

Better Expressiveness for HTM *using* Split Hardware Transactions

Yossi Lev

Brown University & Sun Microsystems Laboratories

Joint work with:

Jan-Willem Maessen

Sun Microsystems Laboratories

Hardware Transactional Memory (HTM)


- Fast, often supports Strong Atomicity
- Restrictions
 - Resource limitations: bounded size
 - ➔ – Expressiveness limitations
 - No true closed/open nesting →
No “nesting constructs”: or-else, retry, etc.
 - No debugging support
 - No long transactions (*context switch, timer interrupts*)

This Work

Improve expressiveness of best effort HTM
using simple software support

- True closed nesting of transactions
- Pause a transaction & Inspect its read/write sets
→ *Debugging, Open Nesting*
- Better survive long transactions

How? Segmentation

- Introduce *Split Hardware Transaction (SpHT)*
 - A split transaction consists of multiple *segments*
 - Each segment executed by its own hardware transaction  *Get strong atomicity
If supported by HTM*
 - using minimal software support combine all to *one atomic operation*

Our goal is to overcome expressiveness limitations
Not to overcome resource limitations

The SpHT Algorithm

- Each segment is executed by a hw transaction.
- **Defer Writes:** log in a write-set
 - **Isolation:** split transaction's partial state not exposed when committing segment's hw transaction
- **Log Reads**, and **validate** them at segment's beginning
 - **Consistency:** address read by *any* segment changes → *current* hw transaction fails.
- **Copy back** by last hw transaction: write-set → shared memory
 - **Atomicity:** last hw transaction runs *all* reads & writes of split transaction.

Hardware: conflict detection. Guarantees atomicity.
Software: read/write tracking. Enhances flexibility.

SpHT for Closed Nesting

- **SpHT_Begin**
begins a (nested) transaction, starting its first segment
- **SpHT_Commit**
ends a (nested) transaction, ending its last segment
- **SpHT_Pause**
pauses a transaction, ending its current segment
- **SpHT_Resume**
resumes a transaction, starting a new segment
- **SpHT_Read / SpHT_Write**
read/write ops, checking & updating the read/write sets

- Segmentation for Closed Nesting
 - End a segment before a nested transaction, Begins a new one after it.

```
atomic {
```

```
  a++
```

```
  atomic {
```

```
    foo()
```

```
  }
```

```
  b++
```

```
}
```

Segmentation for Closed Nesting

- End a segment before a nested transaction, Begins a new one after it.

atomic {

a++

atomic {

foo()

}

b++



SpHT begin()

SpHT_write(&a,1+SpHT_read(&a))

SpHT_pause()

SpHT_begin()

foo()

SpHT_commit()

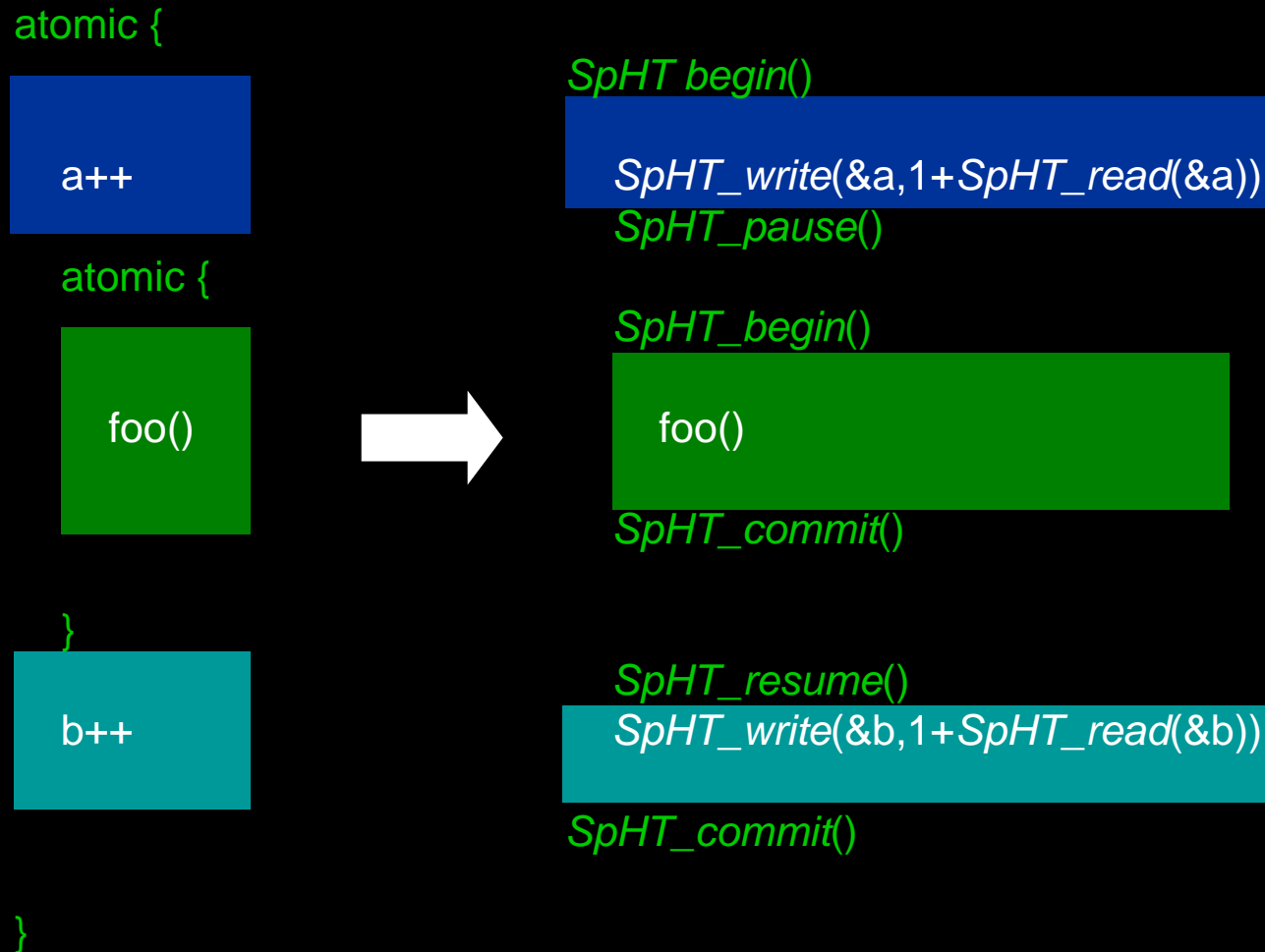
SpHT_resume()

