Database and System Design for Emerging Storage Technologies

by

Steven Pelley

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in the University of Michigan
2014

Doctoral Committee:

Associate Professor Thomas F. Wenisch, Chair
Professor Peter M. Chen
Assistant Professor Michael J. Cafarella
Assistant Professor Zhengya Zhang

ACKNOWLEDGEMENTS

Completing a Ph.D. is a long, daunting process, and I could not have succeeded without help from many people. I would first like to thank my close friends and family for all their support, especially my parents; my sister, Carolyn; and Christie Ferguson. Next I would like to thank my collaborators, advisors, and committee for teaching by example, especially my dissertation advisor, Tom Wenisch; James VanGilder; Jack Underwood; Partha Ranganathan; Jichuan Chang; Kristen LeFevre; Brian Gold; Bill Bridge; Peter Chen; Mike Cafarella; and Zhengya Zhang. Thank you to all of my lab mates and department friends who assisted me with projects, listened to my problems over coffee, released steam playing basketball, or procrastinated with a game of Starcraft, including David Meisner, Faissal Sleiman, Mark Woh, Ben Cassell, Dan Fabbri, Timur Alperovich, Aasheesh Kolli, Richard Sampson, Prateek Tandon, Neha Agarwal, and Jeff Rosen. Finally, thank you to everyone else in the department, Trevor Mudge's lab across the hall, Scott Mahlke's lab next door, the ACAL reading group, and everyone attending my defense.

TABLE OF CONTENTS

LIST OF FIGURES

Figure

LIST OF TABLES

Table

LIST OF APPENDICES

Appendix

ABSTRACT

Database and System Design for Emerging Storage Technologies
by
Steven Pelley

Chair: Thomas F. Wenisch

Emerging storage technologies offer an alternative to disk that is durable and allows faster data access. Flash memory, made popular by mobile devices, provides block access with low latency random reads. New nonvolatile memories (NVRAM) are expected in upcoming years, presenting DRAM-like performance alongside persistent storage. Whereas both technologies accelerate data accesses due to increased raw speed, used merely as disk replacements they may fail to achieve their full potentials. Flash's asymmetric read/write access (i.e., reads execute faster than writes) opens new opportunities to optimize Flash-specific access. Similarly, NVRAM's low latency persistent accesses allow new designs for high performance failure-resistant applications.

This dissertation addresses software and hardware system design for such storage technologies. First, I investigate analytics query optimization for Flash, expecting Flash's fast random access to require new query planning. While intuition suggests scan and join selection should shift between disk and Flash, I find that query plans chosen assuming disk are already near-optimal for Flash. Second, I examine new opportunities for durable, recoverable transaction processing with NVRAM. Existing disk-based recovery mechanisms impose large software overheads, yet updating data in-place requires frequent device synchronization that limits throughput. I introduce a new design, *NVRAM Group Commit*, to amortize synchronization delays over many transactions, increasing throughput at some cost to transaction latency. Finally, I propose a new framework for persistent programming and memory systems to enable high performance recoverable data structures with NVRAM, extending memory consistency with persistent semantics to introduce *memory persistency*.

CHAPTER I

Introduction

For decades disk has been the primary technology for durable and large-capacity storage. Although inexpensive and dense, disk provides high performance only for coarse-grained sequential access and suffers enormous slowdowns for random reads and writes. Recently, several new technologies have emerged as popular or viable storage alternatives. Flash memory, primarily used for mobile storage, has gained traction as a high-performance enterprise storage solution. Nonvolatile Random Access Memories (NVRAM), such as phase change memory and spin-transfer torque RAM, have emerged as high performance storage alternatives [14].

These technologies offer significant performance improvements over disk, while still providing durability with great storage capacity. As drop-in replacements for disk, Flash and NVRAM greatly accelerate storage access. However, the disk interface fails to leverage specific device characteristics. Section 2.1 provides a background on these storage technologies and specifically how their performance differs from disk.

This dissertation investigates how several data-centric workloads interact with future storage technologies, the relevant software and algorithms, and in some instances computer hardware. Specifically, I consider analytics (commonly Decision Support Systems – DSS – popular in "Big Data") and On-Line Transaction Processing (OLTP). Both workload classes have been optimized to surmount disk's constraints, yet storage devices often remain the performance bottleneck and dominant cost. I match each workload to the emerging storage technology that suits it best and address specific opportunities or deficiencies in the software and hardware systems.

1.1 Analytics

Analytics relies on disk to provide enormous data capacity. Typical analytics work-flow involves taking a snapshot of data from an online database and mining this data for com-

plex, yet useful, patterns. While applications do not rely on disk's durability for recovery (in fact, instances that fit in main memory have no need for disk), modern analytics data sets reach peta-byte scale [1], and accessing such large data imposes the dominant bottleneck. Such capacity is currently achieved only by dense disk and Flash memory.

Decades of research have provided modern analytics databases with tools to minimize storage accesses, particularly slow random accesses (e.g., disk-specific indexes, join algorithms to minimize page access and produce large sequential runs). Whereas these optimizations are still effective for Flash, they fail to leverage Flash's ability to quickly read non-sequential data (many optimizations purposefully avoid random access patterns on disk). As examples, I consider access paths (various scan types) and join algorithms. An historic rule of thumb for scans is that an index should be used when less than 10% of rows in a table are returned, otherwise the entire table should be scanned [90]. The intuition is that locating rows from an index requires random reads as well as reading additional pages from the index itself. At sufficiently high selectivities, accessing the entire table, scanning all rows and returning those that satisfy the query, provides a faster access path. One would expect this selectivity (10%) to increase when replacing disk with Flash – Flash is no longer penalized by random reads, preferring any scan that minimizes total page accesses. Similarly, different ad hoc join algorithms (those that do not use indexes: block-nested loops, sort-merge join, and hybrid-hash join) present different storage access patterns and may be variably suited to disk and Flash. These algorithms and query optimization are further discussed in Section 2.2.1.

My results, originally presented in ADMS 2010 [85] and discussed in Chapter III, show that while both previous hypotheses are correct, their significance is negligible. Optimal access path (index vs. table scan) only changes between disk and Flash for a small range of query selectivities, and queries within that range see only a small performance improvement. Additionally, join algorithm choice makes little difference, as optimized join algorithms already minimize storage accesses, regardless of access pattern—join algorithms optimized for disk are already optimized for Flash. I conclude that the page-oriented nature of Flash limits further analytics-Flash optimization. On the other hand, emerging byte-addressable NVRAMs offer finer-grained access. However, analytics does not require persistent storage, instead using NVRAM as a replacement for DRAM. As DRAM-resident analytics techniques are already well established, I instead investigate using NVRAM persistence specifically to provide failure recovery, supporting durable transactions.

1.2 Transaction Processing

Databases have been designed for decades to provide high-throughput transaction processing with disk. Write Ahead Logging (WAL) techniques, such as ARIES [71], transform random writes into sequential writes and minimize transactions' dependences on disk accesses. Section 2.3 outlines modern recovery management, focusing on ARIES. With sufficient device throughput (IOPS) and read-buffering, databases can be made compute-bound and recover near-instantly. NVRAMs provide this massive storage throughput to the masses.

Whereas ARIES is necessary for disk, it presents only unnecessary software overheads to NVRAM. I show that removing ARIES improves transaction throughput by alleviating software bottlenecks inherent in centralized logging. Instead, NVRAM allows data to be updated in-place, enforcing data persistence immediately and providing correct recovery via transaction-local undo logs.

NVRAMs, however, are not without their limitations. Several candidate NVRAM technologies exhibit larger read latency and significantly larger write latency compared to DRAM [14]. Additionally, whereas DRAM writes benefit from caching and typically are not on applications' critical paths, NVRAM writes must become persistent in a constrained order to ensure correct recovery. I consider an NVRAM access model where correct ordering of persistent writes is enforced via *persist barriers*, which stall until preceding NVRAM writes complete; such persist barriers introduce substantial delays when NVRAM writes are slow.

To address these challenges I investigate accelerating NVRAM reads with various cache architectures and capacities, and avoid persist barrier delays by introducing a new recovery mechanism, *NVRAM Group Commit*. Database designs are discussed in Chapter IV. As expected, low latency memory-bus-connected NVRAM needs little additional caching (on-chip caches suffice) and updating data in-place is a simple and viable recovery strategy. However, long latency NVRAM and complex interconnects (e.g., Non-Uniform Memory Architectures – NUMA, PCIe-attached NVRAM, or distributed storage) benefit from DRAM caching and *NVRAM Group Commit*, improving throughput. I investigate specifically how NVRAM read and persist barrier latencies drive OLTP system design. These results and additional evaluations are presented in Chapter V. This work was originally published at VLDB [84].

1.3 Memory Persistency

The previous work looks at how OLTP recovery mechanisms should be designed, considering only the average delay incurred by persist barriers. The final portion of my dissertation investigates specific programming interfaces to order NVRAM writes. Whereas existing memory consistency models provide control over the order and visibility of volatile memory reads and writes across threads, there are no equivalent models to reason about data persistence. Memory consistency may be relaxed, allowing communicating threads to each observe different memory read and write orders. Such memory consistency models improve performance, but require complex reasoning and additional programming mechanisms (memory barriers) to ensure expected behavior. Memory models are described in Section 2.4.

Similarly, NVRAM write order may be relaxed, improving performance by allowing writes to occur in parallel or out of order while still ensuring correct recovery. I introduce *memory persistency*, a framework that extends memory consistency, to reason about NVRAM write order. Relaxed memory persistency models use persist barriers to enforce specific write orders, guaranteeing that data is correctly recovered after failure. I define memory persistency and enumerate the possible design space in Chapter VI. Interestingly, memory persistency models may be de-coupled from the underlying memory consistency models, separately enforcing the order in which writes become durable and the order in which writes become visible to other threads. I introduce a persistent queue benchmark and several memory persistency models in Chapter VII. Finally, I evaluate these models in Chapter VIII. Strict persistency models slow execution nearly 30× relative to existing throughput on volatile, nonrecoverable systems. Relaxed persistency models regain lost throughput by improving NVRAM write concurrency.

1.4 Data Center Infrastructure

The themes of this dissertation include performance, cost efficiency, and reliability. While I focus on storage architectures, I additionally published work regarding the cost and reliability of data center infrastructure. Appendix A contains "Understanding and Abstracting Total Data Center Power," published at WEED 2009 [82]. This work presents power/energy models for all aspects of the data center, including power distribution, battery backups, cooling infrastructure, and IT equipment. Appendix B contains "PowerRouting: Dynamic Power Provisioning for the Data Center," published at ASPLOS 2010 [83]. PowerRouting spreads power distribution responsibility throughout the data center to mini-

mize installed power infrastructure capacity while maintaining reliability, minimizing data center cost. The key insight is that data centers typically over-provision infrastructure, resulting in under-utilized (and often unnecessary) equipment. PowerRouting leverages compute-specific knowledge of the IT workload to more effectively utilize power infrastructure. Both of these works are included without modification.

During my investigations I discovered that, in many regards, industry is ahead of academia at decreasing operating costs and improving infrastructure efficiency. As such, the opportunity to contribute meaningful new techniques to improve infrastructure is rapidly diminishing. Recognizing storage and memory as primary concerns for energy efficiency, reliability, and cost, I focus on new and emerging storage technologies in this dissertation.

1.5 Summary

This dissertation investigates new techniques for accelerating data access for new NVRAM storage technologies, and is organized as follows: Chapter II contains background information on storage technologies, database optimizations, and memory consistency. This background forms the foundation of the work that follows. Chapter III considers taking advantage of Flash's fast random reads to accelerate database analytics. In Chapter IV, I describe potential software designs for OLTP using NVRAM. Chapter V details a methodology to evaluate NVRAM (devices not readily available) on modern hardware and evaluates several aspects of OLTP running on NVRAM. Chapter VI introduces and defines memory persistency. Specific memory persistency models and a persistent queue data structure are proposed in Chapter VII. Finally, I evaluate these memory persistency models in Chapter VIII.

CHAPTER II

Background

This chapter provides details necessary to understand the investigations and experiments in this dissertation. I focus on storage technologies, database analytics optimization, database transaction processing optimizations, and memory consistency models. The purpose of discussing database optimizations is to understand the complications that arise when using disk, and how resulting mechanisms interact with new storage technologies. Additionally, memory consistency forms the foundation for memory persistency.

2.1 Storage Technologies

Many technologies provide storage persistent storage, including disk, Flash, and upcoming NVRAM. For each, I outline the operating principles and important technological trends.

2.1.1 Disk

Disk has been the primary durable and high capacity storage technology for decades. Disk functions by storing data on spinning magnetic platters. Accessing data requires moving the *hard disk head* onto the proper *track*. Once the head settles, it may read or write

	Disk	Flash
Model	WD VelociRaptor 10Krpm	OCZ RevoDrive
Capacity	300gb	120gb
Price	$164	$300
Random Read	10ms	90µs
Seq. Read	120mb/s	190mb/s

Table 2.1: Storage characteristics.

data as the platter rotates and the correct sector within the track reaches the head. Disk capacity increases with the areal size and number of platters, as well as areal density (placing more sectors and tracks within the same area). Because capacity has scaled so well (and continues to), disk remains the lowest cost technology for large datasets and persistent storage.

While popular and inexpensive, disk suffers relatively slow access and undesirable access behavior [97]. Rotational speed limits the rate that data transfers to or from the disk. Further, random reads and writes must first seek to the proper track and then wait until the correct sector reaches the head, a process which takes several ms. Table 2.1 lists the measured performance characteristics of an enterprise disk (2011). This disk achieves nearly 120 MB/s sequential transfers, but random reads take an average of 10ms (only 50 KB/s for 512-byte sectors). As a result, there is a large history of optimization for disk-resident storage, as I discuss in Section 2.3.

2.1.2 Flash Memory

Driven by the popularity of mobile devices, Flash memory has quickly improved in both storage density and cost to the point where it has become a viable alternative for durable storage in enterprise systems. Unlike conventional rotating hard disks, which store data using magnetic materials, Flash stores charge on a floating-gate transistor forming a memory cell. These transistors are arranged in arrays resembling NAND logic gates, after which the "NAND Flash" technology is named. This layout gives NAND Flash a high storage density relative to other memory technologies. Though dense, the layout allows only coarse-grained access and sacrifices read latency—an entire array (a.k.a. page, typically 2KB to 4KB) must be read in a single operation—making NAND Flash more appropriate for block-oriented IO than as a direct replacement for RAM.

One of the difficulties of Flash devices is that a cell can be more easily programmed (by adding electrons to the floating gate) than erased (removing these electrons). Erase operations require both greater energy and latency, and typically can be applied only at coarse granularity (e.g., over blocks of 128KB to 512KB). Moreover, repeated erase operations cause the Flash cell to wear out over time, limiting cell lifetime (e.g., to 10^5 to 10^6 writes [96]). Recent Flash devices further increase storage density by using several distinct charge values to represent multiple bits in a single cell at the cost of slower accesses and even shorter lifetimes.

A Flash-based SSD wraps an array of underlying Flash memory chips with a controller that manages capacity allocation, mapping, and wear leveling across the individual Flash devices. The controller mimics the interface of a conventional (e.g., SATA) hard drive,

allowing Flash SSDs to be drop-in replacements for conventional disks.

As previously noted, Flash SSDs provide substantially better performance than disks, particularly for random reads, but at higher cost [12]. Table 2.1 lists specifications of a typical Flash SSD as compared to a 10,000 RPM hard drive (2011). Though neither of these devices are the highest-performing available today, they are representative of the mid-range of their respective markets. The latency for a random read is over 100× better on the SSD than on the disk, while the sequential read bandwidth is 1.6× better. Unlike disks, where each random read incurs mechanical delays (disk head seek and rotational delays), on SSDs, a random read is nearly as fast as a sequential read.

2.1.3 NVRAM

Nonvolatile memories will soon be commonplace. Technology trends suggest that DRAM and Flash memory may cease to scale, requiring new dense memory technologies [58].

Memory technology characteristics. Numerous technologies offer durable byte-addressable access. Examples include phase change memory (PCM), where a chalcogenide glass is heated to produce varying electrical conductivities, and spin-transfer torque memory (STT-RAM), a magnetic memory that stores state in electron spin [14]. Storage capacity increases by storing more than two states per cell in Multi-level Cells (MLC) (e.g., four distinct resistivity levels provide storage of 2 bits per cell).

While it remains unclear which of these technologies will eventually gain traction, many share common characteristics. In particular, NVRAMs will likely provide somewhat higher access latency relative to DRAM. Furthermore, several technologies are expected to have asymmetric read-write latencies, where writing to the device may take several microseconds [88]. Write latency worsens with MLC, where slow, iterative writes are necessary to reliably write to cells.

Similarly to Flash, resistive NVRAM technologies suffer from limited write endurance; cells may be written reliably a limited number of times. While write endurance is an important consideration, proposed hardware mechanisms (e.g., Start-Gap [87]) are effective in distributing writes across cells, mitigating write endurance concerns.

NVRAM storage architectures. Future database systems may incorporate NVRAM in a variety of ways. At one extreme, NVRAM can be deployed as a disk or Flash SSD replacement. While safe, cost-effective, and backwards compatible, the traditional disk interface imposes overheads. Prior work demonstrates that file system and disk controller latencies dominate NVRAM access times [15]. Furthermore, block access negates advantages of fine-grained access.

Recent research proposes alternative interfaces for NVRAM. Caulfield *et al.* propose Moneta and Moneta Direct, a PCIe attached PCM device [16]. Unlike disk, Moneta Direct bypasses expensive system software and disk controller hardware to minimize access latency while still providing traditional file system semantics. However, Moneta retains a block interface. Condit *et al.* suggest that NVRAM connect directly to the memory bus, with additional hardware and software mechanisms providing file system access and consistency [30]. I later adopt the same atomic eight-byte persistent write, enabling small, safe writes even in the face of failure. NVRAM will eventually connect via a memory interface, but it is unclear how storage technologies will evolve or what their exact performance characteristics will be. Instead of assuming specific NVRAM technologies and interconnects, I measure the affect that NVRAM performance (both latency and bandwidth) has on workload throughput.

2.2 Analytics Optimization

Large scale data processing requires efficient use of storage devices. Relational data is stored in tables (relations). Tables are a collection of rows (tuples), each row containing values for the different columns (attributes) defined on the table. I discuss two important operations common to the relational model that are expected to be affected by Flash's performance characteristics: scans and joins.

2.2.1 Scans

Each data access requires the query optimizer to choose access paths to each table. The goal is to select all relevant rows from the table while retrieving the fewest storage pages, frequently with the use of indexes. Work on access path selection dates back to the late 1970s [99].

There are two classic scan operators implemented by nearly all commercial DBMS systems: relation scan and index scan. An index is a database data structure that maps column values to rows within a table, supporting fast look-ups by that column. Indexes may be ordered (as in an in-memory balanced tree or disk-resident B-Tree) to efficiently retrieve all rows satisfying a range query (e.g., an ordered index on "last name" would accelerate a query asking for all people whose last name is between "Pelley" and "Wenisch"). Additionally, indexes may be clustered (the primary table is stored in sorted/hashed order and supports searching) or non-clustered (the index is separate from the primary store and contains references to rows). A table can only be physically ordered according to one key, limiting the use of clustered indexes. Database indexes are themselves stored on disk.

Many types of indexes exist, but all provide a more direct way to filter specific data than scanning an entire data set.

When no indexes are available, the only choice is to perform a *relation scan*, where all data pages in the table are read from disk and scanned tuple-by-tuple to select tuples that satisfy the query. When a relevant index is available, the DBMS may instead choose to perform an *index scan*, where the execution engine traverses the relevant portion of the index and fetches only pages containing selected tuples as needed. For clustered indexes (i.e., the row itself exists within the index, and all rows are sorted according to the index key), an index scan is nearly always the preferred access path, regardless of the underlying storage device. For non-clustered indexes, whether the optimizer should choose a relation scan or index scan depends on the selectivity of the query (expected number of rows that satisfy the query); relation scans have roughly constant cost regardless of selectivity (cost depends on table size), whereas index scan costs grow approximately linearly with selectivity. When selectivity is low, the index scan provides greater performance because it minimizes the total amount of data that must be transferred from disk. However, as selectivity increases, the fixed-cost (and simpler) relation scan becomes the faster option. Though the relation scan reads the entire table, it can do so using sequential rather than random IO, leveraging the better sequential IO performance of rotating hard disks. A classic rule of thumb for access path selection is to choose a relation scan once selectivity exceeds ten percent [90].

Recent databases implement a third, hybrid scan operator, which I call *rowid-sort scan*. In this scan operator, the unclustered index is scanned to identify relevant tuples. However, rather than immediately fetching the underlying data pages, the rowid (a unique identifier that signifies the row's page and offset within that page) of each tuple is stored in a temporary table, which is then sorted at the end of the index scan. Then, the pages identified in the temporary table are fetched in order, and relevant tuples are returned from the page. The rowid-sort scan has the advantage that each data page will be fetched from disk only once, even if multiple relevant tuples are located on the page. Rowid-sort scan is provided by IBM's DB2, although they may refer to it by a different name. Other databases use different terminology or algorithms to ensure that each heap page is fetched exactly once (for example, PostgreSQL uses a bitmap index, sorting the list of pages with tuples that satisfy the query [64]). Rowid-sort scan is the optimal access path for intermediate selectivities. Section 3.3 investigates choosing an optimal access path depending on whether data resides on disk or Flash.

2.2.2 Joins

One of the most important aspects of query optimization is choosing appropriate join algorithms. The development of join algorithms and optimization strategies dates back over 30 years [99, 101]. Most commercial DBMS systems implement variants of at least three join algorithms: nested-loop join, sort-merge join, and hybrid hash join. At a high level, the nested-loop join iterates over the inner relation for each tuple of the outer relation; the sort-merge join sorts both relations and then performs concurrent scans of the sorted results; and the hybrid hash join forms in-memory hash tables of partitions of the inner relation and then probes these with tuples from the outer relation.

The relative performance of these algorithms depends on a complex interplay of memory capacity, relation sizes, and the relative costs of random and sequential IOs. One example performance model that captures this interplay was proposed by Haas *et al.* [50]. Their model estimates the number of disk seeks and the size of each data transfer and weights each by a cost based on assumed characteristics of the IO device. The model further identifies the optimal buffering strategy for the various phases of each join algorithm. Seek and random/sequential transfer times are central parameters of this model, suggesting that new technologies require new device-specific query optimization.

As accessing large amounts of data on disk can limit system throughput, it is imperative that the query optimizer choose the best query plan. Typical query optimizers use data statistics to approximate query selectivity and cardinality as well as physically-based models to estimate the runtime of candidate query plans. Sections 3.3 and 3.4 will investigate when the query plan changes between disk and Flash, and what performance is lost when the incorrect decision is made (i.e., when assuming disk but actually using Flash).

2.3 Durable and Recoverable Transactions

Database applications typically provide transaction semantics described as ACID (Atomic, Consistent, Isolated, Durable) [48]. While the first three are primarily impacted by the database's concurrency control mechanisms within main memory, transaction durability and database recovery interacts with persistent storage. The goal of recovery management is to ensure that during normal transaction processing no transaction reports that it has committed and is later lost after a system failure, and that any transaction that fails to commit before a failure is completely removed (i.e., no partial updates remain). Additionally, recovery should occur as quickly as possible and allow efficient forward processing. Several schemes provide correct, high performance recovery for disk. I describe ARIES [71], a popular Write Ahead Logging (WAL) system that provides atomic durable transactions.

ARIES. ARIES uses a two-level store (disk and volatile buffer cache) alongside a shared persistent log. The buffer cache is necessary to accelerate reads and defer writes to the disk. Transaction writes coalesce in the buffer cache while being durably recorded in the log as ordered entries describing page updates and logical modifications, transforming random writes into sequential writes.

The log improves disk write performance while simultaneously providing data recovery after failure. Transaction updates produce both redo and undo entries. Redo logs record actions performed on heap pages so that they can be replayed in the event data has not yet written to disk in-place upon failure. Undo logs provide roll back operations necessary to remove aborted and uncommitted transaction updates during recovery. The database occasionally places a checkpoint in the log, marking the oldest update within the database still volatile in the buffer cache (and therefore where recovery must begin). Recovery replays the redo log from the most recent checkpoint to the log's end, reproducing the state at failure in the buffer cache; ARIES "replays history," recovering the failed database. Afterwards, incomplete transactions are removed using the appropriate undo log entries.

While a centralized log orders all database updates, the software additionally enforces that log entries are durable before corresponding buffer cache pages may write back. Transactions commit by generating a commit log entry, which must necessarily become durable after the transaction's other log entries (since the log writes to disk in order). This process guarantees that no transaction commits, and no page writes back to disk, without a durable history of its modifications in the log.

Though complex, ARIES improves database performance with disk. First, log writes appear as sequential accesses to disk, maximizing device throughput. Additionally, aside from reads resulting from buffer cache misses, each transaction depends on device access only at commit to flush log entries. All disk writes to the heap may be done at a later time, off of transactions' critical paths. In this way ARIES is designed from the ground up to minimize the effect of large disk access latencies.

2.4 Memory consistency models

This section provides a background on memory consistency models, outlining four models: Sequential Consistency (SC), Total Store Order (TSO), Relaxed Memory Order (RMO), and Release Consistency (RC). Much of this discussion assumes that caches are completely coherent—that is, any two accesses to a cache line (by any core/thread) have a total order. Release Consistency may guarantee this property only for properly annotated programs. Other systems enforce cache coherence in hardware without annotation.

Consistency models define the order of loads and stores observed by threads. While every thread observes its own execution in program order, it may appear that other threads execute out of order. Processors (and compilers) are generally free to reorder instructions to accelerate performance so long as they produce equivalent results assuming no shared memory accesses (single thread execution). However, loads and stores that are independent from a single-threaded point of view may in fact interact with other threads. Reordering these memory accesses often results in unintended program behavior.

Two popular solutions to this problem are to (1) force all threads to observe the loads and stores of other threads in a single globally defined order (SC) or (2) relax this guarantee, introducing memory barriers that allow the programmer to enforce a certain order when necessary (i.e., relax consistency). While relaxing consistency may provide higher performance, it places a greater burden on programmers to be aware of possible instruction reordering and correctly use memory barriers. Synchronization is often provided by software libraries, shielding the average programmer from the complexities of memory consistency.

The consistency model provides guarantees on observed memory orders that the programmer can expect and mechanisms to enforce additional constraints. Implementations may momentarily violate the consistency model so long as no program is able to observe the violation. For example, implementations are free to *speculate*, executing with relaxed consistency, and later determine whether consistency has been violated. When consistency is violated the implementation must roll back and re-execute memory instructions, providing the illusion that threads execute with strict consistency.

Sequential Consistency. Sequential Consistency [57] provides the most intuitive programming model, yet necessarily the worst performance (although modern techniques use speculation to improve performance). All loads and stores appear in a globally consistent order that is an interleaving of program order from all threads. A popular view of SC is that memory switches between the various processors; only a single processor accesses memory at any single time. Additionally, many architectures define atomic Read-Modify-Write (RMW; e.g., compare-and-swap, atomic add) operations that perform several operations atomically. The programmer need not consider memory instructions reordering and therefore barriers are unnecessary (compiler barriers must still be used).

Total Store Order. Total Store Order [103] provides greater performance than SC at the cost of requiring the programmer to insert memory barriers. Most memory operations are still observed to occur in program order from the perspective of other threads: (1) stores may not reorder with other stores, (2) loads may not reorder with other loads, and (3) a store that occurs after a load may not reorder and appear to bypass that load. However,

loads that occur after a store in program order may reorder and bypass the store, appearing to occur before the store from the perspective of other processors. The rationale for doing so is that load execution is often on the application's critical path—executing a load only after all previous stores are visible additionally slows execution. Executing loads as soon as possible (before prior stores become visible) improves performance.

The programmer is responsible for recognizing any code where allowing a load to bypass a store allows incorrect behavior. A barrier is provided by the architecture for such cases to force loads to delay until previous stores become visible to other threads. Additionally, RMW instructions act as barriers, preventing memory operations from reordering around the RMW instruction. As concurrent programming commonly uses RMW operations, explicit memory barriers are rarely required in practice.

Relaxed Memory Order. Relaxed Memory Order [103] further relaxes consistency, allowing stores and loads to reorder around each other. To enforce the intended behavior the programmer must use memory barriers to constrain memory operation visibility. RMO barriers constrain (1) loads from reordering with other loads, (2) stores from reordering with other stores, (3) loads from bypassing stores, and (4) stores from bypassing loads. These constraints are enforced separately and explicitly. RMO requires frequent barriers for correct thread communication and is generally considered more difficult to program than SC and TSO.

Release Consistency. Release Consistency [44] provides a relaxed consistency model with two instruction instruction—acquire and release. In the absence of these labels threads may observe memory operations in any order and may read stale data (loading an old value when some other processor has recently written to the same address). Reading shared memory requires an *acquire* label. Similarly, after writing to shared memory a *release* label is used to make the updates visible to other processors; acquire and release used as a pair ensure proper synchronization between communicating threads. Additionally, acquire and release labels prevent memory operations from reordering (for example, to ensure that operations in a critical section occur only after the lock is acquired and before the lock is released). RMW operations implicitly contain both acquire and release labels.

Release Consistency improves performance by (1) allowing maximum instruction reordering for unannotated single threaded code, (2) providing precise synchronization, introducing the minimal number of ordering constraints to provide the intended behavior, and (3) enforcing cache coherence only at acquire and release labels, allowing simpler cache coherence hardware. However, release consistency requires the programmer to be aware of and understand acquire and release labels.

A similar trade-off exists for enforcing data persistence—the order in which data be-

comes persistent may be relaxed and deviate from program order stores. Removing NVRAM write constraints improves performance, yet mechanisms must be provided to enforce expected behavior. The interaction between persistence, performance, and ease of programming is explored in Chapter VI.

2.5 Conclusion

This chapter provides a brief history and outline of storage devices, database technologies, and memory consistency. The remainder of my dissertation builds on these ideas to explore how best to use NVRAM in future storage systems.

CHAPTER III

SSD-aware Query Optimization

This chapter asks if database query optimizers must be SSD-aware. The work was published in the Second International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (AMDS 2011), collocated with VLDB [85]. I worked on this project with my advisor, Thomas F. Wenisch, and assistant professor Kristen LeFevre. Refer to Sections 2.1 and 2.2 for background on storage technologies and query optimization.

3.1 Introduction

For decades, database management systems (DBMSs) have used rotating magnetic disks to provide durable storage. Though inexpensive, disks are slow, particularly for non-sequential access patterns due to high seek latencies. With the rapid improvements in storage density and decreasing price of Flash-based Solid State Disks (SSDs), DBMS administrators are beginning to supplant conventional rotating disks with SSDs for performance-critical data in myriad DBMS applications. Though SSDs are often significantly more expensive than conventional disks (in terms of $ per GB), they provide modest (2×) improvements in sequential IO and drastic (over 100×) improvements for random IO, closing the gap between these access patterns.

Many components of modern DBMSs have been designed to work around the adverse performance characteristics of disks (e.g., page-based buffer pool management, B-tree indexes, advanced join algorithms, query optimization to avoid non-sequential IO, prefetching, and aggressive IO request reordering). As SSDs present substantially different performance trade-offs, over the past few years researchers have begun to examine how SSDs are best deployed for a variety of storage applications [13, 23], including DBMSs [59, 120, 62, 8, 110]. A common theme among these studies is to leverage the better random IO performance of SSDs through radical redesigns of index structures [120, 62] and

data layouts [8, 110]. However, even within the confines of conventional storage management and indexing schemes in commercial DBMSs, there may be substantial opportunity to improve query optimization by making it SSD-aware.

In this chapter, I examine the implications that moving a database from disk to Flash SSDs will have for query optimization in conventional commercial DBMSs. I focus on optimization of read-only queries (e.g., as are common in analytics decision support workloads) as these operations are less sensitive to the SSD adoption barriers identified in prior work, such as poor SSD write/erase performance [23] and write endurance [96]. Conventional query optimizers assume a storage cost model where sequential IOs are far less costly than random IOs, and select access paths and join algorithms based on this assumption. The recent literature [8] suggests that on SSDs, optimizers should instead favor access paths using non-clustered indexes more frequently; SSDs will favor retrieving more rows via an index if it reduces the number of pages accessed, even if it increases random IO accesses. Furthermore, as SSDs change the relative costs of computation, sequential, and random IO, the relative performance of alternative join algorithms should be re-examined, and optimizer cost models updated.

My original intent was to leverage this idea to 1) improve query optimization by making device-specific decisions, and 2) accelerate database analytics and reduce operating costs by intelligently placing data on different devices. While Flash is faster than disk, it is generally more expensive per equal capacity. It makes sense to place small amounts of frequently accessed data on Flash, and the rest on disk. Furthermore some data "prefers" to live on Flash as it is naturally accessed in ways that Flash holds a comparative advantage over disk (e.g., queries typically access this data in selectivities or join types that prefer Flash).

Despite this intuition, my empirical investigation using a commercial DBMS finds *it is not necessary to make any adjustments to the query optimizer when moving data from disk to Flash*—an SSD-oblivious optimizer generally makes effective choices. I demonstrate this result, and explore why it is the case, in two steps.

First, I analyze the performance of scan operators. Classic rules of thumb suggest that non-clustered index scans are preferable at low selectivity (i.e., below 10%), whereas a relation scan is faster at high selectivity, because it can leverage sequential IOs. Therefore, the optimizer should prefer index scans at much higher selectivities on SSDs. I demonstrate analytically and empirically that this intuition is false—the range of selectivities for which an index scan operation can benefit from SSDs' fast random reads is so narrow that it is inconsequential in practice.

Second, I measure the relative performance of hybrid hash and sort-merge joins on disk

and Flash. My results indicate that the performance variation between the join algorithms is typically smaller (and often negligible) on Flash, and is dwarfed by the 5× to 6× performance boost of shifting data from disk to SSD. I conclude that because commercial DBMSs have been so heavily optimized to hide the long access latencies of disks (e.g., through sophisticated prefetching and buffering), they are largely insensitive to the latency improvements available on SSDs. Overall, the small and inconsistent performance gains available by making query optimizers SSD-aware are not worth the effort.

3.2 Methodology

The objective of this empirical study is to contrast the performance of alternative scan and join algorithms for the same queries to discover whether the optimal choice of access path or join algorithm differs between SSDs and conventional disks. For either storage device, the optimal access path depends on the selectivity of the selection predicate(s). The optimal join algorithm depends on several factors: the sizes of the inner and outer relations, the selectivity and projectivity of the query, the availability of indexes, and the available memory capacity. The goal is to determine whether the regions of the parameter space where one algorithm should be preferred over another differ substantially between SSD and disk because of the much better random read performance of the SSD. In other words, I am trying to discover, empirically, cases where an access path or join algorithm that is an appropriate choice for disk results in substantially sub-optimal performance on an SSD, suggesting that the optimizer must be SSD-aware.

I carry out this empirical investigation using IBM DB2 Enterprise Server Edition version 9.7. Experiments use the Wisconsin Benchmark schema [10] to provide a simple, well-documented dataset on which to perform scans and joins. Though this benchmark does not represent a particular real-world application, modeling a full application is not my intent. Rather, the Wisconsin Benchmark's uniformly distributed fields allow precise control over query selectivity. Whereas real world queries are more complicated than the simple scans and joins studied here, these simple microbenchmarks reveal the underlying differences between the storage devices and scan/join algorithms most clearly. Alternative "real world" benchmarks (namely TPC-H [109]) complicate matters, making it difficult to discern why different query plans prefer disk or SSD. I aggregate results of all queries to avoid materializing output tables as I am primarily interested in isolating other database operations; typically all results must be materialized regardless of device, so I remove this step to provide a better comparison. I run queries on a Pentium Core Duo with 2GB main memory, a 7200 RPM root disk drive, and the conventional and SSD database disks described

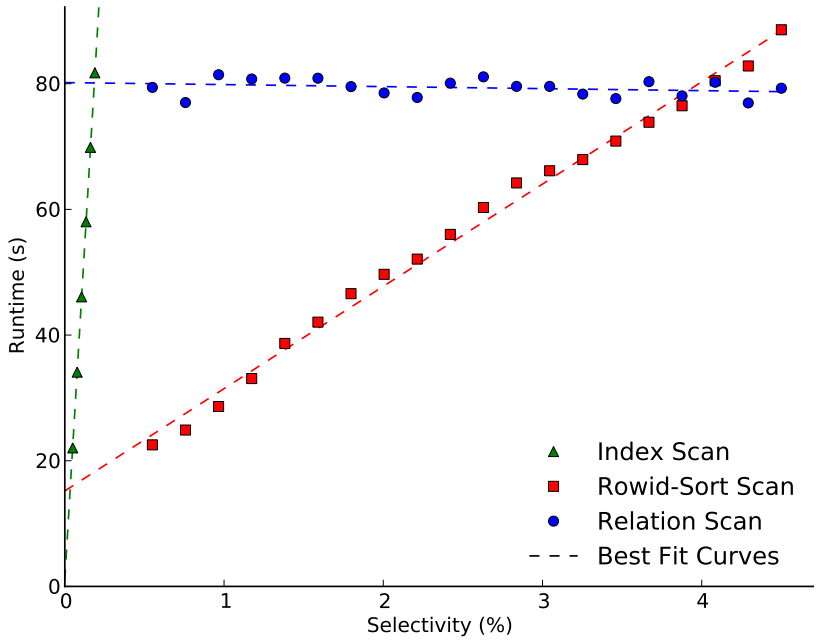


Figure 3.1: Scan operator performance on Disk. Relation scan outperforms the alternatives at selectivities above 4%, while index scan is optimal only for vanishingly small selectivities (e.g., single-tuple queries). Best fit curves drawn for convenience.

in Table 2.1. Both the hard disk and SSD were new at the beginning of the experiments. While other work has shown that SSD performance may degrade over the lifetime of the device I did not observe any change in performance.

Note that I am not concerned with the optimization decisions that DB2 presently makes for either disk or SSD; rather, I am seeking to determine the ground truth of which algorithm a correct optimizer should prefer for each storage device. In general, multiple query plans are achieved by varying the optimizers inputs—disk random and sequential access latencies. Databases often have planner hints to select specific plans (for example, DB2's optimization profiles).

3.3 Scan Analysis

I now turn to the empirical and analytic study of scan operators. I demonstrate that although the expectations outlined in Section 3.1 are correct in principle, the range of selectivities for which an index scan operation benefits from SSDs' fast random reads is so narrow that it is inconsequential in practice.

