OLTP Through the Looking Glass, and What We Found There

Stavros Harizopoulos
HP Labs
Palo Alto, CA
stavros@hp.com

Daniel J. Abadi
Yale University
New Haven, CT
dna@cs.yale.edu

Samuel Madden Michael Stonebraker
Massachusetts Institute of Technology
Cambridge, MA
{madden, stonebraker}@csail.mit.edu

ABSTRACT

Online Transaction Processing (OLTP) databases include a suite of features — disk-resident B-trees and heap files, locking-based concurrency control, support for multi-threading — that were optimized for computer technology of the late 1970's. Advances in modern processors, memories, and networks mean that today's computers are vastly different from those of 30 years ago, such that many OLTP databases will now fit in main memory, and most OLTP transactions can be processed in milliseconds or less. Yet database architecture has changed little.

Based on this observation, we look at some interesting variants of conventional database systems that one might build that exploit recent hardware trends, and speculate on their performance through a detailed instruction-level breakdown of the major components involved in a transaction processing database system (Shore) running a subset of TPC-C. Rather than simply profiling Shore, we progressively modified it so that after every feature removal or optimization, we had a (faster) working system that fully ran our workload. Overall, we identify overheads and optimizations that explain a total difference of about a factor of 20x in raw performance. We also show that there is no single "high pole in the tent" in modern (memory resident) database systems, but that substantial time is spent in logging, latching, locking, B-tree, and buffer management operations.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems — *transaction processing; concurrency.*

General Terms

Measurement, Performance, Experimentation.

Keywords

Online Transaction Processing, OLTP, main memory transaction processing, DBMS architecture.

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

1. INTRODUCTION

Modern general purpose online transaction processing (OLTP) database systems include a standard suite of features: a collection of on-disk data structures for table storage, including heap files and B-trees, support for multiple concurrent queries via locking-based concurrency control, log-based recovery, and an efficient buffer manager. These features were developed to support transaction processing in the 1970's and 1980's, when an OLTP database was many times larger than the main memory, and when the computers that ran these databases cost hundreds of thousands to millions of dollars.

Today, the situation is quite different. First, modern processors are very fast, such that the computation time for many OLTP-style transactions is measured in microseconds. For a few thousand dollars, a system with gigabytes of main memory can be purchased. Furthermore, it is not uncommon for institutions to own networked clusters of many such workstations, with aggregate memory measured in hundreds of gigabytes — sufficient to keep many OLTP databases in RAM.

Second, the rise of the Internet, as well as the variety of data intensive applications in use in a number of domains, has led to a rising interest in database-like applications without the full suite of standard database features. Operating systems and networking conferences are now full of proposals for "database-like" storage systems with varying forms of consistency, reliability, concurrency, replication, and queryability [DG04, CDG+06, GBH+00, SMK+01].

This rising demand for database-like services, coupled with dramatic performance improvements and cost reduction in hardware, suggests a number of interesting alternative systems that one might build with a different set of features than those provided by standard OLTP engines.

1.1 Alternative DBMS Architectures

Obviously, optimizing OLTP systems for main memory is a good idea when a database fits in RAM. But a number of other database variants are possible; for example:

- **Logless databases**. A log-free database system might either not need recovery, or might perform recovery from other sites in a cluster (as was proposed in systems like Harp [LGG+91], Harbor [LM06], and C-Store [SAB+05]).
- **Single threaded databases**. Since multi-threading in OLTP databases was traditionally important for latency hiding in the

face of slow disk writes, it is much less important in a memory resident system. A single-threaded implementation may be sufficient in some cases, particularly if it provides good performance. Though a way to take advantage of multiple processor cores on the same hardware is needed, recent advances in virtual machine technology provide a way to make these cores look like distinct processing nodes without imposing massive performance overheads [BDR97], which may make such designs feasible.

- **Transaction-less databases**. Transactional support is not needed in many systems. In particular, in distributed Internet applications, eventual consistency is often favored over transactional consistency [Bre00, DHJ+07]. In other cases, lightweight forms of transactions, for example, where all reads are required to be done before any writes, may be acceptable [AMS+07, SMA+07].

In fact, there have been several proposals from inside the database community to build database systems with some or all of the above characteristics [WSA97, SMA+07]. An open question, however, is how well these different configurations would perform if they were actually built. This is the central question of this paper.

1.2 Measuring the Overheads of OLTP

To understand this question, we took a modern open source database system (Shore — see http://www.cs.wisc.edu/shore/) and benchmarked it on a subset of the TPC-C benchmark. Our initial implementation — running on a modern desktop machine — ran about 640 transactions per second (TPS). We then modified it by removing different features from the engine one at a time, producing new benchmarks each step of the way, until we were left with a tiny kernel of query processing code that could process 12700 TPS. This kernel is a single-threaded, lock-free, main memory database system without recovery. During this decomposition, we identified four major components whose removal substantially improved the throughput of the system:

Logging. Assembling log records and tracking down all changes in database structures slows performance. Logging may not be necessary if recoverability is not a requirement or if recoverability is provided through other means (e.g., other sites on the network).

Locking. Traditional two-phase locking poses a sizeable overhead since all accesses to database structures are governed by a separate entity, the Lock Manager.

Latching. In a multi-threaded database, many data structures have to be latched before they can be accessed. Removing this feature and going to a single-threaded approach has a noticeable performance impact.

Buffer management. A main memory database system does not need to access pages through a buffer pool, eliminating a level of indirection on every record access.

1.3 Results

Figure 1 shows how each of these modifications affected the bottom line performance (in terms of CPU instructions per TPC-C New Order transaction) of Shore. We can see that each of these

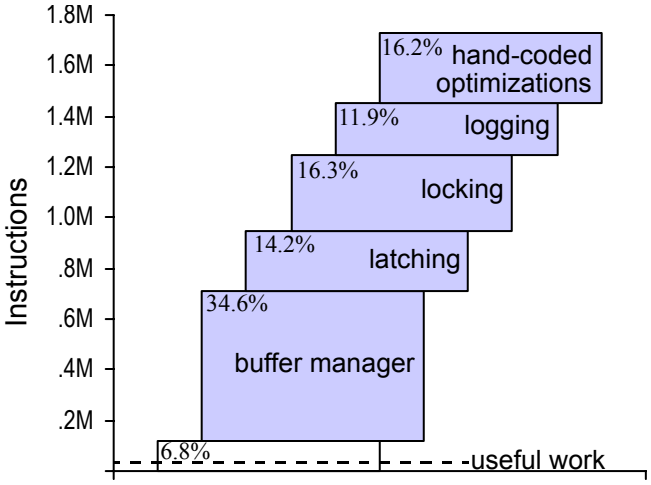

Figure 1. Breakdown of instruction count for various DBMS components for the New Order transaction from TPC-C. The top of the bar-graph is the original Shore performance with a main memory resident database and no thread contention. The bottom dashed line is the useful work, measured by executing the transaction on a no-overhead kernel.

subsystems by itself accounts for between about 10% and 35% of the total runtime (1.73 million instructions, represented by the total height of the figure). Here, "hand coded optimizations" represents a collection of optimizations we made to the code, which primarily improved the performance of the B-tree package. The actual instructions to process the query, labelled "useful work" (measured through a minimal implementation we built on top of a hand-coded main-memory B-tree package) is only about 1/60th of that. The white box below "buffer manager" represents our version of Shore after we had removed everything from it — Shore still runs the transactions, but it uses about 1/15th of the instructions of the original system, or about 4 times the number of instructions in the useful work. The additional overheads in our implementation are due to call-stack depth in Shore and the fact that we could not completely strip out all references to transactions and the buffer manager.

