EXPLOITING SEMANTIC COMMUTATIVITY IN HARDWARE SPECULATION

GUOWEI ZHANG, VIRGINIA CHIU, DANIEL SANCHEZ

MICRO 2016

Massachusetts Institute of Technology

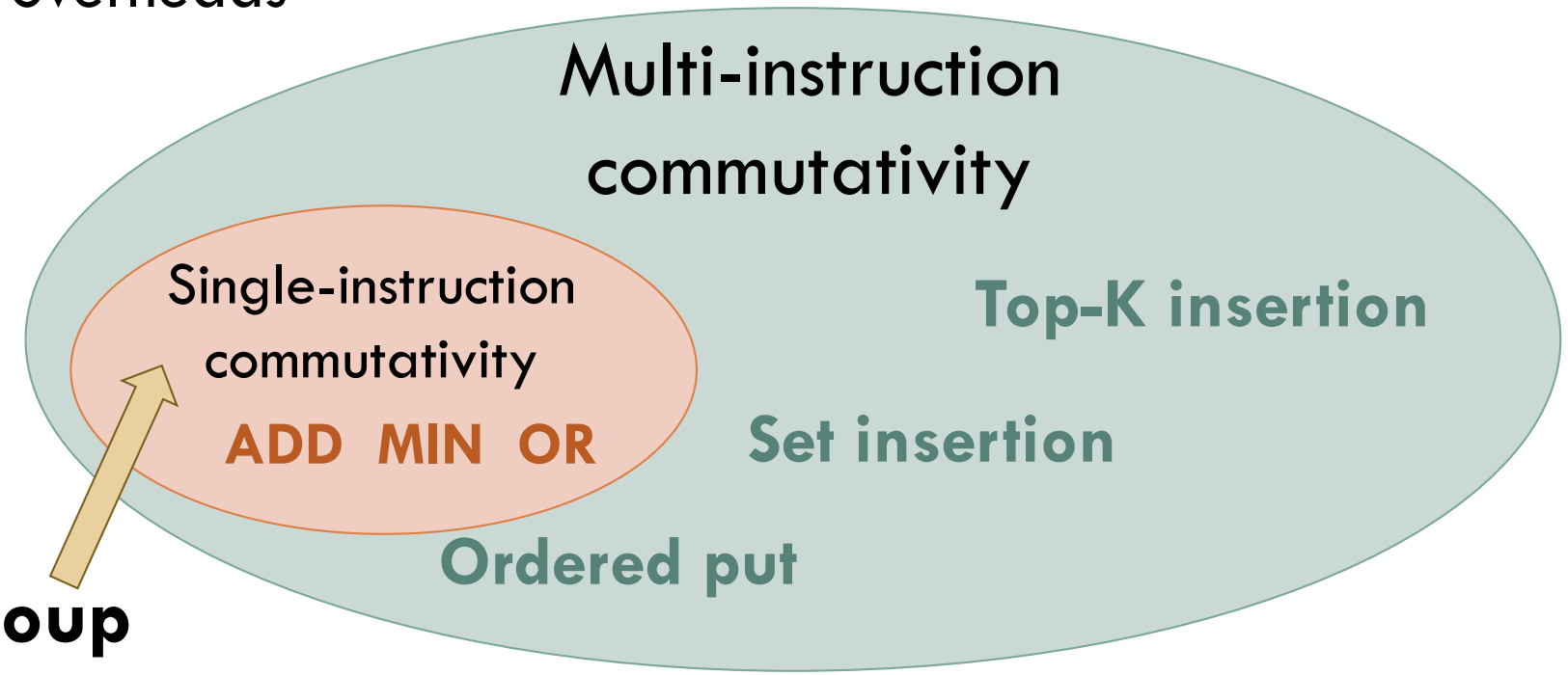
Executive summary

- Exploiting commutativity benefits update-heavy apps
  - Software techniques that exploit commutativity incur high run-time overheads (STM is 2-6x slower than HTM)
  - Prior hardware exploits only single-instruction commutative operations (e.g., addition)

- CommTM exploits multi-instruction commutativity
  - Extends coherence protocol to perform commutative operations **locally and concurrently**
  - Leverages HTM to support **multi-instruction** updates
  - Benefits speculative execution by **reducing conflicts**
  - Accelerates full applications by up to 3.4x at 128 cores

Commutativity

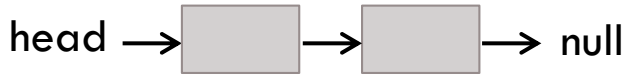
- Commutative operations produce equivalent results when reordered
  - **No true data dependence** → No need for communication
  - Software exploits commutativity but incurs high run-time overheads



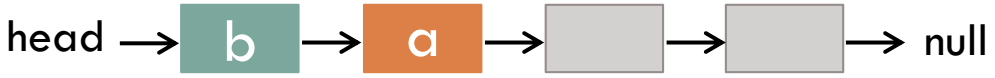
[Zhang et al, MICRO 2015]

Commutativity

- Commutative operations produce equivalent results when reordered
  - **No true data dependence** → No need for communication
  - Software exploits commutativity but incurs high run-time overheads
  - Multi-instruction example: **set (linked-list) insertion**



insert(a); insert(b);

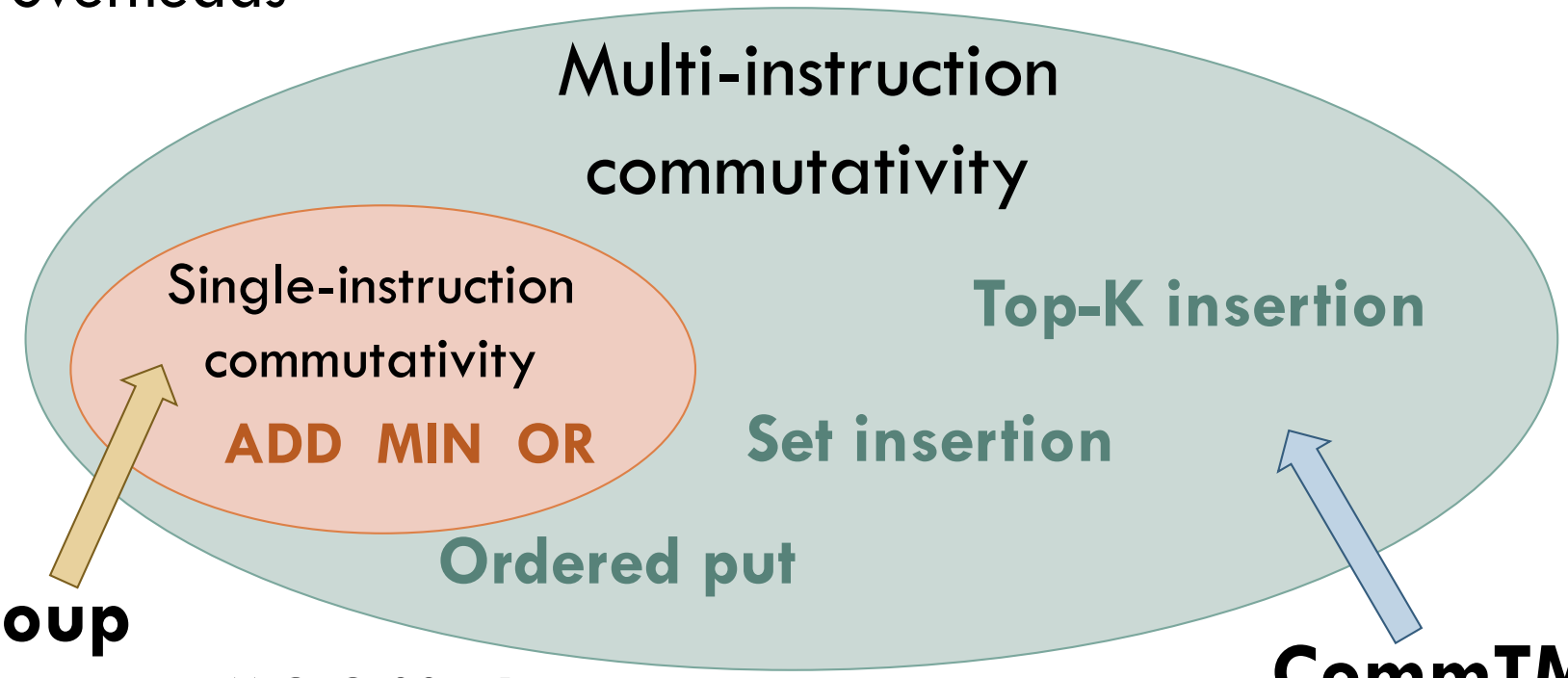


insert(b); insert(a);

