

and designers did not foresee the coming shift to multicore hardware. It would have been difficult to justify spending considerable effort in this area. Today, however, internal scalability of these engines is key to performance as core counts continue increasing.

1.2 Creating a scalable storage manager

Because none of the existing storage managers provide the kind of scalability required for modern multicore hardware, we set out to create a scalable storage manager based on the Shore storage manager. This exercise resulted in Shore-MT, which scales far better than its open source peers while also achieving superior single-thread performance (as shown in the experimental section). We find that the remaining gap between Shore-MT’s scalability and the ideal is due to hardware multithreading in the Niagara processor — threads contend for hardware resources within the processor itself, limiting the scalability software can achieve [29].

We use the lessons learned while creating Shore-MT to distill a set of important principles for developing scalable storage managers, many of which also apply to parallel software in general:

- Efficient synchronization primitives are critical. Poorly-designed or -applied primitives destroy scalability.
- Every component must be explicitly designed for scalability or it will become a bottleneck. Common operations must be able to proceed in parallel unless they truly conflict.
- Hotspots must be eliminated, even when the hot data is read-mostly. The serial overhead imposed by acquiring a shared-mode latch can grow quickly into a bottleneck, for example.
- Abstraction and encapsulations do not mix well with critical sections. We observed large gains by merging data structures with the synchronization primitives that protect them.

We present these principles as both a corroboration of prior work and an example of their practical application in a software system as large and complex as a storage manager. In particular, we demonstrate how systematically applying these principles can transform an existing, poorly performing storage manager to a fully competitive and scalable one.

1.3 Contributions and Paper Organization

To the best of our knowledge, this is the first paper to study and compare the internal scalability of open-source database storage managers on a modern, many-core machine. The contributions of this paper are:

- We identify the bottlenecks which prevent all of the open-source database storage managers we tested from scaling well on a modern multi-core platform. Our experiments show that performance flattens or even decreases after 4-12 hardware contexts.
- We develop Shore-MT — a multithreaded and scalable version of the Shore storage manager [7] — and make it available to the research community.¹
- Shore-MT achieves superior scalability and excellent single-thread performance compared to its peers. It is twice as fast as

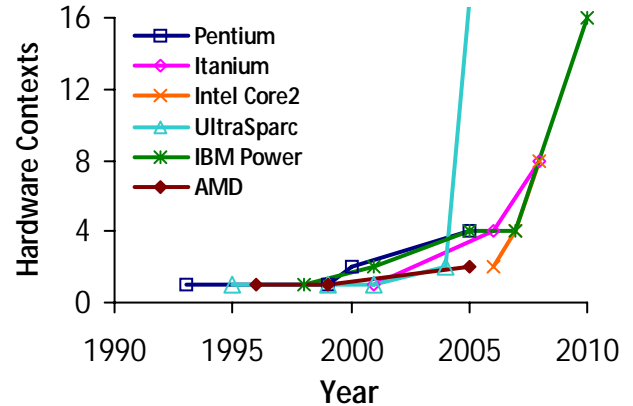


Figure 2. Number of HW contexts per chip as a function of time.

a commercial system and 2-4 times as fast as the fastest open-source system, across all of our experiments.

- Using lessons learned from the development of Shore-MT we show that designers must focus primarily on scalability rather than on single-thread performance. This focus will allow storage managers to exploit the parallelism future multicore chips make available, and our experience with Shore-MT shows that single-thread performance can actually improve as a side effect as well.

This paper is organized into two main parts. The first (Sections 2-4) details the trends and internal components that lead to bottlenecks in current systems, and examines the bottlenecks in each of the four storage managers. The second part (Sections 5-7) presents Shore-MT, measuring its performance and scalability and detailing lessons and principles we learned that can be applied to other engines. We conclude in Section 8.

2. BACKGROUND AND RELATED WORK

This section explains the recent architectural trends that make parallelism and scalability the most critical aspects of modern software design, as well as the main components in a database storage manager that determine the system’s behavior.

2.1 DB Computing in the Multicore Era

Despite Moore’s law doubling transistor density every two years, by the early 2000s it became clear to chip designers that power-related issues[21][10] prohibit commensurate increases in uniprocessor speed. Instead, each new generation of processors incorporates an ever-increasing number of processors (cores) on the same chip, leading the computing world into the so-called “multi-core era.” Section 2 illustrates how core counts of all popular architectures are growing exponentially; uniprocessor systems are increasingly difficult to find due to stagnant single-thread performance. As a result, all software must be able to exploit parallelism for performance rather than depending on succeeding processor generations to provide significant speedup to individual threads. Software designers should focus first on scalability, rather than single-thread performance, to ensure maximum system throughput as core counts grow exponentially.

1. See <http://www.cs.cmu.edu/~stageddb/systems/shoremmt.html>

This shift in processor design has jump-started a wide effort to extract parallelism from all types of existing software. While most database applications are inherently parallel, current database storage managers are designed under the assumption that only a limited number of threads will access their internal data structures during any given instant. Even when the application is executing parallelizable tasks and even if disk I/O is off the critical path, serializing accesses to these internal data structures impedes scalability.

An orthogonal way of scaling up a system to handle multiple requests in parallel is to use a distributed database. In fact, most of today's large database installations run on clusters of machines using either shared-nothing [13][12] or shared-disk [4] configurations. Recent proliferation of virtualization technology [6] allows system administrators to even go one step further and create virtual nodes within a single multicore machine. This is a desirable way of utilizing multicore systems in many applications, but transactional workloads favor a DBMS deployment with a single instance in the (multicore) node. Distributed transactions are notoriously difficult to implement correctly and efficiently [18][11], leading to severe performance degradation when multiple storage managers must participate in a transaction. Given the need for high performance inside a single multicore node, it is important to understand which factors may hinder scalability and discover ways to overcome them.

As we study in this paper, building a scalable database storage engine is challenging. One reason for that is the wide range of operations the storage engine is expected to support. However, if the domain is specific, or the requirements are relaxed the goal for scalability is feasible. For example, recent work proposes systems that provide relaxed consistency semantics, such as Amazon's Dynamo [11], and systems that operate exclusively in main-memory, such as H-Store [31]. These kinds of systems achieve significant gains in performance and/or scalability by giving up features traditionally provided by database storage engines. In specialized domains these systems perform admirably, but general-purpose database storage managers are still necessary for many transactional workloads, and their scalability is crucial.

2.2 Inside a Storage Manager

To provide a base for the discussions in the rest of the paper, this section briefly describes each component and briefly highlights some of the potential scalability challenges it presents. Figure 3 illustrates the major components of a storage manager and how they communicate. The storage manager forms the heart of a database management system. It is responsible for maintaining durability and consistency in the system by coordinating active transactions and their access to data, both in memory and on disk.

