of its ranks is in the StandBy state. The activation of the rank's StandBy state is controlled by its CKE (Clock Enable) signal. The duty cycle of the CKE signal can be collected using the memory controller's performance counters, separately for each rank (T^i_{SB}), as well as the duty cycle of the "logical OR" of the CKE signals for all ranks in the DIMM (T_{SB}).

The DIMMs in our test server consist of two ranks, and the DIMM's power consumption is different when none, one, or both ranks are in the StandBy state. Therefore, the contribution of the StandBy state in the power model is represented by three components. ΔP_{PD} is the additional power consumed when the DIMM is not in the Self Refresh state. ΔP_{SB} is the additional power consumed when *at least one* rank in the DIMM is in the StandBy state. Finally, for each rank with an active CKE signal, the per-rank component ΔP^i_{CKE} is added.

The values for model coefficients were obtained in a calibration step, using two synthetic workloads. One of the calibration workloads is similar to the one described in Section 2.3. This workload consists of random memory accesses, with a constant interval between accesses. This workload was run multiple times, varying the interval between accesses, and the target DIMM. The other calibration workload is a linear memory scan over a large address range. This workload has no parameters and was repeated for each DIMM in the system. During each run of a calibration workload, we measured memory power consumed by the accessed DIMM, and collected power state residencies and operation counts. The model coefficients were calculated by fitting a linear function of power state residencies and operation counts (model inputs) to the measured power values, using the least squares method. The model coefficients and their estimated values for the DIMMs used in the experiments are listed in Figure 2.5.

2.6 Other Techniques for Measuring and Modelling Memory Power

This section summarizes other work on memory power modelling (Subsection 2.6.1, Subsection 2.6.2), simulation (Subsection 2.6.3), and estimation (Subsection 2.6.4).

2.6.1 Architectural Models

Low-level models for a particular device are created by DRAM manufacturers once the devices is in a late stage of development. These models are sometimes called *transistor-level* models and describe all aspects of the device design and are accurate. Manufacturers typically do not release those model publicly. However, the results of modelling on standardized test patterns may be published in a device datasheet instead of actually measured data. Since a model is created for each device, it cannot be used for modelling experimental or future devices, or even devices in early stages of development. There have been efforts to create low-level models that could be adjustable for a wider class of DRAM. Vogelsang [74] proposed a low-level model that offers the same level of accuracy as a transistor-level model. However, in order to implement the model, one has to provide more than 70 parameters, which is the cost of the model's generality. While some of the parameters, such as frequency, timings, and supply voltages, are easy to obtain, others depend on the used technology and materials. Nevertheless, the model can be very useful to assess the trends in future DRAM power consumption.

2.6.2 Datasheet-based Models

DRAM standards define a set of standardized test patterns, for which DRAM manufacturers provide current consumption data in devices' datasheets. The DDR3 standard [6] includes I_{DD} current specifications for 15 test conditions, while DDR4 [7] includes more than 50 such conditions for I_{DD} and I_{PP}, which reflects the increased complexity of DRAM technology. The datasheet specifications can be a result of measurement on a set of devices or be derived from manufacturer's modelling. Describing DRAM power characteristics with a standard current specification would be much more concise than doing so using a low-level model. As a result, this approach is widely used in DRAM power modelling. The downside of datasheet-based current specifications is that transferring them to realistic memory workloads requires a significant amount of guesswork. For example, DDR3 includes the specifications of Precharge Standby current (I_{DD2N}), when all banks are closed, and Active Standby current (I_{DD3N}), when all banks are open. However, it is impossible to tell directly from the datasheet values the active current with a partial number of open banks.

A widely used datasheet-based model of DDR3 DRAM power consumption was published as a Technical Note by Micron [4]. The model takes the average current values from the datasheets and decomposes them to produce power consumption estimates for specific workloads. For the workload description, the model takes percentages of time when a rank

is in a certain state (Precharged vs Active, StandBy vs Power Down), and the number (frequency) of Activate, Precharge, Read, and Write operations.

The accuracy of the Micron model was questioned by Schmidt et al [65]. The major issues were that the state transition effects are ignored and I/O power is overlooked. In DRAM, switching between power states takes a finite time, and requires a certain amount of energy, which may not be directly related to energy consumption of either of the states. As a particularly heinous example, it was found that every time after transitioning to the Self Refresh state DRAM performs a refresh cycle, which energy cost is not accounted for by the Micron model. As a result, power savings of a workload that uses the Self Refresh state are overestimated and the difference is larger when the state state occurs more frequently.

The I/O power modelling is considerably more complicated than the Micron model can accommodate. Memory I/O power modelling, as well as timing, is addressed by the CACTI [72] simulator. Based on a user-supplied description of the physical topology of the memory interface and memory traffic, CACTI performs a circuit-level simulation of the interconnects and produces timing and power estimations. CACTI can be used to evaluate existing, experimental, or future memory channel topologies such as hybrid DRAM/NVM.

A major drawback of DRAM power estimations based on datasheet specifications is related to accuracy of these specifications. They are mostly intended for system builders e.g. to design adequate system cooling and not for measurements per se.

2.6.3 Simulation

In order to apply a memory power model to a specific workload, it is important to understand how memory is accessed. One way to characterize memory access is to run the workload under a system simulator. The simulator executes the processor instructions from the workload, and emulates the behaviour of the CPU, caches, and a memory interface. A number of system-level simulators exist, including GEM5 [16], PTLSim [80], and others. Most system-level simulators' capabilities do not extend past CPU caches or a simple memory channel. Therefore, a separate DRAM simulator, such as DRAMSim [75], or memsim [63] is needed for accurately modelling of DRAM aspects. The result of such simulation is a memory trace, describing time-based low-level memory events such as row activates, data transfers, and state transitions. This resulting memory trace can be either fed directly into the power model, or first aggregated to produce time averages, depending on the model requirements. Some DRAM simulators already integrate a memory power

model, for example, DRAMSim, which uses the Micron model. If the existing memory model is inadequate or not included, another power model is used.

2.6.4 Hardware Estimation

A significant disadvantage of system simulation is its high computational cost. As a result, many studies limit the simulation interval to a few million instruction, which negatively impacts the accuracy. An alternative to using simulation for obtaining memory access information is collecting this information from a live system using *performance counters*. Many processors implement various counters to track events such as branch misprediction or cache misses. Their primary purpose is performance profiling. However, some processors include counters for the memory subsystem as well. Intel Xeon [8] includes a number of counters implemented in the integrated memory controller. The counters may be used to count DRAM Activate, Precharge operations on a per-rank basis, and CAS operations (corresponding to reads and writes) on per-bank basis. Additionally, the time spent in the Power Down, Self Refresh state is also available for reporting.

The Intel XEON processors go even further than just counting DRAM operation. They also include a power model for the DRAM subsystem, integrated in the Running Average Power Limiting (RAPL) framework [25]. The RAPL hardware collects relevant performance counters, runs the power model, and reports accumulated energy values through hardware registers. The RAPL power model, its accuracy, and calibration procedure have not been published. The accuracy of RAPL DRAM power estimations was investigated by Desrochers et al [31]. They compared RAPL energy values with actual measured power in an instrumented server, under a variety of workloads. The accuracy of RAPL energy estimation varied between different servers, but was in general within 20% of the real value. The results matched well when the load was higher. The worst case difference (38%) appeared in one of the systems in the idle state. Overall, RAPL is a valuable tool for DRAM power estimation in live experiments, especially considering the difficulty of adding power measurement instrumentation to a system. Compared to simulation, RAPL is more likely to produce a reliable estimation for a certain workload, because modelling cannot accurately emulate all system behaviour and requires selection of a number of input parameters. RAPL estimation is also available in real time, even for the program under test.

Chapter 3

Memory Power Consumption in Database Workloads

In this chapter, we study memory power consumption for two types of workload: transactional (OLTP) and analytical (OLAP). We characterize memory power consumption *empirically*, by measuring memory power while running the workloads on an experimental server equipped for DIMM power measurements. To study these two types of workloads, we run the TPC-C and TPC-H benchmarks, respectively. These benchmarks have distinct data access patterns. TPC-C is dominated by index access and TPC-H is characterized by large scans and aggregations. We focus on memory-resident workloads. Therefore, in all experiments we ensured that the database fits in main memory and there is no significant disk activity.

First, we report memory power consumption as a function of load. For each workload, we perform multiple test runs varying the transaction rate (in TPC-C) or the number of concurrent queries (in TPC-H). Second, we show how memory power consumption changes with the database size by repeating the experiments for different sizes of the database.

Our results indicate that memory power consumption is not sensitive to the database size, i.e, memory consumes as much power when the database is small as it does when the database is large. Memory power consumption is only slightly sensitive to load in a memory-light transactional workload and more sensitive in memory-heavy analytical workload.

Our analysis also shows that memory power consumption is mostly attributable to background power, which is not directly affected by load. This result explains the weak

memory power sensitivity to database size and load. Any technique for memory power optimization must thus target background power.

Finally, we estimate the power and performance effects of memory interleaving in transactional (OLTP) and analytical (OLAP) workloads. Memory interleaving spreads the memory traffic across all the DIMMs in the system, increasing background power consumption and making memory power optimization difficult. For each workload, we repeat one experiment with and without interleaving. In each case, we report peak throughput and power consumption. Although it is regarded as a performance optimization, memory interleaving has negligible effect on transactional workload performance and small (around 10%) effect on analytical workload.

3.1 Server Configuration

For the experiments in this section, we used a customized test server equipped for DIMM power measurements. The server has dual 8-core Intel Xeon E5-2640 v3 ("Haswell") processors, running at 2.60 GHz, on the Asus Z10PE-D16 motherboard with AMI BIOS dated 01/25/2016. Each processor has four memory channels, with two DIMM slots in each channel. We populated one DIMM slot in each channel with a dual-rank 16GB DDR4-1866 DIMM, totalling 128 GB. We used default BIOS settings in all experiments.

We measured power consumption of each DIMM using the system described in Section 2.4. Due to the space constraints imposed by the current sensors installed in each of the DIMMs slots, every other DIMM slot in the system was left unpopulated. As a result, the memory power consumption in this test server is lower than when all the memory slots are used.

3.2 TPC-C Results

We used Shore-MT [43] for the TPC-C experiments. Shore-MT is a research storage manager, optimized for multi-core systems. It implements a traditional buffer pool with a variant of the CLOCK page replacement policy and ARIES transaction logging and recovery. Shore MT comes with a set of benchmarking tools, called Shore Kits. We used the Shore Kits implementation of the TPC-C workload for our experiments.

In TPC-C, the database scale factor was varied between 100 and 600 warehouses, which translates to an initial database size of approximately 12.5 to 81 GB. Each experiment

consisted of a database generation step and three 30-minute runs. Before each run, the database was restored from a saved copy and the Shore Kits process was restarted. We divided each measurement run into 15-second intervals and collected performance and power data for each interval. Since each run started in cold state, we considered intervals that did not reach 80% of the target throughput as warm-up and discarded the results obtained during these intervals. To reduce the effect of database growth in TPC-C due to data insertion, we only report the results of the first 10 "warm" intervals (2.5 minutes). The database grew between 0.6 and 4 GB during each experiment, depending on the transaction rate. Although database growth introduces variability in the database memory footprint in each experiment, this effect is small because database growth during each run is small compared to the size of available memory.

For TPC-C, we modified the workload generator to insert uniform random think times, with controllable mean, between requests. First, we determined the nominal maximum transaction rate by running the experiment with no think time using the smallest database size. On our system, this was approximately 300000 tpmC. In actual experiments, the think times were then calibrated to produce target throughputs ranging from 1/8 of the maximum to the maximum, in 8 steps. The load generator used 1000 client threads and 16 worker threads. When presenting measurement results as a function of throughput in Section 3.2, we normalize all throughput values to the nominal maximum value.

3.2.1 Memory Power Under TPC-C

For our TPC-C experiments, Figure 3.1 shows the measured average memory power consumption as a function of load (transaction throughput), for three different database sizes. Memory power consumption is highly non-proportional with respect to load. The highest transaction rate we tested is about 8 times higher than the lowest rate. Over this range of workload intensity, memory power grows linearly, but only by about 23%.

Figure 3.1 also shows that, perhaps surprisingly, memory power consumption is not affected by the database size. That is, if we run TPC-C transactions at the same rate against databases of two different sizes, the total memory power consumption is the same. There are several reasons for this. the first reason is memory interleaving, which changes the mapping of the server's physical address space to the DIMMs. Interleaving results in a fine-grained distribution of physical addresses across all of the DIMMs for a given socket. Hence, the memory load is spread across the DIMMs as well.

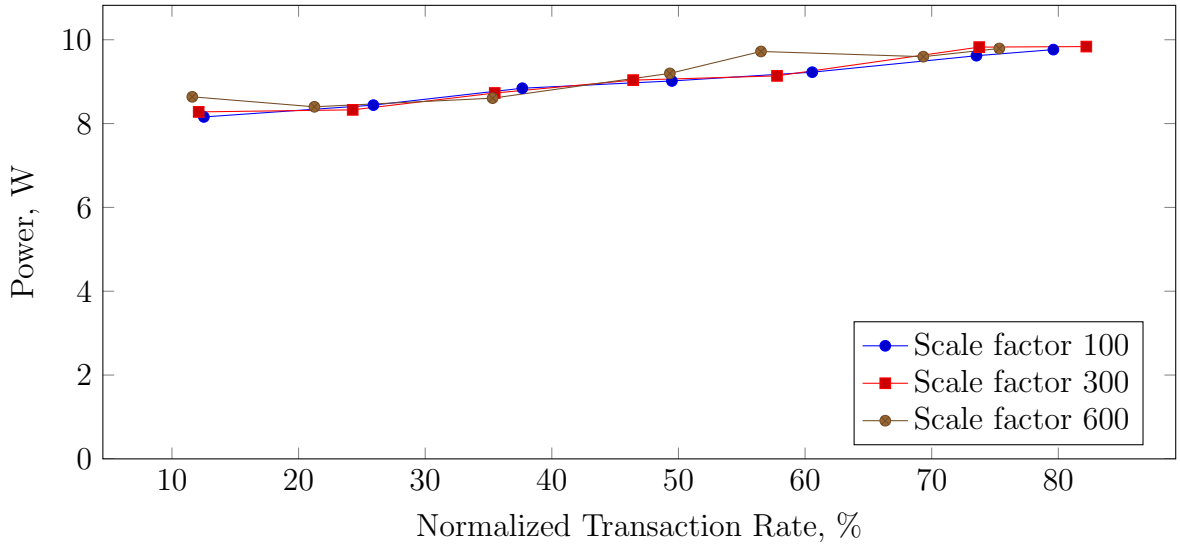

Figure 3.1: Average DRAM power in TPC-C vs. normalized transaction rate, for three database sizes

Second, even without interleaving, the operating system does not try to contain memory allocations in a small number of DIMMs. The operating system does not know the DIMM to address space mapping and cannot take it into consideration when allocating memory.

To further analyze our results, we used the memory power model that was presented in Section 2.5. Figures 3.2, 3.3, and 3.4 illustrate the accuracy of the power model by comparing the measured total memory power consumption with the consumption predicted by the model, for the small (SF 100), medium (SF 300), and large (SF 600) database sizes. For TPC-C, the model slightly overestimates the measured power throughout the load range, but the estimation error is small, with a maximum difference of about 10%.

Using the power model with collected state residency and operation counts, we can break total memory power consumption into background and active components. Figure 3.5 shows the results for the small database size. Memory power consumption is almost entirely attributable to background power. Both background power and active power increase with workload intensity, although the former increases even more than the latter. Even at the maximum load we tested, active power represents a very small fraction (less than 10%) of total power consumption. Active power is low because only a fraction of available memory bandwidth (approximately 120 GB/s) is utilized by TPC-C as shown in Figure 3.6 shows measured memory read and write bandwidths, as a function of load.

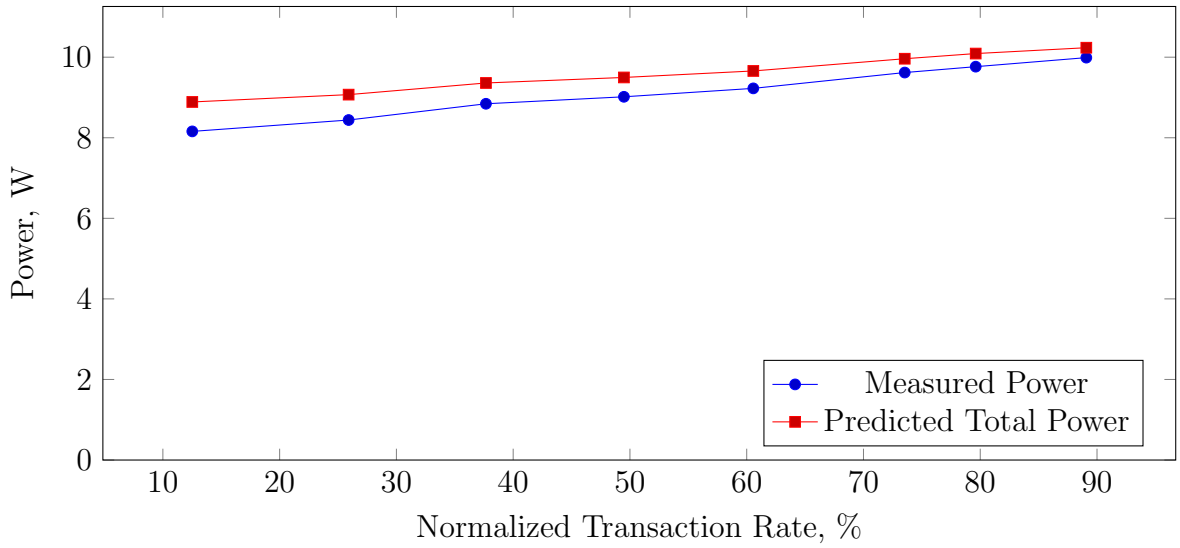

Figure 3.2: Actual and predicted average DRAM power in TPC-C vs. normalized transaction rate, small database (SF=100)

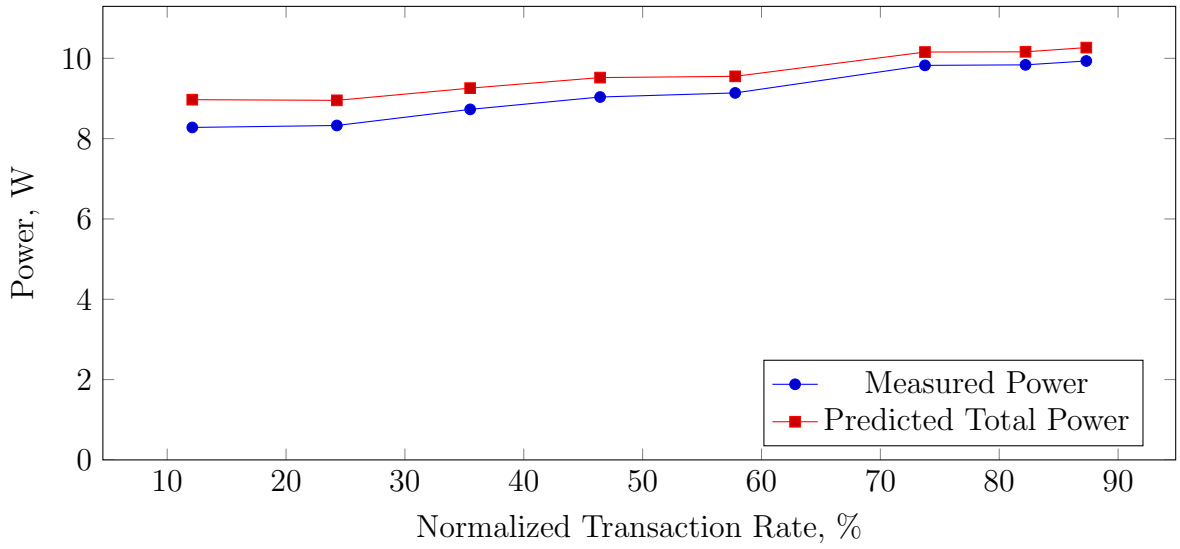

Figure 3.3: Actual and predicted average DRAM power in TPC-C vs. normalized transaction rate, medium database (SF=300)

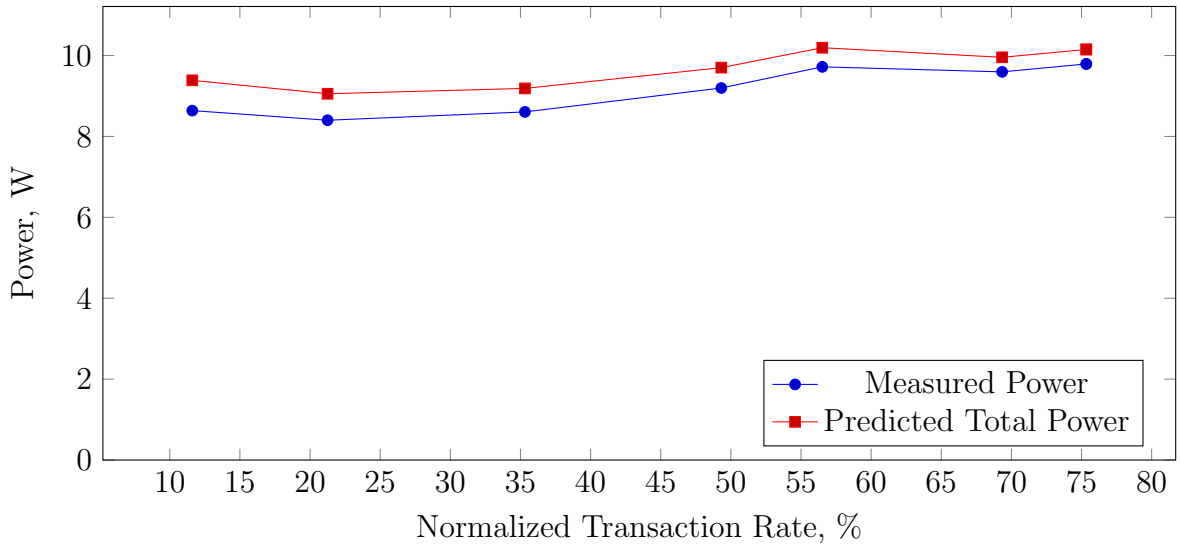

Figure 3.4: Actual and predicted average DRAM power in TPC-C vs. normalized transaction rate with, large database (SF=600)

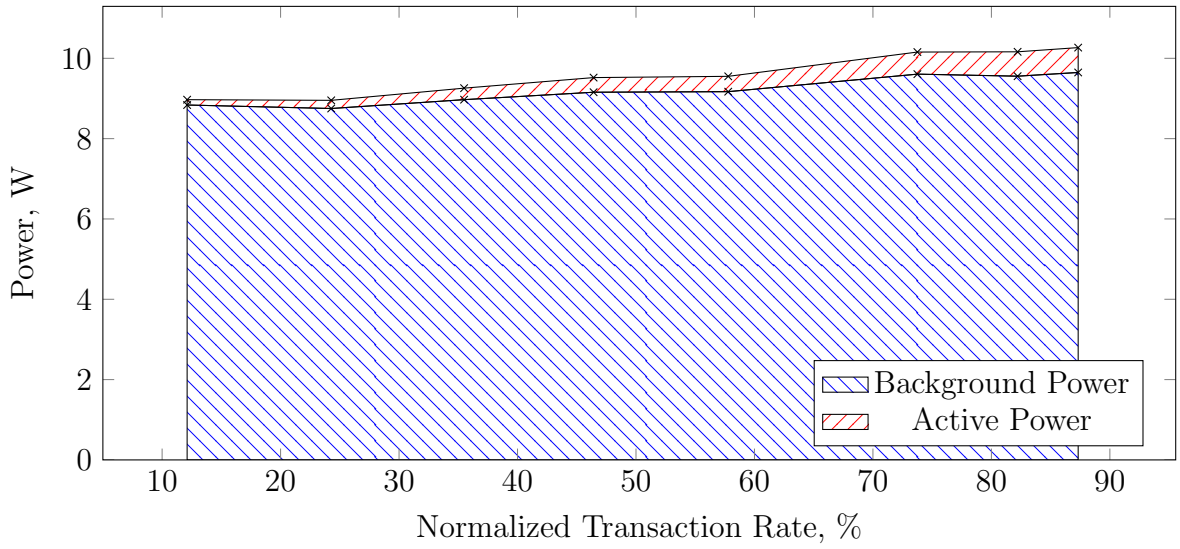

Figure 3.5: Active and background DRAM power breakdown in TPC-C vs. normalized transaction rate, medium database (SF=300)

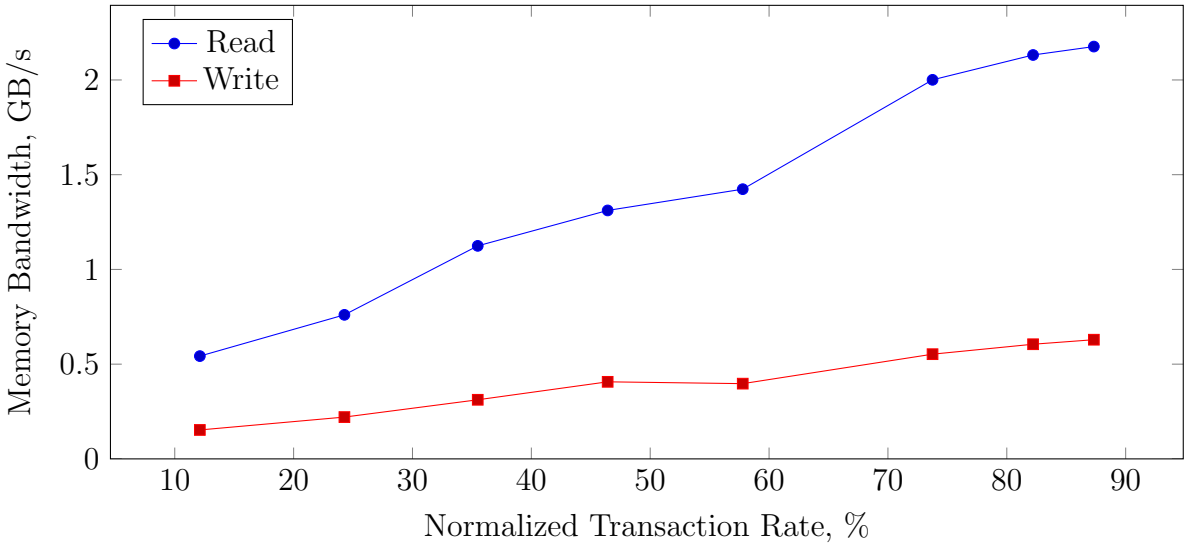

Figure 3.6: Memory read and write bandwidth vs. normalized transaction rate in TPC-C, medium database (SF=300)

Background power increases with load because DIMMs spend more time in the Standby power state as the load increases. Figure 3.7 shows the residency in the Standby and Self Refresh states as functions of load. At the lowest load, DIMMs on average spend only 27% of the time in the Standby state. This rises to 40% at the highest load. Unfortunately, Figure 3.7 also shows that the DIMMs almost never sink all the way into Self Refresh state, even when the load is low. Although we have shown power state residencies for only the small database size, this observation is true for all database sizes we tested. Thus, even a moderate workload does not exhibit enough memory idle time to save power in the Self Refresh state. This represents a lost opportunity, as power consumption in Self Refresh is significantly lower than in other power states.

3.3 TPC-H Results

For TPC-H, we varied the database scale factor from 6 to 72, which resulted in the database sizes ranging from 7.2 GB to 86 GB. We control the system load by controlling the number of concurrent query sessions, while restricting each session to use a single core in the server. Each session executes a batch of all 22 queries of the TPC-H benchmark in a random order, to reduce the possibility of inter-query optimization. By varying the number of concurrent

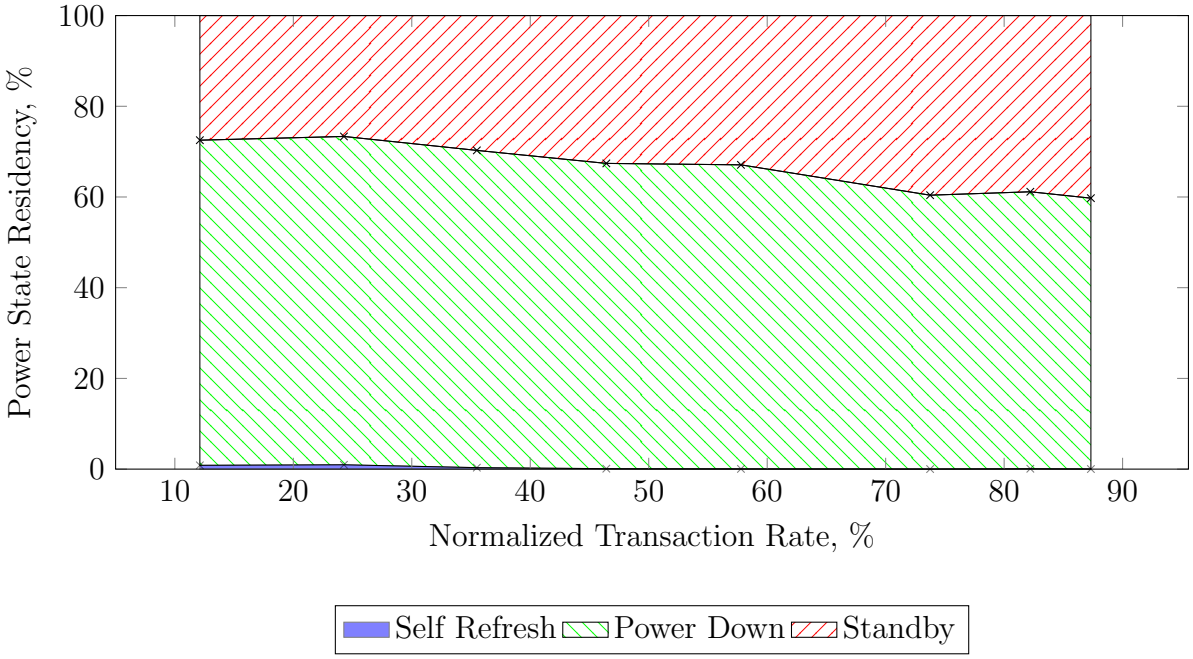

Figure 3.7: Power state residency vs. normalized transaction rate in TPC-C, medium database (SF = 300)

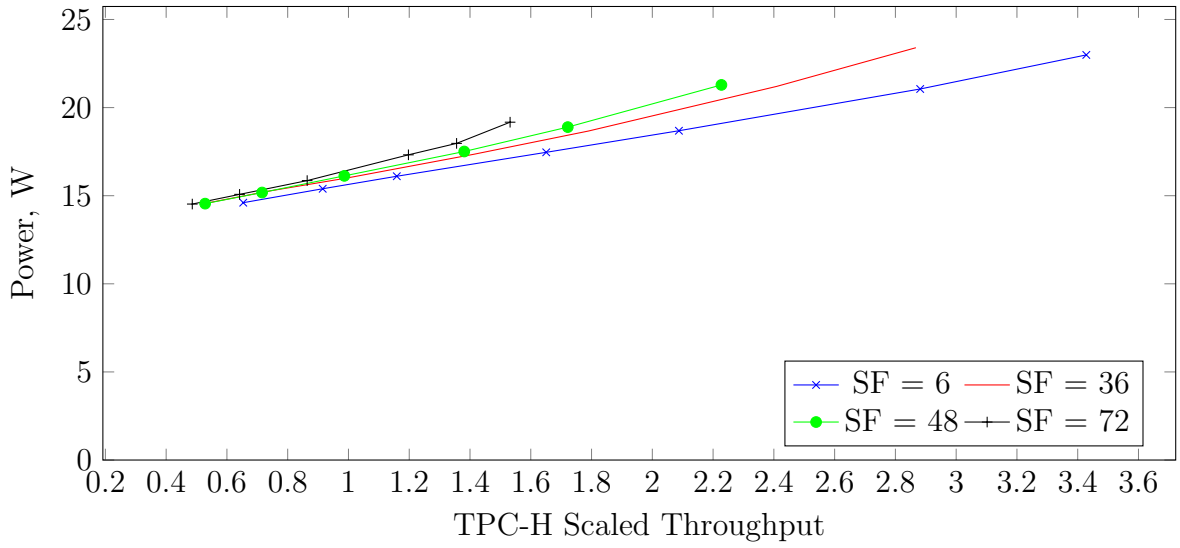

Figure 3.8: Memory power vs. scaled throughput in TPC-H

sessions from 1 to 16, we generate load from 1/16 to 16/16 of full CPU utilization in our system. We repeated each run 5 times, restarting the database and clearing the filesystem cache between runs, and take an average for each metric in the last four runs. To measure throughput, we first compute the reciprocal of the geometric mean of query batch run time of all client sessions, giving a measure of the query completion rate per session. We then multiply this by the number of sessions and the database scale factor, since TPC-H queries take longer for larger databases. This metric, which we refer to as *TPC-H scaled throughput*, approximates the amount of work done by the database system per unit of time.

For OLAP workloads, we ran the TPC-H benchmark on MonetDB [17], which is a column store. MonetDB relies on OS file mapping mechanism to access persistent data. Therefore, it operates best when the mapped files are cached by the OS and degrades when the dataset size exceeds the amount of available memory.

3.3.1 Memory Power Under TPC-H

Figure 3.8 shows measured average memory power consumption as a function of load (TPC-H scaled throughput), for a range of database sizes. Recall that, for a given database size, we vary throughput by varying the number of concurrent query sessions. As was the case

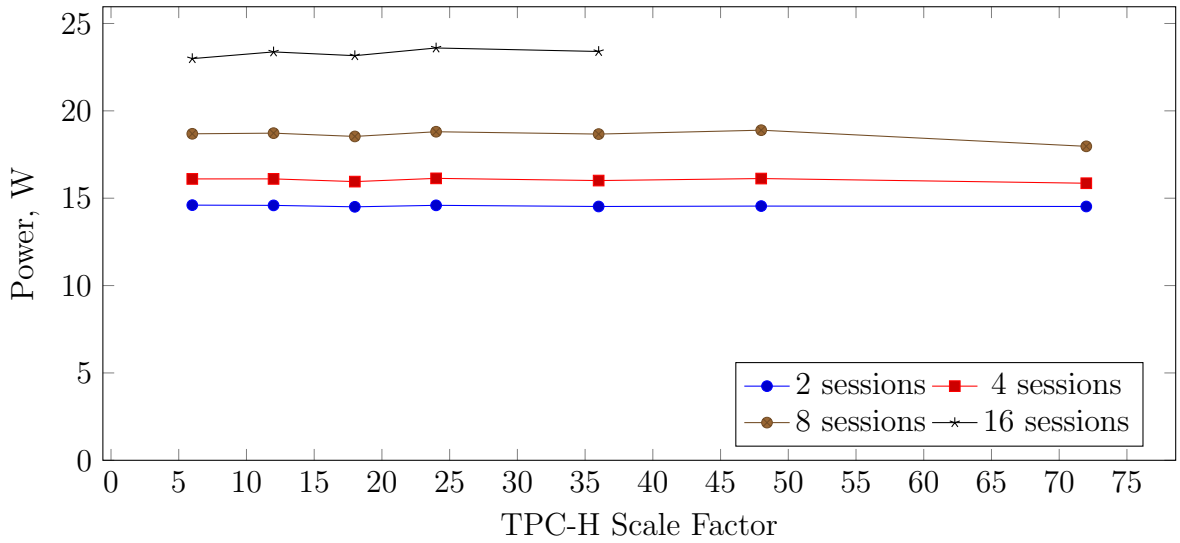

Figure 3.9: Memory power vs. database size in TPC-H

for TPC-C, memory power increases linearly with load, but starts at a relatively high value. Hence, power proportionality is poor. TPC-H is also more memory power hungry than TPC-C. Memory power consumption under our most intensive TPC-H workloads is almost twice as high as that under our peak TPC-C workload.

Figure 3.9 shows the same data as Figure 3.8, but plotted against database size, rather than load. In general, memory power consumption is largely independent of the database size, for the same reasons that it is independent for TPC-C.

We used the power model to further explain the power measurements. First, we compare the actual power measurements to the values of memory power produced by the model in Figures 3.10, 3.11, 3.12, for small (SF 6), medium (SF 48), and large (SF 96) database sizes, respectively. We found that the model's power estimates were very accurate for low workload intensities. As load increased, the model tended to underestimate the actual power consumption. However, the underestimation was never more than 15%.

