Transparent Support for Partial Rollback in Software Transactional Memories

SAPIENZA
UNIVERSITÀ DI ROMA

Alice Porfirio
Alessandro Pellegrini
Pierangelo Di Sanzo
Francesco Quaglia

High Performance and Dependable Computing Systems Group
Sapienza, University of Rome

Euro-Par 2013

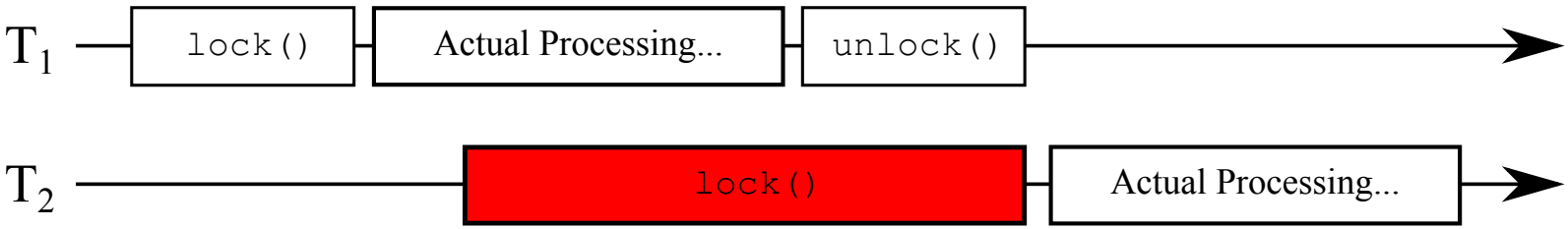
Research Context: Explicit Synchronization

- The most fundamental (and simple!) synchronization primitive is the lock

```
1 void transfer_money(int *bal1, int *bal2, int amount) {
2     lock(&global_lock);
3     if(*bal1 - amount > 0) {
4         *bal1 -= amount;
5         *bal2 += amount;
6     }
7     unlock(&global_lock);
8 }
```

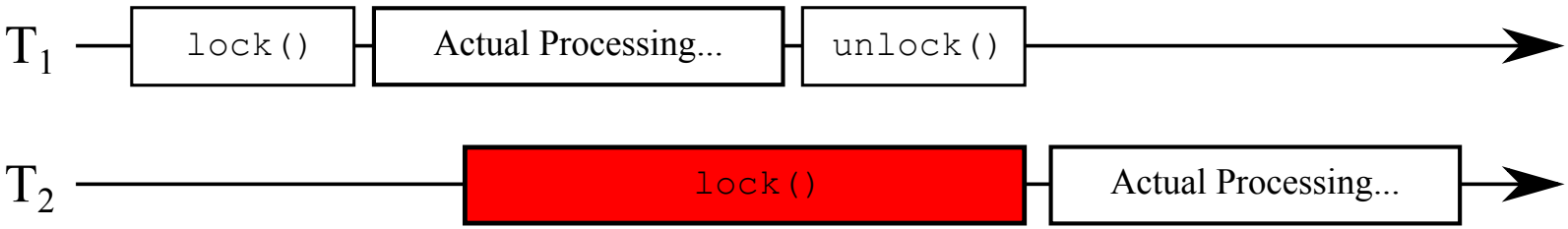
Research Context: Explicit Synchronization (2)

- Locks can be forced to block until released! ☹



Research Context: Explicit Synchronization (2)

- Locks can be forced to block until released! ☹



- It is very simple
- There is no *real* concurrency

Research Context: Fine Grain Locking

- Gives better perfomance
- Poor programmability and transparency
- Debugging is a nightmare

- Yet it still has problems
  - Deadlocks
  - Livelocks
  - Convoying
  - Priority Inversion
  - ...

```
1  void transfer_money(int *bal1,
       int *bal2, int amount) {
2
3      lock(&lock_bal1);
4      if(*bal1 - amount > 0) {
5          *bal1 -= amount;
6          unlock(&lock_bal1);
7
8          lock(&lock_bal2);
9          *bal2 += amount;
10         unlock(&lock_bal2);
11     } else {
12             unlock(&lock_bal1);
13     }
14 }
```

Research Context: Fine Grain Locking (2)

- Locks do not compose!

```
1  void deposit(int *bal, int amnt)
       {
2      lock(&lock_balance);
3      *bal += amnt;
4      unlock(&lock_balance);
5  }
6
7  void withdraw(int *bal, int amnt)
         {
8      lock(&lock_balance);
9      if(*bal - amnt > 0) {
10         *bal -= amnt;
11     }
12     unlock(&lock_balance);
13 }
```

```
1 void transfer(int *bal1, int *
      bal2, int amnt) {
2     withdraw(bal1, amnt);
3     deposit(bal2, amnt);
4 }
5
6 int sum(int *bal1, int *bal2)
      {
7     return *bal1 + *bal2;
8 }
```

