Lecture 18:

Transactional Memory Part II + Course Wrap Up

Parallel Computing

Stanford CS149, Fall 2025

Transactional Memory (TM) Review

Memory transaction
- An atomic and isolated sequence of memory accesses
- Inspired by database transactions

Atomicity (all or nothing)
- Upon transaction commit, all memory writes in transaction take effect at once
- On transaction abort, none of the writes appear to take effect (as if transaction never happened)

Isolation
- No other processor can observe writes before transaction commits

Serializability
- Transactions appear to commit in a single serial order
- But the exact order of commits is not guaranteed by semantics of transaction

Advantages (promise) of transactional memory

Easy to use synchronization construct

- It is difficult for programmers to get synchronization right
- Programmer declares need for atomicity, system implements it well
- Claim: transactions are as easy to use as coarse-grain locks

Often performs as well as fine-grained locks

- Provides automatic read-read concurrency and fine-grained concurrency
- Performance portability: locking scheme for four CPUs may not be the best scheme for 64 CPUs
- Productivity argument for transactional memory: system support for transactions can achieve 90% of the benefit of expert programming with fined-grained locks, with 10% of the development time

Failure atomicity and recovery

- No lost locks when a thread fails
- Failure recovery = transaction abort + restart

Composability

- Safe and scalable composition of software modules

Implementing transactional memory

TM implementation basics

TM systems must provide atomicity and isolation

- While maintaining concurrency as much as possible

Two key implementation questions

- Data versioning policy: How does the system manage uncommitted (new) and previously committed (old) versions of data for concurrent transactions?

- Conflict detection policy: how/when does the system determine that two concurrent transactions conflict?

Data Versioning Policy

Manage uncommitted (new) and previously committed (old) versions of data for concurrent transactions

1. **Eager versioning (undo-log based)**
2. **Lazy versioning (write-buffer based)**

Conflict Detection

Must detect and handle conflicts between transactions

- Read-write conflict: transaction A reads address X, which was written to by pending (but not yet committed) transaction B
- Write-write conflict: transactions A and B are both pending, and both write to address X

System must track a transaction's read set and write set

- Read-set: addresses read during the transaction
- Write-set: addresses written during the transaction

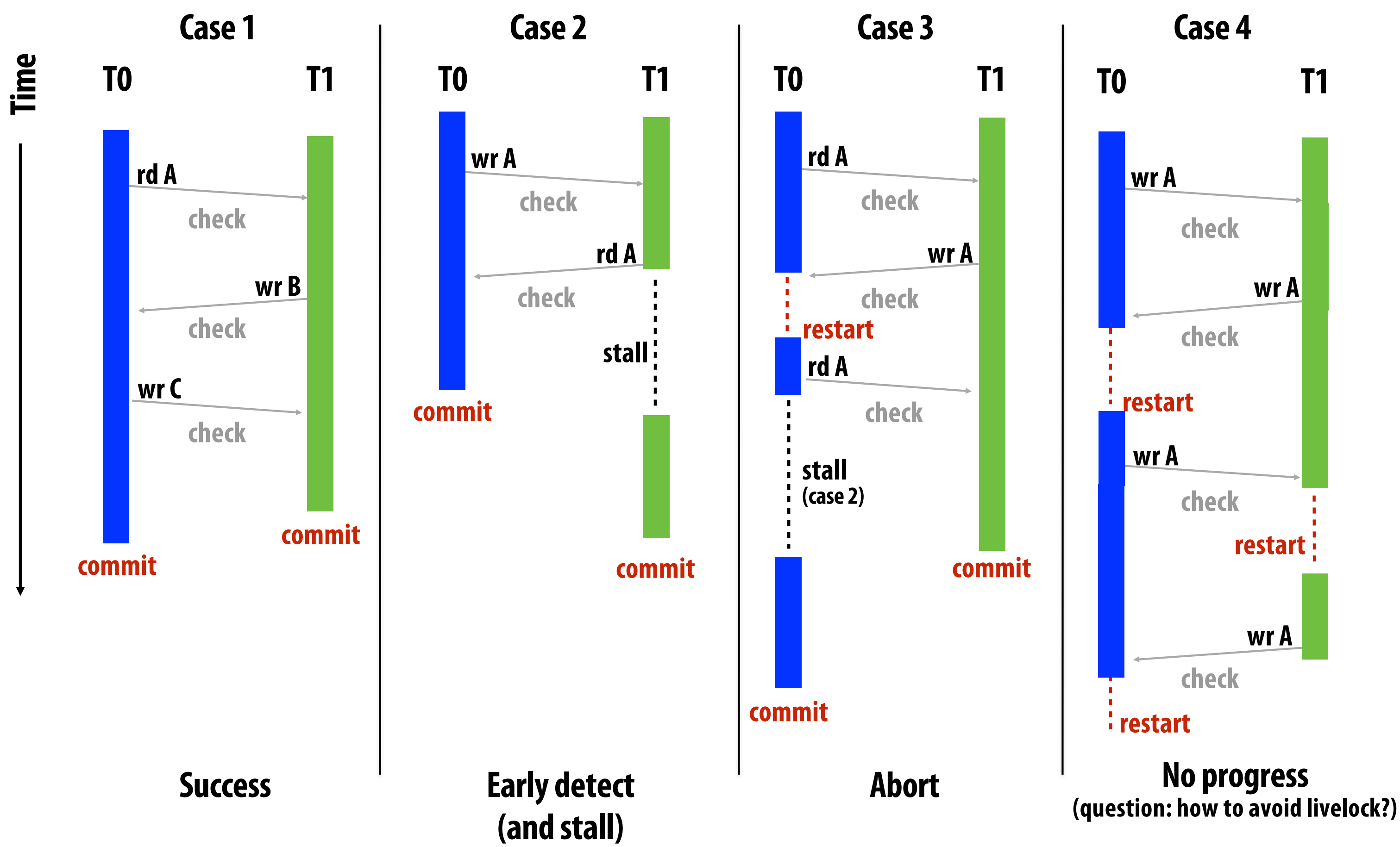
Pessimistic Detection

- Check for conflicts (immediately) during loads or stores
  - Philosophy: "I suspect conflicts might happen, so let's always check to see if one has occurred after each memory operation… if I'm going to have to roll back, might as well do it now to avoid wasted work."

- "Contention manager" decides to stall or abort transaction when a conflict is detected
  - Various policies to handle common case fast

Pessimistic Detection Examples

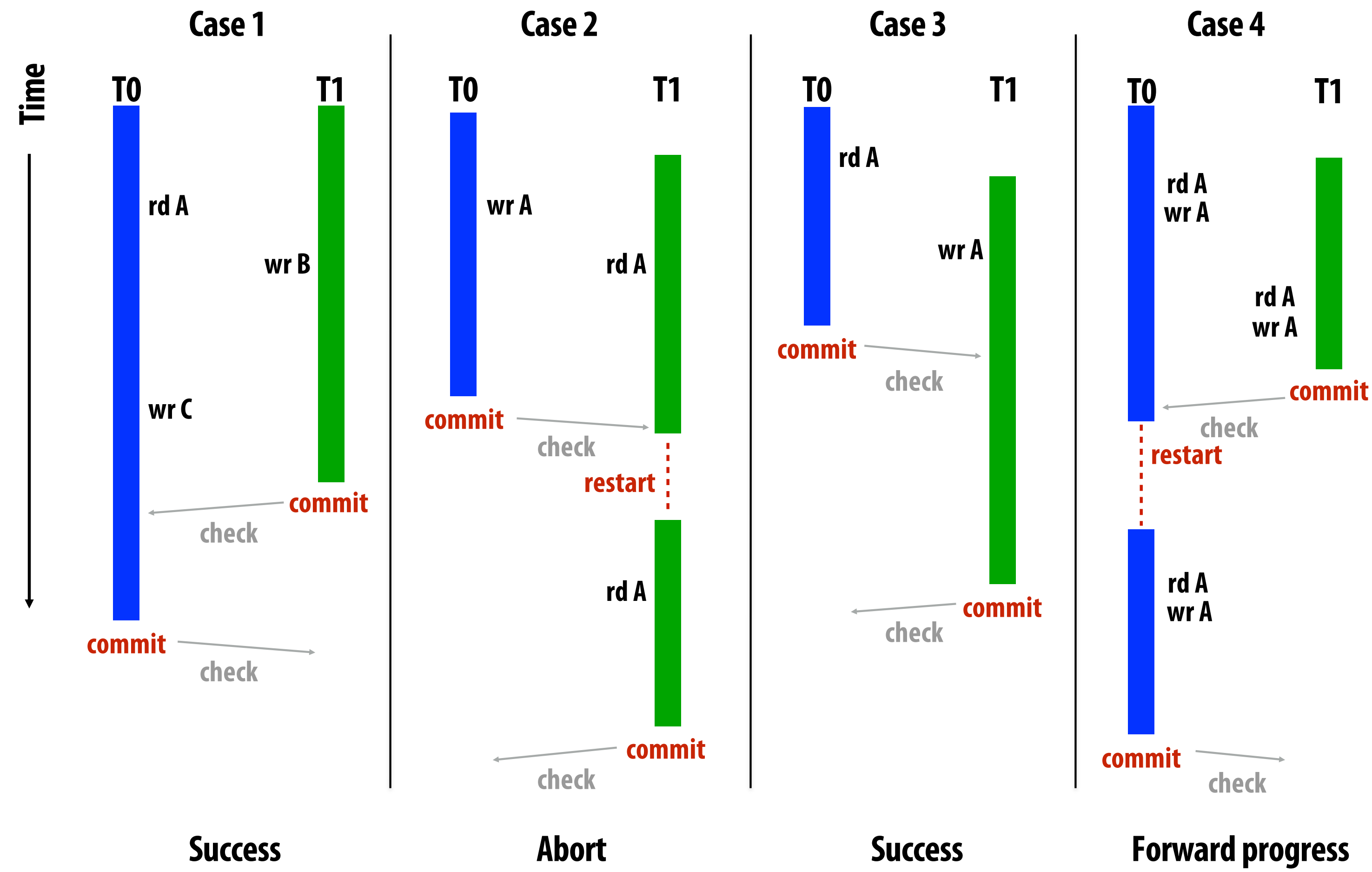
Note: diagrams assume "aggressive" contention manager on writes: writer wins, so other transactions abort)



