KuaFu: Closing the Parallelism Gap in Database Replication

Chuntao Hong[1], Dong Zhou[2], Mao Yang[1], Carbo Kuo[2], Lintao Zhang[1], Lidong Zhou[1]

[1]*Microsoft Research Asia, Beijing, China*
chhong, maoyang, lintaoz, lidongz@microsoft.com

[2]*Tsinghua University, Beijing, China*
dong.zhou.08, byvoid1@gmail.com

***Abstract*—Database systems are nowadays increasingly deployed on multi-core commodity servers, with replication to guard against failures. Database engine is best designed to scale with the number of cores to offer a high degree of parallelism on a modern multi-core architecture. On the other hand, replication traditionally resorts to a certain form of serialization for data consistency among replicas. In the widely used primary/backup replication with log shipping, concurrent executions on the primary and the serialized log replay on a backup creates a serious *parallelism gap*. Our experiment on MySQL with a 16-core configuration shows that the serial replay of a backup can sustain only less than one third of the throughput achievable on the primary under an OLTP workload.**

This paper proposes KuaFu to close the parallelism gap on replicated database systems by enabling concurrent replay of transactions on a backup. KuaFu maintains *write consistency* on backups by tracking transaction dependencies. Concurrent replay on a backup does introduce *read inconsistency* between the primary and backups. KuaFu further leverages multi-version concurrency control to produce snapshots in order to restore the consistency semantics. We have implemented KuaFu on MySQL; our evaluations show that KuaFu allows a backup to keep up with the primary while preserving replication consistency.

I. Introduction

We are witnessing two strong technological trends. First, providing highly available database services on commodity machines is becoming a common practice that makes database replication a necessity, because server failures are no longer negligible exceptions. Second, with the prevalence of multi-core architecture, a database system must achieve high concurrency in transaction processing to fully exploit the potentials in the underlying hardware. Somewhat surprisingly, those two technological trends are creating a serious tension between highly concurrent transaction processing and serialized execution imposed by traditional replication approaches to ensure replica consistency. For example, MySQL, one of the most popular open-source database systems, uses primary/backup replication with log shipping, where a backup executes each transaction in the log sequentially to ensure that the backup is consistent with the primary. As a result, we are observing a significant *parallelism gap* between a high degree of parallelism on the primary and a serialized log replay on a backup.

Such parallelism gap could introduce serious problems in practice. When the primary is heavily loaded with update transactions, backups might no longer be able to keep up with the primary. The system has to either suffer from increased risks of data loss, or to throttle the primary and take a performance hit. While so far not receiving the deserved attention in the research community, there is sufficient evidence to suggest that this is a real issue in practice [1] [2].

In this paper, we propose KuaFu to close the parallelism gap by enabling concurrent replay on a backup. To strike a balance among correctness, reusability, and ease of implementation, KuaFu applies to *row-based logging* with logged events reflecting changes on table rows. To ensure consistency between the primary and backups, KuaFu tracks transaction dependencies and makes sure that a backup preserves the same dependencies observed on the primary.

Tracking transaction dependencies ensures *write consistency* between the primary and the backups. All conflicting writes happen in the same order, and a backup reaches the same state as the primary after log replay is completed. However, write consistency does not constrain non-conflicting writes in different transactions. With concurrent replay on a backup, two non-conflicting transactions T_1 and T_2 might be executed in different orders on the primary and on a backup, leading to *read inconsistency*: If T_1 completes before T_2 on the primary, but T_2 completes before T_1 on the backup, a read on the backup might return a state that reflects the results of T_2, but not those of T_1, a state that never exists on the primary. To allow read operations to be served on backups without this kind of read inconsistency, KuaFu further introduces barriers to create snapshots that are consistent with some past states on the primary by leveraging the existing support of multi-version concurrency control.

In summary, this paper makes the following contributions. First, we have defined the notion of parallelism gap in database systems that arises with the technological trends of multi-core architecture on commodity servers. We have also shown the significance of the gap with evaluations on MySQL. Second, we have developed a solution that leverages transaction dependencies to close the parallelism gap by allowing concurrent replay on a backup. We have carefully analyzed the consistency semantics in terms of write consistency and read consistency, and provided a mechanism for a backup to achieve the same consistency as in the sequential replay case. Third, we have fully implemented KuaFu on MySQL with

less than 1,500 lines of code change. Our detailed evaluations have confirmed that KuaFu allows a backup to keep up with the primary on the same many-core machine, while preserving the desirable consistency.

The rest of the paper is organized as follows. Section II explains the causes and effects of the *parallelism gap* in database replication with a case study. Section III describes the design choices that KuaFu makes and presents its architecture, with Section IV covering the implementation of KuaFu on MySQL. We evaluate KuaFu and share our experiences in Section V, discuss related work in Section VI, and conclude in Section VII.

II. Parallelism Gap: A Case Study

Most modern servers have multiple processor cores. To leverage the power of all the cores, database servers usually run multiple threads to execute transactions concurrently. For high reliability, database replication is used to maintain consistent states across multiple database servers. We observe that traditional database replication mechanisms typically involve some form of serialization to ensure consistency among replicas. This is because concurrent executions tend to introduce non-determinism due to thread-interleaving, which could cause state divergence among replicas. Such replication-related serialization is at odds with the need to support concurrent transaction processing on multi-core processors.

In this section, we perform a case study to show the existence and the severity of the parallelism gap. To show that this is a real problem in practical database systems, we choose MySQL, one of the most widely used open-source database systems. MySQL supports primary/backup replication (primary and backup are also called master and slave in the literature), typically using log shipping. A replicated MySQL instance consists of a primary and one or more backups. Transactions are first executed on the primary. The primary generates logs recording the operations that these transactions have performed, and ships the log entries to the backups. Backups replay the log entries in a serial order to reach the same state as the primary. Read-only transactions may also be served on a backup although they might not reflect the up-to-date states on the primary.

