

Boost + Software Transactional Memory

Boost Library Conference (BoostCon)
May 5, 2009



Raytheon

Customer Success Is Our Mission

Justin E. Gottschlich, Ph.D. Student

Department of Electrical and Computer Engineering
University of Colorado-Boulder

Outline

Purpose

Problem

- The Multicore Revolution
- Mutual Exclusion

Using TM

- Five Interfaces + Beyond

Under the Hood

Roadmap to boost::stm



Toward
 **boost**
STM



Collaborators



Jeremy G. Siek



Manish Vachharajani



Vicente J. Botet Escriba



Maurice Herlihy



Paul Rogers

The Multicore Revolution

And Our Free Lunch

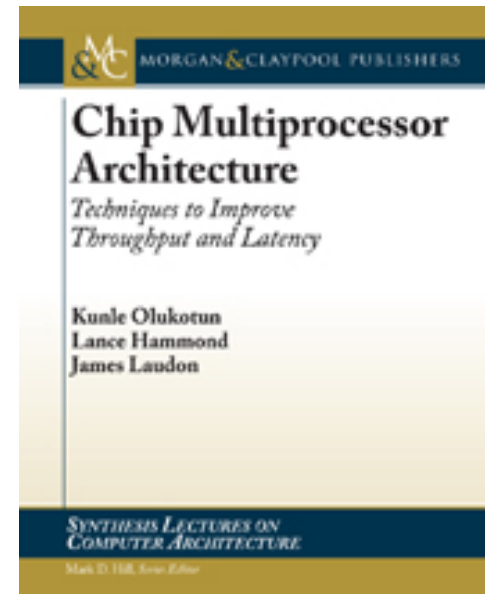


Why Multicore Computers?

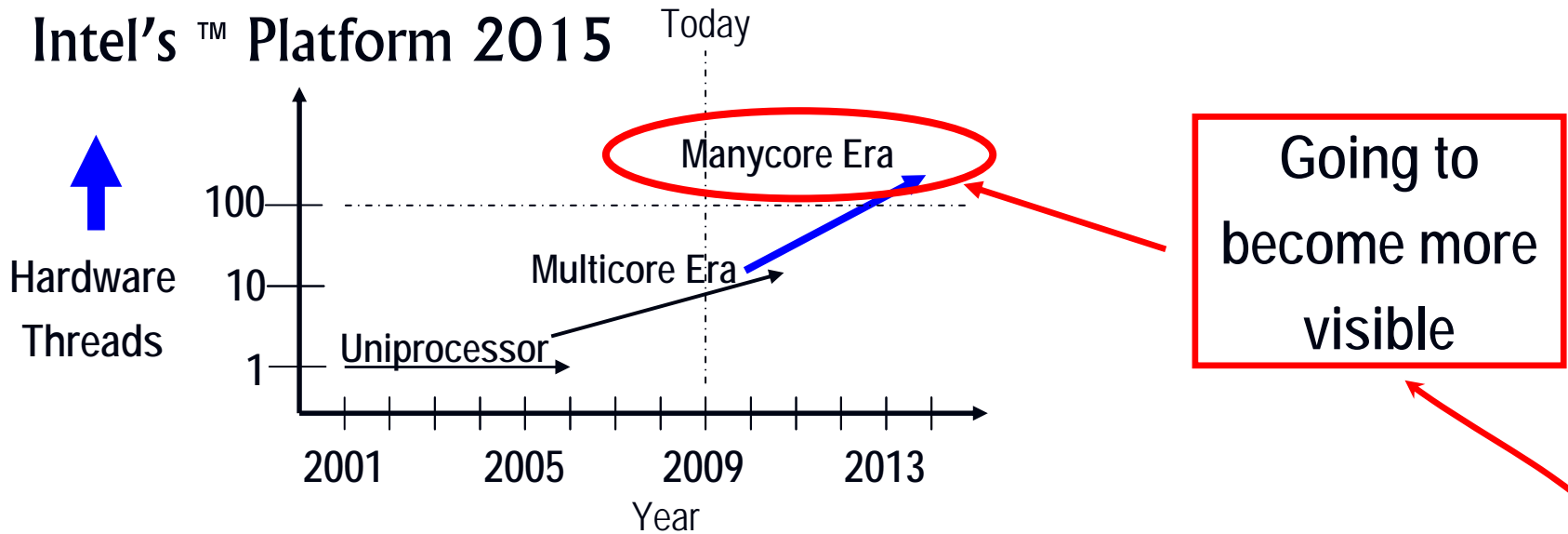
Manufacturers could not
make uniprocessors faster

Heat dissipation a problem

“Large uniprocessors are no
longer scaling in
performance ...”



“The Free (Performance) Lunch Is Over”



Disadvantages:

- No universal performance benefit like uncore
- *Only parallel software benefits*

Parallelizing Code

How do we make software parallel?

- Rewrite code w/ parallel libraries
 - `boost::thread`, `boost::mutex`, `boost::mpi`

So what is the problem?