1.4 Contributions and Paper Organization

The major contributions of this paper are to 1) dissect where time goes inside of a modern database system, 2) to carefully measure the performance of various stripped down variants of a modern database system, and 3) to use these measurements to speculate on the performance of different data management systems — for example, systems without transactions or logs — that one could build.

The remainder of this paper is organized as follows. In Section 2 we discuss OLTP features that may soon become (or are already becoming) obsolete. In Section 3 we review the Shore DBMS, as it was the starting point of our exploration, and describe the decomposition we performed. Section 4 contains our experimentation with Shore. Then, in Section 5, we use our measurements to discuss implications on future OLTP engines and speculate on the performance of some hypothetical data management systems. We present additional related work in Section 6 and conclude in Section 7.

2. TRENDS IN OLTP

As mentioned in the introduction, most popular relational RDBMSs trace their roots to systems developed in the 1970's, and include features like disk-based indexing and heap files, log-based transactions, and locking-based concurrency control. However, 30 years have passed since these architectural decisions were made. At the present time, the computing world is quite different from when these traditional systems were designed; the purpose of this section is to explore the impact of these differences. We made a similar set of observations in [SMA+07].

2.1 Cluster Computing

Most current generation RDBMSs were originally written for shared memory multi-processors in the 1970's. Many vendors added support for shared disk architectures in the 1980's. The last two decades have seen the advent of Gamma-style shared nothing databases [DGS+90] and the rise of clusters of commodity PCs for many large scale computing tasks. Any future database system must be designed from the ground up to run on such clusters.

2.2 Memory Resident Databases

Given the dramatic increase in RAM sizes over the past several decades, there is every reason to believe that many OLTP systems already fit or will soon fit into main memory, especially the aggregate main memory of a large cluster. This is largely because the sizes of most OTLP systems are not growing as dramatically as RAM capacity, as the number of customers, products, and other real world entities they record information about does not scale with Moore's law. Given this observation, it makes sense for database vendors to create systems that optimize for the common case of a memory resident system. In such systems, optimized indices [RR99, RR00] as well as eschewing disk-optimized tuple formats and page layouts (or lack thereof) [GS92] are important to consider.

2.3 Single Threading in OLTP Systems

All modern databases include extensive support for multi-threading, including a collection of transactional concurrency control protocols as well as extensive infiltration of their code with latching commands to support multiple threads accessing shared structures like buffer pools and index pages. The traditional motivations for multi-threading are to allow transaction processing to occur on behalf of one transaction while another waits for data to come from disk, and to prevent long-running transactions from keeping short transactions from making progress.

We claim that neither of these motivations is valid any more. First, if databases are memory resident, then there are never any disk waits. Furthermore, production transaction systems do not include any user waits — transactions are executed almost exclusively through stored procedures. Second, OLTP workloads are very simple. A typical transaction consists of a few index lookups and updates, which, in a memory resident system, can be completed in hundreds of microseconds. Moreover, with the bifurcation of the modern database industry into a transaction processing and a warehousing market, long running (analytical) queries are now serviced by warehouses.

One concern is that multi-threading is needed to support machines with multiple processors. We believe, however, that this can be addressed by treating one physical node with multiple processors as multiple nodes in a shared-nothing cluster, perhaps managed by a virtual machine monitor that dynamically allocates resources between these logical nodes [BDR97].

Another concern is that networks will become the new disks, introducing latency into distributed transactions and requiring the re-introduction of transactions. This is certainly true in the general case, but for many transaction applications, it is possible to partition the workload to be "single-sited" [Hel07, SMA+07], such that all transactions can be run entirely on a single node in a cluster.

Hence, certain classes of database applications will not need support for multi-threading; in such systems, legacy locking and latching code becomes unnecessary overhead.

2.4 High Availability vs. Logging

Production transaction processing systems require 24x7 availability. For this reason, most systems use some form of high availability, essentially using two (or more) times the hardware to ensure that there is an available standby in the event of a failure.

Recent papers [LM06] have shown that, at least for warehouse systems, it is possible to exploit these available standbys to facilitate recovery. In particular, rather than using a REDO log, recovery can be accomplished by copying missing state from other database replicas. In our previous work we have claimed that this can be done for transaction systems as well [SMA+07]. If this is in fact the case, then the recovery code in legacy databases becomes also unnecessary overhead.

2.5 Transaction Variants

Although many OLTP systems clearly require transactional semantics, there have recently been proposals — particularly in the Internet domain — for data management systems with relaxed consistency. Typically, what is desired is some form of eventual consistency [Bre00, DHJ+07] in the belief that availability is more important than transactional semantics. Databases for such environments are likely to need little of the machinery developed for transactions (e.g., logs, locks, two-phase commit, etc.).

Even if one requires some form of strict consistency, many slightly relaxed models are possible. For example, the widespread adoption of snapshot isolation (which is non-transactional) suggests that many users are willing to trade transactional semantics for performance (in this case, due to the elimination of read locks).

And finally, recent research has shown that there are limited forms of transactions that require substantially less machinery than standard database transactions. For example, if all transactions are "two-phase" — that is, they perform all of their reads before any of their writes and are guaranteed not to abort after completing their reads — then UNDO logging is not necessary [AMS+07, SMA+07].

2.6 Summary

As our references suggest, several research groups, including Amazon [DHJ+07], HP [AMS+07], NYU [WSA97], and MIT [SMA+07] have demonstrated interest in building systems that differ substantially from the classic OTLP design. In particular, the MIT H-Store [SMA+07] system demonstrates that removing all of the above features can yield a two-order-of-magnitude

speedup in transaction throughput, suggesting that some of these databases variants are likely to provide remarkable performance. Hence, it would seem to behoove the traditional database vendors to consider producing products with some of these features explicitly disabled. With the goal of helping these implementers understand the performance impact of different variants they may consider building, we proceed with our detailed performance study of Shore and the variants of it we created.

3. SHORE

Shore (Scalable Heterogeneous Object Repository) was developed at the University of Wisconsin in the early 1990's and was designed to be a typed, persistent object system borrowing from both file system and object-oriented database technologies [CDF+94]. It had a layered architecture that allowed users to choose the appropriate level of support for their application from several components. These layers (type system, unix compatibility, language heterogeneity) were provided on top of the Shore Storage Manager (SSM). The storage manager provided features that are found in all modern DBMS: full concurrency control and recovery (ACID transaction properties) with two-phase locking and write-ahead logging, along with a robust implementation of B-trees. Its basic design comes from ideas described in Gray's and Reuter's seminal book on transaction processing [GR93], with many algorithms implemented straight from the ARIES papers [MHL+92, Moh89, ML89].

Support for the project ended in the late 1990's, but continued for the Shore Storage Manager; as of 2007, SSM version 5.0 is available for Linux on Intel x86 processors. Throughout the paper we use "Shore" to refer to the Shore Storage Manager. Information and source code of Shore is available online[1]. In the rest of this section we discuss the key components of Shore, its code structure, the characteristics of Shore that affect end-to-end performance, along with our set of modifications and the effect of these modifications to the code line.

3.1 Shore Architecture

There are several features of Shore that we do not describe as they are not relevant to this paper. These include disk volume management (we pre-load the entire database in main memory), recovery (we do not examine application crashes), distributed transactions, and access methods other than B-trees (such as R-trees). The remaining features can be organized roughly into the components shown in Figure 2.

