Thesis Proposal: 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
2013

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF APPENDICES

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. Wheres 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 thesis 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 to research programming interfaces and hardware designs to enable intuitive, high performance recoverable data structures with NVRAM, extending memory consistency with persistent semantics to introduce *Persistent Memory Consistency*.

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 [11].

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 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 thesis 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 complex, 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 quickly becomes the dominant bottleneck. Such capacity is only reasonably achieved 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 [70]. 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 [65] 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 exhibit nearly balanced read and write capacity in large sequential runs – 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 [55], 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, which I expect will provide high transaction performance and low recovery latency 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 [11]. 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 that 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* to improve 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 is currently under review at VLDB.

1.3 Persistent Memory Consistency

The previous work looks at how OLTP recovery mechanisms should be designed, considering only the average delay incurred by persist barriers. To complete my dissertation I intend to investigate specific programming models that provide persist barriers, their probable performance, and ease of programming. 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. I define the rules that govern persistent write order as *Persistent Memory Consistency*. Relaxed persistent consistency models use persist barriers to enforce specific write orders, guaranteeing that data is correctly recovered after failure. I explore existing solutions, describe why they fall short, and place them into a more precisely defined taxonomy of persistent consistency. I introduce relaxed persistent consistency models and qualitatively reason about their performance in Chapter VI. Interestingly, persistent memory consistency models may be de-coupled from the underlying memory consistency model, separately enforcing the order in which writes becomes durable and the order in which writes become visible to other threads. Finally, I highlight new optimizations for persist barriers and persistent consistency, providing example persistent data structures in Chapter VII. Future work will include performance evaluations, new consistency models, and the design of additional data structures.

1.4 Data Center Infrastructure

The primary themes of this thesis include performance, cost efficiency, and reliability. While I focus on storage architectures, I have published additional work regarding the cost and reliability of data center infrastructure. Appendix A contains "Understanding and Abstracting Total Data Center Power," [63] published at WEED 2009. 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," [64] published at ASPLOS 2010. PowerRouting distributes power infrastructure throughout the data center to minimize 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 this investigation I discovered that, in many regards, industry is well 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 proposal is formatted according to the thesis guidelines for the University of Michigan. Where possible, I intend to use portions of the proposal in my final thesis. The dissertation is organized as follows: Chapter II contains background information on storage technologies and database optimizations. 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. I consider this work complete and do not propose any extension. In Chapter IV I describe potential software designs for OLTP using NVRAM and a methodology to evaluate NVRAM (devices not readily available) on modern hardware. I propose additional work to extend and complete this project in Sections 4.3.1 and 4.3.2. Chapter V uses my methodology to evaluate several aspects of OLTP running on NVRAM. Again, I propose additional evaluations in Section 5.3. Chapters VI and VII describe in-progress work on persistent memory consistency, persistent data structures, and their performance. The ideas contained in those chapters are not mature and I welcome feedback.

CHAPTER II

Background

This chapter provides details necessary to understand the investigations and experiments in this thesis. 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 these will interact with new storage technologies.

2.1 Storage Technologies

I start with a survey of storage technologies including disk, Flash, and upcoming NV-RAM. For each, I provide the operating principles and important technological trends.

2.1.1 Disk

I provide a summary of disk here for a comparison to other technologies. Disk has been the primary durable and high capacity storage technology for decades. Disks function 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 data as the platter rotates and the correct sector within the track reaches the head. Disk capacity increases

	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.

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 dominant, disk exhibits relatively slow access and undesirable access behavior [76]. 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 sector of interest 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 later in the chapter.

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 even in enterprise-class 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 sacrifices byte addressability and some 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 the maximum lifetime of the cell (e.g., to 10^5 to 10^6 writes [75]). 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 [9]. Table 2.1 lists specifications of a typical Flash SSD as compared to a 10,000 RPM hard drive. 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 [45].

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 [11]. 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 dominate, 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 [68]. Write latency worsens with MLC, where slow, iterative writes are necessary to reliably write to a cell.

Similarly to Flash, resistive NVRAM technologies suffer from limited write endurance; cells may be written reliably only a limited number of times. Previously proposed hardware mechanisms (e.g., Start-Gap [67]) are highly effective in distributing writes across cells and can mitigate write endurance concerns. While such work focuses on volatile applications (NVRAM as a DRAM main memory substitute), it may be extended to durable uses.

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 [12]. Furthermore, block access negates advantages of byte addressability.

Recent research proposes alternative device interfaces for NVRAM. Caulfield *et al.*

propose Moneta and Moneta Direct, a PCIe attached PCM device [13]. 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 [23]. 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.

2.2 Analytics Optimization

Large scale data processing requires efficient use of storage devices. Relational data is stored in tables (relations). Tables contains lists of rows (tuples), each containing values for the different columns (attributes) defined on the table. I discuss two important operators within the relational model most affected by Flash's performance characteristics: scans and joins.

2.2.1 Scans

Whenever queries access data the query optimizer must choose access paths to each table. The goal is to select all relevant rows from the table while touching the least number of storage pages, frequently with the use of indexes. Work on access path selection dates back to the late 1970s [79].

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. 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 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 that must be considered here is that they 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; 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 relation scan becomes faster. 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 [70].

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 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. This operation exists under several names. My description here fits the query plan explanation provided by IBM's DB2. Other databases use different terminology or algorithms to ensure that each store page is fetched exactly once (for example, PostgreSQL uses a bitmap index, sorting the list of pages with tuples that satisfy the query [50]). Rowid-sort scan is the optimal access path for intermediate selectivities. Section 3.3 investigates when optimal access path selection changes between disk and Flash.

2.2.2 Joins

One of the most important aspects of query optimization is choosing appropriate join algorithms for queries. The development of join algorithms and optimization strategies dates back over 30 years [79, 81]. 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 and co-authors [38]. 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 optimiziation.

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 optimizing for disk but actually using Flash).

2.3 Durable and Recoverable Transactions

Many database applications require transaction semantics commonly described as ACID (Atomic, Consistent, Isolated, Durable) [**?**]. 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 [55], 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 centralized 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 that describe page updates and actions, 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 store pages so that they can be replayed if data has not yet written to disk in-place. 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 and store; i.e., 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 the store pages write back for each operation. 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, or page writes back to disk, without a durable history of its modifications in the log.

Though complex, ARIES improves database performance with disks. 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 updates to the data store 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 three simple models: Sequential Consistency (SC), Total Store Order (TSO), and Release Consistency. For the remainder of this section I am referring solely to volatile writes without considering for NVRAM persists. 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. While some relaxed memory models may allow reading of stale cached values, I do not consider that here.

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 remote 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). 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 globally defined order (SC) or 2) relax this guarantee, introducing memory barriers that allow the programmer to enforce a certain order when necessary (e.g., TSO, Release Consistency). While relaxing consistency may provide higher performance, it places a greater burden on programmers to be aware of instruction reordering and correctly use memory barriers.

The consistency model provides the programming abstraction and guarantees on observed memory orders that the programmer can expect. 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 rollback and re-execute memory instructions, providing the illusion that threads execute with strict consistency.

Sequential Consistency. Sequential Consistency [**?**] provides the most intuitive programming model, yet necessarily the worst performance (although modern techniques involving speculation improve performance). All loads and stores appear in a globally consistent order that is an interleaving of program order of all threads. The programmer need not consider memory instructions reordering; barriers are unnecessary.

Total Store Order. TSO [82] provides greater performance than SC at the cost of requiring the programmer to insert memory barriers. Most memory operations in each thread are still observed to occur in program order: 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 occur before that load. These rules guarantee that all stores occur in a globally consistent sequential order. However, loads that occur after a store in program order may reorder and bypass the store, appearing to occur before the store. The justification for doing this is that stores are typically not on the application's critical path – they simply write into the cache and do not result in delays. Loads, on the other hand, may stall due to cache misses. Executing loads as soon as possible minimizes delays.

The programmer is responsible for recognizing any code where allowing a load to bypass a store causes an incorrect result. In this case, a barrier is provided by the architecture to force all loads to delay until the store appears to other threads (or re-execute those loads if some other thread issues conflicting stores). Additionally, TSO defines atomic Read-Modify-Write operations (RMW; e.g., compare-and-swap, atomic add) that additionally act as barriers, preventing instructions from reordering. As concurrent programming commonly uses RMW operations, explicit memory barriers are rarely required in practice.

Release Consistency. Release Consistency [34] provides a relaxed consistency model with two barriers – acquire and release. In the absence of barriers threads may observe memory operations in any order. Barriers are always required to enforce memory ordering. Before reading shared memory an *acquire* barrier must be used. Similarly, after writing to shared memory a *release* barrier is used. The programmer may expect that all writes from the releasing thread before the release barrier will be observed by the acquiring thread after the acquire barrier. RMW implicitly contain both acquire and release barriers.

Release Consistency improves performance by (1) relaxing all consistency guarantees for single threaded code and (2) providing precise memory barriers. Acquire and release barriers order only certain types of accesses, leaving other accesses unconstrained. However, release consistency requires the programmer to be aware of and understand these persist barriers. I believe similar trade off exists for persistence – in order to gain performance we relax persistence constraints so that persist order may not match program order and may not appear as a valid interleaving of all persists. The interaction between persistence, performance, and programmability will be explored in Chapters VI and VII.

CHAPTER III

SSD-aware Query Optimization

This chapter determines if database query optimizers must be SSD-aware. The work was originally published in the Second International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (AMDS 2011), colocated with VLDB [65]. I worked on this project with my advisor, Thomas F. Wenisch, and assistant professor Kristen LeFevre. Working under their advisement, I was the sole graduate student and technical contributor (database and programming effort). Refer to Sections 2.1 and 2.2 for necessary background on this topic. While the project ended in a negative result, it was instrumental developing my understanding of databases and storage management. After completing this work I moved from considering Flash to emerging NVRAM technologies; I propose no future work or extensions to this chapter.

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 drop in 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 several factors 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 [10, 17], including DBMSs [46, 98, 49, 6, 89]. A common theme among these studies is to leverage the better random IO performance of SSDs through radical redesigns of index structures [98, 49] and data layouts [6, 89]. 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 [17] and write endurance [75]. 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 [6] 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 accessed in ways that Flash can take advantage of (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 (e.g., 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 [8] 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 allows us to control precisely the selectivity of each query. 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 [88]) 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 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 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

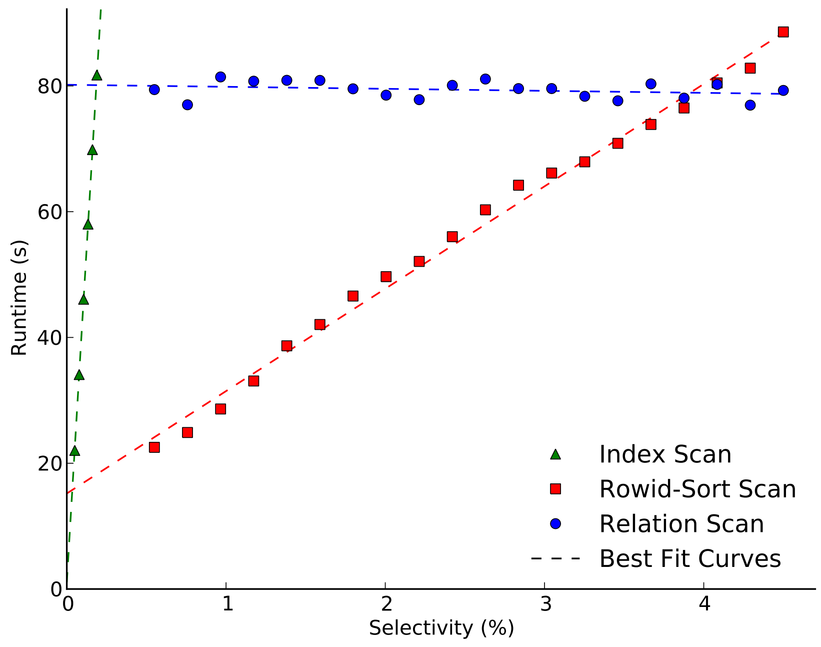


Figure 3.1: Scan operator performance on Disk. Relation scan outperforms the alternatives at selectivities above 4%, while index scan is optimal only for vanishingly small selectivities (e.g., single-tuple queries). Best fit curves drawn for convenience.

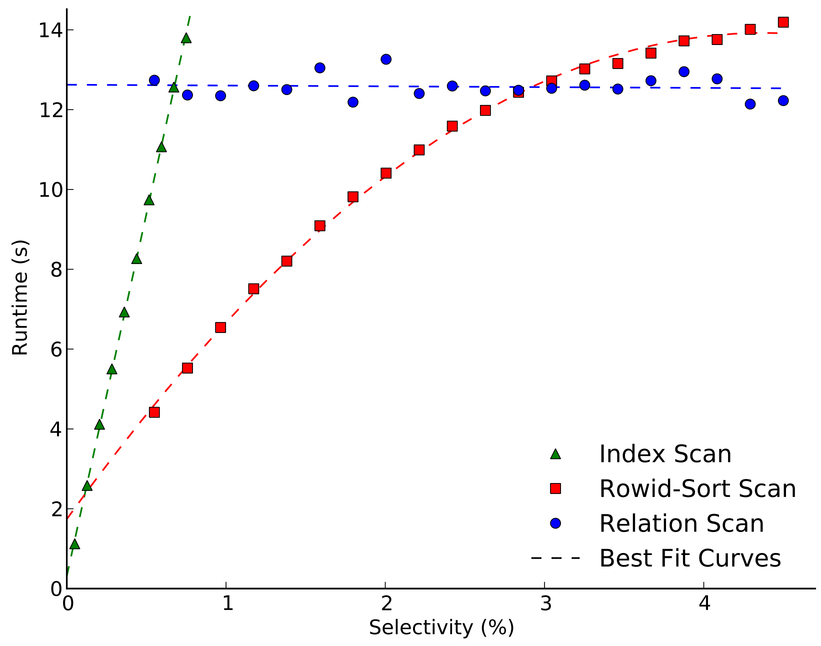


Figure 3.2: Scan operator performance on Flash SSD. Though both break-even points shift as intuition suggests, the selectivities where the optimal decision differs between Disk and SSD are so narrow that the difference is inconsequential in practice. Best fit curves drawn for convenience.

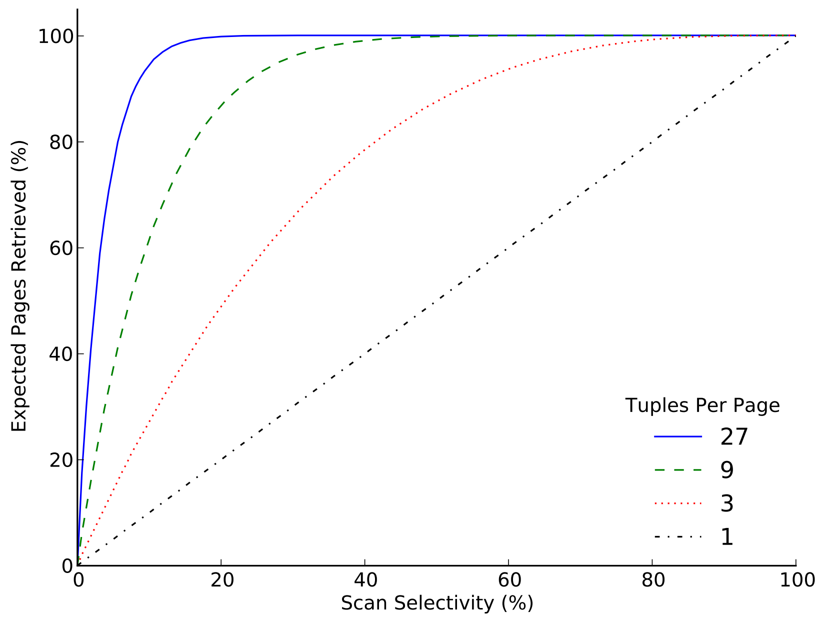


Figure 3.3: Expected page accesses. Index scans touch the majority of pages even at low selectivities.