Optimistic detection

- Detect conflicts when a transaction attempts to commit
  - Intuition: "Let's hope for the best and sort out all the conflicts only when the transaction tries to commit"
- On a conflict, give priority to committing transaction
  - Other transactions may abort later on

Optimistic Detection Examples



TM implementation space (examples)

Software TM systems

- Lazy + optimistic (rd/wr): Sun TL2
- Lazy + optimistic (rd)/pessimistic (wr): MS OSTM
- Eager + optimistic (rd)/pessimistic (wr): Intel STM
- Eager + pessimistic (rd/wr): Intel STM

Hardware TM systems

- Lazy + optimistic: Stanford TCC
- Lazy + pessimistic: MIT LTM, Intel VTM
- **Eager + pessimistic: Wisconsin LogTM** (easiest with conventional cache coherence)

Optimal design remains an open question

- May be different for HW, SW, and hybrid

Software Transactional Memory

```
atomic {
   a.x = t1
   a.y = t2
   if (a.z == 0) {
   a.x = 0
   a.z = t3
   }
}
```

➡

```
tmTxnBegin()
tmWr(&a.x, t1)
tmWr(&a.y, t2)
if (tmRd(&a.z) != 0) {
   tmWr(&a.x, 0);
   tmWr(&a.z, t3)
}
tmTxnCommit()
```

- Software barriers (STM function call) for TM bookkeeping
  - Versioning, read/write-set tracking, commit, . . .
  - Using locks, timestamps, data copying, . . .
- Requires function cloning or dynamic translation
  - Function used inside and outside of transaction

STM Runtime Data Structures

Transaction descriptor (per-thread)

- Used for conflict detection, commit, abort, …
- Includes the read set, write set, undo log or write buffer

Transaction record (per data)

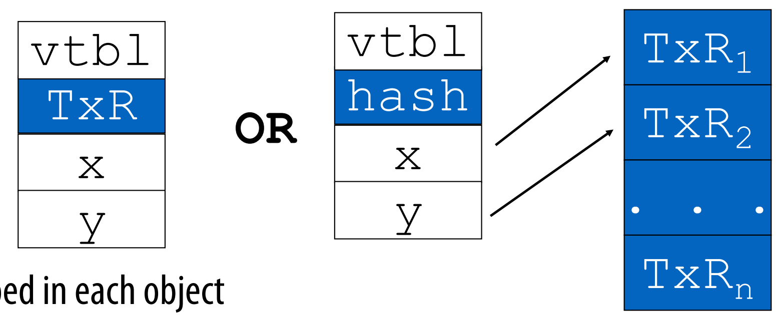
- Pointer-sized record guarding shared data
- Tracks transactional state of data
  - Shared: accessed by multiple readers
    - Using version number or shared reader lock
  - Exclusive: access by one writer
    - Using writer lock that points to owner
  - BTW: same way that HW cache coherence works

Mapping Data to Transaction Records

Every data item has an associated transaction record

Java/C#

```
class Foo {
    int x;
    int y;
}
```



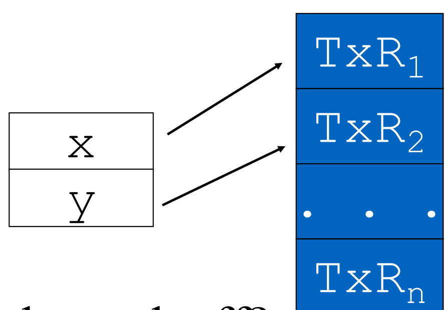
Embed in each object

Hash fields or array elements to global table

f(obj.hash, field.index)

C/C++

```
struct Foo {
    int x;
    int y;
}
```



Address-based hash into global table

Cache-line or word granularity

What's the tradeoff?

Conflict Detection Granularity

Object granularity

- Low overhead mapping operation
- Exposes optimization opportunities
- False conflicts (e.g. Txn 1 and Txn 2)

Element/field granularity (word)

- Reduces false conflicts
- Improves concurrency (e.g. Txn 1 and Txn 2)
- Increased overhead (time/space)

Cache line granularity (multiple words)

- Matches hardware TM
- Reduces storage overhead of transactional records
- Hard for programmer & compiler to analyze

Mix & match per type basis

- E.g., element-level for arrays, object-level for non-arrays

Txn 1

```
a.x = …
a.y = …
```

Txn 2

```
… = … a.z …
```

An Example STM Algorithm

Based on Intel's McRT STM [PPoPP'06, PLDI'06, CGO'07]

- Eager versioning, optimistic reads, pessimistic writes

Based on timestamp for version tracking

- Global timestamp
  - Incremented when a writing xaction commits
- Local timestamp per xaction
  - Global timestamp value when xaction last validated

Transaction record (32-bit)

- LS bit: 0 if writer-locked, 1 if not locked
- MS bits
  - Timestamp (version number) of last commit if not locked
  - Pointer to owner xaction if locked

STM Operations

STM read (optimistic)

- Direct read of memory location (eager)
- Validate read data
  - Check if unlocked and data version ≤ local timestamp
  - If not, validate all data in read set for consistency
- Insert in read set
- Return value

STM write (pessimistic)

- Validate data
  - Check if unlocked and data version ≤ local timestamp
- Acquire lock
- Insert in write set
- Create undo log entry
- Write data in place (eager)

STM Operations (cont)

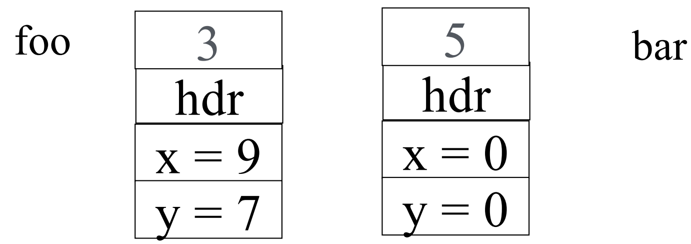
Read-set validation

- Get global timestamp
- For each item in the read set
  - If locked by other or data version > local timestamp, abort
- Set local timestamp to global timestamp from initial step

STM commit

- Atomically increment global timestamp by 2 (LSb used for write-lock)
- If preincremented (old) global timestamp > local timestamp, validate read-set
  - Check for recently committed transactions
- For each item in the write set
  - Release the lock and set version number to global timestamp

