Extending Transactional Memory with Atomic Deferral

Tingzhe Zhou*, Victor Luchangco+, and Michael Spear*

*Lehigh University
+Oracle Labs

Transactional Memory Overview (1)

- Lock
  - Forget to take a lock (data race)
  - Take lock with wrong order (dead-lock)
  - Code re-use problems (composability)
  - Fine-grained locks (difficulty)

- Transaction
  - Atomicity
  - Serializability

```
public void enq(T x) {
  atomic {
    Qnode q = new Qnode(x);
    tail.next = q;
    tail = q;
  }
}
```

Sequential Code

Transactional Memory Overview (2)

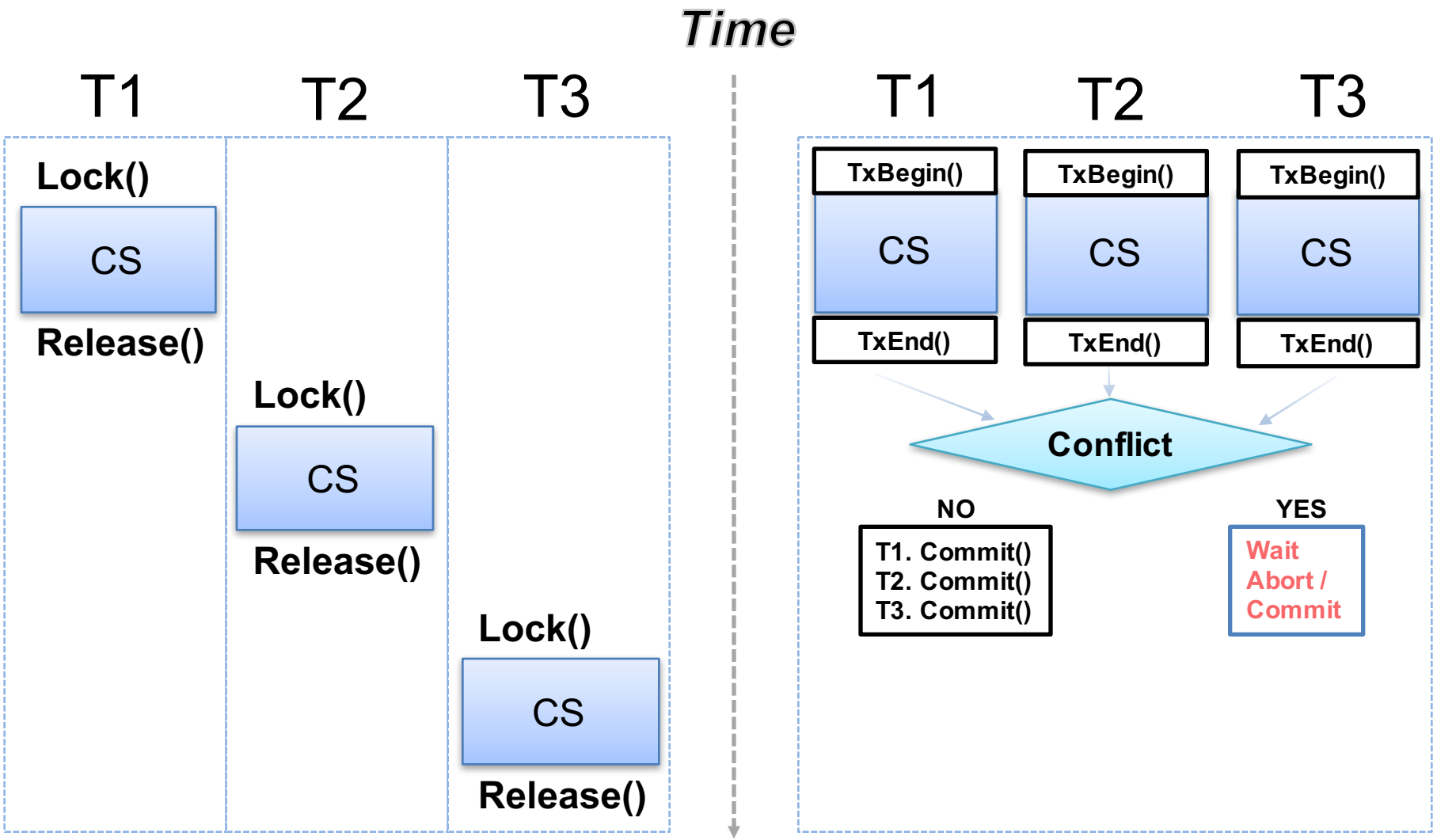
- Software Transactional Memory (STM)
  - instrumentation overhead
  - flexible

```
1  __transaction_atomic {          TxBegin();
2                                  val = TxRead(&counter);
3    counter++;            ==>     val++;
4                                  TxWrite(&counter, val);
5  }                               TxCommit();
```

- Hardware Transactional Memory (HTM)
  - faster
  - no progress guarantee

Buffering	L1 cache (32KB + 32 KB)
Conflict detection	Cache coherence protocol
Abort/Recovery	Invalidate transactional cache line
Commit	Validate transactional cache line

Transactional Memory Overview (3)



Obstacles for Using TM

- Irrevocable operations
  - I/O
  - some system calls
- Long-running operations
  - longer execution time
    - more likely to conflict
  - more memory access
    - STM: conflict, instrumentation overhead
    - HTM: capacity limitation
  - delay other transactions
    - conflicting transactions
    - all concurrent transactions (new finding)

Atomic Deferral

- Via 2PL, the suffix of the transaction remains atomic with the transaction, even though it is not run as a transaction
  - Differs from previous approaches to deferral: arbitrary and complex code allowed in the suffix
- Original motivation: defer an output operation *and its error handling code*
  - Consider writes to an unreliable socket: not just a syscall!
  - Or ensure the `fsync` happens at the right time
- Additional motivation: improve program performance
  - Exclusive use of transactions → correct
  - Addition of locks to protect certain data → avoid transaction overheads, remain correct

Before Atomic Deferral

- Irrevocable Transactions
  - Simplicity (Programmability, Implementation)
  - Limit concurrency
- Deferred Operations
  - Does not constrain concurrency
  - Some output operations can be deferred
  - Data copy
  - Ignore the return value

