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

OSDI 2020

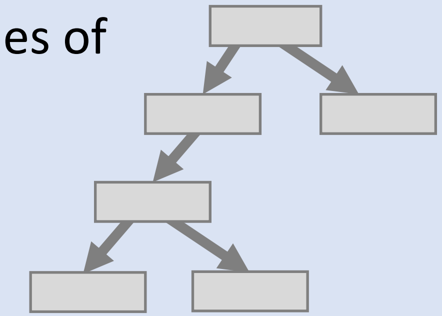
Travis Hance (CMU)
Andrea Lattuada (ETH)
Chris Hawblitzel (MSR)
Jon Howell (VMR)
Rob Johnson (VMR)
Bryan Parno (CMU)

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

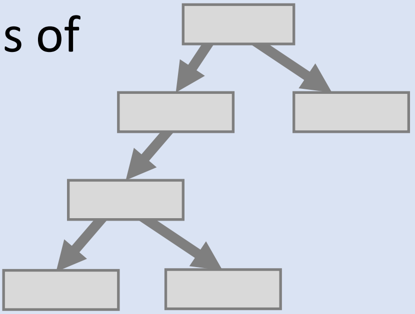
- Stated simply and mathematically

```
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 : Key ⟶ Value

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

Get(k: Key):
  return f(k)
```

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

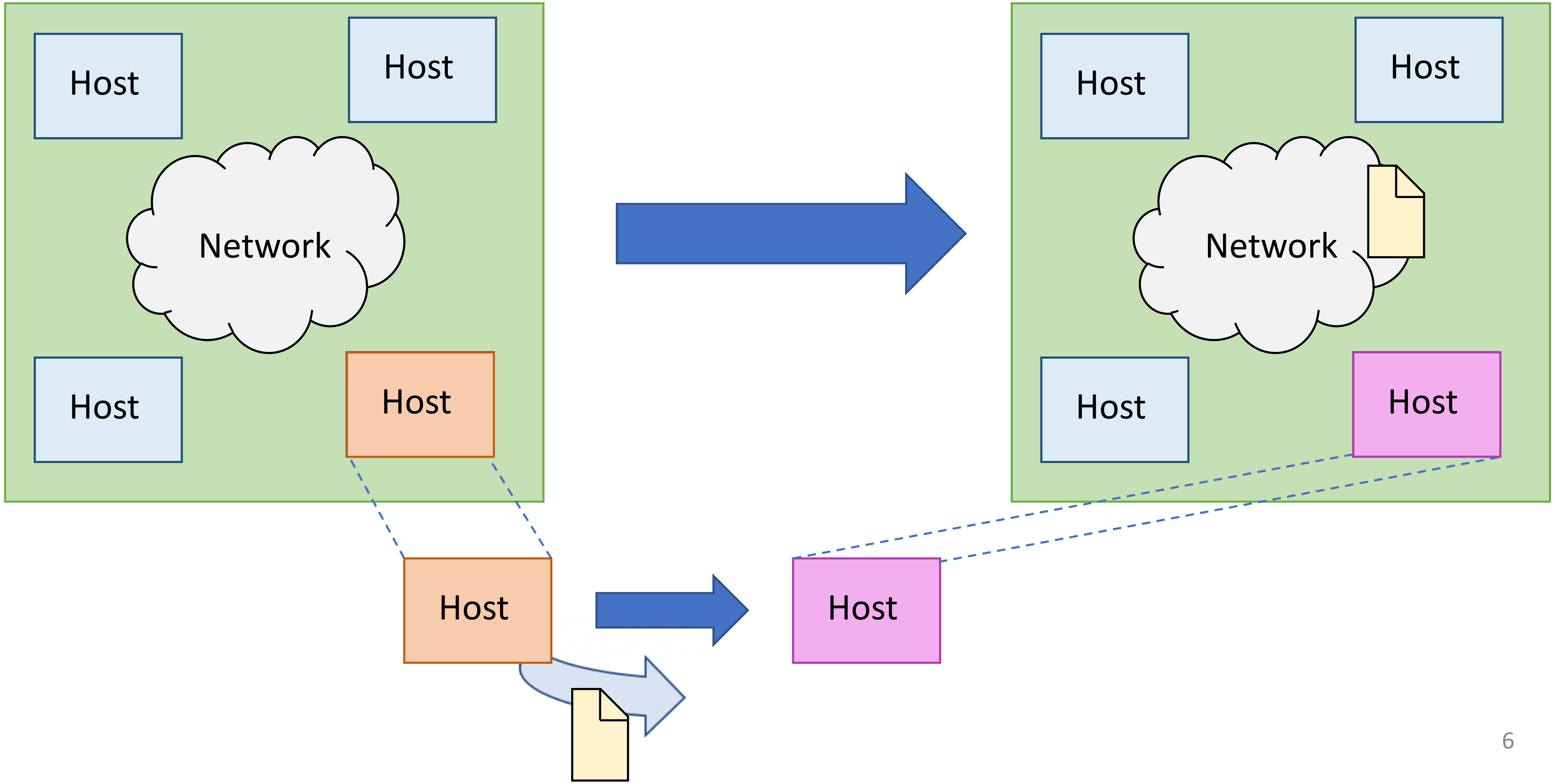
Contributions

- VeriBεtrKV: a complex, verified storage system
  - Crash-safe key-value store based on the **B^{ε}-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



Network system state machine

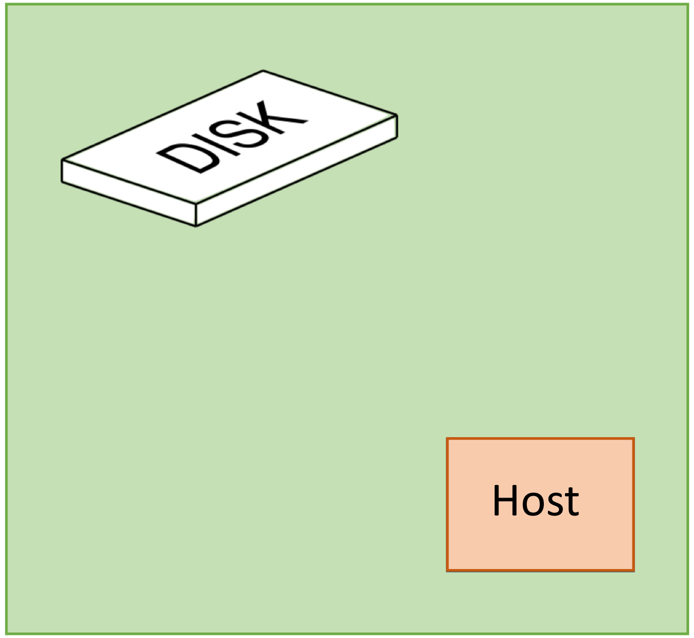
Host state machine

Modeling Asynchronous Systems

- Templated state machine **NetworkSystem<Host>** is defined in terms of **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