Empirical results. I compare the measured performance of the different scan operators as a function of selectivity on SSD and disk. The objective is to find the break-even points where the optimal scan operator shifts from index scan to rowid-sort scan and finally

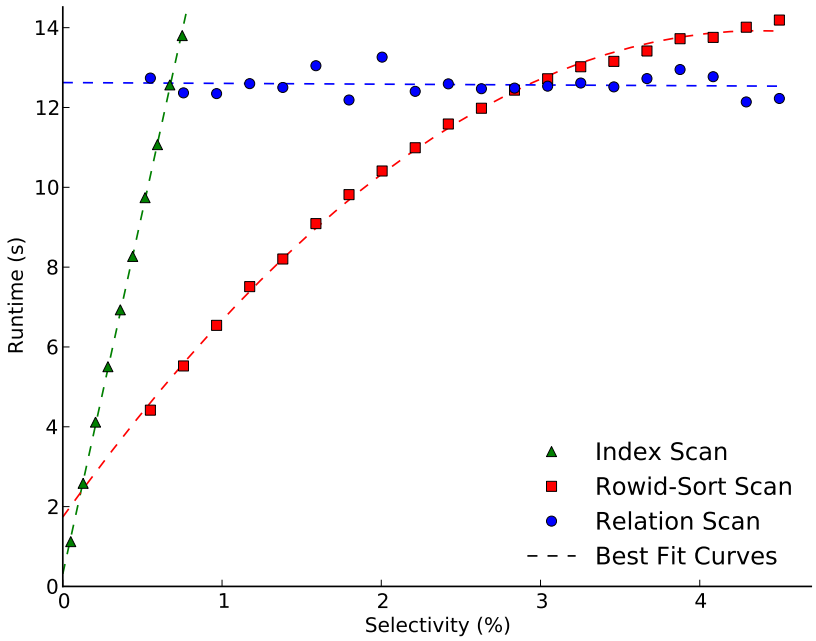


Figure 3.2: Scan operator performance on Flash SSD. Though both break-even points shift as intuition suggests, the selectivities where the optimal decision differs between Disk and SSD are so narrow that the difference is inconsequential in practice. Best fit curves drawn for convenience.

to relation scan on each device, and the performance impact in regions where this decision differs. I issue queries for ranges of tuples using a uniformly distributed integer field on a table with 10 million rows, or roughly 2 GB. I use a pipelined aggregation function to ensure that no output table is materialized.

Figures 3.1 and 3.2 report scan runtime on disk and Flash SSD, respectively. The figures show the measured runtime of each scan (in seconds); lower is better. Recall from Section 3.1 that classic rules of thumb suggest that, on disk, the break-even point between index and relation scan should occur near 10% selectivity, and intuition suggests an even higher break-even point for SSD. Clearly, the conventional wisdom is flawed even for rotating disks; relation scan dominates above selectivities of just 4% (the trends shown in the figure continue to the right). In the intermediate range from about 0.1% to 4% selectivity the rowid-sort scan performs best.

However, the more important analysis is to compare the locations of the break-even points across SSD and disk. Both crossover points shift in the expected directions. The slope of the index scan curve is considerably shallower, and the break-even with the relation scan shifts above 0.5% selectivity. Furthermore, the range in which rowid-sort scan is optimal becomes narrower. Nevertheless, the key take-away is that the range of selectivities for which the optimal scan *differs* across SSD and Disk is vanishingly small. Only a minute

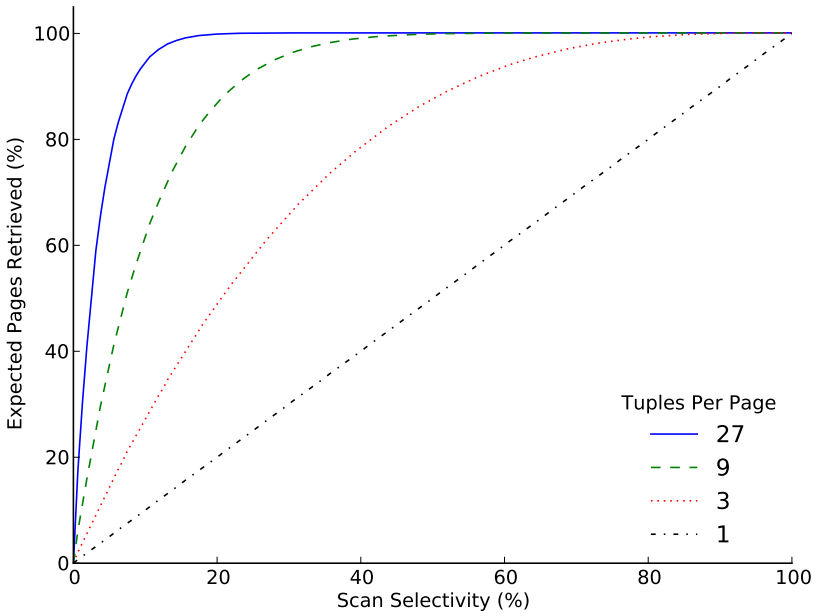


Figure 3.3: Expected page accesses. Index scans touch the majority of pages even at low selectivities.

fraction of queries fit into this range, and queries that make the incorrect decision (between disk and Flash) see a small performance impact. Hence, it is unnecessary for the optimizer to be SSD-aware to choose an effective scan operation.

Whereas these measurements demonstrate my main result, they do not explain why index scans fail to leverage the random access advantage of Flash. I turn to this question next.

Analytic results. The previous results show that index scan underperforms at selectivities far below what the classic 10% rule of thumb suggests. The flaw in the conventional wisdom is that, when there are many tuples per page, the vast majority of *pages* need to be retrieved even if only a few *tuples* are accessed. When the 10% rule is applied to page- rather than tuple-selectivity, the guideline is more reasonable. Yue *et al.* provide an analytical formula for the expected number of pages retrieved given the size of the table, tuples per page, and selectivity [121], assuming tuples are randomly distributed among pages (a reasonable assumption given that each table can be clustered on only a single key). Based on this formula, Figure 3.3 shows the expected percentage of pages retrieved as a function of query selectivity and tuples per page. When a page contains only a single tuple, clearly, the number of tuples and pages accessed are equal. However, as the number of tuples per page increases, the expectation on the number of pages that must be retrieved quickly approaches 100% even at small selectivities. As a point of reference, given a 4kb page size and neglecting page headers, the Wisconsin Benchmark stores 19 tuples per page while

TPC-H's Lineitem and Orders tables store 29 and 30 tuples per page, respectively.

The implication of this result is that, for typical tuple sizes, the vast majority of pages in a relation must be read even if the selectivity is but a few percent. Hence, with the exception of single-tuple lookups, there are few real-world scenarios where scan performance improves with better random access latency under conventional storage managers that access data in large blocks. To benefit from low access latency, future devices will need to provide random access at tuple (rather than page) granularity. Until such devices are available, relation and rowid-sort scans will dominate, with IO bandwidth primarily determining scan performance.

3.4 Join Analysis

I next study the variability in join performance across disk and Flash SSD. Again, the objective is to identify cases where the optimal join algorithm for disk consistently results in grossly sub-optimal performance on Flash SSD. Such scenarios imply that it is important for the optimizer to be SSD aware.

DB2 implements nested loop, sort-merge, and hybrid hash join operators. However, DB2 does not support a block nested loop join; its nested loop join performs the join tuple-by-tuple instead of prefetching pages or other blocks, relying on indexes to provide high performance. Hence, unless the join can be performed in memory, the nested loop grossly underperforms the other two algorithms for ad-hoc queries (those that do not use indexes) regardless of storage device and will not be selected by the query optimizer unless it is the only alternative (e.g., for inequality joins). I therefore restrict the investigation to a comparison of sort-merge and hybrid hash joins.

When a clustered index exists for a particular scan or join this index should almost always be used, regardless of the nature of the storage device. Hence, I do not include clustered indexes in my analysis. Furthermore, I evaluate only ad hoc joins. When indexes are available, the choice of whether or not to use the index is analogous to the choice of which scan operator to use for a simple select query, which is covered by the previous analysis of scans.

Because of the complex interplay between available memory capacity and relation sizes for join optimization [50], I do not have a specific expectation that one join algorithm will universally outperform another on Flash SSD as opposed to disk. Rather, I perform a cross-product of experiments over a spectrum of relation sizes and output projectivities using the Wisconsin Benchmark database. Haas's model demonstrates the importance of the relative sizes of input relations and main memory capacity on join performance; hence

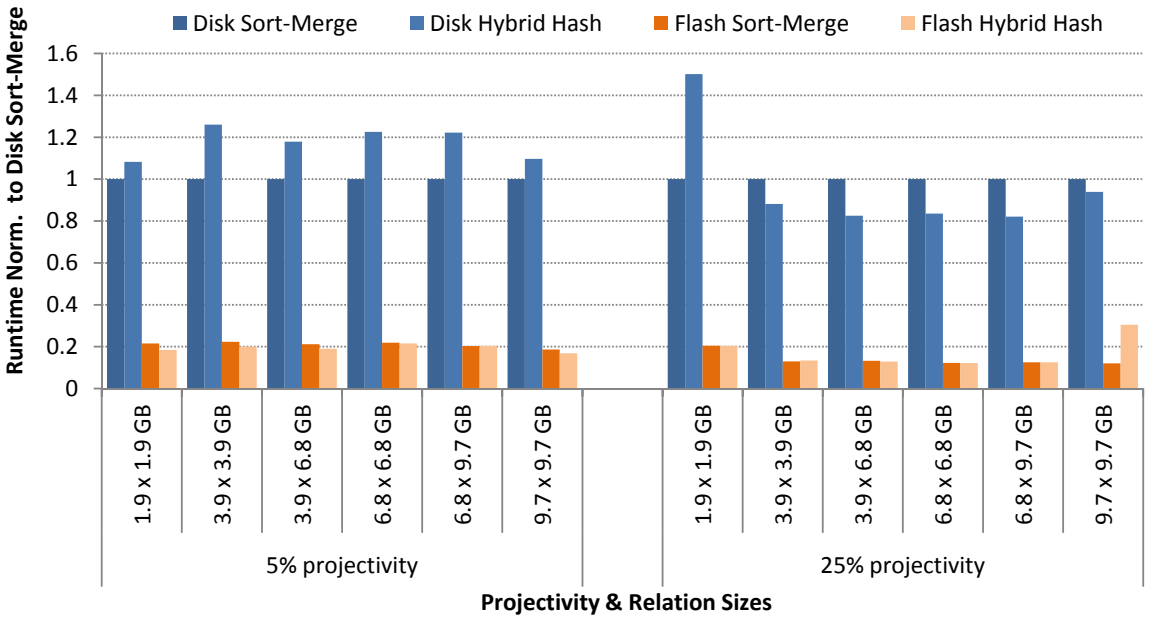


Figure 3.4: Join performance. Join runtimes on Flash SSD and Disk, normalized for each join to the runtime of sort-merge on disk. Though there is significant variability in join algorithm performance on disk, performance variability on SSD is dwarfed by the 6× performance advantage of moving data from disk to SSD.

I explore a range of joins that are only slightly larger than available memory (joining two 1.9GB tables) to those that are an order of magnitude larger (joining two 9.7GB tables). I vary projectivity, having discovered empirically that it significantly impacts the optimal join algorithm on disk, as it has a strong influence on partition size in hybrid hash joins. I execute queries with two projectivities: approximately 5% (achieved by selecting all the integer fields in the Wisconsin Benchmark schema), and approximately 25% (selecting an integer field and one of the three strings in the schema). In all experiments, I perform an equijoin on an integer field, and use an aggregation operator to avoid materializing the output.

I report results in graphical form in Figure 3.4 and absolute run times in Table 3.1. In Figure 3.4, each group of bars shows the relative performance of sort-merge and hybrid hash joins on disk (darker bars) and Flash SSD (lighter bars), normalized to sort-merge performance on disk. Lower bars indicate higher performance. I provide the same data in tabular form to illustrate the runtime scaling trends with respect to relation size, which are obscured by the normalization in the graph.

Two critical results are immediately apparent from the graph. First, Flash SSDs typically outperform disk by 5× to 6× regardless of join algorithm, a margin that is substantially higher than the gap in sequential IO bandwidth, but far smaller than the gap in random IO

Projectivity	Table Sizes	Disk		Flash SSD	
		Sort-merge	Hybrid hash	Sort-merge	Hybrid hash
5%	1.9 × 1.9 GB	187	202	40	34
	3.9 × 3.9 GB	358	451	80	72
	3.9 × 6.8 GB	487	574	103	93
	6.8 × 6.8 GB	649	795	142	140
	6.8 × 9.7 GB	816	997	166	166
	9.7 × 9.7 GB	1084	1189	202	183
25%	1.9 × 1.9 GB	236	355	48	48
	3.9 × 3.9 GB	751	662	97	101
	3.9 × 6.8 GB	947	781	125	122
	6.8 × 6.8 GB	1415	1182	174	173
	6.8 × 9.7 GB	1581	1298	199	199
	9.7 × 9.7 GB	2081	1955	250	634

Table 3.1: Absolute join performance. Join runtimes in seconds. Variability in join runtimes is far lower on Flash SSD than on Disk.

bandwidth (see Table 2.1). Hence, though both join algorithms benefit from the improved random IO performance of SSDs, the benefit is muted compared to the 100× device-level potential. Second, whereas there is significant performance variability between the join algorithms on disk (typically over 20%), with the exception of a single outlier, the variability is far smaller on Flash SSD (often less than 1%). From these results I conclude that although important on disk, the choice of sort-merge vs hybrid hash join on SSD leads to inconsequential performance differences relative to the drastic speedup of shifting data from disk to Flash. Hence, there is no compelling reason to make the query optimizer SSD-aware; the choice it makes assuming the performance characteristics of a disk will yield near-optimal performance on SSD.

I highlight two notable outliers in the results. On disk, the best join algorithm is strongly correlated to query projectivity with the exception of the 1.9GB × 1.9GB join at 25% projectivity. Because the required hash table size for this join is close to the main memory capacity, I believe that this performance aberration arises due to DB2 selecting poor partition sizes for the join. Second, on Flash, I observe a large performance difference (over 2×) between sort-merge and hybrid hash join for the largest test case, a 9.7GB × 9.7GB join at 25% projectivity. For this query, I observe a long CPU-bound period with negligible IO at the end of the hybrid hash join that does not occur for any of the other hash joins. Hence, I believe that this performance aberration is unrelated to the type of storage device, and may have arisen due to the methods employed to coax the optimizer to choose this join

algorithm. In any event, neither of these outliers outweigh the broader conclusion that there is no particular need for the query optimizer to be SSD aware.

3.5 Related Work

Previous work studying the applicability of Flash memory in DBMS applications has focused on characterizing Flash, benchmarking specific database operations on Flash, and designing new layouts, data structures, and algorithms for use with Flash.

Both Bouganim *et al.* and Chen *et al.* benchmark the performance of Flash for various IO access patterns [13, 23]. Bouganim introduces the uFLIP micro-benchmarks and tests their performance on several devices. Chen introduces another set of micro-benchmarks, concluding that poor random write performance poses a significant barrier to replacing conventional hard disks with Flash SSDs. While these micro-benchmarks are instructive for understanding database performance, I focus specifically on the performance of existing scan and join operators on SSD and disk. Others have also benchmarked Flash's performance within the context of DBMS systems. Lee *et al.* investigate the performance of specific database operations on Flash, including multiversion concurrency control (MVCC), external sort, and hashes [59]. Similar to my study, Do *et al.* benchmark ad hoc joins, testing the effects of buffer pool size and page size on performance for both disk and Flash [34]. Although related to this study, neither of these works look at the specific performance differences between disk and Flash for scans and joins and how this might impact query optimization.

Whereas the above works (and this study) focus on measuring the performance of existing databases and devices, others look ahead to redesign DBMS systems in light of the characteristics of Flash. Yin *et al.* and Li *et al.* present new index structures, focusing on maintaining performance while using sequential writes to update the index [120, 62]. Baumann *et al.* investigate Flash's performance alongside a hybrid row-column store referred to as "Grouping" [8]. Similarly, Tsirogiannis *et al.* use a column store motivated by the PAX layout to create faster scans and joins [110].

Interestingly, my findings contradict recommendations from many of these studies. Baumann concludes that SSDs shift optimal query execution towards index-based query plans. The study bases this conclusion on the observation that asynchronous random reads on Flash are nearly as fast as sequential reads. Indeed, the arguments made by Baumann are a key component of the intuition laid out in Section 3.1 that led me to expect a need for SSD-aware query optimization. However, the conclusion neglects the observations discussed in Section 3.3 demonstrating that queries selecting more than a handful of tuples

will likely retrieve the majority of pages in a relation, and thus gain no advantage from fast random IO. Tsirogiannis introduces a join algorithm that retrieves only the join columns, joins these values, and then retrieves projected rows via a temporary index. By the previous argument, scanning for projected data should retrieve the majority of data pages, preferring a relation scan, and thus provide comparable advantage on disk and SSD.

Finally, Bausch *et al.* implement asymmetry-aware query optimization in PostgreSQL [9]. They calibrate and evaluate their system using the TPC-H benchmark and find substantial improvement. However, I believe their calibration and evaluation are skewed and lead to false conclusions (their results are insufficient to demonstrate that query optimizers should be SSD-aware). First, their results show that an external sort of unordered data results results in 95% random read accesses to disk. This is indicative of poorly configured memory buffers; external soft algorithms should exhibit almost entirely sequential access patterns. Furthermore, the results of their calibration (shown in their appendix) differ substantially from expected physical traits. For example, the relative cost of disk's random access is only 6.8× that of a sequential access, the relative cost of Flash's random read access is 5.6× that of its sequential read access (nearly the same as the relative difference on disk), and Flash's sequential writes are roughly 2.5× *slower* than random writes (sequential writes should be faster). Using a physically-based model only makes sense if the model is accurate. I believe the improvement shown is a function of (1) using the total time of 21 queries as the performance and calibration metric instead of arithmetic or geometric mean (query 21 shows improvement of 549% and has a greater runtime than most other queries), (2) using search-based calibration instead of physically derived optimization parameters (e.g., the cost of a disk seek should be measured directly from the disk, not by fitting the model), and (3) the asymmetric model has seven parameters versus the original model's five; these additional degrees of freedom allow more effective search-based calibration even when using an incorrect optimization model. I believe that my work demonstrates fundamental principles suggesting that query optimization sees little benefit from SSD-awareness.

3.6 Conclusion

Flash-based solid state disks provide an exciting new high-performance alternative to disk drives for database applications. My investigation of SSD-aware query optimization was motivated by a hope that the drastically improved random IO performance on SSDs would result in a large shift in optimal query plans relative to existing optimizations. At a minimum, I expected that constants capturing relative IO costs in the optimizer would

require update. This chapter presented evidence that refutes this expectation, instead showing that an SSD-oblivious query optimizer is unlikely to make significant errors in choosing access paths or join algorithms. Specifically, I demonstrated both empirically and analytically that the range of selectivities for which a scan operation can benefit from SSDs' fast random reads is so narrow that it is inconsequential in practice. Moreover, measurements of alternative join algorithms reveal that their performance variability is far smaller on SSDs and is dwarfed by the 5× to 6× performance boost of shifting data to SSD. Overall, I conclude that the small and inconsistent performance gains available by making query optimizers SSD-aware are not worth the effort.

CHAPTER IV

Architecting Recovery Management for NVRAM

The following two chapters consider the impact of using emerging NVRAM technologies for durable transaction processing. Refer to Section 2.1.3 for an overview of storage technologies and Section 2.3 for a description of ARIES, a popular recovery mechanism for disk. The work presented in these two chapters was originally published in VLDB 2014 [84]. I completed this work under the guidance of my advisor, Thomas F. Wenisch, and collaborators at Oracle, Brian T. Gold and Bill Bridge. I would especially like the thank Brian and Bill for bringing industry's point of view and "real world" examples to this collaboration. This chapter outlines the problems with existing disk recovery management, the potential pitfalls of using NVRAM for persistent applications, and a description of several candidate software designs for NVRAM recovery management, to be evaluated in the next chapter.

4.1 Introduction

Emerging nonvolatile memory technologies (NVRAM) offer an alternative to disk that is persistent, provides read latency similar to DRAM, and is byte-addressable [14]. Such NVRAMs could revolutionize online transaction processing (OLTP), which today must employ sophisticated optimizations with substantial software overheads to overcome the long latency and poor random access performance of disk. Nevertheless, many candidate NVRAM technologies exhibit their own limitations, such as greater-than-DRAM latency, particularly for writes [58].

Prior work has already demonstrated the potential of NVRAM to enhance file systems [30] and persistent data structures [115], but has not considered durable, recoverable OLTP. Today, OLTP systems are designed from the ground up to circumvent disk's performance limitations while ensuring that data is properly recovered after system failure. For example, many popular database systems use Write-Ahead Logging (WAL; e.g., ARIES [71])

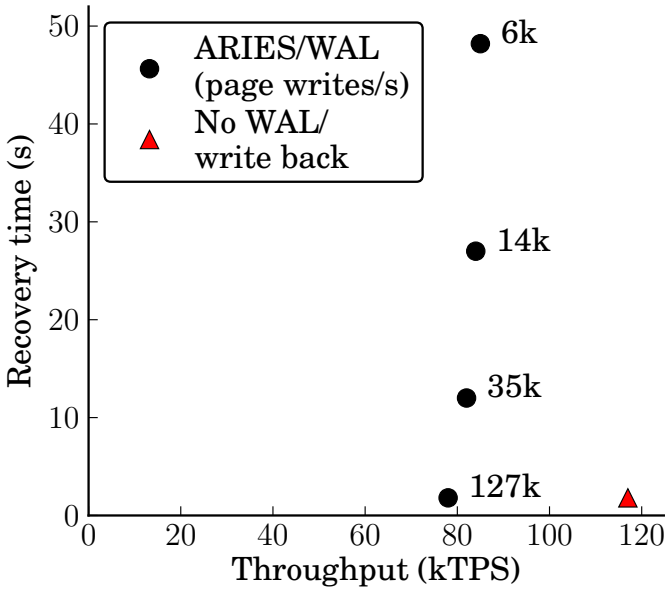


Figure 4.1: TPCB recovery latency vs throughput. Increasing page flush rate reduces recovery latency. Removing WAL entirely improves throughput by 50%.

to avoid expensive random disk writes by instead writing to a sequential log. Although effective at hiding write latency, WAL entails substantial software overheads.

NVRAM offers an opportunity to simultaneously improve database transaction processing throughput and recovery latency by rethinking mechanisms that were designed to address the limitations of disk. Figure 4.1 demonstrates this potential, displaying recovery time and transaction throughput for the TPCB workload running on the Shore-MT storage manager [54] for hypothetical NVRAM devices (see Section 5.1 for a description of the methodology).

The ARIES/WAL points (black circles) in the Figure show forward-processing throughput (horizontal axis) and recovery time (vertical axis) as a function of device write throughput (annotated alongside each point as pages per second). As database throughput can greatly outpace existing storage devices (this configuration requires 6,000 page writes/s to bound recovery; measured disk and Flash devices provide only 190 and 2,500 page writes/s, respectively) I model recovery performance under faster NVRAM using a RAM disk for log and heap while limiting the page flush rate. As intuition would suggest, greater write bandwidth enables more aggressive flushing, minimizing the number of dirtied pages in the buffer cache at the time of failure, in turn reducing recovery time. With enough write bandwidth (in this case, 127,000 flushes/s, or 0.97 GB/s random writes for 8KB pages) the database recovers near-instantly, but forward-processing performance remains compute bound. Achieving such throughput today requires large, expensive disk arrays or

enterprise Flash storage devices; future NVRAM devices might enable similar performance on commodity systems.

NVRAM opens up even more exciting opportunities for recovery management when considering re-architecting database software. Figure 4.1 shows this additional potential with a design point (red triangle) that removes WAL and asynchronous page flushing—optimizations primarily designed to hide disk latency. Throughput improves due to three effects: (1) threads previously occupied by page and log flushers become available to serve additional transactions, (2) asynchronous page flushing, which interferes with transactions as both flusher and transaction threads latch frequently accessed pages, is removed, and (3) transactions no longer insert WAL log entries, reducing the transaction code path. In aggregate these simplifications amount to a 50% throughput increase over ARIES's instantaneous-recovery NVRAM performance. The key take-away is that database optimizations long used for disk only hinder performance with faster devices. In this chapter, I investigate how to redesign durable storage and recovery management for OLTP to take advantage of the low latency and byte-addressability of NVRAM.

NVRAMs, however, are not without their limitations. Several candidate NVRAM technologies exhibit larger read latency and significantly larger write latency compared to DRAM. Additionally, whereas DRAM writes benefit from caching and typically are not on applications' critical paths, NVRAM writes must become persistent in a constrained order to ensure correct recovery. I consider an NVRAM access model where correct ordering of persistent writes is enforced via *persist barriers*, which stall until preceding NVRAM writes complete; such persist barriers can introduce substantial delays when NVRAM writes are slow.

This chapter outlines an approach to architecting recovery management for transaction processing using NVRAM technologies. I discuss potential performance concerns for both disk and NVRAM, proposing software designs to address these problems. In the next chapter I outline an NVRAM performance evaluation framework involving memory trace analysis and precise timing models to determine when OLTP must be redesigned.

4.2 Recovery Management Design

Upcoming NVRAM devices will undoubtedly be faster than both disk and Flash. However, compared to DRAM many NVRAM technologies impose slower reads and significantly slower persistent writes. Both must be considered in redesigning OLTP for NVRAM.

	NVRAM Disk-Replacement	*In-Place Updates*	*NVRAM Group Commit*
Software buffer	Traditional WAL/ARIES	Updates both buffer and NVRAM	Buffer limits batch size
Hardware buffer	Impractical	Slow uncached NVRAM reads	Requires hardware support
Replicate to DRAM	Provides fast reads and removes buffer management, but requires large DRAM capacity		

Table 4.1: NVRAM design space. Database designs include recovery mechanisms (top) and cache configurations (left).

4.2.1 NVRAM Reads

While the exact read performance of future NVRAM technologies is uncertain, many technologies and devices increase read latency relative to DRAM. Current databases and computer systems are not equipped to deal with this read latency. Disk-backed databases incur sufficiently large read penalties (on the order of milliseconds) to justify software-managed DRAM caches and buffer management. On the other hand, main-memory databases rely only on the DRAM memory system, including on-chip data caches. Increased memory latency and wide-spread data accesses may require hardware or software-controlled DRAM caches even when using byte addressable NVRAM.

I consider three configurations of cache management; these alternatives form the three rows of Table 4.1 (subsequent sections consider the recovery management strategies, forming the three columns). The first option, *Software Buffer*, relies on software to manage a DRAM buffer cache, as in conventional disk-backed database systems. The cache may be removed entirely or execution relies solely on a *Hardware Buffer*, as in main-memory databases. Hardware caches are fast (e.g., on-chip SRAM) and remove complexity from the software, but provide limited capacity. Third, one might *replicate to DRAM* all data stored in NVRAM—writes update both DRAM and NVRAM (for recovery), but reads retrieve data exclusively from DRAM. Replicating data ensures fast reads by avoiding NVRAM read latencies (except for recovery) and simplifies buffer management, but requires large DRAM capacity.

4.2.2 NVRAM Writes

Persistent writes, unlike reads, do not benefit from caching; writes persist through to the device for recovery correctness. Additionally, NVRAM updates must be carefully ordered to ensure consistent recovery. I assume that ordering is enforced through a mechanism

Figure 4.2: Durable atomic updates. `persist_wal` appends to the ARIES log using two persist barriers. `persist_page` persists pages with four persist barriers.

```
1:  function PERSIST_WAL(log_buffer, nvram_log)
2:      for entry in log_buffer do
3:          nvram_log.FORCE_LAST_LSN_INVALID(entry)
4:          nvram_log.INSERT_BODY(entry)                    ▷ no LSN
5:      end for
6:      PERSIST_BARRIER()
7:      nvram_log.UPDATE_LSNS()                             ▷ for all entries
8:      PERSIST_BARRIER()
9:  end function
10: function PERSIST_PAGE(page_v, page_nv, page_log)
11:     page_log.COPY_FROM(page_nv)
12:     PERSIST_BARRIER()
13:     page_log.MARK_VALID()
14:     PERSIST_BARRIER()
15:     page_nv.COPY_FROM(page_v)
16:     PERSIST_BARRIER()
17:     page_log.MARK_INVALID()
18:     PERSIST_BARRIER()
19: end function
```

called a *persist barrier*, which guarantees that writes before the barrier successfully persist before any dependant operations after the barrier execute (including subsequent persists and externally visible side effects).

Persist barriers may be implemented in several ways. The easiest, but worst performing, is to delay at persist barriers until all pending NVRAM writes finish persisting. More complicated mechanisms improve performance by allowing threads to continue executing beyond the persist barrier and only delaying thread execution when persist conflicts arise (i.e., a thread reads or overwrites shared data from another thread that has not yet persisted). BPFS provides an example implementation of this mechanism [30]. Regardless of how they are implemented, persist barriers can introduce expensive synchronous delays on transaction threads; the optimal recovery mechanism depends on how expensive, on average, persist barriers become. To better understand how persist barriers are used and how frequently they occur, I outline operations to atomically update persistent data using persist barriers, and use these operations to implement three recovery mechanisms for NVRAM.

Atomic durable updates. Figure 4.2 shows two operations to atomically update NVRAM data. The first, `persist_wal()`, persists log entries into an ARIES log. Shore-MT log entries are post-pended with their Log Serial Number (LSN—log entry file offset). Log

entries are considered valid only if the tail LSN matches the starting location of the entry. I persist log entries atomically by first persisting an entry without its tail LSN and only later persisting the LSN, ordered by a persist barrier. Additionally, I reduce the number of persist barriers by persisting entries in batches, writing several log entries at once (without LSNs), followed by all LSNs. Log operations introduce two persist barriers—one to ensure that log entries persist before their LSNs, and one to enforce that LSNs persist before the thread continues executing.

The second operation, `persist_page()`, atomically persists page data using a persistent undo page log. First, the page's original data is copied to the page log. The page log is marked valid and the dirty version of the page is copied to NVRAM (updated in-place while locks are held). Finally, the log is marked invalid, and will no longer be applied (rolled back) at recovery. Four persist barriers ensure that each persist completes before the next, enforcing consistency at all points in execution. Recovery checks the valid flags of all page logs, copying valid logs back in-place.

The log is always valid while the page persists in-place, protecting against partial NVRAM writes and giving the appearance of atomic page updates. Together, `persist_wal()` and `persist_page()` provide the tools necessary to construct recovery mechanisms. I discuss these mechanisms next, describing their implementation and performance.

NVRAM Disk-Replacement. NVRAM database systems will likely continue to rely on ARIES/WAL at first, using NVRAM as *NVRAM Disk-Replacement*. WAL provides recovery for disk by keeping an ordered log of all updates, as described in Section 2.3. While retaining disk's software interface, NVRAM block accesses copy data between volatile and nonvolatile address spaces. *NVRAM Disk-Replacement* in Shore-MT persists the log and pages with `persist_wal()` and `persist_page()`, respectively. Persists occur on log and page flusher threads, and transaction threads do not delay (except when waiting for commit log entries to persist). *NVRAM Disk-Replacement* provides low recovery latency by aggressively flushing pages, minimizing recovery replay. While requiring the least engineering effort, *NVRAM Disk-Replacement* contains large software overheads to maintain a centralized log and asynchronously flush pages. Next, I leverage NVRAM's low latency to reduce these overheads.

In-Place Updates. Fast, byte-addressable NVRAM allows data to persist in-place, enforcing persist order immediately in a design called *In-Place Updates*. *In-Place Updates* permits the centralized log to be removed by providing redo and undo log functionality elsewhere. I remove redo logs by keeping the database's durable state up-to-date. In ARIES terms, the database maintains its replayed state—there is no need to replay a redo log after failure. Undo logs need not be ordered across transactions (transactions are free to roll

back in any order), and are instead distributed by transaction. Such private logs are simpler and impose fewer overheads than centralized logs. Other databases (such as Oracle) already distribute undo logs in rollback segments and undo table spaces [32]. Transaction undo logs remain durable so that in-flight transactions can be rolled back after failure. Page updates (1) latch the page, (2) insert an undo entry into the transaction-private undo log using `persist_wal()`, (3) update the page using `persist_page()` (without an intermediate volatile page), and (4) release the page latch. This protocol ensures all updates to a page, and updates within a transaction, persist in-order, and that no transaction reads data from a page until it is durable.

Persisting data in-place removes expensive redo logging and asynchronous page flushing, but introduces persist barriers on transactions' critical paths. For sufficiently short persist barrier delays *In-Place Updates* outperforms *NVRAM Disk-Replacement*. However, one would expect transaction performance to suffer as persist barrier delay increases.

In response, I introduce *NVRAM Group Commit*, a recovery mechanism designed to minimize the frequency of persist barriers while still removing WAL. *NVRAM Group Commit* is an entirely new design, committing instructions in large batches to minimize persist synchronization. The next section describes, in detail, the operation and data structures necessary to implement *NVRAM Group Commit*.

4.3 NVRAM Group Commit

The two previous recovery mechanisms provide high throughput under certain circumstances, but contain flaws. *NVRAM Disk-Replacement* is insensitive to large persist barrier delays. However, it assumes IO delays are the dominant performance bottleneck and trades off software overhead to minimize IO. *In-Place Updates*, on the other hand, excels when persist barriers delays are short. As persist barrier latency increases performance suffers, and *NVRAM Disk-Replacement* eventually performs better. Here, I present *NVRAM Group Commit*, coupling *NVRAM Disk-Replacement*'s persist barrier latency-insensitivity with *In-Place Updates*'s low software overhead.

4.3.1 Operating Principles and Implementation

NVRAM Group Commit operates by executing transactions in batches, whereby all transactions in the batch commit or (on failure) all transactions abort. Transactions quiesce between batches—at the start of a new batch transactions stall until the previous batch commits. Each transaction maintains a private ARIES-style undo log, supporting abort and roll-back as in *In-Place Updates*, but transaction logs are no longer persistent.

As batches persist atomically, transactions no longer roll back selectively during recovery (rather, aborted transactions roll back before any data persists), obviating the need for persistent ARIES undo logs. Instead, recovery relies on a database-wide undo log and staging buffer to provide durable atomic batches.

NVRAM Group Commit leverages byte-addressability to reduce persist barrier frequency by ordering persists at batch rather than transaction or page granularity. Batch commit resembles `persist_page()`, used across the entire database, once per batch. Because undo logging is managed at the batch level, transactions' updates must not persist in-place to NVRAM until all transactions in the batch complete (no-steal policy applied to batches). Rather, transactions write to a volatile staging buffer, tracking dirtied cache lines in a concurrent bit field. Once the batch ends and all transactions complete, the pre-batch version of dirtied data is copied to the database-wide persistent undo log, only after which data is copied from the staging buffer in-place to NVRAM. Finally, the database-wide undo log is invalidated, transactions commit, and transactions from the next batch begin executing. On failure the log is copied back, aborting and rolling back all transactions from the in-flight batch. The key observation is that *NVRAM Group Commit* persists entire batches of transactions using four persist barriers, far fewer than required with *In-Place Updates*. Note, however, that it enables recovery only to batch boundaries, rather than transaction boundaries.

I briefly outline two implementation challenges: long transactions and limited staging buffers. Long transactions force other transactions in the batch to defer committing until the long transaction completes. Limited staging buffers, not large enough to hold the entire data set, may fill while transactions are still executing. I solve both problems by resorting to persistent ARIES-style undo logs, as in *In-Place Updates*. Long transactions persist their ARIES undo log (previously volatile), allowing the remainder of the batch to persist and commit. The long transaction joins the next batch, committing when that batch commits. At recovery the most recent batch rolls back, and the long transaction's ARIES undo log is applied, removing updates that persisted with previous batches. Similarly, if the staging buffer fills, the current batch ends immediately and all outstanding transactions persist their ARIES undo logs. The batch persists, treating in-flight transactions as long transactions, reassigning them to the next batch. This mechanism requires additional persistent data structures to allow transaction and batch logs to invalidate atomically.

NVRAM Group Commit requires fewer persist barriers than *In-Place Updates* and avoids *NVRAM Disk-Replacement*'s logging. Batches require four persist barriers regardless of batch length. Persist barrier delays are amortized over additional transactions by increasing batch length, improving throughput. However, increasing batch length defers commit

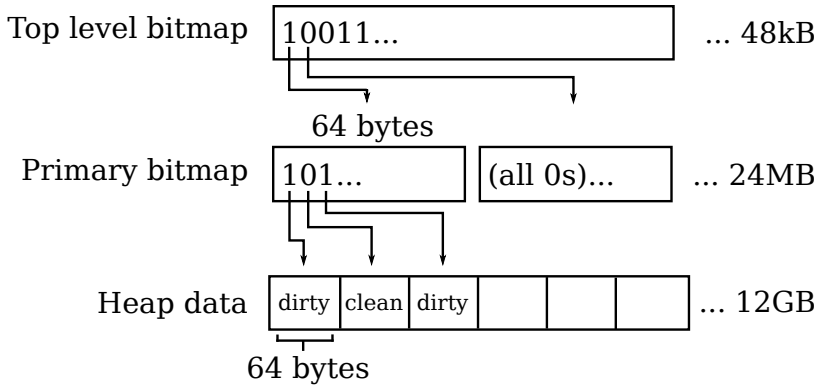


Figure 4.3: Hierarchical bitmap set. Each bit of top level bitmap corresponds to a cache line of bits (64 bytes, 512 bits) in the primary bitmap; top level bit is set if any associated bits in primary bitmap are set (boolean *or* operation). Each bit in the primary bitmap corresponds to a cache line in the database buffer cache and is set if the cache line is dirty (contains modifications since the start of the batch).

for all transactions in the batch, increasing transaction latency. Batch length must be large enough such that batch execution time dominates time spent quiescing transactions between batches.

4.3.2 High Performance Group Commit

NVRAM Group Commit improves on *In-Place Updates* by minimizing persist barrier frequency. However, additional overheads are introduced to (1) track modified addresses while transactions execute, (2) quiesce transactions at batch end, and (3) persist and commit each batch. I outline new mechanisms to accelerate batch performance.

Hierarchical bitmap set. Modified addresses must be tracked so that dirtied data can be quickly located and persisted at the end of each batch. Adding to, iterating over, and clearing the set must be fast. I track modifications at cache line (64-byte) granularity. Batch logging and persisting additionally occurs at this granularity. My first attempts considered a balanced tree set (STL set), which resulted in unacceptably large overheads to add to the set, and a bitmap set, which provided slow iteration—a 12GB buffer cache implies 24MB of bitmap, which requires too many instructions to scan quickly. Instead, I leverage the fact that updates to the buffer cache by each batch are sparse to introduce a new set data structure, the *hierarchical bitmap set*.

The ideal set data structure would use an insert operation similar to the bitmap set (requiring only a bitwise-or operation) but iterate through the set faster. I observe that at the end of each batch only a small portion of the buffer cache is dirty—set iteration should recognize large contiguous blocks of clean addresses and skip these. The hierarchical bitmap

set introduces an additional bitmap, the *top level bitmap*, which indicates if an entire cache line of the *primary bitmap* contains any 1s—each bit of the top level bitmap maps to 512 bits of the primary bitmap. Figure 4.3 shows the hierarchical bitmap set. The top level bitmap (48kB) contains bits corresponding to cache lines of the primary bitmap (24MB) which in turn contains bits corresponding to cache lines of the heap (12GB buffer cache). In the Figure the first bit of the top level bitmap is set, denoting that at least one bit in the first cache line of the primary bitmap must be set. On the other hand, the second bit of the top level bitmap is not set, indicating that no bits in the second cache line of the primary bitmap are set, and that all corresponding cache lines in the heap are clean.

During batch execution addresses are added to the hierarchical bitmap set by setting the correct bits in both the primary and top level bitmaps. Since bits within the same byte in the top level bitmap correspond to different heap pages, and thus must allow concurrent operations, the top level bitmap must be updated using an atomic-or instruction. Iteration is performed by scanning the top level bitmap until a 1 is found, and then scanning the corresponding cache line of the primary bitmap. I additionally optimize bitmap scanning by testing data at eight-byte granularity, then one-byte granularity, and then by bit. Any block without a set bit (i.e., the block equals zero) is skipped. Scanning at successively smaller granularity minimizes the number of instructions necessary to search the bitmap. Processor-specific instructions enable scanning of larger data blocks, but I found additional optimization unnecessary. Finally, the bitmaps are cleared using *memset*; clean portions of the primary bitmap need not be cleared and may be skipped.

