Building Deterministic Transaction Processing Systems without Deterministic Thread Scheduling

Alexander Thomson
Yale University
thomson@cs.yale.edu

Daniel J. Abadi
Yale University
dna@cs.yale.edu

ABSTRACT

Standard implementations of transactional systems such as database systems allow several sources of nondeterminism to introduce unpredictable behavior. The recent introduction of an architecture and execution model that isolates sources of nondeterministic behavior in online transaction processing systems in order to yield deterministic transaction results makes active replication easier and mitigates major scalability barriers. We observe here that (a) this approach would nicely complement other determinism techniques in the assembly of a fully deterministic application stack and (b) the approach does not rely on any special thread-scheduling machinery or deterministic concurrency primitives and even benefits from the nondeterminism inherent in typical OS schedulers.

1. INTRODUCTION & BACKGROUND

Transactions are a useful programming abstraction; transactional middleware systems such as database management systems (DBMSs) and software transactional memory (STM) libraries are ubiquitous components in a wide variety of types of applications. Although sometimes unwieldy in their implementations, such systems represent powerful tools for application developers: program complexity can often be vastly reduced when middleware layers can be made to do a large amount of an application's heavy lifting with regard to data storage/formatting and concurrency control tasks. Unfortunately, standard implementations of transactional systems—and database systems in particular—allow nondeterministic behavior to arise.

The ACID properties, which represent the gold standard of transactionality (and isolation in particular) demand serializability, i.e. the equivalence between actual (potentially highly concurrent) execution of a set of transactions and some sequential execution of those same transactions.

For example, given a set of transactions $\{T_1, T_2, T_3\}$, suppose T_1 and T_3 conflict on a shared resource R_1 (i.e. they both access this resource in a non-commutative way, such as by setting its value if the resource is a variable) while T_2 and T_3 conflict on a resource R_2, but T_1 and T_2 do not conflict with one another at all. Then serializability permits several possibilities:

- execute T_1, T_2 and T_3 sequentially in any order
- execute T_1 and T_2 concurrently, and execute T_3 after they have both completed
- execute T_3 first, and execute T_1 and T_2 concurrently after it has completed

The first possibility above is by definition equivalent in outcome to itself and therefore to a sequential execution, thus satisfying serializability. Since transactions T_1 and T_2 do not conflict—and therefore commute—the second option is equivalent both to the sequential execution T_1-then-T_2-then-T_3 and to the sequential execution T_2-then-T_1-then-T_3, while the third one is equivalent to both T_3-then-T_1-then-T_2 and T_3-then-T_2-then-T_1. Since these all satisfy the ACID properties, typical DBMS implementations employ concurrency control methods that are agnostic to *which* of these executions actually occurs. The serial order to which execution is equivalent is therefore left to the mercy of a wide variety of potentially nondeterministic factors, including OS-level thread-scheduling, possible cache misses and page faults, the possibility of hardware failures, variable network latency if the system is partitioned and/or replicated across multiple physical machines, deadlock detection and resolution mechanisms, and so forth.

1.1 A deterministic transactional system

Our recent paper, *The Case for Determinism in Database Systems* [7], outlines a distributed database system architecture and execution model based closely on two-phase locking that, given a global ordering of transactions as input, executes them in a manner equivalent to *that specific* order while still maintaining high concurrency. We accomplish this by implementing a concurrency control mechanism resembling standard two-phase locking, but with three added invariants:

- **Ordered locking.** For any pair of transactions T_i and T_j which both request locks on some record R, if T_i appears before T_j in the global ordering then T_i must request its lock on R before T_j does. Further, the lock manager must grant each lock to requesting transactions strictly in the order in which those transactions requested the lock.
- **Execution to completion.** Once entering the system, each transaction must go on to run to completion—until it either commits or aborts *due to deterministic program logic*. Thus even in cases where a transaction is delayed for some reason (e.g. due to network latency or hardware failure), the transaction *must* be kept active until either it executes to completion or the replica is killed—even if other transactions are waiting on locks held by the blocked one.