Shore is provided as a library; the user code (in our case, the implementation of the TPC-C benchmark) is linked against the library and must use the threads library that Shore also uses. Each transaction runs inside a Shore thread, accessing both local user-space variables and Shore-provided data structures and methods. The methods relevant to OLTP are those needed to create and populate a database file, load it into the buffer pool, begin, commit, or abort a transaction, and perform record-level operations such as fetch, update, create, and delete, along with the associated operations on primary and secondary B-tree indexes.

1. http:// www.cs.wisc.edu/shore/

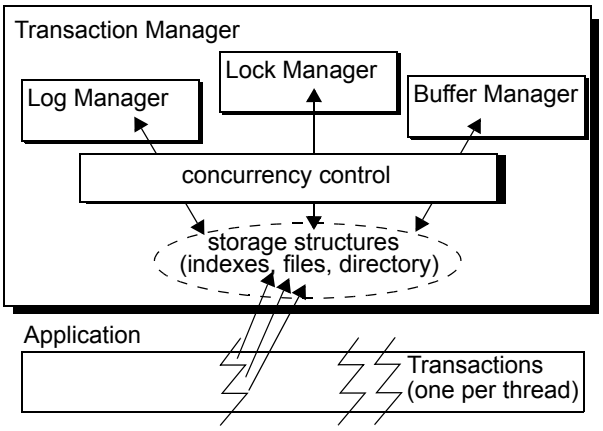

Figure 2. Basic components in Shore (see text for detailed description).

Inside the transaction body (enclosed by begin and commit statements) the application programmer uses Shore's methods to access the storage structures: the file and indexes, along with a directory to find them. All the storage structures use *slotted pages* to store information. Shore's methods run under the transaction manager which closely interacts with all other components. Accessing the storage structures involves calls to the Log Manager, the Lock Manager, and the Buffer Pool Manager. These invocations always happen through a concurrency control layer, which oversees shared and mutually exclusive accesses to the various resources. This is not a separate module; rather, throughout the code, all accesses to shared structures happen by acquiring a *latch*. Latches are similar to database locks (in that they can be shared or exclusive), but they are lightweight and come with no deadlock detection mechanisms. The application programmers need to ensure that latching will not lead to deadlock.

Next, we discuss the thread architecture and give more details on locking, logging, and the buffer pool management.

Thread support. Shore provides its own user-level, non-preemptive thread package that was derived from NewThreads (originally developed at the University of Washington), providing a portable OS interface API. The choice of the thread package had implications for the code design and behavior of Shore. Since threads are user-level, the application runs as a single process, multiplexing all Shore threads. Shore avoids blocking for I/O by spawning separate processes responsible for I/O devices (all processes communicate through shared memory). However, applications cannot take direct advantage of multicore (or SMP) systems, unless they are built as part of a distributed application; that, however, would add unnecessary overhead for multicore CPUs, when simple, non-user level threading would be sufficient.

Consequently, for the results reported throughout this paper, we use single-threaded operation. A system that uses multithreaded operation would consume a larger number of instructions and CPU cycles per transaction (since thread code would need to be executed in addition to transactional code). Since the primary goal of the paper is to look at the cost in CPU instructions of various database system components, the lack of a full multi-threading implementation in Shore only affects our results in that we

Table 1: Possible set of optimizations for OLTP

OLTP properties and new platforms	DBMS modification
logless architectures	remove logging
partitioning, commutativity	remove locking (when applicable)
one transaction at a time	single thread, remove locking, remove latching
main memory resident	remove buffer manager, directory
transaction-less databases	avoid transaction bookkeeping

begin at a lower starting point in total CPU instructions when we begin removing system components.

Locking and logging. Shore implements standard two-phase locking, with transactions having standard ACID properties. It supports hierarchical locking with the lock manager escalating up the hierarchy by default (record, page, store, volume). Each transaction keeps a list of the locks it holds, so that the locks can be logged when the transaction enters the prepared state and released at the end of the transaction. Shore also implements write ahead logging (WAL), which requires a close interaction between the log manager and the buffer manager. Before a page can be flushed from the buffer pool, the corresponding log record might have to be flushed. This also requires a close interaction between the transaction manager and the log manager. All three managers understand log sequence numbers (LSNs), which serve to identify and locate log records in the log, timestamp pages, identify the last update performed by a transaction, and find the last log record written by a transaction. Each page bears the LSN of the last update that affected that page. A page cannot be written to disk until the log record with that page's LSN has been written to stable storage.

Buffer Manager. The buffer manager is the means by which all other modules (except the log manager) read and write pages. A page is read by issuing a *fix* method call to the buffer manager. For a database that fits in main memory, the page is always found in the buffer pool (in the non-main memory case, if the requested page is not in the buffer pool, the thread gives up the CPU and waits for the process responsible for I/O to place the page in the buffer pool). The fix method updates the mapping between page IDs and buffer frames and usage statistics. To ensure consistency there is a latch to control access to the fix method. Reading a record (once a record ID has been found through an index lookup) involves

1. locking the record (and page, per hierarchical locking),

2. fixing the page in the buffer pool, and

3. computing the offset within the page of the record's tag.

Reading a record is performed by issuing a pin / unpin method call. Updates to records are accomplished by copying out part or all of the record from the buffer pool to the user's address space, performing the update there, and handing the new data to the storage manager.

More details on the architecture of Shore can be found at the project's web site. Some additional mechanisms and features are also described in the following paragraphs, where we discuss our own modifications to Shore.

3.2 Removing Shore Components

Table 1 summarizes the properties and characteristics of modern OLTP systems (left column) that allow us to strip certain functionality from a DBMS (right column). We use these optimizations as a guideline for modifying Shore. Due to the tight integration of all managers in Shore, it was not possible to cleanly separate all components so that they could be removed in an arbitrary order. The next best thing was to remove features in an order dictated by the structure of the code, allowing for flexibility whenever possible. That order was the following:

1. Removing logging.

2. Removing locking OR latching.

3. Removing latching OR locking.

4. Removing code related to the buffer manager.

In addition, we found that the following optimizations could be performed at any point:

- Streamline and hardcode the B-tree key evaluation logic, as is presently done in most commercial systems.
- Accelerate directory lookups.
- Increase page size to avoid frequent allocations (subsumed by step 4 above).
- Remove transactional sessions (begin, commit, various checks).

Our approach to implementing the above-mentioned actions is described next. In general, to remove a certain component from the system, we either add a few if-statements to avoid executing code belonging to that component, or, if we find that if-statements add a measurable overhead, we rewrite entire methods to avoid invoking that component altogether.

Remove logging. Removing logging consists of three steps. The first is to avoid generating I/O requests along with the time associated to perform these requests (later, in Figure 7, we label this modification "*disk log*"). We achieve this by allowing group commit and then increasing the log buffer size so that it is not flushed to disk during our experiments. Then, we comment out all functions that are used to prepare and write log records (labeled "*main log*" in Figure 7). The last step was to add if-statements throughout the code to avoid processing Log Sequence Numbers (labeled "*LSN*" in Figure 7).

Remove locking (interchangeable with removing latching). In our experiments we found that we could safely interchange the order of removing locking and latching (once logging was already removed). Since latching is also performed inside locking, removing one also reduces the overhead of the other. To remove locking we first changed all Lock Manager methods to return immediately, as if the lock request was successful and all checks for locks were satisfied. Then, we modified methods related to pin-

ning records, looking them up in a directory, and accessing them through a B-tree index. In each case, we eliminated code paths related to ungranted lock requests.