Privatization Problem

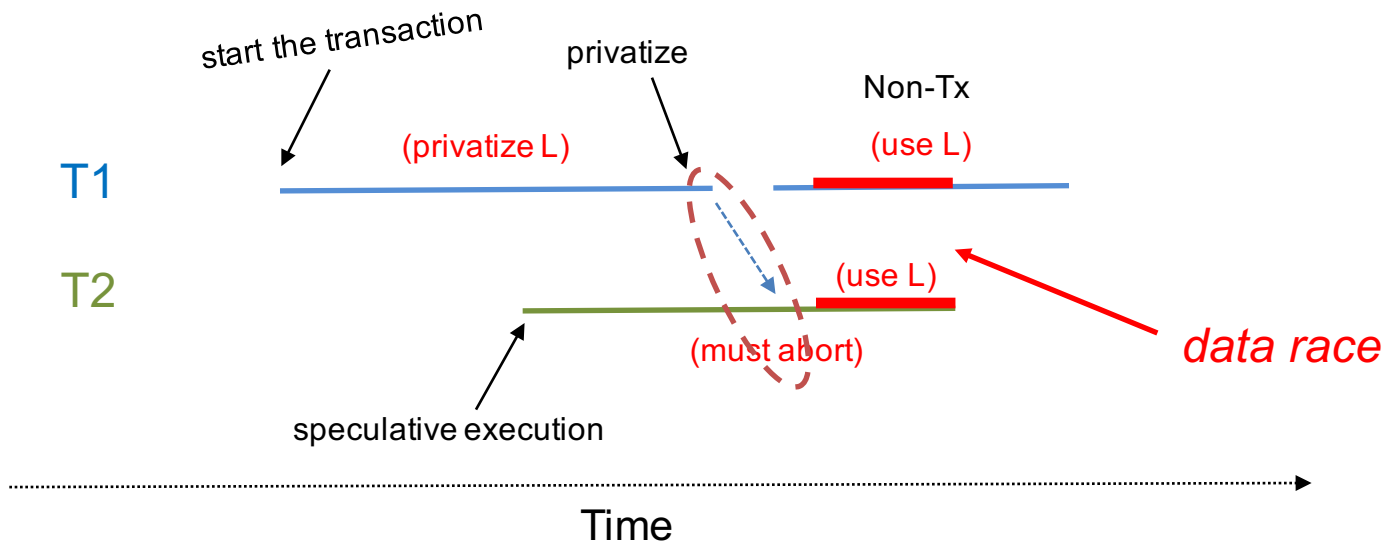
[Zhou, ICPP'17]
[Khyzha, PPoPP'18]

T1

```
__ transaction_atomic {
      node = L->head
      L->head = null
}

// L is privatized
process(node)
```

T2

```
__transaction_atomic{
      i_node = locate(L, i)
      if (i_node != null)
            i_node->data = process(i_node)
}
```