We conducted an experiment with an OLTP workload on MySQL 5.5 with primary/backup replication using log shipping. The OLTP workload is a TPC-C workload without limiting the clients' maximum operation rates. It uses 100 warehouses with a total of about 10GB of data. The transactions in this workload consist of INSERT, UPDATE and DELETE. Different transactions can conflict with each other in this workload. We first run the workload on a primary server for 30 seconds without any backups and generate a log. We then start a backup and record the time it takes to replay the log that the primary has generated. We divide the total number of transactions by the processing time to get the number of transactions per second for both the primary and the backup.

Figure 1 shows the transactions per second for the primary and for the backup with different numbers of CPU cores.

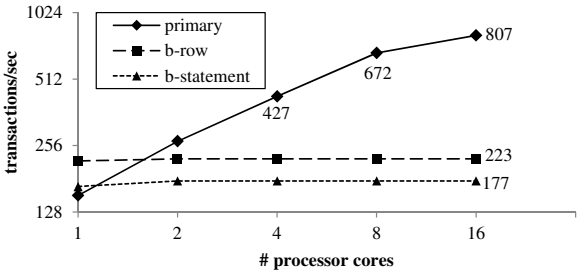

Fig. 1. The throughput of MySQL primary, and the maximum replay throughput of the backup, using different numbers of processor cores. Results with row-base logging and statement-base logging are represented as b-row and b-statement.

MySQL supports two logging methods: *row-based* logging and *statement-based* logging. To confirm that the existence of the parallelism gap is independent of the logging methods, we evaluate the replay performance on a backup in both logging methods. The *primary* data series show the throughput on the primary, given different numbers of cores. The *b-row* and *b-statement* series show the replay throughput on the backup, using row-based logging and statement-based logging, respectively. Because there is only a single thread used to replay the log, the performance on the backup is almost identical with different numbers of cores used.

The figure shows that, with a single core, the throughput on the backup is actually higher than that on the primary. With row-based logging, the replay performance is about 30% better than that of the primary when both servers use a single core. This is due to the lower cost of log replay compared to full transaction execution, as well as the additional read queries performed by the primary. However, when the primary uses more than two cores, the throughput on the primary starts to become higher than that on the backup. With 4 cores, the throughput on the backup using row-based logging is only half of that on the primary. The parallelism gap between the primary and the backup widens even more as more cores are used. When 16 cores are used, the replay on the backup can sustain less than one third of the throughput achievable on the primary.

III. Design of KuaFu

The root cause of the parallelism gap is the tension between determinism (needed for consistency in replication) and concurrency (desirable for performance on multi-core architecture). Any solution that is designed to close the parallelism gap must resolve this tension, while at the same time lives with the constraints from database replication. For example, because database systems are often mature and highly optimized, any complete re-design or even significant code modifications are undesirable and usually impractical. We therefore try to strike the right balance between generality and practicality. On the one hand, we would like the design to rely on well-accepted concepts in database systems and replication mechanisms and to be able to accommodate different low-level implementations. On the other hand, we favor a practical solution that

can be readily incorporated into a production database system like MySQL. In particular, we choose to focus on the widely used primary/backup replication with log shipping, a setup that we used in the experiment shown in Section II. In this section, we describe the design choices we have made and present the architecture of KuaFu.

A. KuaFu Design Choices

The basic idea of KuaFu is simple and straightforward. We let the primary execute transactions concurrently as in the non-replicated case and generate a log of the execution. Instead of having a backup replay the log in serial, KuaFu allows the backup to infer *dependencies* between transactions and allow non-conflicting transactions to execute concurrently during replay.

Making this basic idea work in practice involves careful design decisions on what to log and how dependencies are defined and inferred. A modern database system can be considered as consisting of two logical layers: the SQL layer and the storage layer. The SQL layer is responsible for SQL related tasks such as parsing SQL queries and writing transaction logs. It is also responsible for compiling SQL queries into a series of operations on data tables, which are maintained on the underlying storage layer. The storage layer implements table manipulation and manages the physical storage of tables. Database systems, such as MySQL and Microsoft SQL Server [3], have explicit layered architecture because the clean separation allows different storage engines to be used for customized optimizations.

Replication can be done either at the SQL layer or at the storage layer. KuaFu chooses to focus on replication at the SQL layer, so that the solution is independent of the underlying storage engines. There are further tradeoffs implied by this choice. For replication at the SQL layer the logical semantics (such as transactions and conflicts among transactions) are easily captured in log entries, such logical information might be unavailable when replicating at the storage layer. Moreover, replication at storage layer might lead to a larger log size and require more data to be transferred. On the flip side, replication at the storage layer suffers less from the parallelization gap, because log replay is fast on the backups due to less computation required to apply changes.

There are different ways to accomplish replication even at the SQL layer. The row-based logging and statement logging supported by MySQL are two examples. Statement logging works at the higher level by logging all the SQL statements being executed, while row-based logging works at a lower level by recording all the rows that a transaction modifies. Statement logging tends to result in smaller log sizes, but a backup has to execute those statements fully during relay. To ensure consistency, the execution of the logged SQL statements must be made deterministic. SQL statements with certain non-deterministic functions (e.g., RAND and DATE) must be handled carefully. Re-executing each statement puts extra overhead on a backup, making the parallelism gap more severe as shown in Figure 1. Furthermore, it is hard to check whether or not two transactions are in conflict with each other by looking at their SQL statements.

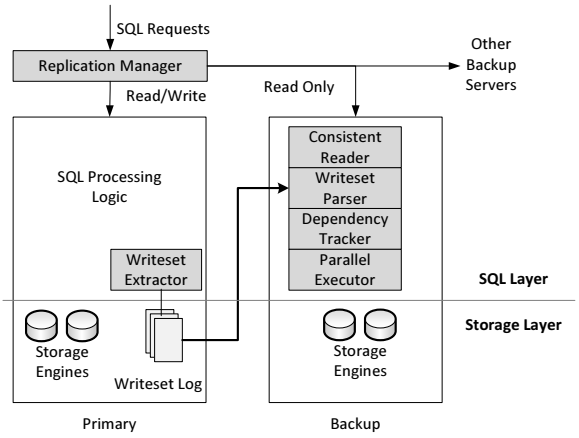

Fig. 2. KuaFu Architecture.