Research Context: Software Transactional Memories

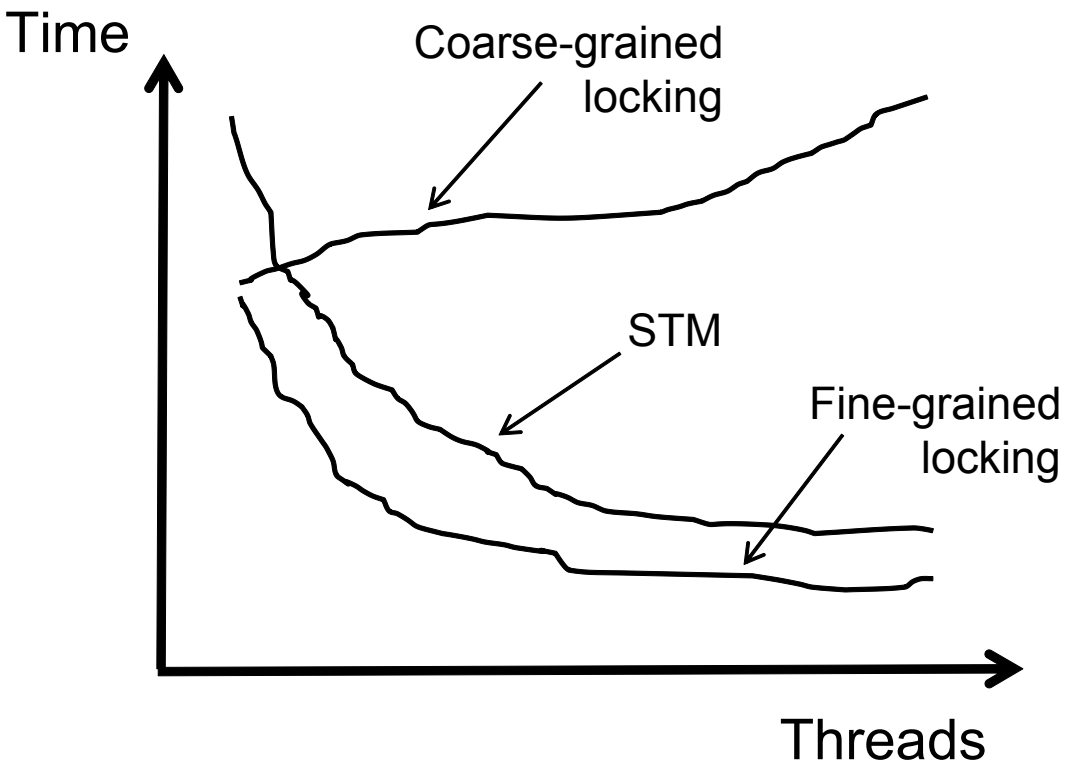
- Key idea:
  - Hide away synchronization issues from the programmer
  - Replace locks with atomic transactions

- Advantages:
  - Avoid deadlocks, priority inversions, convoying
  - Simpler to reason about, verify, compose
  - Provide synchronization transparency in concurrent applications
  - Synchronization code becomes less error-prone

```
1 void transfer(int *bal1, int *bal2, int amnt) {
2     atomic {
3         withdraw(bal1, amnt);
4         deposit(bal2, amnt);
5     }
6 }
```

Research Context: Software Transactional Memories

- Optimistic execution yields performance gains similar to fine grain locking at the simplicity of coarse grain locking



Partial Rollback: Motivations

- In Software Transactional Memories conflicts are handled via Conflict Detection and Management (CDMAN) algorithms
- Generally when a conflict arises some transaction is aborted
  - Thread's execution is rolled back
  - All the work carried out in the transaction is squashed

Partial Rollback: Motivations

- In Software Transactional Memories conflicts are handled via Conflict Detection and Management (CDMAN) algorithms
- Generally when a conflict arises some transaction is aborted
  - Thread's execution is rolled back
  - All the work carried out in the transaction is squashed

- Partial rollback could save a portion of this work
  - It can reduce the overall amount of work to be performed
  - Must be supported with minimal housekeeping overhead

Target CDMAN Algorithm: Commit-Time Locking

- Used in implementations such as TL2 or TinySTM
- Relies on a Global Version Clock (GVC)
  - A shared global counter
  - Atomically incremented by means of atomic CAS
  - Stored as Trastaction-Start Timestamp (TST) when a Tx begins
- Shared data are associated with
  - A lock bit
  - A timestamp

Target CDMAN Algorithm: Commit-Time Locking (2)

- On R/W operations the data's timestamp is compared to TST
- If it is greater, a conflict arises and the Tx is aborted

- At commit time all write locations' lock bits are locked and their timestamps are rechecked
- If no conflicts arise
  - New values are written back into shared objects
  - Their timestamps are updated
  - The GVC is incremented