2.2.1 Buffer pool manager

The buffer pool manager fills two critical functions for the database. First, it presents the rest of the system with the illusion that the entire database resides in main memory, similar to an operating system's virtual memory manager. The buffer pool is a set of "frames," each of which can hold one page of data from disk. When an application requests a database page not currently in

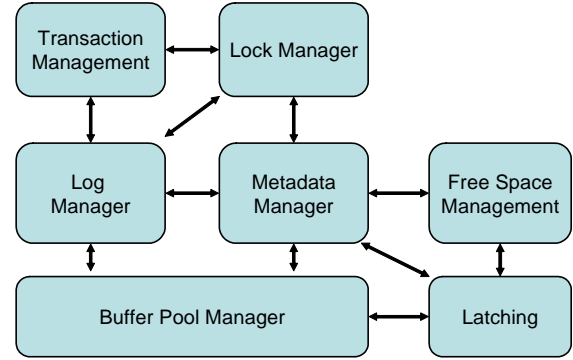


Figure 3. Main components of a storage manager

memory the buffer pool manager must fetch it from disk (evicting some other page) while the application waits. Applications "pin" in-use pages in the buffer pool to prevent their being evicted too soon, and unpin them when finished. Finally, the buffer pool manager and log manager (below) are responsible for ensuring that modified pages are flushed to disk (preferably in the background) so that changes to in-memory data become durable.

Buffer pools are typically implemented as large hash tables in order to find quickly any page requested by an application. Operations within the hash table must be protected from concurrent structural changes caused by evictions, usually with per-bucket mutex locks. Hash collisions and hot pages can cause contention among threads for the hash buckets; growing memory capacities and hardware context counts increase the frequency of page requests, and hence the pressure, which the buffer pool must deal with. Finally, the buffer pool must flush dirty pages and identify suitable candidates for eviction without impacting requests from applications.

2.2.2 Page latches

Pinning a page in the buffer pool ensures that it will remain in memory but does not protect its contents from concurrent modification. To protect its integrity, each page has a reader-writer lock called a latch associated with it; each operation acquires the latch in either read or write mode before accessing the page.²

Latches are a potential bottleneck for two reasons. First, very hot data such as metadata and high-level index pages are accessed so frequently that even compatible read operations can serialize attempting to acquire the latch [20]. Second, latch operations are typically very short, but when an transaction blocks on I/O it can hold a latch for several milliseconds and serialize many transactions behind it.

2.2.3 Lock manager

Database locks enforce logical consistency at the transaction level, ensuring that transactions do not interfere with the correctness of other concurrent transactions. One of the most intuitive (and restrictive) consistency models is "two phase locking"

2. Multi-versioned buffer pools avoid latches entirely by providing copy-on-write semantics for pages in memory. Transactional workloads typically access only a few bytes per page per transaction and quickly fill the buffer pool with near-identical copies of hot pages at the expense of other data.

(2PL), which dictates that a transaction may not acquire any new locks once it has released any. This scheme is sufficient to ensure that all transactions appear to execute in some serial order, though it can also restrict concurrency. In order to balance the overhead of locking with concurrency, the database provides hierarchical locks. In order to modify a row, for example, a transaction acquires a database lock, table lock, and row lock; meanwhile transactions which access a large fraction of a table may reduce overhead by “escalating” to coarser-grained locking at the table level.

Because each row in the database has a logical lock associated with it, the lock manager maintains a pool of locks and lock requests similar to a buffer pool; however, as the number of active locks can change drastically over time, the hash table is likely to have longer chains than the buffer pool, leading to more contention for buckets. Additionally, hierarchical locking results in extremely hot locks which most or all transactions acquire; again, this contention can serialize lock requests that will eventually turn out to be compatible.

2.2.4 Log manager

Database operations log all operations in order to ensure that they are not lost if the system database fails before the buffer pool flushes those changes to disk. The log also allows the database to roll back modifications in the event of a transaction abort. Most storage managers follow the ARIES scheme [25][26], which combines the log, buffer pool manager, and concurrency control schemes into a comprehensive recovery scheme.

The database log is a serial record of all modifications to the database, and therefore forms a potential scalability bottleneck as the number of concurrent operations increases.

2.2.5 Transaction management

The storage manager must maintain information about all active transactions, especially the newest and oldest in the system, in order to coordinate services such as checkpointing and recovery. Checkpointing allows the log manager to discard old log entries, saving space and shortening recovery time. However, no transactions may begin or end during checkpoint generation, producing a potential bottleneck unless checkpoints are very fast.

2.2.6 Free space and metadata management

The storage manager must manage disk space efficiently across many insertions and deletions, in the same way that malloc() and free() manage memory. It is especially important that pages which are scanned regularly by transactions be allocated sequentially in order to improve disk access times; table reorganizations are occasionally necessary in order to improve data layout on disk. In addition, databases store information about the data they store, and applications make heavy use of this metadata.

The storage manager must ensure that changes to metadata and free space do not corrupt running transactions, while also servicing a high volume of requests, especially for metadata.

3. EXPERIMENTAL ENVIRONMENT

Because the basis of this work lies in evaluating the performance of database engines, we begin by describing our experimental environment. All experiments were conducted using a Sun T2000 (Niagara) server [21][10] running Solaris 10. The Niagara chip has an aggressive multi-core architecture with 8 cores clocked at 1GHz; each core supports 4 thread contexts, for a total of 32 OS-visible “processors.” The 8 cores share a common 3MB L2 cache and each of them is clocked at 1GHz. The machine is configured with 16GB of RAM and its I/O subsystem consists of a RAID-0 disk array with 11 15kRPM disks.

We relied heavily on the Sun Studio development suite, which integrates compiler, debugger, and performance analysis tools. Unless otherwise stated every system is compiled using version 5.9 of Sun’s CC. All profiler results were obtained using the ‘collect’ utility, which performs sample-based profiling on unmodified executables and imposes very low overhead (<5%).

3.1 Storage Engines Tested

We evaluate four open-source storage managers: PostgreSQL [30], MySQL [2], BerkeleyDB [1], and Shore [7]. For comparison and validation of the results, we also present measurements from a commercial database manager (DBMS “X”).³ All database data resides on the RAID-0 array, with log files sent to an in-memory file system. The goal of our experiments is to exercise all the components of the database engine (including I/O, locking and logging), but without imposing I/O bottlenecks. Unless otherwise noted, all storage managers were configured with 4GB buffer pools.

In order to extract the highest possible performance from each storage manager, we customized our benchmarks to interface with each storage manager directly through its respective C API. Client code executed on the same machine as the database server, but we found the overhead of clients to be negligible (<5%).

PostgreSQL v8.1.4: PostgreSQL is an open source database management system providing a powerful optimizer and many advanced features. We used a Sun distribution of PostgreSQL optimized specifically for the T2000. We configured PostgreSQL with a 3.5GB buffer pool, the largest allowed for a 32-bit binary.⁴ The client drivers make extensive use of SQL prepared statements.

MySQL v5.1.22-rc: MySQL is a very popular open-source database server recently acquired by Sun. We configured and compiled MySQL from sources using InnoDB as the underlying transactional storage engine. InnoDB is a full transactional storage manager (unlike the default, MyISAM). Client drivers use dynamic SQL syntax calling stored procedures because we found they provided significantly better performance than prepared statements.