- **Conventional parallel programming is very difficult**

Bank Account Problem

Checking account

- Global int **C**
- Controlled by lock **LC**

Savings account

- Global int **S**
- Controlled by lock **LS**

Goal:

- Move \$\$ checking to savings
- Don't bounce checks!
- ***Atomic Operation?***

Hacker's Approach: Deadlock

```
void setCands  
(int c, int s)  
{  
    lock(LC);  
    lock(LS);  
    C = c;  
    S = s;  
    unlock(LC);  
    unlock(LS);  
}
```

```
void getCands  
(int &c, int &s)  
{  
    lock(LC);  
    lock(LS);  
    c = C;  
    s = S;  
    unlock(LC);  
    unlock(LS);  
}
```

```
void getSandC  
(int &s, int &c)  
{  
    lock(LS);  
    lock(LC);  
    s = S;  
    c = C;  
    unlock(LS);  
    unlock(LC);  
}
```

can deadlock

Composing Parallel Libraries

```
void setC(int c)      void getC(int &c)    void setS(int s)    void getS(int &s)
{
  atomic(t) {
    t.w(C) = c;
  } end_atom
}
{
  atomic(t) {
    c = t.r(C);
  } end_atom
}
{
  atomic(t) {
    t.w(S) = s;
  } end_atom
}
{
  atomic(t) {
    s = t.r(S);
  } end_atom
}
```

Real TBoost.STM Code

```
void setCandS
(int c, int s)
{
  atomic(t) {
    setC(c);
    setS(s);
  } end_atom
}

void getCandS
(int &c, int &s)
{
  atomic(t) {
    getC(c);
    getS(s);
  } end_atom
}

void getSandC
(int &s, int &c)
{
  atomic(t) {
    getS(s);
    getC(c);
  } end_atom
}
```

Brief Overview of TM

Transactional Memory (TM)

- Optimistic concurrency
- Synchronization: *transaction*
- Transactions compose
- Generally retried until commit

Transaction (tx): finite set of operations

- Requires **ACI**
 - **Atomic**: all or nothing
 - **Consistent**: only legal memory states
 - **Isolated**: other txes cannot see until committed

Contention Management (CM)

- Mechanism for tx forward progress

Example Transaction and ACI

```
native_trans<int> x = 0, y = 0;
```

```
Thread 1  
atomic(t) {  
  ++t.w(x);  
  --t.w(y);  
} end_atom
```

```
Thread 2  
atomic(t) {  
  ++t.w(x);  
  --t.w(y);  
} end_atom
```

Transaction terminated

```
// CM: Thread 1's tx commits, Thread 2's tx aborts  
  
// end state, x = 1, y = -1, atomic all or nothing  
// abort if x or y change, ensure consistent  
// mid-state, x = 1, y = 0 is isolated
```

TBoost.STM

In Five Interfaces

(And Two Class Templates)



TBoost.STM's native_trans

`native_trans<>`

- Used for all native types
- Arrays ok, not pointers

Examples:

- `native_trans<bool> txBool;`
- `native_trans<float> txFloat;`
- `native_trans<int> txIntArray[100];`

TBoost.STM in Five Interfaces

`void transaction::initialize()`

- Initializes STM, always first call

`void transaction::initialize_thread()`

- Initializes thread, call per thread before using tx

`atomic(<transaction name>)`

`{ <compound> } end_atom`

- Constructs a transaction object
- Executes <compound> until successful

`template <typename T> T&`

`transaction::w(T&)`

- Perform tx write; returns ref of object to write

`template <typename T> T const&`

`transaction::r(T const&)`

- Perform tx read; returns const ref of object to read

`< > = user supplied`

Using native_trans <int >

```
native_trans<int> C = 1000;
```

```
int deposit_and_balance()  
{  
    int c = 100, bal = -1;  
    atomic (t)  
    {  
        t.w(C) += c;  
        bal = t.r(C);  
    } end_  
    return bal;  
}
```

Starting State:

- C = 1000
- bal = -1

Ending State:

- C = 1100
- bal = 1100

Isolated State Captured

Using `native_trans <int> []`

```
native_trans<int> arr[100];

void read_arr(int out[])
{
    atomic(t) {
        for (int i = 0; i < 100; ++i)
            out[i] = t.r(arr[i]);
    } end_atom
}

void write_arr()
{
    atomic(t) {
        for (int i = 0; i < 100; ++i)
            t.w(arr[i]) = i;
    } end_atom
}
```

Starting State:

- arr = 0, 0, 0, ...

Ending State:

- arr = 0, 1, 2, ...

Note: `read_arr()`

- Never sees:
 - 0, 1, 2, 0, 0, ...
- Only: 0, 0, 0, ...
- Or: 0, 1, 2, 3, ...

TBoost.STM's transaction_object

transaction_object

- Used for all user-defined types
- Must be default and copy constructable
- Must have callable operator=
- Uses curiously recurring template

Building transaction_object (char*)

```
class str_trans : public transaction_object <str_trans>
{
    char *b_;
public:
    str_trans() : b_(0) {}
    str_trans(str_trans const &rhs) : b_(0) { cp(rhs.b_); }
    str_trans& operator=(str_trans const &rhs)
    { cp(rhs.b_); return *this; }
    void cp(char const *buf)
    {
        delete [] b_; b_ = 0;
        if (0 == buf) return;
        b_ = new char [strlen(buf)+1]; strcpy(b_, buf);
    }
    char const * buf() const { return b_ ? b_ : ""; }
    ~str_trans() { delete [] b_; }
};
```

Building transaction_object (char*)

```
using namespace std;
str_trans str;

void read_str(string &s)
{
    atomic(t) {
        s = t.r(str).buf();
    } end_atom
}

void write_str(string const &s)
{
    atomic(t) {
        t.w(str).cp(s.c_str());
    } end_atom
}
```

TBoost.STM

Beyond Five Interfaces



How Do We Perform I/O (or Any Irreversible Operation)?