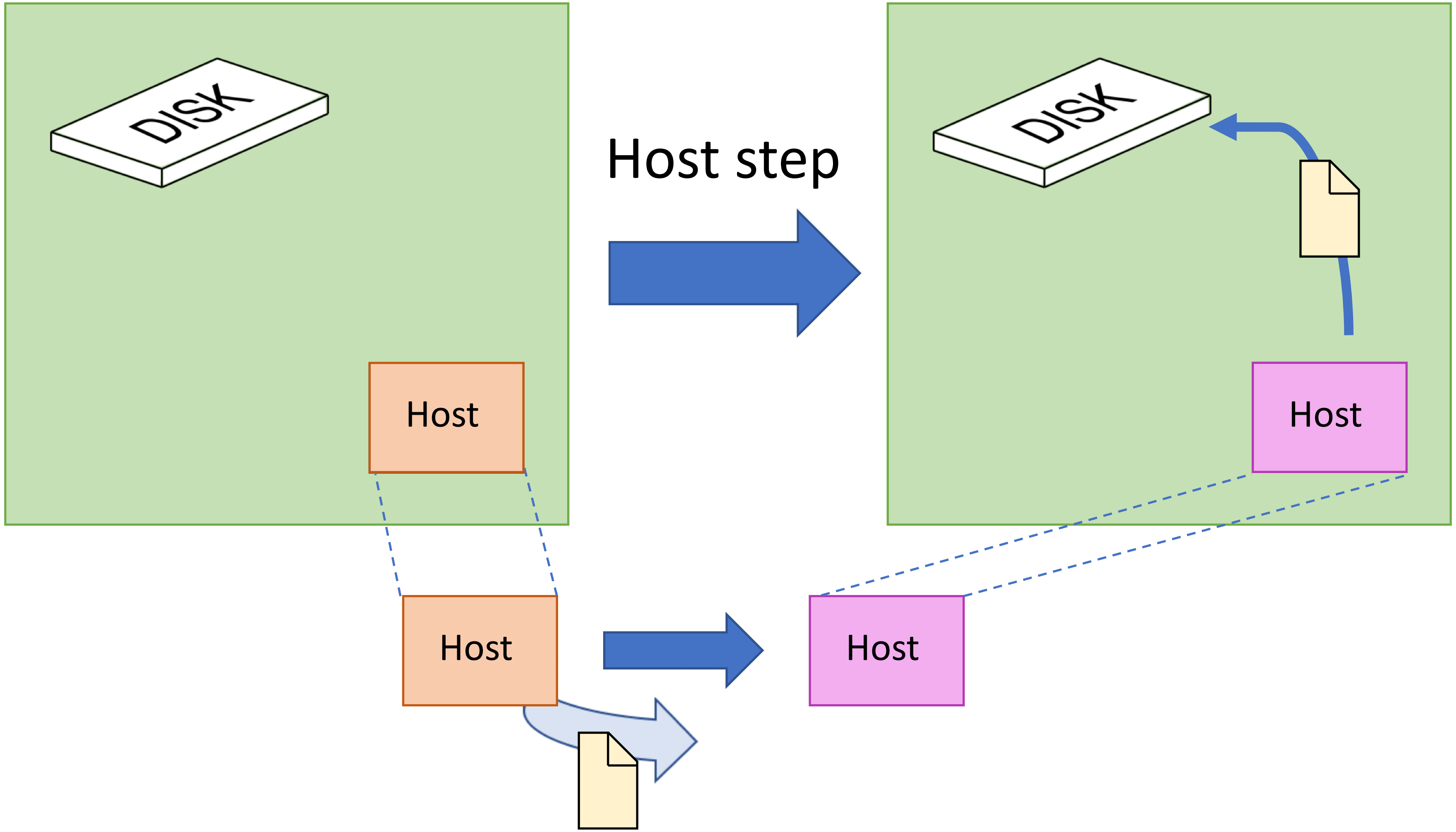
DiskSystem<Host>



Modeling disk systems

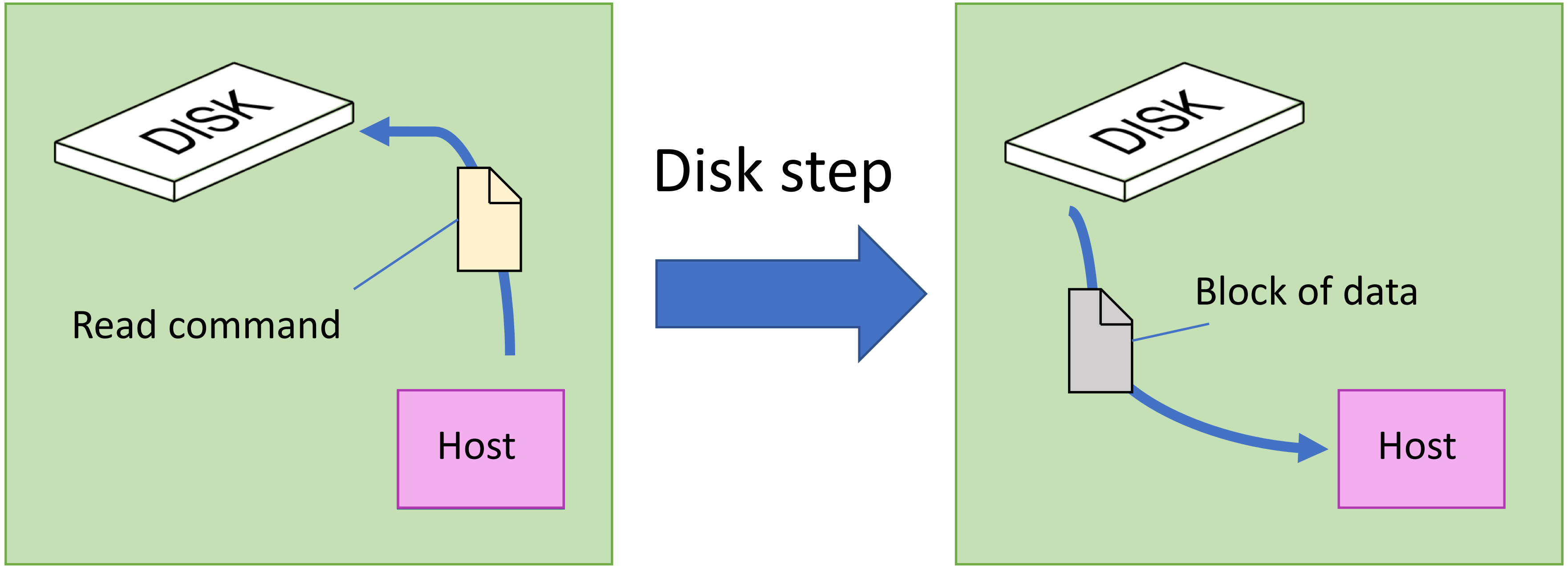
DiskSystem<Host>

Host

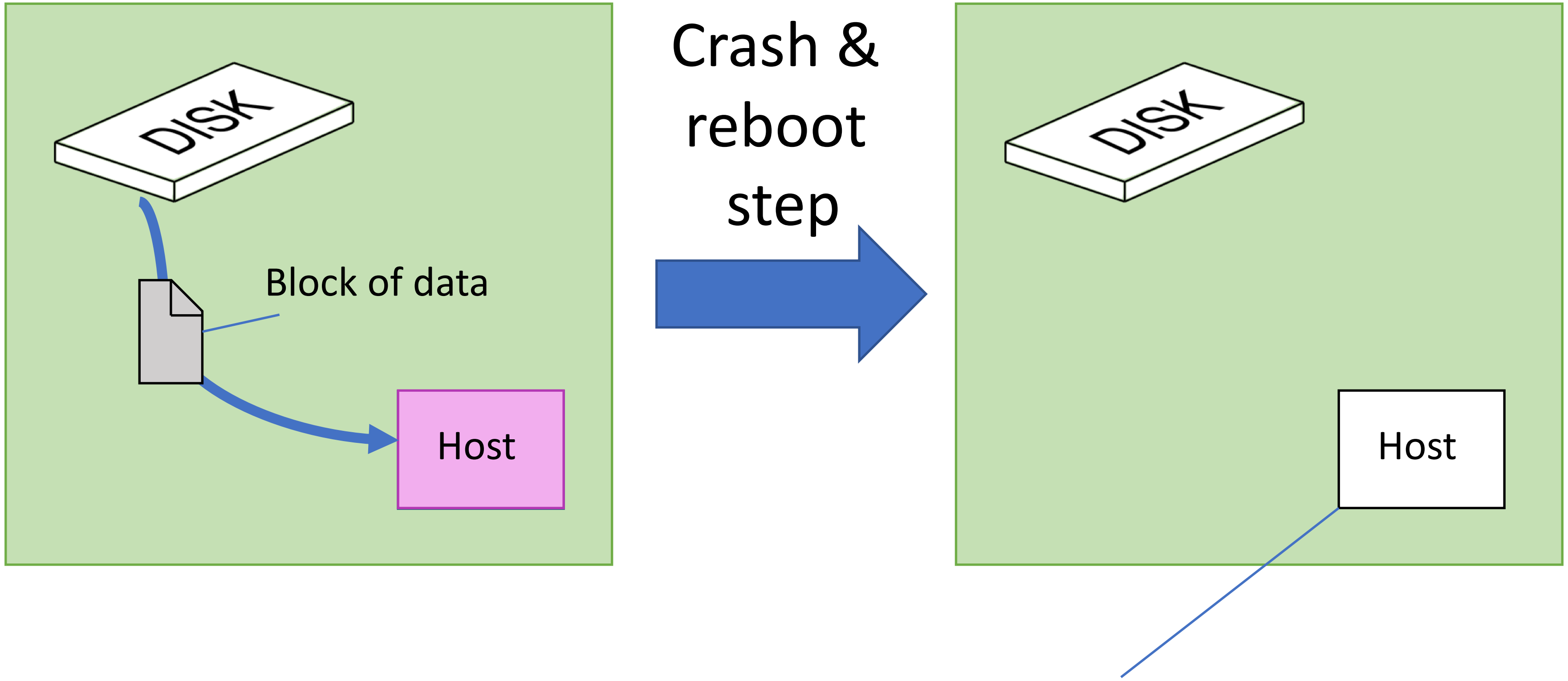


Modeling disk systems

DiskSystem<Host>



Modeling disk systems

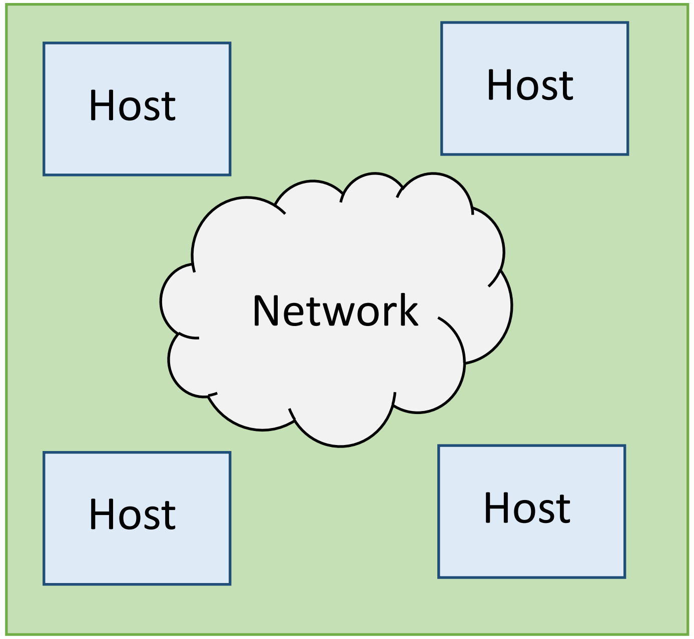


DiskSystem<Host>

Crash & reboot step

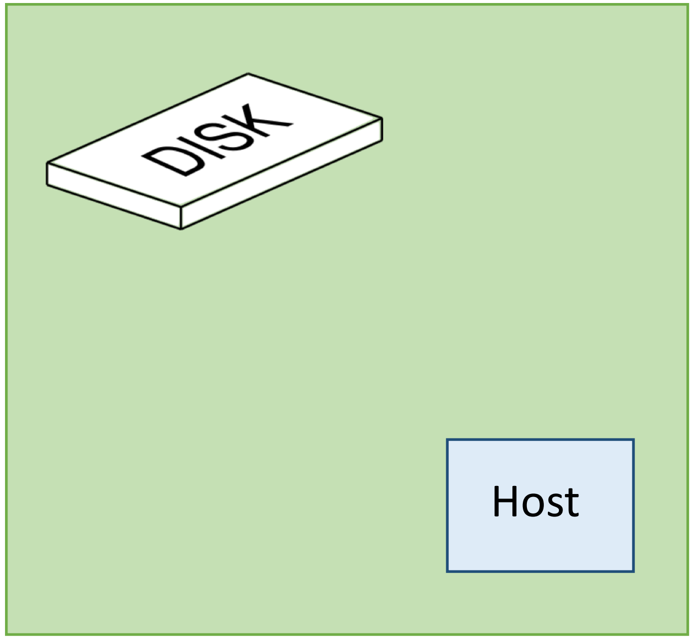
Initial **Host** state

NetworkSystem<Host>



- Network delivering packets
- Packet reordering
- Packet duplication

DiskSystem<Host>



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

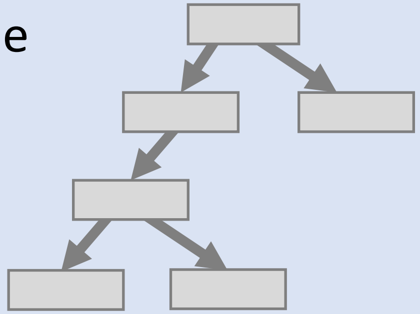
Modeling Disk Systems

- 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

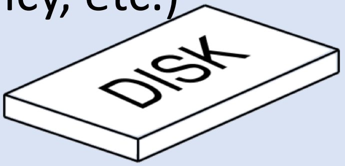
Verifying Persistent Disk Storage Systems

