ADVANCED PARALLEL COMPUTING
LECTURE 06 - TRANSACTIONAL MEMORY

Holger Fröning
holger.froening@ziti.uni-heidelberg.de
Institute of Computer Engineering
Ruprecht-Karls University of Heidelberg

Some material by Falsafi, Hardavellas, Nowatzyk of EPFL, Northwestern, CMU

LOCKING: PRICE OF GUARANTEED CORRECTNESS

LOCKS

Getting locking right

- Taking too few locks
- Taking too many locks
- Taking the wrong locks
- Taking the locks in wrong order
- Freeing locks on error

Locking is expensive

- Coherence actions
- Wait/Transfer/Release

Locking is pessimistic

AN EXAMPLE

```
insert (set s, key k)

lookup (set s, key k)

delete (set s, key k)
```

A concurrent data structure called “set”

Operations on this data structure

Concurrency

- Insert/Delete/Lookup should be mutually exclusive

Now, support: `transfer ( set s1, set s2, key k)`

- Requirement: k has always to be stored in one of both sets
- Never both or neither

Fine grain locking: lock elements, deadlocks!

Coarse grain locking: lock data structure, less concurrency!

AN EXAMPLE

Even with coarse grain locking

- Breaks abstraction: exposes internal lock
- Deadlock: which set's lock to grab first?

An abstract but ideal solution

Where "atomic" has

- Simplicity of coarse grain locking
- Concurrency of fine grain locking
- Without fine-grain locking overheads

```
void transfer (set s1, set s2, key k)
{
  atomic {
    delete (s1, k)
    insert (s2, k)
  }
}
```

=> The promise of Transactional Memory

TRANSACTIONAL MEMORY

Region that executes serially (isolated/atomic)

- Inspired by database transactions, but different

Implementation: **speculative execution**

- Serialize only on dynamic conflicts (eager or lazy)
- E.g., when key manipulated by different threads
- Partly overcomes the granularity/complexity tradeoff
- Avoid conservative serialization of locking

Mutual exclusion: **optimistic** instead of pessimistic

HOT TOPIC / GARTNER'S HYPE CYCLE

Pioneering work

- First idea by Lomet, 1977
- Herlihy+, ISCA1993

Speculative locking

- E.g. Rajwar+, MICRO2001 & ASPLOS 2002

Software Transactional Memory

- E.g. Herlihy+, PODC2003; Harris+, OOPSLA 2003; ...

Hardware Transactional Memory

- E.g. Hammond+, ISCA2004, ASPLOS2004; UTM (HPCA2005); VTM (ISCA2006); LogTM (HPCA2006); ...

