Executing Long-Running Transactions in Synchronization-Free Main Memory Database Systems

Henrik Mühe
TU München
muehe@in.tum.de

Alfons Kemper
TU München
kemper@in.tum.de

Thomas Neumann
TU München
neumann@in.tum.de

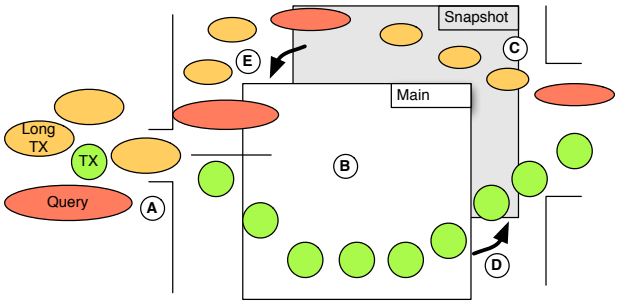
ABSTRACT

Powerful servers and growing DRAM capacities have initiated the development of main-memory DBMS, which avoid lock-based concurrency control by executing transactions serially on partitions. While allowing for unprecedentedly high throughput for homogeneous workloads consisting of short pre-canned transactions, heterogeneous workloads also containing long-running transactions cannot be executed efficiently. In this paper, we present our approach, called 'tentative execution', which retains the high throughput of serial execution for good-natured transactions while, at the same time, allowing for long-running and otherwise ill-natured transactions to be executed. To achieve this, we execute long-running transactions on a consistent snapshot and integrate their effects into the main database using a deterministic and short apply transaction. We discuss various implementation choices and offer an in-depth evaluation based on our main-memory database system prototype HyPer.

1. INTRODUCTION

For hiding I/O latencies, traditional disk-based database systems rely on parallelism which often requires explicit concurrency control mechanisms like two phase locking. Recent main-memory database systems like VoltDB [29], HANA [9] or HyPer [19] use serial execution on disjoint partitions to achieve high throughput without explicit concurrency control. This allows removing the lock manager entirely, which – even in disk-based database systems – has been shown to be a major bottleneck [15, 27]. In main-memory, data accesses are orders of magnitude faster than disk accesses. The lock-manager, however, does not inhibit a significant speedup since it has always resided in main-memory. Therefore, returning to explicit concurrency control is not an option for handling long-running transactions in main-memory DBMSs.

While yielding unprecedented performance for good-natured workloads, serial execution is restricted to a constrained set of transaction types, usually requiring suitable transactions to be extremely short and pre-canned. This makes main-memory database systems using serial execution unsuitable for "ill-natured" transactions like long-running OLAP-style queries or transactions querying external data – even if they occur rarely in the workload.



A) Incoming requests are divided into good- and ill-natured.
B) Good-natured requests are executed using a sequential execution paradigm.
C) Ill-natured requests are tentatively executed on a consistent snapshot.
D) Changes made to the main database are incorporated into the snapshot by a snapshot refresh.
E) Writes on the snapshot are applied to the main database after transaction validation.

Figure 1: Schematic idea of tentative execution.

In our approach, which we refer to as 'tentative execution', the coexistence of short and long-running transactions in main-memory database systems does not require recommissioning traditional concurrency control techniques like two phase locking. Instead, the key idea is to tentatively execute long-running transactions on a transaction-consistent snapshot of the database illustrated in Figure 1, thus converting them into short 'apply transactions'. While the snapshot is already available in our main-memory DBMS HyPer, which will be used and discussed in the evaluation, other systems can implement hardware page shadowing as used in HyPer or employ other snapshotting- or delta-mechanisms as illustrated in Section 3.

Since the transaction-consistent snapshot is completely disconnected from the main database, delays like network latencies or complex OLAP-style data processing do not slow down throughput of "good-natured" transactions (green transactions in Figure 1) running on the main database. If a transaction completes on the snapshot, a validation phase ensures that its updates can be applied to the main database under the predefined isolation level of the DBMS.

The remainder of this paper is structured as follows: In

6th Biennial Conference on Innovative Data Systems Research (CIDR '13) January 6-9, 2013, Asilomar, California, USA.

the following section, we discuss the breadth at which the workload for main-memory DBMS is extended by this work as well as the concrete scenarios discussed in this paper. In Section 3, our tentative execution approach is introduced in detail and implementation choices are offered. Afterwards, in Section 4, we offer a discussion of the system performance when using two phase locking which we compare to the performance of tentative execution in Section 5. There, we also discuss our prototypical implementation of tentative execution which we added to our main memory database system, HyPer. Section 7 concludes this paper.

2. WORKLOAD EXTENSION

In this section, we discuss the range of workloads that will benefit from a more general transaction processing paradigm and give pointers to real-world applications regularly employing transactions of this nature.

2.1 Duration

The focus on short transactions is essential for serial execution as no other transaction running on the same partition can be admitted while another long-running transaction is active. This – of course – causes throughput to plummet making long-running transactions nearly impossible to execute in a vanilla serial execution scheme. Recent research in the area of hybrid database systems, which can execute OLTP transactions as well as OLAP queries on the same state of the database, has lead to the development of the HyPer database prototype [19]. In HyPer, long-running read-only queries can be executed on a consistent snapshot without interfering with transactional throughput, therefore alleviating the problem in the read-only case.

Transactions with a runtime higher than few milliseconds that are not read-only cannot be executed in most recent main-memory database prototypes. This kind of transaction, though, is far from being hard to find. For example, the widespread TPC-E benchmark entails transactions with complex joins which require substantial time to execute.

Apart from complex transactions with high computational demands and therefore long runtime, we additionally identify interactive transactions as a workload that is currently incompatible with the idea of partitioned serial execution. Recently, work involving user-interactive transactions, so called *Entangled Queries* [14], received broad attention in the community highlighting the importance of supporting this workload type. Additionally, we have identified *Available to Promise* as both, a complex as well as a user-interactive transaction type. Here, users are presented with an availability promise for their orders which requires transactional isolation until the user has made a decision. Computing the stock level and therefore availability of the selected products is computationally expensive while interactivity occurs when waiting for the user's decision.

Another source of long-running transactions is application server interactivity. Frequently, application servers retrieve substantial amounts of data from a DBMS, make a complex decision involving other data sources and write the result of this operation back to the DBMS in one single transaction. In this scenario, latencies are typically smaller than waiting times for a user but are still significantly higher than what can be tolerated in a serial execution scheme.

2.2 Partitioning

Among the benchmarks used in the area of main-memory database systems – for example the TPC-C[1], the CH-benCHmark [5] or the voter benchmark[2] – many can be partitioned easily and inhibit only few or no partition crossing transactions. Most prominently, the TPC-C can be easily partitioned by warehouse id limiting the number of transactions that access more than one partition to about 12% as noted in [6]. Oftentimes, the partition crossing characteristics of a benchmark are even removed for the evaluation of main-memory database systems.

Unfortunately, not all workloads can be partitioned as easily as in the case of the benchmarks mentioned above. Curino et al. [6] show that the TPC-E[3] is hard to partition manually though they succeed in finding a promising partitioning scheme using machine learning. Commercial database applications – for instance SAP R/3 – have orders of magnitude more tables than the TPC-E and therefore make finding a simple partitioning which requires only few partition crossing transactions doubtful.

2.3 Scenario used in this work

In this paper, we will focus on application server interactivity since it is a natural addition to widely used benchmarks like the TPC-C. Additionally, it introduces an increase in execution time which is severe enough to render serial execution useless for this kind of transaction. Furthermore, application server interactivity is usually employed in cases where transactional isolation is an absolute requirement making solutions which decrease the isolation level to allow for efficient execution impossible.

