Durability Semantics for Lock-based Multithreaded Programs

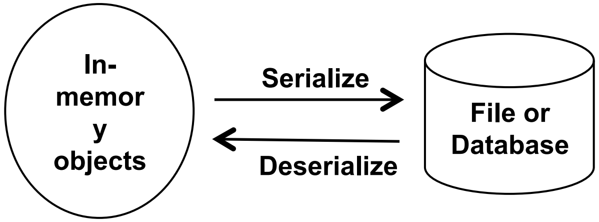
Dhruva R. Chakrabarti, Hans-J. Boehm

Hewlett-Packard Laboratories

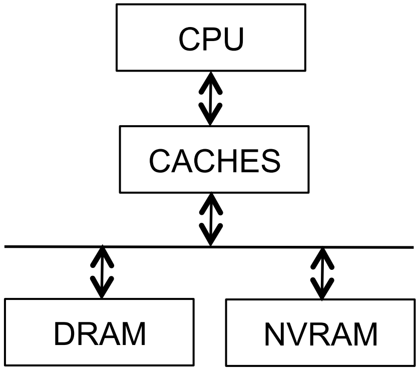
HotPar'13

Do we need a separate durable data representation?

- Conventional durability techniques
  - Separate object and persistent formats
  - Translation code
  - Programmability and performance issues

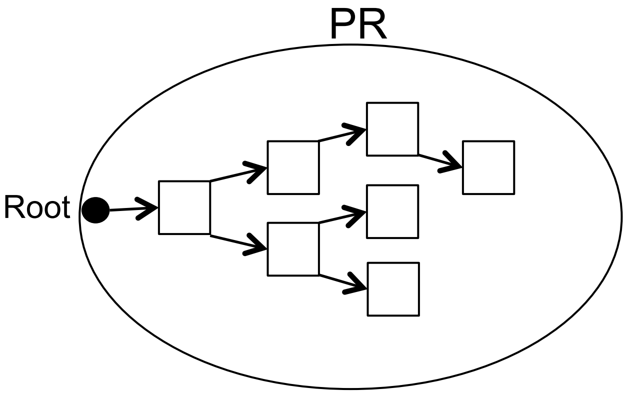


- In-memory durability
  - Enabled by non-volatile memory or NVRAM (such as memristors, PCM, etc.)
  - In-memory objects are durable throughout
  - Byte-addressability simplifies programmability
  - Low load/store latencies offer high performance



Programming Model

Use persistent regions (PR) instead of flat files

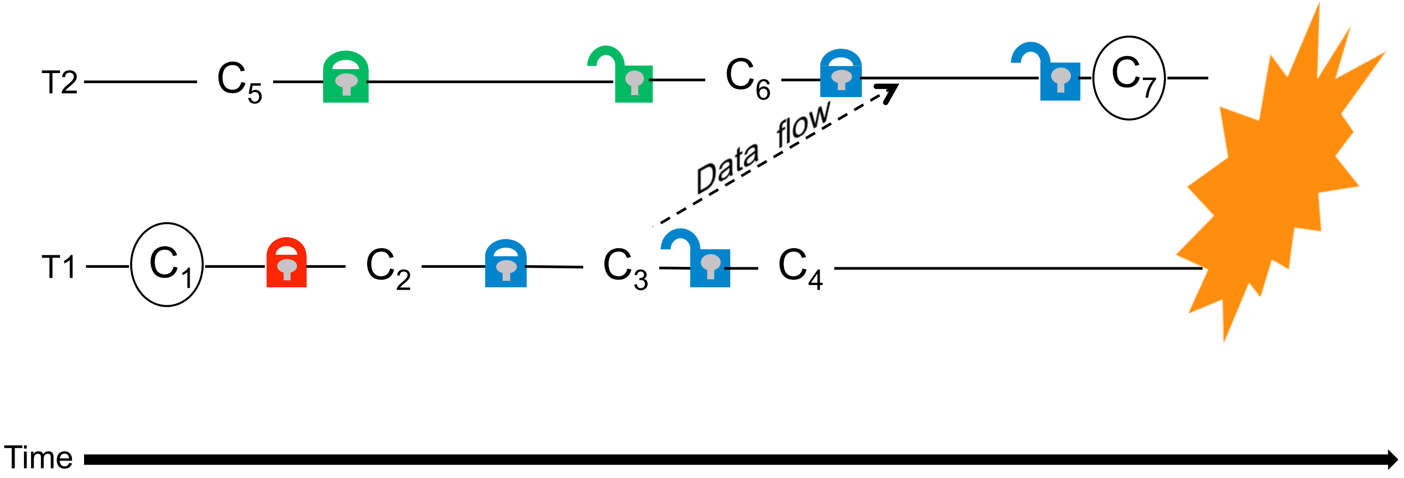


Data not in a PR is considered transient

```
pr = find_or_create_persistent_region(name);
persistent_data = get_root_pointer(pr);
if (persistent_data) {
        // restart code
}
else {
        // initialize persistent_data
}
// use persistent_data
```