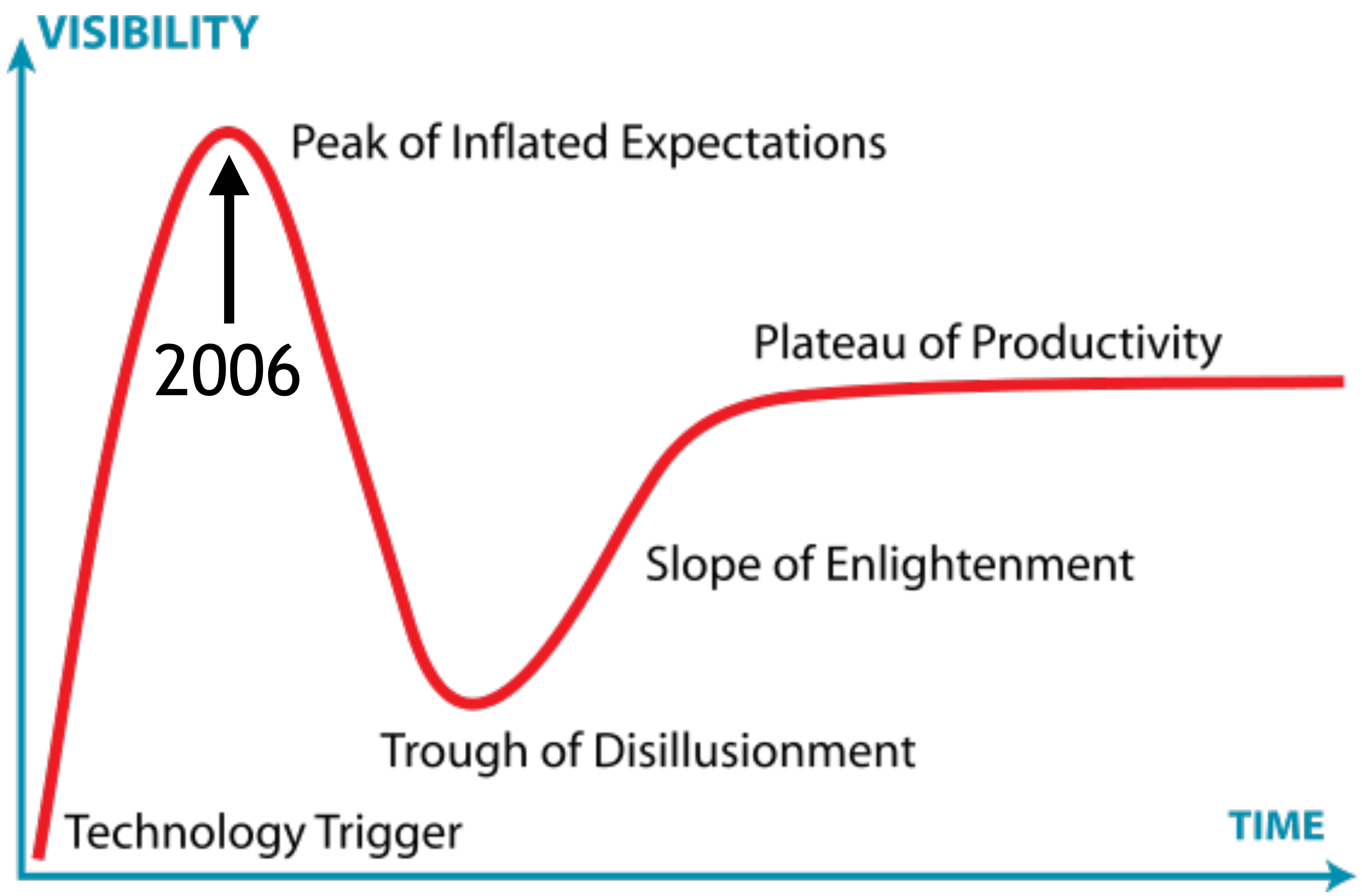
HW/SW Hybrids...

Commercial implementations

- IBM BlueGene/Q (2011), Intel Haswell (2012)

