Abstraction for Crash-Resilient Objects

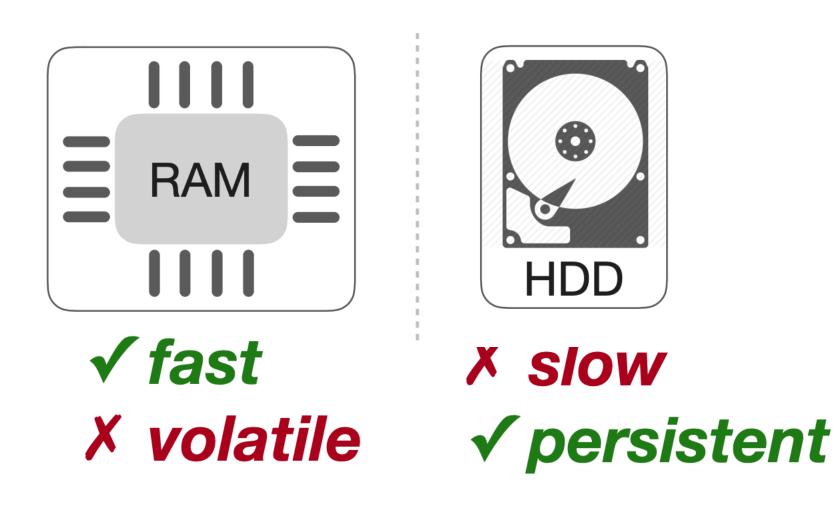
Artem Khyzha Arm Ltd

arm

Ori Lahav
Tel Aviv University



Non-volatile memory



- 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 execution continues

X = 1 and Y = 0 and Y = 0 writes may persist

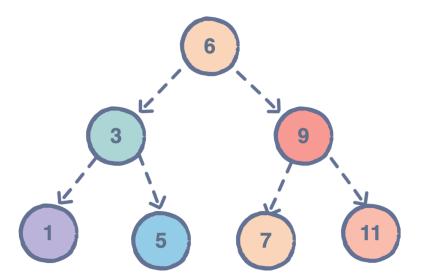
X = 0 and Y = 1 out of order
```

Explicit persist instructions have to be properly placed:

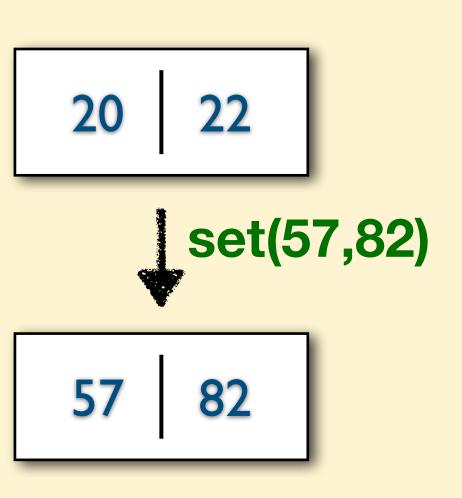
CLFLUSH, CLWB, CLFLUSHOPT, SFENCE

Persistent objects

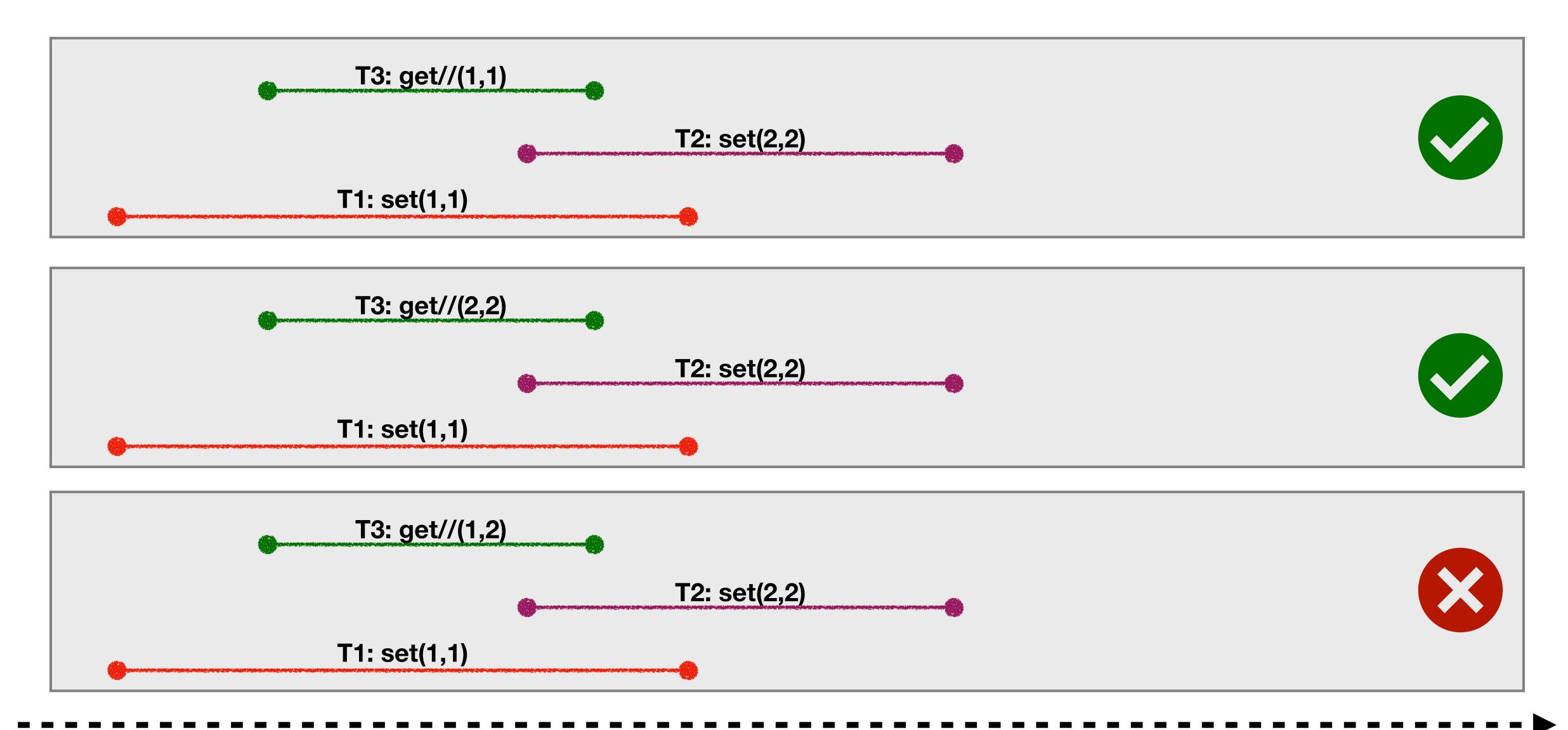
- Concurrent objects able to recover from crashes
 - Maps, queues, stacks...



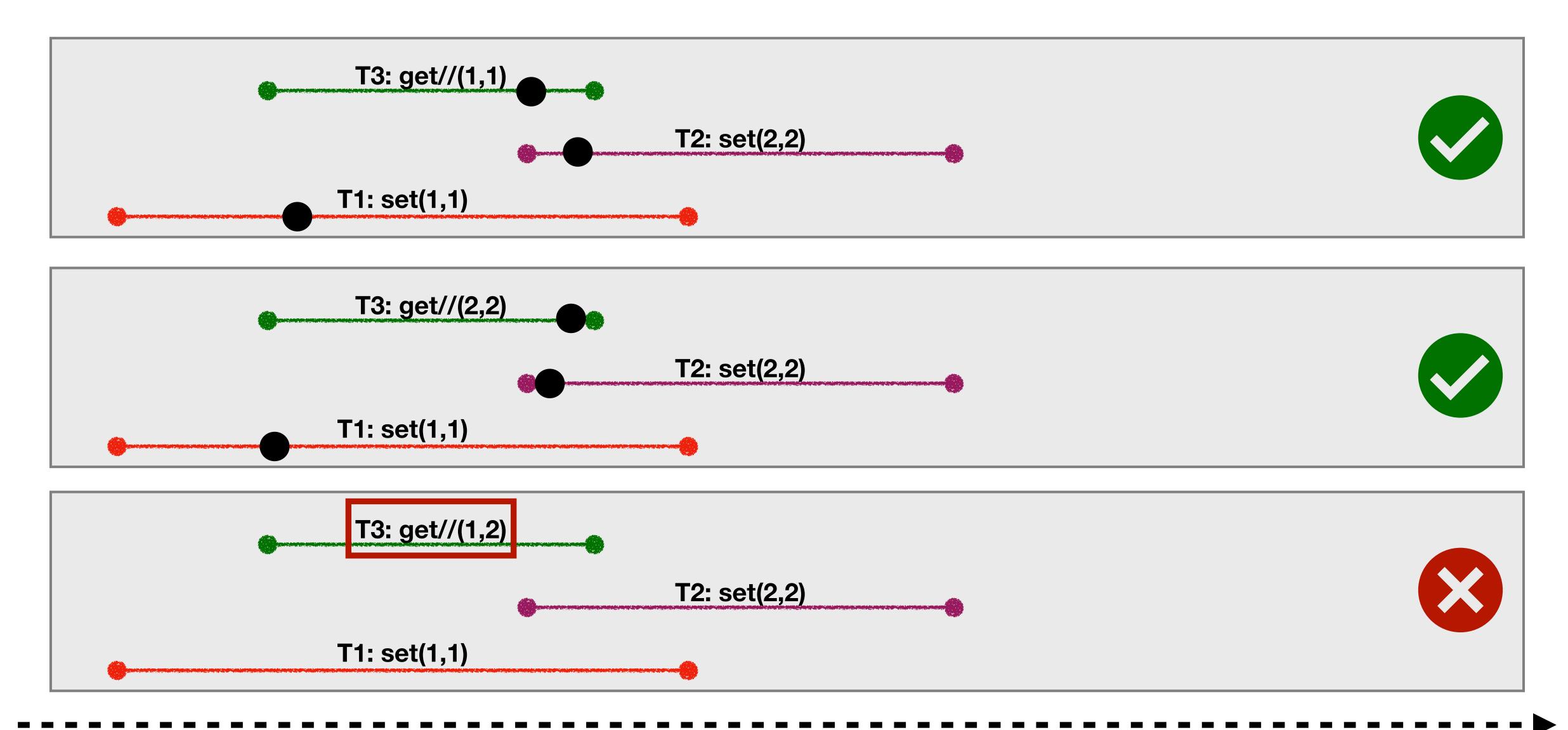
- Persistent pair simple persistent object supporting operations:
 - set(a,b): atomically write a pair of values
 - get: atomically read a pair of values
 - recover: fix pair after crash
 - sync: make sure previous writes to the pair persist



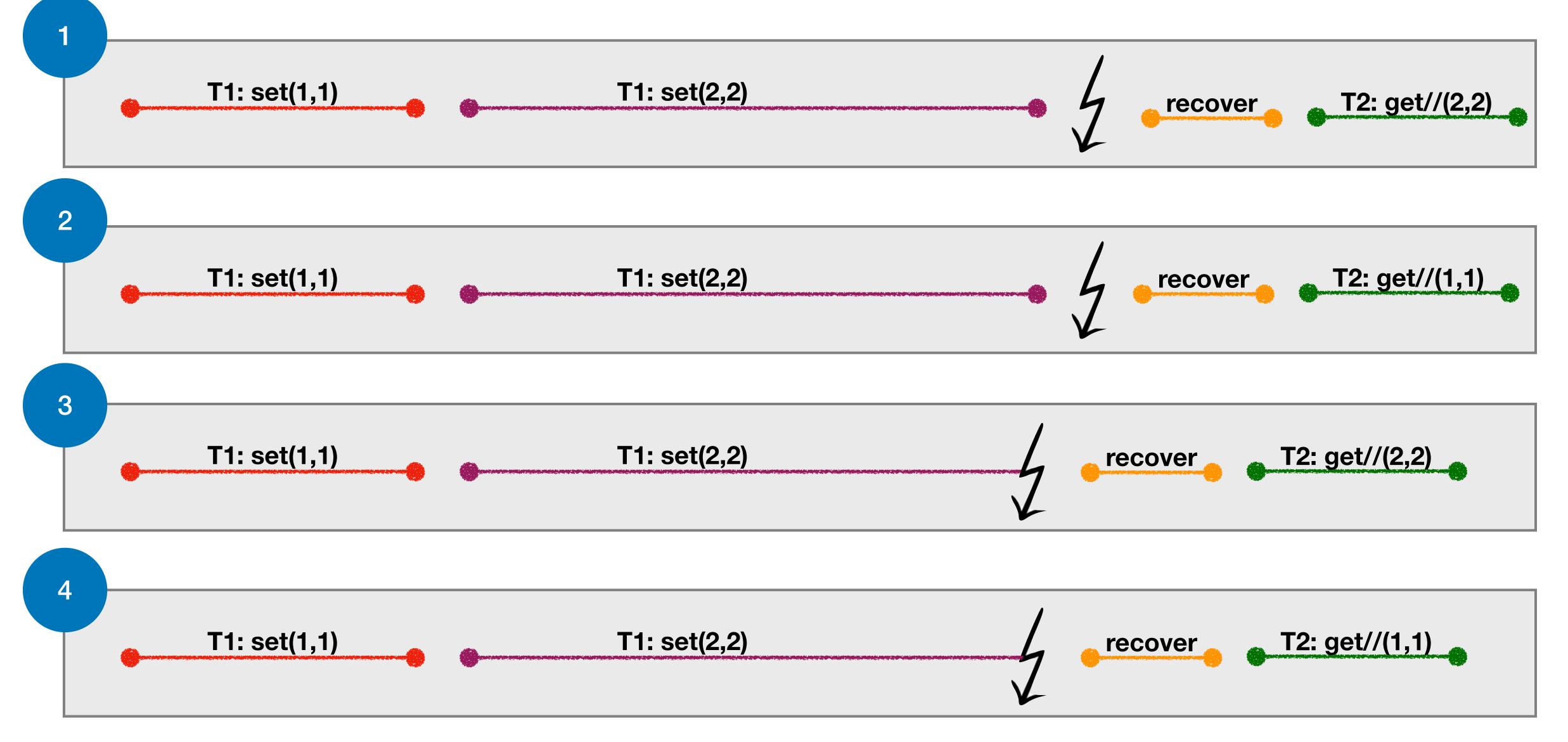
Correctness



Correctness

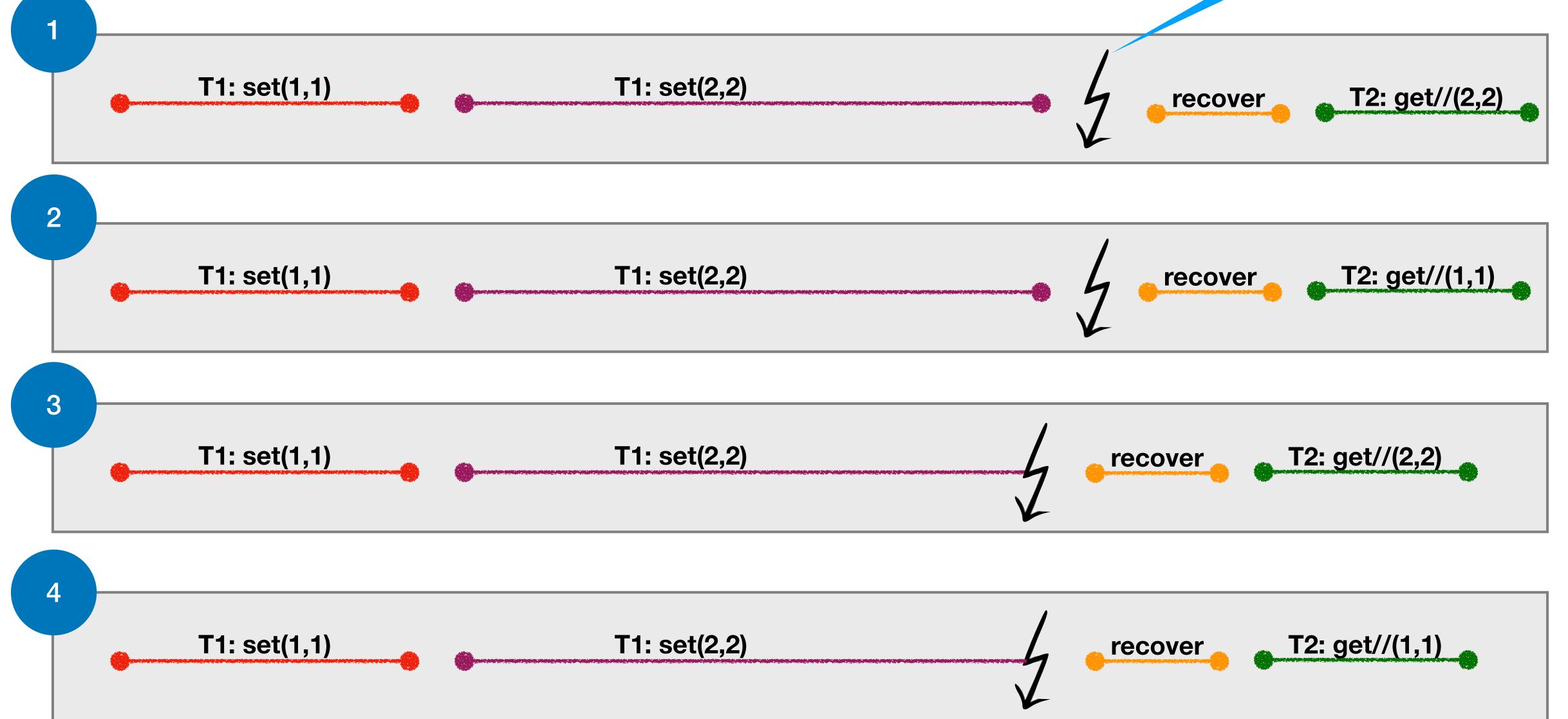


Correctness with crash 1/2

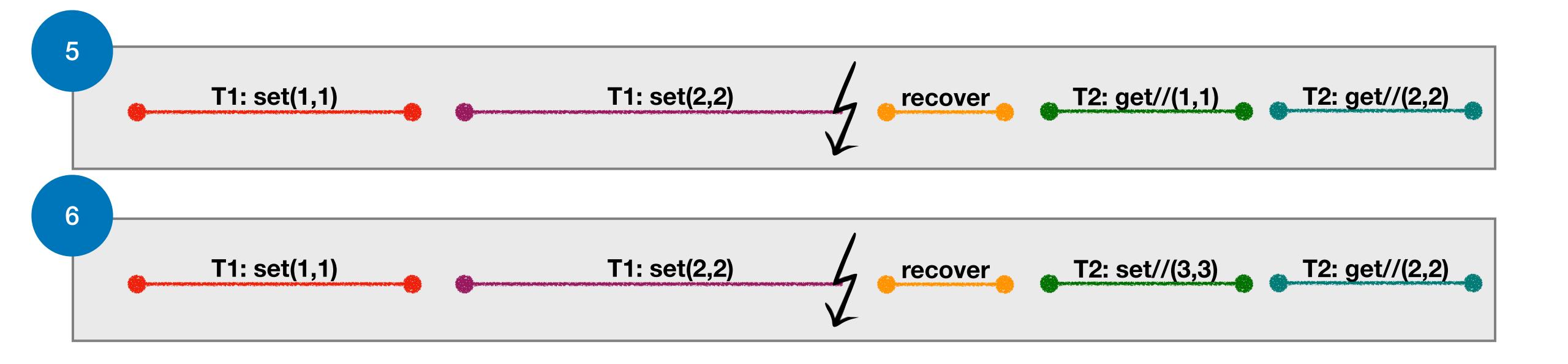


Correctness with crash

System crash



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: Concurrency, correctness, Larch, linearizability, multi-processing, 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