SpHT_write(&b,1+SpHT_read(&b))

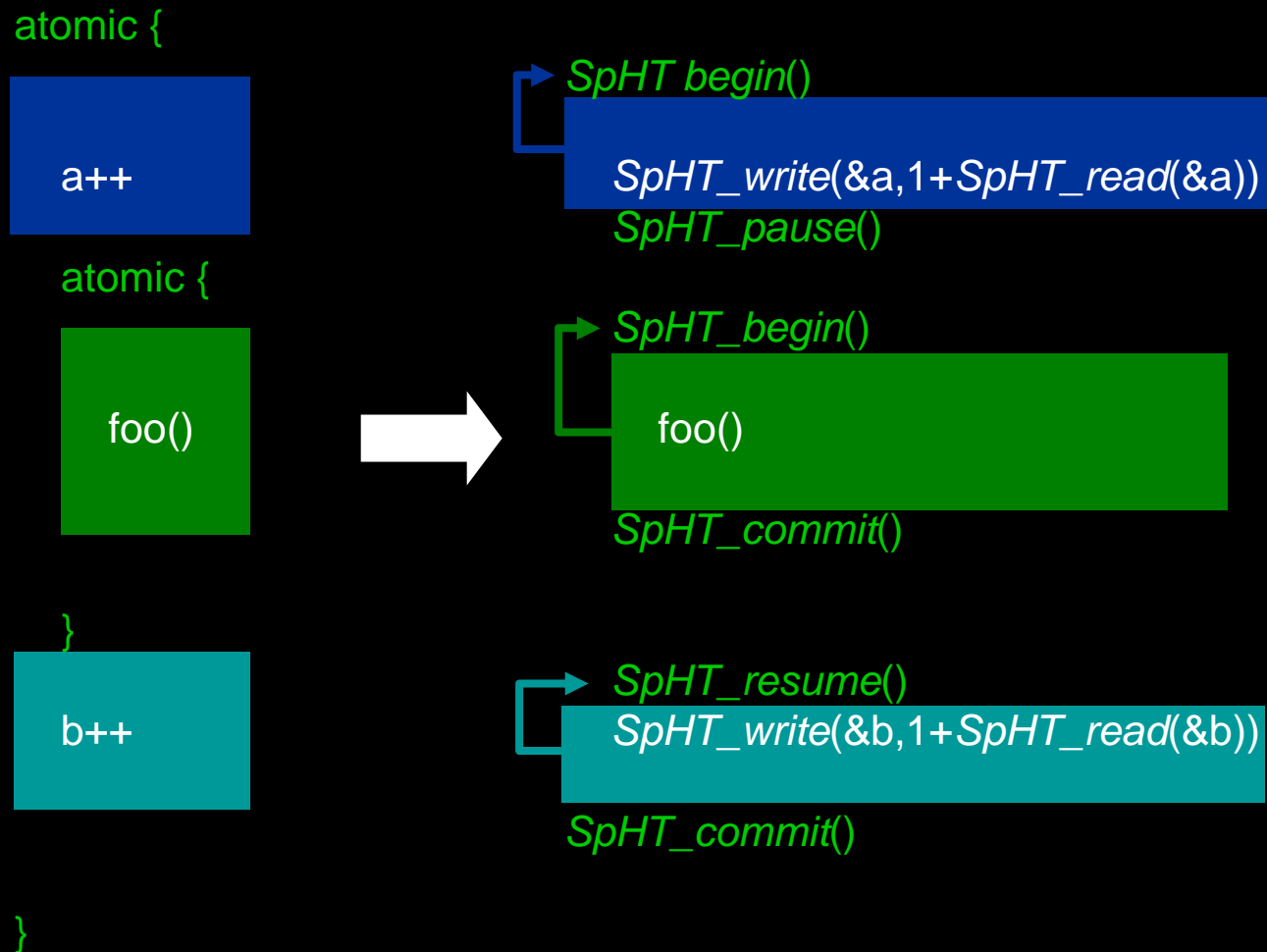
SpHT_commit()

}

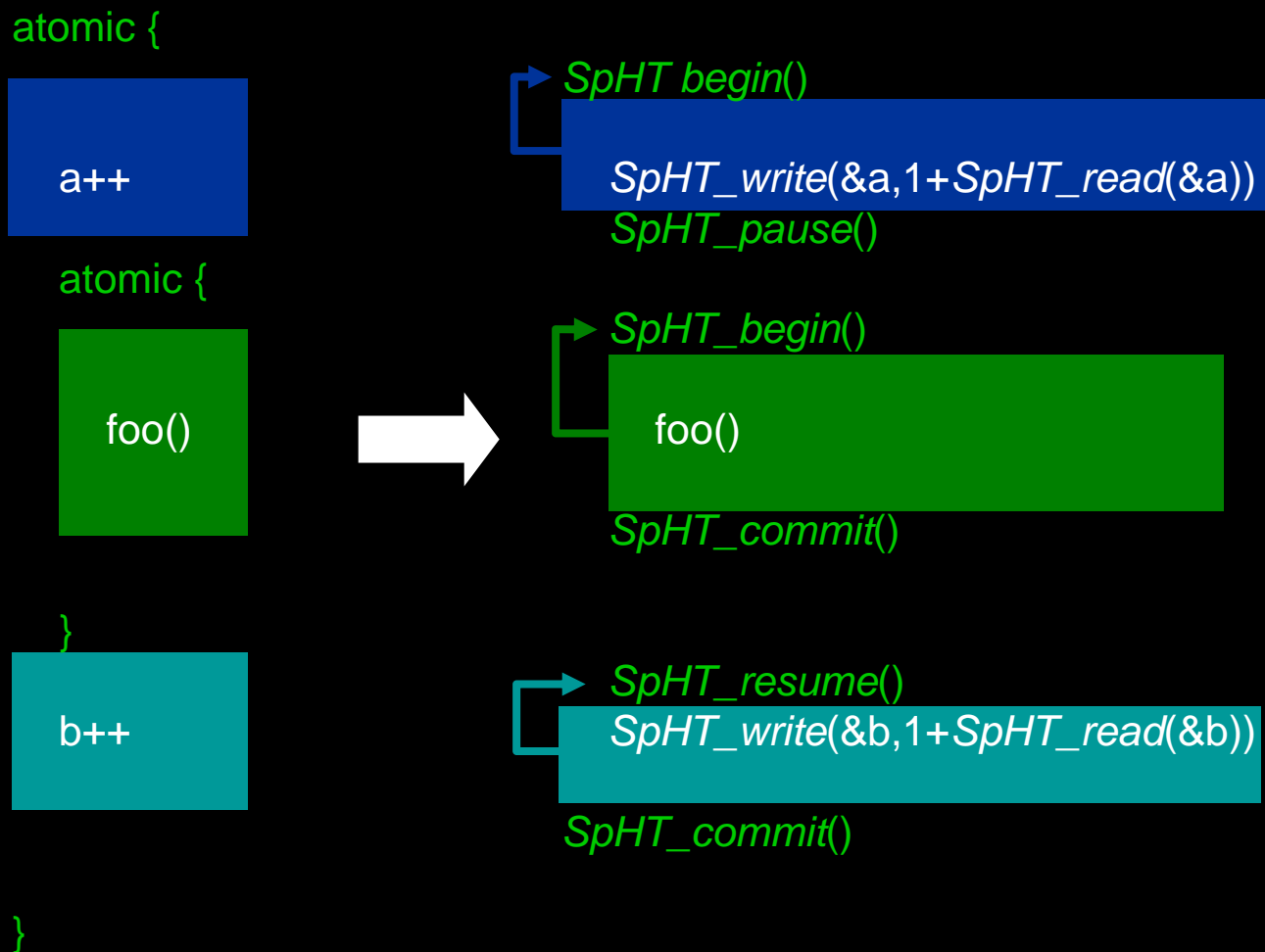
- Two ways a segment's execution may fail
 - HW transaction aborts:



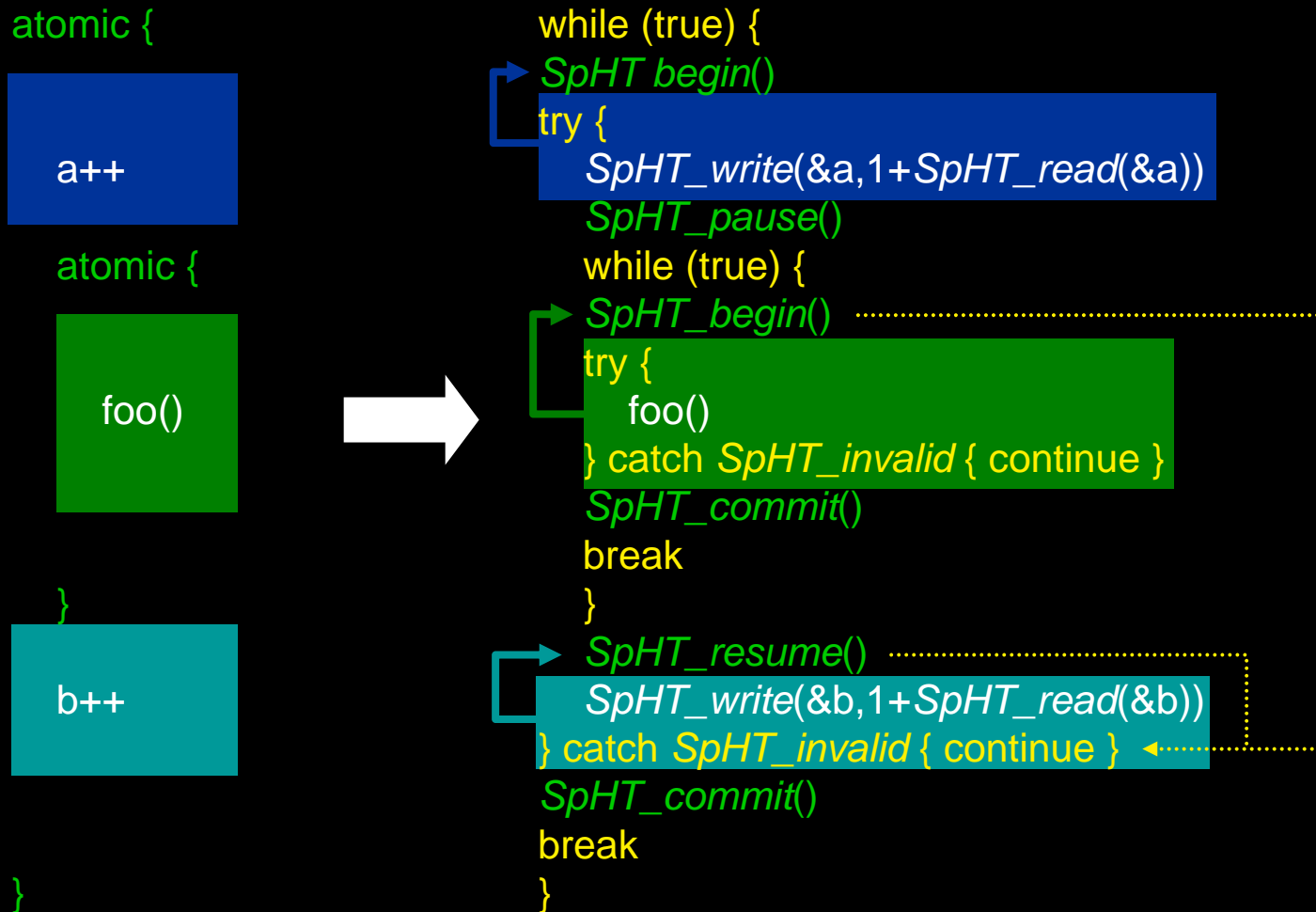
- Two ways a segment's execution may fail
 - HW transaction aborts: control returns to beginning of segment