3. TENTATIVE EXECUTION

The execution of "ill-natured" transactions takes place on a consistent snapshot. Alternative methods like execution on delta structures or using undo log information are in principle possible, as all general results presented here also apply to other mechanisms.

When an ill-natured transaction is detected, it is transferred to the tentative execution engine. The transaction is queued for the next snapshot being created after its arrival. Monitoring is employed during execution on the snapshot to allow for a validation phase on the main database. If the transaction aborts on the snapshot, the abort is reported directly to the user. If the transaction commits, a so called 'apply transaction' is enqueued into the regular sequential execution queue as pictured in step 4), Figure 2. As implied by the name, the apply transaction validates the execution of the original transaction and then applies its writes to the main database state. If validation fails, an abort is reported to the client. Otherwise, successful execution of the original transaction is acknowledged after the apply transaction has committed on the main database.

The remainder of this section details the specific concepts used for identifying transactions that should be run tentatively, monitoring, validation and the general execution strategy of tentative transactions.

[1]See `www.tpc.org/tpcc/default.asp`
[2]See `voltdb.com/sgi-achieves-record-voltdb-benchmarks`
[3]See `www.tpc.org/tpce/`

3.1 Identification of ill-natured transactions

Different mechanisms can be used to separate the workload into good- and ill-natured transactions. A simple approach is limiting the runtime or number of tuples each transaction is allowed to use before it has to finish. When a transaction exceeds this allotment – which can vary depending on the transactions complexity or the number of partitions it accesses – it is rolled back using the undo log and re-executed using tentative execution.

If no interactivity is allowed inside transactions, the rollback after a timeout is transparent to the user. This is because no decision about the success of the transaction, `commit` or an `abort`, has been made and no intermediate results of the transactions could have been observed by the user. This strategy is displayed as step 2) in Figure 2. If it is decided that a transaction should rather be executed tentatively, it is rolled back and reinserted (see step 3)) into the transaction queue with a label marking it as a 'snapshot transaction'. Although simple, a limit-based the approach yields satisfactory results for workloads consisting of many deterministic and short transactions and only some very long-running analytical queries.

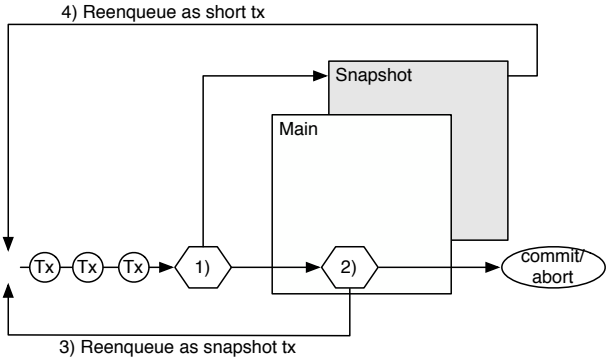


Figure 2: Schematic representation of the tentative execution approach presented in this work.

Apart from limit-based mechanisms, static analysis can be used for the execution of stored procedures. Here, potentially slow accesses to external data which take more than, e.g., a few microseconds to complete, can be identified à priori. Transactions that have already been identified to be long-running by the analysis can be tentatively executed from the start. Instead of relying on automated analysis, the user can also explicitly label transactions as tentative and therefore force execution on the snapshot if that behavior is deemed favorable.

Another possible option for identifying transactions which should be run using tentative execution is collecting statistics on previous executions of each transaction. When limit-based detection has frequently failed executing a certain transaction serially, the scheduler can use this knowledge to schedule the transaction for tentative execution instead of again trying to execute it serially.

3.2 View-serializability

To achieve *view-serializability*, a tentative transaction's read set on the snapshot must be equal to the read set that would have resulted from executing the transaction on the main database. To achieve this, we monitor all reads on the snapshot and validate them against the main database. This ensures that none of the writes performed on the main database by short good-natured transactions invalidate the visible state a tentative transaction was executed on.

Formally, we define *view-serializability* [30]: Let s be a schedule and $RS(s)$ be its *reads-from* relation. Intuitively, the *reads-from* relation contains all triples (t_i, x, t_j) for which a transaction t_j reads the value of the data element x previously written by transaction t_i (A formal definition, which we omit for brevity, can be found in Definition 3.7, [30]). Two schedules s and s' are said to be view-equivalent denoted $s \sim_v s'$ if their *reads-from* relations are equal:

$$s \sim_v s' \Leftrightarrow RF(s) = RF(s')$$

A schedule s is called *view-serializable* iff. a serial schedule s' exists for which $s \sim_v s'$. Intuitively, a schedule s is *view-serializable* when the state of the database read by the transactions in s is the same as the state of the database read by some serial execution of those transactions. We ensure this property by monitoring the reads a tentative transaction performs and validating them against the writes performed in parallel on the main database as detailed below.

3.3 Snapshot isolation

In addition to view-serializability, we offer *snapshot isolation* [1]. Here, the writes of a tentative transaction must be disjoint from those performed in parallel on the main database. This requires monitoring all writes performed on the snapshot such that conflicts with writes performed on the main database can be detected.

Formally, *snapshot isolation* can be defined as the set of schedules which can be generated when enforcing the following two rules [10, 25]:

1. When a transaction t reads a data item x, t reads the last version of x written by a transaction that committed before t started.
2. The write sets of two concurrent transactions must be disjoint.

We enforce 1. by running a transaction t on a snapshot that contains all transactions that have committed before t was admitted and by enforcing a concurrency control protocol like strict 2PL on the snapshot. The latter allows for writes to be done in-place on the snapshot without the danger of a concurrent transaction reading uncommited data from another tentative transaction. The disjoint write sets rule 2. is enforced using the monitoring approach described in the following section.

3.4 Intertransactional read-your-own-writes

While *read-your-own-writes* within one transaction as defined in the SQL standard [16] is fulfilled under both *snapshot isolation* and *view-serializability*, another related anomaly can be observed when using *snapshot isolation*, which we refer to as the intertransactional read-your-own-writes violation.

As an example in the context of tentative execution, consider a user u successfully executing a short transaction t_1 on the database which adds an order with a total value of \$100. Since the transaction is not long-running, it is executed using the sequential execution queue on the main database and commits. Then, u executes a new transaction, t_2, which

counts all orders valued at $100. If – under snapshot isolation – t_2 was dispatched to a snapshot created before t_1, u would not see the effects caused by her previously committed transaction t_1, an anomaly which we refer to as an intertransactional read-your-own-writes violation.

To avoid intertransactional read-your-own-writes violations, we require order preservation analogously to [30, page 102]. For every two transactions t and t' the following must hold: If t is executed entirely before t', all operations of t must come before all operations of t' in the totally ordered history. Intuitively, this ensures that for every transaction t', all effects of all transactions which finished and committed before t' are visible. This behavior is favorable, since users would expect that their transaction – for which a commit was already received – is part of the observed database state.

In our prototypical implementation of the tentative execution approach, adherence to *intertransactional read-your-own-writes* is given under *view-serializability*. Here, the read set is validated such that tentative transactions read the latest committed value for each data item, therefore implicitly fulfilling *intertransactional read-your-own-writes*.

Under *snapshot isolation*, transactions need to be executed on a snapshot which was created after the transaction was admitted. Since we refresh the snapshot periodically as indicated in Section 5.1, we queue all arriving tentative transactions until the next time the snapshot is refreshed, at which point we start their execution. As we can have multiple active snapshots in parallel (cf. Figure 7) and since snapshot creation is cheap, this causes only a minor delay which is noncritical since tentative transactions are long-running in nature.