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 fraction of queries fit into this range, and those 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 the correct 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 scans 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 [99], 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 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 analysis of scans.

Because of the complex interplay between available memory capacity and relation sizes

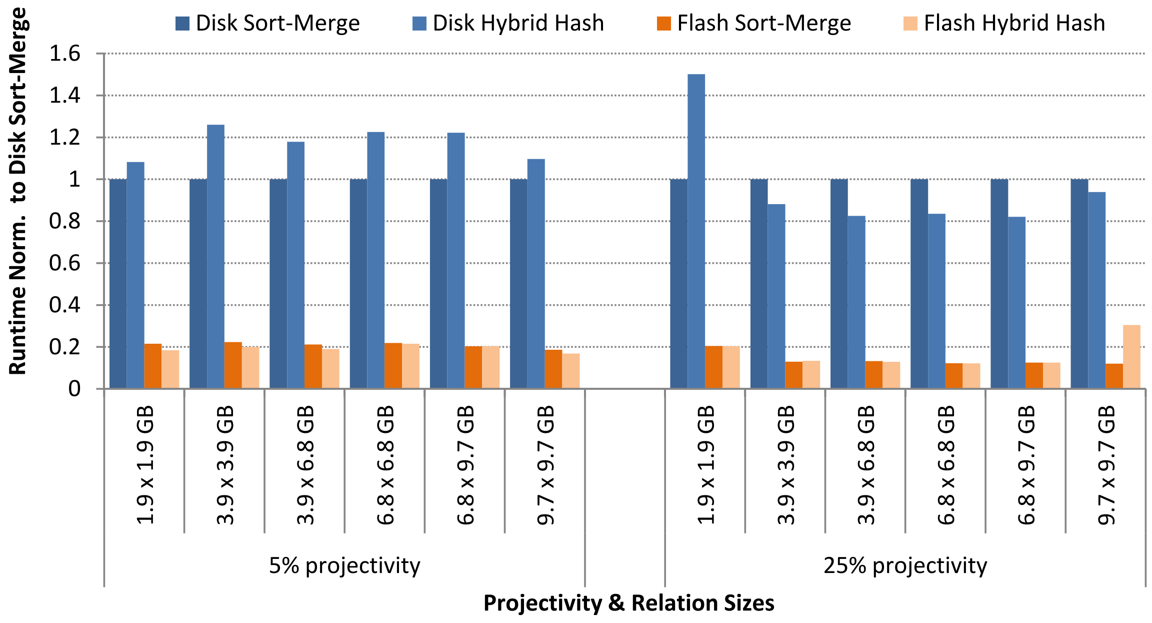


Figure 3.4: Join performance. Join runtimes on Flash SSD and Disk, normalized for each join to the runtime of sort-merge on disk. Though there is significant variability in join algorithm performance on disk, performance variability on SSD is dwarfed by the 6× performance advantage of moving data from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

for join optimization [38], 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 wide 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 I explore a spectrum from 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 runtimes 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

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

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 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 [10, 17]. 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 [46]. 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 [26]. 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 [98, 49]. Baumann *et al.* investigate Flash's performance alongside a hybrid row-column store referred to as "Grouping" [6]. Similarly, Tsirogiannis *et al.* use a column store motivated by the PAX layout to create faster scans and joins [89].

Interestingly, my findings contradict recommendations of these last two 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.

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 demonstrate 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. Work relating to the next two chapters is currently under review at VLDB. I completed this work under the guidance of my advisor, Thomas F. Wenisch, and collaborators at Oracle, Brian T. Gold and Bill Bridge. I was the sole graduate student and technical contributer (programming and running experiments) for the project, although my co-authors heavily contributed to developing the following ideas. 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, a methodology for evaluating NVRAM devices that are not yet available, and a description of several candidate software designs for NVRAM recovery management, to be evaluated in the next chapter. I consider this chapter largely complete with two exceptions. First, I have validated my timing model and intend to include results in the final thesis 4.5.1. Second, my implementation of *NVRAM Group Commit* is more complex than a 12 page journal paper allows room to describe. I intend to include a detailed description in Section 4.3.2.

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 [11]. 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

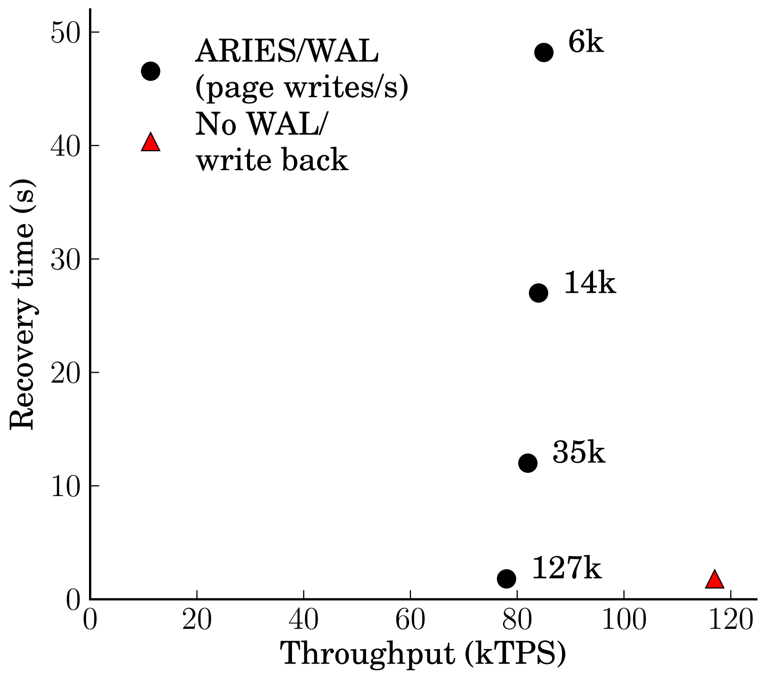


Figure 4.1: TPCB recovery latency vs throughput. Increasing page flush rate reduces recovery latency. Removing WAL entirely improves throughput by 50%.

NVRAM technologies exhibit their own limitations, such as greater-than-DRAM latency, particularly for writes [45].

These NVRAM technologies stand to revolutionize Online Transaction Processing (OLTP), where consistency and durability are paramount, but applications demand high throughput and low latency. Prior work has already demonstrated the potential of these technologies to enhance file systems [36, 23] and persistent data structures [94], but has not considered OLTP. Today, OLTP systems are designed from the ground up to circumvent disk's performance limitations. For example, many popular database systems use Write-Ahead Logging (WAL; e.g., ARIES [55]) 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 forward-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 [42] for hypothetical NVRAM devices (see Section 4.5 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 database throughput can greatly outpace existing storage devices (this configuration requires 6,000 page writes/s to bound recovery at maximum transaction throughput; 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 store 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 if we consider re-architecting database software. The Figure 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 are 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

	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).

disk and NVRAM, proposing software designs to address these problems. Additionally, I outline an NVRAM performance evaluation framework, involving memory trace analysis, code annotation, and precise timing models for OLTP running on existing hardware platforms. Subsequent chapters will build on the designs and methodology presented here to determine when OLTP must be redesigned and what problems might remain.

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.

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 milli-seconds) 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 solely on software to manage a DRAM buffer cache, as in conventional disk-backed database systems. The cache may be removed entirely or execution relies soly 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 only limited capacity. Third, one might *replicate to DRAM* all data that is stored in NVRAM—all writes update both DRAM and NVRAM (for recovery), but reads retrieve data exclusively from DRAM. Replicating data ensures fast reads by avoiding increased 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 generic mechanism called a *persist barrier*, which guarantees that writes before the barrier persist before any dependant operations after the barrier persist.

Persist barriers may be implemented in several ways. The easiest, but worst performing, is to delay threads that issue persist barriers until all pending NVRAM writes successfully persist. 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 [23]. 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). At recovery, a log entry is considered valid only if this 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 (once certain the entry is persistent) persisting the LSN. This order is enforced by inserting a persist barrier between writing the log entry and its LSN. Additionally, I reduce the number of persist barriers by persisting entries in batches, writing several log entries at once (without LSNs), followed by all their LSNs, separated by a single persist barrier. It is entirely possible (yet unlikely) that pre-existing LSN tails already match the

```
persist_wal(log_buffer, nvram_log)
  for entry in log_buffer:
    nvram_log.force_last_lsn_invalid(entry)
    nvram_log.insert_body(entry) # no lsn
  persist_barrier()
  nvram_log.update_lsns()
  persist_barrier()

  persist_page(page_v, page_nv, page_log)
    page_log.copy_from(page_nv)
    persist_barrier()
    page_log.mark_valid()
    persist_barrier()
    page_nv.copy_from(page_v)
    persist_barrier()
    page_log.mark_invalid()
    persist_barrier()
```

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.

log entry's offset; tails must be checked and first reset when this occurs. 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 with the use of a persistent undo page log. First, the page's original data is copied from the NVRAM page to the page log. The page log is persistently 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. Four persist barriers ensure that each update persists before the next, ensuring consistent, persistent state at all points in execution. Recovery checks the valid flags of all page logs, copying any valid log back in-place. The log is always valid while the page persists in-place, protecting against partial NVRAM writes. 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 disk accesses are implemented as copies between the volatile and nonvolatile address spaces. *NVRAM Disk-Replacement* in Shore-MT persists the log and pages with `persist_wal()` and `persist_page()`, respectively. However, persists occur on log and page flusher threads, and transaction threads do not observe persist barrier delays (except when waiting for commit log entries to persist). *NVRAM Disk-Replacement* provides low recovery latency by aggressively flushing pages, minimizing the size of data to recover. 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 updates to persist in-place and enforce persist order immediately, a design called *In-Place Updates*. *In-Place Updates* removes the centralized log by replacing 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 is constantly at its replayed state—there is no need to replay a redo log after failure. Undo logs need not maintain a global order (transactions are already free to roll back in any order), and instead I distribute ARIES undo logs per transaction. Such non-concurrent logs are simpler and impose less overhead than centralized logs. Other databases (such as Oracle) already distribute undo logs in rollback segments and undo table spaces and do not require additional distributed undo mechanisms [25]. Transaction undo logs remain durable so that in-flight transactions at the time of failure can be rolled back. Each page update consists of (1) latching the page, (2) inserting an undo entry into the transaction-private undo log, using `persist_wal()`, (3) updating the page in-place, using `persist_page()` (without an intermediate volatile page), and (4) releasing 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. Recovery applies undo logs for in-flight transactions; there is no need to replay a redo log.

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* (if persist barrier delays are negligible *In-Place Updates* resembles existing non-recoverable in-memory databases). However, one would expect transaction performance to suffer as persist barrier delay increases.

In response, I introduce *NVRAM Group Commit*, a recovery mechanisms 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 also fail to perform in others. *NVRAM Disk-Replacement* is insensitive to large persist barrier delays as it was originally designed for disk. However, it assumes IO delays to be 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, such that *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, allowing only transactions from the oldest batch to execute. 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. ARIES undo logs support concurrent durable transactions. As batches persist atomically, transactions no longer roll back selectively during recovery, 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 limits persist barrier frequency by enforcing persistence by batch rather than by transaction. Persisting a batch resembles `persist_page()`, used across the entire database, once per batch. Because undo logging is managed at the batch level, transactions' updates may not persist in-place to NVRAM until all transactions in the batch complete. Rather, transactions write to a volatile staging buffer, tracking dirtied cache lines in a concurrent bit field. The bit field facilities quickly finding all dirtied data at batch completion. 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 is data 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 to the NVRAM database, 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 present a problem by forcing all other transactions in the batch to defer committing until the long transaction completes. Limited staging buffers, not large enough to fit 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 any in-flight transactions as long transactions, reassigning them to the next batch. Transaction-local ARIES undo logs invalidate as the batch commits, requiring additional persistent data structures to allow transaction and batch logs to invalidate atomically.

NVRAM Group Commit requires fewer persist barriers than *In-Place Updates* yet avoids expensive logging found in *NVRAM Disk-Replacement*. A batch requires only four persist barriers, regardless of batch length. Expensive persist barrier delays can be amortized over additional transactions by increasing batch length, improving throughput. Batch length must be at least large enough to amortize time spent quiescing transactions between batches. However, increasing batch length defers commit for all transactions in the batch, increasing transaction latency.

4.3.2 Proposed Work

NVRAM Group Commit only improves throughput so long as the time between batches to quiesce transactions, locate dirtied data, and persist that data is substantially less than time during each batch where transactions execute. To provide a fair comparison between recovery mechanisms, I implement a functioning prototype of *NVRAM Group Commit* in Shore-MT. Whereas I expect NVRAM to have sufficient bandwidth to persist data quickly, orchestrating data copies, tracking dirty data with low overhead, and locating dirty data efficiently proved to be difficult software problems. The details of this implementation were omitted from the VLDB submission, but are included here, including work to be done.

Concurrent Dirty Set. As described previously, *NVRAM Group Commit* requires each batch to track dirty data. This is done with an efficient concurrent set, tracking the set of buffer pool cache lines dirtied by each batch. In order to maximize transaction throughput, this set must allow low overhead updates by concurrent threads during batch execution, and fast iteration during batch persist.

Conventional implementations of set data structures prove insufficient. A set may be structured as a balanced tree, each node being a range of the dirty set. While such an implementation allows efficient iteration over the set between batches it imposes overheads to transaction execution (as transactions repeatedly search the set for the region they are about to modify). Instead, the set might be designed as a bitmap of dirty cache lines, each bit corresponding to a cache line in the buffer pool. A 12GB buffer pool requires 192MP of bits to track dirty cache lines. This set is efficiently updated with atomic OR operations and cleared at the beginning of each batch. However, iterating over the batch to return dirty addresses is too slow (compared to 10ms batch periods).

Instead, I leverage the observation that dirty regions are rare and sparsely located throughput the buffer pool. Batches contain up to thousands of transactions, yet these transactions can only write a small portion of a 12GB buffer pool. It is therefore necessary to efficiently skip large regions of clean data when iterating through the dirty set. To achieve this, I present a *tiered bitfield set*. The tiered bitfield set contains two bitfields. The first is as described above, where each bit corresponds to a cache line in the buffer pool, referred to as the *primary bitfield*. In addition, there is a higher level bit field, each bit corresponding to a cache line of the primary bit field, called the *top level bitfield*. Thus, a marked bit in the top level bit field indicates that at least one bit within the corresponding cache line of the primary bit field is set, which in turn correspond to dirty cache lines in the buffer pool.

Updating this set requires atomic OR on the necessary bits in both bit fields of the set, a small cost. Iterating over the set involves iterating over the (much smaller) top level bit field, finding *segments* of the primary bit field known to contain at least one set bit. This iteration reduces both the number of instructions and cache/memory lines accessed, minimizing persist time between batches.

I propose to include additional work demonstrating the sparse nature of writes to the buffer pool, as well as timing results that show that fast dirty line tracking and persist is possible. Additionally, I believe this data structure will be effective for any batched persist application.

Concurrent 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. Luckily, unused threads exist – transaction threads that have quiesced between batches.

Instead of sitting idle, these threads participate in persisting each batch.

The buffer pool address space and corresponding portions of the dirty line set are partitioned into several segments, each placed in a task queue. The batch coordinator and transaction threads blocked by the persist process each participate by accepting tasks to and persisting buffer pool partitions (both log and then data in-place). Once all tasks complete the batch commits, allowing the next batch to begin.