STM Example



X1

```
atomic {
 t = foo.x;
 bar.x = t;
 t = foo.y;
bar.y = t; }
```

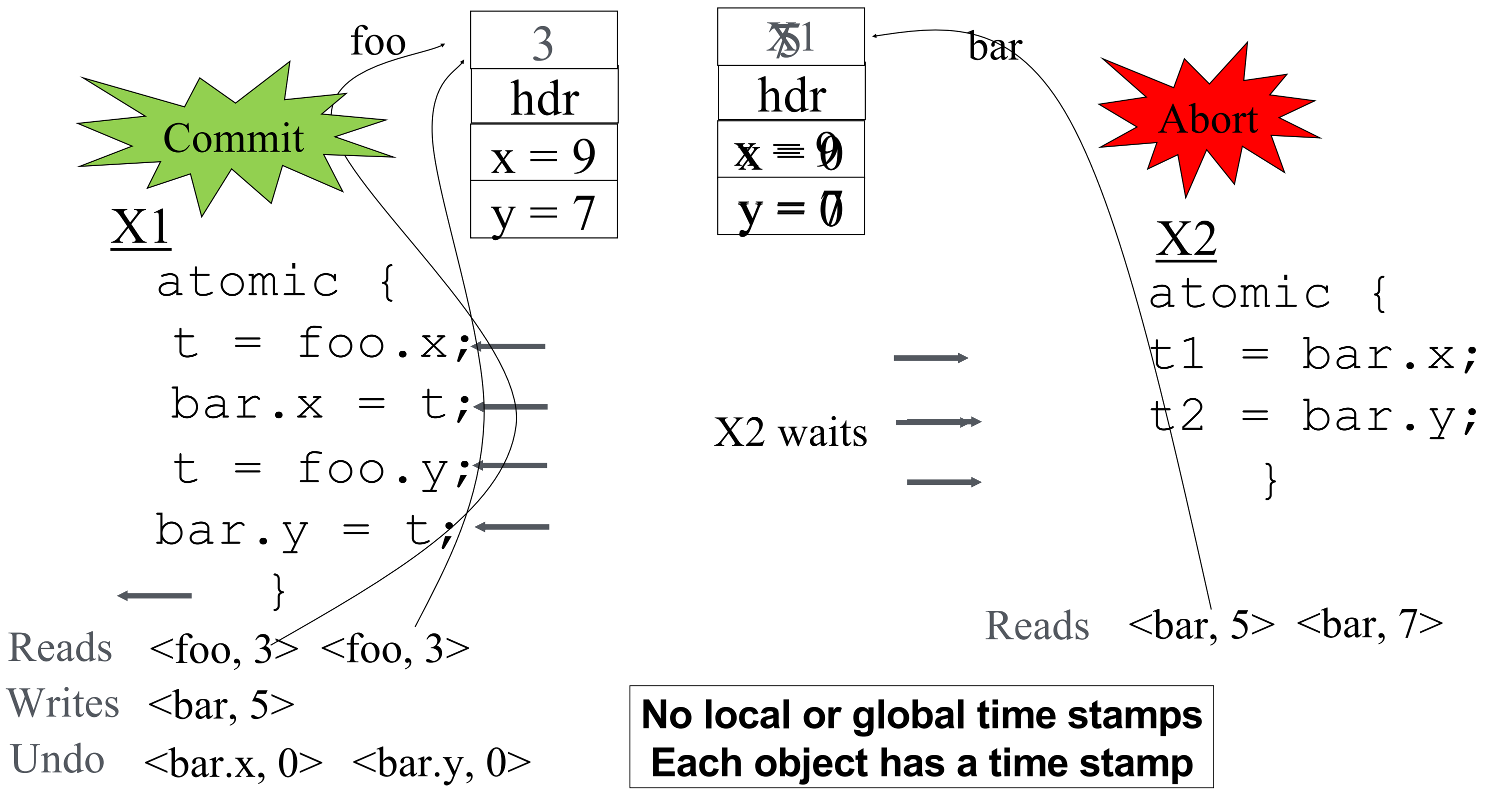
X2

```
atomic {
t1 = bar.x;
t2 = bar.y;
    }
```

X1 copies object foo into object bar

X2 should read bar as [0,0] or [9,7]

STM Example


foo
3
hdr
x = 9
y = 7
bar
hdr
Commit
X1
atomic {
t = foo.x;
bar.x = t;
t = foo.y;
bar.y = t;
}
Reads <foo, 3> <foo, 3>
Writes <bar, 5>
Undo <bar.x, 0> <bar.y, 0>
X2 waits
Abort
X2
atomic {
t1 = bar.x;
t2 = bar.y;
}
Reads <bar, 5> <bar, 7>
No local or global time stamps
Each object has a time stamp

TM Implementation Summary 1

TM implementation

- Data versioning: eager or lazy
- Conflict detection: optimistic or pessimistic
  - Granularity: object, word, cache-line, …

Software TM systems

- Compiler adds code for versioning & conflict detection
  - Note: STM barrier = instrumentation code
- Basic data-structures
  - Transactional descriptor per thread (status, rd/wr set, …)
  - Transactional record per data (locked/version)

Challenges for STM Systems

Overhead of software barriers

Function cloning

Robust contention management

Memory model (strong Vs. weak atomicity)

Optimizing Software Transactions

```
atomic {
   a.x = t1
   a.y = t2
   if (a.z == 0) {
   a.x = 0
   a.z = t3
   }
}
```

```
tmTxnBegin()
tmWr(&a.x, t1)
tmWr(&a.y, t2)
if (tmRd(&a.z) != 0) {
   tmWr(&a.x, 0);
   tmWr(&a.z, t3)
}
tmTxnCommit()
```

- Monolithic barriers hide redundant logging & locking from the compiler

Optimizing Software Transactions

```
atomic {
   a.x = t1
   a.y = t2
   if (a.z == 0) {
   a.x = 0
   a.z = t3
   }
}
```

```
txnOpenForWrite(a)
txnLogObjectInt(&a.x, a)
a.x = t1
txnOpenForWrite(a)
txnLogObjectInt(&a.y, a)
a.y = t2
txnOpenForRead(a)
if(a.z != 0) {
 txnOpenForWrite(a)
 txnLogObjectInt(&a.x, a)
 a.x = 0
 txnOpenForWrite(a)
 txnLogObjectInt(&a.z, a)
 a.z = t3
}
```

- Decomposed barriers expose redundancies

Optimizing Software Transactions

```
atomic {
   a.x = t1
   a.y = t2
   if (a.z == 0) {
   a.x = 0
   a.z = t3
   }
}
```

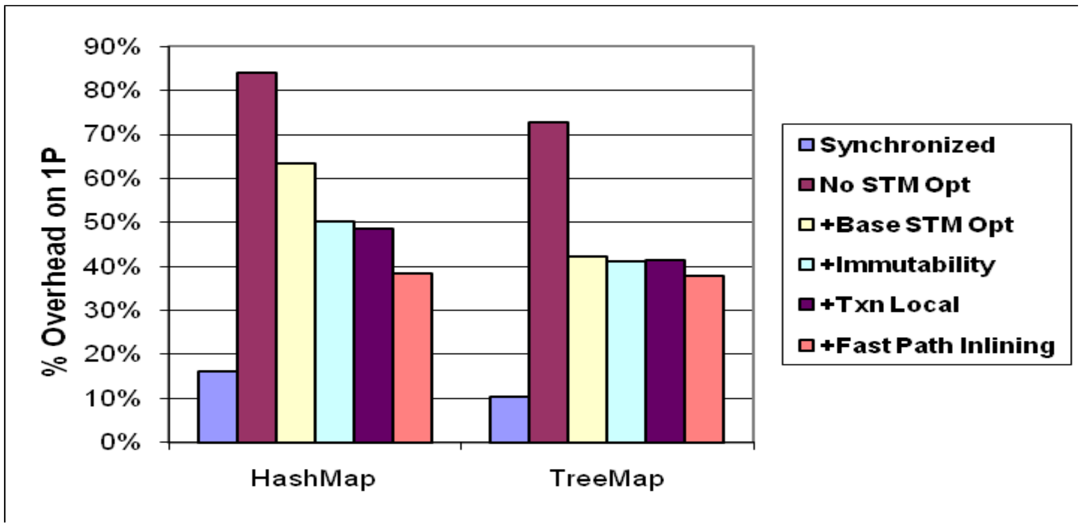
➡

```
txnOpenForWrite(a)
txnLogObjectInt(&a.x, a)
a.x = t1
txnLogObjectInt(&a.y, a)
a.y = t2
if (a.z != 0) {
   a.x = 0
   txnLogObjectInt(&a.z, a)
   a.z = t3
}
```

- Allows compiler to optimize STM code
- Produces fewer & cheaper STM operations

Effect of Compiler Optimizations

1 thread overheads over thread-unsafe baseline



With compiler optimizations

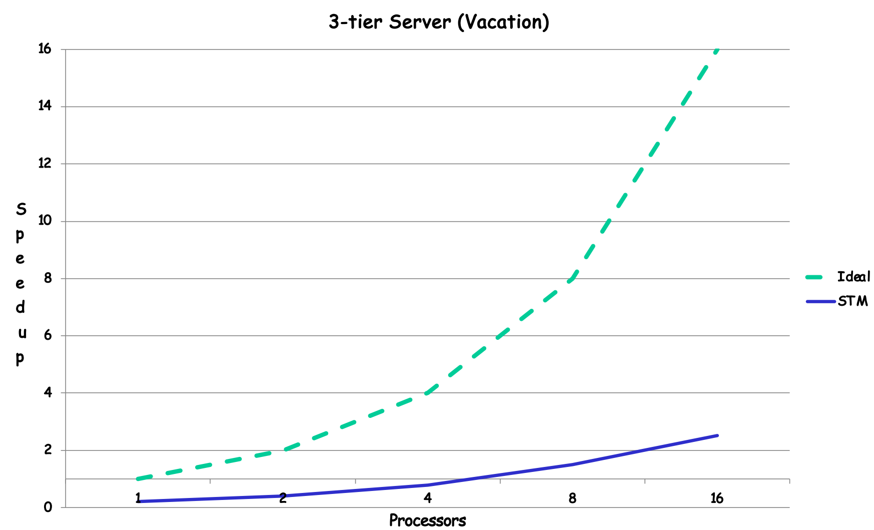
- **<40% over no concurrency control**
- **<30% over lock-based synchronization**

