Dissecting Transactional Semantics and Implementations

Suresh Jagannathan

PURDUE
UNIVERSITY

(S^3)

Observations

- Mainstream adoption of concurrency and distributed programming abstractions
  - ★ Heavy burden on programmer to balance safety and performance
  - ★ Well-known issues with deadlocks, data races, priority inversion, interaction with external actions, etc.
  - ★ Scalability impacted by the use of mutual-exclusion
    - ✦ Finer-grained locks require more care to prove correct
- Advent of multi-core processors
  - ★ Each core can support multiple threads
  - ★ Programmability remains an open question:
    - ✦ How much parallelism can a compiler safely extract?
- Can we simplify concurrent program structure without sacrificing efficiency or scalability?
  - ★ Lock-free data structures and algorithms
  - ★ Software transactions (obstruction-free)

Software Transactions

- Instead of strict synchronization semantics induced by lock-based abstractions,
  - ★Define a relaxed synchronization model:
    - ✦ Decouples shared access from synchronization machinery
    - ✦ Allow concurrent access to shared data provided serialization invariants are not violated.
  - ★Separate specification of program correctness from implementation of a specific solution
    - ✦ Define a guarded region of code protected by a specific concurrency control protocol.
    - ✦ Ideally, applications should be able to overspecify the scope of these regions:
      - The burden of how and when tasks can concurrently access shared data within these regions is shifted from the application to the implementation.

Goals

- Safety
  - ★ Race-freedom
  - ★ No priority inversion
  - ★ Guarantee serializable execution
- Improved performance
  - ★ Access to shared data structures can take place concurrently provided there is no violation of serializability
    - ✦ Imposes weaker constraints on implementations
  - ★ Beneficial impact on scalability
- Software engineering
  - ★ Facilitates new abstractions and methodologies
    - ✦ Can dissect aspects of transactional semantics and implementations for specialized structures and mechanisms.

Outline

- Background and Examples
- Case Study: Implementations
  - ★ Transactional Monitors
- Semantics: A Transactional Object Calculus (optional)
- Case studies: Applications
  - ★ Safe Futures
  - ★ Checkpointing and message-passing

Approaches

- Serial access to shared data using lock-based abstractions
  - ★ Programmer responsible for correct and efficient placement of locks.
- Serializable access to shared data:
  - ★ Provide two important properties:
    - ✦ Atomicity: effects of updates seen all-at-once or not-at-all.
    - ✦ Isolation: while executing within a shared region, effects of other threads not witnessed.
  - ★ Serial execution through locks is a conservative approximation of serializability.
  - ★ Optimistic transactions: allow threads to execute shared (guarded) regions of code assuming serializability will hold.
    - ✦ When it fails, abort and retry.
  - ★ Pessimistic transactions: associate locks with all shared data and acquire when accessed, and release at end of transaction.
    - ✦ Deadlock on lock acquires, requires abort and retry.

Basic Actions

- Start
  - ★monitor access within the dynamic extent of a transaction region
- Log
  - ★Record updates within a transaction in case an abort occurs
- Abort
  - ★Restore global state and retry
- Commit
  - ★Check serializability invariants

Phases

- Optimistic:
  - ★ Read phase: maintain log recording reads and writes to shared data.
  - ★ Validation phase: compare transaction log with global state:
    - ✦ Abort if comparision reveals a serializability violation.
  - ★ Commit phase: update shared data to the heap.
- Pessimistic:
  - ★ Read phase: acquire locks on shared reads and writes.
    - ✦ Log original values to handle aborts.
    - ✦ Abort if a deadlock exists among multiple transactions that require resources (i.e., locks) held by the other.
  - ★ Commit phase: release held locks.
    - ✦ Updates always immediately performed to the global heap.
- The two approaches are not necessarily exclusive:
  - ★ Consider pessimistic writes and optimistic reads.
    - ✦ Allows transactions to eagerly abort on conflicting writes.

Foundational Mechanisms

- Logging –
  - ★versioning used to redirect transactional accesses
  - ★versioning to used to restore aborted transaction
- Dependency tracking –
  - ★ discover violations of serializability
  - ★ discover deadlocks on lock access
    - ✦ Granularity of conflict detect (word vs. object)
- Revocation –
  - ★ undo effects of transactions violating serializability and re-execute them
  - ★ undo effects of deadlock transactions
  - ★ contention management:
    - ✦ When a transaction aborts, when should it run again?
      - ◗ How should livelocks be prevented?
      - ◗ Obstruction-freedom

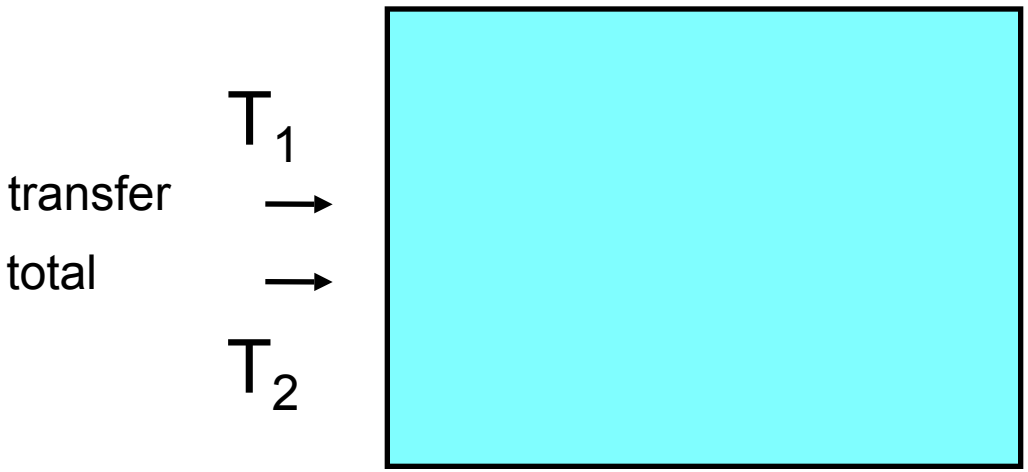
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



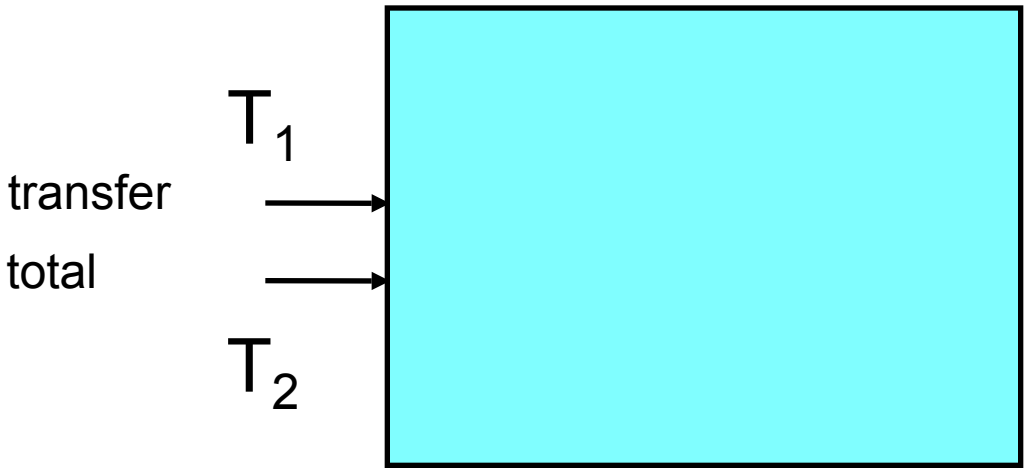
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



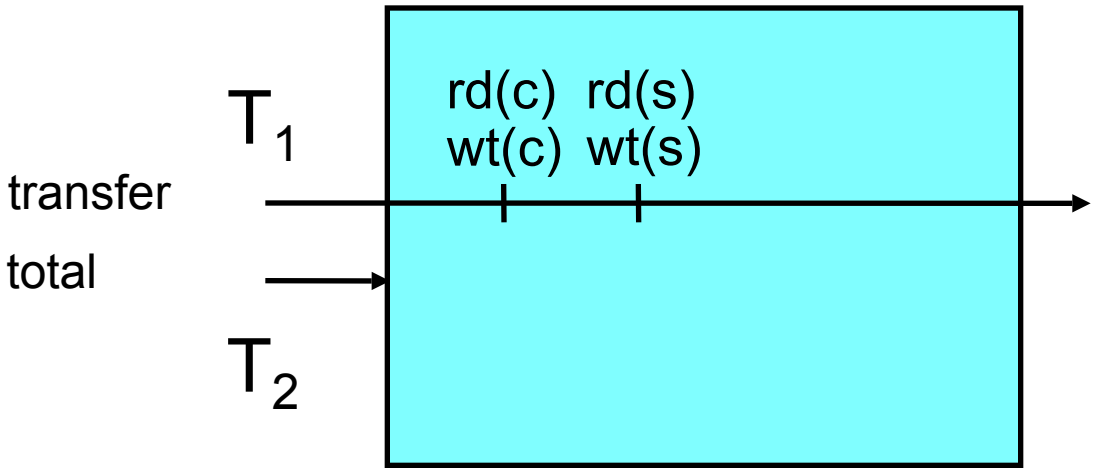
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10

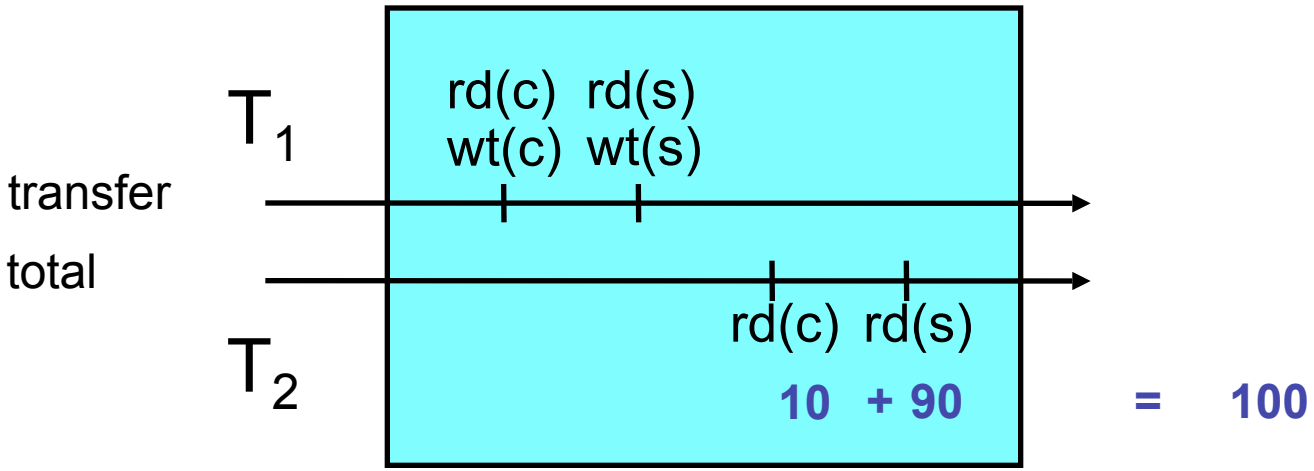


Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



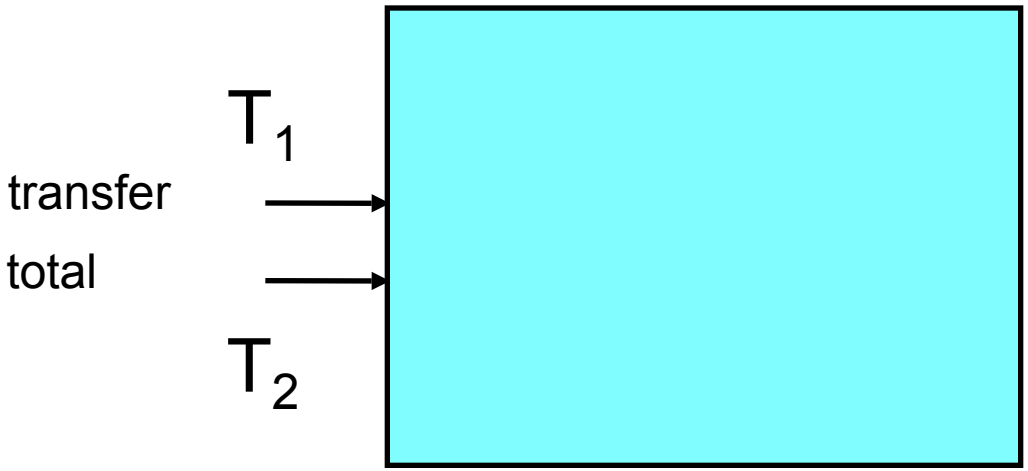
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



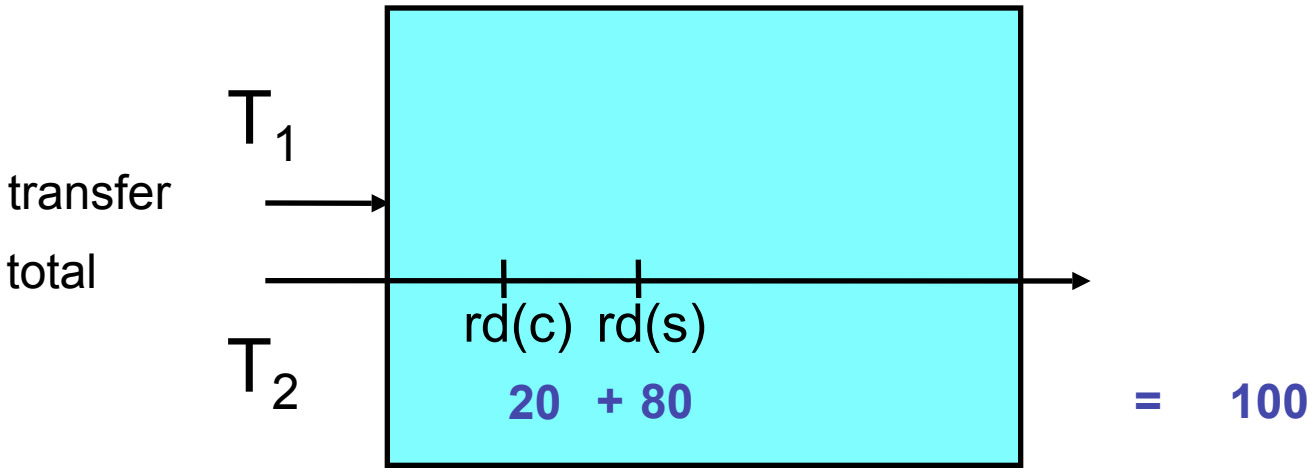
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



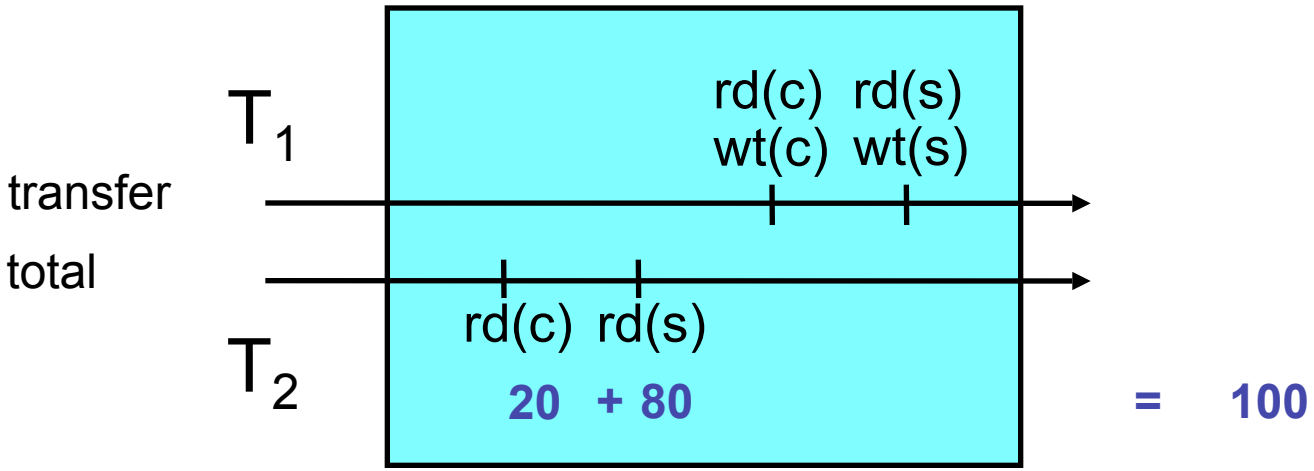
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

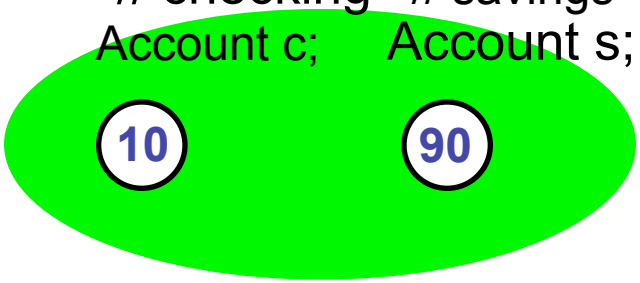
10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

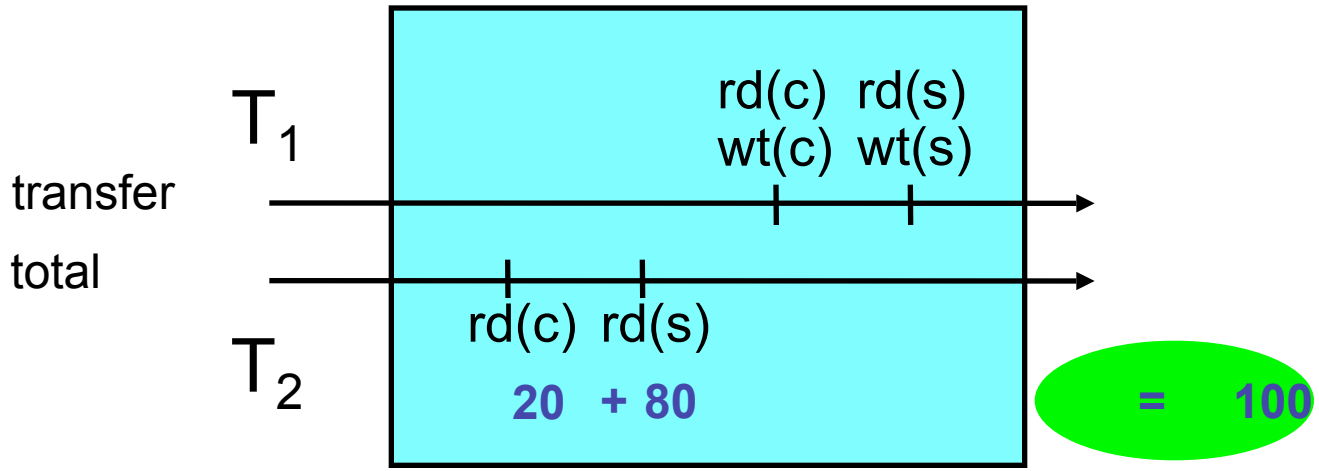
10



Exclusive Monitors




```
                                                    10
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



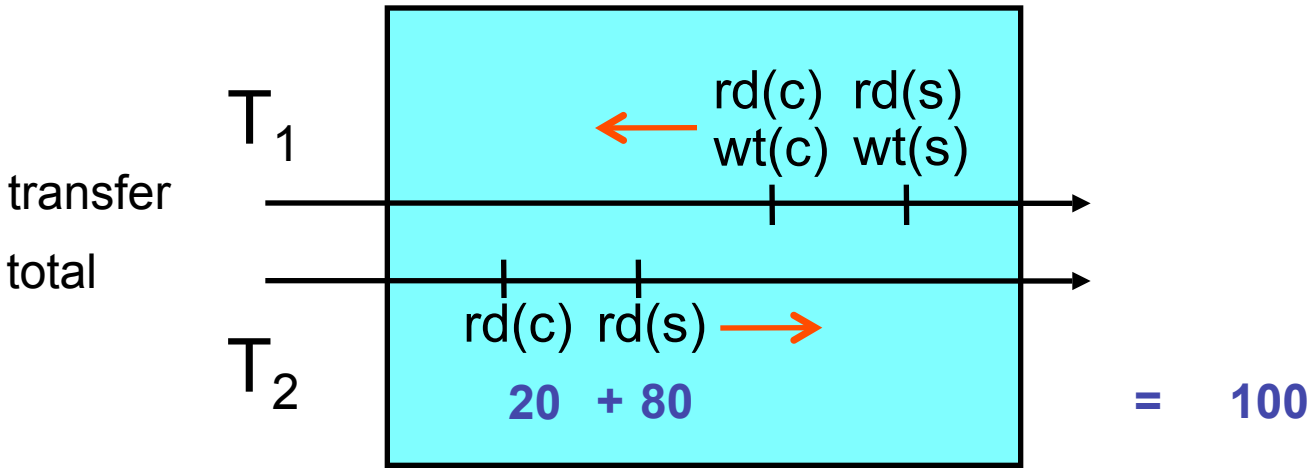
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



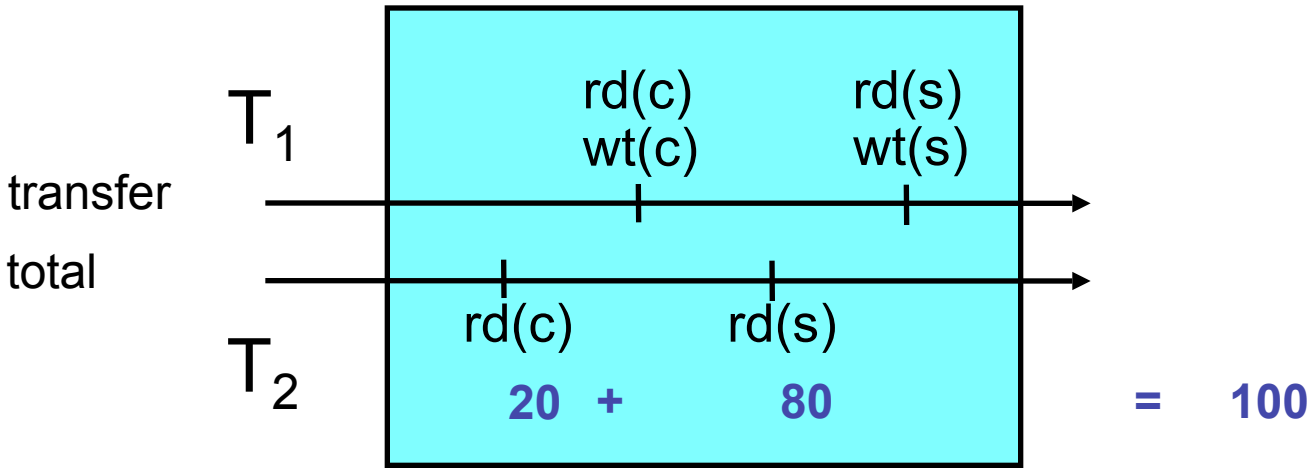
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



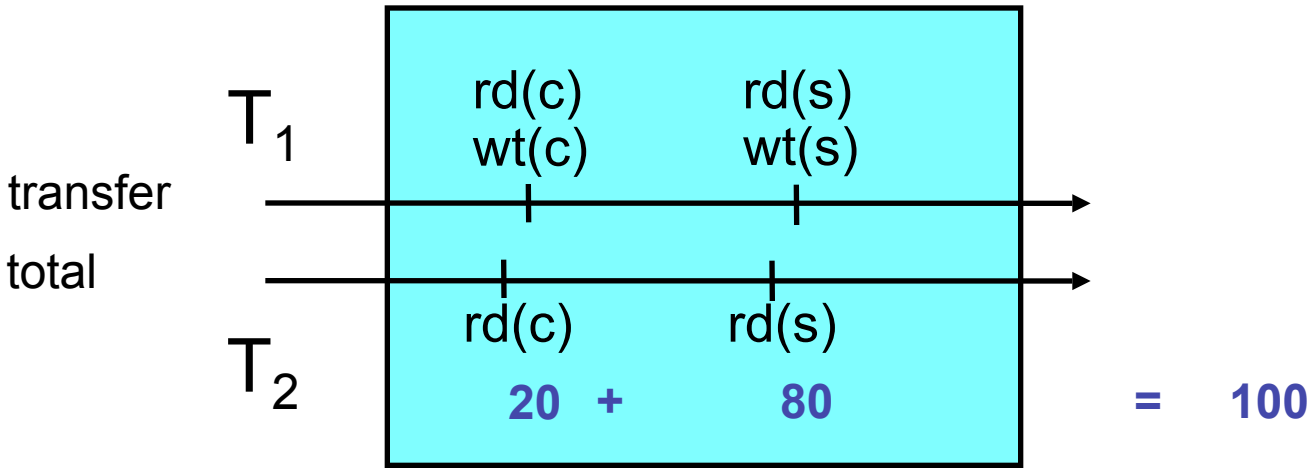
Exclusive Monitors

