Improving OLTP concurrency through Early Lock Release

Manos Athanassoulis† manos.athanassoulis@epfl.ch
Ryan Johnson†‡ ryan.johnson@epfl.ch
Anastasia Ailamaki† anastasia.ailamaki@epfl.ch
Radu Stoica† radu.stoica@epfl.ch

†Ecole Polytechnique Fédérale de Lausanne
Data-Intensive and Applications Laboratory
CH-1015, Lausanne

‡Carnegie Mellon University
Pittsburgh, PA

EPFL-REPORT-152158

ABSTRACT

Since the beginning of the multi-core era, database systems research has restarted focusing on increasing concurrency. Even though database systems have been able to accommodate concurrent requests, the exploding number of available cores per chip has surfaced new difficulties. More and more transactions can be served in parallel (since more threads can run simultaneously) and, thus, concurrency in a database system is more important than ever in order to exploit the available resources.

In this paper, we evaluate Early Lock Release (ELR), a technique that allows early release of locks to improve concurrency level and overall throughput in OLTP. This technique has been proven to lead to a database system that can produce correct and recoverable histories but it has never been implemented in a full scale DBMS. A new action is introduced which decouples the commit action from the log flush to non-volatile storage. ELR can help us increase the concurrency and the predictability of a database system without losing correctness and recoverability. We conclude that applying ELR on a DBMS, especially with a centralized log scheme makes absolute sense, because (a) it carries negligible overhead, (b) it improves the concurrency level by allowing a transaction to acquire the necessary locks when the previous holder of the locks has finished its useful work (and not after committing to disk), and (c), as a result, the overall throughput can be increased by up to 2x for TPCC and 7x for TPCB workloads. Additionally, the variation of the waiting time of the log flush is zeroed because transactions no longer wait for a log flush before they can release their locks.

1. INTRODUCTION

The vast changes in computer hardware have resulted in the necessity to redesign the databases management systems in order to utilize the available technology efficiently [6, 7]. Multi-core computer systems are now the common case and we have to design and maintain systems that perform efficiently under such an environment. In this paper we study the impact of a multi-threaded design in the logging mechanism of database systems and we propose and evaluate a way to address the observed bottleneck. The increase in processors' speed and in memory capacity results in amplifying the importance of the commit time delay due to force writes [5], which becomes a serious bottleneck in achieving high performance in transaction processing.

1.1 Two metrics for DBMSs: Performance and Predictability

The question of performance of database systems running OLTP workloads is here for several decades, but the recent new trends in technology (processors and caches), have opened a lot of opportunities for addressing this issue. In order to utilize the available computational power of a multi-threaded, multi-core (on-chip) system, the software must be massively parallel.

In a complex system like a DBMS, any contention on a module of that system may create a bottleneck which kills performance. Recent research [2][13] has focused on identifying and attacking these bottlenecks, as they appear when a system is optimized for a multi-core environment.

In this paper we exploit the fact that reducing the time period transactions hold database locks can improve scalability, and consequently performance. Our experiments, in an open-source storage manager, show that one important component of the execution time of a transaction is actually spent waiting for the log of the database to be flushed. During that time the transaction is doing nothing but holding the locks. Moreover, our experiments reveal that the variation of the waiting time of the log flushing daemon is very high. For example, as we present in section 2, in an experiment with 100WH TPC-C workload[1], using a server with 64 hardware contexts, the standard deviation of the log flush waiting time was always greater or equal to 100% of the average waiting time. Predictability of response time plays an important role for user experience in some applications. For example, it is not important to have a variation of 1ms in an application where a network is delivering

[1] See section 4 for more details on TPC-C and TPC-B benchmarks.

the results to the end user (since the network can induce a much longer delays). On the other hand, this amount of time could be crucial if a mobile telephone loses the connection to the service provider because the database cannot inform promptly with the new connection parameters. Such a failure will lead to bad quality of services towards the end user. Thus, performance and predictability are two important metrics, describing a modern database system.

Flushing the log of a database creates a bottleneck, because the transactions are waiting for an I/O operation to complete, while holding the acquired locks. That way, they prevent other transactions to acquire these locks and start doing useful work even though the first transactions have finished their computations. We evaluate a technique, named Early Lock Release, according to which the locks held by a transaction can be released earlier than, just after, the log is written to the disk. Our results show that this technique can lead to higher throughput and more predictable duration of transactions, without losing the correctness and recoverability properties.

1.2 Related Work

Group-commit [8] is a first approach to address this bottleneck. Even though it significantly improves the throughput of a transaction processing system, group-commit does not solve the overall problem because it does not remove the delay in response time of individual transactions. Other related work includes [4] where it is proposed to store the log of a database in a flash-SSD disk in order to exploit the fact that the average access time of a flash disk is much lower than the average access time of a mechanical disk. There are, though, several, yet undocumented aspects of the behavior of a flash disk [9] which are currently investigated by the research community. The most important of them is unpredictability of behavior (e.g. access times) of flash devices in respect to time, vendor, flash technology generation and possibly state of the device [9].

In [1], partial strictness in two-phase locking was proven to accept recoverable histories. According to partial strictness a transaction can release all locks immediately after the commit request has arrived. In this paper we expand the theoretical work presented by Soisalon-Soininen and Ylönen. Finally, Wolfson [11] presented algorithms for determining when exactly a transaction can release safely locks before it is finished.

1.3 Early Lock Release (ELR)

Early Lock Release is based on the observation that the locks held by a transaction can be released as soon as the transaction has finished all actions but the commit action. At that point the transactions do nothing but wait for the log-flush before they release their locks and finish.

A new action needs to be introduced, namely Commit Request, in order to decouple in the lifecycle of a transaction the Commit action from the log-flush. To maintain correctness and recoverability properties, several conditions must hold, which we discuss in section 3.

The lifecycle of a transaction is described in Figure 1. Step D.ii (Flush log buffer to disk) is a time consuming operation, which according to our analysis is more than 10% of the total transaction duration even when there is no contention between transactions (for acquiring locks and writing to log buffer). In Figure 2 we experimented with a 100WH TPC-C workload (see Section 4 for more details on experimental setup) and we vary the number of threads from 10 up to 400 for both the baseline and a version of the same system agnostic to the log mechanism.