Motivating Observations

- Reuse durable data structures after process termination
- Reusable data structures must be consistent across failures
  - Invariants must be preserved
- How are invariants identified in a lock-based multithreaded program?
  - No explicit association between a shared datum and the protecting lock
  - Lock acquires can be nested



Contributions

- Consistency semantics for durable data at intermediate program points
  - In spite of arbitrary lock nesting
  - Largely unchanged code
  - Relationship with transactional semantics
- Optimizations
- Initial idea of overheads

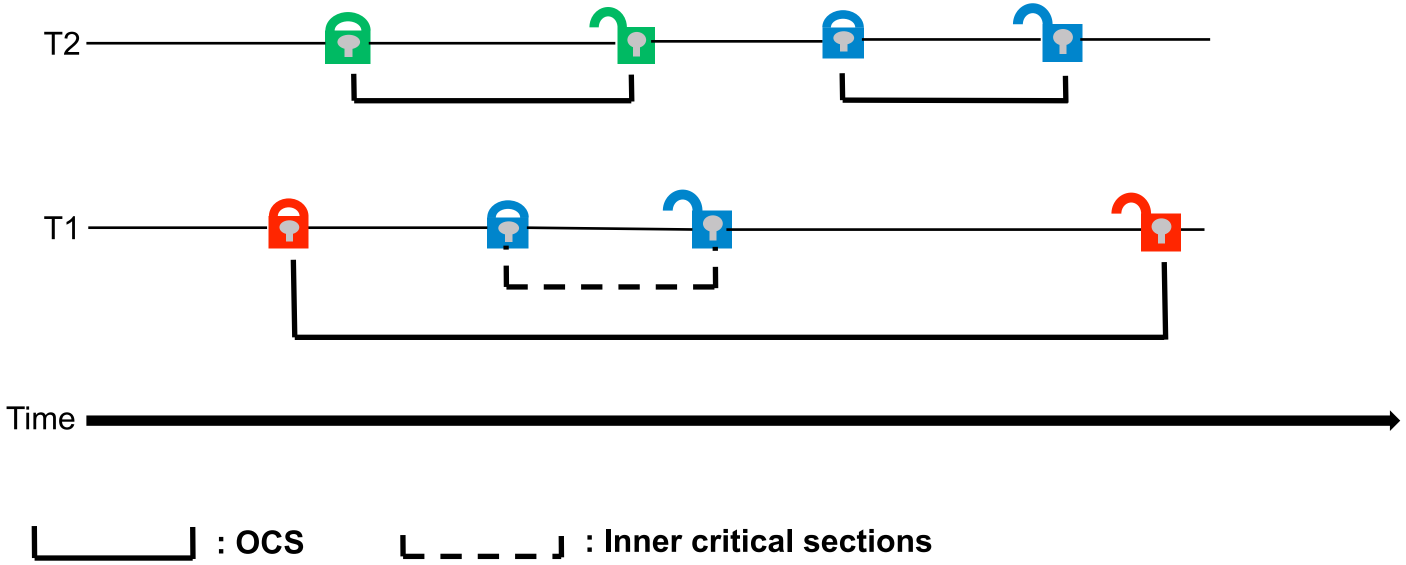
Notion of Consistent Program Points

Unlocked program points are thread-consistent

- Critical sections indicators of consistent states
  - If no locks are held, all data structures should be in a consistent state
- Some restrictions:
  - Client provided locks
  - Serial programs