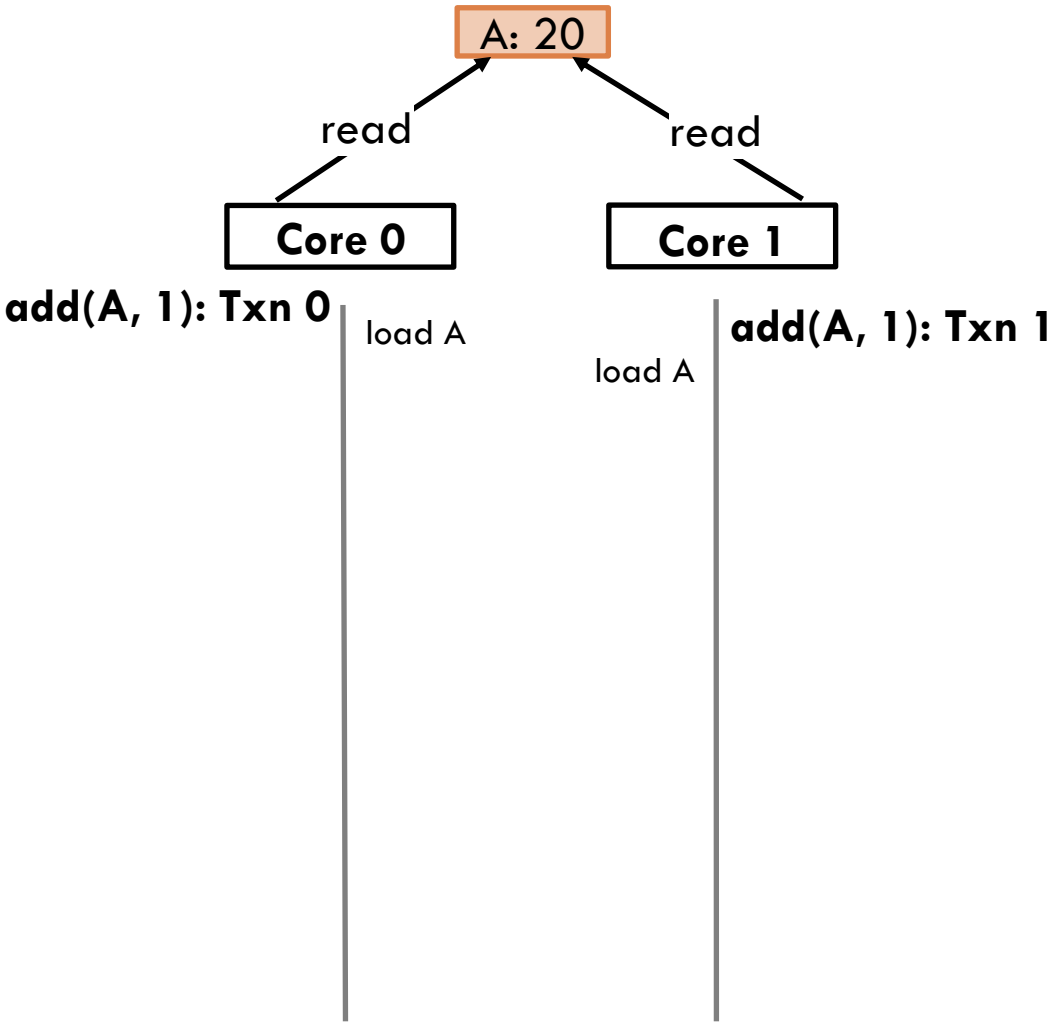
Different but semantically equivalent states

Commutativity

- Commutative operations produce equivalent results when reordered
  - **No true data dependence** → No need for communication
  - Software exploits commutativity but incurs high run-time overheads



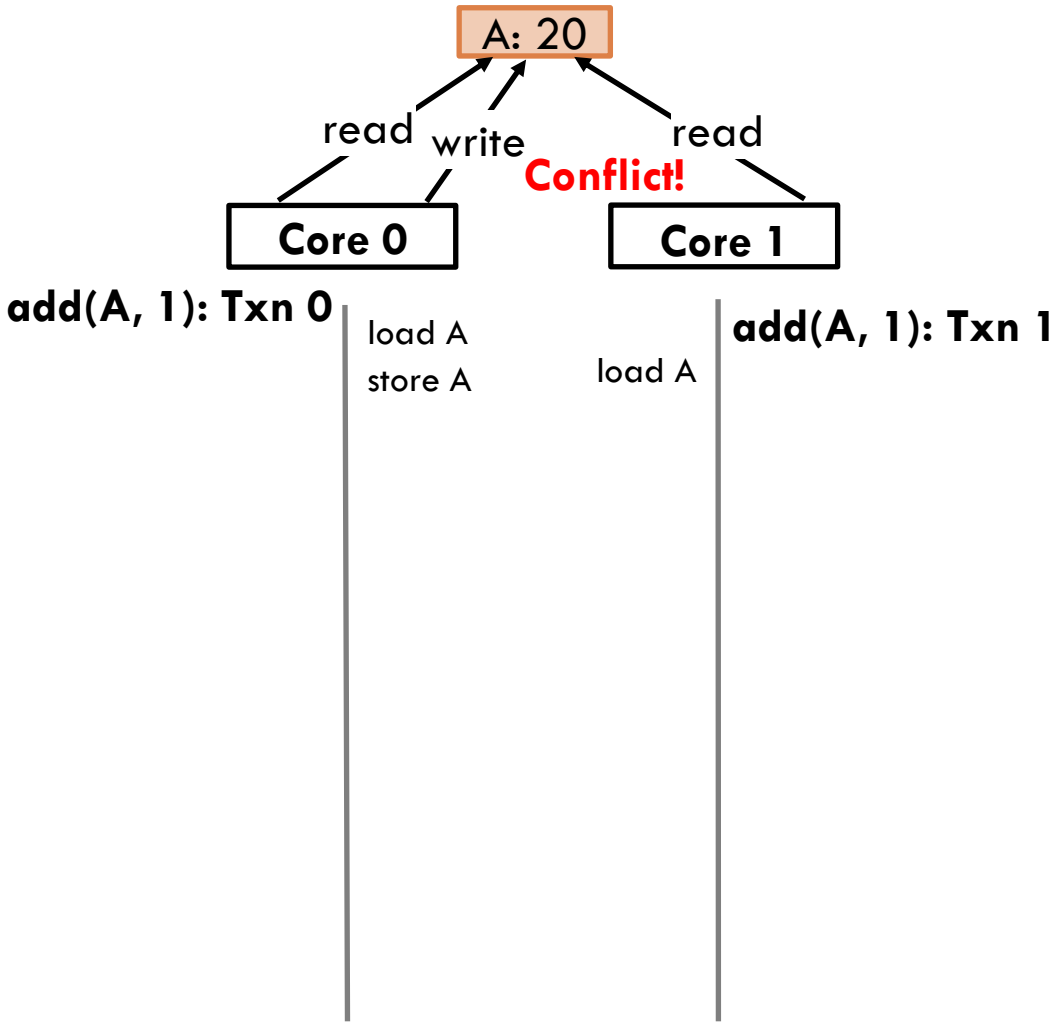
Example: addition in conventional HTM




```
void add (int* counter, int delta) {
    tx_begin();
    int v = load(counter);
    int nv = v + delta;
    store(counter, nv);
    tx_end();
}
```

Core 0	Core 1
add(A, 1); add(A, 1);	add(A, 1); add(A, 1);

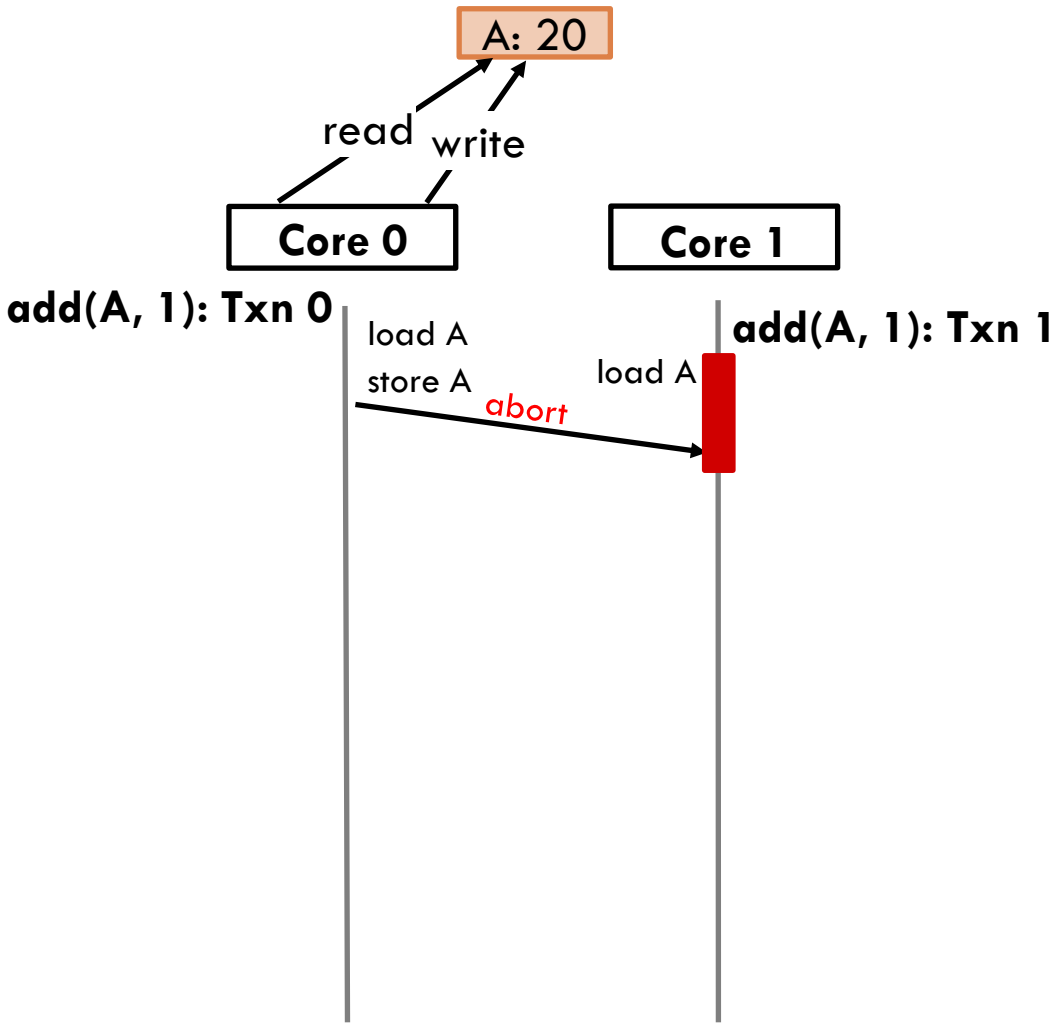
Example: addition in conventional HTM




```
void add (int* counter, int delta) {
    tx_begin();
    int v = load(counter);
    int nv = v + delta;
    store(counter, nv);
    tx_end();
}
```

