An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees

Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, Kunle Olukotun

Computer Systems Laboratory
Stanford University
http://tcc.stanford.edu

Why Hybrid Transactional Memory?

- Transactional Memory (TM) systems are promising
  - Large atomic blocks simplify parallel programming
  - Speed of fine-grain locks with simplicity of coarse-grain locks

- TM can be implemented in either hardware or software
  - Hardware TM (HTM) is fast but inflexible & costly
  - Software TM (STM) is flexible but slow

- Signature-Accelerated TM (SigTM) is a new hybrid TM
  - Uses hardware signatures to accelerate software transactions
    - Fast, flexible, & cost-effective
  - Implements strong isolation of transactional code
    - Correct & predictable execution of software transactions

Outline

What Can We Accelerate?

Compiler →

| High-level | Low-level |

```
ListNode n;
atomic {
  n = head;
  if (n != null) {


    head = head.next;

  }
}
```

```
ListNode n;
STMstart();
  n = STMread(&head);
  if (n != null) {
    ListNode t;
    t = STMread(&head.next);
    STMwrite(&head, t);
  }
STMcommit();
```

- What do these STM functions do?

3

STMstart

- Called at transaction start → init transaction meta data

```
STMstart() {
    checkpoint(); // used to rollback
    other_initialization();
}
```

- Constant overhead cost per transaction
- Expensive only for short transactions

STMread

- Called to read shared data → add to read-set

```
STMread(addr) {

  if (addr in WriteSet) // get latest value
    return WriteBuffer.getValue(addr);
  val = *addr;
  if (!isVersionValid(addr)) // someone wrote?
    conflict_handler();
  ReadSet.insert(addr);
  return val;
}
```

- Building the read-set is expensive
- Overhead cost per transaction varies
  - Locality of read accesses, size of read-set, transaction length

STMwrite

- Called to write shared data → add to write-set

```
STMwrite(addr, val) {
    WriteBuffer.insert(addr, val);
}
```

- Overhead cost per transaction varies
  - Locality of write accesses, size of write-set, transaction length
- Significantly less expensive than `STMread` (reads ≥ writes)

STMcommit

- Called at transaction end → atomically commit changes

```
STMcommit() {
  foreach (addr in WriteSet) // lock write-set
    if (!lock(addr))
      conflict_handler();

  foreach (addr in ReadSet)  // validate read-set
    if (!isVersionValid(addr))
      conflict_handler();

  foreach (addr in WriteSet) // commit write-buffer
    *addr = WriteBuffer.getValue(addr);

  foreach (addr in WriteSet) // unlock write-set
    unlock(addr);
}
```