A concurrent object is a data object shared by concurrent proc condition for concurrent objects that exploits the semantics of degree of concurrency, yet it permits programmers to specify using known techniques from the sequential domain. Lineariza operation applied by concurrent processes takes effect instan invocation and its response, implying that the meaning of a given by pre- and post-conditions. This paper defines linearizab conditions, presents and demonstrates a method for proving the shows how to reason about concurrent objects, given they are I

Categories and Subject Descriptors: D.1.3 [Programming Tec D.2.1 [Software Engineering]: Requirements/Specificati guages]: Language Constructs—abstract data types, concurred and structures; F.1.2 [Computation by Abstract Devices]: F.3.1 [Logics and Meanings of Programs]: Specifying Programs—pre- and post-conditions, specification techniques

General Terms: Theory, Verification

Additional Key Words and Phrases: Concurrency, correct processing, serializability, shared memory, specification

1. INTRODUCTION

1.1 Overview

Informally, a concurrent system consists of a coll that communicate through shared typed objects. message-passing architectures in which the share Strict Linearizability and the Power of Aborting

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

21 November 2003

Abstract—Linearizability is a popular way to define lying objects' operations are instantaneous, then the the concurrent behavior of shared objects. However, implementation remains linearizable when its underlinearizability allows operations that crash to take ef- lying objects are replaced with linearizable implefect at any time in the future. This can be disruptive mentations. This property allows to build complex to systems where crashes are externally visible. In linearizable objects from simpler ones in a modular