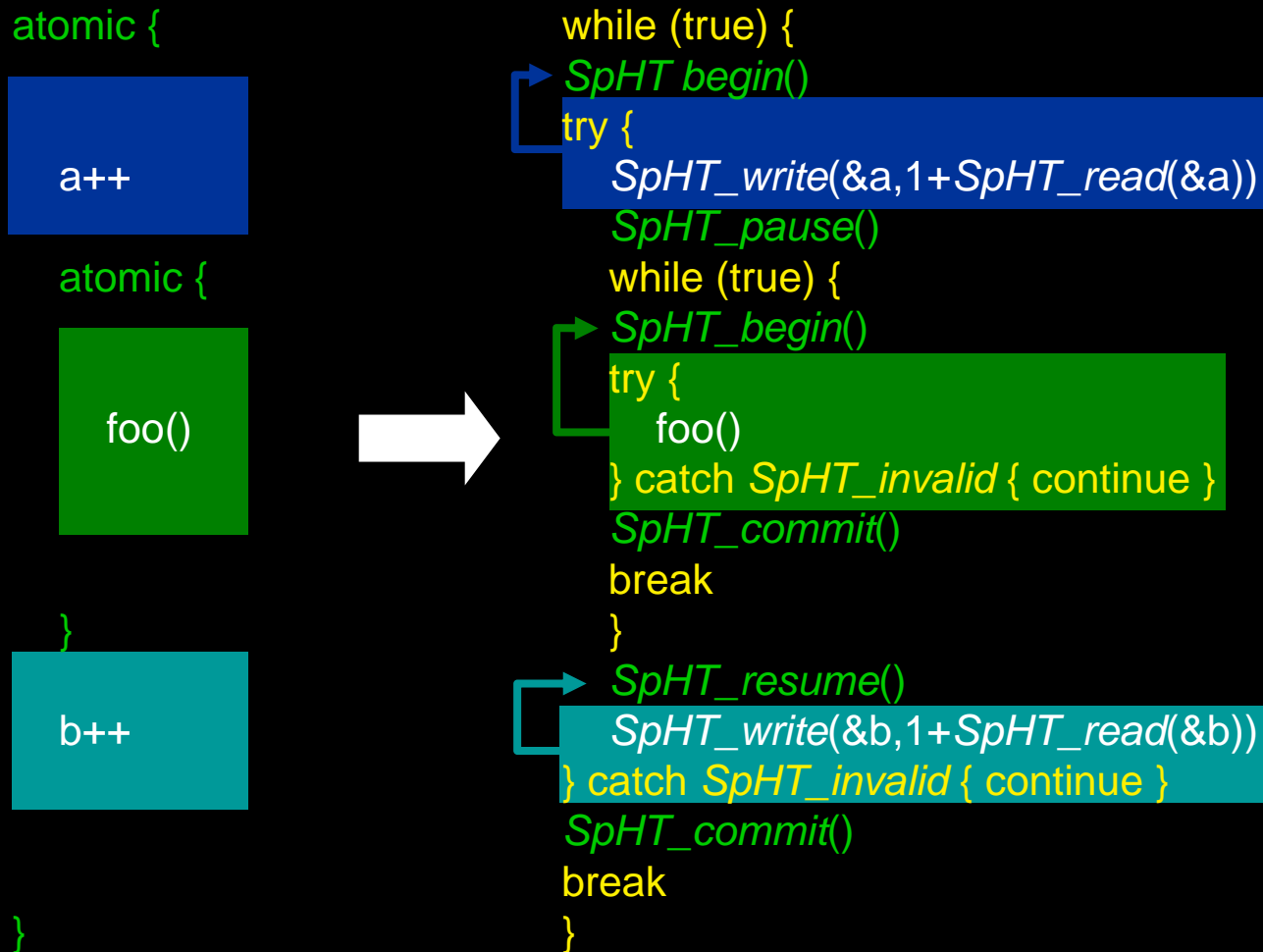
- Two ways a segment's execution may fail
 - HW transaction aborts: control returns to beginning of segment
 - Validation failure: retry the enclosing transaction



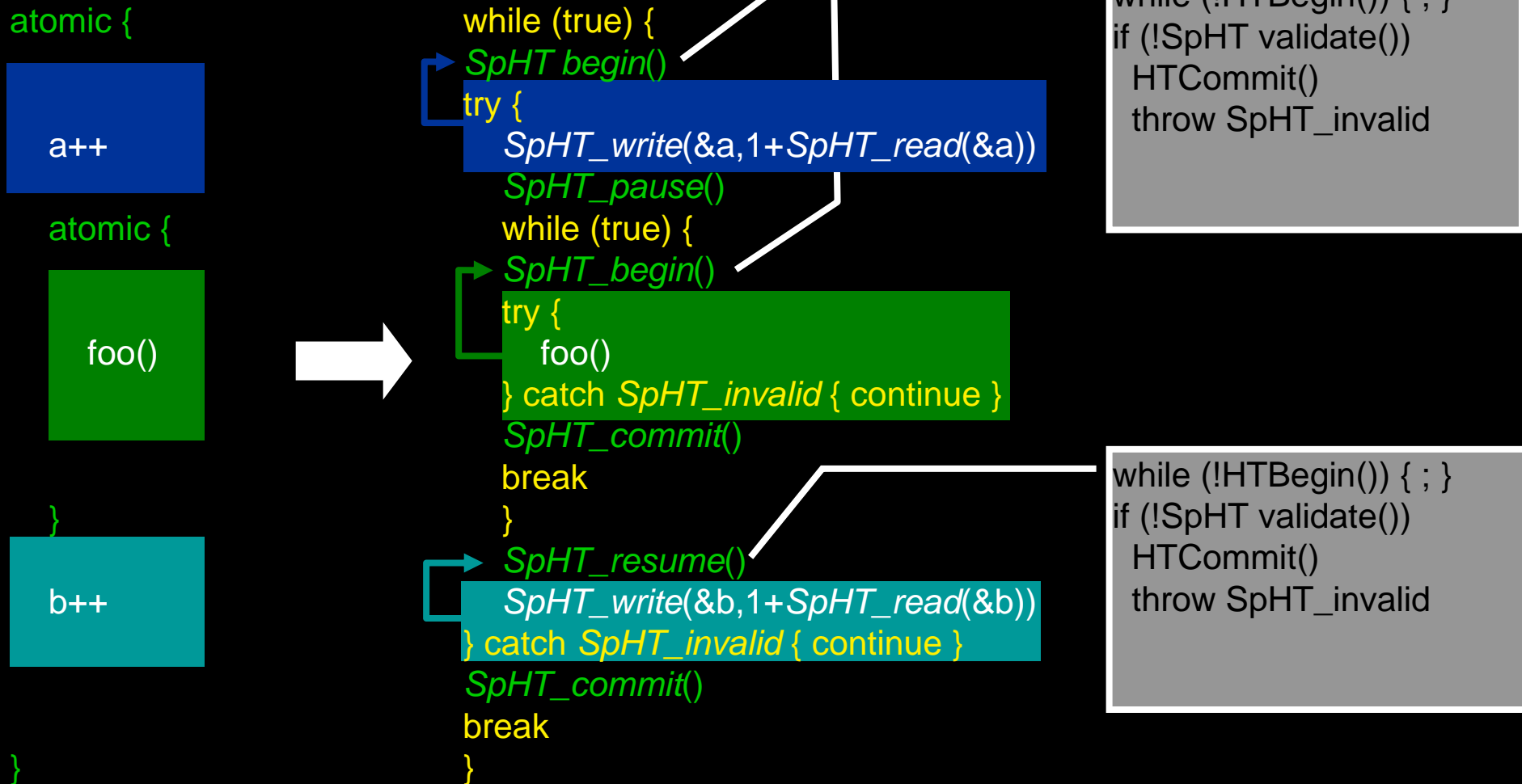
- Two ways a segment's execution may fail
 - HW transaction aborts: control returns to beginning of segment
 - Validation failure: retry the enclosing transaction
 - use exceptions for non-local control transfer