A. Begin
B. Growing phase
   i. Acquire locks (possible over-locking)
C. Reads, Computations and Writes
D. Commit
   i. Write commit to log buffer
   ii. Flush log buffer to disk
E. Release locks
F. Return to user

Figure 1: The lifecycle of a transaction

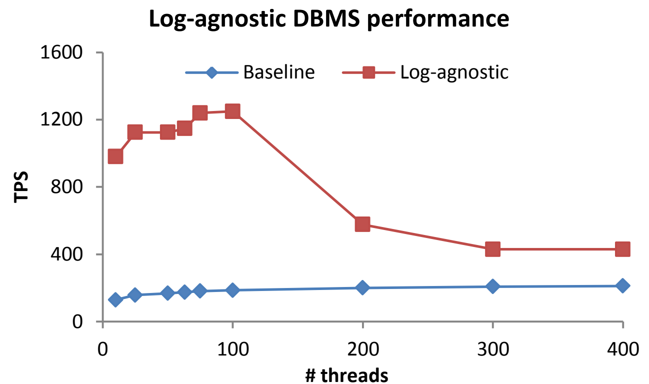

Figure 2: Baseline vs Log-agnostic (no waiting for log flushing) throughput, using a TPCC mix varying number of client threads

The data points presented in Figure 2 show the performance of an open source DBMS as we vary the number of threads in the system and the performance of the same system if we remove completely any obligation for the transactions to wait for the database log. The result is a totally unsafe system which provides theoretical upper bounds of the performance to achieve

when releasing the locks early. Thus, the potential that ELR tries to fulfill is promising, because, as seen in Figure 2 avoiding the waiting time at all, can lead to impressive performance increases.

For up to 100 threads the upper bound is about 7 times the baseline performance and for more threads the upper bound is lower but always above 2x. The system we did these experiments with, had 64 hardware contexts. So, for the rightmost points of the Figure 2, the context switching hurts the potential performance gain, allowing other bottlenecks to hurt the potential impact of ELR on the DBMS' throughput.

We evaluate Early Lock Release, according to which, a transaction can release the locks it holds immediately after it has requested to commit. At that point the transaction has completed all "useful" work and is ready to commit the log to the disk. We investigate the applicability and the impact of that technique for database systems with a single centralized log.

1.4 Contributions

In this paper we implement and analyze a technique described in Partial Strictness in Two-Phase Locking [1], aiming at reducing the log bottleneck from a database system. Soisalon-Soininen and Ylönen proved that this technique is correct but they did not go forward with implementation and experimental results. We implement Early Lock Release inside Shore-MT storage manager [2] offering a multi-core aware environment, which enables us to experiment on the relevance of this technique having in mind the current trends of computer micro-architecture. We experiment with TPC-C and TPC-B workloads and we conclude that ELR can provide significant performance boost with very little overhead, especially, when there is important contention on some part of the underlying database.

The reminder of the paper is organized as follows: In Section 2 we provide data that motivated our experiments and in Section 3 we describe in detail the concept of Early Lock Release and why it provides correct results. In Section 4 we describe our experimental methodology, in Section 5 we present our experimental results and analyses, showing the impact of applying ELR in OLTP workloads, and, finally, in Section 6 we conclude.

2. PERFORMANCE AND PREDICTABILI-TY IN OLTP WORKLOADS

Current database systems do not exploit the parallelism available in modern hardware [2]. A lot of research has been going on addressing this gap between software design and hardware capabilities. In this paper we explore the fact that by releasing locks early we can reduce contention, and thus, increase performance. At the same time, we observe another important issue that can be raised in DBMSs: predictability.

We focus on the behavior of a database system, especially due to the logging mechanism. The fact that all records of the database log have to be written to non-volatile storage (i. e., flush to disk) creates an important bottleneck, already identified in the literature. The throughput of a database system may depend highly on the throughput of the disk device storing the log. At the same time, the access time of disks varies significantly and this variation is added to the response time of the transactions.

In Figure 2 we see that removing completely the waiting for the log to be flushed, we increase performance by up to 7 times. This impressive speedup serves only as a potential maximum gain, which cannot be achieved, because the log is still needed in order to have a database that accepts transactions with ACID properties.

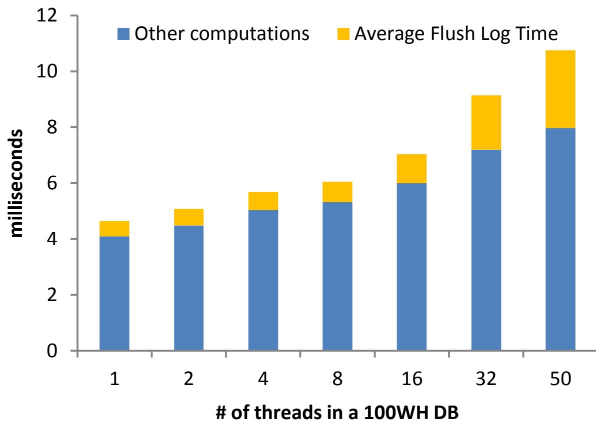

Figure 3: Average TRX duration, 100WH DB, TPC-C mix transactions

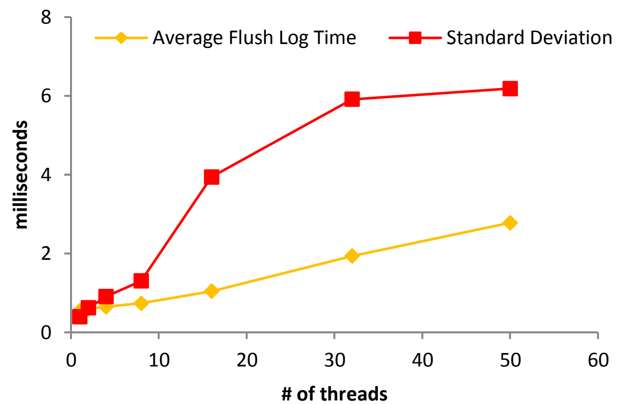

Figure 4: Average Flush log waiting time and standard deviation for 100WH DB, using TPC-C mix transactions

In Figure 3 we present the average duration of a transaction in TPC-C mix varying the number of client threads. Observing these results we see that the log-flush bottleneck is amplified as the number of threads used by the system increases. Thus, we propose to decouple the insertion of the commit log entry into the log buffer and the flushing of the log buffer, during the commit phase. The effect of this decoupling can be important, especially, in cases where there are many threads attempting to acquire locks on the same objects. Following the current trends, where the number of cores per processor is doubling every year [12], it will not be far in the future where thousands of contexts [2] will be available in a single server machine, enabling a DBMS to utilize as many threads.

We believe that releasing the locks before the log has been flushed can enhance the performance of a DBMS. This will be the case, particularly, if (a) there are a lot of transactions fighting for locks for the same database objects and (b) there exists over-locking. Over-locking, happens when the DBMS developer tends to lock in larger granularity than it is really necessary. We experimented with ELR, in a multi-threaded, scalable, open-source storage manager that is developed and maintained by our group and is descendant of Shore [3] storage manager.