3.5 Conflict Monitoring

Our approach is optimistic in that it queues and then executes transactions on a consistent snapshot of the database. This is advantageous as no concurrency control is required for the short- and apply-transaction execution. Similarly to other optimistic execution concepts, for instance [21, 17, 4], a validation phase is required which makes some form of monitoring necessary.

We formalize our monitoring approach as follows. An action that requires monitoring so that it can be verified during the apply phase is called an access. Under *snapshot isolation*, every write is an access and has to be monitored, whereas under *view-serializability*, every read is an access.

For each access performed by the tentative transaction under a given isolation level, we record a 2-tuple

$$(tid, snapshotVersion(tid))$$

and add it to L, the set of monitored accesses. Here, $snapshotVersion(tid)$ is defined as the version of a tuple identified by tid at the point the snapshot was taken. Therefore, it holds that

$$snapshotVersion(tid) = currentVersion(tid)$$

right after a snapshot of the database has been taken.

A tentative transaction is successful if a) it commits on the snapshot and b)

$$\forall (tid, ver) \in L : currentVersion(tid) = ver$$

holds.

Note that a version does not necessarily require a concrete version number or counter per tuple, its value can also be used as a version identifier. We exploit this fact for monitoring accesses which touch only very few tuples and attributes.

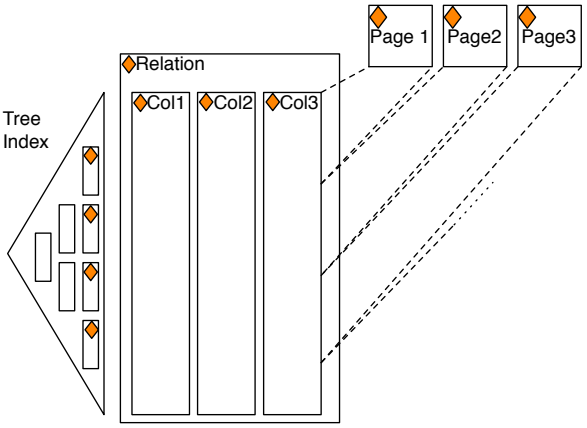


Figure 3: Monitoring using version numbers. The orange diamonds mark possible places where version counters can be employed to achieve different monitoring granularities.

In detail, we employ an adaptive monitoring strategy that depends on the nature of the SQL statement being executed. For requests using the primary index or other unique indexes, we do not use explicit per tuple version numbers but log the values of all attributes which are accessed. If compression is employed, it is sufficient to log the compressed value as long as decompression is possible at a later point in time, during the validation phase.

By logging an attribute value, the version of the accessed data is given implicitly through its value. Therefore, $snapshotVersion(tid)$ is equal to $currentVersion(tid)$ iff. all values of all accessed attributes of the tuple are equal on both the snapshot and the main database.

For statements that access multiple tuples, we vary the granularity at which accesses are logged depending on the access patterns observed on a table. A natural way of noticing changes to the underlying data is to introduce version numbers representing the state of a cluster of tuples. For instance, an entire relation can be versioned as a whole – that is a version counter is increased on every update performed on the relation. When the versions used on the snapshot during tentative execution as well as the version found when applying the transaction on the main database are equal, the datasets used by each transaction are disjoint and therefore conflict free, causing validation to succeed. To achieve other, finer granularities, version counters can be introduced on each column of a relation, on parts of the index, for example B^+-tree leaf nodes or on each memory page.

Our prototype implements the log as attribute values written to a chunk of shared memory. For each read/write of a request that has to be logged, we write all used attribute values as well as the cardinality of the request's result to the log. Since we use shared memory, the tentative transaction and the apply transaction can both access the same log structure which simplifies data sharing and makes explicitly copying the log unnecessary.

For *view-serializability*, we log selects and validate their result against the result of an equivalent select to the main

database during the apply phase. For *snapshot isolation*, we log the set of overwritten tuples and validate that we overwrite the same data on the main database and therefore the data being overwritten has not changed since snapshot creation, fulfilling the disjoint write set requirement of *snapshot isolation*.

In our setting, monitoring is preferential over methods like predicate locking which could be used to track overlaps in read/write sets as well: Monitoring is only required for the few long transactions running on the snapshot, not for the many short transactions operating on the main database. Tracking selection and update predicates would be required on both the main database and the snapshot causing a substantial slowdown for otherwise good-natured transactions.

3.6 Apply phase

During the apply phase, the effects of the transaction as performed on the snapshot are validated on the main database and then applied. This is done by injecting an 'apply transaction' into the serial execution queue of the main database. As opposed to the transaction that ran on the snapshot, the apply transaction only needs to validate the work done on the snapshot, not re-execute the original transaction in its entirety or wait for external resources.

Specifically, we distinguish between two cases: When serializability is requested, all reads have to be validated. To achieve this, it is checked whether or not the read performed on the snapshot is identical to what would have been read on the main database. Depending on the monitoring granularity, the action performed here ranges from actually performing the read a second time on the main database to comparing version counters between snapshot and main.

When *snapshot isolation* is used, the apply transaction ensures that none of the tuples written to on the snapshot have changed on the main database, therefore guaranteeing that the write sets of both the tentative transaction as well as all transactions that have committed on the main database after the snapshot was created are disjoint. This is achieved by either comparing the current tuple values to those saved inside the log or by checking that all version counters for written tuples are equal both during the execution on the snapshot and on the main database.

3.7 Concurrency

Since tentative execution is used for transactions which are unsuitable for sequential execution, it is essential to support concurrency on the tentative execution snapshot. As transactions are compiled differently if they are scheduled to be executed tentatively, we can support popular concurrency control techniques as used in traditional database systems – for instance two-phase locking. In contrast to using 2PL for the entire database, adding the overhead of a centralized lockmanager inside the tentative execution engine is uncritical: Relative to a transaction's runtime and other costs, locking overhead is minimal for transactions executed on the tentative execution snapshot, whereas the overhead of locking would be massive for the short transaction which we execute serially.

When the *view serializability* isolation level is used with tentative execution, a simpler concurrency control mechanism is possible. Since the read set of a tentative transaction is verified during the execution of the apply transaction, we can have multiple transactions run in parallel on the snapshot using latches to maintain physical but not necessarily logical integrity. If two tentative transactions interfere with each other, the conflict will be detected during validation in the apply transaction causing one of the conflicting transactions to abort. Therefore, when using *view serializability*, we can employ this type of optimistic concurrency control on the snapshot. Note that the simplified concurrency paradigm is not applicable for *snapshot isolation*, since writes could be based on an inconsistent view of the database which does not correspond to a previous consistent version.

3.8 Queries

OLAP queries constitute a special case of long-running transactions which do not contain a write component. The tentative execution approach introduced in this work requires no extension for such workloads; OLAP queries are simply forwarded to the snapshot where they are executed analogously to the OLAP execution pattern originally introduced for HyPer [19]. Since no writes are performed, the result of the execution is directly reported to the user without the need for an apply transaction.

Under *snapshot isolation*, no verification needs to be performed since the write set of the OLAP query is empty and therefore cannot conflict with writes on the main database. Under *view-serializability*, the *reads-from* relation of a read-only transaction t is equal to the *reads-from* relation of the serial schedule in which t is executed serially right after the consistent snapshot of the database was taken. Therefore, the execution of OLAP queries using tentative execution fulfills the *view-serializability* requirements as defined in Section 3.2. It is based on multiversion concurrency control like for instance [22] with its mode of execution most closely resembling the multiversion mixed synchronization method, as described by Bernstein, Hadzilacos and Goodman [2, Section 5.5]. There, *updaters* generate a new version for every update they perform on the database whereas *queries* work on a transaction consistent state that existed before the query was started.

