Storage Systems are Distributed Systems (So Verify Them That Way!)

Travis Hance, Carnegie Mellon University
Andrea Lattuada, ETH Zurich
Chris Hawblitzel, Microsoft Research
Jon Howell, VMware Research
Rob Johnson, VMware Research
Bryan Parno, Carnegie Mellon University

Abstract

To verify distributed systems, prior work introduced a methodology for verifying both the code running on individual machines and the correctness of the overall system when those machines interact via an asynchronous distributed environment. The methodology requires neither domain-specific logic nor tooling. However, distributed systems are only one instance of the more general phenomenon of systems code that interacts with an asynchronous environment. We argue that the software of a storage system can (and should!) be viewed similarly. We evaluate this approach in VeriBetrKV, a key-value store based on a state-of-the-art B$^\varepsilon$tree.

In building VeriBetrKV, we introduce new techniques to scale automated verification to larger code bases, still without introducing domain-specific logic or tooling. In particular, we show a discipline that keeps the automated verification development cycle responsive. We also combine linear types with dynamic frames to relieve the programmer from most heap-reasoning obligations while enabling them to break out of the linear type system when needed. VeriBetrKV exhibits similar query performance to unverified databases. Its insertion performance is 24× faster than unverified BerkeleyDB and 8× slower than RocksDB.

1 Introduction

Software verification promises a fundamentally better way of constructing critical systems: Instead of relying on the spotty coverage provided by run-time testing, verification can mathematically guarantee the functional correctness and even the reliability of software at compile time.

In the context of distributed systems, prior work on IronFleet [29] shows how to combine Hoare logic [23, 31], to reason modularly about the behavior of a single program, with TLA-based [40] state-machine reasoning to show that a collection of nodes running that program behaves according to a high-level functional spec when executing in a failure-prone, asynchronous distributed environment. Both techniques employ general-purpose logic, unlike other work in the area that relies on custom languages or logic [17, 18, 58]. The approach is compatible with a reasonable level of automation, modest-scale implementations (2-3K LoC), and performance within a factor of 2 of unverified code.

We observe that the abstraction of a system as a program interacting with a failure-prone, asynchronous environment applies beyond the domain of distributed systems. Indeed, we argue that a very different domain – storage systems – fits quite naturally into this same abstraction, capturing the asynchrony and nondeterminism of crash safety in such systems. Hence we generalize the IronFleet methodology to this new domain, without introducing a domain-specific logic [12], creating a custom language [2], or changing our system's API and implementation to accomodate verification [54].

We evaluate the success of this generalization by constructing VeriBetrKV, a key-value store based on a B$^\varepsilon$tree [7], a complex but asymptotically-compelling write-optimized data structure. Modern persistent key-value stores use write-optimized data structures, such as LSM-trees [19, 20, 26, 37, 43, 53] and B$^\varepsilon$trees [50], in order to efficiently handle random-insertion workloads, which are common in many key-value-store applications. Write-optimized data structures outperform older key-value data structures, such as B-trees, by orders of magnitude. However, these performance gains come at the cost of a significant increase in code complexity.

TLA-based reasoning lets us prove VeriBetrKV's correct behavior under process crashes and under disk sector corruptions, while Hoare logic allows us to reason independently about the implementation-level optimizations needed for performance.

The resulting system is significantly more complex than the closest prior work, a verified in-memory key-value store [29]. The implementation is over 3× larger, and proving it functionally correct and crash safe requires a multi-level (vs. single-level) refinement proof.

Hence, an important contribution of this work are the techniques we developed to apply automated verification at this new scale. To make verification practical for large-scale software development, we need to balance between exploiting automation and controlling it. Ideally, we want *decisive* automation; i.e., automation that quickly tells us whether our code and proofs are correct or incorrect. Decisive automation keeps developers engaged and efficient. Unfortunately, since general-purpose verification is undecidable, most automation tools can also "time out". This pessimal outcome takes longer (by definition) and provides less direction to the developer, harming morale and productivity [21, §9.1][29, §6.3.2].

To escape the plague of time outs, we present a concrete development discipline that rapidly squashes time-out prone code. This ensures developers spend the majority of their time in a tight verification development cycle. For example, 98.3% of the definitions in VeriBetrKV verify in less than 10s, enabling development of larger code bases with less effort.

Many verification frameworks reason only about code operating on immutable data structures [11, 49], leaving optimized code generation to a compiler such as Haskell's. Most real systems code relies on explicit in-place updates for good performance to avoid data copies. Some verification frameworks enable reasoning about heap-manipulating code using various methodologies from the PL literature, from separation logic [12, 52] to dynamic frames [29, 30, 32]. These techniques rely on both programmer annotations and relatively heavy-weight automation (e.g., SMT solvers [16]) to determine whether a modification to one portion of the heap may have affected other objects on the heap. Despite our time-out-prevention discipline, we encountered challenges with such automation: while it works well in small instances, as the system grows more complex, the automation slows significantly, reducing developer productivity (in line with prior reports [29, §6.2]).

Drawing on ideas from commercial (Rust [33]) and research [2, 51, 59] languages, we integrate a lightweight linear type system that gives dramatic automation improvements with dynamic frames when additional flexibility is needed. We rewrote some core components of VeriBetrKV from dynamic frames into linearity, reducing our proof burden by 31–37% without impacting performance. We also leverage linear types to emit optimized C++ code.

Ultimately, our evaluation (§7) shows that VeriBetrKV is able to deliver much of the performance gains promised by sophisticated key-value store data structures. On insertions using a hard disk, it outperforms BerkeleyDB by 24×. However, there is still work to be done. Insertions in VeriBetrKV are about 8× slower than RocksDB, a highly-tuned commercial key-value store.

As with any research prototype, VeriBetrKV comes with limitations. One limitation of VeriBetrKV is that it is presently single-threaded; it can exploit I/O pipelining but not CPU concurrency. Second, the guarantees of verification are limited by the Trusted Computing Base (TCB): the top-level spec of a crash-robust key-value store, the spec of VeriBetrKV's interface to the OS and runtime, and the verifier and compiler (Dafny [41] and Z3 [16]). Finally, we focus on safety and functional correctness: we guarantee that the system does not return incorrect results, but liveness (i.e., the guarantee that an operation will complete in finite time) is out of scope.

In summary, this paper makes the following contributions:

1. A method of specifying crash safety in a clean and extensible way that generalizes a verification methodology for distributed systems. As a demonstration of its extensibility, we enhance the specification to include robustness to disk corruption.
2. A discipline for managing automation that supports scalable system development.
3. The integration of a lightweight linear type system within a general-purpose verification language, and a large-scale concrete case study quantifying the impact of the type system as compared to previous approaches.
4. A prototype key-value store demonstrating that our verification methodolgy can scale to handle the complexities of modern key-value-store data structures.

2 Assumptions and Background

We summarize the assumptions underpinning our verification results (§2.1), the verification techniques we employ (§2.2-2.3), and the prior work from which we take inspiration (§2.4).

2.1 Assumptions

Every verified system makes a guarantee that is predicated on a set of assumptions which can be divided into three categories. First, the user must trust the top-level application-facing specification of the contract provided by the system. We provide a succinct (283 lines) specification for VeriBetrKV in terms of a dictionary that, when it crashes, reverts to its state at the most recent client `sync` (§3.1.2).

Second, the system must specify an interface to the environment (i.e., the rest of the world) the codifies assumptions about how that environment behaves. VeriBetrKV's interface is an asynchronous I/O bus that reorders but does not duplicate or, except during a crash, drop messages. VeriBetrKV's environment models a block-level disk and the possibility of arbitrary crashes and torn writes (§3.1)

Finally, we assume the correctness of our verification tools, including the build system that runs our verifier, Dafny [41], on each file. We modify Dafny to emit C++ code (§5.2), so we also rely on the correctness of the C++ toolchain used to produce an executable. These trusted tools are comparable to those in other systems verification efforts; e.g., systems verified in Coq [15] trust Coq, Coq's extraction to, say, OCaml, and the OCaml compiler and runtime. Prior research indicates that each element in such a toolchain can itself be verified [6, 35, 42, 44, 56].

Despite all of these assumptions, several studies indicate that verification is a qualitatively better way of developing software [22, 24, 63], compared with current state-of-the-art code development techniques. These studies found numerous defects in traditional software, but none in verified software, only in the (unverified) trusted components.

2.2 TLA+-Style State-Machine Refinement

State machine refinement is an important tool in verifying asynchronous systems [1, 25, 38]. A high-level, abstract state machine models the desired behavior of a system (say, a key-value dictionary) capturing essential features (inserts update the dictionary; queries probe it) and abstracting away irrelevant details (e.g., efficient indexing and data representation). A concrete state machine adds more details, showing one way to instantiate the abstract machine (e.g., using a hash table to implement the dictionary). A safety proof then shows that the concrete state machine *refines* the abstract state machine, meaning that every execution of the concrete state machine can be mapped to a possible execution of the abstract one. Hence, reviewing the abstract state machine tells the consumer what to expect from the concrete state machine's behavior.

A strength of this approach is that the high-level machine can abstract over asynchrony. If the concrete machine has many concurrently-moving parts, but one can show a refinement (because most transitions in the concrete model do not change its abstract interpretation), then we know the concurrency is irrelevant to the abstract behavior.

2.3 Floyd-Hoare Verification

Floyd-Hoare reasoning [23, 31] is a popular technique for reasoning about the correctness of single-threaded imperative programs. The developer annotates the program's functions with pre-/post-conditions about the program's state when entering/leaving the function, and a verifier checks that these claims hold for all possible inputs and outputs.

Verification tools based on Floyd-Hoare reasoning typically do not consider asynchronous interactions with an environment, which makes it difficult to reason directly about, e.g., program crashes that might occur at arbitrary points during execution. Hence, prior work in storage-system verification introduced a novel version of Floyd-Hoare reasoning, Crash-Hoare logic, to cope with this particular flavor of asynchrony [12].