Figure 3.13 shows the model's estimated breakdown of memory power into background and active components, for the medium database. This breakdown is similar for all other database sizes except for the largest one, for which active power falls off at high workload intensities when the system is overloaded. The active power component is much larger than it was for TPC-C, because the TPC-H benchmark is much more memory intensive. Nonetheless, most of the power consumption is still due to the background component, as was the case for TPC-C. Background power consumption does not grow with load, as it

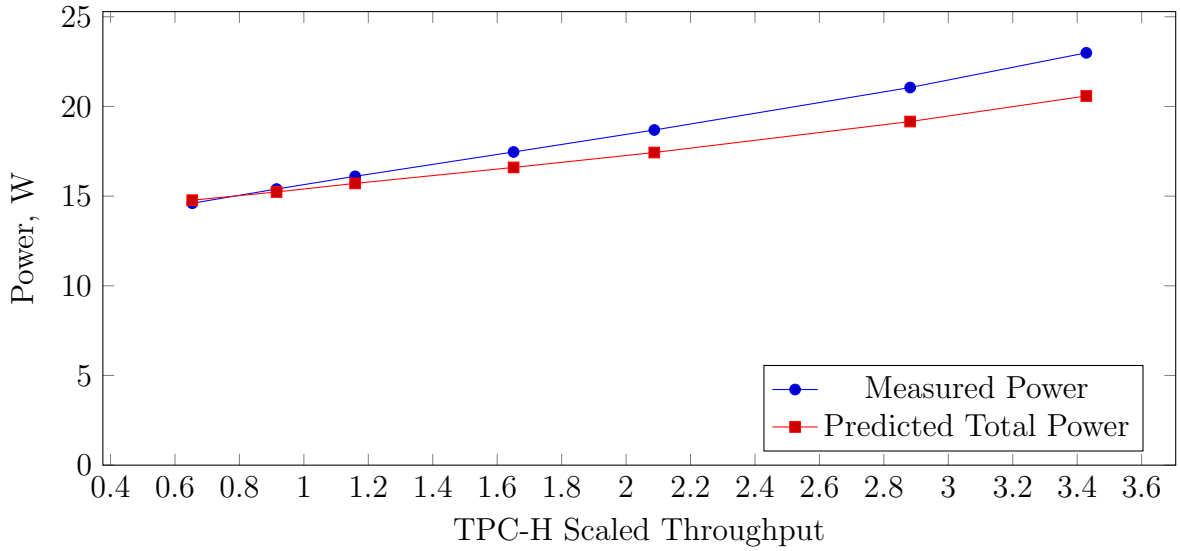

Figure 3.10: Actual and predicted average DRAM power in TPC-H vs. scaled throughput, small database (SF=6)

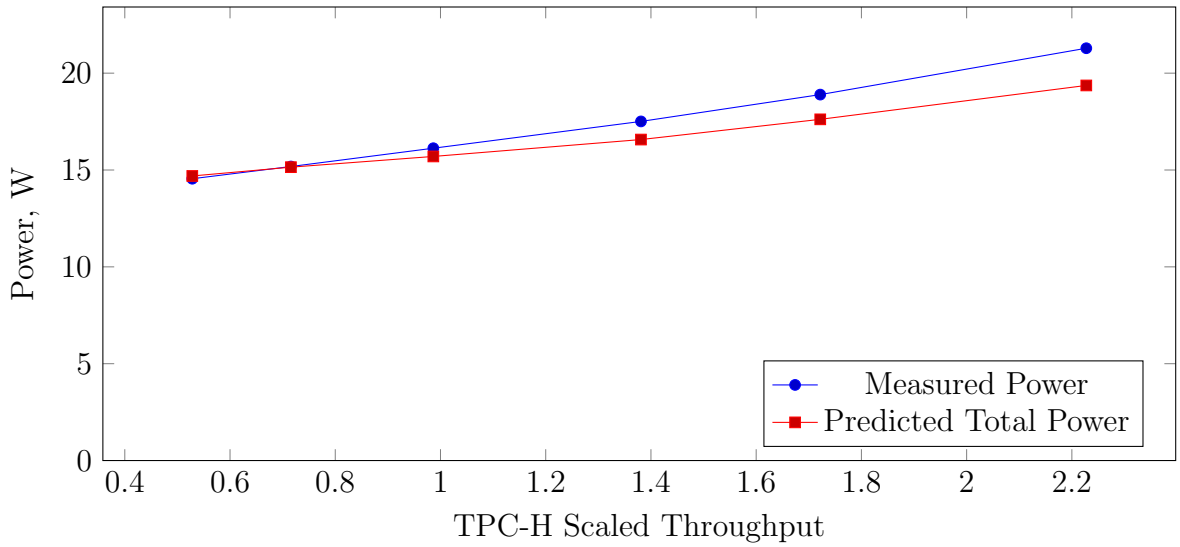

Figure 3.11: Actual and predicted average DRAM power in TPC-H vs. scaled throughput, medium database (SF=48)

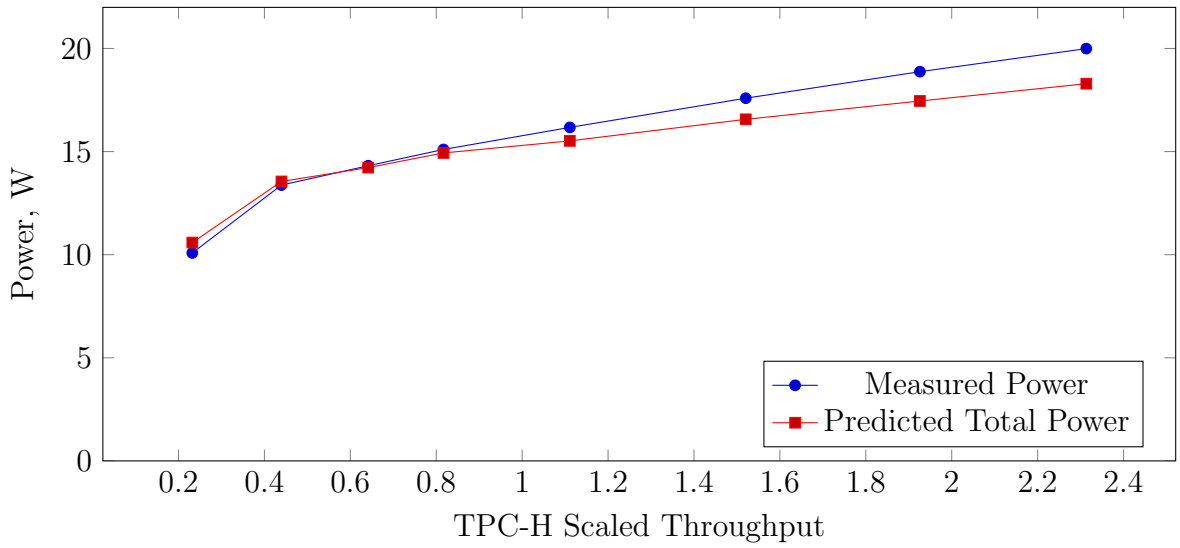

Figure 3.12: Actual and predicted average DRAM power in TPC-H vs. scaled throughput, large database (SF=96)

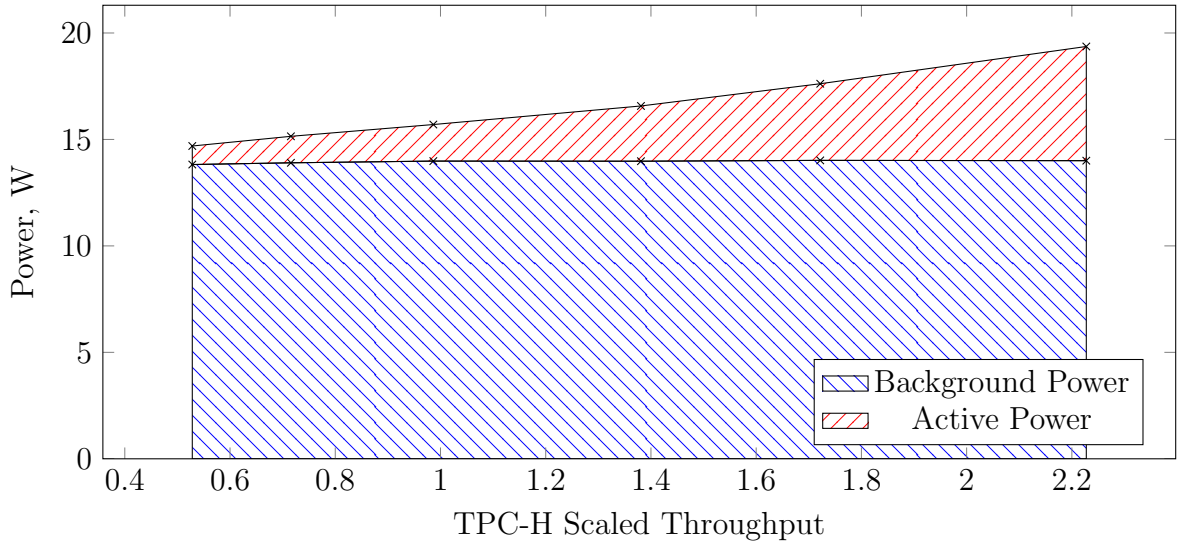

Figure 3.13: Active and background DRAM power breakdown in TPC-H vs. scaled throughput, medium database (SF=48)

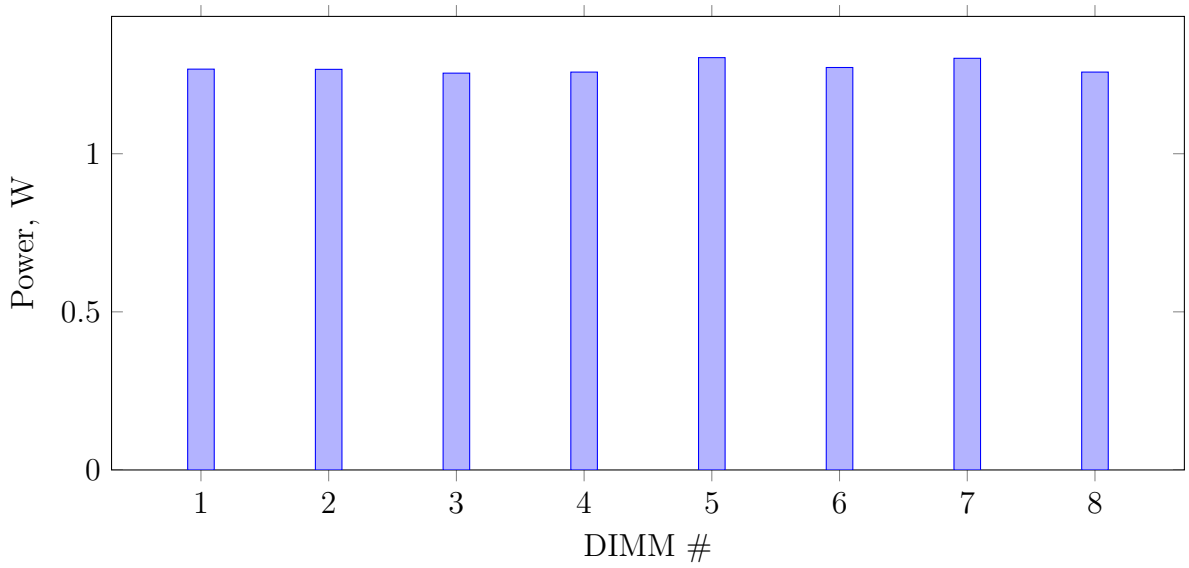

Figure 3.14: Power consumed by individual DIMMs in TPC-C runs with memory interleaving, SF=100, 100% load

does for TPC-C. The reason for this is is that DIMMs are almost always in the Standby power state under TPC-H, even with a single query session. Standby power state residency was at least 98% in all runs.

3.4 Non-Interleaved Memory

Memory interleaving tends to distribute memory accesses uniformly over each processor's DIMMs, increasing power consumption in each DIMM. For example, Figure 3.14 shows the power consumed by each DIMM for a TPC-C run with scale factor 100 and 100% load, with memory interleaving. DIMMs 1 to 4 are connected to CPU socket 1 and DIMMs 5 to 8 are connected to CPU socket 2. All DIMMs consume similar power, despite the fact that the database occupies only about 12.5 GB of the 96 GB available on the server.

By distributing memory accesses across DIMMs, interleaving greatly reduces the lengths of intervals between accesses *in each DIMM*. This makes memory power optimization difficult. Memory power consumption is dominated by background power, which cannot be reduced without introducing sufficiently long idle intervals allowing DIMMs to sink into lower power states. Additionally, even distribution of memory accesses between DIMMs

makes it difficult to add idle intervals to *some* DIMMs without adding a substantial performance penalty for all memory accesses. Thus, memory interleaving is likely to impede any attempt to optimize DIMM power consumption.

In this section, we consider the implications of disabling memory interleaving, since doing so appears to be a prerequisite for memory power optimization. Memory interleaving is a performance optimization, as it allows memory access to be parallelized across DIMMs. Therefore, they key question we wish to answer is how much of a performance impact memory interleaving has on our TPC-C and TPC-H workloads. If it is large, then memory power optimization may require a substantial performance trade-off. If not, then memory power optimization may be possible "for free".

To answer this question, we repeated the TPC-C and TPC-H experiments described in Sections 3.2 and 3.3, but with memory interleaving disabled. Interleaving was disabled by changing interleaving settings in the system BIOS. In the non-interleaved experiments, we did not attempt to control how the database systems made use of virtual memory. Neither did we attempt to control kernel's use of physical memory or the mapping of physical memory to DIMMs.

3.4.1 Performance Impact of Memory Interleaving

Figure 3.15 shows peak TPC-C transaction throughput (no client think times) with and without interleaving, for databases of different sizes. For TPC-C, memory interleaving has no significant effect on performance, regardless of the database size. This is because the TPC-C workload is not very memory intensive, as was shown in Section 3.2.

The TPC-H workload is more memory intensive and disabling memory interleaving does have some negative performance impact. Figure 3.16 shows the total running time of the batch of all 22 TPC-H queries, for small (scale factor 6) and large (scale factor 48) databases, with and without memory interleaving. As the table shows, the performance hit was about 10% on average over all of the TPC-H queries. However, we also found that some TPC-H queries were more sensitive than others to interleaving. Figure 3.17 shows the run time of each individual TPC-H query, with and without interleaving, for the large database. Six queries suffered slowdowns of 20% to 40%, and one query (Query 6) showed about 75% performance degradation. These results suggest that for more memory-intensive database workloads, like TPC-H, memory power optimization is likely to require a performance tradeoff.

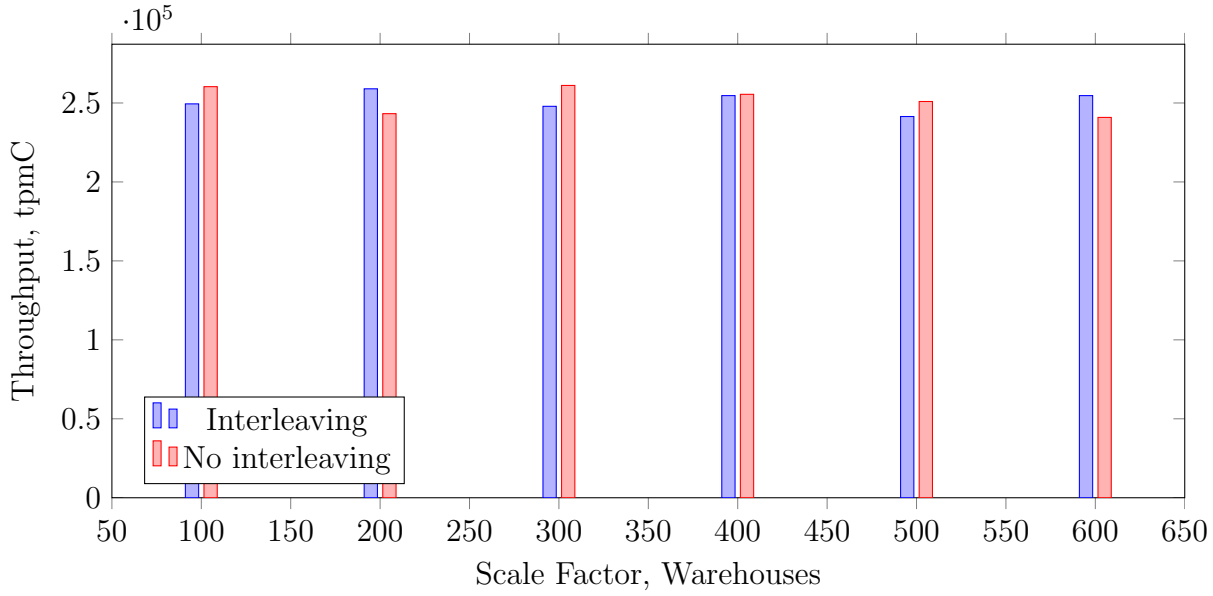

Figure 3.15: Maximum TPC-C throughput in runs with and without memory interleaving

Scale factor	With interleaving	Without interleaving	Relative slowdown
6	22.4	24.6	10%
48	232.2	254.7	9.7%

Figure 3.16: TPC-H total run time, seconds, with and without memory interleaving

3.4.2 Power Impact of Memory Interleaving

Since disabling interleaving does not hurt TPC-C performance, it can be disabled to enable memory power optimization techniques. Our experiments also found that disabling memory interleaving does not, by itself, result in significant memory power savings. It does lead to uneven use of the DIMMs, because of the way data happen to map to DIMMs in our test server. For example, Figure 3.18 shows the residency in any of the low-power states (Power Down and Self Refresh) in each individual DIMM for TPC-C runs with and without interleaving. The variance across the DIMMs is significantly higher in the non-interleaved case. However, these differences do not translate into substantial differences in power consumption. At maximum load, for the small TPC-C database, total memory power consumption was about 10 watts with interleaving, and 9.6 watts without interleaving, a difference of less than 5%. Thus, disabling interleaving should be viewed as a prerequisite for the use

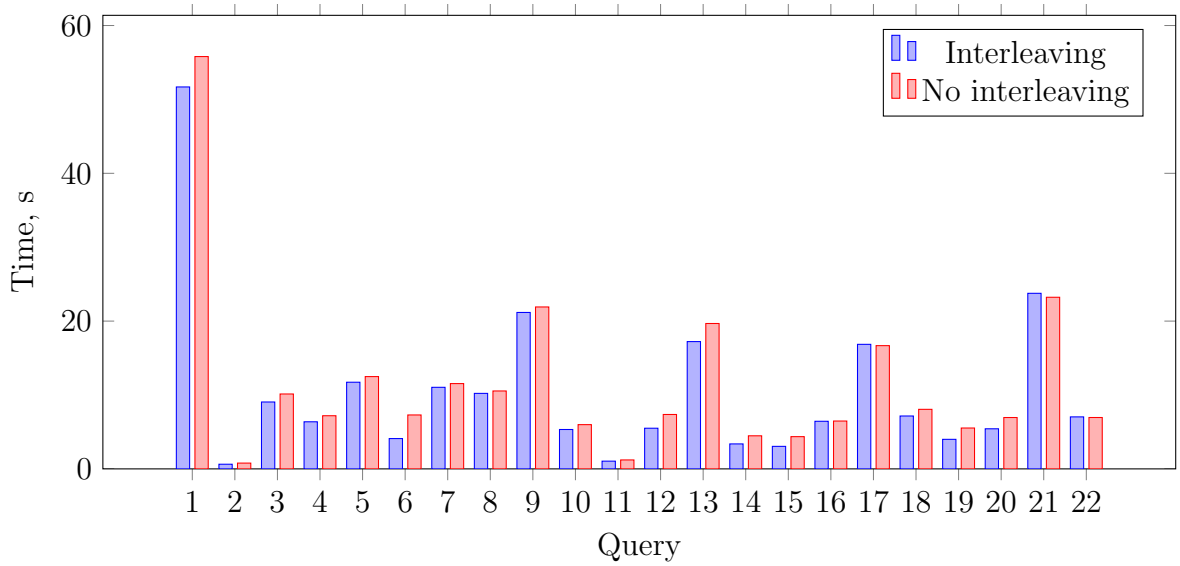

Figure 3.17: Average query run times in TPC-H with and without memory interleaving, SF=48

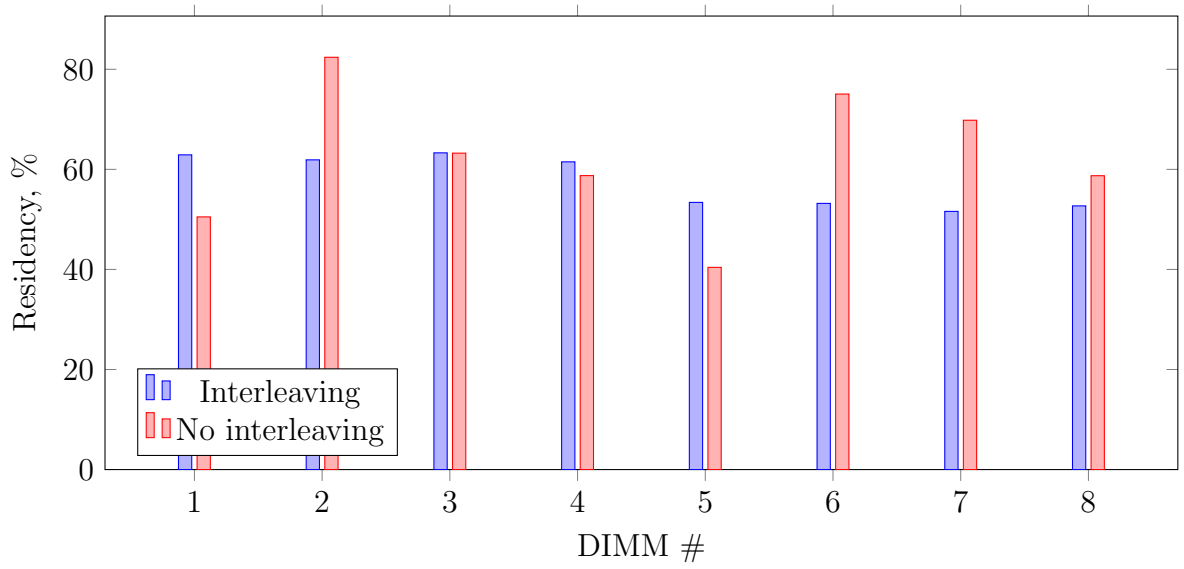

Figure 3.18: Total low-power state residency in individual DIMMs in TPC-C runs with and without memory interleaving, SF=100, 100% load

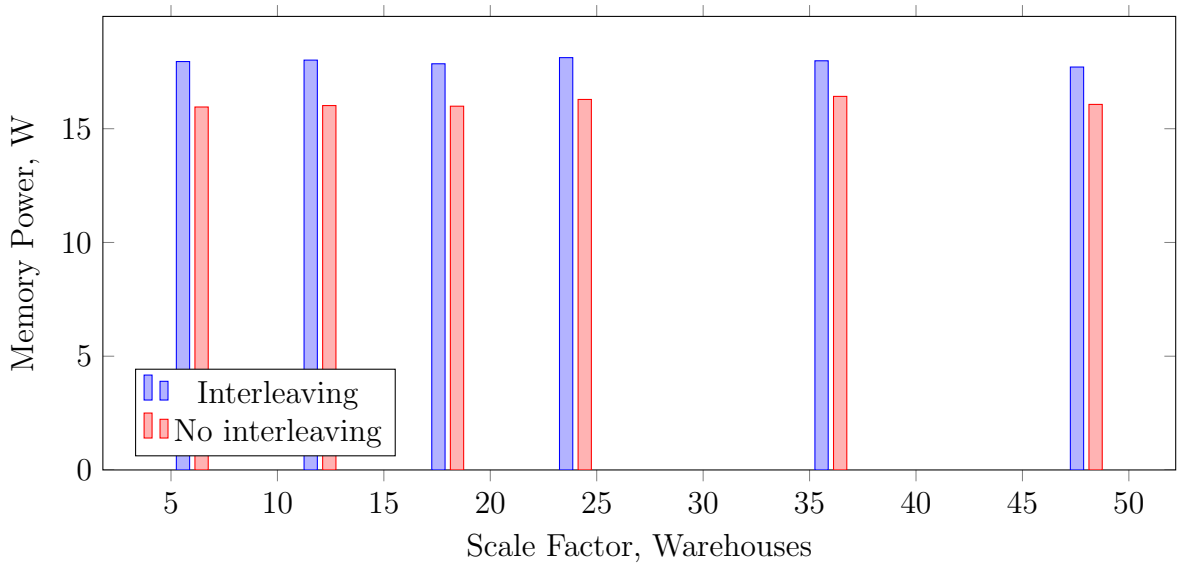

Figure 3.19: Average TPC-H memory power consumption in runs with and without memory interleaving

of application of memory power optimizations, and not as a power-saving technique in its own right.

In TPC-H, disabling memory interleaving reduces memory power consumption by approximately 10%. Average memory power in the entire batch (22 queries), with and without interleaving, is shown in Figure 3.19. Since the queries run longer without interleaving, the total energy impact should also be considered. Overall, there is a small reduction in the total amount of energy consumed for each batch execution. The relative reduction in energy use, for databases of different sizes, is shown in Figure 3.20. As with TPC-C, disabling interleaving is not, by itself, an effective method to reduce memory power consumption in the TPC-H workload.

3.5 Summary of Empirical Results

In this chapter, we studied memory power consumption in transactional and analytical database workloads. In both workloads, memory power consumption does not depend on the database size and stays high when the database is much smaller than the amount of available memory. Memory power consumption is only slightly sensitive to the load

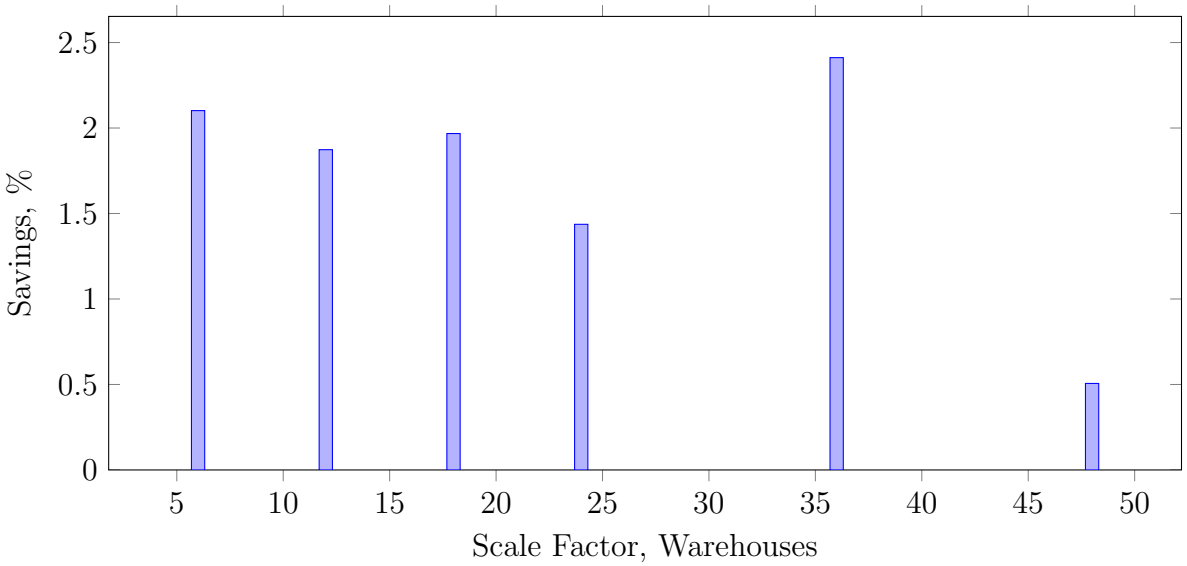

Figure 3.20: Total energy savings in TPC-H when interleaving is disabled

intensity in a transactional workload, showing about 20% difference in power over a range of load spanning an order of magnitude. In the analytical workload, the relative difference between the highest and lowest load is higher, about 60%.

The non-proportional relationship between memory power consumption and changes in database size and load is due to the background power being the bulk of the total consumption. Background power is not proportional to the load but depends on the DIMM state residency, and stays high largely regardless of load and database size. Therefore, any power optimization technique must reduce background power to be effective.

Memory interleaving, often enabled by default, is an impediment in increasing low-power state utilization to target background power consumption. Memory interleaving spreads traffic between DIMMs, which eliminates naturally occurring skew in the access frequency between DIMMs, and makes memory power optimization difficult. Memory interleaving is considered to be a performance optimization, however, we found that its effect on performance is negligible in the transactional workload and small (around 10%) in the analytical workload. However, disabling interleaving does not automatically create memory power savings. Explicit management of memory allocation and access are needed, which will be the focus of the next chapters.

3.6 Related Work

This section describes prior work on how memory power consumption is affected by factors such as memory frequency (Subsection 3.6.1), use of low power states (Subsection 3.6.2), and load (Subsection 3.6.3). The effects of memory size on power consumption in disk-based databases are discussed in Subsection 3.6.3.

3.6.1 Effects of Memory Frequency

Computer memory can work at different frequencies of the DDR interface, which can be changed at a reboot. Although the DDR3 and DDR4 standards define the procedure to dynamically change the frequency at run time, such capability is rarely available in existing implementations, at the time of writing. Nevertheless, since memory frequency is another factor affecting its power consumption, there are studies to explore its effects.

Kumar et al [51] found that a small reduction of memory frequency (from 1066 to 866 MHz) causes a proportional reduction in memory power consumption under load. However, the effect on performance was not proportional to the frequency change and varied between types of queries. Most queries in TPC-H were not affected by this change, so their performance/power ratio improved. However, TPC-H Q1 showed a proportional performance drop and TPC-H Q8 suffered by 20%, which is twice as much as the frequency change, although it is not clear why the performance drop exceeded the frequency reduction. This observation confirms that database queries have varying sensitivity to the memory bandwidth and, even in OLAP workloads, memory bandwidth is not typically critical for performance.

The effects of frequency scaling on *in-memory* database workloads are also discussed in the study by Appuswamy et al [12]. The wider workload set included two synthetic microbenchmarks (in-memory aggregation and scan), an OLAP workload (TPC-H) and an OLTP one (TPC-C). The memory frequency reduction step was more significant than in [51] - 50%, from 1600 to 800 MHz. The power reduction due to this frequency change was less than proportional, about 20%. Confirming previous findings, the performance drop varied between workloads, from being proportional to the frequency change in workloads that saturate memory bus (TPC-H, synthetic aggregation), to less significant in ones with low channel utilization (TPC-C, synthetic scan).

The non-proportional response of memory power consumption to frequency change could lead to a mechanism of saving power in a DBMS that would adjust the frequency based on performance requirements and sensitivity of a particular queries to frequency. For

queries that are not sensitive to frequency, it should always be reduced. For queries that are affected proportionally, frequency can be reduced only if it is acceptable to prioritize energy efficiency over response time. And the rare queries that exhibit poor power efficiency at lower frequencies should run at the maximum frequency all the time. Changing frequency at run time can be potentially easy to implement, if the hardware provided this capability, as no data redistribution is required. However, a method to measure the degree of sensitivity of a particular workload to memory frequency would be required. The shortcoming of such a mechanism is the expected magnitude of power saving is low, up to 15% in the best case [12].

3.6.2 Effects of Power States

DRAM implements low power states to reduce power consumption during periods of inactivity. Appuswamy et al [12] studied the impact of enabling low power states under synthetic and database macrobenchmarks. They collected performance and memory power consumption data in the configurations with low power states disabled ("CKE disable" setting in BIOS) and enabled ("CKE enable"). In all cases, the performance impact was small (less than 5%) while power consumption decreased significantly in all cases except multi-threaded scan and aggregation microbenchmarks. The TPC-C and TPC-H allowed for a factor 1.34 and 1.42 reduction in the power/performance ratio. These workloads are not memory intensive, so memory was able to spend about 90% of time in a low power state. When running on a single thread, the microbenchmarks generated memory traffic low enough that low power residency was between 72% and 84%, saving about 20% of memory power. However, low power state residency dropped to 5% on eight threads and power consumption similar to the baseline with power state disabled.

In conclusion, enabling low power states improves memory power efficiency in most workloads, with the exception of ones with very high memory utilization. Since the hardware is conservative in switching to a low power state, the performance effect was minimal or negligible. The authors only reported the combined residency in the "CKE off" state, which includes any low-power state. Therefore, it is not possible to tell if the deeper Self refresh state is used in various workloads. According to my results in this chapter, the system tends to be very conservative when using the Self Refresh state, as opposed to the Power Down state. Due to this, and also because memory accesses tend to be evenly distributed across DIMMs, the system Self Refresh state is not used much, which limits the power saving potential of a hardware-based power state policy.

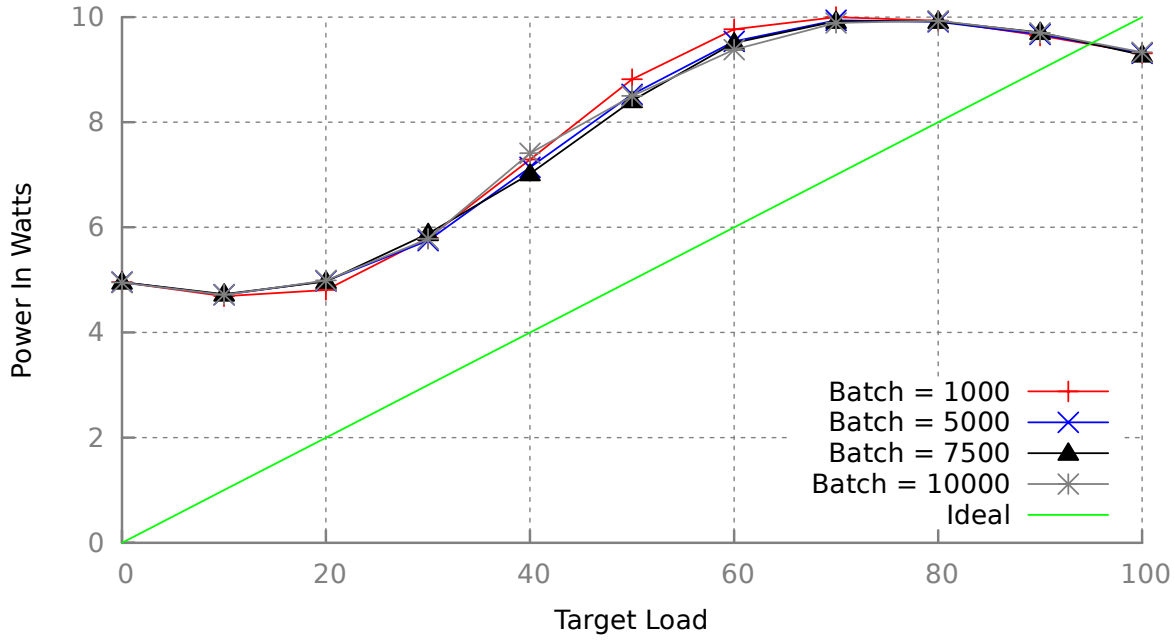

Figure 3.21: Memory power consumption by load, in SPECpower-ssj2008, from Subramaniam et al [70]

3.6.3 Effects of Load

Power consumption versus load was studied by Subramaniam et al [70] in the context of SPEC Power benchmark. SPEC Power allows for gradual adjustment of load level relative to the separately measured maximum sustainable load. Per-component power estimations (CPU core, CPU uncore, CPU package, DRAM) were obtained from RAPL. Memory power consumption showed a characteristic S-shape, shown in Figure 3.21, with a dynamic range of about 2. Memory power, similar to the uncore part of the processors package, was the least power proportional subsystem. The bulk of transition between the minimum and maximum power consumption happens between approximately 20% and 60% load. The authors did not try to explain the shape of the load characteristic. However, based on my characterization of memory power consumption, this unique shape may be related to non-linearity of low power state residency versus load, which results from the timeout-based power state management policy.

There are several studies on database power efficiency that either look at total power consumption only [58], or find that memory consumption is constant vs. other factors [73, 78, 39]. We know that memory consumption does change and these results can be explained by the lack of focus on memory. For example, memory power management can be disabled in the system configuration. Estimating or measuring memory power has also required substantial effort that may not be seen as justifiable in research that is not focused on memory.

Effects of Memory Size in Disk-based Databases

The main purpose of memory in a disk-based DBMS is to serve as a cache (buffer pool) of database pages. The system can work with varying amount of available memory, however, its performance and power consumption will be affected. When such systems employ a large number of disks to increase I/O bandwidth, disks become responsible for the bulk of consumed power. This does not represent a particularly valuable use-case for memory power optimization.

Meza et al [60] studied the power/performance trade-off by varying the number of disks and memory capacity. Peak throughput in a single workload (TPC-H, 300 GB) was used as a measure of system performance and power was measured on the AC side, separately for the main server and storage subsystem. As in many early studies, memory power consumption was found to be constant 4 W per 4 GB DIMM. In total, this translates to approximately 12% of the total system power excluding storage, but only 7% when storage is accounted for. At the same time, memory was critical in improving total power efficiency, for two reasons. First, in a system with memory used as a cache, its memory is always full since it holds only a fraction of the total database. Second, a disk-based storage subsystem has to be configured based on disk throughput and not capacity, which leads to vast disk subsystem overprovisioning (about 50x in this particular study) and its exceptionally high power consumption. Additionally, peak throughput only characterizes systems that are completely busy and partial utilization was not considered. In a system configured in this way, memory cache is highly efficient in reducing the I/O demand and, as a result, system power consumption. Considering the low relative impact of memory on total consumption and the fact that memory is fully used and the system is fully loaded, memory power optimization in these circumstances seems neither worthwhile nor feasible.