```
// checking   // savings
Account c;    Account s;
```

10 90

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



Transactional Monitors

- Monitors executed as optimistic transactions – relaxed interleavings allowed
- Enforce serializable execution
- Effective when contended
- Both exclusive and transactional monitors can co-exist: they produce the same effects (serializability)

Ensuring Serializability

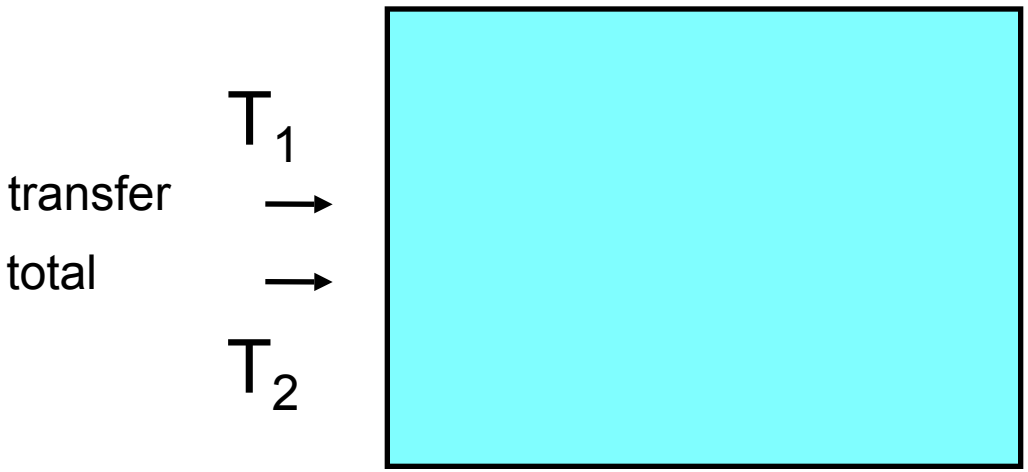
```
// checking   // savings
Account c;    Account s;
```

20 80

atomic

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



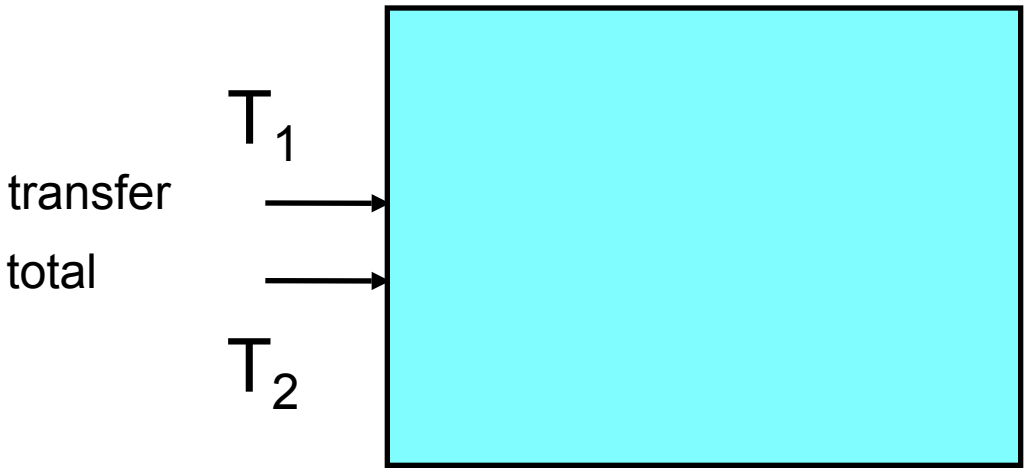
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



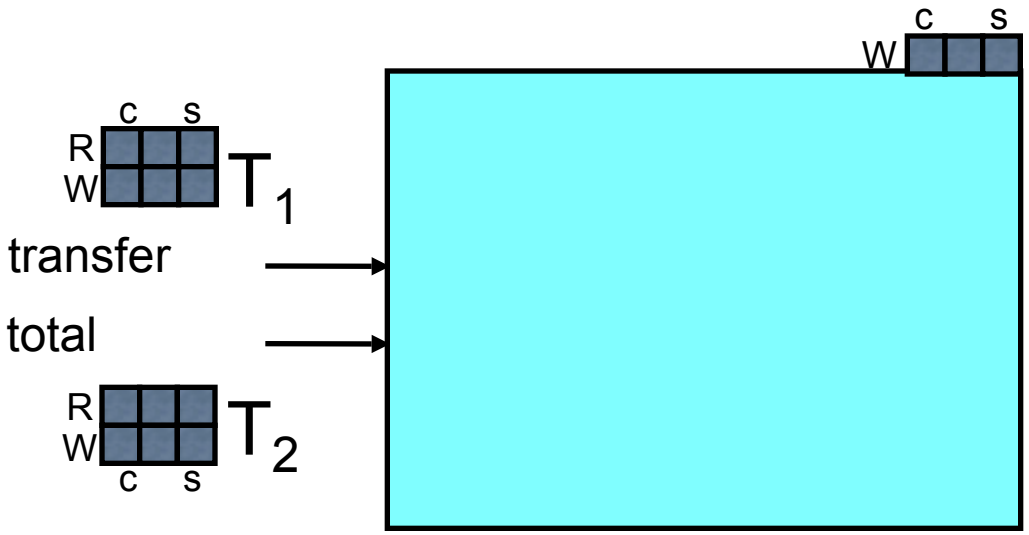
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



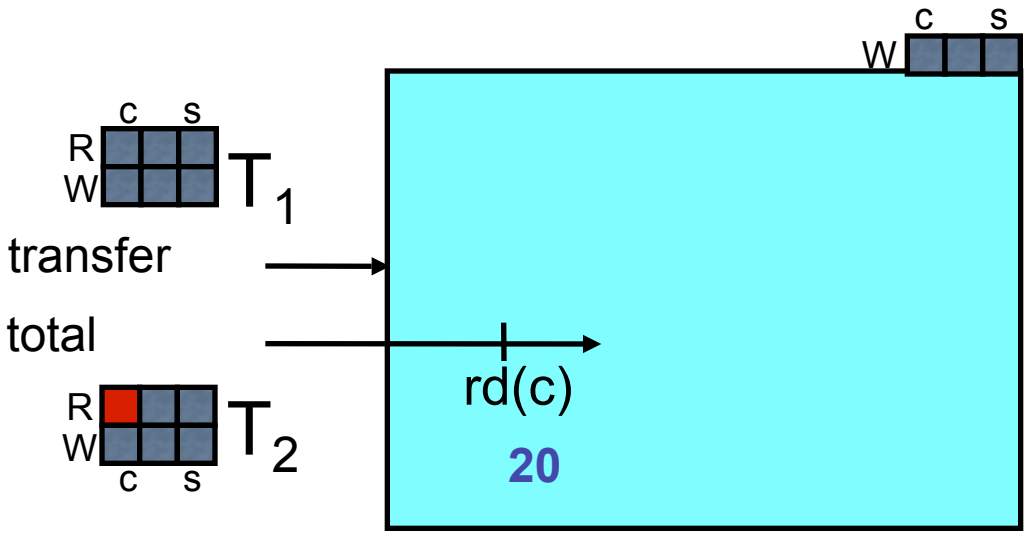
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```

20 80

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10

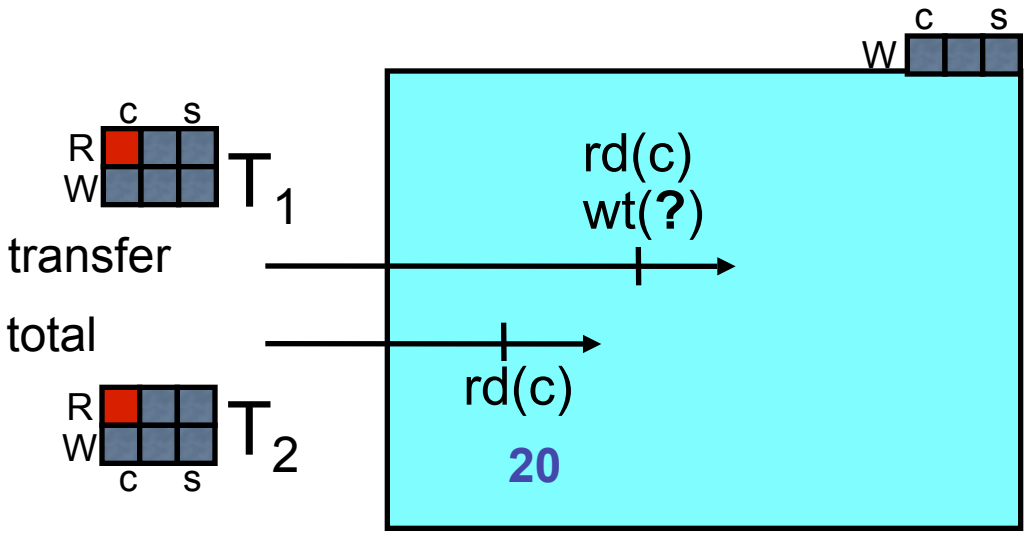


Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```

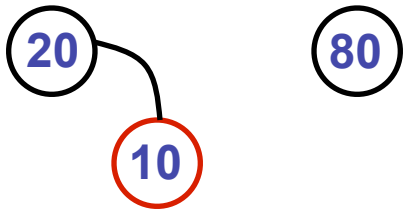
20 80

```
void synchronized transfer (int sum)   // sum = 10
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



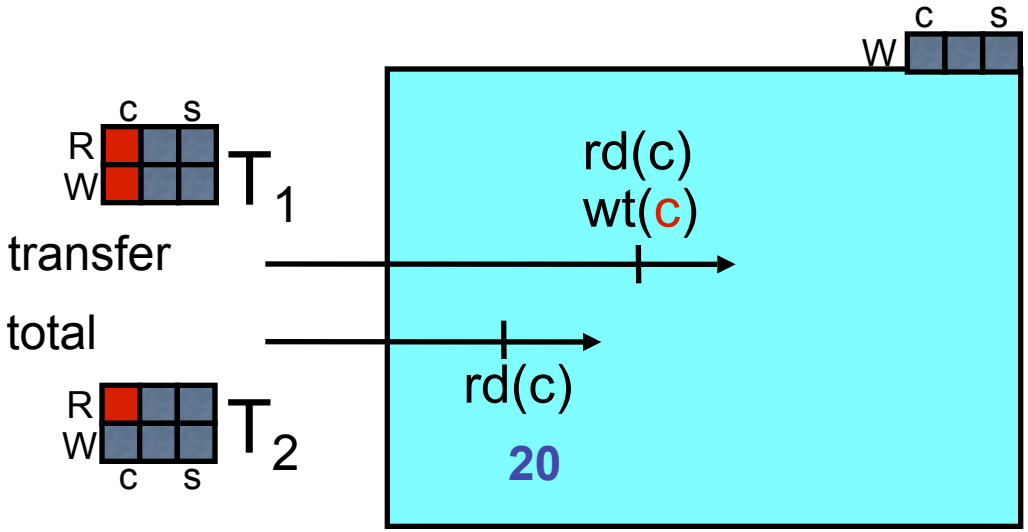
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```



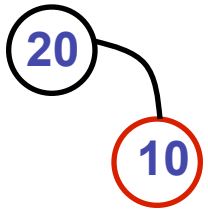

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10

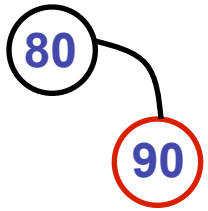


Ensuring Serializability

// checking
Account c;

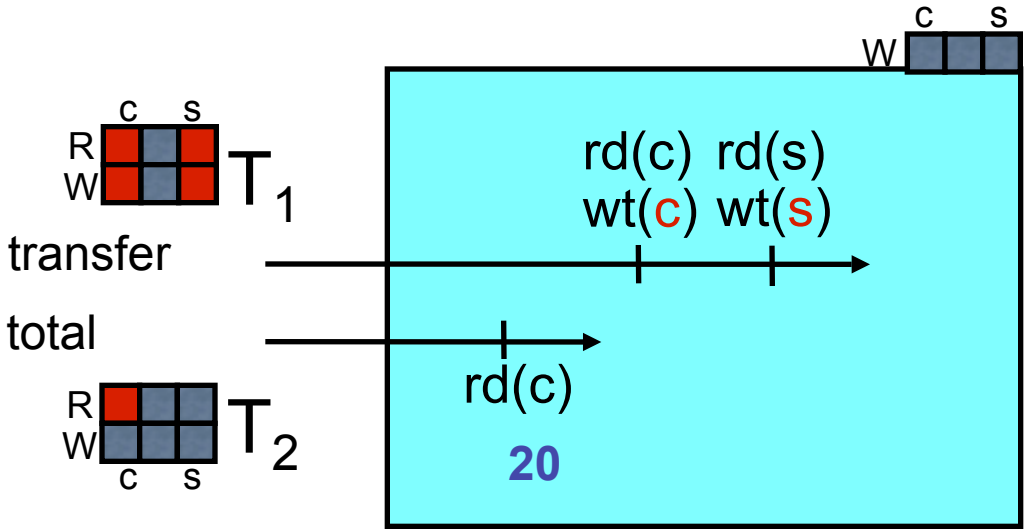


// savings
Account s;



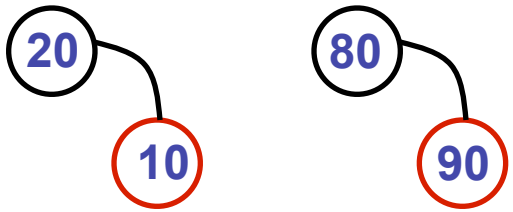

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



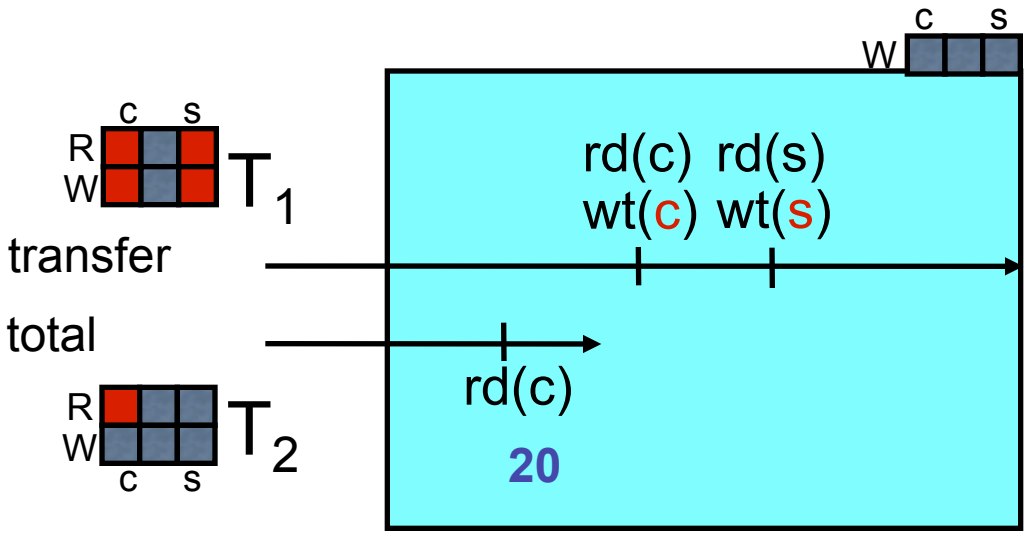
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```



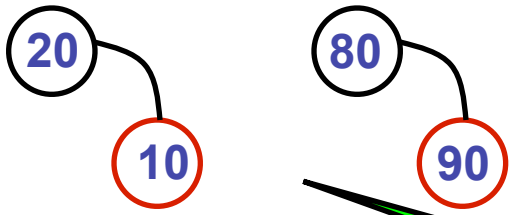

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



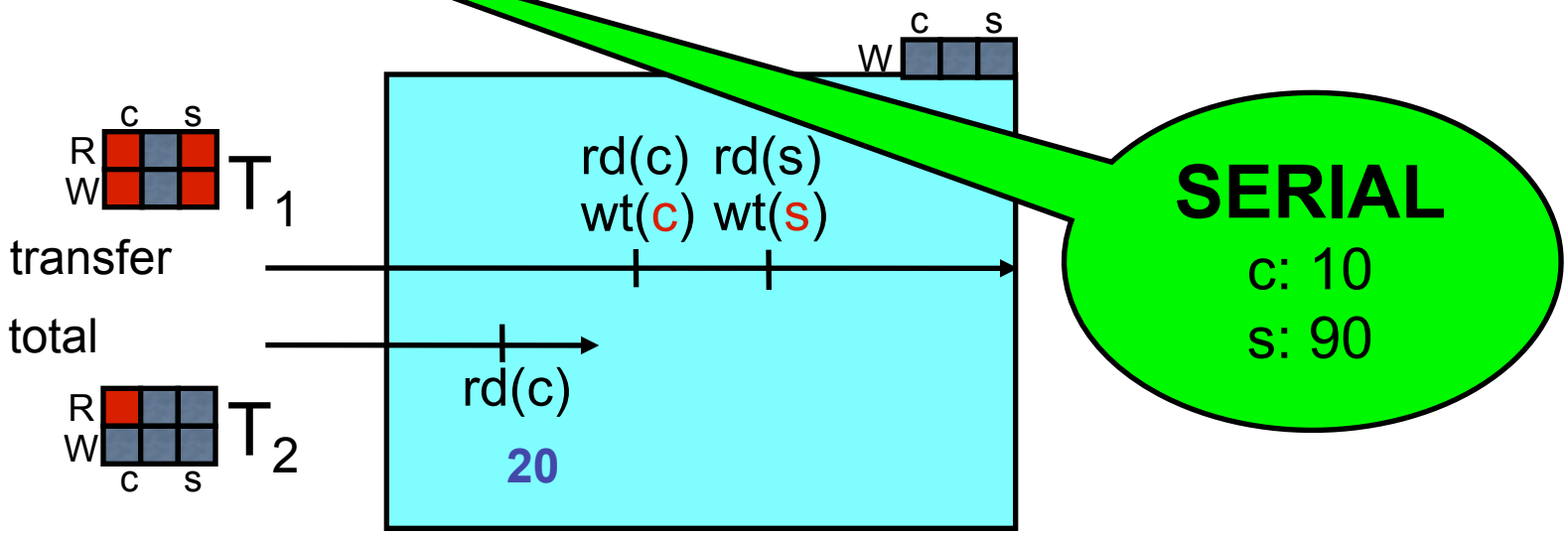
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```



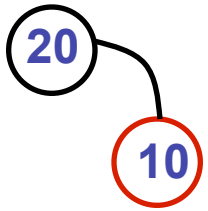

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10

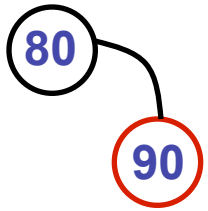


Ensuring Serializability

// checking
Account c;

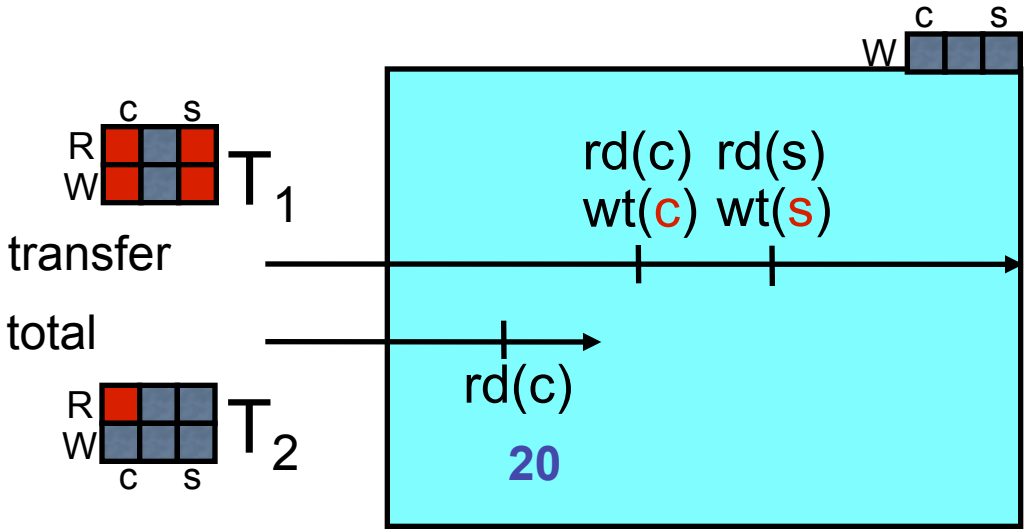


// savings
Account s;



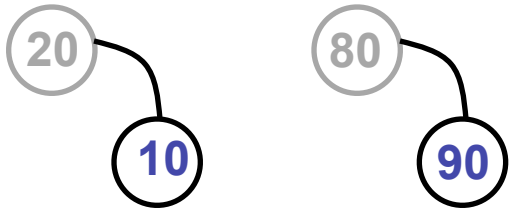

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



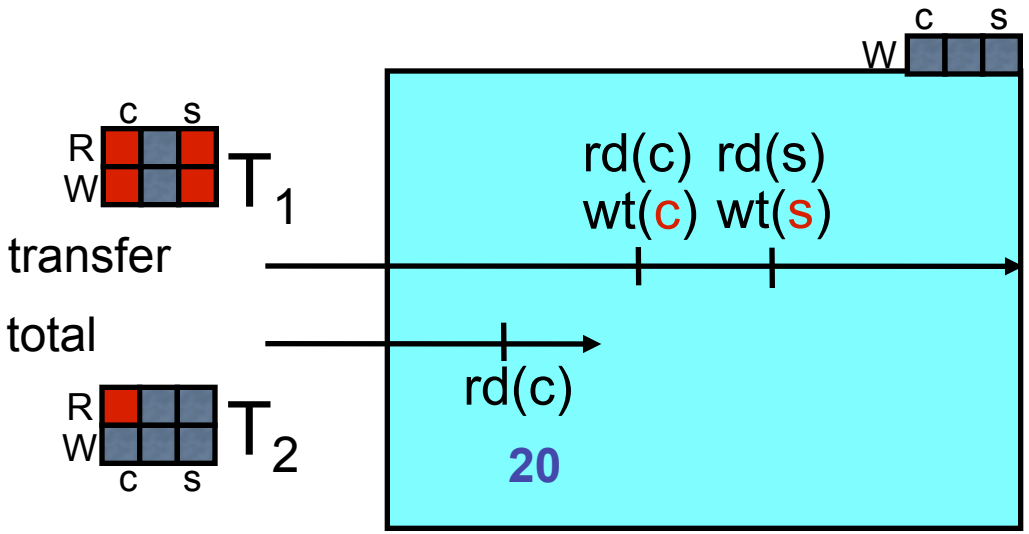
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```



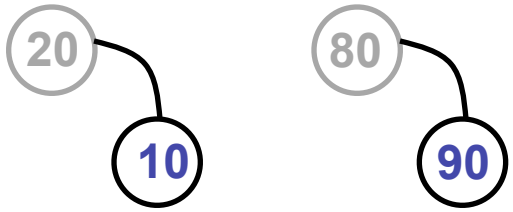

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



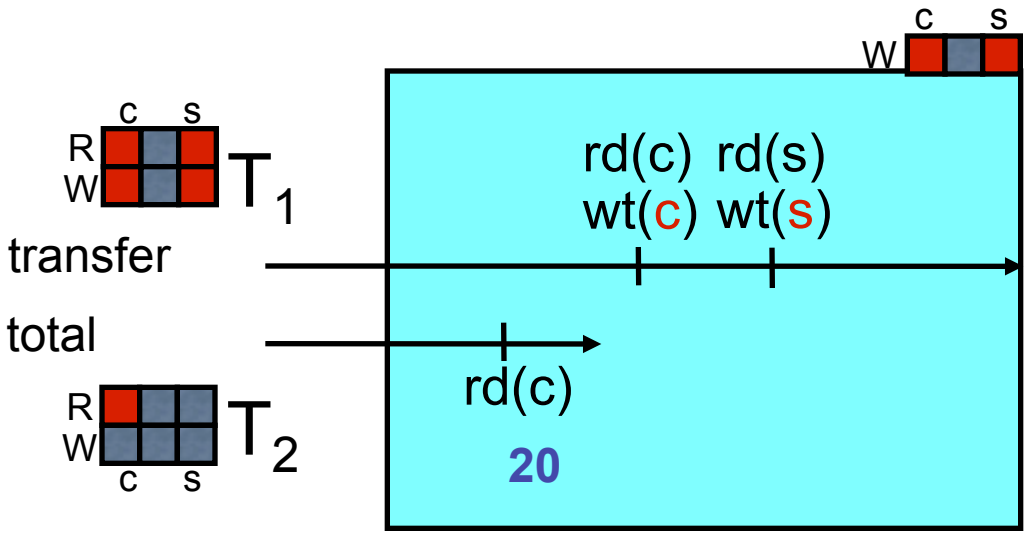
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```



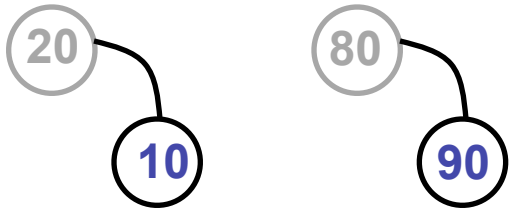

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



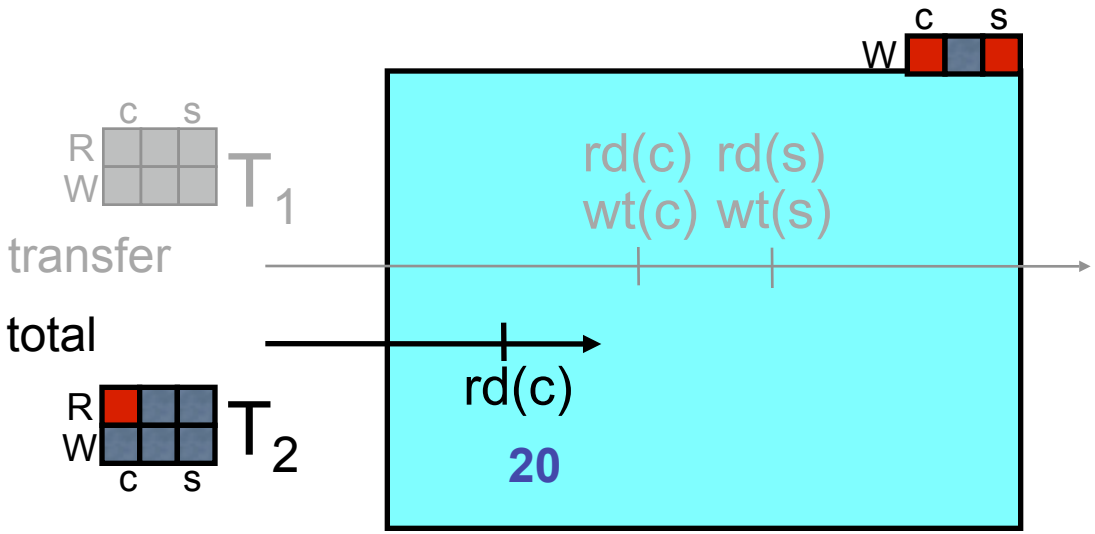
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```



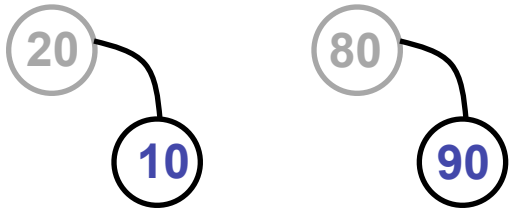

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



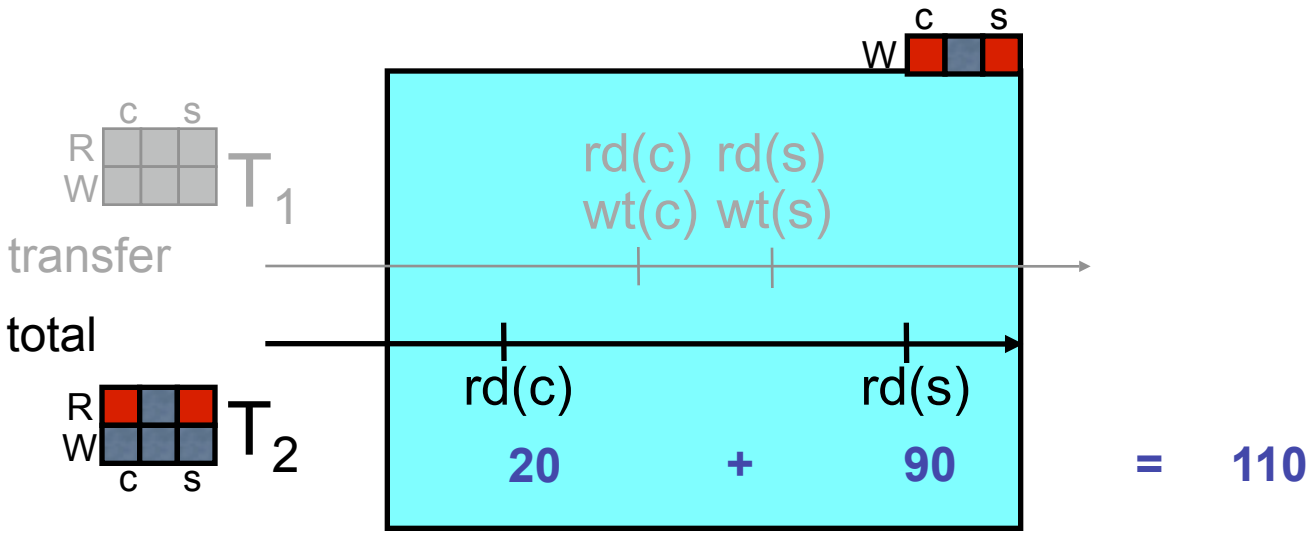
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```