Core 0	Core 1
add(A, 1); add(A, 1);	add(A, 1); add(A, 1);

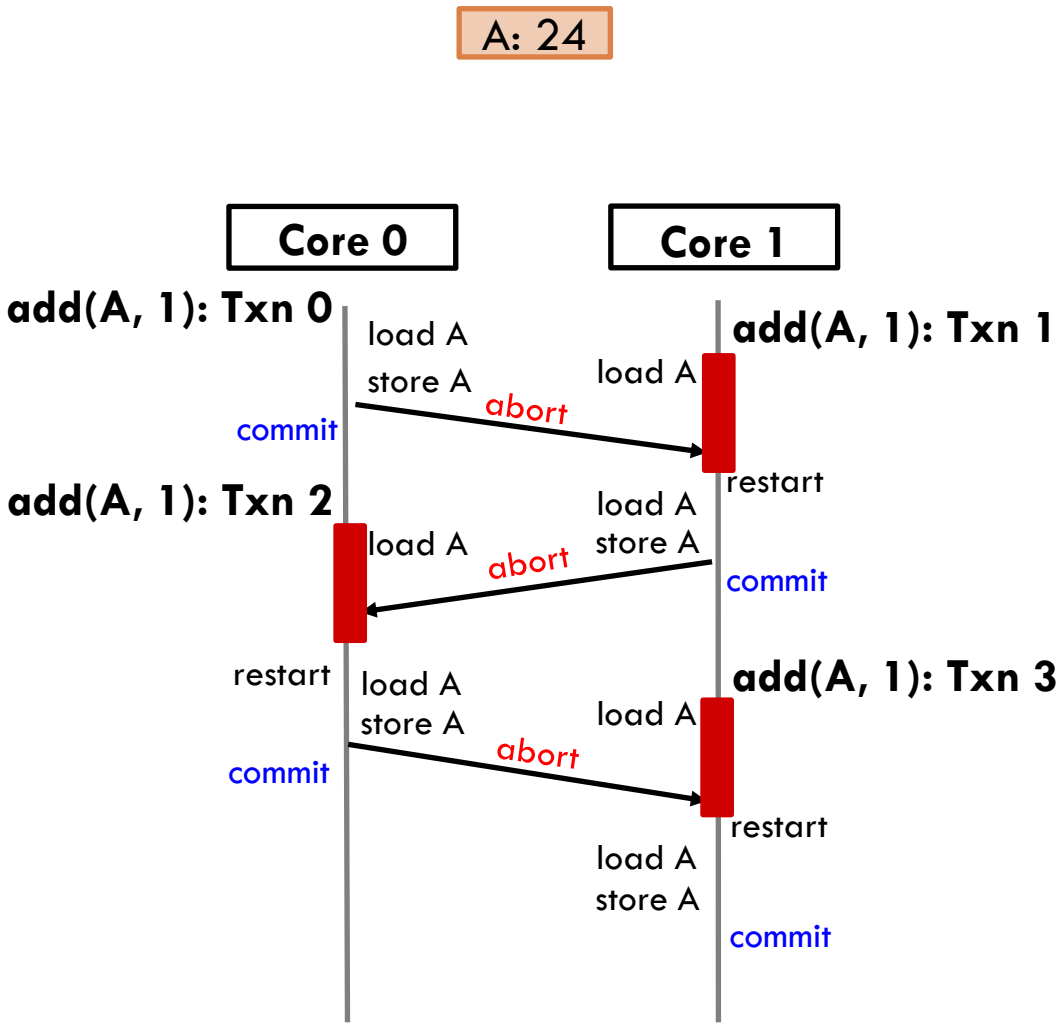
Example: addition in conventional HTM




```
void add (int* counter, int delta) {
    tx_begin();
    int v = load(counter);
    int nv = v + delta;
    store(counter, nv);
    tx_end();
}
```

Core 0	Core 1
add(A, 1); add(A, 1);	add(A, 1); add(A, 1);

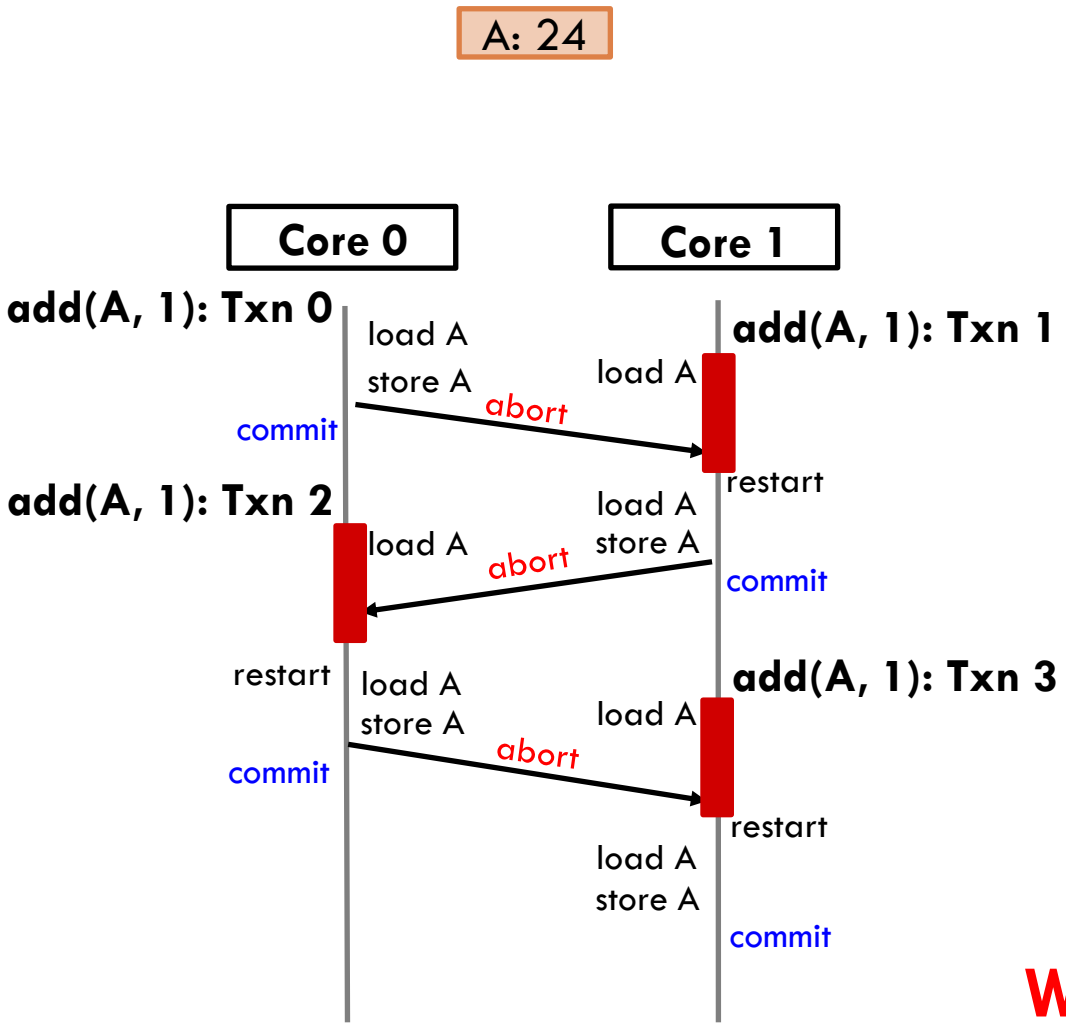
Example: addition in conventional HTM




```
void add (int* counter, int delta) {
    tx_begin();
    int v = load(counter);
    int nv = v + delta;
    store(counter, nv);
    tx_end();
}
```

Core 0	Core 1
add(A, 1); add(A, 1);	add(A, 1); add(A, 1);

Example: addition in conventional HTM




```
void add (int* counter, int delta) {
    tx_begin();
    int v = load(counter);
    int nv = v + delta;
    store(counter, nv);
    tx_end();
}
```

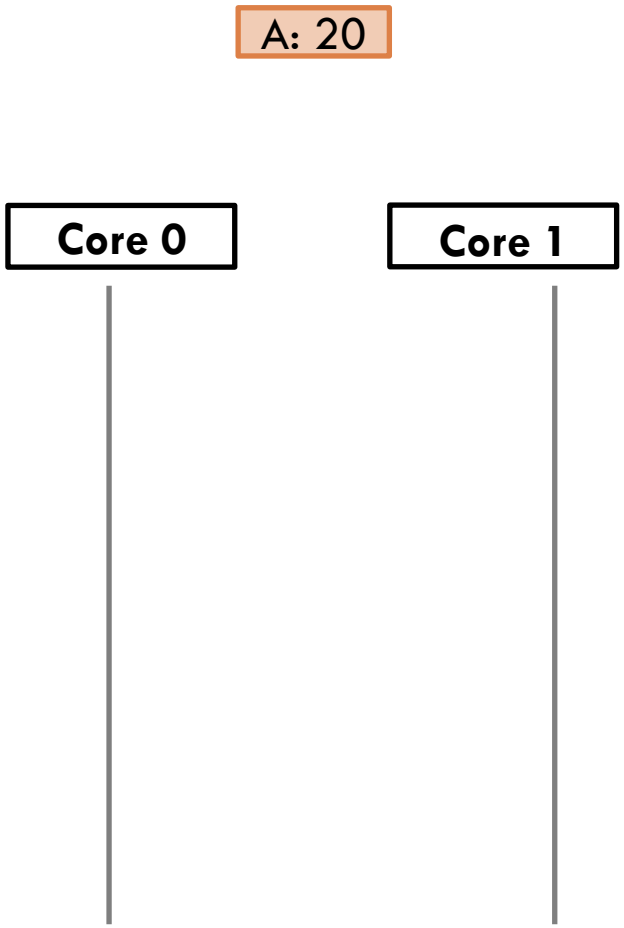
Core 0	Core 1
add(A, 1); add(A, 1);	add(A, 1); add(A, 1);