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

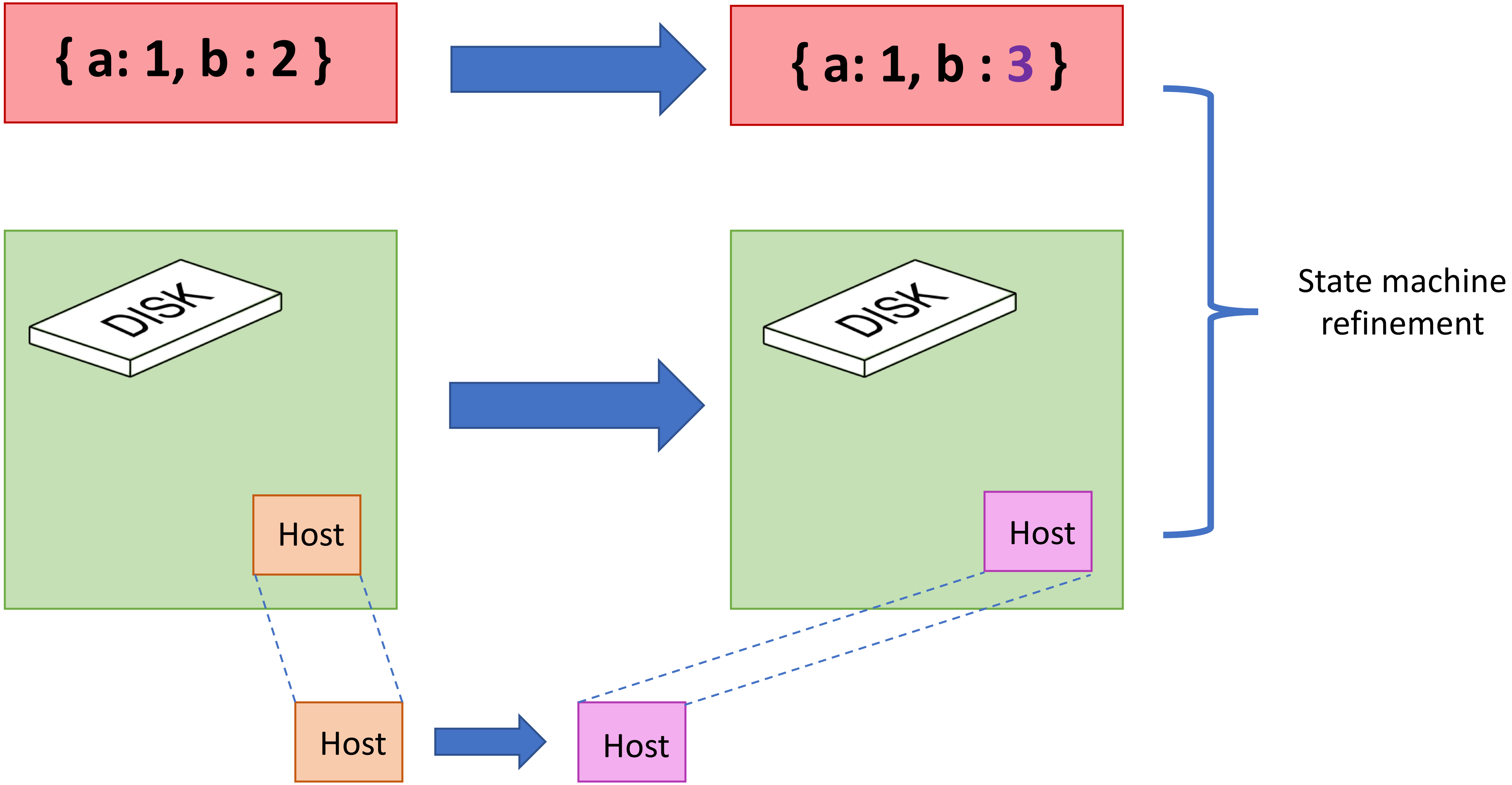


Persistent key-value store specification

- Stated simply and mathematically

```
f : Key ⟶ Value

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

Get(k: Key):
  return f(k)
```

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



Application Spec

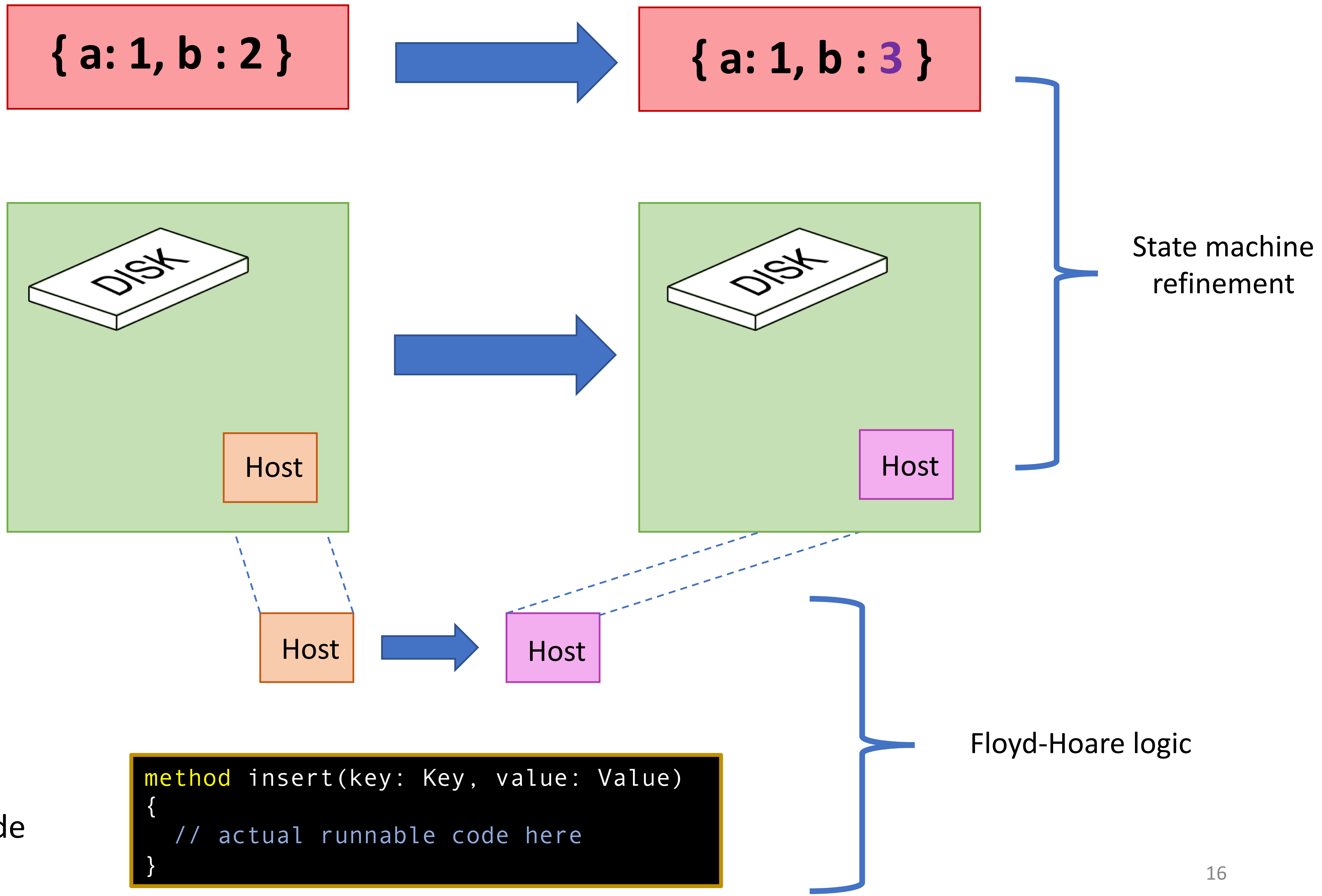
{ a: 1, b : 2 }

{ a: 1, b : 3 }

System state machine

Host model state machine

State machine refinement



Application Spec

System state machine

Host model state machine

- B^{ε}-tree operations
- Caching logic
- Journal logic

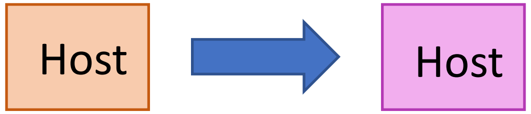
Implementation code

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

Writing Efficient, Verified Code

Host model state machine

- B$^{\epsilon}$-tree operations
- Caching logic
- Journal logic



Implementation code

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

Floyd-Hoare logic

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

Memory Aliasing

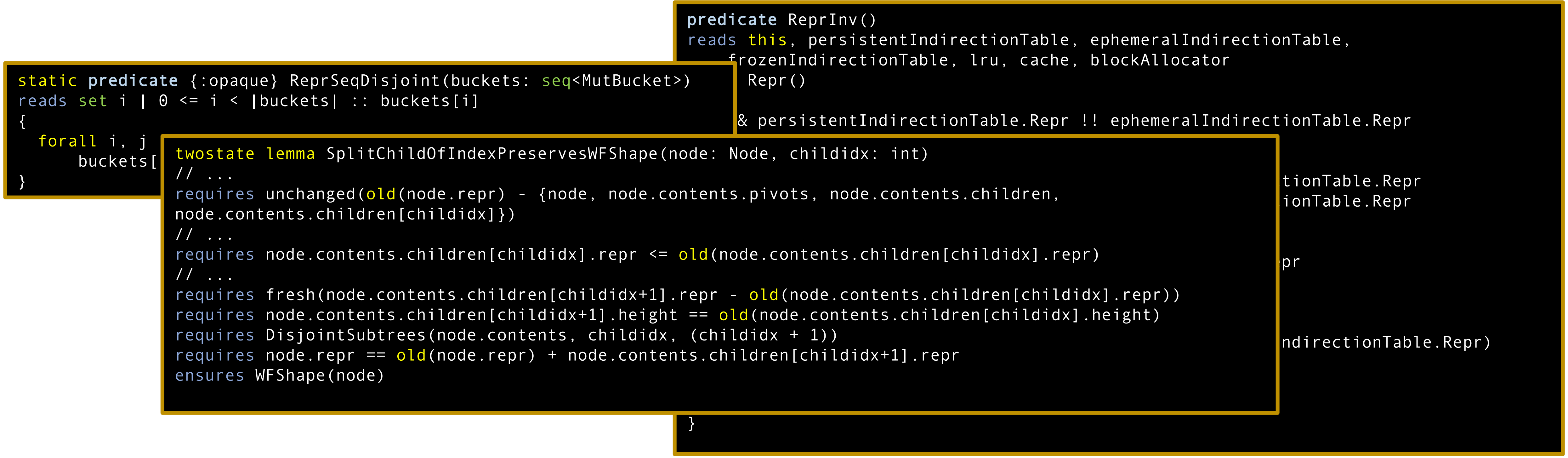
- Dafny uses a memory-reasoning strategy called **dynamic frames**.
    - This strategy requires explicit aliasing information.

```
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.