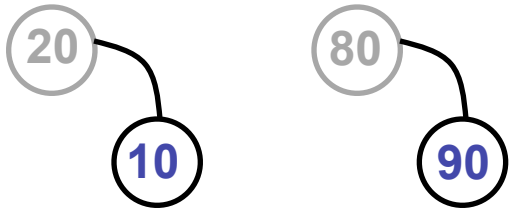

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



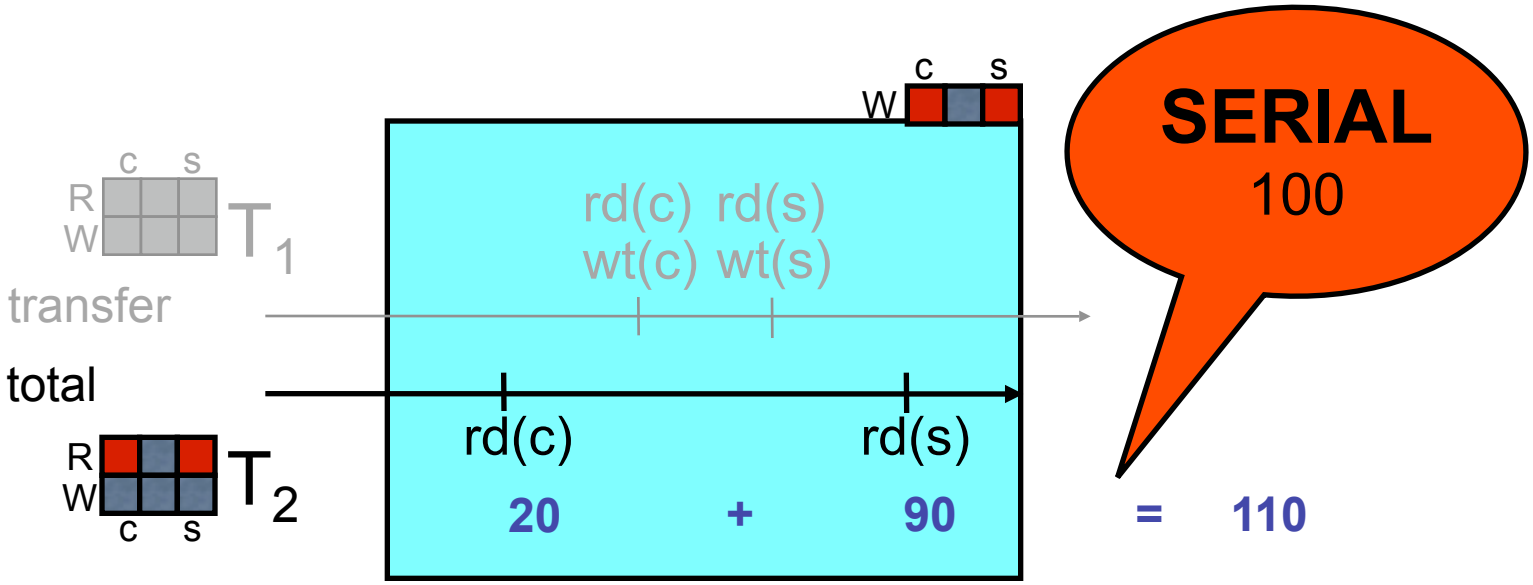
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```



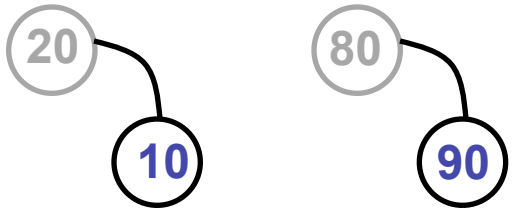

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



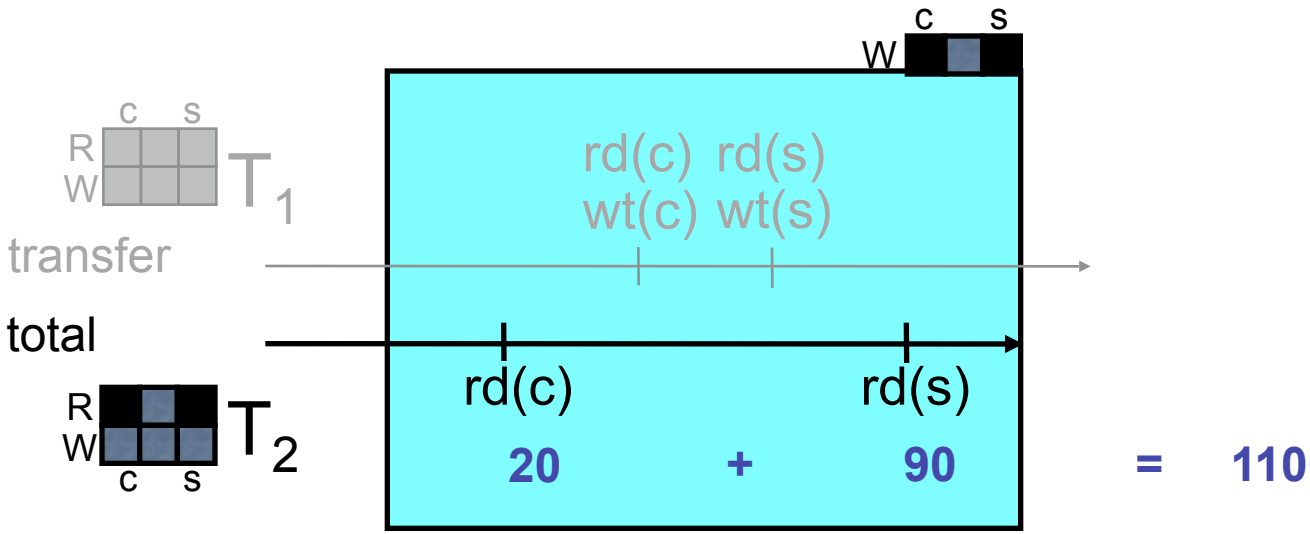
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```



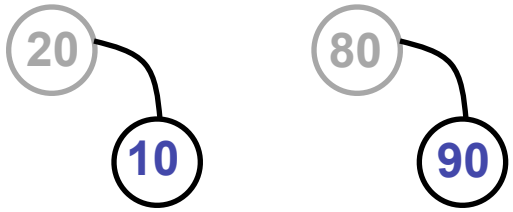

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



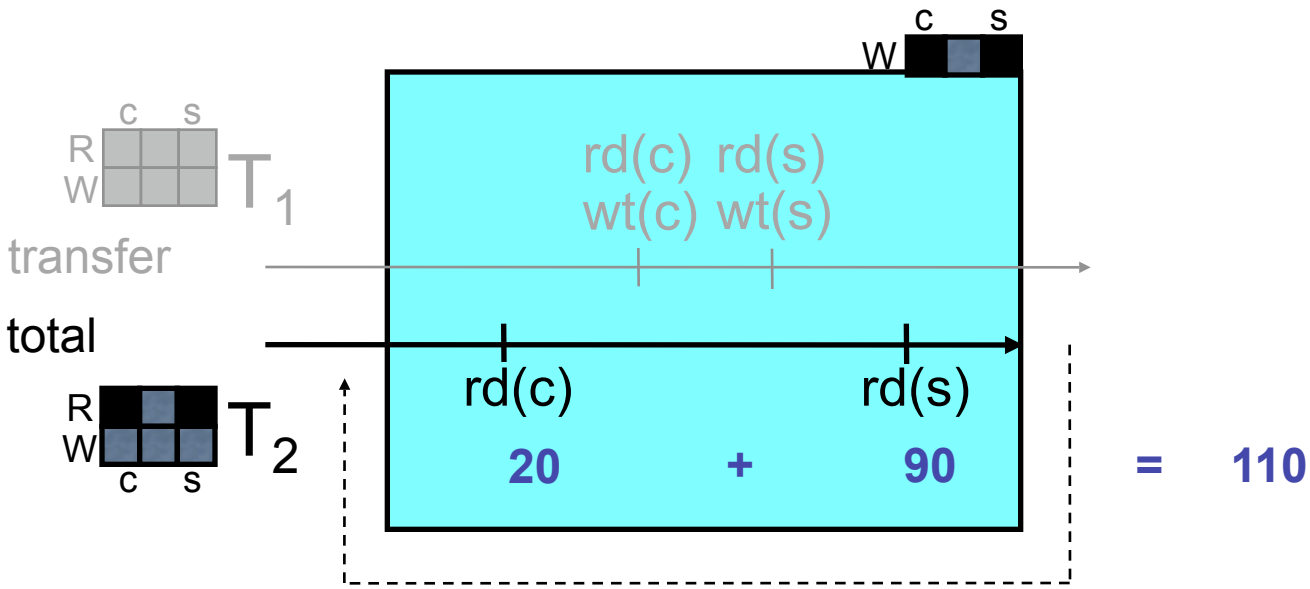
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```



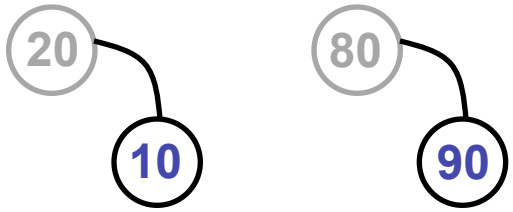

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



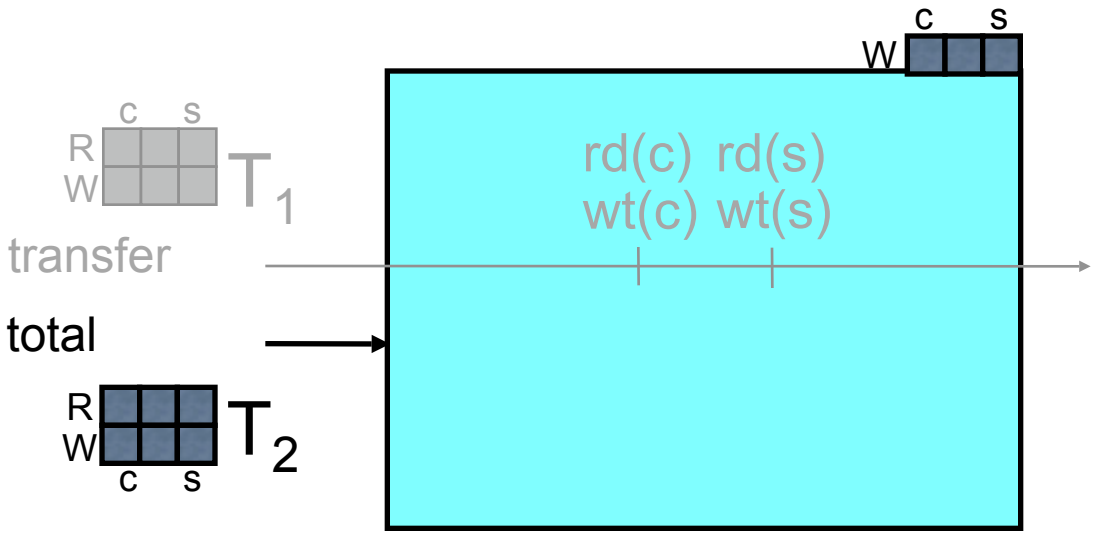
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```



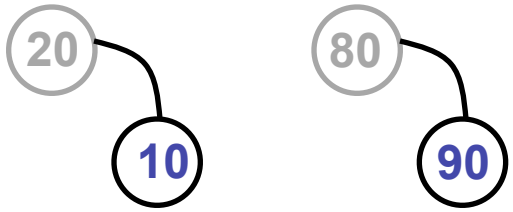

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



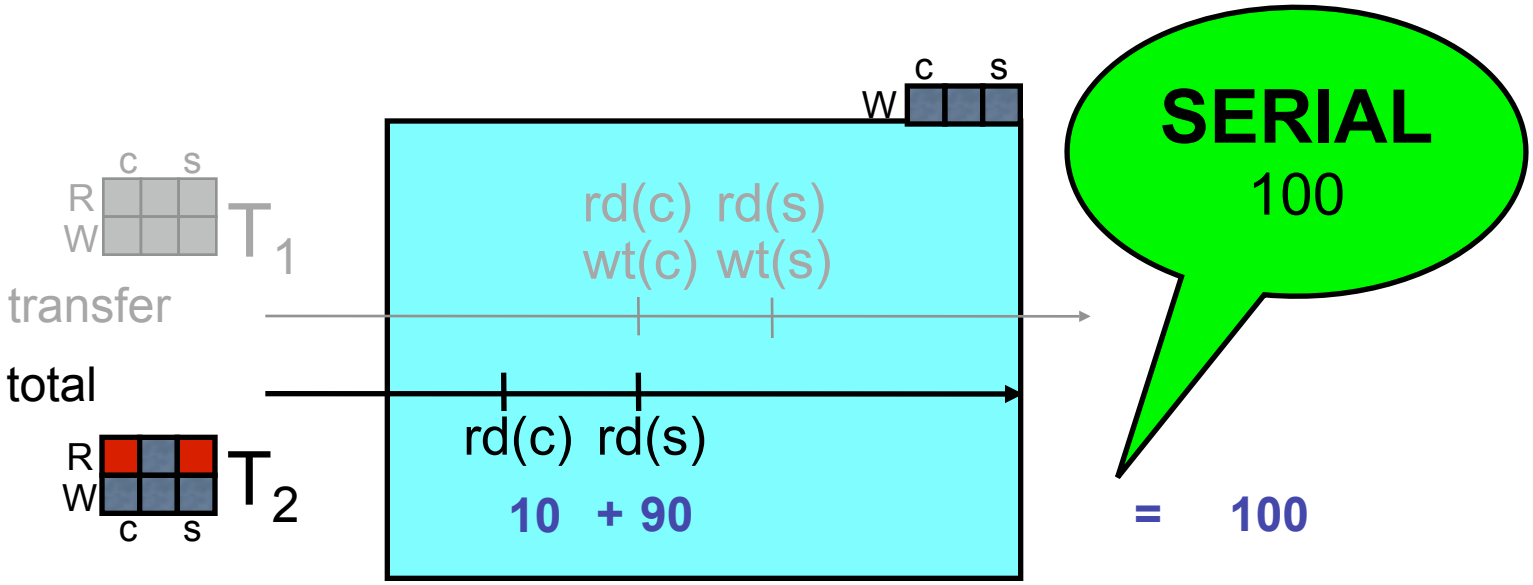
Ensuring Serializability

```
// checking   // savings
Account c;    Account s;
```




```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

10



Design and Implementation Choices

- Transactional memory (atomics) vs. transactional monitors:
  - ★ Using atomics provides stronger safety guarantees
    - ✦ Serializability with respect to all concurrently executing transactions
  - ★ Transactional monitors more closely mirror lock-based programming methodology
- When do writes become visible to the global store?
  - ★ Log writes locally, and update only on commit (redo)
  - ★ Update globally, and revert on abort (undo)
- Should writers witness readers?
  - ★ Visible vs. invisible reads
  - ★ Influences contention management
  - ★ How aggressively should readers be aborted?

Observations

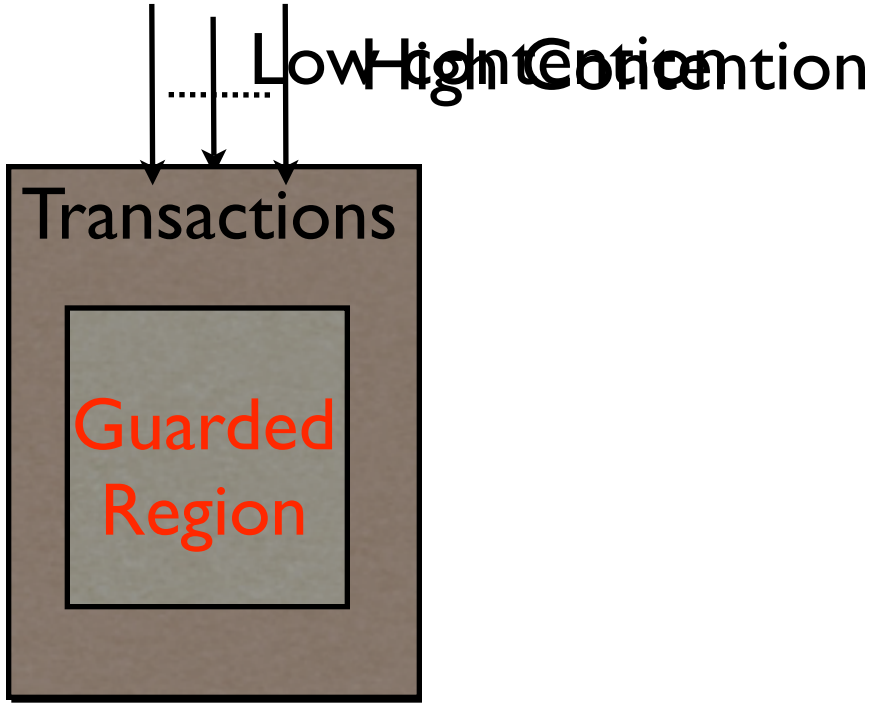
- Classical lock-based approaches to coordinating activities of multiple threads:
  - ★ Impose a heavy burden on programmer to balance safety and performance.
  - ★ Have well-known issues with deadlocks, data races, priority inversion, interaction with external actions, etc.
  - ★ Scalability impacted by the use of mutual-exclusion.
- But ...
  - ★ There is much legacy code (e.g., libraries) that use locks.
  - ★ Well-known tuned implementations.
    - ✦ Thin locks.

Observations

- Software transactions:
  - ★Enforce atomicity and isolation on the regions they protect:
    - ✦ Atomicity: actions within a transaction appear to execute all-at-once or not-at-all.
    - ✦ Isolation: effects of other threads are not witnessed once a transaction starts.
  - ★Conceptually simple programming model
- But ...
  - ★ More complicated implementation model.
    - ✦ Must track atomicity and isolation violations at runtime.
    - ✦ Revocation of effects when violations occur not always possible.
    - ✦ Performance benefit only in the presence of contention.

Reconcilation

- Hybrid Approach:
  - ★ Enforce atomicity and isolation properties using locks when contention is low or when transactional semantics is undesirable or infeasible.
  - ★Enforce these properties using transactions when contention is high and when transactional semantics is sensible.



Goals

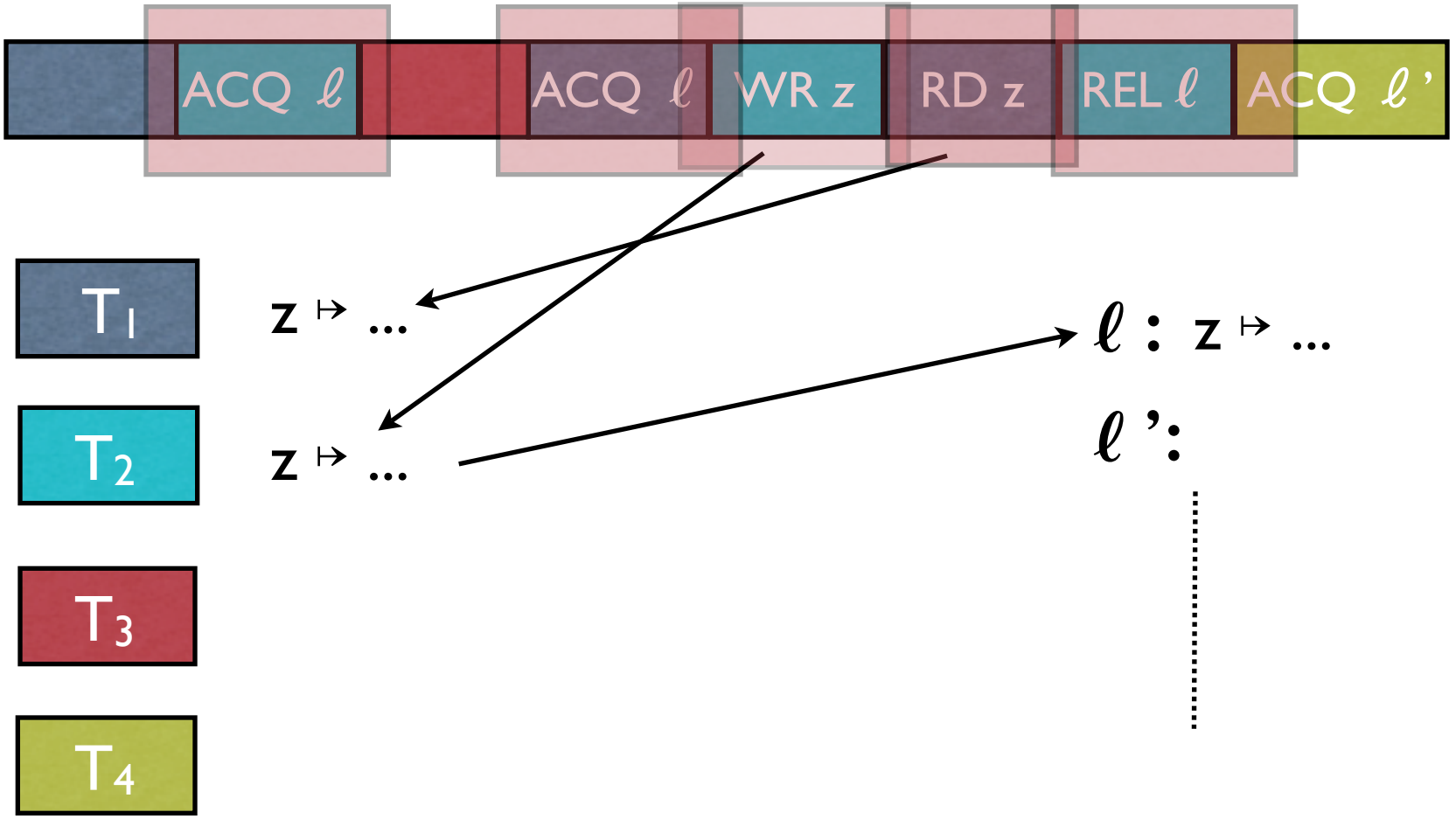
- Protocol choice must be transparent to applications.
  - ★Applications continue to use existing synchronization primitives.
- Transparency does not come at the expense of correctness.
  - ★ Program behavior must not depend on how a guarded region is executed.
  - ★Must work in the presence of nested guarded regions.
- Performance.
  - ★No performance degradation when contention is low.
  - ★Performance improvement when contention is high.

Correctness

- *When is it safe to use hybrid execution?*
- Semantics
  - ★Define a two-tiered execution model:
    - ✦ First tier defines data visibility (memory model) and interleaving
      - ◗ Schedules
      - ◗ Does not define a concurrency control protocol
    - ✦ Second tier defines safety properties on schedules with respect to a specific concurrency control protocol.

Semantics

Schedules



Constraints

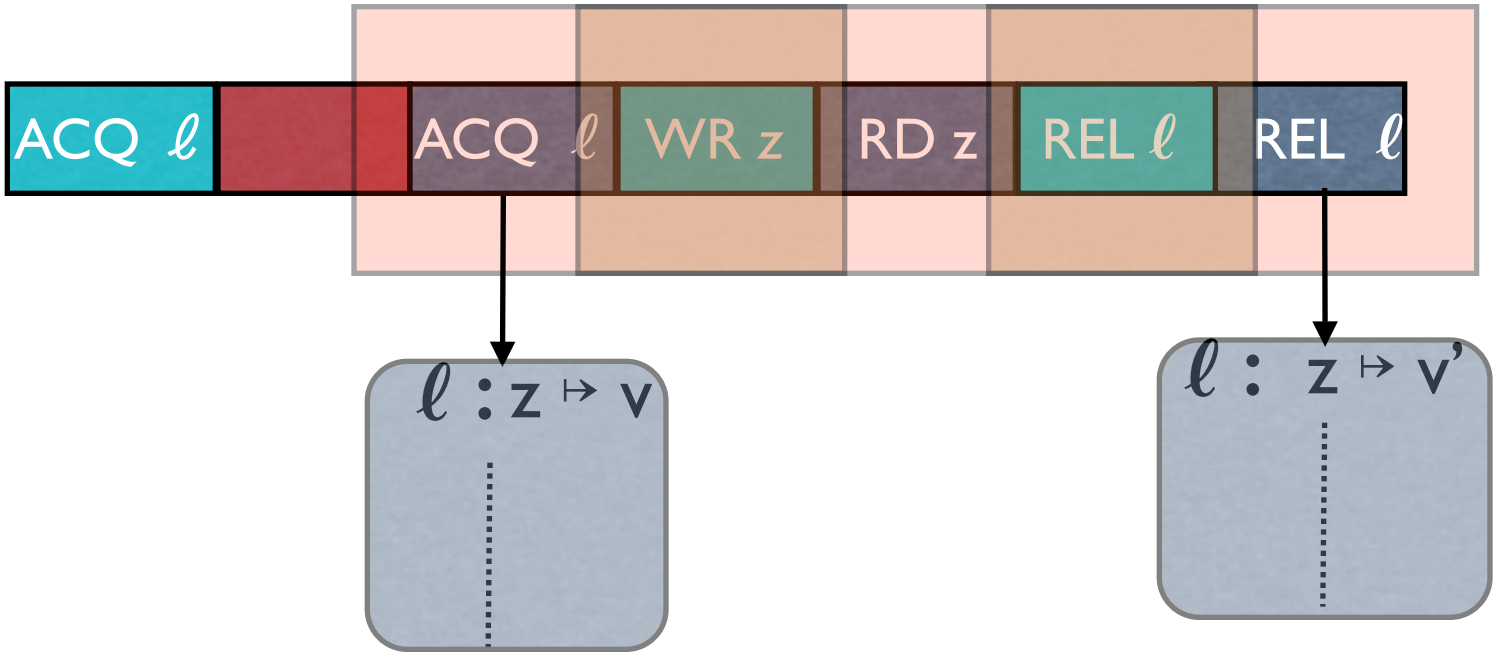
- Impose constraints on schedules to derive specific concurrency protocols.
- Mutual Exclusion: (M-safe schedules)

Multiple threads cannot concurrently execute within the body of a guarded region.

Does not enforce atomicity.

Transactional Constraints

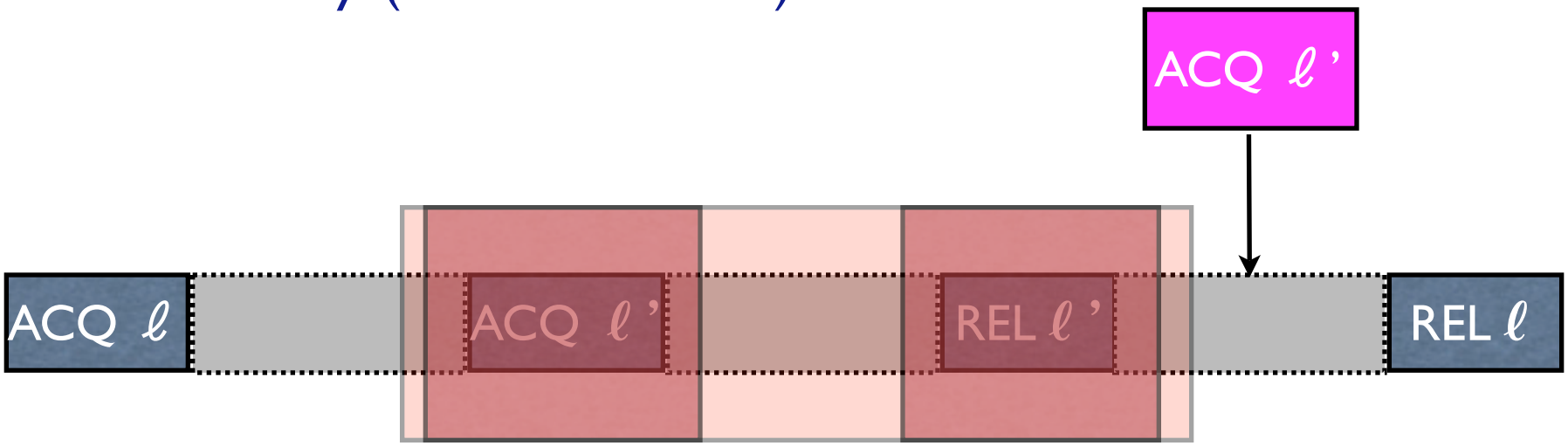
- Isolation: (I-safe schedules)



A non-isolated schedule

Transactional Constraints

- Atomicity: (A-safe schedules)



A non-atomic schedule

Safety

- Any schedule which is both *i-safe* and *a-safe* can be permuted to one which is *m-safe* without change in observable behavior.
  - ★ Can treat synchronized blocks as closed nested transactions in Java programs with *i-safe* and *a-safe* schedules without modifying existing Java semantics.
  - ★ Closed nesting: the effect of a nested synchronized block *B* executed transactionally becomes visible to other transactions only when *B*'s outermost transaction commits.

Design

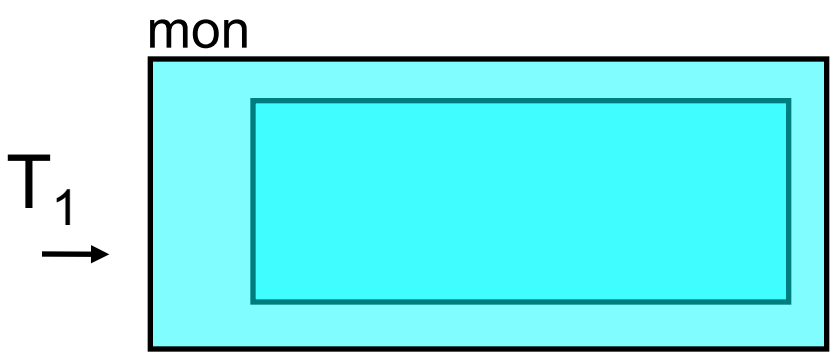
- Consider programs whose generated schedules are *i-safe* and *a-safe*.
  - ★ Execute synchronized blocks and methods
    - ✦ Transactionally, when contention is high.
    - ✦ Serially, when contention is low.
- Closed nested transaction model.
  - ★ Performance challenge
    - ✦ Each monitor defines a locus of contention.
    - ✦ Non-trivial overhead to maintain meta-data to validate transaction safety.
    - ✦ Consider optimizations to reduce this overhead.
      - ▸ Delegate meta-data management from a nested transaction to its parent.

Delegation

T_1

