Capabilities and Limitations of Library-Based Software Transactional Memory in C++

Luke Dalessandro, Virendra Marathe, Michael Spear
Michael Scott

University of Rochester
cs.rochester.edu/research/synchronization

TRANSACT
August 2007

We Want Language Support For Transactional Memory

- Language support is appealing
  - Should make transactional memory easy to use
  - Many analysis benefits
  - etc
- BUT
  - Lots of work to implement
  - High adoption costs
  - Unclear what it should look like (yet)

In The Meantime

- Maybe we can get by with library implementation?
  - Low development overhead
  - Easily distributed
  - Unmatched flexibility
  - Reasonable performance
- Most downsides relate to APIs
  - Complex, incompatible
  - Provide poor compile-time error detection

Typical SW Library APIs

Holy Grail

```
BEGIN_TRANSACTION;
Node* curr = sentinel;
while (curr != NULL)
{
  if (curr->val >= v)
    break;
  curr = curr->next;
}
return ((curr != NULL) &&
        (curr->val == v));
END_TRANSACTION;
```

Typical SW Library APIs

Holy Grail

```
BEGIN_TRANSACTION;
Node* curr = sentinel;
while (curr != NULL)
{
  if (curr->val >= v)
    break;
  curr = curr->next;
}
return ((curr != NULL) &&
        (curr->val == v));
END_TRANSACTION;
```

Object-based with Indirection

```
bool found;
BEGIN_TRANSACTION(t);
Node* curr = sentinel->open_RO(t);
while (curr != NULL)
{
  if (curr->val >= v)
    break;
  curr = curr->next->open_RO(t);
}
found = ((curr != NULL) &&
         (curr->val == v));
END_TRANSACTION(t);
return found;
```

Typical SW Library APIs

Holy Grail

```
BEGIN_TRANSACTION;
Node* curr = sentinel;
while (curr != NULL)
{
  if (curr->val >= v)
    break;
  curr = curr->next;
}
return ((curr != NULL) &&
        (curr->val == v));
END_TRANSACTION;
```

Object-based, No Indirection

```
bool found;
BEGIN_TRANSACTION(t);
found = false;
Validator c_v;
Node* curr =
  sentinel->open_RO(t, c_v);
Node* n = curr->next; validate(c_v);
curr = n->open_RO(t, c_v);
while (curr != NULL) {
  long val = curr->val; validate(c_v);
  if (val >= v)
    break;
  n = curr->next; validate(c_v);
  ...
```

Typical SW Library APIs

Holy Grail

```
BEGIN_TRANSACTION;
Node* curr = sentinel;
while (curr != NULL)
{
  if (curr->val >= v)
    break;
  curr = curr->next;
}
return ((curr != NULL) &&
        (curr->val == v));
END_TRANSACTION;
```

Word-based

```
bool found;
BEGIN_TRANSACTION;
found = false;
Node* curr = sentinel;
while (curr != NULL) {
  if (STM_READ_LONG(&curr->val) >= v)
    break;
  curr = (Node*)STM_READ_PTR(&curr->next);
}
found = ((curr != NULL) &&
  (STM_READ_LONG(&curr->val) == v));
END_TRANSACTION;
return found;
```

Can Library TM Systems Suffice?

- In The Short Term
  - Research community needs
  - TM implementation development and testing
  - Large application development (chicken and egg)
- In The Long Term
  - Naive users
- Current APIs: no and no
- Better APIs: yes and no

TM Hooks And Who Enforces Their Use

	First Access	Per Access	Post Access
DSTM RSTM	Compiler	N/A	N/A
TL2	N/A	User	User
LibLTX	Compiler	N/A	User

RSTM2

- Just an API, **not an implementation!**
  - Back end library implementation can be any published C/C++ library TM
  - Current working implementations include RSTM, ~TL2, RTM, and several others
- Evolved through the use of RSTM
  - RSTM was designed as an open platform to experiment in C++
  - Originally adopted the DSTM interface

RSTM2

Hook	Mechanism	Enforced By
First Access	Smart Pointers	Compiler
Per Access	Smart Pointers	Compiler
Post Access	Field Accessors	Compiler*

*possible to circumvent

Smart Pointers in C++

- Overload `operator->()` and `operator*()` to make objects act like pointers.

```
template <class T>
class smart_ptr
{
    T* impl;
  public:
    T* operator->() { return impl; }
    T& operator*() { return *impl; }
    ... // const members,
        // etc...
};
```

Smart Pointers in RSTM2

- API defines 4 types of smart pointers

State	Name	`operator->()` returns
Shared	`sh_ptr`	N/A
Readable	`rd_ptr`	`const T*`
Writable	`wr_ptr`	`T*`
Private	`un_ptr`	`T*`

Accessors...