- Expensive: scan read-set (1x); scan write-set (3x), locks

How Slow Can STM Be?
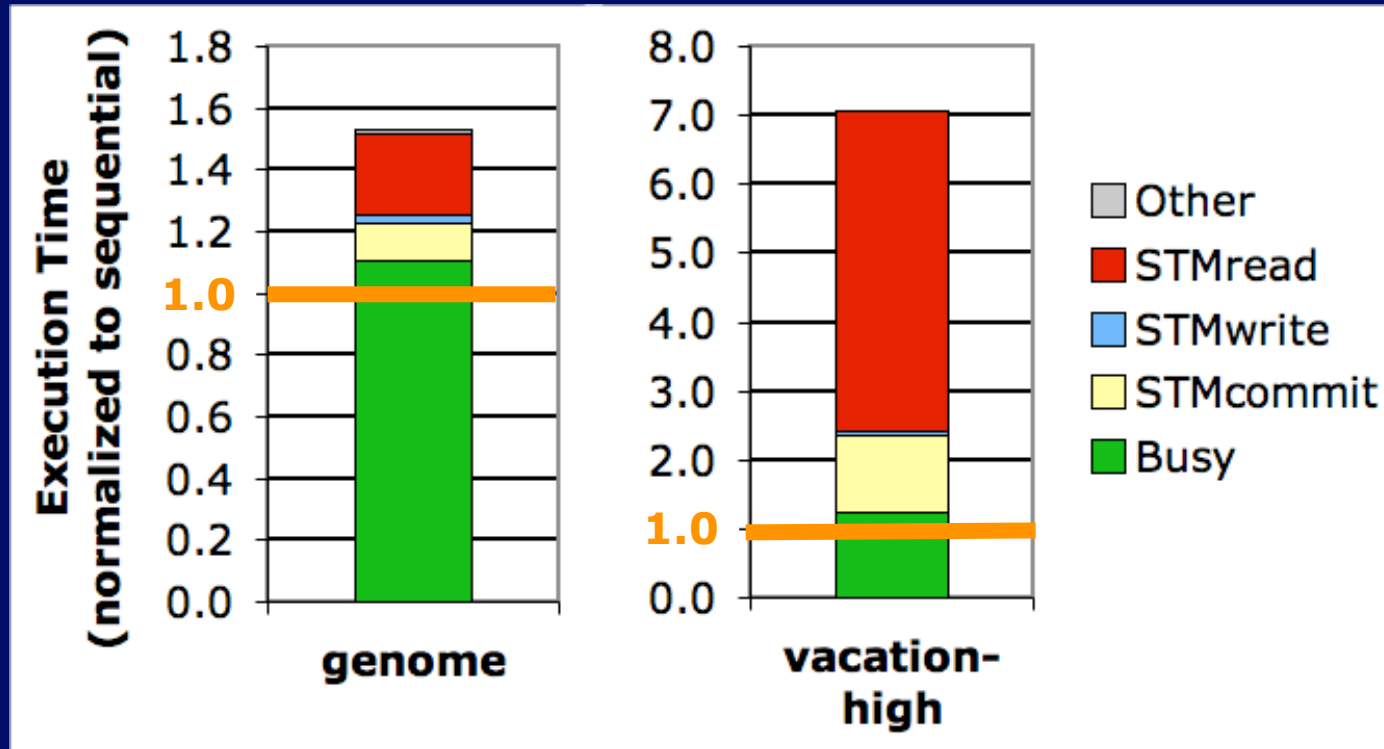
- 1.5x - 7x slowdown over sequential
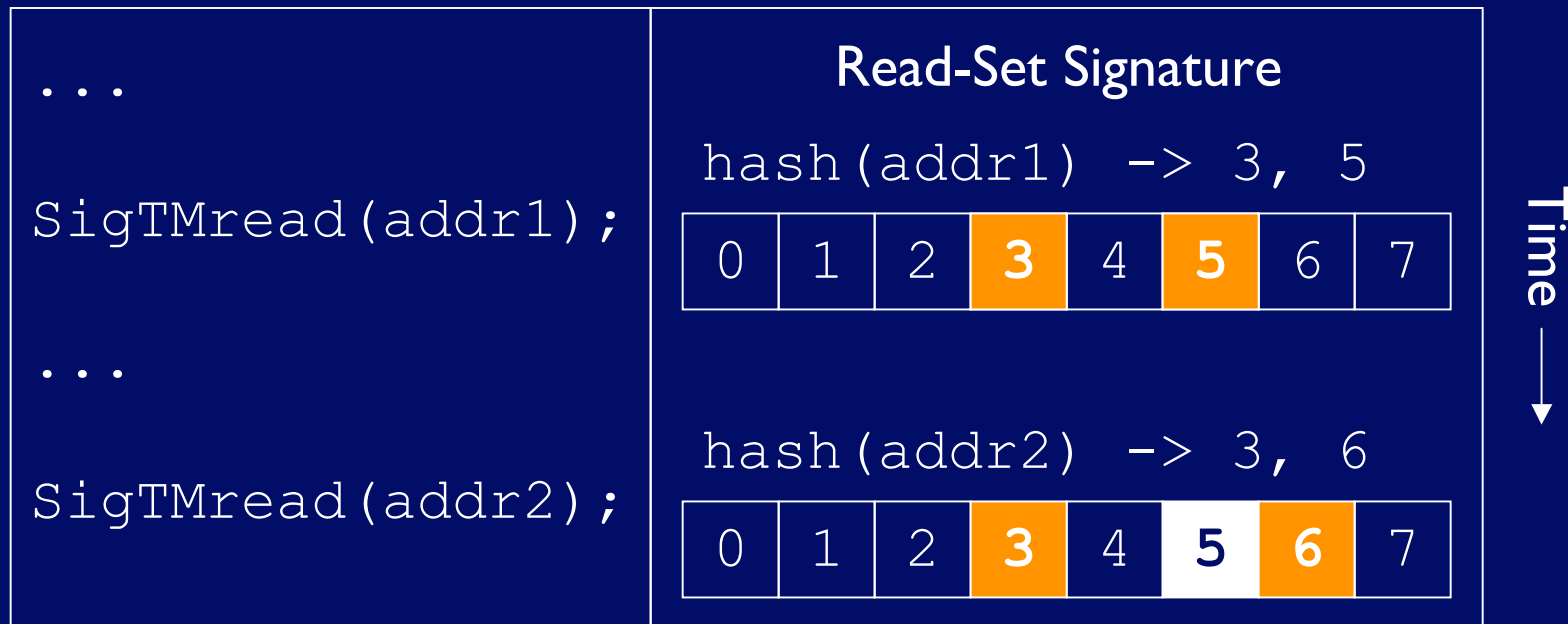- Hybrid TM should focus on `STMread` **and** `STMcommit`