In practice, ordered locking is most easily implemented by requiring transactions to request all locks they will need in their lifetimes immediately upon entering the system. For certain classes of transactions, it may be impossible to know at the time a transaction enters the system exactly which locks it will acquire, because the results of accessing resources early in the transaction may determine what resources the transaction subsequently uses. For example, database management systems are expected to handle transactions that update a record only after locating it by performing a lookup in a secondary index. We handle this case using a technique called Optimistic Lock Location Prediction (OLLP). In this scheme, the transaction is decomposed into two separate transactions:

- the *reconnaissance unit*, which performs all index lookups required to discover the original transaction's full set of required locks (but doesn't cause any actual state changes to be made), then reports this lock set back to the system component tasked with choosing the global order (which can be either the application layer or a special preprocessor in the database system)
- the *execution unit*, which (aided by the reconnaissance knowledge) deterministically executes the original transaction if the indexes have not meanwhile been updated in a manner that changes its read-write set (or if they have, it repeats the reconnaissance step to be rescheduled later)

Conveniently, secondary indexes tend to be updated much less frequently than they are read (especially since the most common data structures used in index implementation—B+ trees—are not designed to handle extremely high update rates), and OLLP therefore seldom results in repeatedly restarting the same transaction.

1.2 Performance & applications

Evaluations of a prototype system implementing deterministic and traditional transaction execution methods side-by-side in the same framework have shown that deterministic transactional execution keeps up with traditional execution so long as one condition is met: there must be no long-running transactions executed at high isolation levels. This condition is increasingly satisfied under real-world deployment conditions, by virtue of two current trends in transaction processing workloads and systems:

- **Short transactions.** Today's real-world OLTP workloads overwhelmingly consist of many short updates—typically implemented as compiled stored procedures rather than sent ad hoc as SQL statements—executing at high isolation levels, with only occasional longer (lower-isolation) queries.
- **No disk stalls.** A typical disk seek takes 5-15ms—a near eternity in an environment in which today's blazing fast processors and monstrous caches often eat through transactions in under 10μs once relevant data has been fetched.[1] Fortunately, today's OLTP data sets increasingly fit entirely in the main memory of a single high-end server—or a cluster of commodity machines.

Deployed in a main-memory-only setting and under typical OLTP workloads, our deterministic transaction execution model comes with three significant benefits *in addition to* the debugging and repeatability perks that typically accompany deterministic behavior:

- **Unsynchronized active replication.** Given identical inputs, deterministic database system replicas will maintain strong consistency (meaning that replica states do not diverge, nor are some replicas forced to lag behind others) without incurring the latency-consistency-overhead tradeoffs inherent in other replication schemes such as log shipping, eventual consistency, and synchronized active replication [5,6,3].
- **No distributed commit protocols.** In existing (nondeterministic) *partitioned* database systems, transactions spanning multiple partitions require a multi-phase commit protocol to ensure atomicity against the possibility of nondeterministic abort at a subset of involved nodes. Since locks must be held for the full duration of this protocol, this necessity sharply drives up lock contention between transactions. The result is that these commit protocols are the primary hurdle to achieving outward scalability of distributed databases on clusters of commodity machines. Our deterministic execution model allows these distributed commit protocols to be shortened—and in many cases omitted completely—sharply mitigating this cost, and therefore promising significant scalability gains. This is because nondeterministic events (such as node failure) cannot cause a transaction to abort (failure is dealt with through on-the-fly failover to an active replica or deterministic recovery of state by replaying history from the input log), so all parts of the protocol that deal with failure or confirmation that the effects of a transaction has made it to stable storage are no longer needed.
- **No ARIES logging.** Careful studies of database system performance has shown that logging components of transactional systems cause at least 10% performance overhead (typically more) due to the physical nature of physical database logging algorithms such as ARIES, where every action that the database makes while performing a transaction is identified and logged to disk [4]. In deterministic database systems, only input must be logged (since database state can at all points be deterministically re-derived from an initial state and a record of subsequent input). Not only is logging input far less labor intensive than logging all