- Macros generate accessors (get/set)

```
class Node {
  // sh_ptr<Node> next;
  GENERATE_FIELD(sh_ptr<Node>, next);
  //int val
  GENERATE_FIELD(int, val);
};

wr_ptr<Node> curr(head);
curr->set_val(10);
```

And Validators

- Getters take reference to smart pointer
- This provides object based libraries a post access validation hook

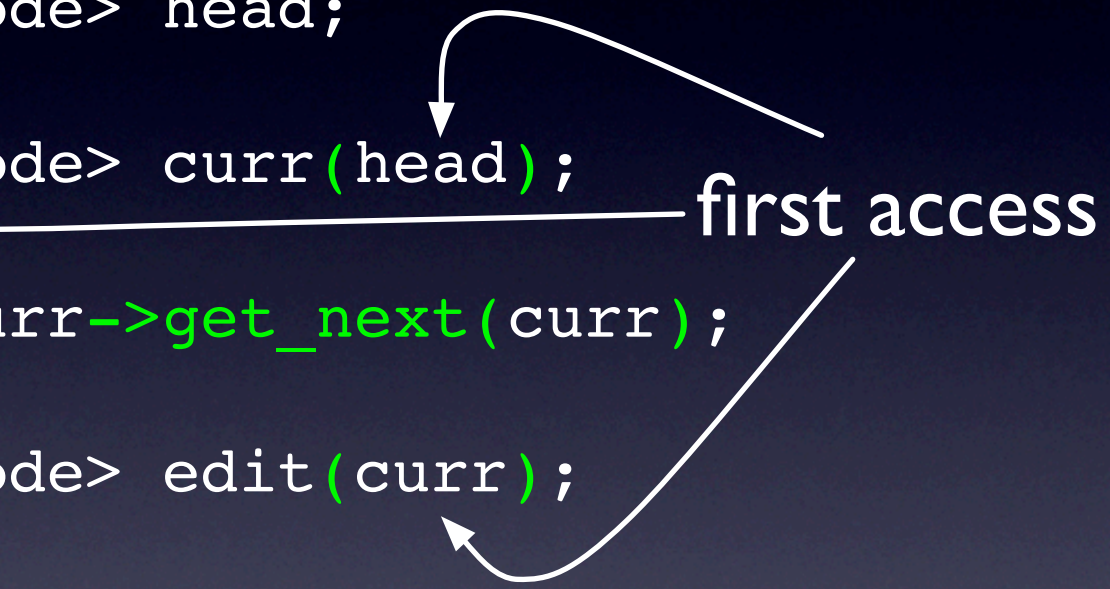
```
// RSTM backend
// rd_ptr caches object version on open
rd_ptr<Node> curr(head);

// accessor compares version on load
sh_ptr<Node> n = curr->get_next(curr);
```

