Abstraction for Crash-Resilient Objects

Artem Khyzha
Arm Ltd

Ori Lahav
Tel Aviv University

ESOP 2022
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

Explicit persist instructions have to be properly placed:

- CLFLUSH
- CLWB
- CLFLUSHOPT
- SFENCE

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

![Diagram showing set(57,82)]
Correctness

T1: set(1,1)
T2: set(2,2)
T3: get//(1,1)

T1: set(1,1)
T2: set(2,2)
T3: get//(2,2)

T1: set(1,1)
T2: set(2,2)
T3: get//(1,2)

Time
Correctness with crash 1/2

1. T1: set(1,1) → T1: set(2,2) → recover → T2: get((2,2))
2. T1: set(1,1) → T1: set(2,2) → recover → T2: get((1,1))
3. T1: set(1,1) → T1: set(2,2) → recover → T2: get((2,2))
4. T1: set(1,1) → T1: set(2,2) → recover → T2: get((1,1))
Correctness with crash 1/2

1. T1: set(1,1) → T1: set(2,2) → T2: get(2,2) → recover

2. T1: set(1,1) → T1: set(2,2) → T2: get(1,1) → recover

3. T1: set(1,1) → T1: set(2,2) → T2: get(2,2) → recover

4. T1: set(1,1) → T1: set(2,2) → T2: get(1,1) → recover

System crash
Correctness with crash 2/2

Diagram showing the operations of T1 and T2, including set and get actions, with recovery indicated.
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 that they are linearizable.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrency; D.2.4 [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, multiprocessor, 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,
Linearizability: A Correctness Condition for Concurrent Objects

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

A concurrent object is a data object shared by concurrent processes. It can have linearizability conditions for concurrent objects that exploit the semantics of the degree of concurrency, and yet permit programmers to specify using known techniques from the sequential domain. Linearizability operations applied by concurrent processes takes effect instantaneously and its response, implying that the meaning of a given by pre- and post-conditions. This paper defines linearizability, presents and demonstrates a method for proving the shows how to reason about concurrent objects, given they are linearizable.

Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent and distributed systems

Additional Key Words and Phrases: Concurrency, correctness, linearizability, sequential, specification

1. INTRODUCTION

1.1 Overview

Informally, a concurrent system consists of a collection of processes that communicate through shared typed objects. A message-passing architecture in which the shared

Strict Linearizability and the Power of Aborting

Marcos K. Aguilera1 and Sven Frehland1
HP Labs, Palo Alto, CA 94304
21 November 2003

Abstract—Linearizability is a popular way to define the concurrent behavior of shared objects. However, linearizability allows operations that crash to take effect at any time in the future. This can be disruptive to systems where crashes are externally visible. If the aborting objects’ operations are instantaneous, then the implementation remains linearizable when its underlying objects are replaced with linearizable implementations. This property allows to build complex linearizable objects from simpler ones in a modular way.

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

Joseph Izrailevitz2, Hammoudi Mendes, and Michael L. Scott
University of Rochester, Rochester, NY 14627-0226, USA
{jhi1, hmendes, scott}@cs.rochester.edu

Durable linearizability

Strict 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
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:

\[
\text{Implementation} \subseteq \text{Specification}
\]

given as code

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

- Include special constructs in the language for intuitive specifications
Goal: Library abstraction theorem

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 \( ????, \) then for every* client program \( C \)

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

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

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

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

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

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

Specification construct

Specification (L#)
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

- 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:
  
  Behaviors(C[L]) ⊆ Behaviors(C[L#])
• We consider the simplest model: **Persistent Sequential Consistency (PSC)**

• can be mapped to **x86-TSO** (by adding appropriate **MFENCE** and **SFENCE**)
The PSC model

\begin{align*}
\{ X = Y = 0 \} \\
X := 1; \\
Y := 1;
\end{align*}

↯↯↯

\begin{align*}
\{ X = Y = 0 \} \\
X := 1; \\
FLUSH(X); \\
Y := 1;
\end{align*}

↯↯↯

\begin{align*}
\{ X = Y = 0 \} \\
X := 1; \\
FLUSH\text{-}OPT(X); \\
SFENCE; \\
Y := 1;
\end{align*}

↯↯↯

- Volatile (loses its contents upon a crash)
- Non-volatile (survives crashes)
The PSC model

{ \( X = Y = 0 \) }
\[ X := 1; \]
\[ Y := 1; \]