In contrast, with row-based logging, we can easily obtain the list of rows being modified for each transaction, making dependency tracking and assessment easy. During replay, two transactions are in conflict if and only if their write sets intersect. Not having to worry about read-write conflict simplifies implementation and increases the degree of parallelism. Row-based logging is a form of *result logging*, compared to the *operation logging* at the SQL statement level. A backup needs only to update the rows with the values in the log, without worrying about the read set of each transaction or re-executing SQL statements. The down side of row-based logging is that it might need to log more data. However, as shown in our experiments, this is not a significant issue in the database system we tested.

B. The Architecture of KuaFu

KuaFu chooses to focus on row-based logging and use write sets to track dependencies among transactions, though the basic idea can be applied to statement logging. Figure 2 shows an overview of KuaFu's architecture, where the gray components have been modified or added by KuaFu to an existing two-layer database system.

The KuaFu system is consisted of three components: the replication manager, the primary, and the backups. Replication manager is responsible for dispatching SQL requests to the servers. All write requests are sent directly to the primary server, and read-only requests can be sent to primary or backup servers depending on the load condition of the servers and the consistency requirements of the queries. Write transactions are first executed on the primary and then propagated to the backups. As the primary commits a transaction, it logs the data modified by that transaction with the writeset [4] extractor. The writeset log records exactly what has been changed by the transaction. By replaying the writeset log, the backups are able to replicate all data modifications performed on the primary. The writeset extractor works in SQL layer, so that different storage engines can share the same extractor.

```
graph = empty_set()
ready_transactions = empty_queue()

dependency_tracker()
{
    while( trx = next_trx_from_parser() )
    {
        trx.extract_writeset()
        vertex v = new_vertex( trx )
        for each u in graph
        {
            if( u.writeset intersects with v.writeset )
            {
                u.followers.push(v)
                v.n_incoming_edges++
            }
         }
        graph.insert(v)
        if( v.n_incoming_edges == 0 )
            ready_transactions.push(v)
    }
}

executor()
{
    // executed in each executor thread
    while( v = ready_transaction.pop() )
    {
        execute_transaction( v.trx )
        for each u in v.followers
        {
            u.n_incoming_edges--
            if( u.n_incoming_edges == 0 )
                ready_transaction.push(u)
        }
    }
    graph.erase(v)
}
```

Fig. 3. Algorithms of the dependency tracker and executors.

A backup reads the writeset log from the primary, parses the log entries using the *writeset parser*, and uses the *parallel executor* to apply changes to the database tables concurrently while respecting the transaction dependency provided by the *dependency tracker*. The parallel executor and the dependency tracker cooperate on a transaction dependency graph G. Graph G is a dynamically changing directed acyclic graph (DAG), in which each vertex denotes a transaction, and an edge from vertex T_1 to vertex T_2 indicates that transaction T_2 depends on T_1 and must be executed after T_1 completes (as in the commit order on the primary recorded on the log). For each transaction T in the log, the dependency tracker adds a new vertex v to the dependency graph G. It also adds edges linking v to transactions that it depends on based on T's writeset. A thread in the parallel executor takes a vertex with no incoming edges to replay, and removes that vertex and its edges from G when completed. The algorithms are shown as pseudo code in Figure 3.

C. Consistency and Reading on Backup

It is obvious that KuaFu ensures *write consistency* in that (i) for each row a backup applies the same sequence of updates in the same order, and (ii) when log replay completes, the backup reaches the same state as the primary. If all reads are served by the primary as well, then KuaFu provides the same consistency semantics as the original replication mechanism that uses serialized replay. This is no longer the case if we allow read transactions to be served on a backup.

In the original replication mechanism with serialized replay, the state on a backup might lag behind because the backup has not managed to replay all transactions that have been committed on the primary. In order to read from backups, clients must be able to tolerate such relaxed consistency. As a running example, assume that the primary has committed transactions T_1, T_2 and T_3 in that particular order. When transactions are replayed in serial, a backup will apply updates in the same order as they are committed on the primary. The state on a backup might reflect T_1 and T_2 only, while the last transaction T_3 is not yet replayed. The backup is therefore only *prefix consistent* with the primary in that reading from backup will get a prefix of all the updates committed on the primary.

Executing transactions in parallel on backups introduces *read inconsistency* that violates the original prefix consistency semantics. Using the same example, if T_1 and T_2 are independent, T_2 might be applied before T_1 on the backup. A read operation might see T_2 but not T_1, which can never happen on the primary. Such inconsistency is temporary until the backup finishes replaying T_1.

Such read inconsistency introduced by concurrent replay on a backup might be undesirable. The *consistent reader* in Figure 2 restores the consistency semantics as provided in the original replication mechanism. It does so by placing a barrier for every n updates such that all updates before a barrier must have been replayed before any of the updates after the barrier can start. We leverage MVCC (Multi-Version Concurrency Control), a mechanism available in many modern databases, to take snapshots at the barriers and only serve reads against the latest snapshot. The snapshots are guaranteed to include a prefixes of all the updates in the primary. By doing so, KuaFu sacrifices performance and freshness for consistency.

IV. Implementation

We have implemented KuaFu on MySQL 5.5.15, which has a layered architecture as in Figure 2 and supports several different storage engines. Our implementation uses the default transactional storage engine InnoDB. It supports row-level locking and multi-version concurrency control (MVCC).

MySQL 5.5 has built-in replication support based on primary/backup with log shipping. It supports both statement-based logging which records SQL statements, and row-based logging, which records row modifications and other operations. Our implementation directly uses the row-based log as the writeset log, making it unnecessary to implement writeset extractor. For other database systems, Plattner et al. [5] showed how to design and implement a writeset extractor.