```
predicate ReprInv()
reads this, persistentIndirectionTable, ephemeralIndirectionTable,
    frozenIndirectionTable, lru, cache, blockAllocator
      Repr()

& persistentIndirectionTable.Repr !! ephemeralIndirectionTable.Repr

tionTable.Repr
ionTable.Repr

pr

ndirectionTable.Repr)

}
```

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

```
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 …

```
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

```
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 VeriBεtrKV

Outline

USTC, CHINA ADSLAB

6 VeriBetrKV

7 Evaluation

8 Conclusion

Outline

USTC, CHINA
ADSLAB

6 VeriBetrKV

7 Evaluation

8 Conclusion

VeriBetrKV(Verified $B^{\varepsilon} - tree\ KV$)

Component

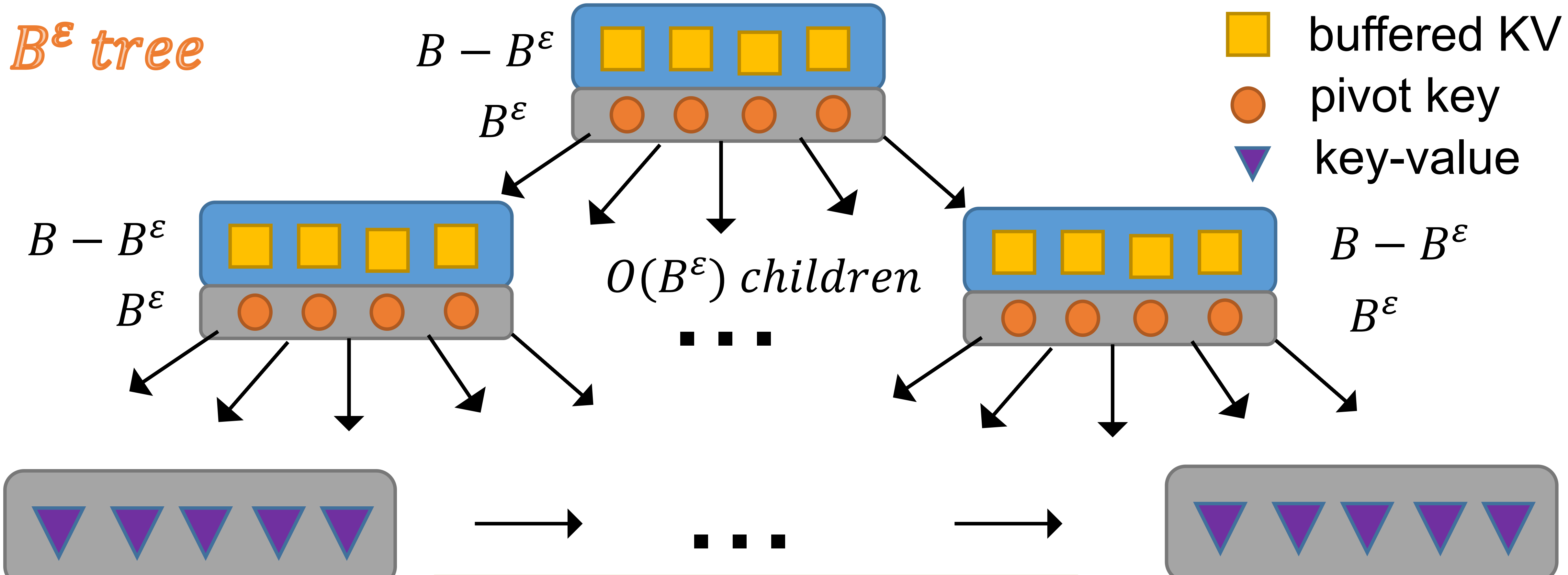
- B^{ε} tree
  - *On disk*
  - *In BlockCache(Memory)*
- Journal
  - *On disk*
  - *In Memory*

functionality

crash safety

VeriBetrKV

B^{ε} tree



- Nodes are larger
- Write optimized(write in buffer)
- query slowed down(larger node)

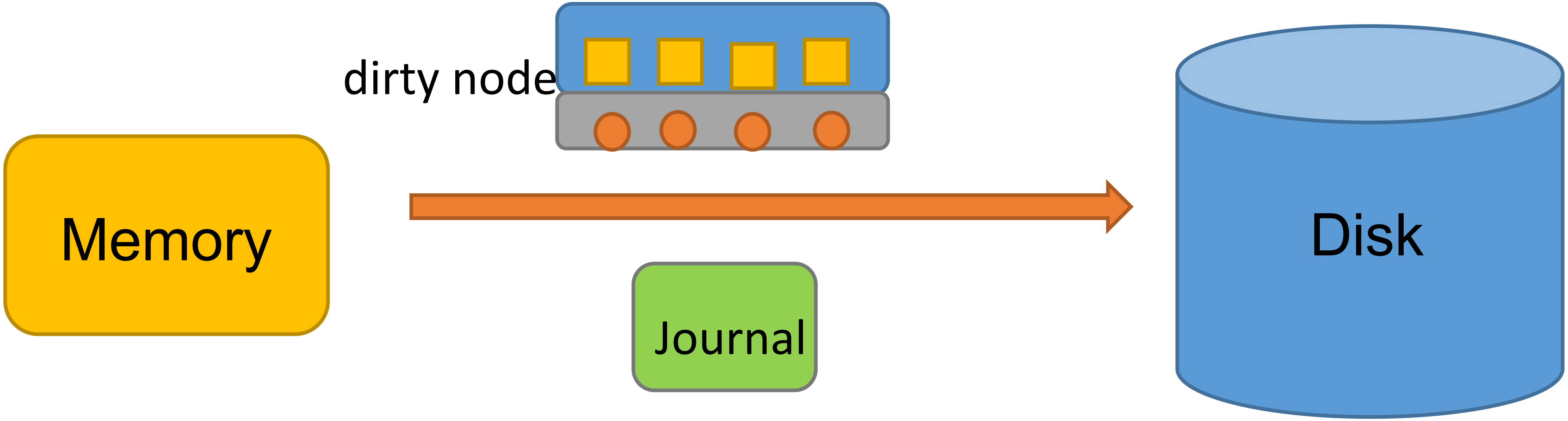
http://supertech.csail.mit.edu/papers/BenderFaJa15.pdf

VeriBetrKV(Verified B^{ε} *tree KV*)

USTC, CHINA ADSLAB

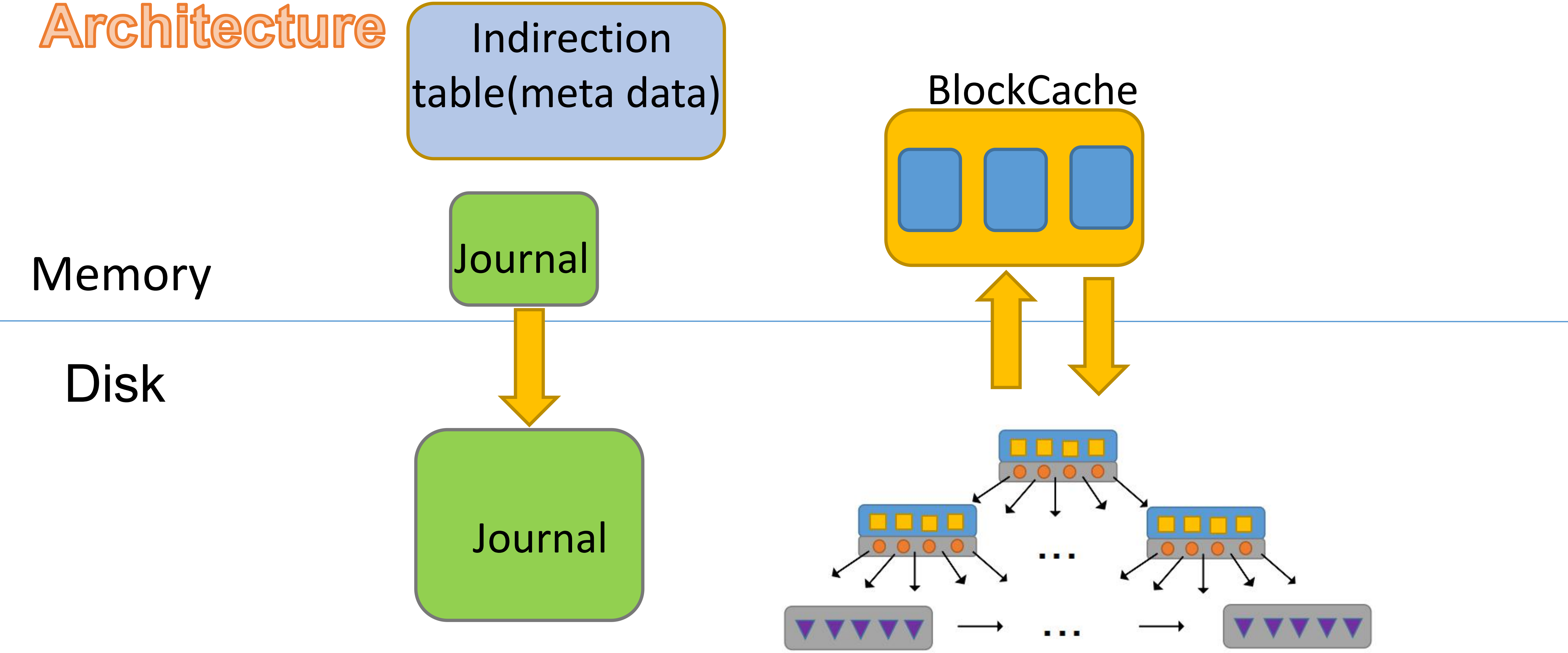
Synchrounization

- Write dirty nodes from BlockCache to disk
- Write journal from memory to disk



VeriBetrKV

Architecture



Proof

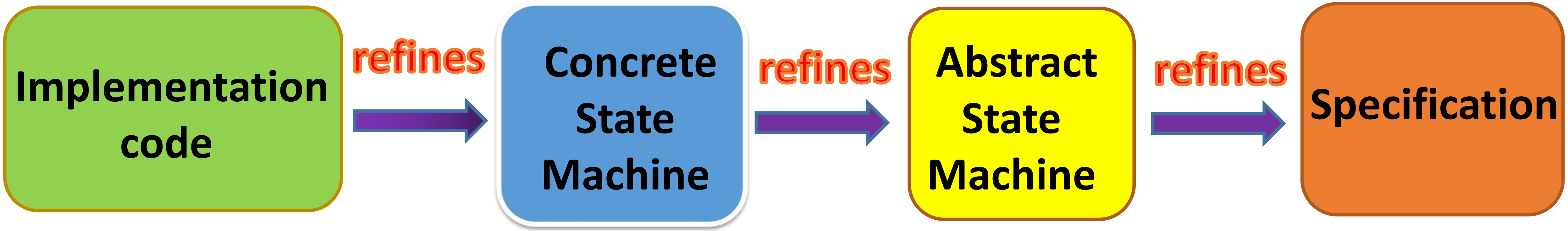
- Refine:
  - Given a concrete state machine T_{conc} and an abstract state machine T_{abs},
  - T_{conc} refines T_{abs}
  - iff every execution of T_{conc} can be mapped to a possible execution of T_{abs}
- Refinement adds detail

Proof

- Refinement for nested model:
  - if A<T> refines B<T> and a refines b,
  - then A<a> refines B<b>

Proof

- The authors build several levels of state machines to describe the asynchronous environment.
- They used modular Hoare logic to prove each step.