SigTM

- SigTM simplifies STM by using simple hardware

	STM	SigTM
Read-set conflict detection	SW (version #)	HW (read-set signature)
Write-set conflict detection	SW (locks)	HW (write-set signature)
Write-set versioning	SW	SW

SigTM Hardware

- SigTM adds a little HW (signatures) to accelerate STM
  - Each HW thread has 2 HW signatures: read-set, write-set
  - No other HW modifications (e.g., no extra cache states)

- `SigTMread` and `SigTMwrite` populate signatures

```
...

SigTMread(addr1);

...

SigTMread(addr2);
```

Read-Set Signature

`hash(addr1) -> 3, 5`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

`hash(addr2) -> 3, 6`

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |

Time →

SigTM Hardware (cont)

- Signatures watch coherence messages
  - SW enables/disables

...	Read-Set Signature
`*addr1 = val`	`hash(addr1) -> 3, 5`

0	1	2	**3**	4	**5**	**6**	7

- On hit in signature, either:
  - Trigger SW abort handler (conflict detection)
  - NACK remote request (isolation enforcement)

- Signatures may generate false conflicts
  - Performance but not correctness issue
  - Reduce with longer signatures & better hash functions

11

SigTMstart

```
SigTMstart() {
    checkpoint(); // used to rollback
    other_initialization();
    enable_read_sig_lookup();
}
```

- Read-set signature starts monitoring coherence messages
  - If hit, signature invokes `conflict_handler()`
  - Continuous validation of read-set

SigTMread

```
SigTMread(addr) {

    if (addr in WriteSet) // get latest value
      return WriteBuffer.getValue(addr);

    // No need to validate addr here

    read_sig_insert(addr);
    return *addr;
}
```

- ▪ `SigTMread` does not need to:
  - Validate read address → continuous validation by HW signature
  - Build software read-set → just add to read-set signature

SigTMwrite

```
SigTMwrite(addr, val) {
    write_sig_insert(addr);
    WriteBuffer.insert(addr, val);
}
```

- `SigTMwrite` **populates write-set signature**
  - **Used during** `SigTMcommit`
- **Write-set versioning still in SW**

SigTMcommit

```
SigTMcommit() {
  enable_write_sig_lookup();
  foreach (addr in WriteSet) // remove from...
    fetch_exclusive(addr);    // ...other caches
  enable_write_sig_nack(); // ensure atomic commit
  disable_read_sig_lookup();
  foreach (addr in WriteSet) // commit write-buffer
    *addr = WriteBuffer.getValue(addr);
  disable_write_sig_lookup();
}
```

- Read-set signature eliminates scan of read-set to validate
- Write-set signature eliminates locks
- Two write-set scans instead of three

How Much Smaller is the Overhead?

