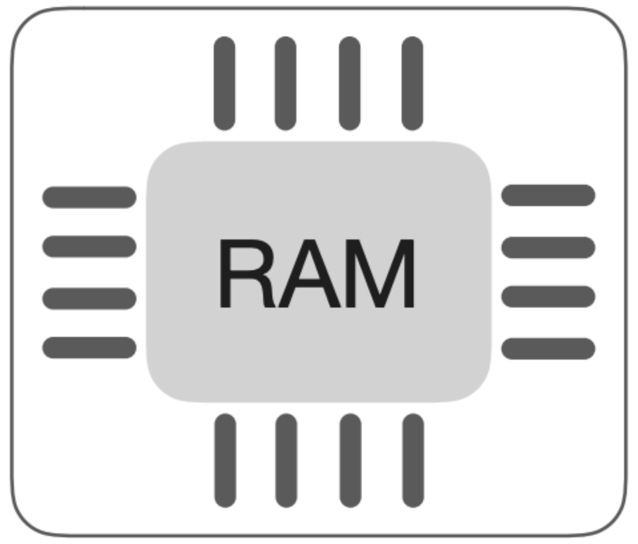
Abstraction for Crash-Resilient Objects

Artem Khyzha
Arm Ltd

Ori Lahav
Tel Aviv University

ESOP 2022

Non-volatile memory

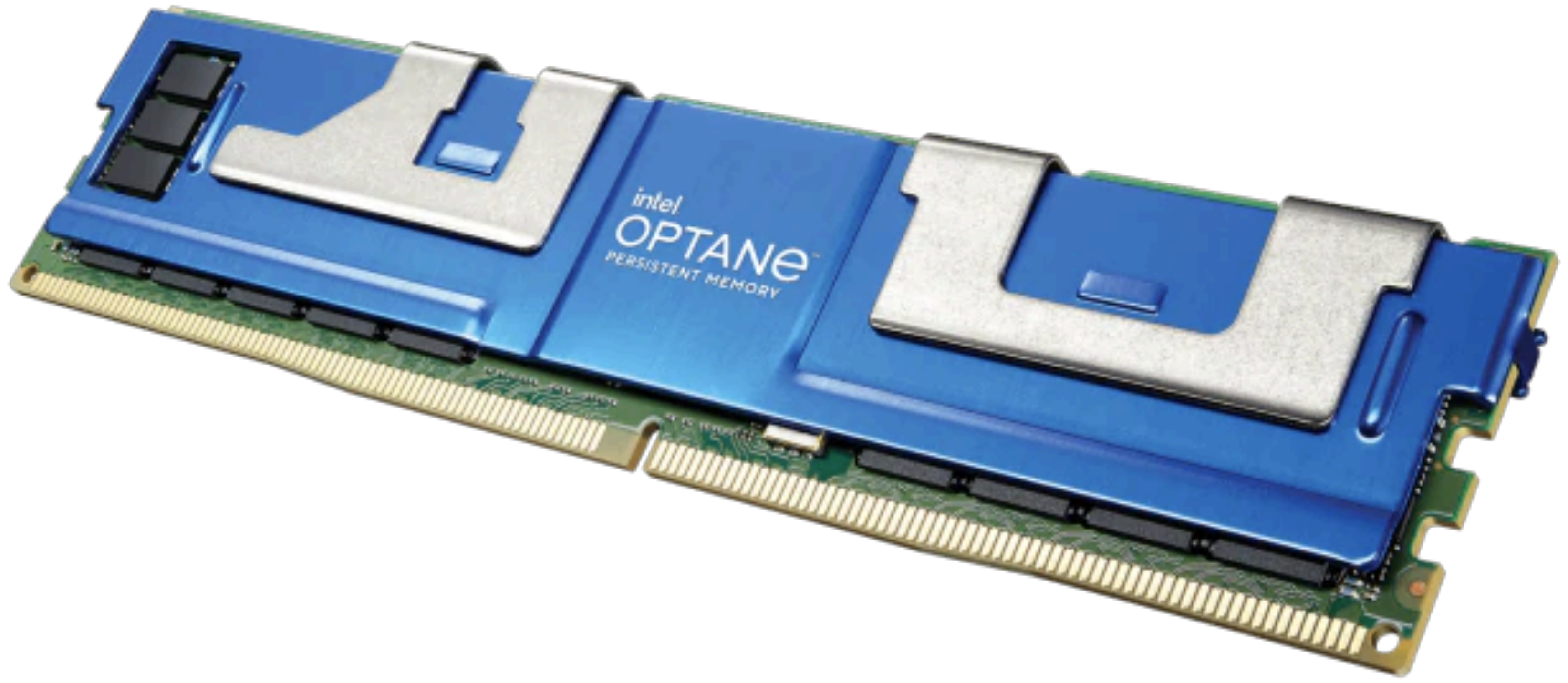


✓ *fast*
✗ *volatile*

✗ *slow*
✓ *persistent*

- NVM provides the best of both worlds:
  - ***fast*** + ***byte-addressable*** (like RAM)
  - ***persistent*** (like HDD)

- Technology is available (e.g., Intel Optane)



New programming challenges

```
{ X = Y = 0 }
   X := 1;
   Y := 1;
    ↯ ↯ ↯
```

Upon recovery, we may get:

```
X = 1 and Y = 1
X = 0 and Y = 0
X = 1 and Y = 0
X = 0 and Y = 1
```

execution continues ahead of persistence

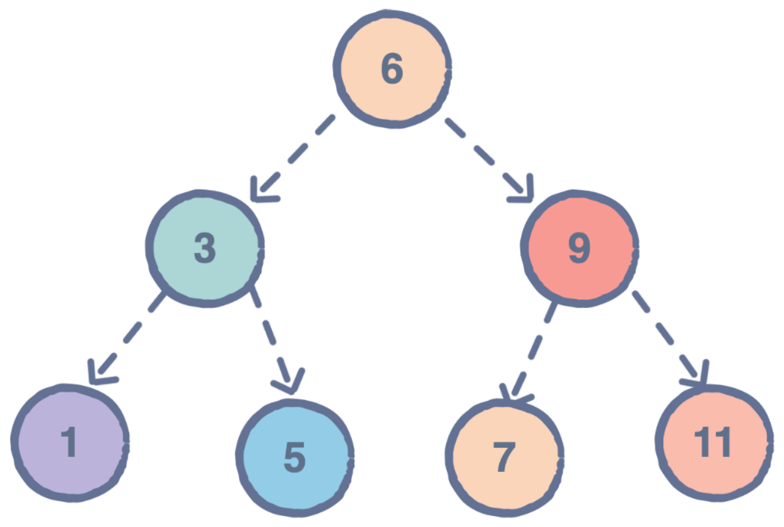
writes may persist out of order

- Explicit persist instructions have to be properly placed:

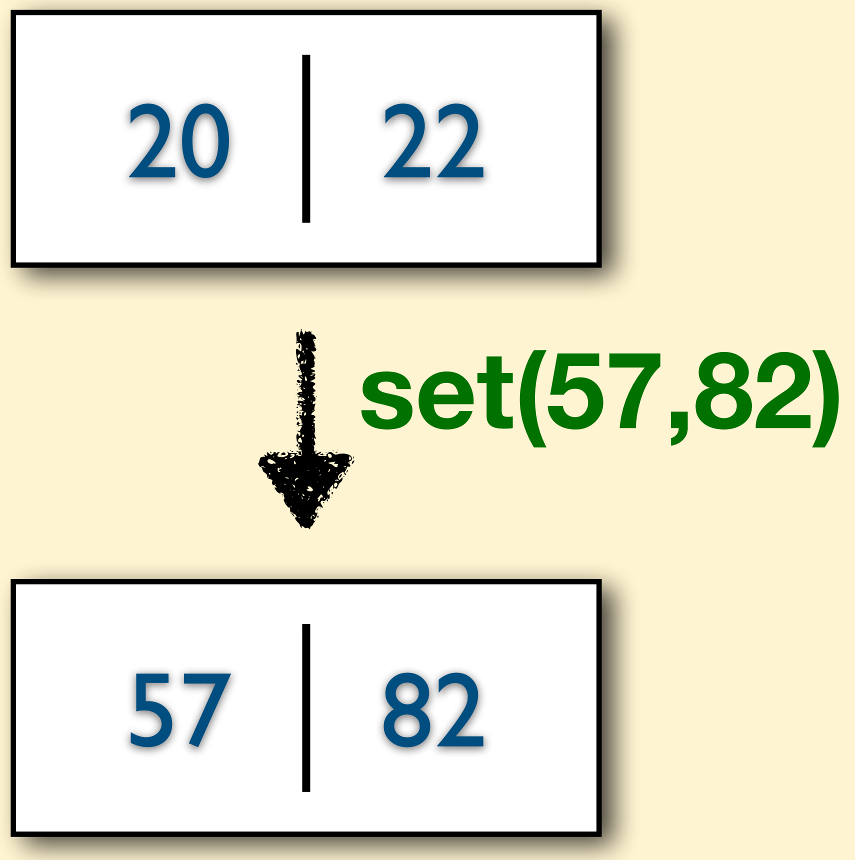
  `CLFLUSH, CLWB, CLFLUSHOPT, SFENCE`

Persistent objects

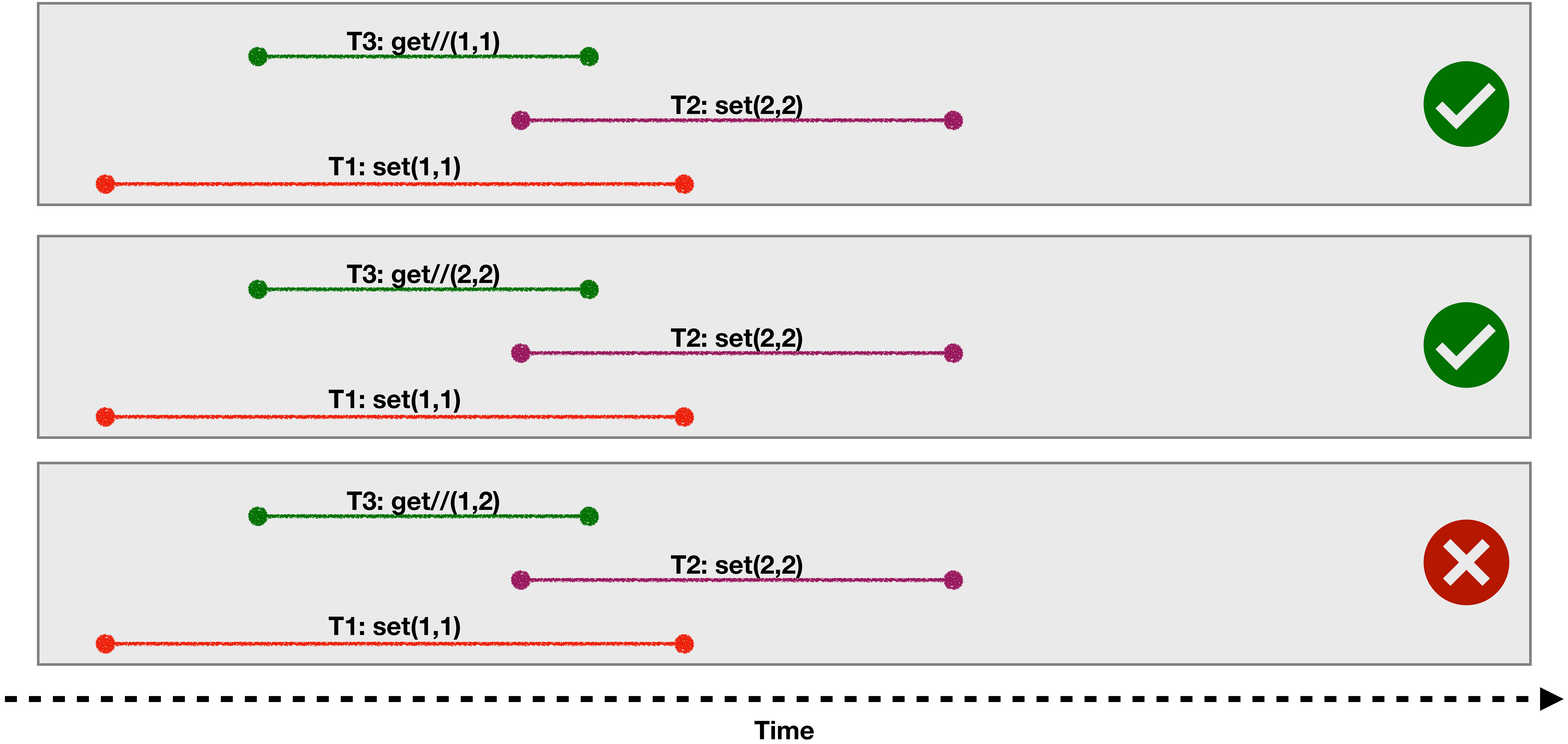
- **Concurrent** objects able to **recover from crashes**
  - Maps, queues, stacks…



