Research statement

Tej Chajed

December 2021

Systems software is a critical fabric for modern technology. Bugs in systems software can have serious consequences, ranging from security vulnerabilities, to data loss, to interruption of services that millions of users rely on. Practitioners recognize the importance of this core software and put enormous effort into testing and fuzzing to improve reliability — as one famous example, sqlite3 has 600 times as much test code as library code — yet despite this effort, bugs are still found in infrastructure like file systems, key-value stores, databases, and browsers.

Formal verification is a promising and systematic approach to write correct systems, which has recently been successfully applied to many systems. A verified system is developed by writing an implementation along with a **proof that the code always meets a specification of desired behavior**. The proof is itself written in a computer, checked mechanically in a process similar to compilation, and developed with a process analogous to software development.

A central challenge in systems verification is going from theoretical foundations, which demonstrate how to reason about small programs, to approaches that can handle real implementations. My research covers a wide range of the software verification stack to bridge the gap, including defining specifications, developing new verification techniques, and proposing new verification-friendly systems designs. I'm interested in designing and verifying practical systems, including ones that have good performance and handle crashes.

1 Verifying a concurrent, crash-safe file system with sequential reasoning

My PhD work has focused on the domain of verifying file systems. Since the file system stores all the persistent data, bugs can cause permanent data loss and correctness is especially important. What makes it hard to implement a file system correctly is that a file system should tolerate *crashes* (where the system reboots unexpectedly) and *concurrency* between multiple applications and the disk, all while aiming for good performance. Verifying a file system is difficult because in the pursuit of performance file-system implementations simultaneously carry out in-memory and I/O processing for multiple operations, and the proof must cover all the interleaving executions of the code.

My main line of research developed and verified **DaisyNFS**, an implementation of the Network File System (NFS) protocol that has a *concurrent implementation*, guarantees operations are *atomic and durable* even if the system crashes, and gets *performance comparable to that of Linux*. Most file systems contend with concurrency throughout the entire implementation. DaisyNFS instead uses a transaction system that isolates the crash safety and concurrency reasoning so that the bulk of the code is verified using easily automated sequential reasoning. The tools and techniques for reasoning about crashes and concurrency in a high-performance system didn't exist in 2015 when I started working on this problem; the transaction system that enables DaisyNFS to get good performance is built on new verification infrastructure that made it possible to reason about a concurrent storage system.

A core challenge in verifying a file system is proving that the system correctly recovers following a crash and reboot at any time. **Crash Hoare Logic (CHL)** [2] is an extension of Hoare logic that introduces the idea of a *crash condition*, an invariant that a particular function maintains at all intermediate points. Crash conditions strengthen the specifications of Hoare logic and enable reasoning about crashes, recovery, and even repeated crashes during recovery. CHL is the basis for **FSCQ, the first verified file system**; I joined Haogang Chen in this effort and contributed to the core theory. The ideas of Crash Hoare Logic were important in most of my subsequent research even as we improved performance and introduced concurrency.

While Crash Hoare Logic can reason about a single recovery procedure, a realistic storage system would have recovery at multiple abstraction levels — for example an application might need to restore its state after the file system recovers. I led the development of Argosy, **the first verification framework with reasoning about nested recovery procedures** [6]. Crashes during recovery are tricky when they interrupt application recovery because the system has to start over with file-system recovery, but Argosy proves that if each recovery procedure is independently verified the two together are also correct.

Realistic storage systems have concurrent implementations, which poses a particular challenge for verification because the proof must reason about thread interleavings combined with a crash at any time. My co-authors and I developed **Perennial, the first framework for mechanized proofs for concurrent programs that supports reasoning about crashes** [7, 9]. Perennial draws from Crash Hoare Logic's basic idea of a crash obligation and integrates this idea into Iris [4], a framework for reasoning about concurrency using separation logic.