Lots of TM papers in the recent years

- 300+ citations in „Transactional Memory", 2nd Ed., 2010



Source: wikipedia.com

SPECULATIVE LOCKING

Correctly synchronizing a program with locks is hard

Fine-grain locking

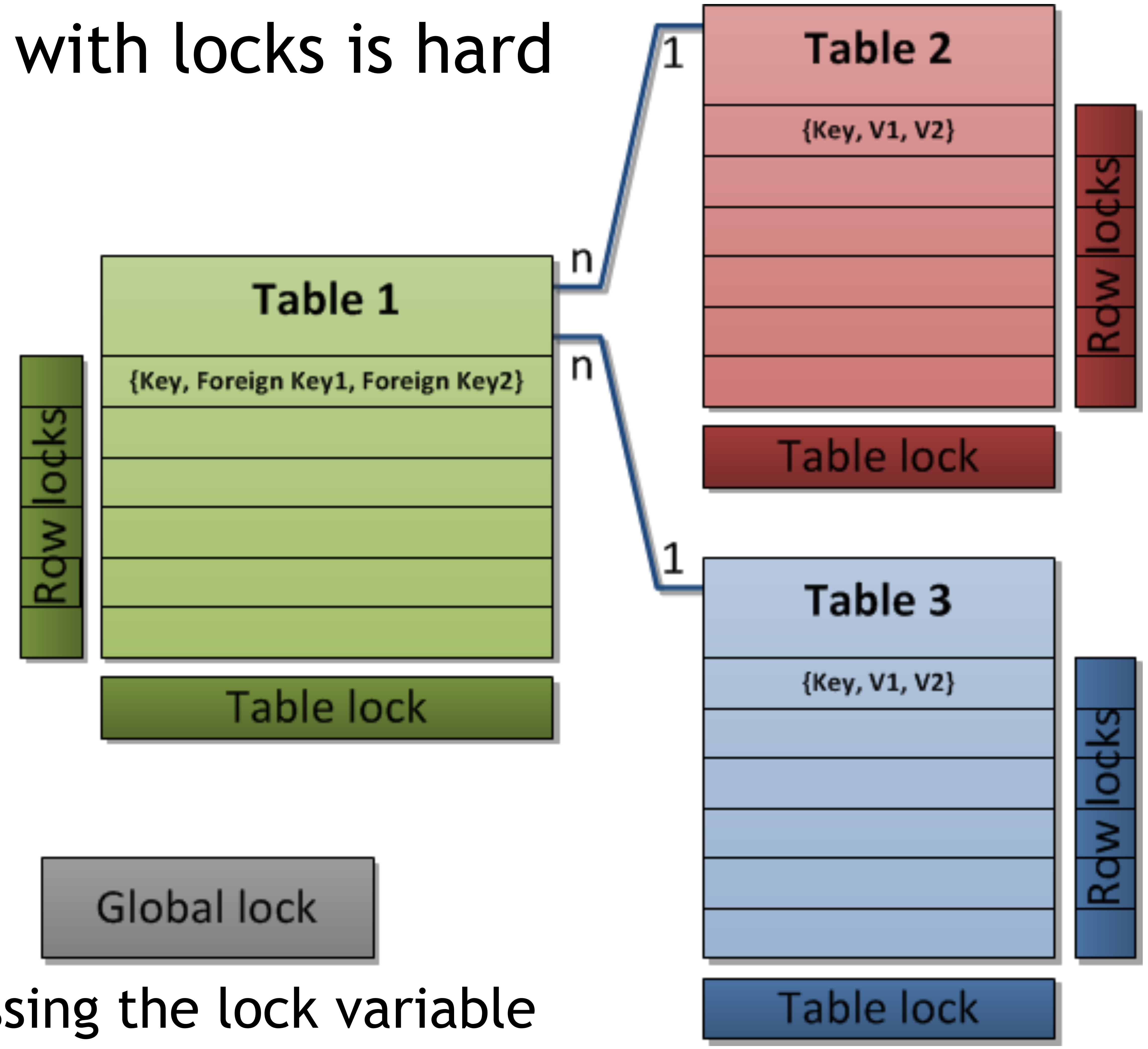
- Difficult to program
- High overhead

Coarse-grain locking

- Poor performance
- Even worse scalability

But, concurrent critical sections usually access disjoint data

- So, they could run in parallel...
- ... except that they conflict on accessing the lock variable



SPECULATIVE LOCK ELISION

- Speculatively execute critical sections in parallel
- Key idea: Detect and elide lock access
  - Upon a lock acquire, don't actually acquire the lock
  - Checkpoint the processor state
  - Run critical sections in parallel
  - Detect conflicting data accesses via coherence protocol
  - Any invalidates before the lock release cause rollback
    - Then retry by acquiring the lock normally
- Advantages
  - No locking overhead, since lock is not actually acquired
  - Allows concurrent execution of non-conflicting critical sections
- How to detect critical sections?
  - Detect temporally silent stores

Ravi Rajwar and James R. Goodman. 2001. Speculative lock elision: enabling highly concurrent multithreaded execution. In Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture (MICRO 34).

INTRODUCTION TO TRANSACTIONAL MEMORY

TM'S BIG IDEAS

Big Idea #1: **no locks, just shared data**

Big Idea #2: **optimistic (speculative) concurrency**

Execute critical sections speculatively, abort on conflict

"Better to beg for forgiveness than to ask for permission"

```
struct account_t { int balance; };
shared struct account_t a[MAX];
int fromID, toID, amount;

begin_transaction();
if (a[fromID].balance >= amount) {
  a[fromID].balance -= amount;
  a[toID].balance += amount;
}
end_transaction();
```

TM'S READ/WRITE SETS

Read set: set of shared addresses critical sections reads

Example: `a[37].balance, a[241].balance`

Write set: set of shared addresses critical section writes

Example: `a[37].balance, a[241].balance`

```
struct account_t { int balance; };
shared struct account_t a[MAX];
int fromID, toID, amount;

begin_transaction();
if (a[fromID].balance >= amount) {
  a[fromID].balance -= amount;
  a[toID].balance += amount;
}
end_transaction();
```

TM'S BEGIN

`begin_transaction()`

Take a local register snapshot

Begin locally tracking read set (remember addresses you've read), see if anyone else is trying to write it

Locally buffer all your writes (invisible to other processes)

=> Local actions only: no lock acquire

```
struct account_t { int balance; };
shared struct account_t a[MAX];
int fromID, toID, amount;

begin_transaction();
if (a[fromID].balance >= amount) {
  a[fromID].balance -= amount;
  a[toID].balance += amount;
}
end_transaction();
```

TM'S END

`end_transaction()`

Check read set: is all your data still valid (no writes to any)?