2.4 Verifying Distributed Systems

In their IronFleet work [29], Hawblitzel et al. show how to compose Hoare logic and TLA+-style state-machine refinement to reason about the safety and liveness of distributed systems. They use Hoare logic to reason locally about a single program's behavior, showing that it conforms to an abstract state machine. They then model the distributed system as another state machine whose state consists of N replicas of the program state machine, along with a network represented as a set of in-flight messages. The system transitions by nondeterministically choosing to allow either one of the N program state machines to advance a step, or the network's state to advance (e.g., by delivering a message). The model captures nodes that can fail-stop and a network that can duplicate, drop, and reorder messages. The top-level verification theorem proves that if the program (e.g., a sharded key-value store) runs in a distributed system as modeled, then its behavior matches a concise app spec – a centralized map.

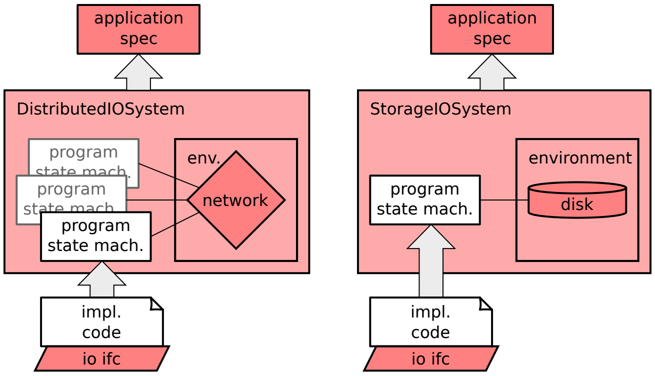


Figure 1: System and environment state machines express how a program interacts with the world to implement an application in IronFleet (left) and VeriBetrKV (right). Shaded shapes are trusted; unshaded shapes are untrusted code and proof.

3 Verifying Storage Systems

To verify storage systems, we observe that embedded within prior work on verifying distributed systems (§2.4) is a general-purpose framework for verifying code that interacts with an asynchronous environment, such as a storage system. This is exciting because it suggests we can use a common methodology to solve a broader class of systems verification problems, and it obviates the need to develop a specialized logic or proof framework for each environment. In this paper, we showcase an instantiation of this general methodology for an asynchronous environment with a single disk and a single processing node. However, we conjecture that the approach generalizes to a variety of systems including multi-node, multi-disk storage systems; indeed, our predecessor, IronFleet [29], has already shown how to model multiple nodes connected by an asyncronous network. Further systems applications might include heterogeneous hardware or device drivers.

In this general framework (Figure 1), the developer uses Hoare logic to prove that their optimized imperative program code, when run synchronously and without crashing, complies with an abstract *program state machine* (e.g., in prior work, this was the code running on each node in the distributed system). The program code uses a trusted API (e.g., for the network) to interact with the outside world, and an important aspect of the refinement proof is showing that the code's interactions with this API match those specified by the program state machine. A second state machine, the *environment*, encodes assumptions about the rest of the world – the parts the program does not control (e.g., the network). A final state machine, the *IOSystem*, composes these two state machines (the program and the environment) and dictates how they interact (e.g., prior work composes N programs running asynchronously, communicating only via the environment's network). The developers prove a top-level theorem showing that the IOSystem state machine refines a simple, high-level application spec (e.g., the distributed key-value store behaves like a centralized map). This theorem guarantees correct execution in practice if the environment behaves as modeled, if the hosts run an executable matching the program, *and* if the combined elements interact as assumed by the system.

In this work, we instantiate this general framework to reason about storage systems by modeling an environment that contains a disk and in which crashes may occur.

3.1 An Environment Model with Crashes

Our model of a storage system is a special case of the IOSystem as described above. It contains two parts: **(i)** the program, which models the executable VeriBetrKV, and **(ii)** the environment: a model of the disk with an I/O bus. Together, these two components form a *StorageIOSystem* (Figure 1).

The state machine representing this StorageIOSystem can transition in three ways. First, the program state machine can transition forward a step, possibly interacting with the I/O bus. Second, the environment can transition, e.g., by having the disk process a read- or write-command from the I/O bus.

Third and finally (and unlike IronFleet DistributedIOSystems), the StorageIOSystem as a whole can perform a *crash*, which models, e.g., a power failure or a kernel panic. Crashing results in a disk state that remembers every write it has acknowledged, but the data at the address of any unacknowledged write may reflect the old value, the newly written value, or a corrupt value (§3.1.1). For the program, a crash simply resets its state machine to its initial (boot-up) condition.

Our top-level proof shows that the StorageIOSystem is a refinement of the application spec (§3.1.2), demonstrating that VeriBetrKV's top-level guarantees are maintained despite an arbitrary number of crashes occurring at nondeterministic times (outside the program's control).

3.1.1 Corruption

Our disk model allows the disk to corrupt its data at any time (i.e., not just during a system crash). There is only one constraint: corruption cannot produce a block with a valid checksum.[1] When VeriBetrKV detects an invalid checksum, it aborts the current query. This ensures correctness, since VeriBetrKV will not return an incorrect query result. This assumption about the disk's behavior is what checksummed storage systems already (informally) assume. We formalize this assumption and make use of it in our correctness proof.

Using checksums means that VeriBetrKV protects against “torn writes”, where a block of the disk is changed to have neither its old value nor the value written, as well as random media corruptions. Of course, some disks do violate our checksum assumption. An adversarially-controlled disk could easily return corrupted data yet with valid checksums. Likewise, a disk which returns stale blocks would also not satisfy our checksum assumption. In either case, it would be impossible for *any* implementation to meet our application spec as written.

However, our methodology provides flexibility in specifying the precise assumptions made about the disk. In principle, an engineer could provide a weaker disk model, and in addition, either provide a cleverer implementation or a weaker application spec to match.

3.1.2 Application Specification

To achieve good performance, a practical storage system cannot afford synchronous writes. Instead, the application calls `sync` when it requires durability; data not synced is permitted to be lost during a crash. The nondeterministic relationship between nondeterministic `crash` and the application `sync` API must appear in the theorem; each is a transition in the application spec state machine.

Intuitively, the application spec of VeriBetrKV says that in the absence of crashes or block corruption, VeriBetrKV acts exactly like a dictionary, always returning the most-recently-written value. In the presence of a crash, VeriBetrKV is allowed to forget data, but no farther back than the most recent sync. Furthermore, “forgetting” is equivalent to the entire dictionary reverting to a consistent, previous snapshot; future crash-free operations proceed forward from this snapshot. Contrast this promise with a filesystem with crash-corrupted metadata: the data may appear complete and valid, but future operations may result in behaviors that violate the filesystem's contract.

Our storage system specification is easy to state, clean, and easy to utilize from a client application. In contrast, contemporary file systems, for performance reasons, decouple `sync` operations on metadata from `sync` on file data. Such guarantees are very difficult to utilize from a client, and in fact difficult to even state precisely; much of DFSCQ is dedicated to stating such a guarantee precisely [11].

3.2 Refinement Verification Techniques

Our methodology requires a proof of a TLA-style state-machine refinement (§2.2) between a StorageIOSystem and our application spec. Due to the complexity of this proof, we break the refinement into a sequence of smaller refinements (§6.2.1). We use the following techniques to organize the proof, separating concerns between the caching subsystem, the journal subsystem, and B$^\varepsilon$tree manipulations.

State-machine composition. In many cases, we define a templated state machine $S\langle T\rangle$ in terms of an abstract subcomponent T. A refinement T' of the subcomponent T can be lifted to a refinement $S\langle T'\rangle$ of the larger state machine $S\langle T\rangle$. This allows us to build up our refinement in terms of refinement proofs for the subcomponents. For example, a B$^\varepsilon$tree tree refines a simple dictionary spec; therefore, a *crash-safe* B$^\varepsilon$tree refines a *crash-safe* dictionary spec.

IOSystem Refinement. Our proof re-uses the concept of an

[1] Therefore, our checksum routine, CRC32C [8], exists within our TCB, so that it can be referenced by our disk model.

IOSystem at several layers of abstraction. For example, at the lowest layer, we model the disk as storing sequences of bytes, but at higher layers, we model the disk in a “type-safe” way, as a collection of nodes. But at all layers, the overall system follows the rules of an IOSystem state machine.

Transposition. High in the abstraction stack, the disk is used in different ways by different modules. For example, the journal models the disk as an array of journal entries, whereas the B$^{\varepsilon}$tree models the disk as storing nodes. At a low level, all the modules together interact with a single, byte-oriented disk. Transposition arguments enable us to split up one StorageIOSystem into multiple, so each can be reasoned about independently.

4 Disciplined Automation

A productive verification workflow uses the developer's time efficiently. This has two aspects: how much tedious typing the developer has to do, and how quickly the verifier replies to the developer's proof attempts.

Functional verification of arbitrary software is undecidable, and hence requires either a limit to expressivity [49, 54, 64] or some degree of manual guidance. In interactive theorem provers [15, 47], developers manually use tactics to tell the prover which steps to take next.

A large verified system has a large number of definitions, such as invariants and state-machine transition relations. An automated program verifier is a great fit for systems verification because so much of the verification work is tedium amenable to automation. Exposing all the definitions to the theorem prover, however, gives the theorem prover a large search space, which can take a long time to explore.

The development cycle is a balance between exploiting automation and controlling it. Faced with a verification failure, the developer must first supply any guidance not provided by automation. That process terminates when the proof passes (because the failure was a weakness of the automation) or the developer discovers an actual flaw. If automation is too weak, the developer burns time tediously typing in the missing proof guidance. If the automation heuristics are too aggressive, the developer burns time waiting for replies from the verifier.