`transaction::make_irrevocable()`

- If fails, throws exception
- Otherwise, *guaranteed* to commit

```
void hello_world() {
    atomic(t) {
        write_str((string)
            "hello concurrent world");
        string s; read_str(s);
        t.make_irrevocable();
        cout << s.c_str();
    } end_atom
}
```

Building lists

```
template <typename T> class list_node :  
    public transaction_object< list_node<T> >  
{  
    list_node *next_;  
    T value_;  
};
```

```
template <typename T> class linked_list  
{  
public:  
    bool lookup(T const &val) const; //assume coded  
    bool insert(list_node<T> const &rhs); // new  
    bool remove(list_node<T> const &rhs); // del  
private:  
    list_node<T> head_;  
};
```

new and delete

Do not use `new` and `delete` for txes!

To create memory

- `T* new_memory()`
- `T* new_memory_copy(T const &rhs)`

To destroy memory

- `delete_memory(T &rhs)`

Building list::insert

```
bool insert(list_node<T> const &rhs) {
    atomic(t) {
        list_node<T> const *headP = &t.r(head_);
        if (NULL != headP->next()) {
            // find location to insert
            list_node<T> *newN = t.new_memory_copy(rhs);
            // set prev, new and next
        } else {
            list_node<T> *newN = t.new_memory_copy(rhs);
            // set head
        }
    } end_atom
    return true;
}
```

Building list::remove

```
bool remove(list_node<T> const &rhs) {  
    atomic(t) {  
        // find the item to remove  
        if (cur->value() == rhs.value()) {  
            // reset pointers  
            t.delete_memory(*cur);  
            t.end(); return true;  
        }  
    } end_atom  
    return false;  
}
```

what does this do?

before_retry

```
atomic(<tx>) { <compound> }  
before_retry { <compound> }
```

- before_retry executes before tx is retried
- useful for cleanup
- changing tx priority, CM strategy, etc.

Building algo::move

```
template <typename C> bool algo::move
(C &l1, C &l2, C::node_type const &v)
{
    bool r = false;
    atomic(t) {
        if (l1.lookup(v) && !l2.lookup(v))
        {
            l1.remove(v); l2.insert(v); r = true;
        }
    } before_retry { r = false; }
    return r;
}
```

References and Smart Pointers

Generally, don't do this:

- `native_trans<int> &x = t.w(X);`
 - a bad idea
- `native_trans<int> &x = t.r(X);`
 - an even worse idea!

Instead, use:

- `boost::stm::read_ptr(transaction &t, T)`
- `boost::stm::write_ptr(transaction &t, T)`

Problems With Holding References

```
native_trans<int> txInt = 0;
```

```
void test_parent() {  
    atomic(t) {  
        native_trans<int> &tx = t.r(txInt);  
        if (0 == tx) test_nested();  
        cout << tx << endl;  
    } end_atom  
}
```

```
void test_nested() {  
    atomic(t) { ++t.w(txInt); } end_atom  
}
```

Use read_ptr

```
native_trans<int> txInt = 0;

void test_parent() {
    atomic(t) {
        read_ptr< native_trans<int> >
            tx(t, txInt);
        if (0 == *tx) test_nested();
        cout << *tx << endl;
    } end_atom
}

void test_nested() {
    atomic(t) { ++t.w(txInt); } end_atom
}
```

Use write_ptr

```
native_trans<int> txInt = 0;

void test_parent() {
    atomic(t) {
        write_ptr< native_trans<int> >
            tx(t, txInt);
        if (0 == *tx) test_nested();
        cout << ++*tx << endl;
    } end_atom
}

void test_nested() {
    atomic(t) { ++t.w(txInt); } end_atom
}
```



Under the Hood: Fast Synchronization



Optimistic Concurrency

Pessimistic concurrency (locks)

- Critical section limited to single thread

Optimistic concurrency (transactions)

- Unlimited threads can execute critical section
- Truly optimistic concurrency
 - Simultaneously *conflicting* critical section execution
 - *Not supported by all TMs*

Important aspects

- Consistency checking
- Updating policies

Consistency Checking

Consistency checking

- Process to verify state of transaction is consistent
- **Inconsistency:** $C_w \cap (I_w \cup I_R) \neq \emptyset$

Validation

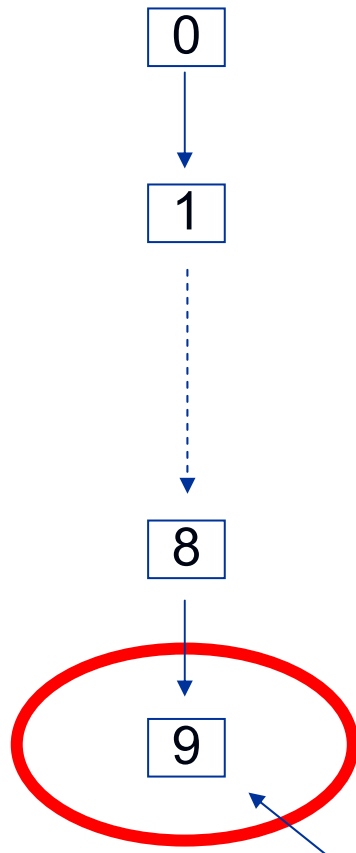
- Each memory location has version #
- When committing
 - tx verifies version # is same
 - increments version for writes