- Yes: commit transaction by committing writes
- No: abort transaction by restoring checkpoint

Writes have to appear atomically with regard to other threads

```
struct account_t { int balance; };
shared struct account_t a[MAX];
int fromID, toID, amount;

begin_transaction();
if (a[fromID].balance >= amount) {
  a[fromID].balance -= amount;
  a[toID].balance += amount;
}
end_transaction();
```

TRANSACTIONAL EXECUTION (CONFLICT)

```
Thread 0

fromID = 241;
toID = 37;

begin_transaction();
if(a[241].balance > 100) {
  ...
  ...
  // detect write to a[241].balance
  // abort, rollback and retry
```

```
Thread 1

fromID = 37;
toID = 241;

begin_transaction();
if(a[37].balance > 100) {
  a[37].balance -= amount;
  a[241].balance += amount;
}
end_transaction();
// no writes to a[241].balance
// no writes to a[37].balance
// commit and proceed
```

TRANSACTIONAL EXECUTION (MORE LIKELY)

```
Thread 0

fromID = 241;
toID = 37;

begin_transaction();
if(a[241].balance > 100) {
 a[241].balance -= amount;
 a[37].balance += amount;
}
end_transaction();
// no write to a[241].balance
// no write to a[37].balance
// commit and proceed
```

```
Thread 1

fromID = 450;
toID = 118;

begin_transaction();
if(a[450].balance > 100) {
  a[450].balance -= amount;
  a[118].balance += amount;
}
end_transaction();
// no write to a[450].balance
// no write to a[118].balance
// commit and proceed
```

Critical sections execute completely in parallel, as there is no conflict detected.