As the system grows, the risk of timeouts grows. We have found it essential to resolve timeouts as soon as they crop up, before there are so many they are difficult to sort out. If a developer observes a $> 20s$ response time, they are expected to stop work and instead resolve the timeout,

The key technique to remedy timeouts is to control how much information the prover has when trying to verify a method or lemma, typically by making fewer definitions visible to the theorem prover. Developers can mark Dafny definitions `opaque`, for example, so that the definitions are hidden by default, except where the developer chooses to explicitly `reveal` the definitions. We use a command-line SMT profiler to pinpoint problematic definitions, i.e., those the solver instantiates too many times in its attempts to construct a proof. Table 7.1 shows that we have followed this discipline with some consistency, and timeouts in VeriBetrKV remain rare.

5 Language Improvements

Verifying low-level systems software means verifying stateful code with in-place updates. Unfortunately, reasoning about updates is painful in the presence of aliasing. Traditional verification tools like Dafny and VCC [13] rely on an SMT solver to reason about aliasing and ownership. For example, Dafny uses dynamic frames [32], where programmers annotate methods with `modifies` clauses to specify which objects each method may modify. With dynamic frames, programmers can write arbitrarily complex expressions to compute the set of modified objects. Programmers can also write arbitrarily complex preconditions and postconditions to specify non-aliasing, usually by specifying the disjointness of various sets of objects used in various modifies clauses. The SMT solver must then reason about these arbitrarily complex expressions, which provides programmers with great flexibility, but painfully slows verification [29, §6.2]. Furthermore, this reasoning is mixed with reasoning about functional correctness properties, often making it confusing for the developer to diagnose errors: does a verification failure indicate something deep about the invariants and states, or just a missing non-aliasing requirement?

Recent work on low-level type-safe languages like Rust [33] point to an alternate strategy, where the language's type checker quickly takes care of memory safety and ensures non-aliasing. Full tools for verifying Rust-like programs [3] are still under development and are not yet as mature as tools like Dafny, Coq, and VCC. Therefore, we use Dafny as a starting point and extend it in a more Rust-like direction in two ways. First, rather than using Dafny's existing high-level code-generation backends, we wrote a C++ backend for Dafny that generates efficient C++ code that does not require a garbage collector (§5.2). Second, we extended Dafny's static type checker to support linear variables (§5.1), which allow us to reason purely functionally about data that can be mutated and manually deallocated. This extended type checking needs no SMT solving and therefore places no burden on the SMT solver. Section 6.1 describes the use of linearity in our implementation. Section 7 quantifies the dramatic reduction in proof code it produces and confirms that our use of linear reasoning has a negligible impact on run-time performance.

Our approach to linearity combines ideas from linear type systems [59] like Cogent [2], linear variables in CIVL [36, 51], and Rust's ownership borrowing. Crucially, our extended type system integrates with Dafny's existing dynamic frames, so that we can use linearity to speed verification where possible and fall back to dynamic frames when we need more flexibility. This allows us to verify the safety of highly-aliased code that would require run-time checks or unsafe code in Rust, or would fall outside Cogent's linearity restrictions.

```
function method seq_get<A>(shared s:seq<A>, i:uint64) : (a:A)
function method seq_set<A>(linear s1:seq<A>, i:uint64, a:A)
  : (linear s2:seq<A>)
function method seq_free<A>(linear s:seq<A>)

method M(a:array<uint64>, o:seq<uint64>,
         linear l:seq<uint64>, shared s:seq<uint64>) ... {
  linear var l2:seq<uint64> := l; // ok: consumes l
  linear var l3:seq<uint64> := l; // error: l already consumed
  var n:uint64 := seq_get(l2, 10); // ok: borrows l2
  l2 := seq_set(l2, 10, 20); // ok: consumes l2, then restores l2
  seq_free(l2); // ok: consumes l2
  seq_free(l2); // error: l2 already consumed
}

class BoxedLinear<A> {
  function Has():bool
  method Give(linear a:A)
    modifies this; requires !Has(); ensures Has(); ...
  method Take() returns(linear a:A)
    modifies this; requires Has(); ensures !Has(); ...
  function method{:caller_must_be_pure} Borrow() : (shared a:A)
    requires Has(); ...
}
```

Figure 2: Using linearity in extended Dafny

5.1 Linear Variables

Since aliasing and mutation are expensive to reason about, we use linearity to express non-aliasing or non-mutation. Specifically, we extend Dafny with a keyword `linear` to express non-aliased, mutable values, and a keyword `shared` to express aliased, immutable values.

Figure 2 shows example code written in our extended version of Dafny. Dafny can express both purely functional code, with no heap modification, and imperative code that allocates and modifies heap data. A Dafny `method` can perform both functional and imperative operations, while a `function method` can perform only purely functional operations. The method `M` demonstrates various kinds of Dafny variables. The variables `a` and `o` use existing Dafny features: `a` is an ordinary pointer to a mutable array in the heap, and `o` is an ordinary immutable sequence. `a` and `o` rely on garbage collection (in C#) or reference counting (in C++) for memory management. The variables `l` and `s` use our extensions to Dafny: `l` is a linear (non-aliased) mutable sequence, and `s` is a shared (potentially aliased) immutable sequence temporarily borrowed from a linear sequence. `l` and `s` do not rely on garbage collection or reference counting. (The declarations `linear l:seq<uint64>` and `shared s:seq<uint64>` are similar in spirit to Rust's declarations `l:&mut[u64]` and `s:&[u64]`, although in Rust's semantics, `l` and `s` are references to sequences, while in Dafny's semantics, `l` and `s` are sequence values, not references.)

The static type checker flags any attempt to duplicate or discard a linear variable like `l` as a type error. In Figure 2, for example, the attempts to assign `l` to both `l2` and `l3` is a type error, as is the attempt to free `l2` twice. The lack of duplication allows efficient in-place updates at run-time, as shown in the call to `seq_set`, which sets one element of a sequence. Despite its efficient implementation, though, the verifier can reason about the call to `seq_set` in a purely functional way [2, 59], without worrying about aliasing and the state of the heap. Like Rust and Cogent, our extension to Dafny allows temporary immutable borrowing from a linear variable, as shown in the call to `seq_get`. Borrowed values are tagged as `shared`, and `shared` variables cannot be returned out of the scope of borrowing.

We also extended Dafny to support linear fields in data structures and linear elements in sequences. In contrast to purely linear systems [2], our system can verify the interoperation between the linear data structures and ordinary heap data (like `array`). First, it supports linear-to-ordinary references: linear data structures can hold ordinary fields and sequence elements, such as Dafny heap pointers and arrays.

Second, to support ordinary-to-linear references, our extension provides a novel trusted class `BoxedLinear<A>`, shown in Figure 2, which stores linear values in ordinary objects. Each `BoxedLinear` object is an ordinary heap object, and references to the object can be freely duplicated, allowing complex aliasing. However, to take the linear value out of a `BoxedLinear` object, the program must prove that the object currently has the linear value. Taking the linear value sets `Has` to false, so that a program can't take the same linear value more than once. This prevents the linear value from being duplicated. In addition, pure functional code (`function methods`) can `Borrow` directly from `BoxedLinear` without modifying `Has`. Restricting the scope of the borrowed value to functional code ensures that no imperative `method` can make `Has` false during the borrow. This approach shows the power of combining SMT solving with type system linearity: linear variables bring economy and clarity to the common cases, while SMT reasoning allows greater flexibility (e.g. using lemmas to prove `Has() == true`) when needed.

5.2 Compiling to C++

Dafny compiles its code to C#, Java, JavaScript, and Go. When building a storage system, however, we want more control over memory layout and management than what these high-level, garbage-collected languages offer.

Hence, we have added a new C++ backend to Dafny. We compile Dafny's immutable datatypes to C++ `structs`, and Dafny's classes and arrays to reference-counted pointers to their C++ equivalents. We implement Dafny's immutable sequences using a C++ `struct` that contains a shared pointer to the underlying values, along with an offset and length into those values. This allows us to optimize operations that extract portions of a sequence; because sequences are immutable, it is safe to implement such operations by returning a new `struct` pointing at the same underlying values but with a different offset and length, rather than copying the values.

When compiling linear variables, we perform updates in-place, rather than copying. Since linear variables cannot be silently discarded, we can rely on explicit deallocation (e.g., `seq_free`) and do not need to reference count them.

Finally, the backend deliberately does not compile Dafny's mathematical integers; it expects the programmer to use Dafny's refinement types to define machine integers that can be (provably) safely compiled to standard C++ integer types.

6 VeriBetrKV: A High-Performance, Verified Key-Value Store

We present a high-level overview of VeriBetrKV's implementation (§6.1) and the structure of its safety proof (§6.2).

6.1 VeriBetrKV's Implementation

VeriBetrKV implements a copy-on-write B$^{\varepsilon}$tree with a logical journal for efficient syncs.

6.1.1 B$^{\varepsilon}$Tree Background

A B$^{\varepsilon}$tree [7] is a write-optimized structure that combines ideas from B-trees and LSM-trees to dramatically improve insertion performance versus a B-tree.

For our purposes, B$^{\varepsilon}$trees have two key properties:

- They support random insertions an order of magnitude faster than B-trees. They achieve this speedup by accumulating newly inserted key-value pairs high in tree and "flushing" items from parent to child in large batches.
- They typically use much larger nodes than a B-tree. B$^{\varepsilon}$tree nodes are often in the range of 1-4MiB, whereas B-tree nodes are in the range of 4-64KiB. This is because B$^{\varepsilon}$trees perform node updates in batch, and hence can afford to update large nodes without incurring high write amplification. As a consequence, queries in an "off-the-shelf" B$^{\varepsilon}$tree are slower than in a B-tree, since each cache miss must read a larger node. Production B$^{\varepsilon}$trees contain optimizations to overcome this problem.

See Bender, et al. [4, 5] for a full exposition of B$^{\varepsilon}$trees and an analytical framework for analyzing their performance.

In VeriBetrKV, we use 2MiB nodes on hard disk, 128KiB nodes on flash, and a fanout of 8.