I propose to include results showing that persist parallelization is necessary, but is an effective way to accelerate batching.

Once both of these optimizations are implemented the system is using all cores to persist, and profiling shows that *memcpy* operations, copies from the buffer pool to the persistent address space, are the primary bottleneck (memcpy is implemented using fast SSE instructions and cannot be optimized further). Persist time is minimized, and there is little room left for improvement.

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 eviction from NVRAM.

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 [2]. 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 [77]. They evaluate their allocation strategies using database traces and queuing models to demonstrate the effectiveness of NVRAM at accelerating persistent writes. Partial Memory Buffers is closest to the middle-top cell of the design space table (*In-Place Updates* with a software-managed DRAM buffer), although that design considers NVRAM as part of a hybrid buffer, not the primary persistent store. None 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*. I am not aware of any previous work that extends disk group commit to reduce the frequency of NVRAM persist barriers. A limited-capacity staging buffer (i.e., one insufficient for the entire data set) may limit batch size, as described above.

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 5.1 I consider the importance of NVRAM caching to transaction throughput. Then, in Section 5.2 I assume a DRAM-replicated data store to isolate read performance from persist performance in evaluating each recovery mechanisms's ability to maximize transaction throughput. The next section describes an evaluation methodology for OLTP on NVRAM.

4.5 Methodology

This section details the methodology for benchmarking transaction processing and modeling NVRAM performance. Experiments use the Shore-MT storage manager [42], including the high performance, scalable WAL implementation provided by Aether [43]. While Aether 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 4.2 shows the experimental system configuration.

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-

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

Table 4.2: Experimental system configuration.

code delay loops to model additional NVRAM latency and bandwidth constraints at 20ns 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 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 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 [52] 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 while considering the effects of page caching. 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 create 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 20ns (latency per loop iteration and RDTSCP) with an accuracy of 2ns.

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. This technique models reservations above 85ns accurately, which is sufficient for my 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 [87]. TPCB contains one transaction class and models a bank executing transactions across branches, tellers, customers, and accounts [86]. TATP includes seven transactions to model a Home Location Registry used by mobile carriers [59]. Table 4.3 shows the workload configuration. I choose a single updating transaction from each workload and size workloads to fit in a 12GB buffer cache. 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 an optimal number of threads per configuration (between 10 and 12).

4.5.1 Proposed Work

While I long ago performed validation of my timing model, I intend to include a quantatative evaluation in the final thesis. Results will include a demonstration that the RDTSCP loop allows precise delays in increments of 20ns. Additionally, I will demonstrate the

Workload	Scale factor	Approx. size	Transaction
TPCC	70	9GB	New order
TPCB	1000	11GB	TPCB
TATP	600	10GB	Update location

Table 4.3: Workloads and transactions. One transaction class from each of three workloads, sized to approximately 10GB.

bandwidth can be accurately modeled so long as bandwidth reservations exceed 85ns. The original validation was performed on a multi-socket system, using two processors. Recent experiments restrict execution to a single socket due to WAL performance concerns. As bandwidth modelling is limited by thread communication latency, I expect single socket execution will allow lower minimum reservations while still accurately modelling bandwidth. However, the multi-socket validation provides a conservative bound, supporting the correctness of my experiments.

4.6 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.

Chen *et al.* consider battery-backed DRAM as a reliable memory in the RIO project [18]. The file cache is treated as a reliable memory and is recovered after "warm" reboots (power is retained). Ng and Chen build on RIO to place a database buffer cache in a reliable memory [60]. However, the mechanisms they investigate are insufficient to provide against many types of failure or ensure proper recovery for truly nonvolatile memories.

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 [4]. Greenan and Miller use NVRAM to store file system meta-data, improving performance while maintaining consistency and durability [36]. Both works continue to assume that disk provides the bulk of persistent storage. 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) [23]. While I assume similar hardware, I additionally consider a broad 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 [22]. Volos *et al.* similarly provide durable memory transactions using Software Transactional Memory (STM) and physical redo logging per transaction [95]. 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 [19]. 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 [94]. 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 [29]. 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.

While different than byte addressable NVRAMs, Flash memory has become an important storage medium. Similar in theme to this work, numerous authors have considered designing databases specifically for Flash ([7], [78]). NVRAM, unlike Flash, allows more efficient in-place updates through byte-addressability, low persist latency, and atomic persists.

Prior work (e.g., H-Store [83]) 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.

4.7 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. Further, I provide a methodology for evaluating these systems based on memory trace analysis and read-hardware timing models. The next chapter uses this methodology to compare these system designs.

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 look at NVRAM read and write performance concerns separately. Additionally, I propose to include additional investigations into important aspects of the system such as bandwidth constraints and device lifetime (based on write endurance), outlined in Section 5.3.

5.1 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).

5.1.1 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, store, 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 5.1. Index accesses represent the great majority of cache line accesses, averaging

	TATP		TPCB		TPCC		Average	
	% lines	lines/ latch	% lines	lines/ latch	% lines	lines/ latch	% lines	lines/ latch
Store	10.57%	5.32	11.71%	6.05	15.47%	4.25	12.58%	5.20
Index	89.43%	11.27	82.41%	12.19	81.18%	11.17	84.34%	11.54
Other	0.00%	0.00	5.89%	7.16	3.36%	3.00	3.08%	3.39
Total		5.53		8.47		6.14		6.71

Table 5.1: 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.

84% 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 11.54 vs 5.20 per store and 3.39 per 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 5.1) 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 5.1 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. Fortunately, caching proves remarkably effective for all workloads. Each workload sees a spike of 10-20% improvement for a cache size of just 20 pages. As cache capacity further increases, each workload's throughput

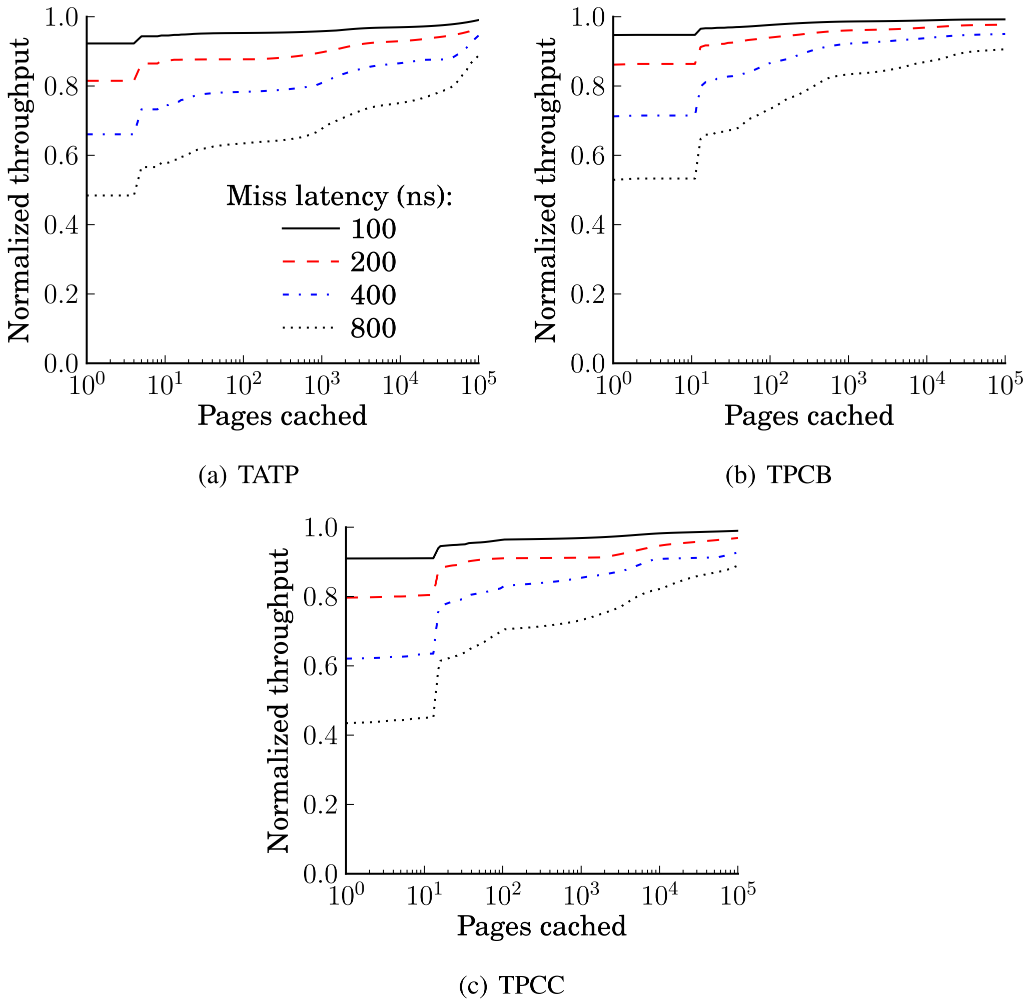
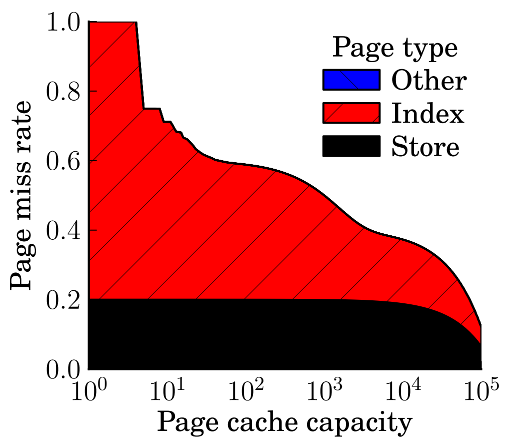
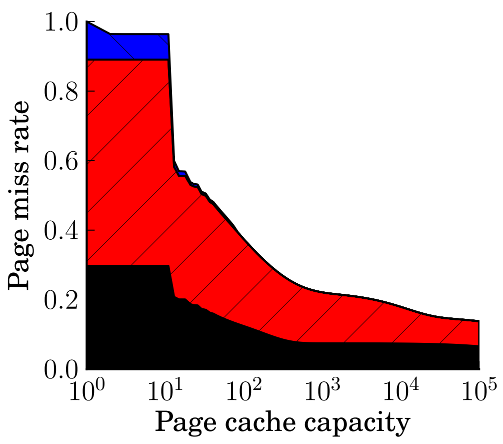


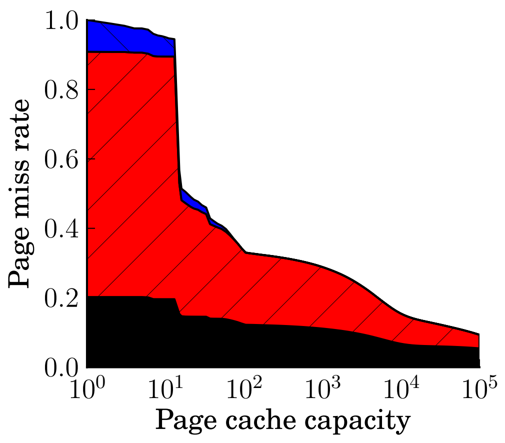
Figure 5.1: Throughput vs NVRAM read latency. 100ns miss latency suffers less than a 10% slowdown over DRAM. Higher miss latencies see large slowdowns, requiring caching. Fortunately, even small caches effectively accelerate reads.



(a) TATP



(b) TPCB



(c) TPCC

Figure 5.2: Page caching effectiveness. High level B+Tree pages and append-heavy store pages cache effectively. Other pages cache as capacity approaches table size.

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 [68].

5.1.2 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 5.2 shows the page miss rate per page type (index, store, or other) as page cache capacity increases. Each graph begins at 1.0 at the left – all page accesses miss for a single page cache. As cache capacity increases, workloads see their miss rates start

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

Table 5.2: Maximum required NVRAM read bandwidth. Workloads require approximately 1 GB/s NVRAM read bandwith, well below technology limits.

to decrease between cache capacity of five and 20 pages. TATP experiences a decrease in misses only in index pages, whereas TPCB and TPCC see a decrease across all page types.

While the behavior is specific to each workload, the results represent trends applicable to most 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.

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 5.2), I compute worst-case NVRAM read bandwidth for each workload, shown in Table 5.2 The considered workloads require at most 1.168 GB/s (TPCC). Since this is substantially lower than expected NVRAM bandwidth and caching reduces the required bandwidth further, I conclude that NVRAM read bandwidth for persistent data on OLTP is not a concern.

5.1.3 Summary

NVRAM presents a new storage technology for which modern database systems have not been optimized. Increased memory read latencies require new consideration for database caching systems. I show that persistently stored 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.

5.2 NVRAM Persist Synchronization

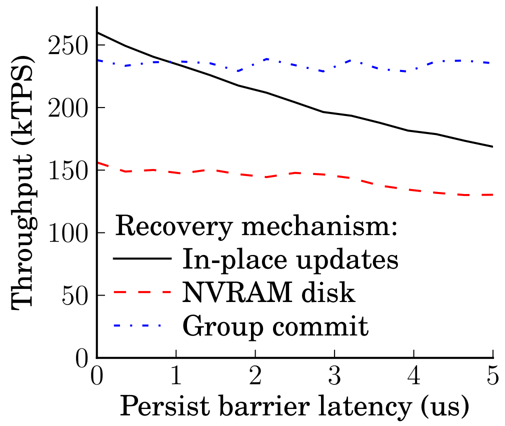
Whereas NVRAM reads benefit from caching, persists must always access the device. Of particular insterest is the cost of ordering persists via persist barriers. Several factors increase persist barrier latency, including ordering persists across distributed/NUMA memory architectures, long latency interconnects (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 4.5 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 (possible with PCIe-attached Flash). Ideally, NVRAM will provide fast enough accesses to use *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.

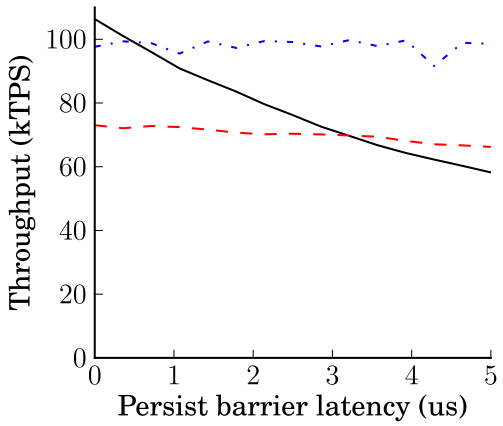
5.2.1 Persist Barrier Latency

Figure 5.3 shows transaction throughput 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.

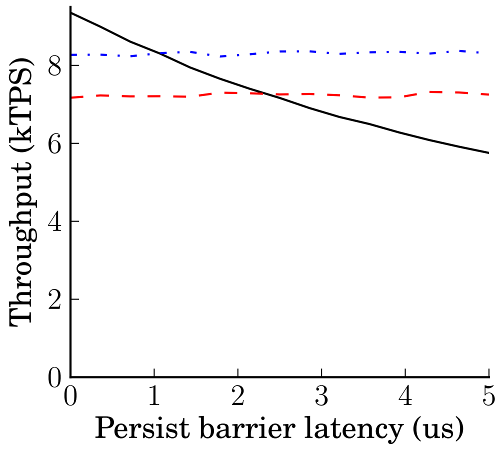
As persist barrier latency increases, each recovery mechanism reacts differently. *In-Place Updates*, as expected, loses throughput. *NVRAM Disk-Replacement* and *NVRAM Group Commit*, on the other hand, are both insensitive to persist barrier latency; their throughputs see only a small decrease as persist barrier latency increases. TATP sees the



(a) TATP



(b) TPCB



(c) TPCC

Figure 5.3: 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.