The fact that the transaction duration depends on the log flush results in high variation of response times. As we can see in Figure 4, as the number of client threads are increased, not only the average duration of waiting the log flush grows, but also the standard deviation of these times is increased as high as 200% of the average. Next, in Figure 5 we can see the distribution of waiting times for a 100WH TPC-C workload using 50 client threads. We see that the values vary for 500ns up to 8ms, which may cause a transaction to wait for the log flush more than the actual duration of its useful work. We are interested in eliminating the effect of this waiting time on the forthcoming transactions that want to acquire the same locks. Thus, we implemented and evaluated Early Lock Release which is described in the following section.

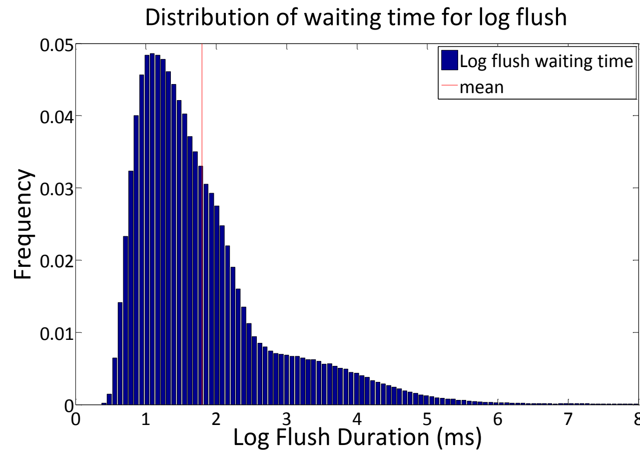

Figure 5: Predictability analysis: Log flush waiting time variation in a 100WH TPC-C workload with 50 concurrent threads serving database requests

3. EARLY LOCK RELEASE

In this section we describe the theoretical results presented by Soisalon-Soininen and Ylönen [1]. These results along with some of our initial experiments, enables us to argue that we lose performance and predictability of a DBMS during the time that a ready-to-commit transaction is waiting for the log flush.

- We lose performance because during that time, all the locks of the transactions are held, and any other transaction trying to acquire the same locks must wait for the first one to commit (that is, to flush the log to the disk), and,
- we lose predictability because the waiting time may vary in an unknown fashion (see Figure 5), because:
    - when a ready-to-commit transaction starts waiting for the log to be flushed, there is no information regarding when was the last log flush, and when will be the next log flush, and,
    - the I/O access time is variable.

However, waiting for the log to be flushed is necessary in order to ensure correctness and durability. Soisalon-Soininen and Ylönen [1] show that we can disconnect releasing the locks and flushing the log to the disk given that certain ordering properties are maintained. These properties will be discussed in the following section.

3.1 Partial Strictness

Two-phase locking can be too restrictive in several cases. In an in-memory database no disk accesses are needed for normal actions, but in commit, the changes should be stored in non-volatile storage, in order to be able to recover from possible failures. The commit action in such a setup is considerably costly as we can also see from Figure 3. On the other hand, much more concurrency would be possible if all locks could be released immediately after the commit request has arrived.

In order to design a system that allows the transactions to release its locks early we need to prove that this is possible. Soisalon-Soininen and Ylönen [1] proved it by defining Partial Strict Histories as follows.

Definition: Partial Strict Histories [1] are the action histories that allow reading and writing from a transaction that has not committed yet but all other actions have been completed and a request for committing the transaction has arrived.

In other words, if a system allows partial strict histories then whenever a transaction has completed the "useful" work and has decided that it will commit it can release its locks. That way it allows other transactions to start working with the same database objects while the former waits for the log flush. Before continuing to more details we should introduce the following notation:

CR_i: Commit Request of transaction i, is the action during which the commit log entry of transaction i is written to the in-memory log buffer

LR_i: Lock Release of transaction i, is the action during which all the locks of transaction i are released

C_i: Commit of transaction i, is the action during which the log up to the point of the commit of transaction i is flushed to non-volatile storage

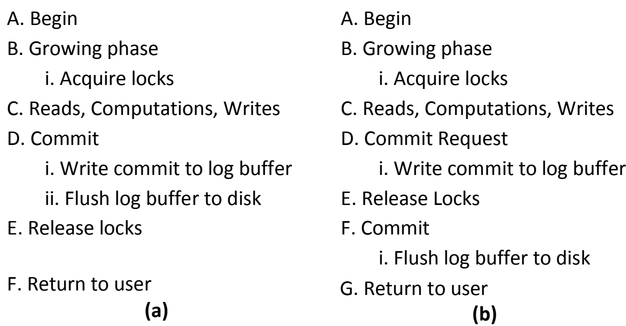

Figure 6: The lifecycle of **(a)** a transaction and **(b)** an ELR-enabled transaction

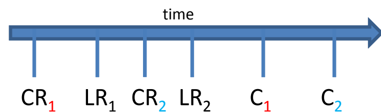

Figure 7: Preserved order for Commit Request and Commit actions of two transactions

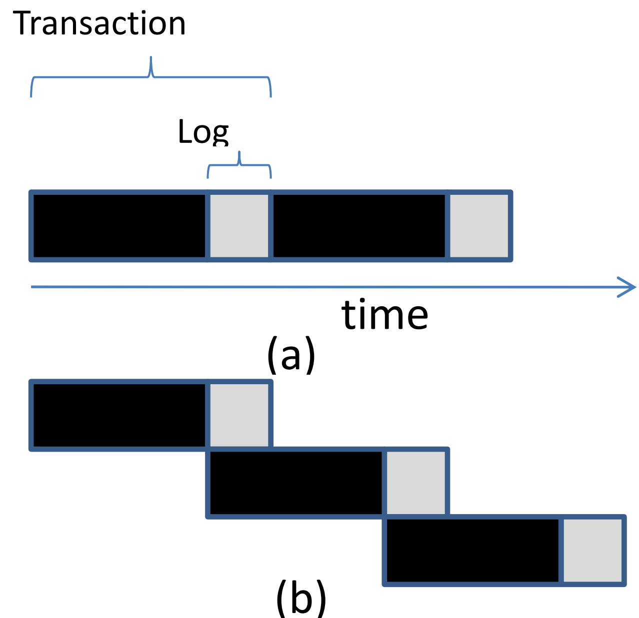

Figure 8: Schematic representation of Transaction execution time of two transactions using the same locks in **(a)** a conventional DBMS and **(b)** in a ELR-enabled system.

3.2 Correctness and Recoverability of ELR using Partial Strict Histories

Soisalon-Soininen and Ylönen [1] have defined Partial Strict Histories and, consequently, Partial Strictness over Two-Phase Locking. Moreover, the proved that Partial Strictness can ensure correctness and recoverability given two more properties discussed below. Partial Strict Histories need the introduction of the action Commit Request, which is actually a different action than Commit.

Below, we summarize the theoretical results presented in [1].