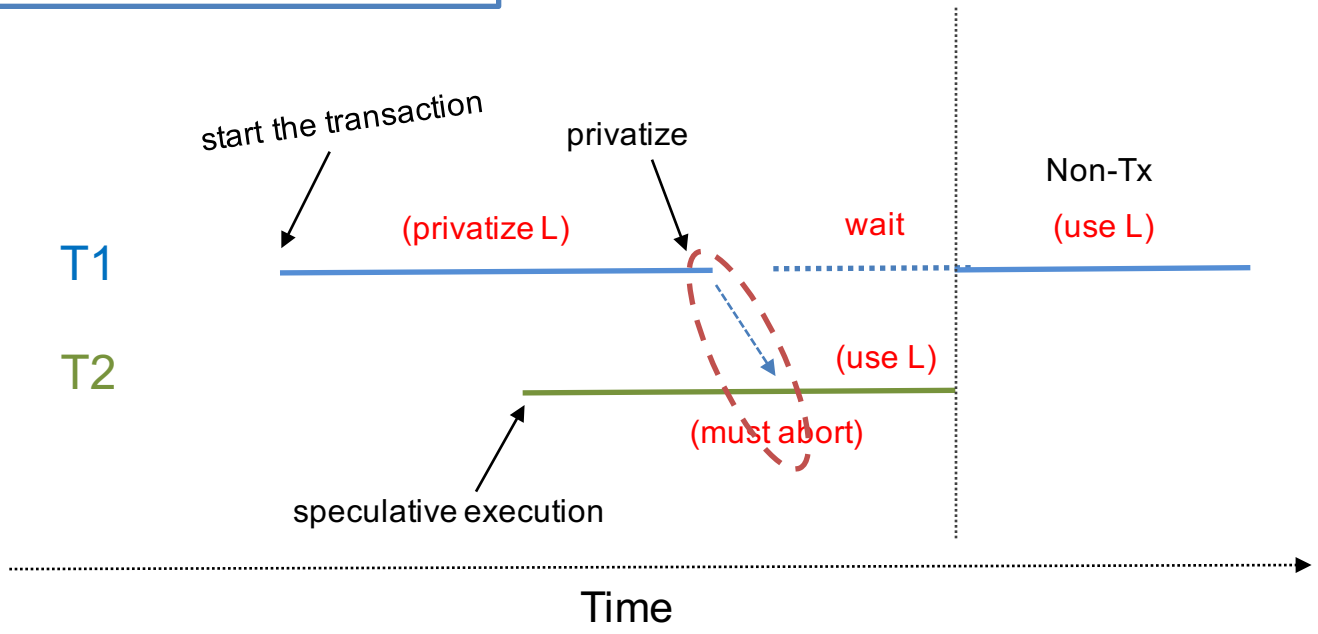
Quiescence in C++ TMTS

T1

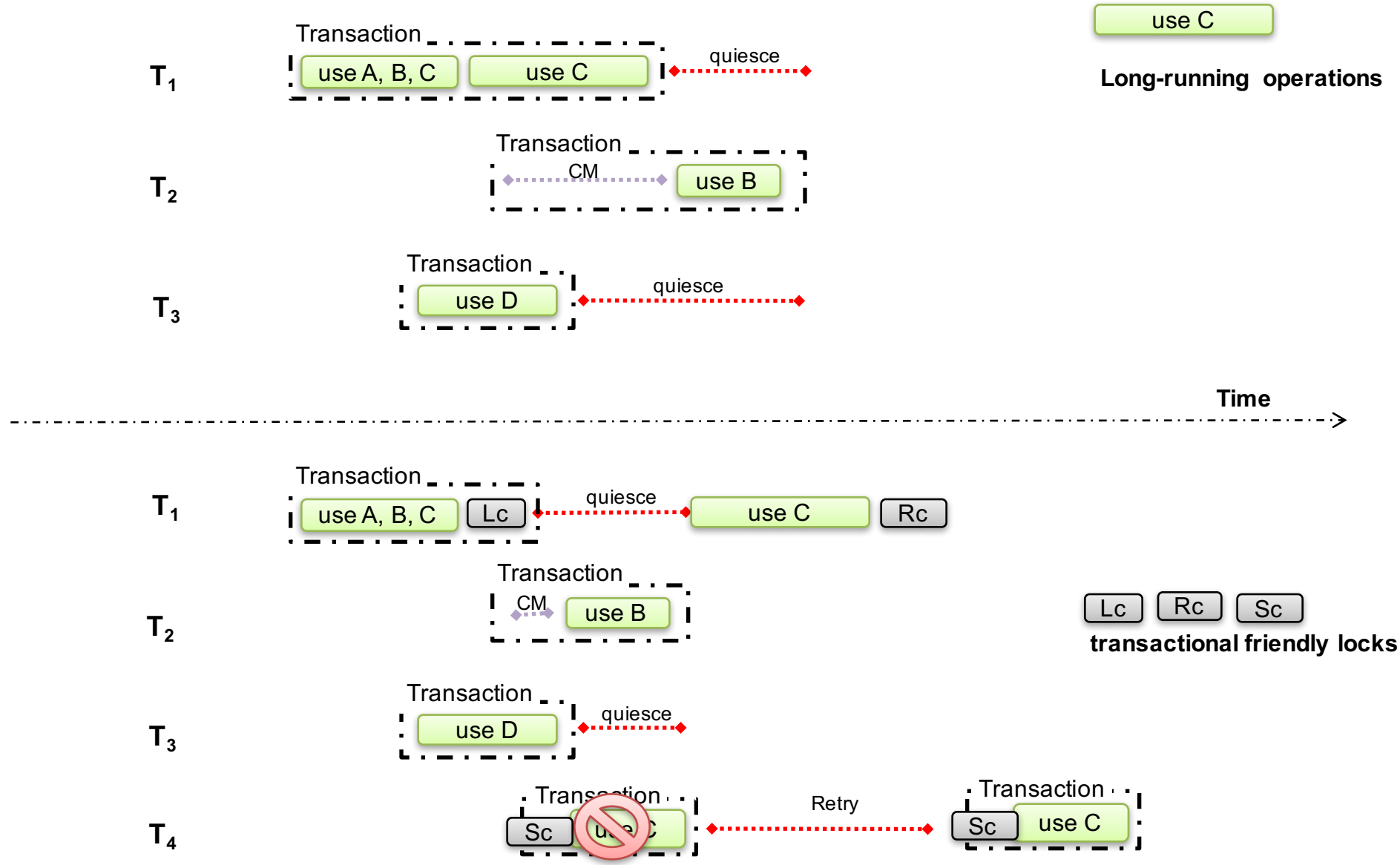
```
__ transaction_atomic {
      node = L->head
      L->head = null
}

// L is privatized
process(node)
```

T2

```
__transaction_atomic{
      i_node = locate(L, i)
      if (i_node != null)
            i_node->data = process(i_node)
}
```



A Motivating Example



Implementing Locks with TM

- Implement the lock as a bool
  - To acquire: set the bool from false to true via a transaction, or `retry` [Harris PPoPP 2005]
  - To release: set the bool from true to false via a transaction
  - To elide: read the bool: if true then `retry`

- Properties:
  - Locks can be acquired and released inside or outside of transactions
  - The use of `retry` ensures threads yield the CPU when the lock is held

Using Locks and Transactions Together

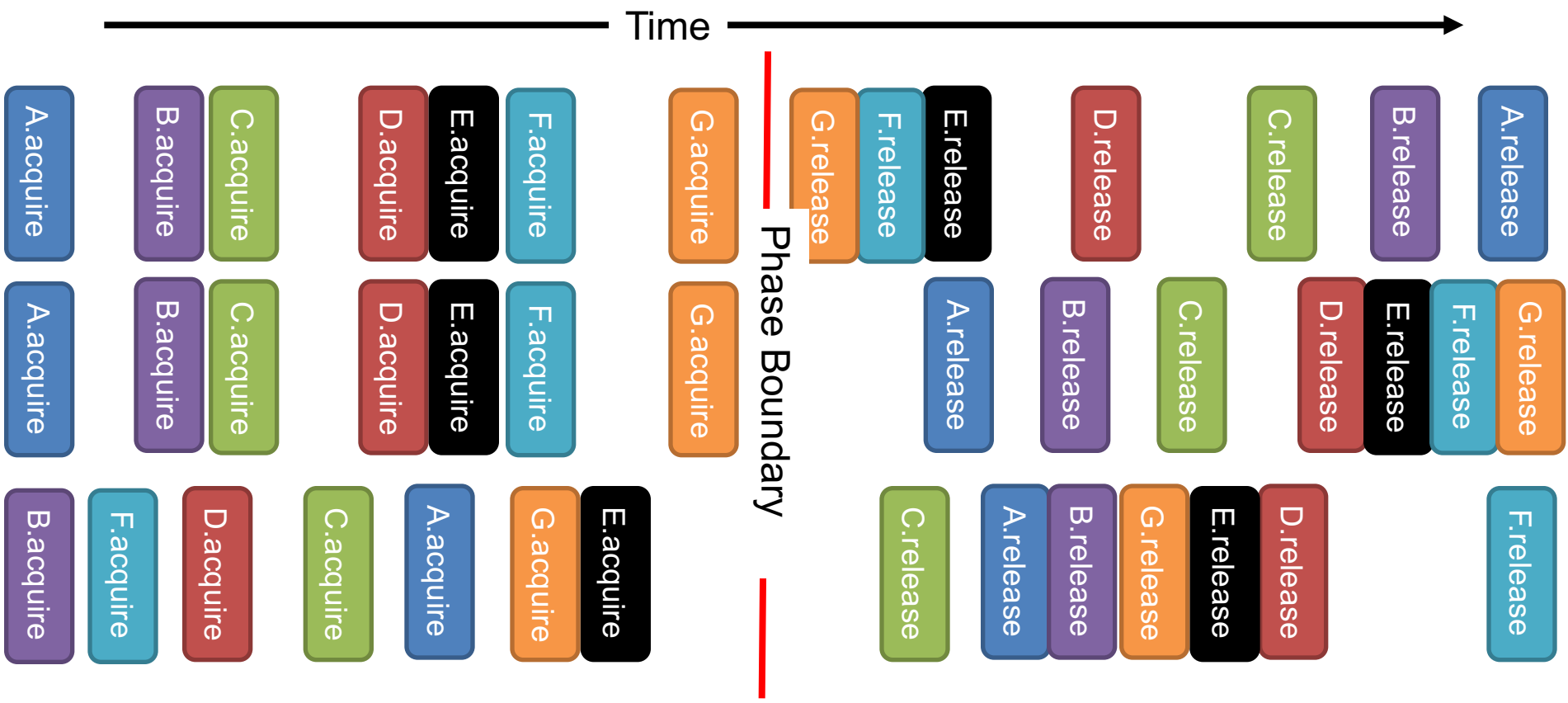
- Seems like a strange proposition…
  - Transactions are heralded as a replacement for locks
  - TM is simpler to use
  - TM scales better when the lock granularity is hard to determine, but conflicts are rare
- But TM is not a silver bullet
  - Can't do irrevocable operations (e.g., I/O) without serialization
  - Hardware TM capacity constraints may result in serialization
  - TM suffers worse from false conflicts