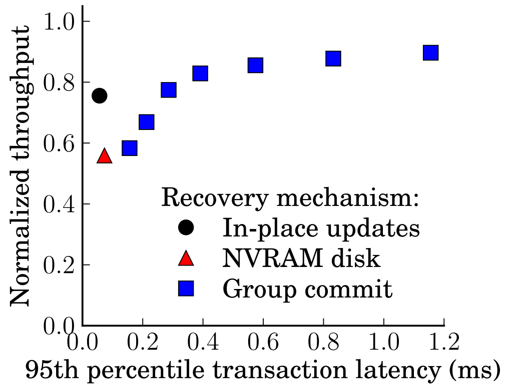
largest throughput decrease for *NVRAM Disk-Replacement* (14% from 0µs to 5µs). The decrease stems from *NVRAM Disk-Replacement*'s synchronous commits, requiring the log flusher thread to complete flushing before transactions commit. During this time, transaction threads sit idle. While both *NVRAM Disk-Replacement* 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.

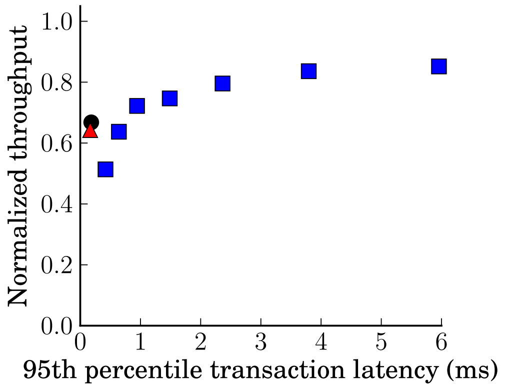
These results suggest different conclusions across storage architectures. NVRAM connected via the main memory bus will provide low latency persist barriers (less than 1µs) and prefer *In-Place Updates*. Other storage architectures, such as distributed storage, require greater delays to synchronize persists. 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.

5.2.2 Transaction Latency

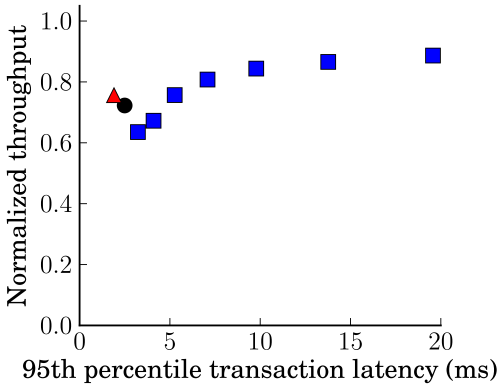
NVRAM Group Commit improves transaction throughput by placing transactions into batches and committing all transactions in a batch atomically. Doing so minimizes and limits the number of inserted persist barriers. However, deferring transaction commit increases transaction latency, especially for the 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



(a) TATP



(b) TPCB



(c) TPCC

Figure 5.4: 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.

latency.

Figure 5.4 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 expectations of modern applications. TPCC, the highest latency transaction, 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.

5.2.3 Summary

Persist barriers used to enforce persist order pose a new 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.

5.3 Proposed Work

I propose to include two additional studies. These studies were originally conducted for an earlier journal submission; the experimental system has changed substantially since then.

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). I intend to study the persist bandwidth requirements for each recovery mechanism.

Recovery mechanisms display interesting behaviors with respect to bandwidth. *NVRAM Disk-Replacement* persists inefficiently with respect to the total number of store bytes updated; even a small page update requires large log entries. However, the log persist makes effective use of cache lines (the granularity that data must transfer to the memory device). Further, when increased recovery latency is allowable, bandwidth usage may be decreased by deferring page flushing, allowing writes to coalesce in pages. The result is that *NVRAM Disk-Replacement* makes reasonable use of bandwidth, and bandwidth constraints are unlikely to limit throughput.

In-Place Updates displays similar bandwidth usage to *NVRAM Disk-Replacement*. All page updates must persist to NVRAM, and these updates tend to use cache lines ineffectively (only a small portion of each cache line changes). Also similarly to *NVRAM Disk-Replacement*, *In-Place Updates* must maintain ARIES logs (although per-transaction and undo-only), requiring large persists.

NVRAM Group Commit, while persisting the least amount of data, is the most sensitive to persist bandwidth constraints. The total quantity of data is reduced by using physical logs – logs copy data with little associated metadata. Additionally, updates in each batch coalesce. Only the first write to a given address produces undo, and only the final write to that address persists in-place. Other versions of data during the batch do not persist. The downside of *NVRAM Group Commit* is that persists occur in bursts between batches. No data persists while the batch executes. Between batches however, all logs and store data persists, placing huge requirements on bandwidth. I intend to include a quantitative study of this behavior.

Device lifetime. NVRAM technologies, much like Flash memory currently, will have limited write endurance; each cell may only be reliably written to 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 [67]. 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. I intend to study expected device lifetime for each recovery mechanism, both assuming that persists to an address always persist to the same cell, and assuming hardware that evenly distributes persists across cells.

When wear-levelling hardware is unavailable *NVRAM Disk-Replacement* manages to prolong lifetime the best of the three recovery mechanisms. Centralized logs evenly persist to the entire log address space. Writes to individual pages may be bounded by limiting the page flush rate, although at the cost of recovery latency. *In-Place Updates* provides the shortest device lifetime. Updates to hot pages and addresses persist each distinct value, quickly wearing out hot cells. *NVRAM Group Commit* limits writes to hot address, but is insufficient to provide lifetime guarantees for expected phase-change write endurance. Persists coalesce within each batch, bounding persists to a single write per cell per batch. However, batches are sufficiently short that hot addresses still experience a high write-rate, wearing out quickly.

The story changes when hardware spreads persists across NVRAM cells. With wear-levelling the primary concern is the total number of persists; all cells experience the same number of persists. While *NVRAM Group Commit* outperforms *NVRAM Disk-Replacement* and *In-Place Updates* in this regard, none of the recovery mechanisms pose a device lifetime concern. Simply put, an insufficient amount of data persists to worry about device lifetime in the presence of hardware wear-levelling. I will include a quantitative evaluation of device lifetime by measuring the rate that persists occur to individual addresses.

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 on-chip or other hardware caches prove sufficient to minimize NVRAM read stalls. I also considered database performance in the presence of persist barrier delays. 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 stalls due to persist synchronization while still removing centralized logging. While *NVRAM Group Commit* increases high-percentile transaction latency, latency is controllable and within modern application constraints.

CHAPTER VI

Persistent Memory Consistency

This chapter motivates and defines *Persistent Memory Consistency*. The ideas here constitute in-progress or future work and I welcome feedback. Refer to Section 2.4 for a discussion of existing memory consistency models. I first define the need for persistenct consistency and then give examples of persistent memory consistency models. In the next chapter I outline persistent programming patterns, why they require relaxed persistent consistency models, and possible optimizations.

6.1 Introduction

Future NVRAMs will provide a persistent store with the programming interface of modern main memories. While such technologies could revolutionize the design of recoverable systems and durable storage, questions remain regarding device performance and the NVRAM programming model. Chapters IV and V considered an abstract *persist barrier* capable of enforcing persist order or blocking until previous persists complete. Instead of considering an implementation or exact semantics of these barriers, I instead looked at the effect average persist barrier latency had on OLTP software design. This chapter delves into the details of persist barriers, considering their implementation, performance impact, and programming model.

Persist barriers exist as a tool for programmers to ensure correct persistence behavior, while at the same time improving performance. An alternative programming model requires that persists occur immediately – execution does not continue until data persists successfully to NVRAM. Such a model guarantees on recovery that the persistent state resembles a valid interleaving of program orders from the various threads. That is, we observe some persistent state of sequentially consistent execution. Whereas Sequential Consistency is a popular model for DRAM programming, long latency NVRAM persists may incur substantial stalls on each persist. Persist barriers remove these stalls by allowing persists between barriers to occur in parallel.

Persist order may additionally be enforced by flushing select cache lines or flushing the entire cache, either blocking until all data persists or receiving an acknowledgement/interrupt. This model most closely matches existing disk interfaces, where the programmer invokes system calls to write individual pages and then calls a sync system call. However, enforcing persistence via cache line flushing requires the programmer to reason about the cache architecture and data layout (e.g., considering if an object crosses cache line boundaries). Instead, we would like a more natural approach to reasoning about persistence that fits existing programming models.

Additional questions remain when memory is shared between threads or processes. Currently, memory consistency models define how threads communicate and what barriers are necessary to ensure expected behavior. Memory consistency models exist because processors (and compilers) prefer to run instructions out of order – a pervasive optimization that significantly complicates multi-threaded programming. While consistency models control the order in which threads observe reads and writes, there is no similar definition for the order in which persists occur, or how persist order is determined across communicating threads. Furthermore, persistent memories often care about the actual timing of persists, not just relative ordering – for example, system calls must make sure that all previous persists have completed before communicating to the outside world. In this case there is no alternative but to block until all data persists. These mechanisms are not present in existing consistency models.

While memory consistency models provide a starting point for persistent consistency models, the performance differences of volatile memory systems/DRAM and NVRAM requires new programming models for NVRAM. In fact, the consistency and persistence models may be de-coupled. That is, the rules that define load and store order might be differ between how *values* are communicated and how *persist order* is enforced.

I wish to extend memory consistency models with persistence semantics to address these concerns. My goals are to determine how multi-threaded consistency interacts with persistence, how to relax persistent consistency models to provide high performance and easily programmable interfaces, and identify programming patterns likely to cause NVRAM performance bottlenecks alongside potential optimizations. While a comprehensive investigation into persistent memory consistency involves considering numerous consistency models, several implementations, and broad range of data structures, I wish to restrict this dissertation to motivating a need for persistent consistency models. To that end, I propose several relaxed consistency models. I will implement simple, significant persistent data structures with each of these models, and plan on developing an evaluation framework for these data structures and consistency models. Importantly, I do not wish to consider hardware implementations of these models. While the performance of persistent systems will invariably depend on the implementation, I feel that a basic understanding of these consistency models is a substantial contribution in itself. My primary challenge is to devise an evaluation of these models independent of their implementations; I welcome any feedback.

The remainder of this chapter discusses the performance implications of persist dependencies before considering examples of persistent memory consistency models. In addition, I examine previous work, highlighting strengths and weaknesses, and placing it into my persistent memory consistency taxonomy.

6.2 Implication of Persist Dependencies

Here I describe how I expect NVRAM, the memory system, and the programming model to affect performance. Later sections will use these assumptions to reason about consistency model and data structure performance.

There are primarily two cases where persists may cause a thread to stall: ordering barriers and sync barriers. Ordering barriers may operate by stalling execution at the barrier until all previous persists complete. More complex memory systems will allow threads to continue executing ahead of persistent state, buffering persists (in the cache or elsewhere in the memory system), and later persisting in the proper order. While decoupling volatile execution state and persistent state removes immediate stalls, these buffers may fill, forcing threads to stall until room becomes available. Therefore, persist throughput is the primary concern – buffers allow threads to execute ahead of persistent state, but the average rate of persist must match the average rate of volatile execution, otherwise stalls necessarily occur.

Persist throughput is primarily limited by the number of persist dependencies in an application. NVRAM memories will allow high throughput so long as persists may occur in parallel. Persist dependencies reduce parallelism, requiring that some set of writes persist entirely before any dependent persists begin (necessary for proper recovery). Since NVRAM cells may take up to microseconds to persist, the longest dependence chain of persists limits persist throughput, additionally limiting execution throughput when buffers fill. Persistent memory consistency models stand to improve throughput and reduce stalls by relaxing persist ordering constraints, minimizing the persist critical path to only the persist order dependencies necessary for recovery correctness. The goal is to reduce persist critical path sufficiently that persist throughput exceeds execution throughput, providing the throughput of a DRAM system with the persistence of NVRAM.

Persistent memory applications also introduce stalls at sync barriers. These are barriers used when interacting with the outside world (e.g., displaying something on-screen, communicating over the network). Unless this external communication can be asynchronously ordered with persists, allowing the thread to continue doing other useful work, the thread must stall for previous persists to complete, reducing throughput. Additionally, end-to-end latency is important for many applications and tasks – greater sync time (while persists drain/flush) leads to increased task latency. Again, persist critical path determines sync latency; if independent writes persist in parallel the time to sync is determined by the longest chain of persists outstanding at the time sync is called.

Finally, a persistent memory consistency model may reduce performance if barriers intended to enforce persist order additionally enforce constraints on volatile execution (e.g., a persist barriers prevents memory instructions to volatile addresses from reordering). While I do not discuss this phenomenon further, I would like to investigate its effects in the future and would appreciate feedback.

The remainder of this chapter proposes persistent memory consistency models to relax persist constraints and improve NVRAM performance.

6.3 Persistent Memory Consistency Models

While memory consistency models restrict the order that loads and stores are observed across threads, they give no guarantees on persistence. I outline several persistent consistency models, starting with a constrained, strict model, and then considering more relaxed models. The purpose of these models is to provide correctness by defining the allowable persistent states during execution. Implementation may only allow these persistent states to be observed at recovery and are free to insert further restrictions, disallowing additional persistent states. Additionally, implementations are free to introduce any optimizations so long as programs are never able to detect a state outside the consistency model (e.g., speculation, in-hardware logging).

6.3.1 Persistent Sequential Consistency

The first model couples persistence to the sequential consistency model as Persistent Sequential Consistency (PSC). This model requires that the underlying volatile memory consistency model is sequential consistency. All loads and stores (including persists) appear to occur in a globally defined order as an interleaving of threads' valid program orders. Whereas volatile memory systems provide this solely by controlling the order in

```
Thread 1:          Thread 2:

A = 1              B = 1
C = B              D = A
```

Figure 6.1: Persistent Sequential Consistency (PSC). All variables in NVRAM and initialized to 0. PSC prevents states {A=0, D=1} and {B=0, C=1}.

which memory actions become visible from each processor, NVRAM used for recovery must treat every point in time as a possible failure, observing the persistent state. Achieving a globally consistent persist order requires that persists *actually* occur in-order from each thread, or that sequential batches of persists occur atomically, so that failure may not observe an intermediate state (NVRAM's atomically persistable size is likely to be small – 8 bytes). Additionally, all data sharing propagates ordering dependencies between persists.

Consider Figure 6.1. The letters correspond to persistent variables, and all are initialized to 0. Assuming threads execute under a sequential consistency model (i.e., they observe shared values from that model) and persist orders are enforced according to the PSC model, at no point may we observe that B=0 and C=1. Doing so would violate the model by allowing thread 1 to observe a persist from thread 2 (B=1) and then persist an additional value before B persists. Even if the program executes according to SC, persistence order must also be observed.

A strict implementation of PSC requires that all persists occur before each thread makes any further progress – volatile execution is coupled to persistent execution. This strict model does not require a sync barrier. A slight relaxation allows volatile state to advance ahead of persistent state, so long as (1) both the volatile and nonvolatile state are valid sequential executions, and (2) the persistent state matches some previous volatile state. This optimization does require a sync barrier; communicating with the outside world requires persistent state catch up with volatile state.

PSC requires all persists within a thread are ordered, and all shared memory accesses additionally order persists. This is also what makes PSC the most intuitive programming model. When persist latency is large (expected to be at least hundreds of nanoseconds up to several microseconds) every persist incurs this penalty; there is no opportunity to execute persists from the same thread in parallel. However, novel implementations might satisfy the PSC model and provide better performance (just as sequential consistency speculates, re-executing memory instructions when SC is violated). The next model relaxes the multi-threaded persist-order constraints.