3.9 Summary

Tentative execution converts an ill-natured transaction into a good-natured one by collecting unknown external values during the execution on the snapshot. From that, an apply transaction which does not require interactivity is generated, which can be executed using high-performance serial execution. In case of successful validation, the apply transaction commits and its effects are equal to the original transaction being run directly on the main database with the specified isolation level. If validation is not successful, the transaction aborts just as if a lock could not be acquired due to a deadlock situation in a system using locking or as if an illegal operation had to be performed when using timestamp based concurrency control (cf. [30]).

When used for the execution of read-heavy OLAP style transactions, coarse granularity monitoring can be used to allow for quick validation of otherwise large amounts of read data. Additionally, *snapshot isolation* can be used to reduce overhead when appropriate – for instance for the concurrent calculation of approximate aggregate values.

4. 2PL IN A MAIN-MEMORY DBMS

Two phase locking is a well known and well researched mechanism for concurrency control in database systems

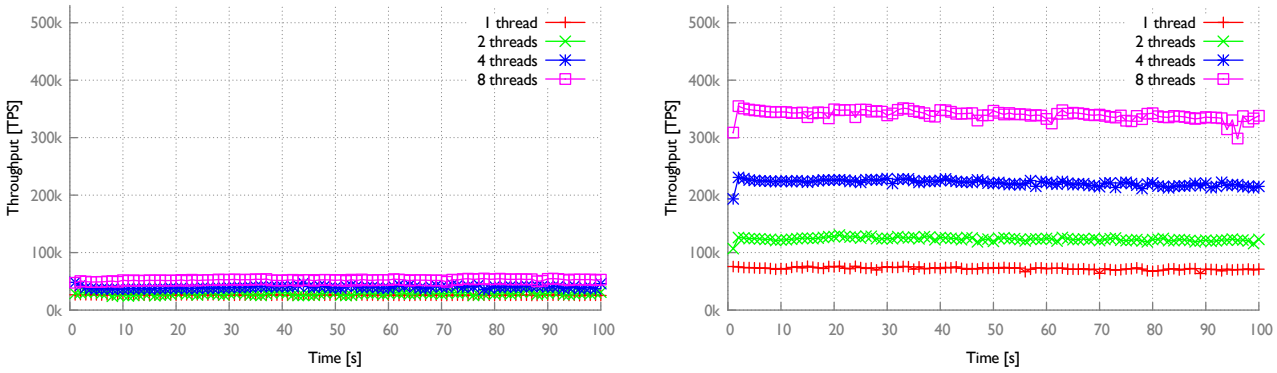


Figure 4: The TPC-C benchmark with 8 warehouses executed using 2PL on the left, partitioned serial execution on the right.

which is widely used, for instance in IBM's DB2 database system. To validate that research in the area of optimizing and extending partitioned serial execution is indeed worthwhile, we have conducted investigations into the overhead that 2PL causes in a main-memory setting. In order to get exact measurements of the overhead, we modified HyPer which currently uses only partitioned serial execution.

We implemented multiple granularity locking with a total of 5 locking modes (IS, IX, S, SIX, X) in a Gray and Reuter [13] style lock manager. For deadlock detection, we rely on an online cycle detection approach as introduced by Pearce et al. [28] for pointer analysis, which we apply to the wait-for graph. Our locking scheme uses coarse granularity locks for accesses which touch more than 5 tuples and fine granularity per-tuple locks otherwise. Since we operate entirely in main-memory, locks are usually held for only a few microseconds rendering a tall locking hierarchy inefficient.

To illustrate the overhead incurred by using 2PL, we ran the well known TPC-C benchmark scaled to 8 warehouses and measured the transactional throughput. In the case of partitioned execution, we partitioned the database by warehouse allowing for many accesses to be restricted to a single partition with an average of 12.5% of the transactions touching multiple partitions. When partitioned execution encounters a so called partition crossing transaction, it locks the entire database effectively disabling parallelism for the duration of said transaction.

In Figure 4, we show how throughput in the TPC-C benchmark varies between 2PL and partitioned execution. We measured the throughput in transactions per second over a 100 seconds long run of the benchmark for a varying number of threads. Clearly, partitioning performs better than locking both in terms of throughput increase per added thread as well as in terms of peak throughput.

The poor performance of 2PL can be explained by looking at profiler information on where time was spend during the execution of the benchmark. In our implementation, roughly 70% of the execution time is spend for locking related tasks as depicted in Figure 5

The comparably high overhead of locking is due to the fact that each transaction inside the TPC-C requires the acquisition of multiple locks on different hierarchy level. During the benchmark run shown in Figure 4, 6.5 million transactions were started which required the acquisition of roughly 400 million locks. 1 Million of these locks could not be fulfilled without waiting for another transaction to release the lock. 40,000 transactions had to be aborted, either because of a cycle in the waits-for graph or because of an unfulfillable lock upgrade request.

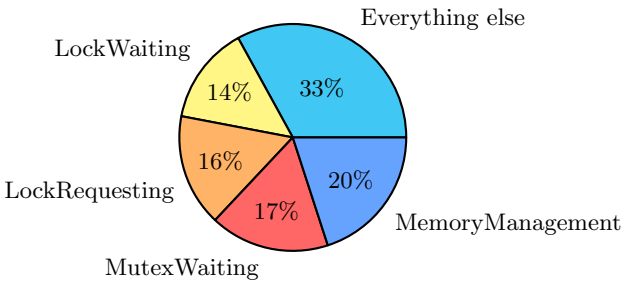


Figure 5: Areas in which time is spend during transaction execution.

Compared to 2PL, partitioned execution scales noticeably better peaking at roughly 350,000 transactions per second (note that redo logs were not persisted in this scenario causing an performance increase of roughly 10%). Here, the overhead of locking shared data which is necessary for the execution of roughly 12,5% of all transactions accounts for about 20% of the execution time. The remaining 80% is exclusively spent on transaction execution explaining the large difference in throughput in which partitioned execution is a factor 7 faster than two-phase locking.

We investigated ways of improving the performance of 2PL in the setting of executing a benchmark like the TPC-C, a benchmark exclusively containing short, pre-canned transactions. The most significant improvement we discovered is replacing lock waiting using condition variables with busy waiting. This is due to the fact that transactions are extremely short and context switches to another thread do not usually pay off, as waiting periods are in the order of a few microseconds. Sadly, this improvement – which allows performance gains of about 20% – is no help when it comes to the execution of transactions different from those in the TPC-C. When an ill-natured transaction is executed with busy waiting enabled, all threads waiting for a lock held by said transaction would actively spin and thus use memory bandwidth as well as completely block an execution unit.

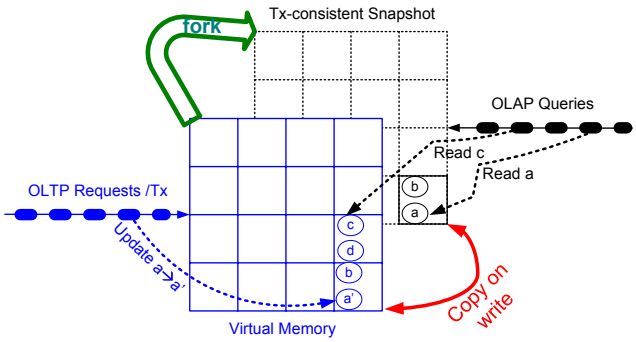


Figure 6: HyPer allows maintaining an arbitrary number of consistent snapshots at low cost using a hardware-supported page-shadowing mechanism.