BerkeleyDB v4.6.21: BerkeleyDB is an open source, embedded database engine currently developed by Oracle and optimized for C/C++ applications running known workloads. It provides full storage manager capabilities but client drivers link against the

3. Licensing restrictions prevent us from disclosing the vendor.

4. The release notes mention subpar 64-bit performance

database library and make calls directly into it through the C++ API, avoiding the overhead of a SQL front end. BerkeleyDB is fully reentrant but depends on the client application to provide multithreaded execution. We note that BerkeleyDB is the only storage engine without row-level locking; its page-level locks can severely limit concurrency in transactional workloads.

Shore: Shore was developed at the University of Wisconsin in the early 1990's and provides features that all modern DBMS use: full concurrency control and recovery with two-phase row-level locking and write-ahead logging, along with a robust implementation of B+Tree indexes. The Shore storage manager is designed to be either an embedded database or the back end for a "value-added server" implementing more advanced operations. Client driver code links directly to the storage manager and calls into it using the API provided for value-added servers. The client code must use the threading library that Shore provides.

3.2 Benchmarks

We evaluate storage managers using a small suite of microbenchmarks which each stresses the engine in different ways. We are interested in two metrics: throughput (e.g. transactions per second) and scalability (how throughput varies with the number of active threads). Ideally an engine would be both fast and scalable, but as we will see, storage managers tend to be either fast or scalable, but not both.

Record Insertion: The first microbenchmark repeatedly inserts records into a database table backed by a B-Tree index. Each client uses a private table; there is no logical contention and no I/O on the critical path.⁵ Transactions commit every 1000 records, with one exception. We observed a severe bottleneck in log flushes for MySQL/InnoDB and modified its version of the benchmark to commit every 10000 records in order to allow a meaningful comparison against the other engines. Record insertion stresses primarily the free space manager, buffer pool, and log manager. We use this benchmark for the primary scalability study in the next section because it is entirely free from logical contention, unlike the other two.

TPC-C Payment: The TPC-C benchmark models a workload of short transactions arriving at high frequency [33]. The Payment transaction updates the customer's balance and corresponding district and warehouse sales statistics. It is the smallest transaction of the TPC-C transaction mix, reading 1-3 rows and updating 4 others. One of the updates made by Payment is to a contended table, WAREHOUSE. Payment stresses the lock manager, log manager, and B-Tree probes.

TPC-C New Order: The New Order is a medium-weight transaction which enters an order and its line items into the system, as well as updating customer and stock information to reflect the change. It inserts roughly a dozen records in addition to reading and updating existing rows. This transaction stresses B-Tree indexes (probes and insertions) and the lock manager.

Payment and New Order together comprise 88% of transactions executed by the TPC-C benchmark. We use them to measure per-

5. All the engines use asynchronous page cleaning and generated more than 40MB/sec of disk traffic during the tests.

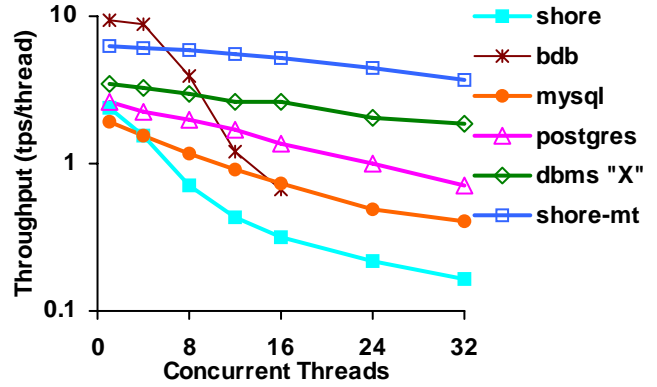


Figure 4. Scalability and performance comparison of Shore-MT vs several open-source engines and one commercial engine. The graph shows the performance of the storage managers for more realistic workloads; as the following sections show, the overall scalability trends are the same for all the benchmarks.

4. EVALUATION OF EXISTING ENGINES

We begin by benchmarking each database storage manager under test and highlight the most significant factors that limit its scalability. Figure 4 compares the scalability of the various engines when we run the insert-only micro-benchmark. Due to lock contention in the transactional benchmarks, the internals of the engines do not face the kind of pressure they do on the insert-only benchmark. Thus we use the latter to expose the scalability bottlenecks at high core counts and to highlight the expected behavior of the transactional benchmarks as the number of hardware contexts per chip continues to increase.

To have a better insight on what is going on we profile the runs with multiple concurrent clients (16 or 24) stressing up the storage engine. Then we collect the results and interpret call stacks to identify the operations where each system spends its time.

PostgreSQL: PostgreSQL suffers a loss of parallelism due to three main factors. First, contention for log inserts causes threads to block (XLogInsert). Second, calls to malloc add more serialization during transaction creation and deletion (CreateExecutorState and ExecutorEnd). Finally, transactions block while trying to lock index metadata (ExecOpenIndices), even though no two transactions ever access the same table. Together these bottlenecks only account for 10-15% of total thread time, but that is enough to limit scalability.

MySQL: MySQL/InnoDB is bottlenecked on two spots. The first one is the interface to InnoDB; in a function called `srv_conc_enter_innodb` threads remain blocked as long as around the 39% of the total execution time. The second one are the log flushes. In another function labeled `log_preflush_pool_modified_pages` the system again experiences large blocking time equal to the 20% of the total execution time (even after increasing transaction length to 10K inserts).

We also observe that mysql spends a non-trivial fraction of its time on two malloc-related functions, `take_deferred_signal` and `mutex_lock_internal`. This suggests a potential for improvement by avoiding excessive use of malloc (trash stacks, object re-use, thread-local malloc libraries...)

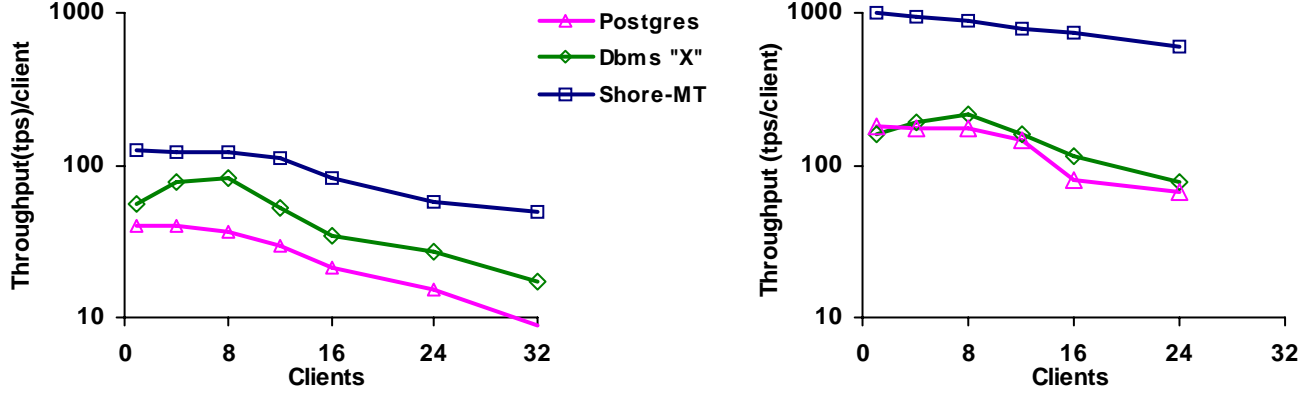


Figure 5. Per-client throughput of Shore-MT, DBMS “X” and PostgreSQL for the New Order (left) and Payment (right) microbenchmarks