[1] The historical reason for the original design decision to allow nondeterministic behavior in database systems is in fact exactly this. Since transactions in the early days of database systems research were likely to stall for long periods of time during disk seeks and user stalls, many transactions would be actively executing at any given time, sharply driving up lock contention. In high-contention environments where some transactions cannot proceed because they are blocked on others that are waiting on a disk accesses, finding useful work to do on newer, potentially contention-free transactions—i.e. performing on-the-fly transaction reordering—is obviously an extremely useful tool. Early database systems therefore profited greatly from a serializability guarantee agnostic to which particular order is emulated.

transaction actions, but the input can even be logged prior to starting the transaction, allowing all logging overhead to occur before any locks are acquired, further reducing system contention. If lightweight checkpointing is used, system recovery can occur from replaying history from a checkpoint (or alternatively from cloning the state of an active replica).

It is important to note here that the *internal* behavior of this type of transaction processing system does not need to be fully deterministic for transactional output to be determined solely by its input. Our current prototype fully maintains its determinism invariant while running on a public cluster of off-the-shelf Linux workstations.

2. CONTRIBUTING TO A SAFER APPLICATION STACK

Since unintentional nondeterminism is the bane of multithreaded software development, fully deterministic environments are extremely desirable to programmers. Because most operating systems supporting real world application deployments take for granted a license to schedule threads and perform other tasks completely free of determinism guarantees, extremely few tools have been developed to support explicitly deterministic multi-threaded execution of any sort, and none (to our knowledge) support efficient deterministic transactional execution.

2.1 The alternatives, and why they're difficult

One very promising approach to quickly building deterministic application components is to run legacy (potentially nondeterministic) applications on operating systems sporting deterministic thread schedulers and other mechanisms for isolating potential sources of nondeterminism, such as dOS or Determinator [8,1]. These systems explicitly tag, log, and synchronize across replicas any nondeterministic events which appear as input to programs or can affect their behavior indirectly.

Almost all of today's database systems, however, make heavy use of nondeterministic resources such as the system clock (most notably in timeout-based deadlock detection mechanisms). While locally logging accesses to the system clock and other nondeterministic events incur prohibitive overhead in itself, in an actively replicated system these values would have to be synchronized across all replicas, which is extremely expensive—especially in the increasingly frequent case of replication over a wide area network.[2]

Furthermore, in the case where within each replica, data is partitioned across multiple physical machines (partitioning and replication are orthogonal characteristics of distributed database systems), all replicas must agree on the order in which any intra-replica messages are delivered—again a prohibitively high inter-replica communications overhead. In addition, because network latency is inherently nondeterministic, non-clock-based timeouts will not detect deadlocks deterministically, and the cost of cycle detection in waits-for graphs spanning multiple machines is much higher than when all transactions are local.

[2]Two alternatives to clock-based deadlock detection are (a) using timeouts built on a timer which measures progress in some deterministic task, such as a repeatedly incremented counter, and (b) performing explicit cycle detection on the waits-for graph. Both of these come with higher overhead than clock-based deadlock detection.

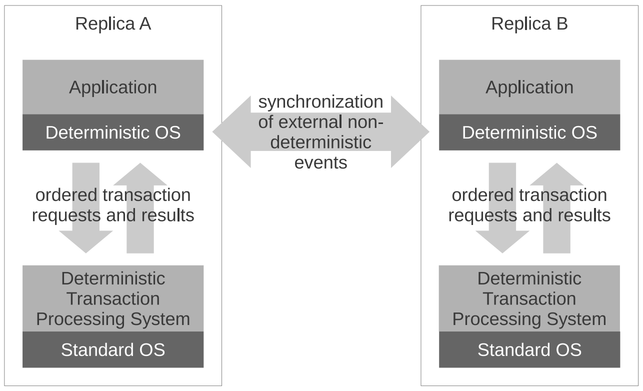
Replica A
Application
Deterministic OS
ordered transaction requests and results
Deterministic Transaction Processing System
Standard OS
synchronization of external non-deterministic events
Replica B
Application
Deterministic OS
ordered transaction requests and results
Deterministic Transaction Processing System
Standard OS

Figure 1: Architecture incorporating a deterministic transaction processing system serving a deterministic application.