STM Question

- Given an optimistic read, pessimistic write, eager versioning STM
- What steps are required to implement the atomic region

```
atomic{
    obj.f1=42;
}
```

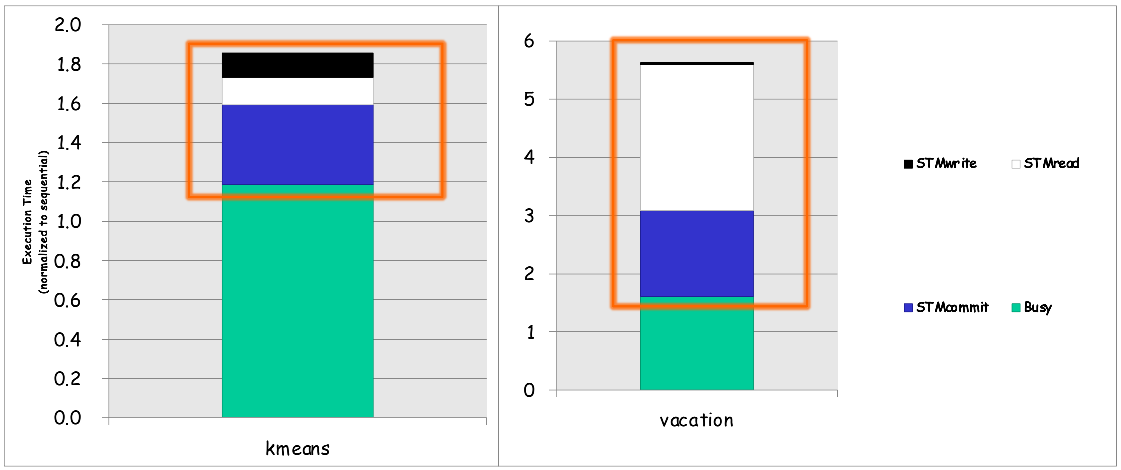
Motivation for Hardware Support



- STM slowdown: 2-8x per thread overhead due to barriers
  - Short term issue: demotivates parallel programming
  - Long term issue: energy wasteful
- Lack of strong atomicity
  - Costly to provide purely in software

Why is STM Slow?

Measured single-thread STM performance



1.8x – 5.6x slowdown over sequential

Most time goes in read barriers & commit

- Most apps read more data than they write

Types of Hardware Support

Hardware-accelerated STM systems (HASTM, SigTM, USTM, …)

- Start with an STM system & identify key bottlenecks
- Provide (simple) HW primitives for acceleration, but keep SW barriers

Hardware-based TM systems (TCC, LTM, VTM, LogTM, …)

- Versioning & conflict detection directly in HW
- No SW barriers

Hybrid TM systems (Sun Rock, …)

- Combine an HTM with an STM by switching modes when needed
  - Based on xaction characteristics available resources, …

	HTM	STM	HW-STM
Write versioning	HW	SW	SW
Conflict detection	HW	SW	HW

Hardware transactional memory (HTM)

Data versioning is implemented in caches
- Cache the write buffer or the undo log
- Add new cache line metadata to track transaction read set and write set

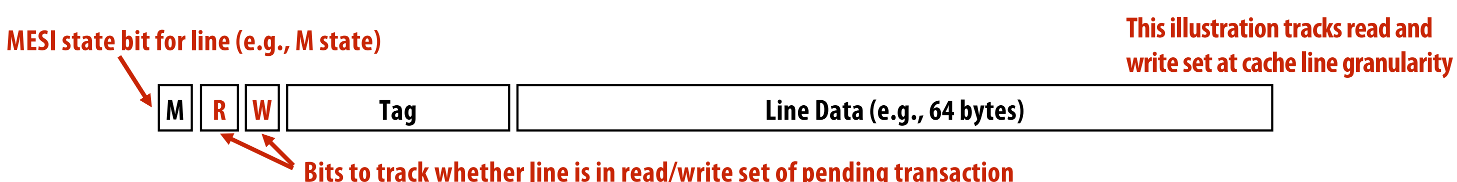
Conflict detection through cache coherence protocol
- Coherence lookups detect conflicts between transactions
- Works with snooping and directory coherence

Note:
- Register checkpoint must also be taken at transaction begin (to restore execution context state on abort)

HTM design

Cache lines annotated to track read set and write set

- R bit: indicates data read by transaction (set on loads)
- W bit: indicates data written by transaction (set on stores)
  - R/W bits can be at word or cache-line granularity
- R/W bits gang-cleared on transaction commit or abort

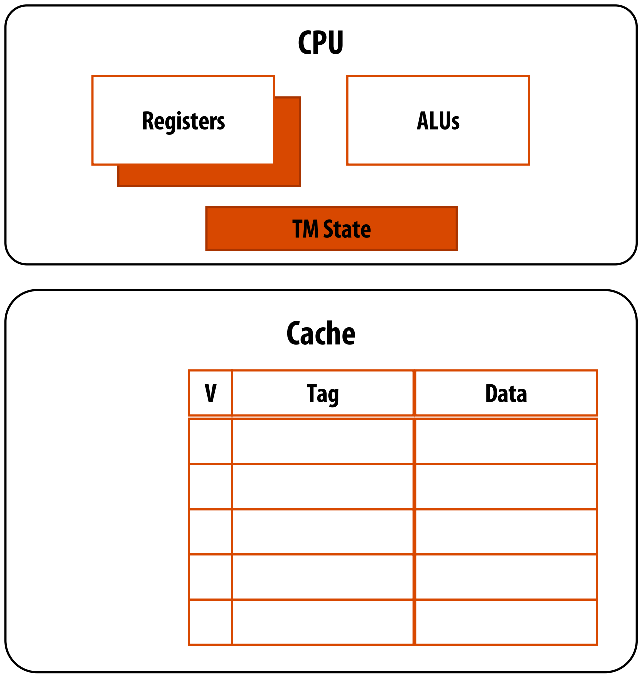


- For eager versioning, need a 2nd cache write for undo log

Coherence requests check R/W bits to detect conflicts

- Observing shared request to W-word is a read-write conflict
- Observing exclusive (intent to write) request to R-word is a write-read conflict
- Observing exclusive (intent to write) request to W-word is a write-write conflict

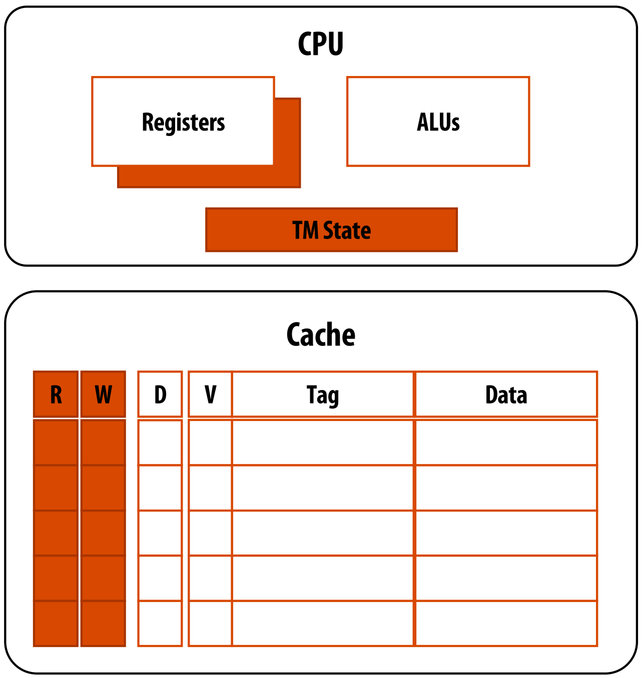
Example HTM implementation: lazy-optimistic



CPU changes

- Ability to checkpoint register state (available in many CPUs)
- TM state registers (status, pointers to abort handlers, ...)