```
synchronized (mon)
 { acc.transfer(10) }
```

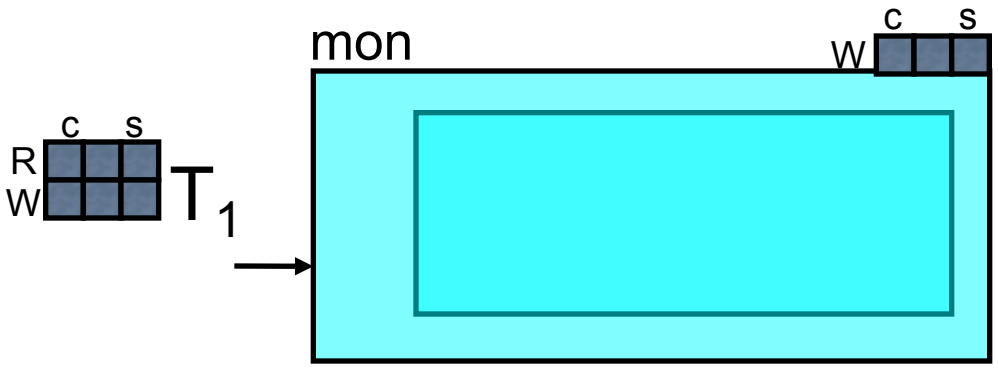
```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



Delegation

T_1 | synchronized (mon) { acc.transfer(10) }

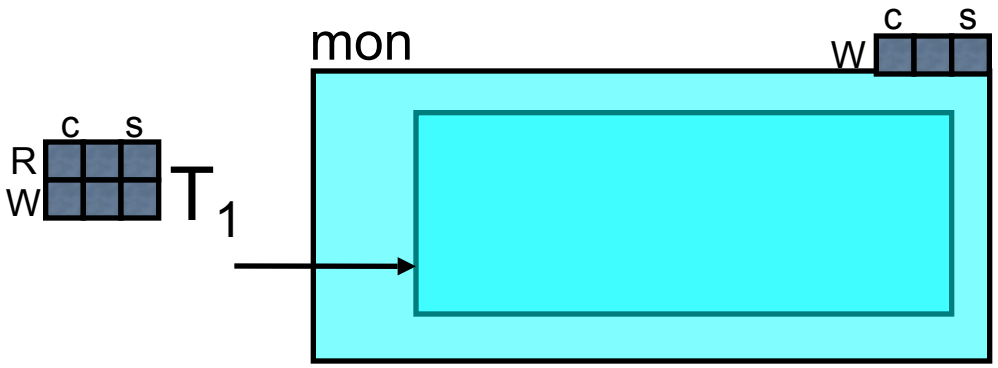
```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



Delegation

T_1 | synchronized (mon) { acc.transfer(10) }

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

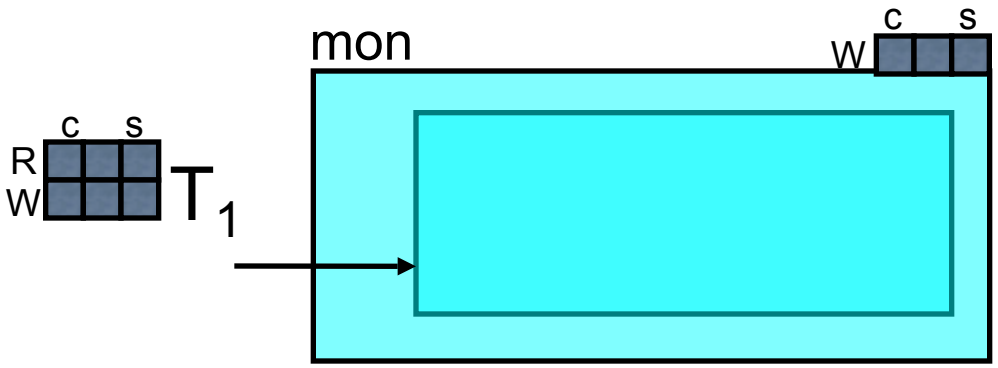


Delegation

T_1 synchronized (mon) { acc.transfer(10) }

T_2 acc.total()

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

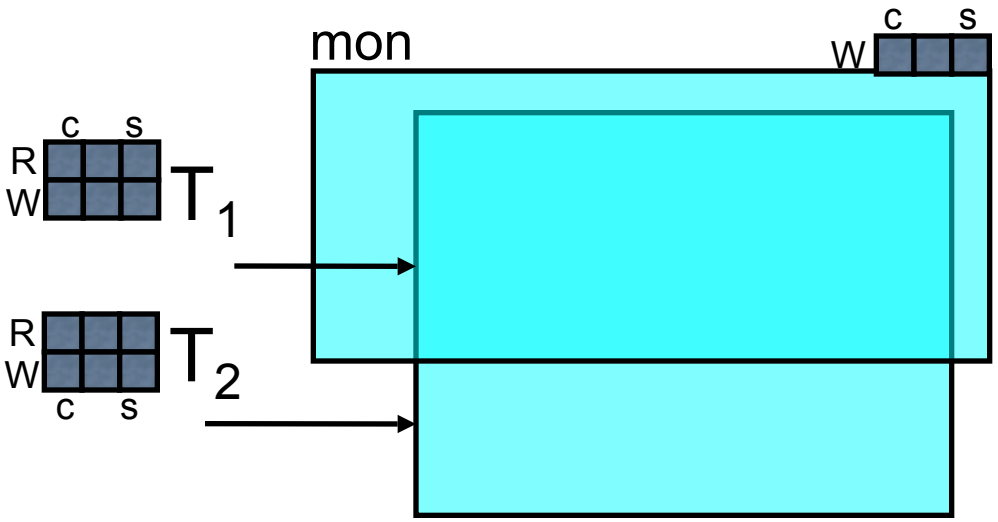


Delegation

T_1 synchronized (mon) { acc.transfer(10) }

T_2 acc.total()

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

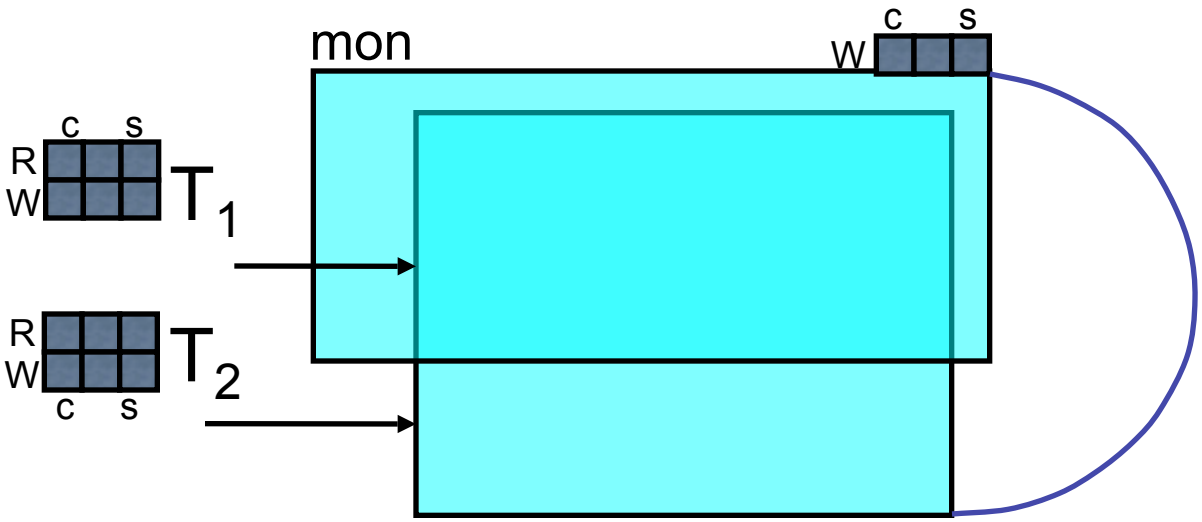


Delegation

T_1 synchronized (mon) { acc.transfer(10) }

T_2 acc.total()

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

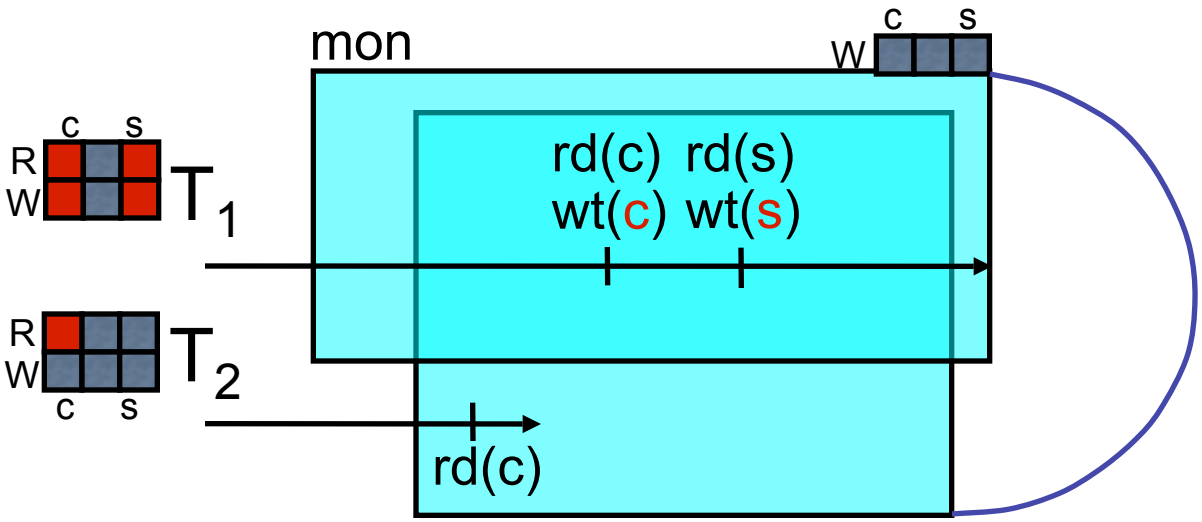


Delegation

T_1 synchronized (mon) { acc.transfer(10) }

T_2 acc.total()

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

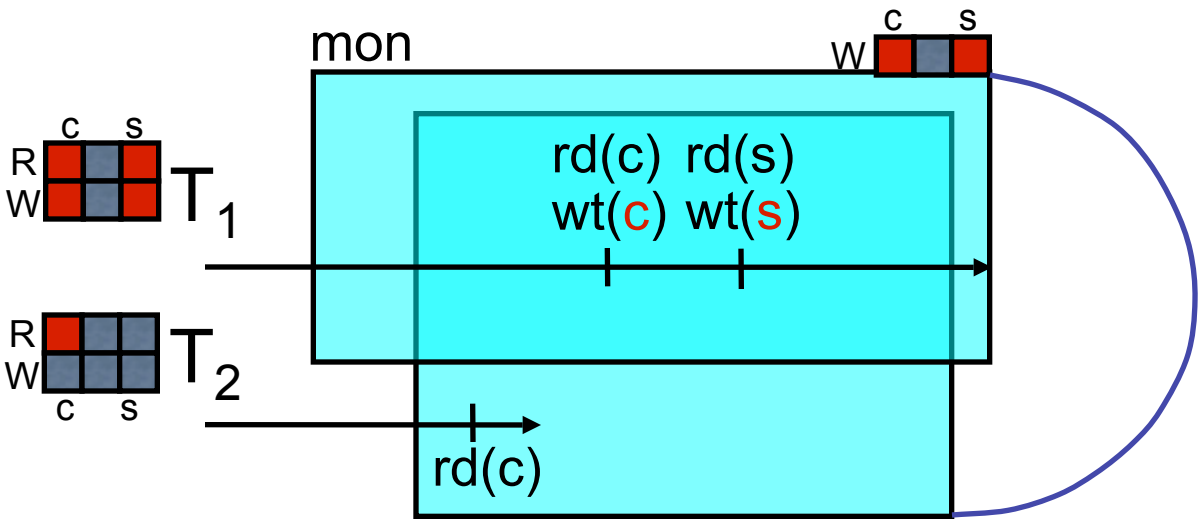


Delegation

T_1 synchronized (mon) { acc.transfer(10) }

T_2 acc.total()

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

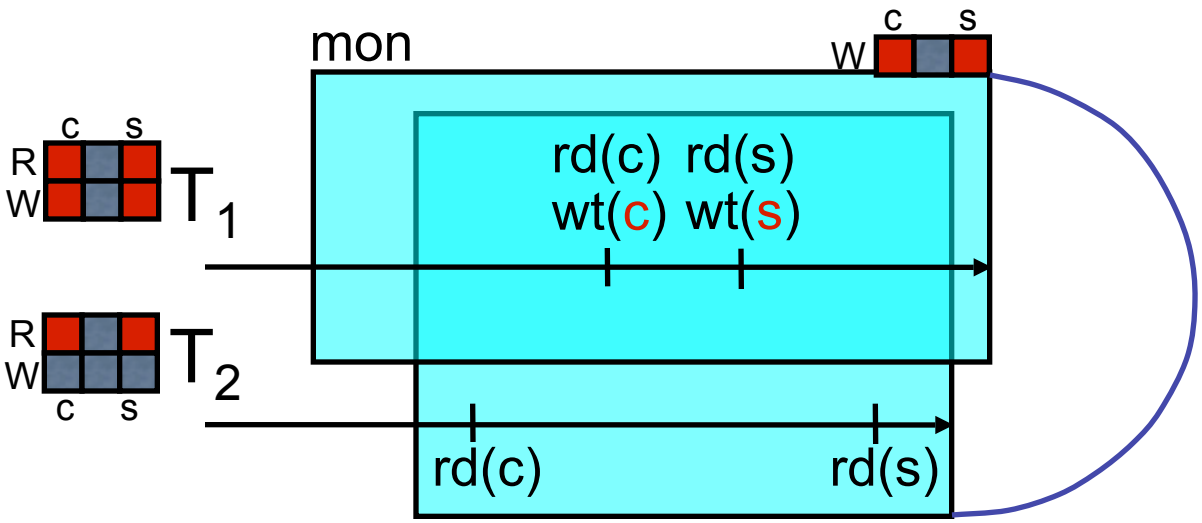


Delegation

T_1 synchronized (mon) { acc.transfer(10) }

T_2 acc.total()

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```

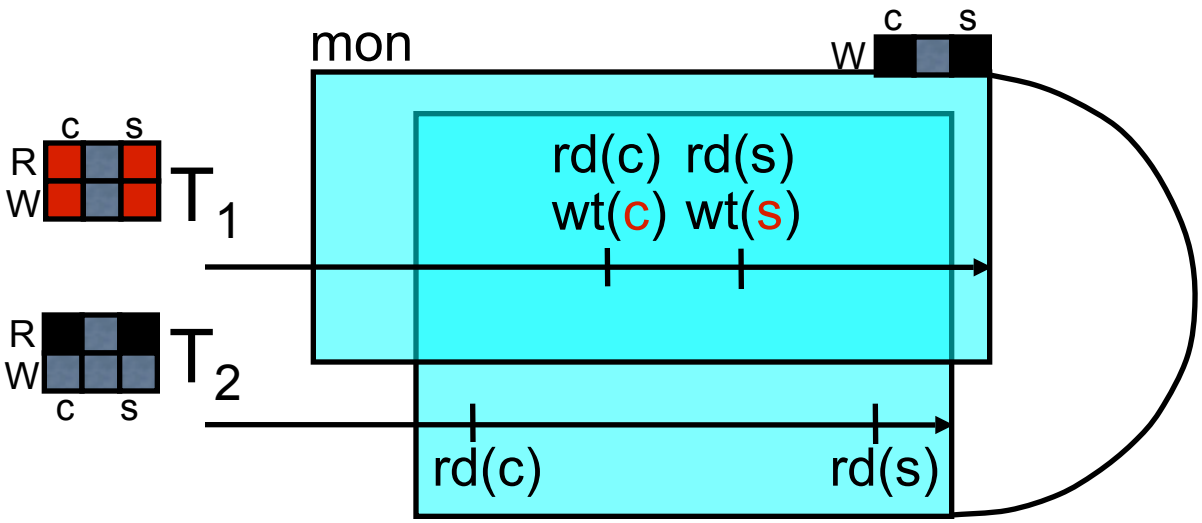


Delegation

T_1 synchronized (mon) { acc.transfer(10) }

T_2 acc.total()

