Jose M. Faleiro — Research Statement

The increasing democratization of server hardware with multi-core CPUs and large main-memories has been one of the dominant hardware trends of the last decade. "Bare metal" servers with tens of CPU cores and over 100 gigabytes of main memory have been available for several years now. Recently, this large scale hardware has also been available via the cloud; for instance, Amazon EC2 now provides instances with 64 physical CPU cores. Due to these hardware trends, database systems, with their roots in uniprocessors and paucity of main-memory, need to rethink their design.

My thesis research investigates the use of *deterministic* database systems to enable efficient and scalable transaction processing on modern hardware. Using deterministic execution as a hammer, my work examines well-established ideas in transaction processing on modern hardware, such as scheduling, concurrency control, and recoverability. This investigation yielded a new understanding of the underlying mechanisms used to implement the above ideas, such as the role of long-duration write locks, the role of timestamps in versioning schemes, the role of database-induced aborts, and so forth. Armed with this understanding, I devised new mechanisms that not only addressed limitations of existing database systems on modern hardware, but also shed new light on long-standing issues in both the research community and real-world systems, such as addressing the gap between strong and weak isolation, effectively exploiting versioning under serializability, and addressing the parallelism gap in log shipping-based replicated systems.

Research Philosophy

While designing solutions for specific problems, my approach is to consider the broadest implications of an algorithm or an implementation technique. My work on multi-version concurrency control, piece-wise visibility, and parallel replication are heavily influenced by this line of thinking. In each of these projects, I thought about the fundamental limitations of existing work rather than addressing specific overheads or bottlenecks. In the process I realized that, in addition to addressing symptoms, the resulting solutions could provide stronger theoretical (i.e., provable) guarantees over the state-of-the-art. I also believe in the importance of engaging with practitioners to ground the assumptions and motivation for research. My ongoing work on parallel replication is directly influenced by outreach efforts to practitioners from industry, and I am currently collaborating with engineers at Facebook to address the problem in MySQL, one of the most widely used storage systems at Facebook. This work is planned to be deployed in production at Facebook, and has the potential to directly impact other production users of MySQL. Finally, I strongly value collaboration; several of my research projects have been collaborations with researchers from other departments and communities. For instance, I have worked with systems researchers at Yale, database systems researchers at UC Berkeley, and most recently software engineers at Facebook.

The remainder of this document is structured as follows. First, I describe my thesis work on deterministic database systems on modern hardware. Second, I describe my work on investigating synchronization in modern main-memory database systems. Third, I describe work that I have done on large-scale distributed systems. Finally, I outline directions for future work.

Deterministic database systems on modern hardware

Lazy transaction evaluation [7]. Conventional database systems execute transactions eagerly; each transaction is typically executed in its entirety, including all reads and writes to the database, before returning a response to clients. However, if a subset of the transaction's writes are only observed by an external read at a later point in time, then the database could make better use of resources by executing those writes lazily. Lazy evaluation could reduce resource requirements during load spikes; a serious problem in real-world settings, often mitigated via error-prone measures such as application server-side batching of requests to amortize the cost of database interactions. In addition, lazy evaluation could also improve cache locality by batching the execution of transactions with overlapping read- and write-sets.

I implemented lazy evaluation by dividing each transaction into two pieces, an eager and a lazy piece. A transaction's eager piece executes immediately in order to return a commit decision to clients and process writes whose values are immediately required. Lazy piece execution is deferred, but must appear to the external world as if it was executed eagerly. This has two implications for the transaction execution. First, once the database promises to commit a transaction, its lazy piece cannot be aborted. Second, the serialization order of multiple transactions' eager and lazy pieces must be consistent. Unfortunately, these two properties are impossible to guarantee in non-deterministic database systems. A deterministic database system, however, totally orders transactions prior to their execution, and can use this total order to consistently order eager and lazy pieces. Furthermore, deterministic database systems only abort transactions if they are explicitly triggered due to logic. Once all logic which could lead to an abort successfully executes, the transaction is guaranteed to commit, even though the transaction as a whole is not executed in its entirety. This remaining portion of the transaction can be executed lazily.

Two aspects of this project directly influenced my later work. The first was an implementation technique; I used a dependency graph-based scheduling mechanism which abstractly captured conflicts between transactions, and could be constructed by relaxing a total order of transactions into a partial order based on transaction conflicts. I later realized that this design pattern could address limitations in multi-version concurrency control protocols [3]. The second was that it was possible to rely on the fact that deterministic systems never abort transactions for non-deterministic reasons to implement new recoverability mechanisms. In my work on piece-wise visibility [5], I made a connection between existing recoverability mechanisms and the fundamental limitations they impose on serializable protocols.