6.3.2 Local Persist Order

This model allows thread communication and data persistence to occur at different granularities. Imagine, as an example, the *NVRAM Group Commit* design from Section 4.3. In this design I required a volatile staging buffer, separate from the primary NVRAM store, where batch updates occur. Only at the end of the batch would I copy the original batch data to an undo log and persist batch updates in-place to NVRAM. An alternative design omits this staging buffer by allowing updates during a batch to persist directly to the NVRAM address space. However, each update must first check if the address region being modified is protected by a segment of the undo log, and if it is not first create and persist the necessary undo. Once a region of memory is protected by undo, threads may update the NVRAM store in-place for the remainder of the batch without considering the order in which data persists. At the end of the batch the batch manager enforces that the last version of data persists before invalidating the log and allowing transactions to commit.

Consider this design under PSC – all persists within a thread occur in-order, and whenever data is shared a persist order is enforced. However, there is no need to enforce a persist order across threads except with respect to individual memory addresses. Data to different addresses may persist in any order so long as 1) associated undo log persists first and 2) the batch manager receives proper acknowledgement at the end of the batch that latest data versions have successfully persisted. Additional cross-thread dependences introduce unnecessary persist-order constraints that increase the critical path of persists, limiting persist throughput.

I relax cross-thread dependences in a model called *Local Persist Order* (LPO). All persists within a thread continue to execute in-order, as in sequential consistency. However, each thread may progress persistent state to various degrees, violating persistent sequential consistency; memory sharing does not enforce a persist-order dependence. While persists from threads to different addresses may reorder freely, persists to the same address must serialize. LPO relies on the underlying memory consistency model to provide this serial order (and therefore the memory consistency model/coherence implementation must provide a serial persist order on individual addresses). For example, again consider Figure 6.1. It is now possible to observe A=1, C=1, but B=0. This occurs if volatile execution observes thread 2 executing entirely before thread 1, but thread 1 persists entirely before thread 2 (which is allowed according to LPO).

The previous situation may be avoided by using explicit persist barriers, which are necessary to enforce persist order between threads. There are many types of barriers, with varying implementations. I highlight a few here. A thread may enforce that all local persists occur before persists from other threads that read the local values. Such *persist order-before* barriers are useful for communicating persistence to another thread. A thread may enforce that all remote persists occur before subsequent local persists in a *persist order-after*, useful for reading persists from remote threads. Both of these may be combined in a *persist total-order*, allowing a thread to completely order all persistent updates with other threads. These barriers are influenced by the *acquire* and *release* barriers of Release Consistency, where barriers enforce an order in only one direction. While these define persistent state, execution still relies on an underlying consistency model to determine what values are communicated and which threads interact.

The batching example might enforce batch persistence by having all transaction, once done persisting data, emit a *persist order-before* and atomically increment a counter/semaphore (initialized to 0). The batch manager polls this counter, and once it reaches the total number of threads (signifying that all threads are done persisting), it emits a *persist order-after* barrier – all persists from the batch manager must occur after transaction thread persists. The batch manager invalidates the log, committing the batch. LPO's guarantee that persists to individual addresses maintain proper order ensure that the last version of data from the batch persist. The use of persist barriers ensures that all store data persist before the batch commits, but during batch execution extraneous cross-thread persist dependences do not slow execution. By using a relaxed persistent memory consistency model we are able to design a more intuitive batching system (no longer requiring a volatile staging buffer) and improve persist performance. Additionally, data persist while the batch executes, minimizing inter-batch delays.

LPO potentially improves performance by minimizing the critical path of persist dependencies, shown in Figure 6.2. Data sharing and the resulting cross-thread persist dependencies increase the critical path of persists. The Figure displays the persist dependencies of two threads as well as shared dependencies. Strict models, such as PSC, order persists between threads, increasing persist critical path (red arrow). LPO, on the other hand, removes such dependencies, except where explicitly created by persist barriers. Since threads persist independently, adding threads provides additional persist throughput, similarly to Symmetric Multi Threading (SMT), which provides memory parallelism on existing systems. LPO, then, is an appropriate model where threads persist independently (no sharing occurs) or where memory sharing should not enforce persist order (such as batching). However, the need to perform all persists in series within a thread remains a concern. The Byte Addressable File System (BPFS) addresses intra-thread persist dependencies by introducing additional persist barriers.

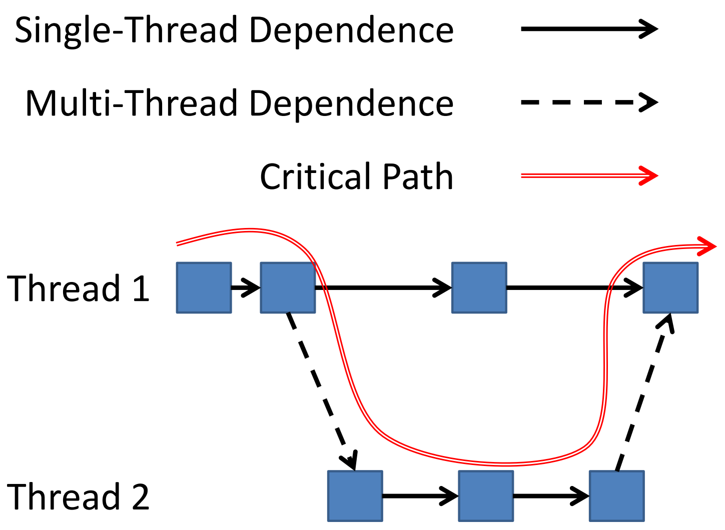


Figure 6.2: Local Persist Order. LPO removes cross-thread persist dependencies, reducing persist critical path (PSC critical path traced in red). Enforcing cross-thread dependencies requires explicit persist barriers.

6.3.3 Byte Addressable Persistent File System

The Byte Addressable Persistent File System (BPFS) [23] is the de-facto standard for existing persistent memory consistency models, used additionally by [22, 29, 94]. Threads persist by writing to the persistent address space, enforcing persist order via *epoch barriers* which divide persists into *epochs*. Persists may not reorder across epoch barriers, but persists within an epoch are free to execute in parallel; BPFS provides a relaxed persistent consistency model (persists may occur out of program order) and uses barriers to enforce constraints. BPFS additionally provides a hardware design, enforcing persist order with only cache modifications (little additional buffering or modifications to the memory system). A table of epochs is kept in the cache of each CPU, with each cache line keeping track of its persist epoch number and process/thread ID (even cache lines with only volatile writes track persist epoch). Persists buffered in the cache write back in the background, making sure that each epoch persists completely before starting to flush the next epoch.

Shared memory accesses enforce an ordering constraint between the epoch that produced the write and the epoch reading/overwriting that data. This persist order is enforced strictly by stalling. If at any point a thread overwrites a cache line from a previous persist epoch (even from the same thread), it stalls until the previous persist epoch completes persisting. Further, if a thread reads a cache line from a persist epoch produced by a remote thread that has not yet persisted it also stalls until that epoch completes persisting (it does

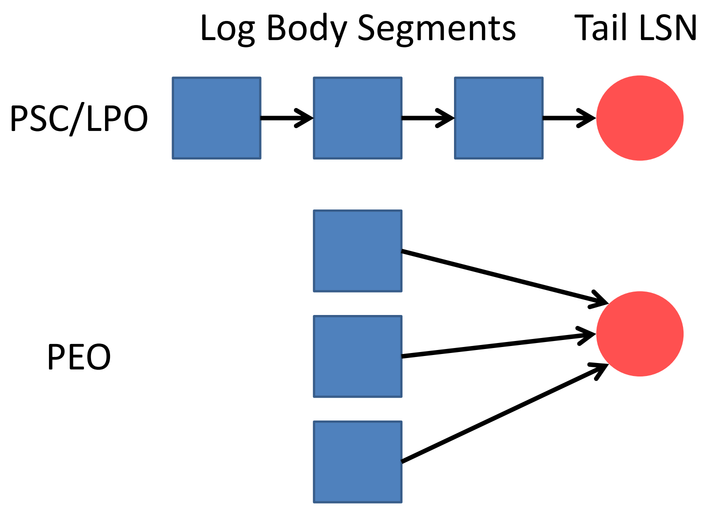


Figure 6.3: Partial Epoch Order. PEO allows regions of NVRAM to persist in parallel while enforcing recovery correctness. `persist_wal` persists entire log entry bodies in parallel using epochs.

not stall if the reading thread is the same as the producing thread). Persists within epochs write to the device in parallel, volatile execution proceeds ahead of persistent state, and cross-thread persist order is correctly enforced (at the cost of stalls).

Consider the ARIES log from Chapter IV, demonstrated in `persist_wal` (Figure 4.2) with persist dependencies shown in Figure 6.3. Persisting a single entry requires first persisting the entry's body, followed by the tail LSN. Most log entries are large (at least 100 bytes), containing information about the transaction, store, page, tuple, and action taken. According to PSC and LPO all persists (8 byte segments) to the log entry body must occur in program order – an unnecessary constraint that increases persist critical path. Instead, BPFS allows the entire log entry body to persist in parallel, and guarantees that the body persists before the LSN using a persist barrier.

In consistency terms BPFS produces a total ordering between persist epochs that communicate through memory (Total Epoch Order – TEO – would be an appropriate name for this model). This ordering resembles transactional memory – producing a total order on memory transactions – except BPFS does not provide concurrency control (conflicts are not detected and transactions can not roll back). However, attempting to provide a total epoch order produces problems; races (data or synchronization) within BPFS epochs may result in deadlock. The authors specifically prohibit races within their implementation, possibly recognizing that they introduce deadlocks. However, preventing synchronization

```
Thread 1:                Thread 2:

persist_bar              persist_bar
A = 1                    B = 2
B = 1                    A = 2
persist_bar              persist_bar
```

Figure 6.4: BPFS deadlock. BPFS attempts to produce a total order on epochs. Races within epochs create a cyclic order-dependence, resulting in deadlock.

races restricts useful programming patterns. I will demonstrate that the implementation as described allows deadlock and later provide example applications that make use of races.

Consider Figure 6.4. Each thread runs a single epoch with two persists to shared memory. The second thread executes these persists in opposite order of the first. If thread 1 writes to A at the same time thread 2 writes to B a deadlock occurs. Thread 1 attempts to write to B, fetching the B's cache line, and observes that thread 2 has not yet persisted, ordering thread 1's epoch after thread 2's. Similarly, thread 2 attempts to write to A, ordering its epoch after thread 1's; a cycle has occurred, preventing forward progress.

This problem may occur even without data races or when persists have a total order. Imagine that A and B on each thread are different variables, each protected by locks, but that thread 1's A (B) is on the same *cache line* as thread 2's A (B). Such *false sharing* results in performance problems for existing volatile cache systems, but in the case of NVRAM and BPFS produces deadlock. Furthermore, one might try to solve this problem by reordering thread 2's writes so that both thread 1 and thread 2 write in the same order, totally ordering the epochs (whoever accesses A first runs to completion while the other stalls). However, relaxed consistency models allow the processor (or compiler) to reorder writes and persists within an epoch, arriving at the original problem. In fact, it is not clear if instructions accessing only volatile data are permitted (or should be permitted) to reorder across epoch boundaries. Next, A and B may represent synchronization races; consider atomic read-modify-write (such as compare-and-swap) to persistent address spaces. BPFS, in assuming no races to persistent data, restricts the use of persistent locks. Finally, certain data structures may specifically intend to allow races within persist epochs. The *NVRAM Group Commit* implementation described above, for example, requires threads to work concurrently towards a persistent state.

BPFS attempts to produce a total order of all epochs, producing deadlocks and preventing useful programming patterns. I believe that the solution is to better define the interaction between persistence and the consistency model. I next introduce a model strongly

```
Thread 1:                        Thread 2:

persist_bar (i)                  persist_bar (i)
persist A                        persist D

persist_bar (ii)                 persist_bar (ii)
persist B                        read X
write X                          persist E

persist_bar (iii)                persist_bar (iii)
persist C                        persist F
```

Figure 6.5: Partial Epoch Order (PEO). PEO considers epochs in different threads containing a memory dependence to be concurrent. This memory dependence does not enforce a persist order between the threads' epochs, but enforces this order on earlier and later epochs. E must persist after A.

influenced by BPFS, that allows concurrent epochs to make forward progress at the cost of a slightly less intuitive interface.

6.3.4 Partial Epoch Order

The third and final model relaxes persist order within a thread similarly to BPFS, but produces only a partial order of epochs between threads. Partial Epoch Order (PEO) considers the epochs of two threads that communicate through memory to be *concurrent*, and thus not ordered. However, later epochs *do* carry the dependence, allowing ordering of persists between threads while avoiding deadlock and allowing inter-thread persist concurrency. When races do not occur within epochs PEO provides the same execution as BPFS. As in LPO persists from concurrent epochs to the same address must have a total order, provided by the underlying consistency model.

Figure 6.5 provides an example. Two threads persist data and communicate through volatile object X, each persisting three epochs. The threads communicate when thread 1, epoch (ii) writes to X, which thread 2 reads in its epoch (ii). According to PEO, epochs (ii) of each thread are concurrent and do not enforce a persist order. Thus, thread 1's persist to B may not necessarily occur before thread 2's persist to E. However, epochs of thread 1 prior to (ii) must necessarily persist before epochs of thread 2 after (ii) – When thread 2 reads a value from thread 1 it orders all later persist epochs after thread 1's earlier epochs. Thread 1's persist to A must occur before thread 2's persist to F – epochs containing shared memory accesses are concurrent, but later epochs carry the persist dependence. The converse is not true – thread 2's persist to D need not persist before thread 1's persist to C. This model produces a partial ordering of persist epochs that all threads observe.

PEO removes the possibility of deadlock by relaxing persist order constraints. However, PEO supports the same programming model as BPFS (a total epoch order). Any program who's epochs contain only volatile races, or only persists (never both) will produce a total order on epochs containing shared memory persists, resulting in the same execution as BPFS. Additionally, PEO's relaxed constraints allow advanced programming models such as batching. Neglecting undo log (which must persist before store data) transactions in a batch may all execute within the same persist epoch. Since these epochs are concurrent no persist order is enforced between persists of different threads. When the batch ends transactions end their epoch and increment the semaphore/counter, as in Section 6.3.2. The batch manager polls the counter until it matches the number of transaction threads, ends its own epoch, and invalidates the log. In practice, transaction threads must use multiple epochs to order undo log persists before data persists; I would like to evaluate whether PEO provides sufficiently relaxed persist constraints or if other models (such as LPO) are required for batching performance.

It is unclear to me if volatile memory and non-memory instructions should be allowed to reorder with persist epoch barriers. This will depend on the actual implementation, and if the hardware is able to easily distinguish persists from nonvolatile writes. For example, thread 2's read to C in the previous example should be allowed to occur before the persist to B so long as the correct epoch order is enforced. As persist barriers might be frequent, not allowing non-persistent instructions to reorder would limit the performance of *volatile* memory instructions.

PEO provides effective single-thread persist parallelism while providing a relatively intuitive model for cross-thread persist-order dependencies. PEO prevents deadlock and allows concurrent access to shared persistent memory spaces without enforcing a persist order.

6.4 Conclusion

This chapter motivated the need for more precise persistent consistency models to allow greater persist performance and enforce correctness while providing an easy to program interface. I proposed three new persistent consistency models and relate them to existing work. While this chapter has focused on persistence semantics and the programming interface, the next chapter demonstrates common persistent data structures, focusing on potential performance concerns.