Perennial consists of a program logic, which organizes a system's proof into specifications for individual functions,

and a set of techniques for formalizing the behavior of concurrent storage systems and their internal layers. The most basic idea is to re-use *invariants* from Iris. An invariant is a property which, once established, the proof system guarantees is maintained by all threads. Perennial turns invariants into a crash reasoning principle, exploiting the fact that if the system crashes the recovery thread can rely on any invariants to hold. Perennial develops several fundamental ideas to integrate crash safety into concurrency reasoning: *local crash obligations* as in Crash Hoare Logic, *logically-atomic crash specifications* that made it possible to verify a storage system in several layers, and a new separation logic primitive called a *crash borrow* for reasoning about ownership of durable state.

Systems verification needs to connect proofs to some implementation, and we want to verify code written in a systems programming language. Writing code at the right level of abstraction is especially important for Perennial's concurrency reasoning to be useful, since the performance benefits of concurrency would be moot if the single-threaded performance of the system were too poor. We developed **Goose, a tool for reasoning about Go code using Perennial** [8]. Goose models a subset of Go suitable for building verified systems, and gives specifications for reasoning about Go primitives like structs, locks, slices, and maps. This approach was essential to getting good performance: Go is a systems programming language with good support for concurrency and a productive basis to build verified software.

As the foundation for DaisyNFS, we implemented and verified a journaling system called GoJournal [9]. Many file systems use journaling, which allows the rest of the file system to safely update multiple pieces of metadata atomically. However, verified journaling poses a *specification challenge* since while GoJournal makes writes atomic, it requires the caller to use appropriate concurrency control to guarantee operations do not conflict. **GoJournal introduces a separation-logic specification for a journaling system.** Perennial's crash specifications and concurrent separation logic allow the specification to capture both the concurrency requirements and crash-safety guarantees of the journal.

A proof based on Perennial ties the GoJournal specification to an efficient implementation that uses multiple cores, merges concurrent operations, and issues all disk I/O without holding any locks for CPU-disk concurrency. Perennial's features were essential not only to specify GoJournal as a whole but also to divide the proof into several layers, each verified against the previous. Modularity was essential since GoJournal is designed for performance and concurrency over verification convenience.

A verified file system requires a good deal of code on top of the journal to implement its data structures and the logic for each operation, all of which requires proof. While GoJournal simplifies the crash safety aspect of verifying this code, concurrency remains a challenge. DaisyNFS thus develops a design where each operation runs within a *transaction*: this results in sequential reasoning for the file-system logic, and the transactional API is efficiently implemented in GoTxn, which layers two-phase locking on top of GoJournal. The DaisyNFS design is itself unusual for the reliance on transactions. In GoTxn we formalized a *program refinement* specification that **formally captures the idea that arbitrary transactions appear to execute atomically both to concurrent transactions and after a crash and reboot**. The proof that GoTxn meets this specification uses the flexibility of Perennial's program logic to implement a logical relations proof strategy adapted for crash and recovery reasoning.

Because GoTxn enables sequential reasoning for the body of each transaction, we were able to write and verify that logic using Dafny, a sequential verification-oriented programming language. As one measure of the reduced proof burden from using Dafny's sequential reasoning, DaisyNFS's **Dafny proof has just a 2:1 proof-to-code ratio** compared with the GoTxn verification which is closer to 20:1. Dafny helped us implement and verify the DaisyNFS file system with a wide range of features and good performance. Across a range of benchmarks running on a fast NVMe disk, **DaisyNFS gets comparable performance to Linux NFS exporting an ext4 file system**.

Overall, my research includes the following intellectual contributions:

- CHL introduces the idea of crash conditions as a way of structuring specifications about crash behavior.
- Argosy introduces recovery refinement to specify an abstraction with its crash and recovery behavior. The main theorem is that recovery refinements compose as desired.
- Perennial introduces crash weakest preconditions (a concurrent extension of crash conditions), a principle for repurposing invariants for crash reasoning, principles for ownership of durable state that transfers to recovery, and a specification style for concurrent storage abstractions.
- Goose takes a useful subset of Go for concurrent systems and presents the user with specifications and theorems for reasoning about code written in that subset using Perennial.
- GoJournal's specification captures the mix of crash atomicity guarantees and concurrency requirements in a high-performance journaling system.
- GoTxn's specification formalizes the intuition that a transaction system makes code appear sequential.
- DaisyNFS includes a verification-friendly design that uses GoTxn's transactional API to make most file-system reasoning sequential.