Strict linearizability

may not take effect (but if it does, it does before the is respected). abort). We show that with abortable operations, there consensus and hence of any object.

unat it is impossible to obtain a strictly-inhearizable write(v), with the usual semantics. If a process p wait-free implementation of objects as simple as invokes write(v) and does not crash, then weak limmulti-reader registers from single-reader ones. To ited effect guarantees that the write can take effect address this problem, we augment our shared ob- only until the time p returns from the write's invocajects by allowing them to abort their operations in tion. This is in contrast to, for example, sequential the presence of concurrency. An aborted operation consistency [8], in which the write can take effect at behaves like an operation that crashes: it may or any arbitrary time in the future (as long as local order

Limited effect is an important property, because it are strictly-linearizable wait-free implementations of prevents old operation instances from suddenly appearing mysteriously. For example, suppose that a client withdraws money from the bank in an automated teller machine. but the machine crashes dur-

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

sistency, and subsumes both existing and proposed instruction set architectures. Using the persistency model, we present an automated transform to convert any linearizable, nonblocking concurrent object into one that is also durably linearizable. We also present a design pattern, analogous to linearization points, for the construction of other, more optimized objects. Finally, we discuss generic optimizations that may improve performance while preserving both safety and liveness.

based on sequential crash-free specifications

Princ This i ARPA

A pre

Laborator

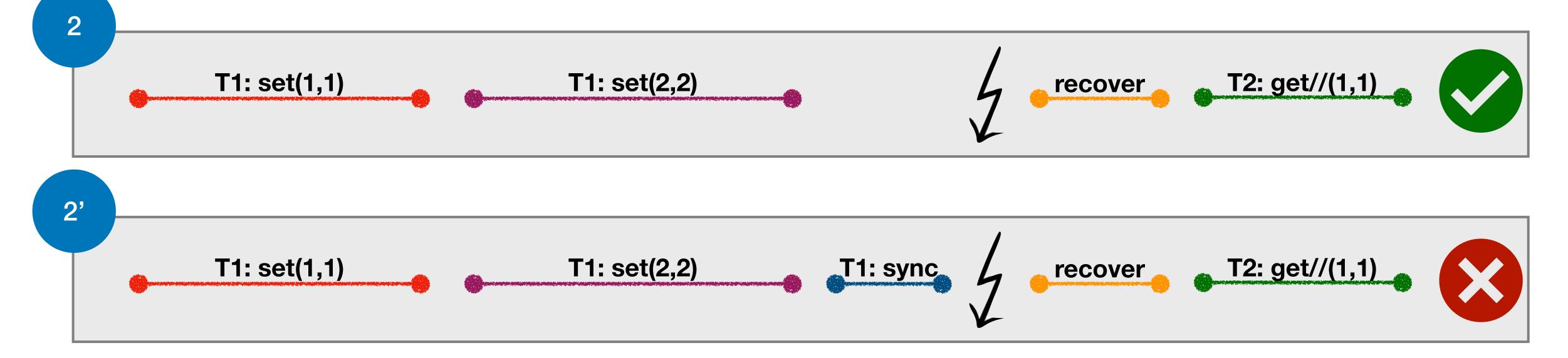
for J. M. Wing was provided in part by the National Science F The views and conclusions contained in this document are the interpreted as representing the official policies, either expresse

response. This simple requirement has many attractive features, from both a conceptual and a pragmatic

(NVM) will become commonplace over the next few years. While the availability of NVM suggests the possibility of keeping persistent data in main memory (not

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



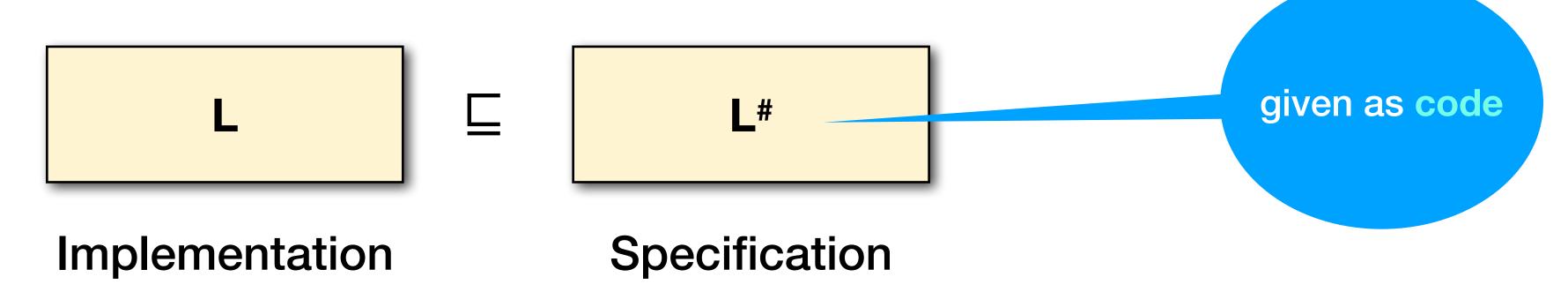
• We possibly want to mix-and-match correctness criteria in the same program

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

Behaviors(C[L]) ⊆ Behaviors(C[L#])

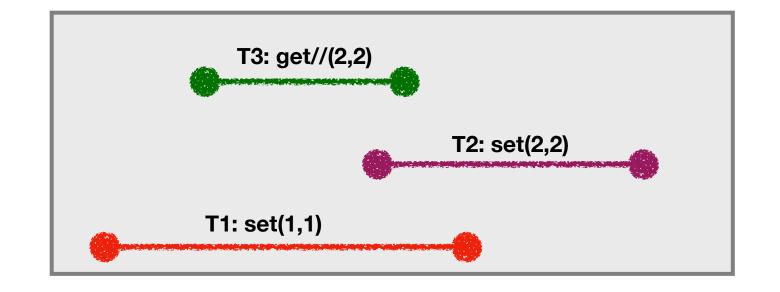
^{*} that uses disjoint memory & follows the calling policy

For volatile objects

If <u>????</u>, then for every* client program **C**Behaviors(**C**[L]) ⊆ Behaviors(**C**[L#])

Histories(MGC[L]) \subseteq Histories(MGC[L#])





- 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

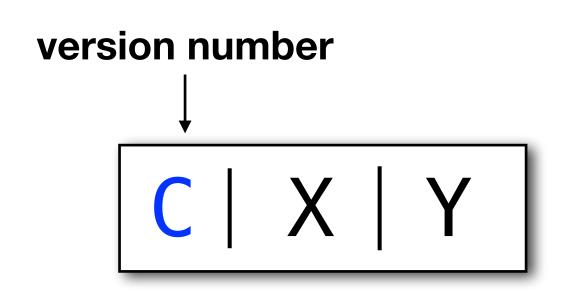
Specification construct