BerkeleyDB: BDB spends the majority of its time on either testing for availability or trying to acquire a mutex — the system spends over 80% of its processing time in two functions with names `_db_tas_lock` and `_lock_try`. Presumably the former is a spinning test-and-set lock while the latter is the test of the same lock. Together these likely form a test-and-test-and-set mutex primitive, which is supposed to scale better than the simple test-and-set. The excessive use of test-and-test-and-set (TATAS) locking justifies the high performance of BDB on low contention cases, since the TATAS locks impose very little overhead on low contention, but fail miserably on high contention.

As we know BerkeleyDB employs page-level locking. The coarse-grained locking by itself imposes scalability problems. The two callers for lock acquisition have names `_bam_search` and `_bam_get_root`. So, in high contention BDB spends most of its time trying to acquire the latches for tree probes. Additionally, we see that the system spends significant amount of time blocked waiting on a `pthread_mutex_lock` and `cond_wait`, most probably because the `pthread` mutexes are used as a fallback plan to acquire the highly contended locks.

This section highlights the bottlenecks we observed in existing storage managers, and which developers cannot ignore if the goal is a true scalable system in emerging many-core hardware. As the PostgreSQL case illustrates, what appears to be a small bottleneck can still hamper scalability as the number of concurrent clients increases

5. PERFORMANCE OF SHORE-MT

We originally set out to make Shore a scalable storage manager. In the quest to achieve scalability in Shore we completely ignored single-thread performance and focused only on removing bottlenecks. This section presents the final outcome, Shore-MT, comparing it with the other storage managers.

Figure 4 and those that follow display performance as the number of transactions per second per thread, plotted on a log-y axis. We use log-y scale on the graphs because it shows scalability clearly without masking absolute performance. Linear y-axis is misleading because two systems with the same scalability will have differently-sloped lines, making the faster one appear less scalable

than it really is. In contrast, a log-y graph gives the same slope to curves with the same scalability.

Shore-MT scales commensurately with the hardware we make available to it, setting the absolute example for other systems to follow. Figure 4 shows the scalability achieved by the different engines running the insert microbenchmark as the number of concurrent threads varies along the x-axis. While single-threaded Shore did not scale at all, Shore-MT exhibits excellent scaling. Moreover, at 32 clients it scales better than DBMS X, a popular commercial DBMS. Therefore, the key ideas upon which a software designer should rely to build scalable software (detailed in Section 7) lead to very good scaling performance. While our original goal was only to achieve high scalability, we also achieved nearly 3x speedup in single-thread performance over the original Shore, leading to a healthy performance lead over the other engines.⁶ We attribute the performance improvement to the fact that database engines spend so much time in critical sections. The process of shortening and eliminating critical sections, and reducing synchronization overhead, had the side effect of also shortening the total code path for a single thread.

As a further comparison, Figure 5 shows the performance of the three fastest storage managers running the New Order (left) and Payment (right) microbenchmarks. Again Shore-MT achieves the highest performance⁷ while scaling as well as the commercial system for New Order. All three systems encounter high contention in the STOCK and ITEM tables, causing a significant dip in scalability for all three around 16 clients. Payment, in contrast, imposes no application-level contention, allowing Shore-MT to scale all the way to 32 threads.

We note that, while both Shore-MT and DBMS “X” scale well up to 32 threads, profiling indicates that both face looming bottlenecks (both in log inserts, as it happens). Our experience tuning Shore-MT suggests that synchronization bottlenecks are an ongoing battle as the number of threads continues to increase. How-

6. BerkeleyDB outperforms the other systems at first, but its performance drops precipitously for more than four clients.

7. Some of Shore’s performance advantage is likely due to its clients being directly embedded in the engine while the other two engines communicate with clients using local socket connections. We would be surprised, however, if a local socket connection imposed 100% overhead per transaction!

ever, Shore-MT proves that even the largest bottlenecks can be addressed using the principles detailed in Section 6. Storage engine designs can make use of them in the future to ensure scalability and take full advantage of the available hardware.

6. ACHIEVING SCALABILITY

We chose the Shore storage manager as our target for optimization for two reasons. First, Shore supports all the major features of modern database engines: full transaction isolation, hierarchical locking, a CLOCK buffer pool with replacement and prefetch hints, B-Tree indexes, and ARIES-style logging and recovery. Additionally, Shore has previously shown to behave like commercial engines at the instruction level [3], making it a good open-source platform for comparing against closed-source engines.

The optimization process from Shore to Shore-MT was straightforward. We began each step by profiling Shore in order to identify the dominant bottleneck(s). After addressing each bottleneck, scalability would either increase, or a previously minor bottleneck would take its place. We then repeated the profiling and optimization steps until Shore became compute-bound for 32 threads.

In the absence of external bottlenecks such as I/O or unscalable database applications, virtually all bottlenecks centered on the various critical sections in Shore’s code. Every major component of the storage manager must communicate with multiple threads, and key data structures must be protected from concurrent accesses that would corrupt them.

6.1 Scalability or Performance First?

Traditionally, software optimization has focused on improving single-thread performance, while addressing scalability almost as an afterthought. Algorithms and data structures within the storage manager are often designed for performance, then modified after the fact to remove any scalability problems that might arise. We find that the “performance first” approach quickly leads to problems as the number of contexts in the system continues to climb exponentially. Optimizations that improve performance significantly for one or a few threads often have a limited impact on performance — or even reduce it — as more threads enter the system. H-Store [31] is an extreme example of this effect, achieving very high single-thread performance but with no possibility for scalability within a single instance of the storage manager.

Three factors lead to tension between speed and scalability. First, optimizations which impact only single-thread performance have the effect of allowing threads to reach bottleneck critical sections faster, increasing contention. The critical sections will then limit throughput or even reduce it if protected with unscalable synchronization primitives. BerkeleyDB, with its TATAS locks, is a good example of this effect (see Section 4). Second, the most scalable synchronization primitives tend to also have the highest overhead. As a result, single-thread performance can potentially drop when the software is modified to be more scalable. Finally, improving single-thread performance raises the bar for scalability. Even if the throughput for many threads does not drop, the scalability will have been reduced when the (now faster) single thread case is used as a baseline.