Traffic

Serialization

Wasted transactional work

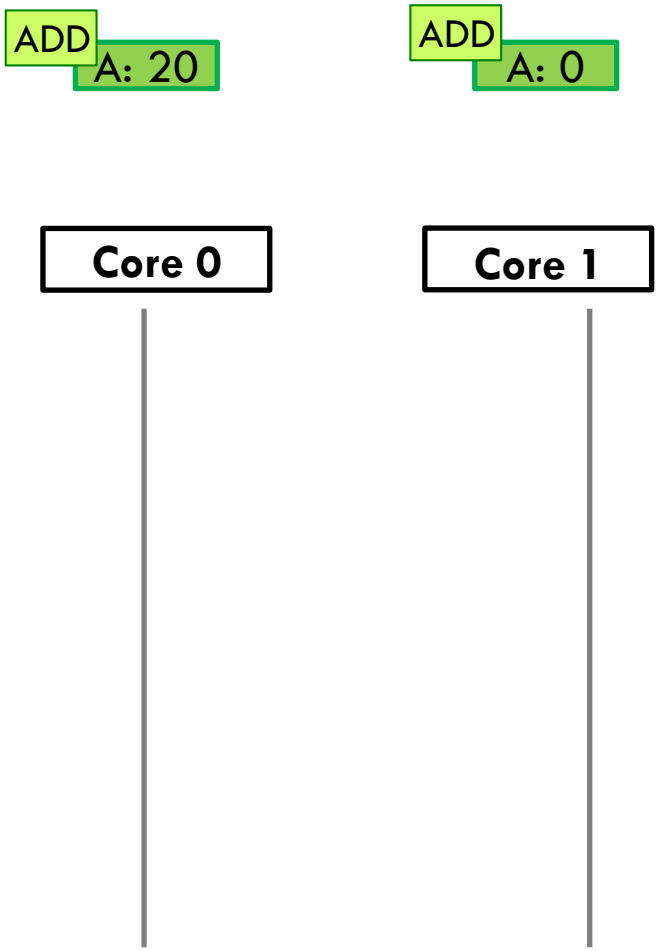
Example: addition in CommTM




```
void add (int* counter, int delta) {
        tx_begin();
        int v = load[ADD](counter);
        int nv = v + delta;
        store[ADD](counter, nv);
        tx_end();
}
```

Core 0	Core 1
add(A, 1); add(A, 1);	add(A, 1); add(A, 1);

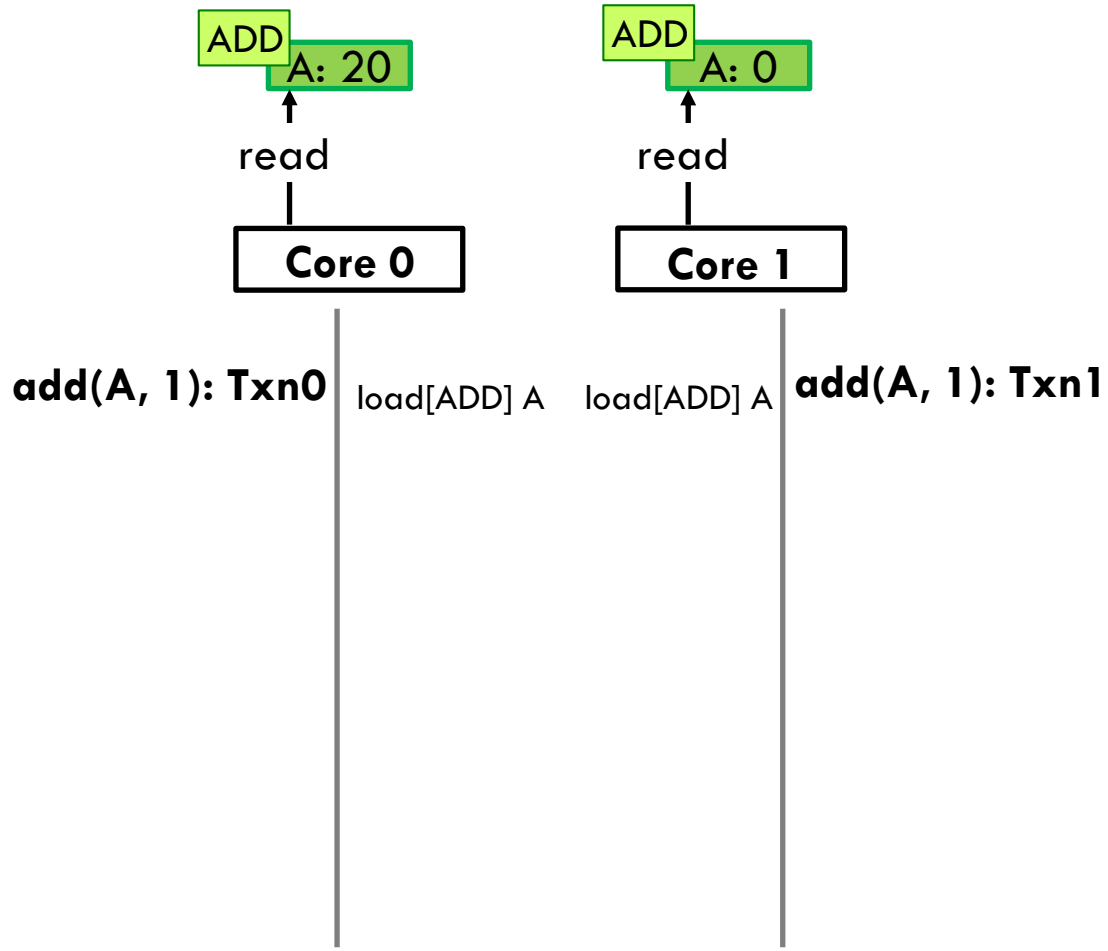
Example: addition in CommTM




```
void add (int* counter, int delta) {
        tx_begin();
        int v = load[ADD](counter);
        int nv = v + delta;
        store[ADD](counter, nv);
        tx_end();
}
```

Core 0	Core 1
add(A, 1); add(A, 1);	add(A, 1); add(A, 1);

Example: addition in CommTM




```
void add (int* counter, int delta) {
    tx_begin();
    int v = load[ADD](counter);
    int nv = v + delta;
    store[ADD](counter, nv);
    tx_end();
}
```

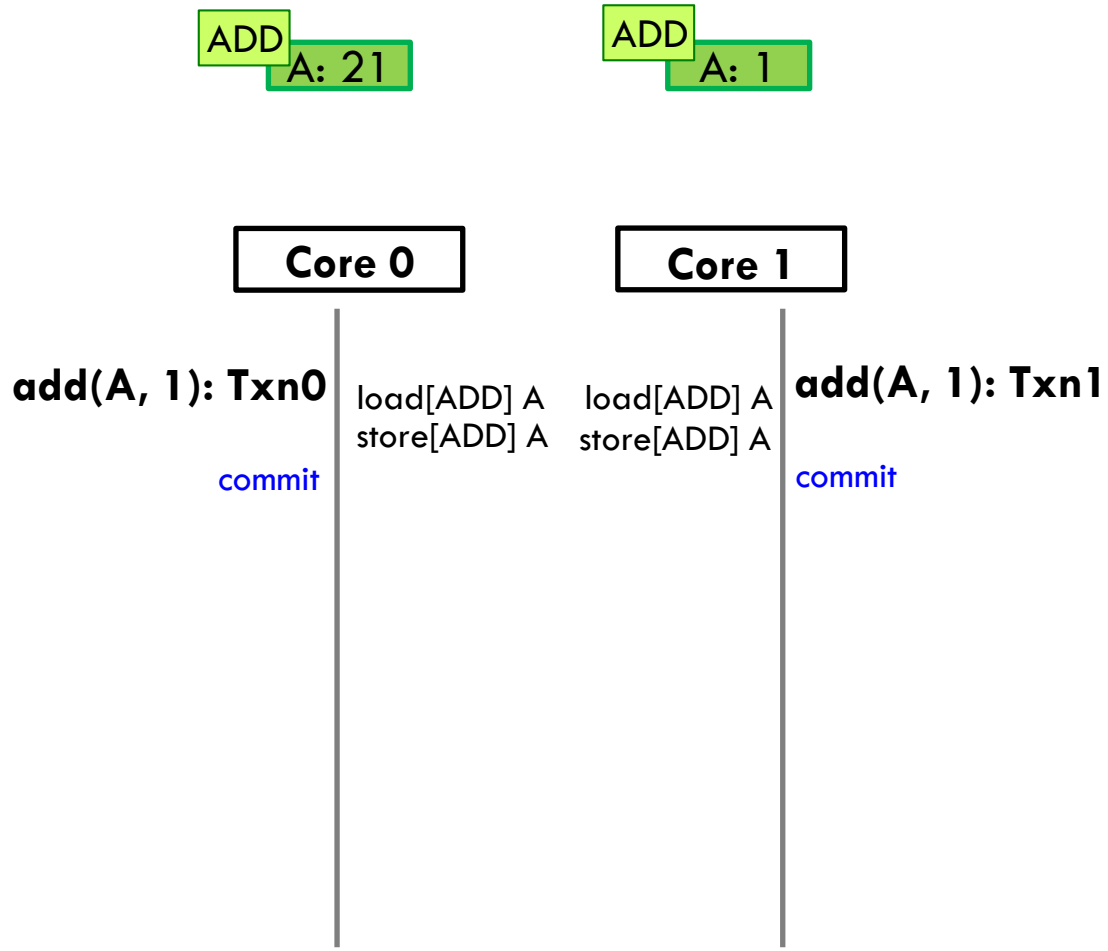
Core 0	Core 1
add(A, 1); add(A, 1);	add(A, 1); add(A, 1);