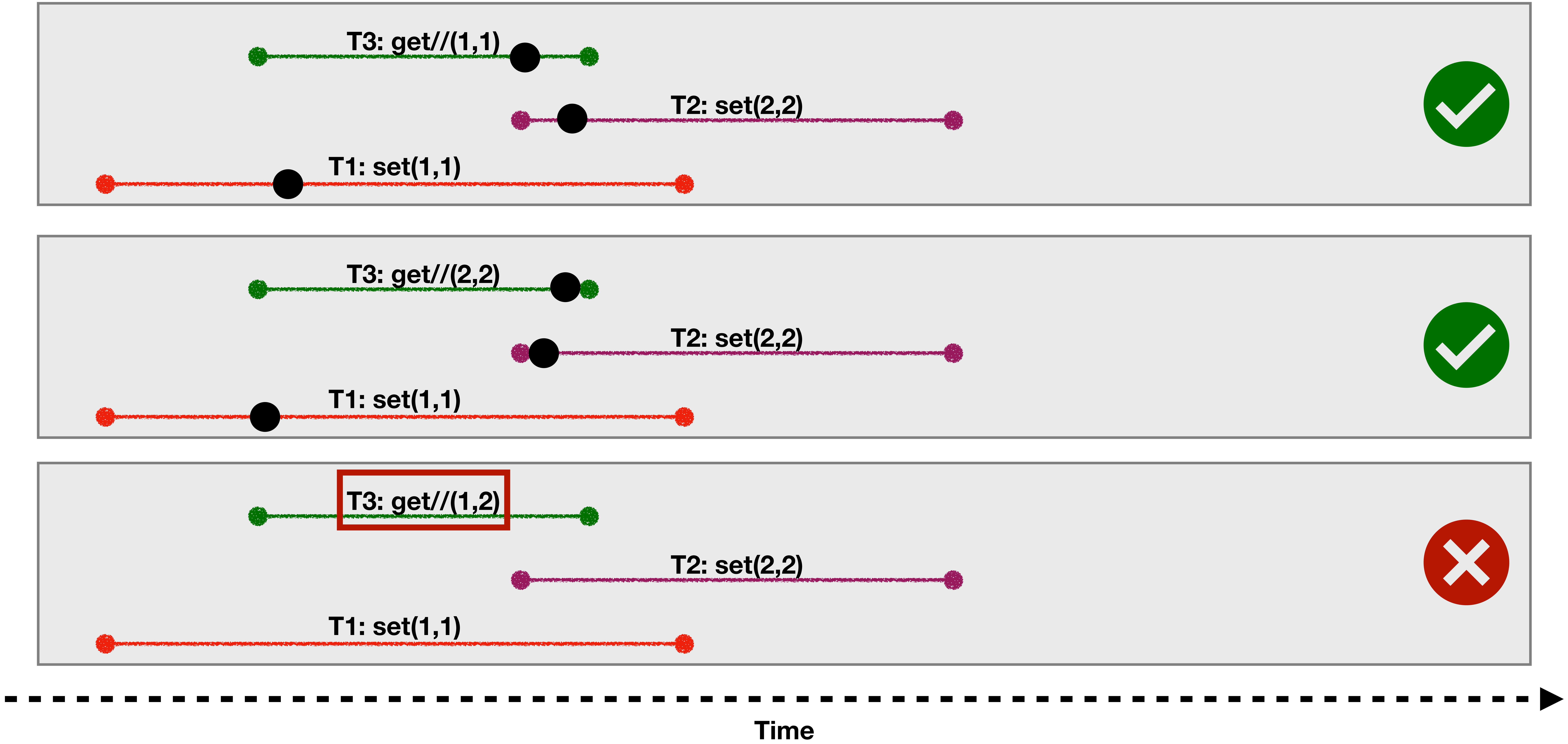
- **Persistent pair** — simple persistent object supporting operations:
  - **set(a,b)**: atomically write a pair of values
  - **get**: atomically read a pair of values
  - **recover**: fix pair after crash
  - **sync**: make sure previous writes to the pair persist