6.1.2 Node-Buffer Data Structures

Each node in a B$^{\varepsilon}$tree contains a buffer of key-value pairs. VeriBetrKV has two representations for in-memory nodes: a serialized format and an in-memory search-tree format. The former avoids marshaling and demarshaling costs for nodes low in the tree, which are updated through batch flushes. We use the search-tree representation only for the root node, which must support single insertions of new key-value pairs from the user.

The in-memory search tree is one of the VeriBetrKV's most performance-critical components. Thus, we originally wrote it using Dafny's dynamic-frames heap reasoning, making it one of the most difficult pieces of code in our implementation.

We then rewrote it using our linear type system (§5). From the verifier's perspective, this eliminated all heap-mutating code. Furthermore, the type system gave immediate feedback on linear typing errors, enabling rapid development. Section 7 quantifies the reduction in proof code and shows that the shift to linear reasoning had no noticeable impact on performance.

The in-memory search tree also demonstrates the utility of the integration between our linear type system and Dafny's builtin Floyd-Hoare reasoning. Each node in our in-memory search tree maintains a linear sequence of linear (pointers to) children. When we split a node, we need to copy half of those child pointers to the new left-hand node, and half of them to the new right-hand node. In a standard linear type system, such as in Rust, we cannot "take" a subset of the values out of a linear array, and we would have to resort to unsafe code or incur the run-time overhead of using an `Optional` type for each array element.

With our linear type system, each linear sequence `s` has an associated boolean ghost[2] sequence, `lseq_has(s)`, that serves the same purpose as the `Has` predicate of the `BoxedLinear` class in Figure 2. When we remove the child pointers for the new left-hand node the ghost sequence becomes `false` for each index `i` that we take. However, this does not prevent taking the remaining children for the right-hand node, since Dafny infers that `lseq_has(children)[i]` is still `true` for those indices.

6.1.3 Caching, Copy-on-Write, and Indirection Tables

VeriBetrKV maintains a cache (BlockCache) of recently accessed nodes, using an LRU eviction policy. The cache is free to write back a node at any time, which is safe because VeriBetrKV uses copy-on-write. The BlockCache is oblivious to the data structure it caches. It resembles a kernel buffer-cache, except **(a)** its allocation unit is a type parameter so we can allocate tree nodes, **(b)** it tracks inter-block references and garbage collects blocks.

VeriBetrKV implements crash safety by maintaining three copy-on-write B$^{\varepsilon}$trees: a *persistent* tree, a *frozen tree*, and an *ephemeral* tree. New inserts go into the ephemeral tree, the frozen tree is in the process of being made durable, and the persistent tree is the previous tree that was made durable.

Each tree is defined by an *indirection table*, which maps logical node IDs to physical disk addresses. Parents refer to children by logical node ID, and the cache is indexed by logical ID. Since nodes are large, the indirection table is small. For example, the indirection table for a 1TiB disk is only 24MiB.

To sync the tree to disk, we write out all dirty nodes, write the indirection table, and then write a superblock pointing to the indirection table. VeriBetrKV keeps two superblocks,

[2] i.e., a data structure used for proof purposes only, not compiled code

alternating between them, and using a counter to detect which one is newer. We call this process a *checkpoint*.

VeriBetrKV ensures that checkpoints capture a point-in-time-snapshot as follows. When we begin a checkpoint, all dirty nodes in the live indirection table are marked as “write-back-before-editing”. Since these nodes are now also referenced by the indirection table of the in-progress checkpoint, we must write them to disk before modifying them in cache. Note that we need not wait for the write to complete: if the system crashes before the write completes, then the system will boot from the previous checkpoint.

The indirection table in VeriBetrKV is implemented as a linear-probing hash table. As with the in-memory search tree, we initially implemented this using Dafny’s dynamic frames. In order to isolate the complexities of heap reasoning, we essentially wrote the hash table twice. The first version used immutable data structures, and served as a precise, low-level description of the hash table’s behavior. We then wrote it a second time using mutable arrays, proving that each step exactly followed the algorithm in the functional model. This approach separated the proof of functional correctness from the proof of correct heap manipulation, but it meant implementing the hash table twice.

We then reimplemented this hash table using our linear Dafny extensions. In this version, the low-level functional model of the algorithm, suitably annotated with `linear` and `shared` keywords, *is* the implementation, cutting the amount of code substantially (§7).

6.1.4 Optimizing Syncs with a Journal

When applications perform frequent syncs, writing out every dirty node for each sync becomes expensive. For example, if an app performs a sync after every insert, then each sync must write out the root node (2MiB) to persist a single insertion.

We solve this problem with a journal of logical operations. Each insertion is recorded in an in-memory journal as well as inserted into the B$^{\varepsilon}$tree. When the application requests a sync, we simply write the in-memory journal to disk.

Journal space is reclaimed as part of a checkpoint.

6.2 VeriBetrKV’s Proof

VeriBetrKV weaves several modularity techniques together to manage the complexity of the code and its correctness argument. We use modular Hoare logic to reason about implementation code and reusable templated state machine models and IOSystem composition (§3.2) to reason about how that code behaves in an asynchronous environment. IOSystems generalize well to this new context of storage systems, and reuse well as we develop a correctness argument up through layers of abstraction. Overall, this blend of techniques is sufficient to modularize the complexity of a modern high-performance storage architecture.

6.2.1 VeriBetrKV’s Refinement Proof

Below, we describe how simple state-machine refinement suffices to reason about the correctness of our B$^{\varepsilon}$tree assuming it existed in memory on a crash-free machine (§6.2.2).

To manage complexity, we modularize our proof by separating the reasoning about the journal subsystem and the crash-safe B$^{\varepsilon}$tree storage subsystem. We further modularize the B$^{\varepsilon}$tree proof by separating caching logic from B$^{\varepsilon}$tree manipulation logic.

6.2.2 Simple State-Machine Refinement

State-machine refinement enables designers to organize high-level correctness arguments in isolation from its low-level details, and then ignore abstract concerns when writing implementation code [40]. For example, suppose we want to prove that a single, unfailing process correctly implements an in-memory B$^{\varepsilon}$tree (① in Figure 3).

- The application spec is a *Map* that updates and queries a key-value relation.
- An *abstract B$^{\varepsilon}$tree* inserts messages into nodes arranged in a tree, where each node is an infinite map. This model has enough detail to define query semantics over that tree, but the (unimplementable) nodes elide detail that is addressed below.
- The *B$^{\varepsilon}$tree* defines the node data structure that clumps the infinite key range into finite buckets at pivot keys.
- *ImplB$^{\varepsilon}$tree* is compilable real imperative code, organized with Hoare logic, with details like mutable data structures and 64-bit integer overflow (gray in the diagram because VeriBetrKV does not have a purely in-memory B$^{\varepsilon}$tree).

The refinement arrow between the *B$^{\varepsilon}$tree* and the *abstract B$^{\varepsilon}$tree* concerns only the relationship between pivot nodes and infinite-map nodes; it ignores higher-level concerns (the tree shape) and lower-level concerns (efficient data structures). This application of refinement gives the developer the freedom to modularize the correctness argument.

6.2.3 VeriBetrKV’s IOSystem Refinement

Of course, VeriBetrKV’s B$^{\varepsilon}$tree does not necessarily run without crashing, and it interacts with an asynchronous byte-level disk, so that the full data structure is stored on disk but cached in memory. Hence, to prove its safety, we repeatedly apply the techniques from §3.2 to prove that the storage IOSystem state machine (§3), when instantiated with VeriBetrKV’s program state machine, refines the application-facing specification (§3.1.2).

A good specification of a crash-safe system needs to be able to describe how data is recovered upon crash. Our specification has an *ephemeral* state and a *persistent* state. All user operations (e.g., queries and inserts) are applied to the ephemeral state; if there are no crashes, the user will observe the behavior of a simple dictionary. On a crash, the ephemeral state reverts to the persistent state; the persistent state therefore represents the state made persistent to disk. Background operations can

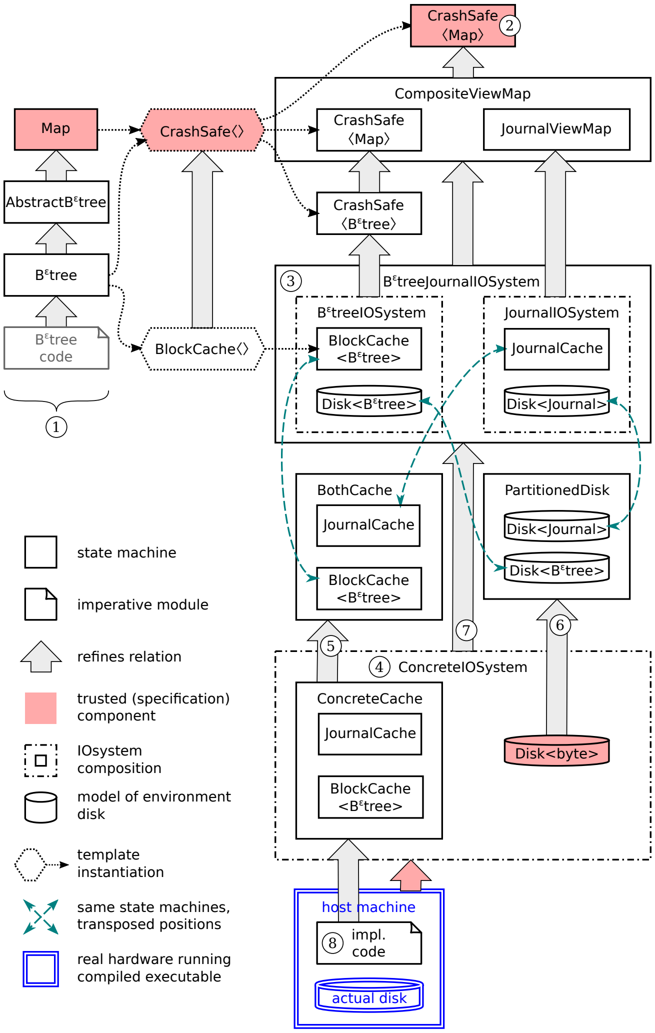

CrashSafe ⟨Map⟩
2
CompositeViewMap
Map
CrashSafe⟨⟩
CrashSafe ⟨Map⟩
JournalViewMap
AbstractB$^{\varepsilon}$tree
CrashSafe ⟨B$^{\varepsilon}$tree⟩
B$^{\varepsilon}$tree
3
B$^{\varepsilon}$treeJournalIOSystem
B$^{\varepsilon}$treeIOSystem
JournalIOSystem
BlockCache <B$^{\varepsilon}$tree>
JournalCache
B$^{\varepsilon}$tree code
BlockCache⟨⟩
Disk<B$^{\varepsilon}$tree>
Disk<Journal>
1
BothCache
JournalCache
BlockCache <B$^{\varepsilon}$tree>
PartitionedDisk
Disk<Journal>
Disk<B$^{\varepsilon}$tree>
state machine
imperative module
refines relation
trusted (specification) component
IOsystem composition
model of environment disk
template instantiation
same state machines, transposed positions
real hardware running compiled executable
5
7
6
4 ConcreteIOSystem
ConcreteCache
JournalCache
BlockCache <B$^{\varepsilon}$tree>
Disk<byte>
host machine
8 impl. code
actual disk