CHAPTER VII

Persistent Programming Patterns

The previous chapter looked at persistent memory programming from the perspective of consistency models and programming interfaces. While necessary to define correct program behavior, I did not consider actual implementations or investigate specific optimizations. This chapter investigates performance more directly, considering two simple persistent programming patterns – a persistent log/buffer and a persistent linked list. While I will describe how I expect the persistent memory consistency models from the previous chapters operate with each pattern, the point is to highlight potential performance bottlenecks and necessary optimizations. I do not yet consider actual programming models or hardware to provide many of these optimizations.

Prior work has examined building data structures for NVRAM. However, these works generally address other issues such as write endurance [19], suggest data structures that intuitively fit NVRAM's programming interface without concern for performance [94], or propose complicated software mechanisms to work around incomplete or inferior persistent consistency models [29]. While BPFS [23] takes a more holistic approach, their model falls short in terms of correctness, multi-threaded performance, and possibly single-thread performance, as I will show shortly. I take a different approach, looking at useful data structures and potential performance concerns and using this insight to imagine new persistent consistency models and optimizations.

7.1 Persistent Buffer

I first discuss a persistent buffer, used as an OLTP database centralized log in [29]. The version presented here is a simplification of the implementation originally proposed, but illustrates the necessary performance concerns. This log buffer contains a persistent data array as well as a persistent counter, marking the end of the valid, persistent region of the

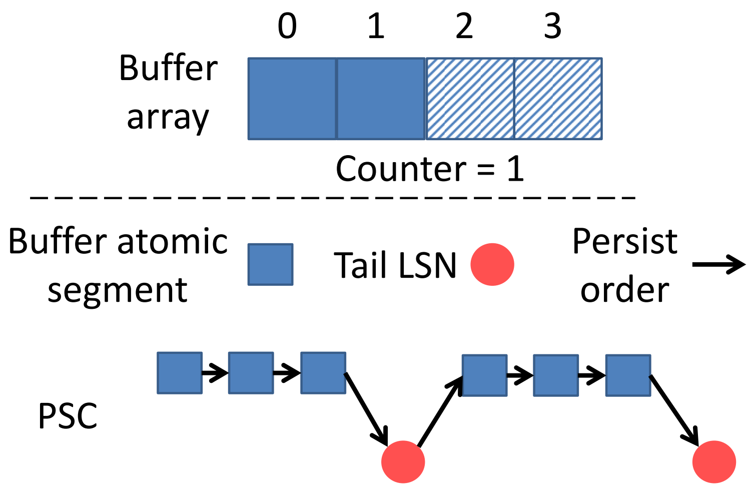


Figure 7.1: Persistent buffer. The persistent buffer constists of a persistent array and counter (top). The persistent counter marks the greatest persistent byte. Persisting to the buffer under PSC places creates between writing buffer data and updating the counter, as well between persisting the counter and the next buffer persist, forming a chain (bottom).

buffer, shown at the top of Figure 7.1. In the example buffer slots 0 and 1 are persistent and valid, while slots 2 and 3 are not (0 and 1 will be recovered on failure). For simplicity, I neglect details that require data to wrap around the end of the buffer and instead pretend that the valid region of the buffer always grows from the base offset of the buffer through the address marked by the counter. Data are inserted into this buffer by acquiring a volatile lock on the structure, reading the counter to determine the next available address, persisting data to that address, and incrementing the counter by the size of the inserted data before releasing the lock. For correctness, buffer data must persist before the new counter value persists, and all counter values must persist in order. These dependencies guarantee that, on recovery, the valid portion of the buffer includes the base address through the address located in the persistent counter.

[29] assumes the BPFS consistency model, noting that updates to the counter field will cause each thread to stall while the previous value persists. Their work provides several complicated optimizations to 1) allow log entries from different threads to persist in parallel (by explicitly removing or re-ordering cross-thread dependencies) and (2) only persist intermediate values of the counter, removing a chain of persist dependencies that would otherwise limit throughput. Such design improves performance but requires additional effort and is unintuitive. Other persistent consistency models might enable these optimiza-

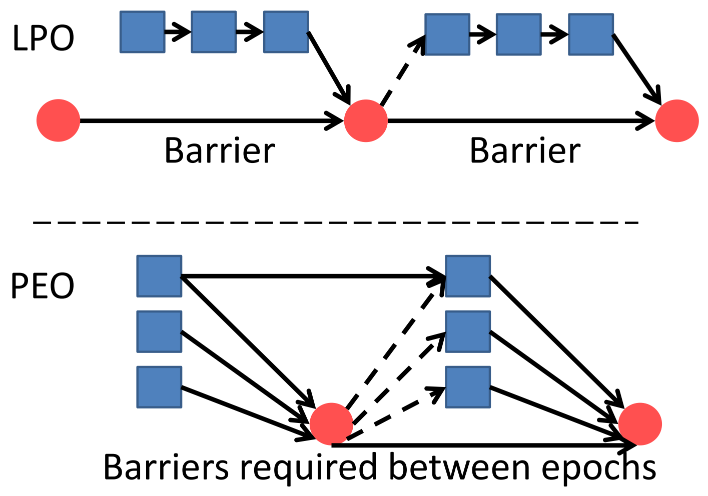


Figure 7.2: Buffer execution under LPO and PEO. LPO (top) removes persist dependencies between the counter value and subsequent buffer persists on other threads. Dependencies remain when adjacent buffer inserts are from the same thread (dashed line). PEO (bottom) allows buffer data to persist in parallel.

tions without the use of error-prone software implementations. I qualitatively evaluation my three consistency models with this buffer, reasoning about persist critical path length per buffer insert.

PSC. Under PSC (recall, persistent sequential consistency) all persists to the buffer and counter form a dependence chain, shown at the bottom of Figure 7.1. Every atomically persistable segment (8 bytes) of buffer data persists in series, followed by the new value of the counter. The next thread to insert into the buffer reads the counter, a shared variable, ordering all subsequent persists after persists of the previous insert. Assuming log entries are 100 bytes on average, each insert requires 14 serial persists (13 to persist the log entry data, 1 to persist the counter). If NVRAM persists take even 100ns (a conservative estimate), entries can only be inserted every 1.4µs, on average (any faster and buffers will eventually fill). This rate is significantly slower than the throughput of modern concurrent buffers, where cache invalidations take anywhere from tens to hundreds of nanoseconds (multi-core to multi-socket, respectively). Without relaxed persistent memory consistency models, NVRAM persists will necessarily create a throughput bottleneck.

LPO. Next I consider the relaxed persistent consistency models introduced in Section 6.3, starting with LPO. Recall that LPO (local persist order) enforces program-order persist order within threads, but not cross-thread dependence order (barriers necessary).

Figure 7.2 shows execution of LPO at the top. Buffer persists continue to serialize and must persist before the counter, but there is no longer a dependence between the counter persisting and later buffer persists from other threads. Counter values must still persist in-order, which will be enforced with a persist barrier or as a result of persists to the same memory address. Dependencies *do* exist between counter persists and subsequent buffer persists on the same thread, shown as a dashed line in the Figure. If every log entry is inserted by a separate thread, LPO results in a persist critical path that involves only counter persists. However, log entries within a single thread must persist in-order, limiting the throughput of each individual thread. LPO is easy to program, as all persists within a thread occur-in order, and provides high throughput given sufficient buffering and a large number of threads to yield persist parallelism.

PEO. The bottom of Figure 7.2 displays buffer execution for PEO – partial epoch order. Partial epoch order allows single-thread persists to occur in parallel by placing them in epochs. Epochs across threads are ordered (or considered concurrent) according to the consistency model and memory sharing. PEO allows buffer data to persist in parallel, and removes ordering constraints between buffer persists and the previous counter persist from another thread (epochs are concurrent – see Figure 7.3). However, buffer persists are ordered after buffer persists of the previous insert (top solid arrow in the Figure), and ordered after counter persists from the same thread (dashed arrows in the Figure). Even a glance at the Figure reveals how complicated cross-thread dependencies and concurrent persist epochs become. While PEO provides a correct model by removing deadlocks from BPFS, and enables optimizations by allowing concurrent persist epochs, this is not a reasonable programming interface.

Assuming subsequent inserts occur from different threads, the persist critical path forms a chain of only one persist epoch per insert (either through the counters or through the buffer persists). Using our earlier assumptions, this model allows inserts every 100ns on average. Such delays are similar to invalidation delays in modern multi-socket CPU systems, but NVRAM may require more than 100ns per serialized persist. Furthermore, PEO requires frequent persist barriers, placing a burden on the programmer. While these epoch barriers are deemed intuitive, they are not as easy to use as sequential consistency (no barriers required).

Reasoning about concurrent epochs is quite difficult. I provide a second implementation of buffer insert under PEO (Figure 7.3, bottom) that allows a total order of persist epochs (TEO – as intended by BPFS). Epochs in this implementation have either volatile races (lock/release) or persists, not both. Following this rule allows a consistent total order

```
PEO:

 persist_bar (i)
 lock_acquire()
 dest = buffer[counter] # reads counter
 memcpy(buffer, entry) # persists buffer

 persist_bar (ii)
 counter += entry.size # persists counter
 lock_release()


TEO:

 persist_bar (i)
 lock_acquire()

 persist_bar (ii)
 dest = buffer[counter] # reads counter
 memcpy(buffer, entry) # persists buffer

 persist_bar (iii)
 counter += entry.size # persists counter

 persist_bar (iv)
 lock_release()
```

Figure 7.3: Buffer insert with PEO/TEO. Pseudocode implements buffer insert with two persist barriers. PEO's concurrent epochs allow epoch (ii) to persist in parallel with epoch (i) of the next insert operation. TEO simplifies persist-order reasoning, separating volatile races from persists. PEO requires four barriers and introduces persist dependencies between persisting buffer data and the previous counter value.

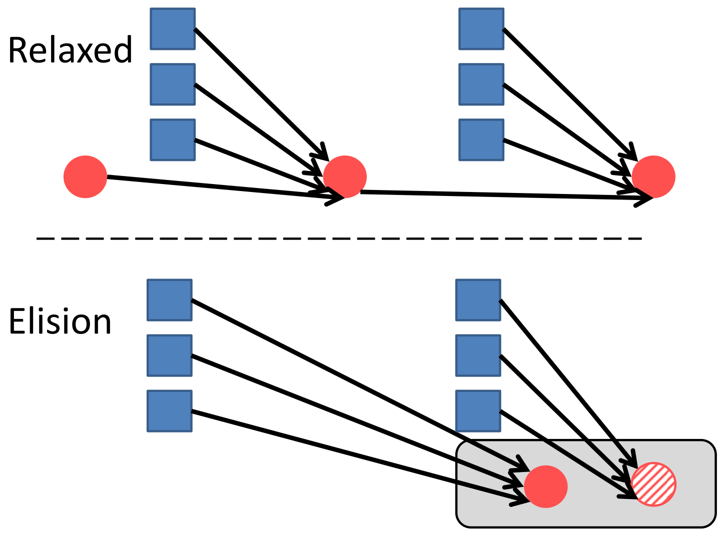


Figure 7.4: Buffer execution under relaxed persistent consistency. Relaxing persistent consistency removes unnecessary cross-thread dependencies (as in LPO), allows epochs to persist in parallel (as in PEO), and removes unnecessary serial epoch dependencies within threads (top). Epochs/dependencies are elided by persisting only select counter values (bottom).

of persist epochs (those epochs that actually persist). This implementation, while more obviously correct, increases persist critical path by introducing a dependence between counter persist and the next buffer data persist (the dashed arrow of Figure 7.2 implies a dependence even for inserts from different threads).

Relaxed persistent consistency. Finally, I imagine a relaxed consistency model for this buffer, shown in Figure 7.4 (top). I will not describe the semantics of this model, but rather the desired persist dependencies. Like PEO buffer data persists in parallel. Like LPO there are no dependencies between counter persists and subsequent buffer persists. This is true even for single-threaded use. Doing so requires more complicated barriers that do not impose serial dependencies between epochs, but instead allow precise, possibly named dependencies. For example, a barrier may declare that an epoch has no earlier persist dependencies, but still enforce that all later epochs persist after it. Such a barrier would allow a programmer to persist buffer data and guarantee that buffer persists never serialize after counter persists.

I introduce an additional optimization to the relaxed consistency model, leveraging the insight from [29] that not all values must persist, shown at the bottom of Figure 7.4. Persistent memory consistency models describe the set of *allowable* persistent states, yet not every allowable state must actually persist. Since the counter is atomically persistable (8 bytes), the memory system may defer persisting new values. Once the buffer data for two (or more) adjacent log entries successfully persist, the counter value for the last entry persists (shaded circle in the Figure), implicitly persisting all intermediate counter values at once, eliding persist epochs entirely. The key is that the counter variable is atomically persistable and is the only persist within its epoch. We might imagine a new epoch barrier that allows only a single persist of atomic persist size. This persist defers, hoping to coalesce with similar persist epochs. A new value eventually persists, but may skip intermediate values so long as all the coalesced persist epochs' dependencies are satisfied. The result is that the persist critical path is now bounded to a constant – persists should never limit the throughput of this buffer (assuming large but finite buffers).

More generally, persist dependencies form a schedule or DAG, with persist epochs forming nodes in the graph and epoch dependencies forming edges. Nodes in the DAG may be combined into an *elision super node* (combining all the nodes' persists) so long as 1) all persists in the super node fit in an atomically persistable size, 2) combining nodes retains the DAG (no cycles introduced), and 3) nodes inherit dependences – the super node inherits all dependencies of its member nodes, all nodes depending on member nodes now depend on the super node. An interesting prospect is that as NVRAM technologies progress, the atomically persistable size may increase (say, from 8 bytes to 64 bytes). Larger atomic persists allow additional persist epochs to coalesce into super nodes, decreasing persist critical path, possibly without user intervention. It remains unclear if this property should be leveraged by hardware or in the consistency model, if at all.

7.2 Persistent Linked List

Finally, I consider a persistent linked list, shown in Figure 7.5. The linked list contains a persistent Head reference, persistent nodes (each containing a *next* reference), and a volatile Tail reference to quickly find the end of the list for insertion (although I assume that at recovery the list will only be traversed from the Head). The top of the figure shows the list, with arrows representing *nexst* references. Unlike the persistent buffer, where log entries persist in parallel and the counter is overwritten, the linked list does not overwrite any values as new nodes are added. For correctness, node data must persist before the node *next* reference pointing to that data.

Strict persistent consistency models will enforce that nodes persist in the order they are inserted. Nodes persist in series, each waiting for the previous node to completely persist, with references still persisting after the data they point to (necessary dependencies shown as solid arrows, unnecessary dependences as dashed arrows at the bottom of the Figure).

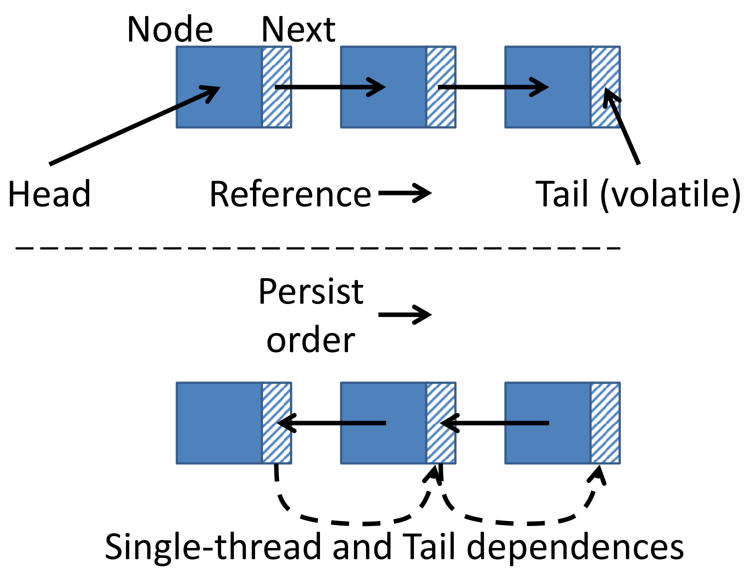


Figure 7.5: Persistent linked list. The linked list constains nonvolatile Head pointer, volatile Tail pointer, and nodes (data – solid square; *next* reference – patterned box). Nodes persist in parallel (data must still persist before references). Dashed lines show unnecessary dependencies enforced by strict persistent consistency models, forming a dependence chain. The valid list is determined at recovery by traversing from Head to the last valid node reference. Implementing efficient sync remains a challenging.