- Measured dynamic instruction counts
  - R = # words in read-set; W = # words in write-set

	STM	SigTM
Read Barrier	19	8
Commit	44 + 16R + 31W	41 + 12W

- Measured single-thread performance relative to sequential

|	STM	SigTM	Improvement
genome	0.65	0.81	1.25x
vacation-high	0.14	0.41	2.93x

Experimental Setup

- Execution-driven simulation to compare: SigTM, STM, HTM

- STAMP: Stanford Transactional Apps for Multiprocessing

  - 4 benchmarks for TM research written in C
    - delaunay: Delaunay mesh generation
    - genome: gene sequencing
    - kmeans: K-means clustering
    - vacation: travel reservation system (similar to SPECjbb2000)

  - Parallelized from sequential code
    - Coarse-grain transactions (intuitive parallel programming)
    - Over 95% of time is spent in transactions

  - STM code is manually optimized (same code for SigTM)
    - HTM code has no instrumentation on reads/writes

How Fast is SigTM?
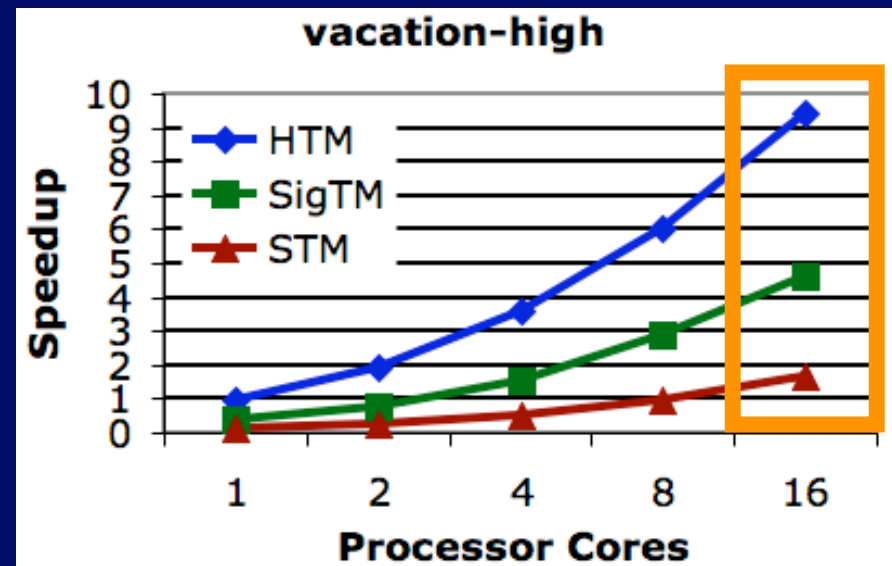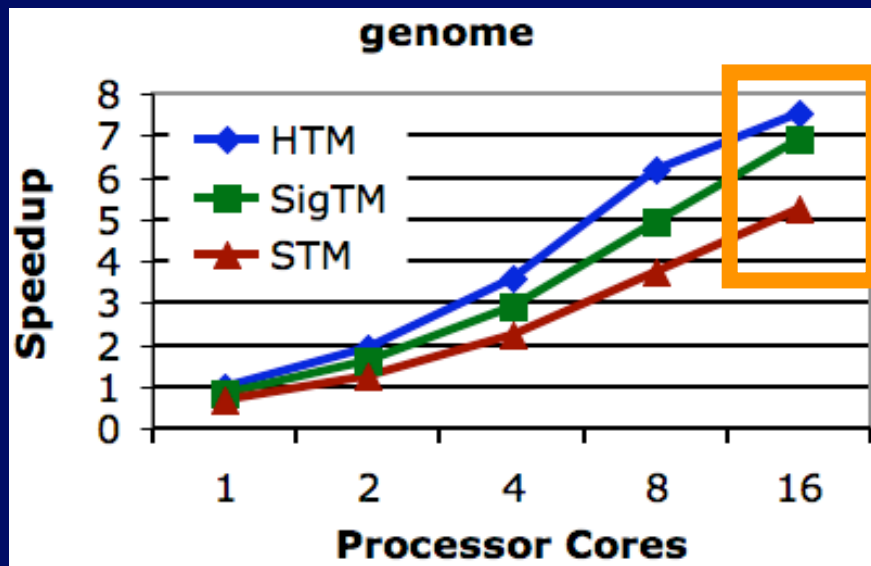
- SigTM faster than STM but slower than HTM