Theorem 1: Partially strict histories are recoverable.

Theorem 2: A partially strict history *H* is strict, provided that in no continuation of *H* (which may be *H* itself) the commit request action is followed by an abort action for the same transaction *T*.

Theorem 3: Whenever commit ordering is preserved, a partially strict two-phase locking accepts only partially strict histories.

Introducing a Commit Request action enables a DBMS to release the locks of the current transaction - even the exclusive locks - immediately upon the arrival of the Commit Request. Interestingly, we can do that without sacrificing any important recovery properties. According to Theorem 3, the property that must hold is that the order of Commit Request actions must be the same as the order that Commit actions are logged (as depicted in Figure 7). Moreover, according to Theorem 2, in order to preserve recoverability, whenever a transaction that performed the Commit Request action has to be aborted then all active transactions must also be aborted. The latter happens as standard procedure of addressing a system crash.

Soisalon-Soininen and Ylönen [1] described an elaborate locking scheme in order to preserve the commit request order in commits. This is not feasible because, firstly, a large number of locks was needed $O(n^2)$, where n is the number of transactions, and, secondly, the dynamic nature of a database system results in a huge variation in the arriving rate of transactions.

We argue that the Commit Request order can be easily preserved in the case of a single serial log with no elaborate locking or other commit-order preserving mechanism required. Implementation-wise the order between Commit Requests and Commits can be preserved using an in-memory log for every Commit Request action which is flushed to the log in the non-volatile storage by a log daemon in a periodic fashion.

In Figure 8 we can see the main impact of applying ELR, in a more intuitive way. The first part (black) of each transaction is the "useful" work done by the transaction, and the second part (light grey) is the amount of time spent waiting for the log to be flushed. If two transactions share the same locks then they are serialized during the execution of both parts. By applying ELR, we can partially overlap two transactions depending on the same locks by overlapping the second part, log flush (light grey), of the first transaction with the first part, "useful computation" (black), of the second transaction to be executed.

4. EXPERIMENTAL METHODOLOGY

This section describes the methodology behind our experiments. Our experimental setup is a Sun T5220 "Niagara II" machine, running Solaris 10. Each one of the 8 available cores on the Niagara chip has 8 hardware contexts, producing a total of 64 hardware contexts available to the OS as separate CPUs. Finally, each core has two execution pipelines, which makes the system able to load and execute instructions from any two threads at the same time. This machine provides more hardware contexts than any other chip today. We choose this CPU because it can offer a programming environment closer to what will be the common case in the future. This available parallelism is a preview of what will become available in the future, as the trend of doubling the number of on-chip cores continues to hold [12].

Architecture	Sun T5220 Niagara II
Operating System	Solaris 10
Cores (Threads/core)	8 (8)
RAM	64GB
HDD throughput	170 IOPS
Storage Manager	Shore-MT
Benchmarks	TPC-C (10WH and 100WH) TPC-B (1000 branches)

Table 1: Experimental Setup

The server described above is equipped with 64GB of memory and the disk on which the database log is being flushed into offers 170 IOPS. The database engine we use is Shore-MT [2], which is a descendant of Shore [3] storage manager that has been modified in order to offer support for the multi-core environment in which modern application should run and scale. We use Shore-MT because it scales well on modern hardware, it is open-source, so, we can implement and apply the technique we want to experiment with. In addition it behaves similarly to commercial database engines at the micro-architectural level [14].

Since Shore-MT is not yet equipped with SQL front-end we hard-coded the transactions of the benchmarks we used. Shore-MT has general-purpose metadata manipulation, but the transaction code has to be schema aware.

In the following sections we present the benchmarks we used for our experiments. In Table 1 we summarize the experimental setup.

4.1 TPC-C Benchmark

TPC-C is an on-line transaction processing benchmark developed by the Transaction Processing Performance Council [15]. In TPC-C nine tables are created, against which five different transactions operate. The transactions do update, insert, delete and abort, as well as primary and secondary index accesses. TPC-C models an on-line transaction-processing database for a retailer. The available transaction cover order creation, payment and delivery, as well as queries regarding order status and current stock level. The size of a TPC-C workload can be described by the number of warehouses that comprise the database instance in question. For example a 10-warehouse workload (10WH) is about 1GB and a 100WH workload is about 10GB.

4.2 TPC-B Benchmark

TPC-B [16] is a benchmark designed to measure the throughput of a system in terms of how many transactions per second a system can perform. It is a database stress test which creates four tables and each transaction accesses them. Unlike TPC-C there is only one type of transaction: a customer deposits and/or withdraws amounts of money for his or hers account at a branch (out of many available branches) of a hypothetical bank. Each transaction is performed by a teller at the branch of the bank. The size of a TPC-B workload is described by the number of branches assumed in the hypothetical bank. A database with 100 branches is about 2GB and a database with 1000 branches is about 20GB.

5. EXPERIMENTAL RESULTS AND ANALYSIS

In this section we present the experimental results and their analysis. Our first experiments aim at verifying the usefulness of the ELR technique. We used a 10WH TPC-C to do our sanity checks and a 100WH database for further analysis. Moreover, we used a TPC-B workload to extend our experiments.

5.1 Sanity check of ELR using 10WH TPC-C workload

We implemented a version of ELR in Shore-MT and we present our first results in this section. The first set of experiments was conducted using the aforementioned experimental setup. The workload that was used was a full TPC-C workload of 10 WHs. In the following figures we present the result of the ELR feature in the throughput of Shore-MT. For these sets of experiments we used an in-memory database and a log stored in the disk.

The baseline of our experiments was the Shore-MT storage system. Figure 9 shows the throughput achieved when using ELR or the baseline system. The x-axis is the number of threads per WH and the y-axis is the throughput of the compared systems in transactions per second. Figure 10 shows the speedup achieved for TPC-C mix and 3 more transactions when executed in the system alone. The x-axis is the number of threads per WH and the y-axis is the speedup as a result of using ELR.

