STM in the Small: Trading Generality for Performance in Software Transactional Memory

Aleksandar Dragojević
Our motivation

• Concurrent data structures
• Hash-tables, skip-lists
  – In-memory database indices
  – Key-value stores

Software Transactional Memory

• Simplifies data structures
  – Ensures correct implementations

• Enables use of complex data structures
  – More complex than when using CAS directly
What is STM performance like?
What is STM performance like?

Graph from: Cascaval et al. “Software transactional memory: why is it only a research toy” ACMQ September 2008
What is STM performance like?

![Graph showing speedup for various benchmarks with 1, 2, 4, 8, and 16 threads. The x-axis represents different benchmarks such as SB7 Read/Write, SB7 Write, Bayes, Genome, Intruder, Kmeans High, Kmeans Low, Labyrinth, Ssca2, Vacation High, Vacation Low, Yada, Hashtable, Linked List, Rbtree, Skiplist. The y-axis represents speedup. ]
What is STM performance like?
What is STM performance like?

![Graph showing speedup for different benchmarks and configurations.](image-url)
What is STM performance like?
What is STM performance like?
What is STM performance like?

Throughput [Mops/s] vs Threads for Skip-list, CAS, and STM.
What is STM performance like?

Throughput [Mops/s]

Threads

CAS

STM

71% increase over Skip-list

78% increase over Skip-list
SpecTM: Specialized STM for Concurrent Data Structures

- CAS: CAS (Compare and Swap) is shown as fast but hard to program.
- STM (Software Transactional Memory): shown as slow but easy to program.
SpecTM: Specialized STM for Concurrent Data Structures
This talk

![Graph showing throughput vs. threads for Skip-list, CAS, and STM.](image)
This talk

![Graph showing Throughput vs. Threads with lines for Skip-list, SpecTM, and CAS.]

- Throughput [Mops/s]
- Threads

- Skip-list
- SpecTM
- CAS

0% 0%
Overview

• STM overheads
• SpecTM
  – Short-transaction API
  – Collocating data and meta-data
  – In-word meta-data
• Evaluation
Overview

• STM overheads

• SpecTM
  – Short-transaction API
  – Collocating data and meta-data
  – In-word meta-data

• Evaluation
STM overheads

- Book-keeping done in software
- Memory accesses become STM calls
- Significant overheads\(^3\)
  - With a single thread 50% overheads
  - Requires 4 threads to outperform sequential code in 75% of cases

STM API

```c
void StartTx();
void CommitTx();
word_t ReadWord(word_t *addr);
void WriteWord(word_t *addr, word_t val);
```
STM read

map(addr)
STM read

```
map(addr)
```

```
0 0 c 1 2 a b 0
```

```
<table>
<thead>
<tr>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>[c1a2aa]</td>
</tr>
<tr>
<td>[c1a2ab]</td>
</tr>
<tr>
<td>[c1a2ac]</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>
```

```
orec table
```
STM read

map(addr) → check RaW

address

0 0 c1 2 a b 0

orec table

... [c1a2aa] [c1a2ab] [c1a2ac] ...

...
STM read

index
...
[orec idx]
...
write-set
...

orcs
... → address-value

map(addr)

check RaW

address
0 0 c 1 2 a b 0

... → 
[c1a2aa]
[c1a2ab]
[c1a2ac]
...
orcs

orec table
STM read

index

write-set

orecs

address-value

map(addr)

check RaW

read orecc

read val

read orecc

o1 == o2 && !locked

address

orec table
STM read (cont’d)

log read
STM read (cont’d)

log read

<table>
<thead>
<tr>
<th>head</th>
</tr>
</thead>
<tbody>
<tr>
<td>curr</td>
</tr>
<tr>
<td>count=2</td>
</tr>
</tbody>
</table>

read-set
STM read (cont’d)
STM read (cont’d)

log read

orec <= validts

head
curr
count=3

read-set

Microsoft Research
STM read (cont’d)

- log read
- orec <= validts
  - head
curr
  count=3
- return val

read-set
STM read (cont’d)

log read

validate

orec <= validts

valid

abort

return val

head

curr

count=3

read-set
STM read (cont’d)

log read

orec <= validts

validate

valid

abort

return val

read-set

head
curr
count=3

read-set
STM read fast-path

- Significantly more expensive than CPU read instructions
  - >10 instructions
- Writes have comparable costs
  - Incurs a compare-and-swap at commit time
Overview