Partial Rollback: Algorithm Sketch

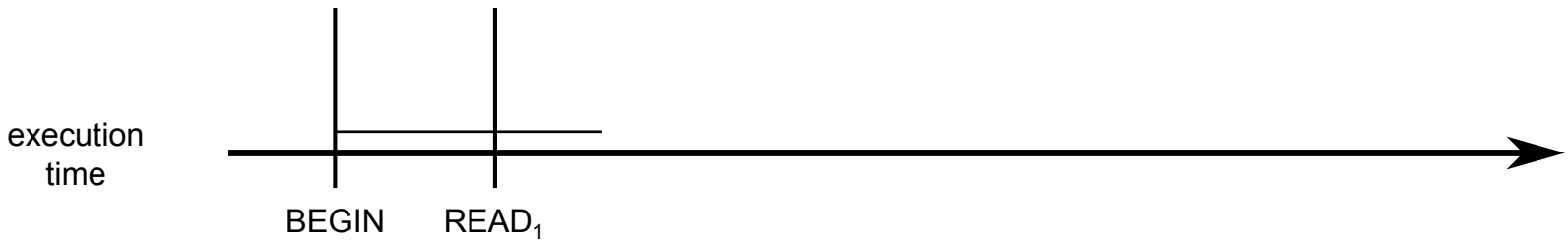
1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point