I map each bit of the top level bitmap to 64 bytes (512 bits) of the primary bitmap to minimize the number of primary bitmap cache lines accessed, but this mapping may be modified. Additional layers of bitmaps may also be added. Whether modification would help depends on the access characteristics and sparseness of data within the set. My structure appears to match my workloads and minimizes insert, iteration, and clearing overheads.

Concurrent batch persist. I have shown that dirty lines can be tracked and iterated over efficiently. However, persisting each batch with the batch coordinator alone results in long persist delays. These delays are due to the *software* overhead of copying data, not NVRAM limitations. To reduce these delays persist operations must be parallelized across threads. I use quiesced transaction threads, formerly waiting for the previous batch to commit, to log and persist the batch and accelerate batch commit.

The buffer cache address space and corresponding portions of the primary bitmap are partitioned into segments and placed in a task queue. The batch coordinator and transaction threads blocked by batch commit each participate by processing tasks, persisting

Workload	Relative performance
TPCC	79.2%
TPCB	86.2%
TATP	83.8%

Table 4.2: Concurrent *NVRAM Group Commit* persist. *NVRAM Group Commit* must use many threads to concurrently persist the batch log and data. Here I show the relative throughput of using only a single thread versus additionally using quiesced transaction threads to accelerate batch persist and commit. Results assume a 20ms batch period.

buffer cache partitions (both log and then data in-place). Once all tasks complete the batch commits, allowing the next batch to begin.

Table 4.2 shows the relative slowdown that results from using only a single thread to persist and commit each batch using a 20ms batch period (methodology discussed in Section 5.1). At worst, 20% throughput is lost (TPCC), yet all workloads suffer at least a 13.8% slowdown. Threads must persist each batch concurrently to achieve the throughput of *In-Place Updates* while remaining resilient to large persist barrier latency.

Using the hierarchical bitmap set and concurrent batch persists minimizes batch overheads. Profiling indicates that *memcpy* operations, copies from the buffer cache to the persistent address space, form the remaining bottleneck (memcpy is implemented using fast SSE instructions and cannot be optimized further), indicated few additional opportunities for optimization.

4.4 Design Space

I describe the space of possible designs given choices regarding NVRAM read and write performance. This discussion ignores possible uses of hard disk to provide additional capacity. Each design works alongside magnetic disk with additional buffer management and the constraint that pages persist to disk before being evicted from NVRAM. Many modern OLTP applications' working sets are small, fitting entirely in main-memory.

Table 4.1 lists the possible combinations of caching architectures and recovery mechanisms. The left column presents *NVRAM Disk-Replacement*, the obvious and most incremental use for NVRAM. Of note is the center-left cell, *NVRAM Disk-Replacement* without the use of a volatile buffer. WAL, by its design, allows pages to write back asynchronously from volatile storage. Removing the volatile cache requires transactions to persist data in-place, but do so only after associated log entries persist, retaining the software overheads of *NVRAM Disk-Replacement* as well as the frequent synchronization in *In-Place Updates*. Thus, this design is impractical.

The middle-column recovery mechanism, *In-Place Updates*, represents the most intuitive use of NVRAM in database systems, as noted in several prior works. Agrawal and Jagadish explore several algorithms for atomic durable transactions with an NVRAM main-memory [3]. They describe the operation and correctness of each mechanism and provide an analytic cost model to compare them. Their work represents the middle column, middle row of Table 4.1 (*In-Place Updates* with no volatile buffer). Akyürek and Salem present a hybrid DRAM and NVRAM buffer cache design alongside strategies for managing cache allocation [98]. Such *Partial Memory Buffers* resemble the middle-top cell of the design space table (*In-Place Updates* with a software-managed DRAM buffer), although their design considers NVRAM as part of a hybrid buffer, not the primary persistent store. Neither of these works considers alternative approaches (such as *NVRAM Group Commit*), to account for large persist barrier latency and associated delays. Additionally, this work extends prior work by providing a more precise performance evaluation and more detailed consideration of NVRAM characteristics.

The right column presents *NVRAM Group Commit*. Each of the three recovery mechanisms may replicate all data between NVRAM and DRAM to ensure fast read accesses, manage a smaller DRAM buffer cache, or omit the cache altogether. In Section 5.2 I consider the importance of NVRAM caching to transaction throughput. Then, in Section 5.3 I assume a DRAM-replicated heap to isolate read performance from persist performance in evaluating each recovery mechanisms's ability to maximize transaction throughput.

4.5 Related Work

To the best of my knowledge, this work is the first to investigate NVRAM write latency and its effect on durable storage and recovery in OLTP. A large body of related work considers applications of NVRAM and reliable memories.

Ng and Chen place a database buffer cache in battery-backed DRAM, treating it as reliable memory [77]. However, the mechanisms they investigate are insufficient to provide ACID properties under any-point failure or protect against many types of failure (e.g., power loss).

Further work considers NVRAM in the context of file systems. Baker *et al.* use NVRAM as a file cache to optimize disk I/O and reduce network traffic in distributed file systems, yet continue to assume that disk provides the bulk of persisted storage [5]. More recently, Condit *et al.* demonstrate the hardware and software design necessary to implement a file system entirely in NVRAM as the Byte-Addressable Persistent File System (BPFS) [30]. While I assume similar hardware, I additionally consider a range of NVRAM

performance and focus instead on databases.

Other work develops programming paradigms and system organizations for NVRAM. Coburn *et al.* propose NV-Heaps to manage NVRAM within the operating system, provide safety guarantees while accessing persistent stores, and atomically update data using copy-on-write [29]. Volos *et al.* similarly provide durable memory transactions using Software Transactional Memory (STM) and physical redo logging per transaction [116]. While these works provide useful frameworks for NVRAM, they do not investigate the effect of NVRAM persist latency on performance, nor do they consider OLTP, where durability is tightly coupled with concurrency and transaction management.

Recently, researchers have begun to focus specifically on databases as a useful application for NVRAM. Chen *et al.* reconsider database algorithms and data structures to address NVRAM's write latency, endurance, and write energy concerns, generally aiming to reduce the number of modified NVRAM bits [24]. However, their work does not consider durable consistency for transaction processing. Venkataraman *et al.* demonstrate a multi-versioned log-free B-Tree for use with NVRAM [115]. Indexes are updated in place, similarly to my *In-Place Updates*, without requiring any logging (physical or otherwise) and while providing snap shot reads. This work considers durability management at a higher level—user transactions—and consistency throughout the entire database. Finally, Fang *et al.* develop a new WAL infrastructure for NVRAM that leverages byte addressable and persistent access [37]. Fang aims to improve transaction throughput but retains centralized logging. I distinguish myself by investigating how NVRAM write performance guides database and recovery design more generally.

More similar to my work, Coburn *et al.* introduce hardware support for durable transactions in NVRAM storage devices [28]. These *editable atomic writes* provide the same functionality as traditional ARIES logs. Such hardware must still serialize and persist log entries; contention while serializing entries and ordering persists may still limit system throughput. Additionally, editable atomic writes use physical logging (updates are recorded as location-data-length tuples), as opposed to logical logging (updates are recorded as a set of data and functions to apply/roll back the update). Logical logging is necessary to precisely control data placement when supporting concurrent transactions. For example, to maintain B+Tree elements in sorted order within a page each transaction must modify large portions of the page (to shift entries), even if those entries were recently modified by another outstanding transaction. It is unclear if physical logging provides all the features expected from ARIES.

Prior work (e.g., H-Store [104]) has suggested highly available systems as an outright replacement for durability. I argue that computers and storage systems will always fail, and

durability remains a requirement for many applications.

4.6 Conclusion

This chapter motivated the need to reconsider system design for NVRAM recovery management. I highlight possible caching architectures as well as three candidate recovery management software designs and their implementations. The next chapter compares these system designs, considering performance related to NVRAM read and persist latencies.

CHAPTER V

An Evaluation of NVRAM Recovery Management

This chapter builds on the previous to investigate the performance effects of different caching architectures and recovery mechanisms for NVRAM. I introduce a methodology for evaluating database performance with upcoming NVRAMs and look at NVRAM read and write performance concerns separately.

5.1 Methodology

This section details the methodology for benchmarking transaction processing and modeling NVRAM performance. Experiments use the Shore-MT storage manager [54], including the high performance, scalable WAL implementation provided by Aether [55]. While Aether additionally provides a distributed log suitable for multi-socket servers, the distributed log exists as a fork of the main Shore-MT project. Instead, I limit experiments to a single CPU socket to provide a fair comparison between WAL and other recovery schemes, enforced using the Linux *taskset* utility. Experiments place both the Shore-MT log and volume files on an in-memory *tmpfs*, and provide sufficiently large buffer caches such that all pages hit in the cache after warmup. The intent is to allow the database to perform data accesses at DRAM speed and introduce additional delays to model NVRAM performance. Table 5.1 shows the experimental system configuration.

Operating System	Ubuntu 12.04
CPU	Intel Xeon E5645 2.40 GHz
CPU cores	6 (12 with HyperThreading)
Memory	32 GB

Table 5.1: Experimental system configuration.

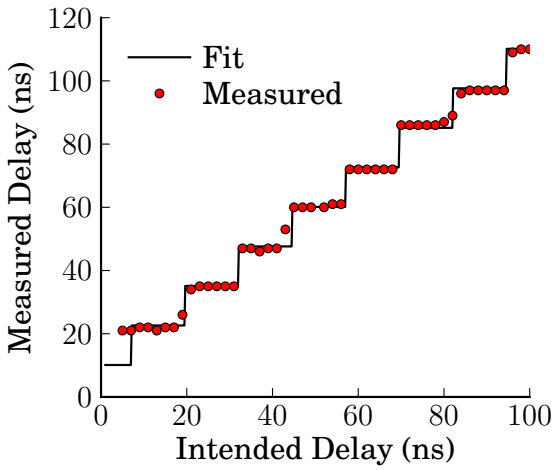


Figure 5.1: Delay precision. Delays are implemented by repeatedly reading the TSC register. Resulting delays form a step function. Inserted delays are within 6ns of the intended delay.

Modeling NVRAM delays. Since NVRAM devices are not yet available, I must provide a timing model that mimics their expected performance characteristics. I model NVRAM read and write delays by instrumenting Shore-MT with precisely controlled assembly-code delay loops to model additional NVRAM latency and bandwidth constraints at 13ns precision. Hence, Shore-MT runs in real time as if its buffer cache resided in NVRAM with the desired read and write characteristics.

I introduce NVRAM delays using the x86 RDTSCP instruction, which returns a CPU-frequency-invariant, monotonically increasing time-stamp that increments each clock tick. RDTSCP is a synchronous instruction—it does not allow other instructions to reorder with it. The RDTSCP loop delays threads in increments of 13ns (latency per loop iteration and RDTSCP) with an accuracy of 2ns.

Figure 5.1 shows the measured delay that results from iterating on an intended delay (both in ns). Points show the median of ten thousand delay trials (a small number of trials result in excessively large delays and skew the mean). The delay function resembles a step function—delays may only be inserted in multiples of the loop iteration latency.

To better understand delay behavior I perform a least squares regression of a step function against the measured data of the form:

$$delay(intended) = a \times floor(intended/a + b) + c$$

which results in an R^2 of .997 and the following parameter values: $a = 12.5$, $b = .434$, and $c = 10.1$. This indicates that each iteration takes 12.5ns.

As delays can only be introduced in steps I subtract 9.3ns from the intended delay to match to the closest step. The resulting delay is within 6.25ns of the intended delay. However, a given delay contains a constant skew (for example, an intended delay of 30ns always results in a 35ns delay). As NVRAM persist latencies are expected to be in the hundreds of ns or greater, such error will negligibly affect my results.

Modeling NVRAM persist bandwidth. In addition to NVRAM latency, I model shared NVRAM write bandwidth. Using RDTSCP as a clock source, I maintain a shared *next_available* variable, representing the next clock tick in which the NVRAM device is available to be written. Each NVRAM persist advances *next_available* to account for the latency of its persist operation. Reservations take the maximum of *next_available* and the current RDTSCP and add the reservation duration. The new value is atomically swapped into *next_available* via a Compare-And-Swap (CAS). If the CAS fails (due to a race with a persist operation on another thread), the process repeats until it succeeds. Upon success, the thread delays until the end of its reservation. The main limitation of this approach is that it cannot model reservations shorter than the delay required to perform a CAS to a contended shared variable. The delay incurred by a CAS instruction depends on contention to the address and the scheduling of threads across cores and processors. I demonstrate that this technique models reservations above 120ns accurately, which is sufficient for my experiments.

Several factors affect the speed of bandwidth reservations (and therefore reservation accuracy) including the number of threads and their placement across sockets and cores. I test the reservation system's accuracy by constraining thread placement while threads repeatedly reserve time and delay until the end of the reservation. If all available time is reserved, reservation overhead is negligible and the modeled bandwidth is accurate. However, when reservation length is sufficiently short reservation overheads will dominate, resulting in unreserved time.

Table 5.2 shows the required reservation length (in ns) to reserve 99% of time with six and 12 threads. Additionally, the Table shows different thread placement across sockets and cores. "Spread" implies assigning threads round-robin to resources, while "pack" indicates that each resource is filled before assigning any threads to the next. For example, assigning six threads in a pack sockets–spread cores policy (on a two socket server, each with six cores and two-way SMT) results in all six threads scheduled on the same socket but each thread on its own core (thus SMT is unused). Such a configuration requires reservations of only 44ns to reserve 99% of bandwidth-time.

Spreading threads across sockets or packing within cores slows reservations by requiring long-latency communication between sockets or forcing threads to contend with each

socket policy	spread cores	pack cores
spread	109	122
pack	44	104

(a) 6 threads

socket policy	spread cores	pack cores
spread	96	101
pack	51	51

(b) 12 threads

Table 5.2: Bandwidth reservation bendmark. Benchmark repeatedly reserves bandwidth as time and delays until end of the reservation. Reservations are inaccurate if entire time cannot be reserved (reservation overhead dominates). Results shown are reservation sizes (in ns) to reserve 99% time. Thread placement is varied across CPU sockets and cores: packed (fill socket/core before allocating new) or spread (round robin assign to resources). 122ns and greater reservations accurately model constrained bandwidth.

other while scheduling instructions on cores. At worst, when threads are spread across sockets and packed within cores a reservation length of 122ns is required to reserve 99% of time.

A similar trend is true when considering 12 threads. Packing threads into a socket completely fills all cores and SMT contexts of the socket (the bottom two cells represent the same configuration), needing 51ns reservations for accurate bandwidth modeling. Spreading threads across sockets while packing cores requires 101ns to reserve 99% of time. The required time decreases from six to 12 threads as more threads are available to reserve time and it is less likely that time will go unreserved.

These results suggest that bandwidth reservations will be accurate, regardless of how threads are scheduled across processors and cores, so long as each reservation exceeds 120ns. My additions to Shore-MT reserve bandwidth for each cache line persisted. The persist bandwidth analysis study (presented later in Section 5.3.3) shows that 35ns per cache line (approximately 1.7GB/s persists) represents sufficient bandwidth to negligibly limit performance. Since at least four cache lines are always reserved together (in `persist_page` and `persist_wal`) the bandwidth reservation is accurate.

NVRAM performance. I introduce NVRAM read and write delays separately. Accurately modeling per-access increases in read latency is challenging, as reads are frequent and the expected latency increases on NVRAM compared to DRAM are small. It is infeasible to use software instrumentation to model such latency increases at the granularity of individual reads; hardware support, substantial time dilation, or alternative evaluation

techniques (e.g., simulation) would be required, all of which compromise accuracy and the ability to run experiments at full scale. Instead, I use offline analysis with PIN [67] to determine (1) the reuse statistics of buffer cache pages, and (2) the average number of cache lines accessed each time a page is latched. Together, these offline statistics provide an average number of cache line accesses per page latch event in Shore-MT. I then introduce a delay at each latch based on the measured average number of misses and an assumed per-read latency increase based on the NVRAM technology.

I model NVRAM persist delays by annotating Shore-MT to track buffer cache writes at cache line granularity—64 bytes—using efficient "dirty" bitmaps. Depending on the recovery mechanism, I introduce delays corresponding to persist barriers and to model NVRAM write bandwidth contention. Tracking buffer cache writes introduces less than a 3% overhead to the highest throughput experiments.

I choose on-line timing modeling via software instrumentation in lieu of architectural simulations to allow experiments to execute at full scale and in real time. While modeling aspects of NVRAM systems such as cache performance and more precise persist barrier delays require detailed hardware simulation, I believe NVRAM device and memory system design are not sufficiently established to consider this level of detail. Instead, I investigate more general trends to determine if and when NVRAM read and write performance warrant storage management redesign.

Recovery performance. Figure 4.1 displays recovery latency vs transaction throughput for the TPCB workload, varying page flush rate. Page flush rate is controlled by maintaining a constant number of dirty pages in the buffer cache, always flushing the page with the oldest volatile update. Experiments run TPCB for one minute (sufficient to reach steady state behavior) and then kill the Shore-MT process. Before starting recovery I drop the file system cache. Reported recovery time includes only the recovery portion of the Shore-MT process; I do not include system startup time nor non-recovery Shore-MT startup time.

Workloads. I use three workloads and transactions in this evaluation: TPCC, TPCB, and TATP. TPCC models order management for a company providing a product or service [108]. TPCB contains one transaction class and models a bank executing transactions across branches, tellers, customers, and accounts [107]. TATP includes seven transactions to model a Home Location Registry used by mobile carriers [76]. Table 5.3 shows the workload configuration. I scale workloads to fit in a 12GB buffer cache. Persist performance experiments use a single write-heavy transaction from each workload while read performance experiments use each workload's full mix. All experiments report throughput as thousands of Transactions Per Second (kTPS). Experiments perform "power runs"—each thread generates and executes transactions continuously without think time—and run

Workload	Scale factor	Size	Write transaction
TPCC	70	9GB	New order
TPCB	1000	11GB	
TATP	600	10GB	Update location

Table 5.3: Workloads and transactions. One transaction class from each of three workloads, sized to approximately 10GB.

	TATP		TPCB		TPCC		Average	
	% lines	lines/ latch	% lines	lines/ latch	% lines	lines/ latch	% lines	lines/ latch
Heap	7.26%	4.19	15.47%	4.25	15.27%	6.10	12.66%	4.85
Index	92.45%	11.82	81.18%	11.17	81.54%	11.44	85.06%	11.48
Other	0.29%	3.00	3.36%	3.00	3.19%	8.31	2.28%	4.77
Total		11.24		9.83		10.52		10.48

Table 5.4: NVRAM access characteristics. "% lines" indicates the percentage breakdown of cache line accesses. "lines/latch" reports the average number of cache line accesses per page latch. Indexes represent the majority of accesses.

an optimal number of threads per configuration (between 10 and 12).

5.2 NVRAM Reads

I first evaluate database performance with respect to NVRAM reads. Many candidate NVRAM technologies exhibit greater read latency than DRAM, possibly requiring additional hardware or software caching. I wish to determine, for a given NVRAM read latency, how much caching is necessary to prevent slowdown, and whether it is feasible to provide this capacity in a hardware-controlled cache (otherwise software caches must be used).

5.2.1 NVRAM Caching Performance

Traces. The NVRAM read-performance model combines memory access trace analysis with the timing model to measure transaction throughput directly in Shore-MT. Traces consist of memory accesses to the buffer cache, collected running Shore-MT with PIN for a single transaction thread for two minutes. I assume concurrent threads exhibit similar access patterns. In addition, I record all latch events (acquire and release) and latch page information (i.e., table id, store type—index, heap, or other). I analyze traces at cache line (64 bytes) and page (8KB) granularity.

These traces provide insight into how Shore-MT accesses persistent data, summarized

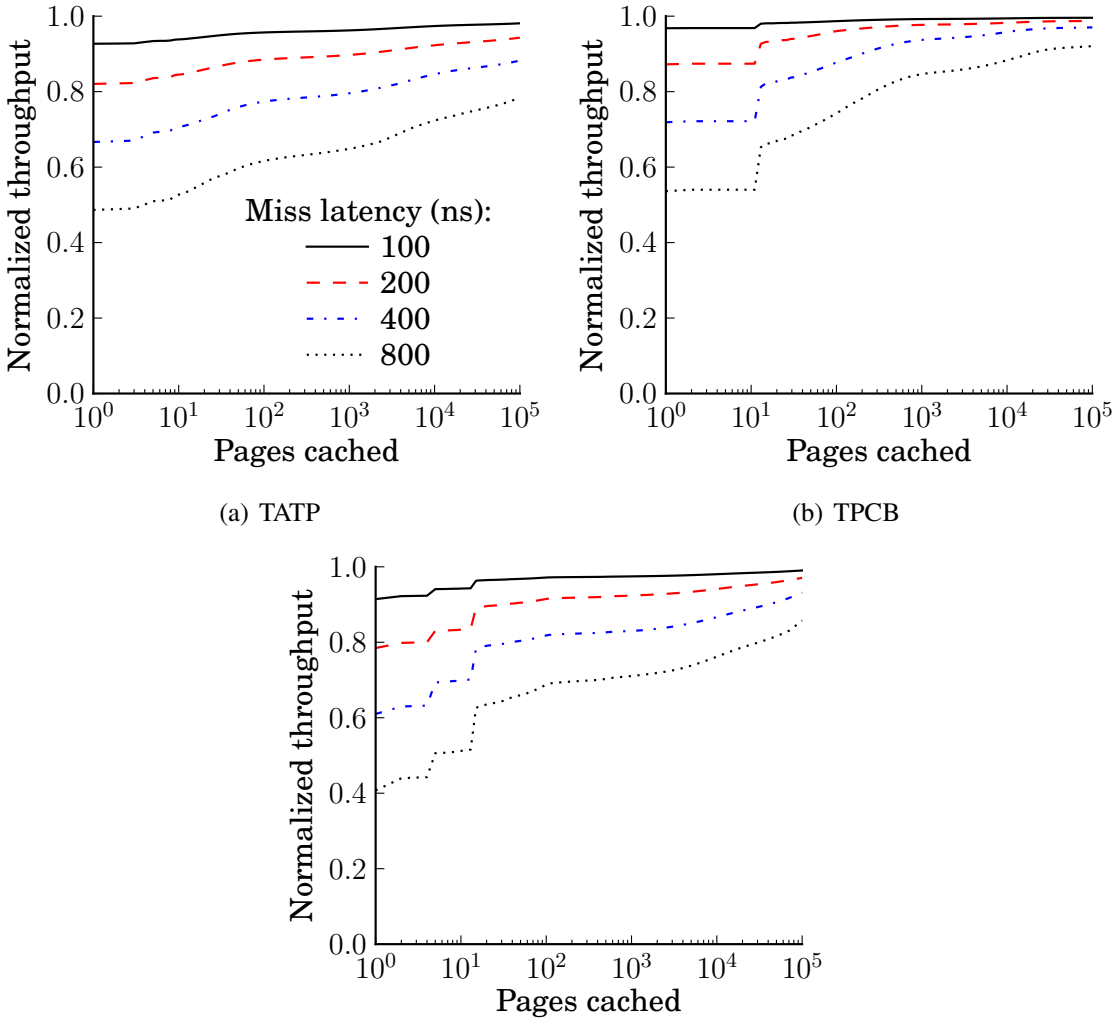
in Table 5.4. Index accesses represent the great majority of cache line accesses, averaging 85% of accesses to NVRAM across workloads. Any caching efforts should focus primarily on index pages and cache lines. Note also that indexes access a greater number of cache lines per page access than other page types (average 11.48 vs 4.85 for heap pages and 4.77 for other page types), suggesting that uncached index page accesses have the potential to introduce greater delays.

Throughput. I create a timing model in Shore-MT from the previous memory traces. Given traces, I perform cache analysis at page granularity, treating latches as page accesses and assuming a fully associative cache with a least-recently-used replacement policy (LRU). Cache analysis produces an average page miss rate to each table. I conservatively assume that every cache line access within an uncached page introduces an NVRAM stall, neglecting optimizations such as out-of-order execution and simultaneous multi-threading that might hide some NVRAM access stalls. The model assumes the test platform incurs a 50ns DRAM fetch latency, and adds additional latency to mimic NVRAM (for example, a 200ns NVRAM access adds 150ns delay per cache line). I combine average page miss rate and average miss penalty (from lines/latch in table 5.4) to compute the average delay incurred per latch event. This delay is inserted at each page latch acquire in Shore-MT, using *In-Place Updates*, to produce a corresponding throughput.

Figure 5.2 shows throughput achieved for the three workloads while varying the number of pages cached (horizontal axis) and NVRAM miss latency (various lines). The vertical axis displays throughput normalized to DRAM-miss-latency's throughput (no additional delay inserted). Without caching, throughput suffers as NVRAM miss latency increases, shown at the extreme left of each graph. A 100ns miss latency consistently achieves at least 90% of potential throughput. However, an 800ns miss latency averages only 50% of the potential throughput, clearly requiring caching. TPCB and TPCC see a 10-20% throughput improvement for a cache size of just 20 pages. As cache capacity further increases, each workload's throughput improves to varying degrees. A cache capacity of 100,000 (or 819MB at 8KB pages) allows NVRAMs with 800ns miss latencies to achieve at least 80% of the potential throughput. While too large for on-chip caches, such a buffer might be possible as a hardware-managed DRAM cache [88].

5.2.2 Analysis

I have shown that modest cache sizes effectively hide NVRAM read stalls for these workloads, and further analyze caching behavior to reason about OLTP performance more generally. Figure 5.3 shows the page miss rate per page type (index, heap, or other) as page cache capacity increases. Each graph begins at one at the left—all page accesses miss

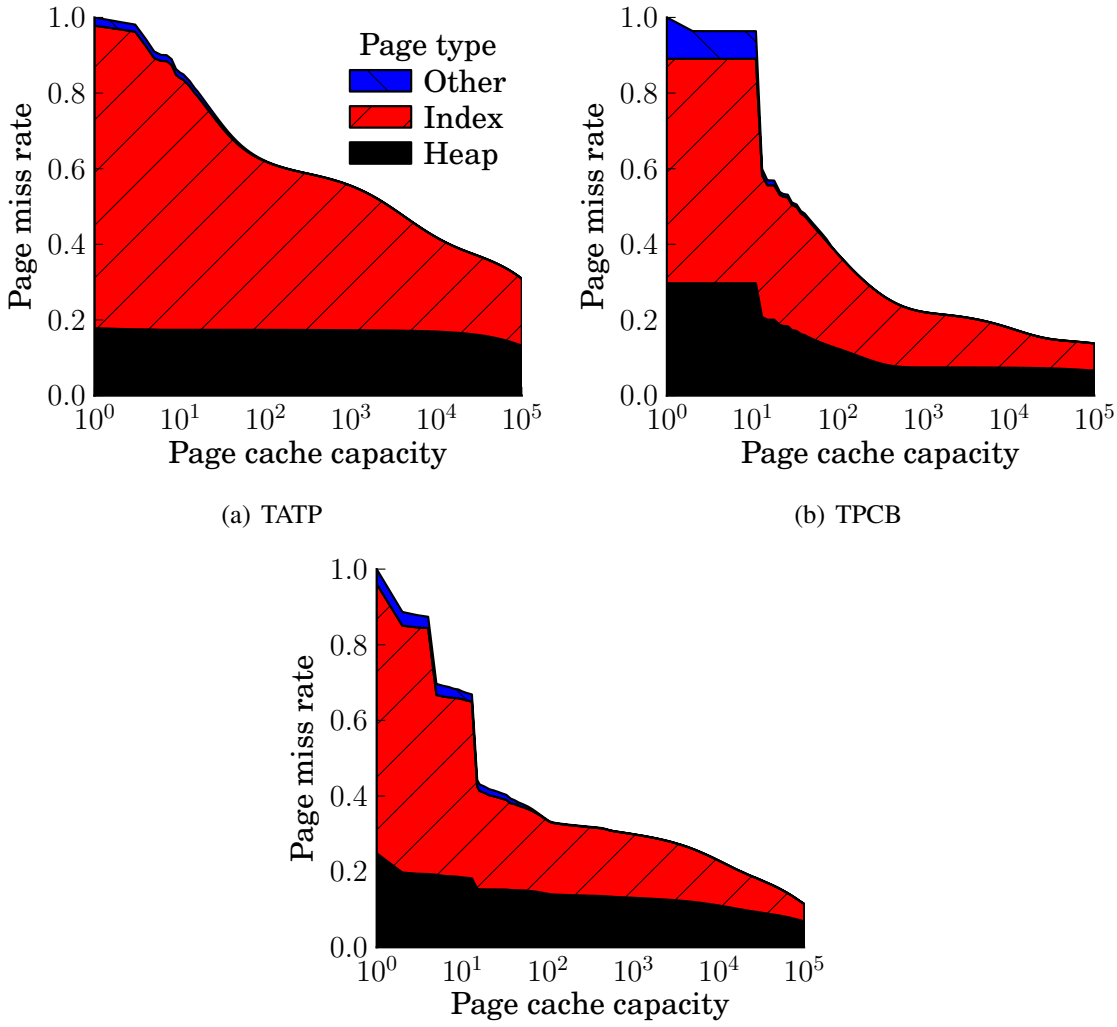


(a) TATP

(b) TPCB

(c) TPCC

Figure 5.2: Throughput vs NVRAM read latency. 100ns miss latency suffers up to a 10% slowdown over DRAM. Higher miss latencies introduce large slowdowns, requiring caching. Fortunately, even small caches effectively accelerate reads.



(a) TATP

(b) TPCB

(c) TPCC

Figure 5.3: Page caching effectiveness. High level B+Tree pages and append-heavy heap pages cache effectively. Other pages cache as capacity approaches table size.

for a single-page cache. As cache capacity increases, workloads see their miss rates start to decrease between cache capacity of five and 20 pages. TATP experiences a decrease in misses primarily in index pages, whereas TPCB and TPCC see decreases across all page types.

While cache behavior is specific to each workload, the results represent trends applicable to many databases and workloads, specifically, index accesses and append-heavy tables. First, all workloads see a decrease in index page misses as soon as B+Tree roots (accessed on every traversal) successfully cache. The hierarchical nature of B+Tree indexes allows high levels of the tree to cache effectively for even a small cache capacity. Additionally, TPCB and TPCC contain history tables to which data are primarily appended. Transactions append to the same page as previous transactions, allowing such tables to cache effectively. Similarly, extent map pages used for allocating new pages and locating pages to append into are frequently accessed and likely to cache. The remaining tables' pages are accessed randomly and only cache as capacity approaches the size of each table. In the case of TPCB and TPCC, each transaction touches a random tuple of successively larger tables (Branch, Teller, and Account for TPCB; Warehouse, District, Customer, etc. for TPCC). This analysis suggests that various page types, notably index and append-heavy pages, cache effectively, accelerating throughput for high-latency NVRAM misses with small cache capacities.

Main-memory databases. While I use Shore-MT (a disk-based storage manager) as a research platform, main-memory database optimizations (e.g., [33, 6, 78]) also apply to byte-addressable NVRAM. Main-memory databases assume heap data resides solely in byte-addressable memory, improving throughput relative to traditional disk-backed storage by removing expensive indirection (i.e., using memory pointers instead of buffer translation tables), reducing overheads associated with concurrency control and latches, and optimizing data layout for caches and main-memory, among other optimizations. While such optimizations will increase transaction throughput, removing non-NVRAM overheads will amplify the importance of read-miss latency (an equal increase in NVRAM read-miss latency will yield a relatively greater drop in performance). At the same time, data layout optimizations will reduce the number of cache lines and memory rows accessed per action (e.g., per latch), minimizing NVRAM read overheads. Investigating main-memory database optimizations for NVRAM remains future work.

Bandwidth. Finally, I briefly address NVRAM read bandwidth. For a worst-case analysis, I assume no caching. Given the average number of cache line accesses per page latch, the average number of page latches per transaction, and transaction throughput (taken from Section 5.3), I compute worst-case NVRAM read bandwidth for each workload, shown

Workload	Bandwidth (GB/s)
TATP	0.977
TPCB	1.044
TPCC	1.168

Table 5.5: Required NVRAM read bandwidth. Workloads require up to 1.2 GB/s read bandwith.

in Table 5.5. The considered workloads require at most 1.2 GB/s (TPCC). Since this is within expected NVRAM bandwidth constraints and caching reduces the required bandwidth further, I conclude that NVRAM read bandwidth for persistent data on OLTP is not a performance concern.

5.2.3 Summary

NVRAM presents a new storage technology, requiring new optimizations for database systems. Increased memory read latencies require new consideration for database caching systems. I show that persistent data for OLTP can be cached effectively, even with limited cache capacity. I expect future NVRAM software to leverage hardware caches, omitting software buffer caches. Next, I turn to write performance for storage management on NVRAM devices.

5.3 NVRAM Persist Synchronization

Whereas NVRAM reads benefit from caching, persists must always access the storage device. Of particular insterest is the cost of ordering persists via persist barriers. Several factors increase persist barrier latency, including ordering persists across distributed/NUMA memory architectures, long latency interconnects (e.g., PCIe-attached storage), and slow NVRAM MLC cell persists. I consider the effect of persist barrier latency on transaction processing throughput to determine if and when new NVRAM technologies warrant redesigning recovery management.

Refer to Sections 4.2 and 5.1 for a more thorough description of recovery mechanisms and experimental setup. All experiments throttle persist bandwidth to 1.5GB/s, which I believe to be conservative (already possible with PCIe-attached Flash). Ideally, NVRAM will provide low latency access, enabling *In-Place Updates*. However, one would expect *In-Place Updates*'s performance to suffer at large persist barrier latencies, requiring either *NVRAM Disk-Replacement* or *NVRAM Group Commit* to regain throughput.

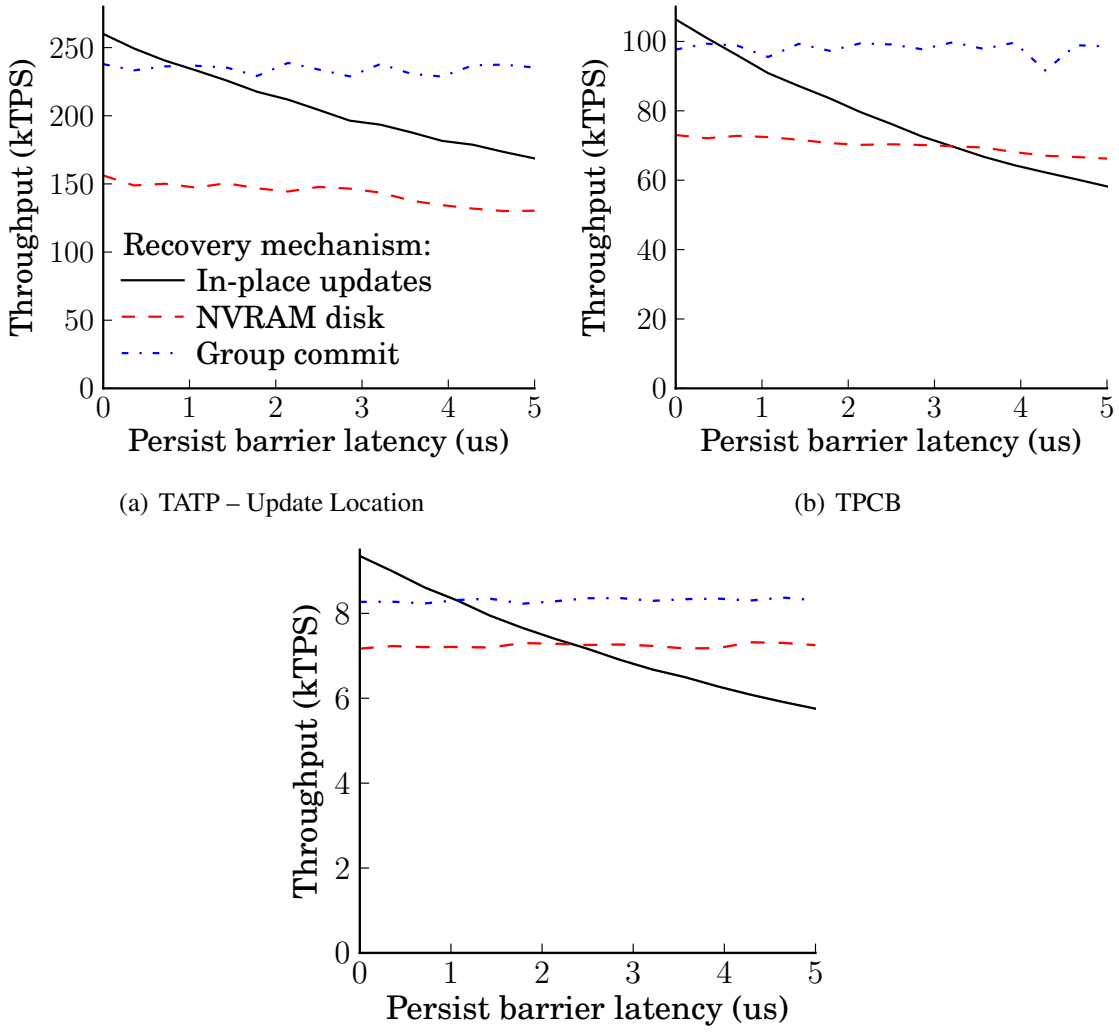
5.3.1 Persist Barrier Latency

Figure 5.4 shows throughput for write-heavy transactions as persist barrier latency increases from 0µs to 5µs, the range believed to encompass realistic latencies for possible implementations of persist barriers and storage architectures. A persist barrier latency of 0µs (left edge) corresponds to no barrier/DRAM latency. For such devices (e.g., battery-backed DRAM), *In-Place Updates* far out-paces *NVRAM Disk-Replacement*, providing up to a 50% throughput improvement. The speedup stems from a combination of removing WAL overheads, removing contention between page flushers and transaction threads, and freeing up (a few) threads from log and page flushers to run additional transactions. *In-Place Updates* also outperforms *NVRAM Group Commit*, providing an average 10% throughput improvement across workloads.

As persist barrier latency increases, each recovery mechanism reacts differently. *In-Place Updates*, as expected, loses throughput. *NVRAM Disk-Replacement* and *NVRAM Group Commit*, on the other hand, are both insensitive to persist barrier latency; their throughputs see only a small decrease as persist barrier latency increases. TATP sees the largest throughput decrease for *NVRAM Disk-Replacement* (14% from 0µs to 5µs). The decrease stems from *NVRAM Disk-Replacement*'s synchronous commits, requiring the log flusher thread to complete flushing before transactions commit. During this time, transaction threads sit idle. While both *NVRAM Disk-Replacement* and *NVRAM Group Commit* retain high throughput, there is a large gap between the two, with *NVRAM Group Commit* providing up to a 50% performance improvement over *NVRAM Disk-Replacement*. This difference, however, is workload dependent, with WAL imposing a greater bottleneck to TATP than to TPCB or TPCC.

Of particular interest are persist barrier latencies where lines intersect—the break-even points for determining the optimal recovery mechanism. Whereas all workloads prefer *In-Place Updates* for a 0µs persist barrier latency, *NVRAM Group Commit* provides better throughput above 1µs persist barrier latency. When only considering *In-Place Updates* and *NVRAM Disk-Replacement* the decision is less clear. Over the range of persist barrier latencies TATP always prefers *In-Place Updates* to *NVRAM Disk-Replacement* (the break-even latency is well above 5µs). TPCB and TPCC see the two mechanisms intersect near 3.5µs and 2.5µs, respectively, above which *NVRAM Disk-Replacement* provides higher throughput. TATP, unlike the other two workloads, only updates a single page per transaction. Other overheads tend to dominate transaction time, resulting in a relatively shallow *In-Place Updates* curve.