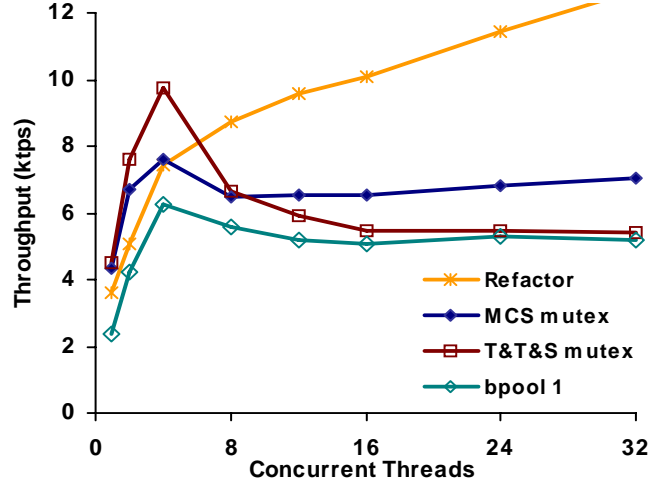


Figure 6. Examples of the kinds of impact optimizations can have on performance and scalability

The graphs in Figure 6 give examples of the kinds of impact optimizations can have on overall performance. Each line in the graph depicts the performance and scalability of optimizations to Shore-MT made between the “bpool 1” and “caching” versions shown in Figure 7 (c.f. Section 7.3). The y-axis of Figure 6 plots system throughput as the number of threads in the system varies along the x-axis. At this stage in Shore’s development the profiler identified a contended pthread mutex in the free space manager as the primary bottleneck in the system.

Our first optimization attempt replaced the pthread mutex with a test-and-test-and-set (T&T&S) mutex having much lower overhead, in hopes of quickly relieving pressure on the critical section [20]. The reduced overhead improved single-thread performance by 90% but did not improve scalability; in fact, *scalability dropped by 50% because single-thread performance doubled with no change for 32 threads*. The next optimization attempt replaced the non-scalable T&T&S mutex with a scalable MCS mutex. This time scalability improved noticeably but the critical section remained contended.

Having exhausted the “easy” approaches for eliminating the bottleneck, we examined the code more closely and determined that the free space manager acquired a page latch while holding the mutex; the page latch acquire was the biggest part of the critical section even in the best case; in the worst case the latch was taken or the page was not in memory, leading to long delays which serialized other threads. By refactoring the code we were able to move the latch acquire outside the critical section, removing the pressure and vastly improving scalability. The overhead we introduced reduced single-thread performance by about 30%, but there was a net gain of about 200% for 32 threads.

This sequence of optimizations illustrate how focusing only on performance for a few threads can be misleading. As we saw in Section 4, several open-source storage managers fell into this trap, performing well at first but failing to scale past 4-8 threads. This observation supports our conclusion that one should focus first on scalability, with performance as a secondary goal.

6.2 Principles for Scalable Storage Managers

We next introduce four fundamental principles for achieving scalability in a database storage manager.

- Efficient synchronization primitives are critical. Poorly-designed or -applied primitives destroy scalability.
- Every component must be explicitly designed for scalability or it will become a bottleneck. Common operations must be able to proceed in parallel unless they truly conflict.
- Hotspots must be eliminated, even when the hot data is read-mostly. The serial overhead imposed by acquiring a shared-mode latch can grow quickly into a bottleneck, for example.
- Abstraction and encapsulations do not mix well with critical sections. We observed large gains by merging data structures with the synchronization primitives that protect them.

As can be seen in Section 7, we applied each principle many times to different parts of the code; the rest of this section gives examples from the development of Shore-MT where each principle had a significant impact on scalability.

6.2.1 Use the right synchronization primitives

Problem: Conceptually, every page search that hits in the buffer pool requires at least three critical sections - one to lock the hash bucket (temporarily preventing other thread from moving the page to other buckets during the search), one to pin it (preventing evictions while the thread is using the page), and a final critical section to request a latch on the page (preventing concurrent modification of the page's data). For hot pages, especially, the critical sections can become a bottleneck as many threads compete for them — even if they all latch the page in read-mode.

Observation: Pinned pages cannot be evicted.

Solution: When a buffer pool search finds a promising page it applies an atomic "pin-if-pinned" operation to the page, which increments the page's pin-count only if it is non-zero (easily implemented using an atomic compare-and-swap). If the page is not already pinned the thread must lock the bucket in order to pin it, but if the conditional pin succeeds, the page cannot have been evicted because the pin count was non-zero (though it may have been moved to a different bucket). The thread then verifies that it pinned the correct page and returns it. In the common case this eliminates the critical section on the bucket.

6.2.2 Shorten or remove critical sections

Problem 1: Shore uses logical logging for many low-level operations such as page allocations. Because of this, Shore must verify that a given page belongs to the correct database table before inserting new records into it. The original page allocation code entered a global critical section for every record insertion in order to search page allocation tables.

Observation: Page allocations do not change often. Also, The information immediately becomes stale upon exit from the critical section. Once the page has been brought into the buffer pool, Shore must check the page itself anyway.

Solution: We added a small thread-local cache to the record allocation routine which remembers the result of the most recent lookup. Because Shore allocates extents of 8 pages and tends to

fill one extent completely before moving on to the next, this optimization cut the number of page checks by over 95%.

Problem 2: The Shore log manager originally used a non-circular buffer and synchronous flushes. Log inserts would fill the buffer until a thread requested a flush or it became full (triggering a flush). The flush operation would drain the buffer to file before allowing inserts to continue.

Observation: Log inserts have nothing to do with flushes as long as the buffer is not full, and flushing should almost never interfere with inserts. Further, using a circular buffer means the buffer never fills as long as flushes can keep up with inserts (on average).

Solution: We converted Shore to use a circular buffer and protected each operation (insert, compensate, flush) with a different mutex. Insert and compensate each use a light-weight queueing mutex, while the slower flush operation uses a blocking mutex. Inserts own the buffer head, flushes own the tail, and compensations own a marker somewhere in between (everything between tail and marker is currently being flushed). Inserts also maintain a cached copy of the tail pointer. If an insert would pass the cached tail, the thread must update the tail to a more recent value and potentially block until the buffer drains. Flush operations acquire the compensation mutex (while holding the flush mutex) just long enough to update the flush marker. Log compensations acquire only the compensation mutex, knowing that anything between the flush marker and the insertion point is safe from flushing. Distributing critical sections over the different operations allows unrelated operations to proceed in parallel and prevents fast inserts and compensations from waiting on slow flushes.

6.2.3 Eliminate hotspots

Problem: Shore's buffer pool was implemented as an open chaining hash table protected by a single global mutex. In-transit pages (those being flushed to or read from disk) resided in a single linked list. Virtually every database operation involves multiple accesses to the buffer pool, and the global mutex became a crippling bottleneck for more than about four threads.

Observation:

- Every thread pins one buffer pool page per critical section.
- Most buffer pool searches (80-90%) hit.
- Most in-transit pages are reads thanks to page cleaning.