Invalidation

- No version #s used
- When committing
 - abort txes reading/writing mem its writing

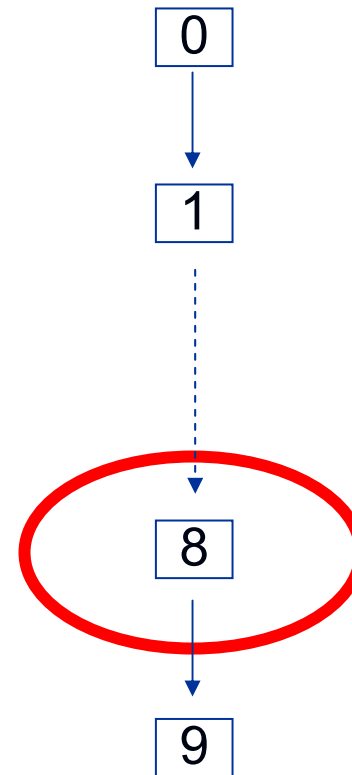
Linked List (Pessimistic)

Step 1: Insert element (9)



new node (9)

Step 2: Lookup element (8)



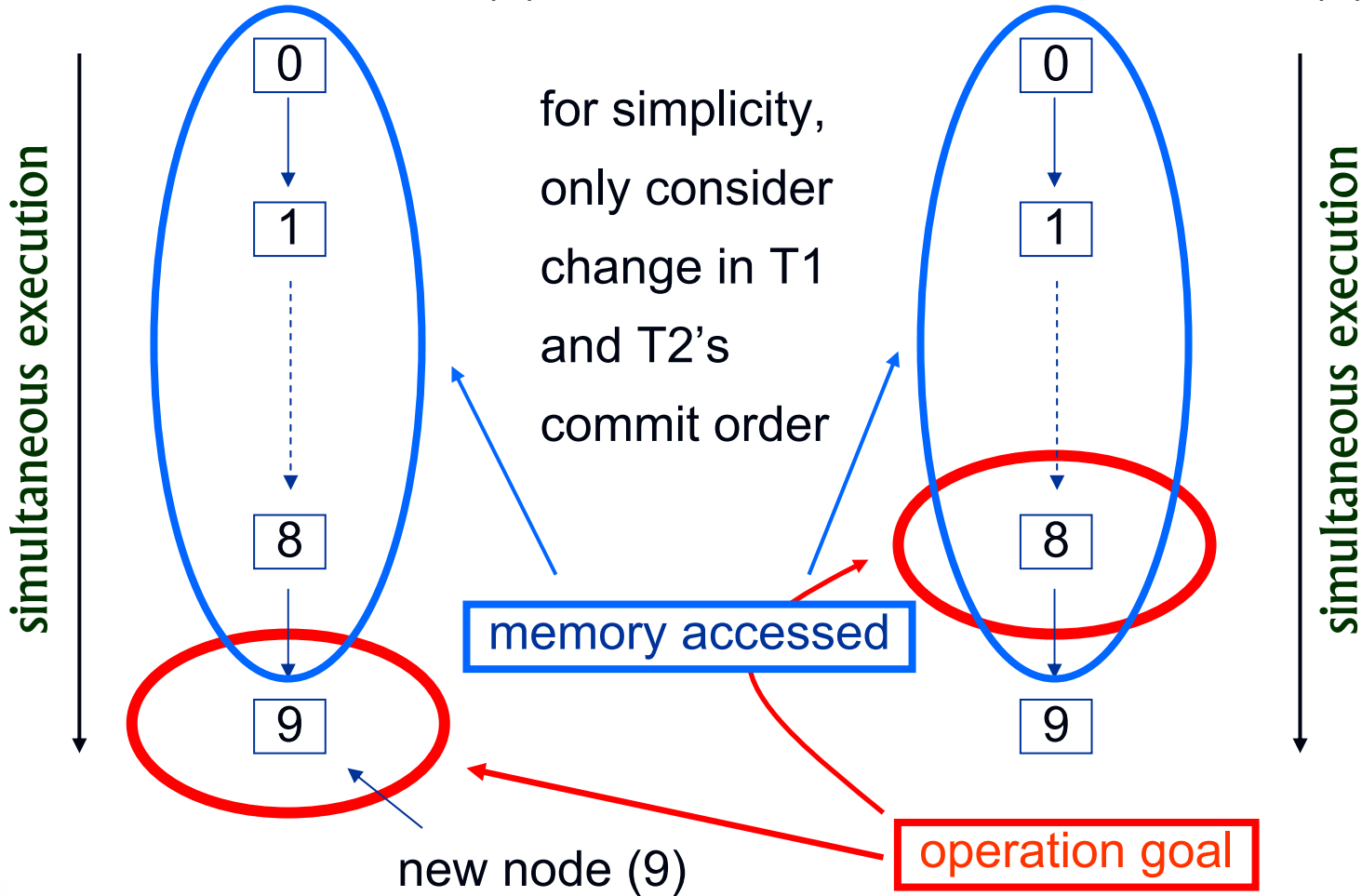
operation goal

serial execution

Linked List (Optimistic)