The previous results show throughput only for a single transaction from each workload. Table 5.6 shows break-even persist barrier latency between *NVRAM Disk-Replace-*



(a) TATP – Update Location

(b) TPCB

(c) TPCC – New Order

Figure 5.4: Throughput vs persist barrier latency. *In-Place Updates* performs best for zero-cost persist barriers, but throughput suffers as persist barrier latency increases. *NVRAM Disk-Replacement* and *NVRAM Group Commit* are both insensitive to increasing persist barrier latency, with *NVRAM Group Commit* offering higher throughput.

Workload	Full mix	Single transaction
TATP	25	12
TPCB	3.2	3.2
TPCC	3.6	2.4

Table 5.6: Break-even persist latency Persist barrier latency (µs) where *NVRAM Disk-Replacement* and *In-Place Updates* achieve equal throughput. Latencies reported for full transaction mixes and single write-heavy transaction per workload.

ment and *In-Place Updates* for these transactions and full transaction mixes. Full transaction mixes contain read-only transactions, reducing log insert and persist barrier frequency (read-only transactions require no recovery). *NVRAM Disk-Replacement* sees improved throughput at 0 µs and *In-Place Updates*'s throughput degrades less quickly as persist barrier latency increases. As a result, the break-even persist barrier latency between these two designs increases for the full transaction mix relative to a single write-heavy transaction and the opportunity to improve throughput by optimizing recovery management diminishes—improved recovery management does not affect read-only transactions and actions.

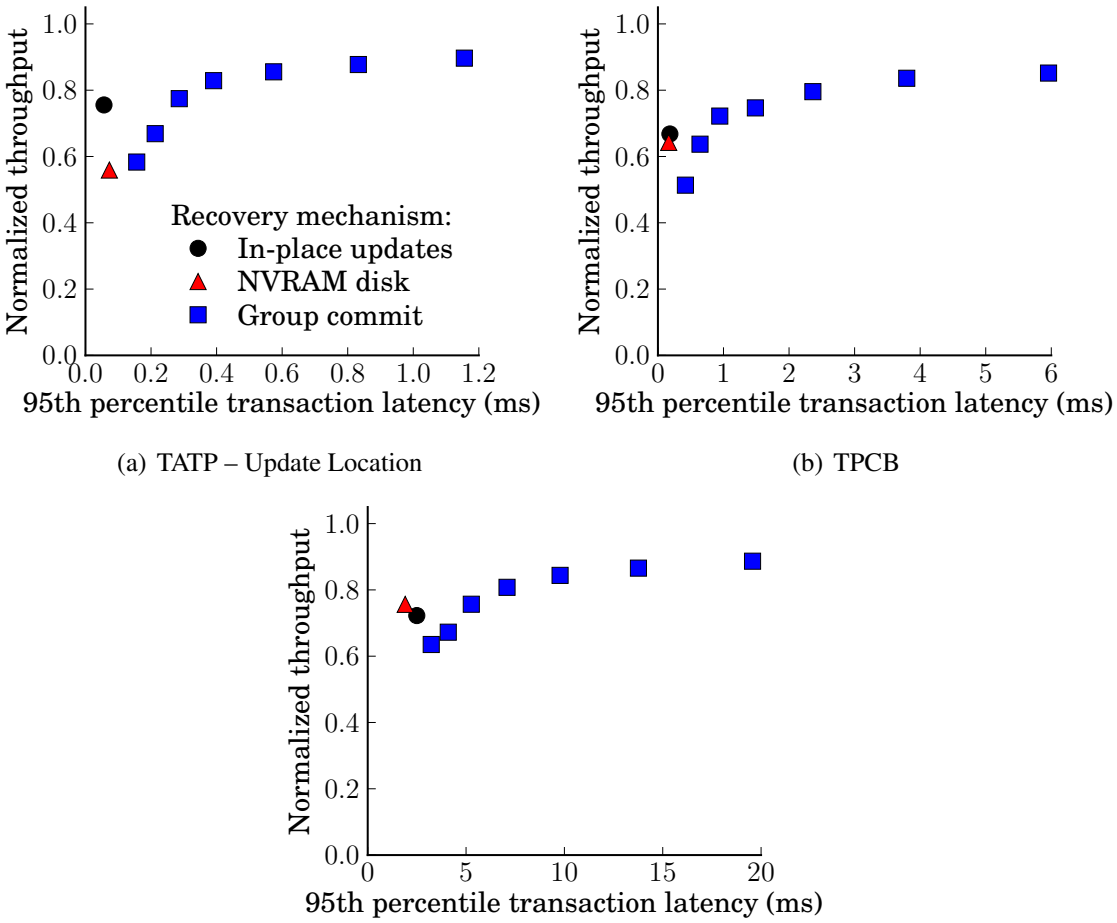
These results suggest different conclusions across storage architectures. NVRAM connected via the main memory bus will provide low latency persist barriers (less than 1µs) and prefer *In-Place Updates*. Other storage architectures, such as distributed storage, require greater delays to synchronize persists. For such devices, *NVRAM Group Commit* offers an alternative to *NVRAM Disk-Replacement* that removes software overheads inherent in WAL while providing recovery. However, *NVRAM Group Commit* increases transaction latency.

5.3.2 Transaction Latency

NVRAM Group Commit improves transaction throughput by placing transactions into batches and committing all transactions in a batch atomically. Doing so minimizes and limits the number of persist barriers. However, deferring transaction commit increases transaction latency, especially for the earliest transactions in each batch. To achieve reasonable throughput, batches must be significantly longer than average transaction latency (such that batch execution time dominates batch quiesce and persist time). The batch period acts as a knob for database administrators to trade off transaction latency and throughput. I use this knob to measure the relationship between throughput and high-percentile transaction latency.

Figure 5.5 shows throughput, normalized to *In-Place Updates* at 0µs persist barrier latency. The results consider a 3µs persist barrier latency, where *NVRAM Group Commit* provides a throughput improvement over other recovery mechanisms. The different *NVRAM Group Commit* points represent different batch periods, and I report the measured 95th percentile transaction latency for all recovery mechanisms. I measure transaction latency from the time a transaction begins to the time its batch ends (Shore-MT does not model any pre-transaction queuing time).

The results illustrate that *NVRAM Group Commit* is capable of providing equivalent throughput to the other recovery mechanisms with reasonable latency increases (no more than 5×). Further, high-percentile transaction latencies fall well below the latency expec-



(a) TATP – Update Location

(b) TPCB

(c) TPCC – New Order

Figure 5.5: 95th percentile transaction latency. All graphs are normalized to 0µs persist barrier latency *In-Place Updates* throughput. Experiments use 3µs persist barrier latency. *NVRAM Group Commit* avoids high latency persist barriers by defering transaction commit, committing entire batches atomically.

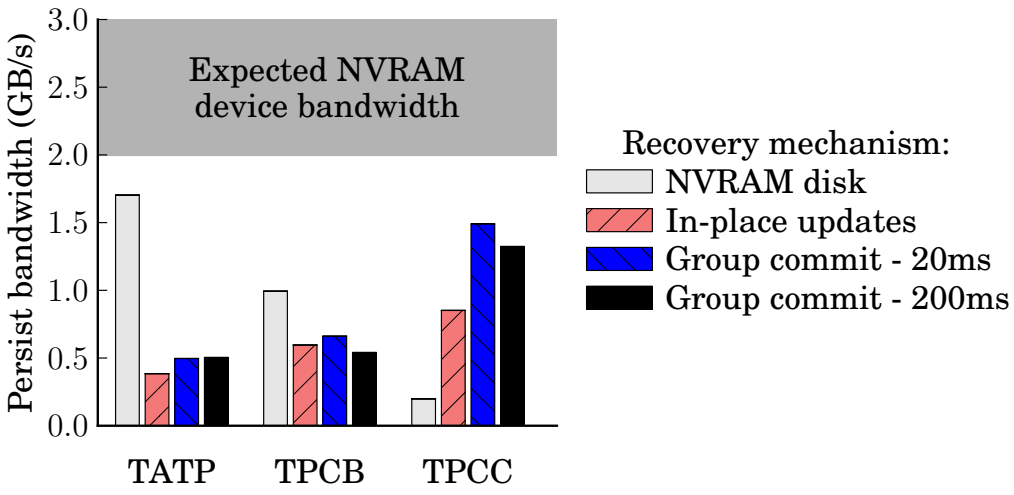


Figure 5.6: Required NVRAM bandwidth. Persist/write bandwidth required to achieve 95% performance relative to no bandwidth constraint. Bandwidth requirements are far below expected device bandwidth.

tations of modern applications. TPCC, the highest latency workload, approaches optimal throughput with a 95th percentile transaction latency of 15ms—similar to latencies incurred by disk-backed databases. For latency sensitive workloads, the batch period can be selected to precisely control latency, and *In-Place Updates* and *NVRAM Disk-Replacement* remain alternatives.

5.3.3 NVRAM Persist Limitations

In addition to persist delays NVRAM may suffer limitations due to persist bandwidth and write endurance. I quantify these limitations and determine how database design affects each.

Persist bandwidth. Different NVRAM storage architectures impose a variety of limitations on persist bandwidth (e.g., memory bus-attached NVRAM will allow greater throughput than PCIe-attached NVRAM). Different software designs place varying requirements on bandwidth.

Figure 5.6 shows the bandwidth required for each workload and recovery mechanism to achieve 95% throughput compared to same configuration with no bandwidth constraint. Each recovery mechanism sees different bandwidth requirements, and required bandwidth varies across workloads.

The *NVRAM Disk-Replacement* configuration flushes pages aggressively (attempts to flush continuously) and consumes all available bandwidth. Since bandwidth is restricted recovery latency may increase (not reflected in the results). Page flushing contends with

log flushing for bandwidth, but so long as log flushes are minimally delayed throughput will not be affected. TATP, which contains short transactions, requires the most bandwidth (1.7GB/s) to reduce transaction commit delays. I believe that asynchronous commit (allowing a new transaction to begin processing on the hardware thread while the previous transaction waits to commit) would lower the bandwidth requirement. TPCC, on the other hand, has long transactions and transaction commit is less of a concern.

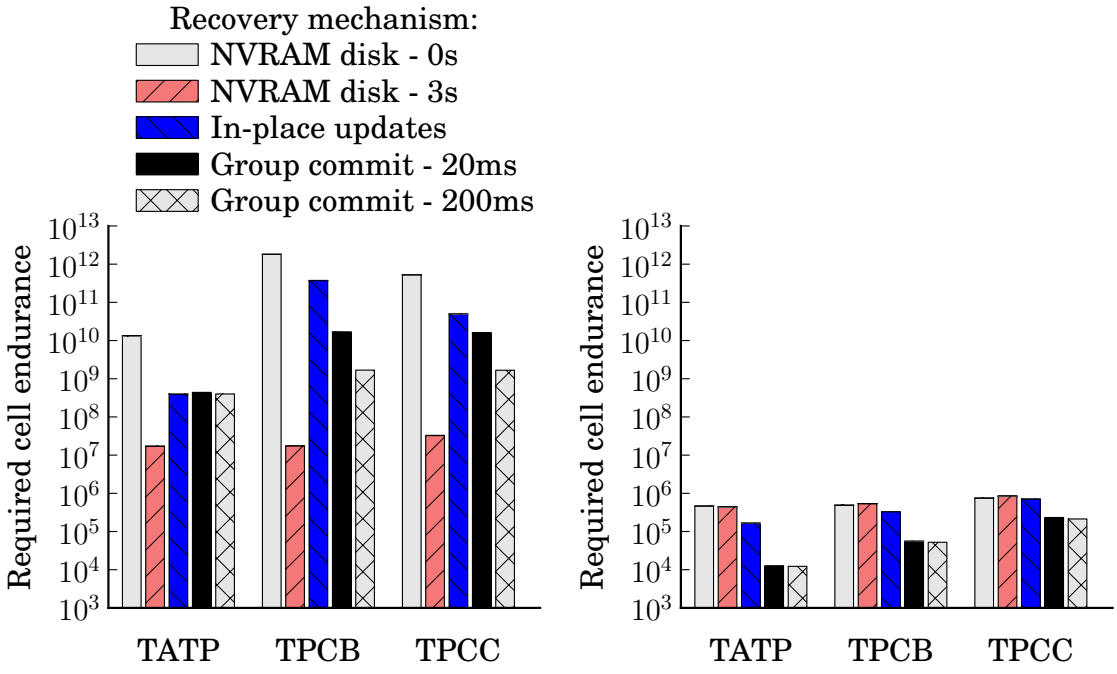
In-Place Updates requires the least bandwidth of the recovery mechanisms. TATP's Update Location transaction needs only 0.4GB/s since each transaction contains a single small update. TPCC's New Order transaction modifies much more data, requiring 0.8GB/s persist bandwidth to achieve 95% throughput.

Finally, *NVRAM Group Commit* sees varying requirements on persist bandwidth. All persists occur at batch boundaries, resulting in persist bursts. I consider two batch periods—20ms and 200ms. TATP and TPCC require relatively small bandwidths while TPCC requires over 1.5GB/s persist bandwidth to achieve 95% throughput. The bursty nature of *NVRAM Group Commit* and large modifications by TPCC's New Order transaction result in persist bandwidth quickly creating a bottleneck. Longer batches help slightly by allowing updates to coalesce within the batch, reducing the total number of bytes persisted. The bandwidth requirements would be reduced by introducing mechanisms to enable early persists (persist data before the batch ends), removing persist bursts.

While persist bandwidth requirements vary, the strictest configuration (*NVRAM Disk-Replacement* for TATP) requires only 1.7GB/s. Such bandwidth is currently possible with existing PCIe Flash memory SSDs, and I believe is attainable for all candidate NVRAM technologies. I expect that persist bandwidth will not be a concern for OLTP systems using NVRAM.

Device lifetime. NVRAM technologies, much like Flash memory currently, will have limited write endurance; each cell may only be reliably written a finite number of times. Device lifetime is generally determined by the most frequently written addresses. Previous work proposes hardware techniques to evenly distribute writes amongst cells by constantly changing the mapping between between memory addresses and the underlying memory cell [87]. These techniques prolong device lifetime by ensuring that frequently written addresses do not wear out individual cells. While the recovery mechanisms presented here each produce varying persist rates with different abilities to naturally distribute persists across addresses, it is unclear if software mechanisms are sufficient to prolong device lifetime.

Figure 5.7 shows the required cell endurance (number of persists) to achieve a 10-year device lifetime assuming the database runs at maximum throughput continuously. All re-



(a) No wear leveling

(b) Perfect wear leveling

Figure 5.7: NVRAM cell endurance for 10-year lifetime. Without hardware wear leveling the most-written cell limits device lifetime. *NVRAM Disk-Replacement* requires conservative page flushing (3s recovery latency vs 0s, instantaneous, recovery) and *NVRAM Group Commit* requires longer batch periods (200ms vs 20ms) to improve device lifetime. We expect hardware wear leveling to always be required with *In-Place Updates*. With perfect wear leveling (all writes occur evenly throughout cells in the device) and a 32GB storage device all workloads and configurations achieve a 10-year device lifetime for cell endurance of 10^6 writes and greater.

covery mechanisms assume *In-Place Updates*'s throughput for a more fair comparison. Persists are tracked at byte granularity. I consider two scenarios: no wear leveling—lifetime is limited by the most frequently written address—and perfect wear leveling—all persists are evenly distributed across all physical addresses (provided in hardware). I consider *NVRAM Disk-Replacement* with both instantaneous recovery and 3s recovery latency, *In-Place Updates*, and *NVRAM Group Commit* with 20ms and 200ms batch latencies. Logs are implemented as circular buffers. Thus, logs do not contribute to the most written byte and have no bearing on device lifetime when hardware wear leveling is unavailable. Perfect wear leveling assumes a 32GB device.

Without hardware wear leveling *NVRAM Disk-Replacement* with instantaneous recovery (aggressive page flushing) and *In-Place Updates* require the greatest cell endurance. Both mechanisms persist each update directly to the NVRAM store. *NVRAM Disk-Replacement* must additionally persist LSNs, which constitute the most frequently persisted addresses, while LSNs are no longer used with *In-Place Updates* and *NVRAM Group Commit*. Both of these configurations require greater cell endurance far greater than 10^8 writes, the endurance expected of phase change memory [14].

NVRAM Disk-Replacement with 3s recovery latency requires much lower cell endurance for a 10 year lifetime. Writes to heap pages coalesce and only occasionally persist to NVRAM. *NVRAM Disk-Replacement* effectively increases device lifetime by reducing the total number of persists and limiting the rate that any single page may write back. However, doing so requires an increase in recovery latency. *NVRAM Disk-Replacement* may possibly provide acceptable software wear leveling so long as recovery latency is reasonable.

Finally, *NVRAM Group Commit* provides wear leveling by only allowing a single persist to each address per batch (all other persists coalesce). However, neither 20ms nor 200ms batches is capable of coalescing enough writes to guarantee a 10 year device lifetime. *NVRAM Group Commit* would require far too large a batch period, increasing transaction latency, to reliably wear-level NVRAM.

When hardware distributes persists evenly across NVRAM cells all recovery mechanisms achieve a 10 year lifetime assuming 10^8 writes per cell. In this case lifetime is determined by the total number of persists rather than the most-persisted address. Due to the inability of software to reliably and sufficiently extend device lifetime, and the effectiveness of hardware wear leveling, I expect future NVRAM SSDs to include hardware wear leveling.

5.3.4 Summary

Persist barriers used to enforce persist order pose a new performance obstacle to providing recoverable storage management with NVRAM. I show that, for memory bus-attached NVRAM devices, ensuring recovery using *In-Place Updates* is a viable strategy that provides high throughput and removes the overheads of WAL. For interconnects and NVRAM technologies that incur larger persist barrier delays, *NVRAM Group Commit* offers an alternative that yields high throughput and reasonable transaction latency. *NVRAM Group Commit*'s batch period allows precise control over transaction latency for latency-critical applications. Finally, I demonstrate that persist bandwidth is not a concern, and that while *NVRAM Disk-Replacement* may provide an effective mechanism to cope with limited write endurance I expect future devices to include hardware wear leveling.

5.4 Conclusion

New NVRAM technologies offer an alternative to disk that provides high performance while maintaining durable transaction semantics, yet existing database software is not optimized for such storage devices. In this chapter, I evaluated recovery management to optimize for NVRAM read and persist characteristics. I found that even small caches effectively reduce NVRAM read stalls. I also considered database performance in the presence of persist barrier delays. Treating NVRAM as a drop-in replacement for disk, *NVRAM Disk-Replacement* retains centralized logging overheads. *In-Place Updates* reduces these overheads, but for large persist barrier latencies suffers from excessive synchronization stalls. I proposed a new recovery mechanism, *NVRAM Group Commit*, to minimize these stalls while still removing centralized logging. While *NVRAM Group Commit* increases high-percentile transaction latency, latency is controllable and within modern application constraints.

This work assumed that persist barriers are expensive, stalling instruction and thread execution at each barrier. However, it is possible for systems to implement high performance barriers that overlap instruction execution with persists while still enforcing proper persist order. Additionally, persist barriers must interact with existing memory execution and the memory consistency model to enforce an order between persists from different threads. The remainder of this dissertation investigates programming interfaces that to enforce persist order while still maximizing persist concurrency.

CHAPTER VI

Memory Persistency

The previous chapters considered how to design recoverable software assuming various latencies for persist barriers. However, the memory system implementation remained unspecified, and ultimately will determine performance. The remainder of this dissertation in turn investigates programming interfaces for persistent memory and how the interface might affect system performance.

This chapter motivates and defines *memory persistency*, a framework for reasoning about and enforcing the order of persist operations. The following chapters propose and evaluate example persistency models. This work was completed in collaboration with my advisor, Thomas F. Wenisch, and Professor Peter M. Chen. Refer to Section 2.4 for a discussion of memory consistency and consistency models.

6.1 Introduction

Emerging nonvolatile memories (NVRAM) promise high performance recoverable systems. These technologies, required as replacements for Flash and DRAM as existing technologies approach scaling limits [58], pair the high performance and byte addressability of DRAM with the durability of disk and Flash memory. Future systems will place these devices on a DRAM-like memory bus, providing systems with throughput similar to DRAM, yet recoverability after failures.

However, ensuring proper recovery requires constraining the order of NVRAM writes. Existing DRAM architectures lack the interface to describe and enforce write ordering constraints; ordering constraints that arise from memory consistency requirements are usually enforced at the processor, which is insufficient for failure tolerance with acceptable performance. Recent work has suggested alternative interfaces to enforce NVRAM write order and guarantee proper recovery, for example durable transactions and persist barriers [116, 30]. While intuitive and suitable to specific applications, I wish to investigate

a more general framework for reasoning about NVRAM write ordering including mechanisms for expressing write constraints that are independent of specific concurrency control mechanisms.

Instead, I recognize that the problem of constraining NVRAM write concurrency resembles memory consistency. Memory consistency restricts the visible order of loads and stores (equivalently, allowable visible memory states) between processors or cores, allowing operations to reorder so long as expected behavior is guaranteed. Memory consistency models provide an interface and set of memory order guarantees for the programmer, but separate the implementation; several distinct implementations may fulfill the same memory consistency model, allowing sophisticated optimization (e.g., speculation [11, 119, 19, 45, 92]). Relaxing the memory consistency model places an additional burden on the programmer to understand the model and insert the correct annotations, but often allows greater performance.

I introduce *Memory Persistency*, a framework motivated by memory consistency to provide an interface for enforcing the order in which NVRAM writes become durable, an operation referred to as a "persist" (as in previous chapters). Memory persistency prescribes the order of persist operations with respect to one another and loads and stores, and allows the programmer to reason about guarantees on the ordering of persists with respect to system failures. The memory persistency model relies on the underlying memory consistency model and volatile memory execution to define persist ordering constraints and the values written to persistent memory.

In the following chapters, I define memory persistency, describe the design space of memory persistency models, and introduce and evaluate several new persistency models. Much like consistency, I identify *strict* and *relaxed* classes of persistency models. Strict persistency relies on implicit guarantees to order persists and couples persistent semantics to the underlying memory consistency model: any two stores to the persistent address space that are guaranteed to be observed in a well-defined order from the perspective of a third processor imply well-ordered NVRAM writes. Thus, the same mechanisms the consistency model provides a programmer to enforce order for stores also enforce order for the corresponding persists. Alternatively, relaxed persistency separates volatile and persistent memory execution, allowing the order of persist operations to differ from the order in which the corresponding stores become visible. Relaxed persistency facilitates concurrent persists even when volatile memory operations become visible according to sequential consistency. Memory persistency further provides an interface for uniprocessors, which may not ordinarily order stores, to specify persist ordering constraints. While separating memory consistency and persistency provides advantages to programmability and performance,

it also introduces new challenges, as separate annotations define allowable reorderings for visibility and persistence of writes to the persistent address space.

Using this framework, I introduce successively relaxed memory persistency models and demonstrate how programmers can exploit the reorderings they allow through several example implementations of a thread-safe persistent queue. I demonstrate that conservative memory consistency (such as sequential consistency) with strict persistency must rely on thread parallelism to enable NVRAM write concurrency. On the other hand, relaxed persistency allows high instruction execution performance, NVRAM write concurrency, and simplified data structures.

Finally, I evaluate my memory persistency models and queue designs. Just as with memory consistency, a memory persistency model is defined separately from its implementation. Instead of assuming specific storage technologies and memory system implementations, I measure NVRAM write performance as the critical path of persist ordering constraints, assuming that NVRAM writes form the primary system bottleneck and that practical memory systems effectively use available concurrency. I demonstrate that relaxed persistency models substantially improve write-concurrency over sequential consistency with strict persistency; for a 500ns NVRAM write latency, these concurrency gains improve performance to the throughput limit of instruction execution—as much as 30x speedup over strict persistency.

6.2 Memory Persistency Goals

Correct recovery of durable systems requires persists to observe some set of happens-before relations, for example, that persists occur before an externally observable action, such as a system call. However, I expect NVRAM writes to be much slower than writes to volatile memory. Provided the minimal set of happens-before relations is observed, the gap between volatile execution and NVRAM write performance can be shrunk by optimizations that increase concurrency. I am interested in defining persistency models that create opportunities for two specific optimizations: *persist buffering*, and *persist coalescing*. Additionally, I intend to propose persistency models that maximize instruction execution throughput and preserve existing thread synchronization patterns, including synchronization to the persistent address space.

Persist Buffering. Buffering durable writes and allowing thread execution to proceed ahead of persistent state greatly accelerates performance [30]. Such buffering overlaps NVRAM write latency with useful execution. To be able to reason about buffering, I draw a distinction between a "store", the cache coherence actions required to make a write (in-

cluding an NVRAM write) visible to other processors, and a "persist", the action of writing durably to NVRAM. Buffering permits persists to occur asynchronously.

Ideally, persist latency is fully hidden and the system executes at native volatile memory speed. With finite buffering, performance is ultimately limited by the slower of the average rate that persists are generated (determined by volatile execution rate) and the rate persists complete. At best, the longest chain (critical path) of persist ordering constraints determines how quickly persists occur (at worst, constraints within the memory system limit persist rate, such as bank conflicts or bandwidth limitations). In defining persistency models, my goal is to admit as much persist concurrency as possible by creating a memory interface that avoids unnecessary constraints. Persistency model implementations might buffer persists in existing store queues and caches or via new mechanisms, such as buffers within NVRAM devices.

Persist Coalescing. I expect that NVRAM devices will guarantee atomicity for persists of some size (e.g., eight-byte atomicity [30]). Persists to the same atomically persistable block may coalesce (be performed in a single persist operation) provided no happens-before constraints are violated. Persist coalescing creates an opportunity to avoid frequent persists to the same address, and allows caching/buffering mechanisms to act as bandwidth filters [46]. Coalescing also reduces the total number of NVRAM writes, which may be important for NVRAM devices that are subject to wear. Larger atomic persists facilitate greater coalescing. Similarly, persistency models that avoid unnecessary ordering constraints allow more coalescing.

While similar gains may be achieved through software (by caching values in nonpersistent memory and precisely controlling when they are persisted), I believe that hardware-enabled persist coalescing is an important feature. Automatic coalescing relieves the programmer of the burden to manually orchestrate coalescing and specify when persists occur via copies. Additionally, automatic coalescing provides backwards compatibility by allowing new devices to increase the size of atomic persists to improve coalescing performance. I do not consider specific hardware implementations to detect when persists may coalesce, but assume that it is an important property of high performance persistent memory systems.

Minimizing non-persistent overheads. Existing data structures intended for volatile memory must be modified for persistent memory, for example, by adding logging or copy-on-write to make persistent updates atomic with respect to recovery. Additionally, new synchronization delays will be introduced with strict persistency if store barriers are inserted to enforce persist order (under strict persistency store barriers are also persist barriers). Data structures modified to be recoverable will ultimately run more slowly even on existing volatile memory systems. The persistency model will determine how invasive

these software transformations are as well as the extent of their performance impact, thus motivating the need to explore the design space of persistency models.

Finally, persistent memory systems must continue to support advanced thread synchronization mechanisms. One area that has not yet been investigated is synchronization to persistent memory, including locks, atomic read-modify-write (e.g., add, exchange, test-and-set), and compare-and-swap. While this work does not attempt to use these primitives, I intend to introduce persistency models that properly define their with NVRAM. Extending existing consistency models provides a framework to define persistent behavior for atomic synchronization primitives.

6.3 Memory Persistency

Recovery mechanisms require specific orders of persists. Failure to enforce this order results in data corruption. A persistency model enables software to label those persist-order constraints necessary for recovery-correctness while allowing concurrency among other persists. As with consistency models, my objective is to strike a balance between programmer annotation burden and the amount of concurrency (and therefore improved performance) the model enables.

I introduce memory persistency as an extension to memory consistency to additionally order persists and facilitate reasoning about persist order with respect to failures. Conceptually, I reason about failure as a *recovery observer* that atomically reads all of persistent memory at the moment of failure. Ordering constraints for correct recovery thus become ordering constraints on memory and persist operations as viewed from the recovery observer. With this abstraction, one may apply the reasoning tools of memory consistency to persistency—any two stores to the persistent memory address space that are ordered with respect to the recovery observer imply an ordering constraint on the corresponding persists. Conversely, stores that are not ordered with respect to the observer allow corresponding persists to be reordered or performed concurrently. The notion of the recovery observer implies that even a uniprocessor system requires memory persistency as the single processor must still interact with the observer (i.e., uniprocessor optimizations for cacheable volatile memory may be incorrect for persistent memory).

Much like consistency models, there may be a variety of implementations for a particular memory persistency model. Like the literature on consistency, I separate model semantics from implementation; my focus in this work is on exploring the semantics. While I do discuss some implementation considerations, I omit details and leave system design and optimization to future work. I divide persistency models into *strict* and *relaxed* classes, and

consider each with respect to the underlying consistency model.

6.3.1 Strict Persistency

The most intuitive memory persistency model is *strict persistency*. Strict persistency couples memory persistency to the memory consistency model, using the existing consistency model to specify persist ordering. Minimal programmer annotations are required—while processor memory barriers are unnecessary, compiler barriers prevent optimizations that interfere with proper recovery. Under strict persistency, the recovery observer participates in the memory consistency model precisely as if it were an additional processor. Hence, any store ordering that can be inferred by observing memory order implies a persist ordering constraint. Persist order must match the (possibly partial) order in which stores are performed in a particular execution.

Conservative consistency models, such as SC, do not allow stores from each thread to reorder from the perspective of other threads; all stores, and therefore persists, occur in each thread's program order. However, such models can still facilitate persist concurrency by relying on thread concurrency (stores from different threads are often concurrent). On the other hand, relaxed consistency models, such as RMO, allow stores to reorder. Using such models it is possible for many persists from the same thread to occur in parallel. However, the programmer is now responsible for inserting the correct memory barriers to enforce the intended behavior, as is currently the case for shared-memory workloads.

Strict persistency unifies the problem of reasoning about allowable memory order and allowable persist order (equivalently, allowable persistent states at recovery); a program that is correctly synchronized with respect to the consistency model is also correctly labeled with respect to the recovery observer. However, directly implementing strict persistency implies frequent stalls—consistency ordering constraints (e.g., at every memory operation under SC and at memory barriers under RMO) stall execution until NVRAM writes complete. A programmer seeking to maximize persist performance must rely either on relaxed consistency (with the concomitant challenges of correct program labeling), or must aggressively employ thread concurrency to eliminate persist ordering constraints. As I will show, decoupling persistency and consistency ordering allows recoverable data structures with high persist concurrency even under SC.

I introduce one important optimization to strict persistency, *buffered strict persistency*, which can improve performance while still guaranteeing strict ordering of persists and visible side effects. Buffered strict persistency allows instruction execution to proceed ahead of persistent state, thus allowing overlap of volatile execution and serial draining of queued persist operations. In terms of the recovery observer, buffered strict persistency allows the

observer to lag arbitrarily far behind other processors in observing memory order. Therefore, the persistent state of the system corresponds to some prior point in the observable memory order. As side effects may otherwise become visible prior to earlier persists, I additionally introduce a *persist sync* operation to synchronize instruction execution and persistent state (i.e., require the recovery observer to "catch up" to present state). The persist sync allows the programmer to order persists and non-persistent, yet visible, side effects.

6.3.2 Relaxed Persistency

Strict persistency provides mechanisms to reason about persist behavior using pre-existing memory consistency models. However, memory consistency models are often inappropriate for persist performance. Conservative consistency models such as SC and TSO serialize the visible order of stores; while high performance implementations of these models exist for volatile instruction execution [102, 63], the high latency of NVRAM persists suggests that more relaxed persistency models may be desirable.

I decouple memory consistency and persistency models via *relaxed persistency.* Relaxed persistency loosens persist ordering constraints relative to the memory consistency model—that is, the visible order of persists (from the perspective of the recovery observer) is allowed to deviate from the visible order of stores. Relaxing persistency requires separate memory consistency and persistency barriers. Memory consistency barriers enforce the visibility of memory operation ordering with respect to other processors, while memory persistency barriers constrain the visible order of persists from the perspective of only the recovery observer.

Relaxing persistency allows systems with conservative consistency, such as SC, to improve persist concurrency without requiring additional threads or complicating thread communication. In later chapters I introduce and explore several relaxed persistency models under SC.

Simultaneously relaxing persistency and consistency allows both the visibility of loads and stores to reorder among processors, and further allows the order of persists to differ from the order of stores. An interesting property of such systems is that memory consistency and persistency barriers are decoupled—store visibility and persist order are enforced separately—implying that persists may reorder across store barriers and store visibility may reorder across persist barriers. Separating store and persist order complicates reasoning about persists to the same address, as I show next.

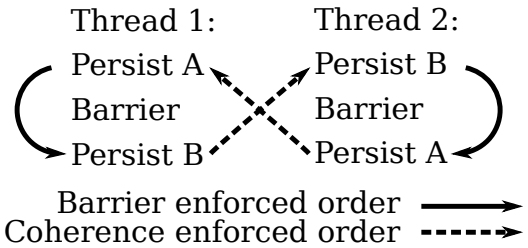


Figure 6.1: Cache Coherence Ordered Persists. Thread 1's store visibility reorders while still attempting to enforce persist order. The resulting persist order cycle is resolved by violating cache coherence persist order or by preventing stores from reordering across persist barriers.

6.3.3 Relaxed Persistency and Cache Coherence

Under SC, both strict and relaxed persistency imply that persists to a single address are totally ordered by cache coherence. A recovery observer can only view stores to a single address in the order observed by cache coherence, and therefore the associated persists must follow this same order. On the other hand, allowing separate memory consistency and persistency barriers can allow persists to occur in an order other than that observed by cache coherence—a behavior that may astonish programmers. This behavior is desirable when synchronization (such as a lock) already guarantees that races cannot occur; treating all persist barriers additionally as consistency store barriers would unnecessarily delay instruction execution.

The example in Figure 6.1 demonstrates that it is not possible to simultaneously (1) allow store visibility to reorder across persist barriers, (2) enforce persist barriers, and (3) enforce persist order to a single address according to the order cache coherence observes those stores. The example considers two distinct persistent objects, A and B. Threads 1 and 2 persist to these objects in different program orders. Thread 1's execution reorders the visibility of stores, while Thread 2 executes its stores in program order. Note that the persist barrier implies that thread 1's persist to B occurs after its persist to A, but the values produced by these operations may become visible to other processors out of program order.

Figure 6.1 additionally annotates the persist order constraints (happens-before relationships) due to persist barriers and cache coherence store order. As shown, these constraints form a cycle and hence cannot be enforced. The cycle can be resolved by either coupling persist and store barriers—every persist barrier also prevents store visibility from reordering—or relaxing cache coherence persist order and providing additional mechanisms to order persists across threads. It remains unclear how to best solve this dilemma.

Store atomicity. Cache coherence provides *store atomicity* for memory consistency models and their resulting memory orders [2]. Store atomicity requires that stores become

visible in their entirety (no partial stores) and that stores are atomically visible to all processors, and thus stores to each address are serializable. Store atomicity is violated when a store is visible to only a subset of processors (although many memory systems allow store buffering and forwarding within the same processor without violating store atomicity). Similar principles apply to memory persistency.

I define a memory persistency model to be *persist-atomic* if persists to the same address are serializable. Equivalently, a model is persist-atomic if recovery establishes a single unique value for each address in the persistent address space. While persist atomicity requires that persists of a certain size occur atomically, persist atomicity additionally places constraints on the order of persists. Our failure model, the recovery observer, implies persist atomicity (each processor sees stores and persists in a well defined order). However, more complex failure models may violate persist atomicity if processors recover different values for the same address (e.g., processors recover using persistent per-processors logs—similar behavior in distributed caches violates store atomicity).

Store and persist atomicity are useful features for defining intuitive consistency and persistency models. However, it is possible for a memory model to exhibit both store and persist atomicity, but for the order of stores and persists to deviate. In such cases persists to the same address are reordered relative to their underlying stores but still retain some total order. I define *strong persist atomicity* as memory models that provide store and persist atomicity and where the order of stores to each address agrees with the order of persists. All persistency models discussed in the remainder of this study provide strong persist atomicity.

6.4 Conclusion

Existing memory systems lack mechanisms to describe and enforce the order of persists. I introduce memory persistency to explore the design space of interfaces and semantics for persistent memory. Memory persistency builds on memory consistency to determine the values for and order of persist operations. Memory persistency may be strict, relying on the consistency model to provide both thread and persist synchronization, or relaxed, decoupling thread synchronization from persist order and requiring additional persist barriers. The next chapter introduces three persistency models and details their expected performance using a thread-safe persistent queue.

CHAPTER VII

Memory Persistency Models

This chapter provides examples of memory persistency models and data structures using the memory persistency framework defined in the previous chapter. I first introduce a concurrent, persistent queue, outlining several design choices and reasoning about persist performance. I then define specific memory persistency models under sequential consistency (SC), demonstrating their use with the persistent queue. The final chapter of this dissertation evaluates these persistency models using the persistent queue.

7.1 Persistent Queue

To understand and evaluate persistency models I first introduce a motivating benchmark: a thread-safe persistent queue. Several workloads require high-performance persistent queues, such as write ahead logs (WAL) in databases and journaled file systems. Previous work investigated the design of an NVRAM log assuming byte-addressable NVRAM with a persist barrier [37]. I extend this work, outlining three queue designs with several persistency models.

Fundamentally, a persistent queue inserts and removes entries while maintaining their order. The queue must recover after failure, preserving proper entry values and order.

My goals in designing a persistent queue are to (1) maximize the instruction execution rate, including multi-thread throughput, (2) improve the persist concurrency of insert operations through greater thread concurrency, and (3) further improve persist concurrency using relaxed persistency. All designs are concurrent (thread-safe) but allow varying degrees of persist concurrency. Additionally, the three designs are fashioned as circular buffers, containing a data segment and head and tail pointers. Psuedo-code for the three designs is shown in Figure 7.1. I outline their execution, recovery, and the minimal necessary persist dependences.

Figure 7.1: Psuedo-code for insert operations. I include the annotations required by relaxed persistency models, discussed in Section 7.2. *PersistBarrier* applies to epoch persistency and strand persistency, *NewStrand* applies only to strand persistency.

Require: *head* is a persistent pointer, *data* a persistent array.

```
 1: sl ← SIZEOF(length)
 2: function INSERTCWL(length, entry)
 3:     LOCK(queueLock)
 4:     NEWSTRAND
 5:     COPY(data[head], (length, entry), length + sl)
 6:     PERSISTBARRIER
 7:     head ← head + length + sl
 8:     UNLOCK(queueLock)
 9: end function
10:
11: function INSERT2LC(length, entry)
12:     LOCK(reserveLock)
13:     start ← headV; headV ← headV + length + sl
14:     node ← insertList.APPEND(headV)
15:     UNLOCK(reserveLock)
16:     NEWSTRAND
17:     COPY(data[head], (length, entry), length + sl)
18:     LOCK(updateLock)
19:     (oldest, newHead) ← insertList.REMOVE(node)
20:     UNLOCK(updateLock)
21:     if oldest then
22:         PERSISTBARRIER
23:         head ← newHead
24:     end if
25:     unlock(reserveLock)
26: end function
27:
28: function INSERTQH(length, entry)
29:     LOCK(queueLock)
30:     NEWSTRAND
31:     start ← head; endPos ← start + length + sl
32:     data[start] ← length; data[endPos] ← 0
33:     PERSISTBARRIER
34:     head ← head + length + sl + 1
35:     UNLOCK(queueLock)
36:     COPY(data[start + sl], entry, length)
37:     PERSISTBARRIER
38:     data[endPos] ← 1
39: end function
```

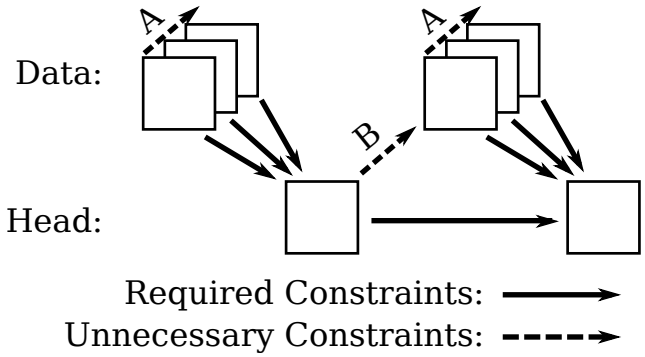


Figure 7.2: Queue Persist Dependences. Persist ordering dependences for *Copy While Locked* and *Two-Lock Concurrent*. Constraints necessary for proper recovery shown as solid arrows; unnecessary constraints incurred by strict persistence appear as dashed arrows and are labeled as A (removed with epoch persistency) and B (further removed by strand persistency).