Solution: We distributed locks in the buffer pool in three stages. The first stage added per-bucket locks to protect the different hash buckets, leaving the global lock to protect in-transit and free lists and the clock hand. However, each page search had to lock a bucket in order to traverse the linked list. This remained a serious bottleneck for hot pages.

The second stage replaced the open chaining hash table with a 3-ary cuckoo hash table [27][14][32]. Cuckoo hashes use N hash functions to identify N locations a value may legally reside in. A collision only occurs if all N locations for a value are full, and is resolved by evicting some other entry from the table. Evicted entries are then re-inserted into one of their other $(N-1)$ slots, potentially causing a cascade of evictions. Cuckoo hashes have two extremely desirable properties. Like open chaining hash tables, deletions are straight-forward. Like closed hash tables,

updates and searches only interfere with each other when they actually touch the same value at the same time. Because the buffer pool is merely a cache, we can also evict particularly troublesome pages in order to end cascades. Cuckoo hashing does face one major drawback, however: operations cost more because they must evaluate multiple high-quality hashes.⁸ In our case, the improvement in concurrency more than offsets the increased cost of hashing by eliminating hot spots on bucket lists.

Unfortunately, increasing the number of threads still further also increased the number of misses per unit time (though the rate remained the same), leading to contention for in-transit lists. On every miss the buffer pool must ensure that the desired page is not currently in-transit. An in-transit-out page cannot be read back in until the dirty data has been written out, and requests for an in-transit-in page must block until the read completes. The large number of transits in progress at any moment led to long linked list traversals and slowed down miss handling. The third stage broke the in-transit lists into several smaller ones (128 in our case) to cut the length of each list. In addition, the buffer pool immediately places in-transit-in pages in the hash table, making it visible to searches and bypassing the transit lists. The thread performing the page read simply holds the page latch in EX mode to block other accesses until the I/O completes. As a result, only true misses search the in-transit-list, which contains only dirty pages currently being evicted. Because effective asynchronous page cleaning makes dirty page evictions very rare, each "in-transit" list is currently a fixed-length array of only one element.

Finally, page misses release the clock hand before evicting the selected replacement frame or attempting to read in the new data. This allows the thread to release the clock hand mutex before attempting expensive I/O. As a result, the buffer pool can service many page misses simultaneously even though an individual page miss is very slow.

6.2.4 Beware of over-abstraction and encapsulation

Problem: Log inserts remained a major bottleneck in Shore, even after decoupling inserts from flushes. In Shore, log inserts occasionally acquire a blocking mutex in order to wake checkpoint and flush threads at regular intervals. In addition, the log manager was highly encapsulated and abstracted in ways that cut through critical sections. For example, the log buffer was embedded deep in the file-handling portion of the log manager in a way that made a circular log difficult to implement. Overall, the insert critical section was far too long.

Observations:

- The log insert critical section only needs to determine where in the buffer the write will (eventually) occur, with what log sequence number (LSN), and whether the buffer currently has space. Once the location of the write is settled, the thread is free to insert the log record at its leisure, perhaps even after waiting for a flush to complete.

8. Cuckoo hashing is extremely prone to clustering with weak hash functions. Our implementation combines three universal hash functions to make one high-quality hash [27].

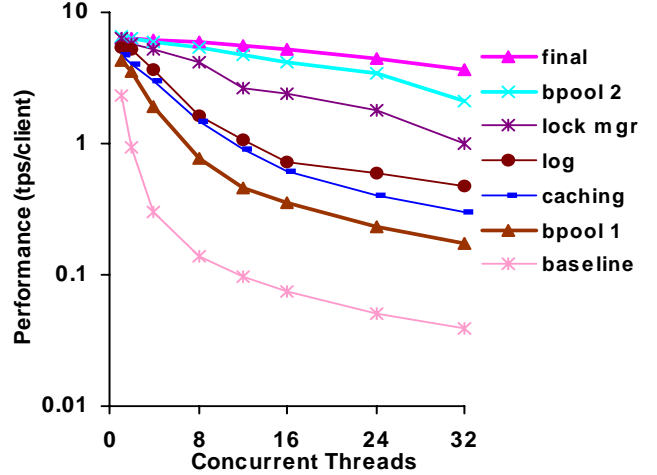


Figure 7. Performance and scalability improvements due to optimizations detailed in Section 7.

- Notifications to daemon threads need not occur immediately - the thread can safely release the insert mutex first, moving the operation off the critical path of other threads.
- Encapsulation is a means, not an end.

Solution: We rewrote the log manager to make encapsulation boundaries surround, rather than cut through, critical sections. The refactoring moved the log buffer above the file handling to finish decoupling inserts from log flushes. Threads perform only the minimum work required to allow them to safely insert a log record later, copying data and notifying daemon threads after releasing the mutex. Finally, because log insert performance is so critical, we redesigned log buffer as an extended queuing lock, combining the functionality of the critical section with the mechanism for protecting it. As each thread enters the log buffer's queue it computes which portion of the log buffer it will eventually write to. It then hands off this information to its successor in the queue at the same time as it exits the critical section. Finally, the log flush daemon follows behind, dequeuing all threads' left-over nodes as it determines which regions of the log buffer to flush. This scheme has three advantages: First, contended state of the log buffer (insert offset) is passed from thread to thread in an orderly fashion, with no contention at hand-off. By combining the functionality of an MCS queuing lock [24] with the log buffer, we also consolidate the code and eliminate overhead. Finally, because a thread only needs to determine where it will eventually insert (not when), it only serializes threads behind it for a short time.

7. FROM SHORE TO SHORE-MT

This section details the process of converting Shore into Shore-MT. We began with the version 5.01 release⁹ and methodically removed scalability impediments identified by Sun's sampling profiler, collect. The improvements clustered into several stages; many optimizations removed a bottleneck, only to have another replace it with no change in scalability. Figure 7 shows the performance of Shore-MT running the insert microbenchmark after

9. Available at <http://ftp.cs.wisc.edu/paradise/sm5.0/>

each major stage of the optimization process. The rest of this section describes the major changes for each phase.

7.1 Baseline

Shore implements a user-level thread library (smthreads) on top of a single operating system thread; auxiliary “diskrw” processes service blocking I/O requests and communicate with Shore through shared memory regions. In order to begin making Shore scalable we first had to replace the thread package with standard POSIX threads (already partly supported) and modify the Shore scheduler to allow them to run in parallel. We also removed the external I/O engines as they were no longer necessary. Because Shore was written for cooperative multithreading on a single OS thread, there were many low-level data races in the code due to unprotected critical sections. We identified these critical sections using code inspection (many were identified by existing comments) as well as Sun’s race detection tool. We then inserted POSIX mutex locks to protect each critical section. Higher-level critical sections were already properly protected because lock contention and I/O could cause inconvenient context switches even under cooperative multithreading. At this stage we also fixed several portability bugs due to our sparc64/solaris environment (Shore only officially supports x86/linux).