```
void synchronized transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float synchronized total ()
  {  return c.balance()+s.balance();  }
```



Delegation Summary

- Optimized version of closed nested transactions
- Setting a delegate – inexpensive
- Only delegate setting required in non-contended case
- Potential for lowering overhead related to nesting even if monitors contended

Mutual Exclusion

- When should transactional execution switch to mutual exclusion?
  - ★Native methods (e.g., I/O)
  - ★Explicit thread synchronization (wait/notify)
  - ★*Absence of contention*
- All parent monitors must be re-acquired in mutual exclusion mode.

Implementation

- Optimistic protocol for reads
- Pessimistic protocol for writes
  - ★Prevent multiple writers to the same object
- Validation phase
  - ★Enforce *i-safe* and *a-safe* constraints
  - ★Discard copies if safety is violated
- Write-back
  - ★Lazily propagate updated copies to the shared heap.
- Implementation in Jikes RVM
  - ★Use read and write barriers to
    - ✦ Create versions
    - ✦ Redirect reads to the appropriate version
    - ✦ Track data dependencies using read/write hash maps

Overheads

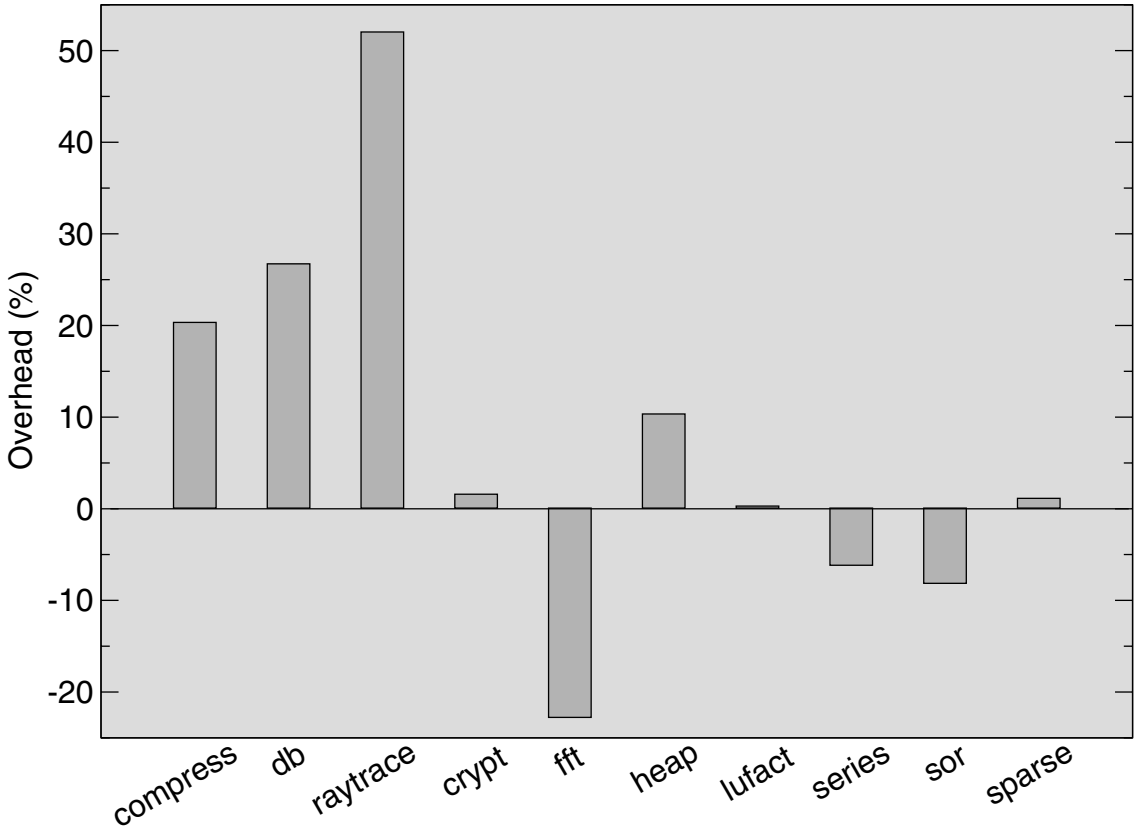
Sources of overhead

- ★Object header expansion
  - ✦ meta-data necessary to enforce transaction safety
    - ◗ forwarding pointers, delegates, hash codes, etc.
- ★Code duplication
  - ✦ Two versions for each method
  - ✦ Still need (fast) read barriers even on non-transactional paths
    - ◗ Access latest version of an object
- ★Triggering transactional execution
  - ✦ Lightweight heuristic to measure contention
    - ◗ Trigger transactional execution when thin-lock is inflated and more than one thread is waiting when locking thread exits.

Barrier Optimizations

- **Goal:** omit barriers on loads of primitive values
- **Problem:** accesses through stale on-stack references
- **Solution:** update references on stack using modified GC stack scanning procedure
  - ★At version creation
    - ✦ eager
  - ★At pre-specified memory"synchronization" points
    - ✦ monitor entry
    - ✦ access to volatile variables
    - ✦ wait/notify operations

Performance: Uncontended Execution



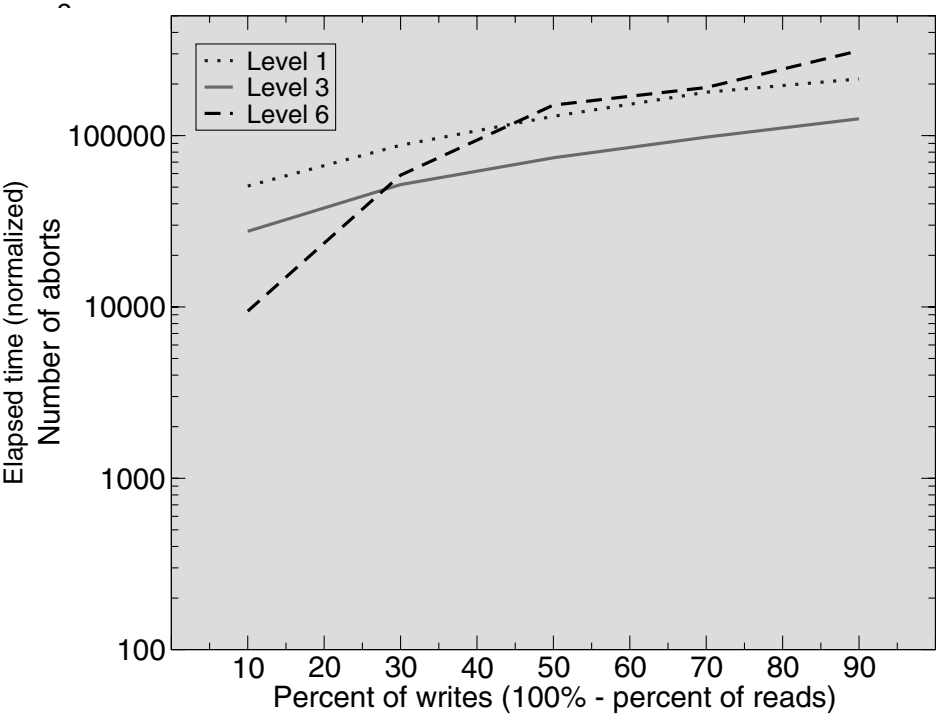
Single-threaded Specjvm98 and Java Grande benchmarks

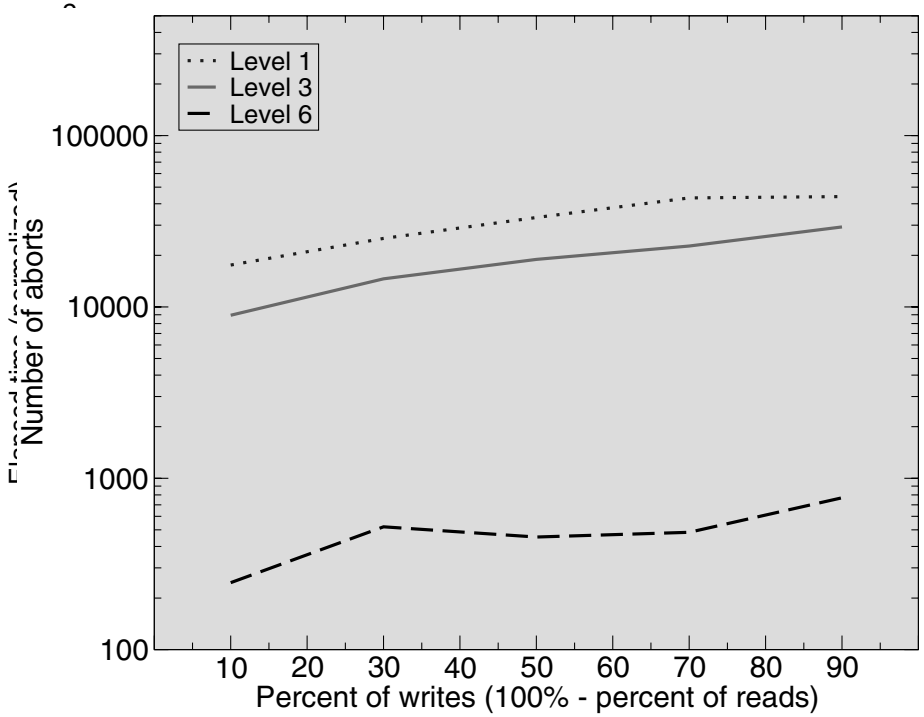
Barriers are primary source of overheads

7% average but large variance

Costs can be significantly reduced through simple compiler optimizations

Performance: Contended Execution





007, a tunable concurrent database benchmark

64 threads, 8 processors

Hybrid execution more resilient to write-biased workloads

Summary

- Effective support for transactions involves efficient implementation of a number of complex actions:
  - ★ logging and copying data to restore program state
  - ★ fast consistency checks to determine if serialization invariants are violated
  - ★ revert thread control-flow to earlier program point in case of abort
- Interaction with other realistic language features add further complications:
  - ★ irrevocable actions (e.g, I/O)
  - ★ native method calls
  - ★ interaction with other concurrency mechanisms (e.g., wait/notify, locks)
  - ★ language memory model and execution semantics
- Can we selectively pick aspects of this implementation space to address other interesting concurrency issues?

A Transactional Calculus

- TFJ is a *concurrent*, *imperative* object calculus with dynamically scoped transactions: onacid and commit
- TFJ supports *multi-threaded* and *nested* transactions

$$
\begin{array}{lcl}
\texttt{P} & ::= & \texttt{0} \mid \texttt{P}|\texttt{P} \mid \boxed{\texttt{t[e]}} \\
\texttt{L} & ::= & \texttt{class C}\,\{\,\overline{\texttt{f}};\overline{\texttt{M}}\,\} \\
\texttt{M} & ::= & \texttt{m}(\overline{\texttt{x}})\,\{\texttt{return e;}\,\} \\
\texttt{e} & ::= & \texttt{x} \mid \texttt{this} \mid \boxed{\texttt{v}} \mid \texttt{e.f} \mid \texttt{e.m}(\overline{\texttt{e}}) \mid \boxed{\texttt{e.f} := \texttt{e}} \mid \\
 & & \texttt{new C}() \mid \boxed{\texttt{spawn e} \mid \texttt{onacid} \mid \texttt{commit}} \mid \texttt{null}
\end{array}
$$

Semantics

- Two-level operational semantics,
- Semantics parameterized by definition of core transactional operations write, read, reflect, commit, spawn
- Labeled reduction relation $\Gamma\ P \xRightarrow{\alpha}_{t} \Gamma'\ P'$

wr	v u	*write*
rd	v	*read*
xt	v	*new*
ac		*start transaction*
co		*commit transaction*
sp		*spawn thread*

Γ is a program state composed of a sequence of thread environments

$t, \mathcal{E}$ associates a thread with its transaction environment

A transaction environment associates a transaction label with a binding environment or log

Read/Write

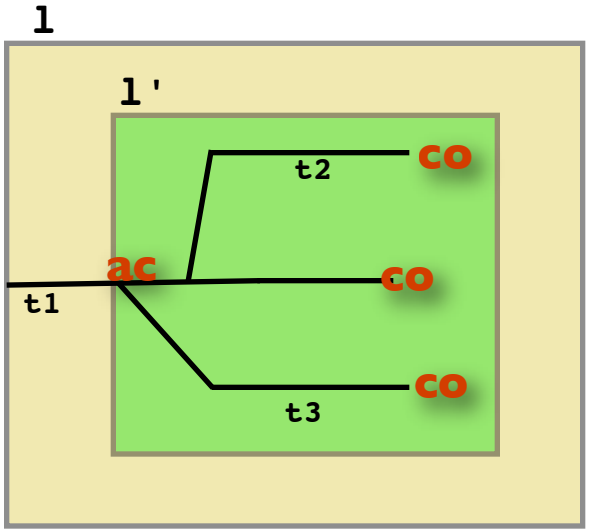
$$\frac{\mathcal{E}', \mathtt{C}(\overline{\mathtt{u}}) = \mathit{read}(\mathtt{v}, \mathcal{E}) \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{f}})}{\mathcal{E}\ \mathtt{v.f_i} \xrightarrow{\mathit{rd}\ \mathtt{v}} \mathcal{E}'\ \mathtt{u_i}} \qquad \textbf{(R-FIELD)}$$

$$\frac{\mathcal{E}', \mathtt{C}(\overline{\mathtt{v}}) = \mathit{read}(\mathtt{v}, \mathcal{E}) \quad \mathcal{E}'' = \mathit{write}(\mathtt{v} \mapsto \mathtt{C}(\overline{\mathtt{v}})\downarrow_{\mathtt{i}}^{\mathtt{v}'}, \mathcal{E}')}{\mathcal{E}\ \mathtt{v.f_i} := \mathtt{v}' \xrightarrow{\mathit{wr}\ \mathtt{v}\mathtt{v}'} \mathcal{E}''\ \mathtt{v}'} \qquad \textbf{(R-ASSIGN)}$$

$$\frac{\boldsymbol{P} = \boldsymbol{P}'' \mid \mathtt{t}[\mathtt{e}] \qquad \mathcal{E}\ \mathtt{e} \xrightarrow{\alpha} \mathcal{E}'\ \mathtt{e}' \qquad \boldsymbol{P}' = \boldsymbol{P}'' \mid \mathtt{t}[\mathtt{e}'] \qquad \Gamma = \mathtt{t}, \mathcal{E} \,.\, \Gamma'' \qquad \Gamma' = \mathit{reflect}(\mathtt{t}, \mathcal{E}', \Gamma'')}{\Gamma\ \boldsymbol{P} \xrightarrow{\alpha}_{\mathtt{t}} \Gamma'\ \boldsymbol{P}'} \qquad \textbf{(G-PLAIN)}$$

Commit

Concurrent threads within a transaction synchronize on commit

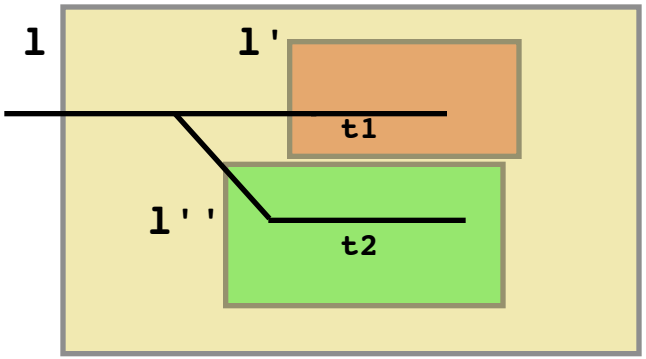


$$\frac{P = P'' \mid \overline{\mathtt{t}[\mathtt{e}]} \qquad \overline{\mathtt{e}} \Downarrow_{\mathtt{commit}} \overline{\mathtt{e}'} \qquad P' = P'' \mid \overline{\mathtt{t}[\mathtt{e}']} \qquad \overline{\mathtt{t}} = \mathit{intranse}(\mathtt{l}, \Gamma) \qquad \Gamma' = \mathit{commit}(\overline{\mathtt{t}}, \mathcal{E}, \Gamma)}{\Gamma\, P \xrightarrow{\mathit{co}}_{\mathtt{t}'} \Gamma'\, P'} \quad \textbf{(G-COMM)}$$

Optimistic Semantics

Per-thread environments as sequences of transaction logs

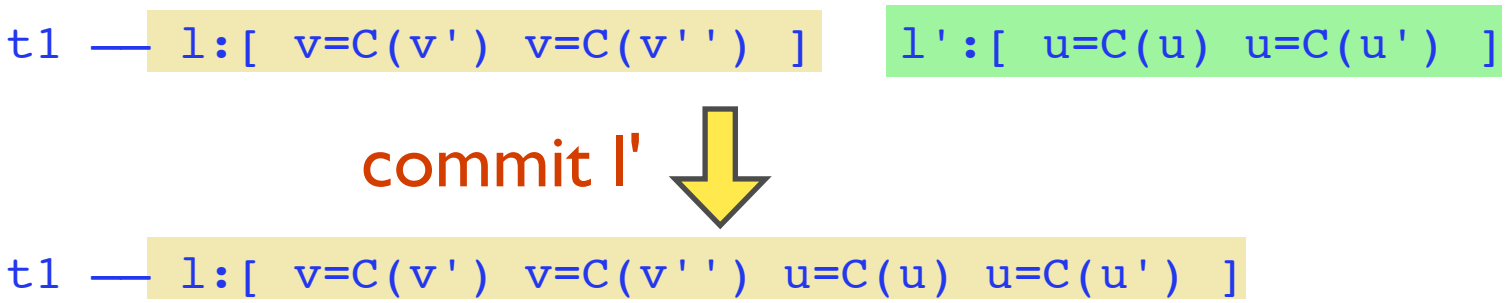
- read adds the object read to the issuing thread's current transaction log
- write adds the new value
- reflect propagates changes from one thread environment to all other threads in the same transaction




```
t1 ⟶ l:[ v=C(v'), v=C(v'')]   l':[ u=C(u) ]
t2 ⟶ l:[ v=C(v'), v=C(v'')]   l'':[ v=C(v'') ]
```

Commit

Commit copies the log of the current transaction into the directly enclosing one



Succeeds if all values read are still current in the enclosing environment

Pessimistic Semantics

2 phase locking:

acquire a lock before reading and writing.

release before commit

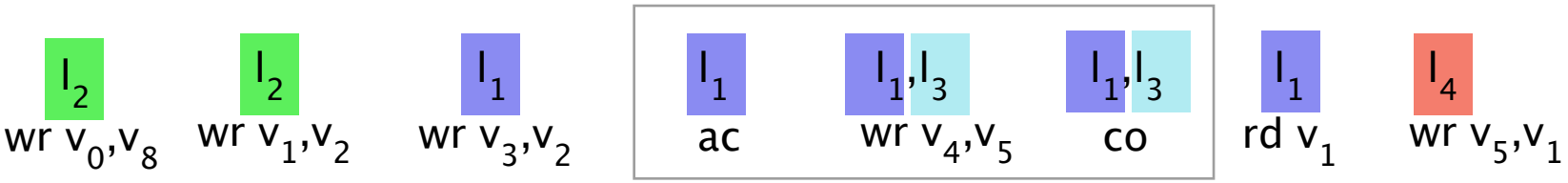
Define a lock environment that maps a lock to the transaction label sequence that specifies the transaction that currently holds it.

$$\frac{\mathcal{E} = \mathcal{E}' \,.\, \mathtt{l}{:}\rho \quad \mathit{findlast}(\mathtt{r}, \mathcal{E}) = \mathtt{C}(\overline{\mathtt{r}}) \quad \mathcal{E}'' = \mathcal{E}' \,.\, \mathtt{l}{:}(\rho \,.\, \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}})) \quad \mathit{checklock}(\mathtt{r}, \mathcal{E}) = \mathtt{true}}{\mathit{read}(\mathtt{r}, \mathcal{E}) \;=\; \mathcal{E}'', \mathtt{C}(\overline{\mathtt{r}})}$$

$$\frac{\mathit{findlast}(\mathtt{r}, \mathcal{E}) = \mathtt{D}(\overline{\mathtt{u}}) \quad \mathcal{E}' = \mathit{acquirelock}(\mathtt{r}, \mathcal{E}) \quad \mathcal{E}'' = \mathcal{E}' \,.\, \mathtt{l}{:}\rho \quad \mathcal{E}''' = \mathcal{E}'' \,.\, \mathtt{l}{:}(\rho \,.\, \mathtt{r} \mapsto \mathtt{D}(\overline{\mathtt{u}}) \,.\, \mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}))}{\mathit{write}(\mathtt{r} \mapsto \mathtt{C}(\overline{\mathtt{r}}), \mathcal{E}) = \mathcal{E}'''}$$

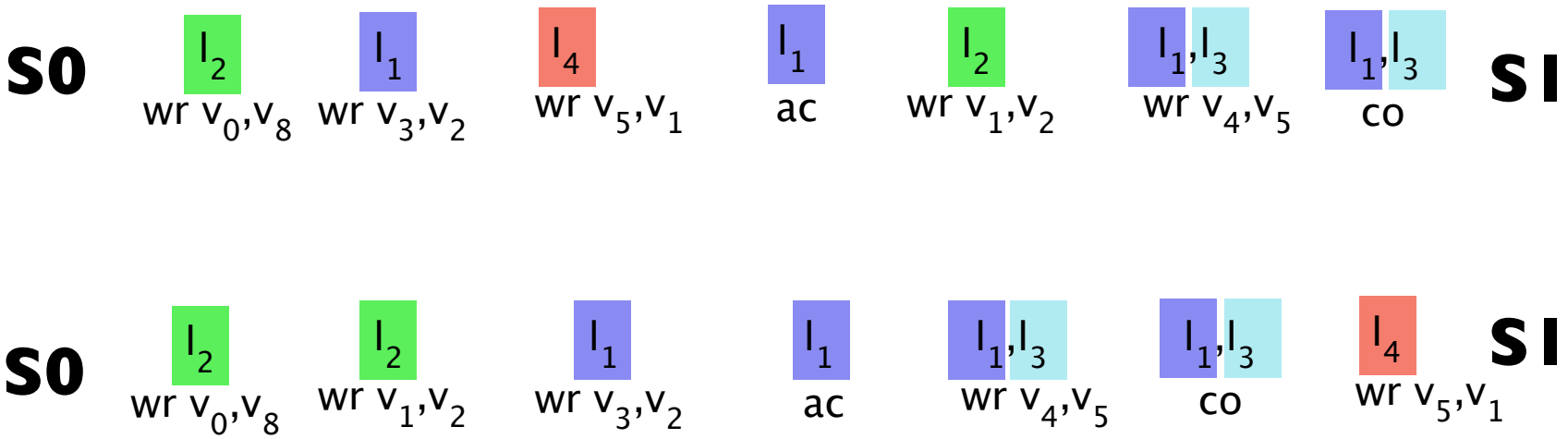
Serial Trace

- A program trace is serial if for all pairs of reductions steps taken by a transaction L, steps occurring between them are taken on behalf of L or transactions nested within L



Soundness

The soundness theorem states that for any trace R, there is an equivalent serial trace R'



Dependencies

Control and data dependencies induce a partial order on actions used to structure transaction traces

	rd t' l'	wr t' l'	xt t' l'	sp t' l'	ac t' l'	co t' l'
rd t l				t = t'	l' < l	
wr t l				t = t'	l' < l	
xt t l				t = t'	l' < l	
sp t l				t = t'	l' < l	
ac t l				t = t'	l' < l	
co t l	l' < l	l' < l	l' < l	t = t'	l' < l	

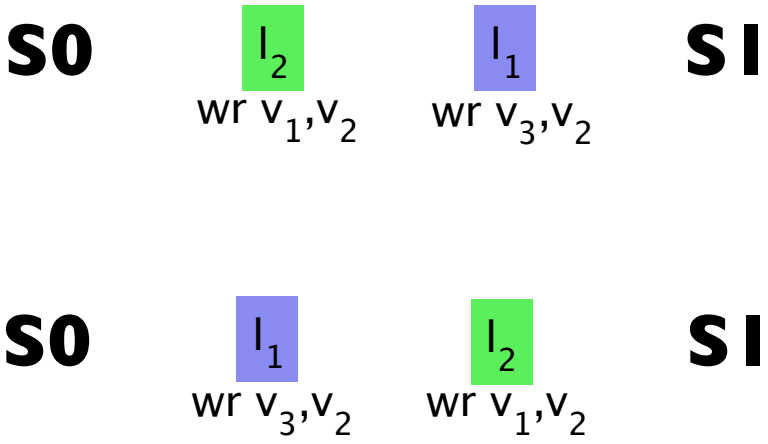
Control

	wr v' u' l'	rd v' l'	xt v' l'
wr v u l	u = v' & l' < l v = v' & l' < l		u = v' & l' < l v = v' & l' < l
rd v l	v = v' & l' < l		v = v' & l' < l
xt v l	v = v' & l' < l		v = v' & l' < l

Data

Permutation

- The key property for proving soundness is the permutation lemma which states that two independent actions can be permuted. Actions are independent if they have no data or control dependency with one another.



Must be proved for each transaction semantics.

Case Study: Futures

If sequential program P is annotated with futures to yield concurrent program P_F, then the observable behavior of P is equivalent to P_F

- Logical serial order trivially satisfied when no side-effects
- Problems arise with mutation of shared data
- Consider futures API in JDK 1.5
- Like transactions, correct implementation of futures requires tracking dependencies
  - ★ But, constraints imposed are stronger: behavior must conform to a serial execution, not a serializable one
  - ★ Pairwise association of concurrent execution states
  - ★ No issues of livelock or deadlock. It is always safe to revert to sequential execution.
- Target applications are those which decompose into speculative units (with little to modest sharing)

Rationale

- Alternative concurrency model
  - ★ No explicit threads
  - ★ Concurrent program easily derived from its sequential counterpart
  - ★ No non-determinism
- Utility
  - ★ Concurrent program development and debugging
  - ★ Convenient way to define arbitrary regions of speculative code
- Best used when (strong notions of) safety dominate performance requirements

Safety Properties

- An access to a location *l* (either a read or write) performed by a future should not witness a write to *l* performed by its continuation.
- The last write to a location *l* performed by a future must occur before the first access to *l* by the continuation.
- How do we maintain these properties?
  - ★version shared data
  - ★track shared data dependencies
  - ★revoke non-serial execution
- These properties must hold even in the presence of exceptions, and irrevocable actions

Using Futures

```
float sum = acc.total();
acc.transfer(10);
print(sum);
```

```
void transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float total ()
  {  return c.balance()+s.balance();  }
```

Using Futures

```
Future f = F[acc.total()];
acc.transfer(10);
print(f.get());
```

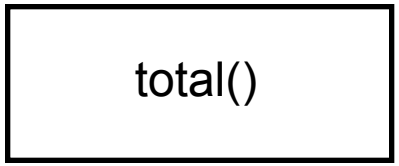
```
void transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float total ()
  {  return c.balance()+s.balance();  }
```

Using Futures

```
Future f = F[acc.total()];
acc.transfer(10);
print(f.get());
```

```
void transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float total ()
  {  return c.balance()+s.balance();  }
```

LOGICAL SERIAL ORDER:

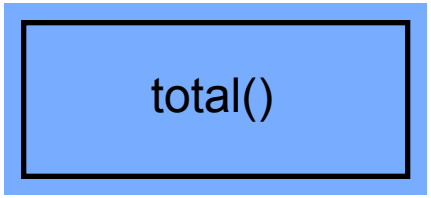


Using Futures

```
Future f = F[acc.total()];
acc.transfer(10);
print(f.get());
```

```
void transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float total ()
  {  return c.balance()+s.balance();  }
```

LOGICAL SERIAL ORDER:



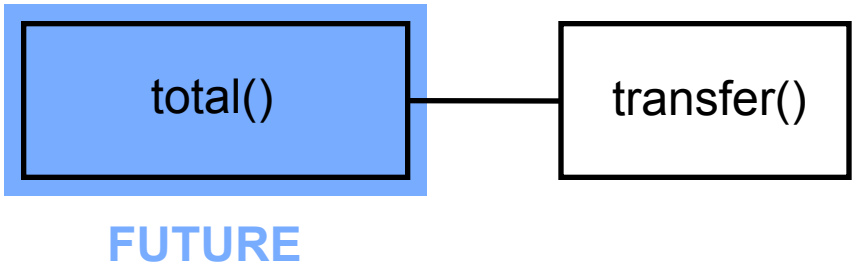
FUTURE

Using Futures

```
Future f = F[acc.total()];
acc.transfer(10);
print(f.get());
```

```
void transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float total ()
  {  return c.balance()+s.balance();  }
```

LOGICAL SERIAL ORDER:

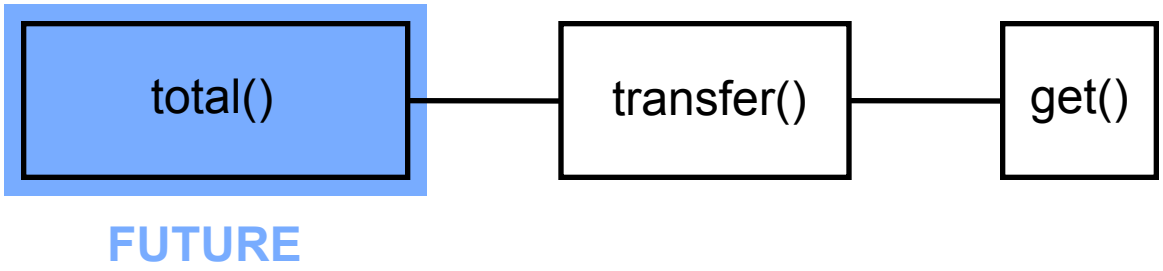


Using Futures

```
Future f = F[acc.total()];
acc.transfer(10);
print(f.get());
```

```
void transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float total ()
  {  return c.balance()+s.balance();  }
```

LOGICAL SERIAL ORDER:

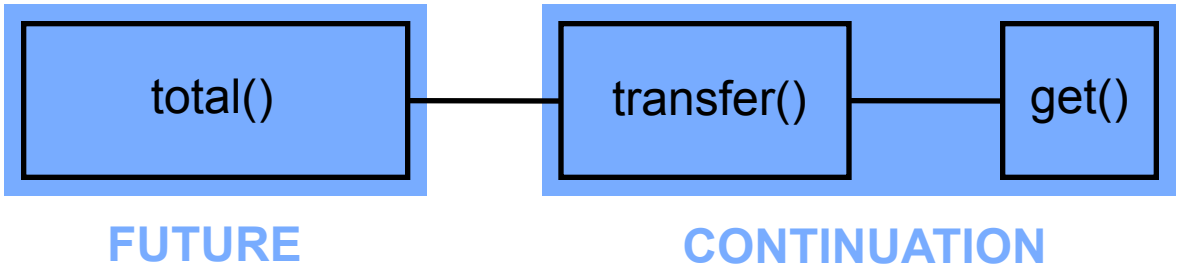


Using Futures

```
Future f = F[acc.total()];
acc.transfer(10);
print(f.get());
```

```
void transfer (int sum)
 {  c.withdraw(sum);
    s.deposit(sum);  }

float total ()
  {  return c.balance()+s.balance();  }
```

LOGICAL SERIAL ORDER:



Safe Futures

- Programmer annotates method calls
- Logical serial order enforced by the run-time
  - ★Futures and continuations encapsulated into optimistic transactions
  - ★Foundational mechanisms shared with transactional monitors
  - ★The notion of logical serial order stronger than serializability
- Consistency checks:
  - ★Data accesses hashed into read and write maps
  - ★Maps used by continuation to detect conflicts for accesses from its future
  - ★Validation at synchronization points (when a future is claimed)
- Log updates by maintaining versions:
  - ★Versions used by future to prevent seeing updates by its continuation
- Aborts:
  - ★Automatic roll-back when conflict detected

Dependency Violations

C_f

```
int i = o.bar;
o.foo = 0;
```

C_f	C_c
read(o)	
	write(o)
	read(o)
write(o)	

(a) Forward

C_c

```
o.bar = 0;
int j = o.foo;
```

C_f	C_c
	write(o)
read(o)	
write(o)	
	read(o)

(b) Backward

Forward dependency violations can be handled by tracking data dependencies.

Backward dependency violations can be handled by versioning updates. Future never sees a premature update by its continuation.

Ensuring Safety

Account c; Account s;

20 80

F1
F2
C

