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

OSDI 2020

Travis Hance (CMU)       Andrea Lattuada (ETH)       Chris Hawblitzel (MSR)
Jon Howell (VMR)         Rob Johnson (VMR)        Bryan Parno (CMU)
What is Verification?

• Mathematical proof that a program is **correct**.
• Proof is checked by a computer (the **verifier**).

**Key-value dictionary implementation**

• Complex data structure
• Handle edge cases
• 100s or 1000s of lines of code

```
Key-value dictionary specification

• Stated simply and mathematically

f : Key → Value

Put(k: Key, v: Value):
  f := f[k ↦ v]

Get(k: Key):
  return f(k)
```
Verifying Persistent Disk Storage Systems

**Persistent key-value store implementation**
- Complex data structure
- Handle edge cases
- 100s or 1000s of lines of code
- Handle asynchronous disk access
- IO-efficient data structure
- Caching (eviction policy, etc.)
- Crash safety
- CPU-efficiency

**Persistent key-value store specification**
- Stated simply and mathematically

\[
f : \text{Key} \rightarrow \text{Value}
\]

\[
\text{Put}(k: \text{Key}, v: \text{Value}) : \\
f := f[k \mapsto v]
\]

\[
\text{Get}(k: \text{Key}) : \\
\text{return } f(k)
\]

- Expose a way for user to confirm data has been persisted
- Data persistence on crash
Contributions

• VeriBεtrKV: a complex, verified storage system
  • Crash-safe key-value store based on the Bε-tree, an established, state-of-the-art, IO-efficient, write-optimized data structure
  • Written in Dafny (compiled via C++)

• General methodology for verifying asynchronous systems

• Linear types combined with Dafny’s dynamic frames to improve the experience of verifying efficient, imperative code
Modeling Disk Systems

• We need a clean & flexible way to encode environmental assumptions.
  • How does the disk work?
  • Assumptions about asynchronicity?
  • What failure scenarios are considered?

• Observation: General problem across asynchronous systems
  • **IronFleet** (2015) uses **state machines** to model networked distributed systems.
  • We generalize and apply to storage systems.
  • No need for a domain-specific logic!
Modeling Asynchronous Systems
Modeling Asynchronous Systems

• Templated state machine $\text{NetworkSystem<Host>}$ is defined in terms of $\text{Host}$ state machine.

• This state machine definition encodes all environmental assumptions!
  • Packet delivery
  • Packet reordering
  • Packet duplication

• We demonstrate that we can use this approach for other asynchronous systems, like our disk system.
Modeling disk systems

DiskSystem<Host>
Modeling disk systems

DiskSystem<Host>

Host step

Host

Host
Modeling disk systems

DiskSystem<Host>

Disk step

Read command

Host

Block of data

Host
Modeling disk systems

DiskSystem<Host>

Crash & reboot step

Initial Host state
NetworkSystem<Host>

- Network delivering packets
- Packet reordering
- Packet duplication

DiskSystem<Host>

- Disk
- IO queue
- Command reordering
- Host failure
- Host reinitialization
- (Limited) spontaneous data corruption
Modeling Disk Systems

• Method: encode any environmental assumptions in the definition of templated state machine `System<Host>
• Natural extension of IronFleet’s method
• Clean split between environmental assumptions (System) and implementation details (Host)
• Environmental assumptions easy to read and understand
Verifying Persistent Disk Storage Systems

**Persistent key-value store implementation**

- Complex data structure
- Handle edge cases
- 1000s of lines of code
- Handle asynchronous disk access
- IO-efficient data structure
- Caching (eviction policy, etc.)
- Crash safety
- CPU-efficiency

**Persistent key-value store specification**

- Stated simply and mathematically

\[
f : \text{Key} \rightarrow \text{Value}
\]

Put(k: Key, v: Value):
\[
f := f[k \mapsto v]
\]

Get(k: Key):
\[
\text{return } f(k)
\]

- Expose a way for user to confirm data has been persisted
- Data persistence on crash
Application Spec

System state machine

Host model state machine

{ a: 1, b : 2 }

{ a: 1, b : 3 }

State machine refinement
Application Spec

\{ a: 1, b : 2 \} \rightarrow \{ a: 1, b : 3 \}

System state machine

Host model state machine
- Bε-tree operations
- Caching logic
- Journal logic

Implementation code

```
method insert(key: Key, value: Value)
{
  // actual runnable code here
}
```
Writing Efficient, Verified Code

- **Goal:** efficient, runnable code that implements this state machine.
  - Imperative code with mutable update-in-place data structures

**Host model state machine**
- $B^\varepsilon$-tree operations
- Caching logic
- Journal logic

**Implementation code**

```csharp
method insert(key: Key, value: Value) {
    // actual runnable code here
}
```

**Floyd-Hoare logic**
Memory Aliasing

• Dafny uses a memory-reasoning strategy called **dynamic frames**.
  • This strategy requires explicit aliasing information.

```dalg
class Point {
  var x: int;
  var y: int;
}

method foo(a: Point, b: Point)
modifies a, b
requires a != b
{
  a.x := 1;
  b.x := b.x - 1;
  assert a.x == 1;
}

method main()
{
  var a := new Point();
  foo(a, a);
}
```
Memory Aliasing