Remove latching (interchangeable with removing locking). Removing latching was similar to removing locking; we first changed all mutex requests to be immediately satisfied. We then added if-statements throughout the code to avoid requests for latches. We had to replace B-tree methods with ones that did not use latches, since adding if-statements would have increased overhead significantly because of the tight integration of latch code in the B-tree methods.

Remove buffer manager calls. The most widespread modification we performed was to remove the buffer manager methods, once we knew that logging, locking, and latching were already disabled. To create new records, we abandoned Shore's page allocation mechanism and instead used the standard malloc library. We call malloc for each new record (records no longer reside in pages) and use pointers for future accesses. Memory allocation can potentially be done more efficiently, especially when one knows in advance the sizes of the allocated objects. However, further optimization of main memory allocation is an incremental improvement relative to the overheads we are studying, and is left for future work. We were not able to completely remove the page interface to buffer frames, since its removal would invalidate most of the remaining Shore code. Instead, we accelerated the mappings between pages and buffer frames, reducing the overhead to a minimum. Similarly, pinning and updating a record will still go through a buffer manager layer, albeit a very thin one (we label this set of modifications "*page access*" in Figure 7).

Miscellaneous optimizations. There were four optimizations we made that can be invoked at any point during the process of removing the above-mentioned components. These were the following. (1) Accelerating the B-tree code by hand-coding node searches to optimize for the common case that keys are uncompressed integers (labeled "*Btree keys*" in Figures 5-8). (2) Accelerating directory lookups by using a single cache for all transactions (labeled "*dir lookup*" in Figure 7). (3) Increasing the page size from the default size of 8KB to 32KB, the maximum allowable in Shore (labeled "*small page*" in Figure 7). Larger pages, although not suitable for disk-based OLTP, can help in a main-memory resident database by reducing the number of levels in a B-tree (due to the larger node size), and result in less frequent page allocations for newly created records. An alternative would be to decrease the size of a B-tree node to the size of a cache line as proposed in [RR99], but this would have required removing the association between a B-tree node and a Shore page, or reducing a Shore page below 1KB (which Shore does not allow). (4) Removing the overhead of setting up and terminating a session for each transaction, along with the associated monitoring of running transactions, by consolidating transactions into a single session (labeled "*Xactions*" in Figure 7).

Our full set of changes/optimizations to Shore, along with the benchmark suite and documentation on how to run the experiments are available online[2].

2. http://db.cs.yale.edu/hstore/

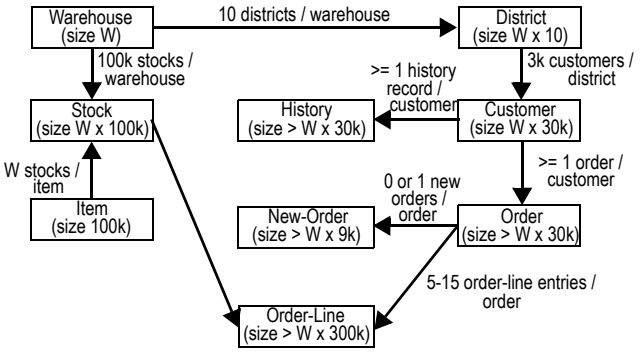

Figure 3. TPC-C Schema.

Next, we move to the performance section of the paper.

4. PERFORMANCE STUDY

The section is organized as follows. First we describe our variant of the TPC-C benchmark that we used (Section 4.1). In Section 4.2 we provide details of the hardware platform, the experimental setup, and the tools we used for collecting the performance numbers. Section 4.3 presents a series of results, detailing Shore performance as we progressively apply optimizations and remove components.

4.1 OLTP Workload

Our benchmark is derived from TPC-C [TPCC], which models a wholesale parts supplier operating out of a number of warehouses and their associated sales districts. TPC-C is designed to represent any industry that must manage, sell, or distribute a product or service. It is designed to scale as the supplier expands and new warehouses are created. The scaling requirement is that each warehouse must supply 10 sales districts, and each district must serve 3000 customers. The database schema along with the scaling requirements (as a function of the number of warehouses W) is shown in Figure 3. The database size for one warehouse is approximately 100 MB (we experiment with five warehouses for a total size of 500MB).

TPC-C involves a mix of five concurrent transactions of different types and complexity. These transactions include entering orders (the New Order transaction), recording payments (Payment), delivering orders, checking the status of orders, and monitoring the level of stock at the warehouses. TPC-C also specifies that about 90% of the time the first two transactions are executed. For the purposes of the paper, and for better understanding the effect of our interventions, we experimented with a mix of only the first two transactions. Their code structure (calls to Shore) is shown in Figure 4. We made the following small changes to the original specifications, to achieve repeatability in the experiments:

New Order. Each New Order transaction places an order for 5-15 items, with 90% of all orders supplied in full by stocks from the customer's "home" warehouse (10% need to access stock belonging to a remote warehouse), and with 1% of the provided items being an invalid one (it is not found in the B-tree). To avoid variation in the results we set the number of items to 10, and always serve orders from a local warehouse. These two changes do not

New Order	Payment
begin	begin
for loop(10)	Btree lookup(D), pin
.....Btree lookup(I), pin	Btree lookup (W), pin
Btree lookup(D), pin	Btree lookup (C), pin
Btree lookup (W), pin	update rec (C)
Btree lookup (C), pin	update rec (D)
update rec (D)	update rec (W)
for loop (10)	create rec (H)
.....Btree lookup(S), pin	commit
.....update rec (S)	
.....create rec (O-L)	
.....insert Btree (O-L)	
create rec (O)	
insert Btree (O)	
create rec (N-O)	
insert Btree (N-O)	
insert Btree 2ndary(N-O)	
commit	

Figure 4. Calls to Shore's methods for New Order and Payment transactions.

affect the throughput. The code in Figure 4 shows the two-phase optimization mentioned in Section 2.5, which allows us to avoid aborting a transaction; we read all items at the beginning, and if we find an invalid one we abort without redoing changes in the database.

Payment. This is a lightweight transaction; it updates the customer's balance and warehouse/district sales fields, and generates a history record. Again, there is a choice of home and remote warehouse which we always set to the home one. Another randomly set input is whether a customer is looked up by name or ID, and we always use ID.

4.2 Setup and Measurement Methodology

All experiments are performed on a single-core Pentium 4 3.2GHz, with 1MB L2 cache, hyperthreading disabled, 1GB RAM, running Linux 2.6. We compiled with gcc version 3.4.4 and O2 optimizations. We use the standard linux utility iostat to monitor disk activity and verify in the main memory-resident experiments there is no generated disk traffic. In all experiments we pre-load the entire database into the main memory. Then we run a large number of transactions (40,000). Throughput is measured directly by dividing wall clock time by the number of completed transactions.

For detailed instruction and cycle counts we instrument the benchmark application code with calls to the PAPI library [MBD+99] http://icl.cs.utk.edu/papi/, which provides access to the CPU performance counters. Since we make a call to PAPI after every call to Shore, we have to compensate for the cost of PAPI calls when reporting the final numbers. These had an instruction count of 535-537 and were taking between 1350 and 1500 cycles in our machine. We measure each call to Shore for all 40,000 transactions and report the average numbers.

Most of the graphs reported in the paper are based on CPU instruction counts (as measured through the CPU performance counters) and not wall clock time. The reason is that instruction counts are representative of the total run-time code path length, and they are deterministic. Equal instruction counts among different components can of course result in different wall clock execution times (CPU cycles), because of different microarchitectural behavior (cache misses, TLB misses, etc.). In Section 4.3.4 we compare instruction counts to CPU cycles, illustrating the components where there is high micro-architectural efficiency that can be attributed to issues like few L2 cache misses and good instruction-level parallelism.