In MySQL's row-based log, each operation is called a log event. Row changes are recorded as row events. The log also contains other events, such as those identifying the start and the end of a transaction. Row events belong to the same transaction are enclosed by the `begin` and `commit` events, enabling us to identify transaction boundaries easily. There are three types of row events, namely `insert`, `update` and `delete`. `Insert` events record the images of the rows being inserted. `Update` events record the *before images* of the rows

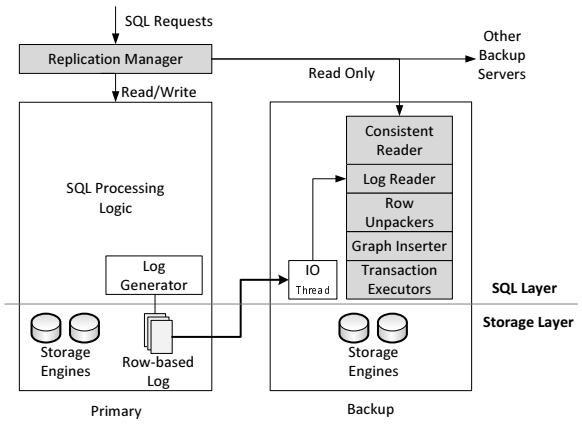

Fig. 4. KuaFu implementation on MySQL.

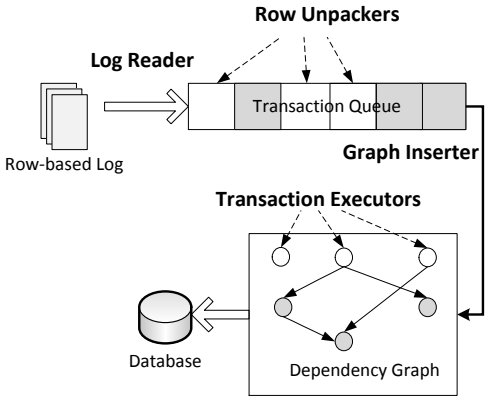

Fig. 5. Concurrent log replay in KuaFu.

being updated as well as the resulting *after images*. `Delete` events record the images of the rows being deleted.

A primary writes the updates to the log when executing transactions. A backup then fetches the log entries from the primary and replays the events in the log. In MySQL, a backup has two threads: the *IO thread* is dedicated to reading the log entries from the primary and storing them on local disk, while the *replay thread* reads the log from local disk and replays these events. Because only a single thread is in charge of parsing and executing the log, log events are executed consecutively regardless of whether it is a row event or other type of event. When a row event is parsed, it contains the type of the operations and the row images. These row images are stored as raw bytes. Before performing the operations, the replay thread must "unpack" these bytes into fields, which requires knowledge of the relevant table schema. Unpacking is actually a bottleneck for KuaFu, as we will see in the next sections.

A. Implementation Overview

Figure 4 shows an overview of the KuaFu implementation, corresponding to the design in Figure 2.

The implementation of the replication manager is straightforward: it analyzes incoming requests, sends the write requests to primary, and sends read requests to primary or backup servers. It keeps track of the version numbers of the latest snapshots of each of the backup servers, so that it can dispatch the read requests to the right backup servers to satisfy freshness constraints.

As mentioned before, since MySQL is able to generate row-based logs, there is no need to re-implement a writeset extractor. We did not modify the code of MySQL for the primary.

A KuaFu backup uses the same IO thread for fetching log entries. After being read, log events need to be unpacked and examined before being sent to multiple threads for replay. Unpacking row images is actually quite expensive and will become a bottleneck if the work is carried out by a single thread. We parallelize the unpacking using multiple *row unpacker* threads. We then use a dedicated thread to insert the transactions into the dependency graph. Therefore we have five different types of replay-related threads on a backup: the IO thread, the log reader thread, the row unpacker threads, the graph inserter thread, and the transaction executor threads.

We also implemented a *consistent reader* on the backups to take snapshots on a backup and serve all reads with the latest snapshot. KuaFu leveraged InnoDB's multi-version system to reduce the cost of taking snapshots.

B. Replay-related Threads

Figure 5 illustrates how KuaFu replays log concurrently on a backup using the five different types of threads. The log reader thread parses log into log events. It packs row log events into transactions and pushes them into a FIFO *transaction queue*. Other types of events are executed directly in the log reader. There is only a single log reader thread, because reading the log and constructing event objects is I/O bound.

When transactions are pushed into the transaction queue, they are originally marked as *packed*. Row unpacker threads are used to unpack these rows. The job of a row unpacker is to search for a packed transaction in the transaction queue, unpack it, and then mark it as *unpacked*. Row unpackers always search for unpacked transactions sequentially, starting from the front of the FIFO queue.

The graph inserter takes the front item of the transaction queue after it is marked as unpacked, and inserts it into the *dependency graph*. We use a single graph inserter to guarantee that transactions are dequeued one by one and inserted into the dependency graph in the enqueue order. We will discuss dependency graph maintenance in detail in the next section.

The transaction executor threads take the transactions that do not depend on any other transactions from the graph, and execute them. After a transaction commits, the corresponding vertex is removed from the dependency graph, and the outgoing edges are also removed.

In addition to row events, the row unpacker and transaction executor threads need to handle "USE DATABASE" and "TABLE MAP" events as well. Row unpacker threads execute these events in order to get table schemas to unpack the row images, and transaction executors execute them before

performing row operations. Therefore, we execute these operations twice, which incur extra overhead. Fortunately, the overhead is largely negligible. KuaFu must also handle log events that alter a table schema or a database layout. We need to make sure transactions committed before those events are not affected by them. Because they are rare operations, KuaFu treats them as barriers and simply waits for all previous events to finish before executing them.

C. Inserting Transactions into the Dependency Graph

Graph inserter is responsible for inserting transactions into the dependency graph. To do this, it needs to infer transaction dependencies. A transaction T_i depends on T_j if T_i proceeds T_j in the log, and T_i is in conflict with T_j. Because the transactions are enqueued and dequeued in the same order, the order in which the transactions are inserted into the graph are guaranteed to be the same as their order in the log. Detecting whether two transactions are conflicting involves checking whether they modified the same row. Assuming B_i is the set of before images of the rows modified by transaction T_i, and A_i is the set of after images, then T_i is conflicting with T_j if $B_i \cap A_j \neq \emptyset$ or $B_j \cap A_i \neq \emptyset$. Therefore it is easy to decide whether two transactions conflict if we can detect whether two row images are the same. If the table has a primary key, then row equality can be determined easily by comparing the primary keys. To avoid comparing the full row images, KuaFu computes signatures for the rows without primary keys and uses them to check row equality.