Partial Rollback: Algorithm Sketch

1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point

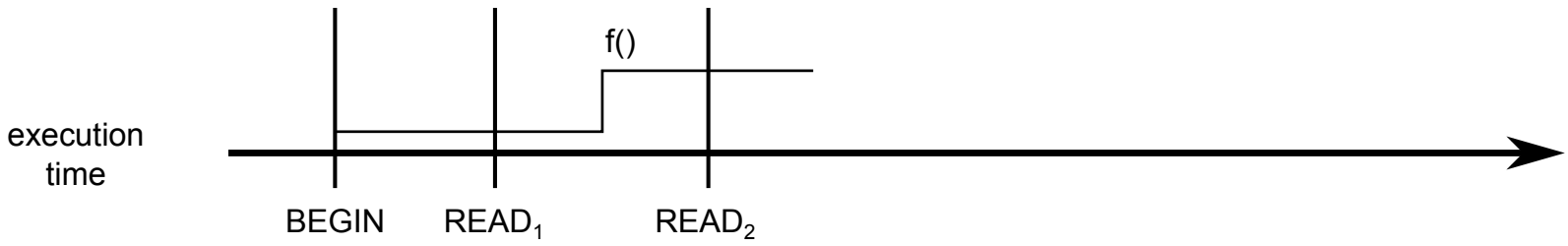
Partial Rollback: Algorithm Sketch

1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point



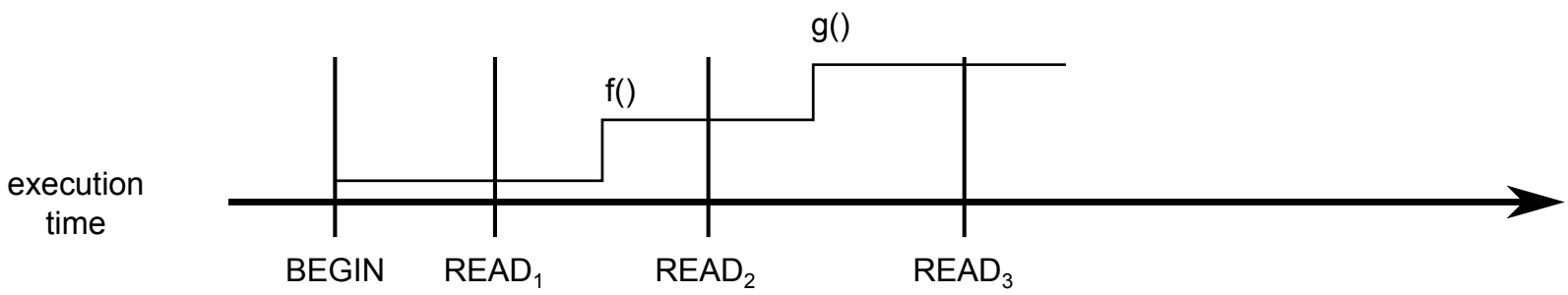
Partial Rollback: Algorithm Sketch

1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point



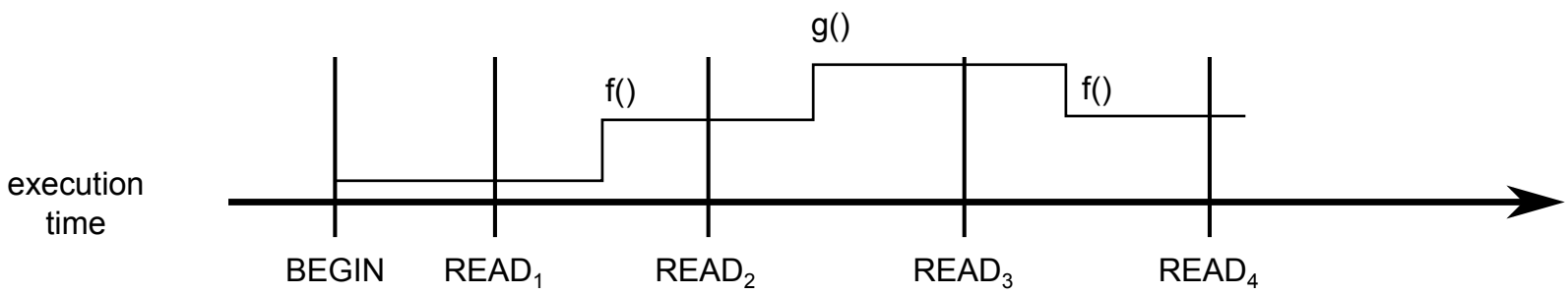
Partial Rollback: Algorithm Sketch

1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point



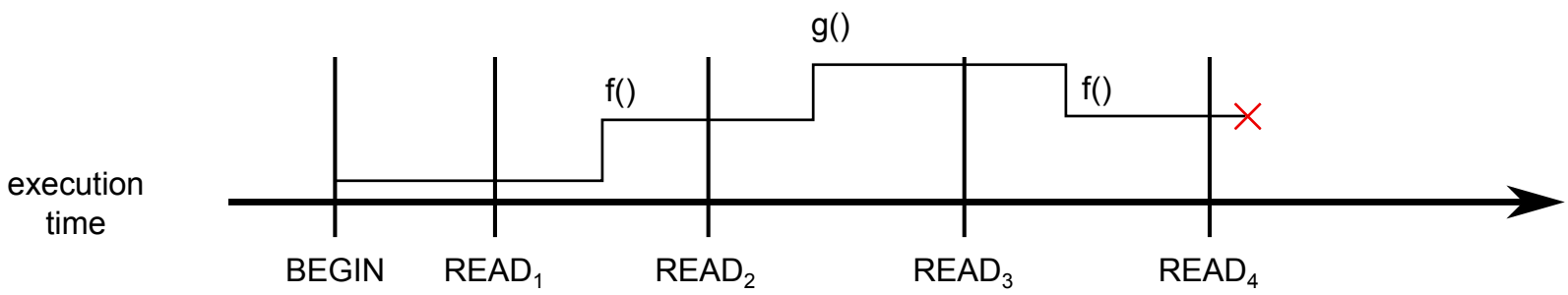
Partial Rollback: Algorithm Sketch

1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point



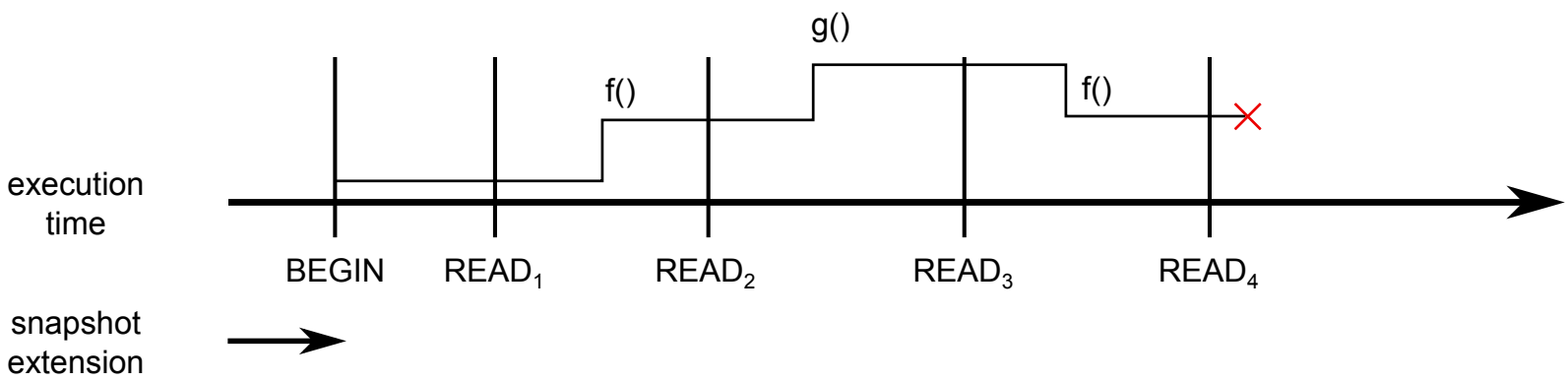
Partial Rollback: Algorithm Sketch

1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point



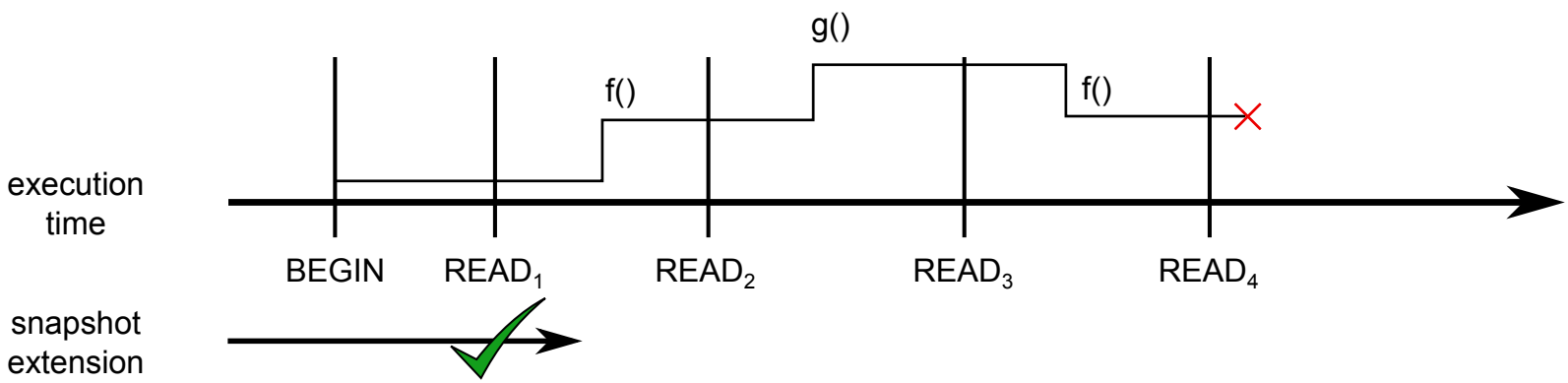
Partial Rollback: Algorithm Sketch