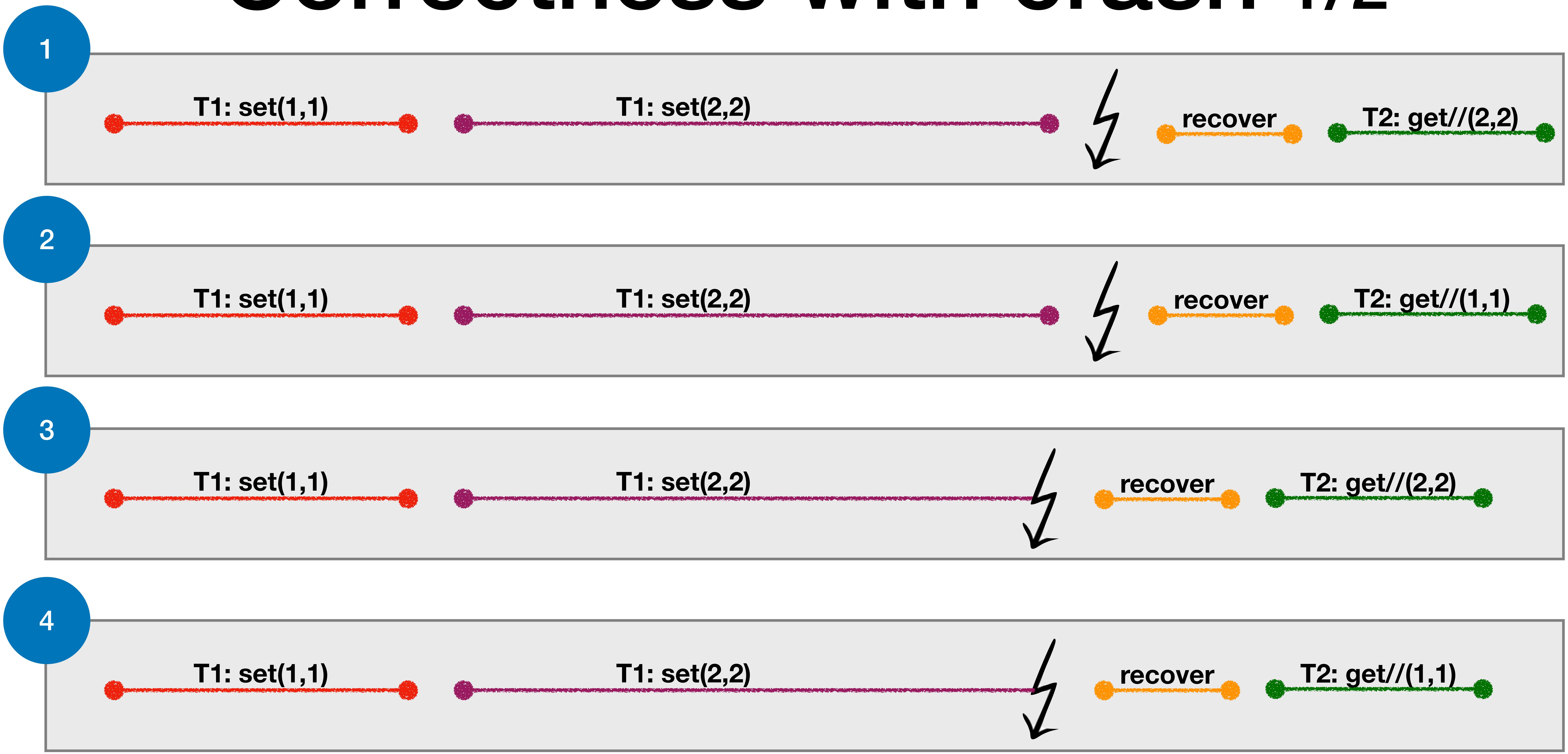
Correctness



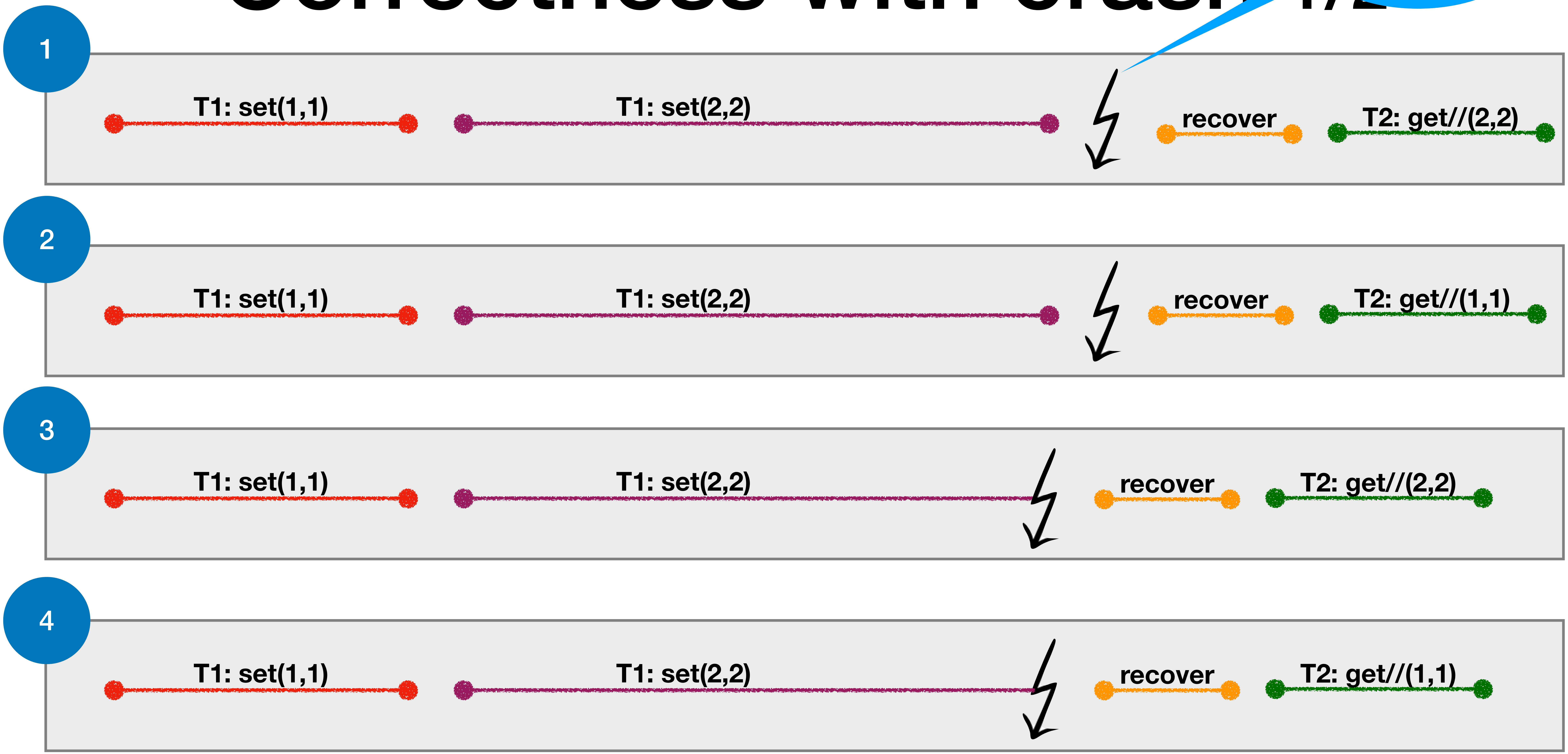
Correctness



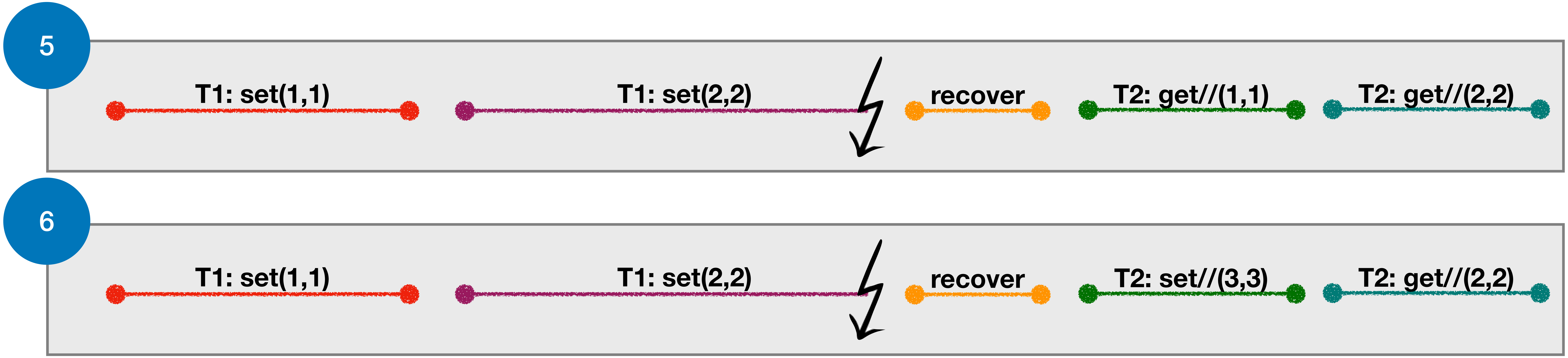
Correctness with crash 1/2



Correctness with crash 1/2



Correctness with crash 2/2



Which linearizability?

Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WING
Carnegie Mellon University

A concurrent object is a data object shared by concurrent processes. Linearizability is a correctness condition for concurrent objects that exploits the semantics of abstract data types. It permits a high degree of concurrency, yet it permits programmers to specify and reason about concurrent objects using known techniques from the sequential domain. Linearizability provides the illusion that each operation applied by concurrent processes takes effect instantaneously at some point between its invocation and its response, implying that the meaning of a concurrent object's operations can be given by pre- and post-conditions. This paper defines linearizability, compares it to other correctness conditions, presents and demonstrates a method for proving the correctness of implementations, and shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.1.3 [**Programming Techniques**]: Concurrent Programming; D.2.1 [**Software Engineering**]: Requirements/Specifications; D.3.3 [**Programming Languages**]: Language Constructs—*abstract data types, concurrent programming structures, data types and structures*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*pre- and post-conditions, specification techniques*

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrrency, correctness, Larch, linearizability, multiprocessing, serializability, shared memory, specification

1. INTRODUCTION

1.1 Overview

Informally, a concurrent system consists of a collection of sequential processes that communicate through shared typed objects. This model encompasses both message-passing architectures in which the shared objects are message queues,

A preliminary version of this paper appeared in the Proceedings of the 14th ACM Symposium on Principles of Programming Languages, January 1987 [21].
This research was sponsored by IBM and the Defense Advanced Research Projects Agents (DOD), ARPA order 4976 (Amendment 20), under contract F33615-87-C-1499, monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB. Additional spport for J. M. Wing was provided in part by the National Science Foundation under grant CCR-8620027. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced

Which linearizability?

Linearizability: A Correctness Condition for Concurrent Objects

MAURICE P. HERLIHY and JEANNETTE M. WI
Carnegie Mellon University

Strict Linearizability and the Power of Aborting

Marcos K. Aguilera* and Svend Frølund†
HP Labs, Palo Alto, CA 94304

21 November 2003

Strict linearizability

Linearizability of Persistent Memory Objects Under a Full-System-Crash Failure Model

Joseph Izraelevitz(✉), Hammurabi Mendes, and Michael L. Scott

University of Rochester, Rochester, NY 14627-0226, USA
{jhi1,hmendes,scott}@cs.rochester.edu

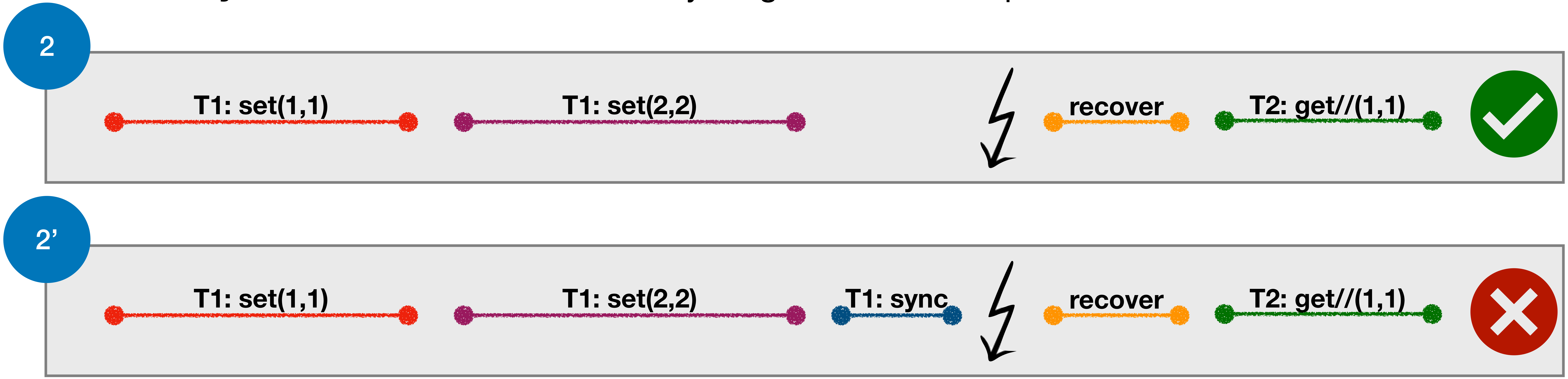
Durable linearizability

Buffered durable linearizability

based on sequential crash-free specifications

Gaps 1/2

- More than one way to interpret sequential crash-free specifications in a crashing environment
- The **sync** method does not mean anything in crash-free specifications



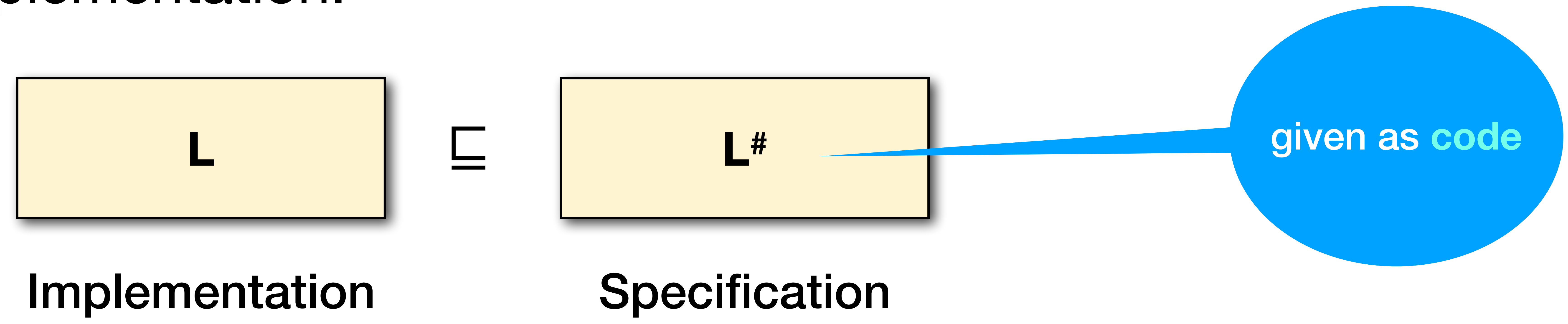
- We possibly want to *mix-and-match* correctness criteria in the same program

Gaps 2/2

- Existing correctness notions were **not** related with **contextual refinement**
  - What ***code*** describes strict/durable/buffered linearizable objects?
  - Cannot be directly used in verification of client programs

Our approach

- Instead of different linearizability-like conditions, we focus on *refinement* w.r.t. another implementation:



- Ensures that the implementation behaves like the specification **under any context**

- Include special constructs in the language for intuitive specifications

Goal: Library abstraction theorem

library correctness criterion

If ____????____, then for every* client program **C**

$$\text{Behaviors}(\mathbf{C}[L]) \subseteq \text{Behaviors}(\mathbf{C}[L^{\#}])$$