- Manually adding aliasing conditions is cumbersome.
  - Number of pairwise conditions grows quadratically.
  - Handling deep data structures requires reasoning about sets of objects.

```plaintext
static predicate {opaque} ReprSeqDisjoint(buckets: seq<MutBucket>)
reads set i | 0 <= i < |buckets| :: buckets[i]
{
  forall i, j | 0 <= i < |buckets| && 0 <= j < |buckets| && i != j ::
    buckets[i].Repr !! buckets[j].Repr
}
```

```plaintext
twostate lemma SplitChildOfIndexPreservesWFShape(node: Node, childidx: int)
// ...
requires unchanged(old(node.repr) - {node, node.contents.pivots, node.contents.children, node.contents.children[childidx]}))
// ...
requires node.contents.children[childidx].repr <= old(node.contents.children[childidx].repr)
// ...
requires fresh(node.contents.children[childidx+1].repr - old(node.contents.children[childidx].repr))
requires node.contents.children[childidx+1].height == old(node.contents.children[childidx].height)
requires DisjointSubtrees(node.contents, childidx, (childidx + 1))
requires node.repr == old(node.repr) + node.contents.children[childidx+1].repr
ensures WFShape(node)
```
Memory Aliasing

• We could just write immutable code instead ...

```plaintext
datatype Point(x: int, y: int)

method foo(
    a: Point,
    b: Point)
returns (a': Point, b': Point)
{
    a' := a.(x := 1);
    b' := b.(x := b.x - 1);
    assert a'.x == 1;
}
```

• This makes verification much easier.
• But copying objects is slower, especially large sequences.
Faster Code with Linear Types

• What if we could:
  • Verify objects as if they were immutable,
  • But have the compiler generate code with in-place updates?

• Use a linear type system to enforce exclusive ownership of objects.
Faster Code with Linear Types

```plaintext
datatype Point(x: int, y: int)

method foo(
    linear a: Point,
    linear b: Point)
returns (linear a': Point,
         linear b': Point)
{
    a' := a.(x := 1);
    b' := b.(x := b.x - 1);
    assert a'.x == 1;
}

method main()
{
    linear var a := Point(0, 0);
    foo(a, a);
}
```
Adding Linear Types to Dafny

• Aliasing errors are now immediate type errors.
• Inspired by prior verification work, Cogent (2016)
• Production languages like Rust also demonstrate that linear semantics are feasible for a lot of systems code.
• When linearity is too constraining, we can still fall back to dynamic frames and theorem-proving.
  • Enables code not expressible in a strict linear type system
  • Used in key places in VeriBétrKV
VeriBεtrKV Implementation

• Code is compiled via a C++ backend for Dafny

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines of code</th>
<th>Total</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Environment model</td>
<td>450</td>
<td>730</td>
<td>Trusted</td>
</tr>
<tr>
<td>Application spec</td>
<td>280</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Executable code</td>
<td>6,500</td>
<td>6,500</td>
<td>Impl</td>
</tr>
<tr>
<td>Host model</td>
<td>2,800</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Refinement Proof</td>
<td>23,000</td>
<td>47,800</td>
<td>Proof</td>
</tr>
<tr>
<td>Floyd-Hoare Proof</td>
<td>22,000</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• Proof : code ratio is ~ 7, comparable to IronFleet.
• System is ~ 3x as large as IronFleet.
Development Process

• Linear types improve both **proof length** and **verification times**.

<table>
<thead>
<tr>
<th>Component</th>
<th>LoC (dynamic frames)</th>
<th>LoC (linear)</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>In-memory hash table</td>
<td>1967</td>
<td>1352</td>
<td>31%</td>
</tr>
<tr>
<td>In-memory search tree</td>
<td>2509</td>
<td>1904</td>
<td>24%</td>
</tr>
</tbody>
</table>

• Maximum method-level interactive verification time dropped 42s → 32s
• 99th percentile dropped 6.1s → 4.8s
• Linear type errors are instant!
Performance Benchmarks

10 million insertion operations, 2GiB RAM, single-threaded
Performance Benchmarks

- VeriBεtrKV’s B^ε-trees beats B-trees on inserts, as expected.
- VeriBεtrKV is still behind RocksDB, one of the fastest, highly-tuned unverified key-value stores.
- VeriBεtrKV lags both BerkeleyDB and RocksDB on queries
  - Memory fragmentation results in smaller effective cache size
  - Missing optimizations needed to match query performance of B-trees
Conclusion

• Defining **System<Host>** state machines is a convenient and flexible way to encode environmental assumptions for system verification.

• Linear type systems are practical for systems code and relieve both developer and verifier burden.

• VeriBεtrKV advances towards performance of state-of-the-art non-verified systems, with much stronger guarantees.

• Thank you
  • thance@andrew.cmu.edu