Implementation code

refines

Concrete State Machine

refines

Abstract State Machine

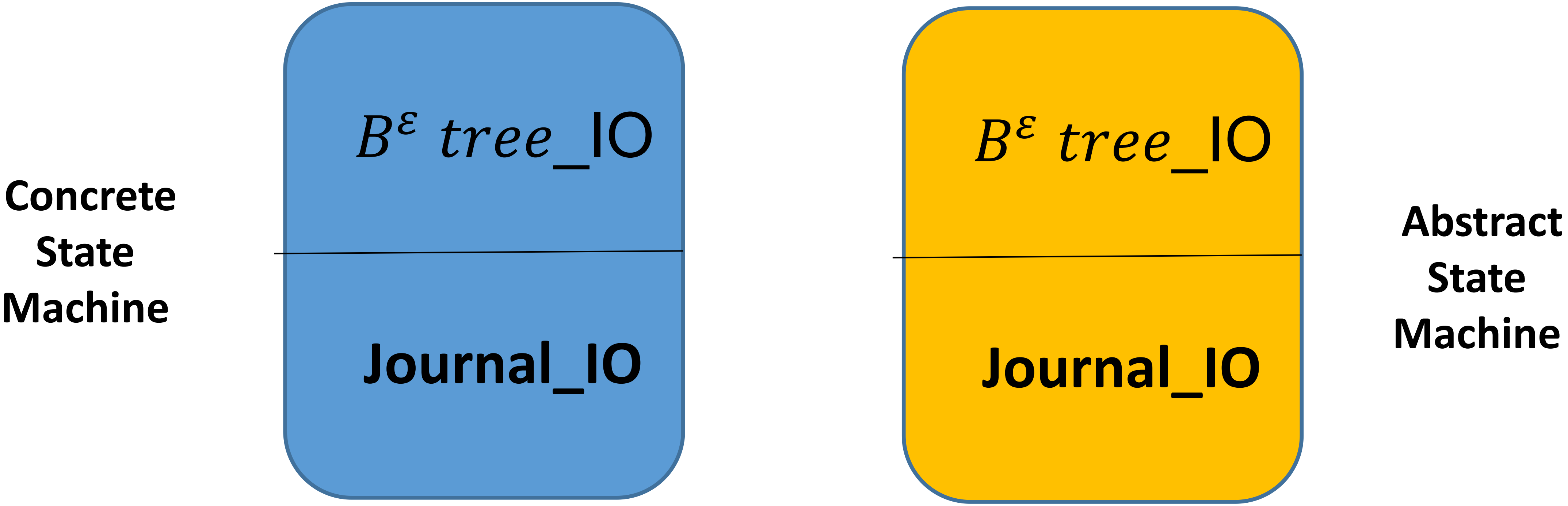
refines

Specification

Modularization

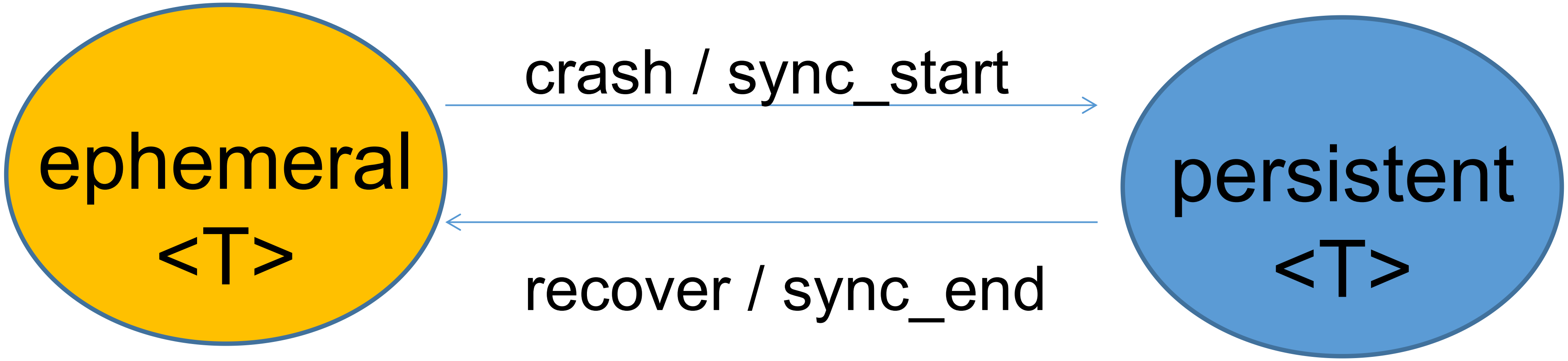
- Seperate the reasoning about
  - $B^{\varepsilon}\ tree$ subsystem
  - Journal subsystem

(Assumption: the journal and $B^{\varepsilon}\ tree$ are not in the same block)



Proof: Figure out Spec first

- We use a state machine to describe how data is recovered from crash.
- We call it CrashSafe<T>, where T is a nested state machine that satisfies the functionality of K-V storage system(with no crash)



ephemeral <T>

crash / sync_start

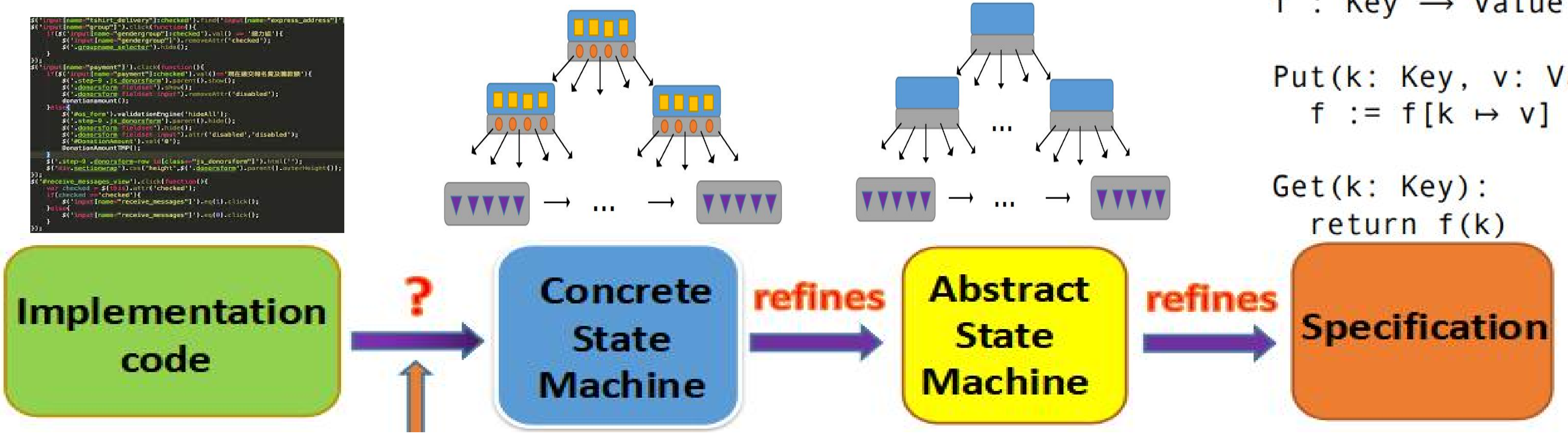
recover / sync_end

persistent <T>

Proof : model B$^{\varepsilon}$ tree and find out T

prove CrashSafe<Map>!!!

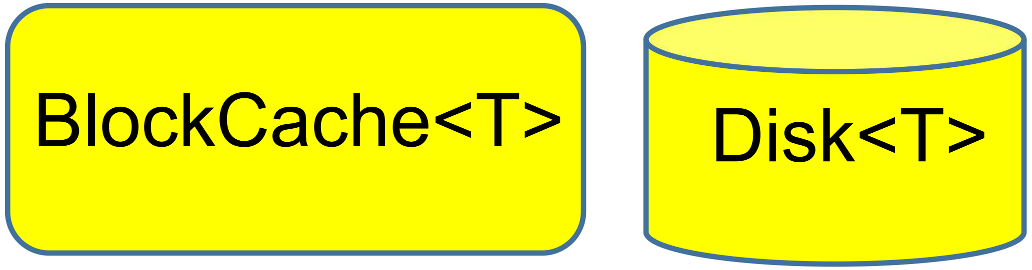
- For an in-memory B^{ε} $tree$ with no crash, its spec is a Map
- Tree structure + abstract node(infinite map) ⇒ abstract B^{ε} $tree$
- Defining node data structure(e.g. finite bucket) ⇒ B^{ε} $tree$

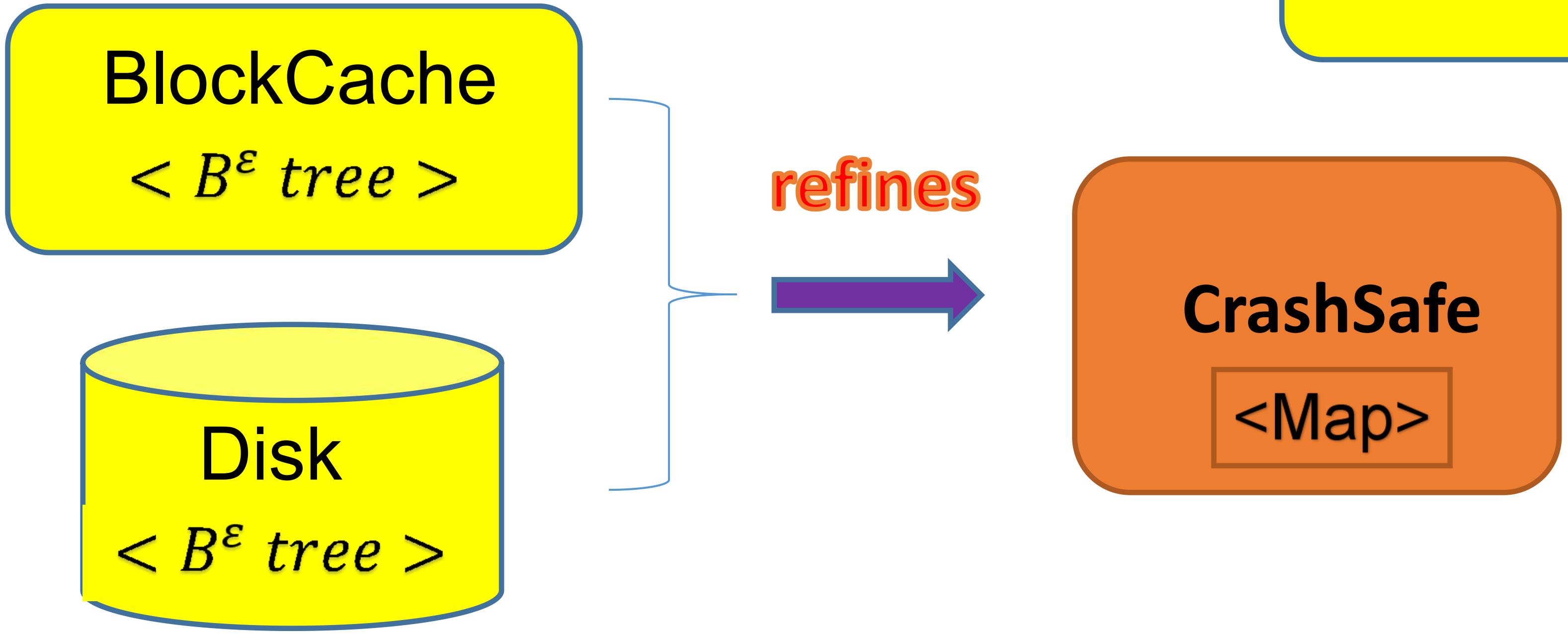


```
f : Key ⟶ Value

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

Get(k: Key):
  return f(k)
```