* that uses disjoint memory & follows the calling policy

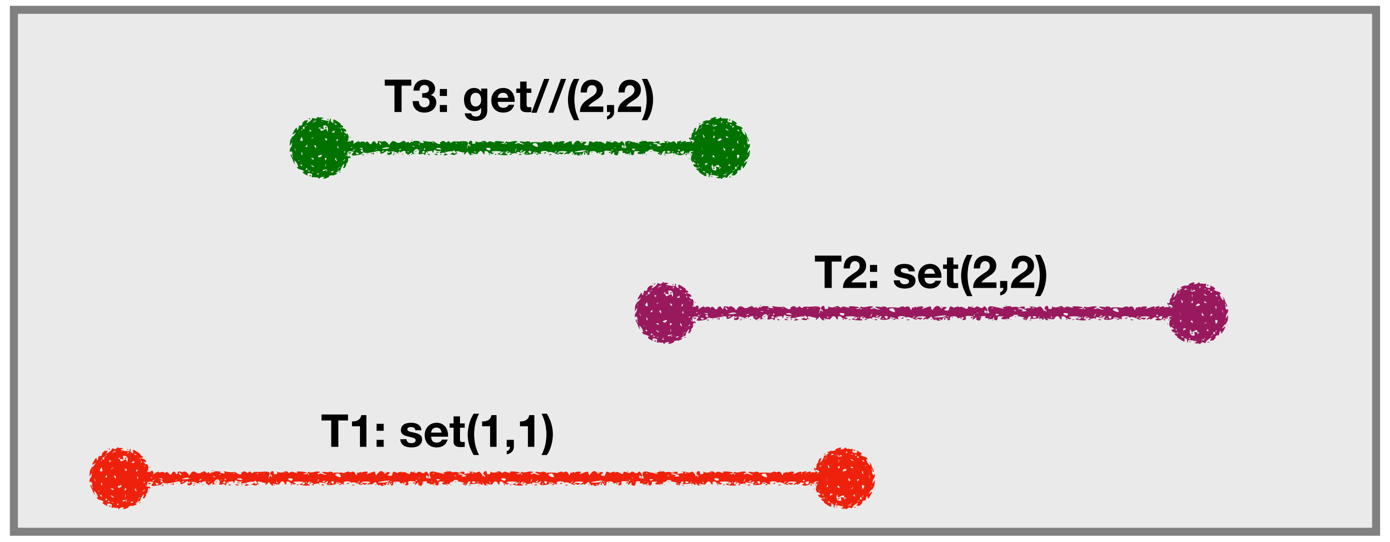
For **volatile objects**

If ????, then for every* client program **C**

$$\text{Behaviors}(\mathbf{C}[L]) \subseteq \text{Behaviors}(\mathbf{C}[L^{\#}])$$

$$\text{Histories}(\mathbf{MGC}[L]) \subseteq \text{Histories}(\mathbf{MGC}[L^{\#}])$$

A particular program that **only** invokes library methods (repeatedly, concurrently, with arbitrary arguments)



- Abstraction theorem: *"a formal confirmation of this folklore"* [Filipovic et al. ESOP'09, TCS'10]
- Extended to x86-TSO [Burckhardt et al. ESOP'12]

Application for a volatile pair

X | Y

```
set(a,b):
  atomic{
      X := a;
      Y := b;
  }
  return;
```

```
get:
  atomic{
      a := X;
      b := Y;
  }
  return(a,b);
```

Specification ($L^{\#}$)

Application for a volatile pair

Specification construct

X | Y

```
set(a,b):
   atomic{
      X := a;
      Y := b;
   }
   return;
```

```
get:
   atomic{
      a := X;
      b := Y;
   }
   return(a,b);
```

Specification (L#)

version number

C | X | Y

```
set(a,b):
  lock();
  C := C + 1;
  X := a;
  Y := b;
  C := C + 1;
  unlock();
  return;
```

```
get:
  BEGIN:   s1 := C;
  if odd(s1) then goto BEGIN
  a := X;
  b := Y;
  s2 := C;
  if s1 ≠ s2 then goto BEGIN
  return(a,b);
```

Implementation (L)

Application for a volatile pair

```
set(a,b):
    lock();
    C := C + 1;
    X := a;
    Y := b;
    C := C + 1;
    unlock();
    return;
```

```
get:
    BEGIN:  s₁ := C;
    if odd(s₁) then goto BEGIN
    a := X;
    b := Y;
    s₂ := C;
    if s₁ ≠ s₂ then goto BEGIN
    return(a,b);
```

$\sqsubseteq$

```
set(a,b):
  atomic{
     X := a;
     Y := b;
  }
  return;
```

```
get:
  atomic{
     a := X;
     b := Y;
  }
  return(a,b);
```

- We have Histories(**MGC**[L]) $\subseteq$ Histories(**MGC**[$L^\#$])
    - shown using a simulation argument
    - proof obligation for the library developer

$\equiv$ standard linearizability when $L^\#$ wraps a sequential implementation in an atomic block

- It follows that for every client program **C:**

Behaviors(**C**[L]) $\subseteq$ Behaviors(**C**[$L^\#$])

Back to NVM

- Multiple formal models:

Persistency Semantics of the Intel-x86 Architecture

AZALEA RAAD, MPI-SWS, Germany
JOHN WICKERSON, Imperial College London, UK
GIL NEIGER, Intel Labs, US
VIKTOR VAFEIADIS, MPI-SWS, Germany

POPL'20

Weak Persistency Semantics from the Ground Up

Formalising the Persistency Semantics of ARMv8 and Transactional Models

AZALEA RAAD, MPI-SWS, Germany
JOHN WICKERSON, Imperial College London, UK
VIKTOR VAFEIADIS, MPI-SWS, Germany

OOPSLA'19

Taming x86-TSO Persistency

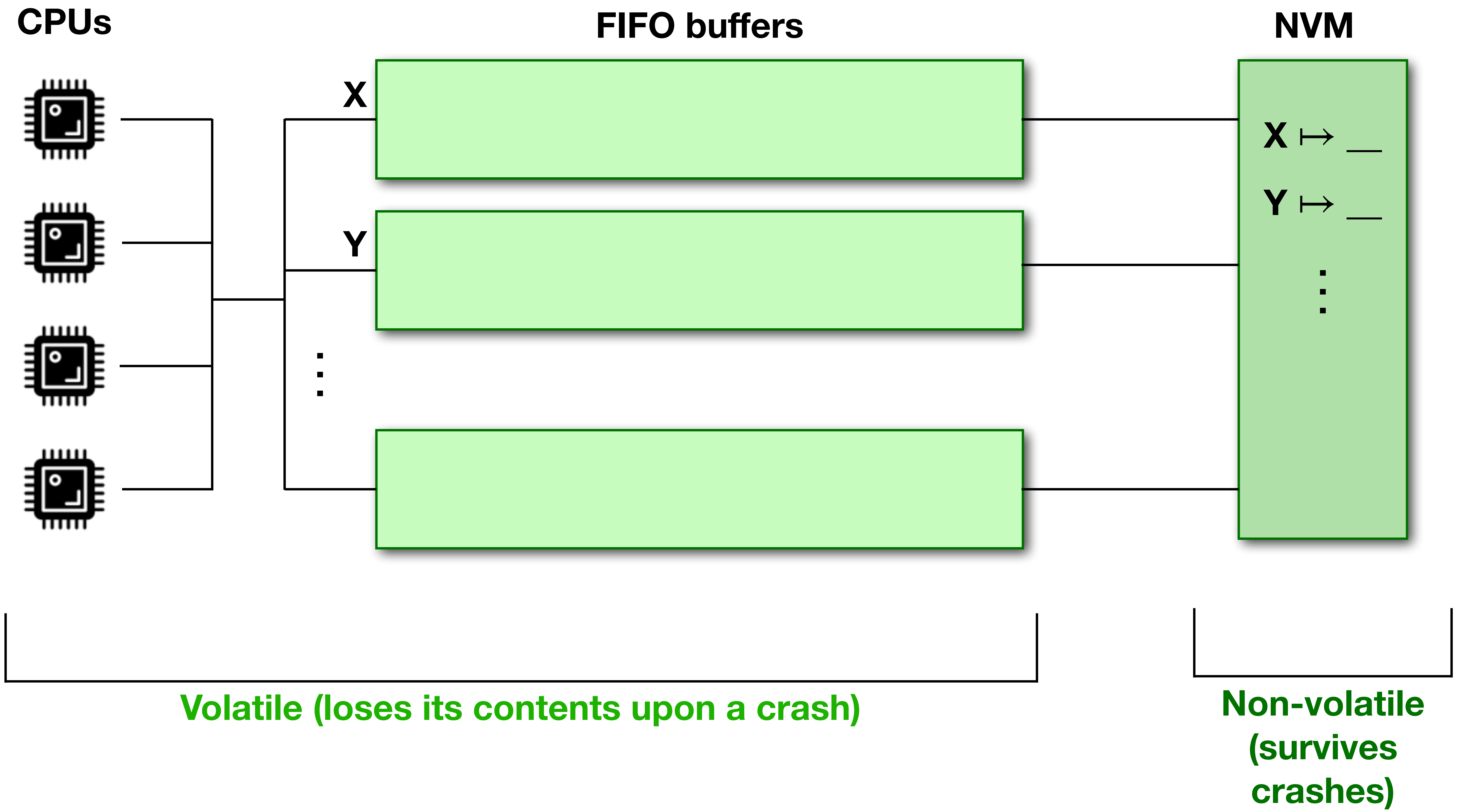
ARTEM KHYZHA, Tel Aviv University, Israel
ORI LAHAV, Tel Aviv University, Israel

POPL'21

- We consider the simplest model: **Persistent Sequential Consistency (PSC)**

- can be mapped to **x86-TSO** (by adding appropriate `MFENCE` and `SFNECE`)