Each time a transaction is inserted into the graph, it must be compared with all the existing transactions in the graph. The time to insert a transaction is proportional to the number of vertices in the graph. If the graph contains too many vertices, it may take too much time for the graph inserter to insert transactions. In our current implementation, we limit the max number of vertices in the graph to be twice the number of transaction executor threads, and throttle the graph inserter when the number of vertices in the graph hits the limit.

D. Reading on Backup

To restore the consistency semantics when we allow reads on backups, KuaFu takes snapshots periodically. Our current implementation takes snapshots for every N transactions, where N is a predefined constant. It is trivial to modify the system to take snapshots with fixed time intervals. In order to make sure that a snapshot is valid, we need to make sure that, when a snapshot is taken, there exists some transaction T_i, such that all the transactions preceding T_i (including T_i) have been replayed, while no transaction following T_i has been replayed. To achieve this, KuaFu creates a barrier in the log reader, stopping it from reading more log events, and waits until the transaction queue and dependency graph are empty. At this time, KuaFu can be sure that all the transactions processed by the log reader have been committed. KuaFu then takes a snapshot and marks the version number of this snapshot as the version of the last transaction seen by the log reader.

Component	Lines of code	Modified files
Log Reader	107	1
Row Unpacker	539	3
Graph Inserter	112	1
Transaction Executor	147	1
Read Support	150	4
Miscellaneous	397	2
Total	**1452**	**11**

TABLE I
IMPLEMENTATION EFFORTS FOR THE COMPONENTS OF KUAFU.

	CPUs	cores	hw threads	DRAM	network
M8	1	4	8	24GB	1Gbps
M48	4	48	48	96GB	1Gbps

TABLE II
MACHINES USED IN THE EXPERIMENTS.

In MySQL, taking a snapshot is an internal mechanism of the storage engine and is hidden from the SQL layer. We modified the interface between the SQL layer and the storage engines, exposing the snapshot related APIs. In particular, we expose two functions in the interface: taking a snapshot, and assigning a snapshot for a transaction. When taking a snapshot, KuaFu invokes the snapshot function at the SQL layer. When serving a read request on the backup, KuaFu assigns the latest snapshot to the request.

E. Implementation Effort

Table I shows the components, the numbers of lines of code used to implement them, and the number of files we modified.

We have modified a total of 1,452 lines of code, a relatively small modification considering the complexity of the MySQL code base. This validates our design choices, which is practical even for a widely used production database system.

V. EVALUATION

A. Experiment Setup

To evaluate the performance of KuaFu, we have conducted experiments on two types of machines, namely M8 and M48. The configurations of these machines are shown in Table II. M8 machines support hyper-threading, with two hardware threads on each core, while the M48 machines have only one hardware thread for each core. In our experiments, the M48 machines are used for testing the scalability of MySQL and KuaFu, while the M8 machines are used for micro-benchmarks. All machines run Windows Server 2008 R2.

An OLTP workload and a set of micro-benchmarks are used in the tests. The OLTP workload is a TPC-C-like workload, with no limitations on the clients' maximum operation rate. It uses 100 warehouses with a total of about 10GB of data. The transactions in this workload is write intensive, each includes at least one INSERT, UPDATE or DELETE operation. Different transactions might be in conflict with each other in this workload. Aside from the OLTP workload, we also use micro-benchmarks to gain insights into the system. The micro-benchmarks use a single table with a size of about 100MB.

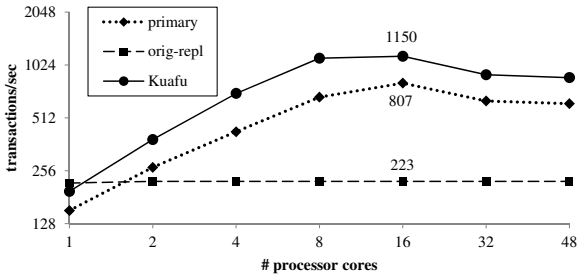

Fig. 6. Throughput of the primary server (primary), the replay throughput of original replication (orig-repl), and replay throughput of KuaFu (KuaFu) on M48, using different numbers of cores on an OLTP workload.

Different operations are performed on the table in different micro-benchmark tests.

Unmodified MySQL 5.5.15 is used on the primary servers, and a modified version is used on the backups. Both versions of MySQL are compiled with Visual C++ 2010. During the tests, the servers are configured with a 16GB buffer, and they flush log files whenever a transaction is committed. For all experiments, we warm the memory cache by performing a checksum on all the tables. The machines are connected with Gigabit Ethernet. In our experiments, the network is not a bottleneck. We observed that enabling logging on the primary server introduces less than 2% overhead, which indicates that disk IO is not a bottleneck, either.

In the following experiments, we first show the overall performance of MySQL and KuaFu on the OLTP workload. We then analyze the overhead due to read-related barriers. We further use micro-benchmarks to test the impact of conflicting transactions on the performance of primary and backup servers. Finally, read performance on the backup servers is presented using the micro-benchmark workload.

B. Scalability and Replay Efficiency

In this subsection, we present the experimental results on the scalability of the primary and the backup servers. We use the OLTP workload in these tests to simulate real-world scenarios. In order to push the scalability of the servers to the limit, we used the 48-core M48 machines. We measure the performance of the servers in terms of transactions per second.

Figure 6 shows the performance of the systems on the M48 machine. There are three sets of data shown in the graph. *Primary* shows the throughput when we execute the OLTP workload on the primary server, with no backups. The *orig-repl* line shows the replay throughput of the original replication mechanism in MySQL, using row-based logging. In this experiment, we first run the primary alone, and then replay the log generated on the backup. The replay throughput can then be obtained by dividing the number of transactions in the log by the time it takes for the backup server to replay the log. The *KuaFu* line shows the throughput of KuaFu. To show how the performance varies with number of cores used, we bind MySQL processes to specified processor cores. We also adjust the number of threads used by MySQL accordingly.