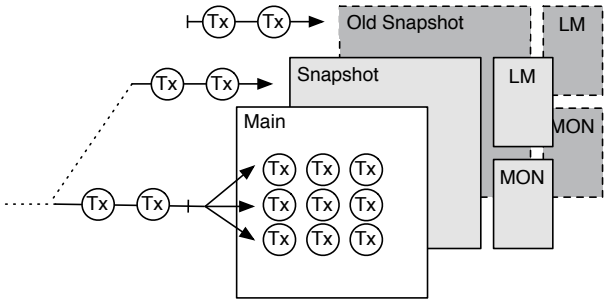


Figure 7: Architecture of the tentative execution approach with two snapshots for tentative execution. The *Snapshot* is a recent snapshot of the database used for executing tentative transactions whereas the *Old Snapshot* finishes the execution of its transaction queue without delaying the creation of a new snapshot or the need for transactions to abort.

5. EVALUATION

In the following Section, we will evaluate the performance of tentative execution in regard to abort rate, throughput and overhead when varying snapshot freshness and executing different workloads.

5.1 HyPer: Snapshot-based OLTP&OLAP

We evaluated our approach on our HyPer prototype database system. HyPer is a hybrid OLTP&OLAP main-memory database system relying on partitioned serial execution for transactions and allowing the execution of long-running read-only workloads by executing them on a consistent snapshot. Since a versatile snapshotting mechanism [23] that has a small memory footprint [11] already exists in HyPer, its usefulness is extended by tentative execution.

HyPer uses hardware page shadowing by cloning the OLTP process (fork) which allows for the cheap creation of an arbitrary number of snapshots which can coexist and share data (cf. Figure 6). Until a modification occurs on a page, memory pages are shared between all snapshots. On modification, a single copy of the memory page is created and the modification is performed on the copy, leading to the original page still being shared by all snapshots on which no modification took place.

This allows for the seamless refresh of a snapshot by cheaply recreating a new snapshot which still shares most pages with both the current snapshot as well the the original database. All transactions queued for execution on the old snapshot can still finish and be applied to the main database whereas the new snapshot can already be used for tentative execution while the old one finishes its work queue.

Using virtual memory for snapshots also allows for optimizing how read/write set logging is done: On most architectures, we could use the dirty-bit available for virtual pages to identify whether or not a page has changed. This is due to the fact that we can unset the dirty-bit when a snapshot is created and it is automatically set for each page when the page is modified. Therefore, no overhead is incurred for regular, good-natured transactions. On apply, we can find conflicts on virtual page granularity by checking the dirty-bits on pages touched by a tentative transaction.

5.2 Database compaction for faster forks

In order to execute transactions with tentative execution, a recent snapshot and therefore the ability to create a snapshot of the database at any time are important. Fresh snapshots minimize unnecessary conflicts with the main database caused by outdated data inside the snapshot. This work uses the compaction mechanism introduced by Funke et al. [11] to further minimize the cost of hardware supported page shadowing as used in the HyPer database system which was originally evaluated in [23].

Compaction is based on the working set theory of Denning [7]. It uses lightweight clustering to separate the database into a hot and a cold part.

While the hot part of the database can be updated in place and resides on small pages in memory; the cold part of the database – which is assumed to rarely change – is stored in an immutable fashion on huge memory pages. When a tuple inside the cold part needs to be updated, it is marked as deleted using a special purpose data-structure containing deletion indicators, copied into the hot part of the database and updated there.

Effectively, this causes cold tuples to be "warmed up" when a modification is required. Hot pages, which do not change anymore, are asynchronously moved to huge pages inside the cold storage part. Funke et al. show, that their mechanism has a negligible runtime overhead for both, OLTP transactions as well as read-only OLAP queries running on a snapshot.

Separating the data into hot and cold parts and storing those parts on differently sized pages increases fork performance since huge pages hold substantially more data per page table entry than small pages. Since 'forking' the database copies the pagetable eagerly and all data in a lazy fashion, the eager copying of the page table becomes faster due to reduced page table size.

5.3 Overhead incurred by tentative execution

First, we want to illustrate the overhead which is incurred by dispatching a transaction to a snapshot, executing it with additional monitoring and applying it to the main database. To show that our approach does not accumulate high runtime

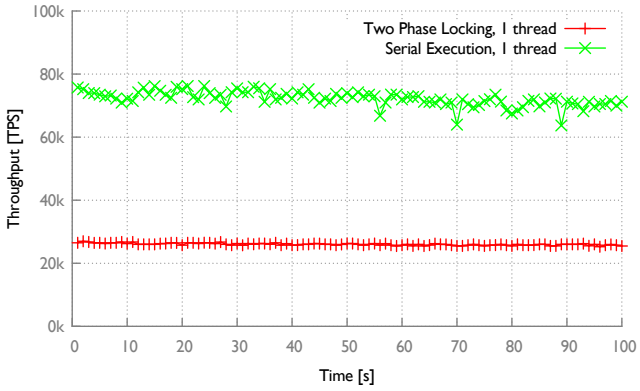


Figure 8: Comparison of 2PL and Serial Execution with a Multi-Programming Level of 1.

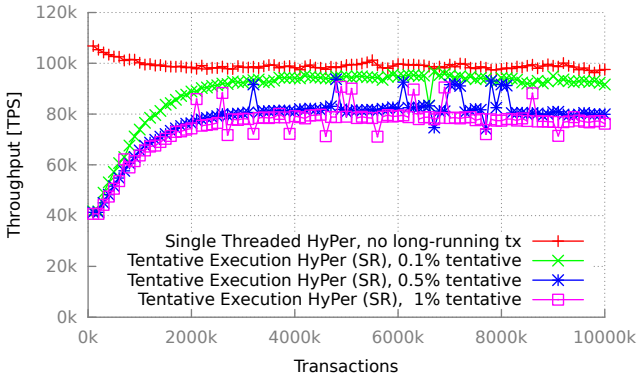


Figure 9: Throughput comparison between vanilla HyPer without any long running transactions and HyPer with long-running transactions using tentative execution.

costs, we ran the TPC-C benchmark and flagged all of its five transactions as being long running. This causes each of the transactions to be run by the tentative execution engine, which we switched to execution without concurrency on the snapshot for a more accurate comparison to regular HyPer.

We ran 10 million transactions distributed as required by the TPC-C and compared the throughput using tentative execution with regular execution of the TPC-C on our HyPer database system. We evaluated both, execution with *snapshot isolation* as well as serializable isolation level.

As a baseline comparison, we executed the TPC-C with Multi-Programming Level 1 to measure the baseline overhead of lock acquisition without taking contention or rejected lock requests into account. Figure 8 shows that going lock management – even without contention – slows processing down by a factor of approximately 3.

Figure 9 shows the throughput of vanilla HyPer versus HyPer with the two tentative execution variants on 100 batches of 100,000 transactions. Separate measurements show that when *all* transactions are executed tentatively (the worst-case scenario), the throughput is approximately cut in half compared to HyPer without tentative execution. This is caused by a multitude of factors: First, each transaction has to be executed just like in vanilla HyPer, so cost cannot possibly be lower. Second, every transaction needs to log the entire read set to memory which effectively converts each read into a read with an additional write operation to the log. Third, the data written to the log will later – in the validation phase – be accessed by a different process, reducing locality. Fourth, we identify records logically in our prototype and therefore need to perform every index lookup both on the snapshot as well as on the main database, thus doubling lookup costs.