The first design, *Copy While Locked* (CWL), serializes insert operations with a lock, first persisting each entry's length and data to the queue's data segment, then persisting the new head pointer. As a result, persists from subsequent insert operations, even if they occur on separate threads, are ordered by lock accesses. If the systems fails before the persist to the head pointer in line 6, the entry is ignored and the insert has failed.

I improve persist concurrency in the second design, *Two-Lock Concurrent* (2LC), by using two different locks to reserve data segment space and persist to the head pointer, respectively. Neither lock is held while entry data persists to the data segment, allowing concurrent persists from different threads. Additionally, a volatile *insert list* is maintained to detect when insert operations complete out of order and prevent holes in the queue. The double-checked locking pattern prevents data races while updating the insert queue. *Two-Lock Concurrent* employs the same recovery as *Copy While Locked*—an entry is not valid and recoverable until the head pointer encompasses the associated portion of the data segment.

Both queue designs use the persistency model to prevent persists to the head pointer from occurring before persists to the data segment. Figure 7.1 includes barriers for two different persistency models (described later in Section 7.2). Additionally, persist dependences (and unnecessary constraints introduced by strict persistency models) are shown in Figure 7.2. Recovery requires that persists to the head pointer are ordered after persists to the data segment from the same insert operation and persists to the head pointer occur in insert-order to prevent holes in the queue (persists to the head pointer may coalesce so long as no ordering constraint is violated). All other persists within the same insert operation and between operations may occur concurrently without compromising recovery correctness.

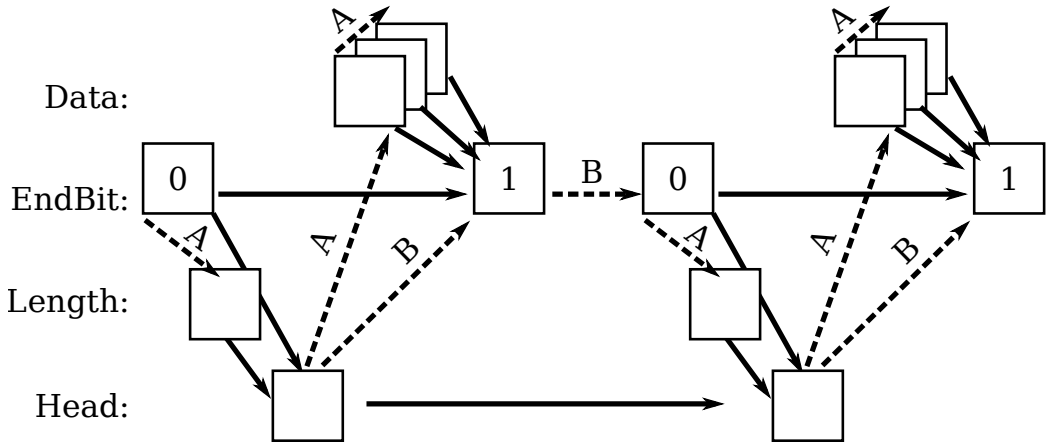


Figure 7.3: *Queue Holes* Dependences. The head pointer may not persist before *endBit* and length; final *endBit* may not persist until the entry persists to the data segment. Strict persistency introduces several unnecessary constraints.

While not necessary for correct recovery, these persist dependences are difficult to describe minimally; ordering mechanisms often introduce unnecessary persist constraints (dashed lines in the Figure). Persistency models that enforce program order of persists in each thread must serialize persists for the data of each entry to the data segment (when the entire entry cannot be persisted atomically), shown as "A" in the Figure. Additionally, persists to the data segment may be ordered after a previous insert's persist to the head pointer, denoted as "B" in the Figure.

The previous designs trade off concurrency and complexity. *Copy While Locked*, while simple, serializes persists between insert operations. On the other hand, *Two-Lock Concurrent* allows greater concurrency, but requires two locks be acquired per insert and a volatile *insert list* be maintained. I consider a third design, *Queue Holes*, that provides both improved persist concurrency and a high execution rate (first introduced in [37]).

Queue Holes prepends each queue entry with its length and appends the entry with an endBit. A single lock is held while reserving queue space and updating the head pointer, but is released prior to persisting the entry into the data segment. An entry is recovered after failure if the entry is located within the region indicated by the head pointer (as in CWL and 2LC) and the entry's endBit is set. The minimal set of persist dependences necessary for correct recovery are shown in Figure 7.3. The length and cleared endBit persist before the head pointer; the data segment persists before setting the endBit; and persists to the same address occur in the order observed by cache coherence (endBits and the head pointer). An inserted entry is recoverable only after the persist at line 37 completes, even though the head pointer is persisted earlier at line 33.

As with the other queue designs many unnecessary persist constraints may be intro-

duced by strict persistency models, shown as dashed lines in Figure 7.3. Many constraints are introduced by persistency models that enforce the program order of persists, labeled "A." Additional constraints may be introduced between persists within an insert operation or across insert operations, labeled "B."

7.2 Memory Persistency Models

Section 6.3 outlined potential classes of persistency models. I now introduce several specific persistency models to be evaluated in the next chapter. All models assume SC as the underlying memory consistency model, and successively relax persistency to introduce specific optimizations. For each model I discuss its motivation, give a definition, describe necessary annotations for and performance of the persistent queues, and offer possible implementations.

7.2.1 Strict Persistency

Motivation. The first persistency model is Strict Persistency, as discussed in Section 6.3. Strict persistency simplifies reasoning about persist ordering by coupling persist dependences to the memory consistency model. No additional persist barriers are required, easing the burden on the programmer. While strict persistency provides an intuitive first model, under SC, it imposes persist ordering constraints that unnecessarily limit persist concurrency for many data structures, and requires programmers to resort to multithreading to obtain concurrency.

Definition. Under strict persistency, persist order observes all happens-before relations implied by execution's dynamic order of memory operations as viewed by the recovery observer. Thus, all persists are ordered with respect to the program order of the issuing thread. Note that, like store operations, persists from different threads that are unordered by happens-before (i.e., the recovery observer cannot distinguish which is first) are concurrent.

Persist Performance. Strict persistency under SC introduces many unnecessary persist dependences. Consequently, strict persistency must rely entirely on thread concurrency to enable concurrent persists. Figures 7.2 and 7.3 illustrate these unnecessary dependences and their causes. Lacking mechanisms to relax persist ordering, strict persistency under SC introduces all the shown dependences (dashed lines).

These dependences are introduced because persists occur in program order under SC. This persistency model lacks the ability to declare two persists from the same thread to be concurrent. However, persist concurrency may be created by using multiple threads

to concurrently insert into the queue. *Two-Lock Concurrent* and *Queue Holes* each allow concurrent inserts, and thus persists from different threads into the data segment are concurrent—The dashed line labeled "B" in Figure 7.2 no longer implies a constraint.

Implementation. A straight-forward implementation of strict persistency stalls issue of subsequent memory accesses until a store and its corresponding persist both complete. Conventional speculation mechanisms may allow loads to speculatively reorder with respect to persistent stores [43]. Buffered strict persistency can be implemented by serializing persists to a single, totally ordered queue in front of persistent memory (e.g., in a bus-based multiprocessor, persists can be queued after they are serialized by the bus). Delays still occur when buffers fill, to drain the queue at persist sync instructions, or under contention to the persist queue.

More advanced implementations might consider distributed queues (i.e., one queue per thread/core) or extensions to the existing cache system. Mechanisms must be introduced to ensure that persists of each thread occur in program order and that persists are ordered by conflicting accesses from different threads, but persists from several threads may occur concurrently (they are not serialized) while persists buffer and execution proceeds ahead of persistent state. Under the definition of SC such implementations must detect load-before-store races between threads, ordering persists prior to the load (on the first thread) before persists after the store (on the second thread). A single store may introduce persist ordering constraints with many threads, each having loaded data from the store's address. It is unclear how to design a system that satisfies these constraints without frequent delays.

While insufficient for SC, TSO might be implemented by recording a "persist counter" and thread in each cache line. Each time a thread persists it increments its persist counter, and each store (both volatile stores and persists) writes the thread and persist counter into the cache line. Persists drain in program order by maintaining a persist queue per thread/core. Persist order between threads is enforced at each load and store by observing the previously recorded thread and counter in the operation's cache line—all subsequent stores from the executing thread must occur after the last persist to the cache line. These inter-thread persist constraints may be recorded in threads' queues or by delaying immediately until the other thread's queue drains. Such a system resembles BPFS (discussed next in Section 7.2.2 and later in Section 7.3) where each persist occurs in its own epoch [30].

Strict persistency under SC may also be implemented using in-hardware NVRAM logs or copy-on-write and indirection to give the appearance of SC while persists occur concurrently. Hardware must ensure both that persists do not occur out of program order for any thread and that conflicting accesses properly enforce persist order across threads. An intriguing possibility would be to leverage existing hardware transactional memories

(HTM), partitioning program execution into transactions that enforce atomicity, isolation, and durability—resembling BulkSC with durable transactions [20]. Transactions must be long enough to minimize transaction overhead (including persist barriers for each durable transaction) and improve persist concurrency (by placing many persists in the same transaction), but short enough to bound resources necessary for atomic transactions (such as a log) and to minimize forward progress lost when a transaction aborts.

7.2.2 Epoch Persistency

Motivation. Strict persistency under SC introduces many persist dependences unnecessary for correct recovery. The most common unnecessary persist dependence occurs due to the program-order constraint of SC. Programs frequently persist to large, contiguous regions of memory that logically represents a single object, but which cannot occur atomically (due to their size). Under strict persistency, the persists serialize. I remove the program-order-implied persist order constraint with *epoch persistency*, allowing consecutive persists from the same thread to reorder and persist in parallel. Doing so, however, requires annotation by the programmer in the form of persist barriers to divide execution into epochs when ordering is required by the recovery algorithm. Epoch persistency additionally allows persists to addresses protected by a lock to reorder with respect to the lock operations (e.g., avoid delaying the lock release while the persist completes); my queue implementations leverage this optimization opportunity.

Definition. Epoch persistency defines a new *persistent memory order* in addition to execution's existing memory order (referred to as the *volatile memory order*). Persistent memory order contains a subset of constraints from the volatile memory order. Any pair of persists ordered in the persistent memory order may not be observed out of that order with respect to the recovery observer.

Volatile memory order satisfies SC. Each thread's execution is additionally separated into *persist epochs* by persist barrier instructions. Epoch persistency provides several rules to inherit memory order constraints from the volatile memory order: (1) any two memory accesses on the same thread and separated by a persist barrier are ordered, (2) any two memory accesses (from the same or different threads) that conflict (they are to the same or overlapping addresses and at least one is a store/persist) assume the order observed from volatile memory order (enforced via cache coherence), and (3) eight-byte persists are atomic with respect to the recovery observer and failure. This last rule is relaxed in my evaluation, requiring only that eight-byte *aligned* persists be atomic.

Persist barriers enforce that no persist after the barrier may occur before any persist before the barrier. Persists within each epoch (not separated by a barrier) are concurrent

and may reorder or occur in parallel. Additional complexity arises in reasoning about persist ordering across threads. I define a *persist epoch race* as persist epochs from two or more threads that include memory accesses (to volatile or persistent memory) that race, including synchronization races, and at least two of the epochs include persist operations. In the presence of a persist epoch race, rules (1) and (2) order accesses prior to the epoch of the earlier access in the race before accesses following the epoch of the later access in the race. Additionally, conflicting accesses are themselves ordered according to the volatile memory order. Consequently, two persists to the same address are always ordered even if they occur in racing epochs.

Discussion. Epoch persistency provides an intuitive mechanism to guarantee proper recovery as it is impossible at recovery to observe a persist from after a barrier while failing to observe a persist from before the same barrier. However, many persists (those within the same epoch) are free to occur in parallel, improving persist concurrency.

As noted in the definition, reasoning about persist order across threads can be challenging. Synchronization operations within persist epochs impose ordering across the store and load operations (due to SC memory ordering), but do not order corresponding persist operations. Hence, persist operations correctly synchronized under SC by volatile locks may nevertheless result in astonishing persist ordering. A simple (yet conservative) way to avoid persist epoch races is to place persist barriers before and after all lock acquires and releases, and to only place locks in the volatile address space. The persist behavior of strict persistency can be achieved by preceding and following all persists with a persist barrier.

Persist epoch races may be intentionally introduced to increase persist concurrency; I discuss such an optimization below. Enforcing persist order between threads with volatile locks requires that the persists be synchronized outside of the epochs in which the persists occur. However, synchronization through persistent memory is possible. Since persists to the same address must follow the order observed by cache coherence, even if they occur in epochs that race, the outcome of persist synchronization is well defined. Hence, atomic read-modify-write operations to persistent memory addresses provide the expected behavior.

Persist ordering example. Figure 7.4 demonstrates a sample memory execution and the resulting persistent memory order. Capital letters denote memory accesses (loads, stores, and persists), accesses labeled by the same letter are to the same address. " | " implies a persist barrier. Arrows show access conflicts and the order that the conflict is observed in volatile memory order.

According to rule (1) of epoch persistency all accesses on the same thread are ordered if separated by a persist barrier (A before B and C; C' and D before E; and E' before F). By

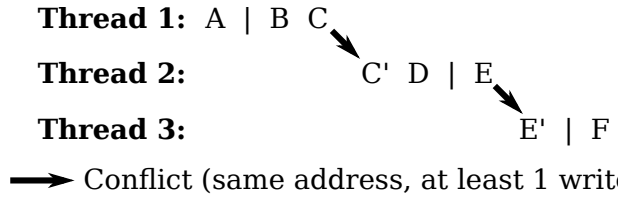


Figure 7.4: Epoch persistency persist order. Persistent memory order is enforced through persist barriers (shown as " | ") and access conflicts. Access A is ordered by transitivity before access F. If these accesses are persists they may not be reordered with respect to the recovery observer. This order is enforced even if accesses are to the volatile address space. Access B is concurrent with all shown accesses from threads 2 and 3, while access D is concurrent with accesses from thread 1.

rule (2) all conflicts are ordered according to volatile memory order (C before C'; E before E'). Transitivity of these rules additionally orders accesses (A before C', E, E', and F; C before E, E', and F; C' and D before E' and F; and E before F). Notably, B is unordered with all shown accesses by threads 2 and 3, and D is unordered with accesses by thread 1 (such accesses are concurrent). Any accesses that are persists may not be observed out of this order by the recovery observer (e.g., a persist to F may not be observed while a persist to A is not).

This order is enforced on conflicts (between C and C'; E and E') to both volatile and persistent address spaces and as long as one of the accesses is a store or persist. This includes the case where the first access is a load and the second a store/persist. Providing an implementation to enforce this constraint remains a challenge, as persists must complete before a subsequent load executes or all load-store conflicts must be detected and enforced at some later time.

The resulting persist order resembles RMO [103] with (at least) one important difference: memory order constraints introduced due to control and register data dependencies do not impose a persistent memory order constraint in epoch persistency. For example, consider that access C is a persist, C' a load, and D a persist whose value depends on C'. According to RMO such a register dependence initiated by a load introduces a (volatile) memory order constraint. Strict persistency under RMO would now allow persist D to be observed without also observing persist C. Epoch persistency, on the other hand, allows such behavior, violating RMO. In general persistent memory order need not order accesses that contain a register dependence, but this may produce unintuitive or unintended behavior.

BPFS. My definition of epoch persistency is inspired by the programming model and hardware extensions for caching persistent memory proposed for the Byte-addressable Persistent File System (BPFS) [30]. However, I introduce several subtle differences that I believe make epoch persistency a more intuitive model. My definition considers all memory

accesses when determining persist ordering among threads, whereas the BPFS memory system orders persists only when conflicts occur to the persistent address space. While the BPFS file system implementation avoids persist epoch races, it is not clear that the burden falls to the programmer to avoid such accesses or what persist behavior results when such races occur (I believe the BPFS authors' intent was to prohibit programs containing such races—the cache implementation deadlocks under persist epoch races containing circular persist dependences). Furthermore, BPFS detects conflicts to the persistent address space by recording the last thread and epoch to persist to each cache line; the next thread to access that line will detect the conflict. Such an implementation, however, cannot detect conflicts where the first access is a load and the second a store. While occasionally unintuitive, such behavior rarely results in unintended behavior and greatly simplifies the memory system. Existing memory consistency models similarly order accesses, including TSO [103]).

Persist Performance. Epoch persistency removes all unnecessary persist dependences that result from strict persistency's program order constraint. All versions of the persistent queue benefit from allowing persist entries to persist to the data segment concurrently. Additionally, *Queue Holes* allows entry length and *endBit* to persist concurrently (although both are ordered with respect to the subsequent persist of the head pointer). Finally, many persist constraints between threads are removed by intentionally allowing persist epoch races. Lock operations occur in the same epoch as the first persists of the insert operation, while unlock operations occur in the same epoch as the last persists; persists from the last epoch of an insert and the first epoch of the subsequent insert are concurrent. As a result, persists protected by a lock now occur concurrently. *Copy While Locked* and *Queue Holes* retain recovery correctness by still ordering all persists to the head marker (persists occur according to cache coherence order).

Figure 7.1 demonstrates how to use epoch persistency's barriers (shown in the code as *PersistBarrier*) within each queue design. The constraints in Figures 7.2 and 7.3 annotated with "A" are removed under epoch persistency relative to strict persistency.

Implementation. BPFS [30] outlines cache extensions that provide a persistency model similar to persist epochs, assuming the TSO consistency model. Modifications must be made to detect load-before-store conflicts (and thus enforce SC rather than TSO ordering) and track conflicts to volatile memory addresses as well as persistent memory addresses; detecting load-before-store conflicts and enforcing persist order without frequent delays remains an open problem. Instead of delaying execution to enforce persist ordering among threads, optimized implementations avoid stalling execution by buffering persists while recording and subsequently enforcing dependences among them, allowing persists to occur asynchronously despite access conflicts.

More advanced techniques might rely on hardware NVRAM logging and hardware transactional memory, partitioning program execution into transactions as discussed above for strict persistency. Under epoch persistency any transaction that contains only a single epoch need not persist atomically, and therefore logging is unnecessary (persist order must still be enforced between transactions). Persist barriers may be used as hints to establish transaction boundaries and minimize durable transaction logging.

7.2.3 Strand Persistency

Motivation. Epoch persistency relaxes persist dependences within and across threads. However, only consecutive persists within a thread may be labeled as concurrent. Likewise, persists from different threads are only concurrent if their epochs race or if they are not synchronized. Many persists within and across threads may still correctly be made concurrent even if they do not fit these patterns. I introduce *strand persistency*, a new model to minimally annotate persist dependences.

Definition. A strand is an interval of memory execution from a single thread. Strands are separated via *strand barriers*; each strand barrier begins a new strand. The strand barrier clears all previously observed persist dependences from the executing thread. Within each strand new and observed persists are ordered using persist barriers according to the epoch persistency model. Rule (1) of epoch persistency may be modified to read: any two memory accesses on the same *strand* separated by a persist barrier assume the order observed from volatile memory order (accesses from different strands never assume a program order constraint, and will only be ordered through conflicts or ordering constraints with accesses in the same strand). Rules (2) and (3) of epoch persistency apply without modification.

Discussion. There are no implicit persist ordering constraints across strands for persists to different addresses on the same thread of execution. Ordering constraints arise only for persists to the same address as implied by cache coherence. Hence, persists on a new strand may occur as early as possible and overlap with all preceding persists. Strand persistency allows programmers to indicate that logical tasks on the same thread are independent from the perspective of persistency. To enforce necessary ordering, a persist strand begins by reading any memory locations after which new persists must be ordered. These reads introduce an ordering dependency due to cache coherence, which can then be enforced with a subsequent persist barrier. This programming interface allows ordering constraints to be specified at the granularity of individual addresses; the minimal set of persist dependences is achieved by placing each persist in its own strand, loading all addresses the persist must depend on, inserting a persist barrier, and then executing the persist.

The performance of epoch persistency is achieved by using a single persist strand on

each thread (strand persistency is equivalent to epoch persistency under such conditions). While I enforce persist order within strands according to epoch persistency, other persistency models may be used (e.g., strict persistency/SC orders persists within strands but new strands clear previously observed dependences).

While intended for persistency, strand persistency additionally functions as a consistency model (*strand consistency*). Strand consistency requires full memory barriers in lieu of persist barriers; loads and stores may not reorder across memory barriers. Additionally, the execution of loads and stores may not reorder across strand barriers for the purpose of observing memory dependences between strands on the same thread, but loads and stores from different strands that do not interact may reorder with respect to other processors and threads. Dependences propagate through memory execution and cache coherence, with strands from the same thread being concurrent if they do not share conflicts through memory (i.e., the visibility of loads and stores from different strands may reorder). Recent trends suggest that such a relaxed consistency model is unnecessary for typical memory architectures as memory systems have tended towards stricter models with improved performance.

Persist Performance. The persistent queue implementations place each insert task in a separate persist strand. The result is that all unnecessary persist constraints are removed, including constraints between inserts from the same thread. Figure 7.1 includes the necessary strand persistency annotations (*NewStrand* and *PersistBarrier*). All unnecessary constraints from Figures 7.2 and 7.3 are removed; those removed in moving from epoch persistency to strand persistency are labeled "B." The required persist dependences (and only those required for correct recovery) remain, maximizing persist concurrency.

Implementation. Strand persistency builds on the hardware requirements to track persist dependencies as in epoch persistency, but further requires mechanisms to separate tracking of dependencies for different strands. In addition to tracking the last thread to access each persistent location, the strand within the thread must also be tracked. Unordered persists on different strands can traverse separate queues (e.g., on separate virtual channels) throughout the persistent memory system. Strand persistency gives enormous implementation latitude and designing efficient hardware to track and observe only the minimal ordering requirements remains an open research challenge. In this work, I focus on demonstrating the potential performance that the model allows.

7.3 Related Work

Durable storage has long been used to recover data after failures. All systems that use durable storage must specify and honor dependencies between the operations that update

that storage. For example, file systems must constrain the order of disk operations to metadata to preserve a consistent file system image [41, 25], and databases must obey the order of durable storage updates specified in write-ahead logging [71].

Specifying and honoring these dependencies becomes harder when the interface to durable storage are loads and stores to a persistent address space. Store instructions to an address space are more frequent and fine grained than update operations when using a block-based interface to durable storage (such as a file system). In addition, CPU caches interpose on store instructions, which leads to the interaction of persistency and cache consistency discussed in this dissertation.

Recent developments in nonvolatile memory technologies have spurred research on how to use these new technologies. Some research projects keep the traditional block-based interface to durable storage and devise ways to accelerate this interface [15]. Other projects provide a memory-based interface to durable storage [29]. My approach follows the path of providing a memory-based interface to durable storage, arguing that the high speed and fine-grained access of new nonvolatile memories provides a natural fit with native memory instructions.

Combining a memory interface to durable storage with multiprocessors adds concurrency control issues to those of durability. Transactions are a common and powerful paradigm for handling both concurrency control and durability, so many authors have proposed layering transactions on top of nonvolatile memory [65, 29, 116, 28]. Similarly, a recent paper proposes to couple concurrency control with recovery management by committing execution to durable storage at the granularity of the outermost critical section [21].

While transactions and critical sections are powerful mechanisms for concurrency control, many programs use other mechanisms besides these, such as conditional waits. Because of this diversity of concurrency control, I believe it is useful to treat the issues of consistency and persistency separately. Just as much work has been done to create a framework of memory consistency models [2], I seek to begin a framework on memory persistency models.

Kiln. Zhao *et al.* recently proposed Kiln, a persistent system using multi-versioning in a persistent cache to provide durable transactions [122]. A key feature of this system is that persistence control and thread synchronization are de-coupled; persistent transactions do not ensure isolated transactions (I believe it is implied that additional synchronization must already ensure that transactions are isolated). Several persistent transactions may occur within a single critical section without the thread synchronization overheads of isolated multi-thread transactions.

In my work I investigate additional interactions between thread communication and per-

sist ordering. Kiln provides no mechanism to allow concurrent persists between interacting threads as I do with epoch persistency and strand persistency. Zhao introduces ordered and unordered transactions, but it is unclear if ordered transactions are totally ordered between all threads or interact with thread synchronization to establish order, and how to enforce any order with unordered transactions (completely unordered transactions do not provide functionality for useful recovery). Finally, as with racing persist epochs, durable transactions that race break transaction atomicity; persists to cache lines that contain conflicting writes from two transactions may reach NVRAM (and be visible at recovery) before or after the transactions commit. A precise persistency model must specify how to avoid such behavior or what intended behavior results from persistent transaction races.

BPFS. The techniques most closely related to those proposed in this dissertation are the primitives for describing persist dependencies in the Byte-Addressable Persistent File System (BPFS) [30]. Condit *et al.* introduce *persist barriers*, similar to existing memory barriers, that constrain the order of writes to the persistent address space. Any two persists separated by a barrier must occur in program order, but persists within the same epoch (interval of execution separated by barriers) are concurrent. Additionally, Condit assumes that NVRAM allows eight byte atomic writes, allowing many updates to write atomically in-place without additional recovery mechanisms. Finally, the proposed cache implementation allows volatile execution of each thread to proceed ahead of the persistent state, although sharing persistent data between threads causes processors to stall. I assume similar mechanisms.

I view BPFS as a single point in the memory persistency design space. While similar to my epoch persistency design, there are subtle, yet important differences, described in Section 7.2.2. I highlight complications with the BPFS programming model, specifically persist epoch races. Epochs that race and form a cycle may cause the BPFS system to deadlock. This problem remains when the access cycle occurs at cache line granularity (false sharing), even if true races do not occur. An important aspect of memory persistency is to precisely define allowable behavior under such scenarios. I investigate the more-general design space of memory persistency and its interactions with memory consistency.

Alternatives to persistent storage. Recent work (e.g., H-Store [104]) suggests highly available systems as an outright replacement for durability. Additionally, battery-backed memory and storage obviate the need for memory persistency, allowing more expensive persist synchronization only when necessary (e.g., Whole-System Persistence [73]). I argue that computers and storage systems will always fail, and durability remains a requirement for many applications.

More importantly, highly available systems and battery-backed storage are not uni-

versally available. Mobile devices, for example, must rely on persistent storage for data recovery. Devices in the ”Internet of Things” will require high performance while recovering after failure. Any device expected to fail frequently (due to low power operation or unreliable power sources) can use memory persistency alongside NVRAM to improve performance while retaining data integrity.

7.4 Conclusion

Relaxed persistency offers new tools to enforce recovery correctness while minimizing delays due to persists. In the next chapter I use these queue designs and persistency models to quantitatively evaluate the opportunity relaxed persistency holds to improve performance with NVRAM.

CHAPTER VIII

An Evaluation of Memory Persistency

The previous two chapters described a recoverable persistent queue data structure as well as three memory persistency models. This chapter uses the queue and models to motivate the need for relaxed memory persistency models. First, I describe the methodology used for this evaluation.

8.1 Methodology

To evaluate persistent queue designs and persistency models I measure instruction execution rate on a real server and persist concurrency via memory traces. All experiments run the queue benchmarks optimized for volatile performance. Memory padding is inserted to objects and queue inserts to provide 64-byte alignment to prevent false sharing. Critical sections are implemented using MCS locks [70], a high-throughput queue based lock. Experiments insert 100-byte queue entries. Instruction execution rate is measured as inserts per second while inserting 100,000,000 entries between all threads using an Intel Xeon E5645 processor (2.4GHz). The remainder of this section describes how I measure persist concurrency for the queue benchmarks.

Persist Ordering Constraint Critical Path. Instead of proposing specific hardware designs and using architectural simulation, I instead measure, via memory traces, the persist ordering constraint critical path. The evaluation assumes a memory system with infinite bandwidth and memory banks (so bank conflicts never occur), but with finite persist latency. Thus, persist throughput is limited by the longest chain (critical path) of persist ordering constraints observed by execution under different memory persistency models. While real memory systems must necessarily delay elsewhere due to limited bandwidth, bank conflicts, and other memory-related delays incurred in the processor, measuring persist ordering constraint critical path offers an implementation-independent measure of persist concurrency.

I measure persist critical path under the following assumptions. Every persist to the persistent address space occurs in place (there is no hardware support for logging or indirection to allow concurrent persists). I track persist dependences at variable granularity (e.g., eight-byte words or 64-byte cache lines). Coarse-grained persist tracking is susceptible to false sharing, introducing unnecessary persist ordering constraints due to accesses to the same cache line but not to overlapping addresses. I similarly track persist coalescing with variable granularity. Every persist attempts to coalesce with the last persist to that address. A persist successfully coalesces if the persist fits within an atomically persistable memory block and coalescing with the previous persist to the same block does not violate any persist order constraints.

Memory Trace Generation. I use PIN to instrument the queue benchmarks and generate memory access traces [67]. Tracing multi-threaded applications requires additional work to ensure analysis-atomicity—application instructions and corresponding instrumentation instructions must occur atomically, otherwise the traced memory order will not accurately reflect execution's memory order. I provide analysis atomicity by creating a bank of locks and assigning every memory address (or block of addresses) to a lock. Each instruction holds all locks corresponding to its memory operands while being traced. In addition to tracing memory accesses, I instrument the queue benchmarks with persist barriers and persistent malloc/free to distinguish volatile and persistent address spaces. My tracing infrastructure is publicly available [81].

Tracing memory accesses in such a way ensures that the trace accurately reflects the order of memory accesses from execution. As only one instruction from any thread can access each address at once, and instructions on each thread occur in program order, the trace observes SC. Thus, all evaluated memory persistency models assume SC as the underlying consistency model.

Performance Validation. It is important that tracing not heavily influence thread interleaving, which would affect persist concurrency. I measure the distance of insert operations between successive inserts from the same thread. I observe that the distribution of insert distance is the same when running each queue natively and with instrumentation enabled, suggesting that thread interleaving is not significantly affected.

Persist Timing Simulation. Persist times are tracked per address (both persistent and volatile) as well as per thread according to the persistency model. For example, under strict persistency each persist occurs after or coalesces with the most recent persists observed through (1) each load operand, (2) the last store to the address being overwritten, and (3) any persists observed by previous instructions on the same thread. Persists' ability to coalesce is similarly propagated through memory and thread state to determine when co-

Threads	Copy While Locked			Two-Lock Concurrent			Queue Holes		
	Strict	Epoch	Strand	Strict	Epoch	Strand	Strict	Epoch	Strand
1	0.034	0.17	**12**	0.080	0.56	**29**	0.032	0.17	**13**
8	0.058	**3.2**	**21**	0.43	**3.4**	**22**	0.25	**1.9**	**20**

Table 8.1: Relaxed Persistency Performance. Persist-bound insert rate normalized to instruction execution rate assuming 500ns persist latency. System throughput is limited by the lower of persist and instruction rates—at greater than 1 (bold) instruction rate limits throughput; at lower than 1 execution is limited by the rate of persists. While strict persistency limits throughput, epoch persistency maximizes performance for many threads and strand persistency is necessary to maximize performance with one thread.

alescing will violate persist ordering constraints—a persist may coalesce if its most recent persist dependence (greatest timestamp) occurs to the same atomically persistable memory block and all other persist dependences are strictly older (timestamp strictly less). Under such situations coalescing will not violate any of the earlier persist dependences. The persistency models differ as to the events that propagate persist ordering constraints through memory and threads.

Next, I use this methodology to establish the need for relaxed persistency models, as well as measure their opportunity to accelerate recoverable systems.

8.2 Evaluation

This section uses the previously described methodology to demonstrate that persist ordering constraints present a performance bottleneck under strict persistency. However, relaxed persistency improves persist concurrency, removing stalls caused by persists. I also show that relaxed persistency models are resilient to large persist latency, allowing maximum throughput as persist latency increases. Finally, I consider the effects of larger atomic persists and coarse-grained persist dependence tracking.

8.2.1 Relaxed Persistency Performance

NVRAM persists are only a concern if they slow down execution relative to non-recoverable systems. If few enough persists occur, or those persists are sufficiently concurrent, performance remains bounded by the rate that instructions execute with few delays caused by persists. To determine system performance, I assume that only one of the instruction execution rate and persist rate is the bottleneck: either the system executes at its instruction execution rate (measured on current hardware), or throughput is limited solely by persist rate (while observing persist dependencies and retaining recovery correctness).

Table 8.1 shows the achievable throughput for the queue benchmarks and persistency models for both one and eight threads assuming 500ns persists. Rates are normalized to instruction execution—normalized rates above one (bold) admit sufficient persist concurrency to achieve the instruction execution rate while normalized rates below one are limited by persists. Instruction execution rates vary between log version and number of threads (not shown).

Strict persistency, the most conservative model, falls well below instruction execution rate, suggesting that memory systems with such restrictive models will be persist-bound. *Copy While Locked* with one thread suffers nearly a 30× slowdown; over-constraining persist order greatly limits workload throughput.

Relaxing persistency improves throughput for persist-bound configurations. Epoch persistency improves persist concurrency by allowing entire queue entries to persist concurrently and removes a number of unnecessary persist constraints via intentional persist epoch races. All queue designs see a substantial increase in throughput, with the eight thread configurations achieving instruction execution rate. At the same time, the one thread configurations remain persist-bound and their throughput suffers relative to a nonrecoverable system. Nevertheless, *Copy While Locked* with one thread is now only 5.9× slower than the instruction execution rate.

While execution for all queue designs with many threads is already compute bound and does not benefit from further relaxing persistency, the single thread configurations require additional persist concurrency to improve performance. Strand persistency allows concurrent persists from the same thread while still ensuring correct recovery. This model enables incredibly high persist concurrency such that all log versions are compute-bound even for a single thread. Sufficiently relaxed persistency models allow data structures and systems that recover from failure while retaining the throughput of existing main-memory data structures.

Persist Latency. The previous results argue for relaxed persistency models under large persist latency. However, for fast enough NVRAM technologies additional persist concurrency is unnecessary to achieve instruction execution rate. Figure 8.1 shows the achievable execution rate (limited either by persist rate or instruction execution rate) for *Copy While Locked* with one thread. The x-axis shows persist latency on a logarithmic scale, ranging from 10ns to 100µs.

At low persist latency all persistency models achieve instruction execution rate (horizontal line formed at the top). However, as persist latency increases each model eventually becomes persist-bound and throughput quickly degrades. Strict persistency becomes persist-bound at only 17ns. Epoch persistency improves persist concurrency—instruction

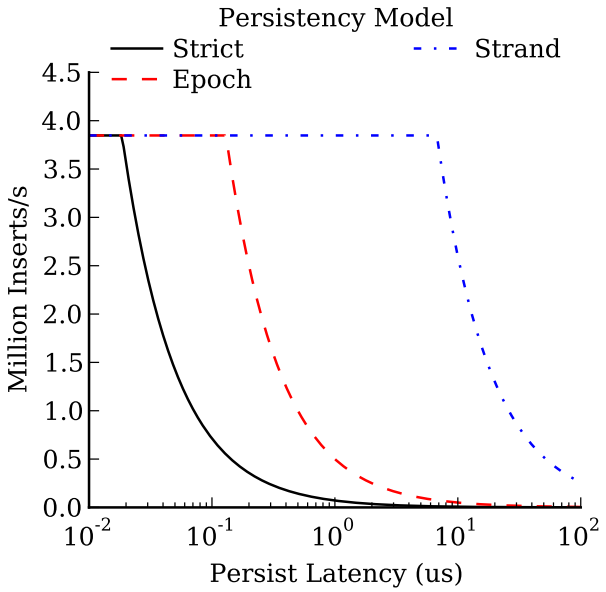


Figure 8.1: Persist Latency. *Copy While Locked*, 1 thread. All models initially compute-bound (line at top). As persist latency increases each model becomes persist-bound. Relaxed models are resilient to large persist latency.

execution rate and persist rate break even at 119ns. While this is a great improvement, I expect most NVRAM technologies to exhibit higher persist latency. Finally, strand persistency offers sufficient persist concurrency to become persist-bound only above 6us, greater than expected NVRAM persist latency.

In all cases throughput quickly decreases once execution is persist-bound as persist latency continues to increase. Persists limit the most conservative persistency models even at DRAM-like write latencies. However, relaxed persistency models are resilient to large persist latencies and achieve instruction execution rate.

8.2.2 Atomic Persist and Tracking Granularity

The previous experiments consider performance for queue designs and persistency models assuming that persist ordering constraints propagate through memory at eight-byte granularity (i.e., a race to addresses in the same eight-byte, aligned memory block introduces a persist ordering constraint according to the persistency model). Additionally, experiments assume that persists occur atomically to eight-byte, aligned memory blocks. Both of these may vary in real implementations; I measure their effect on persist ordering constraint critical path.

Atomic Persist Granularity. Atomic persist granularity is an important factor for per-

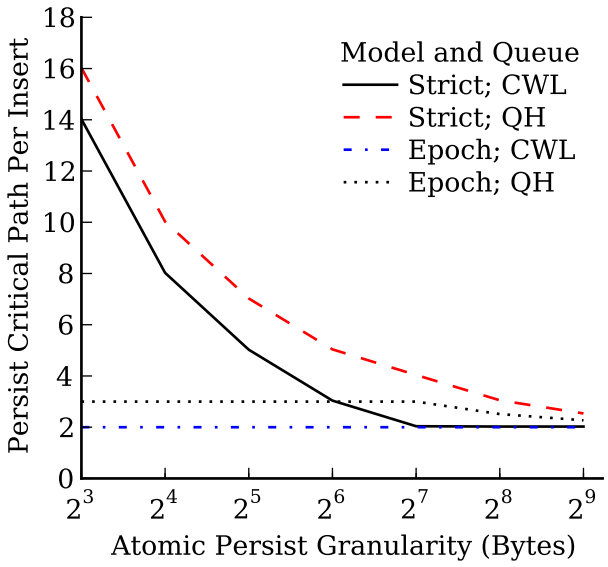


Figure 8.2: Atomic Persist Size. 1 Thread. Large atomic persists allow coalescing, increasing persist concurrency. While effective for strict persistency, large atomic persists do not improve persist concurrency for relaxed models.

sist concurrency and performance. As in [30] I assume NVRAM persists atomically to at least eight-byte (pointer-sized) blocks of memory (a persist to an eight-byte, aligned memory block will always have occurred or not after failure; there is no possibility of a partial persist). However, increasing atomic persist granularity creates opportunities for additional persist coalescing. Nearby or adjacent persists may occur atomically and coalesce so long as no persist dependences are violated. If the originally enforced ordering between two persist operations appeared on the persist dependence critical path, coalescing due to increased atomic persist granularity may decrease the critical path and reduce the likelihood of delay due to persists.