Notion of Failure-Atomic Update Units

Outermost Critical Sections (OCS) are failure-atomic

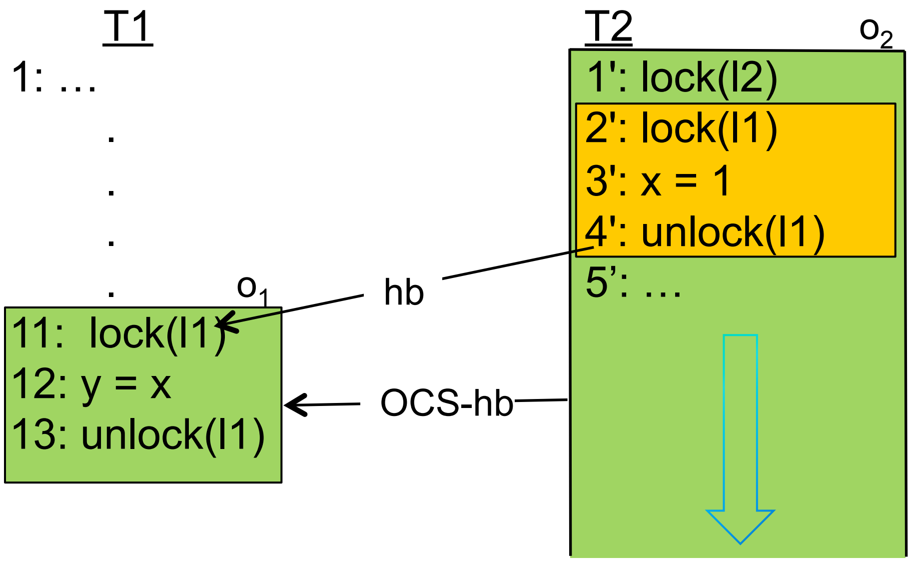


Notion of Durability-related Dependences among OCSes

A completed OCS may depend on an incomplete OCS

- Cause: Isolation and durability boundaries may not match
- Effect: The durable effects of a completed OCS may have to be undone
  - Happens only with nesting

x, y are persistent and initially x=y=0



T1

1: …

.

.

.

.

o_1

11: lock(l1)

12: y = x

13: unlock(l1)

hb

OCS-hb

T2

o_2

1': lock(l2)

2': lock(l1)

3': x = 1

4': unlock(l1)

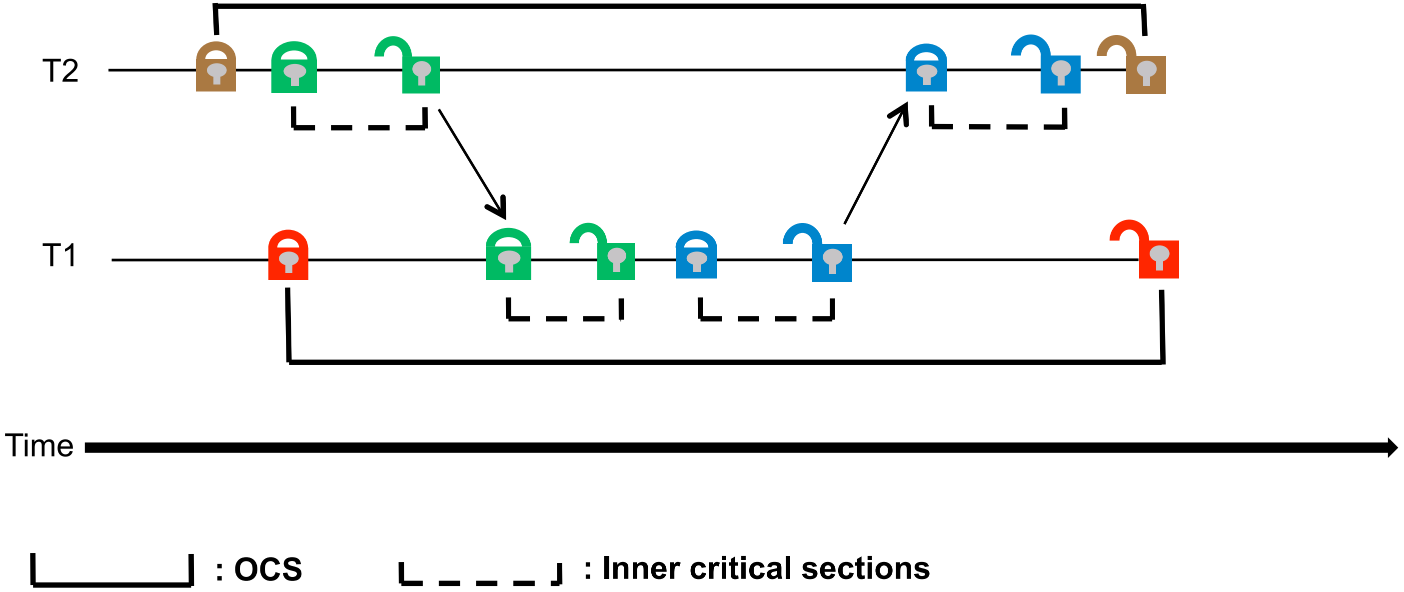
5': …

If the program crashes, can the effects of o_1 be made durable when those of o_2 are not?