IMPLEMENTATION DESIGN SPACE

Four main components

1. Version management

    Logging/buffering

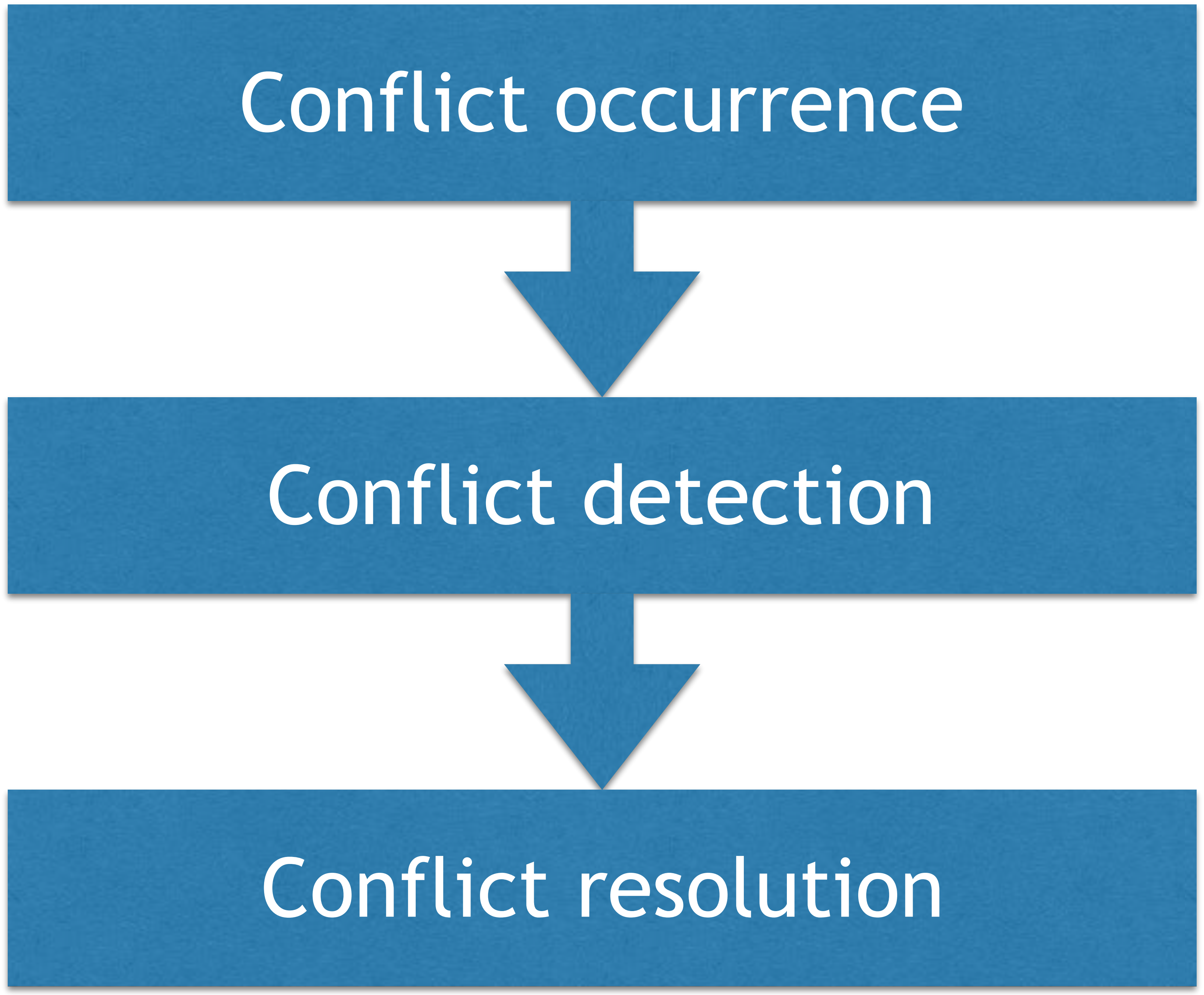
    Register and memory accesses

2. Conflict detection

    Two accesses to a single location, at least one is a write

3. Abort/Rollback

4. Commit



Various implementation approaches

    Hardware TM (HTM), Software TM (STM), Accelerated STM, Hybrids

VERSION MANAGEMENT: PRESERVING REGISTER VALUES

Begin transaction

- Take register checkpoint

Commit transaction

- Free register checkpoint

Abort transaction

- Restore register checkpoint

Register checkpoint includes all registers

- Including control registers like PC, stack pointers, etc.

VERSION MANAGEMENT FOR MEMORY

Lazy version management

Store

Put all writes into write table

Load

If address in write table, read value from write table

“Read own write early”

Otherwise, read from memory

Commit transaction

Write all entries from write table to memory, clear it

Abort transaction

Clear write table (fast)

Eager version management

Store

If address not in write set:
1. Read old value and put it into write log
2. Add address to write set

Write stores directly to memory

Load

Read directly from memory (fast)

Commit transaction

Nothing (fast)

Abort transaction

Traverse log, write logged values back into memory (slow)

CONFLICT DETECTION

Lazy conflict detection

Store

- Add address to write set (if not already present)

Load

- Add address to read set (if not already present)

Commit transaction

- For each address A in write set, and for each other thread T: if A is in T's read set, abort T's transaction

Eager conflict detection

Store

- Add address A to write set (if not already present)
- For each other thread T: if A is in T's write or read set, trigger conflict

Load

- Add address to read set (if not already present)
- For each other thread T: if A is in T's write set, trigger conflict

Conflict: abort either transaction

Commit transaction

- If not already aborted, commit and clear sets

SOFTWARE TRANSACTIONAL MEMORY (STM)

Add extra software to perform TM operations

Version management

- Software data structure for write table (log)
- Eager or lazy

Conflict detection

- Software data structure (lock table)
- Mostly lazy, too high overhead for eager implementations
- Granularity: object or block

Commit

- Need to ensure atomic update of all states
- Grabs lots of locks, or a global commit lock

Many possible implementations and semantics

- Various (disjoint) goals like low sequential overhead, or good scalability, or strong progress guarantees, …

HARDWARE TRANSACTIONAL MEMORY (HTM)

- Key idea: leverage invalidation-based coherence
  - Each cache block has read-only or read-write state
  - Coherence invariant:
    - Many read-only (shared) blocks, or
    - Single read-write block
- Add two bits per cache block: read & write
  - Set on load/store during transactional execution
  - If another processor steals block from cache, abort
    1. Read or write request to cache block with write bit set
    2. Write request to cache block with read bit set

- Low-overhead conflict detection
  - But only if all data fits into cache
  - Eviction: loosing information associated with transaction
- Implementations and semantics
  - ReadTx/WriteTx or atomic blocks (all memory accesses are implicitly transactional)
  - Automatic re-execution after aborts
  - Use of coherence or development of new memory systems for TM
  - Use of write-buffers?