Kumar et al [51] studied memory power in a similar context, for a disk-based DBMS and a DSS workload. However, they also looked at how memory power consumption and system performance are affected by the type of workload. The workload was the TPC-H benchmark, with results broken down by individual queries. Memory capacity was one of

the two control variables, with the other being memory frequency. Memory power showed little power proportionality with load variations and its consumption was proportional only to the number of DIMMs used. The effect of memory capacity was non-uniform for different queries and this variance was significant. Thus, when the amount of memory was halved, performance of some queries was affected by 5% or less while two queries suffered a degradation between 422% and 1241%. This is expected as in a disk-based system memory is used mostly for caching and its value depends on the query access locality. The non-uniform effect of memory capacity on performance can also be used in a power-aware system to adjust the amount of accessible memory. However, this method was considered to be potentially problematic because it would require the application to know the physical layout of memory and place the cached data according to this layout. I use this approach is the foundation of DimmStore, a power-efficient DBMS prototype (described in Chapter4). This study did not provide a detailed analysis why the impact of memory capacity was not uniform between queries. Although it can be explained by changing the cache miss ratio and, consequently, the amount of disk I/O, this is not a complete explanation. For example, it was noted in the study that query plans of some queries changes on a configuration with a different amount of memory. For example, a join algorithm switched from a hash to nested loops implementation. By considering such effects of memory availability on query plans, the trade-off between memory power efficiency and performance can be explored to a greater detail. The results of this study are complementary to my work because a different type of system (disk-based) was in the focus. However, I believe that DSS workloads are also relevant for in-memory databases and should be studied further.

Niemann at al [61] conducted research on power efficiency of several database workloads. One contribution of this work is breaking down power consumption into individual components, CPU, memory, disks, motherboard, and power supply. In this study, the ratio of memory and CPU power consumption was between 0.25 and 0.3 over all the tests. However, CPU and memory consumption was small part of the total as more power went to the mainboard and power supply. Another point of the paper was evaluating server power efficiency versus system and workload configuration. Unfortunately, these results were overly specific to PostgreSQL and mostly affected by its architectural features such as type of session (single-user vs multi-user) and its dependence on OS cache. However, the common finding was that more performant configurations were the most energy efficient.

Chapter 4

DimmStore: Rank-Aware Allocation and Rate-Based Placement

In this chapter, we present DimmStore, a prototype in-memory transactional database management system that aims at reducing memory power consumption. DimmStore targets background power by using memory low-power states as the power-saving mechanism. Memory power states have a high power reduction potential as a DIMM in the lowest-power state (Self Refresh) consumes about 80% less power than it would in the high-power state, ignoring the active power component (Section 2.2). Low-power states need long periods of idleness to get activated, therefore, they see little use in existing systems and workloads (Chapter 2).

DimmStore is designed to create periods of idleness *in some of the DIMMs* by deliberately skewing memory load across the DIMMs. DimmStore achieves that by understanding the physical layout of the memory address space and using the technique of *rank-aware allocation*. With *rate-based placement*, the most-frequently used elements of the database are then placed in a subset (ideally, small) of the DIMMs, and the remaining DIMMs are used for the less frequently used data.

The design of DimmStore is described in Section 4.1. DimmStore is built on top of H-Store, an in-memory transactional DBMS. DimmStore uses the concepts of *eviction* and *uneviction* in H-Store's *anti-caching* feature to achieve and maintain the access skew between DIMMs.

In Section 4.2, we describe how DimmStore interacts with the operating system to discover physical memory layout and allocate memory in particular DIMMs. Operating

system do not currently provide support for these functionalities. Therefore, DimmStore uses ad-hoc workarounds so that we can test and evaluate it.

In Sections 4.4 and 4.5, 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 4.1 at reducing memory power consumption? Second, do these techniques have a significant impact on performance? We consider two transactional workloads. The first (Section 4.4) is the Yahoo! Cloud Serving Benchmark (YCSB) [21], which has simple and easily controllable data access patterns. The second (Section 4.5) is TPC-C [5], which exhibits more complex and dynamic patterns. The power savings for a medium database size are approximately 30% in both workloads. The YCSB workload, which is less memory intensive, showed no performance impact in DimmStore compared to the baseline. In a more memory-heavy TPC-C, the peak throughput degradation was about 10%.

4.1 DimmStore Design

DimmStore is an in-memory transactional database system, based on H-Store [44]. 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.

In traditional database systems, memory load is distributed more-or-less evenly across the DIMMs (see Section 4.2.1). 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*. DimmStore has two memory allocators, one for each region. Whenever DimmStore requires memory, it must choose which region to allocate the memory from. 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 4.2.2, we describe how this rank-aware memory partitioning is accomplished.

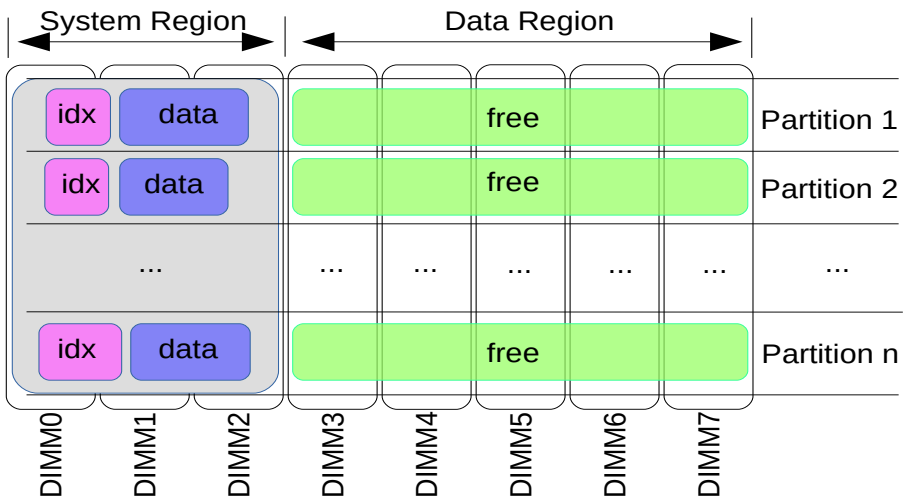

Figure 4.1: DimmStore with a small database

If possible, DimmStore uses *only* memory from the system region. This is illustrated by the DimmStore configuration shown in Figure 4.1. 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 Sections 4.4 and 4.5. 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 4.2. 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 4.2.2, we describe the operating system support that is required, and how we implemented it in our testbed.

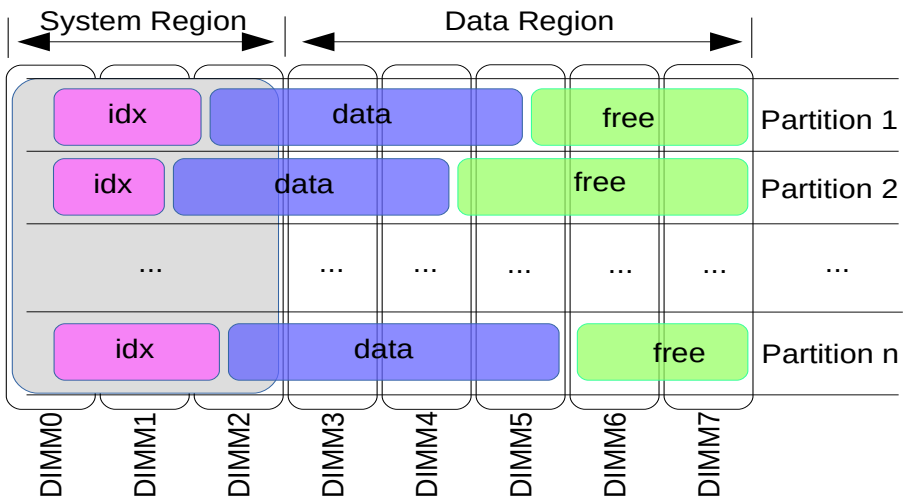

Figure 4.2: DimmStore after spilling to the Data region

4.1.1 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. 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.

Physical memory in the system region is managed in the usual rank-oblivous way by the operating system kernel. Virtual address spaces for all processes, including DimmStore, are allocated space from this region by the kernel.

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 4.1 and 4.2, 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 this extension is left for future work.

4.1.2 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 4.1.3. 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 4.2. 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 4.2.2.

4.1.3 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 [26] 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 global multi-table LRU because it is simpler, but it could easily be modified to use per-table LRU lists in each partition.

4.1.4 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.

4.1.5 Data Loading

We modified the data loading code in H-Store so that tuples can be loaded into the system and data regions directly at system startup. The user may specify a loading memory threshold, which may differ from the default memory threshold that triggers tuple eviction. DimmStore begins loading data into the system region. When the combined data and index size exceeds the loading threshold, the tuple loader starts to load batches of tuples into the data region. Of course, such tuples may not actually be cold, but DimmStore's anti-caching mechanism will gradually adjust tuple locations as the system runs. Specifying a loading memory threshold that is lower than the runtime one can be beneficial to reserve extra space for future index growth.

4.1.6 System region memory reclamation

In H-Store, after a tuple is deleted from a table, the freed space can only be used to insert tuples in the same table. H-Store implements a custom tuple allocator over large blocks obtained from the operating system. When a tuple is deleted, its space is added to a free list which is used for future insertions to the same table only. Deleted tuples' space cannot be used to allocate tuples for other tables.

In DimmStore, tables in the System region are implemented as ordinary in-memory tables. Evicting a "cold" tuple from a table in the System region causes this tuple to be deleted from the in-memory table. Therefore, if a large number of tuples is evicted in a table and the freed space is not used by future insertions or unevictions, the freed space becomes unusable. The problem is particularly significant because the initial database loading fills up the System region without knowing the tuple access frequency in the workload. As a result, a large amount of data is evicted from "cold" tables soon after the workload starts, creating permanently unused "holes" in the tables in the System region.

To solve this problem, we implemented a mechanism to reclaim free tuples' space. For each table, the percentage of the freed space is monitored and once it exceeds a configured threshold, tuples from several blocks in the table are moved into slots taken from the free list. The processed blocks become empty and are freed for possible re-allocation to other tables.

4.2 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 [37, 38, 42, 77], 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.

4.2.1 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 mapping (non-interleaved) 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.

4.2.2 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 4.3. As was noted in Section 4.1.1, 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.

4.3 Experimental Setup

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

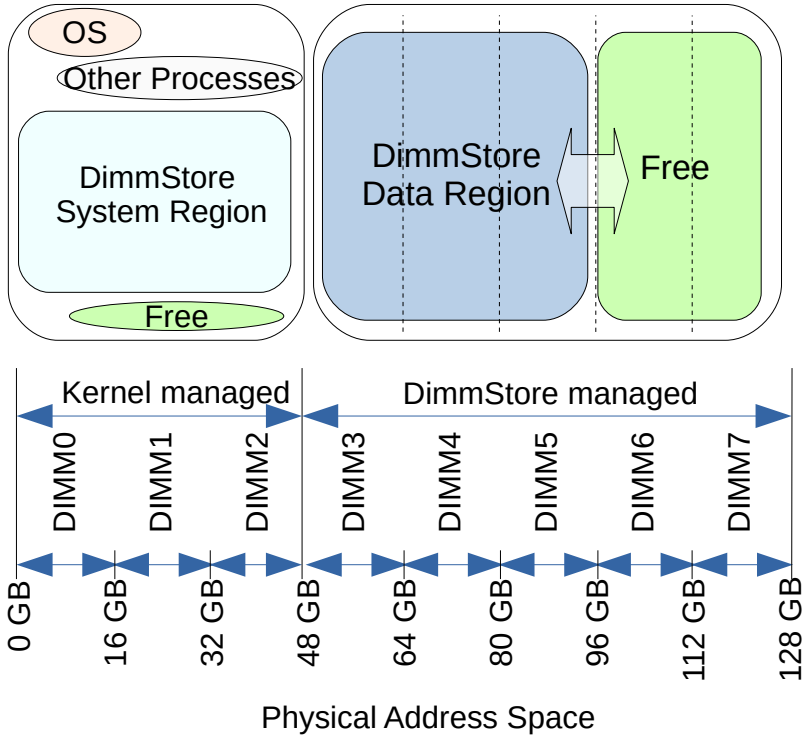

Figure 4.3: Physical memory management in DimmStore

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, as described in Section 2.4. Power consumption of each DIMM in the system is collected during the experiments.

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.

4.4 Experimental Evaluation (YCSB)

Our first set of experiments uses the YCSB workload [21], which has 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 4.4 summarizes the other YCSB workload and DimmStore parameter settings.

Each experimental run consists of three phases: database loading, warm up, and measurement. We ignore measurements collected during the loading and warm up phases. The warm up 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.

Parameter	Values	Default
Zipf skew parameter s	0.5 - 1.2	0.95
Database size	10 - 100 GB	60 GB
Offered load	22.5 - 180 Ktps	90 Ktps
System region size	32 GB	32 GB
Eviction interval t_{evict}	1 ms	1 ms
Eviction volume N_{evict}	64 KB	64 KB
Uneviction probability $p_{unevict}$	$\frac{1}{64}$	$\frac{1}{64}$

Figure 4.4: YCSB Experiment Parameters

4.4.1 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 4.5 shows total memory access rates (reads 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 4.6 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

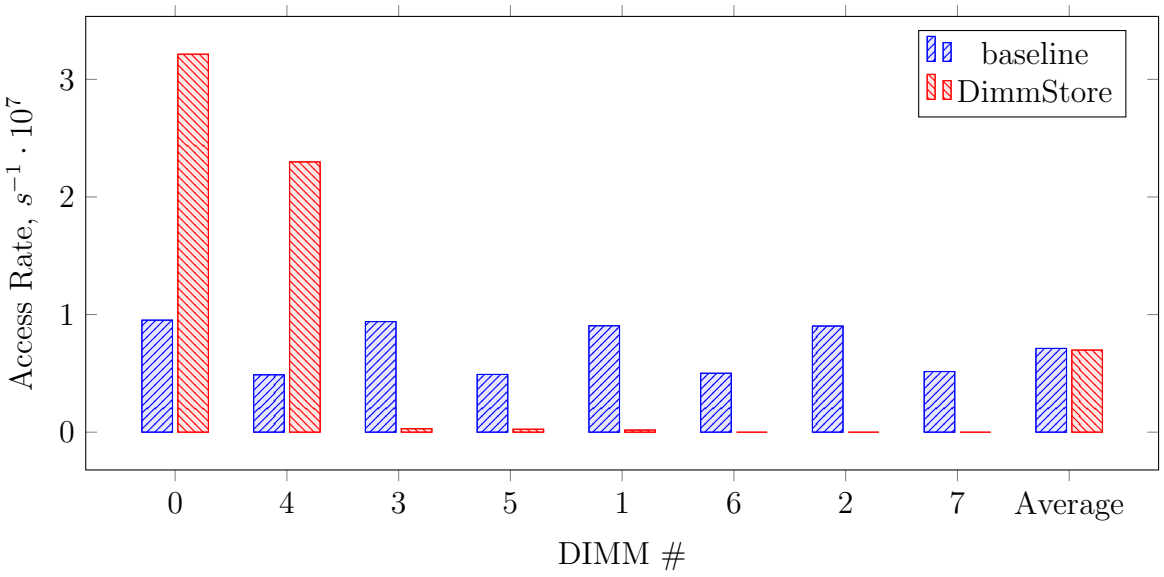

Figure 4.5: YCSB: Individual DIMM access rate, 60 GB database, 90 Ktps. For DimmStore, the system region consists of DIMMs 0 and 4, with the others making up the data region.

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 4.7 and 4.8 illustrate this effect. Figure 4.7 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 StandBy 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

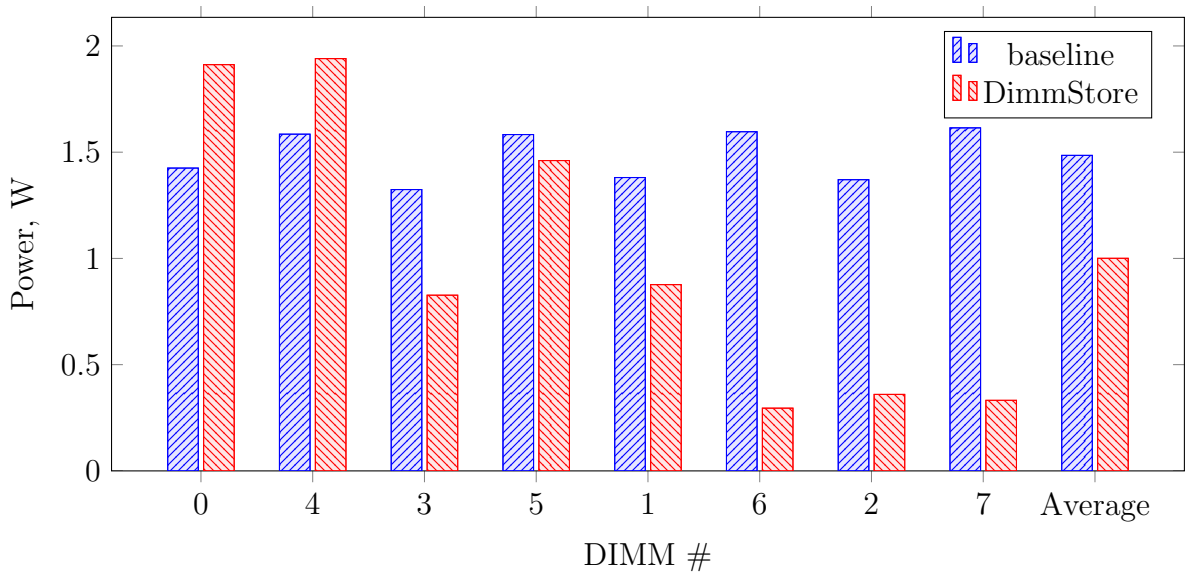

Figure 4.6: YCSB: Individual DIMM power consumption, 60 GB database, 90 Ktps load. For DimmStore, the system region consists of DIMMs 0 and 4, with the others making up the data region.

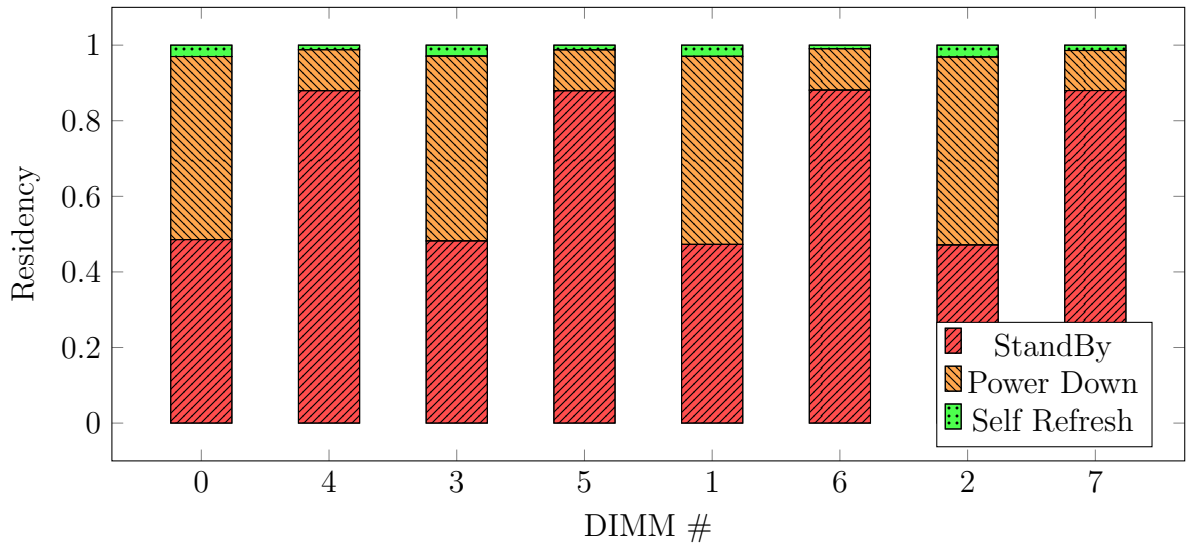

Figure 4.7: YCSB: Average power state residency in the baseline, 60 GB database, 90 Ktps load

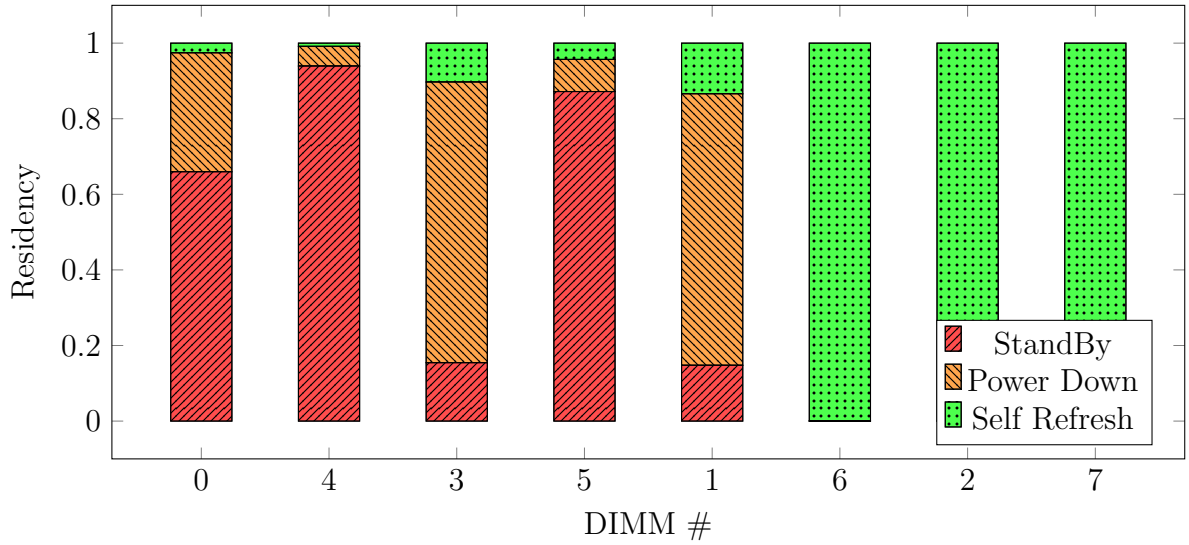

Figure 4.8: YCSB: Average power state residency in DimmStore, 60 GB database, 90 Ktps load

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 4.8 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 0.3 W 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 StandBy 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.

4.4.2 Effects of Database Size

Next, we show how memory power consumption is affected by the database size (Figure 4.9). 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

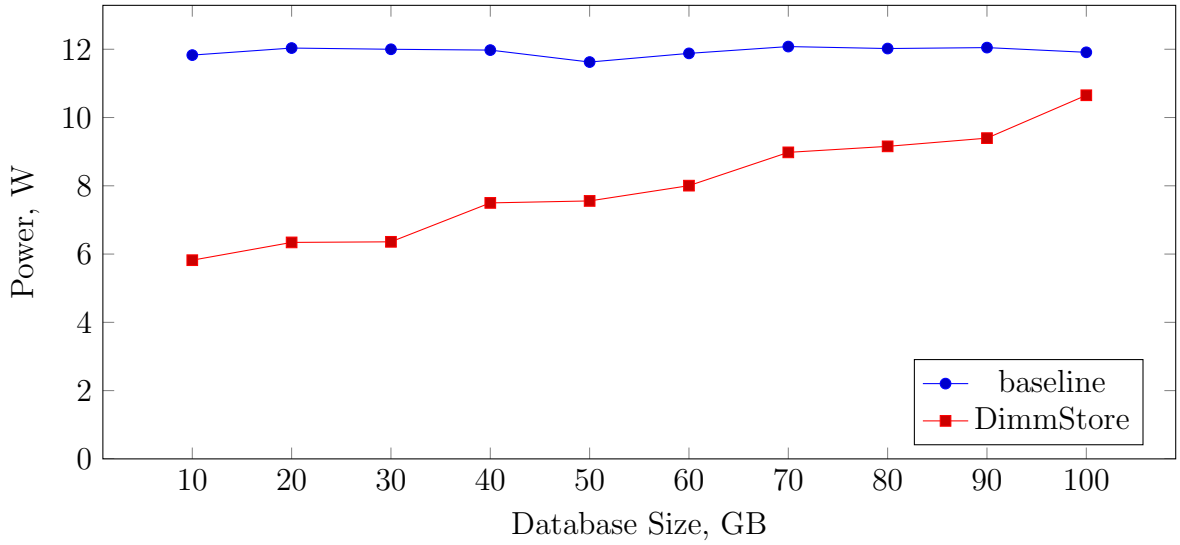

Figure 4.9: YCSB: Memory power consumption by database size, 90 Ktps load.

have low access rates and reduced background power consumption. The corresponding per-DIMM power measurements are shown in Figure 4.10. The total power consumption of the two hot DIMMs' increases by 0.7 W (0.35 W each), while consumption of the six DIMMs in the data region decreases by 2.86 W total (0.48 W each).

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. Each DIMM that is not used saves 0.8 W compared to the average consumption of a DIMM in baseline. When the database is the smallest, this amounts to about 5 W over 6 idle DIMMs, while the two hot DIMMs increase their consumption by only 1.5 W in total.

4.4.3 Effects of Load

To study the effects of load, we fixed the database size at 60GB and varied the transaction request rate. Figure 4.11 shows total memory power consumption as a function of the

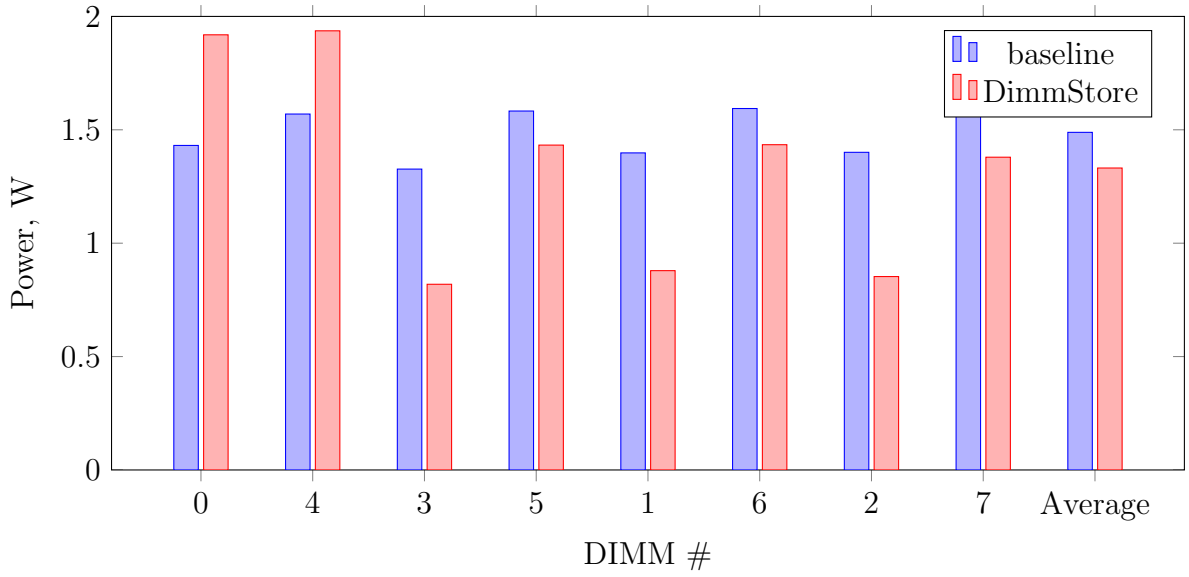

Figure 4.10: YCSB: Individual DIMM power consumption, s=0.95, 80 GB DB (baseline), 100 GB database (DimmStore), 50% load, read-only mix

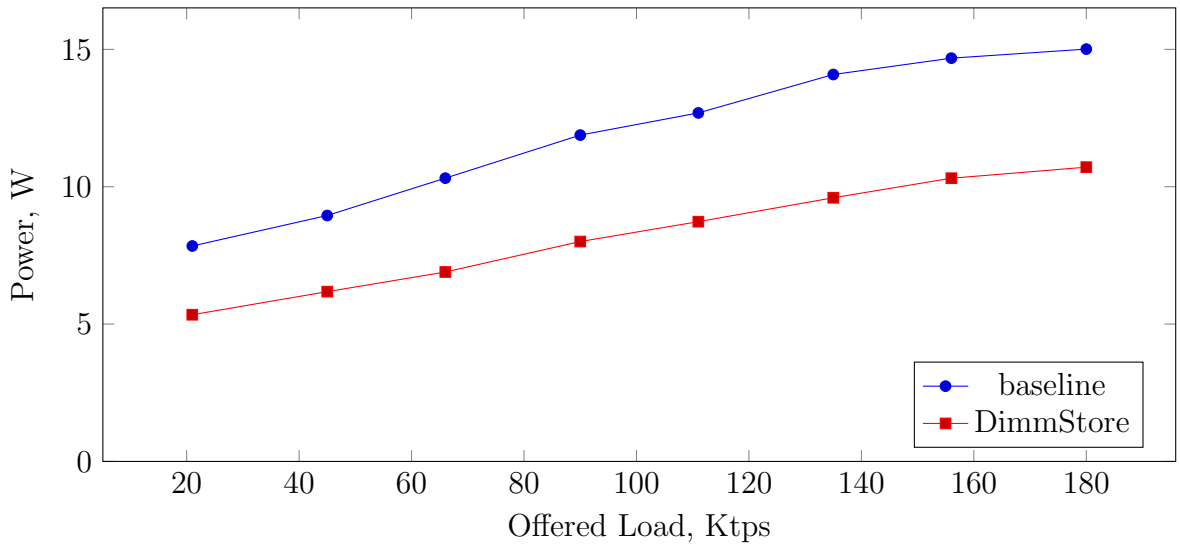

Figure 4.11: YCSB: Memory power consumption by load, 60 GB database

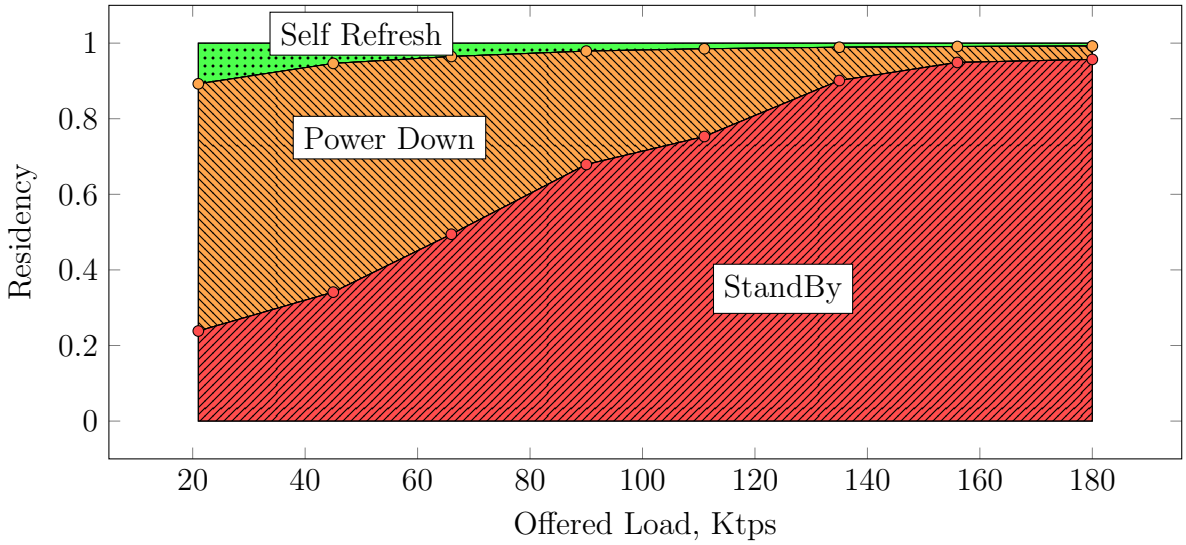

Figure 4.12: YCSB: Average power state residency in the baseline by load, 60 GB database

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 4.12 and 4.13, respectively. Figure 4.12 shows that increasing load increases time spent in the full-power StandBy state, largely at the expense of the Power Down state. For DimmStore, Figure 4.13 shows a similar increase in StandBy state residency, but at the expense of both Self Refresh and Power Down states combined. Self Refresh residency in Baseline is very low, which means there is not enough idleness in memory access pattern for transitions to Self Refresh. The Power Down state is utilized instead due to its short transition interval, but it provides less power savings. Power Down residency decreases with load with explains the change in the power consumption with load.

DimmStore reduces access rate to the DIMMs in the cold region so that the intervals between accessWhile reducing the transaction rate, the Self Refresh residency increases almost linearly, while the Power Down residency stays almost constant. Effectively, Dimm-

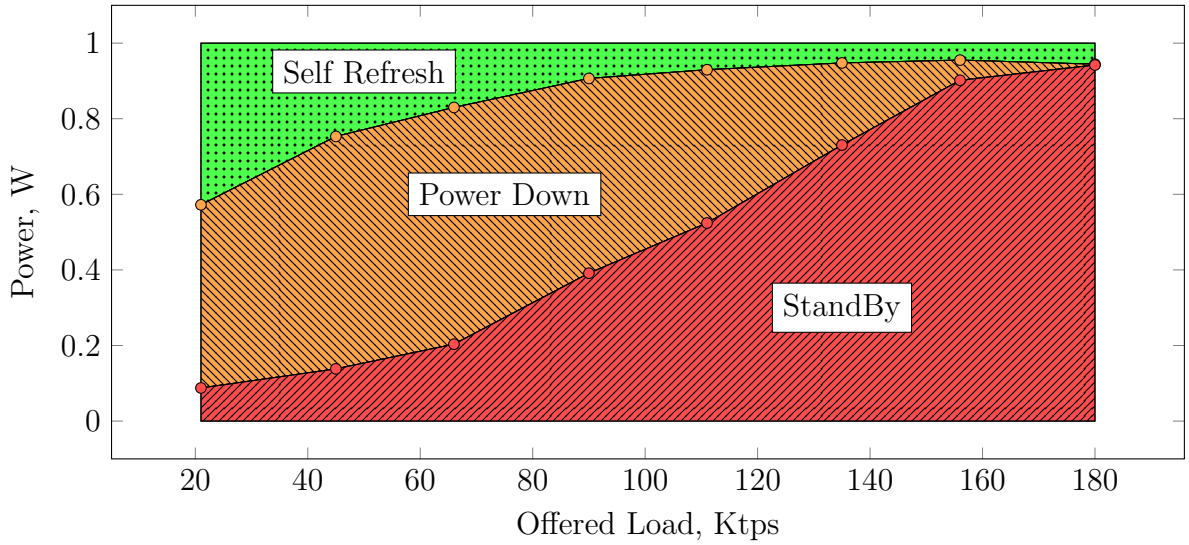

Figure 4.13: YCSB: Average power state residency of non-empty Data region DIMMs in DimmStore by load, 60 GB database.

Store uses Self Refresh, which is a deeper low power state, in DIMM's of the warm region, instead of Power Down during idle periods between memory accesses.

4.4.4 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 towards the hot side of the distribution. As shown in Figure 4.14, 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.

4.4.5 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

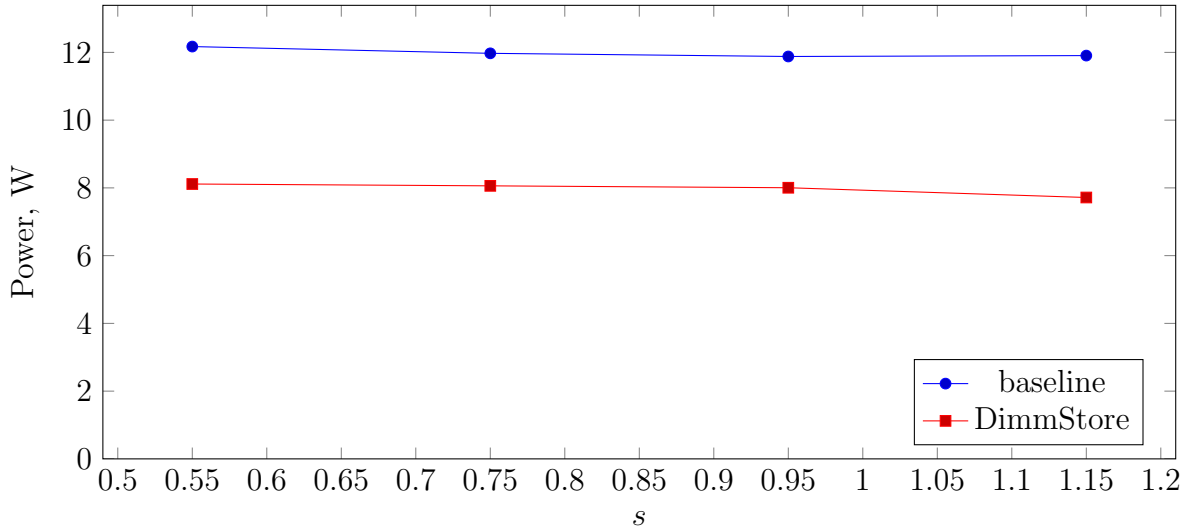

Figure 4.14: YCSB: Memory power consumption by access skew, 60 GB database, 90 Ktps load

for evicting and unevicting tuples from the data region. However, for the YCSB workload, we observed little performance impact from these optimizations.

We evaluate the performance cost of DimmStore's power optimizations by comparing its peak transaction throughput to the one of baseline. Peak throughput in each system is determined by offering it a transaction rate higher than the system can sustain and measuring the actual rate. The benchmark client is also configured in a blocking mode, meaning it senses backpressure 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%.

In addition to peak throughput, we also measured transaction latency at a range of off-peak loads. Figure 4.15 shows mean transaction latency as a function of load. Latencies in the two systems are very similar under this workload.