Figure 3: Structure of VeriBetrKV's correctness argument.

update the persistent state to a newer state. Two transitions, `sync_start` and `sync_end` are defined such that, when `sync_end` runs, the persistent state will have updated to the ephemeral state from the `sync_start` (or a newer version). We call this generic specification CrashSafe⟨T⟩, parameterized over a state machine T. In our case, the top-level specification will be CrashSafe⟨Map⟩.

The CrashSafe⟨T⟩ template is an abstraction of our BlockCache⟨T⟩ template state machine, which interacts with a disk but is oblivious to the data structure it caches. We prove generically that if concrete type T_{conc} refines an abstract type T_{abs}, then an IOSystem containing a BlockCache⟨T_{conc}⟩ and a disk refines a CrashSafe⟨T_{abs}⟩.

We apply this generic result at the next level down ③, where we instantiate the type-oblivious BlockCache⟨T⟩ template with the IO-oblivous B$^{\varepsilon}$tree (§6.2.2). Leveraging refinement ①, this proves that B$^{\varepsilon}$treeIOSystem refines CrashSafe⟨B$^{\varepsilon}$tree⟩ and hence CrashSafe⟨Map⟩.

In a sibling library, we prove that the JournalIOSystem refines JournalViewMap, an abstract summary of journal behavior, including components for journal entries persisted to disk, journal entries being written, and journal entries in memory. Ordinary state machine composition pulls those together into the CompositeViewMap, an abstract summary of the state-machine composition B$^{\varepsilon}$treeJournalIOSystem. The CompositeViewMap is shown to implement a CrashSafe⟨Map⟩: an abstraction function applies updates in JournalViewMap's journals to the map states in CrashSafe⟨Map⟩ to obtain the application-spec CrashSafe⟨Map⟩.

Thus, we have refined from the application spec to a model ③ of the VeriBetrKV's two main components, each modeled as separate systems, each of which uses a high-level "disk" that stores its client's internal datatype representation.

One layer down, the ConcreteIOSystem ④ introduces a real byte-level disk-IO interface. The program component of this IOSystem is the ConcreteCache, which includes marshaling and demarshaling functions on top of the JournalCache and BlockCache⟨B$^{\varepsilon}$tree⟩. The ConcreteCache refines ⑤ the BothCache. The disk component of the ConcreteIOSystem is our low-level disk model, the Disk⟨byte⟩, which refines ⑥ PartitionedDisk via the same marshaling functions. This refinement relies on an invariant that Disk⟨Journal⟩ and Disk⟨B$^{\varepsilon}$tree⟩ access mutually-disjoint regions of the disk. With these two refinements in place, we transpose the four subcomponents (dashed green arrows) and obtain a refinement ⑦ from ConcreteIOSystem to B$^{\varepsilon}$treeJournalIOSystem. This rearrangement is crucial to allow us to reason about BetreeIOSystem and JournalIOSystem separately.

ConcreteCache is the lowest-level model of the program, including all of the components (B$^{\varepsilon}$tree, journal, indirection table, byte-level IO interface). It remains to show that our imperative heap-mutating code ⑧ refines ConcreteCache. To do so, we show that each handler invocation in the code corresponds to a `Next` transition in the ConcreteCache state machine. Because ConcreteCache is a low-level model, this task decomposes nicely along Hoare-logic call-graph boundaries: calls to update the journal advance the JournalCache sub-component, leaving the B$^{\varepsilon}$tree unchanged, and vice versa.

6.2.4 VeriBetrKV's Floyd-Hoare Proof

The top-level API methods of VeriBetrKV's implementation (⑧) use Floyd-Hoare logic to show that their operations correspond to transitions of the ConcreteCache state machine. Of course, the ConcreteCache is only one component of the ConcreteIOSystem (④), and likewise, the implementation code interacts with the disk only via a trusted interface.

At the implementation level, we do not reason about the disk itself—we reason only about interactions via the trusted interface. Transitions of the ConcreteCache state machine are labeled with *disk ops*. Each disk op is either a *no-op* (i.e., no disk interaction), an I/O *request*, or an I/O *response*. The disk-op labels are "visible" to the ConcreteIOSystem: the ConcreteIOSystem state machine is defined in terms of the

Major component	spec	impl	proof
Map, CrashSafe⟨Map⟩	283	82	818
AbstractB$^{\varepsilon}$tree	0	70	2024
B$^{\varepsilon}$tree	0	137	7079
CompositeViewMap	0	26	823
B$^{\varepsilon}$treeIOSystem	0	246	6510
ConcreteIOSystem	270	68	2887
implementation code	180	5380	21697
libraries	477	364	2847
total	1210	6373	44685

Table 1: Line counts [60] by major components in Figure 3.

	hash table		search tree	
Aliasing reasoning	impl	proof	impl	proof
Dynamic frames	289	1678	289	2220
Linear type system	289	1063	373	1531

Table 2: Line counts [60] of two subcomponents, comparing dynamic-frame implementations with our linear type system.[3] Linear typing reduces the proof burden by 31–37%

ConcreteCache's interaction with the disk via disk ops.

Thus, the Floyd-Hoare logic in our implementation does not need to reason about the disk proper. Rather, it only needs to show that each interaction with the trusted disk interface corresponds to a disk op which is valid according to the ConcreteCache state machine.

Overall, our proof shows that an executable built from our implementation's code, if run on a real host with a real disk that meets our assumptions, will behave as the ConcreteSystem does, and hence as a CrashSafe⟨Map⟩. We have connected not just code, but a system with a disk and the possibility of the code crashing, up to the app spec.

7 Evaluation

Our evaluation addresses two main questions:

1. Do our automation-control techniques (§4) and language improvements (§5) improve the developer experience?
2. Can our verification methodology scale to the complexity of a modern key-value-store data structure, and can we deliver the performance gains of write optimization?

7.1 Developer Experience

Measuring Tedium. We estimate the amount of tedium (or conversely, the efficacy of automation) by the ratio of the lines of proof (e.g., lemmas, pre-/post-conditions, loop invariants) to the lines of executable implementation code. This is not a precise model, since it measures a completely verified artifact, where the developers may have cleaned up temporary lines of tedium that were typed in the course of resolving verification failures. However, the proof text in the "cleaned up" code at least reflects the tedium needed to bridge what automation could not manage by itself.

Table 1 gives line counts for VeriBetrKV, organized by the major components shown in Figure 3. We see that the proof ratio for the implementation code is 4:1, which grows to 7:1 when including all system refinement proofs ("total"). This is comparable to the distributed, in-memory key-value store verified in previous work [29], which also reports a 7:1 ratio. However, VeriBetrKV's implementation is 3× larger, indicating that automated verification techniques can scale to larger systems without super-linear effort.

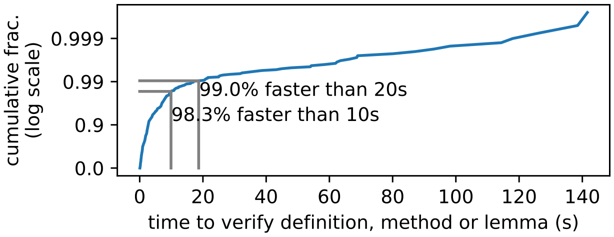


Figure 4: Cumulative distribution of verification times of function definitions, implementation methods, and proof lemmas. Most definitions–99.0%–verify in less than 20s, and 98.3%–verify in less than 10s.

The results also show that VeriBetrKV's implementation is more than 5× larger than its specification, giving us reason to hope that the specification is less likely to contain bugs than an unverified implementation. We have observed no correctness or data loss bugs at runtime; we have seen liveness and performance bugs.

Table 2 compares two VeriBetrKV components that we wrote using both dynamic frames and linear reasoning. The results show that switching to linear reasoning saves tedium, reducing proof overhead by 31–37%. Our qualitative experience was that development of linear code was much more pleasant than dynamic frames because the linear typechecker quickly and unambiguously identifies aliasing problems.

One interesting datum for tedium is the effort developers spend on test infrastructure in conventional development. RocksDB has a 0.99:1 ratio between test and production code (measuring its `db`, `java`, `utilities`, `third-party`, and `tools` directories). BerkeleyDB's ratio is 0.45.

Measuring Proof-Attempt Latency. To assess the success of our discipline for keeping automation under control (§4), we measure the time to verify individual proof units (e.g., definitions, methods, or lemmas) where developers spend most of their time waiting for verification results. This estimates the developer's perception of the latency of the verification-development cycle.

Figure 4 demonstrates that VeriBetrKV almost always exhibits interactive verification times, with 98.3% of definitions verifying in under 10 seconds, and 99.0% in under 20 seconds. This suggests that our timeout-averse development policy is

[3] The implementation of the hash table (with dynamic frames and the linear type system) and search tree (with dynamic frames) coincidentally have identical line counts, despite being unrelated.