Figure 6 shows that the primary server scales well until the number of cores reaches 16. After 16 cores, the throughput falls. Further investigation reveals that this is due to the high cost of inter-cpu-socket locks. The M48 machine has 4 CPUs, each with 12 cores. When we use less than 12 cores, we can bind all the threads to the cores on the same CPU. However, if we use more than 12 cores, we have to use at least 2 CPUs, which means that the locks are now inter-socket. Our profiling results show that the inter-socket locks have poor performance on this machine. As a result, the time used in locks increases dramatically when we scale from 8 threads to 16 threads. For the same reason, KuaFu does not scale beyond 16 cores on M48 either.

A Backup using the original MySQL replication scheme has a throughput about twice the primary when a single core is used. The OLTP workload has transactions composed of both read and write operations. The primary server has to parse and process all the SQL queries, while the log only records the resulting row updates. Replaying the log is faster than fully executing the corresponding queries, making backup faster than the primary when both use a single processor core. Unfortunately, the original replication scheme does not scale. With two cores, a backup can just barely catch up with the primary. Backup begins to fall behind when the primary uses more than four cores.

KuaFu performs slightly worse than the unmodified backup when a single core is used. This is because KuaFu has some extra overhead over the original scheme. The main sources of overhead are data communication and thread synchronization among the log fetcher, the graph inserter, and the transaction executor threads. On the positive side, KuaFu shows good scalability up to 16 cores. Most importantly, a backup in KuaFu always outperforms the primary given the same number of processor cores. This means that, unlike the original scheme, in KuaFu the backups will always be able to catch up with the primary on the same hardware, and will not be a bottleneck for the whole system.

C. Impact of Read-related Barriers

In order to provide read consistency on a backup, KuaFu places barriers between log operations. When a backup hits a barrier, it executes all the transactions before this barrier, takes a snapshot, and then proceeds with the transactions after the barrier. Read requests on a backup are served on the most recent snapshot in order to provide the same prefix consistency as in the original replication scheme.

Frequency of the barriers can affect both read and replay on a backup. The more frequently we place barriers, the fresher the snapshots. On the other hand, barriers limit the number of transactions that can be executed in parallel. They also force the system to wait for the slowest thread, introducing extra overhead. In our current implementation, we place a barrier after every n transactions, where the parameter n is adjustable. It is also easy to put barriers every m millisecond and to impose a hard limit on the latency between snapshots.

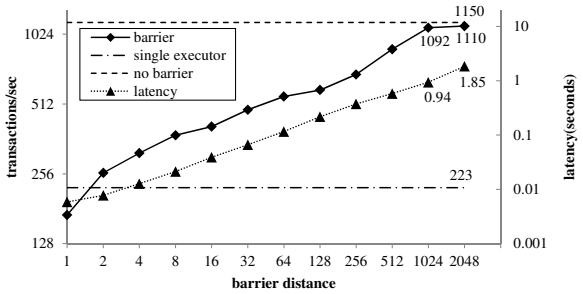

Fig. 7. Replay throughput with different barrier distance (number of transactions between two barriers).

In this section, we test the performance impact of the barriers. We vary the number of transactions between two barriers, and measure the replay throughput. The same workload and test methodology as in the scalability experiments are used. We use 16 cores of the M48 machine, the configuration that offers the best performance.

In Figure 7, the line labeled *barrier* shows the replay throughput when we vary the number of transactions between barriers. For reference, two other sets of data are presented. The *no barrier* line shows the replay throughput when there is no barrier. And the *single executor* line shows the replay throughput when there is a single *transaction executor* thread with no barrier. The *latency* line is the calculated values of the latency between two snapshots.

When the barrier distance is 1, i.e., there is a barrier after each transaction, the replay performance is about 30% lower than the *single executor* case. As the number of transactions between two barriers increases, the barrier overhead is amortized among the transactions. When there are more than 1024 transactions between barriers, the overhead becomes negligible: the *barrier* line gradually approaches the *no barrier* line.

D. Effect of Conflict Ratio

Conflicting transactions cannot be replayed in parallel. The conflict ratio of the transactions can significantly affect the performance of KuaFu.

Because we cannot change the conflict ratio of the OLTP workload, we use a micro-benchmark to test the effect of conflict on replay performance. In this benchmark, we send queries with varying conflict ratios and record the performance of the primary and the backup. The table is divided into *row groups*, each containing 10 rows. Fifty clients continuously generate transactions, each updating all the rows in a random row group. When there is only a single row group, all the transactions are in conflict with each other, and with two row groups, each transaction will be in conflict with half of the other transactions, and so on. The reason we make transactions work on a group of rows instead of a single row is to generate enough work in each transaction to amortize the overhead of BeginTransaction/EndTransaction.

Figure 8 shows the throughput of primary and the backup when different numbers of row groups are used. The experiment was conducted on the M8 machine, using 8 hardware threads. The result shows that the throughput of the backup is always higher than the primary, no matter what the conflict ratio is. With only a single row group, all the transactions are in conflict with each other, thereby forcing the primary into serial execution. Worse, some transactions might get aborted and retried due to contention. On the backup, transactions are found to be dependent and hence only one transaction is executed at a time, avoiding contention between threads. As a result, when there is only a single row group, the backup outperforms the primary by a wide margin. The conflict ratio is reduced as more row groups are added, and the primary's throughput improves. Nevertheless, the throughput of backup grows as well. Even at the other extreme, when the workload contains no conflicting transactions (marked as *NC*, achieved by assigning different portions of the table to different clients), the throughput of backup remains higher than that of the primary.

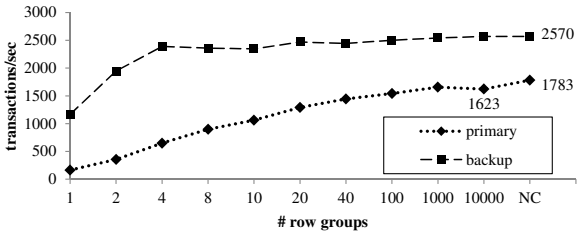

Fig. 8. Throughput with different transaction conflict ratio. The more row groups used, the less conflict; NC stands for no conflict.