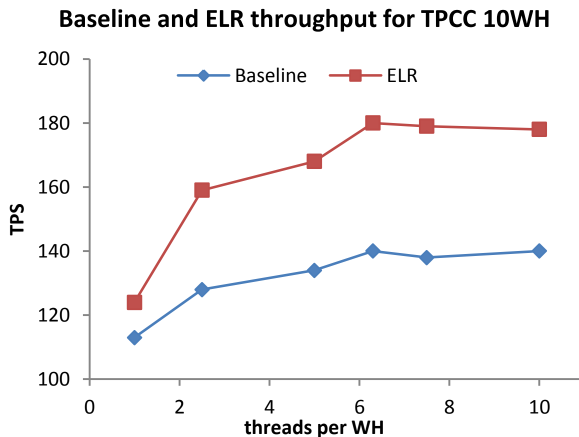

Figure 9: Shore-MT Baseline vs Shore-MT ELR throughput for TPC-C Mix in a micro-benchmark of 10WH

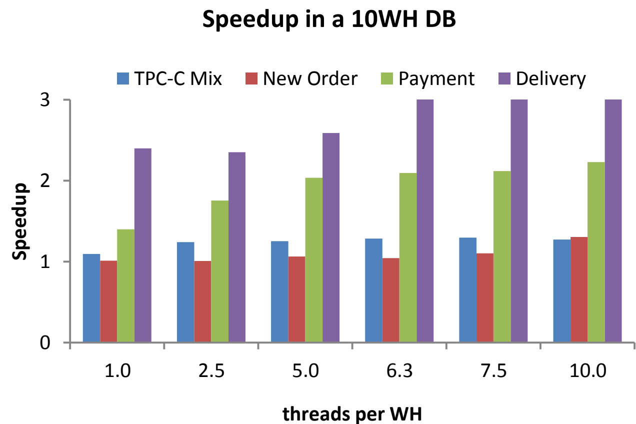

Figure 10: Speedup as a result of ELR for a 10WH database.

As we can see in Figure 9, ELR results in an increase of the performance for TPC-C mix up to 30%. As we expected, the number of threads played a very important role. As the number of threads grows, the probability of two threads requesting the

same locks increases. After 6.3 threads per WH (63 threads in total)[2] we do not observe any additional increase in TPC-C throughput. Actually, for both systems we see a local performance maximum for 63 threads. This is expected as experimental setup has 64 hardware contexts.

In Figure 10 we observe that for the 10WH workload, ELR results in an increase of performance for all different transactions and thread counts. The transaction New Order had a speedup up to 1.31x for 100 threads (10 threads per WH), while Payment reaches speedup of 2x for 5 threads per WH, and remains at 2x for more threads. Finally, Delivery is executed 2.5 times faster for 1 thread per WH and 3 times faster for 6.3 threads per WH or more. The fact that Delivery gets no performance gain for more than 6.3 threads per WH is expected because Delivery is not anymore I/O bounded for so many threads, and thread context switching does not allow for bigger speedup.

5.2 ELR with TPC-C 100WH

Based on our first results we extended our experimentation area with bigger and different workloads. In this section we present the experiments using TPC-C benchmark. We created a workload of 100WH, we stored the database in-memory and the log in disk. We used the same experimental setup (see Table 1) and we experimented for all transactions and several numbers of threads serving transactions.

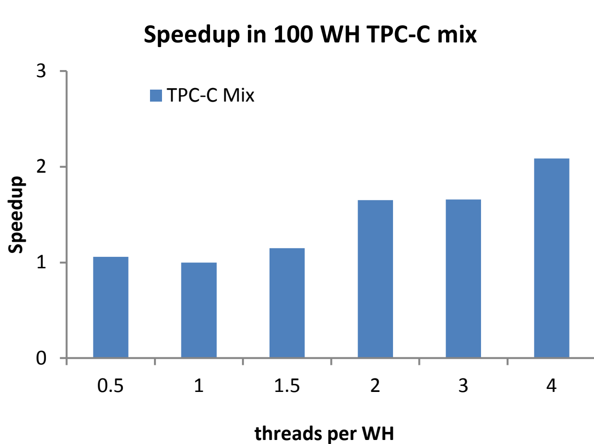

Figure 11: Speedup of throughput of Shore-MT versus Shore-MT with ELR, for TPC-C Mix in a 100WH TPC-C workload

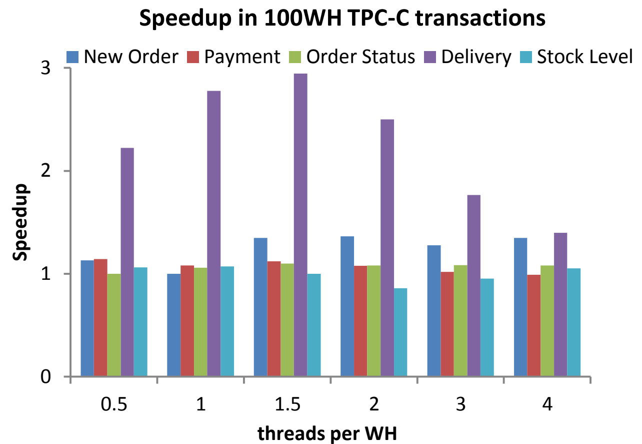

Figure 12: Speedup of throughput in 100WH TPC-C transactions when executed solely, by applying ELR

Figure 11 and Figure 12 present speedup results in a 100WH TPC-C workload. The x-axis for both figures is the number of threads per WH and the y-axis is the speedup ratio. As we can see in Figure 11, at a 100WH TPC-C benchmark, ELR resulted in an important raise of the throughput of the system. To be more specific, for threads per warehouse less than two, there was no significant speedup. With less than 1 thread per warehouse the system performs as fast as the I/O allows it (~170 transactions per second) for both baseline and ELR. In order to exploit the early release of the locks, the system must have enough threads ready to run, benefiting from the locks that have just been released. Thus, when we have enough parallelism, log flushing is overlapped with useful work from transactions, which resulted in accelerating the throughput by as much as 2 times.

In Figure 12 we present the speedup achieved for all available transactions of the TPC-C benchmark, for 0.5 up to 4 threads per WH. For this graph each line represents a set of experiments running only one out of the five available TPC-C transactions. The transactions Order Status and Stock Level are CPU bound because they do not alter the data of the database, so no log records are written. For a transaction being CPU bound, context switching only hurts performance. If the multiprogramming level is higher than the available hardware contexts, the running threads are context switched causing new code and data to be loaded. This results in a higher resource contention, and in a lower throughput. Also, we observe that ELR does not hurt performance of a CPU bound transaction.

5.2.1 Per transaction analysis

Figure 13, Figure 14, Figure 15 and Figure 16 present the throughput achieved in our experiments in transactions per second. The x-axis for all graphs is the total number of threads acting as clients of the database, and on the y-axis we show the throughput achieved from each experiment. For all these graphs, we use the experimental setup described in Section 4.

[2] We use both the terminology "number of threads" and "threads per warehouse". The first is important in order to know how much more parallelism is available, and the second helps us understand the probability of lock conflicts between transactions.

In Figure 13, we see the throughput measured in transactions per second (TPS) for different number of total threads for TPC-C mix. We observe that the baseline for more than 100 threads does not get improved performance. On the other hand, ELR results in increase of throughput for more than 150 threads, where the necessary parallelism has been achieved in order to exploit the fact that locks become available sooner.