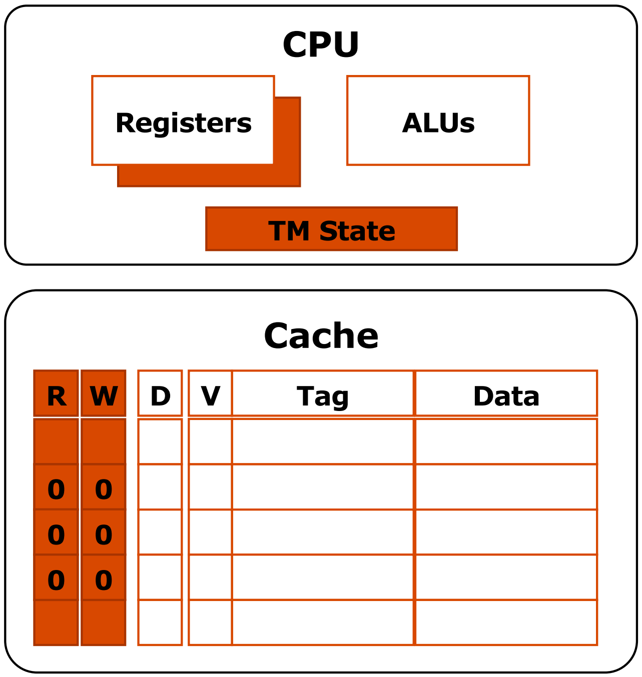
Example HTM implementation: lazy-optimistic



Cache changes

- R bit indicates membership to read set
- W bit indicates membership to write set

HTM transaction execution

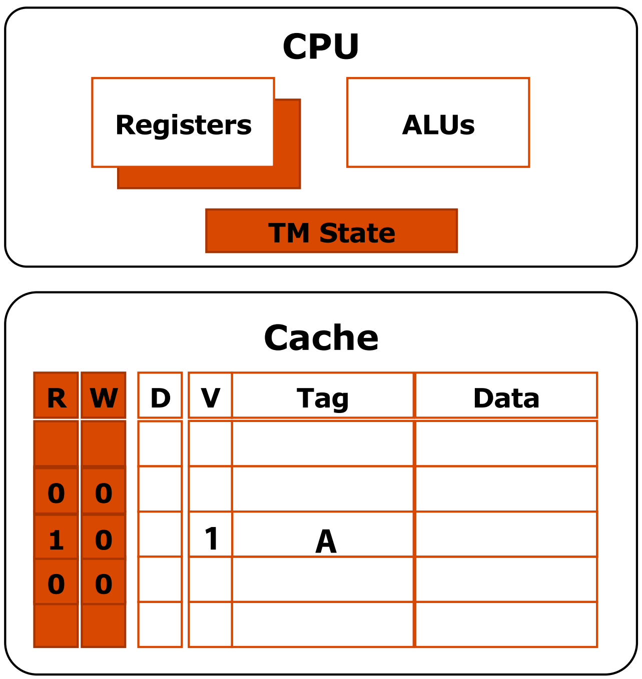



```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

Transaction begin

- Initialize CPU and cache state
- Take register checkpoint

HTM transaction execution

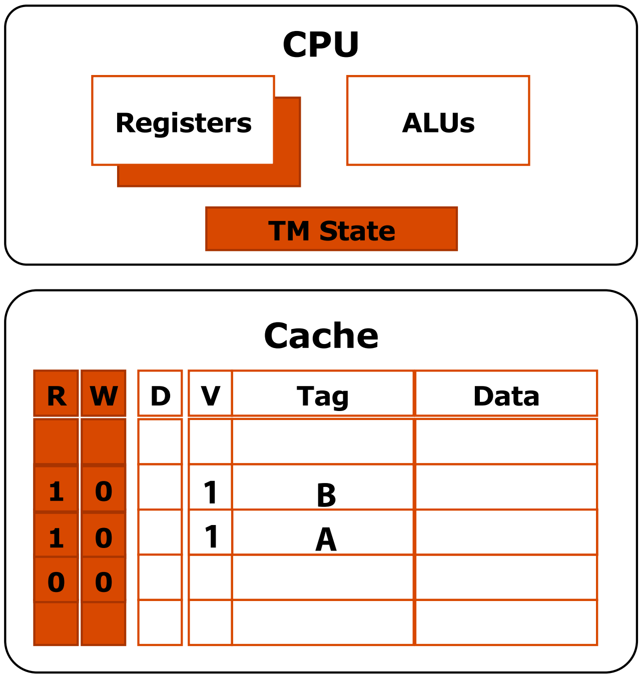



```
Xbegin
    Load A  ←
    Load B
    Store C ⇐ 5
Xcommit
```

Load operation

- Serve cache miss if needed
- Mark data as part of read set

HTM transaction execution

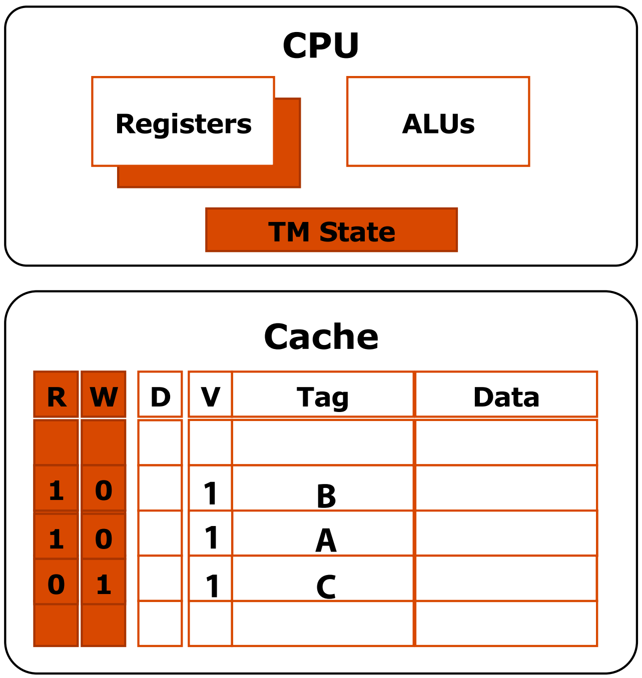



```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

Load operation

- Serve cache miss if needed
- Mark data as part of read set

HTM transaction execution

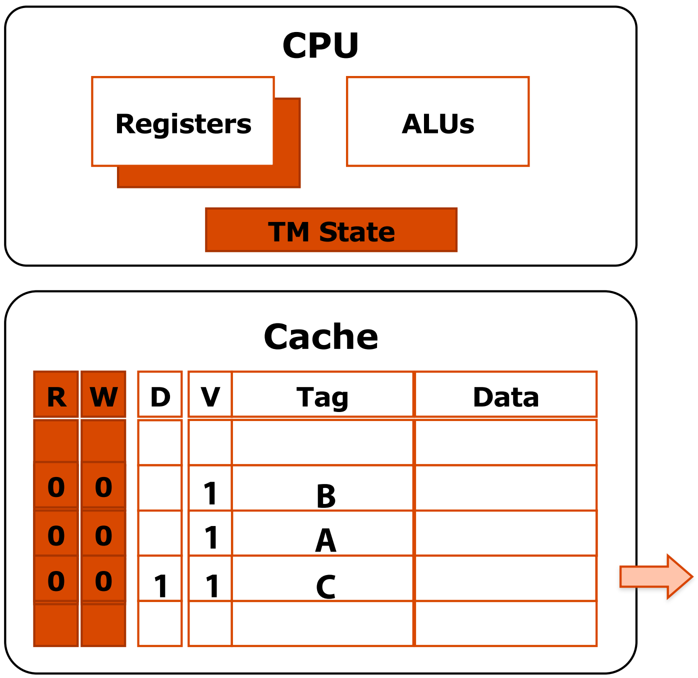



```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

Store operation

- Service cache miss if needed
- Mark data as part of write set (note: this is not a load into exclusive state. Why?)

HTM transaction execution: commit