A low performance impact of DimmStore's power optimizations in YCSB can be explained by its stable tuple access distribution. In YCSB, the probability of hitting a tuple never changes. After a warm up period, DimmStore is able to identify cold data and move it into the data region, where it will tend to remain. Tuple eviction and uneviction rates are the same as the rate of access to cold data, which is low. Thus, the overheads associated with eviction and uneviction are also low.

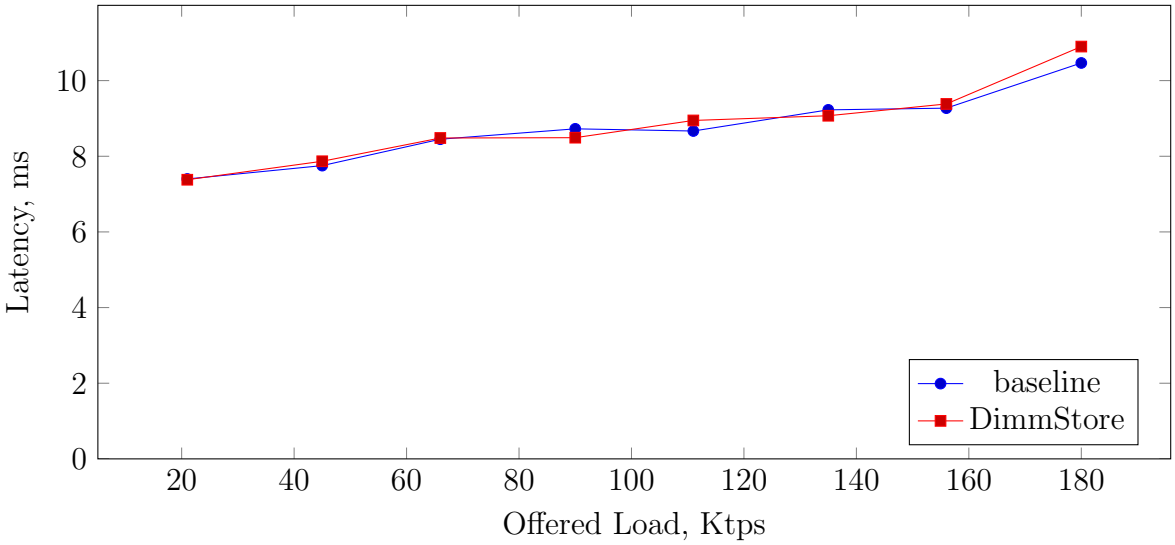

Figure 4.15: YCSB: Average transaction latency by load, 60 GB database

4.4.6 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. Figure 4.16 shows CPU power consumption as a function of load. 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%.

4.4.7 System Region Sizing

In DimmStore, the size of the system region is a configuration parameter, that has to be set by the system administrator before starting the system. The system region cannot be smaller than the size of scratch memory needed by the operating system and the DBMS, and the size of all indexes. Beyond its minimum size, the system region of an optimal size also fit the most frequently used subset of the data tuples to avoid a high volume of data migration due to evictions and unevictions, and the associated heavy performance impact.

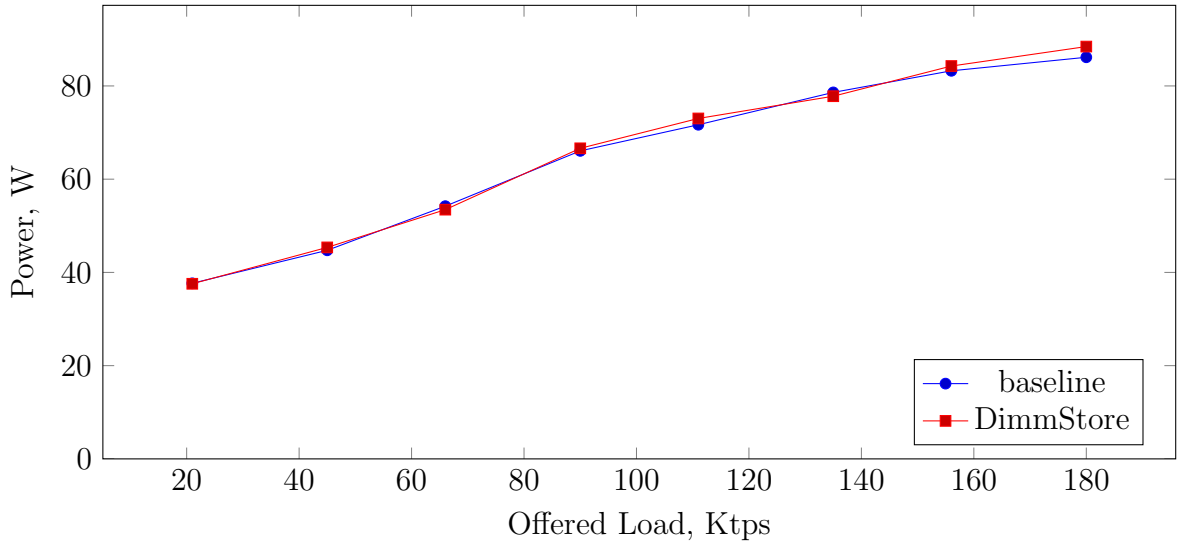

Figure 4.16: YCSB: CPU power consumption by load, 60 GB database

As long as the performance cost of DimmStore is small enough with a System region of a given size, increasing it further will increase memory power consumption due a proportional increases in its background power consumption.

To illustrate this effect, we repeated the experiment with the System region expanded to 4 DIMMs out of 8 total. Figures 4.17 shows memory power consumption and Figure 4.18 shows transaction latency as a function of load, for two configurations of the System memory region. Since the DimmStore's performance impact in the YCSB is minimal, as was shown in Section 4.4.5, the only effect of a larger System region is a higher memory consumption.

4.5 Experimental Evaluation (TPC-C)

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 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.

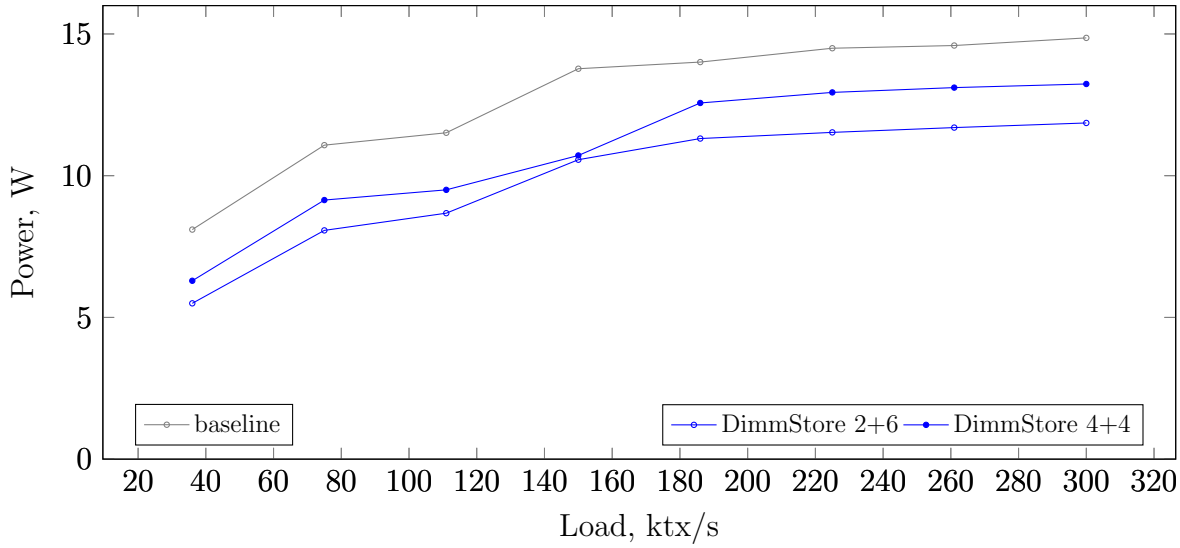

Figure 4.17: YCSB: Total memory power consumption by load, two sizes of the System region, 80 GB database.

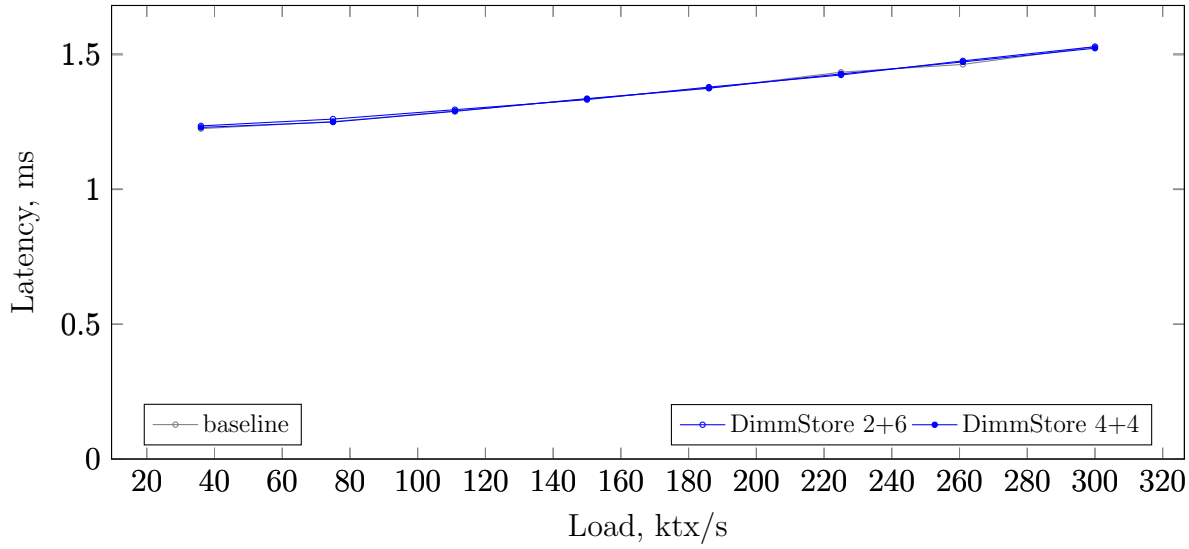

Figure 4.18: YCSB: Total memory power consumption by load, two sizes of the System region, 80 GB database.

Parameter	Values	Default
Database scale factor	100 - 900 warehouses	350
Database size	10 - 90 GB	35 GB
Offered load	8 - 62 Ktps	36 Ktps
System region size	48 GB	48 GB
Eviction interval t_{evict}	1 ms	1 ms
Eviction volume N_{evict}	64 KB	64 KB
Uneviction probability $p_{unevict}$	$\frac{1}{64}$	$\frac{1}{64}$

Figure 4.19: TPC-C Experiment Parameters

4.5.1 Methodology

We used the TPC-C [5] 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 4.19.

Each experimental run has loading, warm up, and measurement phases, as for YCSB. The warm up 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 warm up 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

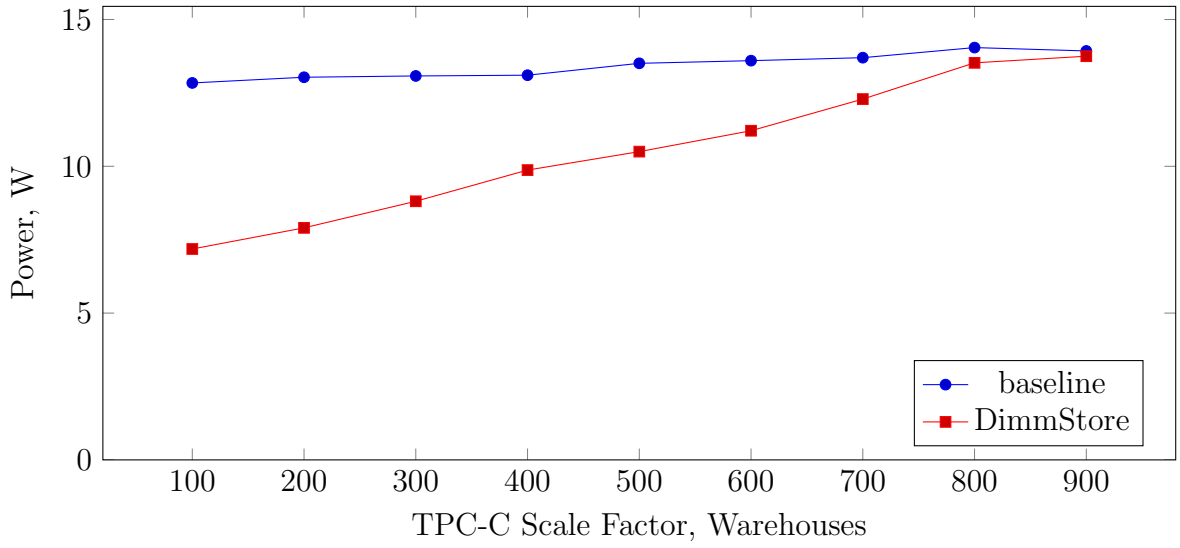

Figure 4.20: TPC-C: Memory power consumption by database scale factor, 36 Ktps load

of the load. We performed runs with offered loads up to 62 Ktps, which is about 90% of the peak load sustainable by the baseline H-Store system.

When presenting measurement results, we exclude the initial warm-up period and report averaged measurements for the following 5-minute interval. Accounting for the warm-up period is necessary because the LRU list of a freshly populated database does not reflect the tuple access distribution in the workload. Therefore, immediately after workload starts, the system makes a poor choice of selecting which tuples to evict. This manifests itself as a sharp spike in the cold region access rate immediately after the workload starts. A local minimum exists because after the initial spike due to the unwarmed LRU list subsides, the cold region access rate starts to slowly increase due to newly inserted index records that gradually reduce the space available for data tuples in the System region.

4.5.2 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 4.20 shows the result of these experiments. The results here are similar to what we observed for YCSB (Figure 4.9). Memory power consumption in the baseline is insensitive to the database size, while DimmStore is able to translate smaller databases

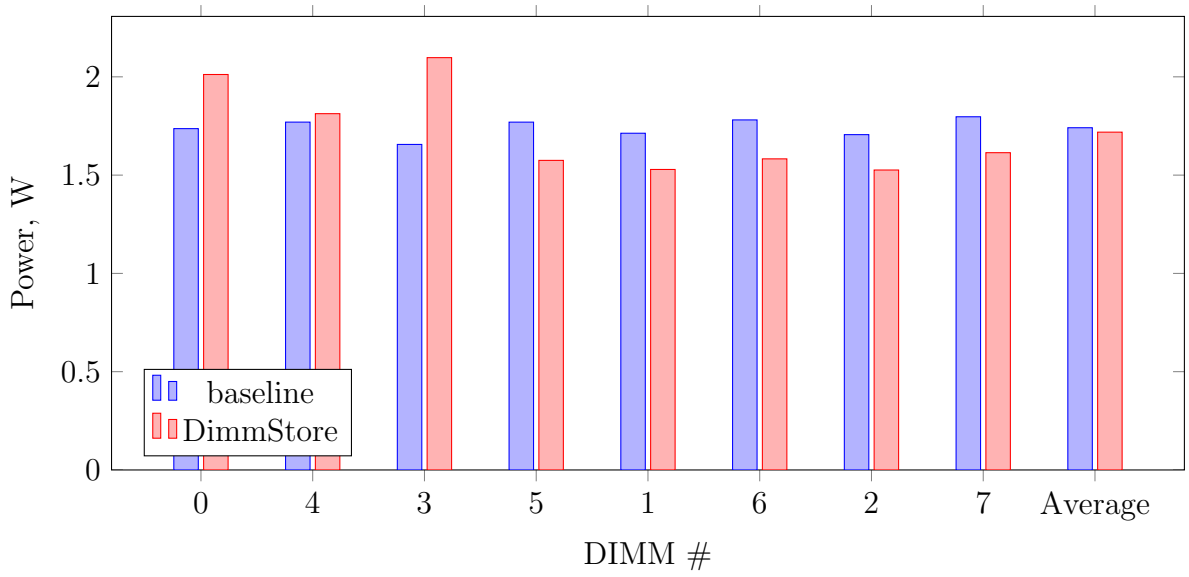

Figure 4.21: TPC-C: Individual DIMM power consumption, DimmStore vs baseline, maximum size database (900 warehouses), 50% load

into memory power reductions. The maximum power savings we observed were about 43%, for the smallest database.

Our TPC-C results differ from YCSB in two ways. First, the memory power savings we observed for the largest databases were smaller for TPC-C than for YCSB. TPC-C's more complex data access patterns make it harder for DimmStore to accurately identify hot and cold tuples. As a result, the memory access rate in the data region is not as low as it is in YCSB. Second, DimmStore's power consumption stops decreasing after the database size falls below 300 warehouses. Once the database is small enough, it fits entirely within the 3-DIMM system region, leaving the entire data region idle. Thus, no further reductions are possible without reconfiguring the testbed to use a smaller system region. This represents a limitation of our testbed, which statically configures the system and data regions, rather than a fundamental limitation of the hardware or the application. In turn, higher warm region access rate causes lower relative power saving in the corresponding DIMMs, as shown in Figure 4.21. Thus, power reduction in the warm region is 0.25 W per DIMM, which is lower than in YCSB. However, the increase in consumption of the System region is 0.36 W per DIMM, which is similar to YCSB. Accounting for the number of DIMMs in each region, the balance becomes only 0.25 W of net power savings.

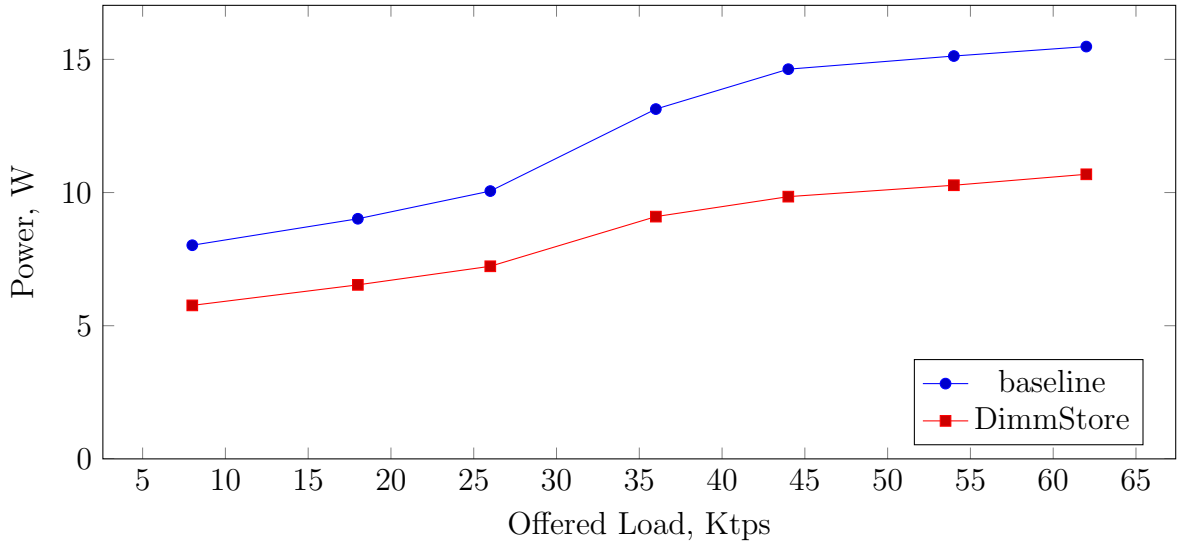

Figure 4.22: TPC-C: Memory power consumption by load, 350 warehouse DB

However, DimmStore draws increasingly less memory power when the database is smaller. Memory power consumption falls almost linearly, until the database becomes so small that it can fit in the System region completely. At that point, DimmStore saves between 30 and 45%, depending on the System region size. The baseline power consumption does not significantly change for various sizes. For the 3 DIMM System region configuration the power balance becomes as follows. Each of the 6 warm DIMMs saves 0.79 W, compared to the average baseline DIMM power, while the 2 hot DIMMs consume 0.24 W more, which nets 3.2 W savings overall. Power consumption does not decrease further with databases smaller than the System region because the memory load is spread between all DIMMs in the System region.

4.5.3 Effects of Load

In our second set of experiments, we fixed the initial database scale factor at 350 warehouses, and varied the offered load. Figure 4.22 shows memory power consumption as function of the offered load. As was the case for YCSB (Figure 4.11), the memory power gap between DimmStore and the baseline is maintained across the load spectrum. For both baseline and DimmStore, power consumption increases with load by about 2.5 W between 25% and 75% load. DimmStore saves approximately 2.5 W in all experiments and this difference is not affected by load. Most part of the power reduction is due to unused

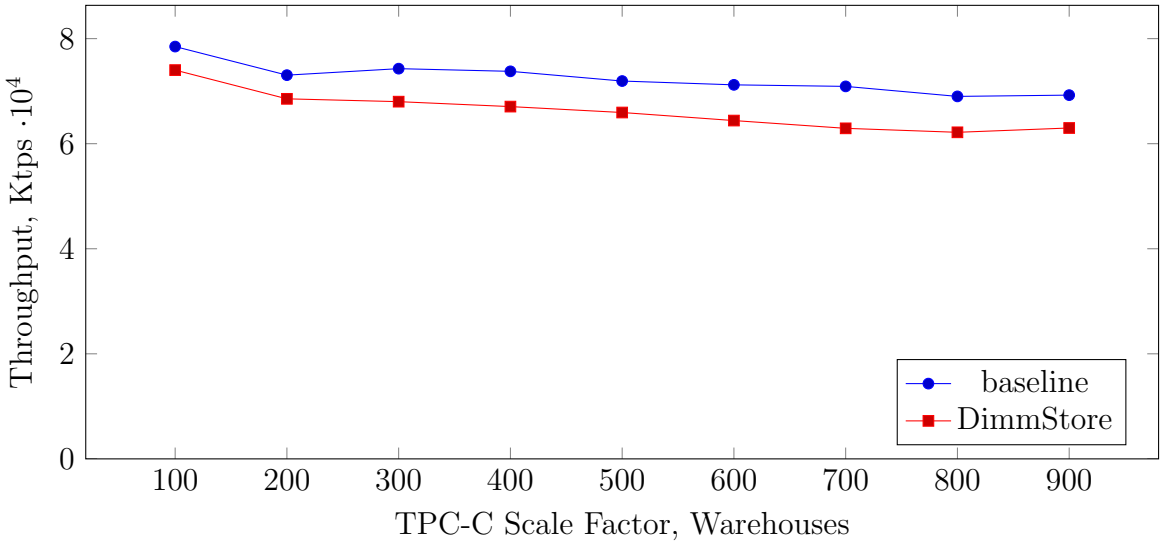

Figure 4.23: TPC-C: Peak throughput by database size

memory space on the last DIMMs in the system, allowing them to stay in the Self Refresh state.

4.5.4 Performance Effects

Next, 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 4.23 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 the throughput gap grows to about 10%.

We also measured transaction latency for both systems at offered loads below peak. Figure 4.24 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

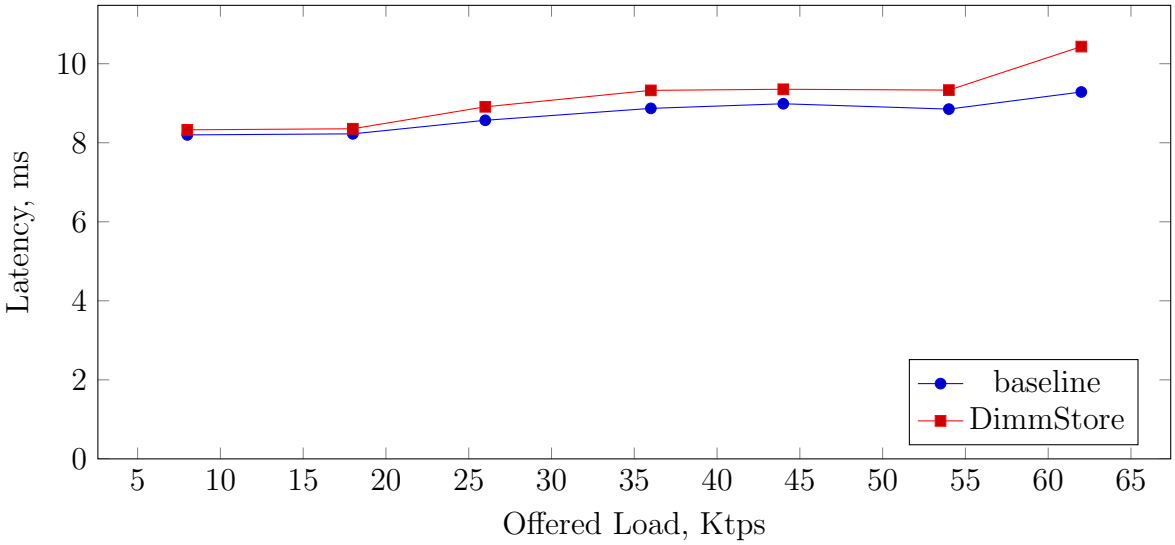

Figure 4.24: TPC-C: Average transaction latency by load, 350 warehouse DB

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.

4.5.5 CPU Power Consumption

As for YCSB, we used RAPL performance counters to measure CPU power, which is shown in Figure 4.25. 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.

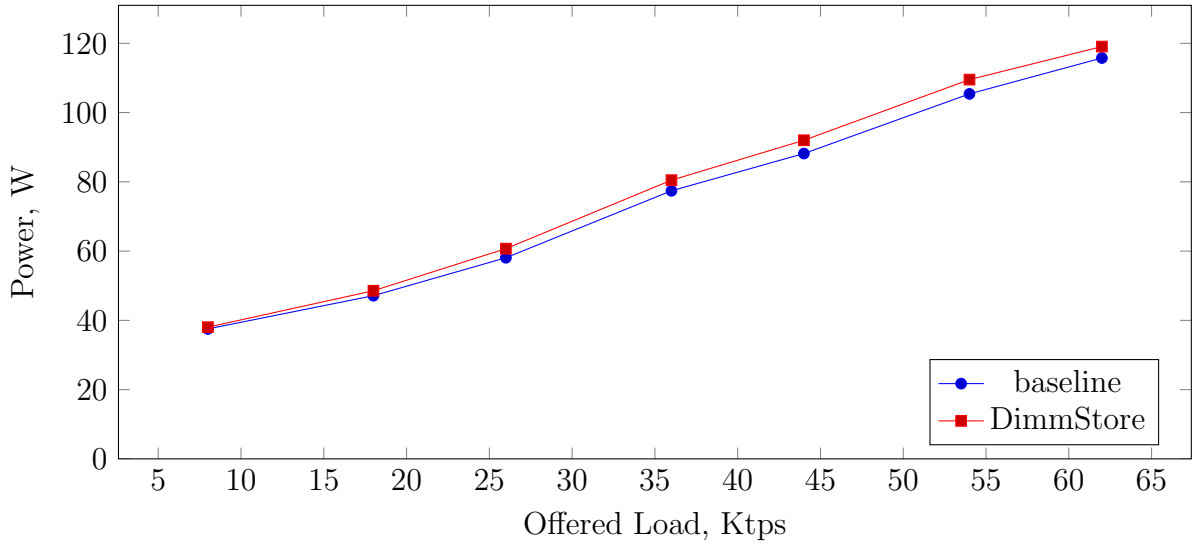

Figure 4.25: TPC-C: CPU power consumption by load, 60 GB database

4.6 Discussion

In this chapter, we presented DimmStore, a prototype of a power-efficient database system that reduces memory power consumption by placing database regions with different access rates into separate sets of DIMMs. We demonstrated that DimmStore reduces memory power consumption by up to 50%. Relative power savings are higher with smaller databases.

DimmStore imposes a moderate impact on system performance, which depends on the workload access patterns. The performance impact is negligible in the YCSB workload, which has simple and static data access pattern. In TPC-C, which access pattern is more complex, the performance impact is below 10%.

DimmStore works by placing rarely accessed data into a memory region, backed by a distinct subset of DIMMs. When the database is smaller than the amount of available memory, only some of the DIMMs in this regions are used. DimmStore minimizes power consumption of the unused DIMMs, which will always stay in the deepest low power state (Self Refresh). The power savings in the used DIMMs depends on the access rate to these DIMMs. If the memory controller employs a conservative power state policy, even the reduced access rate may not result in long enough intervals between access to DIMMs of the Data region.

One technique to reduce the power consumption in used DIMMs of the Data region, once the access is reduced by DimmStore, is to introduce temporary idle periods. It will be discussed in Chapter 5.

4.7 Related Work

DimmStore uses memory low-power states as a mechanism to reduce memory power consumption. Existing work on memory power optimization that are also based on the use of lower-power states are presented in Section 4.7.1. To maximize memory idleness in its Data region, DimmStore relies on classifying data tuples into frequently and infrequently used. Existing work on data classification is covered in Section 4.7.2.

4.7.1 Memory Power Optimization with Power States

Leveraging memory power states has been an important direction of research in power efficient systems. Power states are attractive targets because their power consumption differs significantly, indicating that high energy saving can be achieved if low-power states are better utilized. However, lower power states are associated with delays in memory access. Therefore, the actual question is finding the optimal strategy of state transitions to balance the power and performance goals.

In this section, I describe existing techniques for memory power state management. In current systems, DRAM power state transitions are steered by the memory controller and there are no *direct* means for software to control them. Therefore, I will start with pure *hardware* approaches. However, hardware approaches may be limited in effectiveness because of the small amount of information about the state of the program execution available to the memory controller. Some authors argue for bringing the control of state transitions to the *software*, assuming that a power state control mechanism can be added in the future, if this helps save power. Alternatively, it is also possible to *indirectly* cause memory to transition to a lower-power state by shaping the memory access. This approach is coarser, but it is available for exploration now. The software-based techniques for power state management will be described later in this section.

Hardware-directed Power State Management

The memory controller utilizes complex algorithms to schedule read and write requests from the CPU to satisfy DRAM timing requirements, with the primary goal of maximizing

performance. Minimizing DRAM power consumption has become another goal, but these two goals are in conflict with each other. There are two reasons for this. First, power state transitions incur additional access latency. Second, performance-oriented scheduling causes memory accesses to spread more evenly (over time and hardware interfaces), reducing the opportunity for memory devices to utilize low power states.

A memory controller with the ability to use low power states is said to implement a *power management policy* to balance the conflicting performance and power requirements. Such a policy can be considered a hardware-based memory power optimization technique that is transparent to software running on the system. A simple and widely used memory power management policy is a timeout-based one [53, 34]. In such a policy, a unit of memory, such as a rank, transitions to the low power state after it has been idle longer than a specified threshold time. When multiple states are supported, each is associated with its own threshold value, with longer ones for deeper states. Choosing the right threshold is a balancing act: going to a low power state too soon will negatively impact performance while longer thresholds waste more energy before the transition occurs. More advanced *adaptive policies* [40] consider the state of the scheduling queue(s) to predict future accesses and control the state transitions with better accuracy. The mechanisms based on the memory controller request scheduling are effective in exploiting shallow power states. However, because of a limited queue depth, they cannot create period of idleness long enough to enable deeper power states, such as Self Refresh.

A pure hardware technique proposed by Amin and Chishti [10] extends periods of rank idleness by delaying eviction of some cache lines from the CPU cache. Since the CPU cache has much larger capacity than a reorder queue in the memory controller, there is a potential to delay accesses for longer periods. The proposed cache implementation tracks the rank information for each stored cache line. Among all ranks, some are designated as *prioritized* ranks, and the eviction algorithm is modified to avoid evicting cache lines belonging to prioritized ranks. Periodically, the set of prioritized ranks is rotated to allow the delayed cache lines to be eventually evicted. The potential cost of prolonged cache line residency is cache pollution and the resulting increase in cache misses and a drop in performance. However, balancing the maximum duration of rank prioritization limits the negative effects while allowing idle intervals long enough so that deeper low power states can be used.

Compiler-directed Power State Management

In compiler-driven approaches, code generation for a program being compiled is adjusted to maximize idle intervals in accessing memory ranks. Delaluz et al [27] presented a set of

compiler optimization techniques aimed at reducing the number of used memory ranks in a system with multiple ranks. The techniques use static program analysis and target the program fragments that operate on *arrays*. Examples of array-dominated applications include linear algebra solvers and video processing. The first step of the proposed optimizations is an array allocation algorithm which tries to place program arrays with similar access patterns on the same set of ranks. This technique alone improves the rank access locality of the program, and thus power efficiency. Once the best array placement is computed, the other techniques make further improvements. *Loop Fission* converts a loop consisting of several independent statements into several loops. As long as the independent statements access arrays from different ranks, a smaller set of ranks can be activated at a time. *Loop Splitting* is an orthogonal technique that converts loops over a large array intervals into a series of loops over parts of these intervals. It can reduce the number of used ranks when several arrays spanning multiple ranks are accessed in the loop. Finally, *Array Renaming* reduces the program memory footprint by reusing the space taken by previously processed arrays once they are not needed. The compiler also inserts *power state control instructions* when it expects the set of used memory ranks to change, so an explicit power state control by software is assumed. The advantage of the presented approach is its ability to optimize power consumption of existing code without modifying it. However, since the techniques depend on the details of the memory configuration, such as the size of the ranks, recompilation is required when the program is used on a different system. Since array-like data structures are widely used in DBMS's, especially the main-memory ones, same ideas can be employed when designing its algorithms, so that compiler "magic" is not necessary.

The trade-off between performance and memory power consumption of the code generated by the compiler is explored by Wang and Hu [76]. In their work, variables are partitioned between memory ranks according to the *maximum cut* algorithm. Then, an instruction scheduling algorithm is used to chooses between favouring performance or favouring power efficiency A high-performance schedule is shorter because of more parallelism between memory accesses. However, for a given length of the schedule (number of clock cycles), the proposed algorithm can choose the schedule that maximizes the length of idle intervals in memory ranks. Similar to reordering of memory commands in the memory controller queues, an instruction scheduling algorithm in the compiler can only improve memory idleness over the short term because the reordering distance is limited by program analysis capabilities.

OS scheduler-based techniques

A very common approach to maximize rank idleness is to modify the virtual memory and scheduling subsystems of an operating system. The technique by Delaluz et al [28] gives the control over memory power states to the OS scheduler. The OS tracks memory pages accessed by a process used the "referenced" bit in the page table, which is managed by the CPU's Memory Management Unit (MMU). Assuming that the OS knows the mapping between memory pages and ranks, it can collect per-process *rank usage* information. When a process is scheduled to run, the memory ranks that are known to be used by the process are switched into the active state, and the remaining ranks go into a low-power state. The proposed mechanism is very basic, as it relies on *passive* monitoring of rank usage without an attempt to minimize the rank set by steering the allocation strategy. Memory power efficiency is improved compared to the automatic timeout-based power state management because it eliminates energy consumption during the unnecessary timeout intervals. However, the overall efficiency would depend on whether processes' memory access patterns tend to naturally cluster in a small subset of memory ranks.

Power-Aware Virtual Memory (PAVM) by Huang et al [37] is a step further in the same direction, and addresses the problem of minimizing the set of used ranks per process. Since finding the optimal solution would be computationally expensive, their approach is heuristic-based. The OS keeps track of the memory ranks used by a process (*preferred nodes*) and any further allocations are directed to one of these ranks, as long as there is space available. The preferred node set is expanded by one rank at a time once it runs out of space. Switching the power state of memory ranks is performed by the OS scheduler during a context switch. The study highlights the challenges that arose during evaluation of the practical implementation. A serious problem is memory *sharing* between processes, which causes processes' memory to become scattered across many ranks. Common causes of memory sharing are memory-mapped shared libraries and the page cache. The problem of shared memory was addressed by directing allocation of shared paged to a separate set of ranks and migrating pages back to the process private ranks once they cease to be shared. This study is notable due to its practicality, as it was shown that the proposed technique can be implemented in a real system with little performance overhead. However, its effectiveness relies on whether memory usage can be partitioned by a process. The described approach seems more appropriate for multi-tasked mobile environments where running applications are only occasionally active.

In two related papers [42, 41], threads are clustered into groups so that each group is associated with one memory rank. Memory allocations from a thread group are directed into the associated rank so that the association between thread groups and ranks is main-

tained. The threads in a group are scheduled together so that rank accesses are batched, increasing memory idleness for the remaining ranks. However, the papers provide few details on how thread groups are formed and how shared memory is handled.

The important idea of maximizing rank idleness by exploiting or creating access skew between memory ranks was initially applied at the OS level. The concept of hot and cold ranks for memory power saving was introduced by Huang et al [38]. The authors made the observation that the deeper Self Refresh power state *break-even idle interval* is very long so that this state is almost never entered. The solution would be to separate *memory pages* according to their access frequency so that the pages with higher access rates are placed to the hot ranks, and the ones with lower rates go to the cold ranks. It was assumed that a mechanism exists to estimate access frequency of a memory page, for example, by a memory controller or using page faults. The proposed solution required physical memory pages to be copied between ranks, as described in an earlier work [37], which may introduce additional performance and energy cost. However, since memory page access distribution is uneven, in most workloads it is only necessary to migrate a small percentage (1.5-14.4%) of all pages to maintain sufficient hot/cold rank separation. My work is also based on the same concept of data segregation into hot and cold ranks based on access frequency. However, the solution by Huang et al [38] is application-oblivious and would be implemented at the OS level. In my work, the memory power management is moved to the application (database) level, which allows for a finer granularity of hot/cold separation (tuple level vs page level) and potentially a more accurate and less costly access frequency estimation.