Example: addition in CommTM


```
void add (int* counter, int delta) {
    tx_begin();
    int v = load[ADD](counter);
    int nv = v + delta;
    store[ADD](counter, nv);
    tx_end();
}
```

Core 0	Core 1
add(A, 1); add(A, 1);	add(A, 1); add(A, 1);

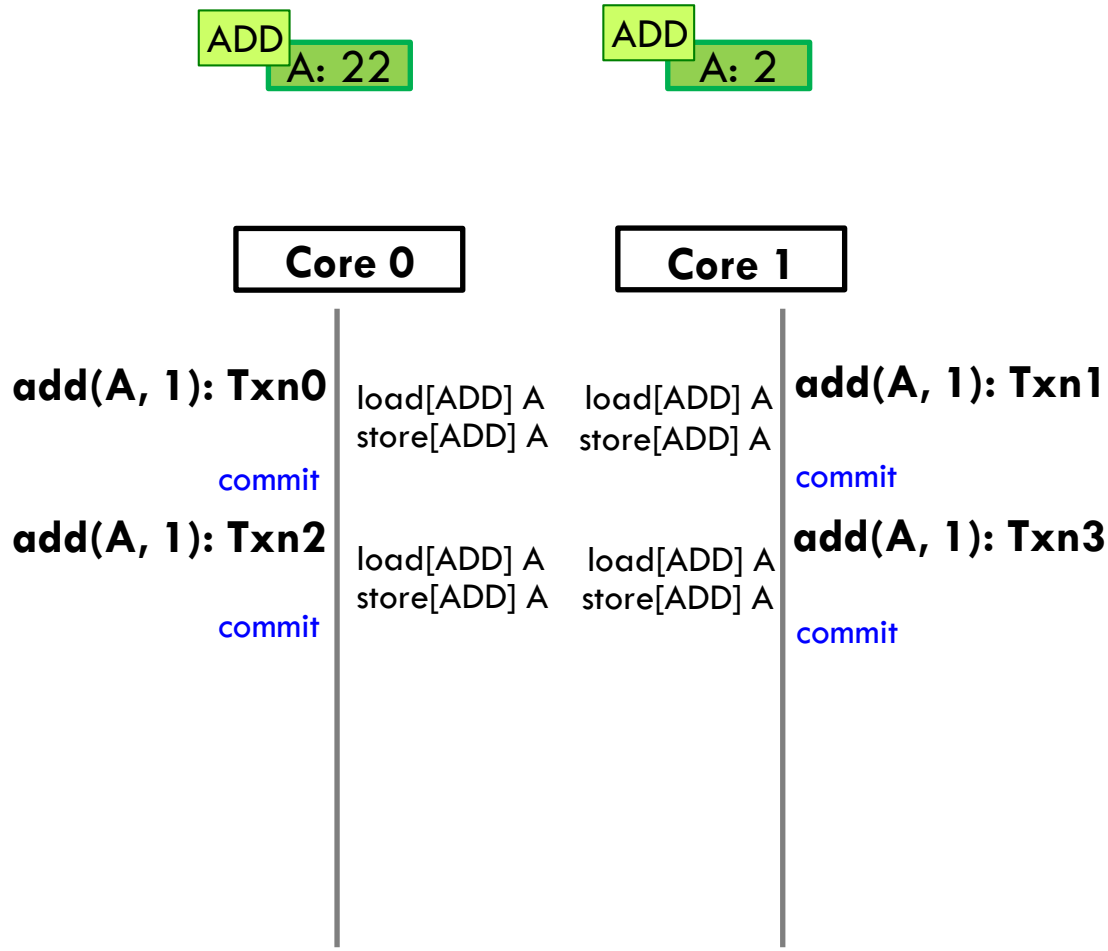
Example: addition in CommTM




```
void add (int* counter, int delta) {
    tx_begin();
    int v = load[ADD](counter);
    int nv = v + delta;
    store[ADD](counter, nv);
    tx_end();
}
```

Core 0	Core 1
add(A, 1); add(A, 1);	add(A, 1); add(A, 1);

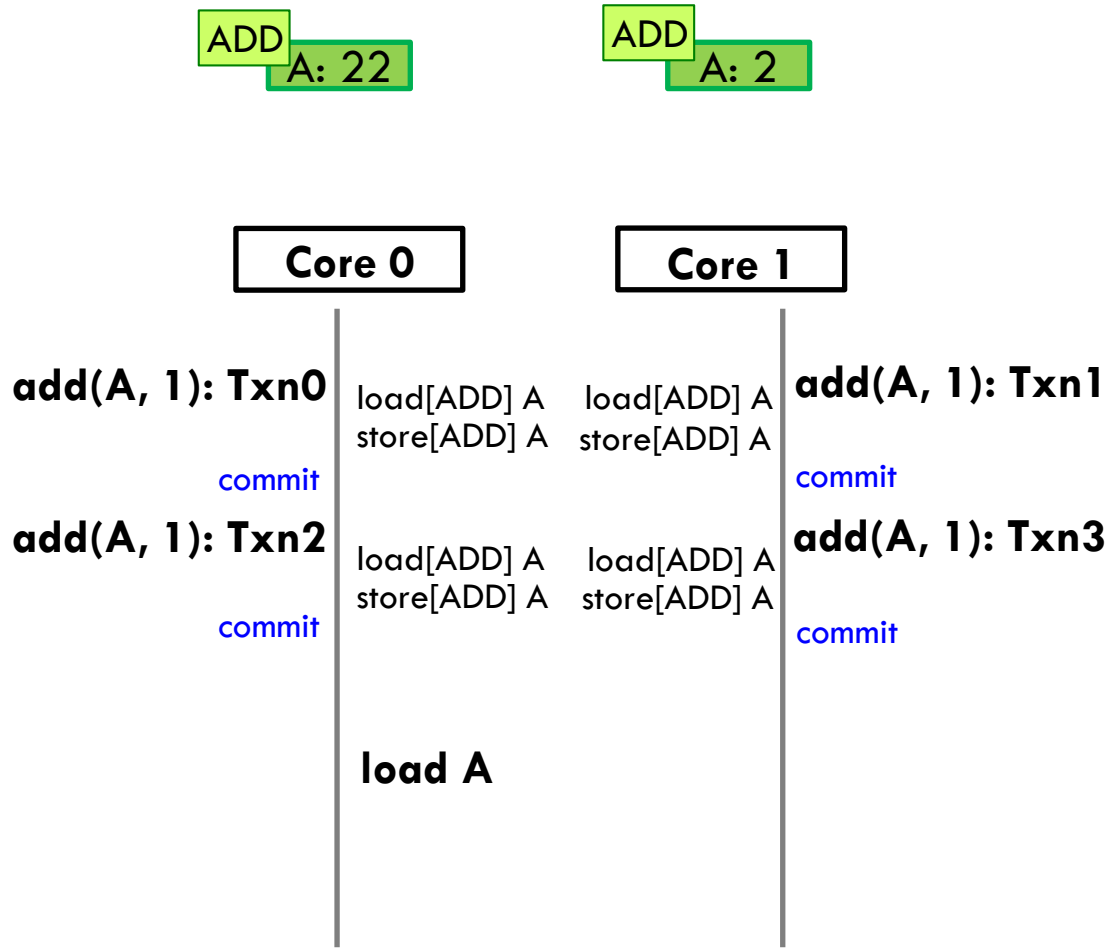
Example: addition in CommTM




```
void add (int* counter, int delta) {
    tx_begin();
    int v = load[ADD](counter);
    int nv = v + delta;
    store[ADD](counter, nv);
    tx_end();
}
```

Core 0	Core 1
add(A, 1);	add(A, 1);
add(A, 1);	add(A, 1);

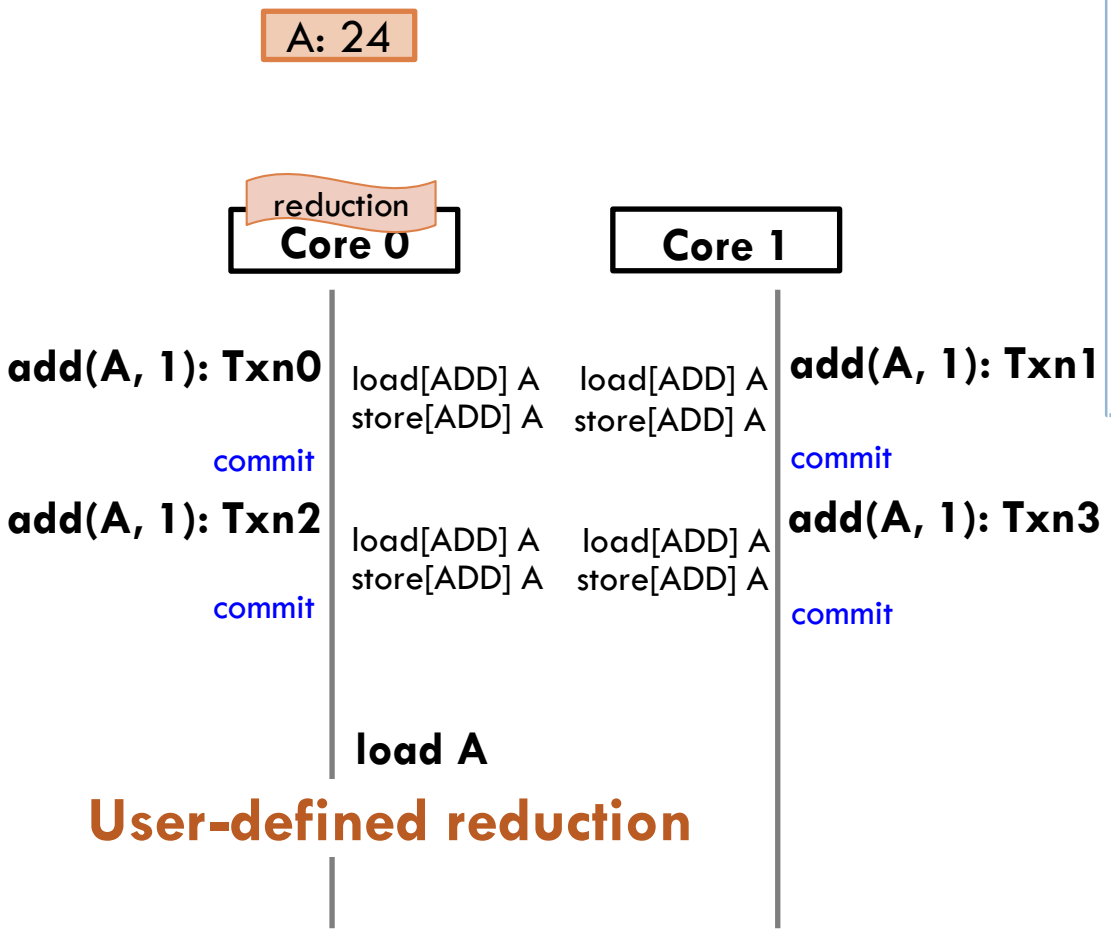
Example: addition in CommTM




```
void add (int* counter, int delta) {
    tx_begin();
    int v = load[ADD](counter);
    int nv = v + delta;
    store[ADD](counter, nv);
    tx_end();
}
```