```
X | Y
```

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

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

Specification (L#)



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

```
get:
 BEGIN: s_1 := C;
 if odd(s<sub>1</sub>) then goto BEGIN
 a := X;
 b := Y;
 S_2 := C;
 if s_1 \neq s_2 then goto BEGIN
 return(a,b);
```

Implementation (L)

Application for a volatile pair

```
set(a,b):
                                                                                set(a,b):
                                                                                                        get:
                       get:
   lock();
                           BEGIN: s_1 := C;
                                                                                  atomic{
                                                                                                           atomic{
   C := C + 1;
                           if odd(s<sub>1</sub>) then goto BEGIN
                                                                                      X := a;
                                                                                                                 a := X;
                           a := X;
   X := a;
                           b := Y;
                                                                                      Y := b;
                                                                                                                 b := Y;
   Y := b;
                           S_2 := C;
   C := C + 1;
                           if s_1 \neq s_2 then goto BEGIN
   unlock();
                           return(a,b);
                                                                                                           return(a,b);
                                                                                  return;
   return;
```

- We have Histories(MGC[L]) ⊆ Histories(MGC[L#])
 - shown using a simulation argument
 - proof obligation for the library developer
- It follows that for every client program C:

≡ standard linearizability
when L# wraps a sequential
implementation in an atomic block

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

Back to NVM

Multiple formal models:

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

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

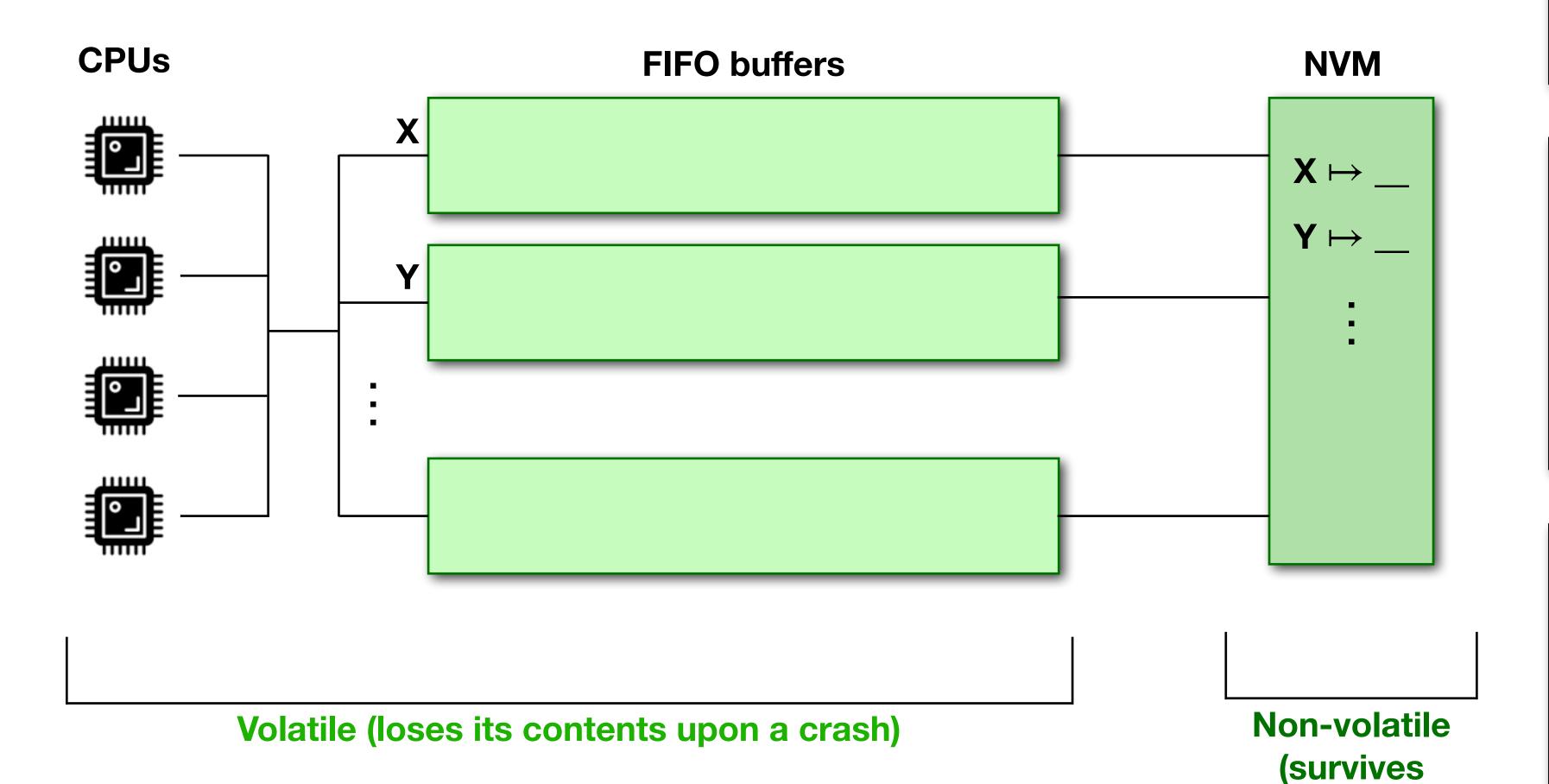
POPL'20

Taming x86-TSO Persistency

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

POPL'21

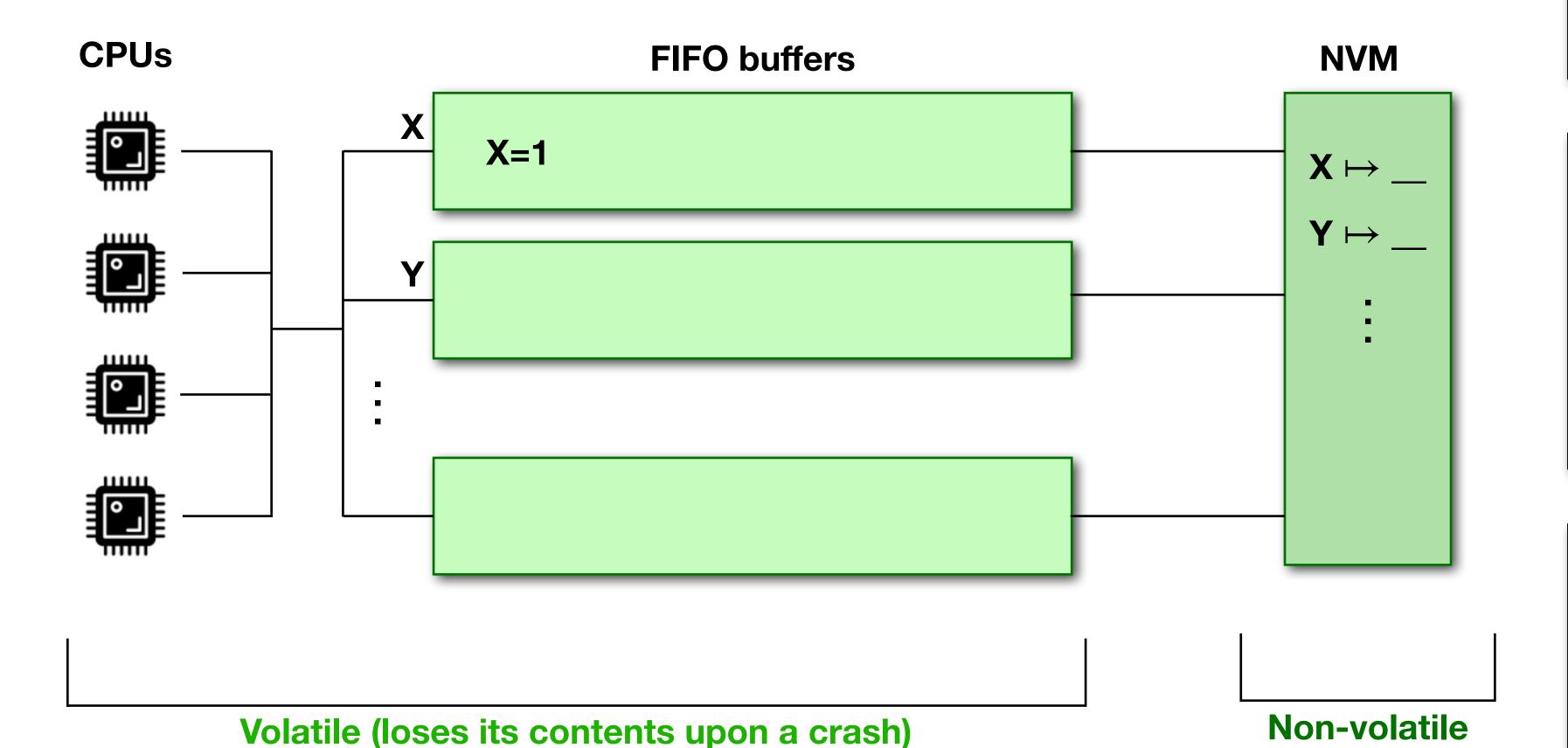
- We consider the simplest model: Persistent Sequential Consistency (PSC)
- can be mapped to x86-TSO (by adding appropriate MFENCE and SFNECE)



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

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

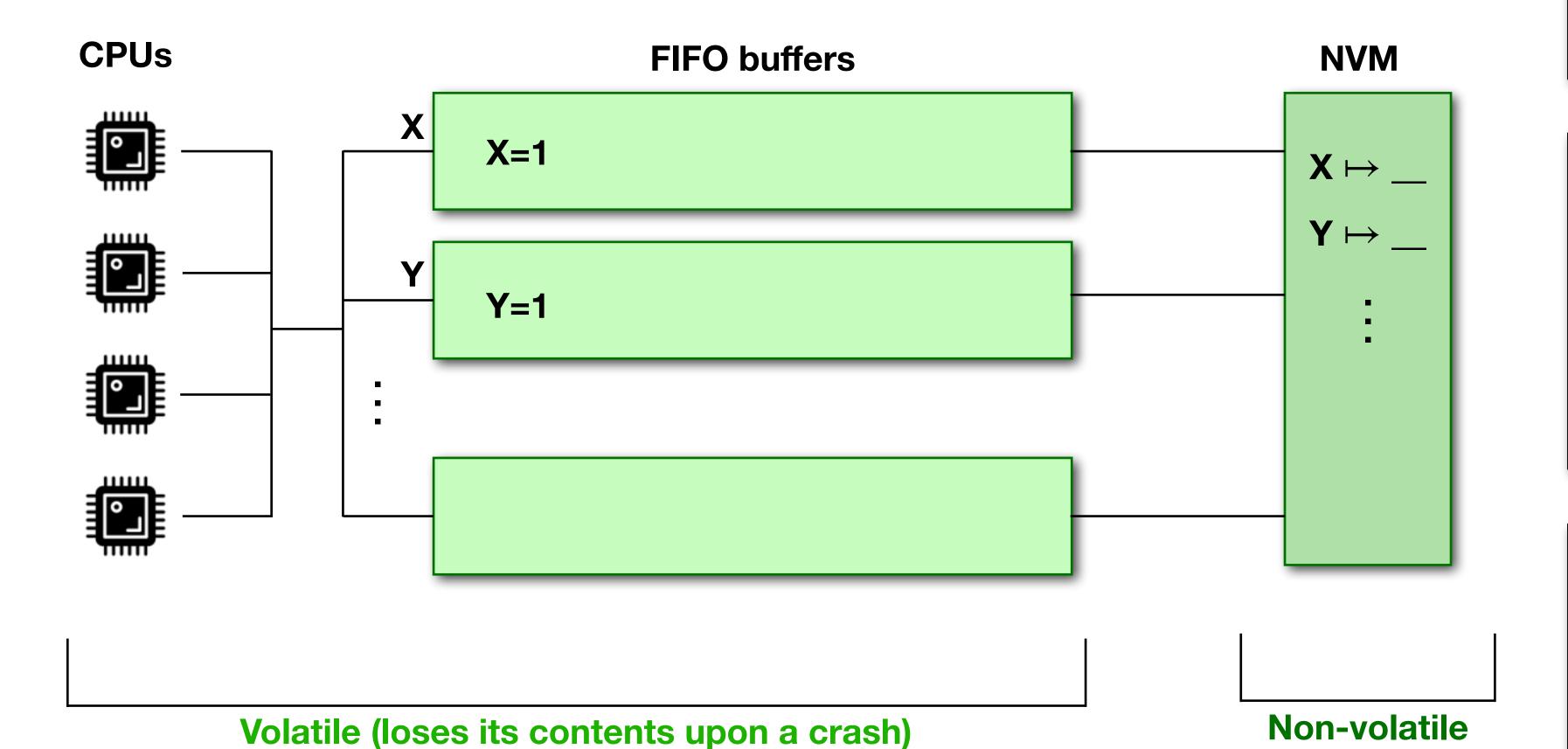
```
{ X = Y = 0 }
    X := 1;
    FLUSH-OPT(X);
    SFENCE;
    Y := 1;
    444
```



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

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

(survives

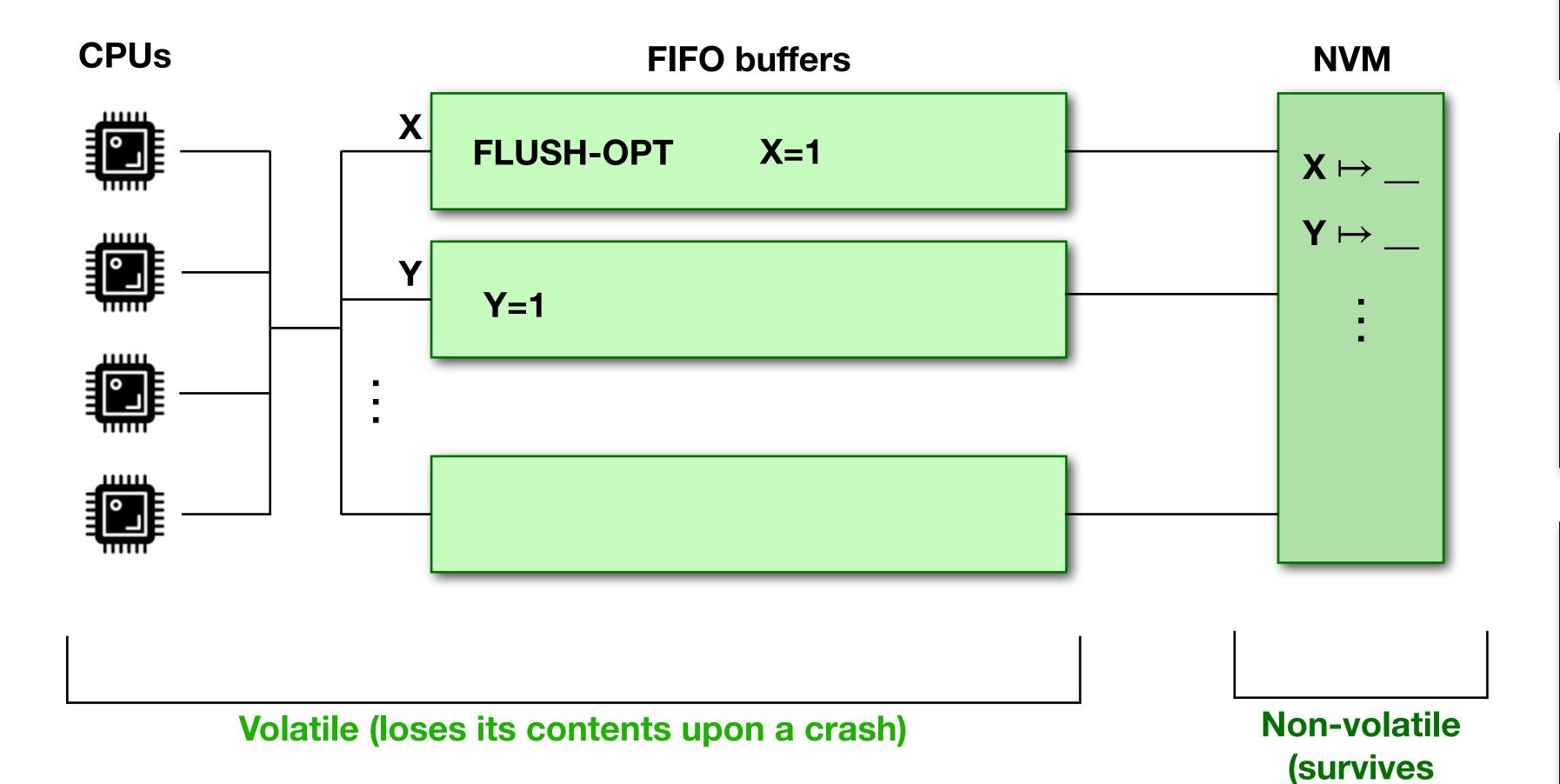