Lock-Based Semantics

- In lock-based programming, serializability is one of the most appealing correctness properties
  - The execution history is equivalent to one in which critical sections are executed without overlapping in time
- Serializability is trivial when there is only one lock
  - TM in C++ is serializable... “as if” one lock protects all transactions
- Serializability is guaranteed when the program obeys two-phase locking
  - An operation executes in two distinct phases: one in which locks are acquired, and one in which they are released

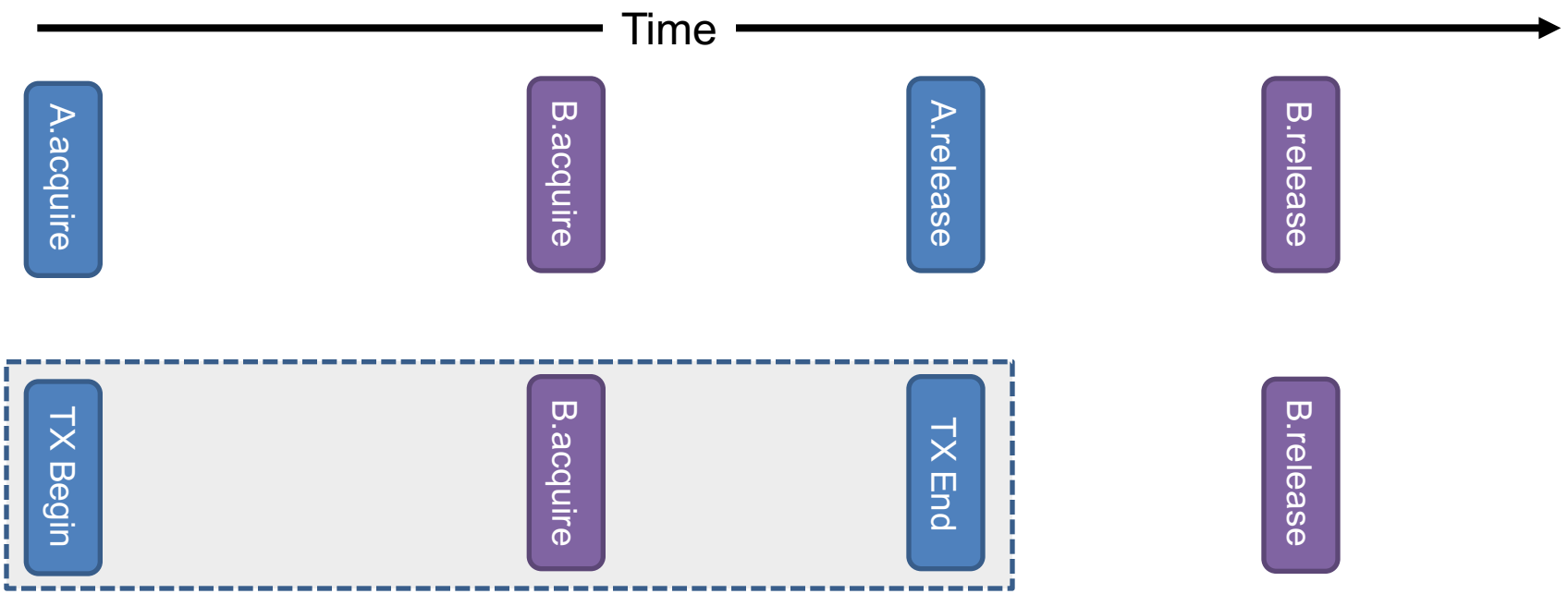
Two-Phase Locking (2PL)

- Each of the following is legal in 2PL



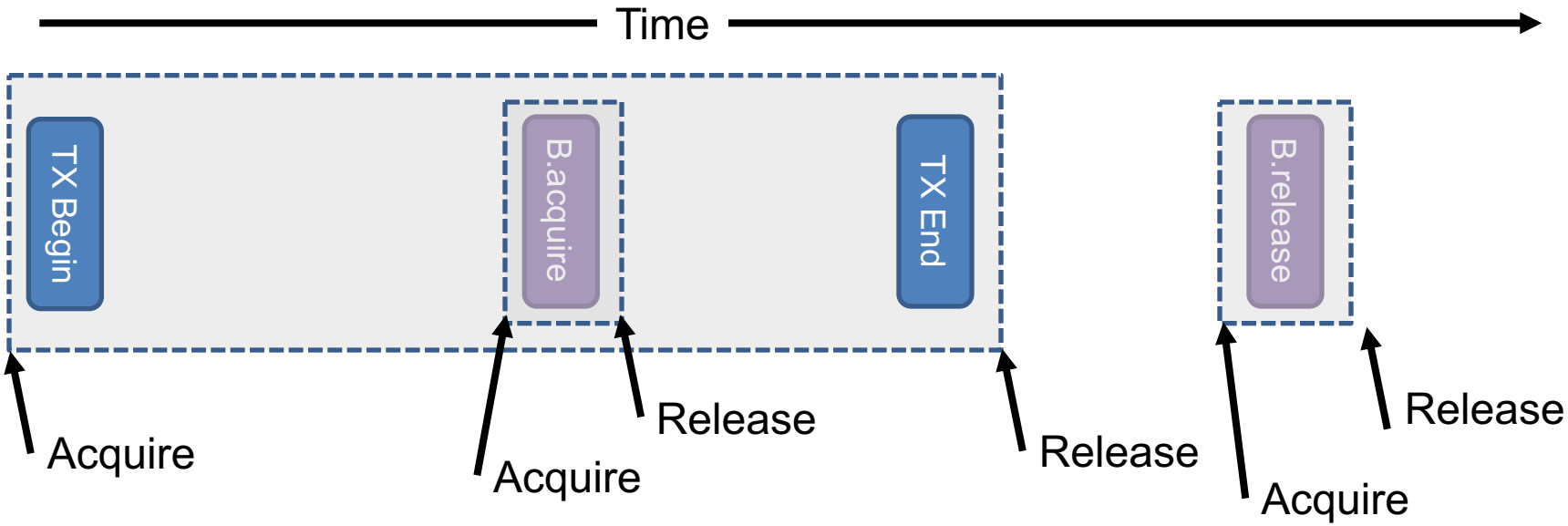
- Recall: serializability ensures correctness, but not progress!