1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point



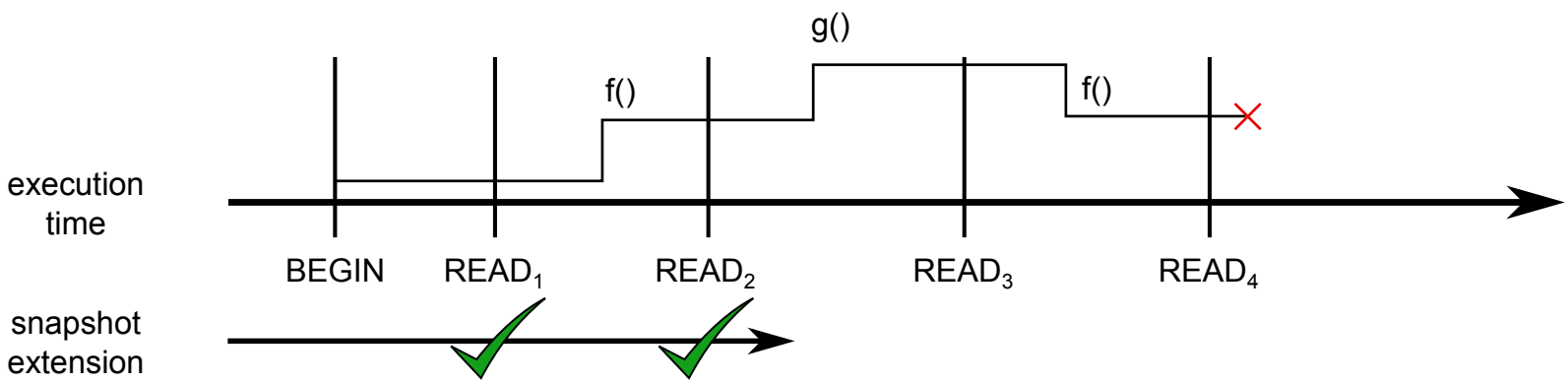
Partial Rollback: Algorithm Sketch

1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point



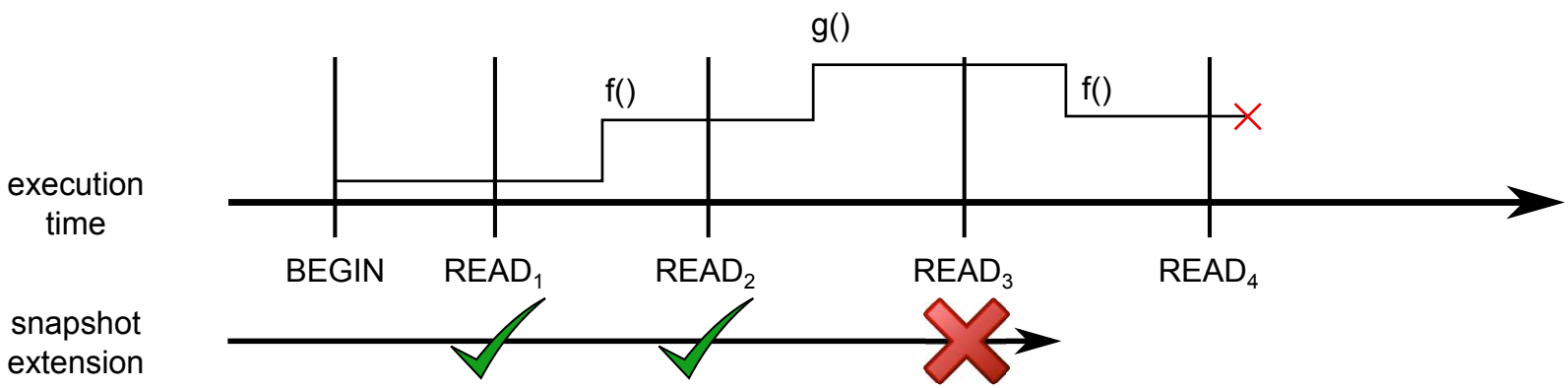
Partial Rollback: Algorithm Sketch

1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point



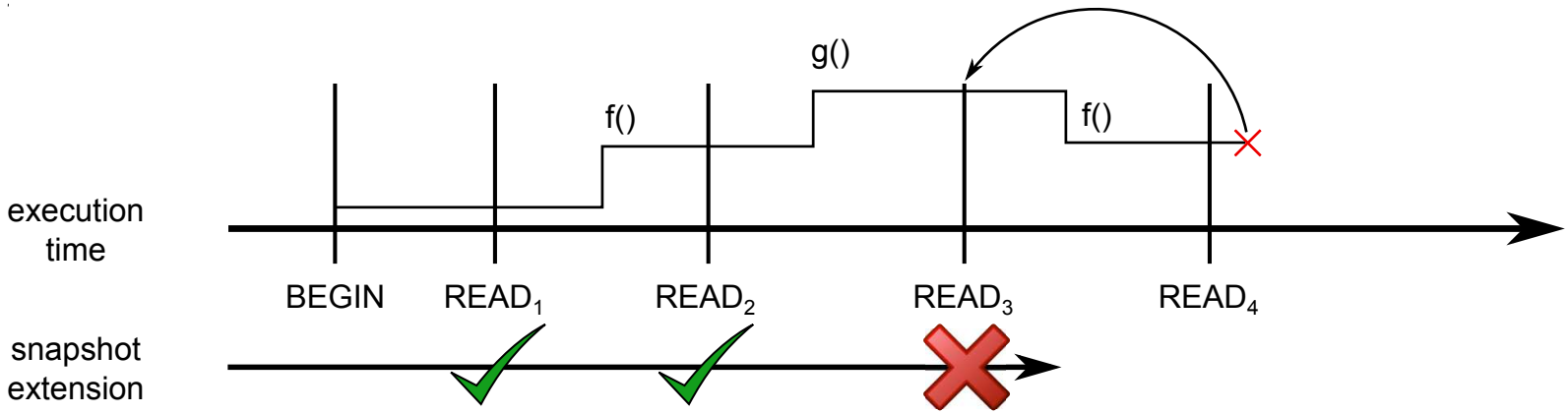
Partial Rollback: Algorithm Sketch

1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point



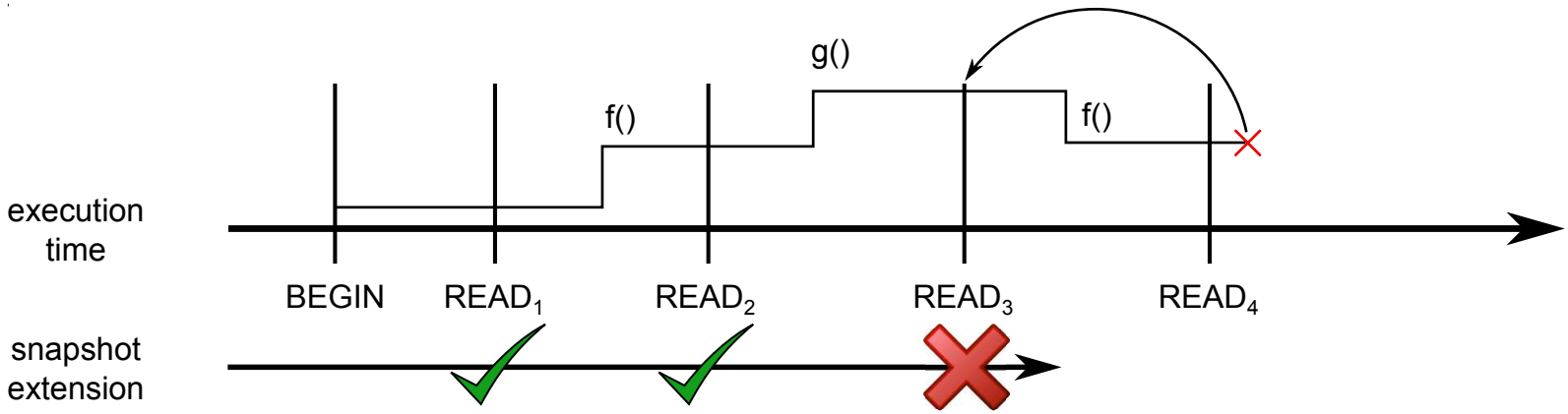
Partial Rollback: Algorithm Sketch

1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point



Partial Rollback: Algorithm Sketch

1. When a read conflict arises, we sequentially extend the snapshot:
   - All previous reads are revalidated
   - The first invalid read is the execution restarting point



2. Coherency between read/write sets is ensured by determining causality relations

Partial Rollback: Algorithm Sketch

3. Thread-private data are transparently logged
   - This is necessary to reproduce the same execution path within the transaction

```
1  int i, j;
2
3  atomic {
4      for(i = 0; i < 10; i++) {
5          j = shared_read(&shared_object);
6          if(i == j) {
7              shared_write(&other_object, j);
8          }
9      }
10 }
```

Partial Rollback: Algorithm Sketch

3. Thread-private data are transparently logged
   - This is necessary to reproduce the same execution path within the transaction

```
1  void f(int *j) {
2    int i;
3
4    atomic {
5        for(i = 0; i < 10; i++) {
6            *j = shared_read(&shared_object);
7            if(i == *j) {
8                shared_write(&other_object, *j);
9            }
10       }
11   }
12 }
```