Multi-version concurrency control [3]. Modern main-memory database systems increasingly store data in a multi-versioned format, where each update creates a new version of a record, while prior values of records are preserved in old versions. Multi-versioning is attractive because, in principle, reads and writes to the same record can be decoupled; reads can be satisfied by old versions, while writes create new versions. However, I realized that serializable multi-version concurrency control (MVCC) protocols do not effectively exploit multi-versioning because it is easier to guarantee serializability by ensuring that reads are primarily directed to the most recently updated versions of records, and hence conflict with writes attempting to install a new update. If not, guaranteeing serializability requires high overhead checks at runtime because the space of permissible reads and writes to records is significantly higher than in a single-version system.

In my work on lazy transactions, I used dependency graphs to resolve transaction conflicts prior to their execution by relaxing a total order of transactions into a partial order based on whether transactions actually conflicted. In this dependency graph based representation of a schedule, the transaction responsible for producing a particular version is determined based on the order writing transaction occur in the graph. Consequently, a reading transaction can determine precisely which writing transaction is responsible for producing the version required to satisfied its read. Using this dependency graph based scheduling mechanism, I developed a new deterministic serializable MVCC protocol, BOHM, and showed that it could provide two concurrency guarantees that are impossible in conventional non-deterministic MVCC protocols. First, BOHM could ensure that reads never blocked writes by relying on the fact that conflicts involving reading and writing transactions could be resolved prior to their execution. Second, BOHM could ensure that writes do not conflict with each other by consistently ordering writes prior to their execution. Both these guarantees translated to real performance gains, BOHM was able to outperform state-of-the-art non-deterministic MVCC protocols by more than 5X on conflict heavy workloads.

Piece-wise visibility [5]. Database systems must ensure that the effects of an aborted transaction are never observed by committed transactions. This property, known as recoverability, is implemented by delaying the visibility of writes; a particular write can only be observed after its corresponding transaction finishes executing *all* its other writes. I noticed that delayed write visibility interacts with serializability's requirement that transactions generally observe the latest value of a record. Any delay in satisfying a read will delay the corresponding reading transaction and thus limit concurrency. Unfortunately, *every* widely used non-deterministic implementation of serializability employs delayed write visibility, and thus inherits its limitations.

Delayed write visibility is a fact of life in non-deterministic database systems because a transaction can be arbitrarily aborted any point during its execution. A deterministic database system, on the other hand, only aborts a transaction if on encountering errors in the transaction's logic, such as explicit abort statements or statements which may violate state invariants. Once all statements which could trigger an abort finish executing, a transaction is guaranteed to commit. Accordingly, I devised a new deterministic concurrency control protocol, piece-wise visibility (PWV), which exploits the above insight. PWV decomposes a transaction into a set of sub-transactions or *pieces*, such that each piece consists of one or more transaction statements. PWV eschews delayed write visibility, making a piece's writes visible as soon as its corresponding transaction is guaranteed to commit, even if one or more pieces from the same transaction have not yet executed. PWV could outperform conventional serializable protocols by over an order of magnitude, and perform competitively with (and often outperform) read committed, a widely used weak consistency level.

Parallel replication (ongoing). The final piece of my thesis research concerns replication lag in multi-core database systems. This project was motivated by conversations I had with practitioners at various large-scale companies, including Facebook, LinkedIn, and Uber, about their pains replicating multi-core database systems. In order to remain available in the presence of machine failures, applications replicate their data using primary-backup replication. In such a setting, a primary server executes requests, and streams a log of its changes to a set of backups. Unlike recovery mechanisms based on page-level parallelism, backups must also serve reads as of a complete consistent prefix of the log, which in turn limits the parallelism of log application on backups. This constrained parallelism can lead to unbounded replication lag, impeding backups' ability to serve fresh reads, and impacting availability by increasing failover times. Accordingly, I am working on a parallel replication mechanism that is guaranteed *never* to fall behind a primary, regardless of the level of isolation and type of concurrency control employed on the primary. This replication mechanism is a direct application of the techniques I developed in my prior work on lazy evaluation, MVCC, and piece-wise visibility. I am currently working on putting these ideas into production at Facebook in the MySQL open-source database system, where it has the potential to significantly simplify database maintenance and the broader ecosystem of systems built on top of MySQL.