Cycle count, however, is susceptible to various parameters, ranging from CPU characteristics, such as cache size/associativity, branch predictors, TLB operation, to run-time variables such as concurrent processes. Therefore it should be treated as indicative of relative time breakdown. We do not expand on the issue of CPU cache performance in this paper, as our focus is to identify the set of DBMS components to remove that can produce up to two orders of magnitude better performance for certain classes of OLTP workloads. More information on the micro-architectural behavior of database workloads can be found elsewhere [Ail04].

Next, we begin the presentation of our results.

4.3 Experimental Results

In all experiments, our baseline Shore platform is a memory-resident database that is never flushed to disk (the only disk I/O that might be performed is from the Log Manager). There is only a single thread executing one transaction at a time. Masking I/O (in the case of disk-based logging) is not a concern as it only adds to overall response time and not to the instructions or cycles that the transaction has actually run.

We placed 11 different switches in Shore to allow us to remove functionality (or perform optimizations), which, during the presentation of the results, we organize into six components. For a list of the 11 switches (and the corresponding components) and the order we apply them, see Figure 7. These switches were described in more detail in Section 3.2 above. The last switch is to bypass Shore completely and run our own, minimal-overhead kernel, which we call "optimal" in our results. This kernel is basically a memory-resident, hand-built B-tree package with no additional transaction or query processing functionality.

4.3.1 Effect on Throughput

After all of these deletions and optimizations, Shore is left with a code residue, which is all CPU cycles since there is no I/O whatsoever; specifically, an average of about 80 microseconds per transaction (for a 50-50 mix of New Order and Payment transactions), or about 12,700 transactions per second.

In comparison, the useful work in our optimal system was about 22 microseconds per transaction, or about 46,500 transactions per second. The main causes of this difference are a deeper call stack depth in our kernel, and our inability to remove some of the transaction set up and buffer pool calls without breaking Shore. As a point of reference, "out of the box" Shore, with logging enabled but with the database cached in main memory, provides about 640 transactions per second (1.6 milliseconds per transaction), whereas Shore running in main memory, but without log flushing provides about 1,700 transactions per second, or about 588 microseconds per transaction. Hence, our modifications yield a factor of 20 improvement in overall throughput.

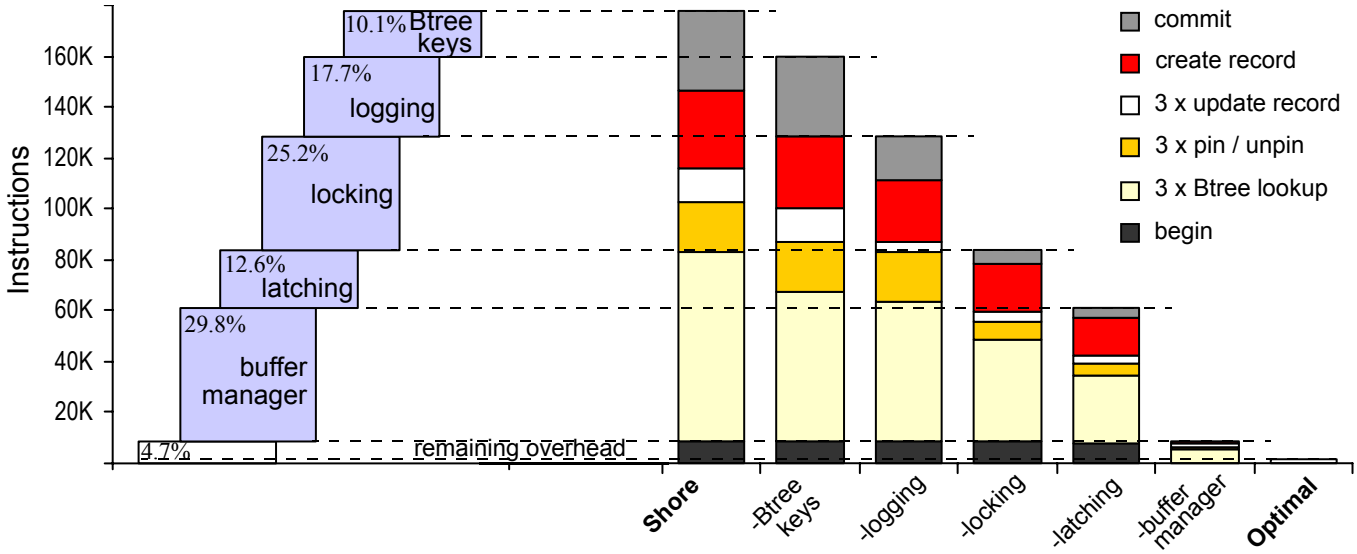

Figure 5. Detailed instruction count breakdown for Payment transaction.

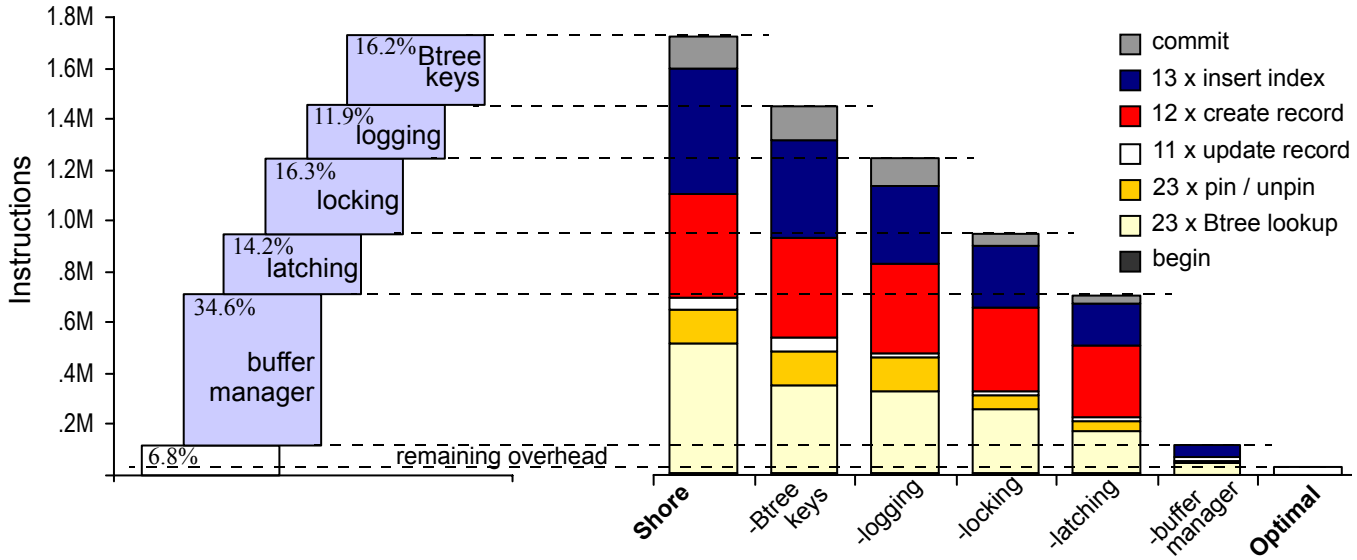

Figure 6. Detailed instruction count breakdown for New Order transaction.