```
Future f1 = F[acc.transfer(10)];
Future f2 = F[acc.total()];
acc.transfer(10);
f1.get();
print(f2.get());
```

T_{F1} ⟶

T_{F2} ⟶

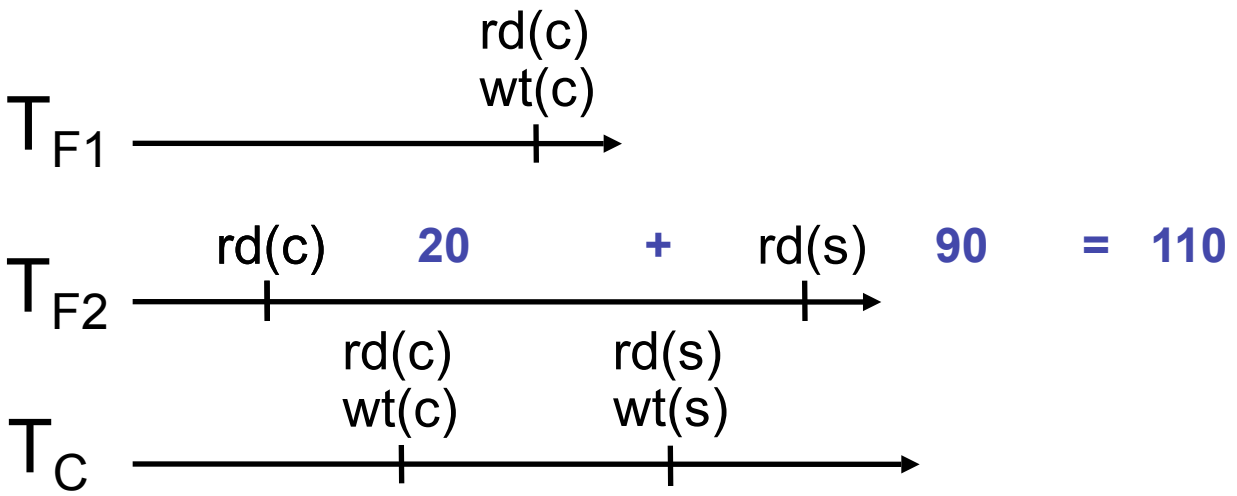
T_{C} ⟶

Ensuring Safety

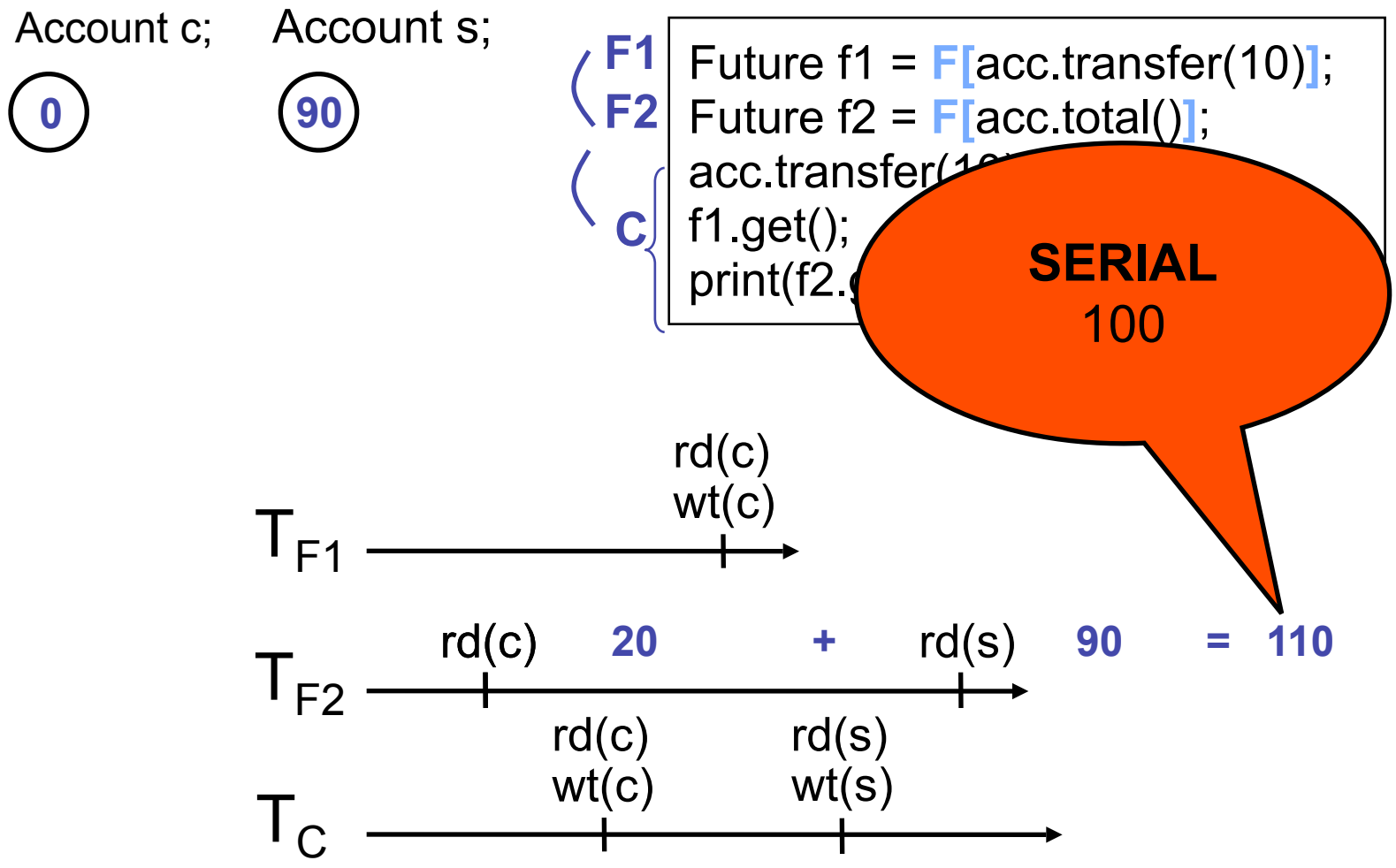
Account c; Account s;

0 90

F1 Future f1 = F[acc.transfer(10)];
F2 Future f2 = F[acc.total()];
C acc.transfer(10);
f1.get();
print(f2.get());



Ensuring Safety



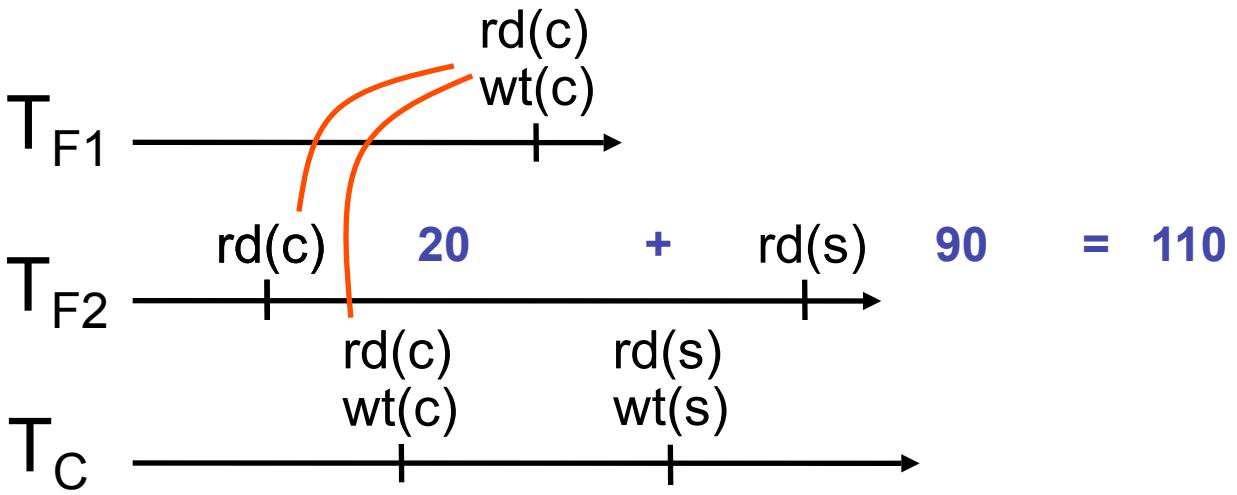
Ensuring Safety

Account c; Account s;

0 90

F1 F2 C

```
Future f1 = F[acc.transfer(10)];
Future f2 = F[acc.total()];
acc.transfer(10);
f1.get();
print(f2.get());
```



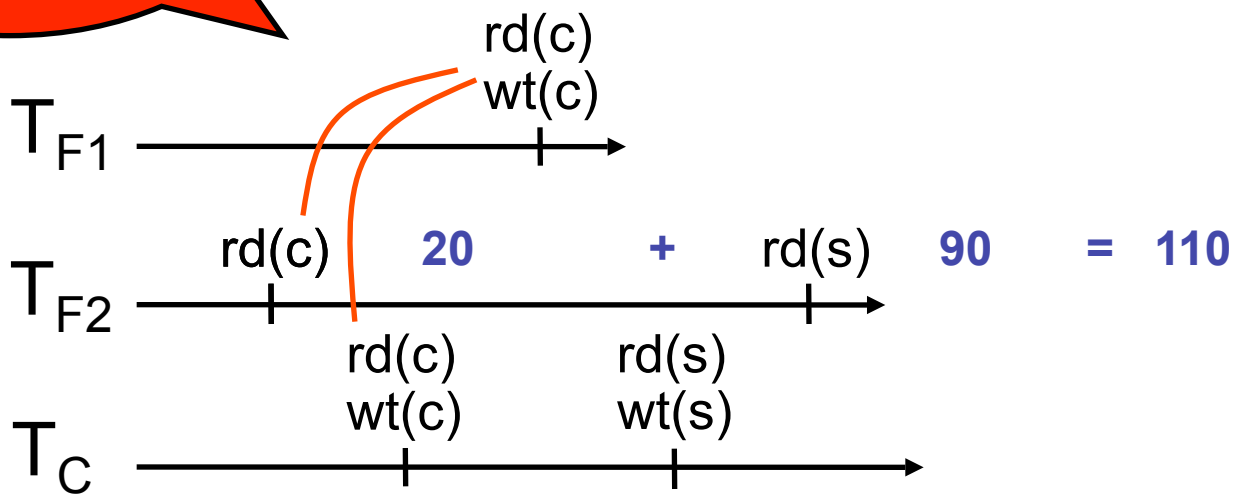
Ensuring Safety

Account c; Account s;

0 90

Forward Violations

F1 Future f1 = F[acc.transfer(10)];
F2 Future f2 = F[acc.total()];
C acc.transfer(10);
f1.get();
print(f2.get());



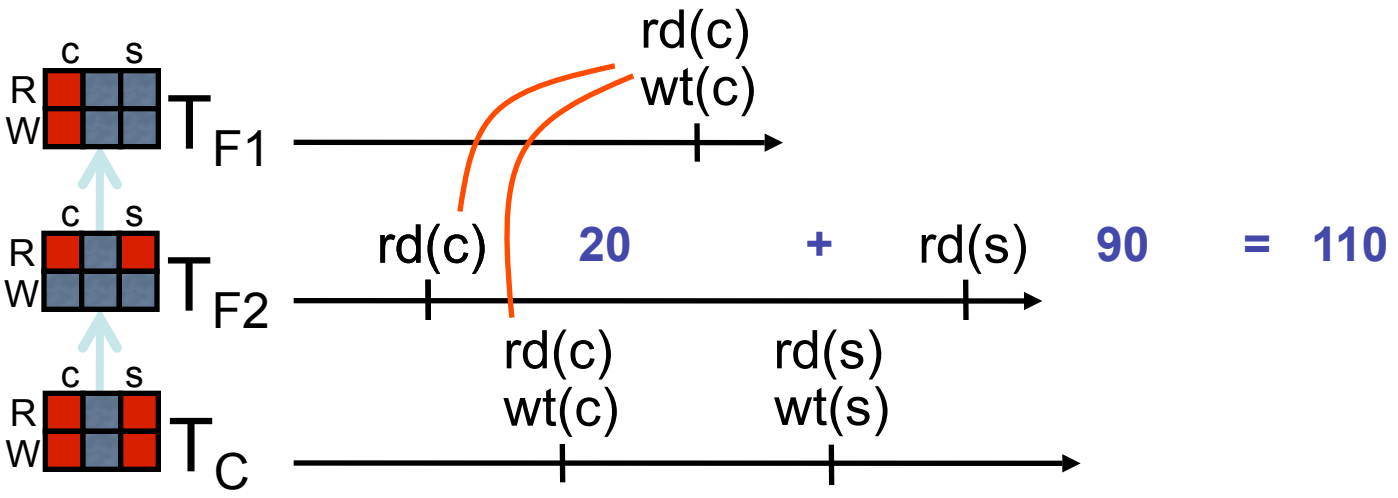
Ensuring Safety

Account c; Account s;

0 90

F1 F2 C

```
Future f1 = F[acc.transfer(10)];
Future f2 = F[acc.total()];
acc.transfer(10);
f1.get();
print(f2.get());
```

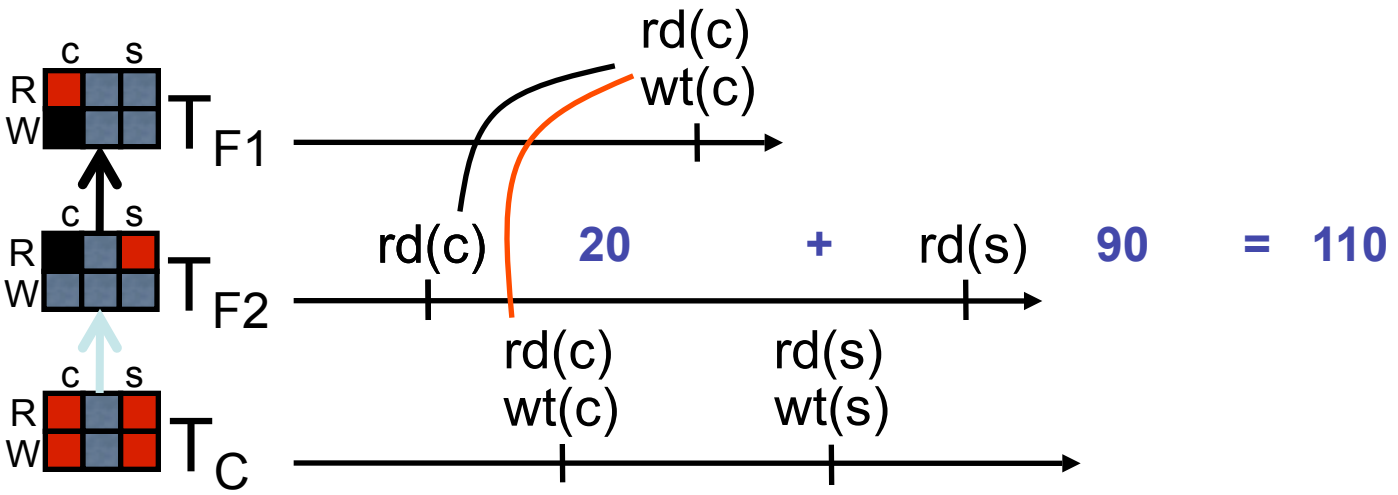


Ensuring Safety

Account c; Account s;

0 90

F1 Future f1 = F[acc.transfer(10)];
F2 Future f2 = F[acc.total()];
C acc.transfer(10);
f1.get();
print(f2.get());

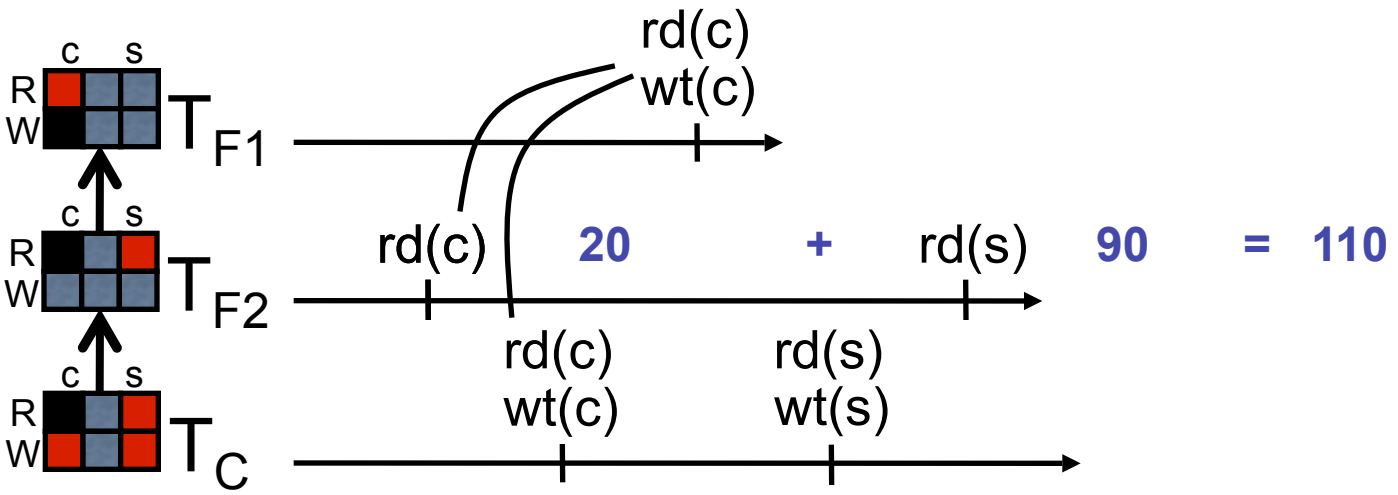


Ensuring Safety

Account c; Account s;

0 90

F1 Future f1 = F[acc.transfer(10)];
F2 Future f2 = F[acc.total()];
C acc.transfer(10);
f1.get();
print(f2.get());

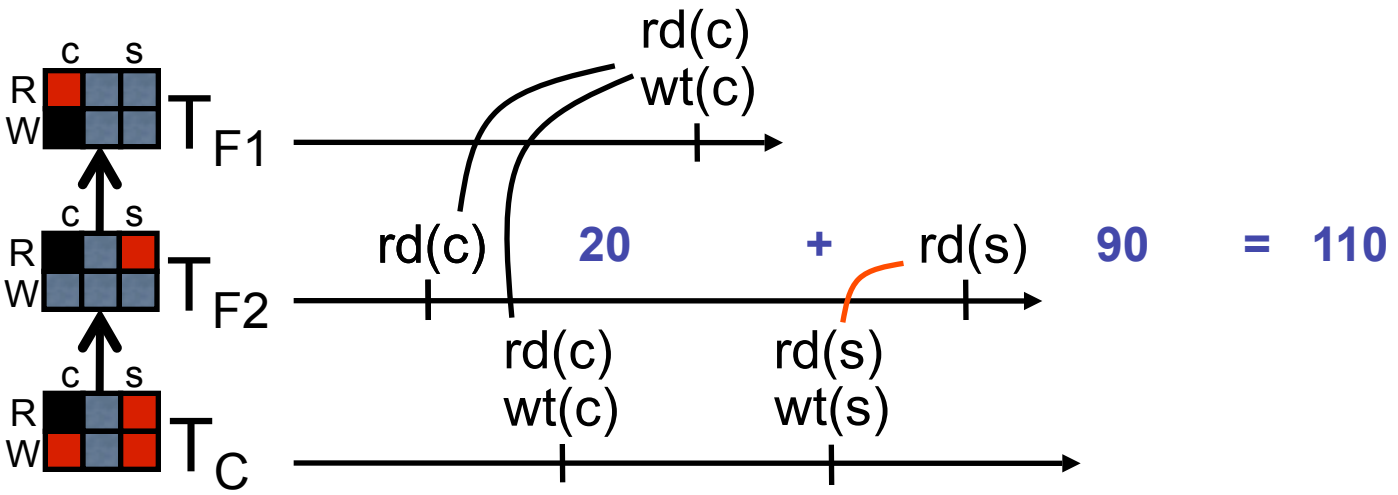


Ensuring Safety

Account c; Account s;

0 90

F1 Future f1 = F[acc.transfer(10)];
F2 Future f2 = F[acc.total()];
C acc.transfer(10);
f1.get();
print(f2.get());



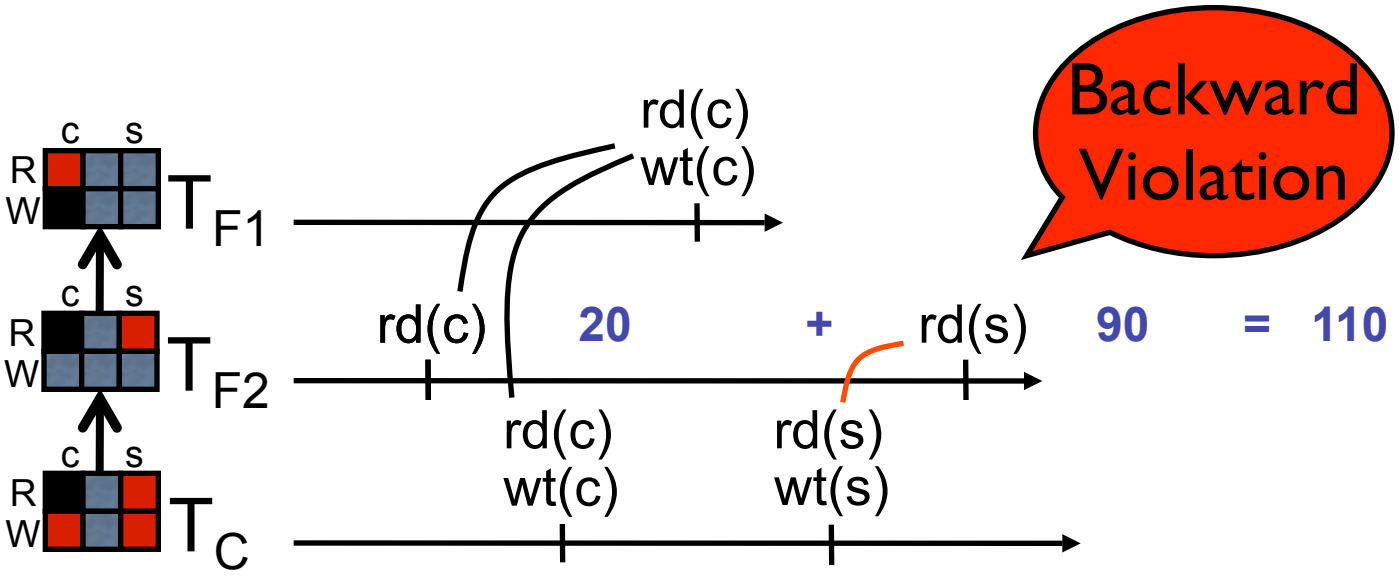
Ensuring Safety

Account c; Account s;

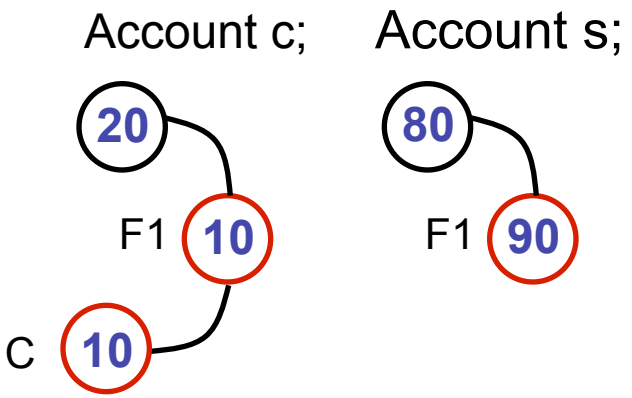
0 90

F1
F2
C

```
Future f1 = F[acc.transfer(10)];
Future f2 = F[acc.total()];
acc.transfer(10);
f1.get();
print(f2.get());
```

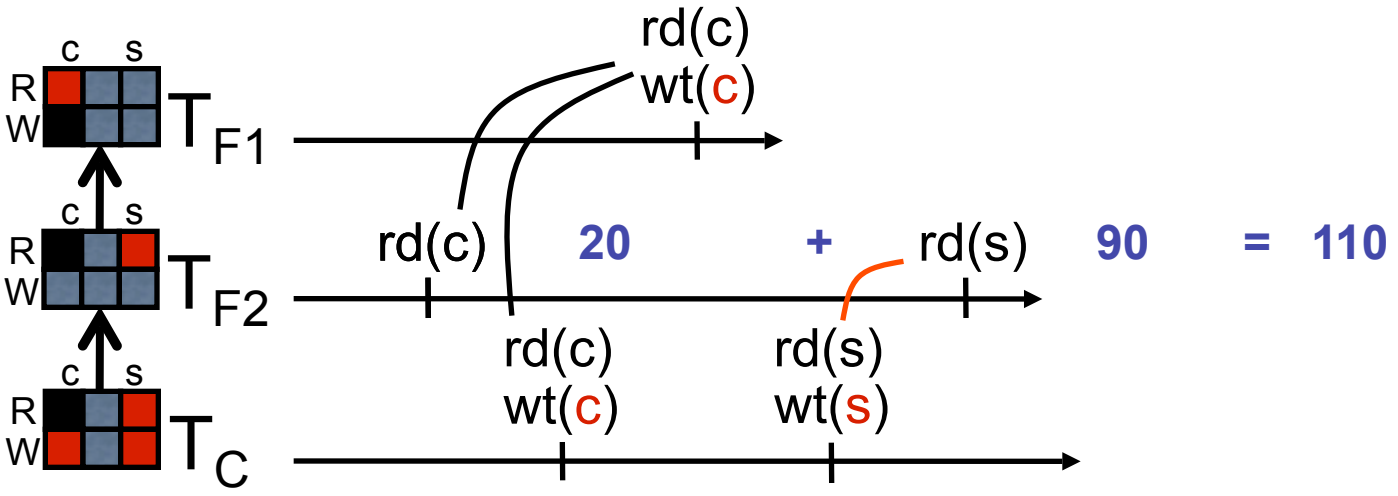


Ensuring Safety

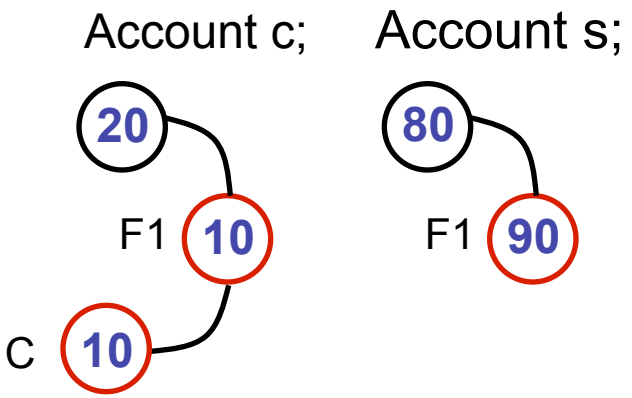


F1
F2
C

```
Future f1 = F[acc.transfer(10)];
Future f2 = F[acc.total()];
acc.transfer(10);
f1.get();
print(f2.get());
```

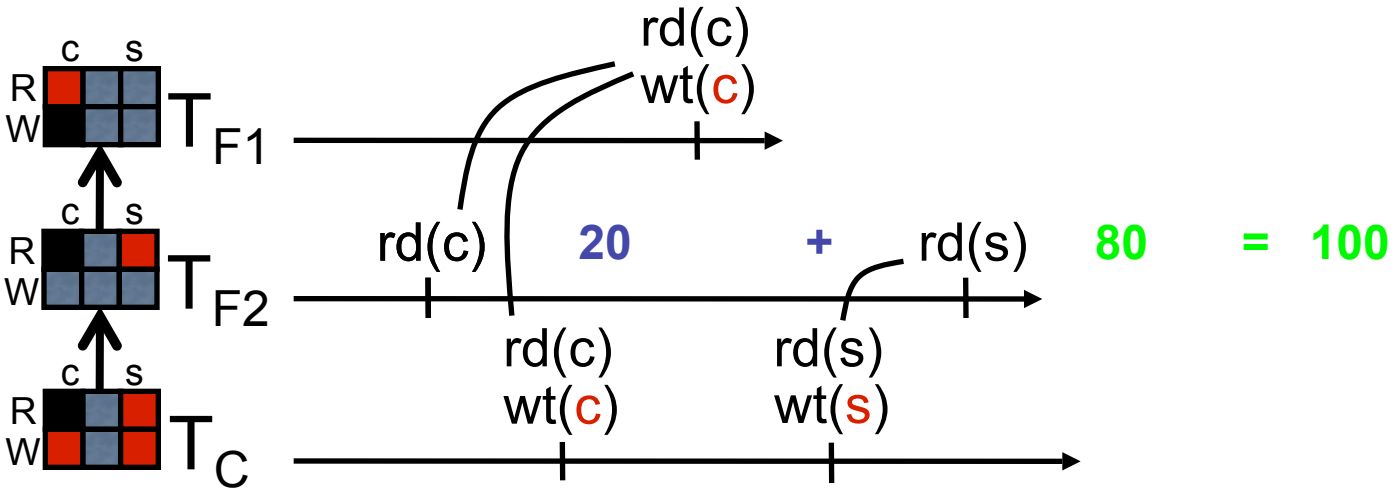


Ensuring Safety



F1
F2
C