Multi-core database systems

Explicit message-passing [9]. A database system's concurrency control related data structures are among the most contention prone because they manage conflicts between requests. This contention is inevitable because the database's entire state is accessible to any thread in the system. To address the limitations of such "shared-memory" architectures, research in the OS community has made the case for employing explicit message passing as an inter-thread communication mechanism. I wanted to take ideas from this research and apply it to addressing contention on concurrency control meta data in database systems. I proposed mediating access to the database system's concurrency control layer via a set of dedicated threads. These *concurrency control threads* were separate from *execution threads*, which were responsible for executing transactions' logic. The two classes of threads communicated with each other via explicit message passing. An implementation of these ideas showed that explicit message passing eliminated the cost of synchronization under conflict heavy workloads, resulting in an order of magnitude improvement in throughput.

Lock-free synchronization [4]. Several recent research papers on multi-core database systems strongly advocated for the use of lock-free synchronization over locks due to their perceived superior scalability. However, in my experience building multi-core deterministic systems, I did not observe any significant differences in scalability between locking and lock-free synchronization. To shed some light on this disparity, I investigated the differences between lock-free and locking synchronization via a set of controlled experiments. I found that claims of superior scalability of lock-free algorithms were misguided. My evaluation found that lock-free algorithms could trigger quadratic queueing delays on multi-core hardware due to retry loops. These quadratic queueing delays can cause scalability collapse under contention, similar to non-scalable locks such as ticket locks.

Distributed systems

Partially ordered shared logs [8]. Large-scale datacenter applications rely on control plane services such as filesystem namenodes, SDN controllers, coordination services, and schedulers. These control plane services must themselves be distributed for fault-tolerance and scalability. Past research showed that it was possible to build distributed control plane services over a totally ordered shared log abstraction. The shared log acts as the source of fault-tolerance and concurrency control, simplifying control plane logic. However, totally ordered shared logs rely on a centralized sequencer to totally order log insertions, which fundamentally limits its scalability, and severely affects availability and latency in the wide-area. Accordingly, with colleagues at

Yale, I explored the idea of partially ordered shared logs, which eschew centralized coordination but retain the fault-tolerance and concurrency control properties of totally ordered logs. We implemented the partially ordered shared log and were able to build several real applications over it, including a ZooKeeper clone and an HDFS compatible namenode. By leveraging partial ordering, our ZooKeeper clone could achieve more than 1.6 million creates/s, a 50X improvement in throughput with no change in ZooKeeper semantics.

Geodistributed actor programming models [1]. During an internship at Microsoft Research, I worked on geodistributing the Orleans actor programming framework, which at the time could only be deployed in a single geographic region. Orleans dynamically mapped actor-ids to their instantiations in a distributed hash table, and synchronously looked up the hash table before calling actor instances. While acceptable in a single datacenter or geographic region, high inter-region latency meant that synchronous lookups were a non-starter in geodistributed deployments. Accordingly, along with colleagues at Microsoft Research, I developed and prototyped an asynchronous geodistributed directory protocol for optimistic actor instantiation and conflict resolution. This asynchronous protocol became the basic building block of geodistributed Orleans, and we were able to demonstrate that our approach came with negligible overhead.

Other research. With colleagues at UC Berkeley, I worked on key-value store system that employed a message-passing and lattice-based state management architecture to scale across the cores of a single server and scale out across servers in a distributed system [10]. To provide developers with a useful rule of thumb while implementing distributed database systems, I proposed the Fairness-Isolation-Throughput (FIT) tradeoff [2], which stated that a distributed database could achieve at most two of the three FIT properties simultaneously. Prior to beginning my PhD studies at Yale, I worked on wait-free algorithms to serve consistent reads over eventually consistent systems [6].

Future Work

Massive scale program analysis. The majority of contemporary program analyses are designed to run sequentially on a single machine. These analyses are thus limited by a lack of parallelism and available memory on a single server. Practical tools therefore often tradeoff precision, and hence utility of analysis, for CPU and memory efficiency. As a consequence, the effective analysis of large codebases has proven extremely challenging. Unfortunately, with the predicted preponderance of physically actuating devices in the future, the need for high-assurance software is more urgent than ever. To address this dilemma, I am interested in infrastructure and algorithms for massively scalable program analysis. One concrete direction I am interested in is parallelizing dataflow analysis. Dataflow analysis is a widespread algorithm whose states belong to a lattice domain and evolve via monotonic transformations. Dataflow analysis could be parallelized using ideas from distributed systems, which show that monotonic transformations of state belonging to lattice domains can converge despite race conditions. Notably, resilience to race conditions has been one of the key enablers of large-scale distributed machine learning; I believe dataflow analysis can similarly be distributed at massive scale.