2PL With Transactions?



Seems OK ☺

What if Locks Implemented via Transactions?



- Standard argument does not work
  - Each transaction is equivalent to acquiring and releasing some lock
- Claim (without proof): transactions for implementing locks don't affect reasoning about 2PL

Programmability

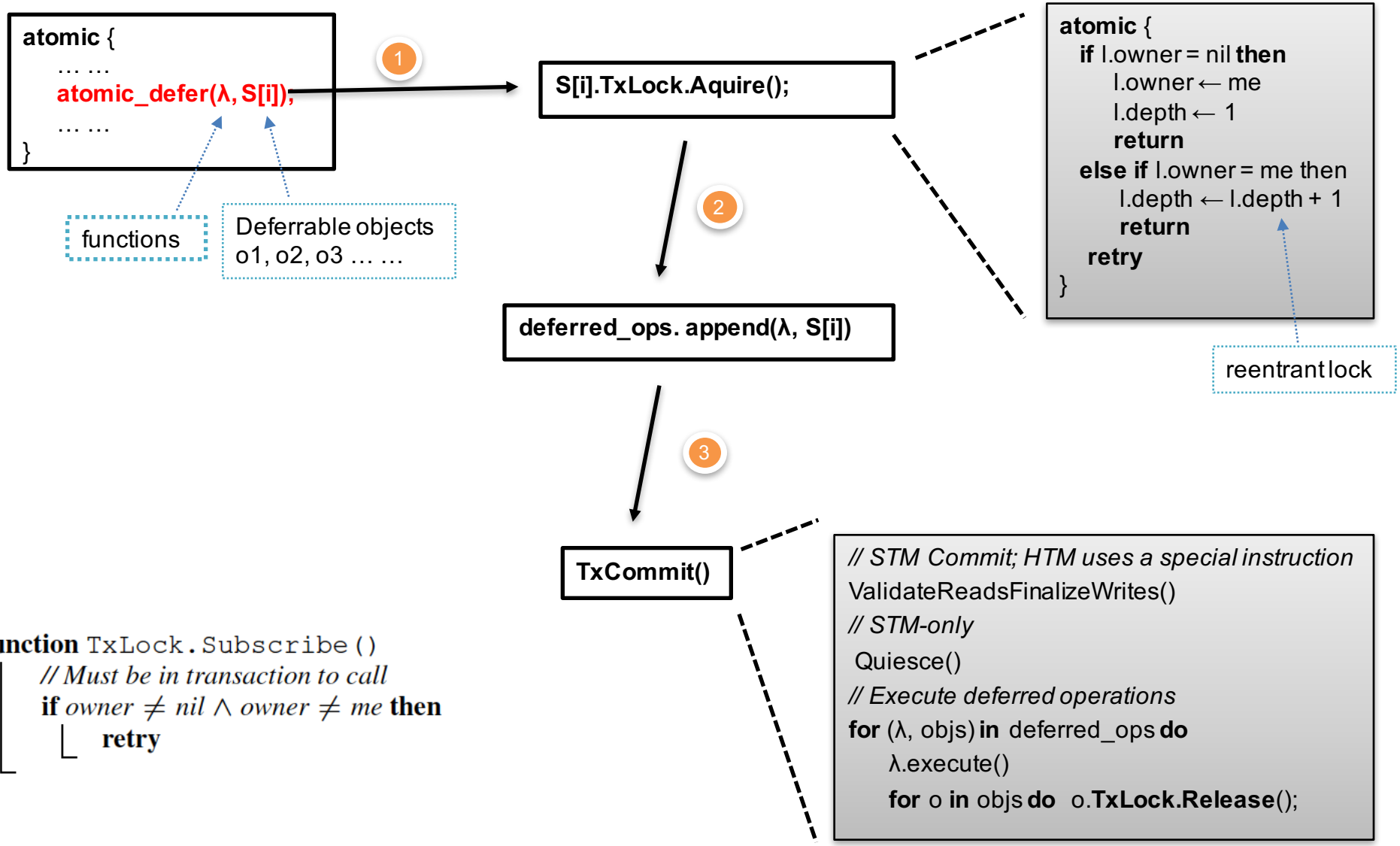
- Two new keywords
  - **Deferrable** annotation on classes
  - **atomic_defer** function

$\lambda \leftarrow () \{ o.expensive() \}$
$\texttt{atomic\_defer}(\lambda, o)$

```
class io_obj
{
    input_stream;
    output_stream;
    … …
}

io_obj  S = new io_obj[N];

λ←() { S[i].out(); }

synchronized {
    … …
    λ();
    … …
}
```