In the Figure 14, Figure 15 and Figure 16 we present similar graphs for New Order, Payment and Delivery. In order to achieve an important performance gain for New Order the system needs enough parallelism (more than 100 threads, which stands for more than 1 thread per WH). On the other hand, Payment is executed more efficient even for 50 threads in the system (0.5 threads per WH) but the performance gain due to ELR remains small even for a higher number of threads. Finally, Delivery which has the lowest throughput (because the transaction is very large and it acquires a many locks, it has the most potential regarding ELR), is sped up by up to 2.5x for 2 threads per WH (200 threads in total). The offered parallelism for 300 and 400 threads changes the Delivery from I/O bound to CPU bound and the speedup decreases since the context switching consumes an important percentage of total executions time.

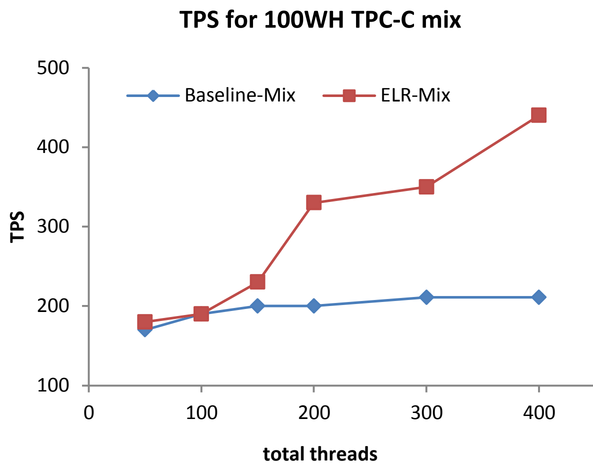

Figure 13: Transactions per second (TPS) for 100WH TPC-C mix without and with ELR

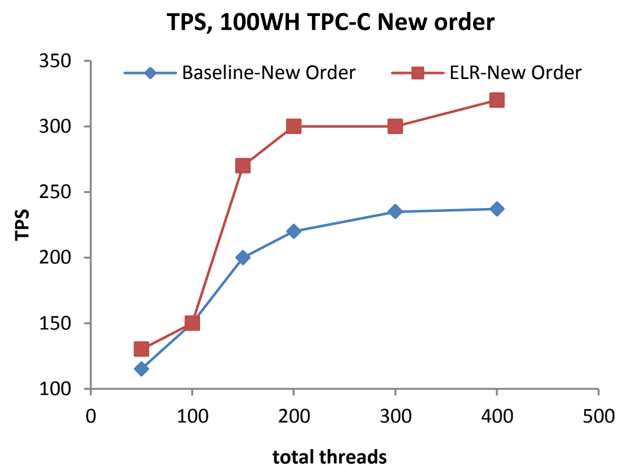

Figure 14: Transactions per second (TPS) for 100WH TPC-C New Order without and with ELR

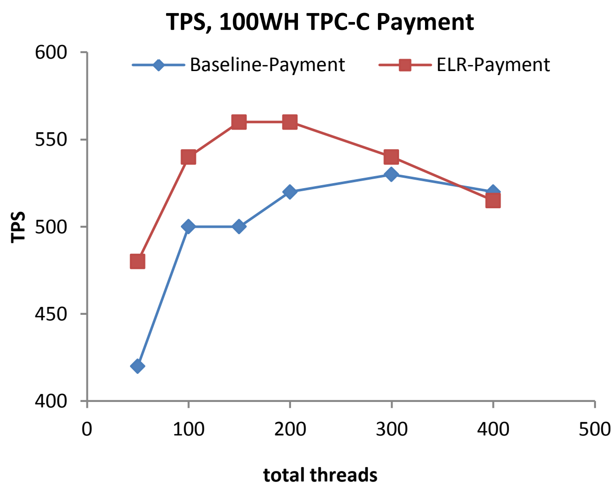

Figure 15: Transactions per second (TPS) for 100WH TPC-C Payment without and with ELR

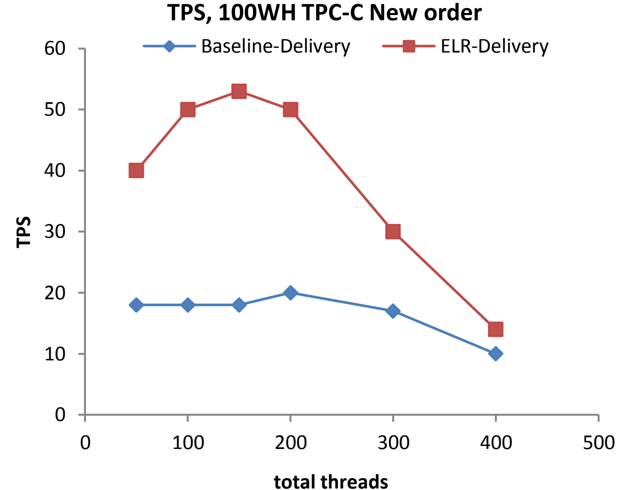

Figure 16: Transactions per second (TPS) for 100WH TPC-C Delivery without and with ELR

An interesting observation is that in Figures 14, 15 and 16, we do not see performance trends explaining the increase in performance depicted in Figure 13. When executing a single transaction we focus on the behavior of a single transaction, losing the flexibility to exploit different characteristics in order to overlap work. So, if a transaction is CPU bound then, as we increase number of threads, the performance will drop. On the other hand, if the transaction is I/O bound, increasing the number of threads may increase the performance up to the point that the offered parallelism can be exploited by the

workload. When executing a combination of all available transactions (TPC-C mix), then the system can exploit the available resources because the transactions have different behavior and there is a balance between CPU and I/O that helps in overlapping work.

5.3 ELR with TPC-B 1000 branches

In this section we present a different set of experiments, during which we use the TPC-B benchmark with a 1000 branches workload, executed on the same experimental setup. The database created has a size of about 20GB and the main metric for this workload is transactions per second. In this set of experiments we added one more parameter, h, which represents the skewness of data accesses and we evaluate its impact on the system. We use the self similar distribution [17] to produce skewed accesses to the branches in order to observe the impact of locality on the baseline system and on the ELR-enabled system.