```
Future f1 = F[acc.transfer(10)];
Future f2 = F[acc.total()];
acc.transfer(10);
f1.get();
print(f2.get());
```



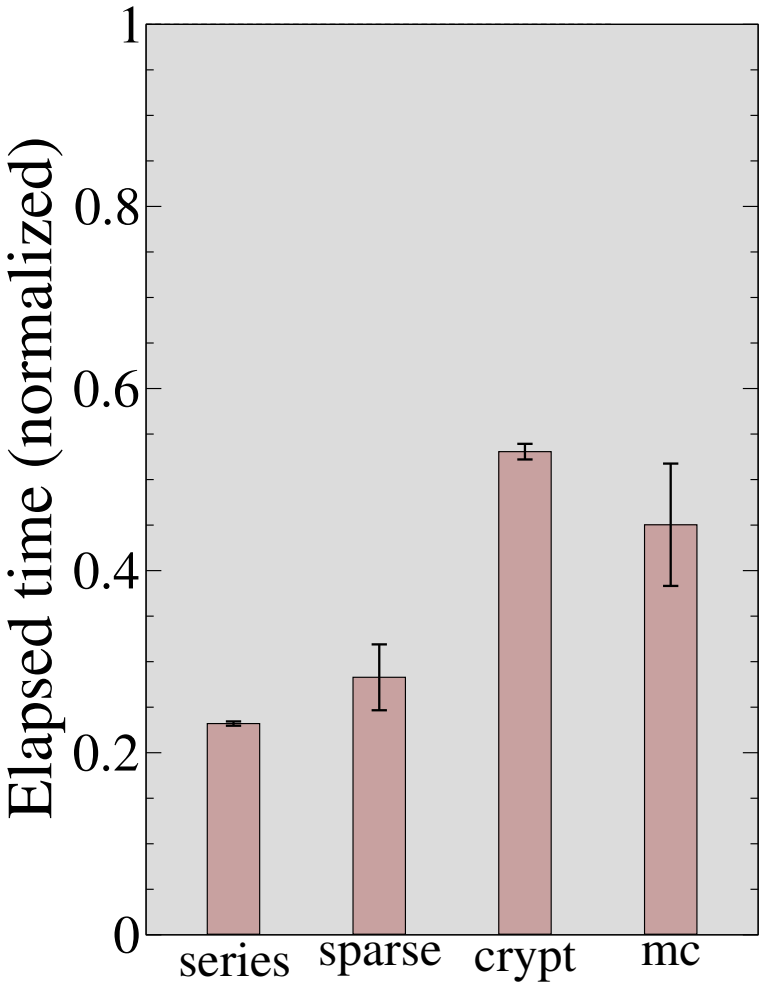
Our Prototype

- Based on IBM's Jikes RVM
- Compiler-injected read and write barriers to intercept shared data accesses
  - ★ Eager update of references on stack:
    - ✦ Version creation
    - ✦ Pre-specified synchronization points
- Bytecode rewriting plus run-time support for automatic roll-back
  - ★Modify runtime to roll-back without running user handlers
- Modification of object headers
  - ★Version access via forwarding pointers
- Experimental results
  - ★Roughly 50% efficiency for modest mutation rates (~ 30%)

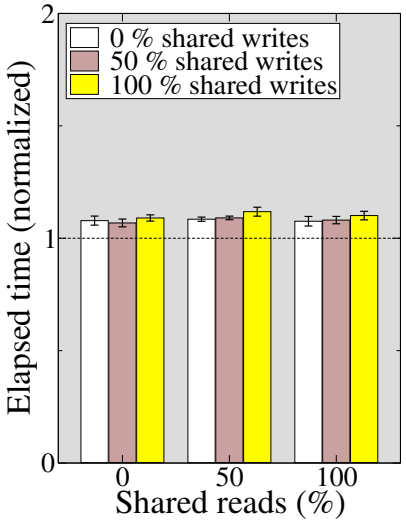
Evaluation

- Selected Java Grande benchmarks
- Modified Multi-User OO7 benchmark
  - ★Standard OO7 design database
    - ✦ Multi-level hierarchy of composite parts
    - ✦ Shared and private modules
  - ★Mixed-mode read/write traversals
- Configuration
  - ★700MHz Pentium 3 (used up to 4 CPUs)
  - ★Average of 5 "hot" runs

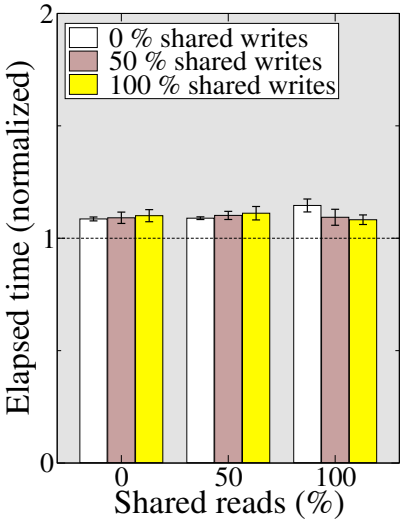
Experimental Results: 4 processor SMP



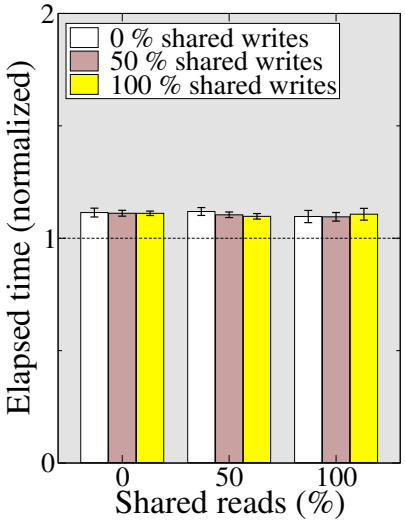
Evaluation



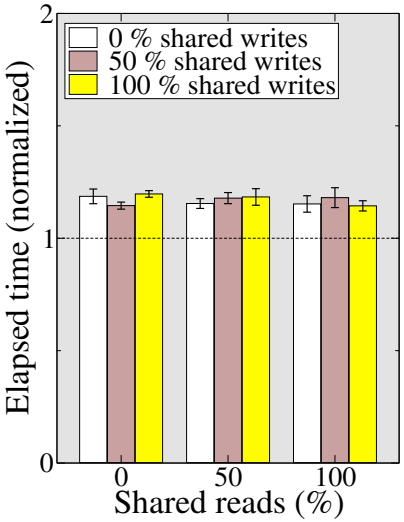
(a) 4% writes, 96% reads



(b) 8% writes, 92% reads



(c) 16% writes, 84% reads

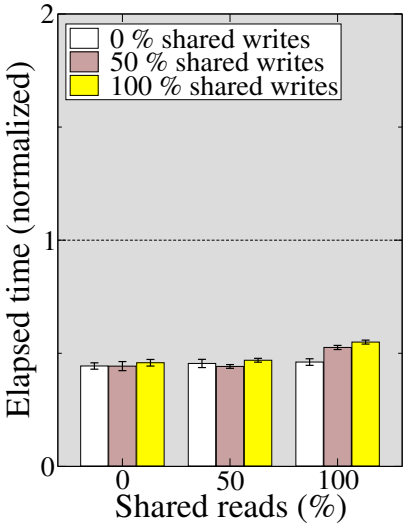


(d) 32% writes, 68% reads

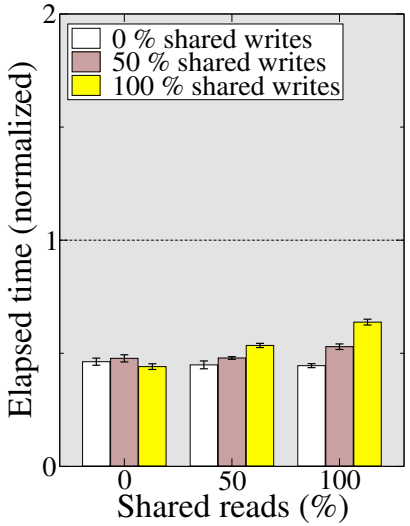
Only one future: measure base overheads.

Range from 8% (4% writes) to 15% (32% writes)

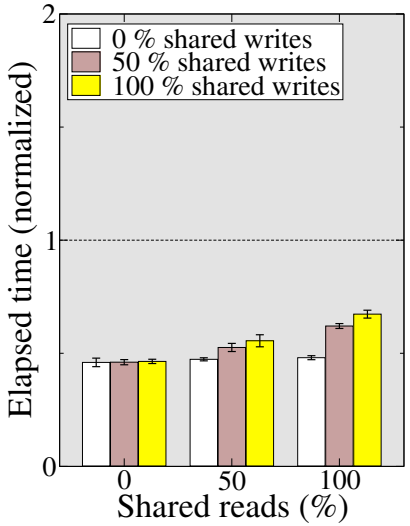
Evaluation



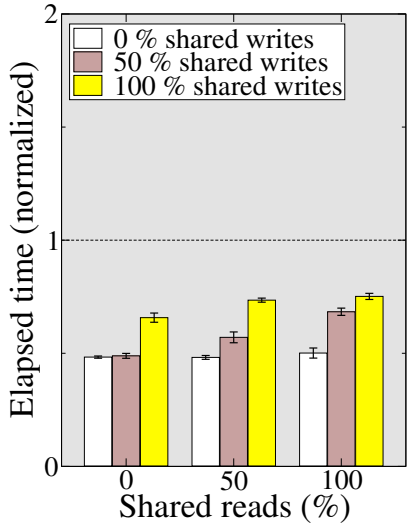
(a) 4% writes, 96% reads



(b) 8% writes, 92% reads



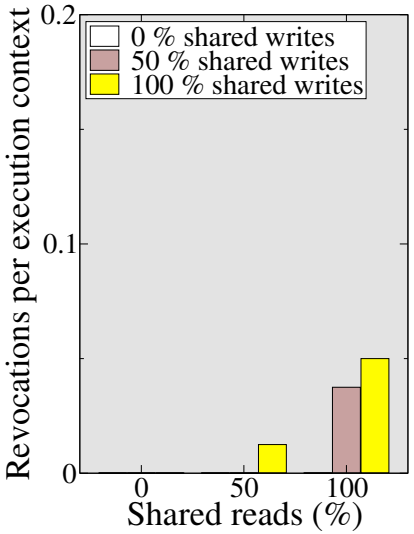
(c) 16% writes, 84% reads



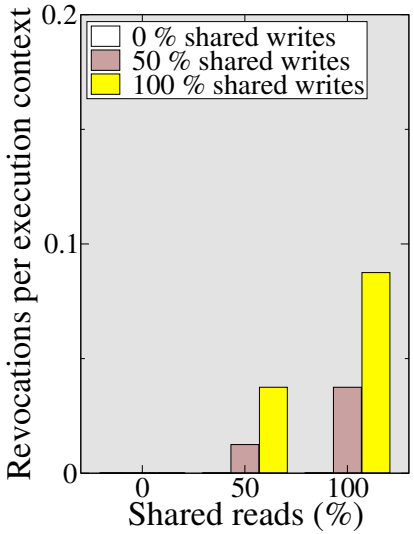
(d) 32% writes, 68% reads

With 4 futures, performance gains range from 55% to 25% over range of write ratios.

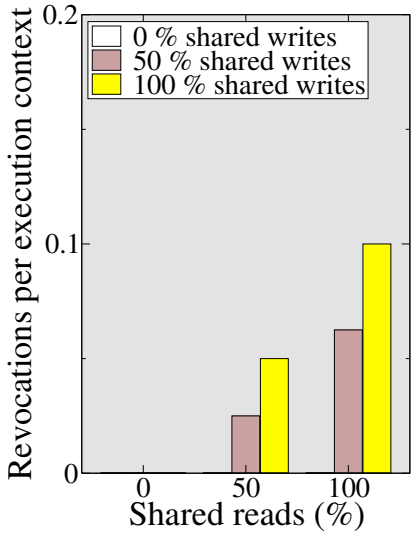
Evaluation



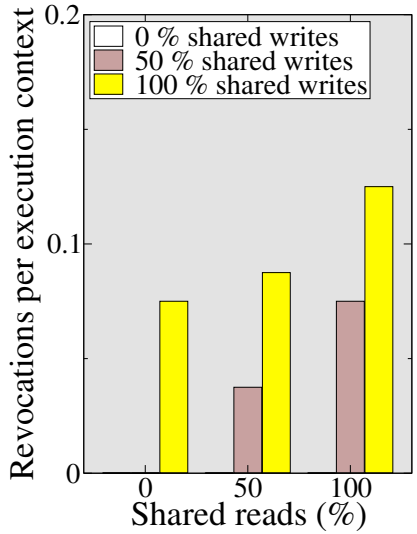
(a) 4% writes, 96% reads



(b) 8% writes, 92% reads



(c) 16% writes, 84% reads



(d) 32% writes, 68% reads

Revocations become more pronounced as shared write percentage increases

Similar structure for new versions created.

Case Study: Modular Checkpointing

- Many faults in long-lived software systems are *transient*:
  - ★Temporary unavailability of a resource:
    - ✦ network timeout
    - ✦ error states in a component repaired by reboot.
  - ★Unreliability of a resource:
    - ✦ packet loss
  - ★Semantic violations:
    - ✦ serializability violations in a transactional system.
- How can such faults be transparently repaired?
  - ★Concurrent threads of control.
  - ★Visible effects
    - ✦ Communication along channels
    - ✦ Shared memory

Robustness

- How can an exception handler ensure that global state is consistent after it executes?
  - ★ Consider thread communication within a handler scope
  - ★ How does a handler revert thread state to one which is consistent with views of other threads?
  - ★ Failure to ensure consistency can lead to deadlock, or erroneous results
- Difficult for applications to enforce consistency statically because of non-determinism and implicit, dynamically-defined thread dependencies
  - ★ If a thread broadcasts some data, how can an application efficiently determine the set of threads that are affected by this data?

Checkpoints

- Checkpoints provide a means to globally revert a computation to an earlier state.
- Transparent approaches: compiler or operating system
- Non-transparent: Library or application-directed
- Our idea:
  - ★ Applications define thread-local program points where checkpoint is feasible.
    - ✦ When a thread attempts to restore execution to a previous checkpoint, control reverts to one of these points for each thread.
    - ✦ The exact checkpoint chosen is calculated dynamically based on lightweight monitoring of thread communication events and effects.

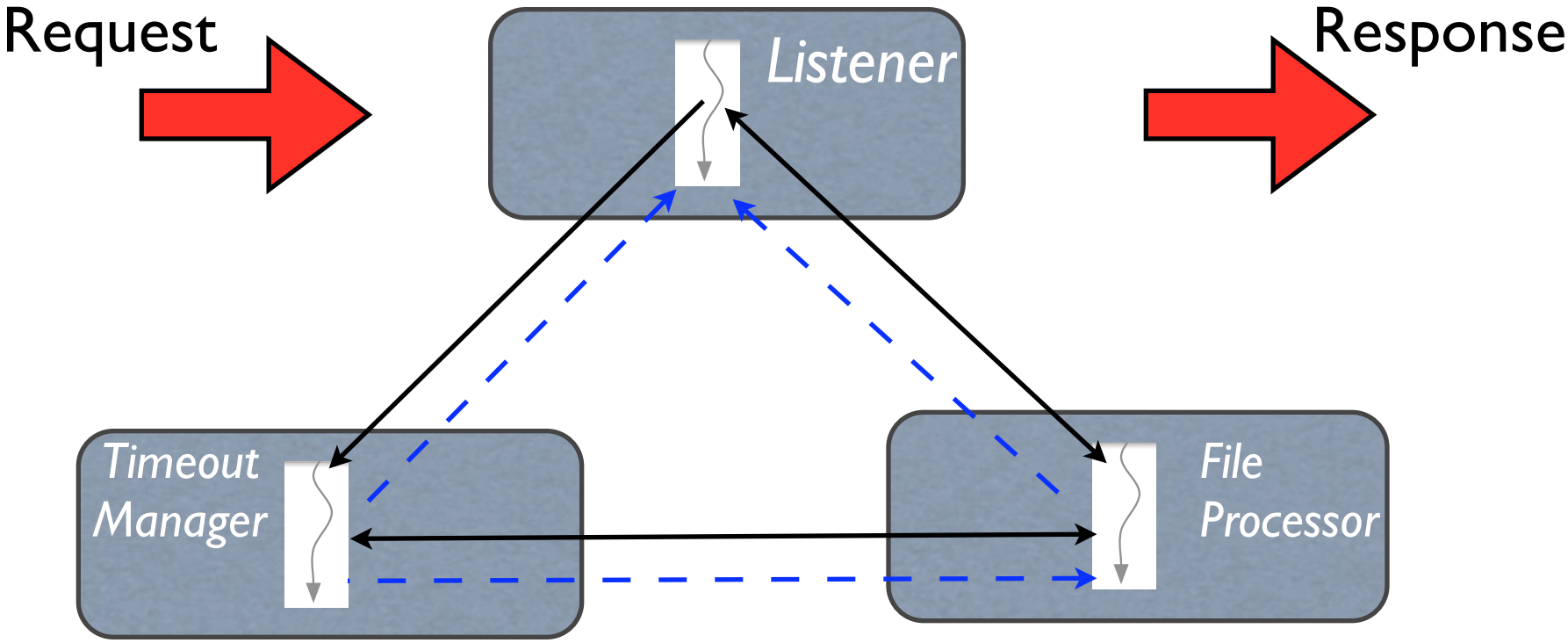
Stabilizers

- Signatures
  - ★ `stable: ('a -> 'b) -> ('a -> 'b)`
  - ★ `stabilize: unit -> 'a`
- Declare monitored section of code
  - ★ Track inter-thread actions including communication and shared memory access
  - ★ Defines a thread-local checkpoint
- Maintain a global dependency structure
  - ★ Construct a global checkpoint from a collection of thread-local ones based on (transitive) thread dependencies
- Serve as building blocks for
  - ★ modular transient fault recovery for Concurrent ML
  - ★ safe software-based speculation
  - ★ open-nested multi-threaded software transactions

Comparison with Transactions

	Transactions	*Stabilizers*
Logging	✓	✓
Atomicity and Isolation	On updates Transaction-specific logs	On stabilization Thread-local checkpoints
Aborts	Transaction-local Lexically-delimted Serializability violation	Global Dynamically computed User-define
Nesting	Idiosyncratic	Uniform
Concurrency control	✓	×
Speculative multithreading	×	✓

Motivation



Swerve is an open source highly concurrent web server written in Standard ML.

Application logic complicated by need to handle transient timeout faults.

Observations

- Non-modular design:
  - ★Recovery from timeout failures requires an explicit protocol distributed among three different modules.
- Alternative strategy:
  - ★Use stabilizers to abstract explicit notification process.
  - ★Have the Timeout manager call stabilize when a timeout occurs.
  - ★Wrap communication events in the modules within stable sections.
  - ★No need for explicit polling
- Implications:
  - ★Timeout recovery expressed without having to embed non-local timeout logic within all threads.
    - ✦ Timeout handling and recovery localized within the Timeout manager.

Example