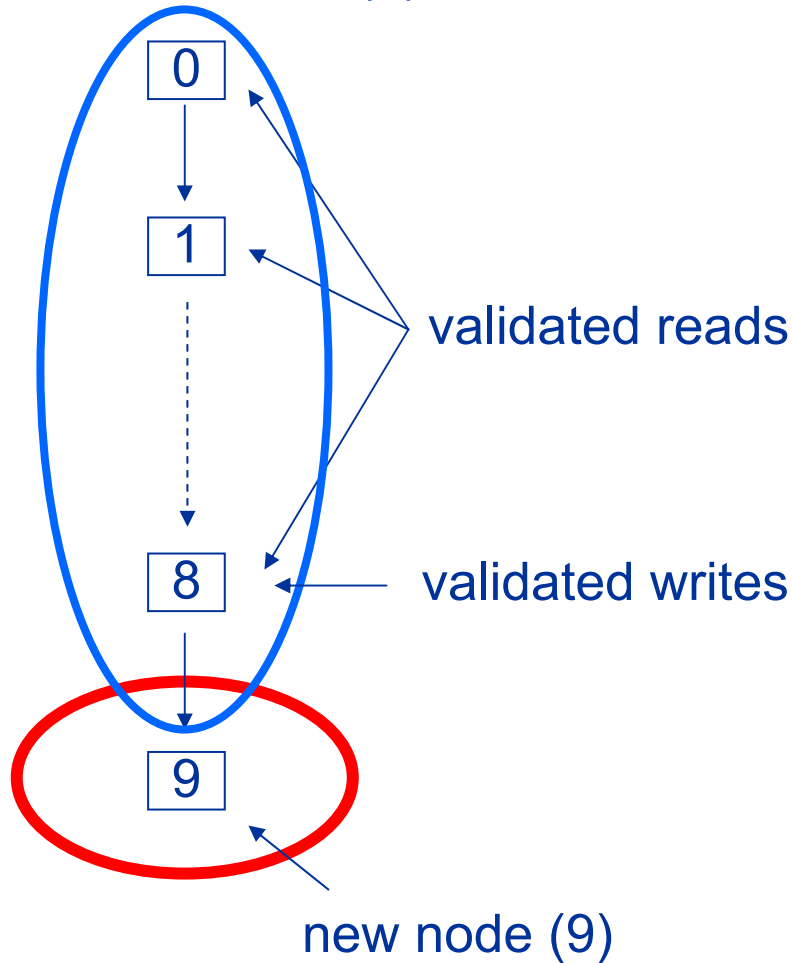
T1: Insert element (9)

T2: Lookup element (8)

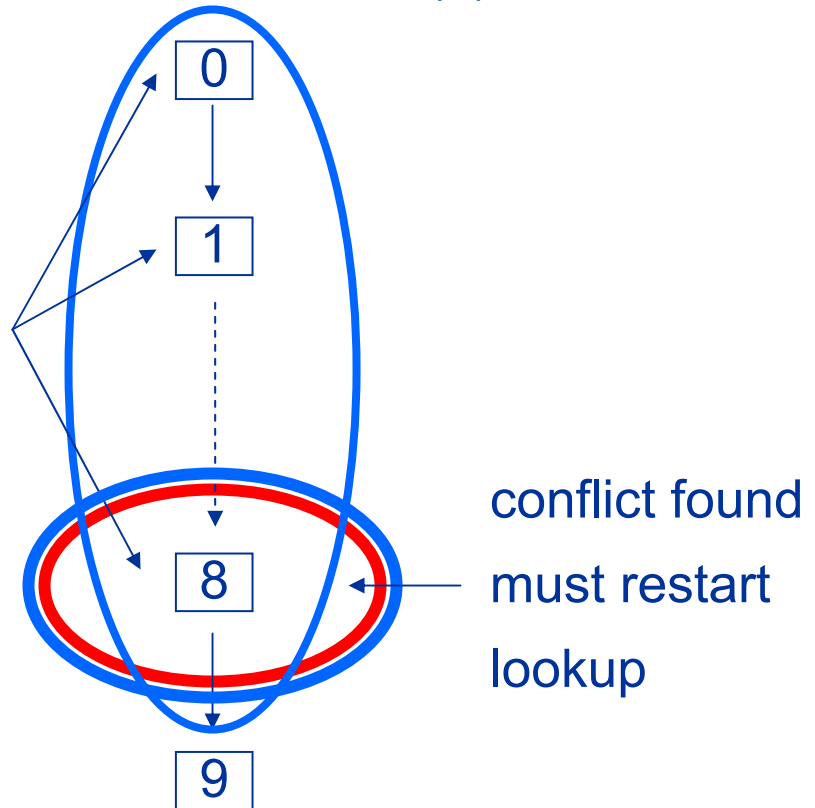


Validation – Insert, Lookup

Insert element (9)