A follow-up work on OS-based memory power management by Wu et al [77] improves on the hot/cold classification memory pages. Instead of relying only on access counters, pages are classified using the MQ algorithm. The MQ algorithm maintains M LRU queues and higher queues store pages that are more recently used in addition to ordering pages by access recency in each queue. The queues are mapped to memory ranks according to their queue number and the relative position in the queue. Once the ideal page placement is produced, a minimum set of migrations is computed that brings the page placements to the previously found mapping. One advantage of the MQ algorithm is its ability to capture both frequency and recency information on page accesses. Another advantage is that the "temperature" of the ranks increases gradually, as opposed to having hot and cold ranks. The proposed technique demonstrates an improved idle time prediction accuracy, which comes close to an offline optimal algorithm.

Improving Memory Rank Idleness in DBMS

Most of the work on improving memory power efficiency did not target a particular class of applications. There are studies to *characterize* memory power in database systems (Section 3.6), but they only suggest directions to save on memory power. One database-specific work by Bae and Jamel [14] aims at limiting memory power consumption in a disk-based DBMS under varying load. They have modified a DBMS to dynamically reduce the buffer pool in order to allow the unused memory to sink into a lower power state. Since the DBMS controls the memory allocation of the buffer pool, it can estimate the required size and adjust the amount of allocated space accordingly. In the baseline, even though some buffers are unused, the physical memory pages are scattered over the memory ranks, preventing any of them from becoming idle. This work highlights the advantage of application-level control of memory allocation over intrinsic hardware and OS-driven methods. The limitation of this work is that there are only two states of the buffer pool: *small* and *large*, each with a predefined size. This limitation reduces the potential of energy efficiency for arbitrary workloads, as well as introducing performance transients during the time when the buffer pool capacity changes.

4.7.2 Data Classification

To save memory power by better utilizing its low power states, longer idle periods are needed. A simple way to increase idleness, is to separate memory space into a *hot* and *cold* regions and maintain access skew between the regions. In this setting, the problem of placing data elements between the regions becomes similar to the classical problem of designing a replacement policy in a buffer pool (cache). The goal of a replacement policy is to ensure a minimum access probability of secondary storage (buffer pool miss), given a smaller buffer pool located in main memory. Scores of replacement policies have been proposed for OS and DBMS buffer pools over the decades. They differ by how accurately they predict the likely used elements, which is workload-specific, and by computational and memory costs. I will briefly outline a few of them, mainly to illustrate the cost aspect. The Least Recently Used (LRU) [13] algorithm is probably the most ubiquitous. The LRU algorithm reasonably well approximates the optimal replacement policy, and is trivial to implement. A typical implementation uses a doubly-linked list, which requires a substantial space cost of two pointers per element. On every access of an element that is currently in the cache (the normal case), at least four random memory writes are needed. The LRU algorithm also fails to effectively handle some important access patterns, such as scans.

The CLOCK policy [23] is a cost-efficient approximation of LRU. The space overhead of CLOCK is only 1 bit per element and at most one memory write is needed per element eviction (amortized). Despite its simplicity, CLOCK performs comparably to LRU and, therefore, widely used in operating systems as a page replacement algorithm. In that case, the access bit is updated by the CPU hardware.

The two general-purpose extensions of LRU and CLOCK correspondingly are ARC [57] and CAR [15]. They substantially improve buffer pool efficiency, handle sequential scans, and reduce the work in the hit path. However, both come with the additional memory overhead for maintaining variable-length lists for elements in the pool, as well as a subset of the *evicted* ones.

In traditional DBMS and OS the element of the buffer pool is a *block* or *page*. The size of both in the order of kilobytes and a space overhead of several words per element is generally acceptable, as well as a cost to update a reference bit in the hit path. However, main-memory databases get away from the concept of a data block and build their data structures directly from individual tuples. For a system that handles datasets larger than its memory, it is natural to identify skew and manage data migration at the individual tuple level. Note that this applies primarily to OLTP systems, as in-memory OLAP databases tend to store data column-wise, which will be discussed later.

Levandoski et al [69] specifically addressed the problem of hot/cold data classification in an OLTP system. They estimated that a traditional LRU, LRU-2, or ARC algorithms would incur at least 25% processing overhead. The corresponding space cost of 16 bytes per tuple would also be significant. Therefore, an *offline* strategy was proposed where tuple rank is not immediately updated during normal operations. Instead, tuple accessed are written to a separate log and access frequencies are computed based on this log only at certain discrete times. Such a two-step approach has the advantage of overall low computation cost and flexibility to schedule the disrupting offline classification portion. For offline classification, an efficient algorithm reads log records *backwards* from the end of the time interval. The algorithm may produce accurate frequency estimates before reading the whole log and terminate early. Moreover, since frequency information is only needed for *hot* tuples, the tuples with low current estimates are discarded so that the size of the data structure is kept constant regardless of the total database size. The accuracy of classification was found experimentally to be *higher* than of LRU-2 and ARC in the TPC-E workload, using the miss rate as the metric. The same offline algorithm is used in the work by Stoica and Ailamaki [68] to adjust the tuple placement in OS memory pages so that the OS paging mechanism can be effective for migration between main memory and secondary storage.

In E-Store [71], tuple-level monitoring is used for dynamic load balancing. The actual hot tuple identification algorithm is expected to work for a short duration, when a partition is found to be overloaded with a per-partition performance monitoring. The algorithm involves counting accesses to individual tuples over a short time interval and partial sorting to extract the hottest tuples. Such a rudimentary algorithm is not suitable for general-purpose continuous tuple identification and its accuracy may be limited by the short data collection interval.

Handling datasets larger than memory is also important for main-memory OLAP systems. Early systems, such as MonetDB [17], rely on OS virtual memory management to exchange pages between memory and storage. The database file is mapped into the process address space and the pages are brought in on access. In the case of memory pressure, the OS will start swapping out pages according to its replacement policy, such as CLOCK.

SAP HANA is a column store that loads whole columns on demand and evicts unused ones according to a weighted LRU strategy [67] where the weight is assigned heuristically. Instead of such a coarse-granular loading and eviction, a column can be configured as *page loadable*. In that case, the three component of the column, data vector, dictionary, and inverted index are split into fragments that can be loaded and unloaded individually.

Chapter 5

Memory Access Gating

In Chapter 4, we introduced DimmStore, a prototype database system aiming at reducing memory power consumption by exploiting and enhancing idleness between database memory accesses. DimmStore increases the average amount of "useful" memory idleness, consisting of longer idle intervals, by concentrating frequently accessed memory in a small number of DIMMs.

DimmStore saves memory power for two reasons. First, because DimmStore uses rank-aware memory allocation and rate-based placement, some DIMMs may become completely idle if there is enough unused memory. The savings depend on the amount of unused memory. Second, because DimmStore places cold data in the Data region, access rates to the used portion of the Data region may become low enough to enable the DIMMs in this region to spend some time in the low power Self Refresh state.

For the second mechanism to be effective, the access rates in the used DIMMs of the Data region must be so low that a substantial percentage of memory interaccess intervals are longer than the Self Refresh timeout. Achieving such low rates requires substantial skew in the access rates between DimmStore's System and Data memory regions. DimmStore will show little or no improvement in memory power efficiency if there is little skew or if it is difficult to identify frequently accessed tuples.

In this chapter, we present and evaluate a complementary technique that specifically targets power consumption in the used DIMMs in the Data region. This technique works by concentrating memory accesses in *time*, as opposed to *space*, as in DimmStore. This is achieved by introduction of forced idle intervals, called *restricted intervals*, during which access to the Data region is *gated*, i.e., it is disallowed.

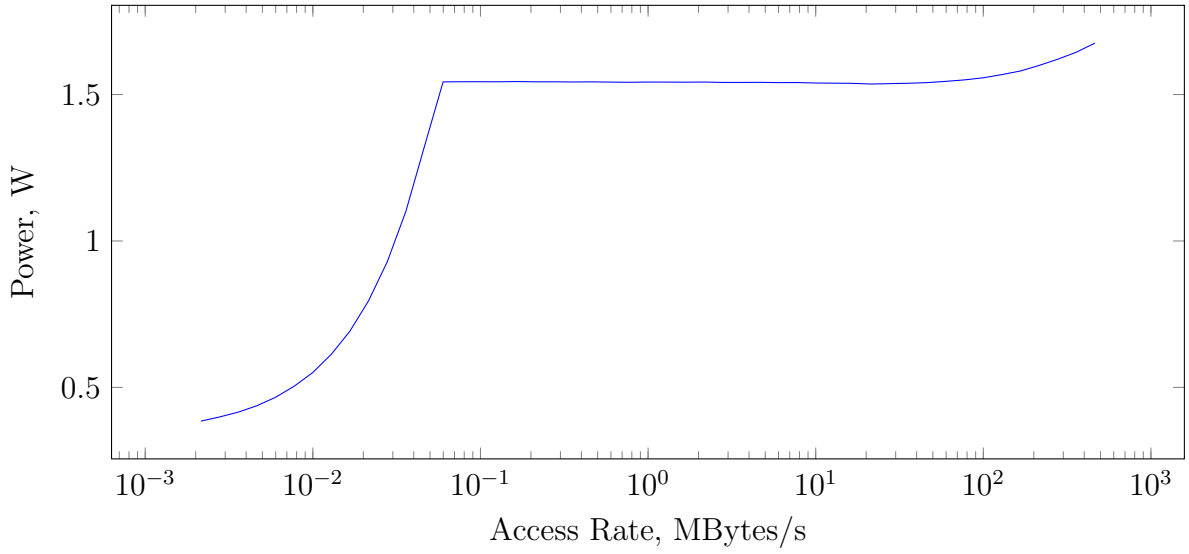

Figure 5.1: DIMM power consumption vs. random access rate, local DIMMs

The memory gating mechanism builds on top of rate-based placement. It is possible to implement a similar mechanism in the baseline system, which treats all memory as a single region. However, access to *all memory* would have to be gated, meaning that no program could access memory during the restricted interval.

5.1 Concept of Resource Gating

Memory is an example of a computational resource that exhibits non-proportional power consumption as a function of load. Power consumption of a memory DIMM only slightly changes with load over almost all load range, except when the load is very low. To illustrate this load to power relationship, we ran the random access rate workload, described in Section 2.3. In each experiment with a certain access rate, the DIMM's power consumption was measured. The measured power as a function of access rate is shown in Figure 5.1 (note the logarithmic horizontal scale). DIMM's power only slightly changes when the access rate spans hundreds of kilobytes to gigabytes per second, but it starts to drop significantly when the access rate is below the very low threshold of about 70 kB/s. This behaviour is determined by the memory's power states and their conservative application by the memory controller, as described in Section 2.3.

For such non-power-proportional resources, power consumption can be reduced by alternating periods of complete idleness with periods of concentrated load. The average utilization remains the same, but it is concentrated in time.

When naturally occurring idle intervals in the resource access stream are not long enough, the system may introduce longer idle intervals by periodically restricting access to the resource for a certain duration, as illustrated in Figure 5.2. We refer to this as *gating*. When a thread (program, transaction) attempts to access the resource during a restricted interval, these accesses will be delayed until the next unrestricted interval, temporarily blocking the program but allowing the resource to remain idle. Although access gating will delay execution of threads that attempt to access the gated resource, threads that do not access *that particular* resource run without interruption. In particular, if a system has multiple independently gated resources, for example, multiple memory DIMMs, only some threads will be affected when a resource is gated.

5.2 Expected Effects of Memory Gating

In this chapter, we consider the application of gating to the DIMMs in DimmStore's Data region. We expect that this will have the following effects:

Increased utilization of low power states and reduced power consumption in the Data region DIMMs. By creating idle periods, restricted intervals allow the memory controller to use lower power states, such as Self Refresh, saving power. Ideally, Data region DIMMs would spend the entirety of each restricted interval in the Self Refresh state. In practice, since the memory controller needs time to detect idleness, the amount of Self Refresh may be slightly lower than that, but still covering much of the restricted interval.

Higher memory utilization in both the System and Data region DIMMs during the unrestricted intervals. Memory utilization during the unrestricted interval will increase, compared to memory utilization without gating. This is because transactions that attempted to access gated DIMMs during the restricted interval will stall, resuming when the restriction is lifted. Increased memory utilization will cause memory to consume more power during the unrestricted intervals. However, we expect this increase to be smaller than the power savings during the restricted intervals for two reasons. First, active power during the unrestricted interval will increase by the same amount as active power is saved during the restricted interval because some memory accesses are merely shifted in

time. More importantly, low power state utilization was likely already low without gating. Thus increased utilization during the unrestricted interval will not be able to cause a large increase in background power consumption since background power will have already been close to its maximum.

Higher transaction response times. Since transactions may stall during the restricted interval, some transaction completions will be delayed. By varying the lengths of the restricted and unrestricted intervals, we expect to observe a trade-off between the increase in transaction latency and memory power savings.

5.3 Memory Gating in DimmStore

To confirm our expectations and quantify their effects, we prototyped memory access gating in DimmStore. Recall from Chapter 4 that DimmStore divides memory into System and Data regions. DimmStore keeps the rarely accessed database tuples in the Data region so that the access frequency to the DIMMs of the Data region is minimized.

Our implementation applies memory access gating to the Data region. The DIMMs of the Data region are made to alternate between *restricted* and *unrestricted* states, or *intervals*, as shown in Figure 5.2.

The lengths of these intervals are statically configured. Transactions attempting to access the Data region during the restricted interval are blocked for the remainder of the restricted interval. As a result, DIMMs in the Data region stay idle at least for the whole duration of each restricted interval. Transactions' accesses to the Data region outside of the restricted interval are unaffected, as are accesses to the System region.

Although the Data region normally consists of multiple DIMMs in the Data region, the prototype implementation treats the whole Data region as a single gated resource. Therefore, restricted intervals are imposed on all DIMMs of the Data region at the same time.

5.3.1 Gating Configuration

With memory gating, the Data region alternates between restricted and unrestricted intervals. Therefore, the gating configuration can be defined by the length of the cycle t

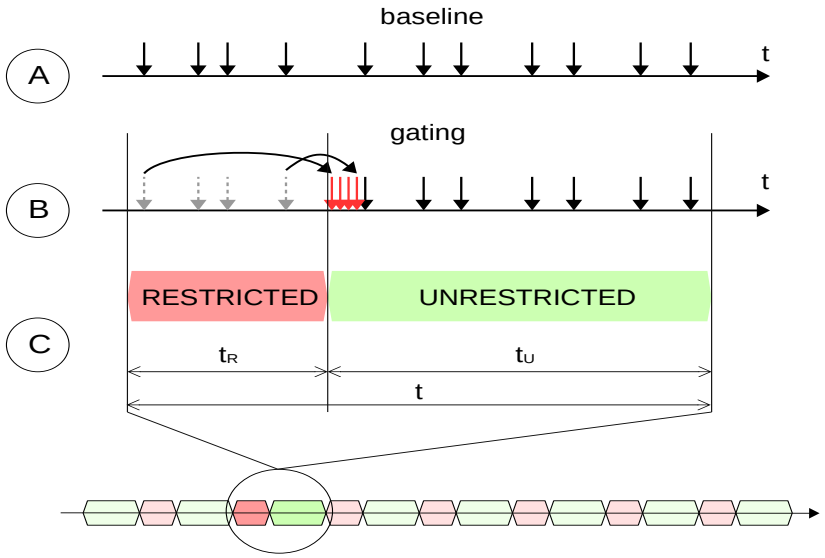

Figure 5.2: Concept of memory gating. A. memory accesses without gating; B. memory accesses modified by gating; C. gating cycle, restricted and unrestricted intervals, and their lengths.

and the length of the restricted interval t_R, as shown in Figure 5.2. Considering the unrestricted interval as "active time", we can relate it to a conventional notion of duty cycle D as $D = 1 - \frac{t_R}{t}$.

In the remainder of the chapter, we will also refer to the *compression factor* k, which is simply the inverse of the duty cycle.

$$k = \frac{1}{1 - \frac{t_R}{t}} = \frac{1}{D} \tag{5.1}$$

Intuitively, compression factor describes how much the utilization of the restricted resource has to increase during the unrestricted interval such that the same average access rate is maintained as without gating. For example, if if the duty cycle is 50%, the compression factor $k = 2$. This means that we expect the memory access rate during the unrestricted interval to be twice as high as it would be without gating.

5.3.2 Enforcing Restricted Intervals

Restricted intervals are enforced by each DimmStore worker thread in the code path that handles accesses to tuples in the Data region. Every time such a tuple is to be accessed, the thread looks up the current time and determines the *offset*, which is the time passed since the start of the current cycle. Total cycle duration is constant, therefore, the offset is the remainder after dividing the current time, expressed in time units since the epoch, by the length of the gating cycle t. Each cycle starts with the restricted interval, therefore, if the offset is less than the restricted interval duration t_R, then the restricted interval is in effect.

If the restricted interval is in effect, the worker thread stalls (sleeps) until the end of current restricted interval. The thread computes the end time of the current restricted interval, and blocks by calling `clock_nanosleep` function.

5.3.3 Indirect Stalls

H-Store, and hence DimmStore, implements a non-blocking architecture, in which one thread processes transactions in each partition. Having only one worker thread per partition allows H-Store to avoid the overhead of synchronization to maintain data consistency. In H-Store, worker threads are not blocked as there are no traditional causes to block such as I/O or database locks.

The single-threaded DimmStore architecture, when combined with gating, results in *indirect stalling*, in addition to the stalling when a transaction attempts to access gated memory. When a transaction stalls, the worker thread in its partition blocks, preventing further transactions in this partition from executing until the end of the restricted interval, even if they do not access the gated memory. Transactions executed by workers in other partitions are not affected.

Because of indirect stalls, the performance of the gated system will degrade significantly, even when transactions rarely access gated memory. In the worst case, when all transactions during the restricted interval stall, the system throughput will be k (compression ratio) times lower than in an ungated system. Since worker threads block on the first access to gated memory during the restricted interval, the actual performance impact will be lower when transactions rarely access gated memory.

If gating is implemented in a system that can execute multiple transactions at the same time, unlike in DimmStore, we expect the impact of gating to be lower than in DmmStore.

5.3.4 Synchronizing Workers

DimmStore uses multiple worker threads, each responsible for its own database partition. Each DIMM in the Data region is shared by all partitions. As a result, each Data region's DIMM may be accessed by any worker thread.

To create idle intervals in the Data region DIMMs, it is necessary for worker threads to enter and exit restricted intervals at the same time. Worker threads read the current time from `gettimeofday` to detect the start of the restricted interval and use absolute time in the `clock_nanosleep` system call to enter the blocking state that ends at the end of the restricted intervals. In Linux, the `gettimeofday` and `clock_nanosleep` system calls provide a time reference that is synchronized between cores and threads. Therefore, there is no need for additional coordination between threads to ensure that the restricted interval starts and ends at the same time.

5.3.5 CPU Cache Write-backs

Stalling DimmStore's worker threads ensures that they will not read or write Data region DIMMs during the restricted interval. However, writes to the Data region during the unrestricted interval may be cached by the processor and written back to memory asynchronously at a later time. Normally, timing of these cache line write-backs is determined

by the processor. If they occur during a future restricted interval, memory idleness during that interval will be significantly reduced.

In DimmStore, there are two sources of writes to the Data region: tuple eviction and in-place tuple updates. Tuple eviction occurs periodically, when a batch of cold tuples from the System region is written to the Data region. In-place tuple updates in the Data region occur because on access, tuples are only moved to the System region with a certain probability, as described in Section 4.1.4. To minimize the number of unevictions and the associated overhead, such as index updates, DimmStore triggers an accessed tuple uneviction only for a configurable fraction of accesses to the Data region. With gating, these in-place updates create modified cache lines for Data region memory.

To solve the problem of cache write-backs during the restricted interval, DimmStore workers issue CLFLUSH instruction after both types of tuple updates in the Data region. The CLFLUSH instruction in Intel processors writes back a cache line to main memory, if it is dirty, and removes the cache line from the processor caches. This solves the problem of deferred write-backs during restricted intervals. However, flushing cache lines may also have a negative impact on system performance as any subsequent access to that cache will have to read it back from the main memory.

5.4 Experimental Evaluation (YCSB)

The goal of this section is to experimentally evaluate the power and performance effects of memory gating. We used our gating prototype, based on DimmStore and described in Section 5.3.2, to evaluate the memory gating mechanism. We will compare it to DimmStore without gating, as well as the baseline H-Store.

The outline of this section is as follows. We first present memory power consumption and system performance, under the YCSB workload, using a default system configuration and database size. Later, we explore how the workload characteristics, system configuration, and gating parameters affect the results. Finally, in Section 5.5, we present a similar analysis for the TPC-C workload.

5.4.1 Evaluation environment

Hardware Configuration

The database system under test runs on a dedicated server instrumented for memory power measurement. The load is generated from a separate load generator (client) machine. The client and server machines are connected with a 1 Gbps switched Ethernet network. Due to the physical separation of the database server and load generating client, power consumed due to load generation is not included in the results.

The database server has two 8-core Intel Xeon E5-2640 v3 processors with a nominal frequency of 2.6 GHz. Each processor has four memory channels with two memory slots in each. One slot in each channel is populated with a 16 GB DDR4 DIMM. Therefore, there are 8 DIMMs in the system, for a total of 128 GB of memory.

In each experiment, the following data is collected on the server machine, as described in Section 2.4:

- measured power, for each DIMM;
- memory controller (RAPL) performance counters, representing per-DIMM power state residency and memory operation rates;
- transactional throughput and latency, measured in DimmStore.

Software Configuration

In all tests, the database is configured with 12 partitions. As dictated by the H-Store architecture, there is exactly one worker thread serving each partition. The worker threads are bound to their own dedicated CPU cores, evenly allocated between the two physical processors. Therefore, six worker threads work on each of the eight-core processors. The remaining cores are reserved as headroom for any demand for CPU cycles by the database threads other than partition workers, and by other system processes.

In both workloads, the Java-based benchmark client bundled with H-Store is used for load generation. Each experiment is run at a predefined target transaction rate, controlled by the benchmark client. The actual transaction rate is also monitored and the runs in which the system cannot keep up with the offered target load are excluded from the results.

Under H-Store's default benchmarking configuration, transactions experience high response times due to internal queueing and batching of requests in both the client and the

server. For example, in the YCSB benchmark at a low load and default settings, the transaction latency as measured at the client was around 10 ms, though it takes only about 50 µs for the server to process the transaction. The long response times are the result of extensive use of queues during many stages of transaction processing in H-Store. In those queues, transactions are accumulated for a period of time, and then are dispatched for processing as a batch. After a batch is processed, the thread goes to sleep for certain amount of time, allowing for the next batch to accumulate. Some sleep times are configurable at run time, but many are hard-coded constants, on both the client and server sides.

These batching delays mask the effect of gating on transaction latency, which we want to measure. Therefore, the degree of batching in the client and server during transaction processing was minimized by making changes to the code and configuration. In particular, the Java classes responsible for load generation (ControlWorker.java), server-side processing (Distributer.java, PartitionExecutor.java) were modified to reduce sleep time constants to 1 ms from existing values of 5 ms or 25 ms. In addition, the parameter *site.txn_incoming_delay* was reduced from 5 ms to 1 ms. The overall effect of these changes was a reduction of the baseline transaction latency from about 10 ms to 1 ms. Although sleep times could be decreased further, doing so would increase the CPU utilization due to the increased frequency of threads polling work queues without blocking. At the same time, that would have a diminishing effect on the latency reduction. Therefore, the baseline latency of 1 ms was chosen as an acceptable trade-off.

Workload configuration

We used the YCSB workload that consists of only Read transactions. Each transaction consists of a single query that reads a tuple from a table by a primary key. The accessed tuples are selected non-uniformly according to a Zipf distribution with a fixed skew parameter $s = 0.95$.

In these experiments, an 80 GB database was used, which occupies approximately two thirds of the available memory in the server. This database size was chosen to simulate a "default" use case, in which much but not all of the server memory is used. The effect of the database size on the power efficiency will be shown in Section 5.4.5.

Memory and Gating Configuration

Experiments with DimmStore use rank-aware memory allocation with identical configuration with and without gating. In DimmStore, 2 DIMMs (32 GB) are used for the System

region, and the remaining 6 DIMMs for the Data region. This configuration maximizes the Data region size, with each processor having the same number of DIMMs in each region.

The database takes 22 GB in the System region for its indexes and hot data, leaving the other 12 GB for non-database memory allocations. The remaining part of the database spills to the Data region, utilizing 4 of its 6 DIMMs. The last 2 DIMMs are empty.

The gating parameters are fixed, with the restricted interval $t_R = 2ms$, total cycle length $t = 8ms$, resulting in a compression factor $k = 1.33$. As it will be shown in Section 5.4.4, these values realize the most savings in power consumption, and increasing them further produces only a marginal improvement.

To evenly split memory load between both memory interfaces, the two DIMMs of the System region belong to different processors. For the same reason, the DIMMs of the Data region are evenly split between processors, utilizing alternating allocation order. Thus, as the database footprint in the Data region increases, memory is first allocated from the DIMM of the first processor. When that DIMM becomes full, memory allocation switches to a DIMM of the second processor, then switches back to the first processor, and so on.

Experiment Configuration

Each experimental run consists of three phases: database population, warm up, and measurement. Measurements collected during the database population and warm up phases are discarded. Even though a main-memory database does not have a disk cache to "warm up", the tuple composition of the System Region constantly changes during the run due to eviction and uneviction activity. Right after the database is populated, the tuple population in the System Region is mostly determined by the database loading order and does not reflect tuple access frequency. Therefore, the first seconds of the workload run see a burst of evictions and unevictions, as the database is adjusting tuple distribution between the System and Data regions according to tuple access rates.

Since the system takes more time to warm up when the transaction rate is lower, the duration of the warm-up and measurement phases is scaled inversely to the relative load. For example, halving the target throughput doubles the length of the warm-up and measurement intervals .

Similarly, the duration of the measurement phase is scaled inversely proportionally to the target transaction rate so that it includes the same number of executed transactions regardless of rate.

Parameter	Values	Default
Database size	60, 80, 100 GB	80 GB
Offered load	37.5 - 300 Ktps	150 Ktps
System region size	32 GB	32 GB
Eviction interval t_{evict}	1 ms	1 ms
Eviction volume N_{evict}	64 KB	64 KB
Uneviction probability $p_{unevict}$	$\frac{1}{64}$	$\frac{1}{64}$
Restricted interval t_R	$2ms$	$2ms$
Compression ratio k	$1.25, 1.33, 2, 3$	1.33

Figure 5.3: YCSB Experiment Parameters

We repeated each experiment 10 times and show the sample mean value. On graphs for total power and latency with default compression factor, we also show 95% confidence intervals CI for the sample mean μ, estimated as:

$$CI = [\mu \pm t_{(0.025,n-1)} \frac{S}{\sqrt{n}}] \tag{5.2}$$

Here, n is the sample size (number of identical experiment runs), $t_{(0.025,n-1)}$ is 97.5-percentile of the Student's distribution with $n-1$ degrees of freedom, and S is the sample standard deviation.

The workload parameters are summarized in Table 5.3.

5.4.2 YCSB Results: Memory Power and Transaction Latency

We begin by evaluating the impact of the gating mechanism on the overall memory power consumption and transaction latency. To do this, we compare the gating prototype to two baselines, DimmStore without gating enabled, and original H-Store.

Figure 5.4 shows total memory power consumption in each system, as a function of load. Each point on the graph represents a single experiment, in which a system under test (H-Store, DimmStore without gating, or DimmStore with gating) was loaded using a fixed transaction rate shown on the X axis. The set of target transaction rates used in these experiments represents eight equally-spaced points, up to the approximate peak throughput of the baseline system (300 ktx/s). In DimmStore with gating, since some of the load during restricted intervals is deferred to unrestricted intervals the peak load

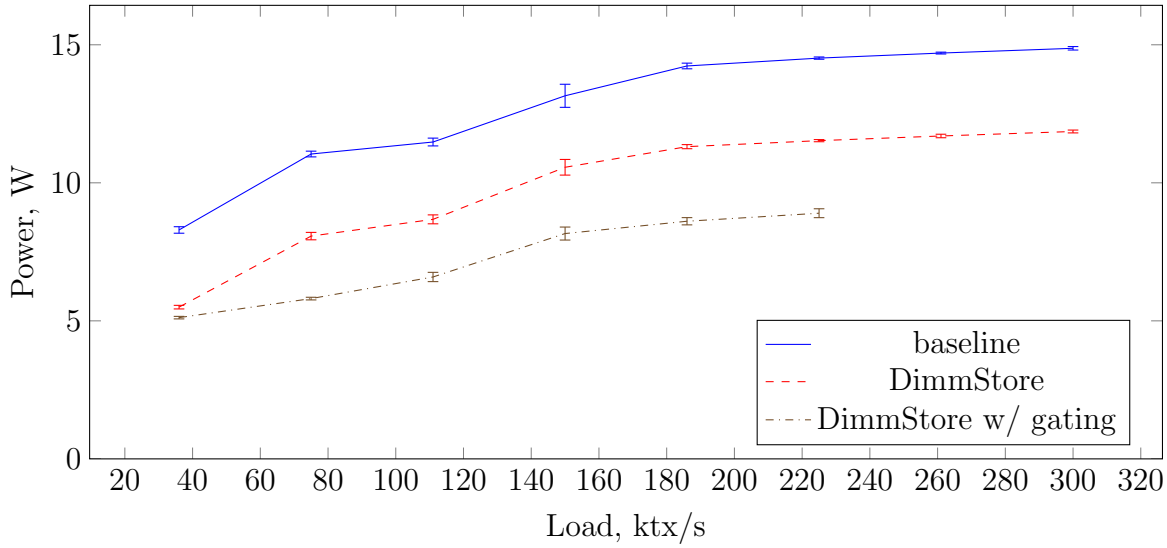

Figure 5.4: YCSB: Total memory power consumption by load, restricted interval duration = 2 ms, compression ratio = 1.33x. Confidence intervals are computed as in (5.2).

is limited to approximately 75% of the peak load without gating. Therefore, DimmStore with gating was tested only up to 225 ktx/s.

Enabling memory access gating in DimmStore reduces memory power consumption by between 9% and 22%, depending on load, relative to DimmStore without gating. Overall, this reduction translates to 37% - 48% power savings over baseline H-Store. The relative power saving is higher at medium and high loads but diminishes at very low load. We will discuss how memory power consumption depends on load in Section 5.4.3.

Transaction latency, measured at the client, is shown in Figure 5.5. Gating results in a latency increase between 0.2 and 0.4 ms, relative to both H-Store and DimmStore without gating. This increase occurs for two reasons: forced blocking of processing threads during the restricted interval, which adds a constant amount of delay independently of load, and the increase in the execution time of each transaction as the transaction rate increases during the unrestricted interval.

5.4.3 Detailed Memory Power Analysis

In this section, we explain how memory gating saves memory power. Recall that memory gating specifically targets power consumption in the used DIMMs of the Data region. To

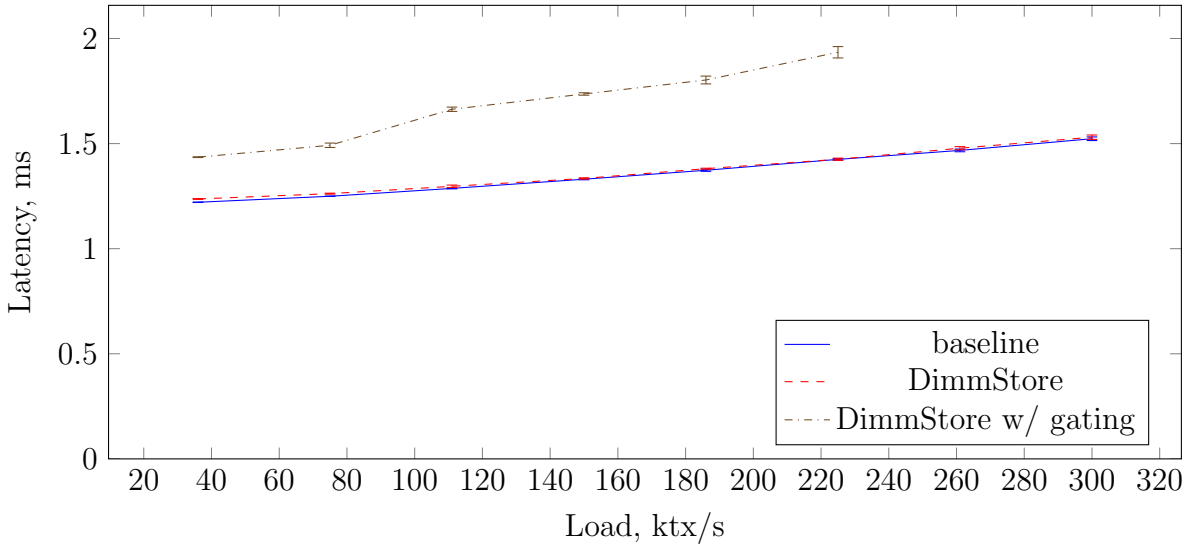

Figure 5.5: YCSB: Transaction latency by load, restricted duration = 2 ms, Compression Ratio = 1.33x. Confidence intervals are computed as in (5.2).

demonstrate that gating works as expected and reduces power consumption in the Data region, we start our analysis by breaking down power consumption by memory region. Later, we show that power savings in the Data region are due to increased low power state utilization. Finally, we show how much of available power saving opportunity is realized by gating in DimmStore.

Memory Power by Region

In DimmStore, there are three groups of DIMMs, based on their relative utilization. Frequently accessed data is concentrated in the System region, causing its DIMMs to handle most memory activity. We call these the "hot" DIMMs. Less frequently accessed database tuples occupy space in Data region DIMMs, these ones are "warm" DIMMs. Depending on the number of DIMMs in the Data region and the database size, some DIMMs in the Data region may stay unutilized. We call these the "cold" DIMMs.

The contribution of each group of DIMMs to memory power consumption as a function of load, in DimmStore with and without gating, is shown in Figure 5.6. We do not include the baseline H-Store system because it does not have the concept of memory regions.

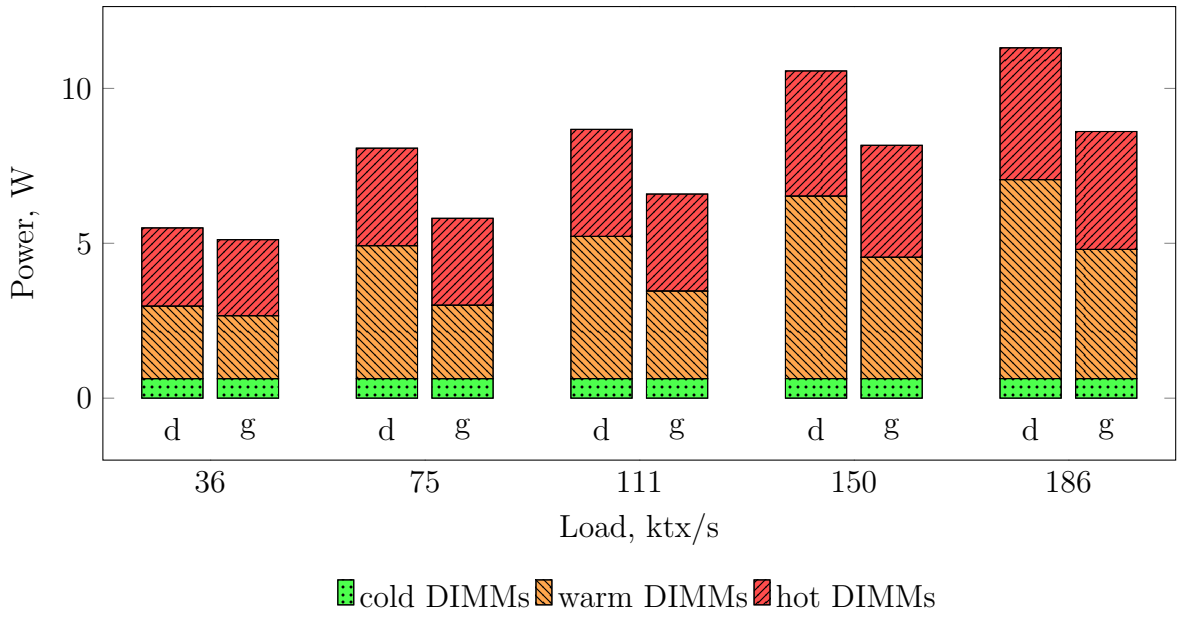

Figure 5.6: YCSB: Total memory power consumption vs load, broken down by DIMM group, restricted interval duration = 2 ms, compression ratio = 1.33x. Column key: "d" - DimmStore, "g" - DimmStore with gating

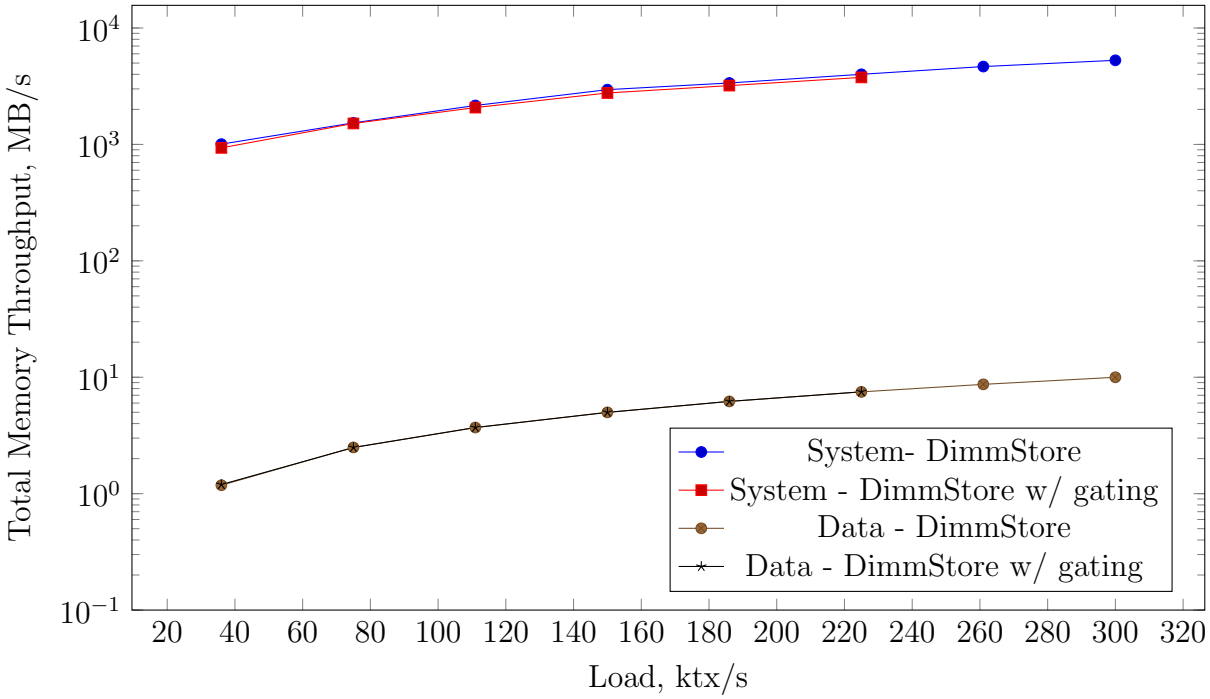

Figure 5.7: YCSB: Total (read and write) memory throughput in the DIMMs of the System Region, by load, restricted interval duration = 2 ms, compression ratio = 1.33x

One observation from Figure 5.6 is that the contribution of the warm DIMMs to total power consumption is significant. In fact, in this system configuration and workload, warm DIMMs are responsible for most of the memory power. This is true despite the fact that warm DIMMs handle much less memory traffic than the hot (System region) DIMMs. The difference between memory traffic between regions in DimmStore (similar with or without gating) is several orders of magnitude, approximately 5 GB/s in the System region vs 10 MB/s in the Data region, at maximum load. For reference, memory throughput in DimmStore with and without gating, as a function of transaction rate, is shown in Figure 5.7, separately for the System region and Data regions.