However, such operation forms a persist dependence chain, limiting persist throughput. Instead, we would like to allow persist nodes to persist in parallel, yet continue to enforce node data-before-reference persist orders. The valid portion of the list is determined at recovery by traversing the list from the Head, truncating the list at the first null or invalid reference.

I do not describe how this list likely performs for my persistent consistency models, but simply note the desired behavior. High performance consistency models will limit persist dependence chains, avoiding the need to persist in series. However, there is now no efficient way to sync the list. While we do not constrain the order in which data persists we still need some mechanisms to *observe* when a specific portion of data has persisted – that is, we must be able to tell when a node and all preceding nodes have persisted. The programmer may determine that the entire list is persistent (or block until it persists) by traversing the list, adding substantial complexity and overhead. Thus, this persistent linked list is an interesting design for applications that require high throughput, rarely sync, and require linked lists (to insert or remove nodes from the middle). Faster, more efficient, yet intuitive sync design remains an interesting challenge.

7.3 Conclusion and Future Work

This chapter considered persistent memory consistency from a performance perspective. I investigated two simple programming patterns, considering how my proposed persistent consistency models might perform and imagining additional optimizations, separate from the programming model, necessary to increase persistent data structure throughput.

For the future, I would like to develop these ideas and design some sort of methodology to support my claim that persistent memory consistency models are a necessary tool and require further investigation. To start, I would like to implement the above data structures, annotating the code with persist dependencies and using a timing model similar to the one outlined in Section 4.5. Additionally, I would consider implementing these models in Shore-MT. Finally, I will consider additional models if the described models do not provide sufficient performance, or I discover that other models provide sufficient performance and are easier to program for. I am especially interested in determining if models stricter than BPFS (fewer barriers) provide sufficient performance. I plan to submit a paper based on these ideas to this upcoming ISCA, November 2013, and defend next year (2014).

Future work (hopefully outside the scope of my thesis) might consider an evaluation framework that does not require code annotation – programs are written truly assuming consistency models and executed via simulation. Additionally, future work will consider actual hardware implementations of useful persistent memory consistency models.

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 [92], with a world-wide carbon-emissions impact similar to that of the entire Czech Republic [53]. 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 [48, 54], scheduling [56, 61], power delivery systems [28], cooling infrastructure [62], 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 [92, 54]; 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.

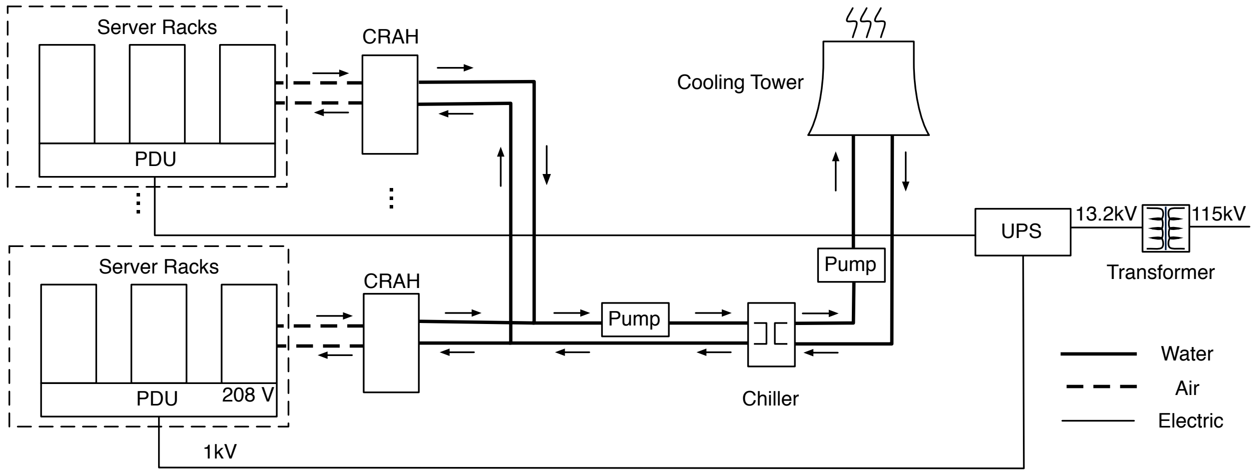


Figure A.1: Power and Cooling Flow.

A.2 Related Work

The literature on computer system power management is vast; a recent survey appears in [44]. A tutorial on data center infrastructure and related research issues is available in [51].

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 [28, 74]. Losses in power conditioning systems and the impact of power overprovisioning are described in [28, 72]. Our prior work proposes mechanisms to eliminate idle power waste in servers [54]. Other studies have focused on understanding the thermal implications of data center design [40, 62].

Power- and cooling-aware scheduling and load balancing can mitigate data center cooling costs [56, 61]. 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.

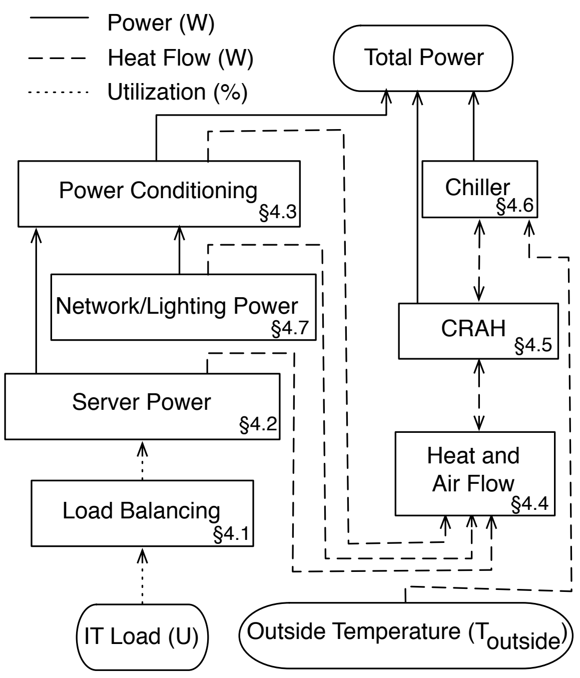


Figure A.2: Data Flow.

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:

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

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 [57].

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 [51, 28, 54, 74]. 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 [74], 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 [72]:

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

where $P_{\mathrm{PDU}_{\mathrm{Loss}}}$ represents power consumed by the PDU, π_{PDU} represents the PDU power loss coefficient, and $P_{\mathrm{PDU}_{\mathrm{Idle}}}$ 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 [72]:

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

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

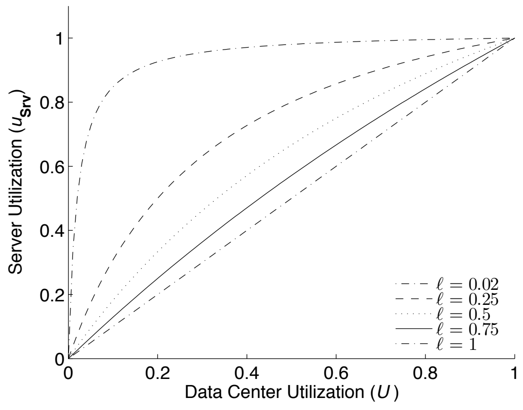


Figure A.3: Individual Server Utilization.

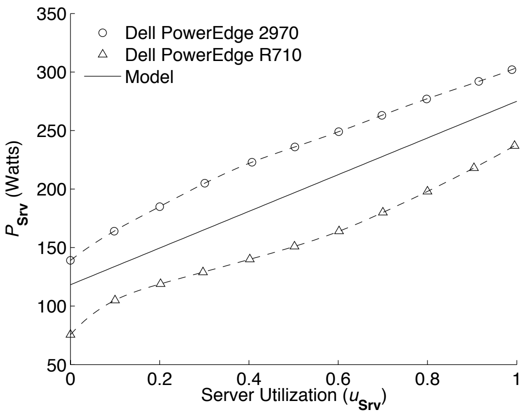


Figure A.4: Individual Server Power.

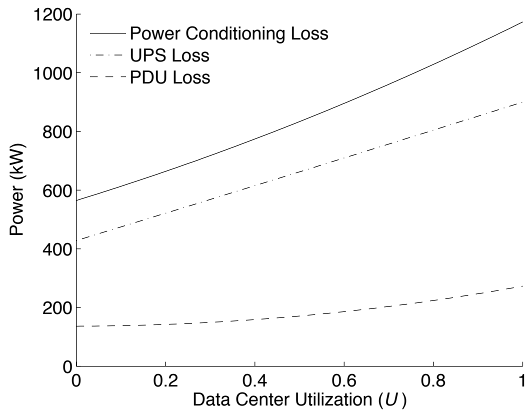


Figure A.5: Power Conditioning Losses.

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 [85, 93]. 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:

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

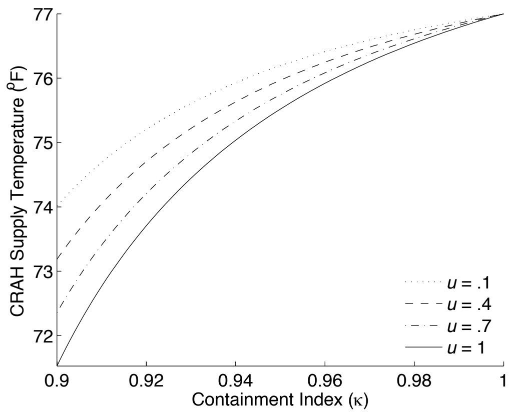


Figure A.6: CRAH Supply Temperature.

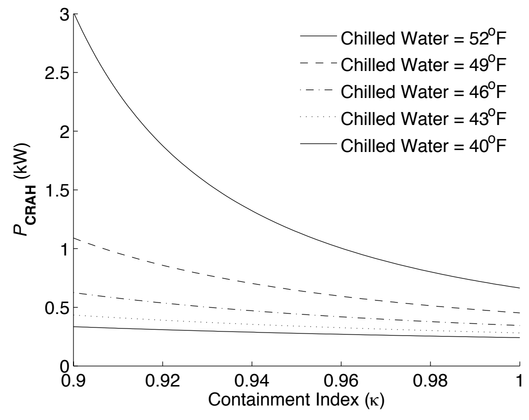


Figure A.7: CRAH Power.

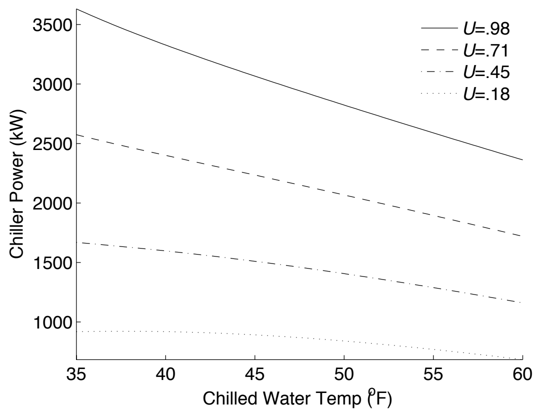


Figure A.8: Chilled Water Temperature.

For this case, q is the heat transferred by the device (either server or CRAH), $\dot{m}_{\mathrm{air}}$ represents the total air flowing through the device, $T_{\mathrm{a_h}}$ the temperature of the air exhausted by the server, and $T_{\mathrm{a_c}}$ 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 results 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 [14]:

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

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_{w_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 [27], 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 [15]. 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

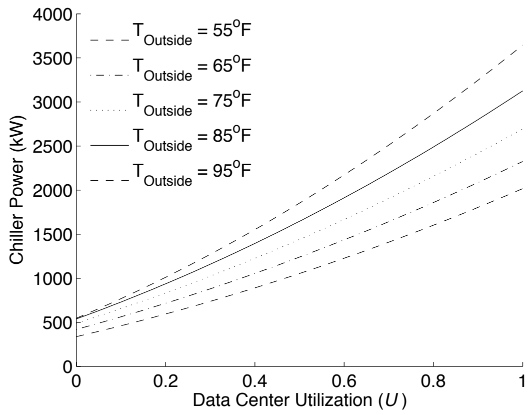


Figure A.9: Effects of U and T_{Outside} on P_{Chiller}

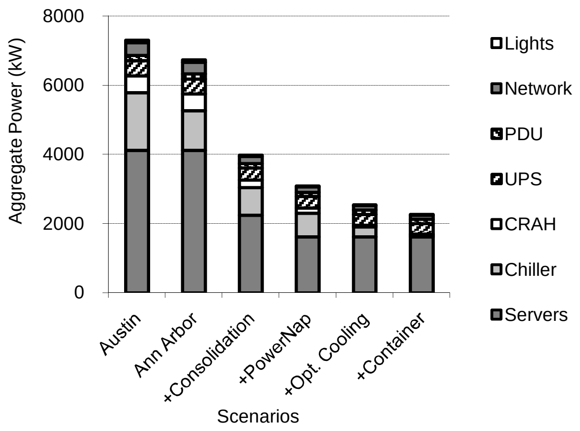


Figure A.10: A Case Study of Power-Saving Features.

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* [54] 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 $T_{\mathrm{w_c}}$ 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*

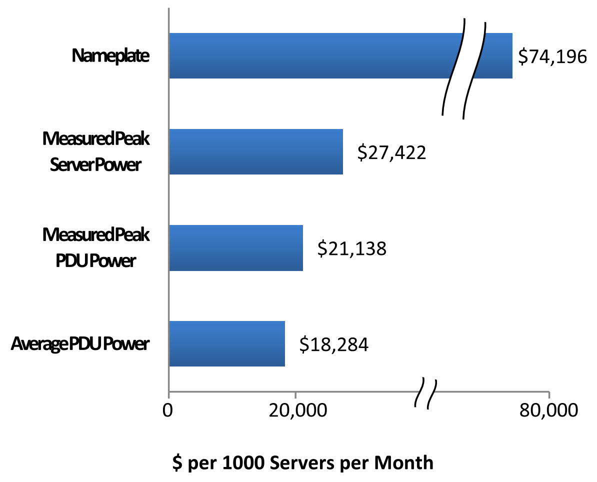


Figure B.1: The cost of over-provisioning. Amortized monthly cost of power infrastructure for 1000 servers under varying provisioning schemes.

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 [51, 90]. 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 [39]. Despite the enormous price tag, over-provisioning remains common at every layer of the power delivery system [39, 51, 28, 47, 71, 35]. 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 [54], 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 [35, 71]. Even in highly-tuned clusters running a single workload, peak utilization is rare, and still falls short of provisioned power capacity [28]. 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 [47, 96] or over ensembles [71, 97]. The most common power-capping approaches rely on throttling server performance to reduce power draw when budgets would otherwise be exceeded [47, 71, 96, 97].

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 [39] assuming high-availability power infrastructure costs \$15 per critical-load Watt [90], 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 [16] or virtual machine migration [21], 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 [80] and declustering in disk arrays [3]), 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 [37]. 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 [33], 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.