The PSC model

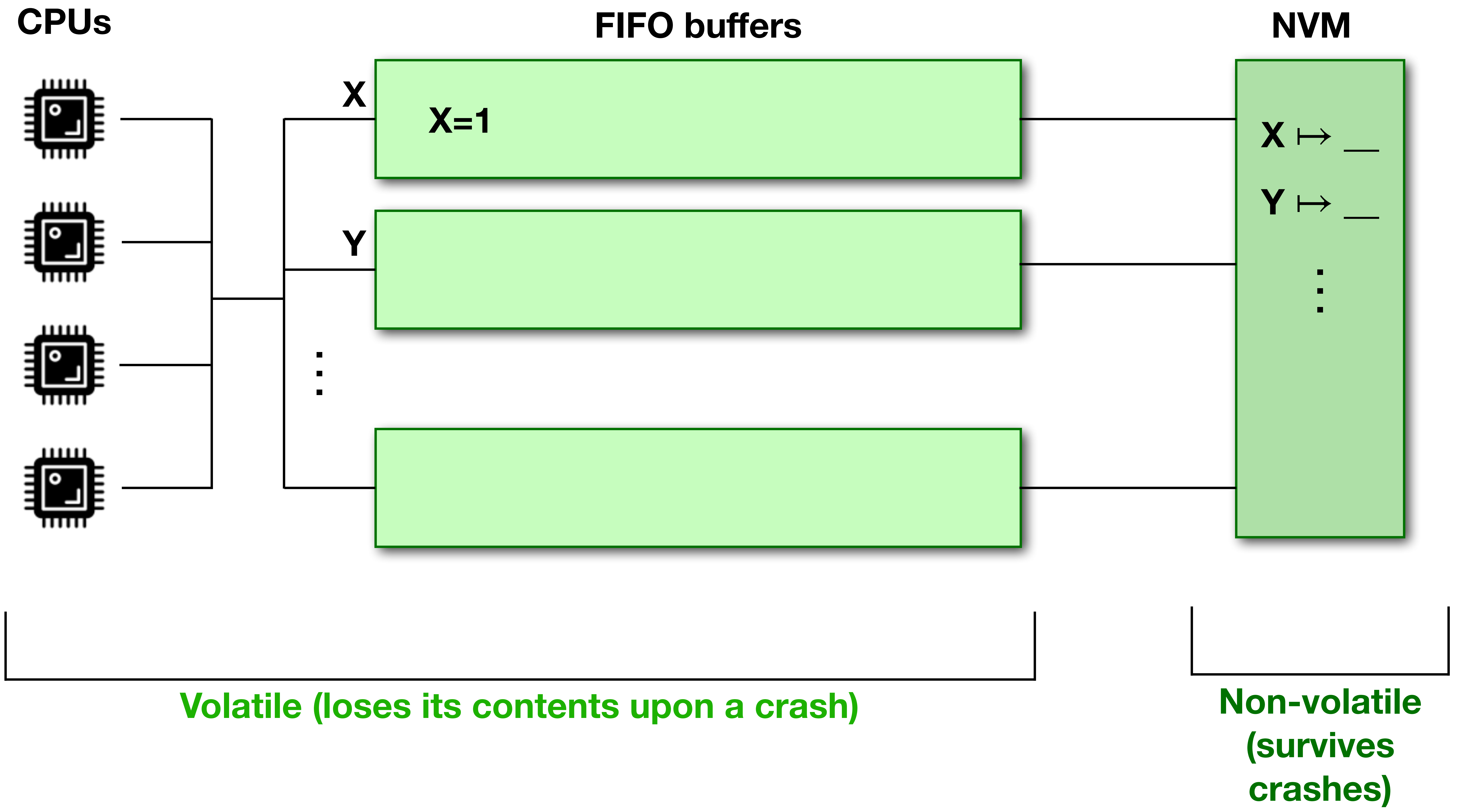



```
{ X = Y = 0 }
X := 1;
Y := 1;
↯↯↯
```

```
{ X = Y = 0 }
X := 1;
FLUSH(X);
Y := 1;
↯↯↯
```

```
{ X = Y = 0 }
X := 1;
FLUSH-OPT(X);
SFENCE;
Y := 1;
↯↯↯
```

The PSC model

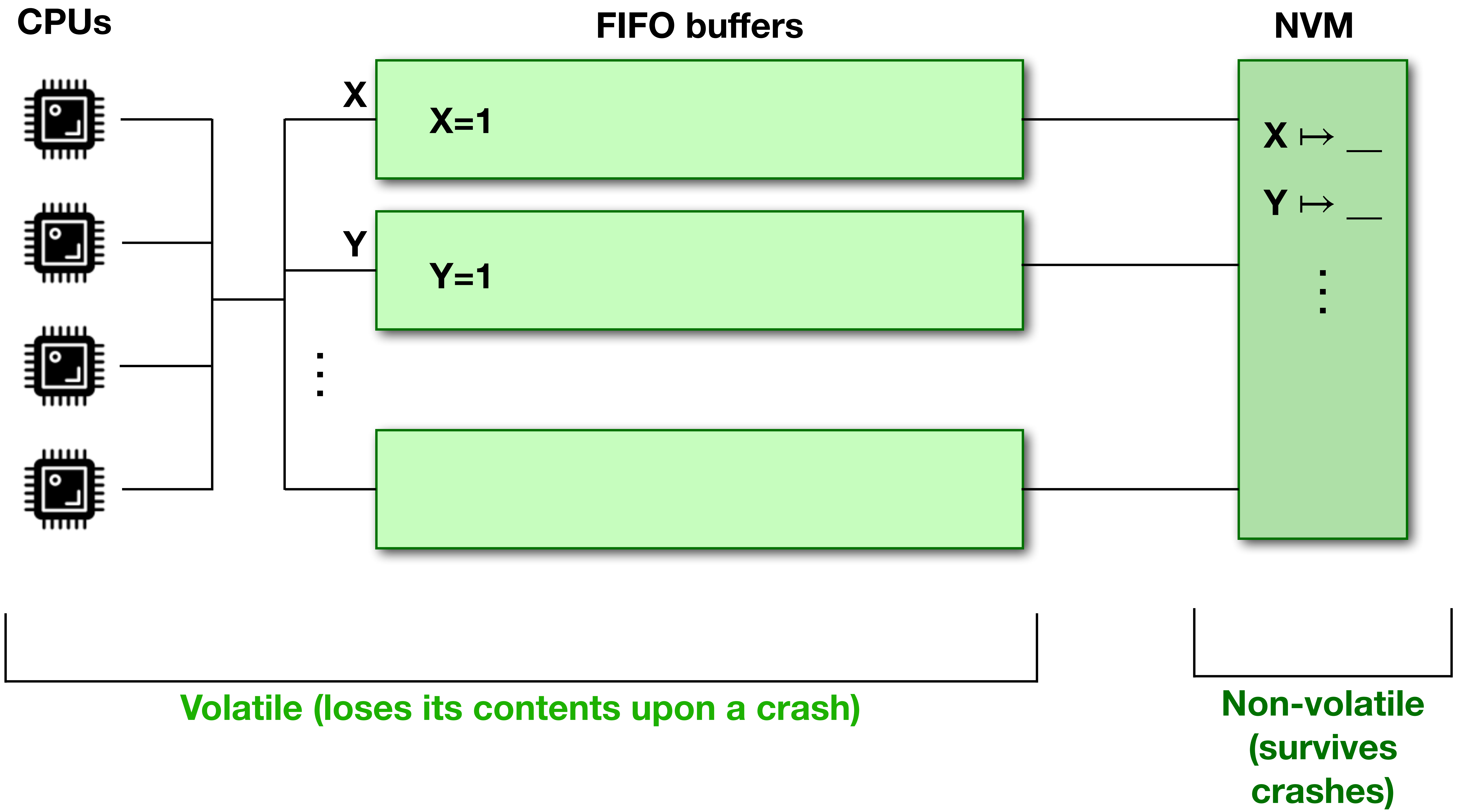



```
{ X = Y = 0 }
X := 1;
Y := 1;
↯↯↯
```

```
{ X = Y = 0 }
X := 1;
FLUSH(X);
Y := 1;
↯↯↯
```

```
{ X = Y = 0 }
X := 1;
FLUSH-OPT(X);
SFENCE;
Y := 1;
↯↯↯
```

The PSC model

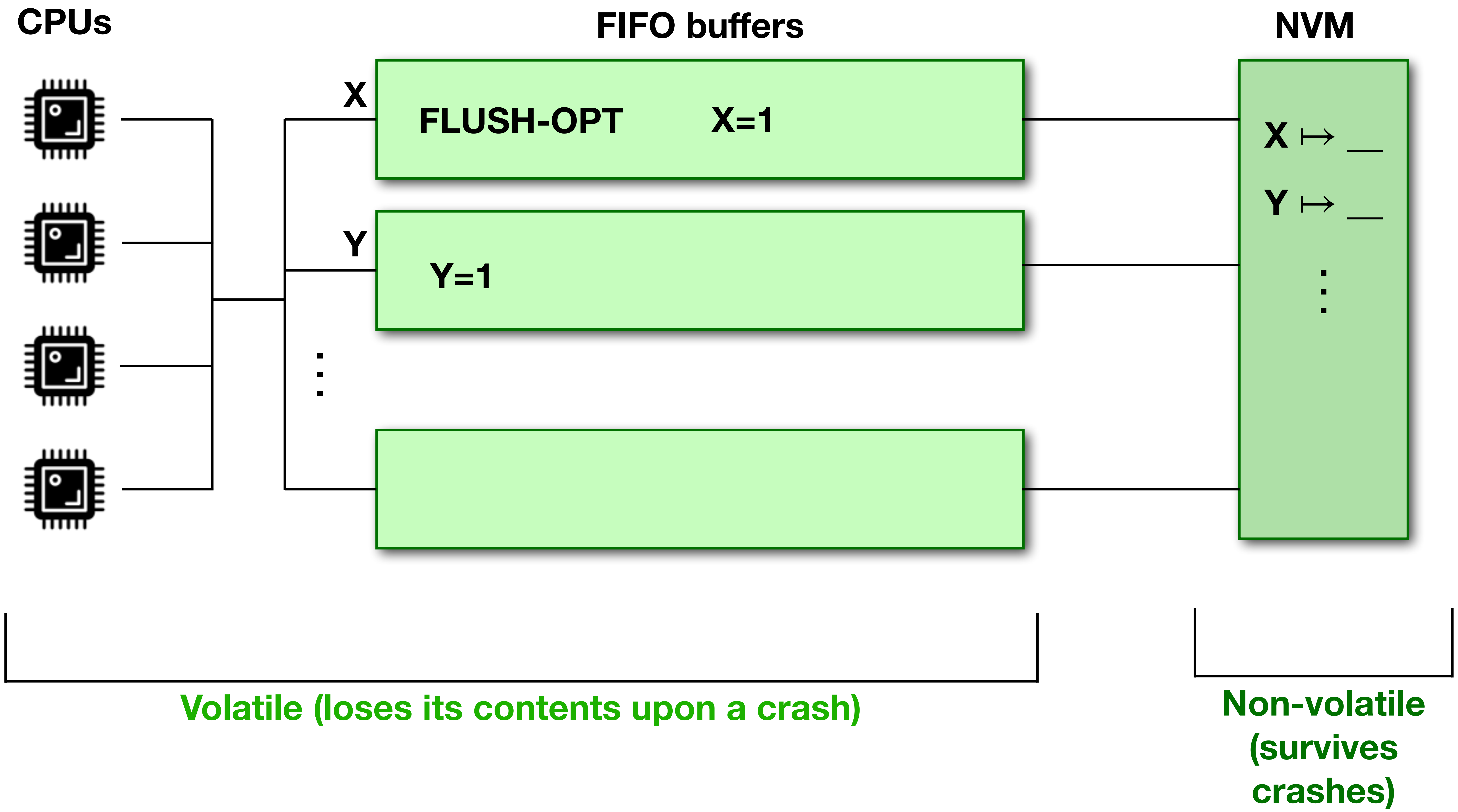



```
{ X = Y = 0 }
X := 1;
Y := 1;
↯↯↯
```

```
{ X = Y = 0 }
X := 1;
FLUSH(X);
Y := 1;
↯↯↯
```

```
{ X = Y = 0 }
X := 1;
FLUSH-OPT(X);
SFENCE;
Y := 1;
↯↯↯
```

The PSC model




```
{ X = Y = 0 }
X := 1;
Y := 1;
↯↯↯
```

```
{ X = Y = 0 }
X := 1;
FLUSH(X);
Y := 1;
↯↯↯
```