Figure 8.2 displays average persist ordering critical path per insert for *Copy While Locked* and *Queue Holes* for both strict persistency and epoch persistency as atomic persist size increases from eight to 512 bytes. At eight-byte persists there is a large separation between strict persistency and epoch persistency. As atomic persist size increases the persist critical path of strict persistency steadily decreases while the critical path of epoch persistency remains largely unchanged. At 512-byte atomic persists (right of the Figure) strict persistency matches epoch persistency. For these queue benchmarks larger atomic persists provide the same improvement to persist critical path as relaxed persistency, but offer no improvement to relaxed models. Increasing atomic persist granularity offers an alternative to relaxed persistency models.

Persist False Sharing. Just as in existing memory systems, persists suffer from false

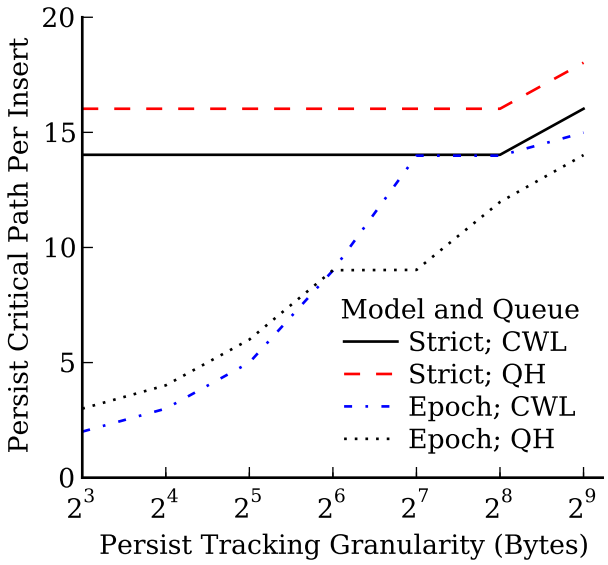


Figure 8.3: Persistent False Sharing. 1 Thread. False sharing negligably affects strict persistency (persists already serialized); relaxed models reintroduce constraints.

sharing, degrading performance. False sharing traditionally occurs under contention to the same cache line even though threads access disjoint addresses in that cache line. Similarly, *persistent false sharing* occurs when a persist ordering constraint is unnecessarily introduced due to the coarseness at which races are observed. Persistent false sharing occurs in races to both persistent and volatile memory, as races to both address spaces establish persist ordering constraints.

Figure 8.3 shows average persist critical path per insert for *Copy While Locked* and *Queue Holes* for both strict persistency and epoch persistency as the granularity at which persist ordering constraints propagate increases. For fine-grained tracking epoch persistency provides a far lower persist critical path than strict persistency. As tracking granularity increases, strict persistency performance remains the same while epoch persistency decreases (critical path increases). At 512-byte tracking granularity strict persistency and epoch persistency provide comparable persist critical paths; many of the persist constraints removed by relaxing consistency are reintroduced through false sharing.

8.3 Future Work

This dissertation lays the foundation for reasoning about NVRAM performance and programming interfaces, culminating in memory persistency. However, I leave it to future work to investigate how best to use persistency to build recoverable systems with NVRAM.

Software for NVRAM. NVRAM should eventually allow data structures with the performance of existing DRAM systems but that recover after failures. Such software will require transformations to introduce logs or indirection, and to properly order writes. These transformations slow execution on even volatile memory systems. However, an instruction-direct memory interface holds the potential to improve execution beyond block and transaction interfaces by increasing thread concurrency and removing frequent copies inherent in block storage interfaces.

New data structures must be developed to bridge the gap between existing high performance main-memory versions and those that provide recovery with disk. For example, ordered indexes are typically implemented as B+Trees with disk, but novel (and complicated) variations exist for main memory, such as highly concurrent balanced trees and lock-free skip lists. With NVRAM and memory persistency one might imagine a recoverable lock-free tree or skip list with bounded recovery time. How to design such data structures remains uncertain, as does their true performance potential.

New models and hardware implementations. Once software has been designed we will require hardware implementations of persistency models and NVRAM memory systems. Existing memory consistency encompasses a broad literature of models and implementations, detailing specific relaxations to improve performance. High performance techniques involving speculation further improve performance for strict consistency models. The same will be true for persistency models, yet I expect persistency models must be further relaxed compared to consistency models (an NVRAM persist may take an order of magnitude longer than making a store visible to other processors). Akin to speculation, logging and indirection (such as copy-on-write) may improve performance of strict persistency models.

Designing high performance and highly concurrent data structures, even without regard for persistency, remains a challenge. To simplify matters researchers have introduced "programmer-centric" memory consistency models—models that provide the appearance of SC when certain criteria are met (e.g., data-race-free-zero [2]). Memory persistency may be similarly simplified by identifying programming patterns and the minimal synchronization constructs to present the appearance of SC while still giving the compiler and processor as much freedom as possible to reorder persists.

Implementing efficient persist coalescing. Finally, the previous chapters assume that coalescing will be an important feature of NVRAM systems; indeed, recent work has already demonstrated that without hardware coalescing the software must cache and coalesce persists, introducing additional instructions and increasing the likelihood of design errors [30, 37]. Coalescing presents several problems such as detecting when coalescing is al-

lowable and determining how long persists should be delayed (delaying a persist increases the chance that it will coalesce with future persists, but may delay important events such as transaction commit). Future work must investigate language, compiler, and hardware mechanisms to provide persist coalescing. It is possible for coalescing to occur only under specific conditions (e.g., coalescing only occurs between persists on the same thread, coalescing only occurs to a limited number of addresses labeled by the programmer), reducing hardware complexity.

8.4 Conclusion

Future NVRAM technologies offer the performance of DRAM with the durability of disk. However, existing memory interfaces are incapable of guaranteeing proper data recovery by properly enforcing persist order. In previous chapters I introduced *memory persistency*, an extension to memory consistency that allows programmers to describe persist order constraints. Additionally, I outlined the design space of possible memory persistency models and detailed three persistency models and their use in implementing a persistent queue. In this chapter I use memory tracing and simulation to demonstrate that strict persistency models suffer a 30× slowdown relative to instruction execution rate, and that relaxed persistency effectively regains this performance. Memory persistency represents a framework for defining programmer interfaces for NVRAM and reasoning about program correctness for recoverable applications.

APPENDICES

APPENDIX A

Understanding and Abstracting Total Data Center Power

Steven Pelley, David Meisner, Thomas F. Wenisch, and James W. VanGilder

The alarming growth of data center power consumption has led to a surge in research activity on data center energy efficiency. Though myriad, most existing energy-efficiency efforts focus narrowly on a single data center subsystem. Sophisticated power management increases dynamic power ranges and can lead to complex interactions among IT, power, and cooling systems. However, reasoning about total data center power is difficult because of the diversity and complexity of this infrastructure.

In this paper, we develop an analytic framework for modeling total data center power. We collect power models from a variety of sources for each critical data center component. These component-wise models are suitable for integration into a detailed data center simulator that tracks subsystem utilization and interaction at fine granularity. We outline the design for such a simulator. Furthermore, to provide insight into average data center behavior and enable rapid back-of-the-envelope reasoning, we develop abstract models that replace key simulation steps with simple parametric models. To our knowledge, our effort is the first attempt at a comprehensive framework for modeling total data center power.

A.1 Introduction

Data center power consumption continues to grow at an alarming pace; it is projected to reach 100 billion kWh at an annual cost of $7.4 billion within two years [113], with a world-wide carbon-emissions impact similar to that of the entire Czech Republic [68]. In

light of this trend, computer systems researchers, application designers, power and cooling engineers, and governmental bodies have all launched research efforts to improve data center energy efficiency. These myriad efforts span numerous aspects of data center design (server architecture [61, 69], scheduling [72, 79], power delivery systems [36], cooling infrastructure [80], etc.). However, with few exceptions, existing efforts focus narrowly on energy-efficiency of single subsystems, without considering global interactions or implications across data center subsystems.

As sophisticated power management features proliferate, the dynamic range of data center power draw (as a function of utilization) is increasing, and interactions among power management strategies across subsystems grow more complex; subsystems can no longer be analyzed in isolation. Even questions that appear simple on their face can become quite complicated.

Reasoning about total data center power is difficult because of the diversity and complexity of data center infrastructure. Five distinct sub-systems (designed and marketed by different industry segments) account for most of a data center's power draw: (1) servers and storage systems, (2) power conditioning equipment, (3) cooling and humidification systems, (4) networking equipment, and (5) lighting/physical security. Numerous sources have reported power breakdowns [113, 69]; Table A.1 illustrates a typical breakdown today. The first three subsystems dominate and their power draw can vary drastically with data center utilization. Cooling power further depends on ambient weather conditions around the data center facility. Even the distribution of load in each subsystem can affect power draws, as the interactions among sub-systems are non-linear

In this paper, our objective is to provide tools to the computer systems community to assess and reason about total data center power. Our approach is two-fold, targeting both *data center simulation* and *abstract analytic modeling*. First, we have collected a set of detailed power models (from academic sources, industrial white papers, and product data sheets) for each critical component of data center infrastructure, which describe power draw as a function of environmental and load parameters. Each model describes the power characteristics of a single device (i.e., one server or computer room air handler (CRAH)) and, as such, is suitable for integration into a detailed data center simulator. We describe how these models interact (i.e., how utilization, power, and heat flow among components) and outline the design of such a simulator. To our knowledge, we are the first to describe an integrated data center simulation infrastructure; its implementation is underway.

Although these detailed models enable data center simulation, they do not allow direct analytic reasoning about total data center power. Individual components' power draw vary non-linearly with localized conditions (i.e., temperature at a CRAH inlet, utilization of an

Table A.1: Typical Data Center Power Breakdown.

Servers	Cooling	Power Cond.	Network	Lighting
56%	30%	8%	5%	1%

individual server), that require detailed simulation to assess precisely. Hence, to enable back-of-the-envelope reasoning, we develop an *abstract model* that replaces key steps of the data center simulation process with simple parametric models that enable analysis of average behavior. In particular, we abstract away time-varying scheduling/load distribution across servers and detailed tracking of the thermodynamics of data center airflow. Our abstract model provides insight into how data center sub-systems interact and allows quick comparison of energy-efficiency optimizations.

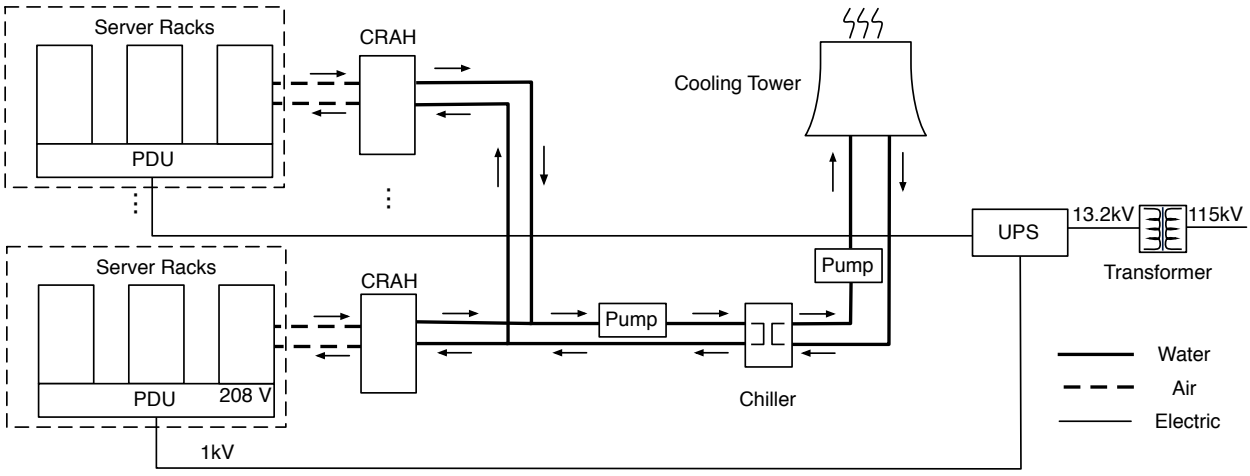


Figure A.1: Power and Cooling Flow.

A.2 Related Work

The literature on computer system power management is vast; a recent survey appears in [56]. A tutorial on data center infrastructure and related research issues is available in [66].

Numerous academic studies and industrial whitepapers describe or apply models of power draw for individual data center subsystems. Prior modeling efforts indicate that server power varies roughly linearly in CPU utilization [36, 95]. Losses in power conditioning systems and the impact of power overprovisioning are described in [36, 93]. Our prior work proposes mechanisms to eliminate idle power waste in servers [69]. Other studies have focused on understanding the thermal implications of data center design [52, 80].

Power- and cooling-aware scheduling and load balancing can mitigate data center cooling costs [72, 79]. Though each of these studies examines particular aspects of data center design, none of these present a comprehensive model for total data center power draw.

A.3 Data Center Power Flow

We begin by briefly surveying major data center subsystems and their interactions. Figure A.1 illustrates the primary power-consuming subsystems and how power and heat flow among them. Servers (arranged in racks) consume the dominant fraction of data center power, and their power draw varies with utilization.

The data center's power conditioning infrastructure typically accepts high-voltage AC power from the utility provider, transforms its voltage, and supplies power to uninterruptible power supplies (UPSs). The UPSs typically charge continuously and supply power in the brief gap until generators can start during a utility failure. From the UPS, electricity is distributed at high voltage (480V-1kV) to power distribution units (PDUs), which regulate voltage to match IT equipment requirements. Both PDUs and UPSs impose substantial power overheads, and their overheads grow with load.

Nearly all power dissipated in a data center is converted to heat, which must be evacuated from the facility. Removing this heat while maintaining humidity and air quality requires an extensive cooling infrastructure. Cooling begins with the computer room air handler (CRAH), which transfer heat from servers' hot exhaust to a chilled water/glycol cooling loop while supplying cold air throughout the data center. CRAHs appear in various forms, from room air conditioners that pump air through underfloor plenums to in-row units located in containment systems. The water/glycol heated by the CRAHs is then pumped to a chiller plant where heat is exchanged between the inner water loop and a second loop connected to a cooling tower, where heat is released to the outside atmosphere. Extracting heat in this manner requires substantial energy; chiller power dominates overall cooling system power, and its requirements grow with both the data center thermal load and outside air temperature.

A.4 Modeling Data Center Power

Our objective is to model total data center power at two granularities: for detailed simulation and for abstract analysis. Two external factors primarily affect data center power usage: the aggregate load presented to the compute infrastructure and the ambient outside air temperature (outside humidity secondarily affects cooling power as well, but, to limit

Table A.2: Variable Definitions.

U	Data Center Utilization	T	Temperature
u	Component Utilization	$\dot{m}$	Flow Rate
κ	Containment Index	C_{p}	Heat Capacity
ℓ	Load Balancing Coefficient	N_{Srv}	# of Servers
E	Heat Transfer Effectiveness	P	Power
π	Loss Coefficient	q	Heat
f	Fractional Flow Rate		

model complexity, we do not consider it here). We construct our modeling framework by composing models for the individual data center components. In our abstract models, we replace key steps that determine the detailed distribution of compute load and heat with simple parametric models that enable reasoning about average behavior.

Framework. We formulate total data center power draw, P_{Total}, as a function of total utilization, U, and ambient outside temperature, T_{Outside}. Figure A.2 illustrates the various intermediary data flows and models necessary to map from U and T_{Outside} to P_{Total}. Each box is labeled with a sub-section reference that describes that step of the modeling framework. In the left column, the power and thermal burden generated by IT equipment is determined by the aggregate compute infrastructure utilization, U. On the right, the cooling system consumes power in relation to both the heat generated by IT equipment and the outside air temperature.

Simulation & Modeling. In simulation, we relate each component's utilization (processing demand for servers, current draw for electrical conditioning systems, thermal load for cooling systems, etc.) to its power draw on a component-by-component basis. Hence, simulation requires precise accounting of per-component utilization and the connections between each component. For servers, this means knowledge of how tasks are assigned to machines; for electrical systems, which servers are connected to each circuit; for cooling systems, how heat and air flow through the physical environment. We suggest several approaches for deriving these utilizations and coupling the various sub-systems in the context of a data center simulator; we are currently developing such a simulator.

However, for high-level reasoning, it is also useful to model aggregate behavior under typical-case or parametric assumptions on load distribution. Hence, from our detailed models, we construct abstract models that hide details of scheduling and topology.

4.1. Abstracting Server Scheduling.

The first key challenge in assessing total data center power involves characterizing server utilization and the effects of task scheduling on the power and cooling systems.

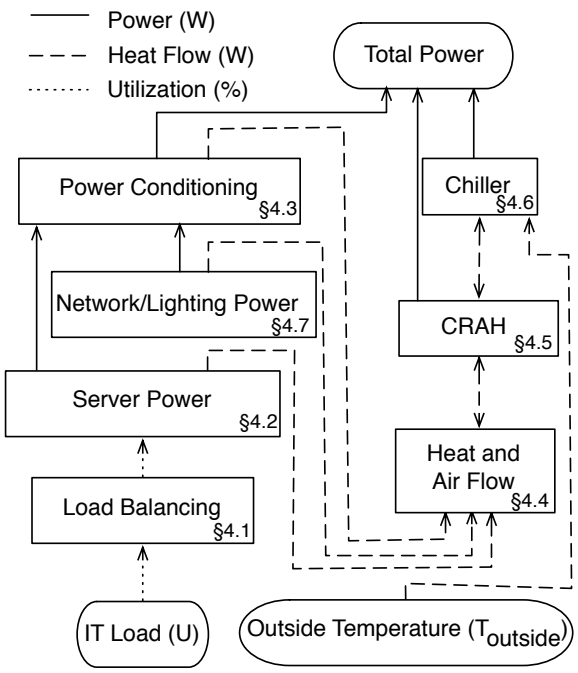


Figure A.2: Data Flow.

Whereas a simulator can incorporate detailed scheduling, task migration, or load balancing mechanisms to derive precise accounting of per-server utilization, we must abstract these mechanisms for analytic modeling.

As a whole, our model relates total data center power draw to aggregate data center utilization, denoted by U. Utilization varies from 0 to 1, with 1 representing the peak compute capacity of the data center. We construe U as unitless, but it might be expressed in terms of number of servers, peak power (Watts), or compute capability (MIPS). For a given U, individual server utilizations may vary (e.g., as a function of workload characteristics, consolidation mechanisms, etc.). We represent the per-server utilizations with u_{Srv}, where the subscript indicates the server in question. For a homogeneous set of servers:

$$U = \frac{1}{N_{\text{Srv}}} \sum_{\text{Servers}} u_{\text{Srv}[i]} \tag{A.1}$$

For simplicity, we have restricted our analytic model to a homogeneous cluster. In real data centers, disparities in performance and power characteristics across servers require detailed simulation to model accurately, and their effect on cooling systems can be difficult to predict [74].

To abstract the effect of consolidation, we characterize the individual server utilization, u_{Srv}, as a function of U and a measure of task consolidation, ℓ. We use ℓ to capture the degree to which load is concentrated or spread over the data center's servers. For a given

U and ℓ we define individual server utilization as:

$$u_{\text{Srv}} = \frac{U}{U + (1 - U)\ell} \tag{A.2}$$

u_{Srv} only holds meaning for the $N_{\text{Srv}}(U + (1-U)\ell)$ servers that are non-idle; the remaining servers have zero utilization, and are assumed to be powered off. Figure A.3 depicts the relationship among u_{Srv}, U, and ℓ. $\ell = 0$ corresponds to perfect consolidation—the data center's workload is packed onto the minimum number of servers, and the utilization of any active server is 1. $\ell = 1$ represents the opposite extreme of perfect load balancing—all servers are active with $u_{Srv} = U$. Varying ℓ allows us to represent consolidation between these extremes.

4.2. Server Power.

Several studies have characterized server power consumption [66, 36, 69, 95]. Whereas the precise shape of the utilization-power relationship varies, servers generally consume roughly half of their peak load power when idle, and power grows with utilization. In line with prior work [95], we approximate a server's power consumption as linear in utilization between a fixed idle power and peak load power.

$$P_{\text{Srv}} = P_{\text{Srv}_{\text{Idle}}} + (P_{\text{Srv}_{\text{Peak}}} - P_{\text{Srv}_{\text{Idle}}})u_{\text{Srv}} \tag{A.3}$$

Power draw is not precisely linear in utilization; server sub-components exhibit a variety of relationships (near constant to quadratic). It is possible to use any nonlinear analytical power function, or even a suitable piecewise function. Additionally, we have chosen to use CPU utilization as an approximation for overal system utilization; a more accurate representation of server utilization might instead be employed. In simulation, more precise system/workload-specific power modeling is possible by interpolating in a table of measured power values at various utilizations (e.g., as measured with the SpecPower benchmark). Figure A.4 compares the linear approximation against published SpecPower results for two recent Dell systems.

4.3. Power Conditioning Systems.

Data centers require considerable infrastructure simply to distribute uninterrupted, stable electrical power. PDUs transform the high voltage power distributed throughout the data center to voltage levels appropriate for servers. They incur a constant power loss as well as a power loss proportional to the square of the load [93]:

$$P_{\text{PDU}_{\text{Loss}}} = P_{\text{PDU}_{\text{Idle}}} + \pi_{\text{PDU}}(\sum_{Servers} P_{\text{Srv}})^2 \tag{A.4}$$

where $P_{\text{PDU}_{\text{Loss}}}$ represents power consumed by the PDU, π_{PDU} represents the PDU power loss coefficient, and $P_{\text{PDU}_{\text{Idle}}}$ the PDU's idle power draw. PDUs typically waste 3% of their input power. As current practice requires all PDUs to be active even when idle, the total dynamic power range of PDUs for a given data center utilization is small relative to total data center power; we chose not to introduce a separate PDU load balancing coefficient and instead assume perfect load balancing across all PDUs.

UPSs provide temporary power during utility failures. UPS systems are typically placed in series between the utility supply and PDUs and impose some power overheads even when operating on utility power. UPS power overheads follow the relation [93]:

$$P_{\text{UPS}_{\text{Loss}}} = P_{\text{UPS}_{\text{Idle}}} + \pi_{\text{UPS}} \sum_{PDUs} P_{\text{PDU}} \tag{A.5}$$

where π_{UPS} denotes the UPS loss coefficient. UPS losses typically amount to 9% of their input power at full load.

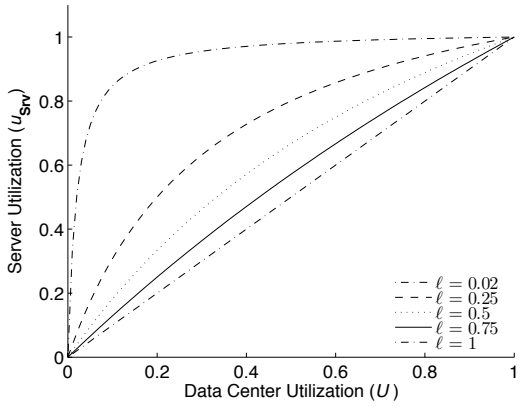


Figure A.3: Individual Server Utilization.

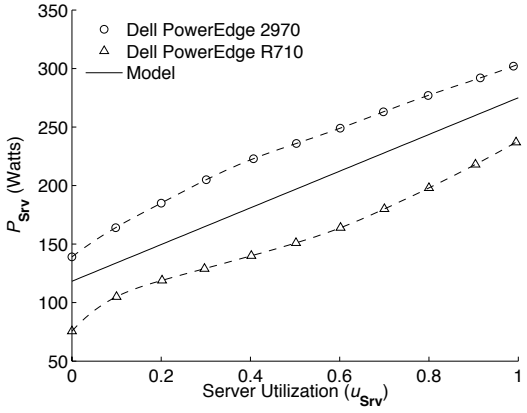


Figure A.4: Individual Server Power.

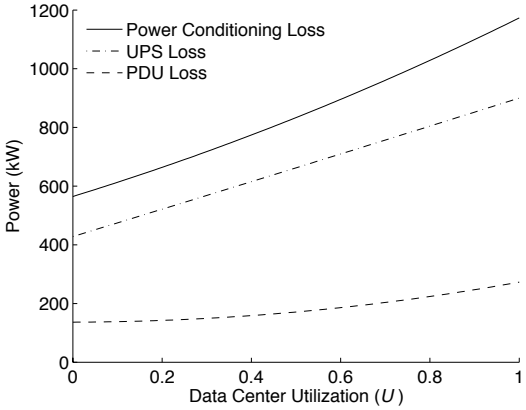


Figure A.5: Power Conditioning Losses.

Figure A.5 shows the power losses for a 10MW (peak server power) data center. At peak load, power conditioning loss is 12% of total server power. Furthermore, these losses result in additional heat that must be evacuated by the cooling system.

4.4. Heat and Air Flow.

Efficient heat transfer in a data center relies heavily on the CRAH's ability to provide servers with cold air. CRAHs serve two basic functions: (1) to provide sufficient airflow to prevent recirculation of hot server exhaust to server inlets, and (2) to act as a heat exchanger between server exhaust and some heat sink, typically chilled water.

In general, heat is transferred between two bodies according to the following thermodynamic principle:

$$q = \dot{m} C_{\mathrm{p}} (T_{\mathrm{h}} - T_{\mathrm{c}}) \tag{A.6}$$

Here q is the power transferred between a device and fluid, $\dot{m}$ represents the fluid mass flow, and C_{p} is the specific heat capacity of the fluid. T_{h} and T_{c} represent the hot and cold temperatures, respectively. The values of $\dot{m}$, T_{h}, and T_{c} depend on the physical air flow throughout the data center and air recirculation.

Air recirculation arises when hot and cold air mix before being ingested by servers, requiring a lower cold air temperature and greater mass flow rate to maintain server inlet temperature within a safe operating range. Data center designers frequently use computational fluid dynamics (CFD) to model the complex flows that arise in a facility and lay out racks and CRAHs to minimize recirculation. The degree of recirculation depends greatly on data center physical topology, use of containment systems, and the interactions among high-velocity equipment fans. In simulation, it is possible to perform CFD analysis to obtain accurate flows.

For abstract modeling, we replace CFD with a simple parametric model of recirculation that can capture its effect on data center power. We introduce a *containment index* (κ), based on previous metrics for recirculation [106, 114]. Containment index is defined as the fraction of air ingested by a server that is supplied by a CRAH (or, at the CRAH, the fraction that comes from server exhaust). The remaining ingested air is assumed to be recirculated from the device itself. Thus, a κ of 1 implies no recirculation (a.k.a. perfect containment). Though containment index varies across servers and CRAHs (and may vary with changing airflows), our abstract model uses a single, global containment index to represent average behavior, resulting in the following heat transfer equation:

$$q = \kappa \dot{m}_{\mathrm{air}} C_{\mathrm{p_{air}}} (T_{\mathrm{a_h}} - T_{\mathrm{a_c}}) \tag{A.7}$$

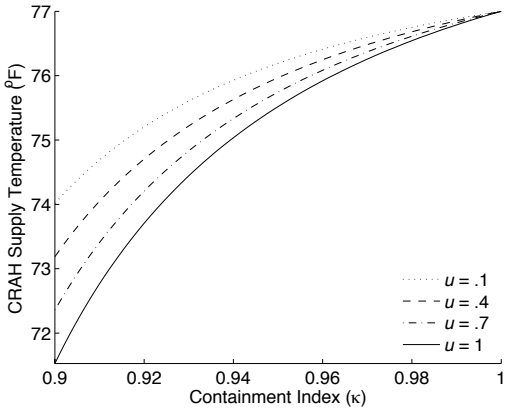


Figure A.6: CRAH Supply Temperature.

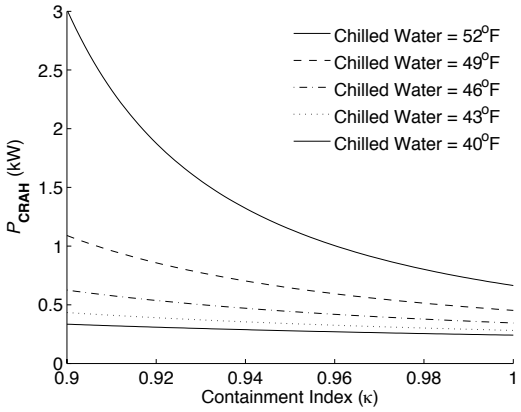


Figure A.7: CRAH Power.

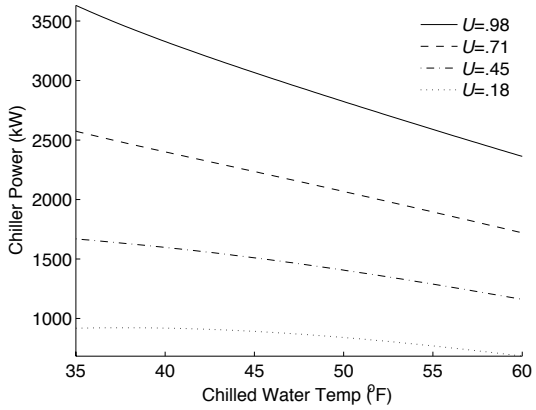


Figure A.8: Chilled Water Temperature.

For this case, q is the heat transferred by the device (either server or CRAH), $\dot{m}_{\text{air}}$ represents the total air flowing through the device, T_{a_h} the temperature of the air exhausted by the server, and T_{a_c} is the temperature of the cold air supplied by the CRAH. These temperatures represent the hottest and coldest air temperatures in the system, respectively, and are not necessarily the inlet temperatures observed at servers and CRAHs (except for the case when $\kappa = 1$).

Using only a single value for κ simplifies the model by allowing us to enforce the conservation of air flow between the CRAH and servers. Figure A.6 demonstrates the burden air recirculation places on the cooling system. We assume flow through a server increases linearly with server utilization (representing variable-speed fans) and a peak server power of 200 watts. The left figure shows the CRAH supply temperature necessary to maintain the typical maximum safe server inlet temperature of 77°F. As κ decreases, the required CRAH supply temperature quickly drops. Moreover, required supply temperature drops faster as utilization approaches peak. As we show next, lowering supply temperature re-

sults in super-linear increases in CRAH and chiller plant power. Preventing air recirculation (e.g., with containment systems) can drastically improve cooling efficiency.

4.5. Computer Room Air Handler.

CRAHs transfer heat out of the server room to the chilled water loop. We model this exchange of heat using a modified effectiveness-NTU method [17]:

$$q_{\text{crah}} = E\,\kappa\,\dot{m}_{\text{air}} C_{\text{p}_{\text{air}}} f^{0.7}(\kappa\, T_{\text{a}_{\text{h}}} + (1-\kappa) T_{\text{a}_{\text{c}}} - T_{\text{w}_{\text{c}}}) \tag{A.8}$$

q_{crah} is the heat removed by the CRAH, E is the transfer efficiency at the maximum mass flow rate (0 to 1), f represents the volume flow rate as a fraction of the maximum volume flow rate, and $T_{\text{w}_{\text{c}}}$ the chilled water temperature.

CRAH power is dominated by fan power, which grows with the cube of mass flow rate to some maximum, here denoted as $P_{\text{CRAH}_{\text{Dyn}}}$. Additionally, there is some fixed power cost for sensors and control systems, $P_{\text{CRAH}_{\text{Idle}}}$. We model CRAH units with variable speed drives (VSD) that allow volume flow rate to vary from zero (at idle) to the CRAH's peak volume flow. Some CRAH units are cooled by air rather than chilled water or contain other features such as humidification systems which we do not consider here.

$$P_{\text{CRAH}} = P_{\text{CRAH}_{\text{Idle}}} + P_{\text{CRAH}_{\text{Dyn}}} f^3 \tag{A.9}$$

As the volume flow rate through the CRAH increases, both the mass available to transport heat and the efficiency of heat exchange increase. This increased heat transfer efficiency somewhat offsets the cubic growth of fan power as a function of air flow. The true relationship between CRAH power and heat removed falls between a quadratic and cubic curve. Additionally, the CRAH's ability to remove heat depends on the temperature difference between the CRAH air inlet and water inlet. Reducing recirculation and lowering the chilled water supply temperature reduce the power required by the CRAH unit.

The effects of containment index and chilled water supply temperature on CRAH power are shown in Figure A.7. Here the CRAH model has a peak heat transfer efficiency of 0.5, a maximum airflow of 6900 CFM, peak fan power of 3kW, and an idle power cost of 0.1 kW. When the chilled water supply temperature is low, CRAH units are relatively insensitive to changes in containment index. For this reason data center operators often choose low chilled water supply temperature, leading to overprovisioned cooling in the common case.

4.6. Chiller Plant and Cooling Tower.

The chiller plant provides a supply of chilled water or glycol, removing heat from the warm water return. Using a compressor, this heat is transferred to a second water loop, where it is pumped to an external cooling tower. The chiller plant's compressor accounts

for the majority of the overall cooling cost in most data centers. Its power draw depends on the amount of heat extracted from the chilled water return, the selected temperature of the chilled water supply, the water flow rate, the outside temperature, and the outside humidity. For simplicity, we neglect the effects of humidity. In current-generation data centers, the water flow rate and chilled water supply temperature are held constant during operation (though there is substantial opportunity to improve cooling efficiency with variable-temperature chilled water supply).

Numerous chiller types exist, typically classified by their compressor type. The HVAC community has developed several modeling approaches to assess chiller performance. Although physics-based models do exist, we chose the Department of Energy's DOE2 chiller model [35], an empirical model comprising a series of regression curves. Fitting the DOE2 model to a particular chiller requires numerous physical measurements. Instead, we use a benchmark set of regression curves provided by the California Energy Commission [18]. We do not detail the DOE2 model here, as it is well documented and offers little insight into chiller operation. We neglect detailed modeling of the cooling tower, using outside air temperature as a proxy for cooling tower water return temperature.

As an example, a chiller at a constant outside temperature and chilled water supply temperature will require power that grows quadratically with the quantity of heat removed (and thus with utilization). A chiller intended to remove 8MW of heat at peak load using 3,200 kW at a steady outside air temperature of 85°F, a steady chilled water supply temperature of 45°F, and a data center load balancing coefficient of 0 (complete consolidation) will consume the following power as a function of total data center utilization (kW):

$$P_{\text{Chiller}} = 742.8U^2 + 1{,}844.6U + 538.7$$

Figure A.8 demonstrates the power required to supply successively lower chilled water temperatures at various U for an 8MW peak thermal load. When thermal load is light, chiller power is relatively insensitive to chilled water temperature, which suggests using a conservative (low) set point to improve CRAH efficiency. However, as thermal load increases, the power required to lower the chilled water temperature becomes substantial. The difference in chiller power for a 45°F and 55°F chilled water supply at peak load is nearly 500kW. Figure A.9 displays the rapidly growing power requirement as the cooling load increases for a 45°F chilled water supply. The graph also shows the strong sensitivity of chiller power to outside air temperature.

4.7. Miscellaneous Power Draws.

Networking equipment, pumps, and lighting all contribute to total data center power. However, the contribution of each is quite small (a few percent). None of these systems

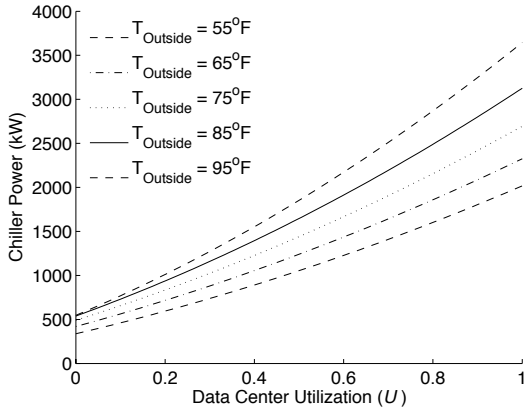


Figure A.9: Effects of U and T_{Outside} on P_{Chiller}

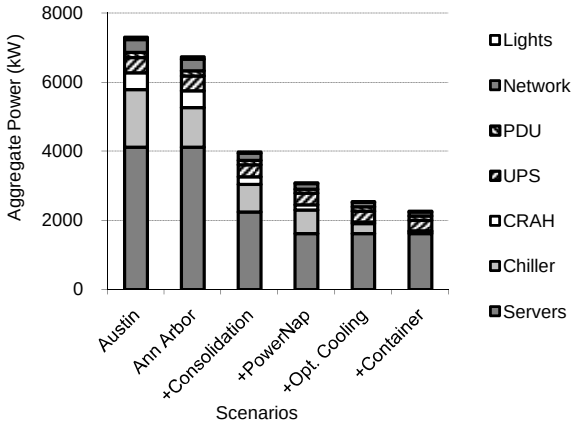


Figure A.10: A Case Study of Power-Saving Features.

have power draws that vary significantly with data center load. We account for these subsystems by including a term for fixed power overheads (around 6% of peak) in the overall power model. We do not currently consider the impact of humidification within our models.

4.8. Applying the Model: A Case Study.

To demonstrate the utility of our abstract models, we contrast the power requirements of several hypothetical data centers. Each scenario builds on the previous, introducing a new power-saving feature. We analyze each data center at 25% utilization. Table A.3 lists our scenarios and Figure A.10 displays their respective power draws. *Austin* and *Ann Arbor* represent conventional data centers with legacy physical infrastructure typical of facilities commissioned in the last three to five years. We use yearly averages for outside air temperature (reported in °F). We assume limited server consolidation and a relatively poor (though not atypical) containment index of 0.9. Furthermore, we assume typical (inefficient) servers with idle power at 60% of peak power, and static chilled water and CRAH air supply temperatures set to 45°F and 65°F, respectively. We scale the data centers such

Table A.3: Hypothetical Data Centers.

Data Center	ℓ	κ	$\frac{P_{\text{Idle}}}{P_{\text{Peak}}}$	$T(F)$	Opt. Cooling
Austin	.95	.9	.6	70	no
Ann Arbor	.95	.9	.6	50	no
+Consolidation	.25	.9	.6	50	no
+PowerNap	.25	.9	.05	50	no
+Opt. Cooling	.25	.9	.05	50	yes
+Containers	.25	.99	.05	50	yes

that the Austin facility consumes precisely 10MW at peak utilization. With the exception of *Austin*, all data centers are located in Ann Arbor.

The outside air temperature difference between *Austin* and *Ann Arbor* yields substantial savings in chiller power. Note, however, that aggregate data center power is dominated by server power, nearly all of which is wasted by idle systems. *Consolidation* represents a data center where virtual machine consolidation or other mechanisms reduce ℓ from 0.95 to 0.25, increasing individual server utilization from 0.26 to 0.57 and reducing the number of active servers from 81% to 44%. Improved consolidation drastically decreases aggregate power draw, but, paradoxically, it increases power usage effectiveness (PUE; total data center power divided by IT equipment power). These results illustrate the shortcoming of PUE as a metric for energy efficiency—it fails to account for the (in)efficiency of IT equipment. *PowerNap* [69] allows servers to idle at 5% of peak power by transitioning rapidly to a low power sleep state, reducing overall data center power another 22%. PowerNap and virtual machine consolidation take alternative approaches to target the same source of energy inefficiency: server idle power waste.

The *Optimal Cooling* scenario posits integrated, dynamic control of the cooling infrastructure. We assume an optimizer with global knowledge of data center load/environmental conditions that seeks to minimize chiller power. The optimizer chooses the highest T_{w_c} that still allows CRAHs to meet the maximum allowable server inlet temperature. This scenario demonstrates the potential for intelligent cooling management. Finally, *Container* represents a data center with a containment system (e.g., servers enclosed in shipping containers), where containment index is increased to 0.99. Under this scenario, the cooling system power draw is drastically reduced and power conditioning infrastructure becomes the limiting factor on power efficiency.

A.5 Conclusion

To enable holistic research on data center energy efficiency, computer system designers need models to enable reasoning about total data center power draw. We have presented parametric power models of data center components suitable for use in a detailed data center simulation infrastructure and for abstract back-of-the-envelope estimation. We demonstrate the utility of these models through case studies of several hypothetical data center designs.