Core 0	Core 1
add(A, 1); add(A, 1);	add(A, 1); add(A, 1);

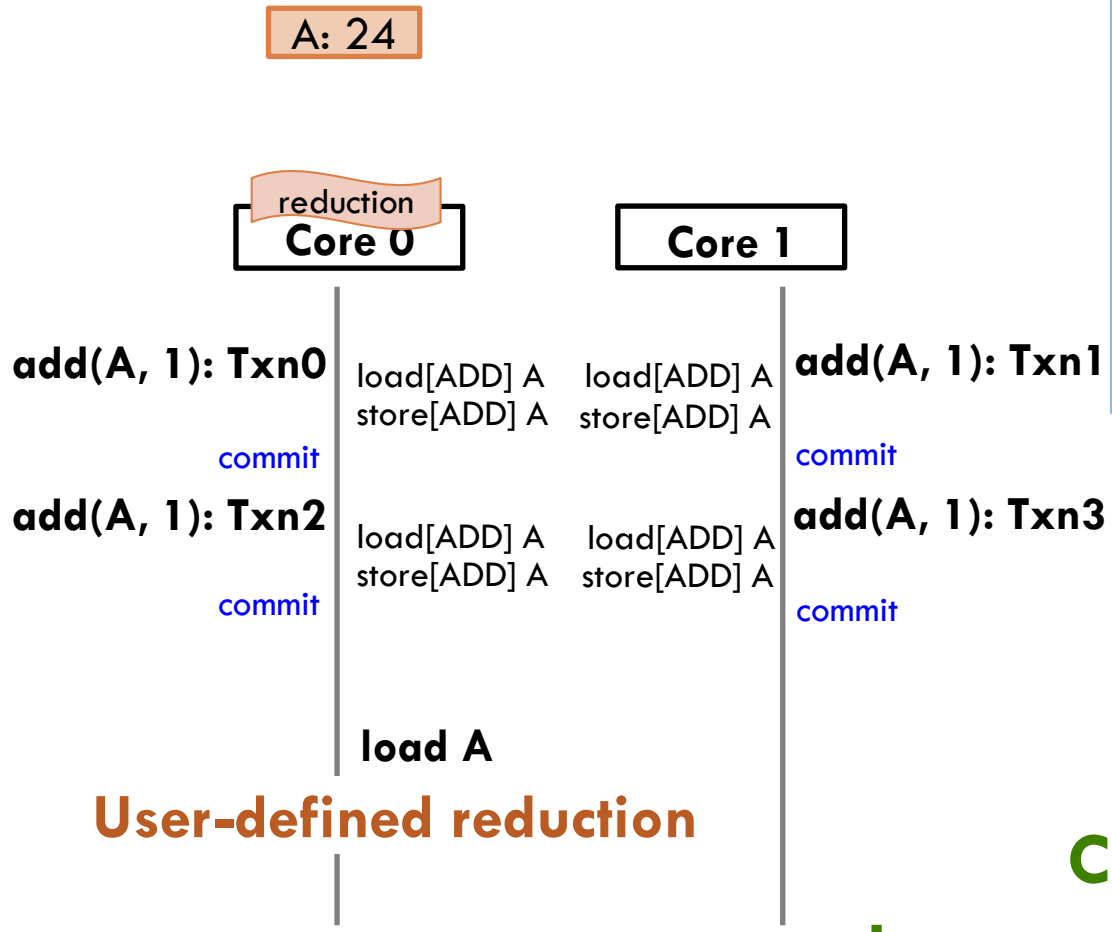
Example: addition in CommTM




```
void add (int* counter, int delta) {
    tx_begin();
    int v = load[ADD](counter);
    int nv = v + delta;
    store[ADD](counter, nv);
    tx_end();
}
```

Core 0	Core 1
add(A, 1);	add(A, 1);
add(A, 1);	add(A, 1);

Example: addition in CommTM




```
void add (int* counter, int delta) {
    tx_begin();
    int v = load[ADD](counter);
    int nv = v + delta;
    store[ADD](counter, nv);
    tx_end();
}
```

Core 0	Core 1
add(A, 1);	add(A, 1);
add(A, 1);	add(A, 1);

Less traffic

Concurrent updates

Less wasted transactional work

Less run-time/memory overheads than STM

CommTM

Programming interface

Transactional update

```
void add (int* counter, int delta) {
        tx_begin();
        int v = load(counter);
        int nv = v + delta;
        store(counter, nv);
        tx_end();
}
```

Programming interface

Transactional update

```
void add (int* counter, int delta) {
        tx_begin();
        int v = load[ADD](counter);
        int nv = v + delta;
        store[ADD](counter, nv);
        tx_end();
}
```

Labeled loads/stores

Programming interface

Transactional update

```
void add (int* counter, int delta) {
        tx_begin();
        int v = load[ADD](counter);
        int nv = v + delta;
        store[ADD](counter, nv);
        tx_end();
}
```

Labeled loads/stores

Non-transactional reduction handler

```
void reduce[ADD] (int* counter, int delta) {
        int v = load[ADD](counter);
        int nv = v + delta;
        store[ADD](counter, nv);
}
```

Programming interface

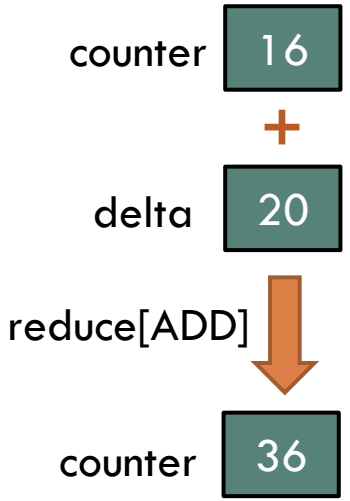
Transactional update

```
void add (int* counter, int delta) {
        tx_begin();
        int v = load[ADD](counter);
        int nv = v + delta;
        store[ADD](counter, nv);
        tx_end();
}
```

Labeled loads/stores

Non-transactional reduction handler

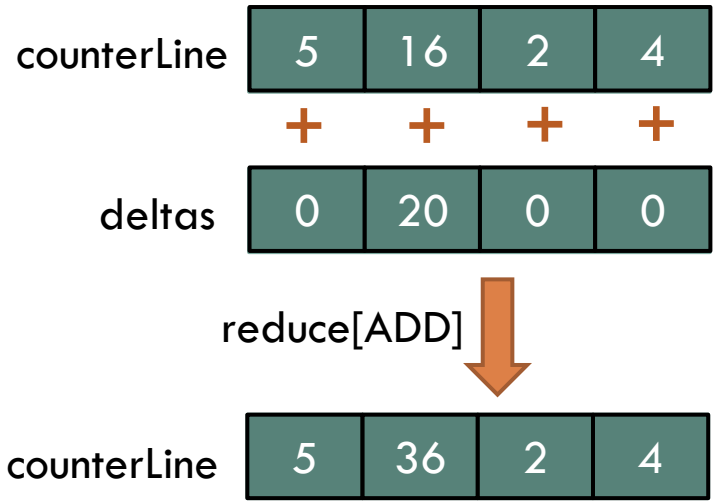
```
void reduce[ADD] (int* counter, int delta) {
        int v = load[ADD](counter);
        int nv = v + delta;
        store[ADD](counter, nv);
}
```



Handling arbitrary object sizes

- For objects smaller than a cache line, assume lines are full of aligned elements and reduce all of them

```
void reduce[ADD] (int* counterLine, int[] deltas) {
    for (int i = 0; I < intsPerCacheLine; i++) {
        int v = load[ADD](counterLine[i]);
        int nv = v + deltas[i];
        store[ADD](counterLine[i], nv);
    }
}
```