Proof: B^{ε} tree_IO

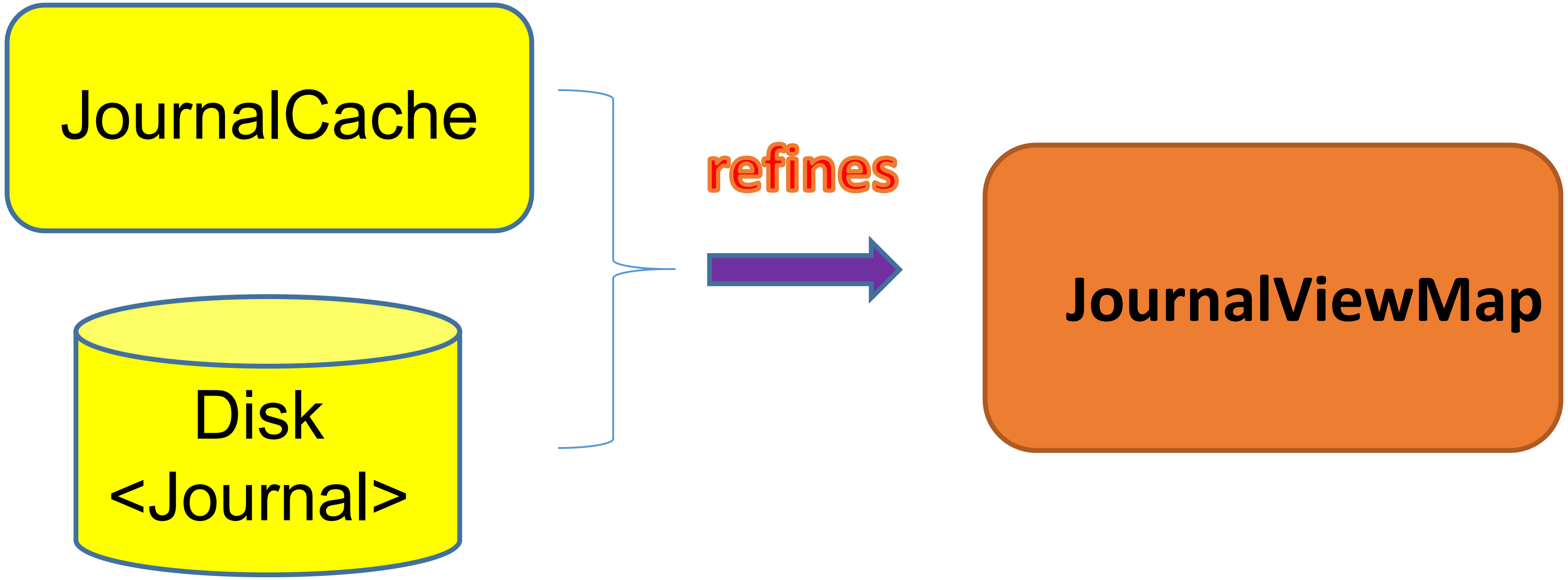
- We define the state machine of BlockCache<T> and Disk<T> to describe their action, where T is the state machine of data structure they stores.





(Note that B^{ε} *tree* refines Map)

Proof: Journal_IO(similar)