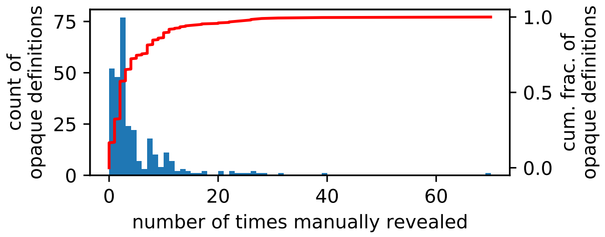


Figure 5: VeriBetrKV manually hides 308 definitions (18% of all system definitions), revealing them 1403 times. This metric reflects the effort involved in explicitly managing automation.

effective. The figure reveals that we did not apply our policy with perfect consistency, as 37 proofs do take longer than 20s to verify. Of those, 54% involve dynamic frames.

It is difficult to capture every place we applied the policy. As a proxy, we can count places where we explicitly hid a definition. This both overestimates the cost of the policy (because sometimes we hid a definition based on intuition, before observing a timeout) and underestimates it (because automation can be controlled with other techniques).

These approximations aside, Figure 5 gives an idea of the burden of controlling automation. We hid 308 definitions, 18% of all definitions in the system. These hidden definitions are manually revealed 1403 times. Of the 308, 52 were never revealed: the salient properties of the definition could be exported as a postcondition without causing timeouts. 74% of hidden definitions are revealed no more than five times; their essential features are captured in lemmas or wrapped into higher-level definitions.

One of the motivations for introducing linear types into Dafny (§5) is to remedy slow verification of dynamic frames. For the fragment of VeriBetrKV we converted to linear reasoning, we compared interactive verification times (as in Figure 4) against the original dynamic frames code. The maximum method-level interactive verification time dropped 10s to 32s, and the 99th percentile dropped 1.3s to 4.8s.

Developers are typically less sensitive to the latency of continuous-integration builds that check that the system as a whole still verifies. For VeriBetrKV, these take 1.8 hours of CPU time, but thanks to the inherent parallelism of modular verification, complete in 11 minutes on 32 cloud machines.

7.2 Performance

Our performance evaluation addresses two questions:

1. Does VeriBetrKV demonstrate the insertion-performance gains of write-optimized data structures?
2. Does our linear extension produce code with performance comparable to hand-written code using dynamic frames?

All experiments were run on cloud instances with directly attached physical storage. The HDD experiments and subcomponent microbenchmarks are run on AWS EC2 d2.xlarge instances, with 4x hardware hyperthreads on a 2.4 GHz Intel Xeon E5-2676 v3. The SSD experiments are run on AWS EC2 i2.xlarge instances, with 4x hardware hyperthreads on a 2.5 GHz Intel E5-2670 v2 and a SATA SSD.

7.2.1 YCSB

Figure 6 shows throughput for VeriBetrKV, BerkeleyDB, and RocksDB on the YCSB benchmarks [14] on hard drive and SSD, including the load phase for workload A, and a uniformly random query workload (labeled as workload "U"). All systems are limited to a single core.

There are three main take-aways from these measurements. First, VeriBetrKV demonstrates the performance gains of using a write-optimized data structure. For the load phase on hard drive, which consists of a pure random insertion workload, VeriBetrKV is over 25× faster than BerkeleyDB. Even on SSD, where random I/O is much cheaper, VeriBetrKV modestly outperforms BerkeleyDB on insertions.

In contrast, VeriBetrKV is roughly 8× slower than RocksDB on hard disk and 4× slower on SSD. Investigation identified three contributing factors: First, where Rocks relies on the kernel buffer cache, VeriBetrKV manages its own cache memory. Its effective cache size is reduced due to malloc fragmentation and conservative allocation to avoid violating the cgroup. Simulating an efficiently-allocated 1.8GiB B^{ε}tree node cache improves performance to over 13K insertions per second on HDD. Second, VeriBetrKV fails to overlap CPU with flush I/O: a twelve-minute run spends four minutes in CPU and eight minutes waiting for I/O, accounting for roughly a $2/3\times$ slowdown. Third, VeriBetrKV performs 1.5× more I/O than Rocks in YCSB-Load; about half of the extra I/O is due to a suboptimal checkpointing policy.

The second main take-away is that queries in VeriBetrKV are about 4× slower than on RocksDB. The fragmentation penalty again explains a 2× factor; with a simulated 1.8GiB cache, VeriBetrKV and Rocks both perform 300K I/Os in serving 1M YCSB-C queries. Furthermore, we observed most Rocks I/Os are 4-8KiB (one or two buffer cache pages), whereas most VeriBetrKV I/Os are 1.5MiB (an entire B^{ε}tree node). This I/O size difference combined with the measured seek and read bandwidth of our hard drives explains the remaining gap. We plan to change our marshalling strategy to enable reading fields without fetching the entire node.

The final take-away is that at a macro-level our linear implementation has essentially the same performance as the version with hand-tuned code using dynamic frames reasoning.

Overall, we conclude that VeriBetrKV demonstrates that a verified system can achieve the performance gains of a write-optimized storage system, but it needs further optimization to match highly-tuned commercial implementations.

7.2.2 Linear Data Structures

Figure 7 shows the performance of our linear and dynamic-frames-based hash-table and search-tree implementations. The main take-away from both experiments is that, even in mi-

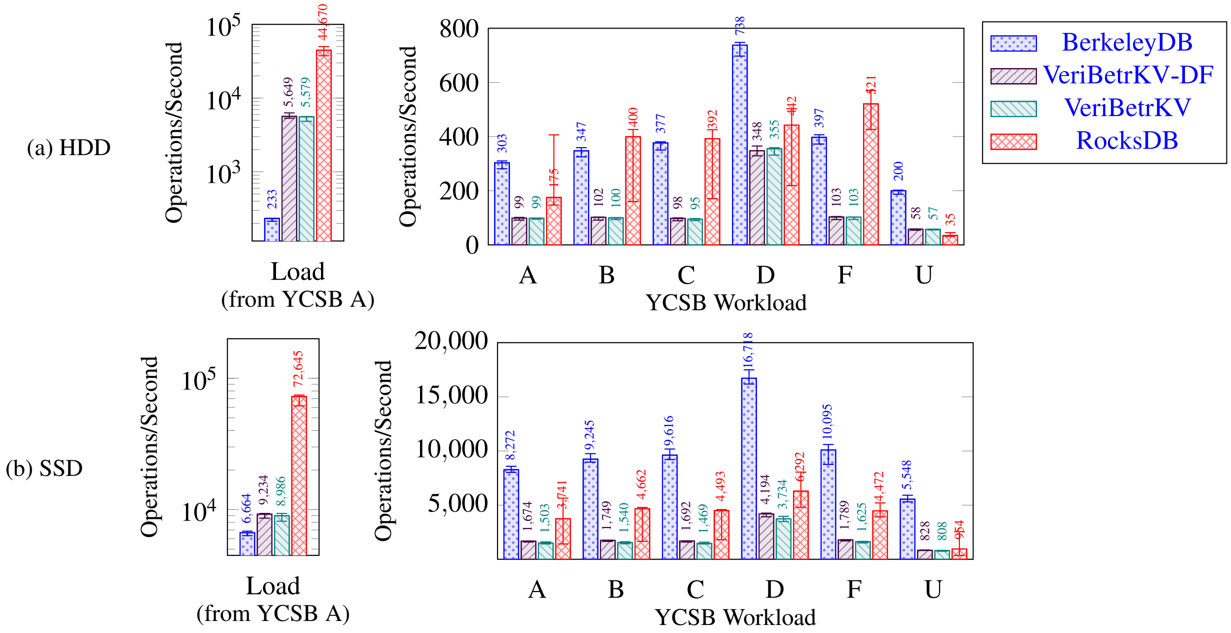
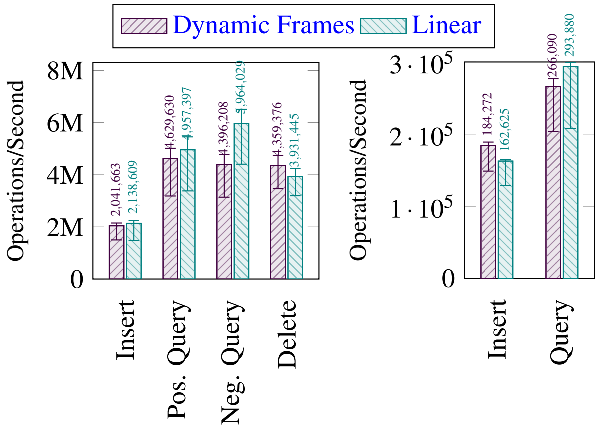


Figure 6: Median throughput of YCSB workloads values running on an HDD (a) and SSD (b) and 2GiB of RAM. VeriBetrKV-DF is a version of our system with hand-tuned code using dynamic frames reasoning. Load is 10M operations (≈ 5GiB of data) and runs are 10000 operations each. Error bars indicate min/max of 6 runs. Higher is better. On an HDD (a), VeriBetrKV insertions are over 25× faster than in BerkeleyDB, but lag RocksDB, by about 9×. VeriBetrKV queries are are about 4× slower than both. On an SSD (b), VeriBetrKV still beats BerkleyDB on random insertions, but VeriBetrKV queries are slower than BerkeleyDB.



(a) Hash table

(b) Search tree

Figure 7: Median throughput of subcomponent microbenchmarks. Higher is better. Error bars are min/max of 6 runs.

crobenchmarks, the linear and non-linear implementations have very comparable performance.

The hash table benchmark inserts 64 million key-value pairs, performs 64 million positive and 64 million negative queries, and then deletes everything in the hash table. The keys are selected pseudo-randomly and are distributed uniformly in the 64-bit key space. The linear version is slightly faster than the non-linear version, except for deletes, which are slightly slower. We suspect the speedup comes primarily from the lack of shared pointer overhead.