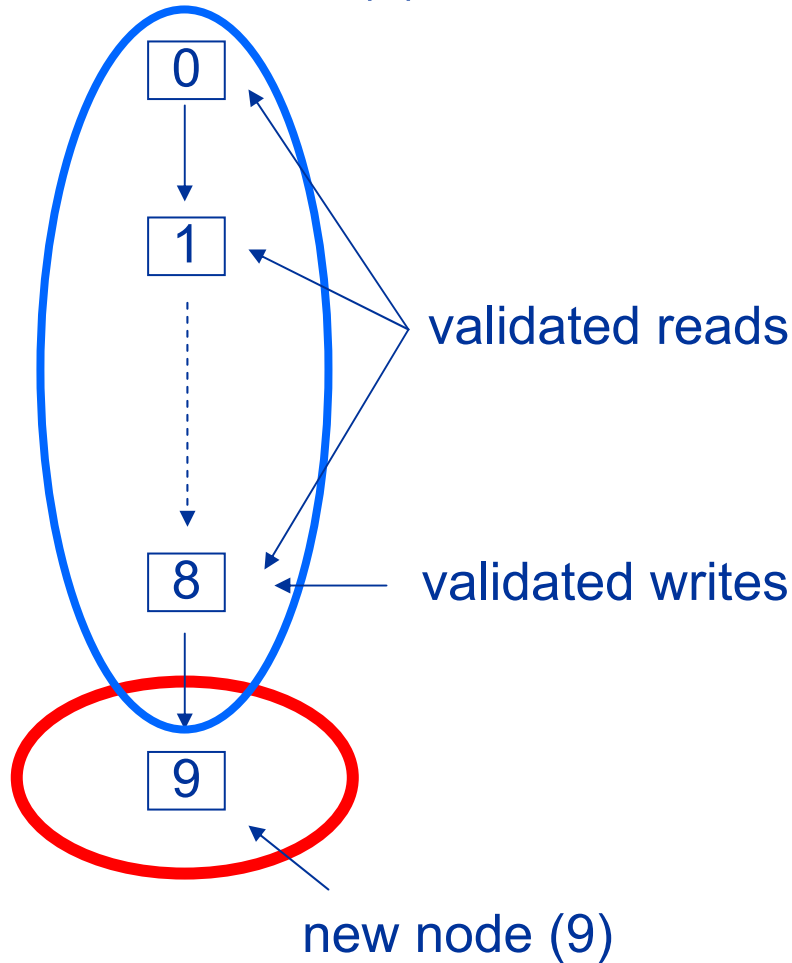
Lookup element (8)



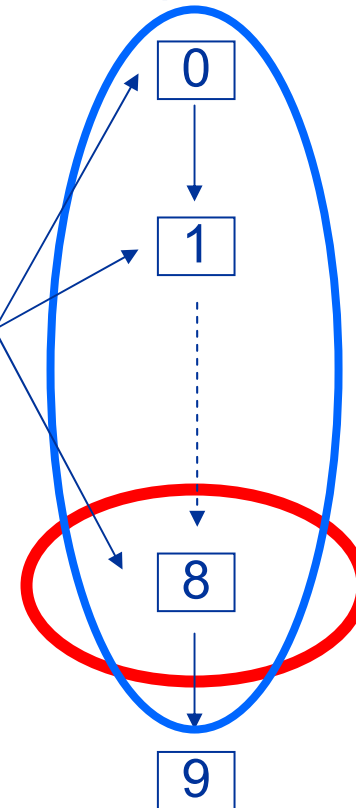
total validated ops:
26 reads, 1 write

Validation – Lookup, Insert

Insert element (9)



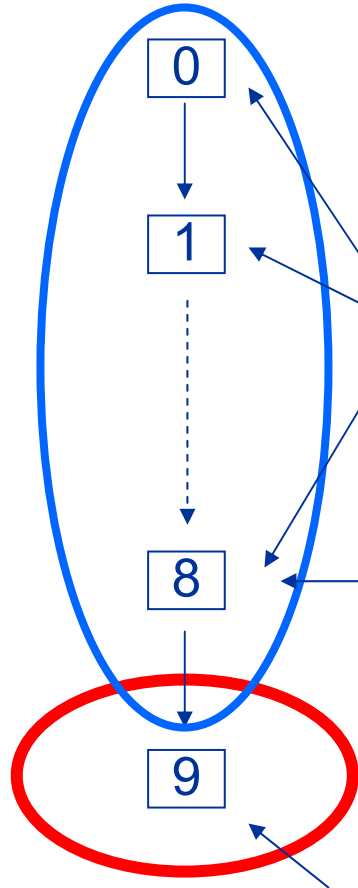
Lookup element (8)



total validated ops:
17 reads, 1 write

Invalidating Linked List

Insert element (9)