The resulting system (“baseline” in Figure 7) was correct but completely unscalable; throughput ranged from 2.4tps at one thread to about 1.2tps for four or more threads.

7.2 Bufferpool Manager

The bufferpool manager was the first major scalability challenge. Shore protected it using a single, global mutex that very quickly became contended. We replaced the global mutex with one mutex per hash bucket. At this point we also applied the atomic pin count update optimization described in Section 6.2.1. Finally, we replaced several key pthread mutex instances with test-and-set spinlocks that acquire a pthread mutex and cond var only under contention. This reduced the overhead of the common (uncontended) case significantly.

These changes (“bpool 1”) boosted scalability significantly, doubling single-thread performance and increasing 32-thread throughput by nearly 5x.

Principles applied:

- shorten or remove critical sections
- eliminate hotspots
- use the right synchronization primitive

7.3 Free Space and Transaction Management

After tuning the buffer pool the next bottlenecks were in the free space and transaction management components Shore. These bottlenecks were all straightforward to address, however. An examination of the code in the free space manager showed that it held a contended mutex while acquiring a page latch (which could be contended or block on I/O). We refactored the operation so that it was safe to release the mutex before acquiring the latch, reducing significantly the pressure on the former. We also discovered a bottleneck on the mutex protecting the transaction list: the most common operations checked the head of the list to determine the

ID of the oldest transaction in the system. We added a local variable to store this ID; callers could read it atomically because IDs are 64-bit integers, and committing transactions would update the ID when they removed themselves from the list.

Finally, we made one more optimization to the buffer pool; we already eliminated one of the three critical sections required to latch a page (the pin operation), and here we added a small array for especially hot pages; instead of protecting the array with a mutex, we changed the search to pin the page, then check its ID before acquiring the latch; if a page eviction occurs before the pin completes the IDs would not match; once the pin succeeds the caller is assured that the page will not move.

The “caching” line in Figure 7 shows the resulting performance. Single-thread speed did not change, but scalability nearly doubled.

Principles applied:

- shorten or remove critical sections
- eliminate hotspots
- eliminate counterproductive abstraction

7.4 Log Manager

Shore’s log manager presents a single API to the rest of the system, implemented as a virtual class backed by a fairly complex hierarchy of subclasses. The entire component was protected by a single mutex, even though there are four major log operations, each with varying cost. In order to allow these (usually compatible) operations to proceed in parallel, we split the log critical section into four, as described in Section 6.2.2. We further optimized the log insert operation, as described in Section 6.2.4, so that insertions serialize just long enough to claim buffer space; the transactions then copy log entries into the space in parallel, blocking on a full buffer if necessary, then notify the log flush daemon when they complete. As a result, the most common log operation — insert — requires an extremely short critical section.

At this point the profiler identified a significant bottleneck in calls to malloc() and free(). Fortunately, Solaris provides a thread-local version of these functions which trades higher memory utilization for zero contention between threads.

The free space manager also became a problem again at this point as well — this time in the metadata check to determine which table a given page ID belongs to. We alleviated the bottleneck by creating a small cache of most recently-used extent ids (an extent is 8 consecutive pages), which allowed the hottest page accesses to avoid accessing metadata pages at all. Last of all, we changed the buffer pool from an open chained hash table to a cuckoo hash at this point (see Section 6.2.3)

The impact of these optimizations is captured by the “log” line in Figure 7. Again, scalability nearly doubled.

Principles applied:

- shorten or remove critical sections
- eliminate hotspots

7.5 Lock Manager

Profiling pointed to the lock manager as the next bottleneck to tackle. Like the bufferpool, the lock manager’s hash table was

protected by a single mutex. However, the lock manager code included support for a mutex per bucket, statically disabled by a single `#define`. In addition, we modified several critical sections in order to shorten them and avoid acquiring nested locks. The last optimization in the lock manager dealt with the lock request pool. The lock manager maintains a pool of pre-allocated lock requests, which it populates and inserts into lock lists as needed; the pool's mutex became a contention point, so we reimplemented it as a lock-free stack where threads can push or pop requests using a single compare-and-swap operation.

Principles applied:

- use the right synchronization primitive
- shorten or remove critical sections
- eliminate hotspots

7.6 Bufferpool Manager Revisited

After fixing the log and lock managers, the bufferpool manager again became a bottleneck. This time mutex protecting the clock replacement algorithm, and the mutex protecting “in-transit” pages became contended. On every page miss, the bufferpool “clock” sweeps through the pages searching for suitable candidates to evict. For efficiency, the clock hand does not pin or latch pages unless they appear promising; we modified the algorithm slightly so it could release the clock hand mutex before pinning and latching the victim; if another thread pinned the page first (making it non-evictable), the clock simply resumes its search.

Once a page has been selected for eviction, it enters an in-transit list while the old page is flushed to disk (if dirty) and the new one read in. This list can become quite large, especially given the random-accesses common to transaction processing. We applied the optimization detailed in Section 6.2.3 in order to shorten the list and distribute it into many smaller lists.

Finally, we added another cache in the free space manager to bypass an $O(n^2)$ algorithm in page allocation routine: searching a linked list of pages to find the last. These changes brought a significant boost in scalability, as indicated by the “bpool 2” line of the figure.

Principles applied:

- shorten or remove critical sections
- eliminate hotspots

7.7 Final Optimizations

The final optimizations to Shore-MT were spread throughout the code base. We created several more caches inside the free space manager to avoid expensive critical sections, including the one described in Section 6.2.2. We also removed an unnecessary search of the lock table initiated by B+Tree probes. The most important optimization involved log checkpoint generation. The soft checkpointing algorithm in Shore builds a list of active transactions and dirty pages contained in the buffer pool at the time of the checkpoint. Unfortunately, traversing a large bufferpool takes a single thread several seconds due to page faults and cache misses; during that time no transaction can begin or complete. We observed that recovery rebuilds the list of dirty pages using information from the log. The only important piece of information is the oldest log sequence number found during the traversal, which

lets the system skip large portions of the log during recovery. We modified the dirty page cleaner threads — which already traverse whole bufferpool, but asynchronously — to track the newest LSN they encounter during each sweep; because they write out the dirty pages they encounter, at the end of a sweep the “newest” LSN is now the oldest, perfect for inclusion in the log checkpoint. We therefore modified the checkpoint thread to simply read this value rather than computing it during the critical section.

The “final” line in the graph shows the performance of the completed Shore-MT. The profiler reported no more significant bottlenecks for this benchmark, which we verified by running multiple copies of Shore-MT in parallel; scalability was the same in both cases, indicating that it utilizes fully the hardware.