- Genome: SigTM 30% faster than STM; within 10% of HTM

- Vacation: SigTM 2.8x faster than STM; 2x slower than HTM
  - Many non-redundant read barriers → large performance difference

How Much Hardware Does it Cost?
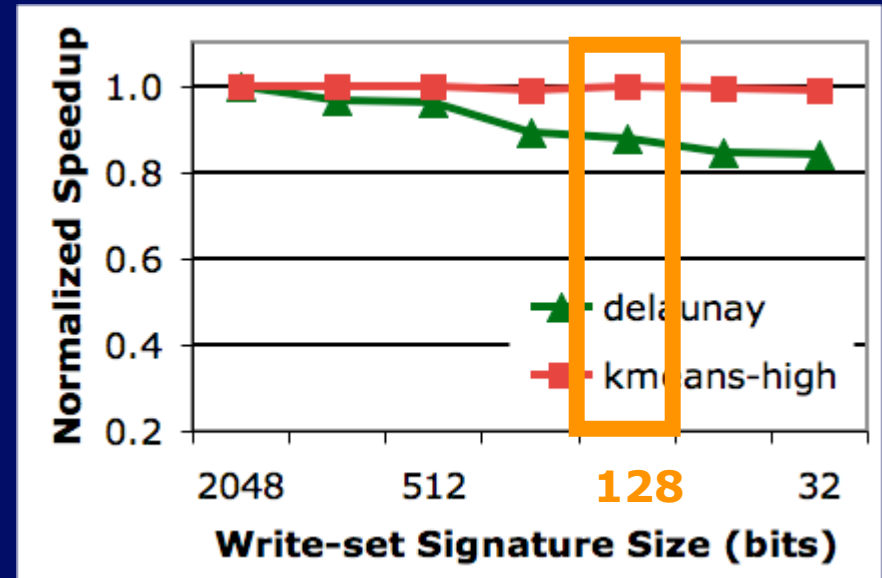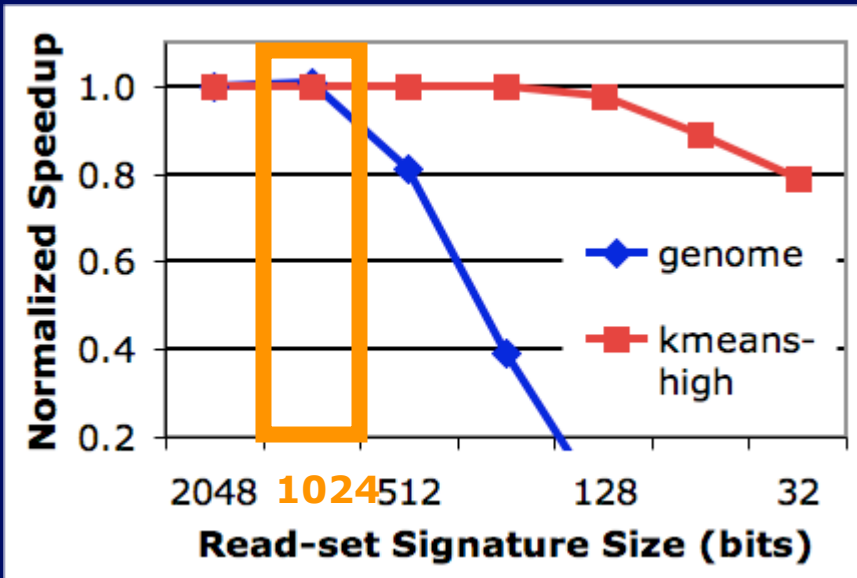
- Decreased signature size to increase false conflicts