The search-tree benchmark measures the time to insert 8 million key-value pairs in pseudo-random order and then query them all in the same order. Performance for the linear version is close to the non-linear version, but slightly faster for queries and slightly slower for inserts. We believe queries are faster due to the elimination of shared pointer overheads, and the insertions are slower due to the overheads of destructing nodes on the way down the tree and reconstructing them on the way back up.

Our modifications to add linear types to Dafny consist of 1900 lines (3%); the C++ backend changes, which include linear features, add another 3100 lines (7.5%).

Overall, our linear type system enables us to construct performant code without the challenges of dynamic frame reasoning.

8 Related Work

IronFleet [29], VeriBetrKV's most direct intellectual ancestor, verifies distributed systems of fail-stop nodes. Its verification strategy uses a refinement hierarchy with just two layers: one from imperative code to a protocol state machine, another from there to application spec. VeriBetrKV's implementation contains more components (an in-memory B-tree and hash table, a block cache, and a journal), and the B$^\varepsilon$tree, its core data structure, is substantially more complicated. Hence VeriBetrKV's implementation is 3× larger than IronFleet's sharded distributed key-value in-memory-only store.

8.1 Verified Storage Systems

FSCQ's Crash Hoare Logic[12] modifies Hoare logic to explicitly reason about crashes, enabling crash reasoning to follow Hoare clauses up the implementation call graph. It does not employ refinement reasoning. FSCQ reasons about potentially repeated crashes during the recovery procedure; VeriBetrKV avoids such reasoning by virtue of a design whose recovery procedure requires no disk writes. FSCQ's implementation is functional code extracted to Haskell, so the framework gives the developer less low-level control than VeriBetrKV.

DFSCQ [11] contributes an application spec for how crashes interplay with the separate `fsync` and `fdatasync` operations. Production file systems like ext4 provide these operations to offer greater performance at the cost of an application spec even more relaxed than the general `sync` operation that covers all updates (as in VeriBetrKV). DFSCQ's implementation exploits this freedom to defer writes.

Yggdrasil uses refinement to show implementation functional correctness relative to a specification [54]. Crashes in the environment are implicit, and the app spec only promises linearity, requiring the implementation to sync on every mutating client operation (or group commit). It cannot exploit the performance benefit of deferring writes until an application-specified `sync`. Its disk model includes asynchronous I/O but has no concept analogous to an IOSystem. Its pushbutton approach to verification constrains the structure of the implementation and proof, but in exchange produces a very favorable proof:code ratio, reported at 1:300.

As discussed in §5, our linear extensions to Dafny are inspired by multiple sources [2, 33, 36, 51, 59]. Most relevant to systems verification is the Cogent language [2], which is a restricted functional language that certifiably compiles to C code. Amani et al. use Cogent to develop two file systems, each about 4K lines of native C. They verify two functional correctness properties of one of the file systems, with the aim of eventually proving both functional correctness and crash safety. The Cogent language is an impressive foundational effort and its certifying compiler provides a stronger guarantee than our simple but trusted changes to Dafny's type system. Our type system's linear variables are similar in spirit to Cogent's linear types, but our work also integrates linearity directly with Dafny's dynamic frames, enabling us to move smoothly back and forth between linear and non-linear reasoning about memory.

It is difficult to make meaningful performance comparisons between our durable key-value store and file systems: File systems provide richer semantics, such as bulk directory rename.

8.1.1 Concurrent Storage

AtomFS [66] is a compute-concurrent file system with fine-grained (per-inode) locks, but it does not reason about crash safety. CSPEC [9] verifies a compute-concurrent mail server absent crash safety. It verifies 215 lines of Coq implementation with 4,050 lines of proof. Perennial [10] verifies a compute-concurrent, crash-safe mail server. Perennial extends a capability separation logic with crash-safety-specific concepts, with which it builds a refinement argument. Perennial verifies 159 lines of concurrent Go with 3,360 lines of proof. Drawing on techniques from these systems would allow VeriBetrKV to scale further via parallelism.

8.2 Automation Strategies

A line of work on "push-button" verification [45, 46, 54, 55, 64, 65] accepts constraints on system structure in exchange for maximizing automation. The developer constrains their imperative code to bounded executions and writes invariants to span independent handler invocations. Such handlers could not walk down a tree of arbitrary depth, as happens in VeriBetrKV's B$^\varepsilon$tree and search tree. Supporting longer bounded executions requires framework improvements [45] rather than creating a modularization job for the developer.

Taube et al. [57] employ a restricted fragment of logic [49] to verify distributed system implementations, including Raft [48] and Multi-Paxos [39]. By restricting the description of the protocol and its properties to a decidable logic, this approach guarantees that a solver can always return either a decisive answer. While the developer still must supply invariants, the remaining proof work is entirely automatic. The cost of this approach is that it force the developer to restate definitions unnaturally, and decidable verification is still subject to combinatorial slowdown as the scope of definitions grows.

The exploration of extreme points in the automation space is promising, but limitations on expressiveness and design motivate us to stick with developer-guided proofs, and instead use automation to make it as cheap as possible.

8.3 Additional Verified Systems

The seL4 verified microkernel is the seminal systems verification project [34], demonstrating the feasibility of verifying software at the scale of thousands of lines. C code refines a Haskell functional model of the implementation, which refines a high-level specification for the behavior of system calls.

CertiKOS [27] proves a concurrent microkernel implementation correct using refinement of state machines it calls "layers" expressed in a side-effect free subset of C. It introduces the notion of contextual refinement to reason about concurrent state machines in isolation [28].

Verdi [61] uses Coq to verify distributed systems by proving the correctness of a series of "system transformers" that take a high-level protocol description running in a friendly environment and transform it into a protocol that is safe in a more hostile environment (e.g., where packets can be dropped). The signature transformer is a verified implementation of Raft [62]. In a sense, Verdi is a distributed systems analog to correctness-preserving compiler transforms.

9 Conclusion

In this work, we extracted a general methodology for verifying asynchronous systems from prior work and applied it to storage systems. In doing so, we developed a verification discipline and a novel integration of linearity with dynamic frame reasoning to reduce the burden of verifying systems code. Because we applied a generic methodology, we expect these improvements to apply equally well to the verification of other asynchronous systems. In future work, we would like to extend the methodology to also support thread-concurrent systems with shared memory, utilizing our linear type system to manage memory ownership.

Ultimately, our implementation and proof of crash safety for VeriBetrKV, a complex, modern storage system, show that automated verification techniques can scale to larger code bases without increasing the proof burden relative to simpler systems.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Gernot Heiser, for useful feedback on the paper. Work at CMU was supported, in part, by grants from a Google Faculty Fellowship, the Alfred P. Sloan Foundation, and the NSF/VMware Partnership on Software Defined Infrastructure as a Foundation for Clean-Slate Computing Security (SDI-CSCS) program under Award No. CNS-1700521. Andrea Lattuada is supported by a Google PhD Fellowship.

A Artifact Appendix

A.1 Abstract

The evaluated artifact is provided as Docker images that contain the source code to VeriBetrKV, build instructions to run the verification as well as run the performance experiments and draw the graphs corresponding to those in §7 with the generated data.

A.2 Artifact check-list

- **Algorithm**: A verified B$^\varepsilon$tree-based key-value store.
- **Program**: VeriBetrKV as described in this paper.
- **Compilation**: Dafny/C++ compiler, included in Docker image.
- **Data set**: YCSB generated workload
- **Run-time environment**: Docker
- **Hardware**: Any x64; provide a data store directory on HDD or SSD as desired
- **Run-time state**: KV store backing files
- **Output**: PDFs containing graphs corresponding to §7
- **Required disk space**: 20GiB
- **Expected experiment run time**: Several hours
- **Public link**: `https://github.com/secure-foundations/veribetrkv-osdi2020/blob/master/README.md`

A.3 Description

A.3.1 How to access

Follow the README at `https://github.com/secure-foundations/veribetrkv-osdi2020/blob/master/README.md`. You can either run the binary Docker distribution, or build it yourself.

A.3.2 Hardware dependencies

You will need any x86 CPU, plus HDD and/or SSD storage devices for the performance measurements.

A.3.3 Software dependencies

All required dependencies are included in the Docker image.

A.3.4 Data sets

Performance experiments use the YCSB benchmark, for which the source and configuration are included in the Docker image.

A.4 Installation

You can either download the GitHub release, `veribetrkv-artifact-hdd`, and load the image with

```
docker load -i veribetrkv-artifact-hdd.tgz
```

or build it yourself with

```
cd docker-hdd
docker build -t veribetrkv-artifact-hdd .
```

A.5 Experiment workflow

The README explains how to launch the experiments by running scripts from outside Docker. The scripts will generate PDFs that reproduce the results from the paper.

A.6 Evaluation and expected result

The graphs in the output PDFs should correspond to those in §7, modulo variation in the experimental hardware.

A.7 Experiment customization

The README at the link above provides details on how to modify the experiment scripts in the Docker container.

A.8 AE Methodology

Submission, reviewing and badging methodology:

- `https://www.usenix.org/conference/osdi20/call-for-artifacts`

References

[1] Abadi, M., and Lamport, L. The existence of refinement mappings. *Theoretical Computer Science 82*, 2 (May 1991).

[2] Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T., Klein, G., and Heiser, G. Cogent: Verifying high-assurance file system implementations. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2016).

[3] Astrauskas, V., Müller, P., Poli, F., and Summers, A. J. Leveraging Rust types for modular specification and verification. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)* (2019).

[4] Bender, M., Pandey, P., Porter, D., Yuan, J., Zhan, Y., Conway, A., Farach-Colton, M., Jannen, W., Jiao, Y., Johnson, R., Knorr, E., McAllister, S., and Mukherjee, N. Small refinements to the DAM can have big consequences for data-structure design. In *Proceeding of the 31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)* (2019).