• STM overheads
• SpecTM
  – Short-transaction API
  – Collocating data and meta-data
  – In-word meta-data
• Evaluation
Overview

• STM overheads
• SpecTM
  – Short-transaction API
  – Collocating data and meta-data
  – In-word meta-data
• Evaluation
Short transactions

• Short read-write transactions
• Short read-only transactions
• Single-access transactions
  – Read, Write, CAS
Short read-write transactions

```c
word_t TxRWRead_1(word_t *addr_1);
word_t TxRWRead_2(word_t *addr_2);
word_t TxRWRead_3(word_t *addr_3);
...

void TxRWCommit_1(word_t val);
void TxRWCommit_2(word_t v1, word_t v2);
void TxRWCommit_3(word_t v1,
                   word_t v2,
                   word_t v3);
...
```
Short read-write transactions

word_t TxRWRead_1(word_t *addr_1);
word_t TxRWRead_2(word_t *addr_2);
word_t TxRWRead_3(word_t *addr_3);
...

void TxRWCommit_1(word_t val);
void TxRWCommit_2(word_t v1, word_t v2);
void TxRWCommit_3(word_t v1,  
                    word_t v2, 
                    word_t v3);
...

Short read-write transactions

word_t TxRWRead_1(word_t *addr_1);
word_t TxRWRead_2(word_t *addr_2);
word_t TxRWRead_3(word_t *addr_3);
...

void TxRWCommit_1(word_t val);
void TxRWCommit_2(word_t v1, word_t v2);
void TxRWCommit_3(word_t v1,
                   word_t v2,
                   word_t v3);
...

Short read-write transactions

```c
word_t TxRWRead_1(word_t *addr_1);
word_t TxRWRead_2(word_t *addr_2);
word_t TxRWRead_3(word_t *addr_3);
...

void TxRWCommit_1(word_t val);
void TxRWCommit_2(word_t v1, word_t v2);
void TxRWCommit_3(word_t v1,  
                    word_t v2,  
                    word_t v3);
...
```
Short transaction optimizations

index

... [orec idx] ...

write-set

... orecs...

address-value

map(addr)

check RaW

read orec

read val

read orec

... o1 == o2 && !locked

address

0 0 c1 2 a b 0

orec table

... [c1a2aa] [c1a2ab] [c1a2ac] ...

...
Short transaction optimizations

- Short transaction optimizations
- Address
- Orec table

```
short transaction optimizations

- Short transaction optimizations
- Address
- Orec table

- Short transaction optimizations
- Address
- Orec table

- Short transaction optimizations
- Address
- Orec table

- Short transaction optimizations
- Address
- Orec table
```

```
short transaction optimizations

- Short transaction optimizations
- Address
- Orec table

- Short transaction optimizations
- Address
- Orec table

- Short transaction optimizations
- Address
- Orec table

- Short transaction optimizations
- Address
- Orec table
```
Short transaction optimizations

index

write-set

orecs

address-value

map(addr)

check RaW

read orecc

read val

read orecc

o1 == o2 && !locked

address

orec table
Short transaction optimizations

index
... [orec idx]
... write-set
... ...
... orecs
... address-value

diagram:
- orec table
- check RaW
- read orec
- read val
- read orec
- o1 == o2 && !locked

flow:
- map(addr)
- address
- 0 0 c 1 2 a b 0
- ... [c1a2aa]
- [c1a2ab]
- [c1a2ac]
- ...

write-set
- ...
Short transaction optimizations

index
[orec idx]

[orecs]

write-set
static

write-set

address-value

map(addr)

check RaW

read orec

read val

read orec

o1 == o2 && !locked
Short transaction optimizations

index
[orec idx]
...

write-set
static

write-set

orecs

address-value

map(addr)

check RaW

read orecc

read val

read orecc

CAS (from write)

o1 == o2 && !locked

address

00 c1 2a b0

orec table

[...]

[c1a2aa]

[c1a2ab]

[c1a2ac]

[...]

write-set

address
Short transaction optimizations (cont’d)

- read-set

log read

- orec <= validts

validate

- valid

- abort

- return val

read-set

head

curr

count=3

read-set
Short transaction optimizations (cont’d)

log read

orec <= validts

head
curr
count=3

merged with write-set

static

read-set

validate

valid

abort

return val
Short transaction optimizations (cont’d)

- log read
- validate
- return val
- abort

orec <= validts

merged with write-set
static

read-set

head
curr
count=3

read-set
Using short transactions

With “traditional” transactions the whole operation is a single transaction
Using short transactions

Split operation into a sequence of short transactions
Using short transactions

“Glue” them together using techniques similar to lock-free algorithms (e.g. pointer marking)
Using short transactions

Is this simpler than using CAS directly?