```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

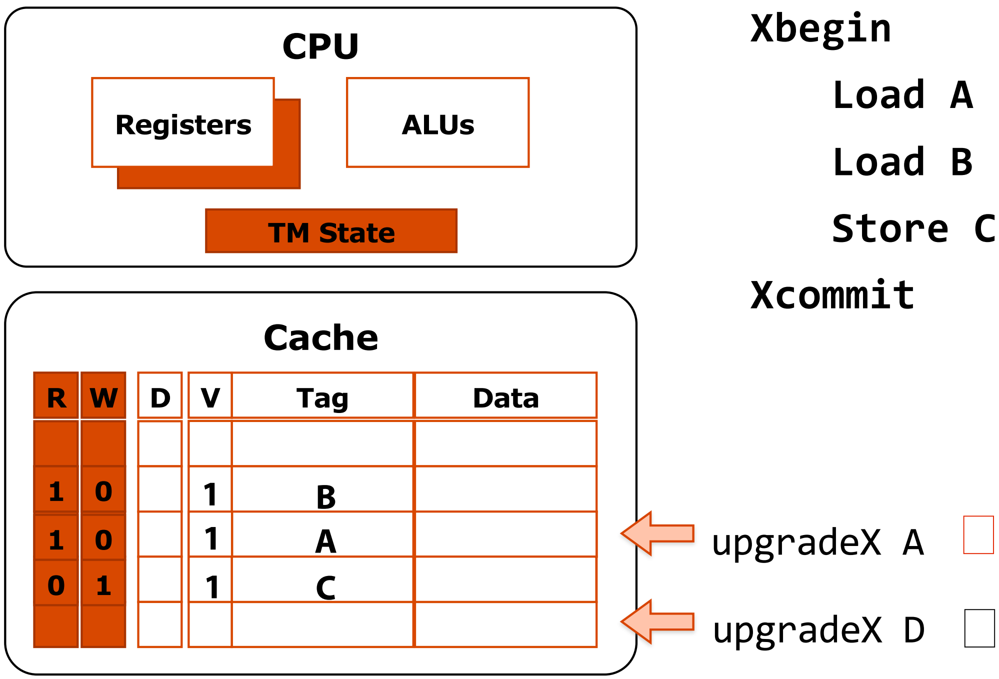
upgradeX C
(result: C is now in dirty state)

Fast two-phase commit

- Validate: request RdX access to write set lines (if needed)
- Commit: gang-reset R and W bits, turns write set data to valid (dirty) data

HTM transaction execution: detect/abort

Assume remote processor commits transaction with writes to A and D




```
Xbegin
    Load A
    Load B
    Store C ⇐ 5
Xcommit
```

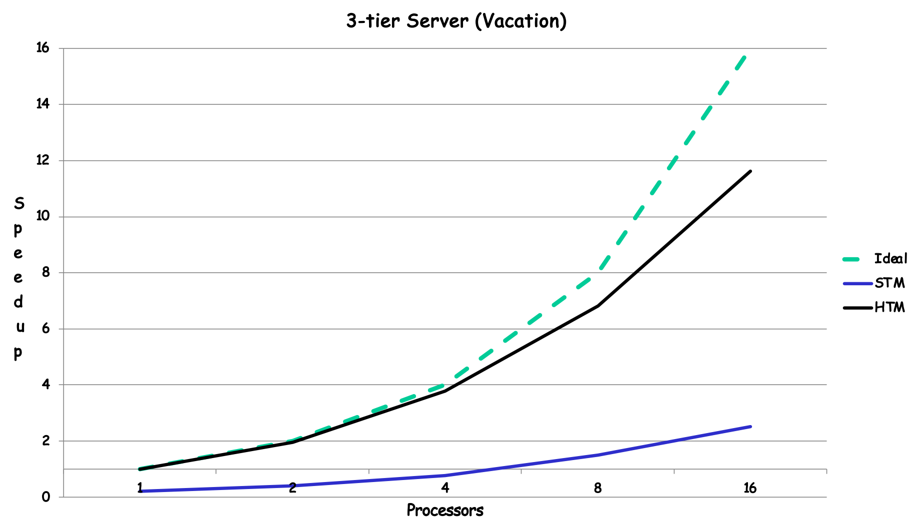
coherence requests from another core's commit

(remote core's write of A conflicts with local read of A: triggers abort of pending local transaction)

Fast conflict detection and abort

- Check: lookup exclusive requests in the read set and write set
- Abort: invalidate write set, gang-reset R and W bits, restore to register checkpoint

HTM Performance Example



- 2x to 7x over STM performance
  - Within 10% of sequential for one thread
  - Scales efficiently with number of processors

Review: Transactional Memory

- Atomic construct: declaration that atomic behavior must be preserved by the system
  - Motivating idea: increase simplicity of synchronization without (significantly) sacrificing performance
- Transactional memory implementation
  - Many variants have been proposed: SW, HW, SW+HW
  - Implementations differ in:
    - Data versioning policy (eager vs. lazy)
    - Conflict detection policy (pessimistic vs. optimistic)
    - Detection granularity (object, word, cache line)
- Software TM systems (STM)
  - Compiler adds code for versioning & conflict detection
    - Note: STM barrier = instrumentation code (e.g. StmRead, StmWrite)
  - Basic data-structures
    - Transactional descriptor per thread (status, rd/wr set, …)
    - Transactional record per data (locked/version)
- Hardware Transactional Memory (HTM)
  - Versioned data is kept in caches
  - Conflict detection mechanisms augment coherence protocol

HTM Example: Transactional Coherence and Consistency

Use TM as the coherence mechanism → all transactions all the time

Successful transaction commits update memory and all caches in the system

P1	P2	P3
Begin T1	Begin T2	Begin T4
Read A	Read A	Read E
Write A, 1	Write E, 3	Write B, 6
Write C, 2	Commit T2	Write C, 7
Read D	Begin T3	Read F
Commit T1	Write C, 4	Commit T4
	Read A	
	Write E, 5	
	Commit T3	

Assumptions

- Lazy and optimistic
- One "commit" per execution step across all processors
- When one transaction causes another transaction to abort and re-execute, assume that the transaction "commit" of one transaction can overlap with the "begin" of the re-executing transaction
- Minimize the number of execution steps

HTM Example: Transactional Coherence and Consistency

P1	P2	P3
Begin T1	Begin T2	Begin T4
Read A	Read A	Read E
Write A, 1	Write E, 3	Write B, 6
Write C, 2	Commit T2	Write C, 7
Read D	Begin T3	Read F
Commit T1	Write C, 4	Commit T4
	Read A	
	Write E, 5	
	Commit T3	

P1			P2			P3		
Action	Read set	Write set	Action	Read set	Write set	Action	Read set	Write set
B T1			B T2			B T4		
R A	A:0		R A	A:0		R E	E:0	
W A, 1	A:0	A:1	W E	A:0	E:3	W B, 6	E:0	B:6
W C, 2	A:0	A:1,C:2	C T2	A:0	E:3	B T4		

HTM Example: Transactional Coherence and Consistency

P1	P2	P3
Begin T1	Begin T2	Begin T4
Read A	Read A	Read E
Write A, 1	Write E, 3	Write B, 6
Write C, 2	Commit T2	Write C, 7
Read D	Begin T3	Read F
Commit T1	Write C, 4	Commit T4
	Read A	
	Write E, 5	
	Commit T3	

P1			P2			P3		
Action	Read set	Write set	Action	Read set	Write set	Action	Read set	Write set
B T1			B T2			B T4		
R A	A:0		R A	A:0		R E	E:0	
W A, 1	A:0	A:1	W E	A:0	E:3	W B, 6	E:0	B:6
W C, 2	A:0	A:1,C:2	C T2	A:0	E:3	B T4		
R D	A:0,D:0	A:1,C:2	B T3			R E	E:3	
C T1	A:0,D:0	A:1,C:2	W C, 5		C:5	W B, 6	E:3	B:6

HTM Example: Transactional Coherence and Consistency

P1	P2	P3
Begin T1	Begin T2	Begin T4
Read A	Read A	Read E
Write A, 1	Write E, 3	Write B, 6
Write C, 2	Commit T2	Write C, 7
Read D	Begin T3	Read F
Commit T1	Write C, 4	Commit T4
	Read A	
	Write E, 5	
	Commit T3	

P1			P2			P3		
Action	Read set	Write set	Action	Read set	Write set	Action	Read set	Write set
B T1			B T2			B T4		
R A	A:0		R A	A:0		R E	E:0	
W A, 1	A:0	A:1	W E	A:0	E:3	W B, 6	E:0	B:6
W C, 2	A:0	A:1,C:2	C T2	A:0	E:3	B T4		
R D	A:0,D:0	A:1,C:2	B T3			R E	E:3	
C T1	A:0,D:0	A:1,C:2	W C, 5		C:4	W B, 6	E:3	B:6
			R A	A:1	C:5	W C, 7	E:3	B:6,C:7
			W E, 6	A:1	C:5,E:6	R F	E:3,F:0	B:6,C:7
				A:1	C:5,E:6	C T4	E:3,F:0	B:6,C:7

HTM Example: Transactional Coherence and Consistency

P1	P2	P3
Begin T1	Begin T2	Begin T4
Read A	Read A	Read E
Write A, 1	Write E, 3	Write B, 6
Write C, 2	Commit T2	Write C, 7
Read D	Begin T3	Read F
Commit T1	Write C, 4	Commit T4
	Read A	
	Write E, 5	
	Commit T3	