Given these basic throughput measurements, we now give detailed instruction breakdowns for the two transactions of our benchmark. Recall that the instruction and cycle breakdowns in the following sections do not include any impact of disk operations, whereas the throughput numbers for baseline Shore do include some log write operations.

4.3.2 Payment

Figure 5 (left side) shows the reductions in the instruction count of the Payment transaction as we optimized B-tree key evaluations and removed logging, locking, latching, and buffer manager functionality. The right part of the figure shows, for each feature removal we perform, its effect on the number of instructions spent in various portions of the transaction's execution. For the Payment transaction, these portions include a begin call, three B-tree lookups followed by three pin/unpin operations, followed by three updates (through the B-tree), one record creation and a commit call. The height of each bar is always the total number of instructions executed. The right-most bar is the performance of our minimal-overhead kernel.

Our B-tree key evaluation optimizations are reportedly standard practice in high-performance DBMS architectures, so we perform them first because any system should be able to do this. Removing logging affects mainly commits and updates, as those are the portions of the code that write log records, and to a lesser degree B-tree and directory lookups. These modifications remove about 18% of the total instruction count.

Locking takes the second most instructions, accounting for about 25% of the total count. Removing it affects all of the code, but is especially important in the pin/unpin operations, the lookups, and commits, which was expected as these are the operations that must acquire or release locks (the transaction already has locks on the updated records when the updates are performed).

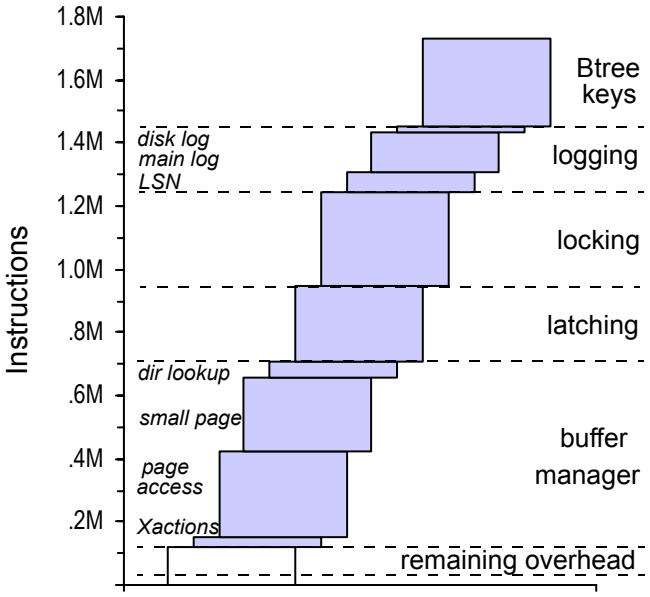

Figure 7. Expanding breakdown for New Order (see Section 3.2 for the labels on the left column).

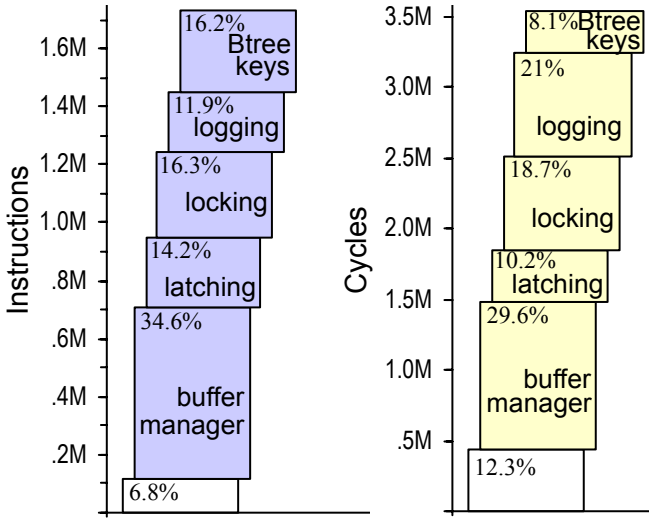

Figure 8. Instructions (left) vs. Cycles (right) for New Order.

Latching accounts for about 13% of the instructions, and is primarily important in the create record and B-tree lookup portions of the transaction. This is because the buffer pool (used in create) and B-trees are the primary shared data structures that must be protected with latches.

Finally, our buffer manager modifications account for about 30% of the total instruction count. Recall that with this set of modifications, new records are allocated directly with malloc, and page lookups no longer have to go through the buffer pool in most cases. This makes record allocation essentially free, and substantially improves the performance of other components that perform frequent lookups, like B-tree lookup and update.

At this point, the remaining kernel requires about 5% (for a 20x performance gain!) of the total initial instruction count, and is about 6 times the total instructions of our "optimal" system. This analysis leads to two observations: first, all six of the major components are significant, each accounting for 18 thousand or more instructions of the initial 180 thousand. Second, until all of our optimizations are applied, the reduction in instruction count is not dramatic: before our last step of removing the buffer manager, the remaining components used about a factor of three fewer instructions than the baseline system (versus a factor of 20 when the buffer manager is removed).

4.3.3 New Order

A similar breakdown of the instruction count in the New Order transaction is shown in Figure 6; Figure 7 shows a detailed accounting of all 11 modifications and optimizations we performed. This transaction uses about 10 times as many instructions as the Payment transaction, requiring 13 B-tree inserts, 12 record creation operations, 11 updates, 23 pin/unpin operations, and 23 B-tree lookups. The main differences in the allocation of instructions to major optimizations between New Order and Payment are in B-tree key code, logging, and locking. Since New Order adds B-tree insertions in the mix of operations, there is more relative benefit to be had by optimizing the key evaluation code (about 16%). Logging and locking now only account for about 12% and 16% of the total instructions; this is largely because the total fraction of time spent in operations where logging and locking perform a lot of work is much smaller in this case.

The buffer manager optimizations still represent the most significant win here, again because we are able to bypass the high overhead of record creation. Looking at the detailed breakdown in Figure 7 for the buffer manager optimization reveals something surprising: changing from 8K to 32K pages (labelled "small page") provides almost a 14% reduction in the total instruction count. This simple optimization — which serves to reduce the frequency of page allocations and decrease B-tree depth — offers a sizeable gain.

4.3.4 Instructions vs. Cycles

Having looked at the detailed breakdown of instruction counts in the Payment and New Order transactions, we now compare the fraction of time (cycles) spent in each phase of the New Order transaction to the fraction of instructions used in each phase. The results are shown in Figure 8. As we noted earlier, we do not expect these two fractions to be identical for a given phase, because cache misses and pipeline stalls (typically due to branches) can cause some instructions to take more cycles than others. For example, B-tree optimizations reduce cycles less than they reduce instructions, because the Shore B-tree code overhead we remove is mainly offset calculations with few cache misses. Conversely, our residual "kernel" uses a larger fraction of cycles than it does instructions, because it is branch-heavy, consisting mostly of function calls. Similarly, logging uses significantly more cycles because it touches a lot of memory creating and writing log records (disk I/O time is not included in either graph). Finally, locking and the buffer manager consume about the same percentage of cycles as they do instructions.

5. IMPLICATIONS FOR FUTURE OLTP ENGINES

Given the performance results in the previous section, we revisit our discussion of future OLTP designs from Section 2. Before going into the detailed implications of our results for the design of various database subsystems, we make two high level observations from our numbers:

- First, the benefit of stripping out any one of the components of the system has a relatively small benefit on overall performance. For example, our main memory optimizations improved the performance of Shore by about 30%, which is significant but unlikely to motivate the major database vendors to re-engineer their systems. Similar gains would be obtained by eliminating just latching or switching to a single-threaded, one-transaction-at-a-time approach.
- The most significant gains are to be had when multiple optimizations are applied. A fully stripped down system provides a factor of twenty or more performance gain over out-of-the-box Shore, which is truly significant. Note that such a system can still provide transactional semantics, if only one transaction is run at a time, all transactions are two phase, and recovery is implemented by copying state from other nodes in the network. Such a system is very, very different from what any of the vendors currently offers, however.