“Glue” them together using techniques similar to lock-free algorithms (e.g. pointer marking)
Using short transactions

Is this simpler than using CAS directly?

Yes: transactions eliminate tricky races

“Glue” them together using techniques similar to lock-free algorithms (e.g. pointer marking)
Mix short and ordinary transactions

Implement corner cases using ordinary transactions
Example

Remove 20

```
10
10
10
```
```
20
20
```
```
30
```
```
40
40
40
```
Example

Remove 20

TxSingleRead

```
10
10
10
```

```
20
20
```

```
30
```

```
40
40
40
```

Example

Remove 20

TxSingleRead
Example

Remove 20

TxSingleRead
Example

Remove 20

Short read-write transaction to mark the node and unlink it
Example

Remove 20

Short read-write transaction to mark the node and unlink it
Example

Remove 20

This becomes trickier when using CAS directly
Example

Remove 20

We have to use several CASes

CAS

10
10
10
20
20
30
40
40
40
Example

Remove 20

We have to use several CASes

CAS
Example

Remove 20

We have to use several CASes

CAS

10
10
10
20
20 ×
30
40
40
40
Example

Remove 20

We have to use several CASes

CAS

<table>
<thead>
<tr>
<th>10</th>
<th>20</th>
<th>10</th>
<th>30</th>
<th>40</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

✗

✗
Example

Remove 20

We have to use several CASes

CAS

10
10
10

20
20

✓

✓

30

40
40
40
Example

Remove 20

We have to use several CASes

CAS

10
10
10

20 ×
20 ×

30

40
40
40
Example

Remove 20

We have to use several CASes

CAS
Example

Remove 20

We have to use several CASes
Example

Remove 20

There might be a race at each of these steps

CAS
Example

Remove 20

We need to make sure other operations clean-up for us

CAS
Example

Remove 20

*All* interactions between operations have to be handled correctly
Example

Remove 20

E.g. removal of an element that is still being added to the list

CAS
Overview

• STM overheads

• SpecTM
  – Short-transaction API
  – Collocating data and meta-data
  – In-word meta-data

• Evaluation
“Traditional” STM data layout

<table>
<thead>
<tr>
<th>W₁</th>
<th>W₂</th>
<th>W₃</th>
<th>W₄</th>
<th>...</th>
</tr>
</thead>
</table>

application data

<table>
<thead>
<tr>
<th>O₁</th>
<th>O₂</th>
<th>O₃</th>
<th>O₄</th>
<th>...</th>
</tr>
</thead>
</table>

orecs
“Traditional” STM data layout

General
False conflicts
Cache misses

application data

... W₂ W₃ W₄ ...

... W₁...

... O₁ O₂ O₃ O₄ ...

orecs
Collocate data and meta-data

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$W_1$</td>
<td>$O_1$</td>
</tr>
<tr>
<td>$W_2$</td>
<td>$O_2$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>$W_3$</td>
<td>$O_3$</td>
</tr>
<tr>
<td>$W_4$</td>
<td>$O_4$</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Simplify mapping
No false conflicts
Same cache line
Collocated meta-data optimizations

index

... [orec idx]

... write-set

... static

... write-set

... orece

... address-value

map(addr)

check RaW

read orece

read val

read orece

o1 == o2 && !locked

orece table

0 0 c1 2 a b 0

c1a2aa
c1a2ab
c1a2ac

...
Collocated meta-data optimizations

index
...   write-set
[orec idx]
...   static
write-set
...   orecs

address-value

map(addr)
check RaW
read orecc
read val
read orecc

o1 == o2 && !locked

[ orecc table ]
...   [c1a2aa]
[ c1a2ab ]
[ c1a2ac ]
...
Collocated meta-data optimizations

index

write-set

orecs

address-value

map(addr)

check RaW

read orecc

read val

read orecc

o1 == o2 && !locked

write-set

static

orecs

address

simplified mapping

better cache locality

0 0 c 1 2 a b 0

orec table

...
Pure value-based validation

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$W_1$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$W_2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$W_3$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$W_4$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Restrict values
Restrict access patterns
Single-word operations
Value-based optimizations

- Address-value
- Write-set
- Orecs

Index

- ... [orec idx] ...