Principles applied:

- shorten or remove critical sections
- eliminate hotspots

7.8 Lessons Learned

This section and the previous one show clearly how a few fundamental principles, applied systematically, can transform a storage engine from virtually single-threaded to fully scalable. As we explained in the beginning of Section 6, we expect that designers can apply these principles to any storage manager to improve its scalability in similar fashion. Though we distilled these principles and lessons from our experience with Shore-MT, these results generalize to other storage engines for the following reasons:

1. Shore uses state of the art algorithms and behaves like a commercial dbms at the microarchitectural level. This suggests that Shore is a typical case does roughly the same kinds of things as the commercial application.
2. Shore started out with the worst scalability of the open source engines, indicating that other engines which scale poorly can be improved using with a similar approach.
3. Shore ended up scaling as well as, or better than, the commercial engine. This indicates that the implementation of the different components, rather than a fundamental design flaw that might be specific to Shore.
4. Our optimizations were guided by a small set of very generic approaches for improving scalability. These approaches are therefore likely to be effective in other storage managers as well as other domains.

Finally, we note that achieving scalability is not a one-time event. Fixing one bottleneck can expose others, and adding more hardware contexts to the system will eventually cause contention for some critical section. With the number of hardware contexts doubling each processor generation it is never safe to assume a critical section or algorithm will not become a bottleneck.

8. CONCLUSION

As we enter the multicore era, database storage managers must provide scalability in order to achieve the high performance users demand. Though modern open source storage managers are not currently up to the task, our experience converting Shore to the scalable Shore-MT suggests that much progress is possible. With careful benchmarking and analysis, we identify bottlenecks that inhibit scalability; repairing them creates Shore-MT, a multi-

threaded storage manager which exhibits excellent scalability and superior performance when compared to both its peers as well as to a popular commercial database system. Most importantly, the lessons we identified provide a useful reference as new bottlenecks appear in the future.

9. ACKNOWLEDGEMENTS

The authors thank the Database Research Group at Columbia University for providing access to their T2000 Server on short notice following a hardware failure. We also thank the anonymous reviewers for their helpful comments. This work was partially supported by grants and equipment from Intel; a Sloan research fellowship; an IBM faculty partnership award; NSF grants CCR-0205544, CCR-0509356, IIS-0133686, and IIS-0713409; and an ESF EurYI grant.

10. REFERENCES

- [1] Oracle BerkeleyDB. <http://www.oracle.com/technology/products/berkeley-db/index.html>
- [2] MySQL. <http://www.mysql.com>.
- [3] A. Ailamaki, D. J. DeWitt, and M. D. Hill. "Walking Four Machines By The Shore." In Proc. CAECW, 2001.
- [4] R. Bamford, D. Butler, B. Klots, and N. MacNaughton. "Architecture of Oracle Parallel Server." In Proc. VLDB, 1998.
- [5] P. A. Bernstein, and N. Goodman. "Multiversion Concurrency Control - Theory and Algorithms." ACM TODS, 8(4), 1983.
- [6] E. Bugnion, S. Devine, and M. Rosenblum. "Disco: running commodity operating systems on scalable multiprocessors." In Proc. SOSP, 1997.
- [7] M. Carey, D. J. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. K. Tan, O. Tsatalos, S. White, M. Zwilling. "Shoring up persistent applications." In Proc. SIGMOD, 1994.
- [8] H. T. Chou, and D. J. DeWitt. "An Evaluation of Buffer Management Strategies for Relational Database Systems". In Proc. VLDB, 1985.
- [9] T. Craig. "Building FIFO and priority-queueing spin locks from atomic swap." Technical Report TR 93-02-02, University of Washington, Dept. of Computer Science, 1993.
- [10] J. D. Davis, J. Laudon and K. Olukotun. "Maximizing CMP Throughput with Mediocre Cores". In Proc. PACT, 2005.
- [11] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Voshall, and W. Vogels. "Dynamo: Amazon's highly available key-value store." In Proc. SOSP, 2007.
- [12] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. Hsiao, and R. Rasmussen. "The Gamma Database Machine Project." IEEE TKDE 2(1), 1990.
- [13] D. J. DeWitt, and J. Gray. "Parallel Database Systems: The Future of High Performance Database Systems." Commun. ACM, 35(6), 1992.
- [14] D. Fotakis, R. Pagh, P. Sanders, and P. G. Spirakis. "Space efficient hash tables with worst case constant access time." In STACS, 2003.
- [15] J. Gray. "Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King." Gong Show Presentation at CIDR, 2007.
- [16] J. Gray, and A. Reuter. "Transaction Processing: Concepts and Techniques." Morgan Kaufmann Publishers, Inc., 1993.
- [17] B. He, W. N. Scherer III, and M. L. Scott. "Preemption Adaptivity in Time-Published Queue-Based Spin Locks." In Proc. HiPC, 2005.
- [18] P. Helland. "Life beyond Distributed Transactions: an Apostate's Opinion." In Proc. CIDR, 2007.
- [19] J. M. Hellerstein, and M. Stonebraker. "Anatomy of a Database System." In Readings in Database Systems, 4th ed.
- [20] R. Johnson, I. Pandis, A. Ailamaki. "Critical sections: re-emerging scalability concerns for database storage engines." In Proc. DaMoN, 2008.
- [21] P. Kongetira, K. Aingaran, and K. Olukotun. "Niagara: A 32-Way Multithreaded SPARC Processor". IEEE MICRO, 2005.
- [22] P. Lehman, and B. Yao. "Efficient locking for concurrent operations on B-trees." ACM TODS, 6(4), 1981.
- [23] P. Magnussen, A. Landin, and E. Hagersten. "Queue locks on cache coherent multiprocessors." In Proc. IPPS, 1994.
- [24] J. Mellor-Crummey, and M. Scot. "Algorithms for scalable synchronization on shared-memory multiprocessors." ACM TOCS, 9(1), 1991.
- [25] C. Mohan, "ARIES/KVL: a key-value locking method for concurrency control of multi-action transactions operating on B-tree indexes." In Proc. VLDB, 1990.
- [26] C. Mohan, and F. Levine. "ARIES/IM: an efficient and high concurrency index management method using write-ahead logging". In Proc. SIGMOD, 1992.
- [27] R. Pagh, and F. F. Rodler. "Cuckoo Hashing". In Proc. ESA, 2001.
- [28] P. Ranganathan, K. Gharachorloo, S. V. Adve, and L. A. Barroso. "Performance of Database Workloads on Shared-Memory Systems with Out-of-Order Processors." In Proc. ASPLOS, 1998.
- [29] Y. Ruan, V. Pai, E. Nahum, J. Tracey. Evaluating the impact of simultaneous multithreading on network servers using real hardware. ACM SIGMETRICS, 33(1), 2005.
- [30] M. Stonebraker, and L. A. Rowe. "The Design of Postgres." In Proc. SIGMOD, 1986.
- [31] M. Stonebraker, S. Madden, D. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. "The End of an Architectural Era (It's Time for a Complete Rewrite)." In Proc. VLDB, 2007.
- [32] M. Thorup. "Even strongly universal hashing is pretty fast." In Proc. SODA, 2000.
- [33] TPC (Transaction Processing Performance Council). TPC Benchmark C (OLTP) Standard Specification, Revision 5.9.