5.1 Concurrency Control

Our experiments showed a significant contribution (about 19% of cycles) of dynamic locking to total overhead. This suggests that there is a large gain to be had by identifying scenarios, such as application commutativity, or transaction-at-a-time processing, that allow concurrency control to be turned off. However, there are many DBMS applications which are not sufficiently well-behaved or where running only one transaction at a time per site will not work. In such cases, there is an interesting question as to what concurrency control protocol is best. Twenty years ago, various researchers [KR81, ACL87] performed exhaustive simulations that clearly showed the superiority of dynamic locking relative to other concurrency control techniques. However, this work assumed a disk-based load with disk stalls, which obviously impacts the results significantly. It would be highly desirable to redo these sorts of simulation studies with a main memory workload. We strongly suspect that some sort of optimistic concurrency control would be the prevailing option.

5.2 Multi-core Support

Given the increasing prevalence of many-core computers, an interesting question is how future OLTP engines should deal with multiple cores. One option is to run multiple transactions concurrently on separate cores within a single site (as it is done today); of course, such parallelism requires latching and implies a number of resource allocation issues. Our experiments show that although the performance overhead of latching is not particularly high (10% of cycles in the dominant transaction, New Order), it still remains an obstacle in achieving significant performance improvements in OLTP. As technologies (such as transactional memory [HM93]) for efficiently running highly concurrent programs on multicore machines mature and find their way into products, it will be very interesting to revisit new implementations for latching and reassess the overhead of multithreading in OLTP.

A second option is to use virtualization, either at the operating system or DBMS level, to make it appear that each core is a single-threaded machine. It is unclear what the performance implications of that approach would be, warranting a careful study of such virtualization. A third option, complementary to the other two, is to attempt to exploit intra-query parallelism, which may be feasible even if the system only runs one transaction at a time. However, the amount of intra-query parallelism available in a typical OLTP transaction is likely to be limited.

5.3 Replication Management

The traditional database wisdom is to support replication through a log-shipping based active-passive scheme; namely, every object has an "active" primary copy, to which all updates are first directed. The log of changes is then spooled over the network to one or more "passive" backup sites. Recovery logic rolls the remote database forward from the log. This scheme has several disadvantages. First, unless a form of two-phase commit is used, the remote copies are not transactionally consistent with the primary. Hence, reads cannot be directed to replicas if transaction-consistent reads are required. If reads are directed to replicas, nothing can be said about the accuracy of the answers. A second disadvantage is that failover is not instantaneous. Hence, the stall during failures is longer than it needs to be. Third, it requires the availability of a log; our experiments show that maintaining a log takes about 20% of total cycles. Hence, we believe it is interesting to consider alternatives to active-passive replication, such as an active-active approach.

The main reason that active-passive replication with log shipping has been used in the past is that the cost of rolling the log forward has been assumed to be far lower than the cost of performing the transaction logic on the replica. However, in a main memory DBMS, the cost of a transaction is typically less than 1 msec, requiring so few cycles that it is likely not much slower than playing back a log. In this case, an alternate active-active architecture appears to make sense. In this case, all replicas are "active" and the transaction is performed synchronously on all replicas. The advantage of this approach is nearly instantaneous failover and there is no requirement that updates be directed to a primary copy first. Of course, in such a scenario, two-phase commit will introduce substantial additional latency, suggesting that techniques to avoid it are needed — perhaps by performing transactions in timestamp order.

5.4 Weak Consistency

Most large web-oriented OLTP shops insist on replicas, usually over a WAN, to achieve high availability and disaster recovery. However, seemingly nobody is willing to pay for transactional consistency over a WAN. As noted in Section 2, the common refrain in web applications is "eventual consistency" [Bre00, DHJ+07]. Typically, proponents of such approach advocate resolving inconsistencies through non-technical means; for example, it is cheaper to give a credit to a customer who complains

than to ensure 100% consistency. In other words, the replicas eventually become consistent, presumably if the system is quiesced.

It should be clear that eventual consistency is impossible without transaction consistency under a general workload. For example, suppose transaction 1 commits at site 1 and aborts or is lost at site 2. Transaction 2 reads the result of transaction 1 and writes into the database, causing the inconsistency to propagate and pollute the system. That said, clearly, there must be workloads where eventual consistency is achievable, and it would be an interesting exercise to look for them, since, as noted above, our results suggest that removing transactional support — locking and logging — from a main memory system could yield a very high performance database.

5.5 Cache-conscious B-trees

In our study we did not convert Shore B-trees to a "cache-conscious" format [RR99, RR00]. Such an alteration, at least on a system without all of the other optimizations we present, would have only a modest impact. Cache-conscious research on B-trees targets cache misses that result from accessing key values stored in the B-tree nodes. Our optimizations removed between 80% to 88% of the time spent in B-tree operations, without changing the key access pattern. Switching from a stripped-down Shore to our minimal-overhead kernel — which still accesses the same data — removed three quarters of the remaining time. In other words, it appears to be more important to optimize other components, such as concurrency control and recovery, than to optimize data structures. However, once we strip a system down to a very basic kernel, cache misses in the B-tree code may well be the new bottleneck. In fact, it may be the case that other indexing structures, such as hash tables, perform better in this new environment. Again, these conjectures should be carefully tested.

6. RELATED WORK

There have been several studies of performance bottlenecks in modern database systems. [BMK99] and [ADH+99] show the increasing contribution of main memory data stalls to database performance. [MSA+04] breaks down bottlenecks due to contention for various resources (such as locks, I/O synchronization, or CPU) from the client's point of view (which includes perceived latency due to I/O stalls and preemptive scheduling of other concurrent queries). Unlike the work presented here, these papers analyze complete databases and do not analyze performance per database component. Benchmarking studies such as TPC-B [Ano85] in the OLTP space and the Wisconsin Benchmark [BDT83] in general SQL processing, also characterize the performance of complete databases and not that of individual OLTP components.

Additionally, there has been a large amount of work on main memory databases. Work on main memory indexing structures has included AVL trees [AHU74] and T-trees [LC86]. Other techniques for main memory applicability appear in [BHT87]. Complete systems include TimesTen [Tim07], DataBlitz [BBK+98], and MARS [Eic87]. A survey of this area appears in [GS92]. However, none of this work attempts to isolate the components of overhead, which is the major contribution of this paper.

7. CONCLUSIONS

We performed a performance study of Shore motivated by our desire to understand where time is spent in modern database systems, and to help understand what the potential performance of several recently proposed alternative database architectures might be. By stripping out components of Shore, we were able to produce a system that could run our modified TPC-C benchmark about 20 times faster than the original system (albeit with substantially reduced functionality!). We found that buffer management and locking operations are the most significant contributors to system overhead, but that logging and latching operations are also significant. Based on these results, we make several interesting observations. First, unless one strips out all of these components, the performance of a main memory-optimized database (or a database without transactions, or one without logging) is unlikely to be much better than a conventional database where most of the data fit into RAM. Second, when one does produce a fully stripped down system — e.g., that is single threaded, implements recovery via copying state from other nodes in the network, fits in memory, and uses reduced functionality transactions — the performance is orders of magnitude better than an unmodified system. This suggests that recent proposals for stripped down systems [WSA97, SMA+07] may be on to something.