- For objects larger than a cache line, use a level of indirection

Example: set (linked-list) insertion

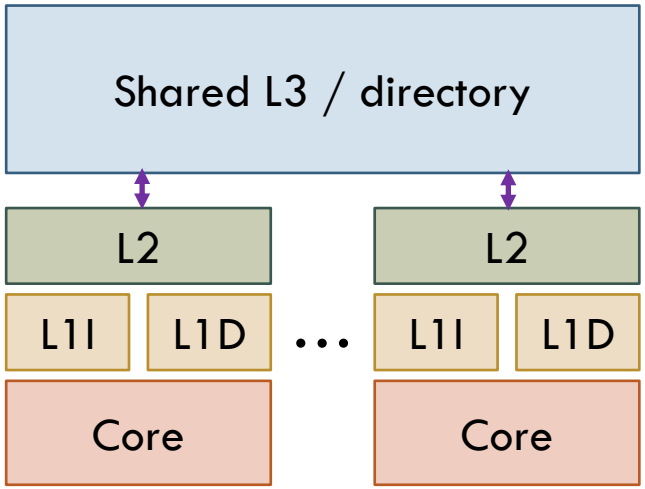

```
void insert (SetDesc* s, Node* n) {
        tx_begin();
        Node* head = load[INSERT](s);
        n->next = head;
        store[INSERT](s, n);
        if (head == nullptr)
                store[INSERT](s+sizeof(Node*), n);
        tx_end();
}
```

```
Struct SetDesc {
        Node* head;
        Node* tail;
};
Struct Node {
        …
        Node* next;
};
```

```
void reduce[INSERT] (SetDesc* s0, SetDesc* s1) {
        if (s1->head == nullptr) return;
        Node* head0 = load[INSERT](s0);
        if (head0 == nullptr) {
                store[INSERT](s0, s1->head);
        } else {
                Node* tail0 =
load[INSERT](s0+sizeof(Node*));
                tail0->next = s1->head;
        }
        store[INSERT](s0+sizeof(Node*), s1->tail);
}
```

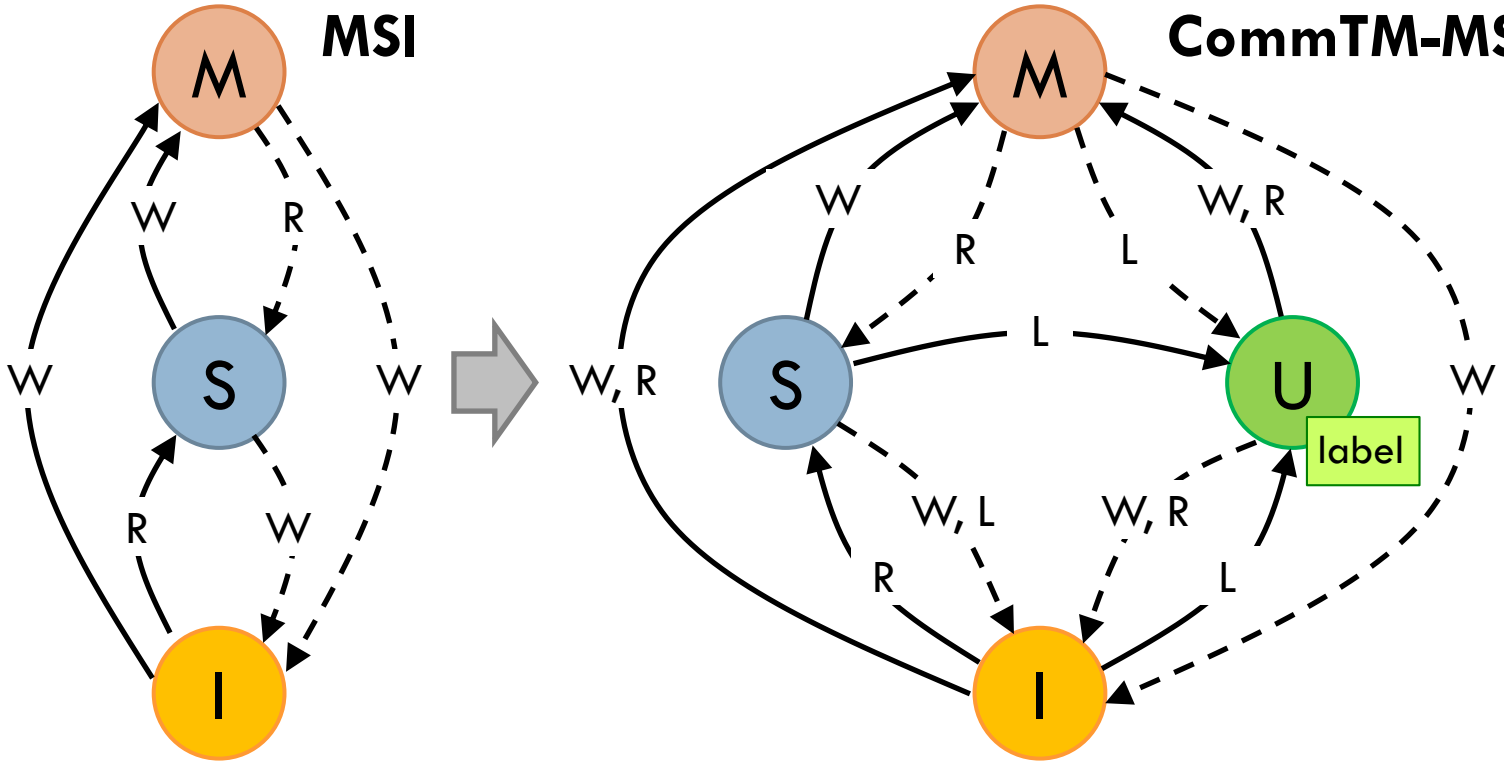
Implementation

- Baseline HTM
  - MESI coherence protocol
  - Eager conflict detection
  - Timestamp-based conflict resolution
  - Lazy version management (buffer speculative data in L1s)



CommTM can be applied to other HTMs and hardware speculation techniques

Coherence protocol



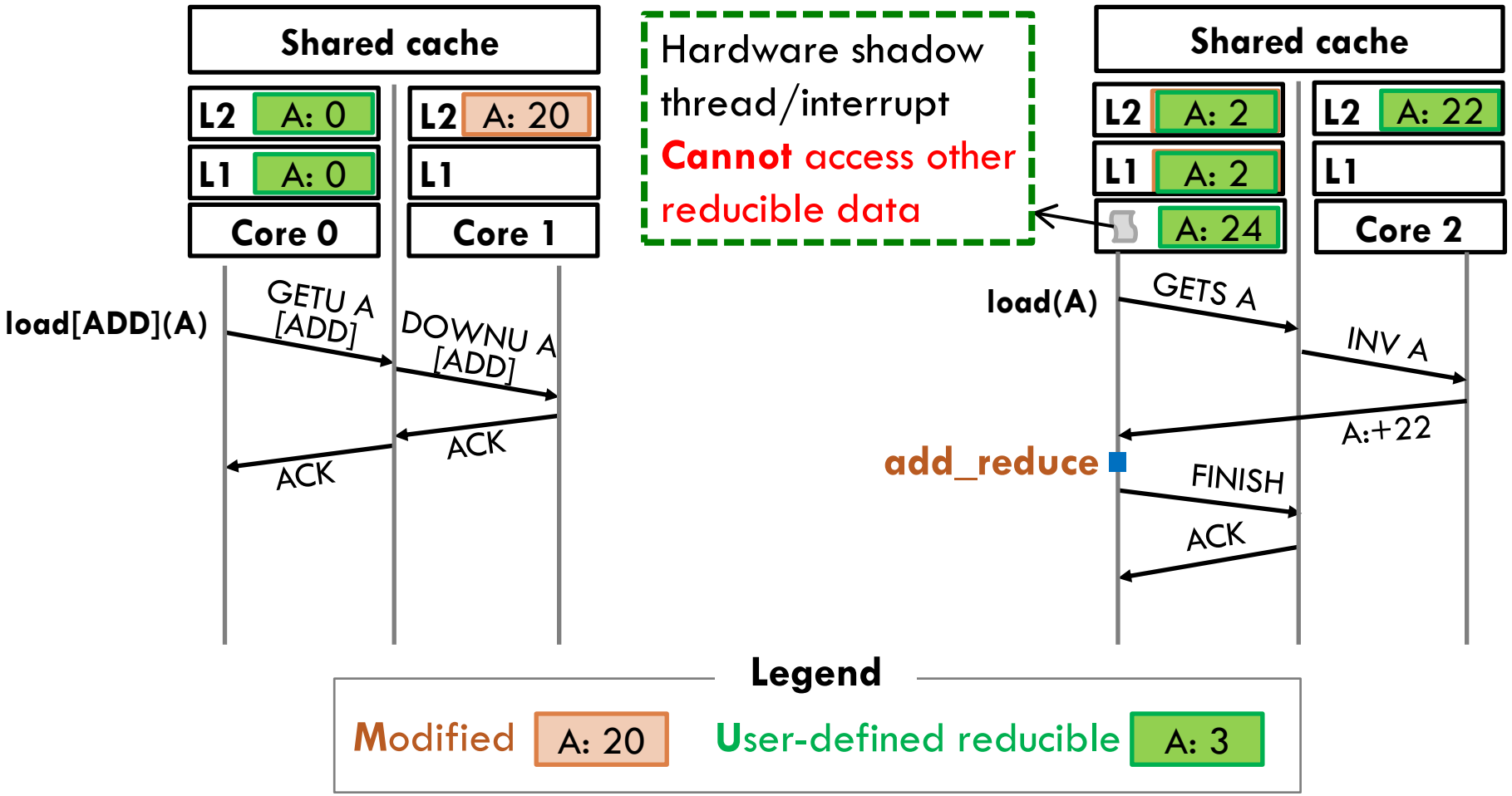
Legend

Transitions	Initiated by own core (gain permissions)				
	Initiated by others (lose permissions)				
States	Modified	**S**hared (read-only)	Invalid	**U**ser-defined reducible	
Requests	**R**ead	**W**rite	**L**abeled load/store		