Partial Rollback: Algorithm Sketch

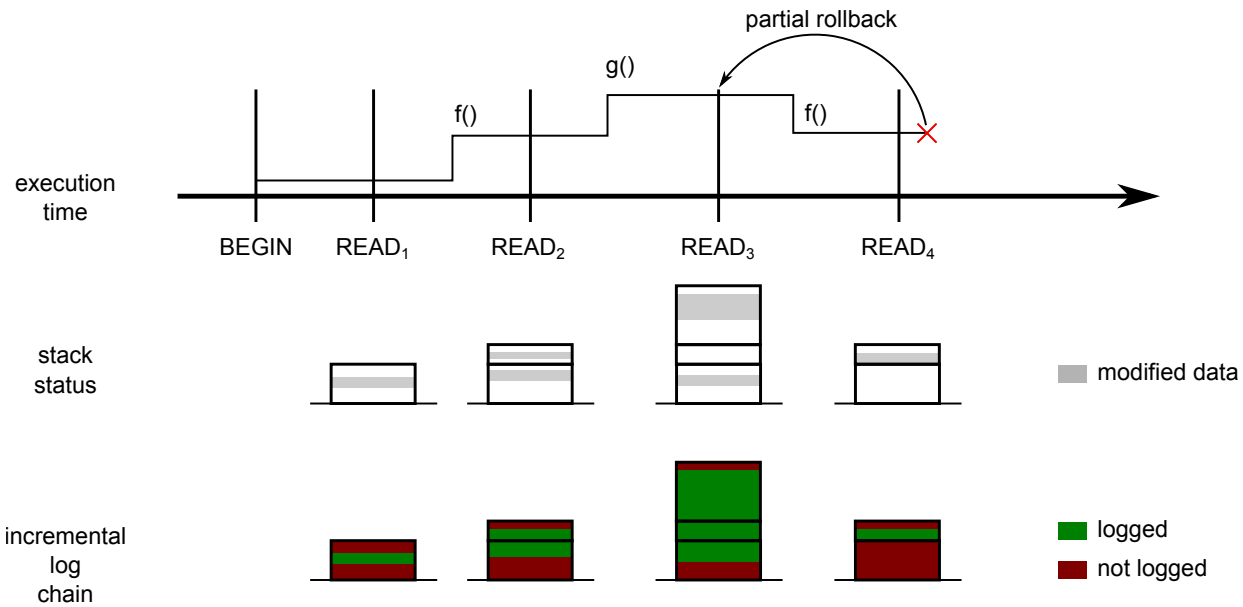
3. Thread-private data are transparently logged
   - The application-level code is instrumented by relying on Hijacker, an assembly-level code manipulator, targeting x86/x86_64 assembly code
     - `mov` instruction are identified at compile time
     - a lightweight assembly module is inserted before them
     - it fastly computes the target address of the memory update operation

Partial Rollback: Algorithm Sketch

3. Thread-private data are transparently logged
   - The application-level code is instrumented by relying on Hijacker, an assembly-level code manipulator, targeting x86/x86_64 assembly code
     - `mov` instruction are identified at compile time
     - a lightweight assembly module is inserted before them
     - it fastly computes the target address of the memory update operation
   - When a Tx begins, a stack-update interval is defined $I = [sp, sp]$
   - The computed destination/size of `mov` instructions is used to update the boundaries of I
   - Before a transactional read, Hijacker inserts code to log the stack-update interval and CPU state
   - After the stack-update interval is logged, we again set $I = [sp, sp]$

Partial Rollback: Algorithm Sketch

4. Upon a partial rollback, the CPU state and stack state are put back in place
   - CPU states are managed via System V `getcontext()`/`setcontext()`
   - A stack restore is performed incrementally
   - The stack log chain is backward traversed
   - Already-updated portions are no longer updated



Advantages of the approach

- We enforce full transparency: no modification to the application code is required
- Update-stack regions are contiguous: efficient logs via optimized `movs` instructions
- Pointer variables are trivially handled
- Compilers can optimize out stack frames without affecting correctness