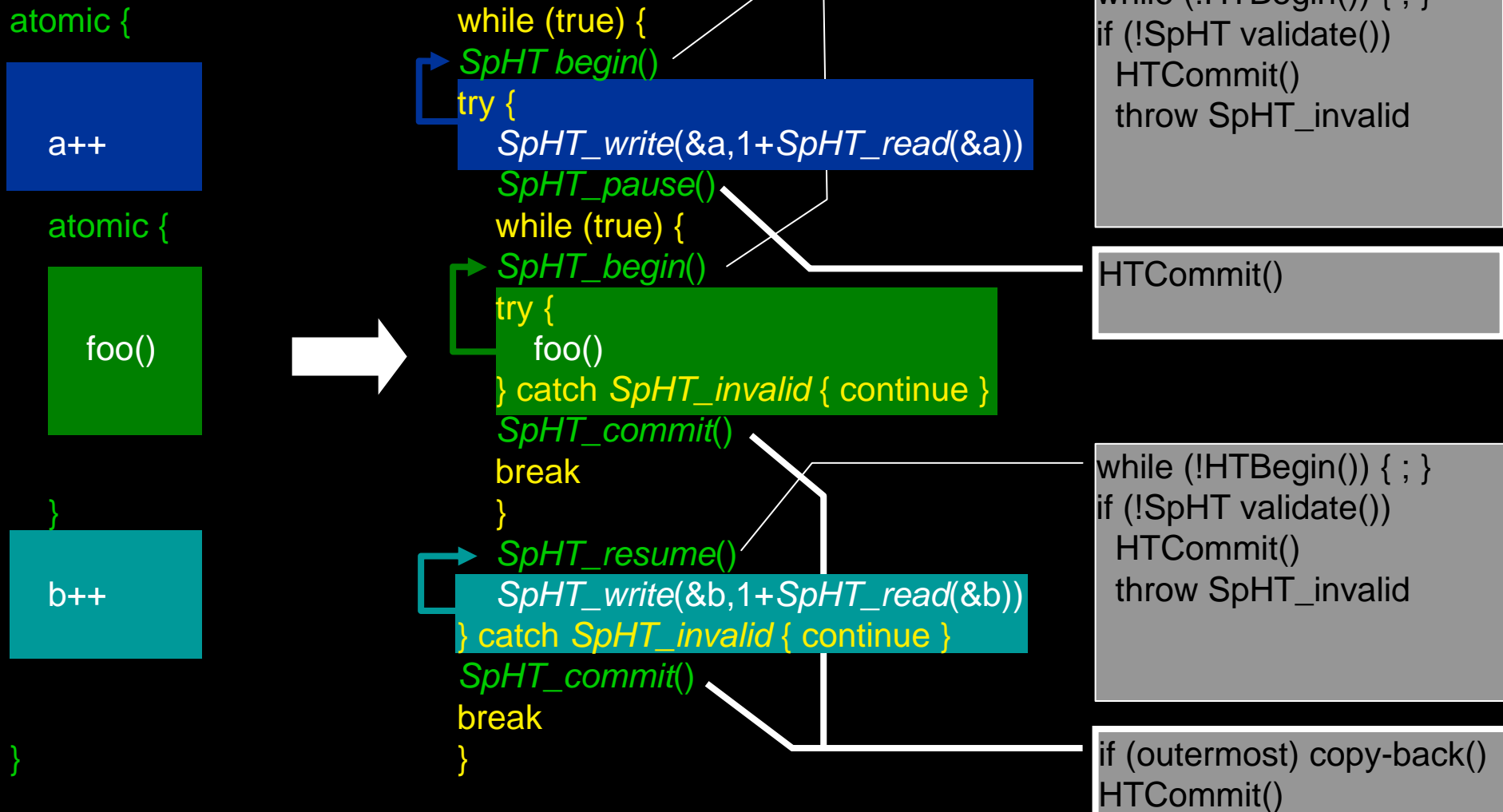
- SpHT Operations Implementation:



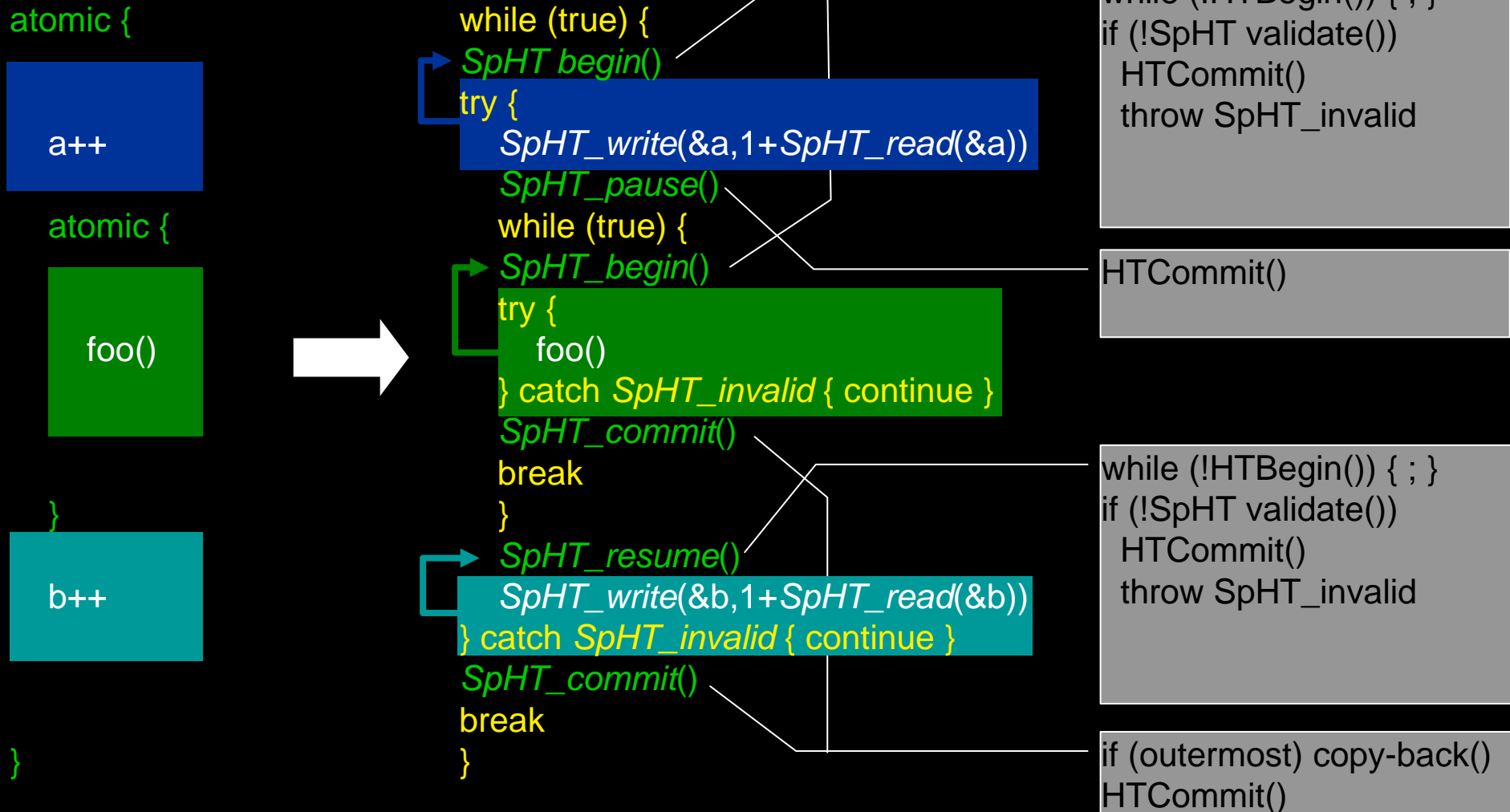
- SpHT Operations Implementation:
 - HW Interface: HTBegin, HTCommit
 - HTBegin returns true when transaction begins, false when it fails