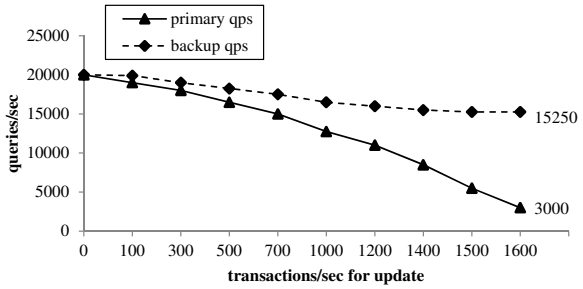

Fig. 9. Read performance of a two-machine system with different update rates.

E. Read Performance

In this experiment, we test the read performance of the system under different update rates. We use the micro-benchmark with 50 clients updating the table continuously and another 50 clients sending read queries at the same time. We use a system composed of a primary and a backup, each with the same hardware configuration as the M8 machine.

Figure 9 presents the numbers of queries per second that the servers are able to serve as we vary the update rates on the primary. The result shows that the read performance on both the primary and the backup decreases as the update rate increases. When there are no updates, the primary and the backup can both serve 20,000 read requests per second. When there are update requests, the backup is able to serve

more reads than the primary, because replaying an update on the backup incurs only about half the workload as it takes to execute the query on the primary. Moreover, read queries on the primary conflict with update transactions, leading to further slowdown. This argues for the need to dispatch read requests to the backups.

VI. Related Work

Replication in database systems is a rich field and has been carefully examined by many researchers over the years. We will cover only the most relevant work in this section. A well-known paper by Jim Gray [6] proposes to categorize database replication approaches in two dimensions. The first dimension is where updates take place. Different approaches in this dimension includes *primary copy* (also called primary/backup or master/slave) and *update everywhere*. KuaFu focuses on the primary/backup class of replication. The second dimension distinguishes between *eager* and *lazy* replication approaches, where eager replication keeps all replicas synchronized by updating all the replicas as part of the transaction, while lazy replication propagates updates to other replicas after the transaction commits. KuaFu can be applied to both.

Replication by Log Shipping. Primary/backup replication is typically implemented via log-shipping [7], [8]. Many primary/backup schemes for replicating databases have been proposed [4], [5], [9], [10], [11], [12]. Plattner et. al [4], [5] proposed Ganymed, a primary/backup scheme for replicating snapshot isolated databases in clusters of machines. In Ganymed, a primary handles all update transactions, while read-only transactions are handled by backups. Transactions are serialized at the commit time on the primary and their writesets are extracted for updating. However, during update propagation, transactions commit serially at the backups in order to guarantee that the backups converge to the same state as the primary. Many popular database systems use a similar mechanism to implement replication. KuaFu is designed to address the parallelism gap in this setting.

Daudjee et.al [9] proposed a lazy database replication scheme that guarantees *strong session one-copy serializable (1SR)*. Their algorithms can be applied directly in KuaFu to guarantee strong session 1SR if the original database system guarantees serializability. They further proposed another scheme to achieve snapshot isolation in a lazily replicated database systems. If the underlying DBMS has snapshot isolation [13], then their scheme can not only ensure *Strong Session Snapshot Isolation*, but also apply updates on the backups concurrently. KuaFu eliminates the requirement of snapshot isolation by automatically inferring transaction dependency. KuaFu requires that the storage engine take snapshot in a backup to support consistent read on a backup.

Replication via Consensus. Consensus protocols such as Paxos [14] can be used to build a replicated state machine for database systems. Traditional state-machine replication suffers from the same tension between parallelism (for performance) and serialized execution (for consistency). KuaFu can be extended to address this tension for state-machine replication. Unlike in the primary/backup replication with log shipping, where transactions are committed on the primary first before forwarded to the backups, replicas must reach a consensus in state-machine replication before a transaction is committed. This can be done in the same way as in Tribble [15], with the mechanism in KuaFu used for record and (parallel) replay: a replica acting as a *leader* execute the transactions to generate the logs, while the others replaying the logs. In contrast, Tribble adopts a general approach of recording the dependencies of requests by logging their synchronization operations and preserving their orders during replay, which incurs high overhead for most database systems that use light-weight locks extensively.

The main complication of extending KuaFu to state-machine replication is related to leader changes (or even multiple leaders in some cases). Although the consensus protocol ensures consistency, a demoted leader might need to roll back transactions that the replicas fail to reach consensus on, while ensuring that the effects of those transactions are never exposed. Such rollbacks are possible with the writeset logs, but KuaFu has not implemented this feature yet.

M. Kapritsos et. al. [16] advocated the notion of execute-verify for replicated state machine on multi-core servers. This approach allows a backup to execute requests in parallel speculatively and to roll back in case of state divergence. This technique works best for workloads with low conflict ratio.

Replication at Different Layers. Primary/backup replication can be implemented at different layers of a database system, including the SQL-statement layer, the table-row layer and the virtual-machine layer.

Thomson et. al [11] presents a transactional database execution model which guarantees equivalence to a predetermined serial execution. Their approach can be adopted in databases to implement replication on the SQL statement layer. Database replicas running exactly the same database software with the same initial state and receiving identical sequences of SQL requests will have identical state. However, as noted by the authors, the deterministic scheme lowers throughput when there are long-running transactions.

RemusDB [12] provides high availability of database systems by running database systems on virtual machines and replicating the virtual machines. This method effectively puts replication on the memory and disk layer. Such replication scheme removes the dependency on database systems, at the expense of increased communication cost.

Deterministic Replay and Execution. KuaFu is also related to deterministic record and replay because a backup is essentially replaying the log from the primary to reach an identical state. Deterministic recording and replay has attracted a lot of attentions recently in the systems community [17], [18], [19], [20]. Although in theory, a generic record and replay tool can be used by treating a database system as a multi-threaded program, the overhead tends to be high. KuaFu is designed specifically for database systems. It leverages writesets to

track dependencies among transactions, and can thus avoid recording a massive amount of low-level information such as thread interleaving.

Deterministic execution by Jimenez-Peris et. al. [21] and Wenbing et. al. [22] proposed a deterministic thread scheduler to make sure that each replica executes transactions with multiple threads, whereas our work does not control thread scheduling and targets database replication specifically.