Reducible-state transitions

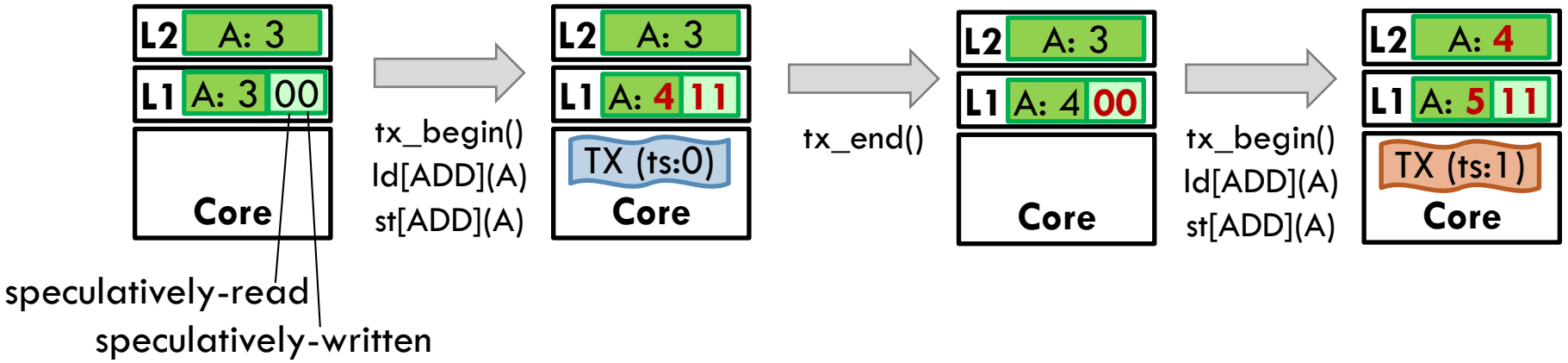
Entering U state triggered by **labeled load/store**

Leaving U state with non-transactional **reductions**



Transactional execution

- Speculative value management for U state is analogous to M state



- Invalidation to speculatively accessed data in U state triggers a **conflict**

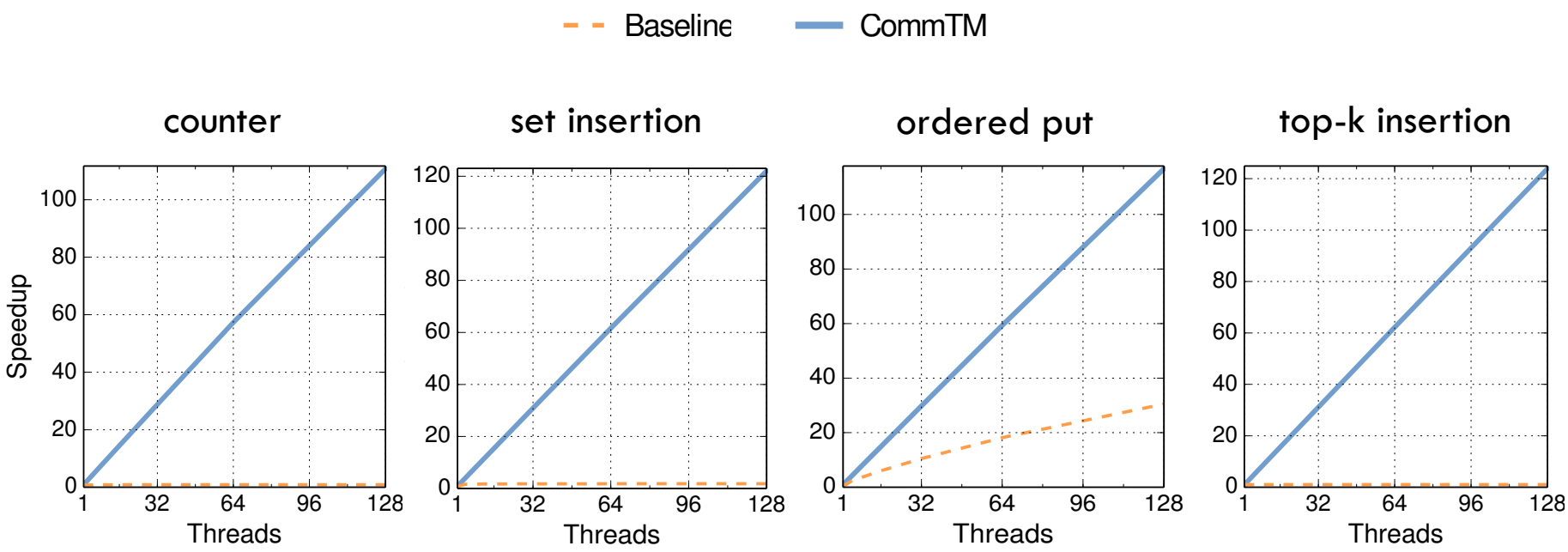
Gather requests allows more concurrency

- Motivation
  - **Conditional commutativity:** Operations commute only when reducible data meets some conditions
  - **Frequent reductions** triggered by condition checks limit concurrency

- Gather requests allow partial updates to move across caches **without leaving the reducible state**
  - Achieves higher concurrency (e.g., for reference counting)

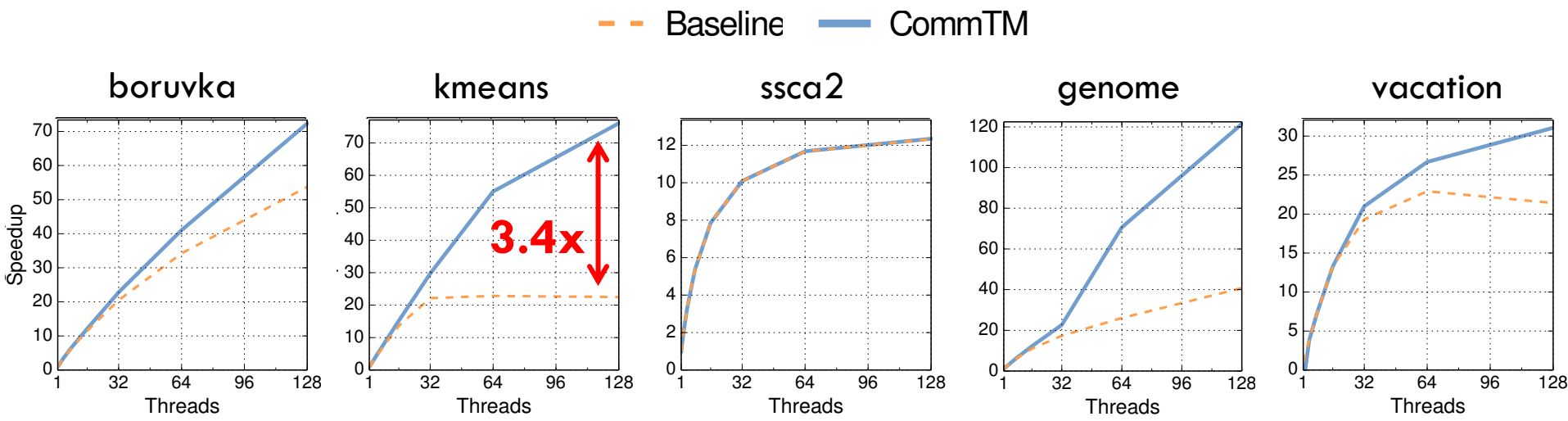
Evaluation

Evaluation on microbenchmarks

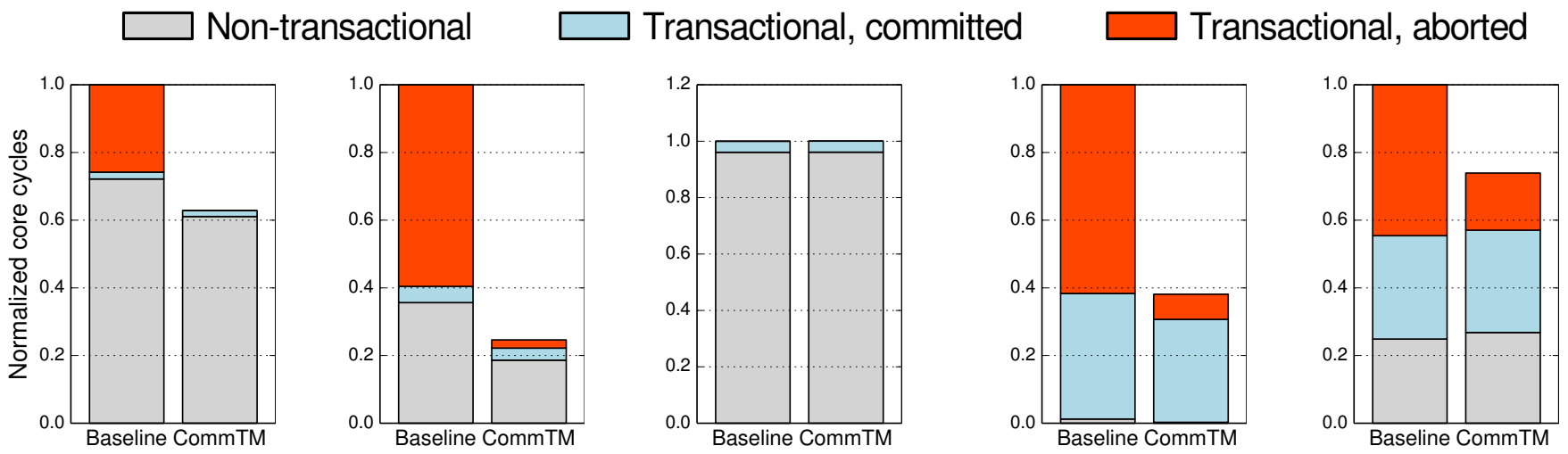


Up to 128x speedup over baseline TM

Evaluation on full applications



Breakdown of core cycles at 128 threads (lower is better)



Conclusions

- Leverages HTM to support multi-instruction operations
- Extends coherence protocol to allow local and concurrent updates
- Bridges the gap between software and hardware speculation
- Reduces conflicts and serialized transactions significantly
- Accelerates challenging workloads by up to 3.4x at 128 cores

Thanks For Your Attention!

Questions are welcome!

Massachusetts
Institute of
Technology