Although it may be possible to engineer around these difficult problems, it is clearly a difficult problem to force a traditional transaction processing system to behave deterministically using only OS-level tools without severely hampering performance.

2.2 Where deterministic transaction processing fits in

A transaction processing engine designed to be explicitly deterministic would complement other determinism techniques nicely in a deterministic application stack.

Figure 1 illustrates an example architecture combining applications running on a deterministic operating system such as dOS or Determinator [8,1] with a deterministic transaction processing middleware layer like we propose. Since each instance of the transactional system accepts as input only an ordered set of transaction requests from the application—which will of course be identical between replicas of the deterministic application server—the transaction processing system replicas will not need to communicate with one another at all.

3. NONDETERMINISTIC SCHEDULING

As mentioned in section 1.1, our deterministic transaction execution protocol requires no determinism guarantees on the part of its operating system. Standard nondeterministic scheduling and unreliable or ill-ordered network message delivery do not detract from the deterministic relationship between input transaction sequences and externally observable behavior. There also are several subtle advantages to being able to run this kind of transaction processing system on off-the-shelf systems.

First, allowing any given deployment of the system to schedule threads in the best manner for its specific hardware setup can be extremely useful. To illustrate this, suppose we have a sequence of transactions $\{T_1, T_2, T_3, T_4\}$ where T_1, T_2, and T_3 do not conflict with one another at all (i.e. they can be executed as concurrently as we like), but where T_4 conflicts with all three of the previous transactions (i.e. we cannot begin executing it until they are all complete). Suppose also that the transaction processing application is replicated on two unlike machines, replica A and replica B (in the manner illustrated in Figure 1), where replica A will take extra time to execute T_2—e.g., due to a cache miss occurring only on one machine since the machines could have different cache sizes—while replica B will take extra time to execute T_1—e.g., because its ALU is slower than replica A's at certain operations. Figure 2 illustrates several possible execution schedules that might occur if each replica has two hardware contexts executing transactions. Schedule 1 represents a good schedule for replica A, but if it is chosen deterministically, then replica B will also choose schedule 1, with the same interleaving of transaction begin and end events. In this case, it is forced to start T_3 only after T_1 finishes, and it must stretch T_2 out to complete when T_3 does, resulting in poor CPU resource utilization, increased transaction latency, and—ultimately—reduced system throughput.

If deterministic thread scheduling were instead to yield schedule 2 at both replicas, replica A would up with a similarly suboptimal execution trace. A better scenario would be to execute the set of transactions with schedule 1 at replica A and schedule 2 at replica B—which is only possible if threads are free to yield CPU resources to one another based on nondeterministic factors such as the real wall-clock time that different machines actually spend executing each transaction.

In the case of systems that partition their data across multiple machines at each replica, even worse scenarios with more dramatic performance costs present themselves in the event of variations in intra-replica network latencies.[3]

Second, replication across several machines running a heterogeneous mix of platforms, improves overall availability, since different replicas tend to take different code paths through the underlying OS, and therefore rare bugs or vulnerabilities in an operating system or other software component of a given platform are less likely to trigger correlated failures. Portability to any platform—not only those which provide convenient determinism guarantees—is therefore especially valuable in transaction processing systems where high availability is usually considered critical.

Finally, the architecture we propose is also compatible with traditional (non-deterministic) transactional execution —the set of valid executions in our system is, after all, a direct subset of the set of executions allowed in standard transaction processing systems. Our system therefore incorporates the possibility to transition on-the-fly (and almost seamlessly) between deterministic mode and traditional execution mode, which can increase concurrency under some workload and execution conditions. Combining this property with an application execution environment supporting determinism guarantees—which are presumably equally easy to turn off dynamically—we also envision a selectively-deterministic platform where a user may control in real time which applications execute “normally” and which ones deterministically. Applications for such a platform would include development and testing, as well as deployment of systems where some components must execute deterministically (to ensure replayability or strongly consistent replication, for example) while less important components—or highly nondeterministic legacy code—forgo the modest overhead that comes with enforcing determinism.