↯↯↯

{ \( X = Y = 0 \) }
\[ {X := 1}; \]
\[ \text{FLUSH}(X); \]
\[ Y := 1; \]

↯↯↯

{ \( X = Y = 0 \) }
\[ X := 1; \]
\[ \text{FLUSH-OPT}(X); \]
\[ \text{SFENCE}; \]
\[ Y := 1; \]

↯↯↯

Volatile (loses its contents upon a crash)

Non-volatile (survives crashes)

FIFO buffers

CPU

NVM

CPUs

X

Y

X=1

NVM
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;  
↯↯↯

CPUs

FIFO buffers

X = 1

Y = 1

NVM

X ↦ __
Y ↦ __
…

Volatile (loses its contents upon a crash)

Non-volatile (survives crashes)

CPU

Volatile

Non-volatile

FIFO buffers

NVM
The PSC model

\[
\begin{align*}
\{ & X = Y = 0 \} \\
X & := 1; \\
Y & := 1; \\
\end{align*}
\]

\[\vdash \vdash \vdash\]

\[
\begin{align*}
\{ & X = Y = 0 \} \\
X & := 1; \\
\text{FLUSH}(X); \\
Y & := 1; \\
\end{align*}
\]

\[\vdash \vdash \vdash\]

\[
\begin{align*}
\{ & X = Y = 0 \} \\
X & := 1; \\
\text{FLUSH-OPT}(X); \\
\text{SFENCE}; \\
Y & := 1; \\
\end{align*}
\]

\[\vdash \vdash \vdash\]

CPUs

FIFO buffers

NVM

Volatile (loses its contents upon a crash)

Non-volatile (survives crashes)
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
Persistent blocks

- We introduce a construct to control persistency:

  \[
  \text{persist \{ ... \}}
  \]
  
  - All writes persist \textit{simultaneously}
  
  - \textbf{No earlier} than the block ends
  
  - May be forced by a \texttt{FLUSH} instruction
 Specification #1 of a persistent pair

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

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

cancel:
  return;

durable pair
```

T1: set(1,1)  T2: get/(1,1)  recover
Specification #2 of a persistent pair

\[ \text{set}(a,b): \]
\[
\text{atomic}\{ \\
\text{persist}\{ \\
\quad X = a; \\
\quad Y = b; \\
\}\text{FLUSH}(X); \\
\}\text{return};
\]

\[ \text{get}: \]
\[
\text{atomic}\{ \\
\quad a = X; \\
\quad b = Y; \\
\}\text{return}(a,b);
\]

\[ \text{sync}: \]
\[
\text{FLUSH}(X); \\
\text{return};
\]

\[ \text{recover}: \]
\[
\text{return};
\]

T1: \text{set}(1,1) \quad T1: \text{set}(2,2) \quad \text{recover} \quad T2: \text{get}(/(1,1))
Library abstraction theorem

If [???], then for every client program $C$

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

Candidate library correctness criterion:

$\text{Histories}(\text{MGC}[L]) \subseteq \text{Histories}(\text{MGC}[L#])$
Challenge in NVM

Implementation

```plaintext
foo:
    return;
```

Specification

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

BROKEN!

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

\[ \text{Histories(MGC[L])} \subseteq \text{Histories(MGC[L#])} \]
Solution #1

• Introduce a “local SFENCE instruction”

```c
foo:
    return;
```

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

{ A = 0 }
A := 1;
FLUSH-OPT(A);
foo();
-conscious
{ 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

\[
\text{foo: return;}
\]

\[
\text{foo: SFENCE; return;}
\]

Histories(MGC[L]) \notin Histories(MGC[L#])
Library abstraction theorem

include CALL, RETURN, and SFENCE transitions

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

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

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:

\[
\begin{align*}
\text{Histories}(\text{MGC}[L_1]) & \subseteq \text{Histories}(\text{MGC}[L_1\#]) \\
\text{Histories}(\text{MGC}[L_2]) & \subseteq \text{Histories}(\text{MGC}[L_2\#]) \\
\text{Histories}(\text{MGC}[L_1 \uplus L_2]) & \subseteq \text{Histories}(\text{MGC}[L_1\# \uplus L_2\#])
\end{align*}
\]
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
Calling policies

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

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

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

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

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

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

To show that \(C\) adheres to policy, should we use \(L\) or \(L\#\)?
If \( \text{Histories}(MGC_{\text{policy}}[L]) \subseteq \text{Histories}(MGC_{\text{policy}}[L^\#]) \), then for every client program \( C \) that adheres to policy:

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

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

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

Thank you!