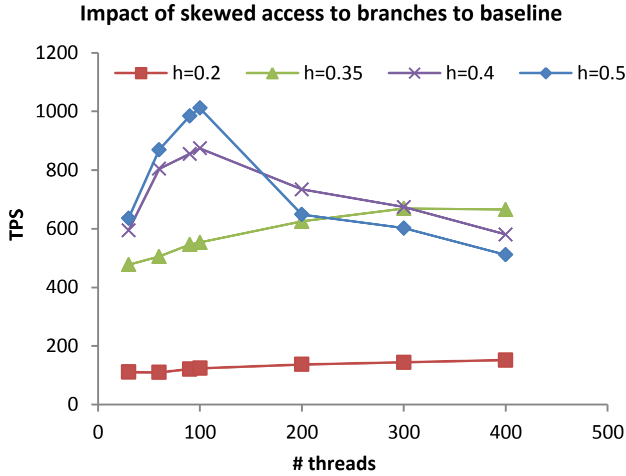

Figure 17: Baseline performance for different access skewness

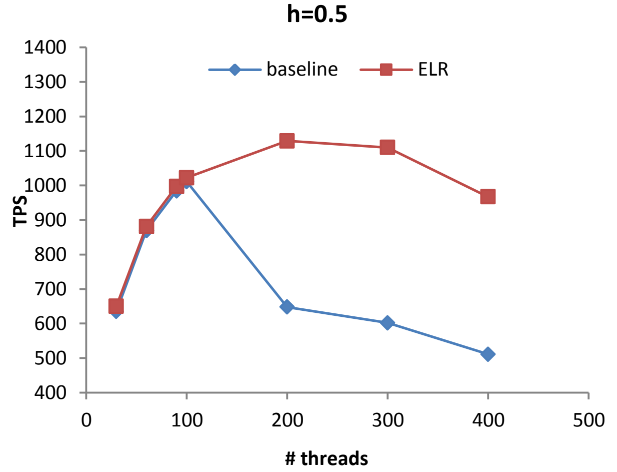

Figure 18: Baseline vs ELR throughput for h=0.5

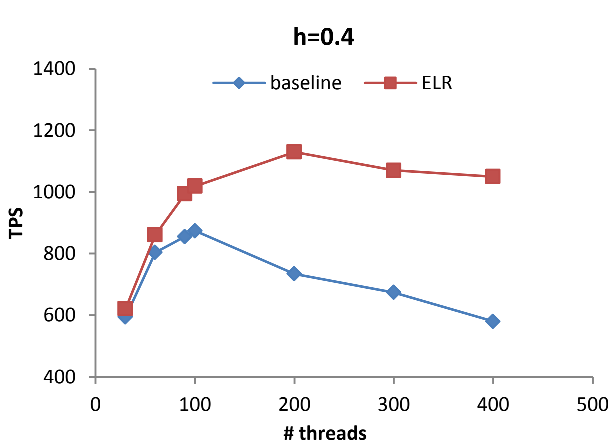

Figure 19: Baseline vs ELR throughput for h=0.4

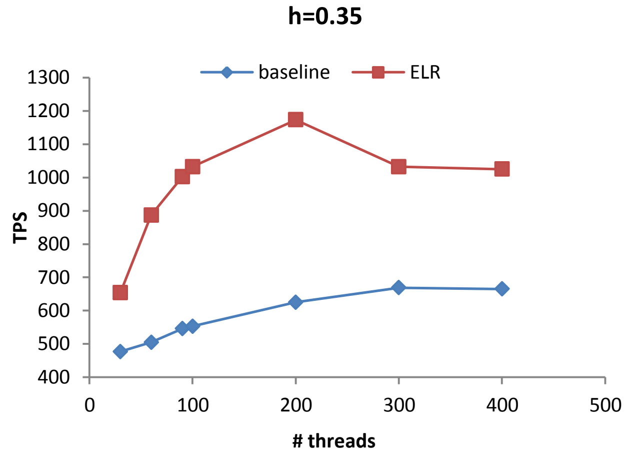

Figure 20: Baseline vs ELR throughput for h=0.35

A value S following the self similar distribution can be computed as follows:

$$S = 1 + \lfloor n \cdot X^{(log(h)/log(1-h))} \rfloor$$

where $n + 1$ is the maximum value that S can take, h is a parameter tuning the skewness and X follows the uniform distribution in $[0,1)$. For example, for $h = 0.2$ 80% of the accesses will be directed to 20% of the branches. The boundary case is $h = 0.5$, for which there is no effect on the produced data (50% of the accessed are directed to 50% of the branches which is equivalent to uniform).

In the following graphs we present the throughput observed for different number of threads used, as well as for different values of parameter h.

In Figure 17 we see the impact of the use of the skewed accesses distribution to the performance of the baseline system. In the x-axis there is the number of total threads of the system and in the y-axis there is the observed throughput in transactions per second. The same format is used for the figures to follow.

Increasing the skewness of the accesses makes the baseline perform worst because there is more contention in several parts of the database. This leads to many transactions fighting over the same locks. Moreover, since there are only 64 hardware contexts available, as the number of threads increases, the system cannot sustain more concurrency and the performance drops.

In the following graphs we see the performance of the baseline for four representative values of h, contrasted to the performance of the ELR-enabled system.

In Figure 18 we see the performance of baseline and ELR for $h = 0.5$, which means that the distribution of accesses is uniform. The baseline and the ELR perform similarly when there is not enough parallelism to gain from the fact that the locks are released early. As soon as the number of threads is sufficient ELR increases the performance by up to 2x. The absolute performance of ELR drops due to context switching following the same trend as baseline. In Figure 19 we present the throughput of the baseline and the ELR-enabled system for $h = 0.4$, introducing some skewness (60% of the accesses are directed to 40% of the branches). The baseline is more contended than in previous graph, but the ELR performs in a similar way regardless of the contention. This happens because transactions release locks early enough for other active transactions to continue doing "useful" work.