[5] Bender, M. A., Farach-Colton, M., Jannen, W., Johnson, R., Kuszmaul, B. C., Porter, D. E., Yuan, J., and Zhan, Y. An introduction to B^{ε}-trees and write-optimization. *;login: The USENIX Magazine 40*, 5 (Oct. 2015), 22–28.

[6] Böhme, S., and Weber, T. Fast LCF-style proof reconstruction for Z3. In *Proceedings of Interactive Theorem Proving* (2010).

[7] Brodal, G. S., and Fagerberg, R. Lower bounds for external memory dictionaries. In *Proceedings of the 13th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)* (Baltimore, MD, USA, 2002), pp. 39–48.

[8] Castagnoli, G., Brauer, S., and Herrmann, M. Optimization of cyclic redundancy-check codes with 24 and 32 parity bits. *IEEE Transactions on Communications 41*, 6 (1993), 883–892.

[9] Chajed, T., Kaashoek, M. F., Lampson, B., and Zeldovich, N. Verifying concurrent software using movers in CSPEC. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2018).

[10] Chajed, T., Tassarotti, J., Kaashoek, M. F., and Zeldovich, N. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP)* (Hunstville, ON, Canada, Oct. 2019).

[11] Chen, H., Chajed, T., Konradi, A., Wang, S., Ileri, A., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Verifying a high-performance crash-safe file system using a tree specification. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (Shanghai, China, Oct. 2017).

[12] Chen, H., Ziegler, D., Chajed, T., Chlipala, A., Kaashoek, M. F., and Zeldovich, N. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP 2015)* (Monterey, California, Oct. 2015).

[13] Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., and Tobies, S. VCC: A practical system for verifying concurrent C. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics* (2009).

[14] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (2010).

[15] Coq Development Team. The Coq Proof Assistant https://coq.inria.fr/.

[16] de Moura, L. M., and Bjørner, N. Z3: An efficient SMT solver. In *Proceedings of the Conference on Tools and Algorithms for the Construction and Analysis of Systems* (2008).

[17] Desai, A., Gupta, V., Jackson, E., Qadeer, S., Rajamani, S., and Zufferey, D. P: Safe asynchronous event-driven programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (2013).

[18] Drăgoi, C., Henzinger, T. A., and Zufferey, D. Psync: A partially synchronous language for fault-tolerant distributed algorithms. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2016).

[19] FaceBook, Inc. MyRocks: A RocksDB Storage Engine with MySQL. http://myrocks.io/.

[20] FaceBook, Inc. RocksDB: A persistent key-value store for fast storage environments. https://rocksdb.org/.

[21] Ferraiuolo, A., Baumann, A., Hawblitzel, C., and Parno, B. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2017).

[22] Fisher, K., Launchbury, J., and Richards, R. The HACMS program: Using formal methods to eliminate exploitable bugs. *Philosophical Transactions A, Math Phys Eng Sci. 375*, 2104 (Sept. 2017).

[23] Floyd, R. Assigning meanings to programs. In *Proceedings of Symposia in Applied Mathematics* (1967).

[24] Fonseca, P., Kaiyuan Zhang, X. W., and Krishnamurthy, A. An empirical study on the correctness of formally verified distributed systems. In *Proceedings of ACM EuroSys Conference* (Apr. 2017).

[25] Garland, S. J., and Lynch, N. A. Using I/O automata for developing distributed systems. *Foundations of Component-Based Systems 13* (2000).

[26] Google, Inc. LevelDB. https://github.com/google/leveldb.

[27] Gu, R., Koenig, J., Ramananandro, T., Shao, Z., Wu, X. N., Weng, S.-C., Zhang, H., and Guo, Y. Deep specifications and certified abstraction layers. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (2015).

[28] GU, R., SHAO, Z., CHEN, H., WU, X., KIM, J., SJÖBERG, V., AND COSTANZO, D. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2016).

[29] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving practical distributed systems correct. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (Oct. 2015).

[30] HAWBLITZEL, C., HOWELL, J., LORCH, J. R., NARAYAN, A., PARNO, B., ZHANG, D., AND ZILL, B. Ironclad apps: End-to-end security via automated full-system verification. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (October 2014).

[31] HOARE, T. An axiomatic basis for computer programming. *Communications of the ACM 12* (1969).

[32] KASSIOS, I. T. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM 2006: Formal Methods* (2006).

[33] KLABNIK, S., NICHOLS, C., AND RUST COMMUNITY. The Rust Programming Language https://doc.rust-lang.org/book/.

[34] KLEIN, G., ANDRONICK, J., ELPHINSTONE, K., MURRAY, T., SEWELL, T., KOLANSKI, R., AND HEISER, G. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems 32*, 1 (2014).

[35] KUMAR, R., MYREEN, M. O., NORRISH, M., AND OWENS, S. CakeML: a verified implementation of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)* (Jan. 2014).

[36] LAHIRI, S. K., QADEER, S., AND WALKER, D. Linear maps. In *Proceedings of the 5th ACM Workshop on Programming Languages Meets Program Verification* (2011).

[37] LAKSHMAN, A., AND MALIK, P. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (Apr. 2010), 35–40.

[38] LAMPORT, L. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems 16*, 3 (May 1994).

[39] LAMPORT, L. The part-time parliament. *ACM Trans. Comput. Syst. 16*, 2 (May 1998), 133–169.

[40] LAMPORT, L. *Specifying Systems: The TLA+ Languange and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[41] LEINO, K. R. M. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)* (2010).

[42] LEROY, X. Formal verification of a realistic compiler. *Communications of the ACM (CACM) 52*, 7 (2009), 107–115.

[43] MONGODB, INC. The WiredTiger Storage Engine. http://www.wiredtiger.com/.

[44] MULLEN, E., PERNSTEINER, S., WILCOX, J. R., TATLOCK, Z., AND GROSSMAN, D. Œuf: Minimizing the Coq extraction TCB. In *Proceedings of the ACM Conference on Certified Programs and Proofs (CPP)* (2018).

[45] NELSON, L., BORNHOLT, J., GU, R., BAUMANN, A., TORLAK, E., AND WANG, X. Scaling symbolic evaluation for automated verification of systems code with serval. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019).

[46] NELSON, L., SIGURBJARNARSON, H., ZHANG, K., JOHNSON, D., BORNHOLT, J., TORLAK, E., AND WANG, X. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017).

[47] NIPKOW, T., PAULSON, L., AND WENZEL, M. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, 2002.

[48] ONGARO, D., AND OUSTERHOUT, J. In search of an understandable consensus algorithm. In *Proceedings of the USENIX Annual Technical Conference (ATC)* (June 2014).

[49] PADON, O., MCMILLAN, K. L., SAGIV, M., PANDA, A., AND SHOHAM, S. Ivy: Safety verification by interactive generalization. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2016).

[50] PERCONA LLC. The PerconaFT Storage Engine. https://github.com/percona/PerconaFT.

[51] QADEER, S., TASIRAN, S., AND HAWBLITZEL, C. Automated and modular refinement reasoning for concurrent programs. In *Computer Aided Verification (CAV)* (2015).

[52] REYNOLDS, J. C. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science* (2002).

[53] SCYLLA, INC. ScyllaDB: The real-time big data database. https://www.scylladb.com.

[54] SIGURBJARNARSON, H., BORNHOLT, J., TORLAK, E., AND WANG, X. Push-button verification of file systems via crash refinement. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (Nov. 2016).

[55] SIGURBJARNARSON, H., NELSON, L., CASTRO-KARNEY, B., BORNHOLT, J., TORLAK, E., AND WANG, X. Nickel: A framework for design and verification of information flow control systems. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (2018).

[56] SWAMY, N., HRIŢCU, C., KELLER, C., RASTOGI, A., DELIGNAT-LAVAUD, A., FOREST, S., BHARGAVAN, K., FOURNET, C., STRUB, P.-Y., KOHLWEISS, M., ZINZINDOHOUÉ, J.-K., AND ZANELLA-BÉGUELIN, S. Dependent types and multi-monadic effects in F*. In *Principles of Programming Languages* (2016).

[57] TAUBE, M., LOSA, G., MCMILLAN, K. L., PADON, O., SAGIV, M., SHOHAM, S., WILCOX, J. R., AND WOOS, D. Modularity for decidability of deductive verification with applications to distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (2018).

[58] V. GLEISSENTHALL, K., KICI, R. G., BAKST, A., STEFAN, D., AND JHALA, R. Pretend synchrony: Synchronous verification of asynchronous distributed programs. vol. 3, Association for Computing Machinery.

[59] WADLER, P. Linear types can change the world! In *Proceedings of the IFIP TC 2 Working Conference on Programming Concepts and Methods* (1990).

[60] WHEELER, D. A. SLOCCount. Software distribution. http://www.dwheeler.com/sloccount/.

[61] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)* (June 2015).

[62] WOOS, D., WILCOX, J. R., ANTON, S., TATLOCK, Z., ERNST, M. D., AND ANDERSON, T. Planning for change in a formal verification of the raft consensus protocol. In *ACM Conference on Certified Programs and Proofs (CPP)* (Jan. 2016).

[63] YANG, X., CHEN, Y., EIDE, E., AND REGEHR, J. Finding and understanding bugs in C compilers. In *PLDI* (June 2011).

[64] ZAOSTROVNYKH, A., PIRELLI, S., IYER, R., RIZZO, M., PEDROSA, L., ARGYRAKI, K., AND CANDEA, G. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (2019).

[65] ZHANG, K., ZHUO, D., AKELLA, A., KRISHNAMURTHY, A., AND WANG, X. Automated verification of customizable middlebox properties with Gravel. In *17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020, Santa Clara, CA, USA, February 25-27, 2020* (2020).

[66] ZOU, M., DING, H., DU, D., FU, M., GU, R., AND CHEN, H. Using concurrent relational logic with helpers for verifying the AtomFS file system. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP 2019)* (Hunstville, ON, Canada, Oct. 2019).