NO! y=1, x=0 is not a consistent state.

OCS-hb relation may be cyclic

Inner critical sections can cause cyclic OCS-hb



All effects in the involved OCSes must appear to be visible in persistent memory at the same time

An Implementation Overview

- All lock operations and writes to persistent memory locations logged
- hb-relations between lock releases and acquires captured in the log
- Logs maintained in non-volatile memory
- Unnecessary log entries periodically pruned
- Some optimizations implemented
- Cache lines flushed at appropriate points

Some Preliminary Experimental Results

- NVRAM-based programs 2-3 orders of magnitude faster than disk-based ones
- But what’s the overhead of adding durability to transient data structures?

[1]Runtime comparison of 2 durable applications with the transient version as the baseline

Apps	**Slowdown** (num_threads=4)		**Statistics** (num_threads=4)		
	nvram	*nvram-nf*	*#OCSes*	*#stores*	*#logs*
Dedup	50%	33%	260K	900K	1.4M
Memcached	160%	60%	4M	22M	30M

nvram: durable version
nvram-nf: durable version without cache line flushes
#OCSes: Total number of OCSes encountered dynamically
#stores: Total number of dynamic store operations in NVRAM
#logs: total number of log entries created in NVRAM

Dedup: A deduplication kernel from the PARSEC benchmark suite. The hashtable maintaining unique key-value pairs of chunks of input stream is made durable.

Memcached: Starting with the original key-value cache implementation, the cache, LRU lists, and the slab allocator information are made durable.

---

[1] **DRAM used to simulate NVRAM on a RedHat Linux Intel Xeon x86-64 machine.**

Conclusions

- Presented a technique for identifying intermediate application-wide consistent states in a lock-based program

- NVRAM enables an efficient implementation

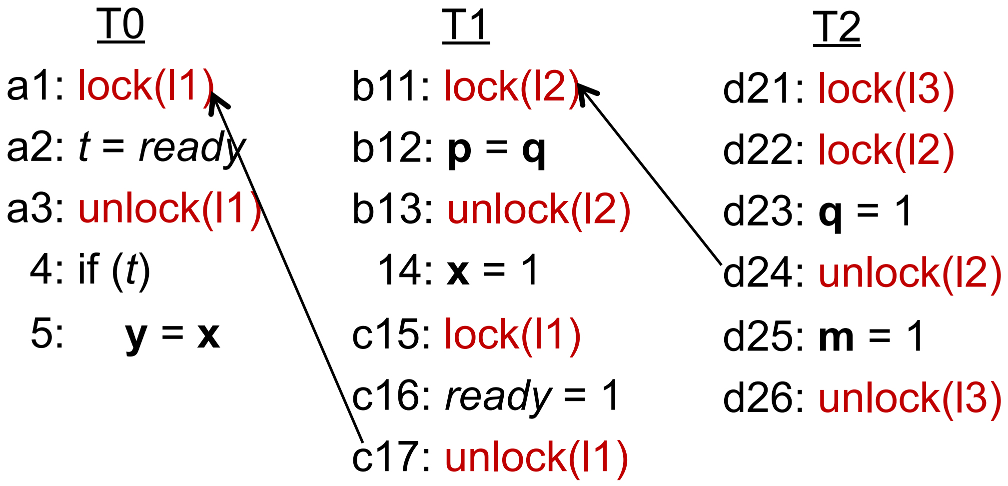
Backup

Optimizations/Pitfalls

- Is log elision applicable to durable updates outside an OCS?
- Is it mandatory to track every OCS-hb, specifically ones that involve OCSes with updates to transient locations alone?