Putting It Together

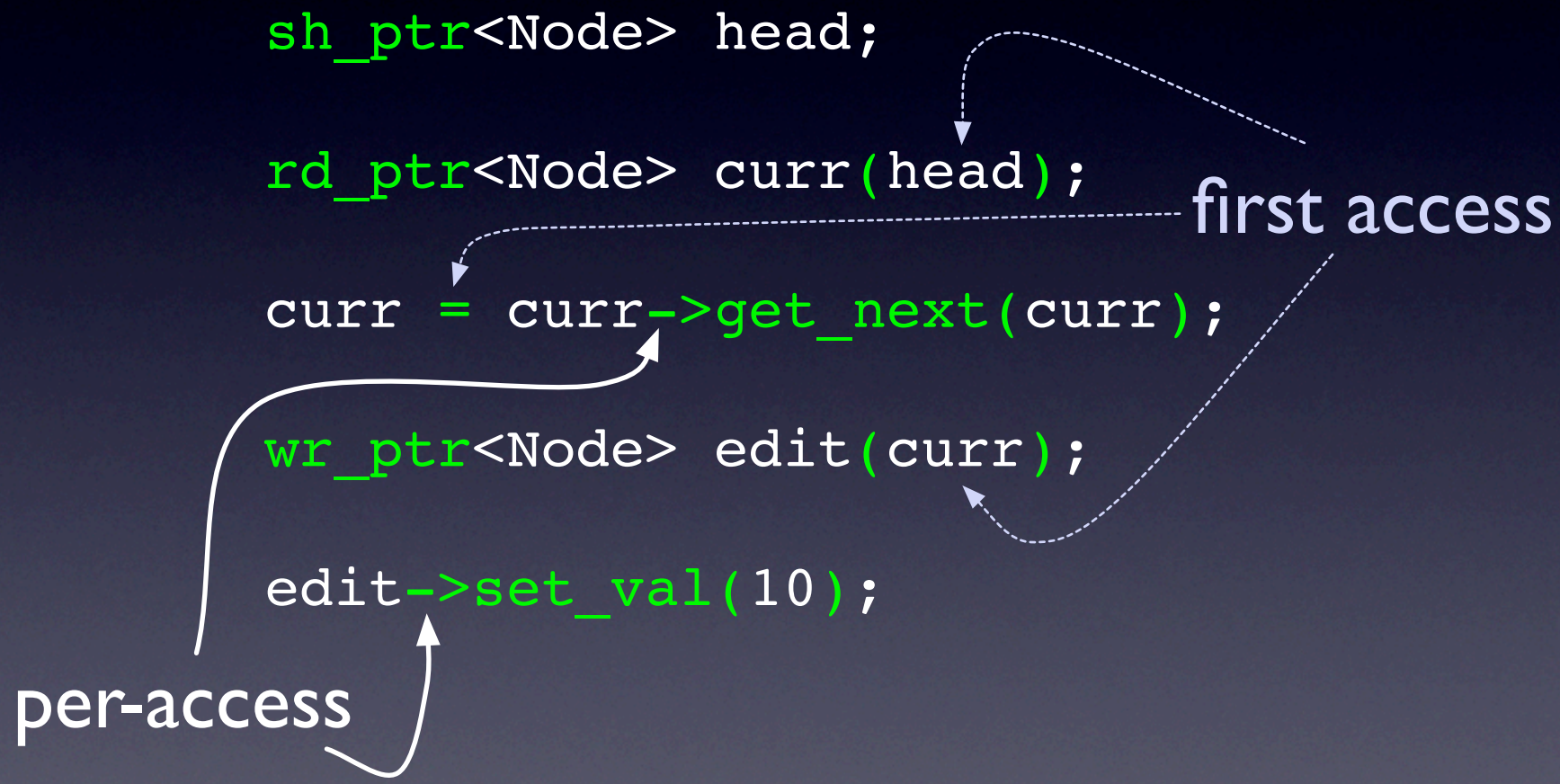
```
sh_ptr<Node> head;

rd_ptr<Node> curr(head);

curr = curr->get_next(curr);

wr_ptr<Node> edit(curr);

edit->set_val(10);
```

Putting It Together

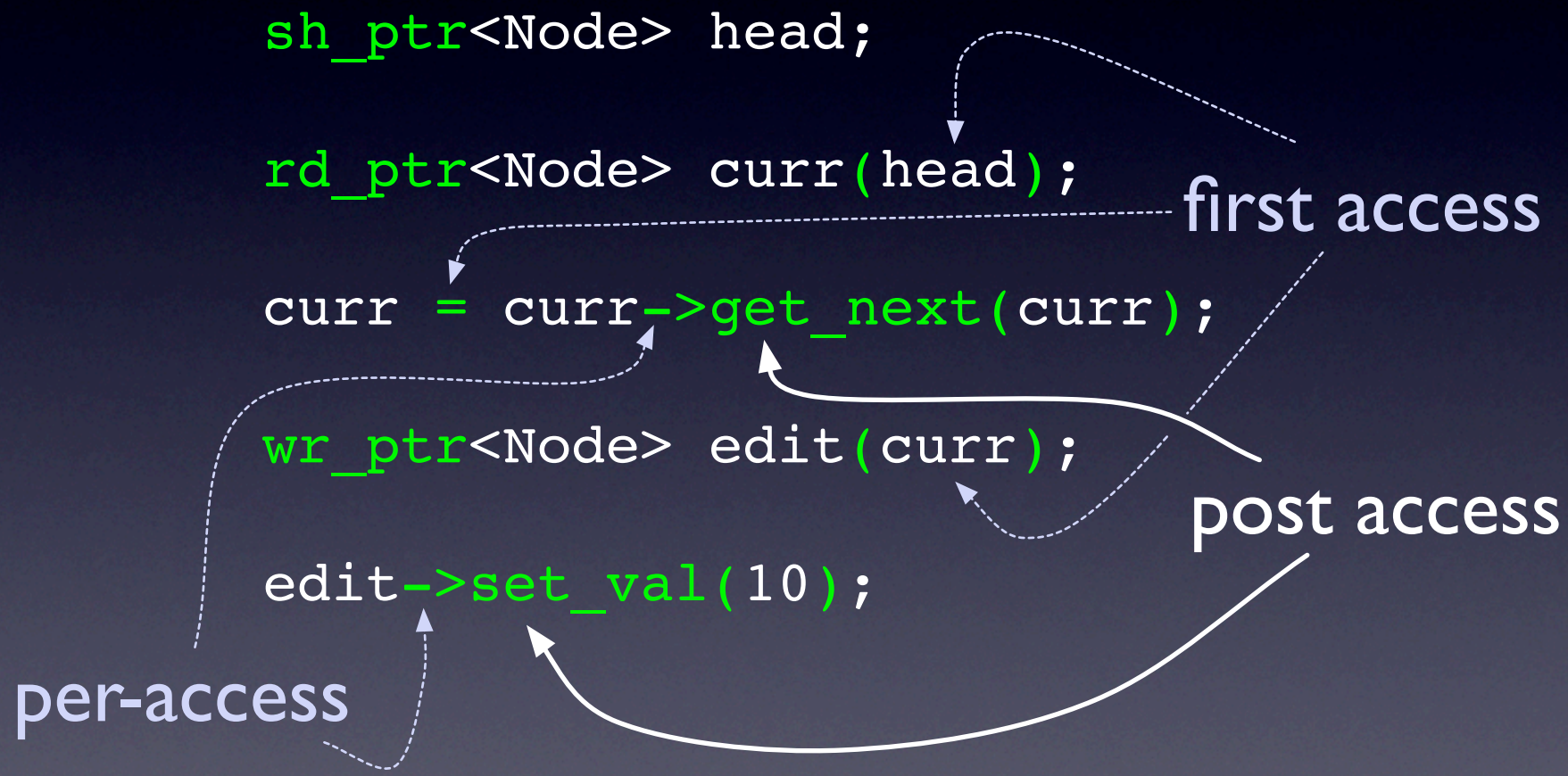
```
sh_ptr<Node> head;

rd_ptr<Node> curr(head);

curr = curr->get_next(curr);

wr_ptr<Node> edit(curr);

edit->set_val(10);
```