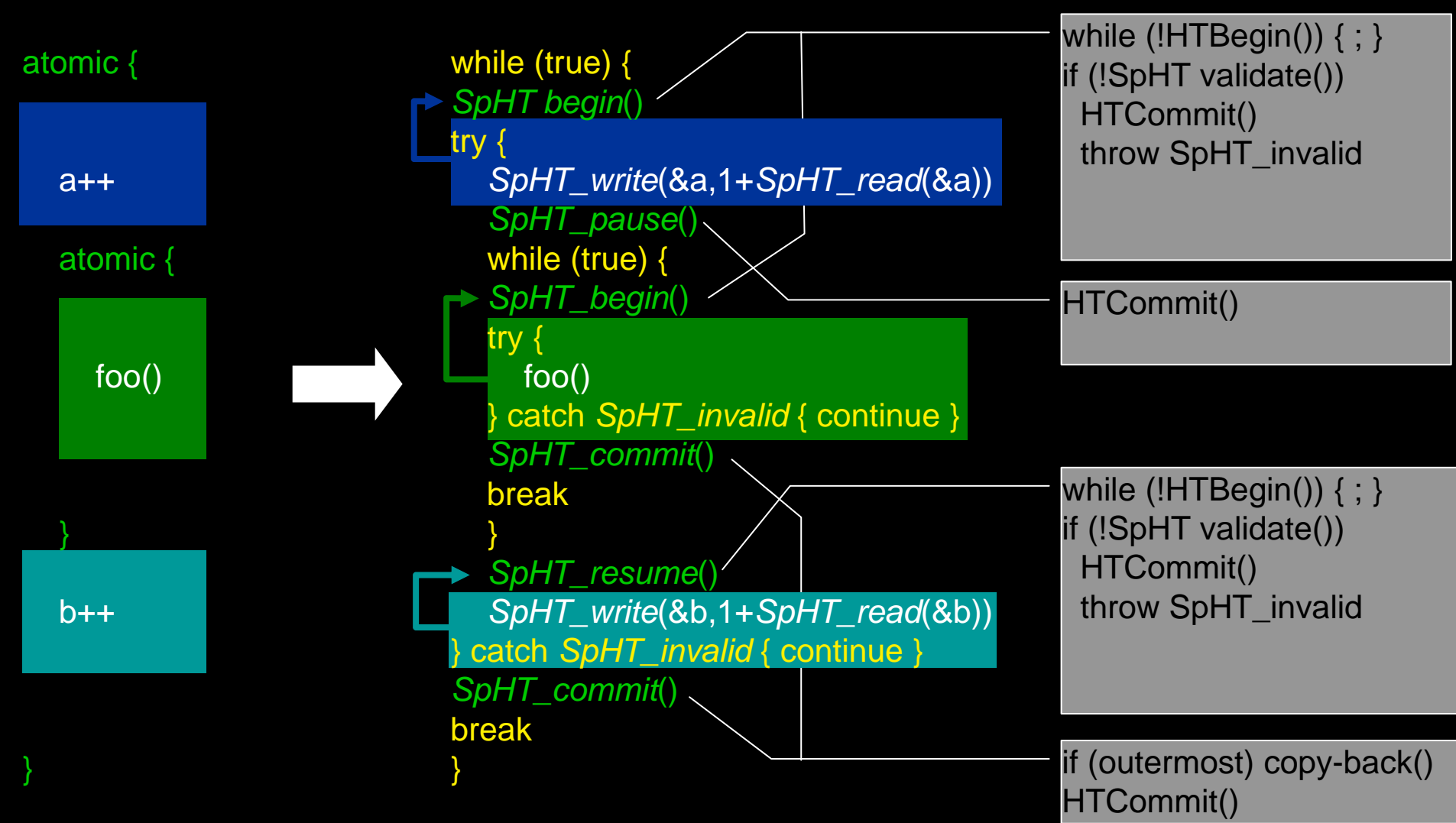
- SpHT Operations Implementation:
 - HW Interface: HTBegin, HTCommit
 - HTBegin returns true when transaction begins, false when it fails