Database Replication in the Industry. Due to performance concerns, most commercial database systems, such as Oracle 11g [23], IBM DB2 [24] and Microsoft SQL Server 2008 [3] use log-shipping to implement replication. In the open source community, two of the most popular database systems, MySQL [25] and PostgreSQL [26] both use log-shipping as their built-in replication solution. KuaFu focuses specifically on this form of log-shipping based replication.

Ad-hoc approaches do exist to close the parallelism gap. Among them, sharding, also called data partitioning, is the most widely accepted [27]. In this approach, a data set is divided into several database instances to get concurrency at the database level. There are also proposals to leverage the group-commit capability of a database system by replaying the group committed transactions in parallel [28]. Other proposals include speculative execution of logged transaction, and configuring backups with better hardware [1] [2]. These approaches are largely orthogonal to KuaFu, as KuaFu tackles the problem directly by executing non-conflicting transactions in parallel on a backup.

VII. Conclusion

Concurrency for performance and replication for reliability are driven by the prevalence of multi-core architecture and the wide-spread use of commodity-machine clusters. Yet the tension between the two creates a significant practical problem of the parallelism gap. A serialized execution on backups limits the throughput of highly concurrent transaction processing on existing database systems. By making a set of important design choices, KuaFu provides a simple, effective, and practical solution to close the parallelism gap. It involves a small number of changes to the existing database systems, requires no changes to the non-replicated database systems, and is shown to close the gap on a real database system while preserving the consistency semantics.

References

[1] B. Schwartz, "MySQL limitations part 1: Single-threaded replication," Website, 2010, http://www.mysqlperformanceblog.com/2010/10/20/mysql-limitations-part-1-single-threaded-replication.

[2] P. Zaitsev, "Fighting MySQL replication lag," Website, 2009, http://www.percona.com/ppc2009/PPC2009_Fighting_MySQL_Replication_Lag.pdf.

[3] K. Delaney, P. Randal, and K. Tripp, *Microsoft SQL Server 2008 Internals*. Microsoft Press, 2009.

[4] C. Plattner and G. Alonso, "Ganymed: Scalable replication for transactional web applications," in *Proceedings of the 5th ACM/IFIP/USENIX International Conference on Middleware*, 2004, pp. 155–174.

[5] C. Plattner, G. Alonso, and M. T. Özsu, "Extending DBMSs with satellite databases," *The VLDB Journal*, vol. 17, pp. 657–682, 2008.

[6] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, 1996, pp. 173–182.

[7] C. A. Polyzois and H. García-Molina, "Evaluation of remote backup algorithms for transaction-processing systems," *ACM Transaction on Database Systems*, vol. 19, pp. 423–449, 1994.

[8] R. P. King, N. Halim, H. García-Molina, and C. A. Polyzois, "Management of a remote backup copy for disaster recovery," *ACM Transaction on Database Systems*, vol. 16, pp. 338–368, 1991.

[9] K. Daudjee and K. Salem, "Lazy database replication with ordering guarantees," in *Proceedings of the 20th International Conference on Data Engineering*, 2004, pp. 424–435.

[10] D. Khuzaima and S. Kenneth, "Lazy database replication with snapshot isolation," in *Proceedings of the 32nd International Conference on Very Large DataBases*, 2006, pp. 715–726.

[11] A. Thomson and D. J. Abadi, "The case for determinism in database systems," *Proceedings of VLDB Endowment*, vol. 3, pp. 70–80, 2010.

[12] U. F. Minhas, S. Rajagopalan, B. Cully, A. Aboulnaga, K. Salem, and A. Warfield, "RemusDB: Transparent high availability for database systems," in *Proceedings of the 37th International Conference on Very Large DataBases*, vol. 4, 2011, pp. 738–748.

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

[14] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems*, vol. 16, pp. 133–169, 1998.

[15] Z. Guo, C. Hong, M. Yang, L. Zhou, and L. Zhuang, "Paxos made parallel," Microsoft Research Asia, Tech. Rep. 118, 2012.

[16] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin, "All about Eve: Execute-verify replication for multi-core servers," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012, pp. 237–250.

[17] G. Altekar and I. Stoica, "ODR: Output-deterministic replay for multi-core debugging," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009, pp. 193–206.

[18] G. W. Dunlap, D. G. Lucchetti, M. A. Fetterman, and P. M. Chen, "Execution replay of multiprocessor virtual machines," in *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2008, pp. 121–130.

[19] M. Xu, R. Bodik, and M. D. Hill, "A "flight data recorder" for enabling full-system multiprocessor deterministic replay," in *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003, pp. 122–135.

[20] D. Lee, B. Wester, K. Veeraraghavan, S. Narayanasamy, P. M. Chen, and J. Flinn, "Respec: Efficient online multiprocessor replay via speculation and external determinism," in *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, 2010, pp. 77–90.

[21] R. Jimenez-peris, M. Patino-Martłnez, S. Arevalo, and J. Carlos, "Deterministic scheduling for transactional multithreaded replicas," in *Proceedings of the IEEE 19th Symposium on Reliable Distributed Systems*, 2000, pp. 164–173.

[22] W. Zhao, L. E. Moser, and P. M. Melliar-Smith, "Deterministic scheduling for multithreaded replicas," in *Proceedings of the 10th IEEE International Workshop on Object-oriented Real-time Dependable Systems*, 2005, pp. 74–81.

[23] K. Deshpande, *Oracle Streams 11g Data Replication*. McGraw-Hill, 2008.

[24] L. J. Gu, L. Budd, A. Cayci, C. Hendricks, M. Purnell, and C. Rigdon, *A Practical Guide to DB2 UDB Data Replication v8*. IBM Corporation, 2002.

[25] *MySQL 5.5 Manual*, http://dev.mysql.com/doc/refman/5.5/en/replication-howto.html.

[26] *PostgreSQL 9.1.3 Documentation*, http://www.postgresql.org/docs/9.1/interactive/high-availability.html.

[27] L. Soares, "Feature preview: The multi-threaded slave," Website, 2011, http://d2-systems.blogspot.com/2011/04/mysql-56x-feature-preview-multi.html.

[28] Knielsen, "Parallel replication of group-committed transactions," Website, 2011, http://askmonty.org/worklog/Server-RawIdeaBin/?tid=184.