Optimizing thread-consistent updates

Elide logging outside an OCS, if possible



T0

```
a1: lock(l1)
a2: t = ready
a3: unlock(l1)
  4: if (t)
  5:     y = x
```

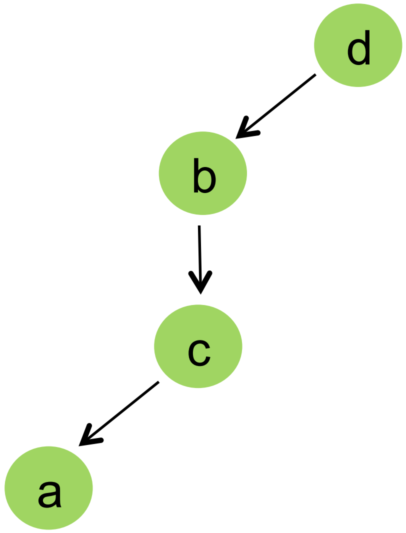
T1

```
b11: lock(l2)
b12: p = q
b13: unlock(l2)
  14: x = 1
c15: lock(l1)
c16: ready = 1
c17: unlock(l1)
```

T2

```
d21: lock(l3)
d22: lock(l2)
d23: q = 1
d24: unlock(l2)
d25: m = 1
d26: unlock(l3)
```

OCS level hb-relations

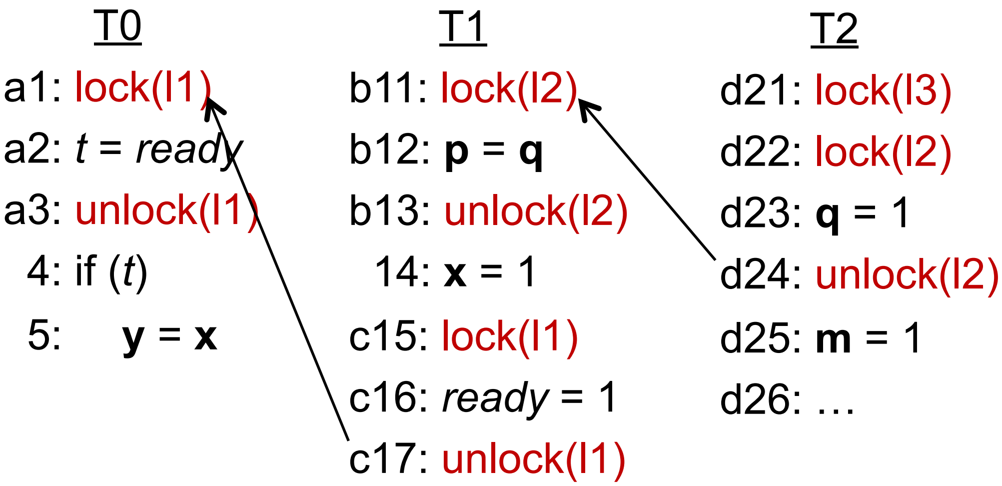


Elide logging of line 14 since OCS ‘b’ will not be undone.

Elide logging of line 5 since OCS ‘a’ will not be undone.

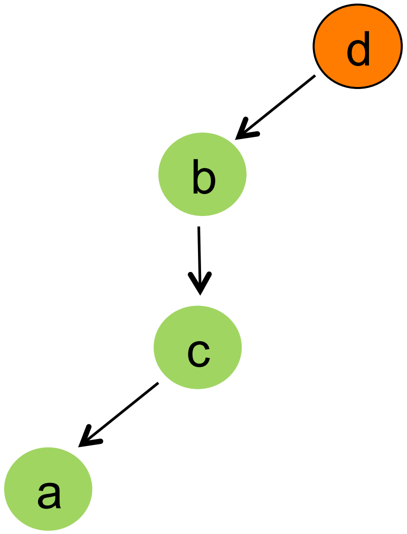
Every hb-relation must be captured

x, **y**, **m**, **p**, and **q** are shared and persistent.
t is local, *ready* is shared. Both are transient.
Initially **x** = **y** = **m** = **p** = **q** = *t* = *ready* = 0



T0	T1	T2
a1: lock(l1)	b11: lock(l2)	d21: lock(l3)
a2: *t* = *ready*	b12: **p** = **q**	d22: lock(l2)
a3: unlock(l1)	b13: unlock(l2)	d23: **q** = 1
4: if (*t*)	14: **x** = 1	d24: unlock(l2)
5: **y** = **x**	c15: lock(l1)	d25: **m** = 1
	c16: *ready* = 1	d26: …
	c17: unlock(l1)	

OCS level hb-relations



If OCS d fails but all hb-relations are captured,
all values are reset to a consistent state

If OCS d fails but all hb-relations are not captured,
y = 1 while others are 0, an inconsistent state