With *snapshot isolation*, throughput is lower than with serializability. This seems counter-intuitive at first since the TPC-C reads more tuples than it updates and therefore the amount of data that needs to be logged and verified should be smaller for *snapshot isolation* compared to serializability. It is caused by the fact that view-serializable transactions can concurrently execute on the snapshot using latching whereas transactions under snapshot-isolation require a more complex concurrency control system on the snapshot, in our case two phase locking although others are possible. Note that although traditional concurrency control is a major source of overhead for short, good-natured transactions, no slow overhead for good-natured transactions is added when concurrency control is only used on the snapshot.

When comparing both tentative execution variants with vanilla HyPer, a different slope of the curves can be observed for the first million transactions. The decrease in throughput for vanilla HyPer comes from tree indexes rapidly growing in depth at the start of the execution. For the two tentative execution variants, the increase in throughput is due to copy-on-write operations used for snapshot maintenance being less frequent once old tuples are no longer updated and only new tuples are inserted and later updated.

For the frequent case of only a small fraction of the workload being identified as tentative, Figure 9 displays a throughput comparison. Here, a small fraction varying between 0.1% and 1% of the workload was executed using tentative execution. After a short ramp-up phase – which is caused by copy-on-writes after snapshot creation – throughput increases up to a level of roughly 80% of the transaction rate achievable with an unmodified version of HyPer.

In the unlikely worst-case where every transaction has to be executed tentatively, tentative execution takes about twice as long compared to regular execution in HyPer. This is caused mainly by added monitoring and validation overhead as well as operations like index lookups which have to be performed twice, once on the snapshot and the second time on the main database. In total, we consider the added overhead to be negligible for the expected ratio of ill-natured transactions.

5.4 Snapshot freshness versus commit rate

To measure the effect of snapshot freshness on commit rates, we added a third kind of payment transaction to the TPC-C which requires a credit check during the transaction before adding funds to a customers account and committing. The delay caused by the credit check varies between 1ms and 10ms with uniform distribution. Since a customer is technically able to commence another order and pay for it using a different method, the customers account balance can change during execution forcing the transaction to commit. The code for our 'paymentByCredit' transaction can be found in Appendix B.

Figure 10 shows the commit rate of the tentative 'payment-

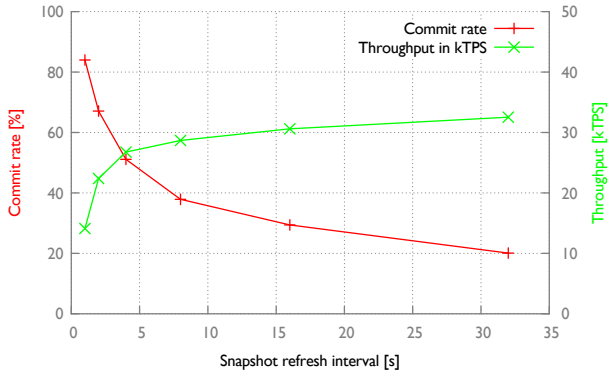


Figure 10: Commit rate and throughput of a tentative 'paymentByCredit' transaction depending on the refresh frequency of the snapshot.

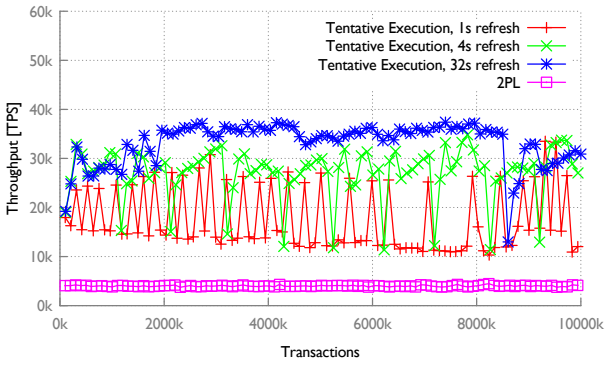


Figure 11: Throughput of tentative execution with varying refresh intervals compared to execution of the same workload using 2PL.

ByCredit' transaction as well as the total system throughput depending on the snapshot refresh interval. At 32s on the x-Axis, the snapshot is being refreshed every 32s seconds causing more tentative transactions to abort due to reading invalid data than when the snapshot is refreshed more frequently, for instance every 4 seconds. Therefore, as can be seen in the figure, the commit rate of the 'paymentByCredit' transaction decreases with less frequent refreshes and converges towards zero. This is an expected result as data becomes severely outdated when the snapshot is not refreshed. It should be noted that the transaction rate of 40,000 transactions per second combined with a total number of only 150,000 customers, each customer on average issues an order at the unrealistic rate of every 9 seconds, making the snapshot and the main database diverge quickly.

Total throughput, as opposed to the commit rate, increases with longer refresh intervals as can also be witnessed in Figure 10. This is due to the fact that 're-forking', i.e. recreating the snapshot and therefore refreshing it, requires the system to be quiesced when using hardware page shadowing as is the case in HyPer. With longer usage intervals before a snapshot is refreshed, transaction processing is quiesced less frequently causing an increase in throughput.

Even with very short re-fork intervals, tentative execution still performs favorably compared to execution using 2PL, as illustrated in Figure 11. The throughput for tentative execution is substantially higher than for locking, even when the snapshot is refreshed every second. The oscillations visible in the three instances of tentative execution displayed in Figure 11 stem from quiescing the database to fork which lowers throughput for some of the data points. The effect is less pronounced for long re-fork intervals, for example every 32s as pictured in blue, as only one refresh happens during the course of the benchmark. The red line, displaying tentative execution with a refresh interval of 1 seconds, exhibits a higher oscillation frequency with the lowered overall performance being due to the cost of quiescing the system and initial copy-on-write costs after creating a new snapshot.

5.5 Snapshot isolation versus serializability

While serializability offers classical consistency guarantees, *snapshot isolation* is widely used in commercial database systems and its implications are well understood [25]. In the following section we offer a brief summary of the effects of the different isolation levels on tentative execution and try to give guidelines in which scenario each isolation level is useful in this context.

5.5.1 *Monitoring memory consumption*

By definition, the amount of information that needs to be verified after a transaction has finished on the snapshot varies between *snapshot isolation* and *view serializability*. *Snapshot isolation* only requires the write set of tentative and regular transactions to be conflict free, whereas *view serializability* requires that the read set on the snapshot as well as main database is equivalent. Therefore, in terms of memory consumption, *snapshot isolation* is favorable for read-heavy workloads. This is emphasized as inserts do not need to be validated in terms of write set collisions but only constraint violations, which in our prototypical implementation is free as we recompute all inserts during the apply phase instead of explicitly logging the inserted values (cf. Section 3.5).

One way of monitoring the readset of a transaction to achieve *view-serializability* is logging all data read as well as the cardinality of all index accesses. If the actual data as well as the index cardinality are equivalent between snapshot and main database, the user transaction would have made the same decisions on either copy of the database and therefore the transaction can commit. Figure 1 shows the number of tuples deleted, inserted, updated and read during each read/write transaction of the TPC-C benchmark as well as the size of the tentative execution log for read logging. In addition to read-set logging, the log size required for write set logging as used under *snapshot Isolation* is shown.