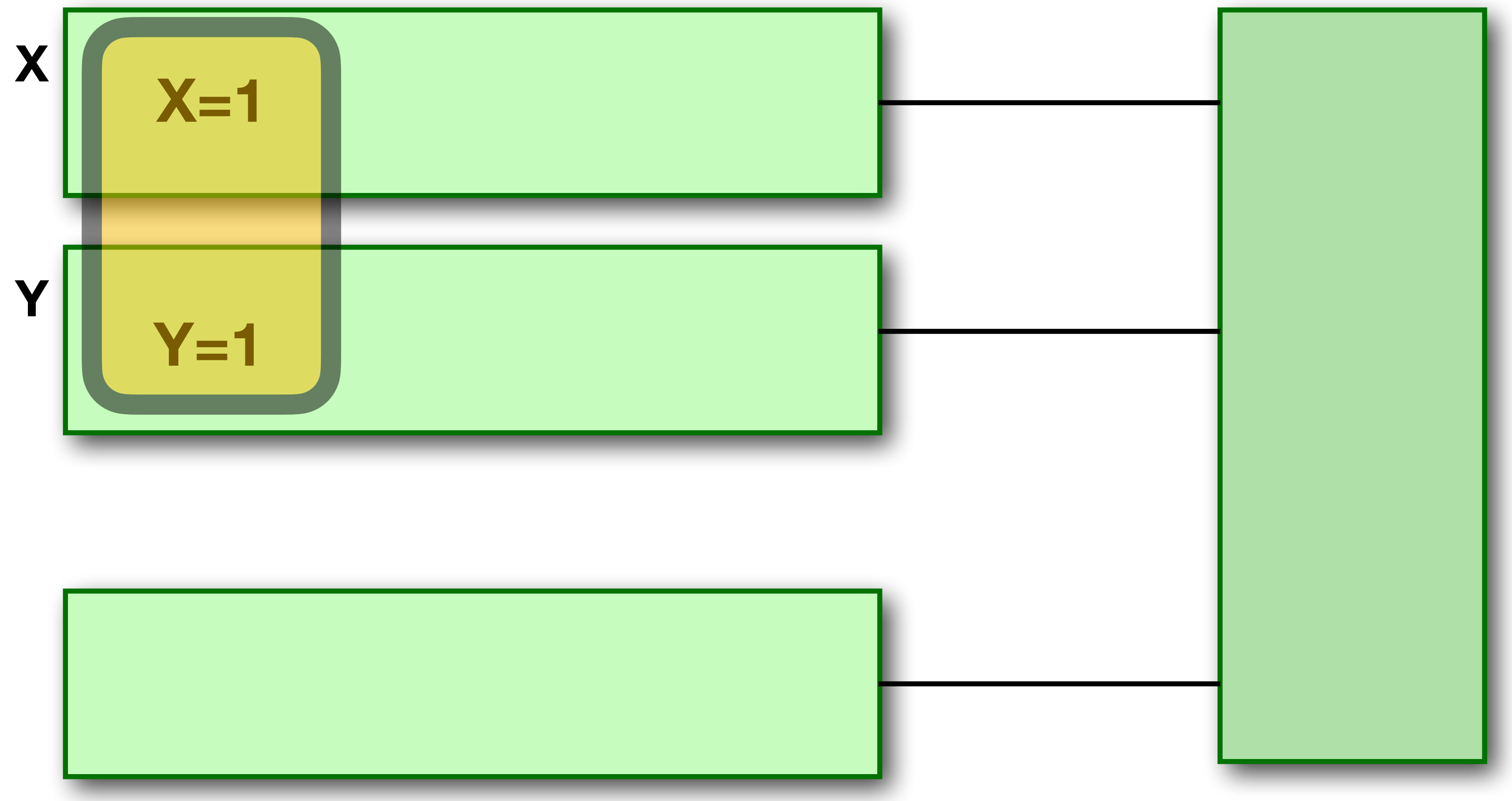
```
{ X = Y = 0 }
X := 1;
FLUSH-OPT(X);
SFENCE;
Y := 1;
↯↯↯
```

Persistent blocks

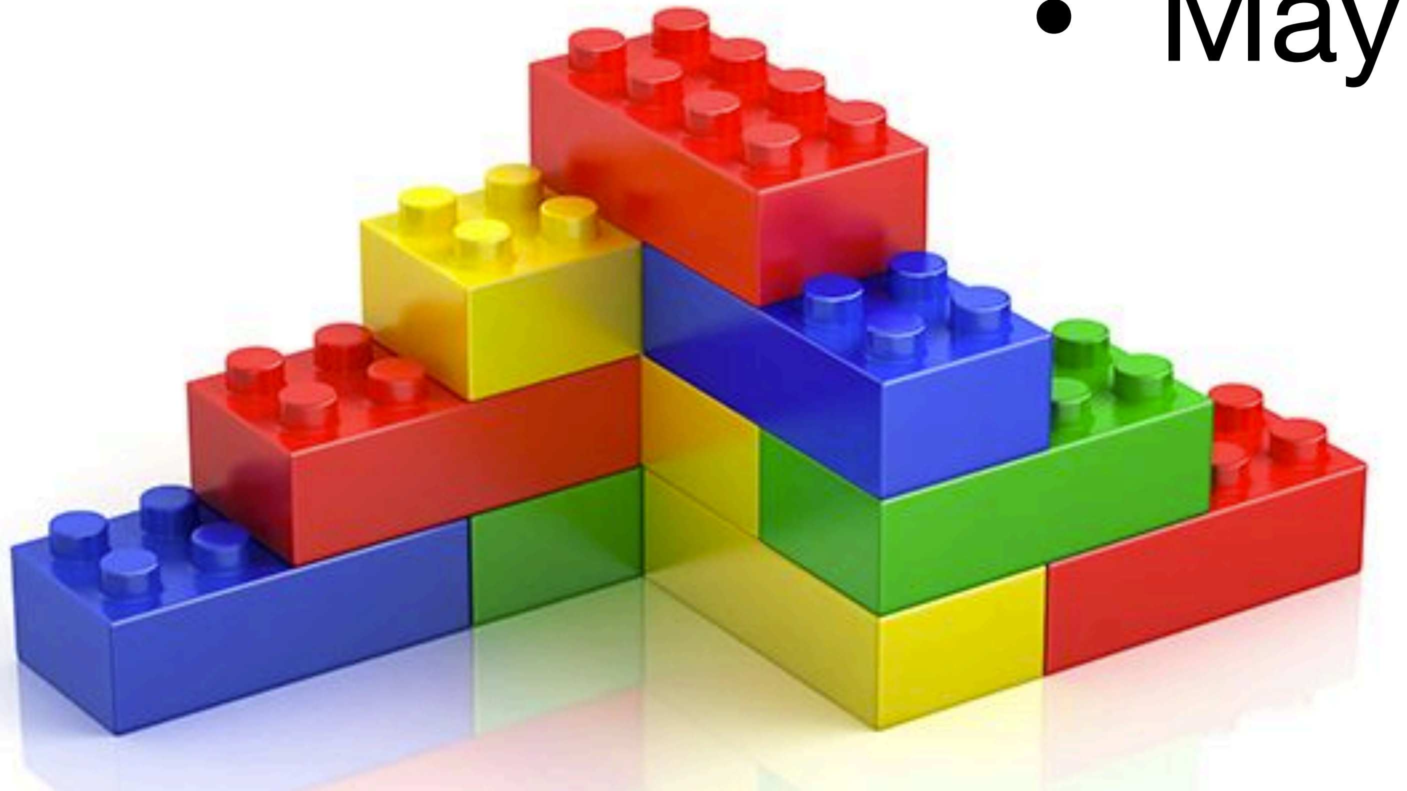
- We introduce a construct to control persistency:

  persist { … }

  - All writes persist **simultaneously**
  - **No earlier** than the block ends
  - May be forced by a **`FLUSH`** instruction




```
persist {
    X := 1;
    Y := 1;
}
```