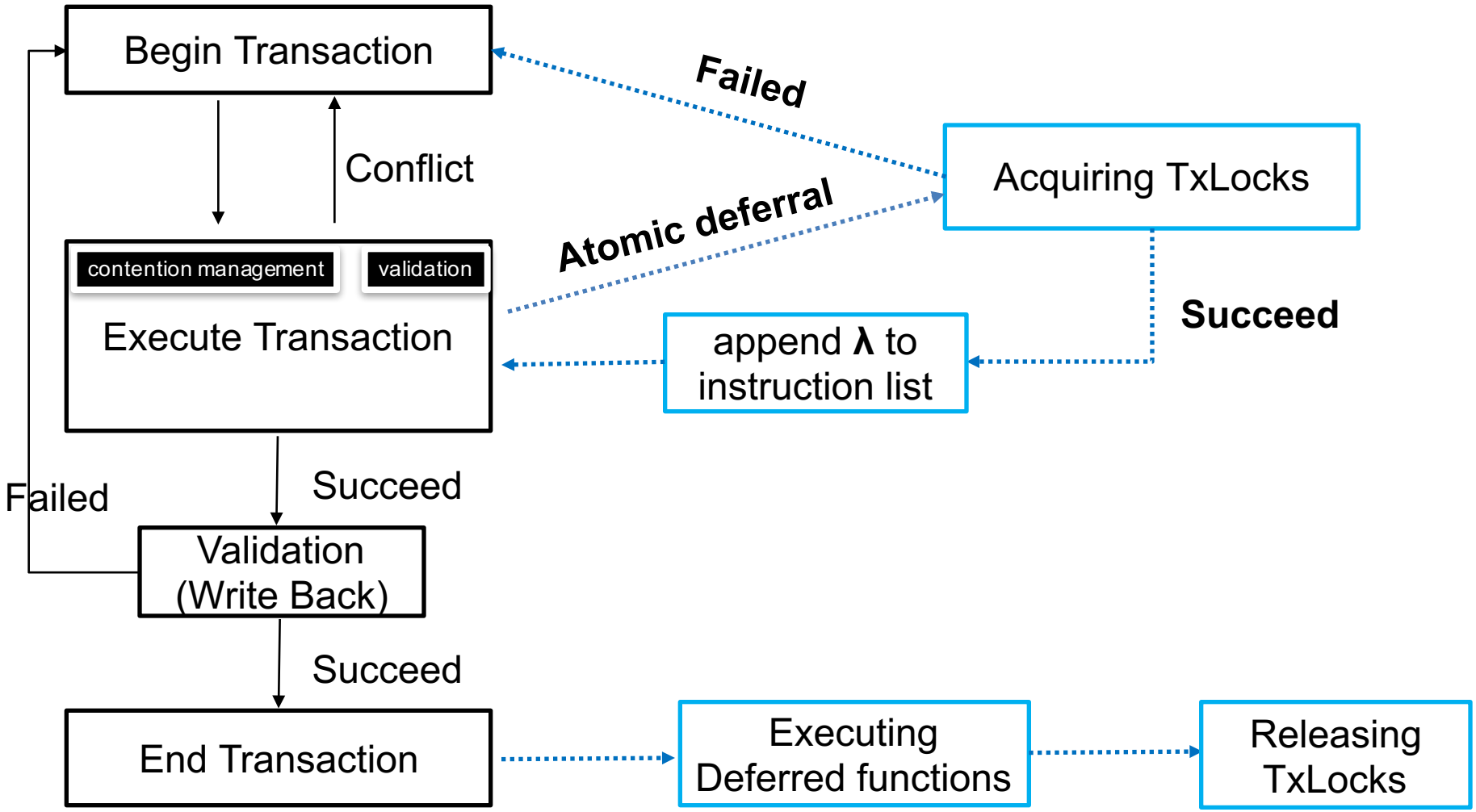
```
class io_obj public Deferrable
{
    input_stream;
    output_stream;
    … …
}

io_obj  S = new io_obj[N];

λ←() { S[i].out(); }

atomic {
    … …
    atomic_defer(λ, S[i]);
    … …
}
```

What's behind the scene?




```
function TxLock.Subscribe()
    // Must be in transaction to call
    if owner ≠ nil ∧ owner ≠ me then
        retry
```

High-level Execution

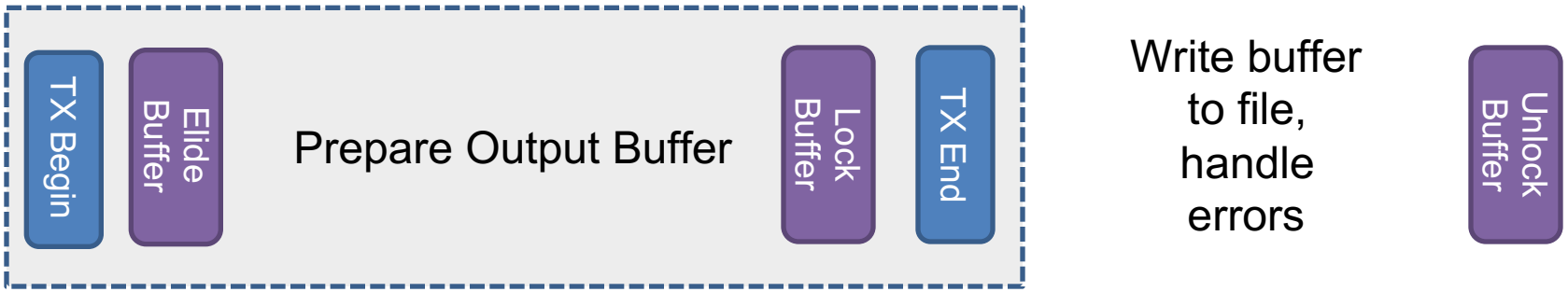


Practical Concerns

- Programmer may violate two phase locking
  - Wrapping transactions in deferred operation
  - Accessing objects without subscribe the corresponding Txlocks.

Example #1: Output Operations

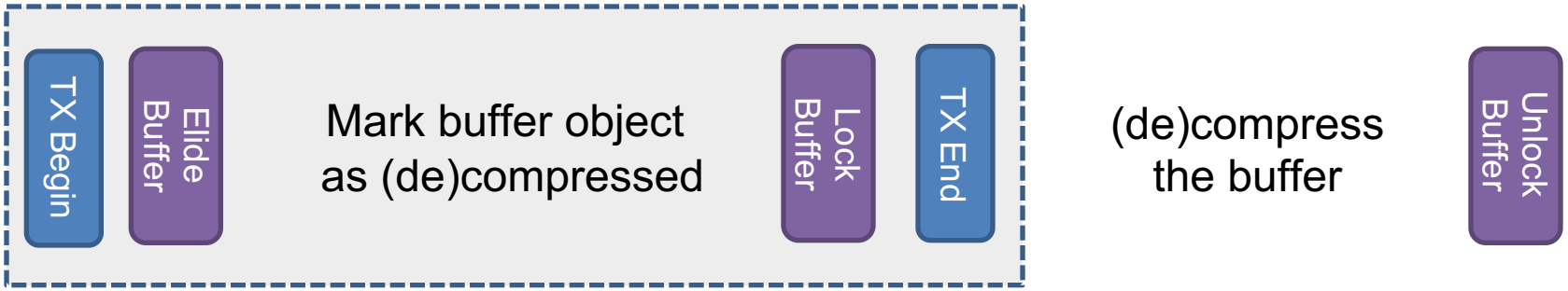
- System calls (e.g., writing to a file) cannot be done speculatively → must run transaction in isolation
  - With atomic deferral, system call is not in the transaction



- Concurrent accesses to buffer from within transactions must use the `elide()` instruction on the buffer's lock to respect mutual exclusion if lock is held
- Concurrent accesses to buffer from outside transactions must acquire the buffer's lock

Example #2: Long-Running Operations

- Long-running, pure functions lead to slowdown
  - Instrumentation overhead in software TM
  - Capacity constraints in hardware TM
- Example: (de)compression in PARSEC dedup
  - Given a byte stream, produce a new byte stream



  - Other users of byte stream must use the buffer lock's elide operation before checking if buffer (de)compressed