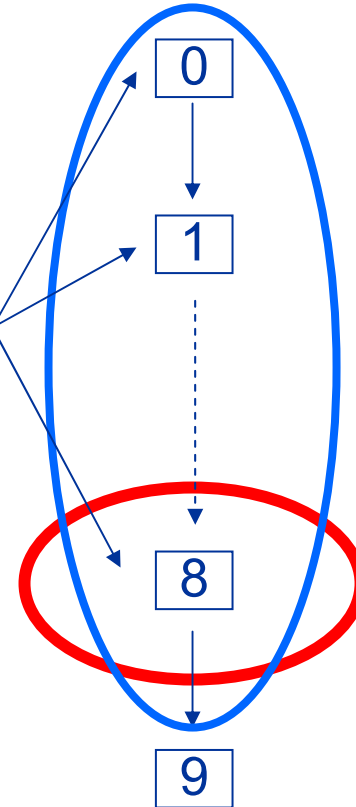


~~validated reads~~

validated writes

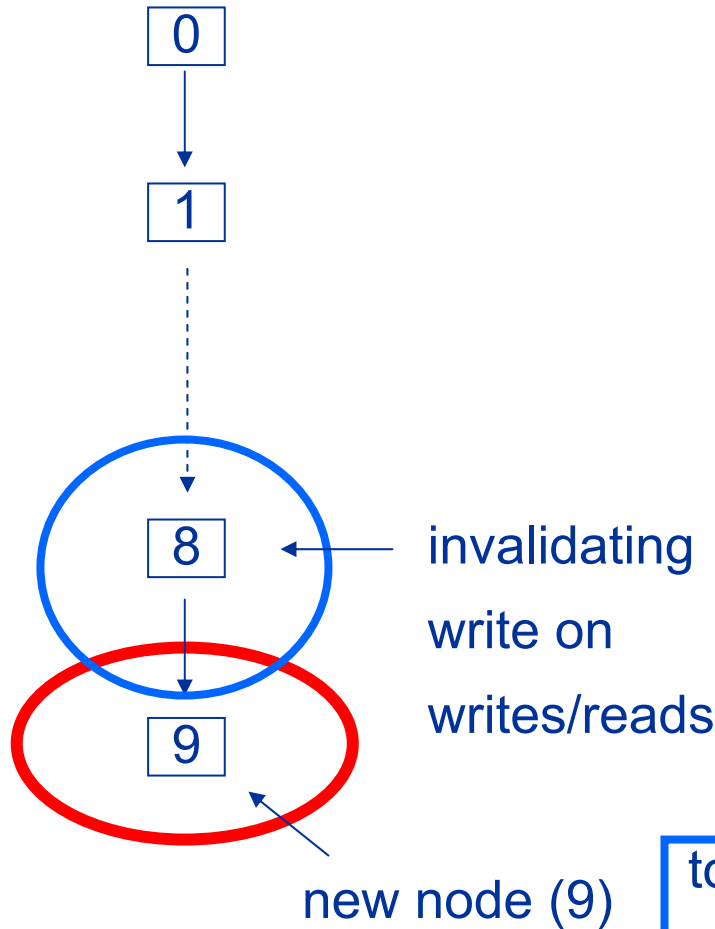
new node (9)

Lookup element (8)

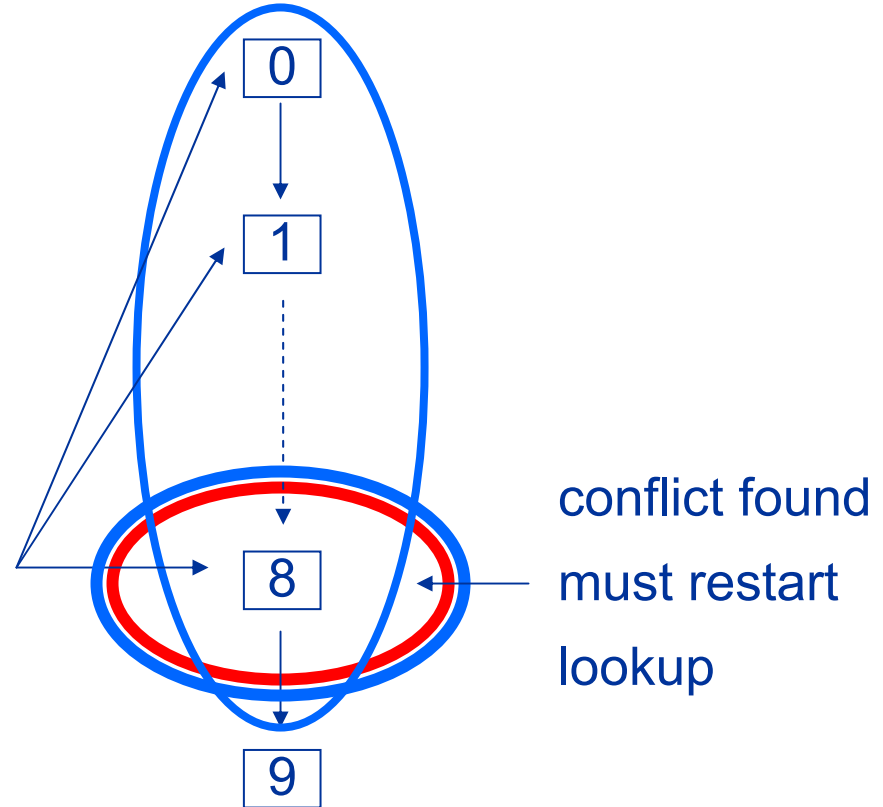


Invalidation – Insert, Lookup

Insert element (9)



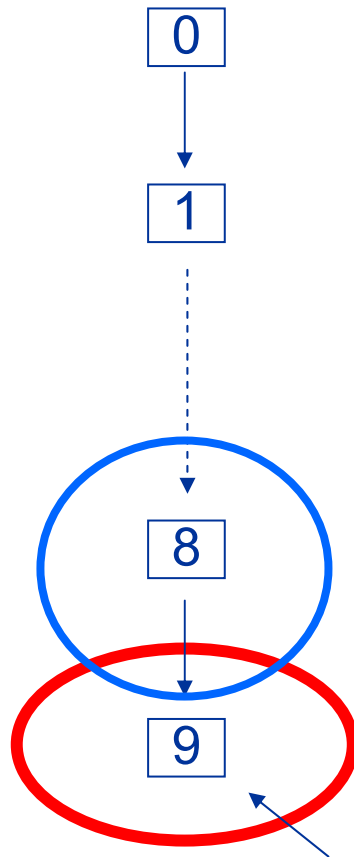
Lookup element (8)