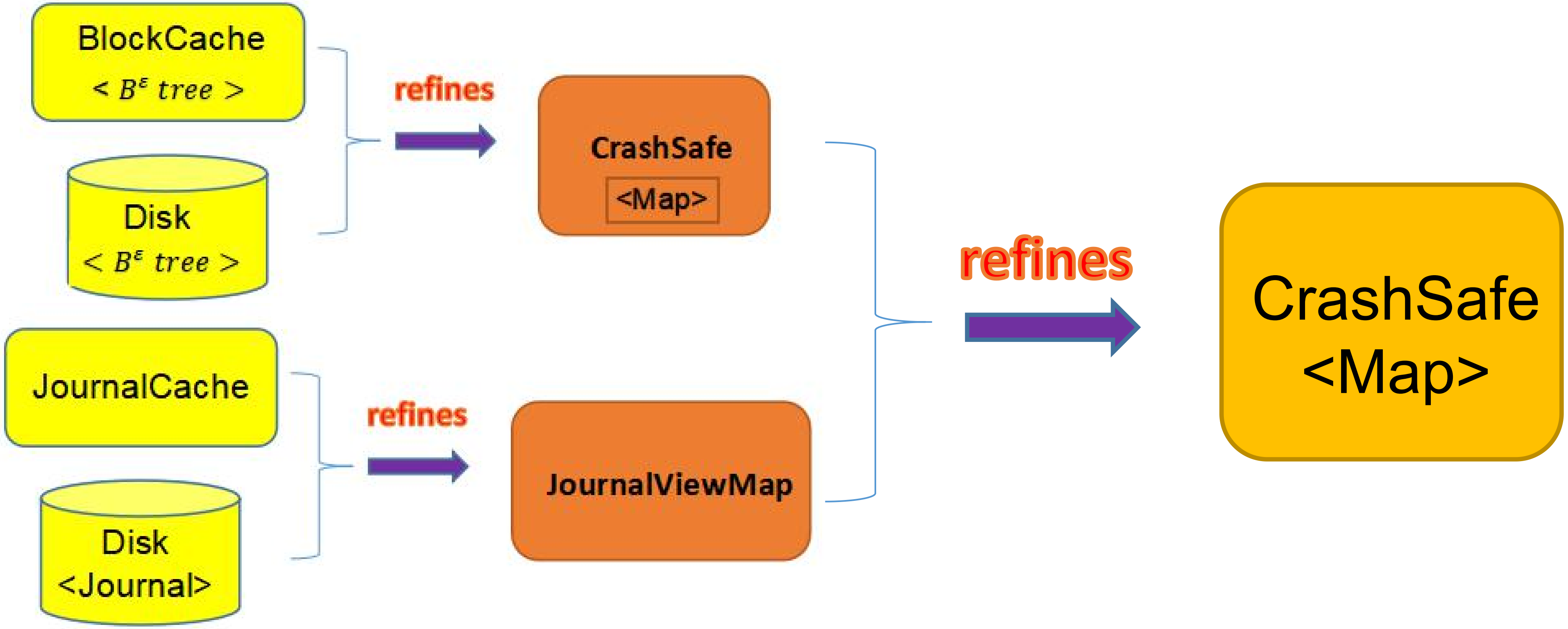
JournalCache

Disk <Journal>

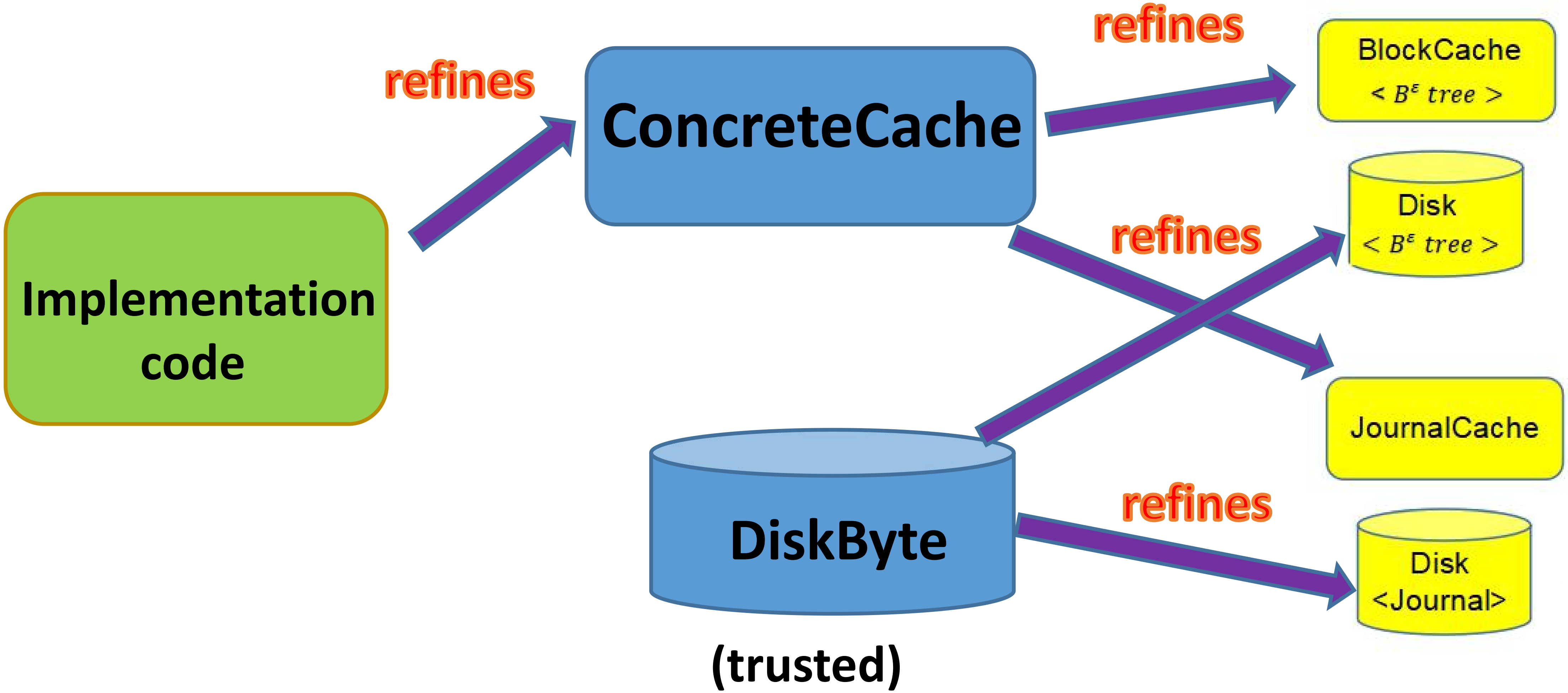
refines

JournalViewMap

Proof: So far, we have refinement



Proof: Let's refine Cache and Disk respectively



Implementation code

refines

ConcreteCache

refines

BlockCache $< B^\varepsilon \text{ tree} >$

refines

Disk $< B^\varepsilon \text{ tree} >$

JournalCache

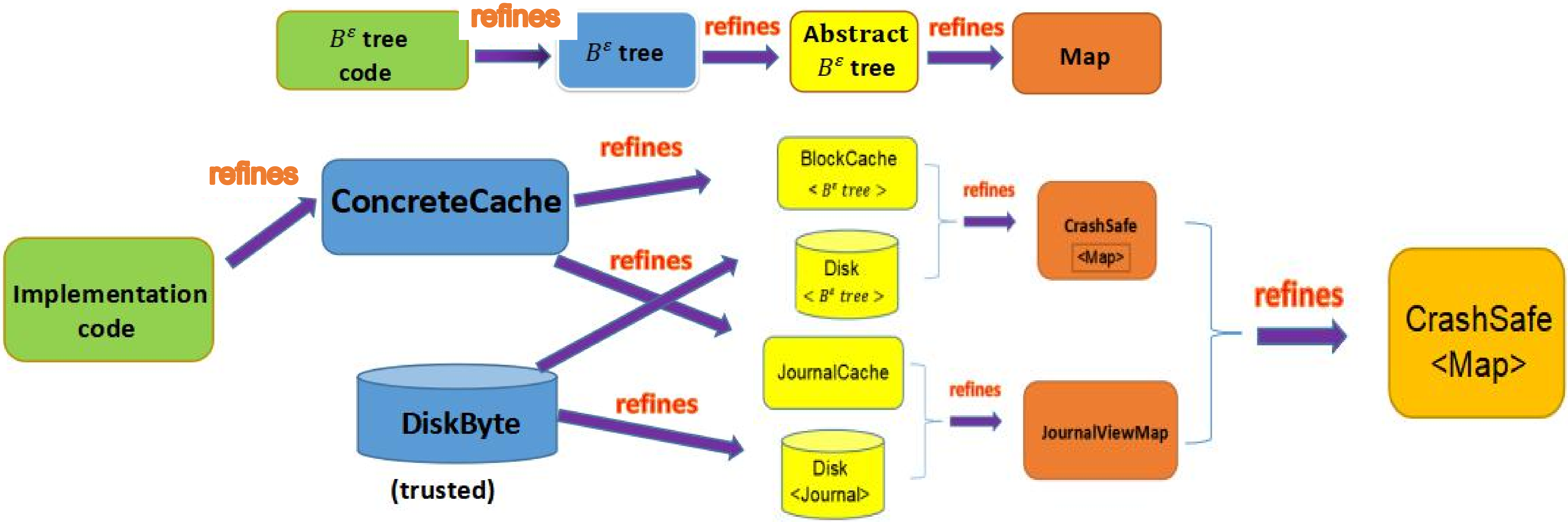
DiskByte

(trusted)

refines

Disk <Journal>

Proof: Overall structure



(trusted)

Outline

6 VeriBetrKV

7 Evaluation

8 Conclusion

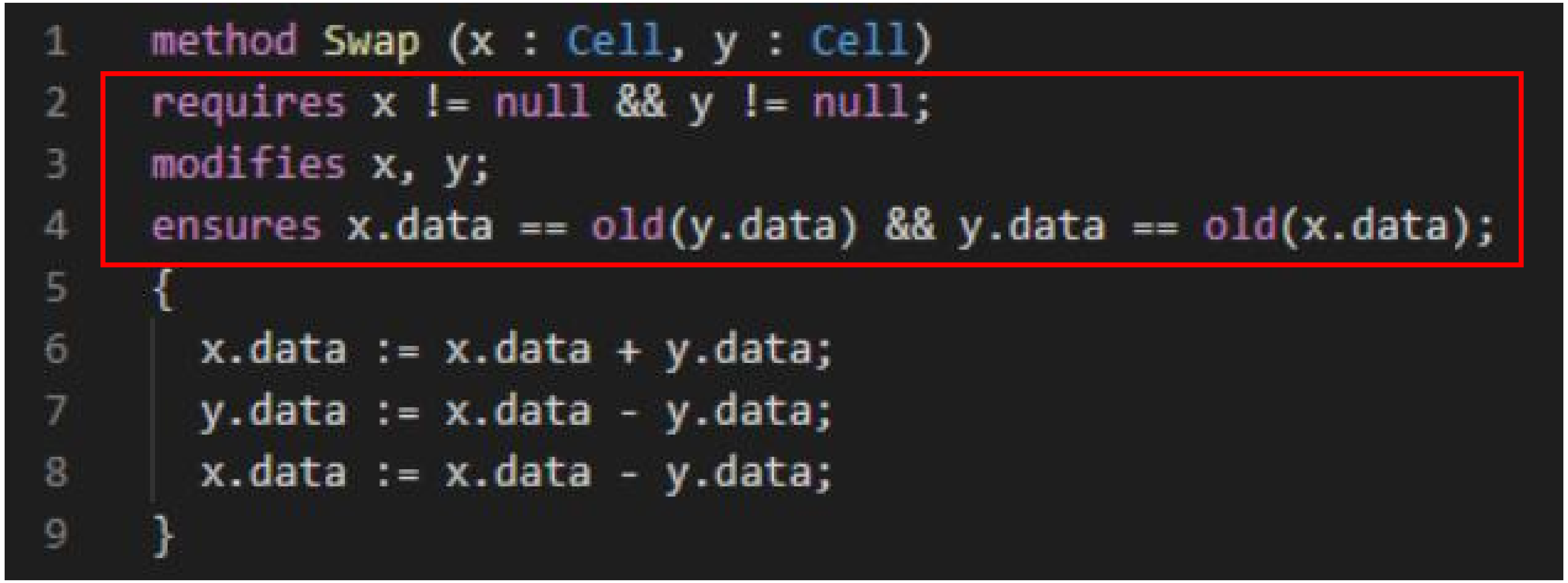
Evaluation

We focus on 2 points

- Does the automation tool improve developer experience?
- btw, can we deliver the performance gains of write optimization?

Evaluation

Developer experience: $$\frac{line\ of\ proof}{line\ of\ impl\ code}$$

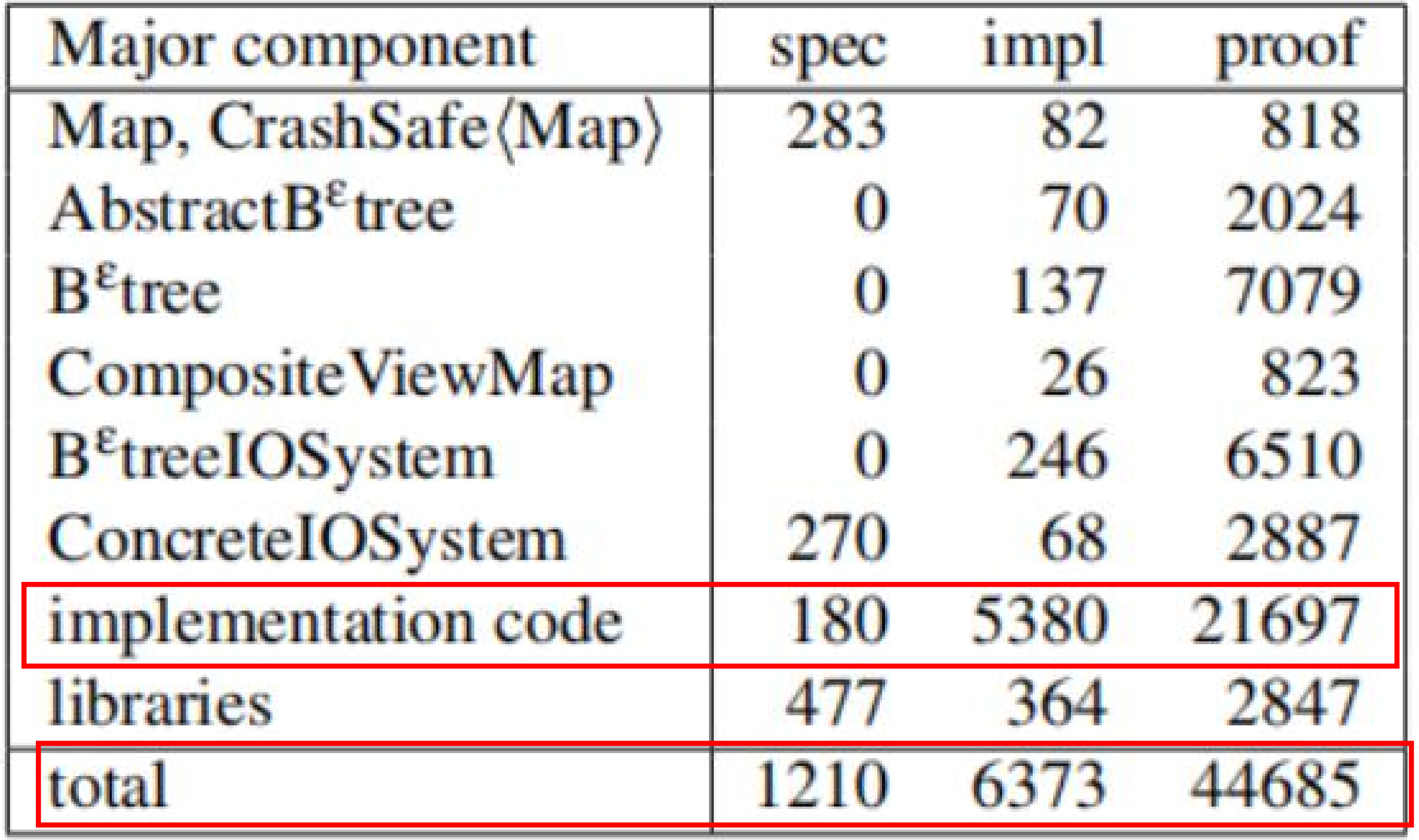


```
method Swap (x : Cell, y : Cell)
requires x != null && y != null;
modifies x, y;
ensures x.data == old(y.data) && y.data == old(x.data);
{
  x.data := x.data + y.data;
  y.data := x.data - y.data;
  x.data := x.data - y.data;
}
```