We can also see from Figure 5.6 that gating reduces memory power in warm DIMMs only. This is the expected result as gating only targets the Data region.

Power measurements shown in Figure 5.6 represent sums of power consumption in all DIMMs in each group (hot, warm, and cold). These sums depend on the region configuration and database size, i.e. the number of DIMMs in each region and the number of used DIMMs in the Data region. To make power consumption easier to interpret and compa-

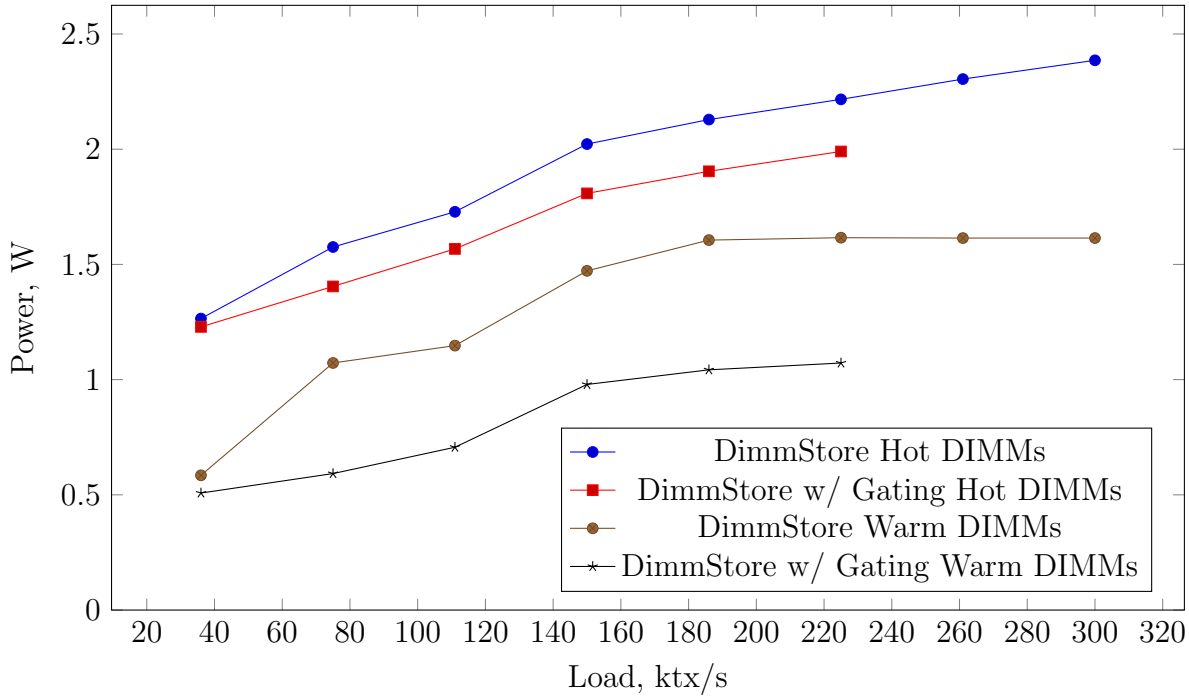

Figure 5.8: YCSB: Average per-DIMM power consumption in the DIMMS of the System region ("hot") and used DIMMs of the Data region ("warm"), by load, restricted interval duration = 2 ms, compression ratio = 1.33x

rable between regions of different sizes, we normalize them by the number of DIMMs in each group and show the result in Figure 5.8. Memory gating reduces average power consumption in warm DIMMs by between 15% and 43%. The highest relative saving occurs at medium load.

Power State Use by Region

The power savings in the warm DIMMs are due to an increased use of low power states, resulting in lower background power consumption. The power state residency for warm Data and System region DIMMs is shown in Figure 5.9. Gating substantially increases Self Refresh utilization in warm DIMMs under all loads except the lowest. At the lowest load, the memory is accessed infrequently enough that the Self Refresh state can be used even without gating.

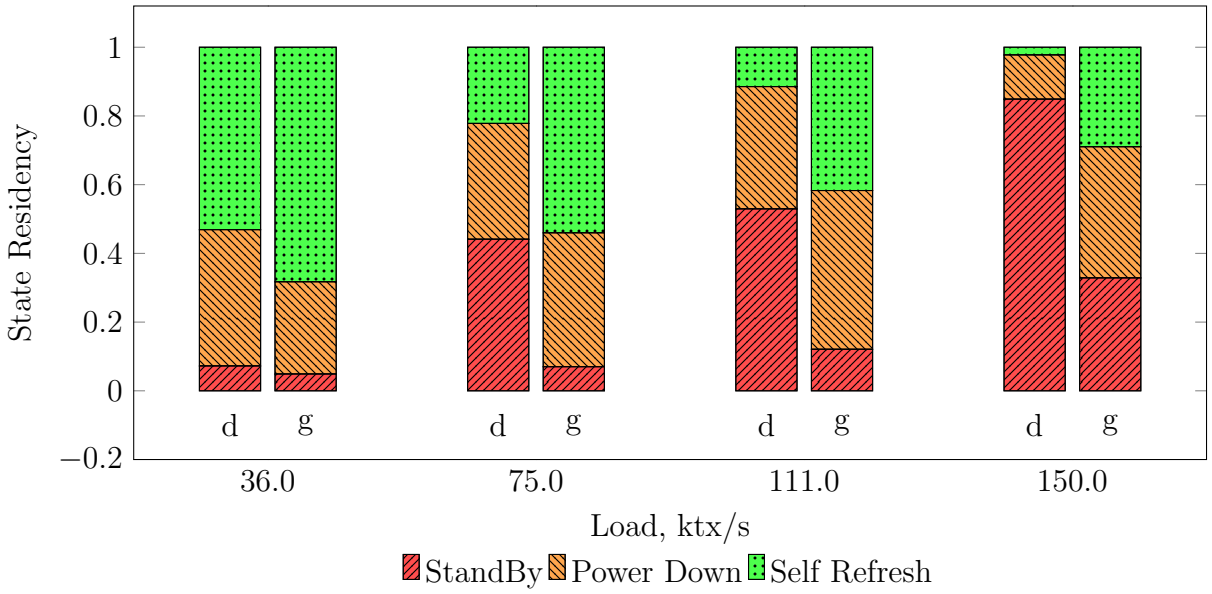

Figure 5.9: YCSB: Average power state residency in warm DIMMs, restricted interval duration = 2 ms, compression ratio = 1.33x. Column key: "d" - DimmStore, "g" - DimmStore with gating

Although power reductions in the System region are not a goal of the gating mechanism, Figure 5.8 also shows that memory power consumption in the System region also decreases, but by a smaller amount, between 4% and 14%. The reduction of power consumption in the System region is a unintended result of the restricted interval implementation, which causes worker threads to block for a portion of the restricted interval.

Memory Power by Type

To put the effectiveness of power saving techniques in DimmStore in perspective, and to see how much of power saving opportunity is realized, we now show memory power consumption *by power component*. Figure 5.10. shows memory power components, found using the memory model from Section 2.5, and aggregated across all DIMMs in the server. The background power components other than Refresh power are lumped together as *Variable Background power (VBP)*. For DimmStore, VBP is further broken down for the System and Data regions. The memory gating in DimmStore directly targets only the VBP in the Data region. Therefore, the relative amount of reduction in the VBP in the Data region is an indication of the actual gating effectiveness, or how much of the saving potential is realized in a particular workload. As shown in Figure 5.10, actual reduction in the VBP in the Data Region due to gating is between 32% (low load) and 61% (medium load).

The remaining power components are not targeted by the gating mechanism and are not expected to significantly change. The Refresh power component is the largest of these but it is intrinsic to operating a DIMM while retaining data. The Variable background Power component in the System region is also significant, but this region handles almost all database memory activity, as well as those not managed by the database system. Therefore, significantly reducing power consumption in the System region is outside of scope of this work.

CPU Power

CPU power consumption, reported by RAPL power monitoring counters, is shown in Figure 5.11. CPU power consumption is reduced compared to the ungated DimmStore by between 26% and 5%. The observed reduction CPU power consumption is unintended and was not studied further. However, a possible explanation of this phenomenon is switching of the processor cores to idle states (C-states) during the restricted intervals as worker threads, which are bound to cores, become blocked.

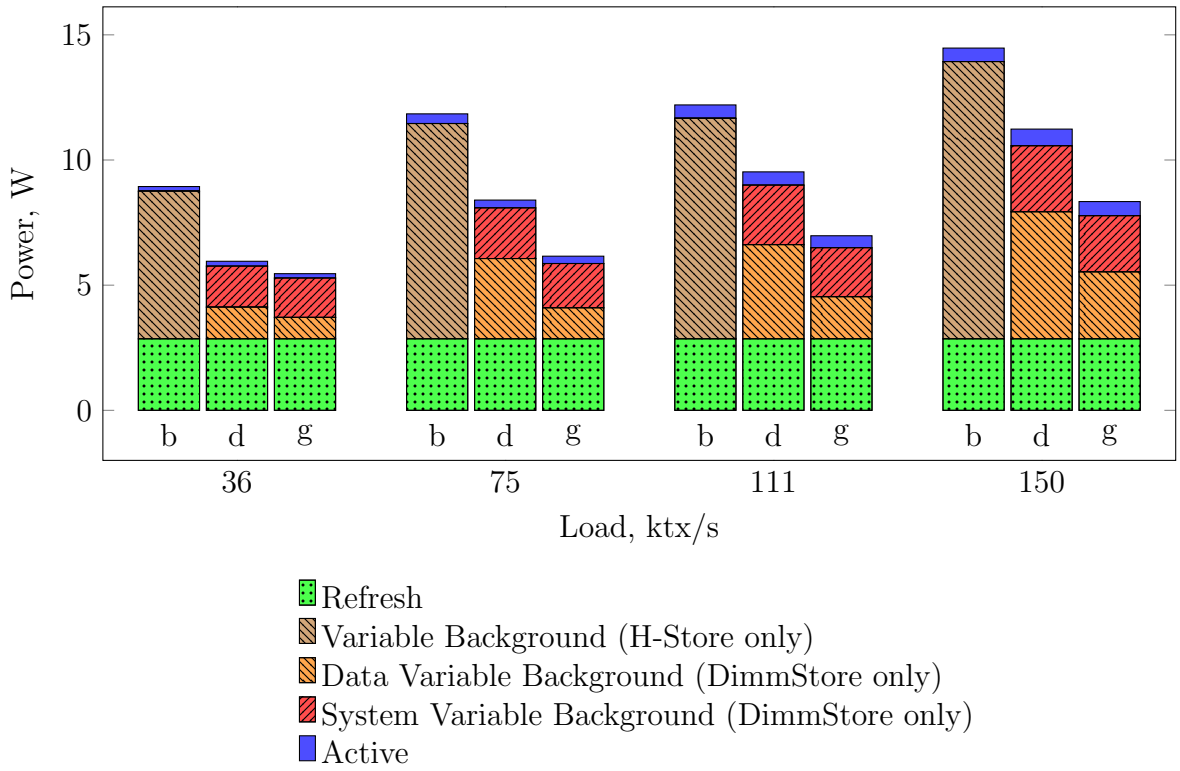

Figure 5.10: YCSB: Total memory power consumption by load broken down by type and region, restricted interval duration = 2 ms, compression ratio = 1.33x. Column key: "b" - baseline H-Store, "d" - DimmStore, "g" - DimmStore with gating

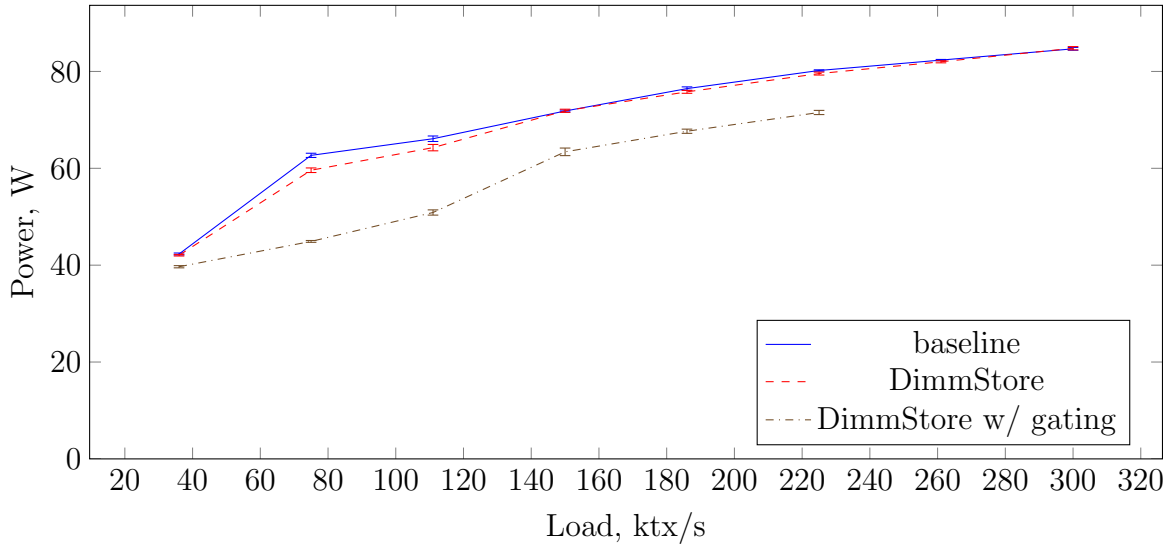

Figure 5.11: YCSB: Total CPU power consumption by load, restricted interval duration = 2 ms, compression ratio = 1.33x. Confidence intervals are computed as in (5.2).

5.4.4 YCSB: Effects of Gating Parameters

In this section, we explore how gating configuration affects power consumption and transaction latency. In DimmStore, gating is controlled by two configuration parameters, the length of the restricted interval t_R and the total length of the gating cycle t. Recall, that we defined the compression factor $k = \frac{1}{1-\frac{t_R}{t}}$ (Section 5.3.1).

The length of the restricted interval, t_R, sets the minimum amount of idle time in each gating period that is available for using low power states, such as Self Refresh. Generally, each DIMM is expected to spend most of the restricted interval in a low power state. This amount of low-power state per gating period is load-independent. Idle time *outside* of the restricted interval may add more low power state utilization, and this addition will be higher with lower load. Since the compression factor k determines the the memory load increase during the unrestricted interval, we tested the effects of the gating parameters by varying t_R and k.

To understand the impact of these parameters on memory power and transaction latency, we ran experiments with different parameter values. We considered all combinations of 20 values for the restricted interval, in the range from 1 to 6 ms, and 8 values of compression factor, in the range from 1.25x to 3x. Therefore, the parameter space represents

a rectangular grid in which every combination of the parameter values corresponds to a separate test run.

First, we describe the *combined* effect of both parameters by showing the general shape of the trade off between memory power and system performance due to changes in the parameters. In other words, can power consumption be further reduced by tuning the parameters, and will the transaction latency necessarily increase by doing so?

This trade off can be illustrated using scatterplots in the latency/power coordinate system. Figures 5.12, 5.13, and 5.14 show the scatterplots for low, medium, and high loads, respectively. Each point in a plot corresponds to one experiment with a distinct combination of gating parameters. In addition to points obtained from experiments with gating, each plot includes a point from baseline H-Store (marked as a solid square), and from DimmStore without gating (marked as a solid circle).

In the lowest load setting (Figure 5.12), gating parameters do not substantially reduce memory power consumption relative to DimmStore, regardless of parameter settings. The maximum power reduction from any gating configuration is less than 10%. However, depending on the parameter values, transaction latency may almost double. At medium and high loads (Figures 5.13, 5.14), the gating parameters control the tradeoff between memory power consumption and transaction latency. However, the incremental power reduction becomes smaller once the latencies increase and stops improving as the latency reaches about 2.5 ms in the case of medium load. At higher load, the trade off between power consumption and additional latency becomes less clear and and occurs in the region of longer latencies.

Given that the gating parameters affect power savings and transaction latency, we will discuss how to tune them. First, we consider the restricted interval length. Figures 5.15 to 5.17 show memory power consumption and transaction latency as a function of restricted interval length (t_R), for several values of the compression factor (k). Increasing restricted interval length reduces power consumption when the interval length is less than 2 - 3 ms, and increasing it further provides little additional power savings. The reason for this diminishing return is that the longer the restricted interval is, the smaller fraction of it is not used for the Self Refresh state due to a constant Self Refresh timeout. Since transaction latency continues to increase with higher restricted intervals values, there is no practical advantage in setting the interval length above 3 ms.

To study the effect of the compression factor k, we fix the restricted interval at 2 ms and vary the compression factor at various loads. The power consumption and transaction latency, as functions of load, for different compression factors, are shown in Figures 5.18 and 5.19, respectively. In contrast to restricted interval length, the compression factor

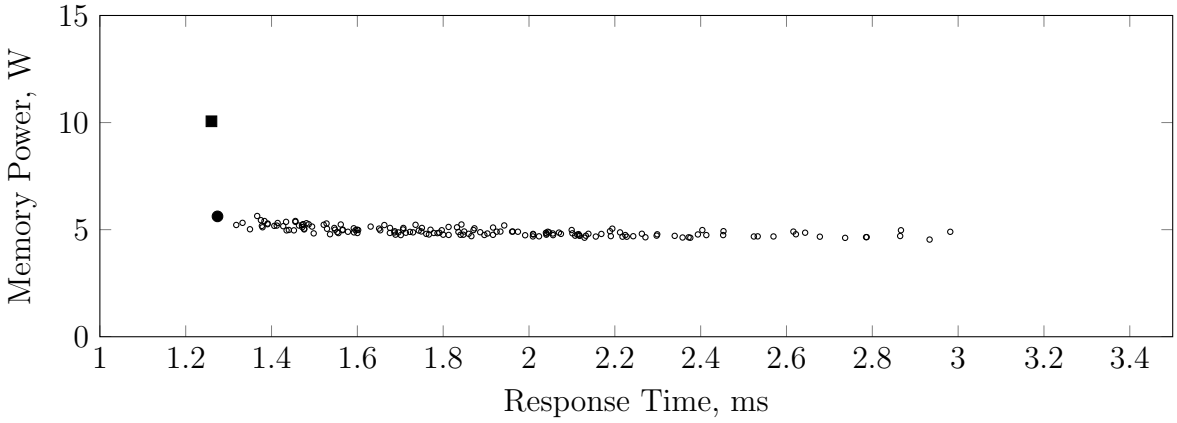

Figure 5.12: YCSB: Memory power versus response times, low load.

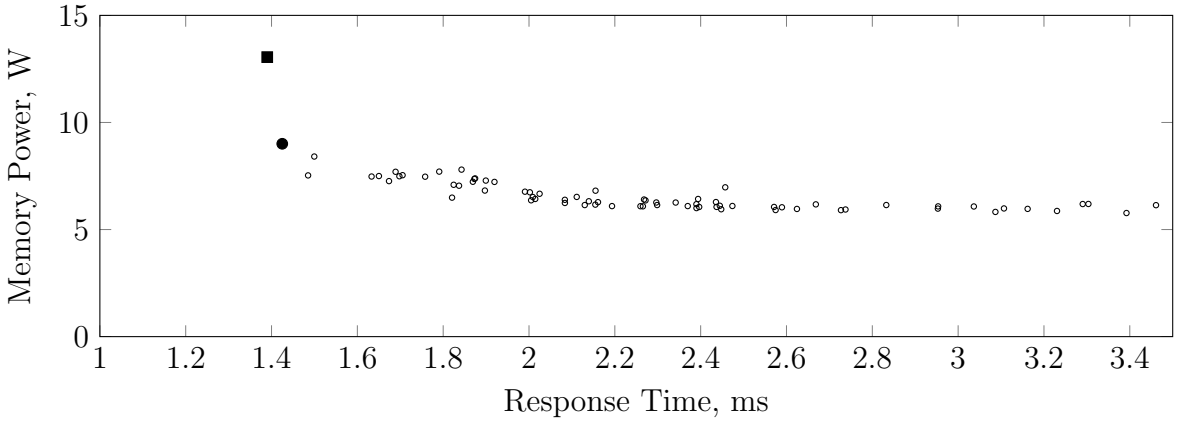

Figure 5.13: YCSB: Memory power versus response times, medium load.

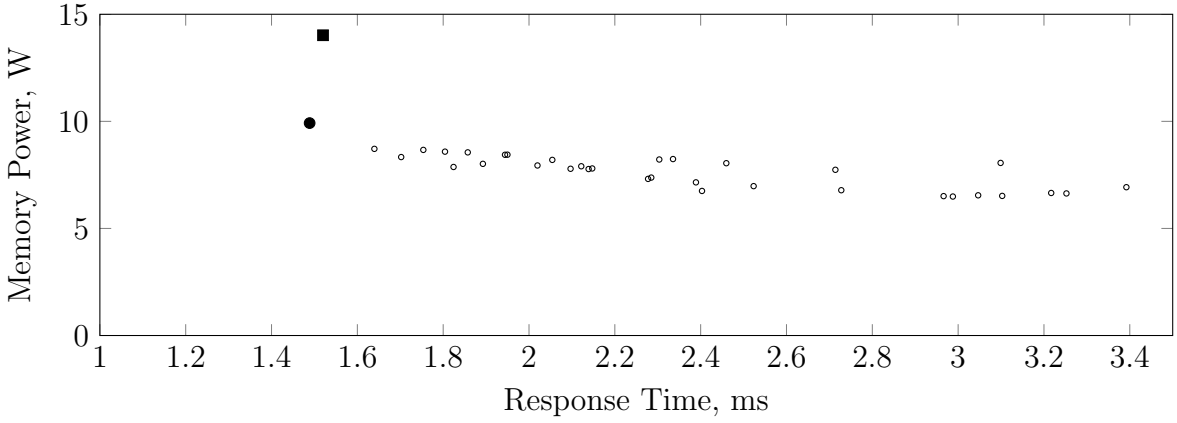

Figure 5.14: YCSB: Memory power versus response times, high load.

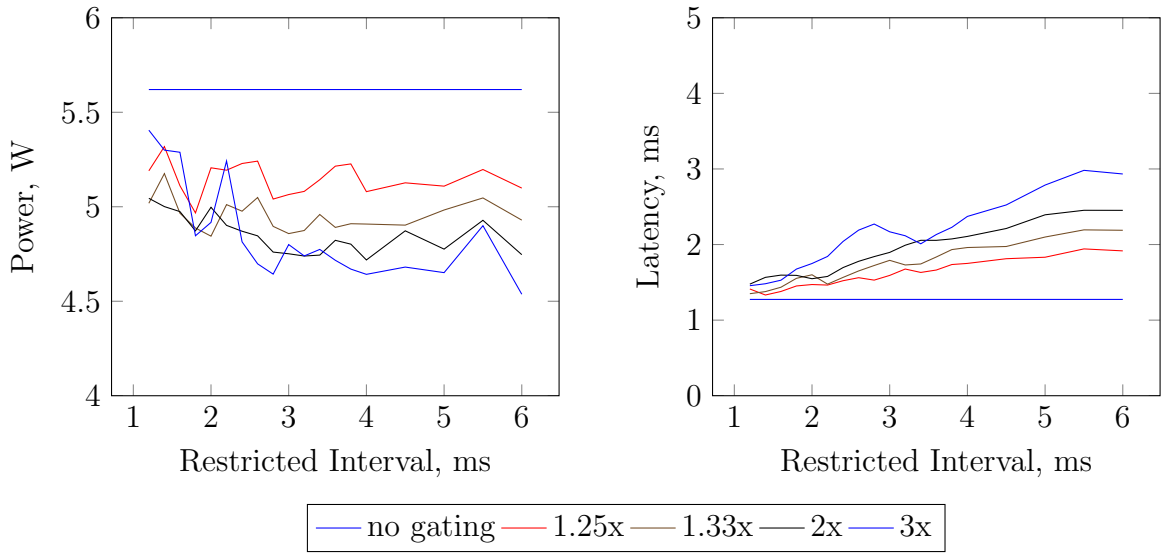

Figure 5.15: YCSB: Memory power consumption and transaction latency as functions of restricted interval length, for different compression factors, low load (25%).

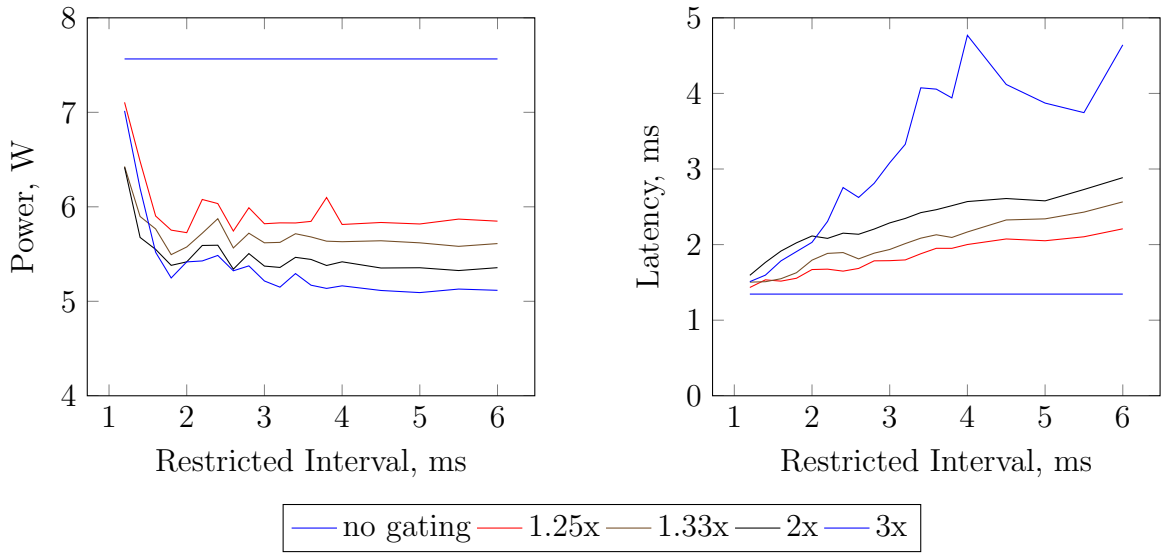

Figure 5.16: YCSB: Memory power consumption and transaction latency as functions of restricted interval length, for different compression factors, medium load (50%).

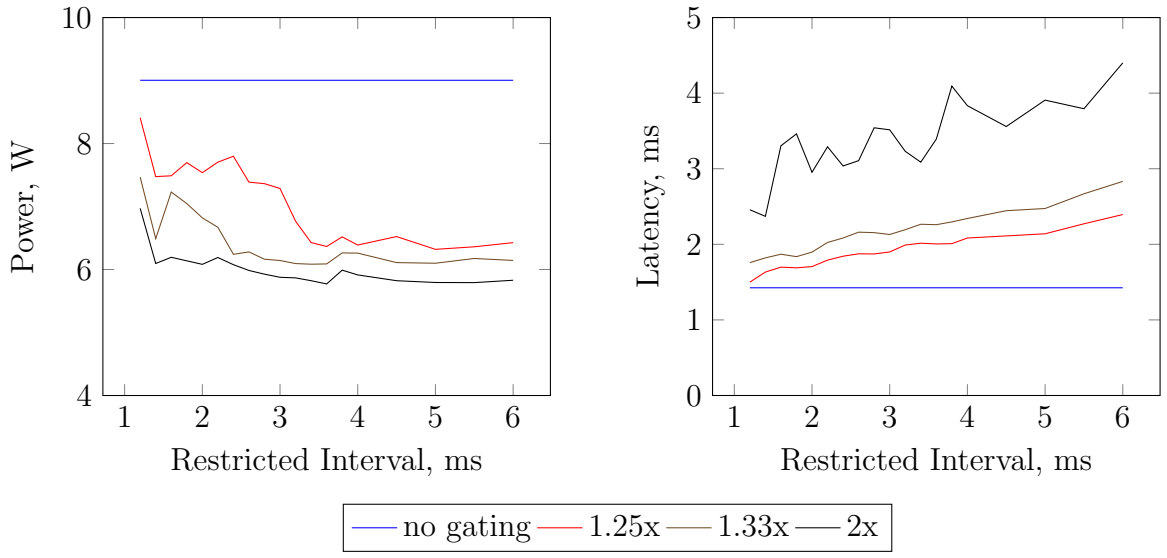

Figure 5.17: YCSB: Memory power consumption and transaction latency as functions of restricted interval length, for different compression factors, high load (75%).

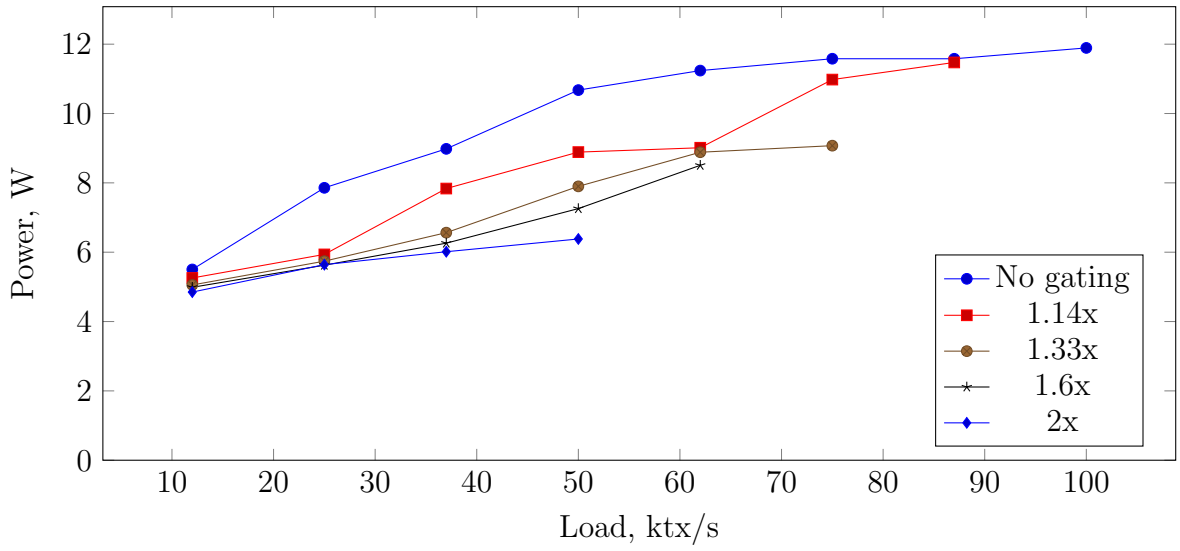

Figure 5.18: YCSB: memory power consumption by load and compression factor k, restricted interval duration = 2 ms

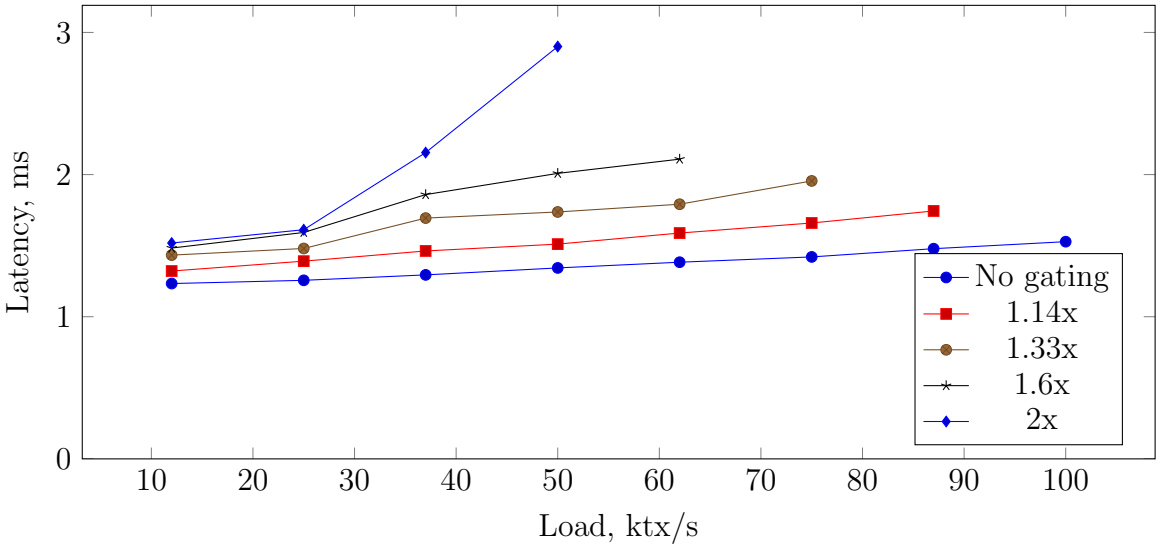

Figure 5.19: YCSB: Latency by load and compression factor k, restricted interval duration = 2 ms

affects power consumption over its full permissible range. Similarly, higher values of the compression factor consistently increase transaction latency. Therefore, it is not possible to give a single recommendation for setting of the compression factor. For power sensitive application, it should be set to the maximum possible value, depending on load. For applications that impose quality of service requirements, the value of the compression factor can be set to a maximum value for a permissible latency penalty.

5.4.5 Effects of Database Size(YCSB)

In this section, we will vary the size of the database to understand how memory power consumption is affected by these changes. We consider 60 GB and 100 GB database sizes, in addition to the default 80 GB size. The 100 GB database was too big for the 2 DIMM System region because its indexes and hot data did not fit into the 22 GB section of the System region reserved for the database. Therefore, we increased the System region to 4 DIMMs and repeated the experiment for the smaller database sizes using the 4 DIMMs System region.

The 4 DIMM System region configuration imposes a 44 GB limit on the hot data and indexes, which is twice the limit imposed by the 2 DIMM configuration. In both configu-

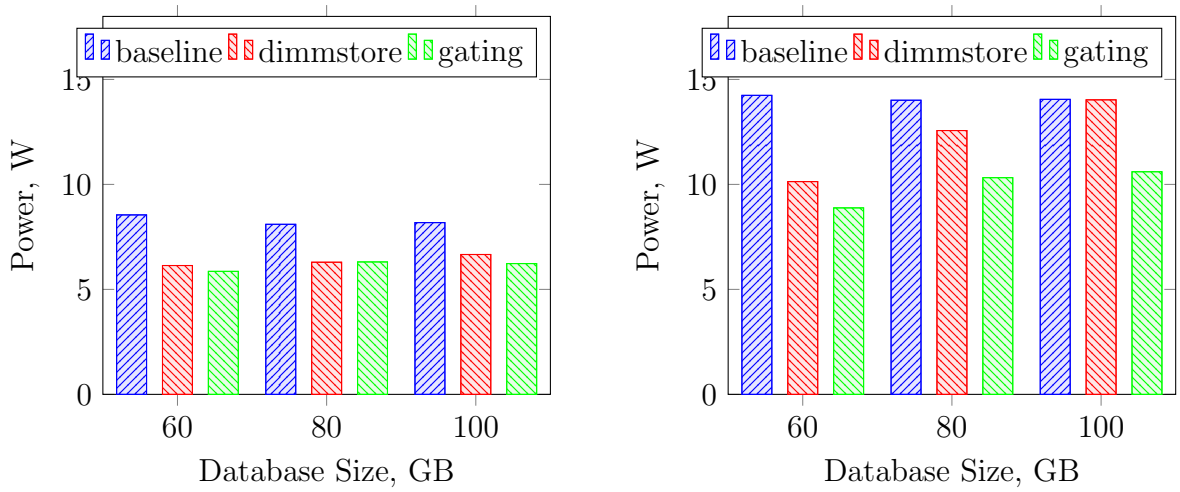

Figure 5.20: YCSB: Total memory power consumption by load, three database sizes, 2 DIMM System region, gating configuration with restricted interval duration = 2 ms, compression ratio = 1.33x. Left: low load (approx. 40 ktx/s), right: medium load (approx 180 ktx/s).

rations, the System and Data regions are symmetrically split between the two processors in the server. The other experimental settings are identical to the ones in the previous YCSB experiments.