Impact. Perennial has seen some direct adoption from groups at Aarhus University and MPI-SWS, who are using it as the basis for program logics for non-volatile RAM. This research requires extending Perennial to handle reasoning

about weak memory behaviors but leverages all of its mechanisms for reasoning about the persistent memory across crashes and reboots.

Perennial and Goose are also the basis for a new research project at MIT on reasoning about distributed systems. Goose is used to tie this reasoning to efficient implementations, while Perennial is being extended to reason about concurrency between both machines and threads.

FSCQ [2] was the first verified file system; since then, several other groups have created verified file systems that go beyond FSCQ, including Yggdrasil, AtomFS, and VeriBetrFS. The first version of Perennial [7] describes how to encode *refinement* reasoning into separation logic, including with crash reasoning. Researchers at VMWare Research and CMU are working on incorporating these reasoning principles into Dafny. Perennial has also influenced practitioners at Amazon, who are interested in formal reasoning in the development of the next generation of ShardStore, a key-value storage system in Amazon S3.

2 Combining systems design and verification

My work involves both systems and verification thinking, each of which informs the other. When going from systems to verification, this means taking existing designs and formalizing why they work in the proof. Bringing a verification perspective to systems entails coming up with new designs that are more obviously correct, using verification to sharpen the layers and organization of the system.

Formalizing systems intuition. *Recovery helping* is a reasoning principle in Perennial [7] where an operation logically appears to take place during recovery. The technique was partly inspired by concurrency helping where one thread completes an operation on behalf of another, typically on a data structure. Recovery helping arose out of trying to give a precise correctness argument to a storage system where the linearization point ended up being in the recovery code.

GoJournal and GoTxn intuitively make a sequence of reads and writes atomic; atomicity is the right intuition but verifying these systems requires a precise contract under which operations are atomic. In GoJournal we formalized this contract using separation logic to capture the concurrency control expected from the caller, and then in GoTxn we formalized the transaction system's stronger guarantee that transactions are atomic.

Verification-friendly design. In DaisyNFS we developed the system design based on thinking about the verification story: using transactions greatly simplifies the proof since the code within a transaction appears to execute sequentially. While file systems do use atomicity from journaling, DaisyNFS is a novel file-system design where every operation fits into a transaction — using transactions more aggressively is what enabled fully sequential reasoning for the rest of the file-system code. The insight for this system-design problem came from thinking about verification.

In GoJournal developing the proof helped us understand the code and invariants for the write-ahead log (an internal component of the journal). Rather than thinking about avoiding bugs in concrete executions, verification forced a principled design based on defining the abstract state of the write-ahead log and how it is affected by a crash.

3 Other verified systems

Verification is useful for systems beyond storage, and for properties other than crash safety. For example, one domain where verification is particularly valuable is security because the assurances of a proof cover all inputs, and attackers are motivated to find even obscure inputs that trigger vulnerabilities.

Efficient parsing. During an internship at Microsoft Research working with the Project Everest team, I worked on parsing for a verified implementation of TLS 1.3. Parsing in TLS is challenging because the standard involves over 200 serialized types, all of which must be manipulated efficiently for the overall implementation to be fast. I developed a specification and verification approach for writing efficient imperative parsers with the existing functional parsers as specifications, including a notion of a *validator* that allows using serialized data in-place without copying it. These core definitions went into a verified parsing framework called EverParse [5].

Deferred durability. FSCQ has a simple specification that promises each file-system operation such as `create` and `rename` is persisted to disk immediately and atomically; that is, if the system crashes in the middle of an operation it appears to have occurred either completely or not at all. For better performance, real file systems implement optimizations that relax the persistence and atomicity guarantees, allowing operations to be buffered in memory and for data writes to be reordered with metadata operations. In followup work we developed DFSCQ [1], which specifies and verifies a particularly tricky feature from ext4 of log-bypass writes, where file data is written directly rather than going through a write-ahead log. A key contribution of DFSCQ is a **metadata-prefix specification that precisely states what data must be on disk after a crash**, with a proof that the implementation meets this specification even in the presence of buffered operations and log-bypass writes.