- Performance sensitive to read-set signature length
  - 1024 bits is recommended

- Performance insensitive to write-set signature length
  - 128 bits is recommended

Outline

- Introduction

- SigTM Performance

- **SigTM Strong Isolation**

- **Related Work**

- **Conclusion**

Example Program: Privatization

Thread 1

```
ListNode n;
atomic {
  n = head;
  if (n != null)
    head = head.next;
}
// use n.val many times
```

Thread 2

```
atomic {
  ListNode n = head;
  while (n != null) {
    n.val++;
    n = n.next;
  }
}
```

- Two acceptable outcomes:
  - T1 commits first; T1 privatizes & uses non-incremented `n.val`
  - T2 commits first; T1 privatizes & uses incremented `n.val`

- Works correctly with lock-based synchronization
  - Race-free program

Unpredictable Results with STM?

Thread 1

```
ListNode n;
atomic {
  n = head;
  if (n != null)
    head = head.next;
}
// use n.val many times
```

Thread 2

```
atomic {
  ListNode n = head;
  while (n != null) {
    n.val++;
    n = n.next;
  }
}
```

- All STMs may lead to unexpected results with this code
  - T1 may use both old & new value after privatization

- Cause: non-transactional accesses are not instrumented
  - Non-Tx writes do not cause Tx to abort
  - Tx commit not isolated with respect to non-TX accesses

22

Strong Isolation

- <u>Definition</u>: transactions are isolated from non-Tx accesses

- HTM → inherent strong isolation
  - Non-Tx cause coherence messages
  - Conflict detection mechanism enforces strong isolation

- STM → supplemented strong isolation
  - Additional barriers needed in non-Tx accesses
  - Some can be optimized but still a source of overhead

- SigTM → inherent strong isolation
  - Without additional instrumentation or overhead

How SigTM Provides Strong Isolation

Initially: x=0

```
// T1            // T2
atomic {          ...
   t=x;           ...
   ...         ← x=10;
   x=t+1;         ...
}                 ...
```

- **Non-Tx write to read-set?**
  - Hits in read-set signature → transaction aborts

Outline

- Introduction

- SigTM Performance

- SigTM Strong Isolation

- **Related Work**

- **Conclusion**

SigTM and Other Hybrid TMs

- Kumar (PPoPP'06) and HyTM (ASPLOS'06)
  - Require significant cache modifications for HTM
  - Need 2 versions of transaction code

- HASTM (MICRO'06)
  - Requires cache modifications (expensive for nesting)
  - Cache updates from prefetching / speculation problematic

- RTM (ISCA'07 – later today)
  - Requires significant cache modifications (TMESI)
    - Cache handles common case conflict detection <u>and</u> buffering
  - Poor performance (slower than sequential…)

- **None has strong isolation without barriers in non-Tx**

SigTM and Signature-based HTMs

- Bulk (ISCA'06)
  - First use of signatures for TM
  - Requires additional HW for write versioning

- LogTM-SE (HPCA'07)
  - Additional HW to implement undo log
  - Additional HW to remember recently logged lines
  - Recommended smaller signatures (32–64 bits)

Conclusions

- SigTM is a hybrid TM that:

  - Uses minimal additional hardware
    - 1K bits for read-set signature; 128 bits for write-set signature
    - No modification to caches

  - Reduces the runtime overhead of SW transactions
    - Eliminates SW read-set, locks, and time stamps
    - Continuous validation of read-set by HW signatures

  - Leads to good performance
    - Outperforms STM by 30% – 280%
    - Slowdown compared to HTM is 10% – 100%

  - Delivers strong isolation for predictable behavior

Questions?

STAMP

Stanford Transactional Applications for Multiprocessing

A new benchmark suite designed for TM research

http://stamp.stanford.edu