Total memory power consumption for various database sizes is shown in Figure 5.20, for low and medium transaction rate settings. At low load, gating does not reduce memory power consumption over DimmStore, for all database sizes. This extends the results obtained in Section 5.4.2) for the medium database size.

At a medium load, memory power savings due to gating are larger for larger databases. DimmStore's power consumption increases with larger database sizes and becomes as high as in the baseline when the database uses all the DIMMs in the Data region. With gating, increasing the database size leads to smaller increases in memory power consumption than in DimmStore. This is because gating improves the utilization of low power states in the used DIMMs of the Data region.

Parameter	Values	Default
Database scale factor	450 warehouses	450 warehouses
Database size	45 GB	45 GB
Offered load	7.5 - 60 Ktps	30 Ktps
System region size	48 GB	48 GB
Eviction interval t_{evict}	1 ms	1 ms
Eviction volume N_{evict}	64 KB	64 KB
Uneviction probability $p_{unevict}$	$\frac{1}{64}$	$\frac{1}{64}$
Restricted interval t_R	$2ms$	$2ms$
Compression ratio k	$1.33, 2$	1.33

Figure 5.21: TPC-C Experiment Parameters

5.5 Impact of Gating: TPC-C

In this section, we will show the effects of memory gating on memory power consumption and transaction latency in a different workload, TPC-C. The TPC-C benchmark represents a more complex workload, simulating a transactional application. The benchmark consists of 5 transaction types and 9 tables, featuring a variety of access patterns to tuples in different tables and transactions.

The default System region configuration is is the same as in YCSB experiments, with the System region consisting of 2 DIMMs, one DIMM in each CPU socket, with a total of 32 GB. The remaining six DIMMs, 96 GB, makes up the Data region.

We used a default database scale factor of 450 warehouses, resulting the database size of 62 GB, including the indexes. The system uses 22 GB of the System region for hot data and indexes. The remaining data, approximately 40 GB, is in the Data region, where it occupies three of the six available DIMMs. The remaining 3 DIMMs of the Data region are unused (cold). In the TPC-C workload, the database grows during the run, so the used part of the Data region expands. The amount of database expansion is approximately 3.5 GB total over the measurement interval. As with YCSB experiments, each experiment was repeated 10 times and the mean value is shown in figures. The parameters of the experiments are summarized in Figure 5.21.

The design of the TPC-C experiments is similar to the YCSB design, as described in Section 5.4.1. In each experiment, the database system (baseline H-Store, DimmStore, or DimmStore with gating) executes transactions generated from a separate client machine, at a fixed rate. The system is allowed to warm up for 5 minutes and the next 5 minutes is

the measurement interval, for the nominal 60 ktps load. For lower than nominal loads, the warm-up and measurement intervals are proportionally stretched so that the same amount of work is performed by the DBMS during each interval.

During the measurement interval, we measure power consumed by each DIMM in the server, record transaction latency data from the client generator, and collect DRAM RAPL performance counters representing power state residencies. For presentation, the measured quantities are averaged over the measurement interval.

The workload was run at several load settings, each representing a fraction of the nominal maximum transaction rate, set at 60 ktps. The nominal maximum transaction rate corresponds to the maximum load the baseline system could sustain. However, the maximum sustainable load was lower in DimmStore, therefore, the 100% (60 ktps) load points are not included in the DimmStore results.

In the experiments with gating, the length of the restricted interval is 2 ms, and two settings for the compression ratio are used: 1.33x and 2.0x.

TPC-C: Memory Power

Total memory power consumption for the three systems, with two settings of gating compression ratio (k) is shown in Figure 5.22. Enabling gating with the compression factor of 1.33 reduces the power consumption by 5-15% compared to DimmStore without gating, with the higher power savings for higher load. Further increasing the compression factor to 2.0 reduces the power consumption by approximately the same amount. The additional power savings due to gating are lower in absolute terms compared to those achieved by DimmStore over the baseline, but still significant, considering they occur on top of the reduction achieved by DimmStore.

Memory power reduction is mainly caused by the increased utilization of the Self Refresh state in the used DIMMs of the Data region, i.e., the warm DIMMS, as it was the case for YCSB. The power state residency versus load in warm DIMMs for two gating settings (2 ms restricted interval, 1.33x and 2x compression) is shown in Figure 5.23.

With gating, the Self Refresh state utilization does not depend on load and its residency is close to the ratio of the restricted interval to the full cycle length $1 - \frac{1}{k}$, where k is compression ratio. For reference, this ratio is $\frac{1}{4}$ for the 1.33x compression, and $\frac{1}{2}$ for the 2x compression. Therefore, Self Refresh state must be in effect during much of the restricted interval but not used during the unrestricted interval. This is in contrast with the YCSB workload (Section 5.4.3) where Self Refresh residency substantially increased at lower loads.

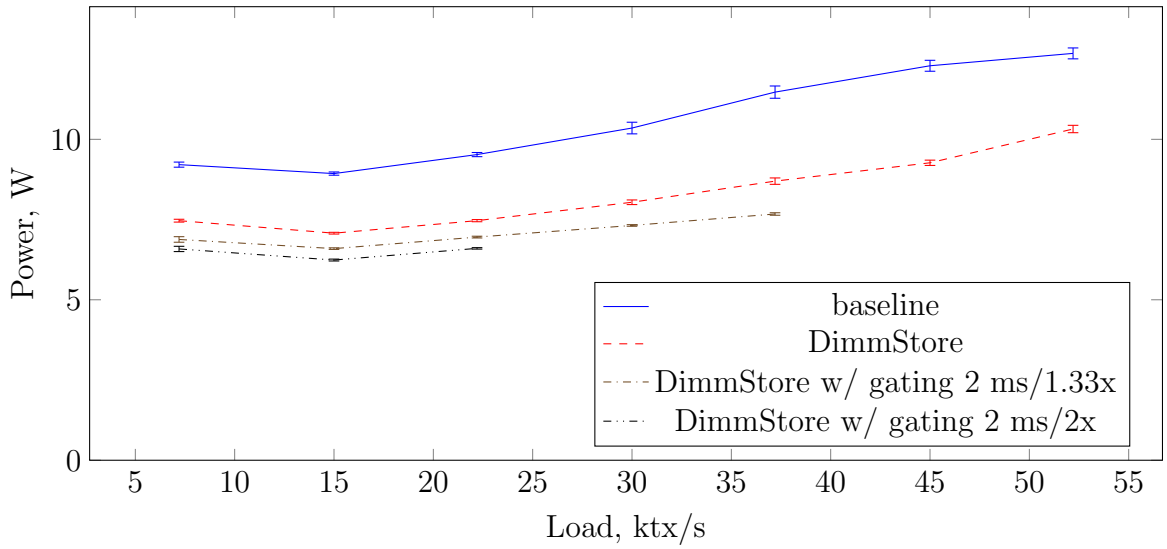

Figure 5.22: TPC-C: Memory power consumption in baseline, DimmStore without gating, and with various gating parameters, versus load. Confidence intervals are computed as in (5.2).

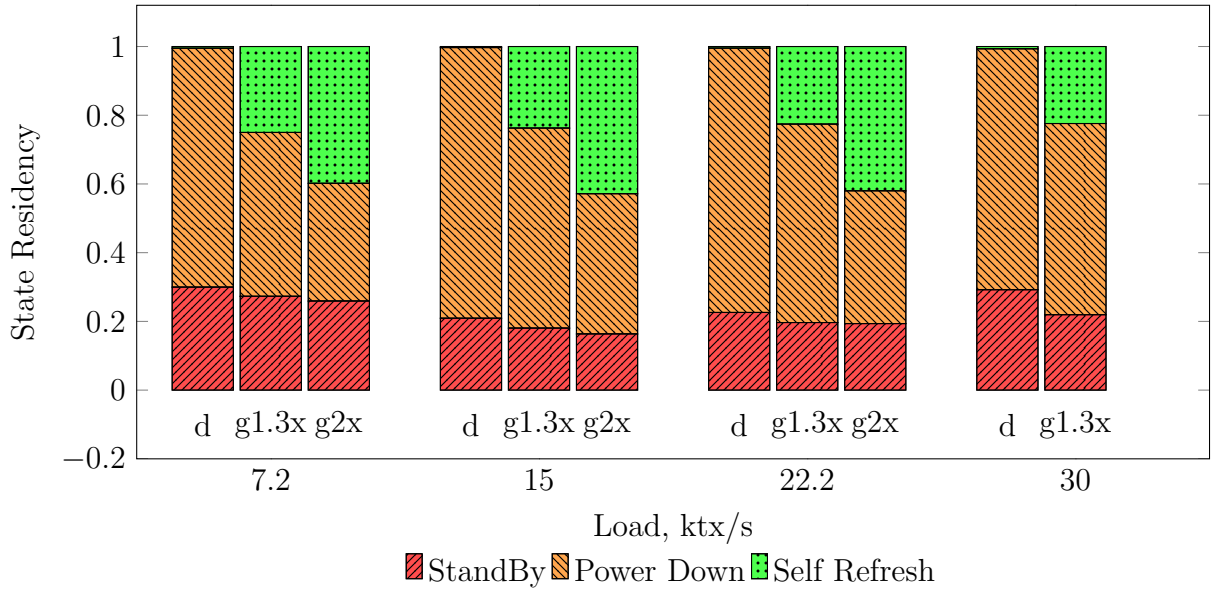

Figure 5.23: TPC-C: Average power state residency in warm DIMMs, restricted interval duration = 2 ms, compression ratio = 1.33x and 2x. Column key: "d" - DimmStore, "g1.3x" - DimmStore with gating $k = 1.33$, "g2x" - DimmStore with gating $k = 2$

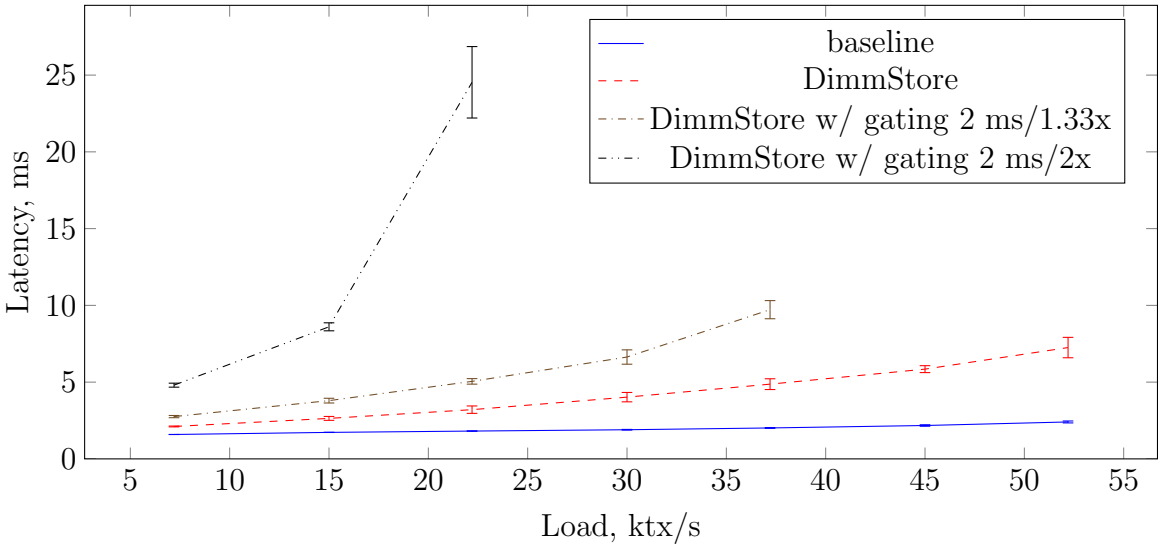

Figure 5.24: TPC-C: Transaction latency in baseline, DimmStore without gating, and with various gating parameters, versus load. Confidence intervals are computed as in (5.2).

Even though gating increases the Self Refresh state utilization, this happens at the expense of the Power Down state. This means that the natural idle time that exists between memory accesses without gating can make use of the Power Down state. With gating, these short idle intervals are coalesced together during the restricted interval, but the total usage of all lower power states is not substantially increased. These two factors (no increase in the Self Refresh residency at lower loads, the Self Refresh state replacing existing Power down state) explain the lower power savings in a more memory-intensive TPC-C workload, compared to YCSB.

TPC-C: Transaction Latency

While memory gating reduces TPC-C memory power consumption, it also introduces a significant latency penalty. The transaction latencies are shown in Figure 5.24. The increase in latency is modest (0.5 - 1.5 ms, 30-60%) for the lower compression factor setting (1.33x) as long as the load is low (<50%). However, even with the lower compression factor, the additional latency grows to 6 ms (150%) for higher utilization. With the 2x compression, transaction latencies rise rapidly with increasing load.

With gating, some increase in latency is expected because the system capacity is reduced. In the worst case, when no transactions can execute during the restricted interval,

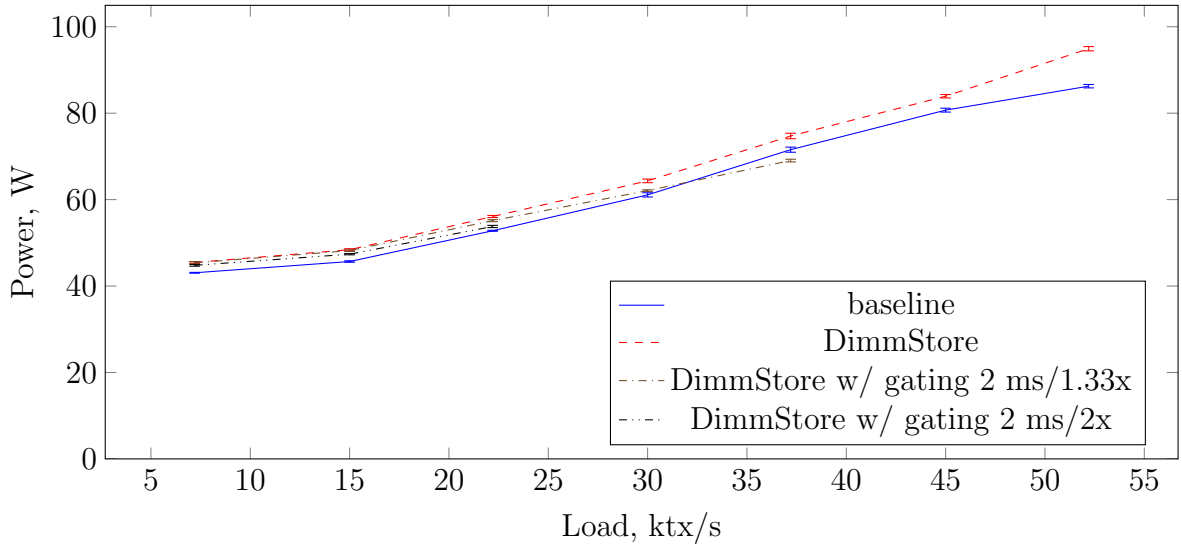

Figure 5.25: TPC-C: CPU power consumption in baseline, DimmStore without gating, and with various gating parameters, versus load. Confidence intervals are computed as in (5.2).

system utilization during the unrestricted interval is increased by the same value as the compression factor. For example, system utilization of 50% with compression factor of 1.33 becomes 66% during the unrestricted interval, if no work can be performed during the restricted interval.

TPC-C: CPU Power Consumption

Estimated total CPU power consumption, obtained from RAPL, is shown in Figure 5.25. Compared to DimmStore without gating, CPU power consumption stays the same for low loads and decreases by approximately 8% with higher loads. As CPU power consumption in DimmStore is higher than in the baseline by 4-5%, gating compensates for this increase in low loads and additionally saves a small amount in a higher load.

5.6 Discussion

In this chapter, we described the memory access gating mechanism, which was implemented in DimmStore. Memory access gating saves additional power in the used DIMMs of the

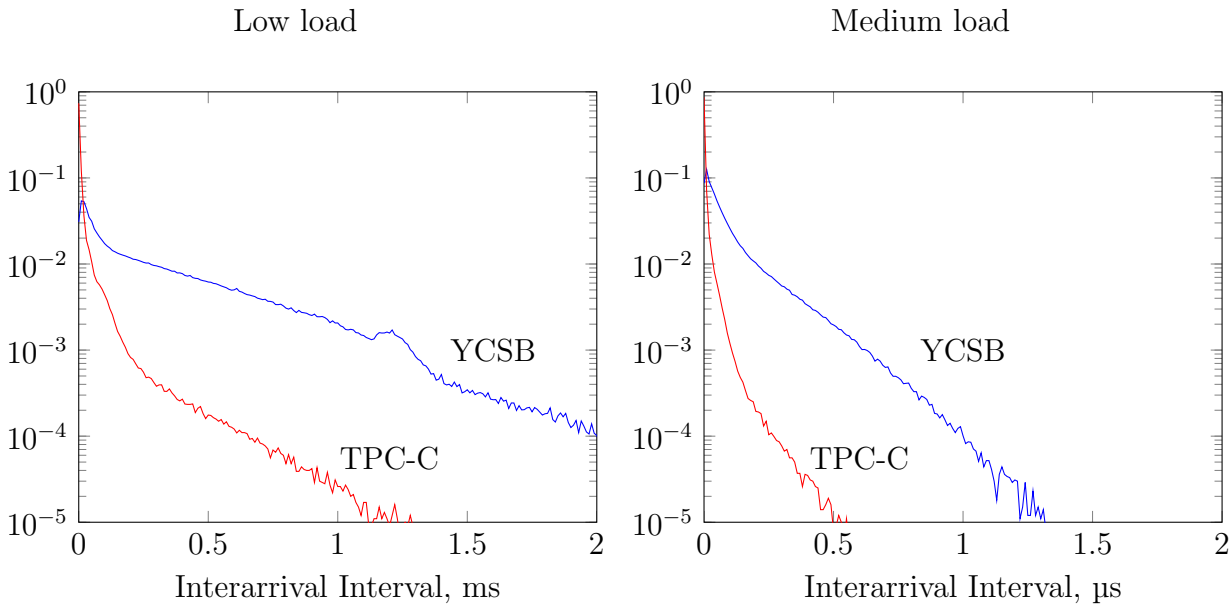

Figure 5.26: Inter-access interval length distribution in a used DIMM of the Data region, without gating (pdf)

Data region, compared to DimmStore without gating. Memory power savings occur in most of the load range, except very low loads, and are approximately 20% in the YCSB workload and between 5% and 10% in the TPC-C workload in typical settings.

Memory gating imposes a performance cost. Temporary blocking of worker threads adds to transaction latency and increases system utilization when the threads are not blocked. The observed increase in transaction response times over DimmStore without gating was 20%-30% in YCSB and 30%-100% in TPC-C.

The difference in power savings and performance impact between the two workloads are due to their different memory access characteristics. Due to more complex and non-regular access patterns, caching of frequently used tuples in the System region is less effective in TPC-C. As a result, the average access rate to DIMMs of the Data region is higher and the average interval between the accesses is shorter. We collected a trace of memory accesses to one of the DIMMs in the Data region and its probability density function is shown in Figure 5.26.

Increasing the degree of access gating by using higher values for the compression factor and the length of the restricted interval leads to a higher performance impact. In other words, there is a tradeoff between power savings and performance. We did not develop an automated mechanism to navigate this tradeoff to find an optimal gating configuration according to the application requirements. This is left for future work.

The access gating prototype, as implemented in DimmStore, has several limitation that reduce its power efficiency and worsen the performance impact. These limitations are due to the underlying H-Store architecture, which does not allow for reordering of transaction processing. Accessing a DIMM in the Data region during the restricted intervals forces the partion's worker thread to delay processing other transactions, even those that would not access the Data region.

If the gating is implemented in a database system that supports out-of-order transaction processing, restricted intervals could be imposed in a *rolling* fashion. In that case, restricted intervals are forced only on a subset of DIMMs at a time. This technique would reduce the average amount of worker thread blocking for the same total number of restricted intervals. We believe that the rolling restricted intervals technique could further improve power efficiency with a lower performance impact.

5.7 Related Work

Memory gating mechanism in DimmStore aims to further increase memory low power state utilization by implementing restricted intervals when access to a memory resource is forbidden. The concept of low power states is applicable to system components other than memory. For processors, the lower power states are usually referred to as C-states. A line of related research targets maximization of the processor C-states utilization by periodically putting processors to sleep. In [33], a *request batching* technique is used in a Web server. Incoming network requests are accumulated by the network adapter for a certain duration of time, called *batching timeout*. The duration of the batching timeout is dynamically adjusted based on the resulting quality of service level. It was found that the request batching policy provides greater energy savings for light workloads, compared to a policy based on Dynamic Voltage Scaling (DVS). This work targets a similar type of workload (web requests) as DimmStore (database transactions) and uses a request batching technique, which is similar to memory access gating in DimmStore. However, the system is assumed to consist only of a single element (processor) and request batching affects all requests handled by the system. In contrast, DimmStore defines and maintains two memory regions such that their access rates differ greatly, and applies an access gating

technique to the region with the lower rate. As a side note, it is interesting that the response time target used for evaluation in that study is 50 ms (90th %-ile), which is an order of magnitude longer than typical response times with DimmStore and gating.

A technique to reduce the power cost of C-state transitions due to clock interrupts in virtual machines is used in the IdlePower architecture [11]. The technique targets virtual machines in an idle state and adjusts the amount of interrupt delay is heuristically adjusted based on the machine's utilization history.

DreamWeaver [59] extends the idea of forced sleep times for multi-core systems. The reason that request batching performs poorly on multiprocessors with parallel request execution is the variance in the request processing times. For processors implementing per-package C-states, continuing execution of a straggling request will destroy much of the low power state potential. This problem will become more prevalent with more parallelism in the processor. The proposed solution is to suspend all cores as soon as any single one becomes idle, while also putting a bound on the amount of delay for each request. The desired result is having the processor switch between states when all cores are busy or all are idle. The work to implement forced idle states in Dreamweaver is offloaded to a secondary low-power processor. Although memory is not considered in that work, the problem of coordinating idle time between worker threads is also very actual for DimmStore because memory DIMMs are shared between the threads. DimmStore addresses this problem in a simpler way, by imposing a predefined single schedule of restricted and unrestricted intervals for its Data region.

A separate class of work focuses on scheduling algorithms that maximize idle time on uni- and multiprocessors for energy savings. Typically, task scheduling problems apply in the real-time systems context, where a priori task execution information can be utilized. A continuum of scheduling algorithms has been proposed that balance the requirement for the tasks to meet deadlines, energy utilization, and computational complexity for various system models. A family of Energy-Saving Rate-Harmonized Scheduling (ES-RHS) algorithms [64, 32] schedules periodic tasks, according to their priorities, on boundaries of the periodic time interval, called Harmonizing Period. To produce optimal energy savings, *forced sleep* is inserted at the beginning of each Harmonizing Period, which duration is determined based on the worst-case execution time of all tasks. As a result, all idle time in the period is coalesced into a single idle interval. Although the real-time task scheduling has very distinct setting from where DimmStore operates, there are obvious analogies between periodic insertion of forced sleep in ES-RHS and restricted intervals in DimmStore. To address the problem of energy saving scheduling in multicores that go into C-states synchronously, the same work [32] defines the SyncSleep Scheduling, where

known real-time tasks are partitioned between cores, each running ES-RHS, to maximize common sleep time.

Typically, when processors are put to sleep, much of the entire system will become idle and transition to some sort of a lower power state. This especially applies to memory, and even though existing work does not specifically targets memory, with any technique that puts the processor into a sleep state the memory power consumption will be reduced as well. The real-time scheduling problem of Maximizing Common Idle Time (MCIT) is explored in [20] in the context of multiprocessors. Effects of several MCIT algorithms on power consumption of shared memory are evaluated in [35].

Chapter 6

Additional Related Work

In this chapter, we include references to related work on topics not previously covered in individual chapters. The first group of references describe existing approaches to memory power savings that do not rely on memory power states. These approaches are not specific to databases and include application of memory voltage and frequency control (Section 6.1) and saving on memory refresh energy (Section 6.2). We also present work on power saving techniques in DBMS, which do not specifically target memory. These include power-efficient query planning (Section 6.3), and application of processor DVFS in database systems (Section 6.4).

6.1 Memory Voltage and Frequency Control

Adjusting operating frequency according to changing application performance demand is a widely used technique to control energy consumption in a variety of computing systems, from smartphones to servers. In an operational state, the power consumed by CMOS circuits increases with the switching rate of logic gates. Since the switching rate does not necessarily directly correspond to the amount of useful work performed by the circuit, reducing frequency can help minimize energy consumption when utilization is low. Additionally, the supply voltage can be reduced as timing requirements become less tight at lower frequencies. The combined effect of power reduction due to frequency and voltage scaling is used in a technique called Dynamic Voltage and Frequency Scaling (DVFS). DVFS has been successfully used in CPU power management, but is not yet available for memory.

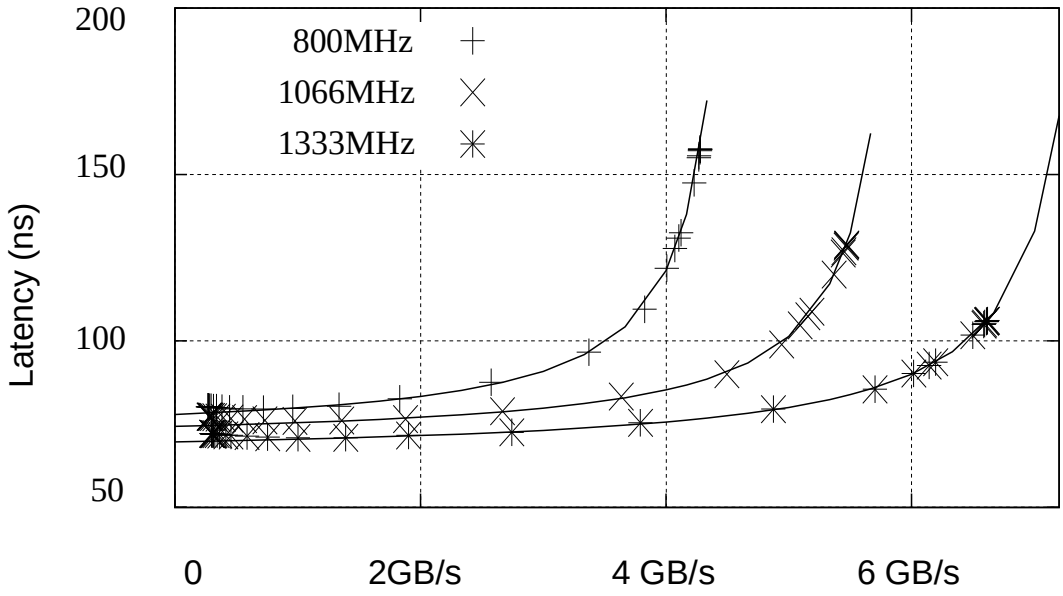

Figure 6.1: Memory latency in as a function of channel bandwidth demand, from David et al [24]

David et al [24] conducted an extensive experimental study of the potential of memory DVFS. Existing systems often allow a user to statically change memory frequency between reboots. Modern DRAM standards [6, 7] also define a procedure to change frequency at run time. However, there is no procedure to change voltage during operation, even though some systems include programmable DRAM voltage regulators. A number of potential issues need to be addressed to dynamically adjust voltage, such as data retention and need for interface recalibration. Nevertheless, it was found that minimum stable voltages of DDR3 decrease with frequency, albeit not significantly, with the reduction from 1333 to 800 MT/s. The time to switch voltage was estimated to be about 20 μs. According to the results of measurement and modelling, reducing frequency alone from 1333 MT/s in steps to 1066 MT/s and 800 MT/s would correspond to power reduction of about 8% in each step. The additional effect of voltage scaling was estimated to be about 6% per step.

Memory frequency reduction proportionally reduces the available bandwidth. The proposed power management strategy should increase memory frequency for applications with higher memory bandwidth demand. Particularly interesting was the observation of how frequency reduction affects the memory access latency, illustrated in Figure 6.1. As long the required bandwidth is much lower than the available bandwidth, the increase in latency is small and is determined by the longer transfer time. The latency starts to increase visibly due to waiting in the memory controller queues as the required bandwidth becomes roughly half of the peak. This observation led to a simple memory frequency control algorithm that uses constant bandwidth ranges for each frequency setting. The actual memory bandwidth is estimated during an interval, and memory switches to the frequency set for the corresponding bandwidth range. This technique showed the reduction of memory energy between 3.5-4.5% in the *non-memory-intensive* subset of SPEC CPU benchmark, with negligible impact on performance *for all the benchmarks* in the set.

Reducing memory frequency may negatively affect application performance because it reduces the memory throughput. The holy grail of DVFS is a frequency control algorithm that would minimize the consumed energy while keeping performance degradation to the minimum. Therefore, current research is largely focused on choosing the right spot on the energy-performance trade-off imposed by DVFS.

Deng et al [30] proposes present MemScale, a control algorithm for DVFS that adjusts the memory frequency in response to changes in the workload. The algorithm is based on the observation that performance of a workload with a higher ratio of instructions that produce the last level cache (LLC) misses is more sensitive to frequency reduction. Therefore, the counters representing the number of instructions executed and the number of LLC misses are used to characterize the workload in the current time interval. Based on this information, expected performance and power consumption is predicted by a corresponding

model, once for each of the available frequencies, and the one that entails a minimal power consumption for a fixed allowed performance degradation is chosen for the next interval.

This work does not investigate practical feasibility of DVFS, nor does it experimentally characterize memory power consumption under DVFS. The evaluation of the algorithm is based on the combination of simulation, modelling, and a number of assumptions, such as quadratic dependency of power consumption on voltage. Overall, the expected memory energy savings in the SPEC benchmark are between 6-31% with performance degradation restricted to 10%. For *non-memory-intensive* programs, the algorithms achieve maximum savings (about 31%) with small performance degradation (<2%), while the memory-intensive ones show less energy saving (5-20%) with higher degradation (4-8%).

An extension to MemScale for systems with multiple memory controllers, MultiScale, by Deng et al [29], is motivated by the adoption of multicore systems that exhibit non-uniform load between channels, or composed or heterogeneous hardware. Utilization skew between channels calls for per-channel, or per-controller, DVFS. The performance and power model is the same as in [30], but now an application may access more than one memory channel. By adjusting the frequencies of all channels, the system minimizes the total estimated energy while restricting its relative performance degradation.

Recent work by Sharifi [66] is a further development in DVFS control algorithms. As in MultiScale, memory frequency is adjusted independently for each memory channel in a multi-core system. Running applications are grouped according to their performance sensitivity to memory frequency and assigned to a CPU core with a channel operating at the optimal frequency for each group. A novel idea is considering *burstiness* of memory accesses when estimating the sensitivity. Thus, instead of using the counter of memory accesses as a measure of application memory *heaviness*, a counter for *distinct groups of memory accesses* is used. The intuition behind this idea is not deeply investigated, but it is based on the observation that applications with bursty memory access patterns tend to be less affected by a reduction in memory frequency, the average memory access rate being the same. Compared to the baseline from [30], the new algorithm improves power saving and reduces performance degradation due to DVFS on mixed and memory-intensive workloads.

The difficulty of assessing various DVFS algorithms comes from the fact that there is no support for memory DVFS in existing systems. The evaluation of proposed algorithms in [30, 29, 66] is performed under a simulator, which raises questions on how much of the improvement can be expected under real-life conditions.

Variable frequency support has been improved in the most recent standard of mobile DRAM, LP-DDR4 [1]. Even though previous DDR standards defined a procedure to

change frequency at run time, memory would need to undergo a set of time-consuming calibration sequences to adjust interface parameters to new timings. In LP-DDR4, mode registers that store calibrated settings are duplicated so that switching between two frequency operating points is possible without retraining. This improvement reduces the frequency switching time and can potentially improve effectiveness of workload-controlled DVFS implementations.

In general, managing memory power with DVFS is mostly complementary to the approaches based on memory power states. Power states provide higher power differential, while reducing frequency can be effective even when there are no long enough idle intervals. Since my strategy of memory power management causes highly skewed bandwidth utilization between channels, it may be useful to combine it with a per-channel DVFS mechanism.

6.2 Reducing Memory Refresh Energy

A separate category of work on memory power optimization is related to energy consumed when memory is idle. DRAM memory requires periodic refresh of its rows to retain information. Refreshing contributes to the background portion of DRAM energy consumption that does not depend on how actively memory is being used. Refresh power is the lowest level of a memory rank power consumption at which information is guaranteed to be safe. Quantitatively, it closely corresponds to power consumption in the Self Refresh state.

Refresh power is especially important for mobile devices that spend much time in a sleep state. However, refresh power can also be manipulated when regions of DRAM do not hold valid data or this data is not critical. Two existing methods can potentially be used to reduce the Self Refresh power: reducing the refresh rate and completely disabling refresh at the granularity of a memory rank or its subset. Since refresh is critical for data retention, both methods cause potential loss of all or part of the stored data. Let us consider the latter idea first.

Most of the current DRAM devices support a feature called Partial Array Self Refresh (PASR) that allows the system to restrict the subset of the array that is subject to refresh in the Self Refresh state. Reduction of the refreshed portion is accompanied by a reduction of power consumption, although not linearly. Thus, for the oldest LPDDR standard, refreshing 1/16 of the array reduces the Self Refresh power by 62% [18]. The Maximum Power Savings mode in LPDDR4 [1] can be seen as a coarse granular version of PASR, when refresh is disabled for a whole rank. Potential integration of PASR into Linux kernel

to reduce power consumption in a sleep mode was investigated by Brandt et al [18]. Since switching to or from a sleep mode can be made in a restricted OS code path that only uses a small dedicated memory region, all other physical allocated memory can be "compressed" into one of the refresh-enabled regions of DRAM during the transition. On device wake-up, a reverse "decompression" process takes place which restores the original memory map before the rest of the system is made operational. Kjellberg [49] presented a low-level kernel driver for a finer granularity PASR mode in LPDDR2 devices, which makes memory "compression" and "decompression" unnecessary. The driver is integrated with the kernel memory allocator and tracks usage of individual DRAM banks, which is the PASR unit in LPDDR2. Upon entering the sleep mode, the PASR is configured to exclude the unused banks from refresh. The proposed mechanism can only save power when the device is inactive. Despite these limitations, even in a running system memory can be in the Self Refresh state for significant time, while some of the memory remains unallocated. For such circumstances, Coquelin and Pallardi [22] describe a similar modification of the Linux kernel that aims at using PASR to avoid refreshing of unused physical memory. The proposed mechanism consists of a low-level driver that configures PASR through memory registers, and a high-level *governor* that controls memory compaction. However, there appears to be significant difficulties in designing an effective governor for a general purpose operating system, therefore, the proposed techniques have not made it to existing systems.

However, PASR is a potentially valuable method to save power in large-scale database systems when memory is underutilized. As with rank-aware allocation, there is gap between hardware capabilities and tools available to software as there is no interface for the software to describe the usage patterns of memory allocations. Closing this gap would mean designing system interfaces that would expose the power structure of the system and provide memory intensive applications, such as DBMS, tools to exploit these capabilities.

PASR and the Maximum Power Saving mode can be seen as an additional low-power state that allows the stored data to be lost. The common requirement for any power-state based power strategy is its ability to concentrate data with similar access characteristics into consecutive intervals of the physical address space. Once this is achieved, any remaining intervals may become large enough so that the associated PASR region, or a whole rank, can be switched to the state with no refresh.

An orthogonal method to reduce refresh energy is to reduce the rate of refresh. Such a reduction is possible both in Self Refresh state and when refresh commands are issued by the memory controller during normal operation. It has been noted that standard refresh interval (64 ms) is very conservative and increasing it to 1 second only causes errors at the rate of 4.0×10^{-8} [55]. This observation has lead to the design of Flikker [55], a system that saves memory energy but creates a possibility of a data loss in the part of

the program data that is not critical, such as lossy compressed images. Sparkk [56] is another, even more complex system of the same type, which stores *bits* of numeric values of various importance in memory modules that are refreshed at different rates. Therefore, less important bits are more likely to be corrupted while the important ones are stored intact. Shifting errors to less significant bits allows the system to reduce the average refresh rate for the same *perceived* level of data corruption. Techniques based on controlling the refresh interval establish a class of *approximate memory storage systems*, while general-purpose mainstream systems are not designed to accept any data corruption.