Confidentiality. File systems serve an important function of keeping users' data confidential. Specifying exactly what confidentiality means is challenging, since typically file systems protect file data but not all metadata between users. SFSCQ is a verified file system with a specification that formalizes this security property with *data noninterference* [3]. SFSCQ's proof uses a technique called *sealed blocks* to layer

the confidentiality proof on top of DFSCQ's existing functional correctness proof. I am generally interested in **incremental and modular verification techniques as a way to make verification more practical**, in this case by separating functional correctness from confidentiality.

4 Future directions

My research agenda is to identify a new domain where verification is valuable, then develop the specifications, verification techniques, and system designs that make verification possible in that domain. So far I have executed this agenda for concurrent and crash-safe storage systems, culminating in a verified file system. In my future work I'm excited to expand the reach of verification to other domains and build new systems with formal verification.

Security reasoning. Proving that systems preserve confidentiality is a difficult aspect of security reasoning. Testing in this domain is especially difficult, since it is hard to determine whether information has leaked from merely running the system. Formal verification has room in this area to define new specifications, since systems are often interested in protecting some data but not all (for example, file systems typically make total free space public), and also to scale verification up to larger systems. As in SFSCQ [3] I am interested in reducing the proof burden by factoring out reasoning about confidentiality from functional correctness.

Partial failures. Distributed systems face concurrency and crashes, like storage systems, but include an additional challenge in the form of partial failures where some but not all nodes reboot. In this domain I'm especially interested in advancing research that covers implementations and encompasses a whole system, both its clients and servers. I'd also like to extend the recovery reasoning in Perennial to recovery in a distributed system, where nodes need to re-join by talking to existing servers.

Mixing verification styles. I'm interested in exploring verification designs that combine general, interactive proofs of core infrastructure with simpler, automated proofs for clients. GoTxn and DaisyNFS are one example, where the infrastructure is a transaction system that makes the caller's code atomic. I have experience with both automated and interactive verification, and such systems can combine the best of both styles of verification. Future work in this area could include other transaction systems, or go beyond storage to distributed infrastructure like map-reduce or the AWS Lambda interface.

Safe software upgrades. Systems are not static artifacts but instead evolve over time. This raises the challenge of reasoning about software upgrades, such as when new software accesses old data, or in a distributed system when nodes running different versions communicate with each other. Upgrades are a good fit for verification because like storage, bugs can lead to permanent data corruption, and thoroughly testing the interactions between two pieces of software is extremely difficult. The core concept of a crash invariant might be directly applicable here, albeit one that is shared between the old and new versions of the code; I am interested in exploring the details and connecting the theory to practical code.

References

[1] Haogang Chen, **Tej Chajed**, Alex Konradi, Stephanie Wang, Atalay Ileri, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying a high-performance crash-safe file system using a tree specification. In *SOSP*, Shanghai, China, October 2017.

[2] Haogang Chen, Daniel Ziegler, **Tej Chajed**, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare Logic for certifying the FSCQ file system. In *SOSP*, Monterey, CA, October 2015.

[3] Atalay Ileri, **Tej Chajed**, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Proving confidentiality in a file system using DiskSec. In *OSDI*, Carlsbad, CA, October 2018.

[4] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: a modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming*, 28:e20, 2018.

[5] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, **Tej Chajed**, Nadim Kobeissi, and Jonathan Protzenko. EverParse: Verified secure zero-copy parsers for authenticated message formats. In *USENIX Security*, Santa Clara, CA, 2019.

[6] **Tej Chajed**, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Argosy: Verifying layered storage systems with recovery refinement. In *PLDI*, Phoenix, AZ, June 2019.

[7] **Tej Chajed**, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *SOSP*, Hunstville, ON, Canada, October 2019.

[8] **Tej Chajed**, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent Go code in Coq with Goose. In *CoqPL*, New Orleans, LA, January 2020.

[9] **Tej Chajed**, Joseph Tassarotti, Mark Theng, Ralf Jung, M. Frans Kaashoek, and Nickolai Zeldovich. GoJournal: a verified, concurrent, crash-safe journaling system. In *OSDI*, July 2021.