Experiment

- 4-core/8-thread Intel Core i7-4770 CPU running at 3.40GHz.
- Code

Listing 6: An example of deferring I/O and system calls

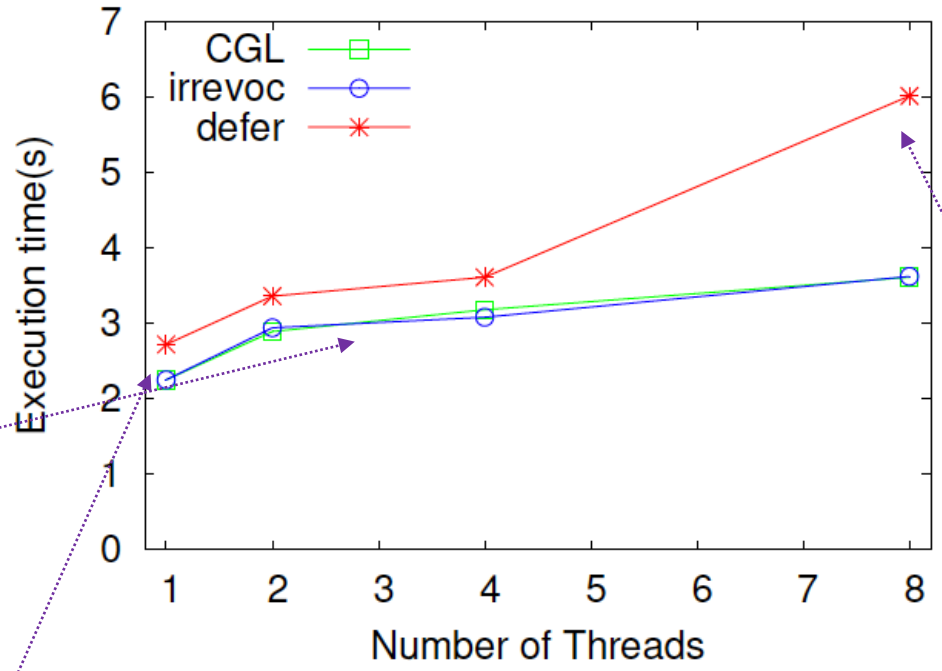
```
// Encapsulate streams in a Deferrable
object
class defer_file: public Deferrable
    input     // input stream
    output    // output stream

// An array of files
dfs: defer_file[]
```

```
// Operation to be deferred
1  λ ←(id, content)
       // Read File
2      if ¬dfs[id].input.open() then
       error
       // Get the length of the file
3      dfs[id].input.seekg(0, end)
4      len ← dfs[id].input.tellg()
5      dfs[id].input.close()
       // Write to the file and close
6      tmp ← format(content, len)
7      dfs[id].output.write(tmp)
8      dfs[id].output.close()
```

```
// Irrevocable version of benchmark
1  synchronized
2      content ← ...
3      id ← ...
4      λ(id, content)

// atomic_defer version of benchmark
1  atomic
2      content ← ...
3      id ← ...
4      atomic_defer(λ(id, content),
       dfs[id])
```

atomic_defer performance (1 file descriptor)

No concurrency



irrevocable Tx is carefully optimized in GCC TM

Retry is not supported by c++ TMTS

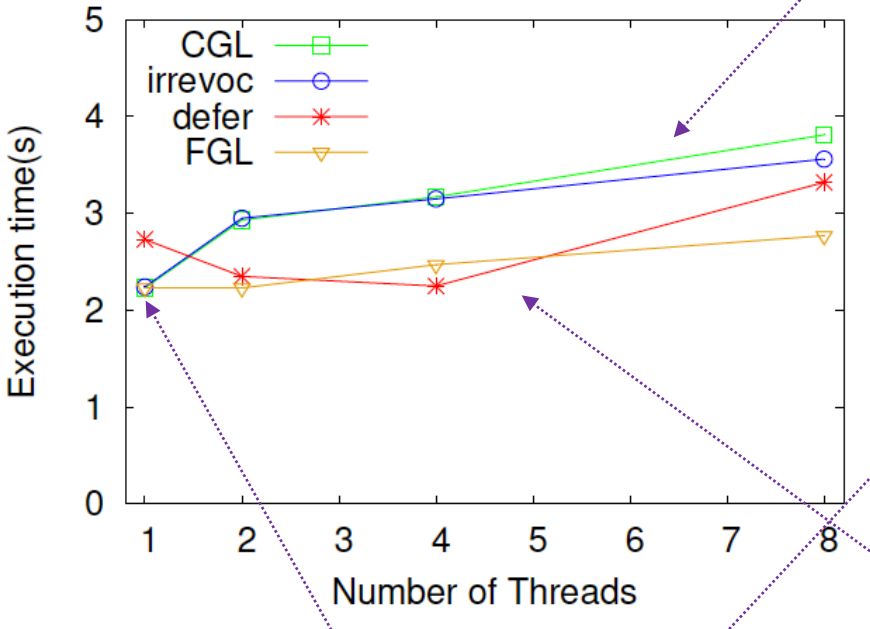
Latency: Lambda, Instrumentation… …

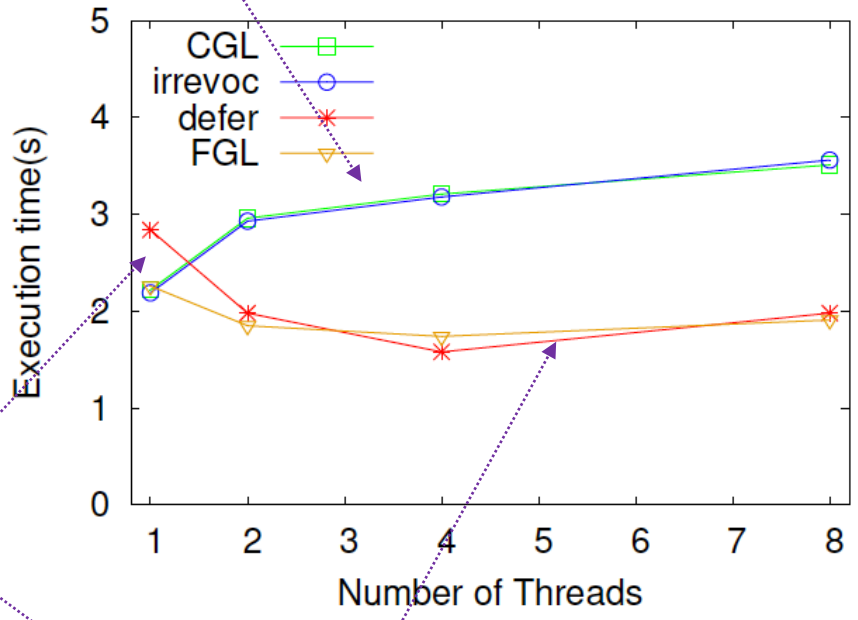
atomic_defer performance (2, 4)