Map(addr)

- Check RaW

Read orec

- Read val

Read orec

- CAS (from write)

O1 == o2 & !locked

Orecs

- Simplified mapping
- Better cache locality

Address

- 0 0 c 1 2 a b 0

Orec table
Value-based optimizations

- Value-based optimizations address orecc table
- write-set optimizations
- Simplified mapping better cache locality
- o1 == o2 && !locked
Value-based optimizations

<table>
<thead>
<tr>
<th>Value-based optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>write-set</td>
</tr>
<tr>
<td>[orec idx]</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>write-set</td>
</tr>
<tr>
<td>orecs</td>
</tr>
<tr>
<td>index</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>[orec idx]</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>address-value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>c</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>c</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>a</td>
</tr>
</tbody>
</table>

write-set

orecs

index

map(addr)

check RaW

read orec

read val

read orec

o1 == o2 && !locked

simplified mapping

better cache locality

orec table

write-set

static

Map

check RaW

read orec

read val

read orec

 CAS (from write)

address

orec table
API changes
API changes

word_t ReadWord(TVar *addr);
void WriteWord(TVar *addr, word_t val);

word_t TVarToWord(TVar addr);
TVar WordToTVar(word_t val);
API changes

Transactions access TVar, not word_t

```c
word_t ReadWord(TVar *addr);
void WriteWord(TVar *addr, word_t val);
```

```c
word_t TVarToWord(TVar addr);
TVar WordToTVar(word_t val);
```
API changes

Transactions access TVar, not word_t

```c
word_t ReadWord(TVar *addr);
void WriteWord(TVar *addr, word_t val);
```

Calls to update TVar non-transactionally

```c
word_t TVarToWord(TVar addr);
TVar WordToTVar(word_t val);
```
Pure value-based short read-write

read val
Pure value-based short read-write

read val

!locked
Pure value-based short read-write

read val

!locked

CAS
Pure value-based short read-write

- read val
- !locked
  - abort
  - CAS
Pure value-based short read-write
Pure value-based short read-write

读取值

非锁定

失败

CAS

成功

日志

写入集中

静态
Pure value-based short read-write

1. Read val
2. Non-locked? (CAS)
   - Yes: Success
     - Log
     - Return val
   - No: Abort

Notes:
- CAS (Compare and Swap)
- Write-set
- Static
Pure value-based short read-write

1. Read val
2. Check if locked
   - If not locked, proceed to CAS
   - If locked, abort
3. Attempt to CAS
   - If successful, log and return val
   - If failed, handle failure
STM vs. SpecTM

STM

SpecTM
Overview

• STM overheads
• SpecTM
  – Short-transaction API
  – Collocating data and meta-data
  – In-word meta-data

• Evaluation
Evaluation

Throughput [Mops/s]

Skip-list (90% lookups)

STM

CAS

Threads
Evaluation

Throughput [Mops/s]

Threads

Skip-list (90% lookups)

CAS

STM

240%
Evaluation

Throughput [Mops/s] vs. Threads

- Skip-list (90% lookups)
- CAS
- SpecTM-Short
- STM

48% increase in throughput with CAS compared to STM.
Evaluation

Throughput [Mops/s]

Skip-list (90% lookups)

CAS

SpecTM-Short-Colloc

SpecTM-Short

STM

Threads
Evaluation

Throughput [Mops/s]

Threads

Skip-list (90% lookups)
CAS
2%
SpecTM-Short-Val
SpecTM-Short-Colloc
SpecTM-Short
STM
Evaluation

Hash-table (90% lookups)

Throughput [Mops/s]

Threads
Evaluation

Hash-table (90% lookups)

Throughput [Mops/s]

Threads

SpecTM-Short-Val
SpecTM-Short-Colloc
SpecTM-Short
CAS
STM

0%

0 5 10 15
Conclusions

• Trade generality for performance in SpecTM
  – Restrict STM API
  – Control data layout

• STM-based data structures perform as well as the ones built directly from CAS

• Do we need SpecTM with upcoming Intel TSX?
  – Best-effort limited-size transactions
SpecTM: Specialized STM for Concurrent Data Structures