Evaluation

Developer experience:

$$\frac{line\ of\ proof}{line\ of\ impl\ code}$$



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

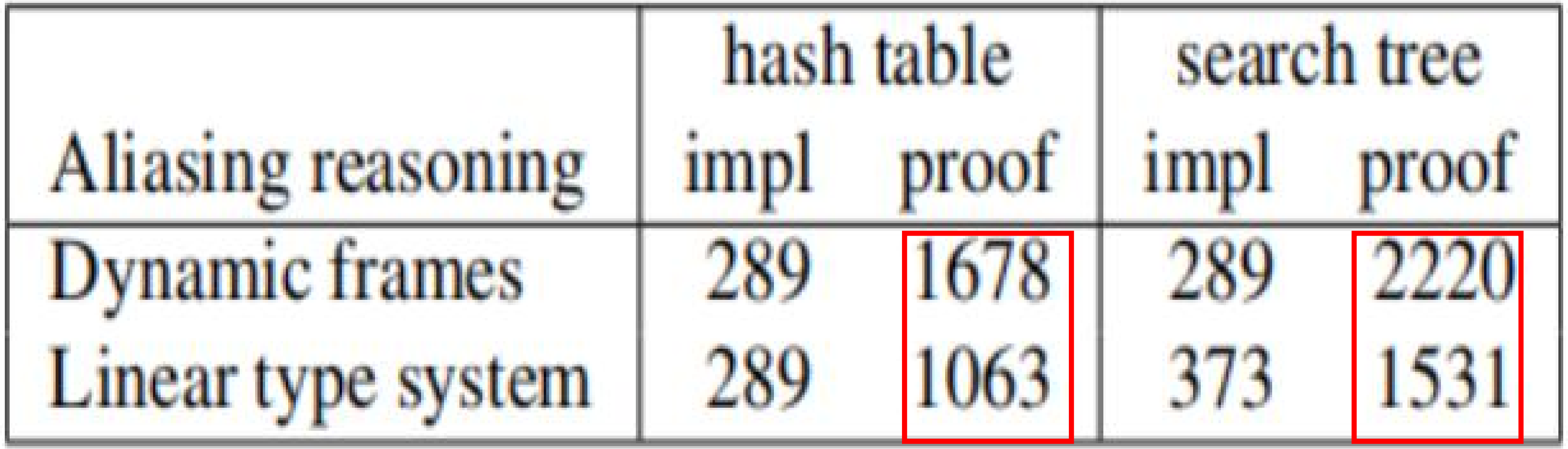
4:1

7:1

Compared to IronFleet, it can scale to a larger system

Evaluation

Dynamic frames vs Linear type system



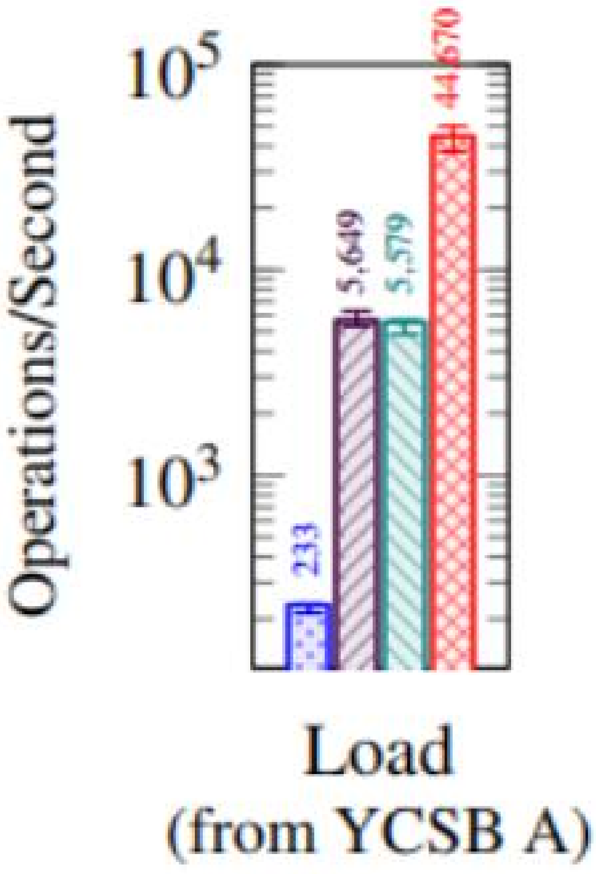
	hash table		search tree	
Aliasing reasoning	impl	proof	impl	proof
Dynamic frames	289	1678	289	2220
Linear type system	289	1063	373	1531

Linear typing reducs the proof burden by 31–37%

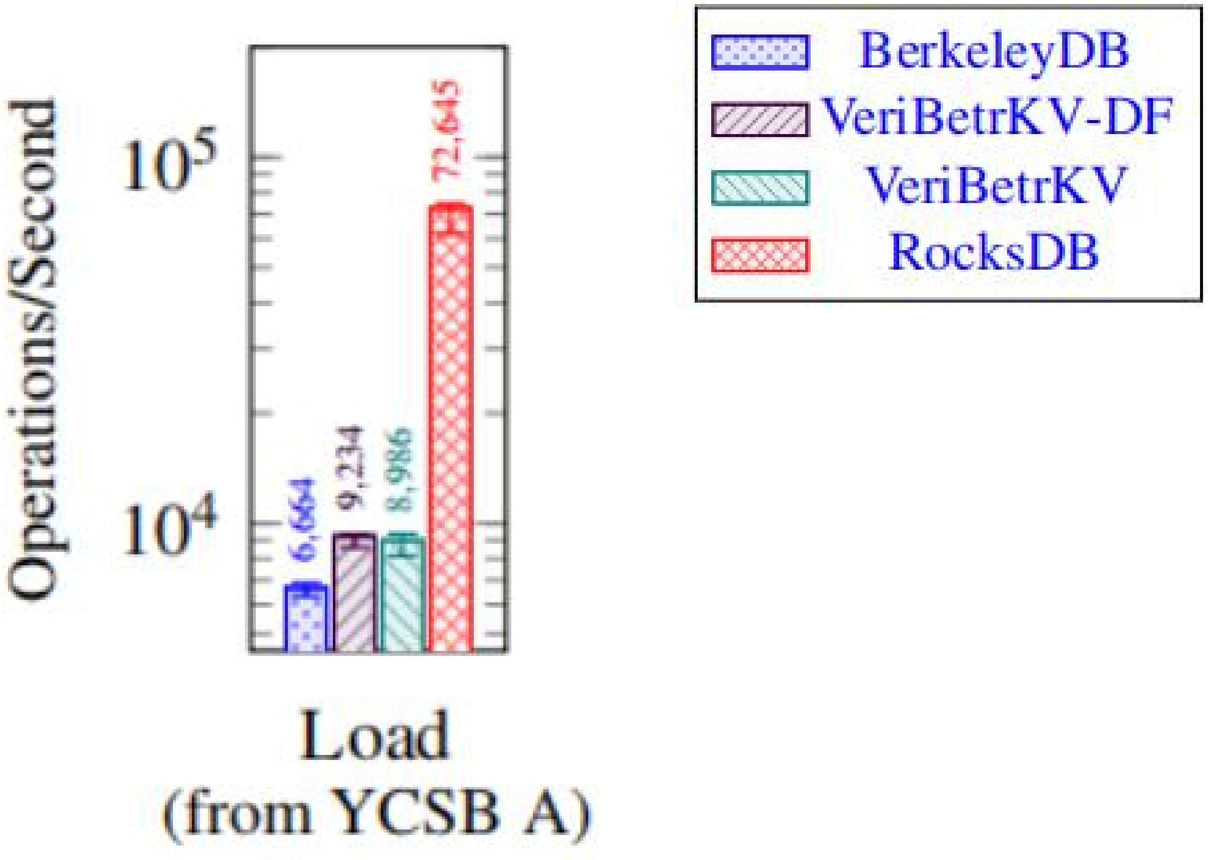
Evaluation

performance: random write

HDD

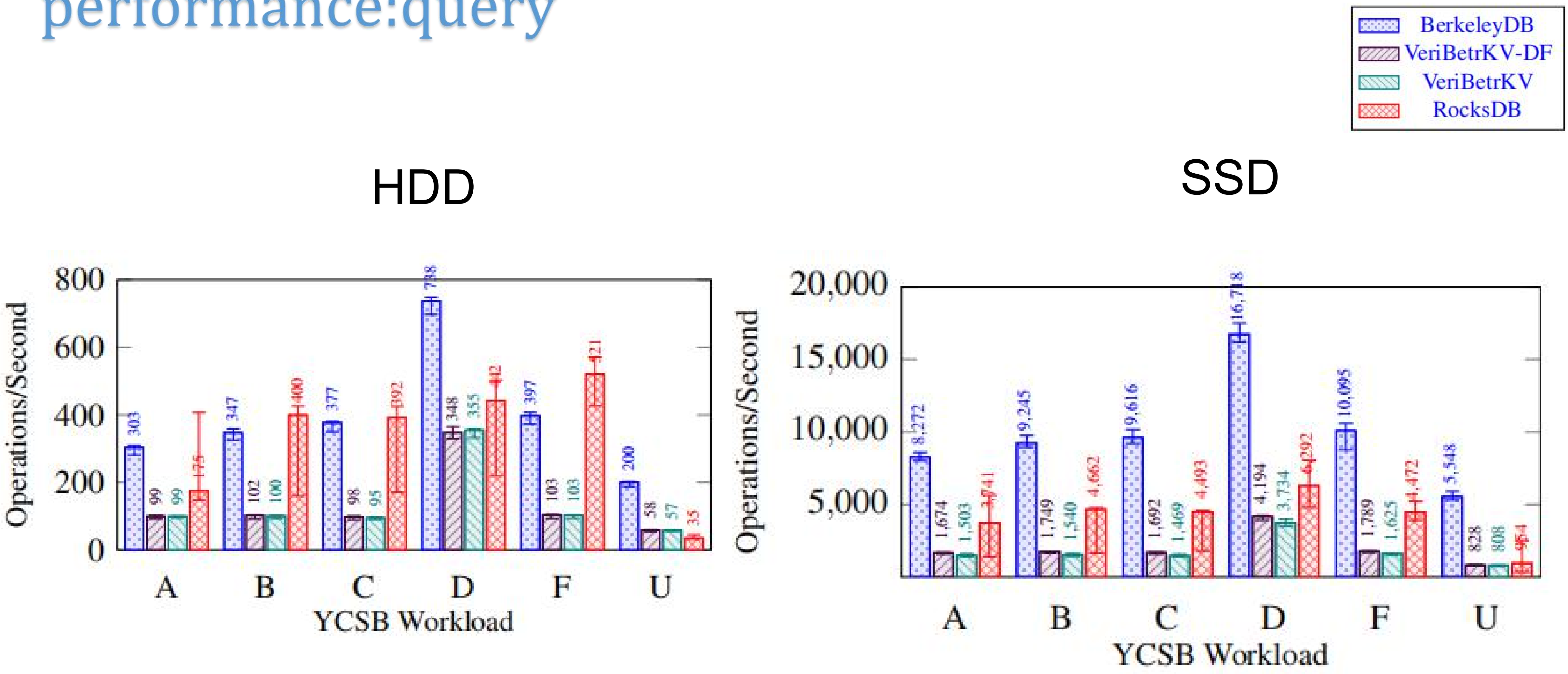


SSD



Evaluation

performance:query



HDD

SSD

Outline

USTC, CHINA
ADSLAB

6 VeriBetrKV

7 Evaluation

8 Conclusion

Conclusion

All in all, this paper presents

- general methodology for verifying asynchronous systems from prior work.
- a Key-Value storage system that advances towards performance of state-of-the-art non-verified systems, with much stronger guarantees