APPENDIX B

Power Routing: Dynamic Power Provisioning in the Data Center

Steven Pelley, David Meisner, Pooya Zandevakili,
Thomas F. Wenisch, and Jack Underwood

Data center power infrastructure incurs massive capital costs, which typically exceed energy costs over the life of the facility. To squeeze maximum value from the infrastructure, researchers have proposed over-subscribing power circuits, relying on the observation that peak loads are rare. To ensure availability, these proposals employ power capping, *which throttles server performance during utilization spikes to enforce safe power budgets. However, because budgets must be enforced locally—at each power distribution unit (PDU)—local utilization spikes may force throttling even when power delivery capacity is available elsewhere. Moreover, the need to maintain reserve capacity for fault tolerance on power delivery paths magnifies the impact of utilization spikes.*

In this paper, we develop mechanisms to better utilize installed power infrastructure, reducing reserve capacity margins and avoiding performance throttling. Unlike conventional high-availability data centers, where collocated servers share identical primary and secondary power feeds, we reorganize power feeds to create shuffled *power distribution topologies. Shuffled topologies spread secondary power feeds over numerous PDUs, reducing reserve capacity requirements to tolerate a single PDU failure. Second, we propose* Power Routing, *which schedules IT load dynamically across redundant power feeds to: (1) shift slack to servers with growing power demands, and (2) balance power draw across AC phases to reduce heating and improve electrical stability. We describe efficient*

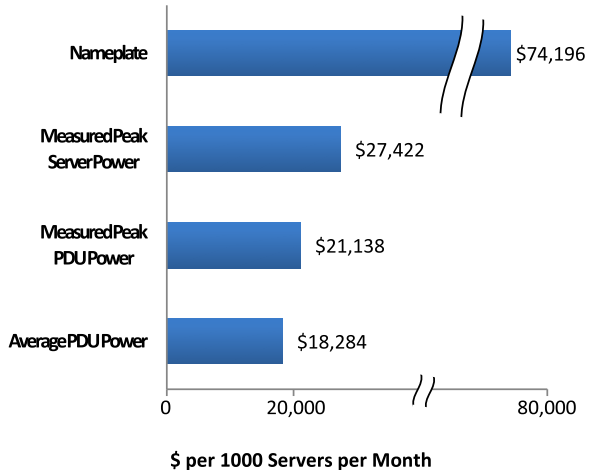


Figure B.1: The cost of over-provisioning. Amortized monthly cost of power infrastructure for 1000 servers under varying provisioning schemes.

heuristics for scheduling servers to PDUs (an NP-complete problem). Using data collected from nearly 1000 servers in three production facilities, we demonstrate that these mechanisms can reduce the required power infrastructure capacity relative to conventional high-availability data centers by 32% without performance degradation.

B.1 Introduction

Data center power provisioning infrastructure incurs massive capital costs—on the order of $10-$25 per Watt of supported IT equipment [66, 111]. Power infrastructure costs can run into the $10's to $100's of millions, and frequently exceed energy costs over the life of the data center [51]. Despite the enormous price tag, over-provisioning remains common at every layer of the power delivery system [51, 66, 36, 60, 91, 47]. Some of this spare capacity arises due to deliberate design. For example, many data centers include redundant power distribution paths for fault tolerance. However, the vast majority arises from the significant challenges of sizing power systems to match unpredictable, time-varying server power demands. Extreme conservatism in nameplate power ratings (to the point where they are typically ignored), variations in system utilization, heterogeneous configurations, and design margins for upgrades all confound data center designers' attempts to squeeze more power from their infrastructure. Furthermore, as the sophistication of power management improves, servers' power demands will become even more variable [69], increasing the data center designers' challenge.

Although the power demands of individual servers can vary greatly, statistical effects make it unlikely for all servers' demands to peak at the same time [47, 91]. Even in highly-tuned clusters running a single workload, peak utilization is rare, and still falls short of

provisioned power capacity [36]. This observation has lead researchers and operators to propose *over-subscribing* power circuits. To avoid overloads that might impact availability, such schemes rely on *power capping* mechanisms that enforce power budgets at individual servers [60, 117] or over ensembles [91, 118]. The most common power-capping approaches rely on throttling server performance to reduce power draw when budgets would otherwise be exceeded [60, 91, 117, 118].

Figure B.1 illustrates the cost of conservative provisioning and the potential savings that can be gained by over-subscribing the power infrastructure. The graph shows the amortized monthly capital cost for power infrastructure under varying provisioning schemes. We calculate costs following the methodology of Hamilton [51] assuming high-availability power infrastructure costs $15 per critical-load Watt [111], the power infrastructure has a 15-year lifetime, and the cost of financing is 5% per annum. We derive the distribution of actual server power draws from 24 hours of data collected from 1000 servers in three production facilities (details in Section B.5.1). Provisioning power infrastructure based on nameplate ratings results in infrastructure costs over triple the facility's actual need. Hence, operators typically ignore nameplate ratings, instead provisioning infrastructure based on a measured peak power for each class of server hardware. However, even this provisioning method overestimates actual needs—provisioning based on the observed aggregate peak at any power distribution unit (PDU) reduces costs 23%. Provisioning for less-than-peak loads can yield further savings at the cost of some performance degradation (e.g., average power demands are only 87% of peak).

Power capping makes over-subscribing safe. However, power budgets must enforce local (PDU) as well as global (uninterruptible power supply, generator and utility feed) power constraints. Hence, local spikes can lead to sustained performance throttling, even if the data center is lightly utilized and ample power delivery capacity is available elsewhere. Moreover, in high-availability deployments, the need to maintain reserve capacity on redundant power delivery paths to ensure uninterrupted operation in the event of PDU failure magnifies the impact of utilization spikes—not only does the data center's direct demand rise, but also the potential load from failover.

An ideal power delivery system would balance loads across PDUs to ensure asymmetric demand does not arise. Unfortunately, since server power demands vary, it is difficult or impossible to balance PDU loads statically, through clever assignment of servers to PDUs. Such balancing may be achievable dynamically through admission control [22] or virtual machine migration [27], but implies significant complexity, may hurt performance, and may not be applicable to non-virtualized systems. Instead, in this paper, we explore mechanisms to balance load through the *power delivery infrastructure*, by dynamically

connecting servers to PDUs.

Our approach, *Power Routing*, builds on widely-used techniques for fault-tolerant power delivery, whereby each server can draw power from either of two redundant feeds. Rather than designating primary and secondary feeds and switching only on failure (or splitting loads evenly across both paths), we instead centrally control the switching of servers to feeds. The soft-switching capability (already present for ease of maintenance in many dual-corded power supplies and rack-level transfer switches) acts as the foundation of a power switching network.

In existing facilities, it is common practice for all servers in a rack or row to share the same pair of redundant power feeds, which makes it impossible to use soft-switching to influence local loading. Our key insight, inspired by the notion of skewed-associative caches [100] and declustering in disk arrays [4]), is to create *shuffled distribution topologies*, where power feed connections are permuted among servers within and across racks. In particular, we seek topologies where servers running the same workload (which are most likely to spike together) connect to distinct pairs of feeds. Such topologies have two implications. First, they spread the responsibility to bear a failing PDU's load over a large number of neighbors, reducing the required reserve capacity at each PDU relative to conventional designs. Second, they create the possibility, through a series of switching actions, to route slack in the power delivery system to a particular server.

Designing such topologies is challenging because similar servers tend to be collocated (e.g., because an organization manages ownership of data center space at the granularity of racks). Shuffled topologies that route power from particular PDUs over myriad paths require wiring that differs markedly from current practice. Moreover, assignments of servers to power feeds must not only meet PDU capacity constraints, they must also: (1) ensure that no overloads occur if any PDU fails (such a failure instantly causes all servers to switch to their alternate power feed); and (2) balance power draws across the three phases of each alternating current (AC) power source to avoid voltage and current fluctuations that increase heating, reduce equipment lifetime, and can precipitate failures [49]. Even given a shuffled topology, power routing remains challenging: we will show that solving the dynamic assignment of servers to PDUs reduces to the partitioning problem [42], and, hence, is NP-complete and infeasible to solve optimally. In this paper, we address each of these challenges, to contribute:

- **Lower reserve capacity margins.** Because more PDUs cooperate to tolerate failures, shuffled topologies reduce per-PDU capacity reserves from 50% of instantaneous load to a $1/N$ fraction, where N is the number of cooperating PDUs.

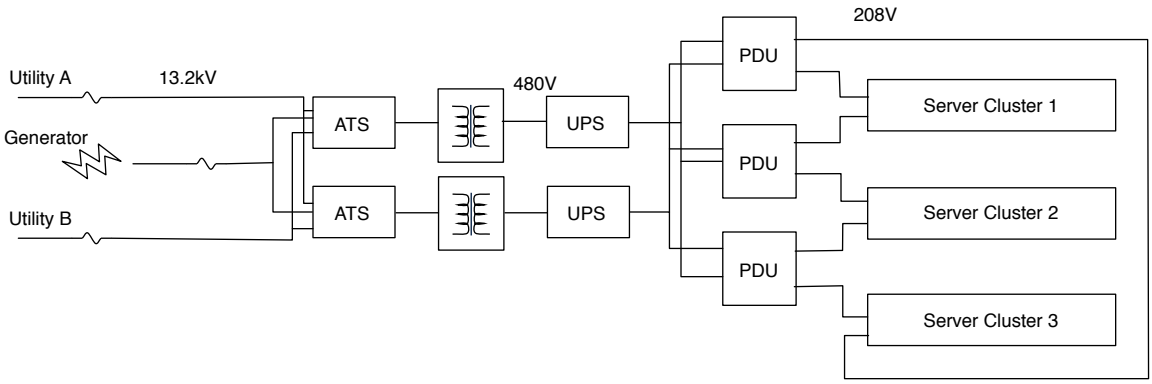


Figure B.2: Example power delivery system for a high-availability data center.

- **Power routing.** We develop a linear programming-based heuristic algorithm that assigns each server a power feed and budget to minimize power capping, maintain redundancy against a single PDU fault, and balance power draw across phases.
- **Reduced capital expenses.** Using traces from production systems, we demonstrate that our mechanisms reduce power infrastructure capital costs by 32% without performance degradation. With energy-proportional servers, savings reach 47%.

The rest of this paper is organized as follows. In Section B.2, we provide background on data center power infrastructure and power capping mechanisms. We describe our mechanisms in Section B.3 and detail Power Routing's scheduling algorithm in Section B.4. We evaluate our techniques on our production data center traces in Section B.5. Finally, in Section B.6, we conclude.

B.2 Background

We begin with a brief overview of data center power provisioning infrastructure and power capping mechanisms. A more extensive introduction to these topics is available in [66].

Conventional power provisioning. Today, most data centers operate according to power provisioning policies that assure sufficient capacity for every server. These policies are enforced by the data center operators at system installation time, by prohibiting deployment of any machine that creates the potential for overload. Operators do their best to estimate systems' peak power draws, either through stress-testing, from vendor-supplied calculators, or through de-rating of nameplate specifications.

In high-availability data centers, power distribution schemes must also provision redundancy for fault tolerance; system deployments are further restricted by these redundancy

requirements. The Uptime Institute classifies data centers into tiers based on the nature and objectives of their infrastructure redundancy [112]. Some data centers provide no fault tolerance (Tier-1), or provision redundancy only within major power infrastructure components, such as the UPS system (Tier-2). Such redundancy allows some maintenance of infrastructure components during operation, and protects against certain kinds of faults, but numerous single points-of-failure remain. Higher-tier data centers provide redundant power delivery paths to each server. Power Routing is targeted at these data centers, as it exploits the redundant delivery paths to shift power delivery capacity.

Example: A high-availability power system. Figure B.2 illustrates an example of a high-availability power system design and layout for a data center with redundant distribution paths. The design depicted here is based on the power architecture of the Michigan Academic Computer Center (MACC), the largest (10,000 square feet; 288 racks; 4MW peak load including physical infrastructure) of the three facilities providing utilization traces for this study. Utility power from two substations and a backup generator enter the facility at high voltage (13.2 kVAC) and meet at redundant automated transfer switches (ATS) that select among these power feeds. These components are sized for the peak facility load (4MW), including all power infrastructure and cooling system losses. The ATS outputs in turn are transformed to a medium voltage (480 VAC) and feed redundant uninterruptible power supply (UPS) systems, which are also each sized to support the entire facility. These in turn provide redundant feeds to an array of power distribution units (PDUs) which further transform power to 208V 3-phase AC.

PDUs are arranged throughout the data center such that each connects to two neighboring system clusters and each cluster receives redundant power feeds from its two neighboring PDUs. The power assignments wrap from the last cluster to the first. We refer to this PDU arrangement as a *wrapped topology*. The wrapped topology provides redundant delivery paths with minimal wiring and requires each PDU to be sized to support at most 150% of the load of its connected clusters, with only a single excess PDU beyond the minimum required to support the load (called an "N+1" configuration). In the event of any PDU fault, 50% of its supported load fails over to each of its two neighbors. PDUs each support only a fraction of the data center's load, and can range in capacity from under ten to several hundred kilowatts.

Power is provided to individual servers through connectors (called "whips"), that split the three phases of the 208VAC PDU output into the 120VAC single-phase circuits familiar from residential wiring. (Some equipment may operate at higher voltages or according to other international power standards.) Many modern servers include redundant power supplies, and provide two power cords that can be plugged into whips from each PDU. In

such systems, the server internally switches or splits its load among its two power feeds. For servers that provide only a single power cord, a rack-level transfer switch can connect the single cord to redundant feeds.

The capital costs of the power delivery infrastructure are concentrated at the large, high-voltage components: PDUs, UPSs, facility-level switches, generators, transformers and the utility feed. The rack-level components cost a few thousand dollars per rack (on the order of $1 per provisioned Watt), while the facility-level components can cost $10-$25 per provisioned Watt [66, 111], especially in facilities with such high levels of redundancy. With Power Routing, we focus on reducing the required provisioning of the facility-scale components while assuring a balanced load over the PDUs. Though circuit breakers typically limit current both at the PDU's breaker panels and on the individual circuits in each whip, it is comparatively inexpensive to provision these statically to avoid overloads. Though Power Routing is applicable to manage current limits on individual circuits, we focus on enforcing limits at the PDU level in this work.

Phase balance. In addition to enforcing current limits and redundancy, it is also desirable for a power provisioning scheme to balance power draw across the three phases of AC power supplied by each PDU. Large phase imbalances can lead to current spikes on the neutral wire of a 3-phase power bus, voltage and current distortions on the individual phases, and generally increase heat dissipation and reduce equipment lifetime [49]. Data center operators typically manually balance power draw across phases by using care in connecting equipment to particular receptacles wired to each phase. Power Routing can automatically enforce phase balance by including it as explicit constraints in its scheduling algorithm.

Power capping. Conservative, worst-case design invariably leads to power infrastructure over-provisioning [47, 91, 36, 118]. Power capping mechanisms allow data center operators to sacrifice some performance in rare utilization spikes in exchange for substantial cost savings in the delivery infrastructure, without the risk of cascading failures due to an overload. In these schemes, some centralized control mechanism establishes a power budget for each server (e.g., based on historical predictions or observed load in the previous time epoch). An actuation mechanism then enforces these budgets.

The most common method of enforcing power budgets is through control loops that sense actual power draw and modulate processor frequency and voltage to remain within budget. Commercial systems from IBM [86] and HP [53] can enforce budgets to sub-watt granularities at milli-second timescales. Researchers have extended these control mechanisms to enforce caps over multi-server chassis, larger ensembles, and entire clusters [91, 60, 117, 38], examine optimal power allocation among heterogeneous servers [39]

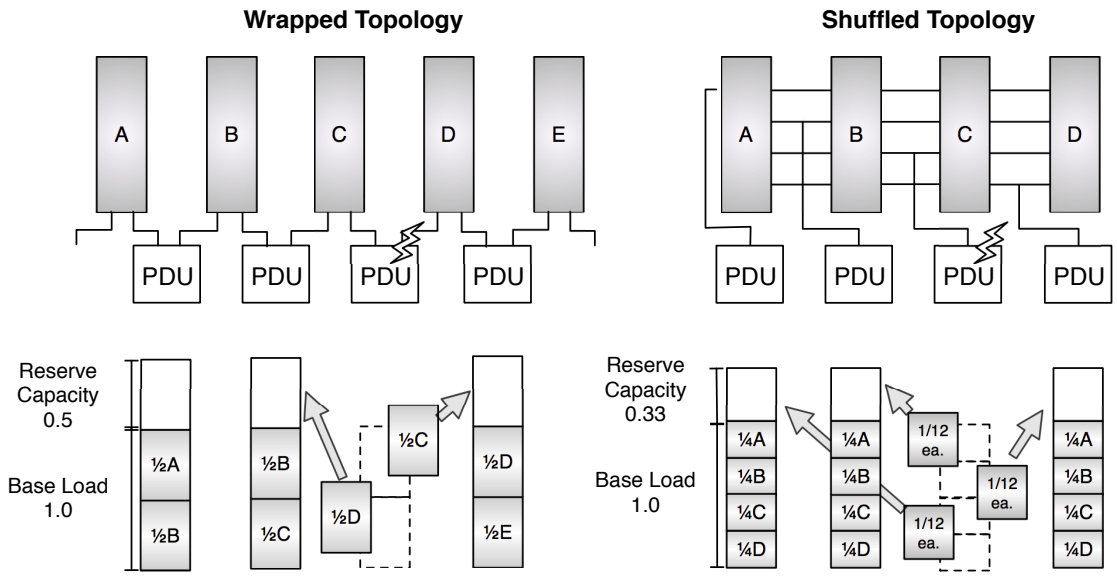
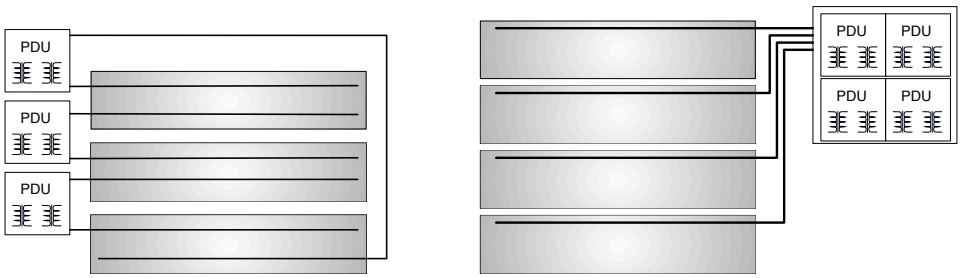


Figure B.3: Reduced reserve capacity under shuffled topologies (4 PDUs, fully-connected topology).

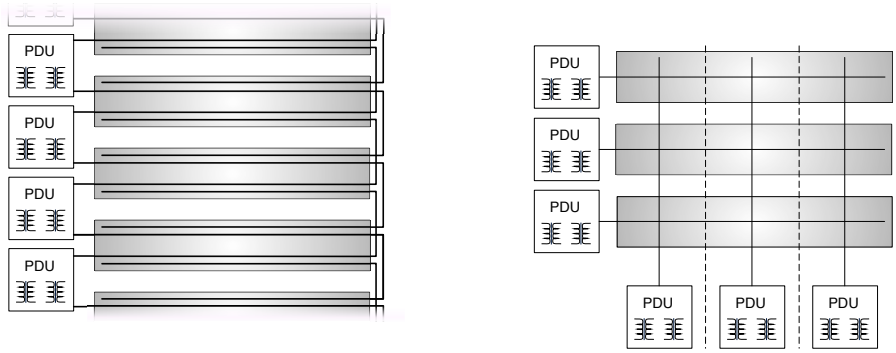
and identify the control stability challenges when capping at multiple levels of the power distribution hierarchy [89, 118]. Others have examined extending power management to virtualized environments [75]. Soft fuses [47] apply the notion of power budgets beyond the individual server and enforce sustained power budgets, which allow for transient overloads that the power infrastructure can support. Finally, prior work considers alternative mechanisms for enforcing caps, such as modulating between active and sleep states [40].

Like prior work, Power Routing relies on a power capping mechanism as a safety net to ensure extended overloads can not occur. However, Power Routing is agnostic to how budgets are enforced. For simplicity, we assume capping based on dynamic frequency and voltage scaling, the dominant approach.

Though rare, peak utilization spikes do occur in some facilities. In particular, if a facility runs a single distributed workload balanced over all servers (e.g., as in a web search cluster), then the utilization of all servers will rise and fall together [36]. No scheme that over-subscribes the physical infrastructure can avoid performance throttling for such systems. The business decision of whether throttling is acceptable in these rare circumstances is beyond the scope of this study; however, for any given physical infrastructure budget, Power Routing reduces performance throttling relative to existing capping schemes, by shifting loads among PDUs to locate and exploit spare capacity.



(a) Wrapped (conventional) (b) Fully-connected



(c) Serpentine (d) X-Y

Figure B.4: Shuffled power distribution topologies.

B.3 Power Routing.

Power Routing relies on two central concepts. First, it exploits *shuffled topologies* for power distribution to increase the connectivity between servers and diverse PDUs. Shuffled topologies spread responsibility to sustain the load on a failing PDU, reducing the required reserve capacity per PDU. Second, Power Routing relies on a *scheduling* algorithm to assign servers' load across redundant distribution paths while balancing loads over PDUs and AC phases. When loads are balanced, the provisioned capacity of major power infrastructure components (PDUs, UPSs, generators, and utility feeds) can be reduced, saving capital costs. We first detail the design and advantages of shuffled topologies, and then discuss Power Routing.

B.3.1 Shuffled Topologies.

In high-availability data centers, servers are connected to two PDUs to ensure uninterrupted operation in the event of a PDU fault. A naive (but not unusual) connection topology provisions paired PDUs for each cluster of machines. Under this data center design, each PDU must be sized to support the full worst-case load of the entire cluster; hence, the power infrastructure is 50% utilized in the best case. As described in Section B.2, the more

sophisticated "wrapped" topology shown in Figure B.2 splits a failed PDU's load over two neighbors, allowing each PDU to be sized to support only 150% of its nominal primary load.

By spreading the responsibility for failover further, to additional PDUs, the spare capacity required of each PDU can be reduced—the more PDUs that cooperate to cover the load of a failed PDU, the less reserve capacity is required in the data center as a whole. In effect, the reserve capacity in each PDU protects multiple loads (which is acceptable provided there is only a single failure).

Figure B.3 illustrates the differing reserve capacity requirements of the wrapped topology and a shuffled topology where responsibility for reserve capacity is spread over three PDUs. The required level of reserve capacity at each PDU is approximately X/N, where X represents the cluster power demand, and N the number of PDUs cooperating to provide reserve capacity. (Actual reserve requirements may vary depending on the instantaneous load on each phase).

The savings from shuffled topologies do not require any intelligent switching capability; rather, they require only increased diversity in the distinct combinations of primary and secondary power feeds for each server (ideally covering all combinations equally).

The layout of PDUs and power busses must be carefully considered to yield feasible shuffled wiring topologies. Our distribution strategies rely on overhead power busses [94] rather than conventional under-floor conduits to each rack. The power busses make it easier (and less costly) to connect many, distant racks to a PDU. Power from each nearby bus is routed to a panel at the top of each rack, and these in turn connect to vertical whips (i.e., outlet strips) that supply power to individual servers. The whips provide outlets in pairs (or a single outlet with an internal transfer switch) to make it easy to connect servers while assuring an appropriate mix of distinct primary and secondary power feed combinations.

Though overhead power busses are expensive, they still account for a small fraction of the cost of large-scale data center power infrastructure. Precise quantification of wiring costs is difficult without detailed facility-specific architecture and engineering. We neglect differences in wiring costs when estimating data center infrastructure costs, and instead examine the (far more significant) impact that topologies have on the capacity requirements of the high-voltage infrastructure. The primary difficulty of complex wiring topologies lies in engineering the facility-specific geometry of the large (and dangerous) high-current overhead power rails; a challenge that we believe is surmountable.

We propose three shuffled power distribution topologies that improve on the wrapped topology of current high-availability data centers. The *fully connected* topology collocates all PDUs in one corner of the room, and routes power from all PDUs throughout the entire

facility. This topology is not scalable. However, we study it as it represents an upper bound on the benefits of shuffled topologies. We further propose two practical topologies. The *X-Y* topology divides the data center into a checkerboard pattern of power zones, routing power both north-south and east-west across the zones. The *serpentine* topology extends the concept of the wrapped topology (see Figure B.2) to create overlap among neighboring PDUs separated by more than one row.

Each distribution topology constrains the set of power feed combinations available in each rack in a different manner. These constraints in turn affect the set of choices available to the Power Routing scheduler, thereby impacting its effectiveness.

Wrapped Topology. Figure 2.4(a) illustrates the wrapped topology, which is our term for the conventional high-availability data center topology (also seen in Figure B.2). This topology provides limited connectivity to PDUs, and is insufficient for Power Routing.

Fully-connected Topology. Figure 2.4(b) illustrates the fully-connected topology. Under this topology, power is routed from every PDU to every rack. As noted above, the fully-connected topology does not scale and is impractical in all but the smallest data centers. However, one scalable alternative is to organize the data center as disconnected islands of fully-connected PDUs and rack clusters. Such a topology drastically limits Power Routing flexibility, but can scale to arbitrary-sized facilities.

Serpentine Topology. Figure 2.4(c) illustrates the serpentine topology. Under this topology, PDUs are located at one end of the data centers' rows, as in the wrapped topology shown in Figure B.2. However, whereas in the wrapped topology a power bus runs between two equipment rows from the PDU to the end of the facility, in the serpentine topology, the power bus then bends back, returning along a second row. This snaking bus pattern is repeated for each PDU, such that two power busses run in each aisle and four busses are adjacent to each equipment row. The pattern scales to larger facilities by adding PDUs and replicating the pattern over additional rows. It scales to higher PDU connectivity by extending the serpentine pattern with an additional turn.

X-Y Topology. Figure 2.4(d) illustrates the X-Y topology. Under this topology, the data center is divided into square zones in a checkerboard pattern. PDUs are located along the north and west walls of the data center. Power busses from each PDU route either north-south or east-west along the centerline of a row (column) of zones. Hence, two power busses cross in each zone. These two busses are connected to each rack in the zone. This topology scales to larger facilities in a straight-forward manner, by adding zones to the "checkerboard." It scales to greater connectivity by routing power busses over the zones in pairs (or larger tuples).

B.3.2 Power Routing.

Power Routing leverages shuffled topologies to achieve further capital cost savings by under-provisioning PDUs relative to worst-case demand. The degree of under-provisioning is a business decision made at design time (or when deploying additional systems) based on the probability of utilization spikes and the cost of performance throttling (i.e., the risk of failing to meet a service-level agreement). Power Routing shifts spare capacity to cover local power demand spikes by controlling the assignment of each server to its primary or secondary feed. The less correlation there is among spikes, the more effective Power Routing will be at covering those spikes by shifting loads rather than throttling performance. Power Routing relies on a capping mechanism to prevent overloads when spikes cannot be covered.

Power Routing employs a centralized control mechanism to assign each server to its primary or secondary power feed and set power budgets for each server to assure PDU overloads do not occur. Each time a server's power draw increases to its pre-determined cap (implying that performance throttling will be engaged), the server signals the Power Routing controller to request a higher cap. If no slack is available on the server's currently active power feed, the controller invokes a scheduling algorithm (detailed in Section B.4) to determine new power budgets and power feed assignments for all servers to try to locate slack elsewhere in the power distribution system. The controller will reduce budgets for servers whose utilization has decreased and may reassign servers between their primary and secondary feeds to create the necessary slack. If no solution can be found (e.g., because aggregate power demand exceeds the facilities' total provisioning), the existing power cap remains in place and the server's performance is throttled.

In addition to trying to satisfy each server's desired power budget, the Power Routing scheduler also maintains sufficient reserve capacity at each PDU to ensure continued operation (under the currently-established power budgets) even if any single PDU fails. A PDU's required reserve capacity is given by the largest aggregate load served by another PDU for which it acts as the secondary (inactive) feed.

Finally, the Power Routing scheduler seeks to balance load across the three AC phases of each PDU. As noted in Section B.2, phase imbalance can lead to numerous electrical problems that impact safety and availability. The scheduler constrains the current on each of the three phases to remain within a 20% margin.

The key novelty of Power Routing lies in the assignment of servers to power feeds; sophisticated budgeting mechanisms (e.g., which assign asymmetric budgets to achieve higher-level QoS goals) have been extensively studied [91, 60, 117, 38, 89, 118, 75, 39]. Hence, in this paper, we focus our design and evaluation on the power feed scheduling

mechanism and do not explore QoS-aware capping in detail.

B.3.3 Implementation.

Power Routing comprises four elements: (1) an actuation mechanism to switch servers between their two redundant power feeds; (2) the centralized controller that executes the power feed scheduling algorithm; (3) a communications mechanism for the controller to direct switching activity and assign budgets; and (4) a power distribution topology that provisions primary and secondary power feeds in varying combinations to the receptacles in each rack.

Switching power feeds. The power feed switching mechanism differs for single- and dual-corded servers. In a single-corded server, an external transfer switch attaches the server to its primary or secondary power feed. In the event of a power interruption on the active feed, the transfer switch seamlessly switches the load to the alternative feed (a local, automatic action). The scheduler assures that all PDUs have sufficient reserve capacity to supply all loads that may switch to them in the event of any single PDU failure. To support phase balancing, the transfer switch must be capable of switching loads across out-of-phase AC sources fast enough to appear uninterrupted to computer power supplies. External transfer switches of this sort are in wide-spread use today, and retail for several hundred dollars. In contrast to existing transfer switches, which typically switch entire circuits (several servers), Power Routing requires switching at the granularity of individual receptacles, implying somewhat higher cost. For dual-corded servers, switching does not require any additional hardware, as the switching can be accomplished through the systems' internal power supplies.

Control unit. The Power Routing control unit is a microprocessor that orchestrates the power provisioning process. Each time scheduling is invoked, the control unit performs four steps: (1) it determines the desired power budget for each server; (2) it schedules each server to its primary or secondary power feed; (3) it assigns a power cap to each server (which may be above the request, allowing headroom for utilization increase, or below, implying performance throttling); and (4) it communicates the power cap and power feed assignments to all devices. The control unit can be physically located within the existing intelligence units in the power delivery infrastructure (most devices already contain sophisticated, network-attached intelligence units). Like other power system components, the control unit must include mechanisms for redundancy and fault tolerance. Details of the control unit's hardware/software fault tolerance are beyond the scope of this study; the challenges here mirror those of the existing intelligence units in the power infrastructure.

The mechanisms used in each of the control unit's four steps are orthogonal. As this

study is focused on the novel scheduling aspect of Power Routing (step 2), we explore only relatively simplistic policies for the other steps. We determine each server's desired power budget based in its peak demand in the preceding minute. Our power capping mechanism assigns power budgets that throttle servers to minimize the total throttled power.

Communication. Communication between the control unit and individual servers/-transfer switches is best accomplished over the data center's existing network infrastructure, for example, using the Simple Network Management Protocol (SNMP) or BACnet. The vast majority of power provisioning infrastructure already supports these interfaces. Instantaneous server power draws and power budgets can also typically be accessed through SNMP communication with the server's Integrated Lights Out (ILO) interface.

Handling uncontrollable equipment. Data centers contain myriad equipment that draw power, but cannot be controlled by Power Routing (e.g., network switches, monitors). The scheduler must account for the worst-case power draw of such equipment when calculating available capacity on each PDU and phase.

B.3.4 Operating Principle.

Power Routing relies on the observation that individual PDUs are unlikely to reach peak load simultaneously. The power distribution system as a whole operates in one of three regimes. The first, most common case is that the load on all PDUs is below their capacity. In this case, the power infrastructure is over-provisioned, power capping is unnecessary, and the entire data center operates at full performance. At the opposite extreme, when servers demand more power than is available, the power infrastructure is under-provisioned, all PDUs will be fully loaded, and power capping (e.g., via performance throttling) is necessary. In either of these regimes, Power Routing has no impact; the power infrastructure

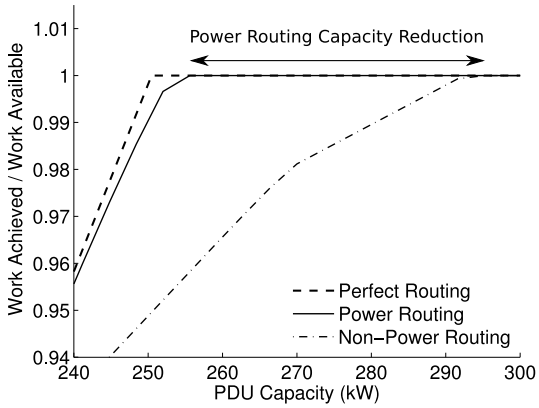


Figure B.5: Shuffled Topologies: 6 PDUs, fully-connected

is simply under- (over-) provisioned relative to the server demand.

Power Routing is effective in the intermediate regime where some PDUs are overloaded while others have spare capacity. In current data centers, this situation will result in performance throttling that Power Routing can avoid.

To illustrate how Power Routing affects performance throttling, we explore its performance envelope near the operating region where aggregate power infrastructure capacity precisely meets demand. Figure B.5 shows the relationship between installed PDU capacity and performance throttling (in terms of the fraction of offered load that is met) with and without Power Routing (6 PDUs, fully-connected topology) and contrast these against an ideal, perfectly-balanced power distribution infrastructure. The ideal infrastructure can route power from any PDU to any server and can split load fractionally over multiple PDUs. (We detail the methodology used to evaluate Power Routing and produce these results in Section B.5.1 below.)

The graph provides two insights into the impact of Power Routing. First, we can use it to determine how much more performance Power Routing achieves for a given infrastructure investment relative to conventional and ideal designs. This result can be obtained by comparing vertically across the three lines for a selected PDU capacity. As can be seen, Power Routing closely tracks the performance of the ideal power delivery infrastructure, recovering several percent of lost performance relative to a fully-connected topology without power routing.

The graph can also be used to determine the capital infrastructure savings that Power Routing enables while avoiding performance throttling altogether. Performance throttling becomes necessary at the PDU capacity where each of the three power distributions dips below 1.0. The horizontal distance between these intercepts is the capacity savings, and is labeled "Power Routing Capacity Reduction" in the figure. In the case shown here, Power Routing avoids throttling at a capacity of 255 kW, while 294 kW of capacity are needed without Power Routing. Power Routing avoids throttling, allowing maximum performance with less investment in power infrastructure.

B.4 Scheduling

Power Routing relies on a centralized scheduling algorithm to assign power to servers. Each time a server requests additional power (as a result of exhausting its power cap) the scheduler checks if the server's current active power feed has any remaining capacity, granting it if possible. If no slack exists, the scheduler attempts to create a new allocation schedule for the entire facility that will eliminate or minimize the need for capping. In addition

to considering the actual desired power budget of each server, the scheduler must also provision sufficient reserve capacity on each feed such that the feed can sustain its share of load if any PDU fails. Finally, we constrain the scheduler to allow only phase-balanced assignments where the load on the three phases of any PDU differ by no more than 20% of the per-phase capacity.

The scheduling process comprises three steps: gathering the desired budget for each server, solving for an assignment of servers to their primary or secondary feeds, and then, if necessary, reducing server's budgets to meet the capacity constraints on each feed.

Whereas sophisticated methods for predicting power budgets are possible [26], we use a simple policy of assigning each server a budget based on its average power demand in the preceding minute. More sophisticated mechanisms are orthogonal to the scheduling problem itself.

Solving the power feed assignment problem optimally, even without redundancy, is an NP-Complete problem. It is easy to see that power scheduling $\in$ NP; a nondeterministic algorithm can enumerate a set of assignments from servers to PDUs and then check in polynomial time that each PDU is within its power bounds. To show that power scheduling is NP-Complete we transform PARTITION to it [42]. For a given instance of PARTITION of finite set A and a size $s(a) \in \mathbb{Z}+$ for each $a \in A$: we would like to determine if there is a subset $A' \in A$ such that the $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$. Consider A as the set of servers, with $s(a)$ corresponding to server power draw. Additionally consider two PDUs each of power capacity $\sum_a s(a)/2$. These two problems are equivalent. Thus, a polynomial time solution to power scheduling will yield a polynomial time solution to PARTITION (implying power scheduling is NP-Complete).

In data centers of even modest size, brute force search for an optimal power feed assignment is infeasible. Hence, we resort to a heuristic approach to generate an approximate solution.

We first optimally solve a power feed assignment problem allowing servers to be assigned fractionally across feeds using linear programming. This linear program can be solved in polynomial time using standard methods [31]. From the exact fractional solution, we then construct an approximate solution to the original problem (where entire servers must be assigned a power feed). Finally, we check if the resulting assignments are below the capacity of each power feed. If any feed's capacity is violated, we invoke a second optimization step to choose power caps for all servers.

Determining optimal caps is non-trivial because of the interaction between a server's power allocation on its primary feed, and the reserve capacity that allocation implies on its secondary feed. We employ a second linear programming step to determine a capping

strategy that maximizes the amount of power allocated to servers (as opposed to reserve capacity).

Problem formulation. We formulate the linear program based on the power distribution topology (i.e., the static assignment of primary and secondary feeds to each server), the desired server power budgets, and the power feed capacities. For each pair of power feeds we calculate $Power_{i,j}$, the sum of power draws for all servers connected to feeds i and j. (Our algorithm operates at the granularity of individual phases of AC power from each PDU, as each phase has limited ampacity). $Power_{i,j}$ is 0 if no server shares feeds i and j (e.g., if the two feeds are different phases from the same PDU or no server shares those PDUs). Next, for each pair of feeds, we define variables $Feed_{i,j}i$ and $Feed_{i,j}j$ to account for the server power from $Power_{i,j}$ routed to feeds i and j, respectively. Finally, a single global variable, $Slack$, represents the maximum unallocated power on any phase after all assignments are made. With these definitions, the linear program maximizes $Slack$ subject to the following constraints:

$\forall i, j \neq i$, i and j are any phases on different PDUs:

$$Feed_{i,j}i + Feed_{i,j}j = Power_{i,j} \tag{B.1}$$

$$\sum_{k \neq i} Feed_{i,k}i + \sum_{l in j' s PDU} Feed_{i,l}l + Slack \leq Capacity(i) \tag{B.2}$$

And constraints for distinct phases i and j within a single PDU:

$$|\sum_{k \neq i} Feed_{i,k}i - \sum_{k \neq j} Feed_{j,k}j| \leq .2 \times Capacity(i, j) \tag{B.3}$$

With the following bounds:

$$-\infty \leq Slack \leq \infty \tag{B.4}$$

$$\forall i, j \neq i : Feed_{i,j}i, Feed_{i,j}j \geq 0 \tag{B.5}$$

Equation B.1 ensures that power from servers connected to feeds i and j is assigned to one of those two feeds. Equation B.2 restricts the sum of all power assigned to a particular feed i, plus the reserve capacity required on i should feeds on j's PDU fail, plus the excess slack to be less than the capacity of feed i. Finally, equation B.3 ensures that phases are balanced across each PDU. A negative $Slack$ indicates that more power is requested by servers than is available (implying that there is no solution to the original, discrete scheduling problem without power capping).

We use the fractional power assignments from the linear program to schedule servers to feeds. For a given set of servers, s, connected to both feed i and feed j, the fractional solution will indicate that $Feed_{i,j}i$ watts be assigned to i and $Feed_{i,j}j$ to j. The scheduler

must create a discrete assignment of servers to feeds to approximate the desired fractional assignments as closely as possible, which is itself a bin packing problem. To solve this sub-problem efficiently, the scheduler sorts the set s descending by power and repeatedly assign the largest unassigned server to i or j, whichever has had less power assigned to it thus far (or whichever has had less power relative to its capacity if the capacities differ).