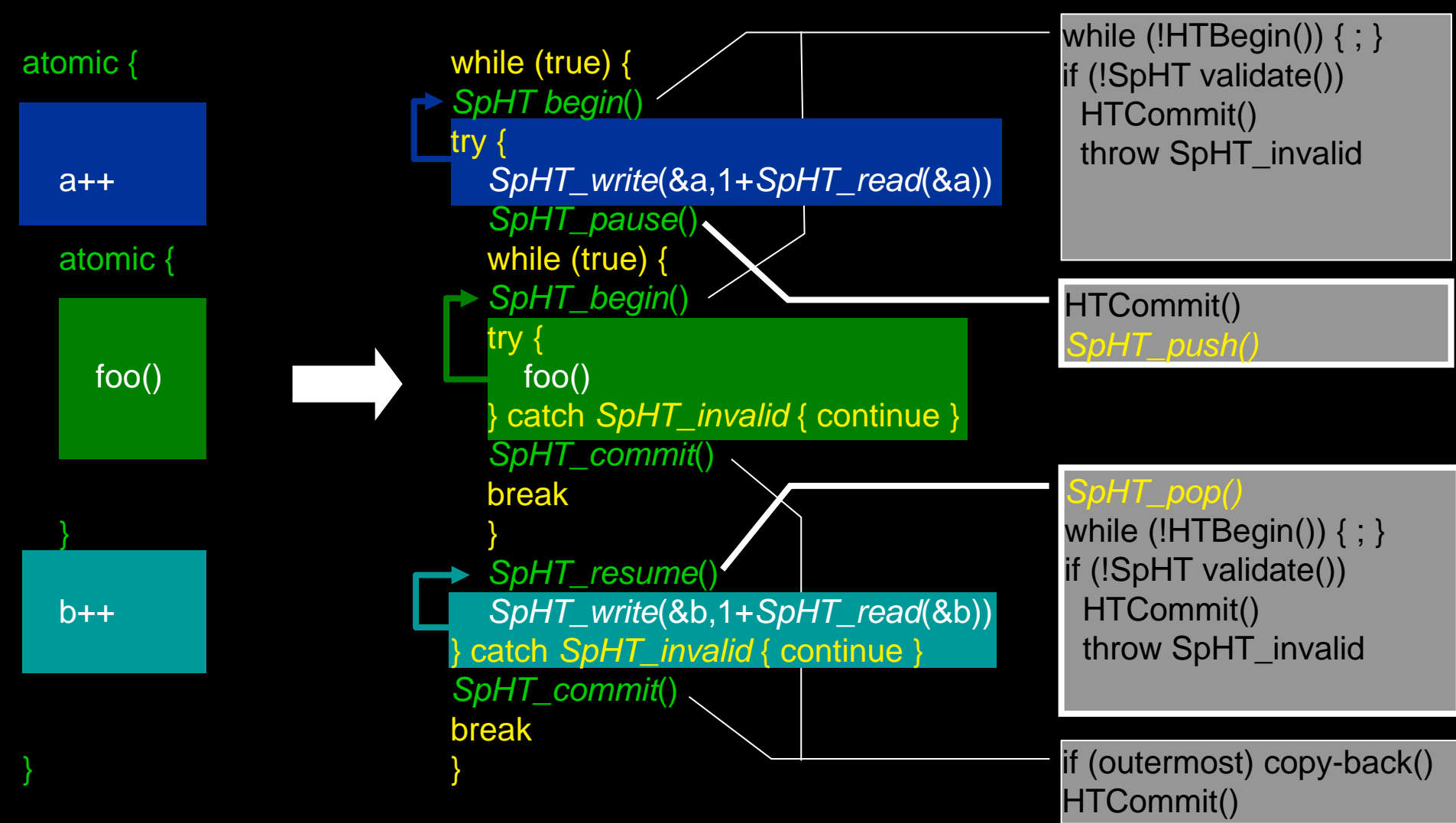
What about the state of the Read/Write Sets ?



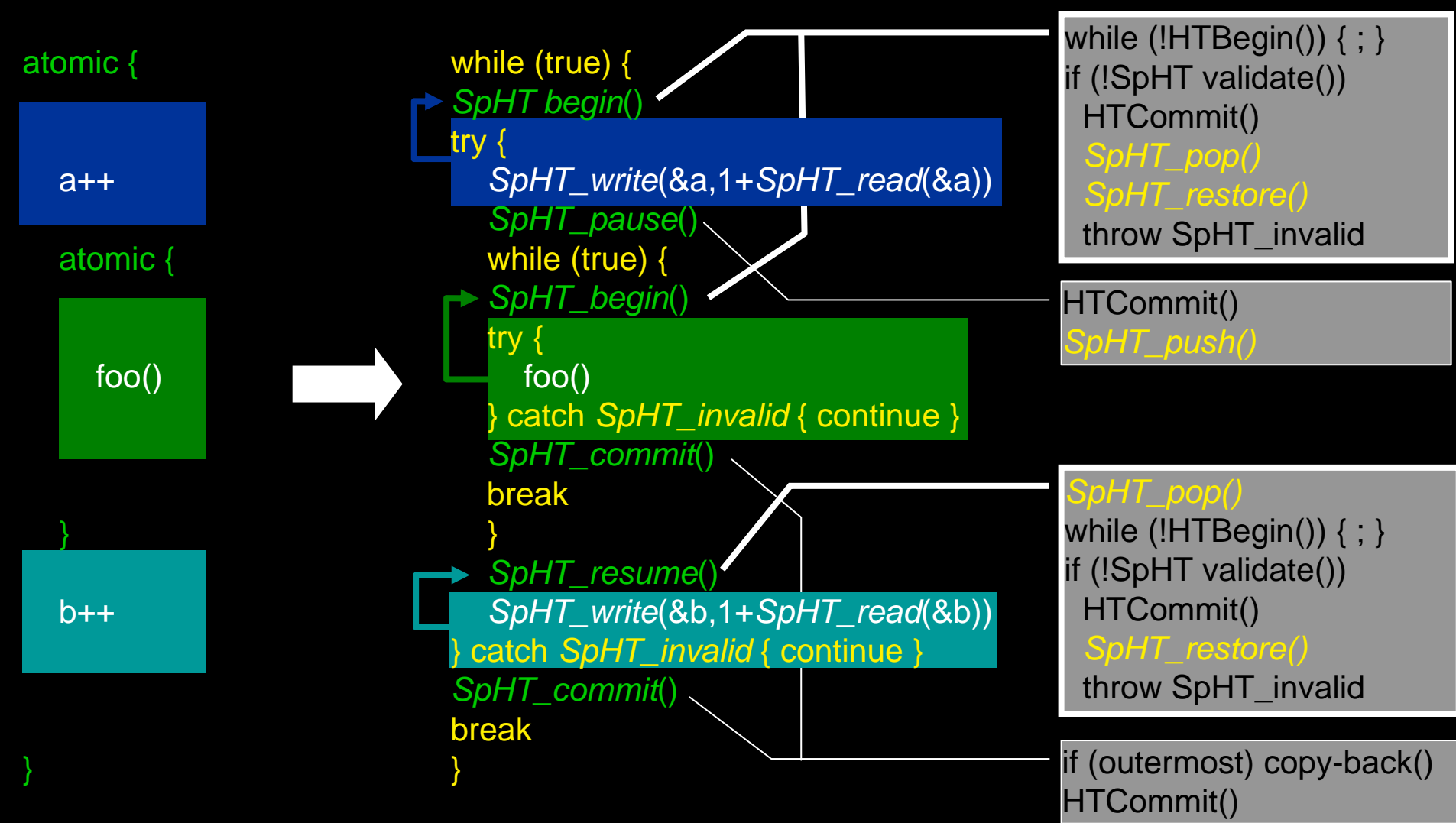
- Stack of Read/Write Set states
 - Manipulated by *SpHT_Push*, *SpHT_Pop*, and *SpHT_Restore*
- Maintain the stack invariant:
 - Top-stack state is the beginning state of innermost transaction.*