Write log size is computed by adding the number of bytes deleted and updated to the log. The rationale behind this is that write conflict checking requires to check if the values overwritten or deleted are similar on the snapshot as well as on the main database. Apart from the actual content, the number of deleted and updated tuples has to be saved in the general case. In Table 1, an optimization is possible: Since all updated or deleted tuples are accessed using unique indexes, the update and delete operations are guaranteed to either fail (causing a rollback on the snapshot and therefore causing the log to be discarded) or succeed for exactly one

Tx	Unit	Avg	Min	Max
neworder	tuples delete	0.09	0	14
	bytes deleted	7.31	0	1120
	tuples inserted	11.98	4	17
	bytes inserted	851.70	320	1252
	attrs updated	1.01	1	2
	bytes updated	4.04	4	8
	attrs read	55.01	25	80
	bytes read	602.16	260	887
	index accesses	24.02	13	34
	AVG read log	602.16B + 24.02*8B = **794.32B**		
	AVG write log	7.31B + 4.04B = **11.35B**		
delivery	**AVG read log**	1636.34B + 249.39*8B = **3631.46B**		
	AVG write log	120B + 518.78B = **638.78B**		
payment	**AVG read log**	640.27B + 6.2*8B = **689.87B**		
	AVG write log	74.18B + 64.00B = **138.18B**		

Table 1: Log sizes in the read/write transactions of the TPC-C.

tuple. For inserts, it suffices to recheck for key violations during the apply phase.

The read log size is determined by the space required for logging all attributes which have been read during the execution on the snapshot. Additionally, the cardinality of all select statements has to be written to the log to ensure that no tuples were "missing" on the snapshot which are now visible on the main database. For both, read and write logs, all externally supplied values, for instance by the user or an external application server, are also added to the log to be incorporated in the apply transaction.

As illustrated in Table 1, the three read/write transactions of the TPC-C differ in terms of the size of their read vs. write log. This is expected since, for instance, `neworder` only inserts tuples which are implicitly checked during the apply phase by making sure no index properties are violated. Updates are only performed on one integer which is incremented with the next neworder id, causing only a minimal amount of data to be written to the write log. The `neworder` transaction accesses multiple tables to read data used in the newly created order entry, resulting in the higher memory consumption of the read log. For the three TPC-C transactions shown here, read-log size is consistently larger than write-log size.

5.5.2 Abort rate

The selected isolation level has a direct impact on the number of transactions that have to be aborted due to conflicts. For serializability, no changes to the read set of a tentative transaction are allowed during its runtime to allow the transaction to eventually commit on the main database. This includes changes to tuple values as well as changes in the cardinality of each selection's result. This can lead to long-running transactions suffering from high abort rates due to reading frequently-changing (hotspot) tuples.

Consider a transaction which computes and saves the total turnover for a warehouse (see Appendix A for an example). Here, the read set for the transaction will likely vary between snapshot and main database since an order might have arrived for the warehouse between snapshotting and the application of the tentative transaction, leading to a high number of aborts under view serializability. It is however likely, that the computed sum needs to represent a valid state but not the most recent one, which can be achieved using *snapshot isolation*. Here the transaction would apply iff. the aggregate being written has not been modified between snapshot creation and the tentative transaction's commit.

Besides the isolation level, the degree of detail in monitoring directly influences the number of transactions which commit. If, for example, a set of tuples is monitored using version counters on the B^+-Tree leafs of the primary index, a modification of one of the indexed tuples leads to all tentative modifications of any of the tuples indexed by the same leaf node to be rejected and therefore causes an abort. Thus, finer log granularities reduce the number of unnecessary aborts while at the same time increasing the overhead in both time and space caused by monitoring read/write sets.

6. RELATED WORK

Snapshot isolation, as Berenson et al. [1] pointed out, is an important relaxed isolation level for database systems. It has well-researched properties and anomalies which were, for instance, examined in [25, 18]. Jorwekar et al. [18] investigated the automatic detection of anomalies under *snapshot isolation*. Extending *snapshot isolation* to gain serializable schedules has been investigated by Fekete et al. [10]. *Snapshot isolation* is being used in practice, for instance as the default isolation level in Oracle database systems [26].

Harizopoulos et al. [15] found that concurrency control, primarily using a lock manager, is a major bottleneck in disk based database systems. Focussing more specifically on the contention which occurs in a 2PL lock manager, Pandis et al. [27] found that the central nature of the lock manager is a major source for lock contention causing a significant slowdown – especially as the number of cores increases. They devised data centric execution for disk-based systems implemented in their prototype database system, DORA. There, a chunk of data is assigned to each thread instead of assigning a specific transaction to each thread. Their approach increases data locality and reduces contention inside the lock manager.

Jones et al. [17] describe an approach to increase throughput in a distributed cluster setting by hiding delays caused by using two phase commit. They allow the tentative execution of new transactions as soon as a partition crossing transaction has finished work on one but not all partitions. Through optimistically executing followup transactions, they show that the throughput of pre-canned deterministic transactions can be significantly increased.

Bernstein et al. [3] introduced Hyder an optimistic approach for high performance transaction processing in a scaled out environment without any manual partitioning. Their key algorithm, called MELD [4], merges the log-file structured transaction states and handles conflicts during optimistic execution. Without any partitioning, MELD ensures that finished distributed transactions are merged into the last committed database state if they do not conflict.

Dittrich et al. [8] use a log-file structure to allow executing both OLTP as well as OLAP on the same database.

Larson et al. [21] discuss efficient concurrency control mechanisms for main-memory database systems. Apart from single-version locking, they introduce and evaluate multi-version locking and multi-version optimistic concurrency control based on timestamps ranges. They find that optimistic multi-version storage performs favorable compared to locking, especially when long-running transactions are part of the workload.

Nightingale et al. [24] employ speculative execution in the context of distributed file systems. They show that optimistic execution can be used to hide network delays when the outcome of a resource modification is highly predictable. Changes to resources r are done in place and an undo log structure is created for each updated resource. If another transactions tries to access a modified resource, it blocks or is marked as speculative and therefore dependent on the outcome of the transaction which originally modified r.

Gray et al. [12] explored using condensed apply transactions in disconnected application scenarios. Actually, their mechanism predates this papers in calling transactions which are run independently from the main database and later validated *tentative transactions*. Gray et al. validate their apply transactions (which they refer to as *base transactions*) with hard coded acceptance criteria instead of read or write set logging. They reduce synchronization cost on both main as well as disconnected databases.

The issue of managing long-running workloads in traditional, lock-based DBMSs has been investigated by Krompass et al. [20]. They use policy based scheduling taking multiple target dimensions into account to reduce the negative impact caused by long-running transactions in the workload.

7. CONCLUSION

Two emerging hardware trends will dominate database system technology in the near future: Increasing main memory capacities of several TB per server and an ever increasing number of cores to provide abundance of compute power. Therefore, it is not astonishing that main-memory database systems have recently attracted tremendous attention. To effectively exploit this massive compute power it is essential to entirely re-engineer database systems as the control and execution strategies for disk-based databases are inappropriate. In main-memory, databases using serial execution scale extremely well as there is no I/O latency slowing down the execution of transactions. This observation also led to the design of VoltDB/H-Store.

Unfortunately, so far, the serial execution paradigm excluded complex queries and long running transactions from the workload. With this paper, we have achieved a decisive step in allowing universal long duration transactions to coexist with short, pre-canned transactions – without slowing down their serial execution. This coexistence is achieved by exploiting the snapshot concept that was originally devised in HyPer to accommodate complex queries on the transactional data. Here, we developed the tentative execution method to pre-process long-running transactions in such a workload and then re-inject them as "condensed" apply transactions into the regular short transaction workload queue. Our performance evaluation proves that the high throughput for short transactions can indeed be preserved while, at the same time, accommodating "ill-natured" long-running transactions.