P1			P2			P3		
Action	Read set	Write set	Action	Read set	Write set	Action	Read set	Write set
B T1			B T2			B T4		
R A	A:0		R A	A:0		R E	E:0	
W A, 1	A:0	A:1	W E	A:0	E:3	W B, 6	E:0	B:6
W C, 2	A:0	A:1,C:2	C T2	A:0	E:3	B T4		
R D	A:0,D:0	A:1,C:2	B T3			R E	E:3	
C T1	A:0,D:0	A:1,C:2	W C, 5		C:5	W B, 6	E:3	B:6
			R A	A:1	C:5	W C, 7	E:3	B:6,C:7
			W E, 6	A:1	C:5,E:6	R F	E:3,F:0	B:6,C:7
				A:1	C:5,E:6	C T4	E:3,F:0	B:6,C:7
			C T3	A:1	C:5,E:6			

Hardware transactional memory support in Intel Haswell architecture

- New instructions for "restricted transactional memory" (RTM)
  - xbegin: takes pointer to "fallback address" in case of abort
    - e.g., fallback to code-path with a spin-lock
  - xend
  - xabort

  - Implementation: tracks read and write set in L1 cache
- Processor makes sure all memory operations commit atomically
  - But processor may automatically abort transaction for many reasons (e.g., eviction of line in read or write set will cause a transaction abort)
    - Implementation does not guarantee progress (see fallback address)
  - Intel optimization guide (ch 12) gives guidelines for increasing probability that transactions will not abort

Summary: transactional memory

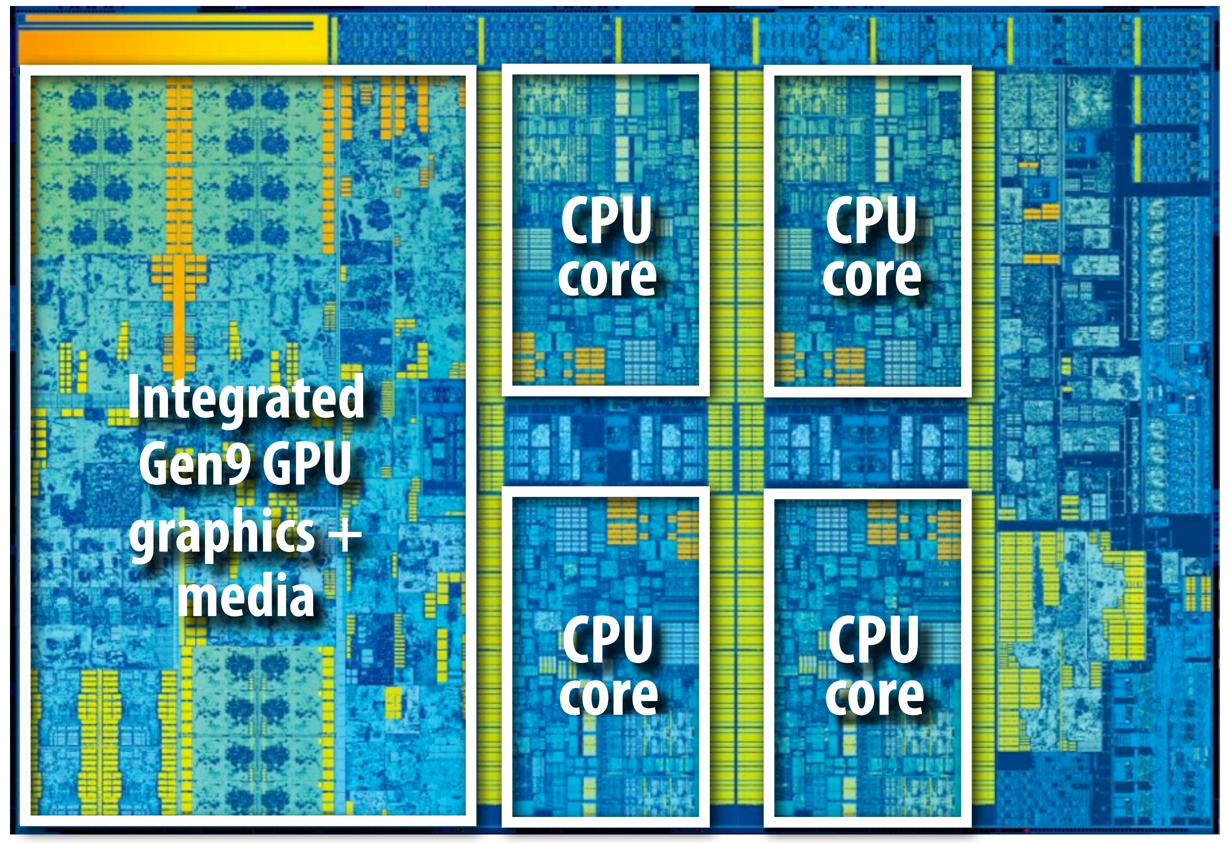
- Atomic construct: declaration that atomic behavior must be preserved by the system
  - Motivating idea: increase simplicity of synchronization without (significantly) sacrificing performance
- Transactional memory implementation
  - Many variants have been proposed: SW, HW, SW+HW
  - Implementations differ in:
    - Versioning policy (eager vs. lazy)
    - Conflict detection policy (pessimistic vs. optimistic)
    - Detection granularity (object, word, cache line)
- Software TM systems
  - Compiler adds code for versioning & conflict detection
    - Note: STM barrier = instrumentation code
  - Basic data-structures
    - Transactional descriptor per thread (status, rd/wr set, …)
    - Transactional record per data (locked/version)
- Hardware transactional memory
  - Versioned data is kept in caches
  - Conflict detection mechanisms built upon coherence protocol

Course Wrap Up

😎 😂 💃 🕺

(Students)

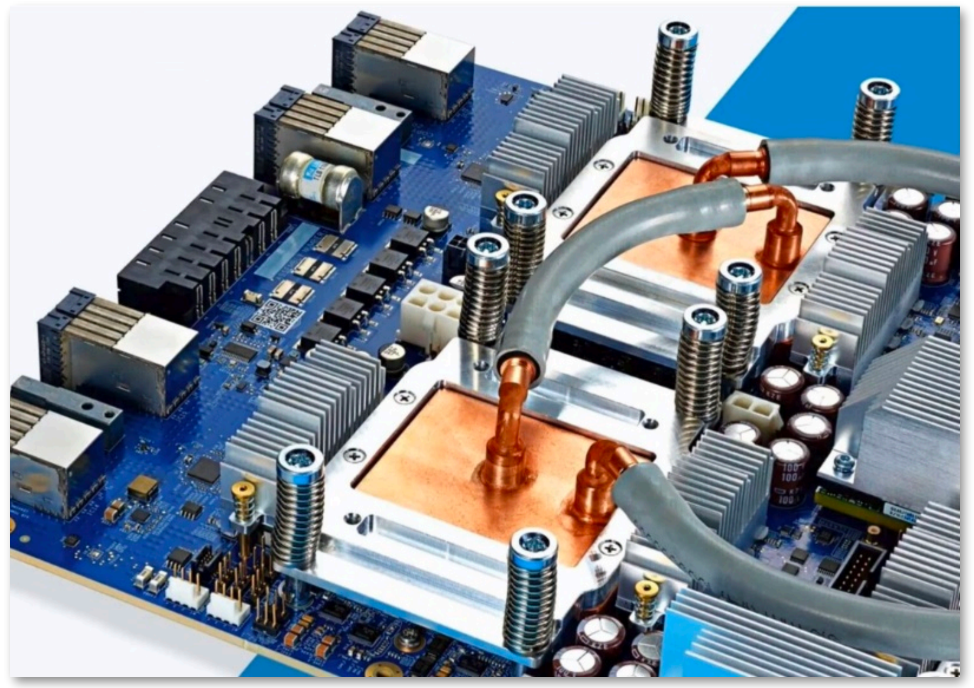
For the foreseeable future, the primary way to obtain higher performance computing hardware is through a combination of increased parallelism and hardware specialization.