6.3 Energy Efficient Query Planning

The memory energy required to execute a particular database query may vary depending *how* the query is executed. For example, it was found that energy cost of the same query may vary by a factor of four when different query plans are chosen [62]. Such a large difference indicates that there is a high potential to save energy if the optimizer is made aware of *query energy cost*, in addition to execution cost. Not surprisingly, an impressive amount of research exists that explores the direction of power efficient query planning.

There are more factors that may affect the power cost of a query. Psaroudakis et al [62] look at the level of CPU parallelism and frequency to see how they affect CPU and memory power and system performance. Two algorithms typical for an in-memory DBMS, *concurrent partitioned* scan and *parallel aggregation*, in a multi-socket system, were considered. In particular, for a given level of load, how many cores in each of the CPU in the system should be allocated to query processing, to maximize the ratio of performance to combined energy consumption of the CPU and memory? It was found that that the optimal performance/energy ratio is achieved with the smallest number of CPUs used, such that the memory bus in each used socket is saturated. Knowing the amount of work that saturates the socket memory bus in a socket, we should evenly spread it among all the cores in the CPU, choosing the lowest sufficient frequency. This result can be explained if memory is the *least power proportional* component in the system, therefore, it is the most energy efficient when a unit of memory is fully utilized or completely idle. In my work, I have made the same observation and use a strategy to concentrate memory workload, but I do that with a finer granularity - memory ranks, instead of all memory attached to a CPU socket.

The power-performance behaviour is too complex to directly codify in a query optimizer, therefore, an open problem is navigating over the search space of query plans with

added energy parameters. A feasible strategy would be to build and calibrate a power-performance model on a number of operational points and use this model in the optimizer to associate a power cost for a query plan. A work by Gotz et al [36] had the focus of finding the best CPU frequency and number of threads, referred to as the *sweet spot*, in the context of a complex (TPC-H) workload. However, the paper stops short of suggesting an automated approach to integrate this information into the query optimizer.

Lang et al [52] look at the effects of query plan selection on energy consumption and efficiency. They show that, for example, different join algorithms, e.g. hash-join or merge-join, *may* have different energy/performance ratios. The best performance plan, which is normally selected by the optimizer, may not be the most power efficient. If a certain penalty in query response time is allowed, as specified in the Service Level Agreement (SLA), the system could save some energy by exploiting the energy/performance trade-off. To make this possible, the query optimizer must consider the energy cost of each operator. The proposed energy model predicts this cost as a weighted sum of expected CPU cycles, I/O reads and writes, and the number of memory accesses. The model only needs five parameters to learn and achieves an average error rate of around 3% and a peak error rate of 8%.

A similar way of leveraging a cost-based query optimizer to balance between query times and consumed energy is presented by Xu et al [79]. In their work, the single query cost metric combines both time and power costs using an adjustable weight factor. The trade-off between time and power costs forms essentially a Pareto curve, which shape is determined by the weight factor, for a query with a fixed total cost. The power cost component is computed by associating power costs for elementary operations, such as processing a tuple by the CPU or reading a page from the disk. Once the power component is integrated into the cost metric, the query optimizer can consider it in the otherwise unmodified plan search procedure.

The effectiveness of database energy optimization based on query plan selection depends on availability of reasonable plan candidates that realize this trade-off. For workloads that are dominated by simple and short queries, which is common e.g. in OLTP, there just may not be enough possible plans to choose from. On the other had, such workloads execute queries from multiple clients in parallel, which provides opportunities to adjust relative execution *scheduling* of the queries. For these workloads, the other possible approach is to adjust scheduling of individual queries.

6.4 CPU Frequency Scaling in DBMS

CPU frequency and voltage scaling (DVFS) is a general technique, usually applied to all applications running in a system.

In these techniques, both application performance and power consumption are modelled to choose the DVFS operating point that balances the power consumption and performance degradation. For a database server, the accuracy of performance modelling can be improved by allowing the database engine itself to act as a DVFS controller. Korkmaz et al [50] presents an algorithm to control CPU frequency scaling and scheduling for transactional workloads. In such a workload, the level of system performance can be characterized by response times of individual transactions. Provided with a *latency target* for transactions and estimating their expected execution times at various frequencies, the system can choose a frequency setting such that the majority of transactions during a time interval satisfy their latency targets. To further tighten the energy budget, POLARIS schedules transaction execution according to their deadlines. The combined effect of frequency scaling and transaction reordering allows the system to save energy under a variety of load conditions while keeping the number of missed latency targets to a minimum. A related but less sophisticated algorithm, without transaction reordering, was described earlier by Kasture et al [47].

A significant feature of existing techniques for achieving CPU power efficiency is their reliance on frequency settings (P-states) while *idle states* (C-states) do not seem to be useful as long as the processor is at least minimally loaded. Memory, in contrast, does not currently support frequency scaling, and current progress in power efficiency fully relies on idle states (low power states). However, the theoretical application of memory frequency scaling has been explored (see Section 3.6.1) and one should expect its practical implementation in near future. If that happens, the ideas of DBMS-resident frequency control could be extended to *both* CPU and memory.

Chapter 7

Conclusion

Memory power consumption is a significant contributor to servers' energy footprint, but is also relatively unexplored area in the wider space of power optimization research. In this work, we attempt to understand how memory consumes power and develop techniques to reduce the memory power consumption of main-memory database systems. Main-memory database systems are an important case for memory power optimization because of their requirement to fit the entire database in memory, including space for future growth.

We characterized the memory power consumption in existing database systems (Chapter 3) and showed that memory power consumption stays high regardless of the database size, and only weakly depends on load. This happens because the bulk of the power consumption is background power, which depends on power state residencies, but is not proportional to load. Memory allocation and accesses in existing systems tend to be spread evenly across DIMMs and over time. Even when memory is not fully used and load is lower than peak, accessing all DIMMs at the same rate does not create long idle intervals on any DIMMs. Long idle intervals are required for low-power states to be used. In the following chapters, we develop two orthogonal mechanisms to reduce background power consumption.

In Chapter 4, we present a power-saving DBMS prototype, DimmStore, based on rate-based placement and rank-aware memory allocation. With these techniques, frequently used data is concentrated in a subset of DIMMs, so that the access rate in the remaining DIMMs is reduced. We demonstrated significant power savings (up to 50%) in DimmStore in transactional workloads with minimal performance impact. Most memory power savings occur due to minimization of power consumption in unused regions of memory when database size is smaller than the amount of available memory. Power consumption in rarely

used DIMMs is also reduced, however, the amount of reduction depends on the residual access rate to these DIMMs. Workloads with simpler and more regular access patterns, such as YCSB, demonstrate larger power savings than more complex workloads.

In Chapter 5, we augment DimmStore with a *memory access gating* mechanism, which targets memory power consumption in the region with rarely-accessed data. Although basic DimmStore reduces the average access rate in that region, these memory accesses were still evenly distributed over time. Memory access gating inserts long periods of idleness in the rarely accessed DIMMs by periodically stalling threads that attempt to access those DIMMs. DimmStore with memory access gating specifically reduces power consumption in used but rarely accessed DIMMs. Added longer idle intervals and shifting some accesses to other time intervals impacts performance. We analyzed the power/performance tradeoff of gating and discussed the ways to tune gating parameters to navigate this trade-off according to application requirements.

Although I successfully prototyped the power saving techniques in an open-source main-memory DBMS, I expect more consistent results in a system where memory power optimization is a part of its design. To enable a clean implementation of rank-aware allocation, the operating system should provide a system interface for applications to discover the physical memory layout and route memory allocations into specified physical memory regions. Such an interfaces may be modelled after the set of APIs for management of threads and memory allocations in NUMA systems, such as `libnuma`. Possible designs of system interfaces for rank-aware allocations are discussed in more detail in Section 8.3.1.

The rank-aware allocation and rate-based placement techniques are generally applicable to data management applications and have little performance overhead. Therefore, I would recommend to implement them in new and existing systems to improve power efficiency. The biggest consideration is a mechanism to identify and maintain access rate information for data items, which should be tailored to the data structures used in the system.

The memory access gating technique produced workload-dependent power savings, with a higher performance impact. It would be beneficial to include it when the system may experience a high variance in load and the application can tolerate an increase in response times. An important implementation consideration is integration of idle interval scheduling due to gating with other work scheduling in the system. In particular, the system should be able to insert the idle intervals with good timing accuracy and execute the batch of delayed work without interruption. With a careful implementation, a favorable balance of performance overhead and power savings can be achieved.

Chapter 8

Future Work

In this chapter, we outline future directions for the work on memory power consumption, started in this thesis. In Section 8.1, we discuss possible improvements to the techniques used in DimmStore. Later in Section 8.2, we elaborate on extending the techniques from this work to other workloads and types of systems. In Section 8.3, we consider hypothetical support from operating systems that would make the techniques described in this work more practical.

8.1 DimmStore improvements

8.1.1 Rank-aware index allocation

In its current implementation, DimmStore always allocates memory for indexes in the System region. This reduces the amount of scarce System region memory available for storing hot data tuples. Since every key in a secondary index corresponds to a tuple in the primary table store, it may be possible to distinguish between frequently and infrequently used keys and place them to the System and Data regions, respectively. The difficulties in rank-aware index key placement are the following:

- In existing indexes, the location (address) of a key is uniquely determined by the data structure used by the index. For example, in a B-Tree, a key must located in a block according to its value. In a hash table, the location of the key in the table is determined by its hash value. It is not possible to separately store the cold subset of keys in a different location.

- Index keys are often small and the memory overhead associated with computing and adding and maintaining frequency information will increase storage and run time costs.
- The flag indicating that the tuple is in the Data region iself currently is also stored in the index. Not storing this flag for tuples in the Data region will require a two-step search procedure, first in the System region, and if not found, in the data region.

A Hypothetical Hot/Cold B-Tree Index

In a B-Tree, keys are organized in blocks of a fixed size (nodes), using pointers to form a tree. A B-tree search traverses the tree starting from the root and ending in a leaf node containing the key. Due to the fanout, the upper levels of the tree are accessed more often than the lower levels. If only a small number of keys in the leaf nodes are frequently accessed, it will beneficial to store frequently-accessed keys in a separate set of nodes and move the least accessed nodes to the Data region. Essentially, the index will consist of two trees, hot and cold. The hot part of the index will likely be smaller and will have fewer levels. Searching in such an index would first involve searching the hot part, and if the value was not found, continue searching the cold part. Since the hot part is smaller, even an increased number of total block traversals may impose only a small cost, which will be offset by the reduction in memory use.

Since the keys in leaf nodes uniquely correspond to tuples, an additional mechanism to track the frequency of access to the keys is not needed. The keys in leaf nodes are moved between the hot and cold parts at the same time the corresponding tuples are evicted or unevicted. The keys in the upper levels of the cold part can be assumed to be frequently used and always stored in the System region.

The potential problem with two-stage hot-cold index is frequent searches for non-existent keys. Such searches will always fall back to search in the cold part, unnecessarily increasing the access rate to the Data region. If such searches are expected, this problem can be addressed by adding markers for frequently searched non-existing keys in the hot part of the index and using additional Bloom filters.

8.1.2 Dynamic Sizing of the System region

In DimmStore, the number of DIMMs in the System region is configured before starting the system. A smaller System region is beneficial for memory power reduction (because of

more Data region DIMMs) but it would incur a higher cost (performance and, potentially, power too) of evictions and unevictions. The two issues here are to determine the optimal size of the System region, and to implement the incremental reclassification of one DIMM from System to Data region, or vice verse.

Since the System region is similar to a cache, the algorithms used for dynamically sizing caches should also apply. Alternatively, a simple heuristic approach may be based on limiting (or fixing) the rate of evictions while allowing uneviction to occur according to the workload. To keep the System region constant, these processes must be balanced. A too small System region would cause an increase in the rate of unevictions. If the eviction rate does not change, the System region will naturally grow. As the System region grows, its miss rate is expected to decrease, eventually balancing out evictions and unevictions when it reaches a certain size.

Extending or shrinking the System region would involve data migration. In contrast to caches, tuples in the System region are not duplicated in the Data region. However, the data migration may happen asynchronously and the additional cost will amortize over time. When extending or shrinking the Data region by one DIMM, the chosen DIMM will be temporarily shared by both regions. Allocating space for new (inserted or unevicted) tuples will use free space in this DIMM. The tuples that existed in this DIMM while it was in the Data region, will be evicted at the first priority, possibly even without observing the eviction rate limit.

8.1.3 Shared Data Region

The current DimmStore implementation applies the H-Store partitioning scheme to both System and Data region. Essentially, every partition has its own fraction of the total memory, divided into the partition's System and Data regions. This was done because in H-Store, all partitions are completely independent. However, since partitions access their Data regions very infrequently, it may be possible to share a single Data region between partitions with a minimal synchronization cost.

The benefit of the shared Data region would an improved memory allocation efficiency in case partitions allocate memory unevenly.

8.1.4 Rate-aware Data Region

Currently, DimmStore does not take advantage of having multiple DIMMs in the Data region. It allocates memory starting from the physical address of the first DIMM, and

continues to allocate memory sequentially in all DIMMs. Therefore, the distribution of tuples in the Data region does not follow their access rate.

There is almost certainly a skew in the access rate to tuples in the Data region. It may be possible to reduce the total power consumption by migrating tuples to each DIMM, according to their access rate. This additional access skew to the Data region's DIMMs may reduce the residual access to the coldest set of DIMMs so that the inter-access intervals are long enough for the Self Refresh power state.

In addition, if memory gating is used, the "colder" Data region's DIMMs may be configured with more aggressive gating settings than the "warmer" DIMMs. If shared Data region is also implemented (Section 8.1.3), the access rate skew between DIMMs may be higher as it will include all the tuples in the Data region.

8.2 Future Extensions

8.2.1 Analytical Workloads

In this work, we focus on transactional workloads. These workloads are characterized by a high rate of transactions and small amount of work in each. Tuples are typically accessed using indexes with little scanning. Transactions typically arrive from many clients and are largely independent. Memory accesses are predominantly random and little of the total memory bandwidth is used.

The other class of database workload is Online Analytical Processing (OLAP). OLAP workloads are antithetical to OLTP workloads. User queries process large amount of data, often with large scans. These properties of the workload result in higher memory bandwidth utilization, which makes it more difficult to save memory power.

The first concern with OLAP workloads in conjunction with memory power saving techniques is memory interleaving. As it was discussed in Chapter 3, disabling interleaving reduced query performance. However, it is important that different queries were affected to different degree, which means that different queries have different requirements for memory throughput. Since the only other performance bottleneck is the CPU, different queries have different ratio of needed CPU cycles and memory bandwidth.

Analytical queries typically consist of several operators, each implementing a known algorithm. For each query operator, its performance requirements may be estimated. In the worst case, a memory-bound operator will saturate the memory bus and we cannot

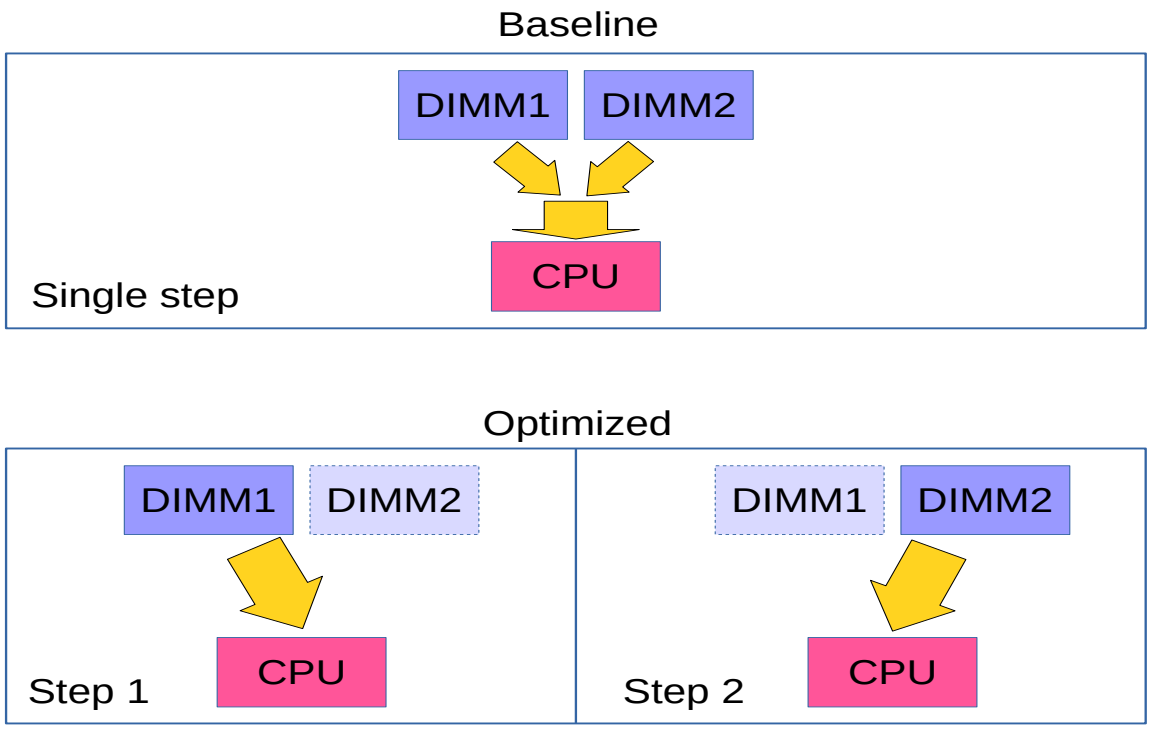

Figure 8.1: Data flow between DIMMs and processor, for a query operator requires the bandwidth of one CPU and two DIMMs (memory channels).

expect to save memory power. However, if the operator is CPU-bound and the required memory bandwidth is lower than the maximum available from all channels, it may be still obtained using a smaller number of memory channels. In analytical DBMS, for example in MonetDB, data is often broken down to blocks so that one block is processed at a time. The primary goal is performance optimization due to more efficient use of processor caches.

A memory power-efficient approach to OLAP may be based on scheduling processing of blocks located in particular DIMMs to control the memory bandwidth, as illustrated in Figure 8.1. A query operator that needs the total memory bandwidth provided by two channels to saturate all CPUs, will start processing only blocks locating in DIMMs installed in two channels. The remaining DIMMs will be unused during this time. Once these data blocks are processed, the system will switch to process the data blocks located in DIMMs of the next two channels, and so on.

8.2.2 NUMA Systems with Large Number of Nodes

The DimmStore design is also applicable to system with a larger number of processors and NUMA nodes. Since the great majority of memory accesses are directed to the System region, NUMA strategies related to memory allocation are only important for the System region and should be the same as in the baseline system.

If the workload exhibits a high degree of access locality, it would be beneficial to allocate a separate System region in every node in the system and pin the worker threads to the local CPU. If the access locality is poor, pinning threads to cores will provide less advantage.

If the workload is unevenly distributed between nodes, it may be beneficial to allocate the System region only on a subset of nodes, leaving the other nodes with Data region memory only. The worker threads should preferably use the cores of the nodes with System regions. In that case, memory power consumption may be reduced because of a smaller System region.

8.3 System Support

8.3.1 Support form Platforms/BIOS

Memory Configuration Discovery

The system should be able to report its memory information for two reasons. First, knowing its configuration is necessary when deploying load in a distributed system. Second, it is needed for rank-aware allocations.

Currently, memory information is maintained by the platform BIOS and can be reported to the operating system through the DMI interface [3] and applications can read it with the `dmidecode` tool. This information includes the list of memory modules, their sizes, frequency, and NUMA location. This information is sufficiently complete to estimate system's capabilities for the purpose of load balancing. However, the DMI information in out test server does not indicate interleaving configuration and reported modules' address ranges did not correspond to actual physical addresses as configured in the memory controller. With interleaving, multiple modules may serve one address range, which DMI is likely not able to represent.

For rank-aware allocation, the platform should additionally provide information to the operating system derived from the memory controller configuration. This information should include the following items:

- the list of memory regions with identical configuration and their accurate address ranges;
- the configuration for each region, such as interleaving granularity and state management policy;
- the mapping of regions to physical devices as reported by DMI.

Dynamic and Partial Memory Configuration

Currently, memory configuration is a cumbersome process. Memory settings are set manually by the boot-time BIOS program and require a lengthy reboot to take affect. Changing memory configuration for different application requirements can be significantly sped up if the memory controller can be reconfigured from the operating system. The Linux and Windows operating systems can already "hot-add" and "hot-remove" memory devices to survive such reconfiguration. Based on scarce information available for Intel memory controllers[8], it can be speculated that multiple regions with different interleaving configurations can be supported at the same time. For example, a system with 16 DIMMs on 4 channels may be configured with interleaving the first DIMM in each channel, proving a 4 DIMM region of "fast" memory, and without interleaving for the remaining 12 DIMMs. For power savings, the non-interleaved DIMMs should be able to use the Self Refresh state independently from other DIMMs on the same channel.

Explicit Control of Memory Power States

Memory power efficiency could be improved if the software could directly switch memory power states. Currently, such state transitions are triggered by a timeout in the memory controller. Direct state control could eliminate the need to enforce long idle intervals during which memory is in a higher-power state only to induce a state transition. Alternatively, a similar effect can be achieved by keeping the state transitions to the controller but allowing the software to set timeouts dynamically.

8.3.2 Support from Operating Systems

Operating systems do not currently provide an interface for applications to request memory from a particular physical memory region. This is the reason why DimmStore "hides"

memory from the operating system. The major drawback of this approach is the requirement for superuser privileges for the DBMS process. To make physical memory allocation more practical, the operating system should provide tools that can be based on existing capabilities. The existing mechanisms in Linux operating system that serve a similar purpose are the `madvise` and `set_mempolicy` interfaces, and the `hugetlbfs` filesystem. The `madvise` system call allows the application to apply an integer bit mask for a virtual memory region, according to the usage pattern it expects for this region. Some of the flags set performance expectations for the region. For example, there are bits to indicated expected sequential or random access patterns, and whether this region will be accessed soon. The limitation of this interface is due to a limit on the number of bits in the bit mask and lack of additional arguments. The `set_mempolicy` interface controls on which nodes of the NUMA system memory allocations should take place for the current process. This call takes a mask of NUMA nodes and is not directly applicable to control memory allocations within a node, but a similar call may be added that accepts a mask of *physical memory modules* instead. Additionally, the call should apply to particular memory allocations, for example, specified as a virtual address range. The `hugetlbfs` file system allows the system administrator to "mount" memory backed up by huge pages into a directory such that all files in that directory will use huge pages. The directory can be made to be owned by any user so superuser privileges may not be required. A similar system mechanism could be envisioned to dynamically associate physical memory regions with user virtual memory allocations.

References

[1] Low power double data rate 4 (LPDDR4). JEDEC standard, JEDEC SOLID STATE TECHNOLOGY ASSOCIATION. 121, 122

[2] MK4096. 4096 x 1 bit dynamic RAM. Datasheet, MOSTEK, 1973. 7

[3] Desktop management interface specification. version 2.0. Specification, Desktop Management Task Force, 1998. 134

[4] TN_41_01: Calculating memory system power for DDR SDRAM. Technical note, Micron, 2007. 8, 13, 17

[5] Transaction Processing Performance Council. TPC BENCHMARK C. Standard Specification. Revision 5.11, 2010. 44, 67

[6] DDR3 SDRAM. JEDEC STANDARD JESD79-3F, JEDEC SOLID STATE TECHNOLOGY ASSOCIATION, July 2012. 6, 17, 120

[7] DDR4 SDRAM. JEDEC STANDARD JESD79-4A, JEDEC SOLID STATE TECHNOLOGY ASSOCIATION, October 2013. 6, 8, 9, 12, 17, 120

[8] Intel Xeon processor E5 and E7 v4 product families uncore performance monitoring reference manual. JEDEC standard, Intel, April 2016. 9, 19, 135

[9] MCP3914. 3v eight-channel analog front end. Datasheet, Microchip, 2020. 12

[10] Ahmed M. Amin and Zeshan A. Chishti. Rank-aware cache replacement and write buffering to improve DRAM energy efficiency. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design - ISLPED '10*, page 383. ACM Press, 2010. 75

[11] Hrishikesh Amur, Ripal Nathuji, Mrinmoy Ghosh, Karsten Schwan, and Hsien-Hsin S Lee. Idlepower: Application-aware management of processor idle states. In *Proceedings of the Workshop on Managed Many-Core Systems, MMCS*, volume 8. Citeseer, 2008. 116

[12] Raja Appuswamy, Matthaios Olma, and Anastasia Ailamaki. Scaling the memory power wall with DRAM-aware data management. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, pages 3:1–3:9. ACM, 2015. 1, 38, 39

[13] Maurice J. Bach. *The Design of the UNIX Operating System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986. 79

[14] Chang S Bae and Tayeb Jamel. Energy-aware memory management through database buffer control. In *Proceedings of the Third Workshop on Energy-Efficient Design (WEED)*, 2011. 79

[15] Sorav Bansal and Dharmendra S. Modha. Car: Clock with adaptive replacement. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, FAST '04, pages 187–200, Berkeley, CA, USA, 2004. USENIX Association. 80

[16] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. 18

[17] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in MonetDB. *Commun. ACM*, 51(12):77–85, December 2008. 28, 81

[18] Todd Brandt, Tonia Morris, and Khosro Darroudi. Analysis of the PASR standard and its usability in handheld operating systems such as Linux, 2007. 122, 123

[19] Karthik Chandrasekar, Benny Akesson, and Kees Goossens. Improved power modeling of DDR SDRAMs. In *2011 14th Euromicro Conference on Digital System Design*, pages 99–108. IEEE, August 2011. 8, 13

[20] Jessica Chang, Harold N. Gabow, and Samir Khuller. A model for minimizing active processor time. In Leah Epstein and Paolo Ferragina, editors, *Algorithms – ESA 2012*, pages 289–300, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 117

[21] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. SoCC*, pages 143–154, 2010. 44, 54

[22] Maxime Coquelin and Loc Pallardy. PASR framework. saving the power consumption of the unused memory, 2012. 123

[23] F. J. Corbat. A paging experiment with the multics system. In *In Honor of P. M. Morse*, pages 217–228. MIT Press, 1969. 80

[24] Howard David, Chris Fallin, Eugene Gorbatov, Ulf R. Hanebutte, and Onur Mutlu. Memory power management via dynamic voltage/frequency scaling. page 31. ACM Press, 2011. xv, 119, 120

[25] Howard David, Eugene Gorbatov, Ulf R. Hanebutte, Rahul Khanna, and Christian Le. RAPL: Memory power estimation and capping. In *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '10, pages 189–194. ACM, 2010. 13, 19

[26] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stan Zdonik. Anti-Caching: A new approach to database management system architecture. *PVLDB*, 6(14):1942–1953, 2013. 47

[27] V. Delaluz, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Energy-oriented compiler optimizations for partitioned memory architectures. In *Proceedings of the 2000 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '00, pages 138–147. ACM, 2000. 75

[28] V. Delaluz, A. Sivasubramaniam, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Scheduler-based DRAM energy management. page 697. ACM Press, 2002. 77

[29] Qingyuan Deng, David Meisner, Abhishek Bhattacharjee, Thomas F. Wenisch, and Ricardo Bianchini. MultiScale: memory system DVFS with multiple memory controllers. page 297. ACM Press, 2012. 121

[30] Qingyuan Deng, David Meisner, Luiz Ramos, Thomas F. Wenisch, and Ricardo Bianchini. MemScale: active low-power modes for main memory. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems - ASPLOS '11*, page 225. ACM Press, 2011. 120, 121

[31] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. A validation of DRAM RAPL power measurements. pages 455–470. ACM Press, 2016. 19

[32] Sandeep DSouza, Anand Bhat, and Ragunathan Rajkumar. Sleep scheduling for energy-savings in multi-core processors. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 226–236, 2016. 116

[33] Mootaz Elnozahy, Michael Kistler, and Ramakrishnan Rajamony. Energy conservation policies for web servers. In *Proceedings of the 4th Conference on USENIX Symposium on Internet Technologies and Systems - Volume 4*, USITS'03, page 8, USA, 2003. USENIX Association. 115

[34] Xiaobo Fan, Carla Ellis, and Alvin Lebeck. Memory controller policies for DRAM power management. In *Proceedings of the 2001 International Symposium on Low Power Electronics and Design*, ISLPED '01, pages 129–134. ACM, 2001. 10, 75

[35] Chenchen Fu, Yingchao Zhao, Minming Li, and Chun Jason Xue. Maximizing common idle time on multicore processors with shared memory. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(7):2095–2108, 2017. 117

[36] S. Gotz, T. Ilsche, J. Cardoso, J. Spillner, T. Kissinger, U. Assmann, W. Lehner, W.E. Nagel, and A. Schill. Energy-efficient databases using sweet spot frequencies. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*, pages 871–876, December 2014. 125

[37] Hai Huang, Padmanabhan Pillai, and Kang Shin. Design and implementation of power-aware virtual memory. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, 2003. 50, 77, 78

[38] Hai Huang, Kang G. Shin, Charles Lefurgy, and Tom Keller. Improving energy efficiency by making DRAM less randomly accessed. page 393. ACM Press, 2005. 50, 78

[39] Volker Hudlet. True energy-efficient data processing is only gained by energy-proportional DBMSs, 2010. 41

[40] Ibrahim Hur and Calvin Lin. A comprehensive approach to DRAM power management. In *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*, pages 305–316. IEEE, 2008. 75

[41] Gangyong Jia, Guangjie Han, Jinfang Jiang, and Joel J.P.C. Rodrigues. PARS: A scheduling of periodically active rank to optimize power efficiency for main memory. 58:327–336, December 2015. 77

[42] Gangyong Jia, Xi Li, Jian Wan, Liang Shi, and Chao Wang. Coordinate page allocation and thread group for improving main memory power efficiency. In *Proceedings of the Workshop on Power-Aware Computing and Systems*, HotPower '13, pages 7:1–7:5. ACM, 2013. 50, 77

[43] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-mt: A scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 24–35, New York, NY, USA, 2009. ACM. 21

[44] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-Store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, 2008. 44

[45] Alexey Karyakin and Kenneth Salem. An analysis of memory power consumption in database systems. In *Proceedings of the 13th International Workshop on Data Management on New Hardware*, DAMON '17, pages 2:1–2:9, New York, NY, USA, 2017. ACM. iv, 13, 15

[46] Alexey Karyakin and Kenneth Salem. Dimmstore: Memory power optimization for database systems. *Proc. VLDB Endow.*, 12(11):1499–1512, July 2019. iv

[47] Harshad Kasture, Davide B. Bartolini, Nathan Beckmann, and Daniel Sanchez. Rubik: Fast analytical power management for latency-critical systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO-48, pages 598–610, New York, NY, USA, 2015. ACM. 126

[48] Brent Keeth, R. Jacob Baker, Brian Johnson, and Feng Lin. *DRAM Circuit Design: Fundamental and High-Speed Topics*. Wiley-IEEE Press, 2nd edition, 2007. 6

[49] Henrik Kjellberg. Partial Array Self-Refresh in Linux, 2010. 123

[50] Mustafa Korkmaz, Martin Karsten, Kenneth Salem, and Semih Salihoglu. Workload-aware cpu performance scaling for transactional database systems. In *Proceedings*

of the 2018 International Conference on Management of Data, SIGMOD '18, pages 291–306, New York, NY, USA, 2018. ACM. 126

[51] Karthik Kumar, Kshitij Doshi, Martin Dimitrov, and Yung-Hsiang Lu. Memory energy management for an enterprise decision support system. In *Proceedings of the 17th IEEE/ACM International Symposium on Low-power Electronics and Design*, ISLPED '11, pages 277–282. IEEE Press, 2011. 38, 41

[52] Willis Lang, Ramakrishnan Kandhan, and Jignesh Patel. Rethinking query processing for energy efficiency: Slowing down to win the race. 34:12–23, 2011. 125

[53] Alvin R. Lebeck, Xiaobo Fan, Heng Zeng, and Carla Ellis. Power aware page allocation. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IX, pages 105–116. ACM, 2000. 75

[54] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T.W. Keller. Energy management for commercial servers. 36(12):39–48, December 2003. 1

[55] Song Liu, Karthik Pattabiraman, Thomas Moscibroda, and Benjamin G. Zorn. Flikker: saving DRAM refresh-power through critical data partitioning. page 213. ACM Press, 2011. 123

[56] Jan Lucas, Mauricio Alvarez-Mesa, Michael Andersch, and Ben Juurlink. Sparkk: Quality-scalable approximate storage in DRAM. 2014. 124

[57] Nimrod Megiddo and Dharmendra S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, pages 115–130, Berkeley, CA, USA, 2003. USENIX Association. 80

[58] David Meisner, Brian T. Gold, and Thomas F. Wenisch. PowerNap: Eliminating server idle power. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 205–216. ACM, 2009. 41

[59] David Meisner and Thomas F. Wenisch. Dreamweaver: Architectural support for deep sleep. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, page 313324, New York, NY, USA, 2012. Association for Computing Machinery. 116

[60] Justin Meza, Mehul A. Shah, Parthasarathy Ranganathan, Mike Fitzner, and Judson Veazey. Tracking the power in an enterprise decision support system. In *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '09, pages 261–266. ACM, 2009. 41

[61] Raik Niemann, Nikolaos Korfiatis, Roberto Zicari, and Richard Gbel. Does query performance optimization lead to energy efficiency? a comparative analysis of energy efficiency of database operations under different workload scenarios. abs/1303.4869, 2013. 42

[62] Iraklis Psaroudakis, Thomas Kissinger, Danica Porobic, Thomas Ilsche, Erietta Liarou, Pnar Tzn, Anastasia Ailamaki, and Wolfgang Lehner. Dynamic fine-grained scheduling for energy-efficient main-memory queries. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, pages 1:1–1:7. ACM, 2014. 124

[63] K. Rajamani. Memsim user's guide. Research report rc23431, IBM, 2004. 18

[64] Anthony Rowe, Karthik Lakshmanan, Haifeng Zhu, and Ragunathan Rajkumar. Rate-harmonized scheduling and its applicability to energy management. *IEEE Transactions on Industrial Informatics*, 6(3):265–275, 2010. 116

[65] Daniel Schmidt and Norbert Wehn. Dram power management and energy consumption: A critical assessment. In *Proceedings of the 22Nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*, SBCCI '09, pages 32:1–32:5, New York, NY, USA, 2009. ACM. 18

[66] 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 *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 210–220, September 2017. 121

[67] Reza Sherkat, Colin Florendo, Mihnea Andrei, Anil K. Goel, Anisoara Nica, Peter Bumbulis, Ivan Schreter, Günter Radestock, Christian Bensberg, Daniel Booss, and Heiko Gerwens. Page as you go: Piecewise columnar access in sap hana. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1295–1306, New York, NY, USA, 2016. ACM. 81

[68] Radu Stoica and Anastasia Ailamaki. Enabling efficient OS paging for main-memory OLTP databases. 2013. 80

[69] Radu Stoica, Justin J. Levandoski, and Per-Ake Larson. Identifying hot and cold data in main-memory databases. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)*, ICDE '13, pages 26–37, Washington, DC, USA, 2013. IEEE Computer Society. 80

[70] Balaji Subramaniam and Wu-chun Feng. Towards energy-proportional computing for enterprise-class server workloads. page 15. ACM Press, 2013. xii, 40

[71] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. E-store: Fine-grained elastic partitioning for distributed transaction processing systems. *Proc. VLDB Endow.*, 8(3):245–256, November 2014. 81

[72] Shyamkumar Thoziyoor, Jung Ho Ahn, Matteo Monchiero, Jay B. Brockman, and Norman P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 51–62, Washington, DC, USA, 2008. IEEE Computer Society. 18

[73] Dimitris Tsirogiannis, Stavros Harizopoulos, and Mehul A. Shah. Analyzing the energy efficiency of a database server. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 231–242. ACM, 2010. 41

[74] Thomas Vogelsang. Understanding the energy consumption of dynamic random access memories. pages 363–374. IEEE, December 2010. 17

[75] David Wang, Brinda Ganesh, Nuengwong Tuaycharoen, Kathleen Baynes, Aamer Jaleel, and Bruce Jacob. Dramsim: A memory system simulator. *SIGARCH Comput. Archit. News*, 33(4):100–107, November 2005. 18

[76] Zhong Wang and Xiaobo Sharon Hu. Energy-aware variable partitioning and instruction scheduling for multibank memory architectures. 10(2):369–388, April 2005. 76

[77] Donghong Wu, Bingsheng He, Xueyan Tang, Jianliang Xu, and Minyi Guo. RAMZzz: Rank-aware dram power management with dynamic migrations and demotions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 32:1–32:11. IEEE Computer Society Press, 2012. 50, 78

[78] Zichen Xu. Building a power-aware database management system. In *Proceedings of the Fourth SIGMOD PhD Workshop on Innovative Database Research*, IDAR '10, pages 1–6. ACM, 2010. 41

[79] Zichen Xu, Yi-Cheng Tu, and Xiaorui Wang. Exploring power-performance tradeoffs in database systems. In *2010 IEEE 26th International Conference on Data Engineering (ICDE)*, pages 485–496, March 2010. 125

[80] M. T. Yourst. Ptlsim: A cycle accurate full system x86-64 microarchitectural simulator. In *2007 IEEE International Symposium on Performance Analysis of Systems Software*, pages 23–34, April 2007. 18

[81] H. Zheng, J. Lin, Z. Zhang, E. Gorbatov, H. David, and Z. Zhu. Mini-rank: Adaptive DRAM architecture for improving memory power efficiency. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 210–221, November 2008. 8