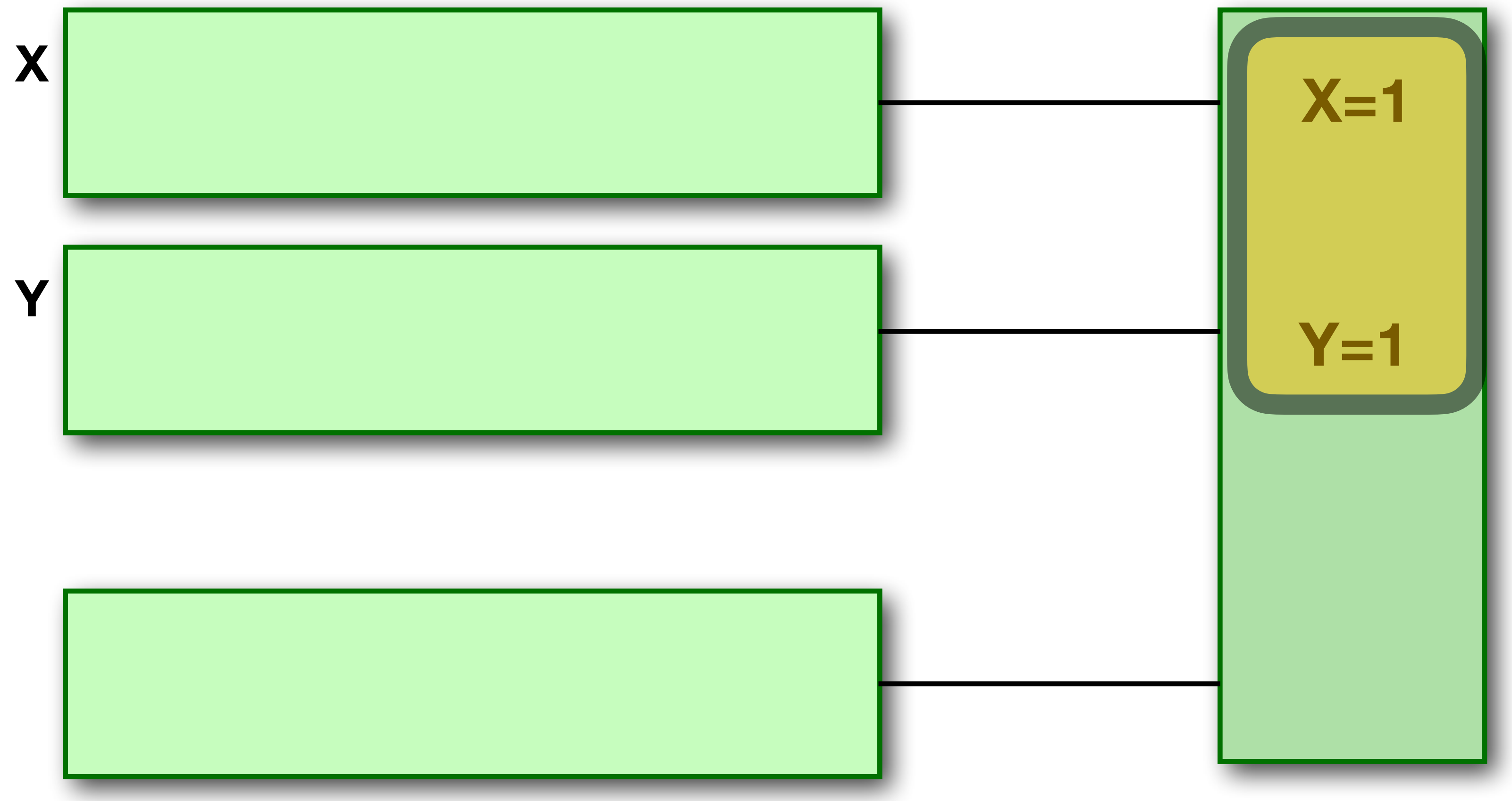


Persistent blocks

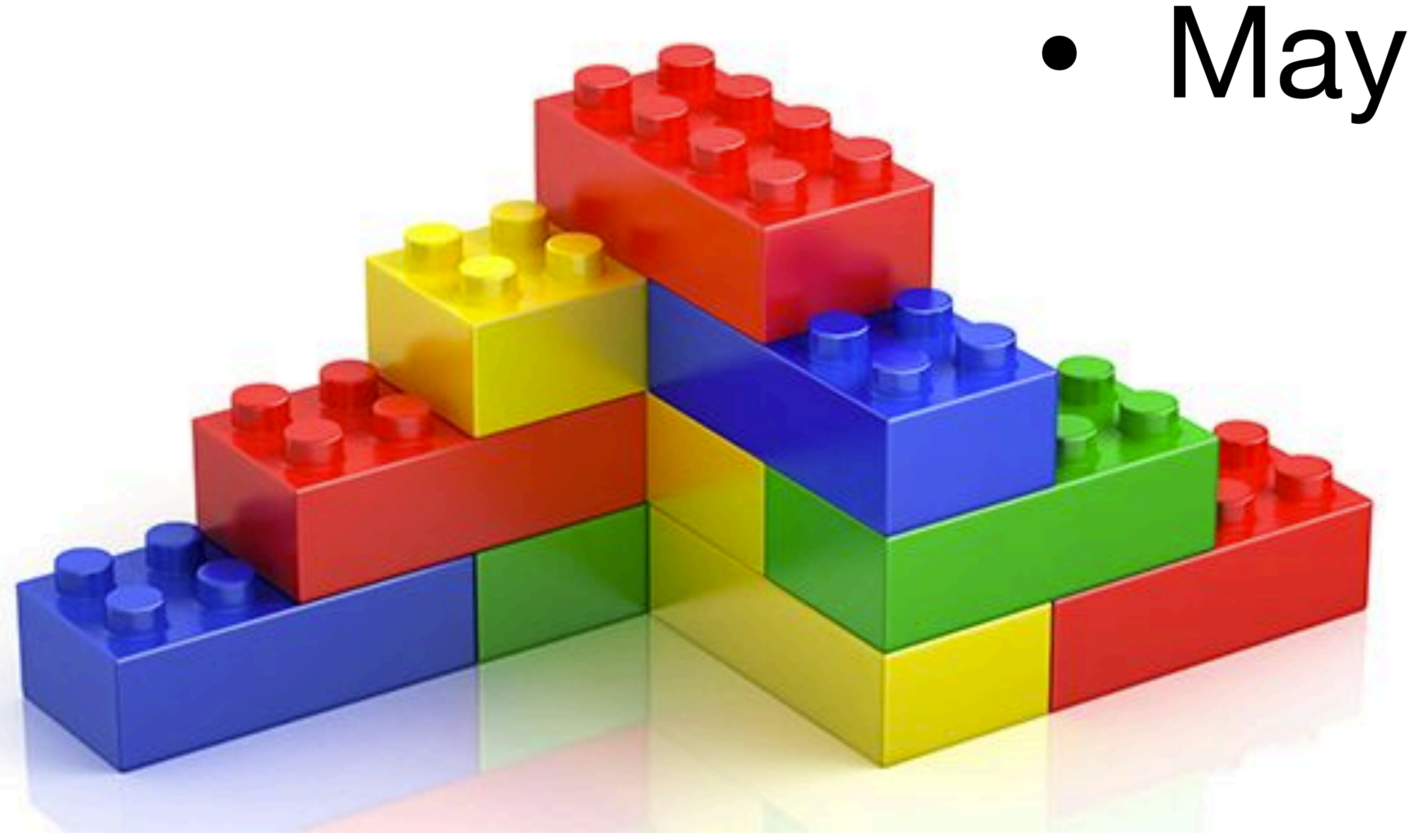
- We introduce a construct to control persistency:

  persist { … }

  - All writes persist **simultaneously**
  - **No earlier** than the block ends
  - May be forced by a **`FLUSH`** instruction




```
persist {
    X := 1;
    Y := 1;
}
```



Specification #1 of a persistent pair

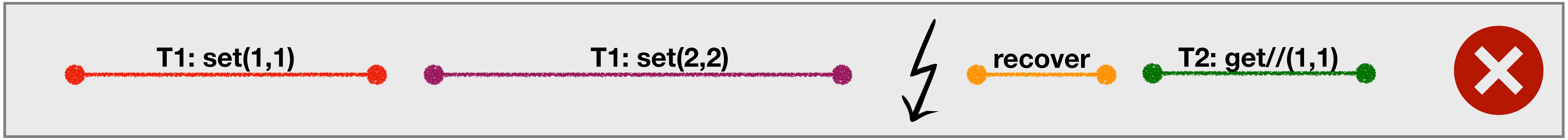
```
set(a,b):
  atomic{
    persist{
      X = a;
      Y = b;
    }
    FLUSH(X);
  }
return;
```

```
get:
  atomic{
      a = X;
      b = Y;
  }
  return(a,b);
```

```
sync:
    return;
```

```
recover:
    return;
```

durable pair



Specification #2 of a persistent pair

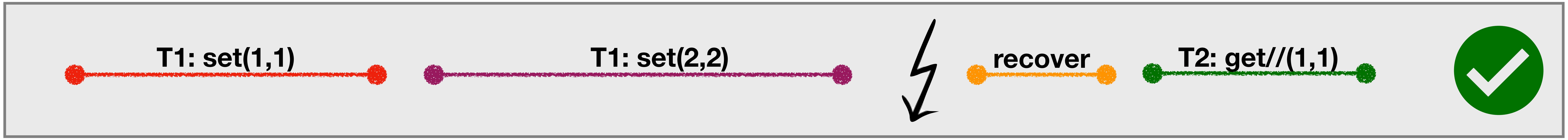
```
set(a,b):
  atomic{
    persist{
      X = a;
      Y = b;
    }
    ~~FLUSH(X)~~;
  }
return;
```

```
get:
  atomic{
      a = X;
      b = Y;
  }
  return(a,b);
```

```
sync:
  FLUSH(X);
  return;
```

```
recover:
  return;
```

buffered durable pair



Library abstraction theorem

If ____????____, then for every* client program **C**

$$\text{behaviors}(\mathbf{C}[L]) \subseteq \text{behaviors}(\mathbf{C}[L^{\#}])$$

Candidate library correctness criterion:

$$\text{Histories}(\mathbf{MGC}[L]) \subseteq \text{Histories}(\mathbf{MGC}[L^{\#}])$$

Challenge in NVM

Implementation | **Specification**

```
foo:
    return;
```

$\stackrel{?}{\sqsubseteq}$

```
foo:
    SFENCE;
    return;
```

✔ Histories(**MGC**[L]) $\subseteq$ Histories(**MGC**[L$^{\#}$])

```
{ A = 0 }
A := 1;
FLUSH-OPT(A);
foo();
↯↯↯
{ A = ?? }
```

BROKEN!

- **SFENCE** has a global effect
- Client-library interaction is **not** fully captured in passed and returned values

Solution #1

- Introduce a "local SFENCE instruction"

```
foo:
    return;
```

$\sqsubseteq$

```
foo:
    LSFENCE(X);
    return;
```

```
{ A = 0 }
A := 1;
FLUSH-OPT(A);
foo();
↯↯↯
{ A = ?? }
```

- **LSFENCE** effect by library is confined to library code

can be either 0 or 1 for both L and L#

Solution #2

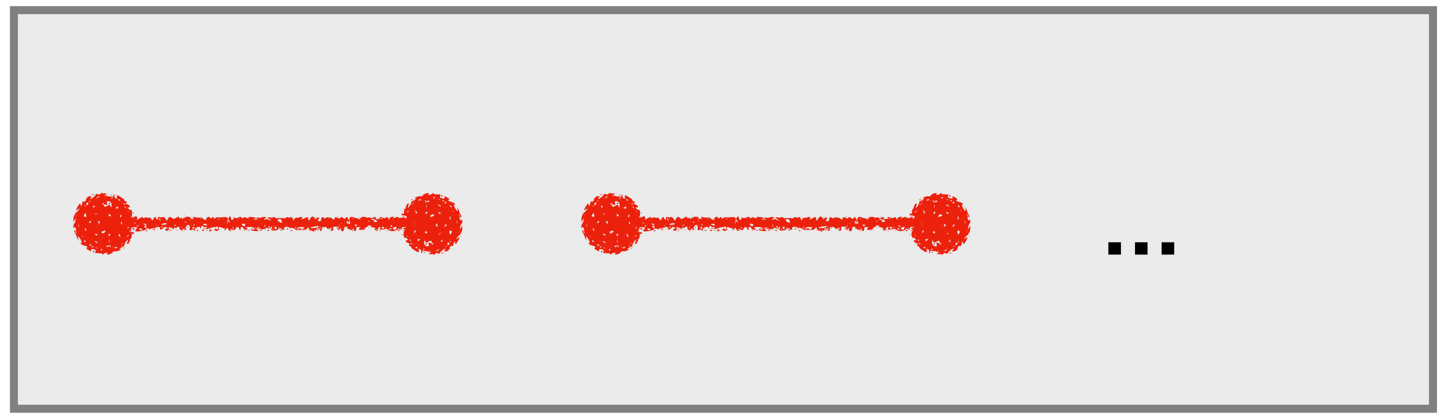
- Make histories more expressive

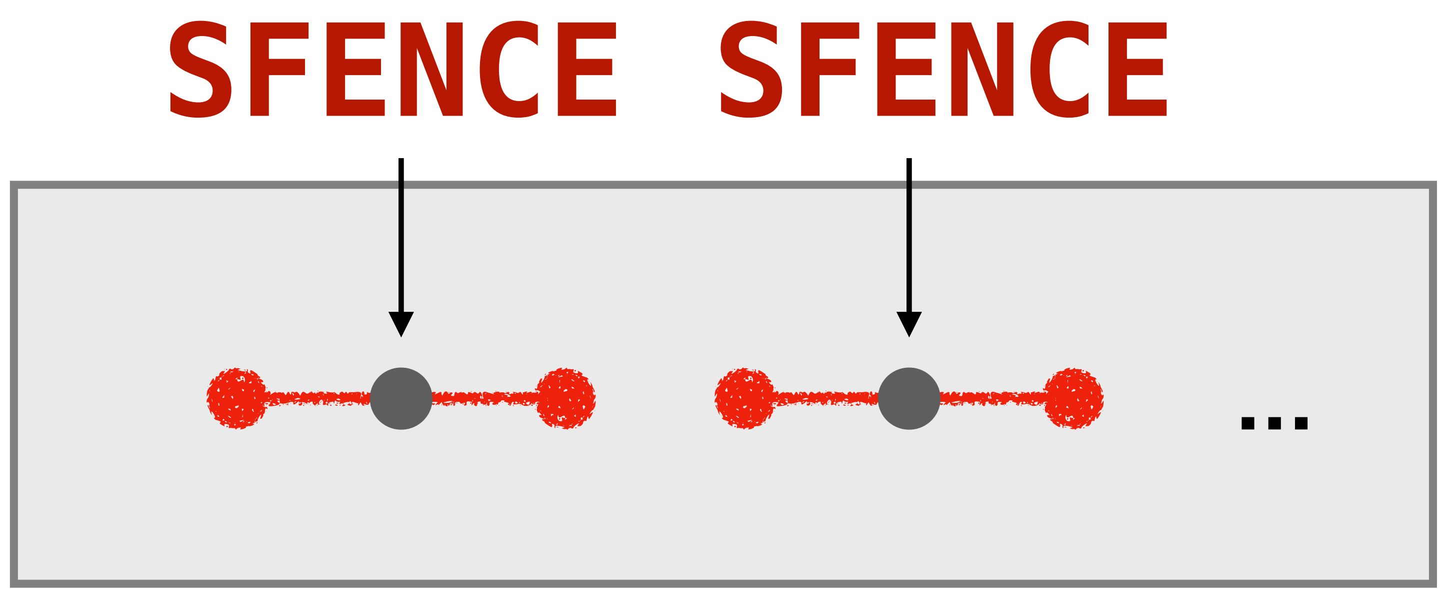
```
foo:
    return;
```

$\not\sqsubseteq$

```
foo:
    SFENCE;
    return;
```

Histories(**MGC**[L]) $\not\subseteq$ Histories(**MGC**[L$^\#$])