```
{ X = Y = 0 }
X := 1;
Y := 1;
```

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

```
{ X = Y = 0 }
    X := 1;
    FLUSH-OPT(X);
    SFENCE;
    Y := 1;
    444
```

(survives



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

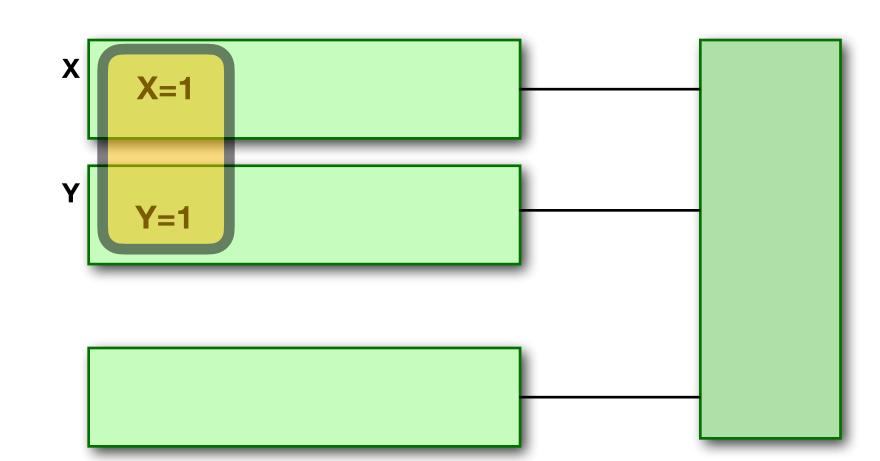
```
{ X = Y = 0 }
X := 1;
FLUSH(X);
Y := 1;
```

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



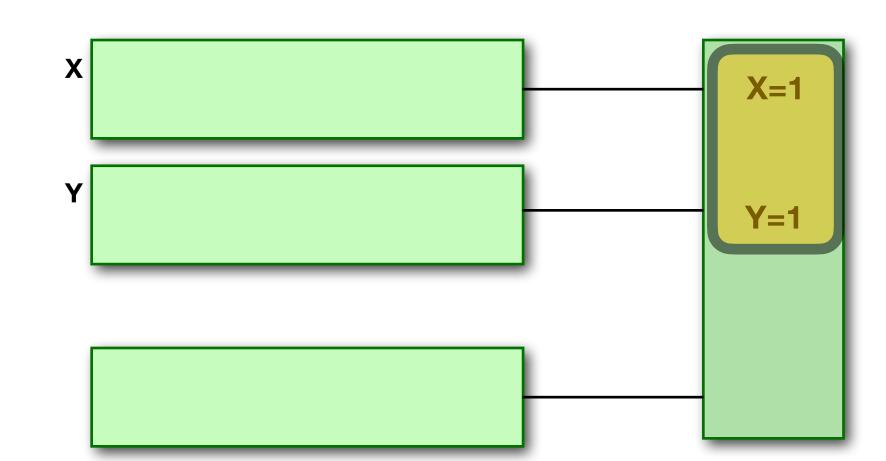
```
persist {
    X := 1;
    Y := 1;
}
```

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