HTM VS STM

HTM

Requires hardware (~~announced~~ ~~available~~ ~~cancelled~~ available)

Simple for bounded case

Unbounded TM in hardware really complicated

Size: tracking conflicts after cache overflow

Duration: context switching during transactions

Cache block granularity for conflicts

STM

Here today (prototype compilers from Intel and others)

Generally weaker semantics

Slow (2x or more single-thread overhead)

Adds lots of instructions to memory operations

HYBRID TRANSACTIONAL MEMORY

Hardware-accelerated STM

Add special hardware tracking features

Under control of software

Can reduce STM overhead, but perhaps not enough

Hybrid HTM/STM

Use HTM mode most of the time

Revert to STM only on overflows and such

Getting the interaction right is actually really tricky

SOME CAVEATS

Although transactional memory really looks promising...

What if: read or write set bigger than cache?

What if: transaction gets scheduled out in the middle?

What if: transaction wants to do IO or syscalls?

Concurrency issues (see databases)

- Between transactions & between TM ops
- Linearizability (atomicity) & serializability (basic correctness condition)

How do we translate lock-based programs into TM programs?

- Replacing `lock_acquire` with `begin_transaction` doesn’t always work

Several kinds of transaction semantics

- Are transactions atomic relative to code outside of transactions?

TRANSACTIONS ARE NOT CRITICAL SECTIONS

What‘s wrong with this code?

Thread 0	**Thread 1**
`begin_transaction();`	`begin_transaction();`
`flagA = true;`	`while (!flagA);`
`while (!flagB);`	`flagB = true;`
`//update m`	`//update n`
`end_transaction();`	`end_transaction();`

A less contrived example:

Thread 0	**Thread 1**
`begin_transaction();`	`begin_transaction();`
`...`	`...`
`queueA->enqueue(val1);`	`queueB->enqueue(val2);`
`while (queueB->empty());`	`while (queueA->empty());`
`//access queueB`	`//access queueA`
`...`	`...`
`end_transaction();`	`end_transaction();`

These are not necessarily programming errors...

MORE ISSUES: CONSISTENCY DURING TRANSACTIONS

Consistency during transactions not covered in DBs!

- Strict serializability provides an intuitive model for the execution of committed transactions – nothing about failures
- Nothing about the interactions of TM-/non-TM accesses

Inconsistent Reads

- Example: eager VM and lazy CD => zombie or “doomed” transactions
- User is responsible to do an “incremental validation”
- Validating n locations requires n memory accesses; if for each access (total m): O(n*m)

Thread 0

```
do {
  StartTx();
  int tmp_1 = ReadTx(&x);





  int tmp_2 = ReadTx(&y);
  while (tmp_1 != tmp_2) { }
} while (!CommitTx());
```

Thread 1

```


do {
  StartTx();
  WriteTx(&x, 10);
  WriteTx(&y, 10);
} while (!CommitTx());
```

LCD: For each address A in write set, and for each other thread T: if A is in T’s read set, abort T’s transaction

IMPLEMENTATIONS

STM

- Mainly two types
  - Fully validates: validate complete read set after each transactional read: more concurrency at a higher price
  - Global version number: avoid full validation by comparing version numbers at the end of the transaction (one of the best trade-offs)
- Overhead: 1.5-120x single-thread execution time
- Mostly dead respectively an extension to HTM => hybrid HTM

HTM

- For two decades studied in academia
- Now first commercial solutions available
- Intel Transactional Synchronization Extensions (TSX): XACQUIRE/XRELEASE & XBEGIN/XEND
- Intel‘s ~~Haswell~~ Broadwell, IBM‘s Blue Gene/Q, Oracle: future SPARC will include memory versioning, AMD: either goes its own way with ASF, or joins Intel‘s HLE/RTM

SUMMARY

Transactional Memory is a promising technique to address:

- Programming (synchronization) complexity
- Overhead associated with traditional locking

Caveats

- Transactional Memory != transactions in databases
- Transactions != critical sections
- Mixture of transactional and non-transactional traffic
- Pessimistic locking => probability of deadlocks
- Optimistic TM => probability of livelocks

SUMMARY: TM OVERVIEW

HTM: likely lazy VM, eager CD
STM: eager/lazy VM, likely lazy CD