Two files

CGL and Irrevoc

Four files



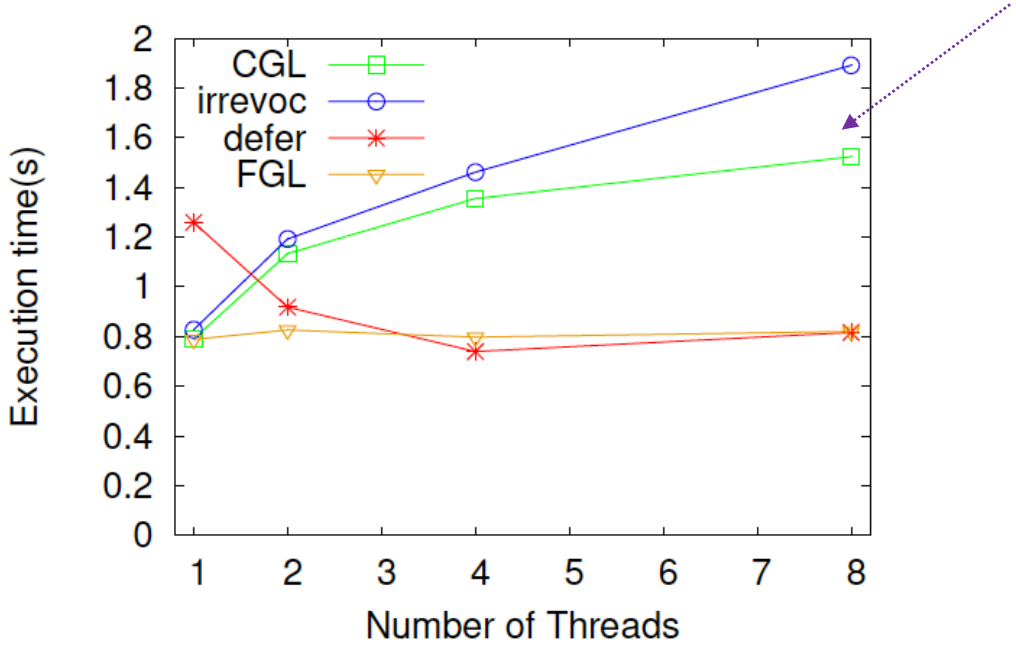


Atomic_defer scales !!!

Latency: Lambda, Instrumentation… …

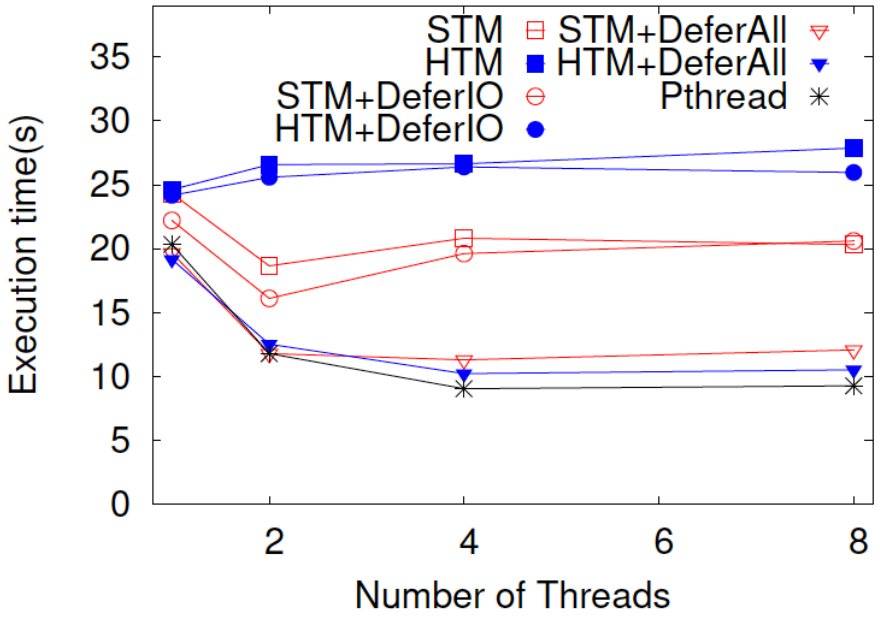
atomic_defer performance (4, small)

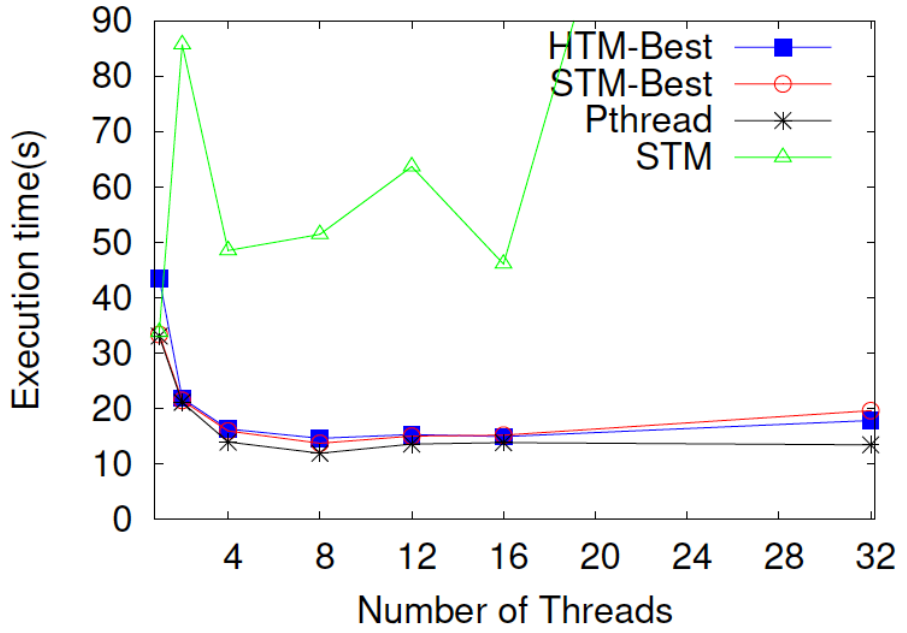
Irrevocable transaction shows its overhead, it performs even worse than CGL



Parsec Dedup Kernel

- 18-core/36-thread Intel E5-2699 V3 CPUs running at 2.30GHz.





Conclusions

- Non-atomic I/O deferral isn't enough
  - Network I/O is more than a syscall... Need to handle errors atomically!
- Locking can be an optimization for transactional programs
  - Avoid copying
  - Calls to elide() can be handled by compiler
- Next step: more workloads
  - Focus thus far: output stage of pipeline parallelism
  - Other opportunities: management of open file descriptors in MySQL, logging operations in cloud applications, asynchronous file output, ...

Q & A

- Thank you !
- Contact Info:
  - Tingzhe Zhou: tiz214@lehigh.edu