```
<u>set(a,b):</u>
  atomic{
    persist{
      X = a;
       Y = b;
    FLUSH(X);
 return;
```

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

```
sync:
  return;
                durable
                  pair
recover:
  return;
```

Specification #2 of a persistent pair

```
<u>set(a,b):</u>
  atomic{
     persist{
      X = a;
       Y = b;
     FLUSH(X);
 return;
```

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

```
sync:
 FLUSH(X);
 return;
                buffered
                durable
                  pair
recover:
  return;
```

Library abstraction theorem

```
If <u>????</u>, then for every* client program C
behaviors(C[L]) ⊆ behaviors(C[L#])
```

Candidate library correctness criterion:

Histories(MGC[L]) \subseteq Histories(MGC[L#])

Challenge in NVM

<u>Implementation</u>

Specification

```
foo:
return;
```

```
?
⊑
```

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

✓ Histories(MGC[L]) ⊆ Histories(MGC[L#])

```
{ A = 0 }
A := 1;
FLUSH-OPT(A);
foo();
444
{ 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; 
□ LSFENCE(X);
return;
```

```
{ A = 0 }
A := 1;
FLUSH-OPT(A);
foo();
444
{ A = ??}
```

• LSFENCE effect by library is confined to library code

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

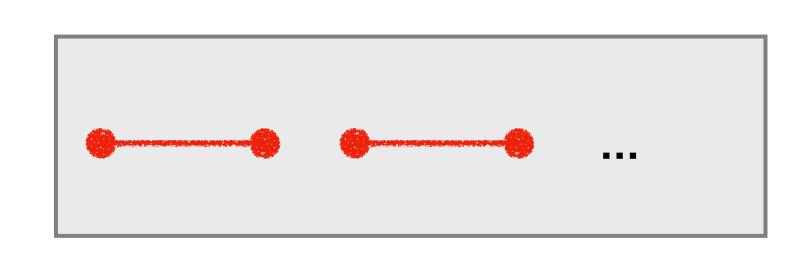
Solution #2

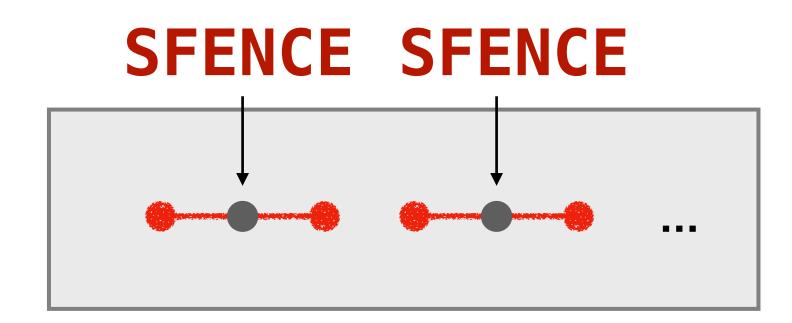
Make histories more expressive

```
foo:
return;

foo:
specific foo:
specific foo:
return;
```

Histories(MGC[L]) ⊈ Histories(MGC[L#])





Library abstraction theorem

include CALL, RETURN, and SFENCE transitions

If Histories(MGC[L]) ⊆ Histories(MGC[L#]),

then for every* client program C:

Behaviors(C[L]) ⊆ Behaviors(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:

```
Histories(MGC[L<sub>1</sub>]) \subseteq Histories(MGC[L<sub>1</sub>#])
Histories(MGC[L<sub>2</sub>]) \subseteq Histories(MGC[L<sub>2</sub>#])
Histories(MGC[L<sub>1</sub> \uplus L<sub>2</sub>]) \subseteq Histories(MGC[L<sub>1</sub># \uplus L<sub>2</sub>#])
```

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

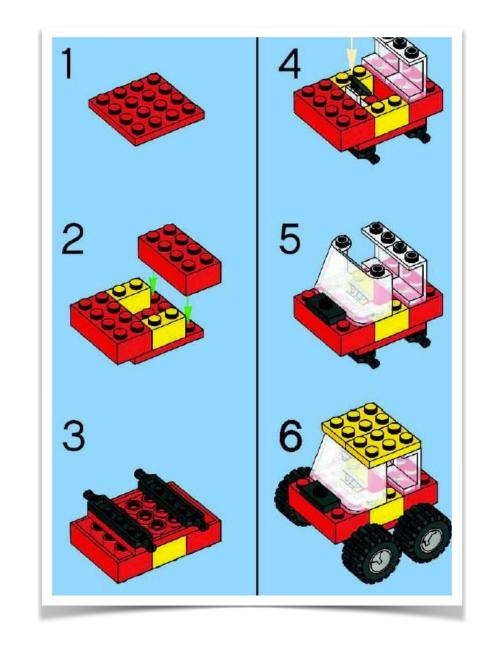
buffered durable pair

Calling policies

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

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

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

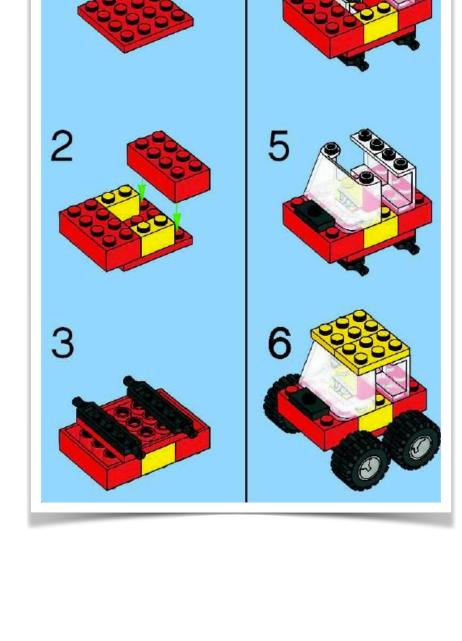


Calling policies

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

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

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



To show that C adheres to policy, should we use L or L#?

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

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

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