Library abstraction theorem

include CALL, RETURN, and SFENCE transitions

If Histories(**MGC**[L]) $\subseteq$ Histories(**MGC**[L$^{\#}$]),
then for every* client program **C:**

$$\text{Behaviors}(\mathbf{C}[L]) \subseteq \text{Behaviors}(\mathbf{C}[L^{\#}])$$

in the extension of the **PSC** model with **persistence blocks** and **local sfences**

Library abstraction theorem

- The key of the **proof** is a **“composition lemma”**:
  - Allows us to compose traces of client and library provided that they induce the same history
  - Formally shows that client-library interaction is fully captured in histories
- **Compositionality** (a.k.a. **locality**) is a corollary:

$$\frac{\text{Histories}(\text{MGC}[L_1]) \subseteq \text{Histories}(\text{MGC}[L_1^\#]) \qquad \text{Histories}(\text{MGC}[L_2]) \subseteq \text{Histories}(\text{MGC}[L_2^\#])}{\text{Histories}(\text{MGC}[L_1 \uplus L_2]) \subseteq \text{Histories}(\text{MGC}[L_1^\# \uplus L_2^\#])}$$

Application for pairs

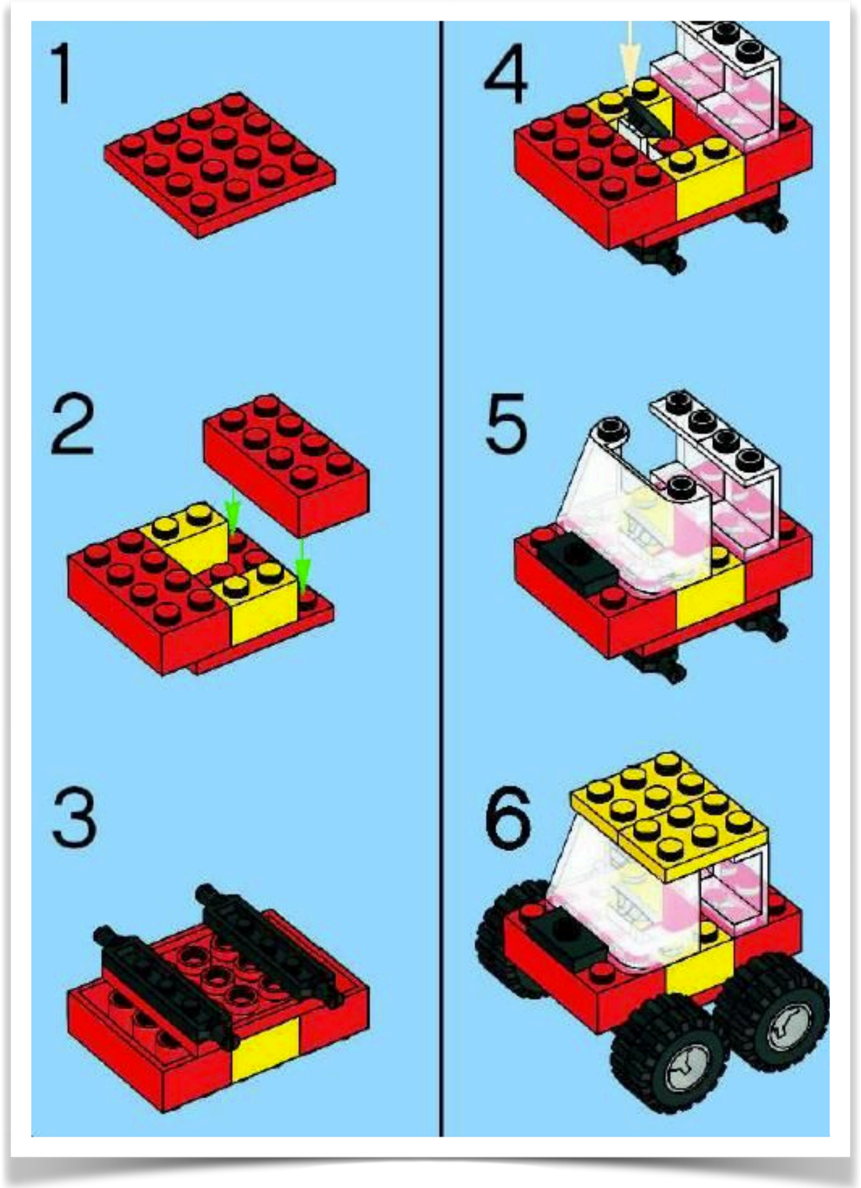
- We studied two toy implementations of persistent pairs:
    - a durable pair - using a **redo log**
        - log values before writing
        - recovery finishes the job if the write crashes after logging
    - a buffered durable pair - using a **checkpoint mechanism**
        - Persist the pair at every **sync**
          (without **sync**- only volatile memory is used!)
        - recovery resets to the latest checkpoint
- We demonstrated the library correctness condition w.r.t. the corresponding specification

⇒ Client programs using pairs may assume the simple specification

durable pair

buffered durable pair

Calling policies

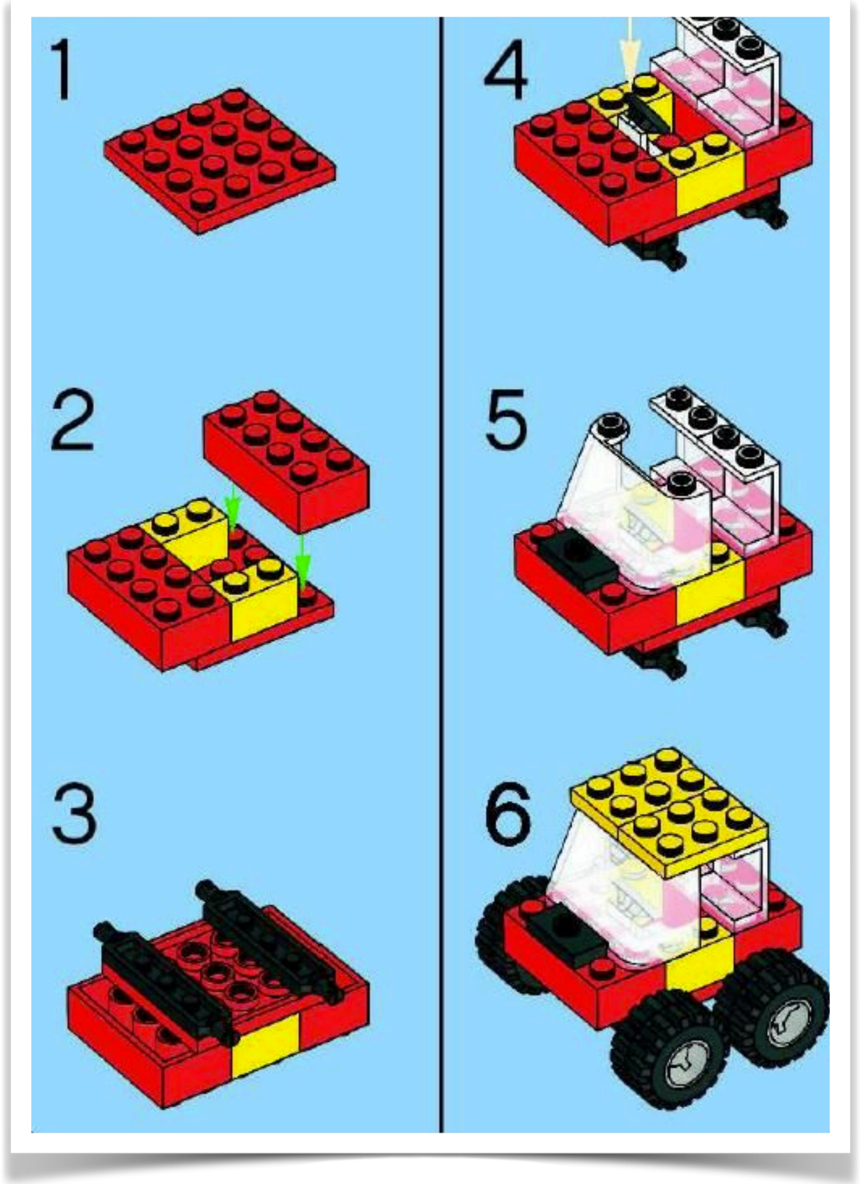


- Many libraries require a "calling policy", e.g.
  - **recovery** after crash
  - **single** producer
  - consume **non-empty** collections

If Histories($\mathbf{MGC}_{policy}$[L]) $\subseteq$ Histories($\mathbf{MGC}_{policy}$[$L^{\#}$]),
then for every client program **C** that adheres to policy**:**

Behaviors(**C**[L]) $\subseteq$ Behaviors(**C**[$L^{\#}$])

Calling policies



- Many libraries require a “calling policy”, e.g.
  - **recovery** after crash
  - **single** producer
  - consume **non-empty** collections

If Histories($\mathbf{MGC}_{policy}$[L]) ⊆ Histories($\mathbf{MGC}_{policy}$[$L^\#$]),
then for every client program **C** that adheres to policy**:**

Behaviors(**C**[L]) ⊆ Behaviors(**C**[$L^\#$])

To show that **C** adheres to **policy**, should we use L or $L^\#$?

If Histories($\mathbf{MGC}_{\text{policy}}$[L]) $\subseteq$ Histories($\mathbf{MGC}_{\text{policy}}$[L$^{\#}$]),
then for every client program **C** that adheres to policy:

$$\text{Behaviors}(\mathbf{C}[L]) \subseteq \text{Behaviors}(\mathbf{C}[L^{\#}])$$

To show that C adheres to policy, should we use L or L$^{\#}$?

- We prove that using L$^{\#}$ works
- **Abstraction theorem can be applied without any knowledge of L!**
- Seemingly circular reasoning is resolved by looking at minimal violations

Conclusion

- A **correctness condition** for libraries under NVM
- Formally related to **contextual refinement**
- Assume **PSC** semantics (x86-TSO is future work)
- Novel specification constructs: **persist { … }**, **LSFENCE(…)**
- Support **calling policies**
- Can encode (strict) (buffered) durable linearizability

I am hiring!
http://www.cs.tau.ac.il/~orilahav/

Thank you!