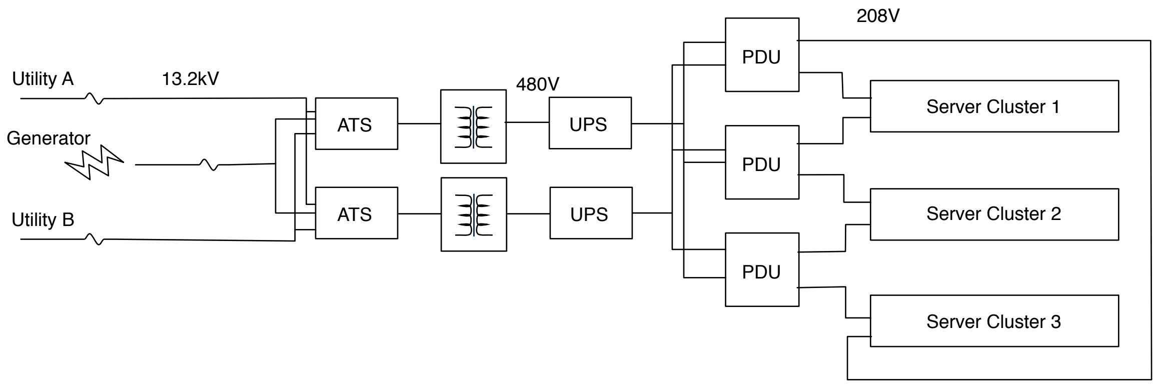


Figure B.2: Example power delivery system for a high-availability data center.

- **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 [51].

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 [91]. 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 [51, 90], 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 [37]. 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 [35, 71, 28, 97]. 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 [66] and HP [41] 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 [71, 47, 96, 30], examine optimal power allocation among heterogeneous servers [31]

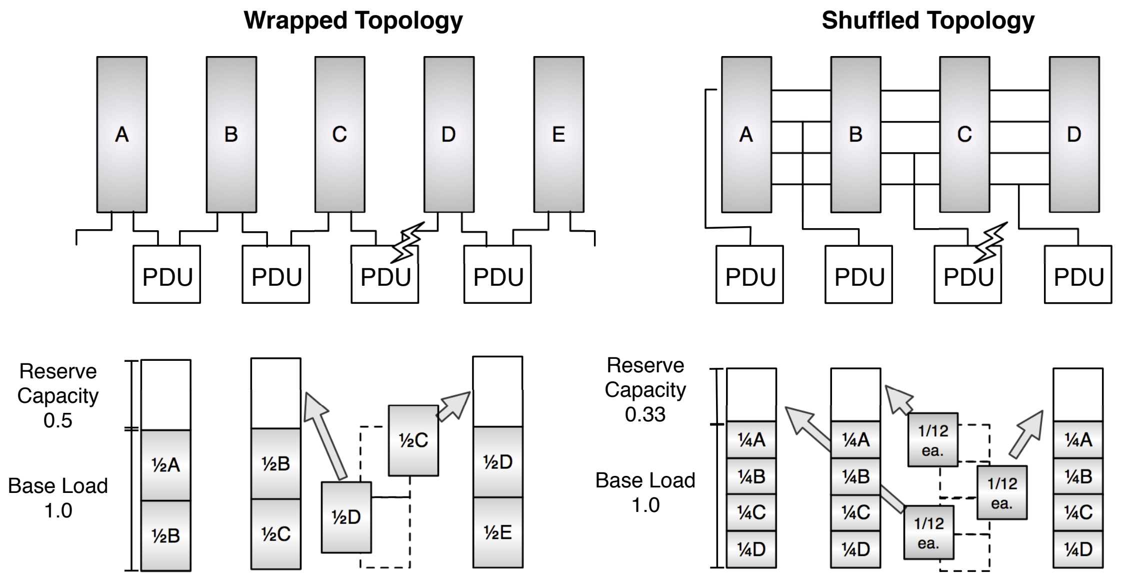
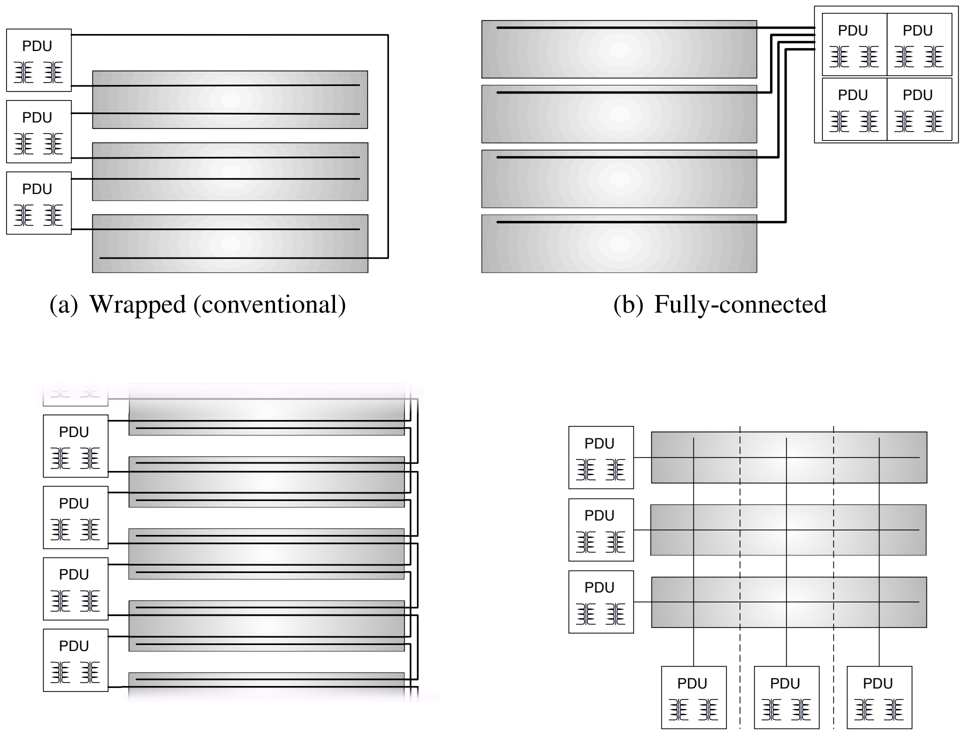


Figure B.3: Reduced reserve capacity under shuffled topologies (4 PDUs, fully-connected topology).

and identify the control stability challenges when capping at multiple levels of the power distribution hierarchy [69, 97]. Others have examined extending power management to virtualized environments [58]. Soft fuses [35] 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 [32].

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 [28]. 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 [73] 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 [71, 47, 96, 30, 69, 97, 58, 31]. 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

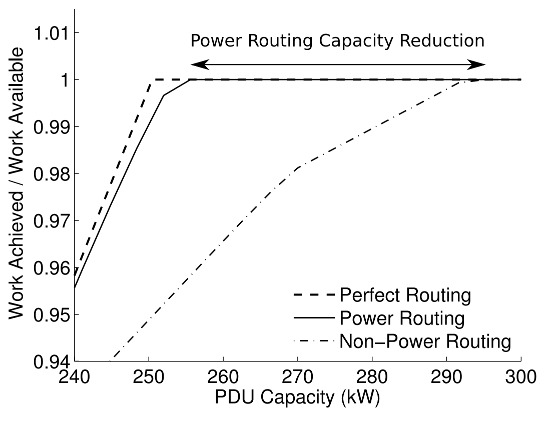


Figure B.5: Shuffled Topologies: 6 PDUs, fully-connected

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 [20], 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 [33]. 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 [24]. 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 [28]. 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 [84]. 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 approximate power draw in between these points.

Prior work [28, 71] 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.

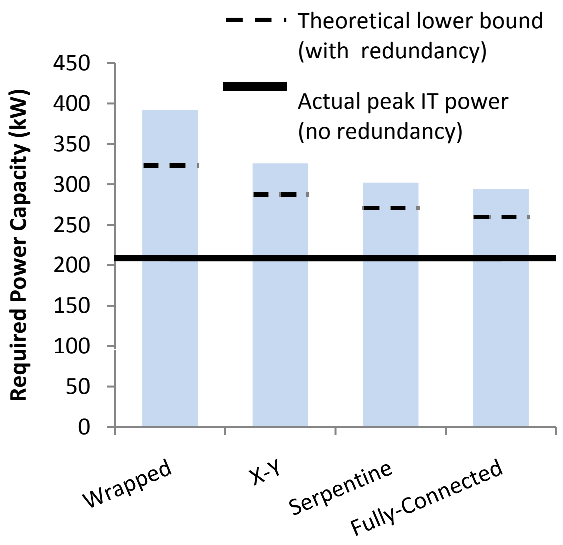


Figure B.6: Minimum capacity for redundant operation under shuffled topologies (no Power Routing).

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-

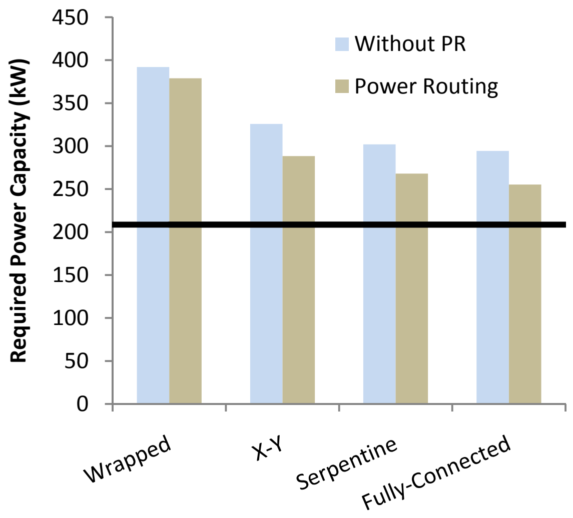


Figure B.7: Power Routing infrastructure savings as a function of topology.

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.

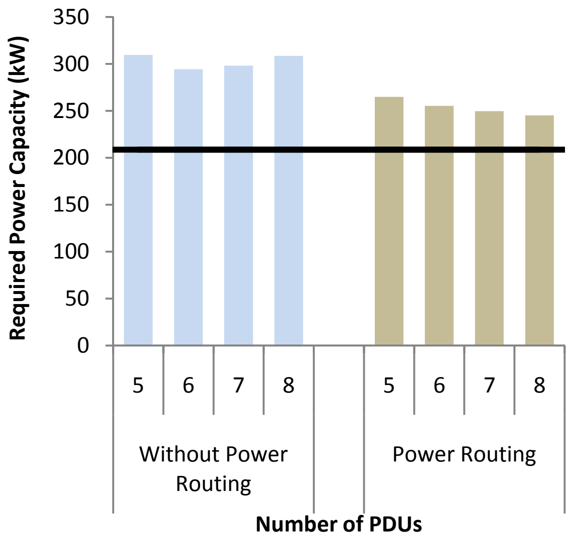
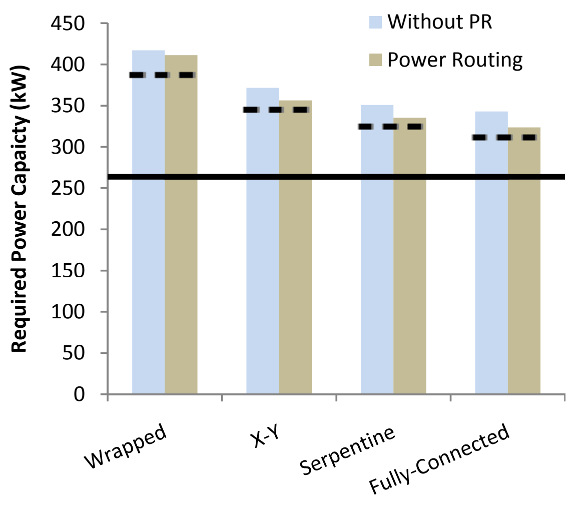


Figure B.8: Sensitivity of the fully-connected topology to number of PDUs.

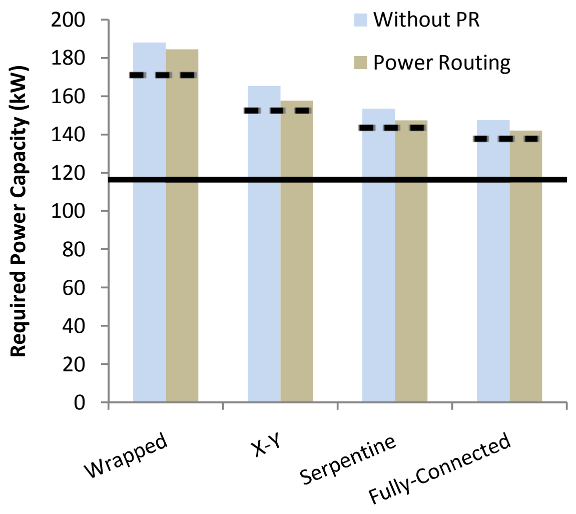
To explore these effects, we construct Power Routing test cases for 1000-server synthetic clusters where each server runs the same application. We do not study the web search application in isolation; in this application, the utilization on all servers rise and fall together, hence, the load on all PDUs 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 [5]), 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 [54], which allows servers to sleep during the millisecond-scale idle periods between task arrivals. We repeat the experiment shown in Figure B.7 under

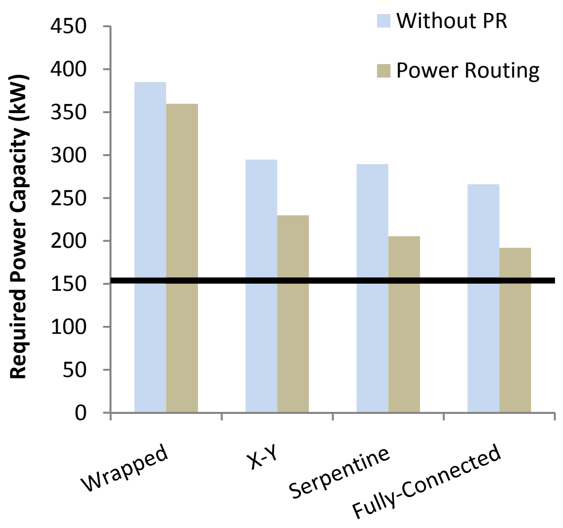


Figure B.10: Impact with energy-proportional servers.

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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] Philip A. Bernstein, Colin W. Reid, and Sudipto Das. Hyder - a transactional record manager for shared flash. In *Proc. of the 5th Biennial Conf. on Innovative Data Systems Research*, pages 9–20, 2011.

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

[9] 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.

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

[11] 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.

[12] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

[18] Peter M. Chen, Wee Teck Ng, Subhachandra Chandra, Christopher Aycock, Gurushankar Rajamani, and David Lowell. The rio file cache: surviving operating system crashes. In *Proc. of the 7th International Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 74–83, 1996.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

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

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

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

[36] Kevin M. Greenan and Ethan L. Miller. Reliability mechanisms for file systems using non-volatile memory as a metadata store. In *Proc. of the 6th International Conf. on Embedded Software*, pages 178–187, 2006.

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

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

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

[40] 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.

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

[42] 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.

[43] 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.

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

[45] 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.

[46] 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.

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

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

[49] 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.

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

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

[52] 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.

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

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

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

[60] 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.

[61] 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.

[62] 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.

[63] 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.

[64] 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.

[65] 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.

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

[67] 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.

[68] 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.

[69] 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*, 2008.

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

[71] 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.

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

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

[74] 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.

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

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

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

[78] M. Sarwat, M. Mokbel, X. Zhou, and S. Nath. Fast: a generic framework for flash-aware spatial trees. *Advances in Spatial and Temporal Databases*, pages 149–167, 2011.

[79] 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.

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

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

[82] SPARC International Inc. The SPARC Architecture Manual V8. http://www.sparc.org/standards/V8.pdf.

[83] 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.

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

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

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

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

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

[89] 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.

[90] 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.

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

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

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

[94] 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.

[95] 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.

[96] 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.

[97] 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.

[98] 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.

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