Intel Core i7 CPU + integrated GPU and media

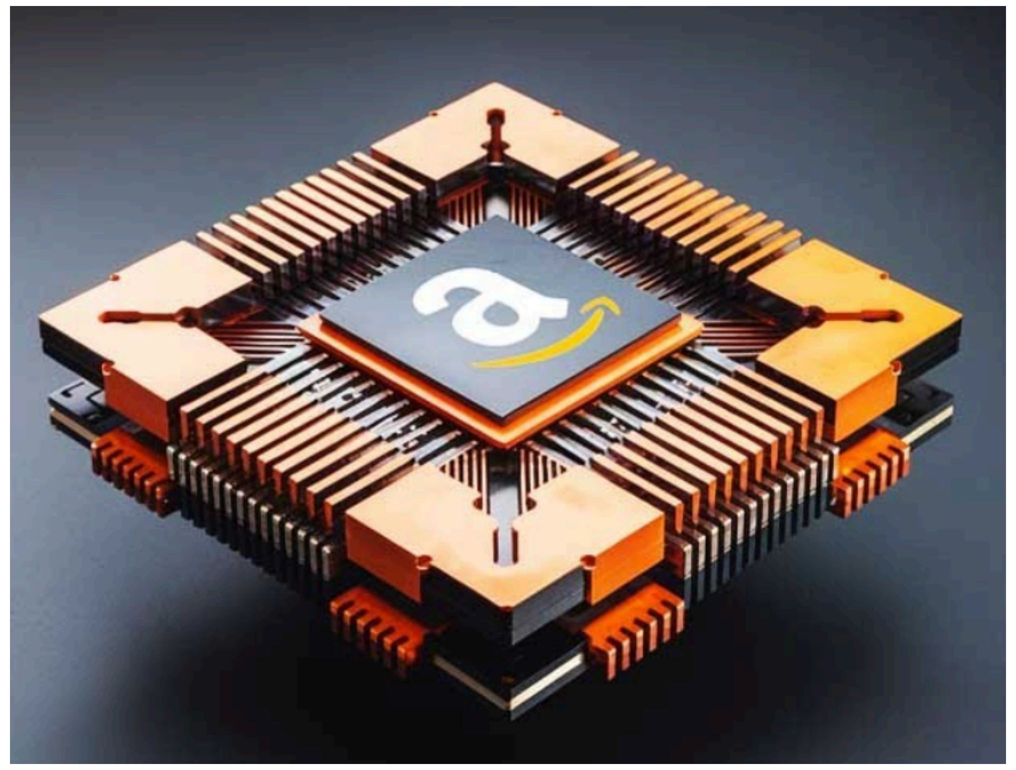
NVIDIA GPU
(single SMM core)
32 wide SIMD
2048 CUDA/core threads per SMM
Tensor Cores

FPGA
(reconfigurable logic)



Google TPU
AI Accelerator

Apple A11
Heterogeneous SoC
multi-core CPU + multi-core GPU + media ASICs + AI basics



AWS Trainium
AI Accelerator

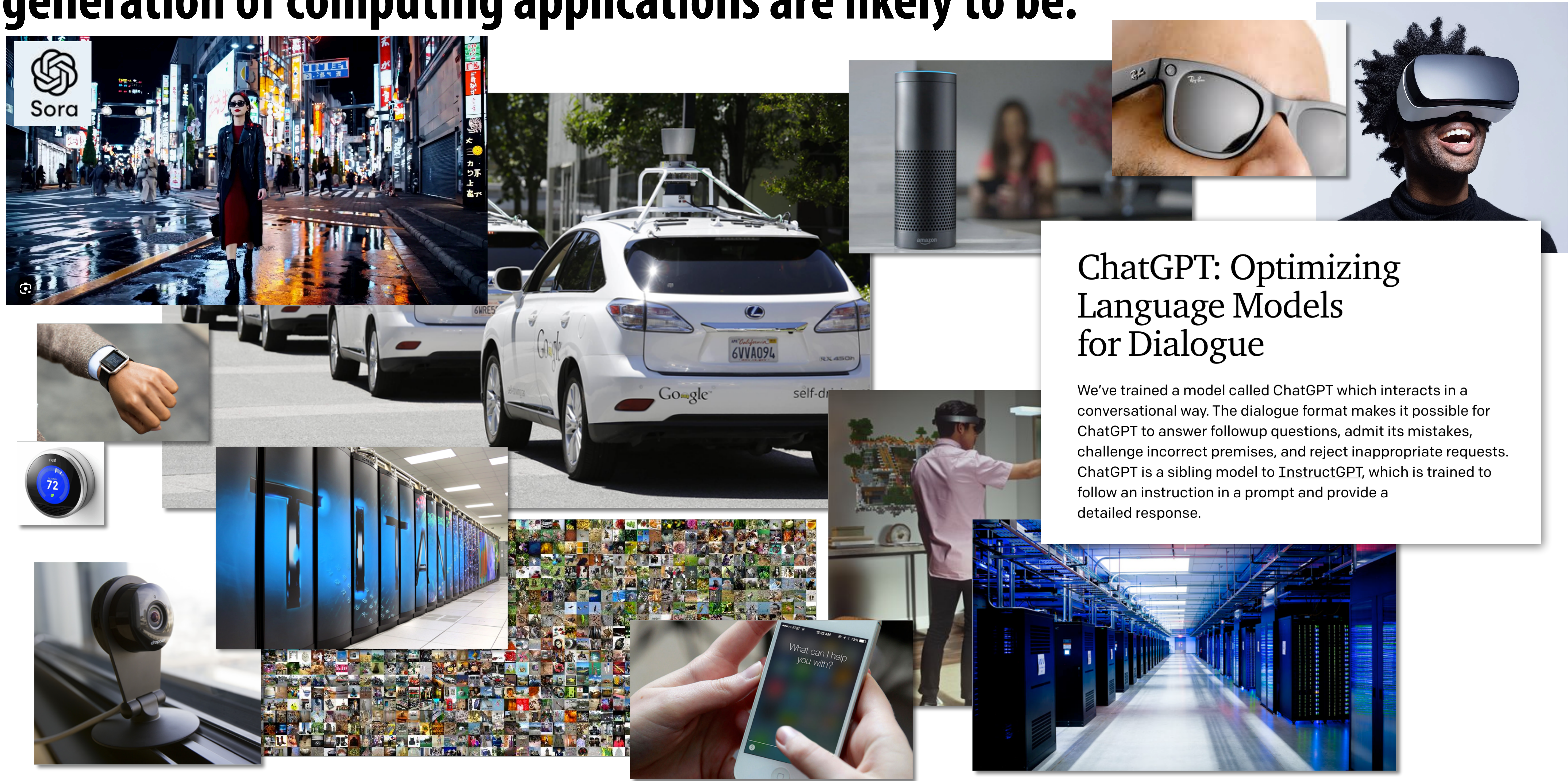
Modern software is surprisingly inefficient compared to the peak capability of modern machines

A lot of performance is currently left on the table (increasingly so as machines get more complex, and parallel processing capability grows)

Extracting this performance stands to provide a notable impact on many compute-intensive fields
(or, more importantly enable new applications of computing!)

Given current software programming systems and tools, understanding principles of how a parallel machine works is important to achieving high performance.

This is very important given how exciting (and efficiency-critical) the next generation of computing applications are likely to be.



Key issues we have addressed in this course

Identifying parallelism

(or conversely, identifying dependencies)

Efficiently scheduling work

1. Achieving good workload balance

2. Overcoming communication constraints:

Bandwidth limits, dealing with latency, synchronization

Exploiting data/computation locality = <u>efficiently managing state</u>!

We discussed these issues at many scales and in many contexts

Heterogeneous mobile SoC
Single chip, multi-core CPU
Multi-core GPU
CPU+GPU
Clusters of machines
AI accelerator hardware

Key issues we have addressed in this course

How throughput-oriented hardware works

Multiple cores, hardware-threads, SIMD

Specialized AI accelerators

Abstractions that help structure code to be efficient

Data parallel thinking

Functional parallelism

Transactions

Tasks

SPMD

Next steps

Other relevant classes

CS 217: Hardware Accelerators for Machine Learning

(Winter, Kunle's course)

CS/EE 282: Computer Systems Architecture

CS 348K: Visual Computing Systems

(Spring, Kayvon)

Design of high-performance hardware/software systems for processing images and video (ray tracing, video analysis, smartphone camera processing, NeRF/AI-based graphics, fast data labeling, etc)

After taking this course, you might be able to play a role in ongoing Stanford research in parallel computing!

Come talk to us!

HOW TO GET STARTED:

Common scenario:

Student: ***"I'm really interested in parallel systems research, is there anything I could do in your lab?"***

Kayvon's reaction: ***will this be a good fit for the student?***

To do well in a research lab setting, the student must be both highly motivated, have some background in the area to help them succeed, and be willing to put real time into it for the quarter.

- **Have they taken CS149?**
- **Are there examples of them going beyond expectations on programming projects in CS149?**
- **Have they worked on anything of the sort before in related classes or prior internships?**

Example ongoing projects in my lab

- Algorithmic techniques to boost the efficiency of LLM agents without model fine-tuning (agile agent optimization and development)
- Compiler efforts to design abstractions that make it easier for the programmer to experiment with scheduling parallel work onto threads, thread blocks, and warps
- High-performance (1M fps) world simulation engines designed to maximize throughput when training AI agents
- A high performance virtual human athlete simulator
- Using AI agents as "play testers" to help humans design games/virtual environments that meet certain goals of play (level of difficulty, fairness, etc.)
- The CS149 assistant agent (a virtual CA for CS149)

In the time remaining…
Ask us anything!