If a server cannot be assigned to either feed without violating the feed's capacity constraint, then throttling may be necessary to achieve a valid schedule. The server is marked as "pending" and left temporarily unassigned. By the nature of the fractional solution, at most one server in the set can remain pending. This server must eventually be assigned to one of the two feeds; the difference between this discrete assignment and the optimal fractional assignment is the source of error in our heuristic. By assigning the largest servers first we attempt to minimize this error. Pending servers will be assigned to the feed with the most remaining capacity once all other servers have been assigned.

The above optimization algorithm assumes that each pair of power feeds shares several servers in common, and that the power drawn by each server is much less than the capacity of the feed. We believe that plausible power distribution topologies fit this restriction.

Following server assignment, if no feed capacity constraints have been violated, the solution is complete and all servers are assigned caps at their requested budgets. If any slack remains on a feed, it can be granted upon a future request without re-invoking the scheduling mechanism, avoiding unnecessary switching.

If any capacity constraints have been violated, a new linear programming problem is formulated to select power caps that maximize the amount of power allocated to servers (as opposed to reserve capacity for fail-over). We scale back each feed such that no PDU supplies more power than its capacity, even in the event that another PDU fails. The objective function maximizes the sum of the server budgets. We assume that servers can be throttled to any frequency from idle to peak utilization and that the relationship and limits of frequency and power scaling are known a priori. Note, however, that this formulation ignores heterogeneity in power efficiency, performance, or priority across servers; it considers only the redundancy and topology constraints of the power distribution network. An analysis of more sophisticated mechanisms for choosing how to cap servers that factors in these considerations is outside the scope of this paper.

B.5 Evaluation

Our evaluation demonstrates the effectiveness of shuffled topologies and Power Routing at reducing the required capital investment in power infrastructure to meet a high-

availability data center's reliability and power needs. First, we demonstrate how shuffled topologies reduce the reserve capacity required to provide single-PDU-fault tolerance. Then, we examine the effectiveness of Power Routing at further reducing provisioning requirements as a function of topology, number of PDUs, and workload. Finally, we show how Power Routing will increase in effectiveness as server power management becomes more sophisticated and the gap between servers' idle and peak power demands grows.

B.5.1 Methodology

We evaluate Power Routing through analysis of utilization traces from a large collection of production systems. We simulate Power Routing's scheduling algorithm and impact on performance throttling and capital cost.

Traces. We collect utilization traces from three production facilities: (1) *EECS servers*, a small cluster of departmental servers (web, email, login, etc.) operated by the Michigan EECS IT staff; (2) *Arbor Lakes Data Center*, a 1.5MW facility supporting the clinical operations of the University of Michigan Medical Center; and (3) *Michigan Academic Computer Center (MACC)*, a 4MW high-performance computing facility operated jointly by the University of Michigan, Internet2, and Merit that runs primarily batch processing jobs. These sources provide a diverse mix of real-world utilization behavior. Each of the traces ranges in length from three to forty days sampling server utilization once per minute. We use these traces to construct a hypothetical high-availability hosting facility comprising 400 medical center servers, 300 high performance computing nodes, and a 300-node web search cluster. The simulated medical center and HPC cluster nodes each replay a trace from a specific machine in the corresponding real-world facility. The medical center systems tend to be lightly loaded, with one daily utilization spike (which we believe to be daily backup processing). The HPC systems are heavily loaded. As we do not have access to an actual 300-node web search cluster, we construct a cluster by replicating the utilization trace of a single production web server over 300 machines. The key property of this synthetic search cluster is that the utilization on individual machines rises and falls together in response to user traffic, mimicking the behavior reported for actual search clusters [36]. We analyze traces for a 24-hour period. Our synthetic cluster sees a time-average power draw of 180.5 kW, with a maximum of 208.7 kW and standard deviation of 9 kW.

Power. We convert utilization traces to power budget requests using published SPECPower results [105]. Most of our traces have been collected from systems where no SPECPower result has been published; for these, we attempt to find the closest match based on vendor descriptions and the number and model of CPUs and installed memory. As SPECPower only provides power at intervals of 10% utilization, we use linear interpolation to approxi-

mate power draw in between these points.

Prior work [36, 91] has established that minute-grained CPU utilization traces can predict server-grain power draw to within a few percent. Because of the scope of our data collection efforts, finer-grained data collection is impractical. Our estimates of savings from Power Routing are conservative; finer-grained scheduling might allow tighter tracking of instantaneous demand.

To test our simulation approach, we have validated simulation-derived power values against measurements of individual servers in our lab. Unfortunately, the utilization and power traces available from our production facilities are not exhaustive, which precludes a validation experiment where we compare simulation-derived results to measurements for an entire data center.

Generating data center topologies. For each power distribution topology described in Section B.3.1, we design a layout of our hypothetical facility to mimic the typical practices seen in the actual facilities. We design layouts according to the policies the Michigan Medical Center IT staff use to manage their *Arbor Lakes* facility. Each layout determines an assignment of physical connections from PDUs to servers. Servers that execute similar applications are collocated in the same rack, and, hence, in conventional power delivery topologies, are connected to the same PDU. Where available, we use information about the actual placement of servers in racks to guide our placement. Within a rack, servers are assigned across PDU phases in a round-robin fashion. We attempt to balance racks across PDUs and servers within racks across AC phases based on the corresponding system's power draw at 100% utilization. No server is connected to two phases of the same PDU, as this arrangement does not protect against PDU failure. We use six PDUs in all topologies unless otherwise noted.

Metrics. We evaluate Power Routing based on its impact on server throttling activity and data center capital costs. As the effect of voltage and frequency scaling on performance varies by application, we instead use the fraction of requested server power budget that was not satisfied as a measure of the performance of capping techniques. Under this metric, the "cost" of failing to supply a watt of requested power is uniform over all servers, obviating the need to evaluate complex performance-aware throttling mechanisms (which are orthogonal to Power Routing). Our primary evaluation metric is the minimum total power delivery capacity required to assure zero performance throttling, as this best illustrates the advantage of Power Routing over conventional worst-case provisioning.

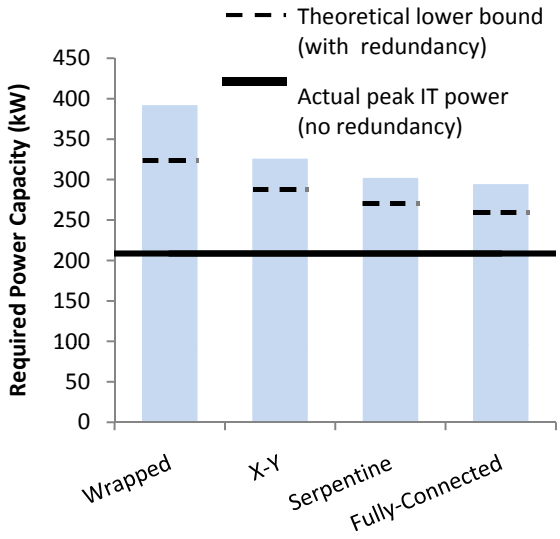


Figure B.6: Minimum capacity for redundant operation under shuffled topologies (no Power Routing).

B.5.2 Impact of Shuffled Topologies

We first compare the impact of shuffled topologies on required power infrastructure capacity. Shuffled topologies reduce the reserve capacity that each PDU must sustain to provide fault tolerance against single-PDU failure. We examine the advantage of several topologies relative to the baseline high-availability "wrapped" data center topology, which requires each PDU to be over-provisioned by 50% of its nominal load. We report the total power capacity required to prevent throttling for our traces. We assume that each PDU must maintain sufficient reserve capacity at all times to precisely support the time-varying load that might fail over to it.

Differences in the connectivity of the various topologies result in differing reserve capacity requirements. For an ideal power distribution infrastructure (one in which load is perfectly balanced across all PDUs), each PDU must reserve $\frac{1}{c+1}$ to support its share of a failing PDU's load, where c is the *fail-over connectivity* of the PDU. Fail-over connectivity counts the number of distinct neighbors to which a PDU's servers will switch in the event of failure. It is two for the wrapped topology, four for serpentine, and varies as a function of the number of PDUs for X-Y and fully-connected topologies. As the connectivity increases, reserve requirements decrease, but with diminishing returns.

To quantify the impact of shuffled topologies, we design an experiment where we statically assign each server the best possible primary and secondary power feed under the constraints of the topology. We balance the average power draw on each PDU using each server's average power requirement over the course of the trace. (We assume this average

to be known a priori for each server.)

In Figure B.6 each bar indicates the required power capacity for each topology to meet its load and reserve requirements in all time epochs (i.e., no performance throttling or loss of redundancy) for a 6 PDU data center. For 6 PDUs, the fail-over connectivities are 2, 3, 4, and 5 for the wrapped, X-Y, serpentine, and fully-connected topologies, respectively. The dashed line on each bar indicates the topology's theoretical lower-bound capacity requirement to maintain redundancy if server power draw could be split dynamically and fractionally across primary and secondary PDUs (which Power Routing approximates). The gap between the top of each bar and the dashed line arises because of the time-varying load on each server, which creates imbalance across PDUs and forces over-provisioning. The solid line crossing all bars indicates the data center's peak power draw, ignoring redundancy requirements (i.e., the actual peak power supplied to IT equipment).

Topologies with higher connectivity require less reserve capacity, though the savings taper off rapidly. The X-Y and serpentine topologies yield impressive savings and are viable and scalable from an implementation perspective. Nevertheless, there is a significant gap between the theoretical (dashed) and practical (bar) effectiveness of shuffled topologies. As we show next, Power Routing closes this gap.

B.5.3 Impact of Power Routing

Power Routing effectiveness. To fully explore Power Routing effectiveness, we repeated the analysis above for all four topologies (wrapped, X-Y, serpentine, and fully-connected) and contrast the capacity required to avoid throttling for each. For comparison, we also reproduce the capacity requirements without Power Routing (from Figure B.6). We show results in Figure B.7. Again, a dashed line represents the theoretical minimum capacity necessary to maintain single-PDU fault redundancy for our workload and the given topology; the solid line marks the actual peak IT power draw. Because the overall load variation in our facilities is relatively small (HPC workloads remain pegged at near-peak utilization; the medical facility is over-provisioned to avoid overloading), we expect a limited opportunity for Power Routing. Nonetheless, we reduce required power delivery capacity for all topologies (except wrapped) by an average of 12%.

From the figure, we see that the sparsely-connected wrapped topology is too constrained for Power Routing to be effective; Power Routing requires 20% more than the theoretical lower bound infrastructure under this topology. The three shuffled topologies, however, nearly reach their theoretical potential, even with a heuristic scheduling algorithm. Under the fully-connected topology, Power Routing comes within 2% of the bound, reducing power infrastructure requirements by over 39kW (13%) relative to the same topology with-

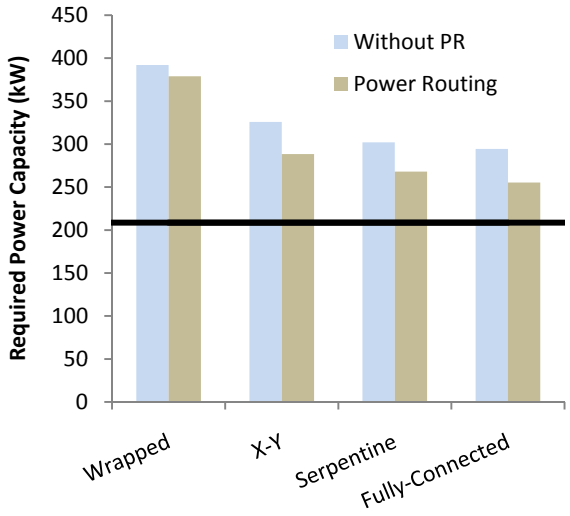


Figure B.7: Power Routing infrastructure savings as a function of topology.

out Power Routing and more than 35% relative to the baseline wrapped topology without Power Routing. Our result indicates that more-connected topologies offer an advantage to Power Routing by providing more freedom to route power. However, the the more-practical topologies yield similar infrastructure savings; the serpentine topology achieves 32% savings relative to the baseline.

Sensitivity to number of PDUs. The number of PDUs affects Power Routing effectiveness, particularly for the fully-connected topology. Figure B.8 shows this sensitivity for four to eight PDUs. For a fixed total power demand, as the number of PDUs increases, each individual PDU powers fewer servers and requires less capacity. With fewer servers, the variance in power demands seen by each PDU grows (i.e., statistical averaging over the servers is lessened), and it becomes more likely that an individual PDU will overload. Without Power Routing, this effect dominates, and we see an increase in required infrastructure capacity as the number of PDUs increases beyond 6. At the same time, increasing the number of PDUs offers greater connectivity for certain topologies, which in turn lowers the required slack that PDUs must reserve and offers Power Routing more choices as to where to route power. Hence, Power Routing is better able to track the theoretical bound and the required power capacity decreases with more PDUs.

B.5.4 Power Routing For Low Variance Workloads

The mixed data center trace we study is representative of the diversity typical in most data centers. Nevertheless, some data centers run only a single workload on a homogeneous cluster. Power Routing exploits diversity in utilization patterns to shift power delivery slack; hence, its effectiveness is lower in homogeneous clusters.

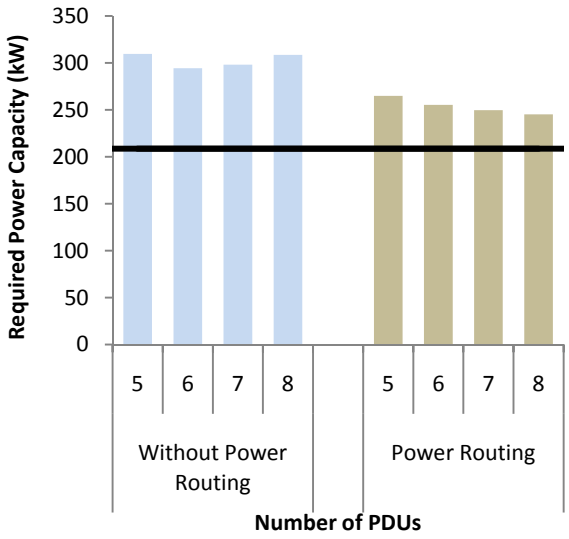
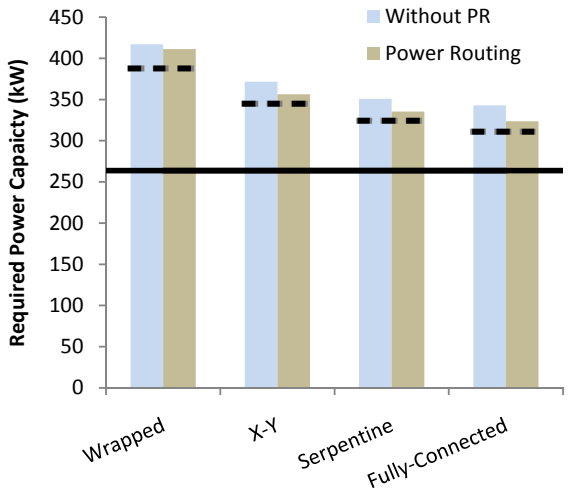


Figure B.8: Sensitivity of the fully-connected topology to number of PDUs.

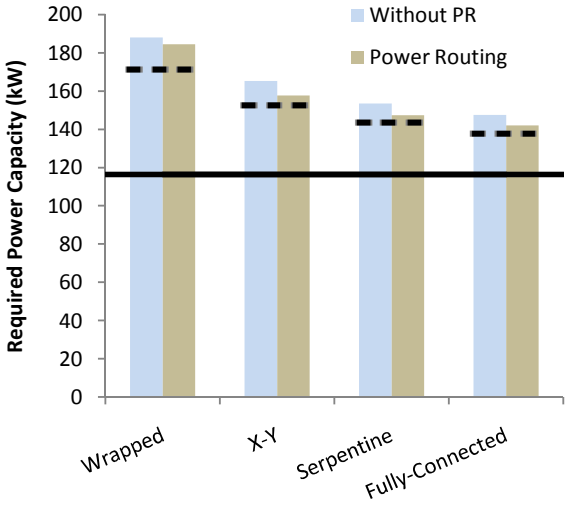
To explore these effects, we construct Power Routing test cases for 1000-server synthetic clusters where each server runs the same application. We do not study the web search application in isolation; in this application, the utilization on all servers rise and fall together, hence, the load on all PDUs is inherently balanced and there is no opportunity (nor need) for Power Routing. Instead, we evaluate Power Routing using the medical center traces and high performance computing traces, shown in Figures 2.9(a) and 2.9(b), respectively.

The high performance computing cluster consumes a time-average power of 114.9 kW, a maximum of 116.4 kW, and a standard deviation of 0.8 kW while the medical center computing traces consume a time-average power of 254.6 kW, with maximum 263.6 kW and standard deviation 2.4 kW. In both cases, the variability is substantially lower than in the heterogeneous data center test case.

Although Power Routing comes close to achieving the theoretical lower bound infrastructure requirement in each case, we see that there is only limited room to improve upon the non-Power Routing case. Even the baseline wrapped topology requires infrastructure that exceeds the theoretical bound by only 7.5% for the high performance computing cluster and 5% for the medical data center. We conclude that Power Routing offers substantial improvement only in heterogeneous clusters and applications that see power imbalance, a common case in many facilities.



(a) Arbor Lakes (clinical operations)



(b) MACC (high-performance computing)

Figure B.9: Power Routing effectiveness in homogeneous data centers.

B.5.5 Power Routing With Energy-Proportional Servers

As the gap between servers' peak and idle power demands grows (e.g., with the advent of energy-proportional computers [7]), we expect the potential for Power Routing to grow. The increase in power variance leads to a greater imbalance in power across PDUs, increasing the importance of correcting this imbalance with Power Routing.

To evaluate this future opportunity, we perform an experiment where we assume all servers are energy-proportional—that is, servers whose power draw varies linearly with utilization—with an idle power of just 10% of peak. This experiment models servers equipped with PowerNap [69], which allows servers to sleep during the millisecond-scale idle periods between task arrivals. We repeat the experiment shown in Figure B.7 under

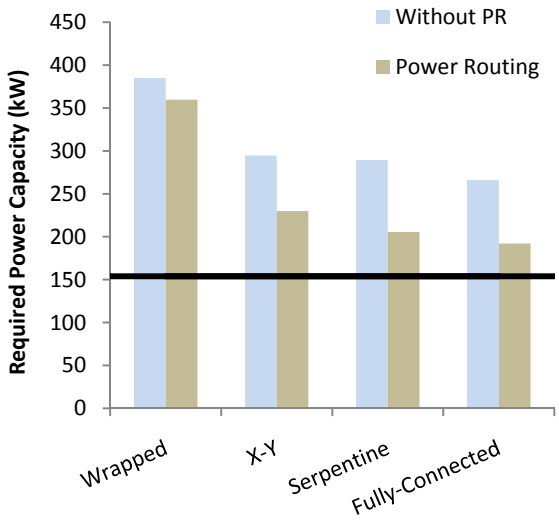


Figure B.10: Impact with energy-proportional servers.

this revised server power model. The results are shown in Figure B.10. Under these assumptions, our traces exhibit a time-average power of 99.8 kW, maximum of 153.9 kW, and standard deviation of 18.9 kW.

Power Routing is substantially more effective when applied to energy-proportional servers. However, the limitations of the wrapped topology are even more pronounced in this case, and Power Routing provides little improvement. Under the more-connected topologies, Power Routing is highly effective, yielding reductions of 22%, 29%, and 28% for the X-Y, serpentine, and fully-connected topologies, respectively, relative to their counterparts without Power Routing. As before, the more-connected topologies track their theoretical lower bounds more tightly. Relative to the baseline wrapped topology, a serpentine topology with Power Routing yields a 47% reduction in required physical infrastructure capacity. It is likely that as computers become more energy-proportional, power infrastructure utilization will continue to decline due to power imbalances. Power Routing reclaims much of this wasted capacity.

B.5.6 Limitations

Our evaluation considers workloads in which any server may be throttled, and our mechanisms make no effort to select servers for throttling based on any factors except maximizing the utilization of the power delivery infrastructure. In some data centers, it may be unacceptable to throttle performance. These data centers cannot gain a capital cost savings from under-provisioning; their power infrastructure must be provisioned for worst case load. Nonetheless, these facilities can benefit from intermixed topologies (to reduce reserve capacity for fault tolerance) and from the phase-balancing possible with Power

Routing.

B.6 Conclusion

The capital cost of power delivery infrastructure is one of the largest components of data center cost, rivaling energy costs over the life of the facility. In many data centers, expansion is limited because available power capacity is exhausted. To extract the most value out of their infrastructure, data center operators over-subscribe the power delivery system. As long as individual servers connected to the same PDU do not reach peak utilization simultaneously, over-subscribing is effective in improving power infrastructure utilization. However, coordinated utilization spikes do occur, particularly among collocated machines, which can lead to substantial throttling even when the data center as a whole has spare capacity.

In this paper, we introduced a pair of complementary mechanisms, shuffled power distribution topologies and Power Routing, that reduce performance throttling and allow cheaper capital infrastructure to achieve the same performance levels as current data center designs. Shuffled topologies permute power feeds to create strongly-connected topologies that reduce reserve capacity requirements by spreading responsibility for fault tolerance. Power Routing schedules loads across redundant power delivery paths to shift power delivery slack to satisfy localized utilization spikes. Together, these mechanisms reduce capital costs by 32% relative to a baseline high-availability design when provisioning for zero performance throttling. Furthermore, with energy-proportional servers, the power capacity reduction increases to 47%.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] Data, data everywhere. *The Economist*, 2010. Special Report: Managing Information.

[2] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: a tutorial. *IEEE Computer*, 29(12):66–76, December 1996.

[3] Rakesh Agrawal and H. V. Jagadish. Recovery algorithms for database machines with nonvolatile main memory. In *Proceedings of the Sixth International Workshop on Database Machines*, pages 269–285, 1989.

[4] G. Alvarez, W. Burkhard, L. Stockmeyer, and F. Cristian. Declustered disk array architectures with optimal and near-optimal parallelism. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, 1998.

[5] Mary Baker, Satoshi Asami, Etienne Deprit, John Ouseterhout, and Margo Seltzer. Non-volatile memory for fast, reliable file systems. In *Proc. of the 5th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 10–22, 1992.

[6] Chuck Ballard, Dan Behman, Asko Huumonen, Kyosti Laiho, Jan Lindstrom, Marko Milek, Michael Roche, John Seery, Katriina Vakkila, Jamie Watters, and An oni Wolski. IBM solidDB: Delivering data with extreme speed. 2011.

[7] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *Computer*, 40(12), 2007.

[8] Stephan Baumann, Giel de Nijs, Michael Strobel, and Kai-Uwe Sattler. Flashing databases: expectations and limitations. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, 2010.

[9] Daniel Bausch, Ilia Petrov, and Alejandro Buchmann. Making cost-based query optimization asymmetry-aware. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 24–32, New York, NY, USA, 2012. ACM.

[10] Dina Bitton, David J. Dewitt, and Carolyn Turbyfill. Benchmarking database systems - a systematic approach. In *Proceedings of the Very Large Database Conference*, 1983.

[11] Colin Blundell, Milo M.K. Martin, and Thomas F. Wenisch. Invisifence: Performance-transparent memory ordering in conventional multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 233–244, New York, NY, USA, 2009. ACM.

[12] Simona Boboila and Peter Desnoyers. Performance models of flash-based solid-state drives for real workloads. In *Proceedings of the 2011 IEEE 27th Symposium on Mass Storage Systems and Technologies*, MSST '11, pages 1–6, Washington, DC, USA, 2011. IEEE Computer Society.

[13] Luc Bouganim, Bjrn r Jnsson, and Philippe Bonnet. uFLIP: understanding flash IO patterns. In *Fourth Biennial Conference on Innovative Data Systems Research*, 2009.

[14] G. W. Burr, B. N. Kurdi, J. C. Scott, C. H. Lam, K. Gopalakrishnan, and R. S. Shenoy. Overview of candidate device technologies for storage-class memory. *IBM J. of Research and Development*, 52:449–464, 2008.

[15] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proc. of the 43rd International Symp. on Microarchitecture*, pages 385–395, 2010.

[16] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. In *Proc. of the 17th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 387–400, 2012.

[17] Y. A. Çengel. *Heat transfer: a practical approach*. McGraw Hill Professional, 2 edition, 2003.

[18] CEC (California Energy Commission). The nonresidential alternative calculation method (acm) approval manual for the compliance with california's 2001 energy efficiency standards, april 2001.

[19] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: Bulk enforcement of sequential consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 278–289, New York, NY, USA, 2007. ACM.

[20] Luis Ceze, James Tuck, Pablo Montesinos, and Josep Torrellas. Bulksc: Bulk enforcement of sequential consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 278–289, New York, NY, USA, 2007. ACM.

[21] Dhruva R. Chakrabarti and Hans-J. Boehm. Durability Semantics for Lock-based Multithreaded Programs. In *Proceedings of the 2013 USENIX Workshop on Hot Topics in Parallelism (HotPar)*, June 2013.

[22] Jeffrey S. Chase, Darrell C. Anderson, Prachi N. Thakar, Amin M. Vahdat, and Ronald P. Doyle. Managing energy and server resources in hosting centers. *SIGOPS Oper. Syst. Rev.*, 35(5), 2001.

[23] Feng Chen, David A. Koufaty, and Xiaodong Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the 11th International Joint Conference on Measurement and Modeling of Computer Systems*, 2009.

[24] Shimin Chen, Phillip B. Gibbons, and Suman Nath. Rethinking database algorithms for phase change memory. In *Proc. of the 5th Bienniel Conf. on Innovative Data Systems Research*, pages 21–31, 2011.

[25] Vijay Chidambaram, Thanumalayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the 2013 Symposium on Operating System Principles*, November 2013.

[26] Jeonghwan Choi, Sriram Govindan, Bhuvan Urgaonkar, and Anand Sivasubramanium. Profiling, prediction, and capping of power consumption in consolidated environments. In *MASCOTS*, September 2008.

[27] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live migration of virtual machines. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation (NSDI)*, 2005.

[28] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. From aries to mars: Transaction support for next-generation, solid-state drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 197–212, New York, NY, USA, 2013. ACM.

[29] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. NV-Heaps: making persistent objects fast and safe with next-generation, non-volatile memories. In *Proc. of the 16th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 105–118, 2011.

[30] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through byte-addressable, persistent memory. In *Proc. of the 22nd Symp. on Operating Systems Principles*, pages 133–146, 2009.

[31] T Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[32] Oracle Corporation. Oracle database documentation library. http://docs.oracle.com/cd/E16655_01/server.121/e17643/storage.htm#CACJFFJI.

[33] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. Hekaton: SQL server's memory-optimized OLTP engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1243–1254, 2013.

[34] Jaeyoung Do and Jignesh M. Patel. Join processing for flash SSDs: remembering past lessons. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, 2009.

[35] DOE (Department of Energy). Doe 2 reference manual, part 1, version 2.1, 1980.

[36] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz Andre Barroso. Power provisioning for a warehouse-sized computer. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA)*, 2007.

[37] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. High performance database logging using storage class memory. In *Proc. of the 27th International Conf. on Data Engineering*, pages 1221–1231, 2011.

[38] Mark E. Femal and Vincent W. Freeh. Boosting data center performance through non-uniform power allocation. In *Proceedings of Second International Conference on Autonomic Computing (ICAC)*, 2005.

[39] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, and Charles Lefurgy. Optimal power allocation in server farms. In *Proceedings of ACM SIGMETRICS 2009 Conference on Measurement and Modeling of Computer Systems*, 2009.

[40] Anshul Gandhi, Mor Harchol-Balter, Rajarshi Das, Charles Lefurgy, and Jeffrey Kephart. Power capping via forced idleness. In *Workshop on Energy-Efficient Design*, 2009.

[41] Gregory R. Ganger, Marshall Kirk McKusick, Craig A. N. Soules, and Yale N. Patt. Soft Updates: A Solution to the Metadata Update Problem in File Systems. *ACM Transactions on Computer Systems*, 18(2):127–153, May 2000.

[42] MR Garey, D.S. Johnson, R.C. Backhouse, G. von Bochmann, D. Harel, CJ van Rijsbergen, J.E. Hopcroft, J.D. Ullman, A.W. Marshall, I. Olkin, et al. *A Guide to the Theory of Computers and Intractability*. Springer.

[43] Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Two techniques to enhance the performance of memory consistency models. In *In Proceedings of the 1991 International Conference on Parallel Processing*, pages 355–364, 1991.

[44] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th annual international symposium on Computer Architecture*, ISCA '90, pages 15–26, New York, NY, USA, 1990. ACM.

[45] Chris Gniady, Babak Falsafi, and T. N. Vijaykumar. Is sc + ilp = rc? In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, ISCA '99, pages 162–171, Washington, DC, USA, 1999. IEEE Computer Society.

[46] James R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, ISCA '83, pages 124–131, New York, NY, USA, 1983. ACM.

[47] Sriram Govindan, Jeonghwan Choi, Bhuvan Urgaonkar, Anand Sivasubramaniam, and Andrea Baldini. Statistical profiling-based techniques for effective power provisioning in data centers. In *Proceedings of the 4th ACM European Conference on Computer systems (EuroSys)*, 2009.

[48] Jim Gray. The transaction concept: virtues and limitations (invited paper). In *Proceedings of the seventh international conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[49] T.M. Gruzs. A survey of neutral currents in three-phase computer power systems. *IEEE Transactions on Industry Applications*, 26(4), Jul/Aug 1990.

[50] Laura M. Haas, Michael J. Carey, Miron Livny, and Amit Shukla. Seeking the truth about ad hoc join costs. *VLDB Journal*, 6(3), 1997.

[51] James Hamilton. Internet-scale service infrastructure efficiency. Keynote at the International Symposium on Computer Architecture (ISCA), 2009.

[52] Taliver Heath, Ana Paula Centeno, Pradeep George, Luiz Ramos, Yogesh Jaluria, and Ricardo Bianchini. Mercury and freon: temperature emulation and management for server systems. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.

[53] HP Staff. HP power capping and dynamic power capping for ProLiant servers. Technical Report TC090303TB, HP, 2009.

[54] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: a scalable storage manager for the multicore era. In *Proc. of the 12th International Conf. on Extending Database Technology*, pages 24–35, 2009.

[55] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. Aether: a scalable approach to logging. *Proc. VLDB Endow.*, pages 681–692, September 2010.

[56] Stefanos Kaxiras and Margaret Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan Claypool, 2008.

[57] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, September 1979.

[58] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, 2009.

[59] Sang-Won Lee, Bongki Moon, Chanik Park, Jae-Myung Kim, and Sang-Woo Kim. A case for flash memory SSD in enterprise database applications. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2008.

[60] C. Lefurgy, X. Wang, and M. Ware. Power capping: A prelude to power shifting. *Cluster Computing*, 11(2), 2008.

[61] Charles Lefurgy, Karthick Rajamani, Freeman Rawson, Wes Felter, Michael Kistler, and Tom W. Keller. Energy management for commercial servers. *Computer*, 36(12), 2003.

[62] Yinan Li, Bingsheng He, Qiong Luo, and Ke Yi. Tree indexing on flash disks. In *Proceedings of the 2009 IEEE International Conference on Data Engineering*, 2009.

[63] Changhui Lin, Vijay Nagarajan, Rajiv Gupta, and Bharghava Rajaram. Efficient sequential consistency via conflict ordering. *SIGARCH Comput. Archit. News*, 40(1):273–286, March 2012.

[64] Heikki Linnakangas. http://www.postgresql.org/message-id/464F3C5D.2000700@enterprisedb.com.

[65] David E. Lowell and Peter M. Chen. Free Transactions with Rio Vista. In *Proceedings of the 1997 Symposium on Operating Systems Principles*, pages 92–101, October 1997.

[66] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer*. Morgan Claypool, 2009.

[67] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, 2005.

[68] Jennifer Mankoff, Robin Kravets, and Eli Blevis. Some computer science issues in creating a sustainable world. *IEEE Computer*, 41(8), August 2008.

[69] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: eliminating server idle power. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2009.

[70] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[71] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, pages 94–162, March 1992.

[72] J. Moore, J. S. Chase, and P. Ranganathan. Weatherman: Automated, online and predictive thermal mapping and management for data centers. In *ICAC '06: Proceedings of the 2006 IEEE International Conference on Autonomic Computing*, 2006.

[73] Dushyanth Narayanan and Orion Hodson. Whole-system persistence. In Tim Harris and Michael L. Scott, editors, *ASPLOS*, pages 401–410. ACM, 2012.

[74] R. Nathuji, A. Somani, K. Schwan, and Y. Joshi. Coolit: Coordinating facility and it management for efficient datacenters. In *HotPower '08: Workshop on Power Aware Computing and Systems*, December 2008.

[75] Ripal Nathuji and Karsten Schwan. Virtualpower: coordinated power management in virtualized enterprise systems. In *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

[76] Simo Neuvonen, Antoni Wolski, Markku Manner, and Vilho Raatikka. Telecom Application Transaction Processing Benchmark. http://tatpbenchmark.sourceforge.net.

[77] Wee Teck Ng and Peter M. Chen. Integrating reliable memory in databases. In *Proc. of the International Conf. on Very Large Data Bases*, pages 76–85, 1997.

[78] Oracle America. Extreme performance using Oracle TimesTen in-memory database, an Oracle technical white paper. July 2009.

[79] Luca Parolini, Bruno Sinopoli, and Bruce H. Krogh. Reducing data center energy consumption via coordinated cooling and load management. In *HotPower '08: Workshop on Power Aware Computing and Systems*, December 2008.

[80] C.D. Patel, R. Sharma, C.E. Bash, and A. Beitelmal. Thermal considerations in cooling large scale high compute density data centers. In *The Eighth Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Systems.*, 2002.

[81] Steven Pelley. Atomic memory trace. https://github.com/stevenpelley/atomic-memory-trace.

[82] Steven Pelley, David Meisner, Thomas F Wenisch, and James W VanGilder. Understanding and abstracting total data center power. In *Workshop on Energy-Efficient Design*, 2009.

[83] Steven Pelley, David Meisner, Pooya Zandevakili, Thomas F. Wenisch, and Jack Underwood. Power routing: dynamic power provisioning in the data center. In *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS XV, pages 231–242, New York, NY, USA, 2010. ACM.

[84] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. Storage management in the nvram era. *PVLDB*, 7(2):121–132, 2013.

[85] Steven Pelley, Thomas F. Wenisch, and Kristen LeFevre. Do query optimizers need to be SSD-aware? In *Workshop on Accelerating Data Management Systems using Modern Processor and Storage Architectures*, 2011.

[86] P. Popa. Managing server energy consumption using IBM PowerExecutive. Technical report, IBM, 2006.

[87] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proc. of the 42nd International Symp. on Microarchitecture*, pages 14–23, 2009.

[88] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proc. of the 36th International Symp. on Computer Architecture*, pages 24–33, 2009.

[89] Ramya Raghavendra, Parthasarathy Ranganathan, Vanish Talwar, Zhikui Wang, and Xiaoyun Zhu. No "power" struggles: coordinated multi-level power management for the data center. In *Proceeding of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, March 2008.

[90] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw Hill, 3rd edition, 2002.

[91] Parthasarathy Ranganathan, Phil Leech, David Irwin, and Jeffrey Chase. Ensemble-level power management for dense blade servers. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA)*, 2006.

[92] Parthasarathy Ranganathan, Vijay S. Pai, and Sarita V. Adve. Using speculative retirement and larger instruction windows to narrow the performance gap between memory consistency models. In *Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '97, pages 199–210, New York, NY, USA, 1997. ACM.

[93] N. Rasmussen. Electrical efficiency modeling for data centers. Technical Report #113, APC by Schneider Electric, 2007.

[94] N. Rasmussen. A scalable, reconfigurable, and efficient data center power distribution architecture. Technical Report #129, APC by Schneider Electric, 2009.

[95] Suzanne Rivoire, Parthasarathy Ranganathan, and Christos Kozyrakis. A comparison of high-level full-system power models. In *HotPower '08: Workshop on Power Aware Computing and Systems*, December 2008.

[96] David Roberts, Taeho Kgil, and Trevor Mudge. Integrating NAND flash devices onto servers. *Communications of the ACM*, 52, April 2009.

[97] Chris Ruemmler and John Wilkes. An introduction to disk drive modeling. *Computer*, 27(3):17–28, March 1994.

[98] Kenneth Salem and Sedat Akyürek. Management of partially safe buffers. *IEEE Trans. Comput.*, pages 394–407, March 1995.

[99] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1979.

[100] André Seznec. A case for two-way skewed-associative caches. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA)*, 1993.

[101] L.D. Shapiro. Join processing in database systems with large main memories. *ACM Transactions on Database Systems*, 11(3), 1986.

[102] Abhayendra Singh, Satish Narayanasamy, Daniel Marino, Todd Millstein, and Madanlal Musuvathi. End-to-end sequential consistency. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 524–535, Washington, DC, USA, 2012. IEEE Computer Society.

[103] SPARC International, Inc. *The SPARC Architecture Manual*, version 9 edition, 1994.

[104] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proc. of the 33rd International Conf. on Very Large Data Bases*, pages 1150–1160, 2007.

[105] The Standard Performance Evaluation Corporation (SPEC). SPECpower Benchmark Results. http://www.spec.org/power_ssj2008/results.

[106] R. Tozer, C. Kurkjian, and M. Salim. Air management metrics in data centers. In *ASHRAE 2009*, January 2009.

[107] Transaction Processing Performance Council (TPC). TPC-B Benchmark. http://www.tpc.org/tpcb/.

[108] Transaction Processing Performance Council (TPC). TPC-C Benchmark. http://www.tpc.org/tpcc/.

[109] Transaction Processing Performance Council (TPC). TPC-H Benchmark. http://www.tpc.org/tpch/.

[110] Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. Query processing techniques for solid state drives. In *Proceedings of the 35th SIGMOD International Conference on Management of Data*, 2009.

[111] W. Turner and J. Seader. Dollars per kW plus dollars per square foot are a better datacenter cost model than dollars per square foot alone. Technical report, Uptime Institute, 2006.

[112] W. Turner, J. Seader, and K. Brill. Industry standard tier classifications define site infrastructure performance. Technical report, Uptime Institute, 2005.

[113] U.S. EPA. Report to congress on server and data center energy efficiency. Technical report, USEPA, August 2007.

[114] J. W. VanGilder and S. K. Shrivastava. Capture index: An airflow-based rack cooling performance metric. *ASHRAE Transactions*, 113(1), 2007.

[115] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proc. of the 9th Usenix Conference on File and Storage Technologies*, pages 61–75, 2011.

[116] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: lightweight persistent memory. In *Proc. of the 16th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 91–104, 2011.

[117] Xiaorui Wang and Ming Chen. Cluster-level feedback power control for performance optimization. In *Proceedings of the 14th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2008.

[118] Xiaorui Wang, Ming Chen, Charles Lefurgy, and Tom W. Keller. SHIP: Scalable hierarchical power control for large-scale data centers. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2009.

[119] Thomas F. Wenisch, Anastasia Ailamaki, Babak Falsafi, and Andreas Moshovos. Mechanisms for store-wait-free multiprocessors. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 266–277, New York, NY, USA, 2007. ACM.

[120] Shaoyi Yin, Philippe Pucheral, and Xiaofeng Meng. A sequential indexing scheme for flash-based embedded systems. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 2009.

[121] P. C. Yue and C. K. Wong. Storage cost considerations in secondary index selection. *International Journal of Parallel Programming*, 4, 1975.

[122] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. Kiln: Closing the performance gap between systems with and without persistence support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.