[3]Interestingly, this effect is perfectly analogous to the phenomenon discussed in the footnote[1] on page 2, where on-the-fly transaction reordering proves profitable against disk latency-rooted variations in transaction duration. In this case, however, we only reorder the execution of *logically commutable* transactions (relaying commit messages back to the application in the original order, of course), so that the end system state is unaffected. Note that since threads share system data that they access at the start and end of each transaction (e.g. lock manager data structures and thread pool metadata), it would be extremely difficult for a scheduling library to automatically guess at this commutativity via memory ownership tracking or other static analysis techniques in the style of CoreDet and increase concurrency accordingly [2].

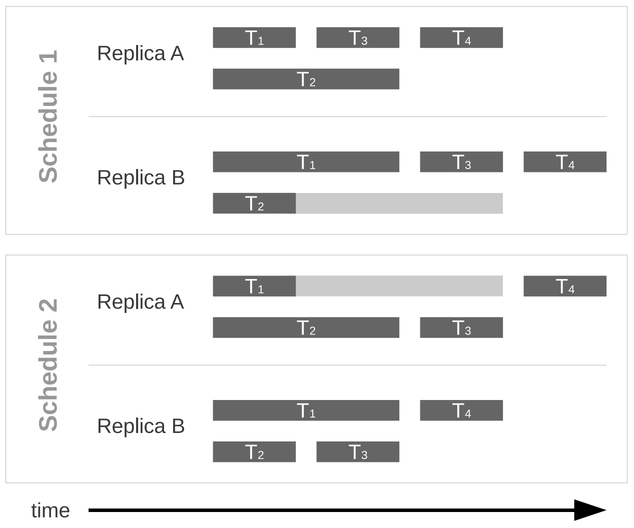

Figure 2: Two possible execution schedules for a pair of replicas of a transactional system. Dark grey denotes the shortest time in which a transaction could execute; light gray denotes artificial stretching out of transaction execution to maintain equivalent instruction interleaving with that of the opposite replica's schedule.

4. CONCLUSION

Implementing determinism in transactional systems need not simply leverage deterministic primitives of the underlying OS. Rather, concurrency control and commit protocols of the transactional system can be modified, thereby allowing the transactional system to run on legacy operating systems for improved performance and robustness to correlated failures. Such a deterministic database system nonetheless fits into a deterministic application stack that includes other tools and applications running on a deterministic OS.

Perhaps surprisingly, running a deterministic transaction processing system on top of a traditional OS, not only is the overhead of determinism minimal, but it can actually *improve* performance relative to traditional (completely nondeterministic) approaches due to the ability to omit contention-increasing distributed commit protocols. As the number of applications that could profit performance-wise by deterministic execution, we expect an increasing number of database systems to implement deterministic concurrency control in the future.

5. REFERENCES

[1] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. *Efficient system-enforced deterministic parallelism.* In OSDI, 2010.

[2] T. Bergan, O. Anderson, J. Devietti, L. Ceze, and D. Grossman. *CoreDet: A compiler and runtime system for deterministic multithreaded execution.* In ASPLOS, 2010.

[3] J. Gray, P. Helland, P. O'Neil, and D. Shasha. *The dangers of replication and a solution.* In Proc. of SIGMOD, pages 173-182, 1996.

[4] S. Harizopoulos, D. J. Abadi, S. R. Madden, and M. Stonebraker. *OLTP through the looking glass, and what we found there.* In SIGMOD, Vancouver, Canada, 2008.

[5] K. Krikellas, S. Elnikety, Z. Vagena, and O. Hodson. *Strongly consistent replication for a bargain.* In ICDE, pages 52-63, 2010.

[6] C. A. Polyzois and H. Garcia-Molina. *Evaluation of remote backup algorithms for transaction-processing systems.* ACM Trans. Database Syst., 19(3):423-449, 1994.

[7] A. Thomson and D. J. Abadi. *The case for determinism in database systems.* In VLDB, Singapore, 2010.

[8] Tom Bergan, Nick Hunt, L. Ceze, and S. Gribble. *Deterministic process groups in dOS.* In OSDI, 2010.