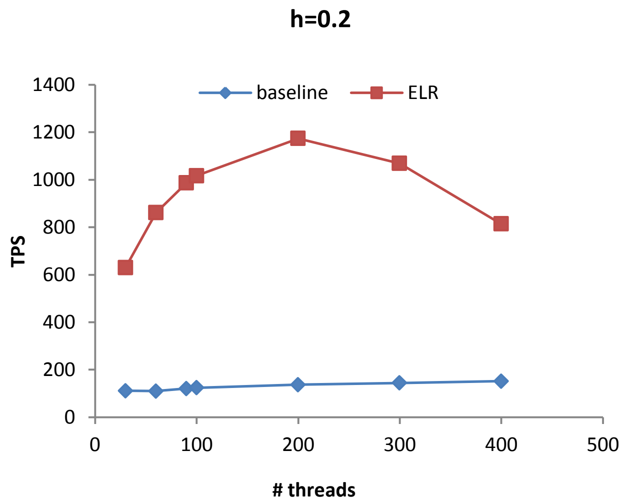

Figure 21: Baseline vs ELR throughput for h=0.2

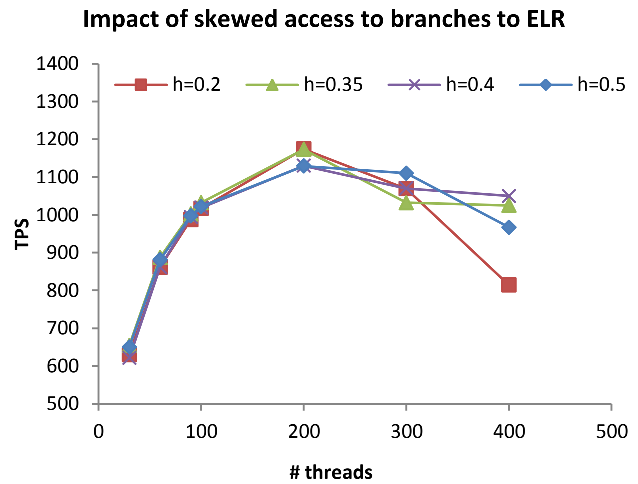

Figure 22: ELR performance for different access skewness

Higher contention decreases baseline performance but not ELR as seen in Figure 20 and Figure 21. Actually ELR performs very similarly for all different values of h (see Figure 22) leading to speedups from 2x up to 8x. The reason that ELR has so steady performance is that the impact of the (h-induced) contention to the baseline is cancelled by allowing the system to execute concurrently all eligible transactions. When the number of threads is high, then the system execute concurrently all transactions that do not try to acquire the same locks as default behavior. As we increase skewness of accesses distribution more contention is introduced, thus more transactions try to access the same part of the database. This leads to lower throughput for the baseline, because transaction hold hot locks while waiting for the log flush. On the other hand, ELR has the same throughput for all different levels of skewness because all locks, including the hot ones, are released as soon as the useful work is finished. In both cases the databases should be locked during the useful computation but if the transaction is not going to be aborted there is no point in holding the locks waiting for the log flush.

6. CONCLUSIONS AND FUTURE WORK

In this paper we evaluated a technique called Early Lock Release. According to this technique a transaction can release the locks immediately after the transaction has finished its "useful" work and has requested to commit. This will result in boosting the overall performance. There are, though, two open issues: recoverability and durability.

Recoverability can be ensured as long as the order of commit requests is preserved as the order of commits in the log. A single log can ensure that this property will hold and, as future work, we plan to explore the notion of ELR in a distributed

log environment. Durability is also ensured as long as the log is flushed to the disk. So, whenever a crash occurs, all active transactions have to be aborted.

After a series of experiments with TPC-C and TPC-B workloads, we concluded that the evaluated technique can be applied to modern database systems when we want to address the log bottleneck. The log of a database can be the bottleneck of database in cases of in-memory databases with log stored in non-volatile storage, or with databases for which the device were the data are stored can sustain such throughput to stress the throughput of the device hosting the log of the database. We see no reason why ELR cannot be implemented and used in a DBMS in order to boost its performance since the overhead is minimal and the performance gain can be up to 2x, without losing correctness and recoverability properties.

As future work we plan to test the behavior of a system implementing distributed logging and ELR at the same time. Distributed logging capability in a DBMS can lift a part of the burden of log-flush bottleneck, and combined with ELR can enhance the transactional processing performance.

Distributed logging can be implemented either in a shared-nothing fashion or in a shared-something fashion. Following a shared-nothing approach will allow ELR to be implemented easily in local transactions, but it should pose new questions as far as the durability is concerned. On the other hand, a shared-something approach can potentially incorporate the benefits of ELR and, on the same time, fulfill the necessary properties to ensure recoverability and durability. One approach could be to apply a partial order of all log entries in all different distributed logs. Following such an approach, the adopted partial order should ensure that two different commit requests should always happen (i.e. re-done) in the same order. If the same ordering is preserved in the log-flushing mechanism then early lock release mechanism will maintain the recoverability property.

7. REFERENCES

[1] Soisalon-Soininen, E., and Ylonen, T. "Partial Strictness in Two-Phase Locking", *ICDT 1995*.

[2] Johnson, R., Pandis, I., Hardavellas, N., and Ailamaki, A. "Shore-MT: A Scalable Storage Manager for the Multicore Era", *EDBT 2009*.

[3] Carey, M., DeWitt, D. J., Franklin, M., Hall, N., McAuliffe, M., Naughton, J., Schuh, D., Solomon, M., Tan, C. K., Tsatalos, O., White, S., and Zwilling, M. "Shoring up persistent applications", *SIGMOD 1994*.

[4] Lee, S.-W., Moon, B., Kim, J.-M., and Kim, S.-W. "A Case for Flash Memory SSD in Enterprise Database Applications", *SIGMOD 2008*.

[5] Stonebraker, M., Madden, S., Abadi, D., Harizopoulos, S., Hachem, N. and Helland, P. "The End of an Architectural Era (It's Time for a Complete Rewrite)", *VLDB 2007.*

[6] Harizopoulos, S., and Ailamaki, A. "A Case for Staged Database Systems", *CIDR 2003.*

[7] Hardavellas, N., Pandis, I., Johnson, R., Mancheril, N., Ailamaki, A., and Falsafi, B. "Database Servers on Chip Multiprocessors: Limitations and Opportunities", *CIDR 2007.*

[8] DeWitt, D., Katz, R., Olken, F., Shapiro, L., Stonebraker, M., and Wood, D. "Implementation techniques for main memory database systems", *SIGMOD 1984.*

[9] Bouganim, L., Jonsson, B., and Bonnet, P." uFLIP: Understanding Flash I/O Patterns", *CIDR 2009.*

[10] Eich, M. "A classification and comparison of main memory database recovery techniques", *Data Engineering*, pp. 332-339, 1987.

[11] Wolfson, O. "An algorithm for early unlocking of entities in database transactions", *Journal of Algorithms*, 7:146-156, 1986.

[12] Davis, J., Laudon, J., and Olukotun, K., "Maximizing CMP Throughput with Mediocre Cores", In Proc. PACT, 2005.

[13] Boncz, P., Zukowski, M., and Nes, N., "MonetDB/X100: Hyper-Pipelining Quert Execution", *In Proc. CIDR, Asilomar, CA, USA, 2005*.

[14] Ailamaki, A., DeWitt, D. J., and Hill, M. D. "Walking Four Machines By The Shore", *In Proc. CAECW*, 2001.

[15] Transaction Processing Performance Council (TPC). *TPC Benchmark C: Standard Specification*. Available online at http://www.tpc.org/tpcc/spec/tpcc_current.pdf.

[16] Transaction Processing Performance Council (TPC). *TPC Benchmark B: Standard Specification*. Available online at http://www.tpc.org/tpcb/spec/tpcb_current.pdf.

[17] Gray, J., Sundaresan, P., Englert, S., Baclawski, L., and Weinberger, P., "Quickly Generating Billion-Record Synthetic Databases", *In Proc. SIGMOD pp. 243-252, 1994.*

[18] Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., and Yelick, K., "The Landscape of Parallel Computing Research: A View from Berkeley", *Technical Report Technical Report No. UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.*