Putting It Together



Putting It Together



List Search With RSTM2

Holy Grail

```
BEGIN_TRANSACTION;
Node* curr = sentinel;
while (curr != NULL)
{
  if (curr->val >= v)
    break;
  curr = curr->next;
}
return ((curr != NULL) &&
        (curr->val == v));
END_TRANSACTION;
```

Any Library Type

```
bool found;
BEGIN_TRANSACTION;
found = false;
rd_ptr<Node> curr(sentinel);
while (curr != NULL)
{
  if (curr->get_val(curr) >= v)
    break;
  curr = curr->get_next(curr);
}
found = ((curr != NULL) &&
  (curr->get_val(curr) == v));
END_TRANSACTION;
return found;
```

What have we gained?

- Normalized interface
  - Write an app or benchmark once, test any library implementation or feature you want
  - Automated code sharing between transactional and non-transactional contexts with template metaprogramming
  - Clean enough for experts to use
- Nearly no dependence on programmer discipline (compiler enforced hooks)
- Enables us to write non-trivial applications
  - Delaunay mesh generation
  - See papers on app [IISWC '07], privatization [UR TR 915]

API Annoyances

- Arbitrary restrictions remain
  - `this` is not a smart pointer
  - No exceptions in shared object constructors
  - No nonlocal returns
  - Explicit transactional types (`Node`)
- Programming awkwardness
  - Accessors an validators
  - Template based code sharing
  - Error messages are baffling
- These could be solved with simple compiler modifications

More Fundamental Issues

- Shared data is explicitly specified at object level
  - Requires program design from a shared object perspective
- Only shared objects revert on abort
  - This seems like the wrong semantics
  - Particularly annoying for simple loop indices
- Many open research questions require compiler level knowledge
  - eg. how much can be inferred about shared vs. privatized data use?

Conclusions

- RSTM2 **is** suitable for short term needs
  - Allows write-once TM benchmarks
  - Simplifies large-scale application development
  - Good framework for investigating TM semantics
- BUT it is **not** appropriate for naive users
  - Generic programming is complicated
  - Explicit use of 4 types of pointers too much
  - Annoyances are overwhelming
- We feel that no pure library will suffice long term (not surprising in retrospect)

Future Work

- Implement mappings to more back ends
- Write more applications, benchmarks
- Implement lightweight compiler support to fix "the annoyances"
- Investigation of TM semantics choices and their consequences

Thanks

- The rest of the RSTM team:
  Arrvindh Shriraman, Aaron Rolett,
  Sandhya Dwarkadas
- Download RSTM2 and write your own apps and benchmarks for the last time
- http://www.cs.rochester.edu/research/synchronization/