8. ACKNOWLEDGMENTS

We thank the SIGMOD reviewers for their helpful comments. This work was partially supported by the National Science Foundation under Grants 0704424 and 0325525.

9. REPEATABILITY ASSESSMENT

All the results in this paper were verified by the SIGMOD repeatability committee. Code and/or data used in the paper are available at http://www.sigmod.org/codearchive/sigmod2008/

10. REFERENCES

[ACL87] Agrawal, R., Carey, M. J., and Livny, M. "Concurrency control performance modeling: alternatives and implications." *ACM Trans. Database Syst. 12(4)*, Dec. 1987.

[AMS+07] Aguilera, M., Merchant, A., Shah, M., Veitch, A. C., and Karamanolis, C. T. "Sinfonia: a new paradigm for building scalable distributed systems." In *Proc. SOSP,* 2007.

[AHU74] Aho, A. V., Hopcroft, J. E., and Ullman, J. D. "The Design and Analysis of Computer Algorithms." Addison-Wesley Publishing Company, 1974.

[ADH+99] Ailamaki, A., DeWitt, D. J., Hill, M. .D., and Wood, D. A. "DBMSs on a Modern Processor: Where Does Time Go?" In *Proc. VLDB,* 1999, 266-277.

[Ail04] Ailamaki, A. "Database Architecture for New Hardware." Tutorial. In *Proc. VLDB*, 2004.

[Ano85] Anon et al. "A Measure of Transaction Processing Power." In *Datamation*, February 1985.

[BBK+98] Baulier, J. D., Bohannon, P., Khivesara, A., et al. "The DataBlitz Main-Memory Storage Manager: Architecture, Performance, and Experience." In *The VLDB Journal*, 1998.

[BDT83] Bitton, D., DeWitt, D. J., and Turbyfill, C. “Benchmarking Database Systems, a Systematic Approach.” In *Proc. VLDB*, 1983.

[BHT87] Bitton, D., Hanrahan, M., and Turbyfill, C. “Performance of Complex Queries in Main Memory Database Systems.” In *Proc. ICDE,* 1987.

[BMK99] Boncz, P. A., Manegold, S., and Kersten, M. L. “Database Architecture Optimized for the New Bottleneck: Memory Access.” In *Proc. VLDB,* 1999.

[Bre00] Brewer, E. A. “Towards robust distributed systems (abstract).” In *Proc. PODC*, 2000.

[BDR97] Bugnion, E., Devine, S., and Rosenblum, M. “Disco: running commodity operating systems on scalable multiprocessors.” In *Proc. SOSP*, 1997.

[CDF+94] Carey, M. J., DeWitt, D. J., Franklin, M. J. et al. “Shoring up persistent applications.” In *Proc. SIGMOD,* 1994.

[CDG+06] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E. “Bigtable: A Distributed Storage System for Structured Data.” In *Proc. OSDI*, 2006.

[DG04] Dean, J. and Ghemawat, S. “MapReduce: Simplified Data Processing on Large Clusters.” In *Proc. OSDI*, 2004.

[DHJ+07] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. “Dynamo: amazon’s highly available key-value store.” In *Proc. SOSP*, 2007.

[DGS+90] DeWitt, D. J., Ghandeharizadeh, S., Schneider, D. A., Bricker, A., Hsiao, H., and Rasmussen, R. “The Gamma Database Machine Project.” *IEEE Transactions on Knowledge and Data Engineering* 2(1):44-62, March 1990.

[Eic87] Eich, M. H. “MARS: The Design of A Main Memory Database Machine.” In *Proc. of the 1987 International workshop on Database Machines*, October, 1987.

[GS92] Garcia-Molina, H. and Salem, K. “Main Memory Database Systems: An Overview.” *IEEE Trans. Knowl. Data Eng.* 4(6): 509-516 (1992).

[GR93] Gray, J. and Reuter, A. “Transaction Processing: Concepts and Techniques.” Morgan Kaufmann Publishers, Inc., 1993.

[GBH+00] Gribble, S. D., Brewer, E. A., Hellerstein, J. M., and Culler, D .E. “Scalable, Distributed Data Structures for Internet Service Construction.” In *Proc. OSDI,* 2000.

[Hel07] Helland, P. “Life beyond Distributed Transactions: an Apostate’s Opinion.” In *Proc. CIDR*, 2007.

[HM93] Herlihy, M. P. and Moss, J. E. B. “Transactional Memory: architectural support for lock-free data structures.” In *Proc. ISCA*, 1993.

[KR81] Kung, H. T. and Robinson, J. T. “On optimistic methods for concurrency control.” *ACM Trans. Database Syst. 6(2):213-226*, June 1981.

[LM06] Lau, E. and Madden, S. “An Integrated Approach to Recovery and High Availability in an Updatable, Distributed Data Warehouse.” In *Proc. VLDB*, 2006.

[LC86] Lehman, T. J. and Carey, M. J. “A study of index structures for main memory database management systems.” In *Proc. VLDB,* 1986.

[LGG+91] Liskov, B., Ghemawat, S., Gruber, R., Johnson, P., Shrira, L., and Williams, M. “Replication in the harp file system.” In *Proc. SOSP*, pages 226-238, 1991.

[MSA+04] McWherter, D. T., Schroeder, B., Ailamaki, A., and Harchol-Balter, M. “Priority Mechanisms for OLTP and Transactional Web Applications.” In *Proc.ICDE*, 2004.

[MHL+92] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. “ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging.” *ACM Trans. Database Syst. 17(1):94-162*, 1992.

[Moh89] Mohan, C. “ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes.” 1989, Research Report RJ 7008, Data Base Technology Institute, IBM Almaden Research Center.

[ML89] Mohan, C. and Levine, F. “ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging.” 1989, Research Report RJ 6846, Data Base Technology Institute, IBM Almaden Research Center.

[MBD+99] Mucci, P. J., Browne, S., Deane, C., and Ho, G. “PAPI: A Portable Interface to Hardware Performance Counters.” In *Proc. Department of Defense HPCMP Users Group Conference,* Monterey, CA, June 1999.

[RR99] Rao, J. and Ross, K. A. “Cache Conscious Indexing for Decision-Support in Main Memory.” In *Proc. VLDB*, 1999.

[RR00] Rao, J. and Ross, K. A. “Making B+- trees cache conscious in main memory.” In *SIGMOD Record, 29(2):475-486,* June 2000.

[SMK+01] Stoica, I., Morris, R., Karger, D. R., Kaashoek, M. F., and Balakrishnan, H. “Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications.” In *Proc. SIGCOMM*, 2001.

[SAB+05] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O’Neil, E., O’Neil, P., Rasin, A., Tran, N., and Zdonik, S. “C-Store: A Column-oriented DBMS.” In *Proc. VLDB*, 2005.

[SMA+07] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P. “The End of an Architectural Era (It's Time for a Complete Rewrite).” In *Proc. VLDB*, 2007.

[Tim07] Oracle TimesTen. http://www.oracle.com/timesten/index.html. 2007.

[TPCC] The Transaction Processing Council. TPC-C Benchmark (Rev. 5.8.0), 2006. http://www.tpc.org/tpcc/spec/tpcc_current.pdf

[WSA97] Whitney, A., Shasha, D., and Apter, S. “High Volume Transaction Processing Without Concurrency Control, Two Phase Commit, SQL or C.” In *Proc. HPTPS*, 1997.