- Stack of Read/Write Set states
 - Manipulated by *SpHT_Push*, *SpHT_Pop*, and *SpHT_Restore*
- Maintain the stack invariant:
 - Top-stack state is the beginning state of innermost transaction.*



- Stack of Read/Write Set states
 - Manipulated by *SpHT_Push*, *SpHT_Pop*, and *SpHT_Restore*
- Maintain the stack invariant:
 - Top-stack state is the beginning state of innermost transaction.*



Additional Notes

- Support **nesting constructs**
orElse, user level abort, retry, etc.
- **Integrate with HyTM or PhTM**
 - Run concurrently as is with HW transactions
 - Also with SW transactions
with same overhead like HyTM

Optimizations

- **Exploit HTM features**
 - Non-transactional access during a HW transaction
 - Use reported conflict address to avoid validation
- Avoid validation on every segment
- Roll back multiple levels

One important question:

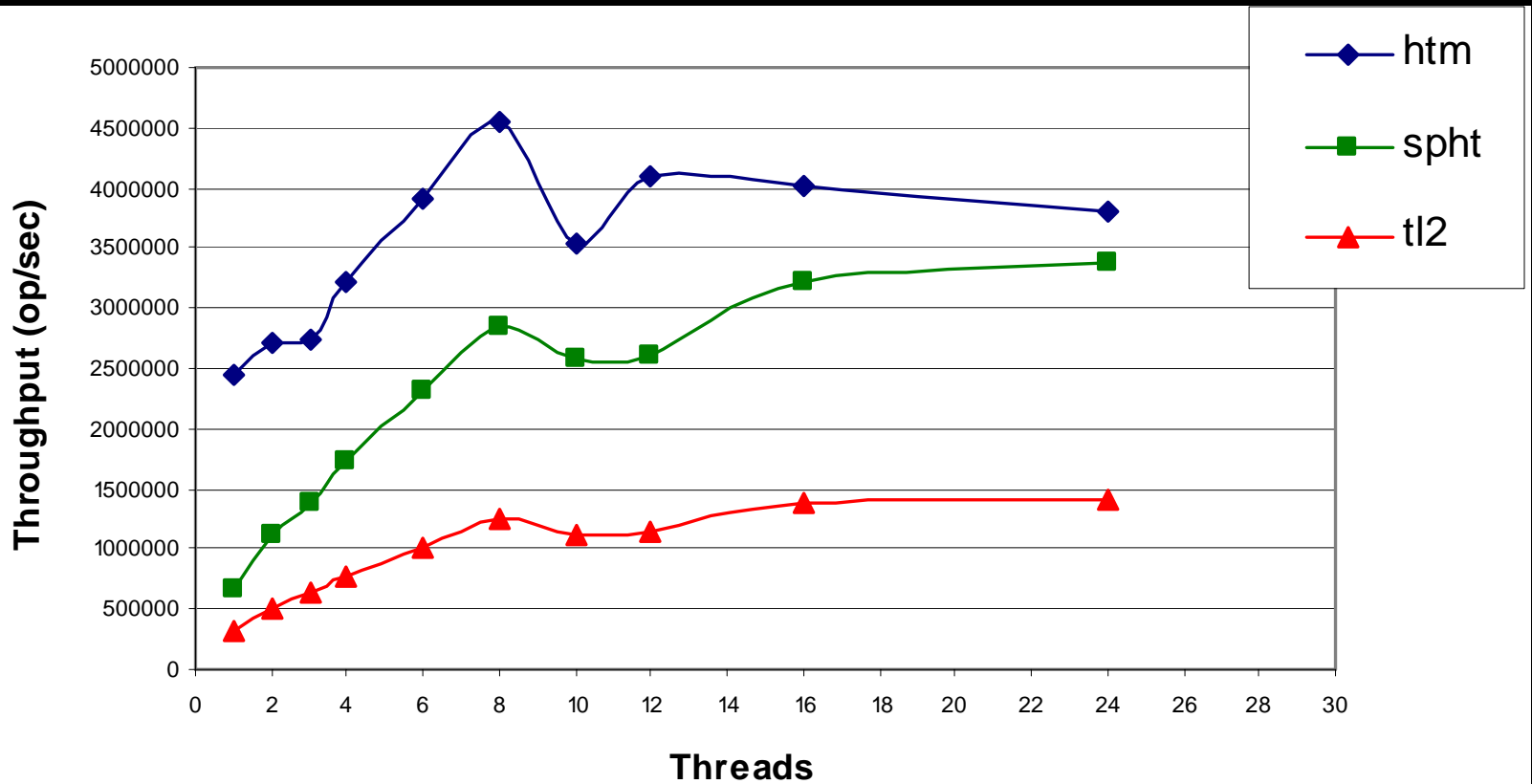
Are we faster than STM?

Evaluation

How much benefit do we get by handling conflicts in hardware?

- **SpHT Prototype** in C++
 - Local read/write sets: Arrays, fast lookup for write set
 - No stack mechanism at this point
 - Can retry current segment
 - Is not likely to change the answer
- **Simulated HTM** support: variant of LogTM
- Compared **TL2**, **SpHT**, and **pure HTM**.
- Benchmark: **RBTtree**, 10% Ins, 10% Del, 80% lookup

Results



Experiments with Nesting

- In the paper:
 - Used hand-coded contrived example
 - Compared
 - No Nesting
 - Closed Nesting
 - Open Nesting
 - Try-Atomic

Summary

- **Split Hardware Transactions**

- Hardware: handle conflicts
- Software: read/write tracking

- **Segmentation:**

One atomic operation, multiple HW Transactions

- **More flexibility with best-effort HTM**

- Closed and Open Nesting
- Debugging Support
- Long Transactions