```
let val c = channel()
    val c' = channel()
    fun g y = ... recv(c) ... recv(c')
              ...
              raise Timeout
              ...
              in handle Timeout => ...
    fun f x = let val _ = spawn(g(...))
                  val _ = send(c,x)
                  ...
              in if ...
                 then raise Timeout
                 else ...
              end handle Timeout => ...
in spawn(f(arg))
end
```

What happens if f raises a timeout exception?

Must re-execute it, erasing effects from the earlier evaluation

Determining the set of events that must be restored depends on dynamic scheduler events.

Example

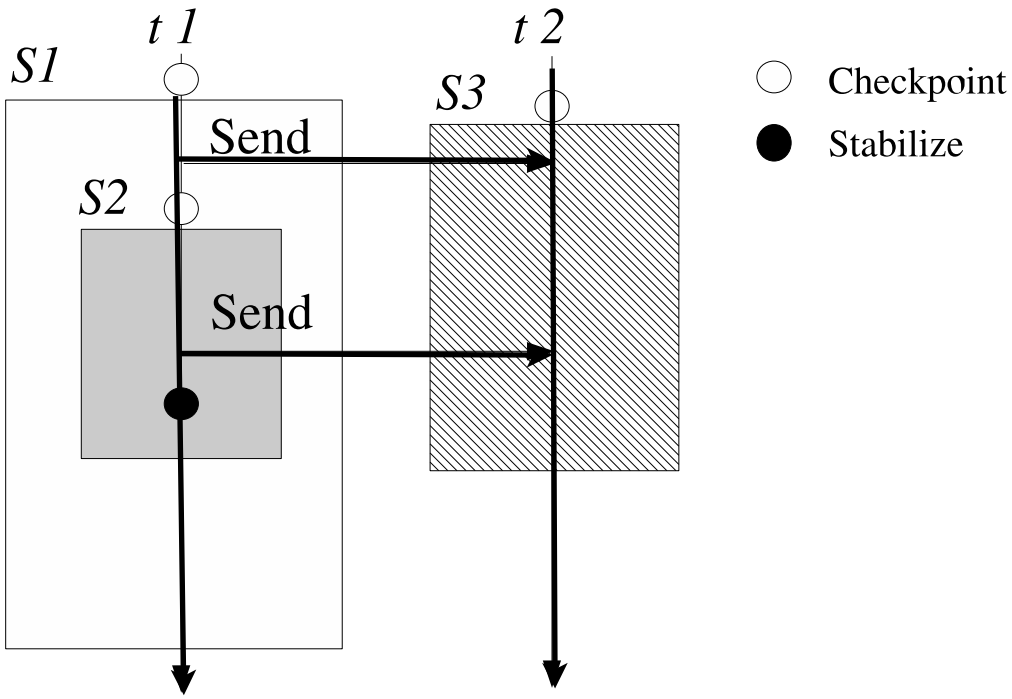
```
let val c = channel()
    val c' = channel()
    fun g y = ... recv(c) ... recv(c')
              ...
              raise Timeout
              ...
              in handle Timeout => ...
    fun f x = stable fn () =>
                  let val _ = spawn(g(...))
                      val _ = send(c,x)
                         ...
                  in if ...
                     then raise Timeout
                     else ...
                  end handle Timeout => stabilize()  ()
in spawn(f(arg))
end
```

A timeout exception reverts the computation to a state in which the spawn of g, and its receipt on channel c have been discarded.

Behavior

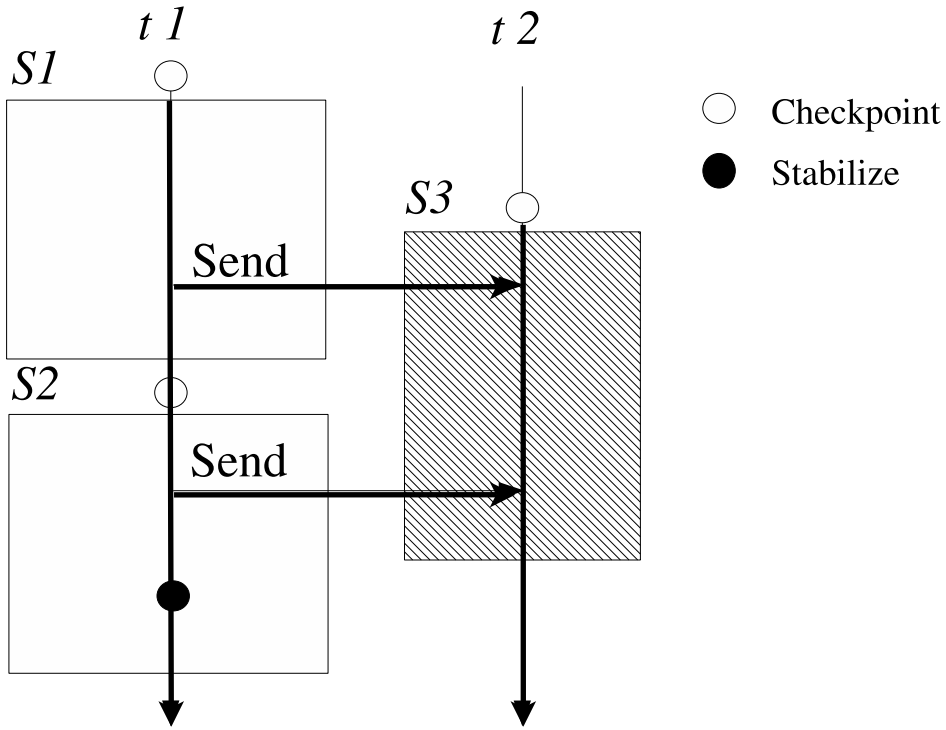
- Stable sections defined by programmer
- Safety violations explicit
  - ★ Not limited to serializability violations
- Save continuations for control
- Version updates
  - ★ Channel communication
  - ★ Shared variables
- Abort semantics
  - ★ Revert control to globally consistent state based on communication events observed within a stable section.
  - ★ Basis for dealing aborts in optimistic multi-threaded and open-nested (speculative) transactions.

Example



Sections chosen for rollback depends upon communication actions performed

Example



Sections chosen for rollback depends upon communication actions performed

Semantics

- Define a call-by-value functional core with threads and synchronous channel communication.
- First attempt:
  - ★ Grab entire checkpoint of program state.
    - ✦ Restore all threads to saved point.
- Core language:

$$
\begin{array}{rcl}
P & ::= & P \| P \mid \mathtt{t}[\mathtt{e}]_{\overline{\delta}} \\
\mathtt{e} & ::= & \mathtt{x} \mid \mathtt{l} \mid \lambda\, \mathtt{x}.\mathtt{e} \\
 & \mid & \mathtt{mkCh}() \mid \mathtt{send}(\mathtt{e},\mathtt{e}) \mid \mathtt{recv}(\mathtt{e}) \mid \mathtt{spawn}(\mathtt{e}) \\
 & \mid & \mathtt{stable}(\mathtt{e}) \mid \overline{\mathtt{stable}}(\mathtt{e}) \mid \mathtt{stabilize}
\end{array}
$$

$$
\begin{array}{rcll}
\delta & \in & \mathit{StableId} & \\
\mathtt{v} & \in & \mathit{Val} & = \quad \mathtt{unit} \mid \lambda\, x.e \mid \mathtt{l} \\
\alpha, \beta & \in & \mathit{Op} & = \quad \{\text{LR, SP,COMM,SS,ST,ES}\} \\
\Lambda & \in & \mathit{StableState} & = \quad \mathit{Process} \times \mathit{StableMap} \\
\Delta & \in & \mathit{StableMap} & = \quad \mathit{StableId} \xrightarrow{\text{fin}} \mathit{StableState}
\end{array}
$$

$$
E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{e}] \quad ::= \quad P \| \mathtt{t}[E[\mathtt{e}]]_{\overline{\delta}}
$$

Global Checkpoint

$$\frac{\begin{array}{c}\forall \delta \in Dom(\Delta), \quad \delta' \geq \delta \\ \Delta' = \Delta[\delta' \mapsto (E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{stable}(\lambda\,\mathtt{x.e})(\mathtt{v})], \Delta)] \\ \Lambda = \Delta'(\delta_{min}), \quad \delta_{min} \leq \delta \quad \forall \delta \in Dom(\Delta')\end{array}}{E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{stable}(\lambda\,\mathtt{x.e})(\mathtt{v})], \Delta \overset{\text{SS}}{\Longrightarrow} E^{\mathtt{t},P}_{\delta.\overline{\delta}}[\overline{\mathtt{stable}}(\mathtt{e}[\mathtt{v}/\mathtt{x}])], \Delta[\delta' \mapsto \Lambda]}$$

$$E^{\mathtt{t},P}_{\delta.\overline{\delta}}[\overline{\mathtt{stable}}(\mathtt{v})], \Delta \overset{\text{ES}}{\Longrightarrow} E^{\mathtt{t},P}_{\overline{\delta}}[\mathtt{v}], \Delta - \{\delta\}$$

$$\frac{\Delta(\delta) = (P', \Delta')}{E^{\mathtt{t},P}_{\delta.\overline{\delta}}[\mathtt{stabilize}], \Delta \overset{\text{ST}}{\Longrightarrow} P', \Delta'}$$

maintain ordering of stable sections

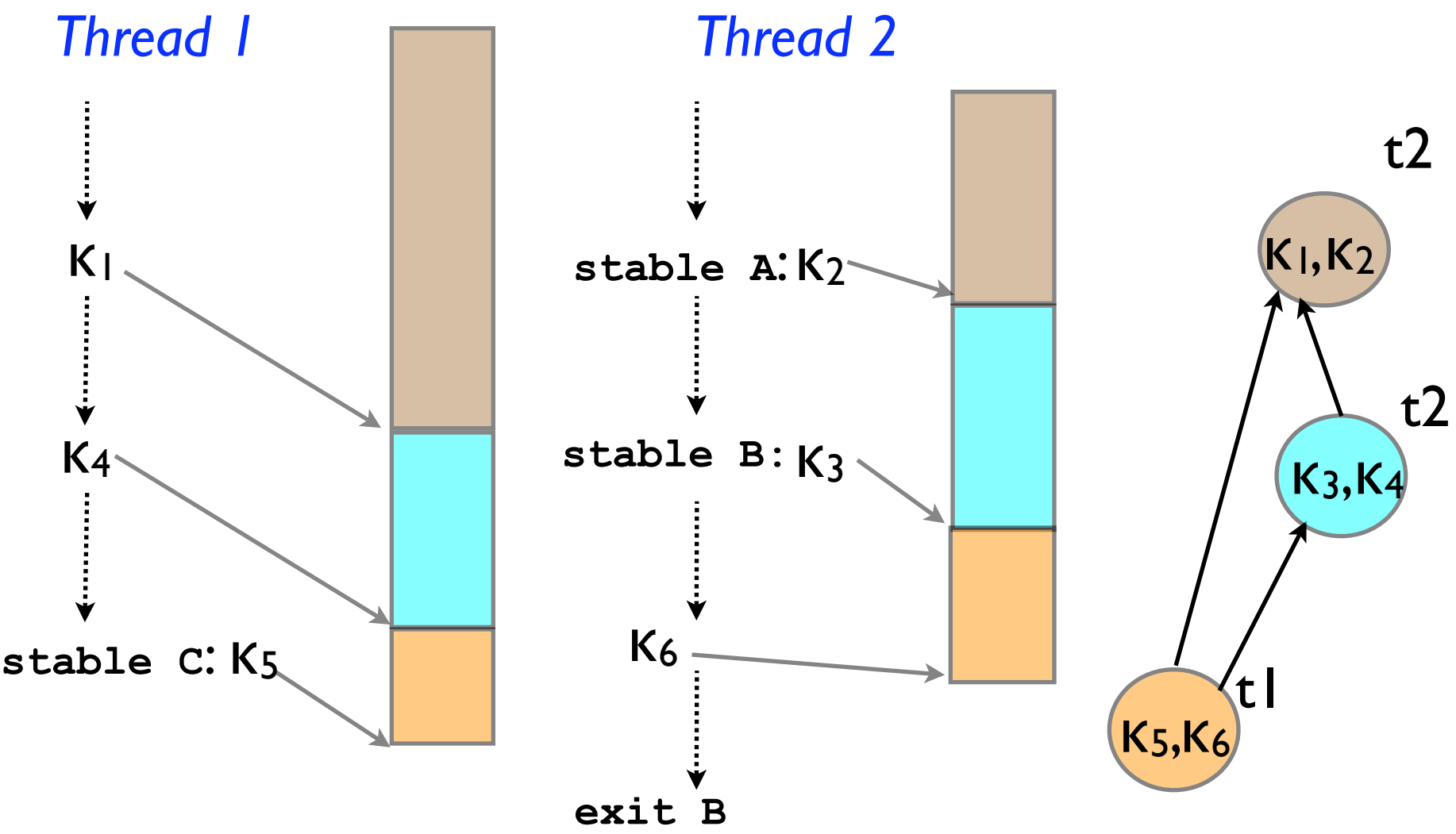
capture thread state

find least common ancestor

associate global checkpoint with stable section

restore to checkpoint saved for current stable section

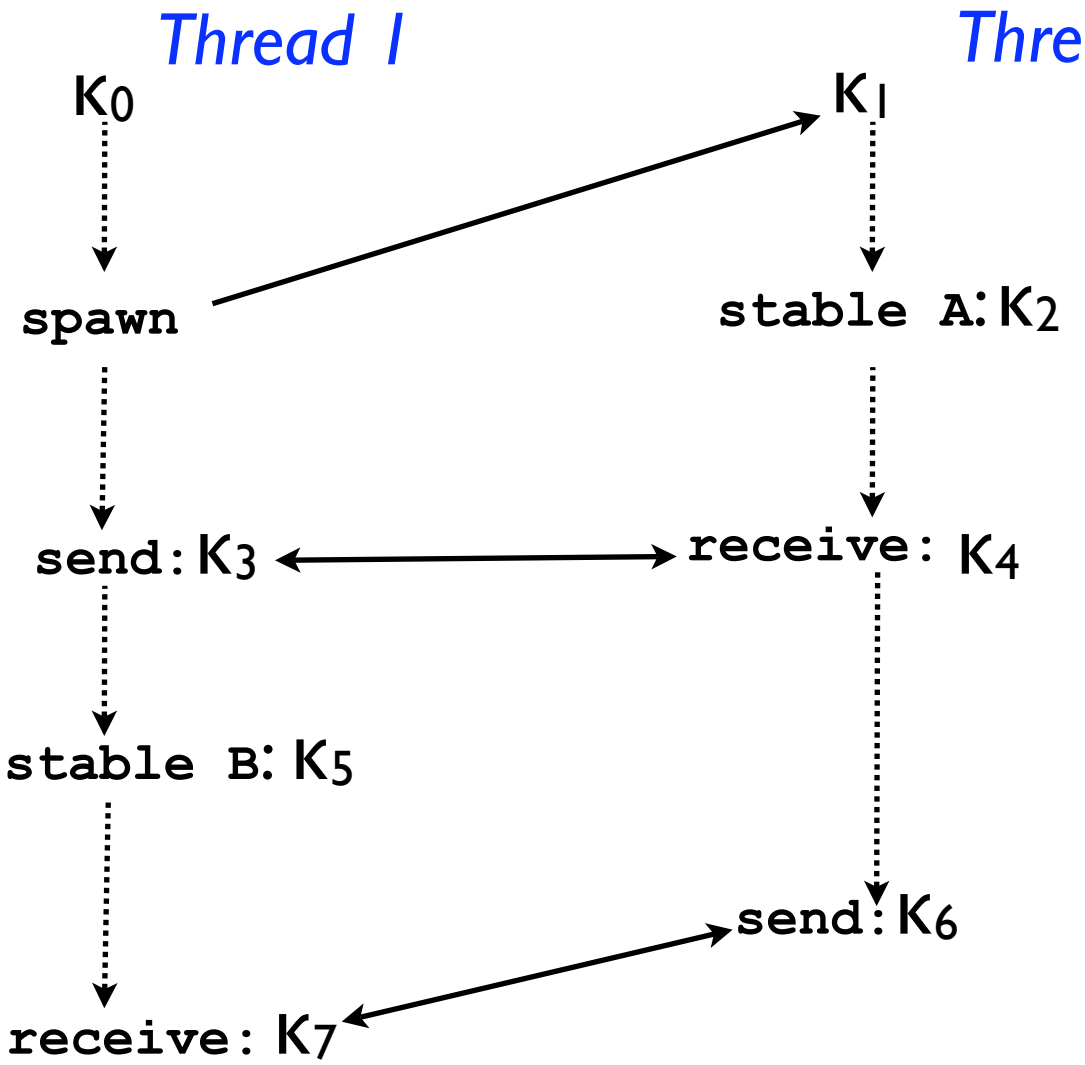
Global Checkpoint

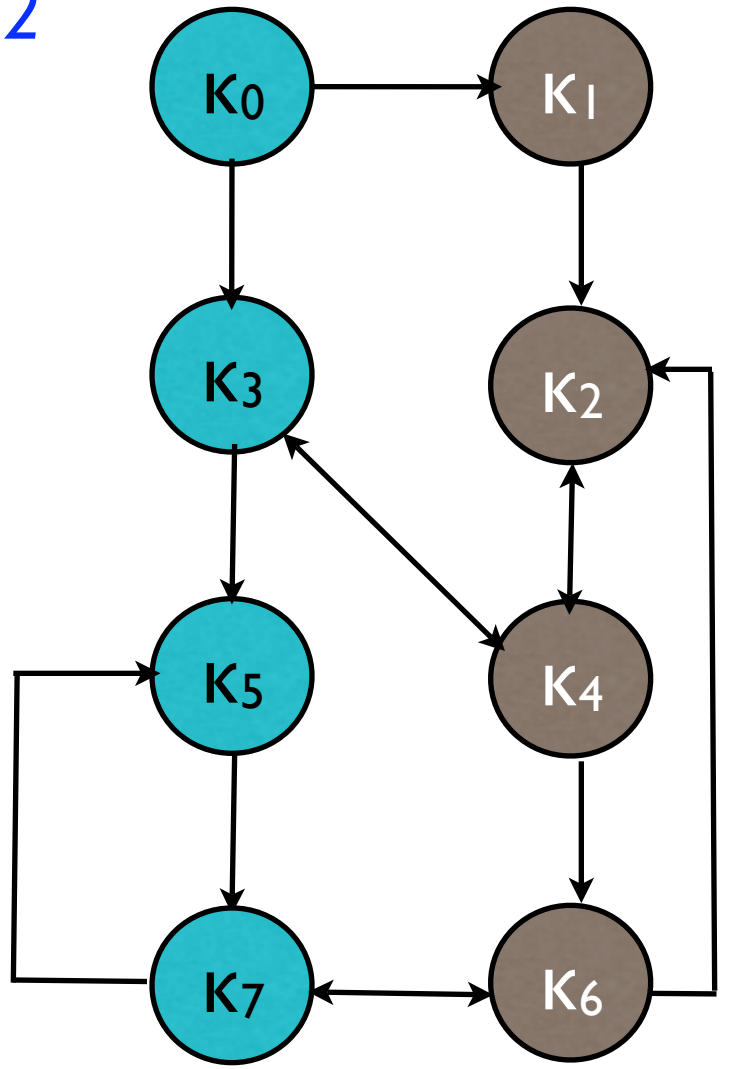


Can we do better?

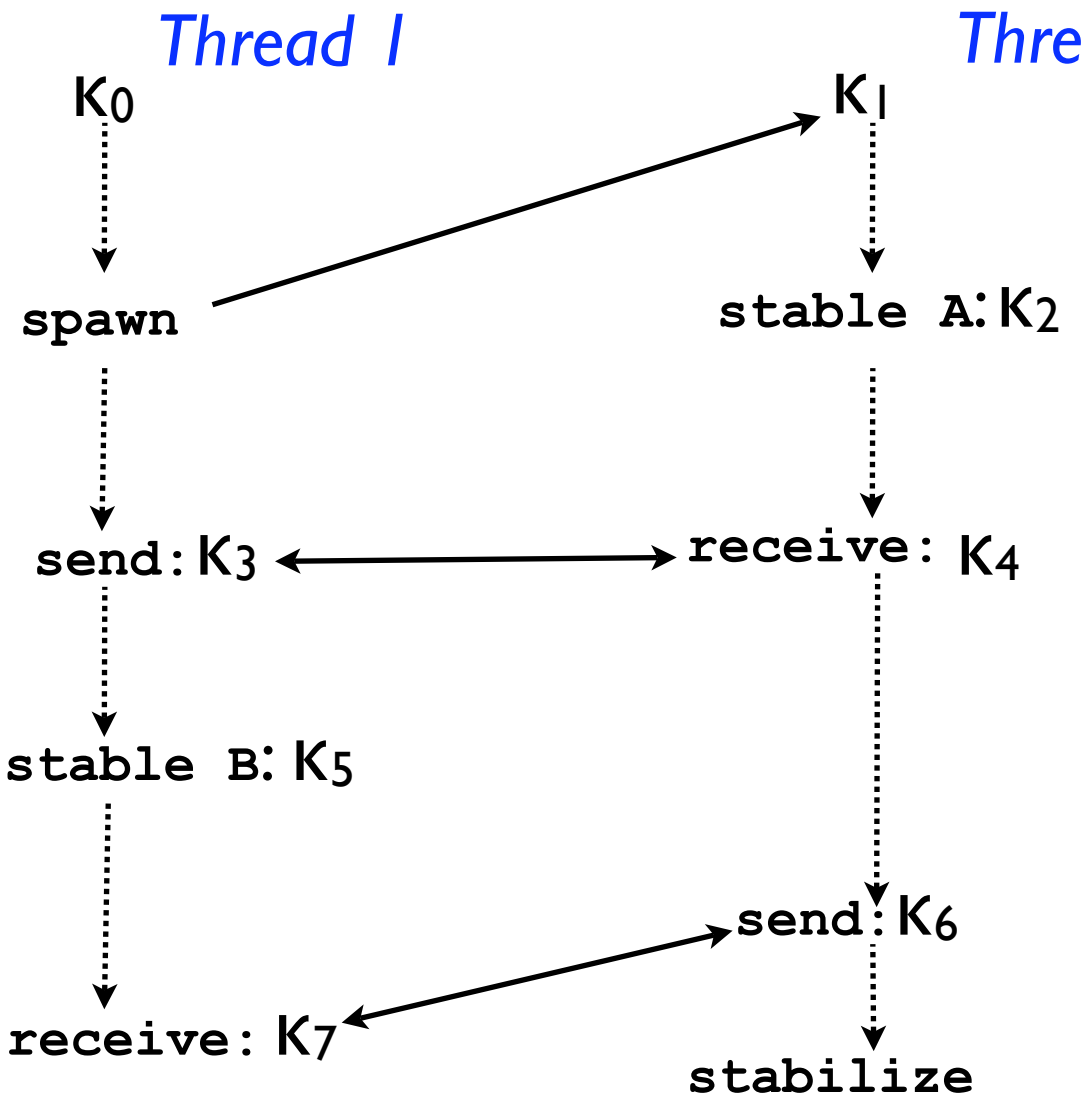
- Global checkpoints simple to describe, but ...
  - ★ hard to implement: requires global coordination to capture state
  - ★ overly conservative: restored checkpoint may revert computation unnecessarily
  - ★ does not take communication among threads into consideration
- Incremental construction:
  - ★ restore thread state based on the actions witnessed by threads
  - ★ build a dependency graph that tracks communication events and establishes a temporal ordering on thread-local actions
  - ★ use graph reachability on this graph to determine thread-local checkpoints.

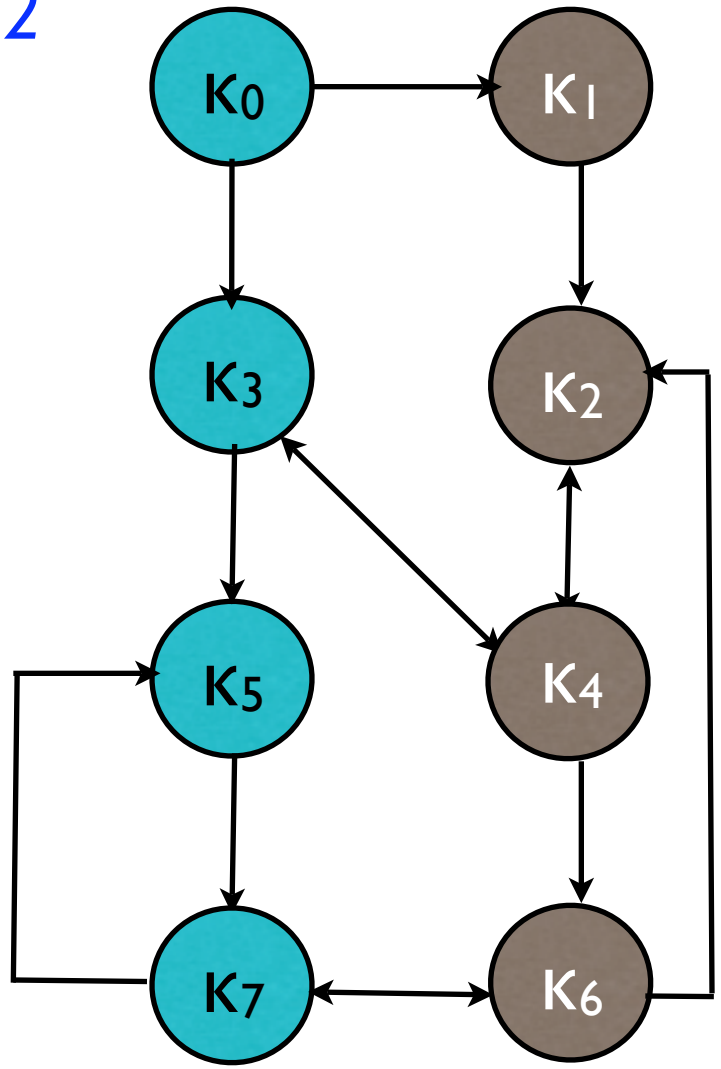
Incremental Checkpoint





Incremental Checkpoint

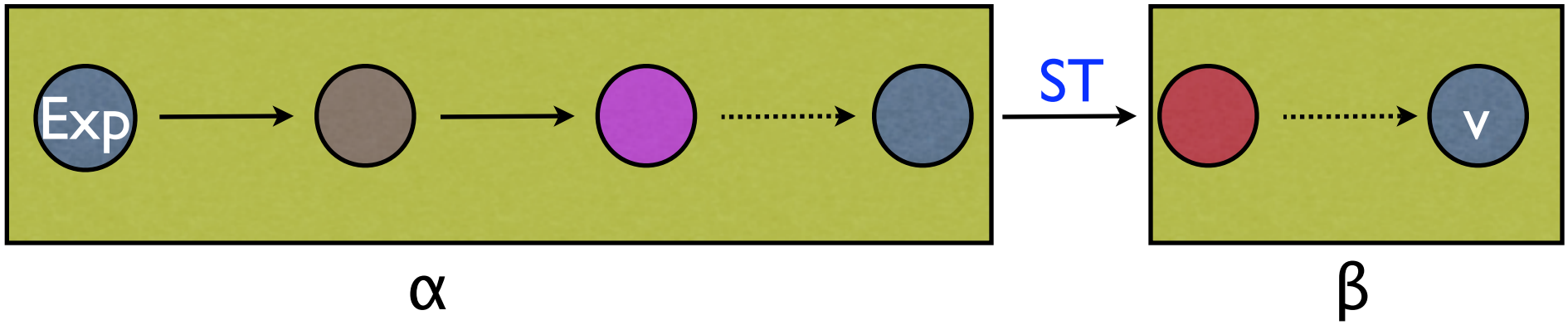




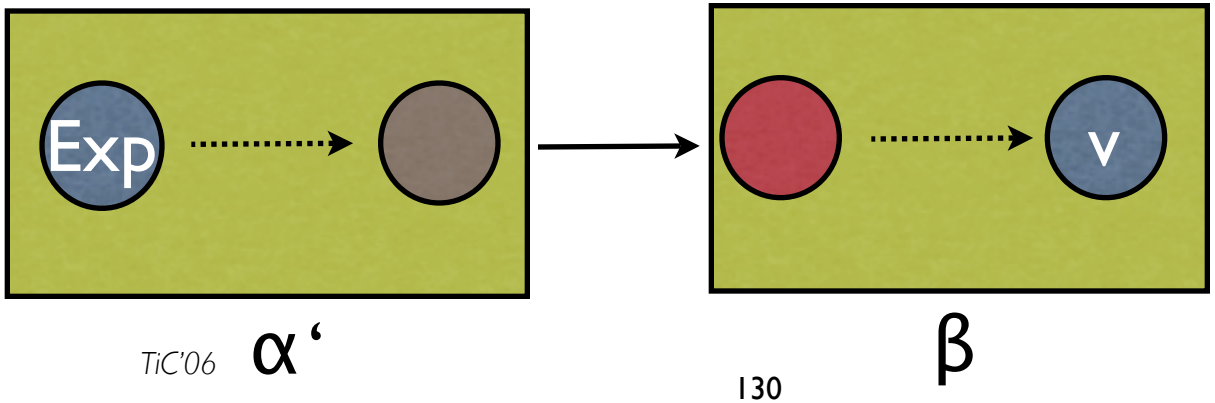
Garbage collection

Characteristics

- Properties:
  - ★Safety: A stabilize action never yields an infeasible state.



α β



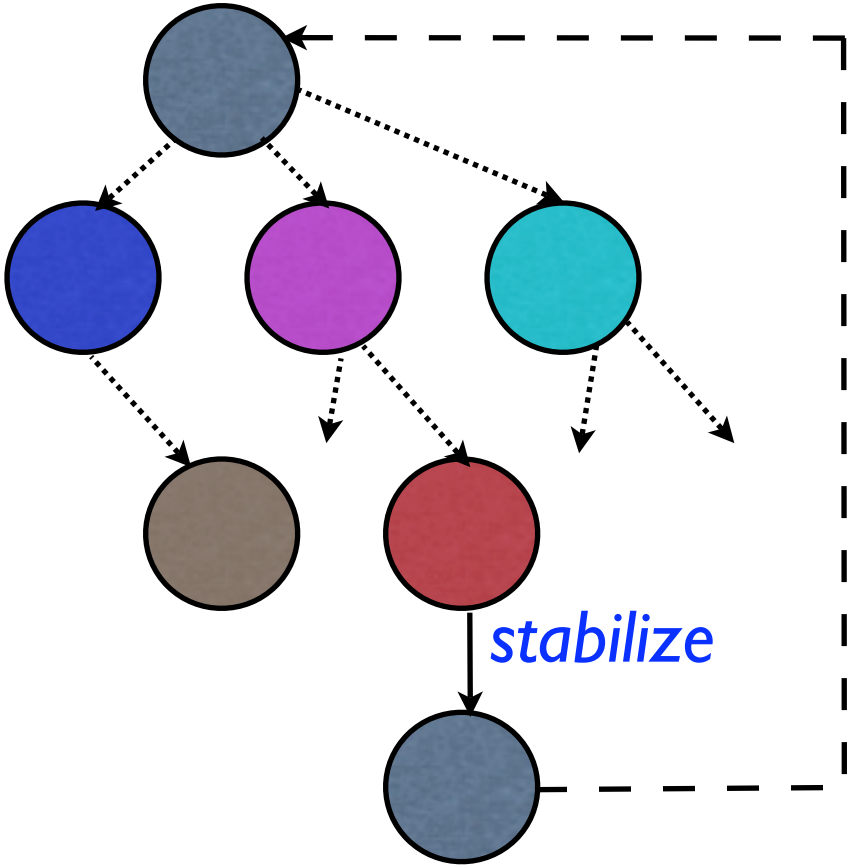
α' β

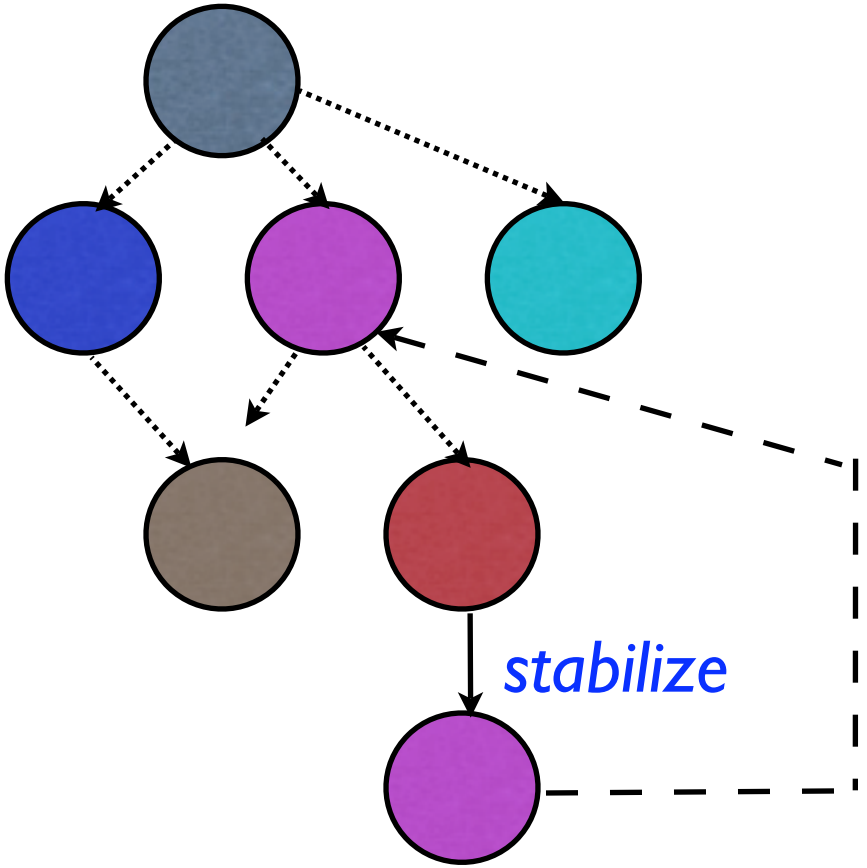
$\alpha' \leq \alpha$

stabilization can never manufacture new states (S³)

Characteristics

- Properties:
  - ★Correspondence: Incremental checkpointing is more efficient than global checkpointing.





Overheads

- Implemented in MLton
  - ★ Insertion of read and write barriers
  - ★ Compensations
  - ★ hooks in the CML library to update the dependency graph
- Overheads to maintain checkpoints small, roughly 6%
  - ★ eXene: a windowing toolkit
  - ★ Swerve: a web server

	Threads	Channels	Events	Shared Writes	Shared Reads	Graph Size (MB)	Runtime Overheads (%)
Triangle	205	79	187	88	88	.19	.59
N-Body	240	99	224	224	273	.29	.81
Pretty	801	340	950	602	840	.74	6.23
Swerve	10532	231	902	9339	80293	5.43	6.60

Restoration Costs

Requests	Graph	Channels		Threads	Runtime
	Size	Num	Cleared	Affected	(milli-seconds)
20	1130	85	42	470	5
40	2193	147	64	928	19
60	3231	207	84	1376	53
80	4251	256	93	1792	94
100	5027	296	95	2194	132

Swerve web server

Stabilization performed after a varying number of concurrent requests.

Instrumented Recovery

Benchmark	Channels		Threads		Runtime
	Num	Cleared	Total	Affected	milli-seconds
Swerve	38	4	896	8	3
eXene	158	27	1023	236	1.9

Swerve: induce a timeout every 10 requests.
eXene: induce packet loss every 10 packets.

Open Questions

- Long-lived and first-class transactions
  - ★ mixing implementation strategies safely and profitably
  - ★Consistency properties
- Open nesting
  - ★ Compensations
- Atomic data sets vs. atomic code regions
- STM for multicore:
  - ★making non-thread-safe code thread-safe
- Safe futures of arbitrary size and scope
  - ★Interaction with threads
- Stabilizers
  - ★ self-adjusting data structures (memoization)
  - ★ program slicing

Conclusions

- Software transactional implementations are necessarily complex.
  - ★ Address issues of versioning, rollback, and global consistency checks
  - ★ Efficient implementations possible, but non-trivial
- Can extract features of these implementations to address other interesting concurrency problems:
  - ★ safe speculative execution via futures
  - ★ safe checkpointing
- Much to be gained by exploring non-lock centric concurrency abstractions
- See http://www.cs.purdue.edu/s3
  - ★ Acknowledgments:
    - ✦ Adam Welc, Antony Hosking: *transactional monitors, safe futures*
    - ✦ Jan Vitek: *Transactional featherweight Java*
    - ✦ Lukas Ziarek, Philip Schatz: *stabilizers*