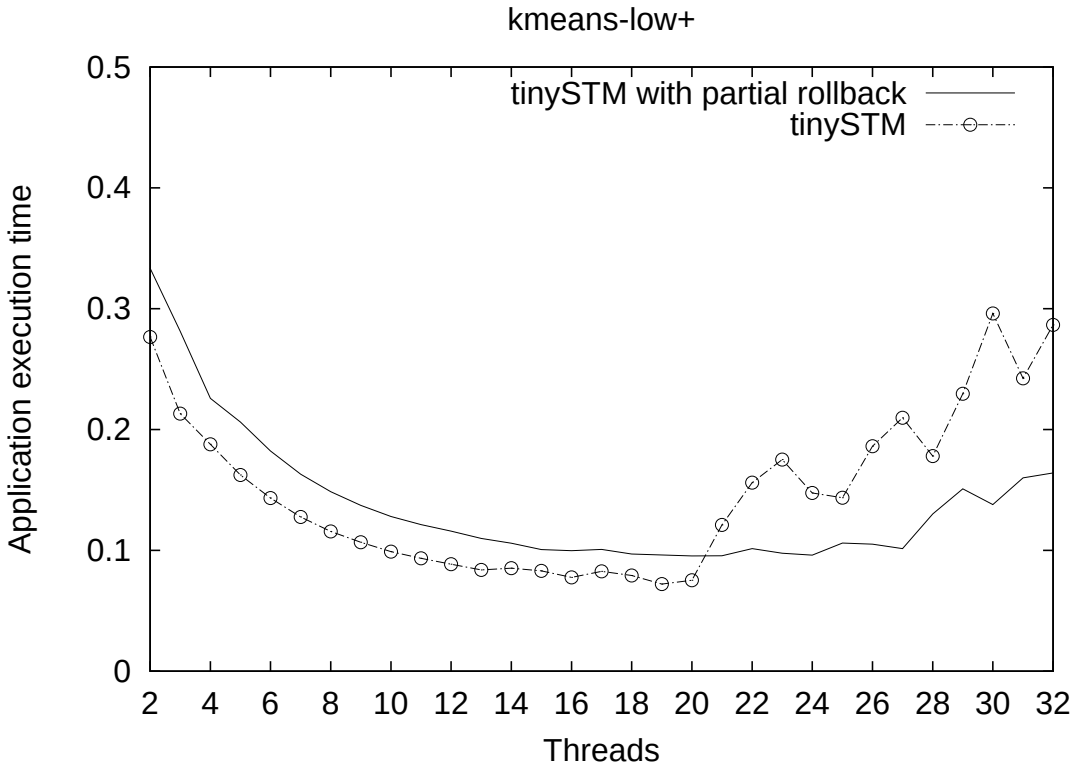
Experimental Results

- Implementation within TinySTM
- Reference benchmark: STAMP STM suite
  - ssca2
  - kmeans
- No bias towards conflicting in early phase of Txs
- We vary the amount of data on which they operate

- 32-core HP ProLiant server, NUMA architecture
- 64 Gb RAM
- Linux Kernel 2.6.32-5-amd64 – Debian 6

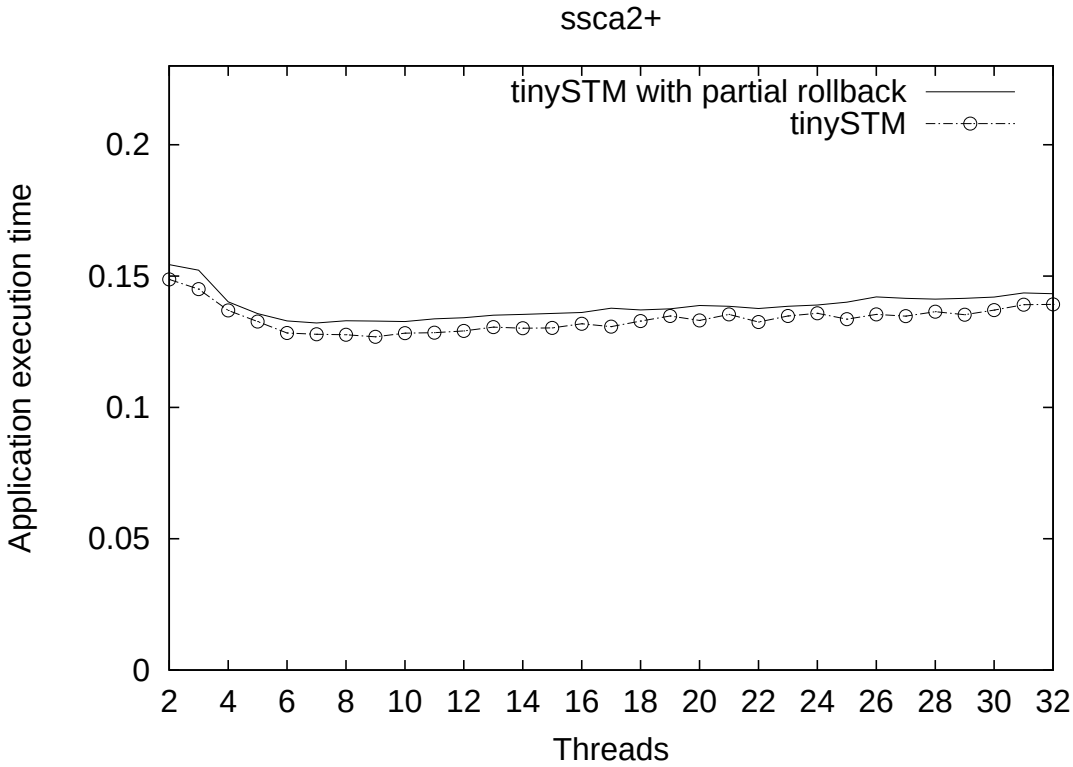
Experimental Results

- Medium transactional computation, small data set: *high contention*
- Execution time reduced by 40%



Experimental Results

- Small transactional computation, medium data set: *low contention*
- Limited overhead, $\leq 7\%$



Thanks for your attention

Questions?

pellegrini@dis.uniroma1.it
http://www.dis.uniroma1.it/∼pellegrini
http://www.dis.uniroma1.it/∼hpdcs