In conclusion, combining a state of the art main-memory database system, snapshotting using hardware page shadowing and tentative execution allows executing a wide range of workloads. With tentative execution, we can now support short, pre-canned transactions at high throughput while at the same time executing OLAP queries as well as long-running read/write transactions on a consistent snapshot.

8. REFERENCES

[1] H. Berenson, P. A. Bernstein, J. Gray, J. Melton, E. J. O'Neil, and P. E. O'Neil. A Critique of ANSI SQL Isolation Levels. *CoRR*, abs/cs/0701157, 2007.

[2] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - A Transactional Record Manager for Shared Flash. In *CIDR*, pages 9–20, 2011.

[4] P. A. Bernstein, C. W. Reid, M. Wu, and X. Yuan. Optimistic Concurrency Control by Melding Trees. *PVLDB*, 4(11):944–955, 2011.

[5] R. Cole, F. Funke, L. Giakoumakis, W. Guy, A. Kemper, S. Krompass, H. A. Kuno, R. O. Nambiar, T. Neumann, M. Poess, K.-U. Sattler, M. Seibold, E. Simon, and F. Waas. The mixed workload CH-benCHmark. In *DBTest*, page 8, 2011.

[6] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB*, 3(1):48–57, 2010.

[7] P. J. Denning. The working set model for program behaviour. *Commun. ACM*, 11(5):323–333, 1968.

[8] J. Dittrich and A. Jindal. Towards a one size fits all database architecture. In *CIDR*, pages 195–198, 2011.

[9] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner. SAP HANA database: data management for modern business applications. *SIGMOD Record*, 40(4):45–51, 2011.

[10] A. Fekete, D. Liarokapis, E. J. O'Neil, P. E. O'Neil, and D. Shasha. Making snapshot isolation serializable. *ACM TODS*, 30(2):492–528, 2005.

[11] F. Funke, A. Kemper, and T. Neumann. Compacting Transactional Data in Hybrid OLTP & OLAP Databases. *PVLDB*, 5(11):1424–1435, 2012.

[12] J. Gray, P. Helland, P. E. O'Neil, and D. Shasha. The dangers of replication and a solution. In H. V. Jagadish and I. S. Mumick, editors, *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada, June 4-6, 1996*, pages 173–182. ACM Press, 1996.

[13] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[14] N. Gupta, L. Kot, S. Roy, G. Bender, J. Gehrke, and C. Koch. Entangled queries: enabling declarative data-driven coordination. In *SIGMOD*, pages 673–684, 2011.

[15] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. OLTP through the looking glass, and what we found there. In *SIGMOD*, pages 981–992, 2008.

[16] A. N. S. Institute. *American national standard for information systems: database language, SQL*. 1986.

[17] E. P. C. Jones, D. J. Abadi, and S. Madden. Low overhead concurrency control for partitioned main memory databases. In *SIGMOD*, pages 603–614, 2010.

[18] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. Automating the detection of snapshot isolation anomalies. In *VLDB*, pages 1263–1274, 2007.

[19] A. Kemper and T. Neumann. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*, pages 195–206, 2011.

[20] S. Krompass, H. A. Kuno, J. L. Wiener, K. Wilkinson, U. Dayal, and A. Kemper. Managing long-running queries. In *EDBT*, pages 132–143, 2009.

[21] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. M. Patel, and M. Zwilling. High-performance concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4):298–309, 2011.

[22] D. B. Lomet, A. Fekete, R. Wang, and P. Ward. Multi-version concurrency via timestamp range conflict management. In *ICDE*, pages 714–725, 2012.

[23] H. Mühe, A. Kemper, and T. Neumann. How to efficiently snapshot transactional data: hardware or software controlled? In *DaMoN*, pages 17–26, 2011.

[24] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. *ACM Trans. Comput. Syst.*, 24(4):361–392, 2006.

[25] R. Normann and L. T. Østby. A theoretical study of 'snapshot isolation'. In *ICDT*, pages 44–49, 2010.

[26] Concurrency control, transaction isolation and serializability in SQL92 and Oracle7. White paper, Oracle Corporation, Oracle Corporation, 500 Oracle Parkway, Redwoord City, CA 94065, 1995.

[27] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-Oriented Transaction Execution. *PVLDB*, 3(1):928–939, 2010.

[28] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online Cycle Detection and Difference Propagation: Applications to Pointer Analysis. *Software Quality Journal*, 12(4):311–337, 2004.

[29] VoltDB LLC. VoltDB Technical Overview. `http://voltdb.com/resources`, 2010.

[30] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery*. Morgan Kaufmann, 2002.

APPENDIX

A. WAREHOUSEREVENUE

```
transaction aggregateWarehouseTurnover(int w_id) {
  select sum(ol_amount) as turnover
    from orderline ol where ol.w_id=w_id;
  update turnover_aggregates ta
    set ta.turnover=turnover where ta.w_id=w_id;
}
```

B. PAYMENTBYCREDIT TRANSACTION

```
transaction paymentByCredit(int w_id,int d_id,
  int c_w_id, int c_d_id,timestamp h_date,
  numeric(6,2) h_amount,timestamp datetime,int c_id)
{  select c_data,c_credit,c_balance
   from customer c where c.c_w_id=c_w_id and
   c.c_d_id=c_d_id and c.c_id=c_id;
   var numeric(6,2) c_new_balance+=h_amount;

   -- Approval processing
   approval_check(c_id,h_amount);

   if (c_credit='BC') {
      var varchar(500) c_new_data;
      sprintf (c_new_data,'%s␣|%4d␣%2d␣%4d␣%2d␣'+
        '%4d␣$%7.2f␣%12c',c_data,c_id,c_d_id,
        c_w_id,d_id,w_id,h_amount,h_date);
      update customer set
        c_balance=c_new_balance,c_data=c_new_data
        where customer.c_w_id=c_w_id and
        customer.c_d_id=c_d_id and
        customer.c_id=c_id;
   } else {
      update customer set c_balance=c_new_balance
        where customer.c_w_id=c_w_id
        and customer.c_d_id=c_d_id
        and customer.c_id=c_id;
   }
   insert into history values(c_id,c_d_id,c_w_id,
     d_id,w_id,datetime,h_amount,'credit');
}
```

C. DEMO DESCRIPTION

Our demonstration will show how partitioned serial execution excels with good-natured workloads. The main benchmark used for the demonstration uses a pre-canned version of the TPC-C where wait times have been removed to demonstrate high transaction rates without the need for loading thousands of separate warehouses. We also demonstrate how HyPer executes read-only workloads on an arbitrarily recent snapshot of the database as illustrated in Figure 6. Our benchmark setup, called the CH-benCHmark, is a combination of the TPC-C and the TPC-H in parallel. A detailed specification is available in [5].

Tentative execution is demonstrated using the same basic benchmark extended with long-running transactions which trigger the tentative execution mechanism. We will offer an interactive GUI to monitor throughput, change the workload and observe commit rates. Therefore, the audience can try other long-running demo transactions different from the examples shown in this paper to test custom usage scenarios and the versatility of the system.

The demonstration takes place locally on a portable demo machine as well as remotely on a Dell PowerEdge R910 server, currently being sold for roughly 40,000€ equipped with 4x 8 cores (16 hardware threads) Intel Xeon X7560 processors and 1TB of memory. The increased number of cores helps demonstrating the wider range of workloads a system with tentative execution can achieve by increasing the utilization of the available hardware.

In total, our demonstration encompasses a full functional implementation of tentative execution. This allows to demonstrate a main-memory database system capable of execution heterogeneous workloads consisting of short transactions, OLAP-style queries and ill-natured long transactions – that do not interfere with each other.