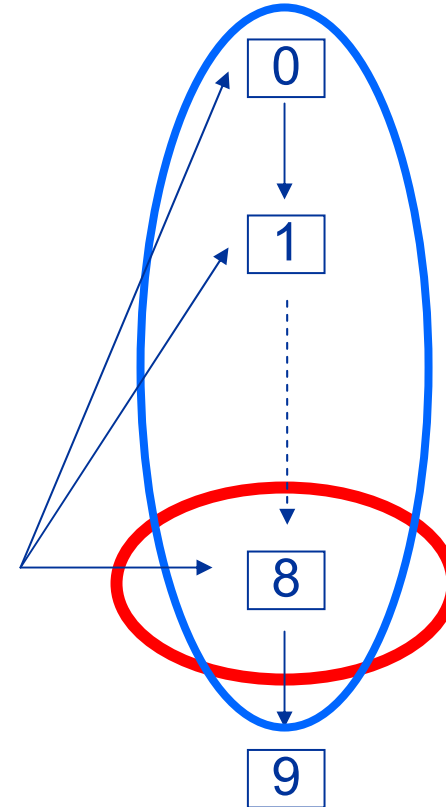
total invalidated ops:
9 reads unopti, 3 reads opti

Invalidation – Lookup, Insert

Insert element (9)



Lookup element (8)



all reads, no
invalidation
needed

new node (9)

total invalidated ops: 0

Consistency Checking

Validation

- Great for short txes with many threads

Invalidation

- Great for long txes with fewer threads
- Great for read-dominated workloads
 - Mathematically optimal for read-only txes

TBoost.STM supports both

Updating Policies

Direct Updating

```
atomic(t) { t.w(C) = c; } end_atom }
```

1) copy mem

2) modify

3.a) if commit,
do nothing

original

3.b) if **abort**, restore original

Deferred Updating

```
atomic(t) { t.w(C) = c; } end_atom }
```

1) copy mem

original

2) modify

3.a) if abort,
do nothing

3.b) if **commit**, copy to global

Updating Policies

Direct Updating

- Advantages

fast commit

early writer conflict

detectio

TBoost.STM supports both

- Disadvantages

single writer per mem

speculative contention

management

Deferred Updating

- Advantages

fast abort

truly optimistic

time

commit detection

(no speculation)

- Disadvantages

slow commit

Putting It All Together

```
void setCandS(){  
    atomic(t) {  
        setC(-100);  
        setS(100);  
    } end_atom  
}
```

thread1

Truly optimistic
concurrency

```
void getSandC  
(int &s, int &c){  
    atomic(t) {  
        getS(s);  
        getC(c);  
    } end_atom  
}
```

thread2

C = 0, S = 100

thread1

setC(-100)

setS(100)

C1()

time

thread2

0 = getS()

100 = getC()

C2()

C = 100, S = 0

C = 100, S = 0



Consistency and Updating in TBoost.STM

Consistency

- `#define PERFORM_VALIDATION 1`
- `#define PERFORM_VALIDATION 0`

Updating

- `transaction::do_deferred Updating()`
- `transaction::do_direct Updating()`

More Optimizations

TBoost.STM v.0.2

- Competitive with TL2 (state-of-the-art STM)
- + **10x faster** than DracoSTM v.0.1
- Downloadable from website / boost-sandbox

Roadmap to **boost::stm**

... And The Big Open Problems



TBoost.STM + Boost

Code in Boost sandbox

Identify necessary interfaces

- Future extensions key to success
- Discussion with Boost community

Goal: review of **boost::stm v1.0** early 2010

TBoost.STM + Research

Transactions + locks + others

- Implementation and formal semantics

Efficiency

- Transactional boosting, open nesting, etc.

Test Suites

- Integrate with STAMP, STMBench7
 - *STMBench7 researchers helping us now*

And much more ...

Parallel Computing Publications

Shifting the Parallel Programming Paradigm (Best Presentation Award)

[Raytheon Information Systems and Computing (ISaC), March 2009]

Lock-Aware Transactional Memory [ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), March 2009]

Optimizing Consistency Checking for Memory-Intensive Transactions [ACM Symposium on Principles of Distributed Computing (PODC), August 2008]

C++ Move Semantics for Exception Safety and Optimization in Software Transactional Libraries [International Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS), July 2008]

Extending Contention Managers for User-Defined Priority-Based Transactions [ACM Workshop on Exploiting Parallelism with Transactional Memory and other Hardware Assisted Methods (EPHAM), April 2008]

DracoSTM: A Practical C++ Approach to Software Transactional Memory [ACM Symposium on Library-Centric Software Design (LCSD), October 2007]

Exploration of Lock-Based Software Transactional Memory [MS Thesis, 2007]

Questions?

Justin E. Gottschlich
(gottschi@colorado.edu)

Department of Electrical and Computer Engineering
University of Colorado-Boulder