Bug finders for database applications. I am also interested in using program analysis to address difficulties in building database-backed applications. For instance, there is a real need to understand the implications of weak consistency on applications' high level constraints. Database systems abstractly allow applications to tradeoff consistency for performance via consistency levels. However, these consistency levels are specified at the granularity of permissible interleavings between low-level reads and writes to database objects, and it is consequently extremely challenging to determine whether it is permissible for an application to employ weak consistency without corrupting its state. An application's state is thus susceptible to silent corruption, recovery from which is a laborious, error-prone exercise. To address this problem, I believe we can exploit the fact that the operational semantics of weak consistency are well-defined in an implementation agnostic fashion. Using this well-defined operational semantics, interleavings between an application's transactions could be systematically explored via appropriately abstracted model checking. Such systematic exploration could determine precisely which of an application's constraints might be violated due to weak consistency.

Data management for service oriented applications. Modern large-scale applications are moving away from architectures consisting of monolithic application servers and databases. Driven by the economic incentive for quick iteration and technology advances in containerization, applications increasingly employ service oriented architectures consisting of multiple smaller applications or *services* interacting via explicit message-passing. Fragmenting application state across services means that no single system component has a global view of state, which in turn complicates guaranteeing well-understood properties in the monolithic context. For instance, to guarantee request atomicity, a monolithic application can rely on atomicity mechanisms implemented by a database system. Service oriented applications, however, cannot rely on such mechanisms because no single database system has access to the application's state in its entirety. This functionality gap is similar to technology trends around distributed key-value systems a decade ago, where systems gave up functionality such as multi-object transactions and consistent replication, only to reimplement them a few years later. To head off this scenario, I am interested in data management frameworks and abstractions for service oriented applications to provide guarantees such as request atomicity, consistent caching, and request memoization, guarantees that are difficult to make without a global view of state.

Log-structuring considered harmful. Driven by hardware trends of NVRAM and RDMA, there has been recent interest in significantly lowering the latency profiles of database and other storage systems. Recent research has focussed on exploiting the byte-addressability and low-latency random IO of this hardware, but still implements important system components, such as recovery subsystems, in a log-structured manner. Log-structuring can simplify the implementation of guarantees such as atomicity (by ensuring that a single durable write is sufficient to commit a transaction) and recoverability (by allowing uncommitted transactions to share fate under failures). Unfortunately, log-structuring can interact badly with large-scale multi-core hardware; the tail of the log becomes a single point of contention, and in turn necessitates some form of batching to amortize the cost of synchronization. When combined with the low latency of NVRAM and RDMA, this batching can shift latency bottleneck of these systems from storage to CPU; recent papers have shown that it takes on the order of nanoseconds or single digit microseconds to perform durable writes, but in a set of simple microbenchmarks, I found that it takes on the order of tens of microseconds of batching to avoid bottlenecking on the tail of the log on large-scale multi-core hardware. Thus, in order to effectively exploit NVRAM and RDMA on multi-core hardware, I believe that future research will have to fundamentally rethink the design of important database system components to avoid log-structuring.

References

[1] Philip A. Bernstein, Sebastian Burckhardt, Sergey Bykov, Natacha Crooks, Jose M. Faleiro, Gabriel Kliot, Alok Kumbhare, Muntasir Raihan Rahman, Vivek Shah, Adriana Szekeres, and Jorgen Thelin. Geo-distribution of actor-based services. In *OOPSLA*, 2017.

[2] Jose M. Faleiro and Daniel J. Abadi. Fit: A distributed database performance tradeoff. *IEEE Data Engineering Bulletin*, 38(1), 2015.

[3] Jose M. Faleiro and Daniel J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11), 2015.

[4] Jose M. Faleiro and Daniel J. Abadi. Latch-free synchronization in database systems: Silver bullet or fool's gold? In *CIDR*, 2017.

[5] Jose M. Faleiro, Daniel J. Abadi, and Joseph M. Hellerstein. High performance transactions via early write visibility. *PVLDB*, 10(5), 2017.

[6] Jose M. Faleiro, Sriram Rajamani, Kaushik Rajan, G. Ramalingam, and Kapil Vaswani. Generalized lattice agreement. In *PODC*, 2012.

[7] Jose M. Faleiro, Alexander Thomson, and Daniel J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD*, 2014.

[8] Joshua Lockerman, Jose M. Faleiro, Juno Kim, Soham Sankaran, Daniel J Abadi, James Aspnes, and Mahesh Balakrishnan. The flexlog: A partially ordered shared log. *Under submission. Draft: https://tinyurl.com/flexlog-pdf.*

[9] Kun Ren, Jose M. Faleiro, and Daniel J. Abadi. Design principles for scaling multi-core oltp under high contention. In *SIGMOD*, 2016.

[10] Chenggang Wu, Jose M. Faleiro, Yihan Lin, and Joseph M. Hellerstein. Indy: A kvs for any scale. *Under submission. Draft: https://tinyurl.com/indy-kvs-pdf.*