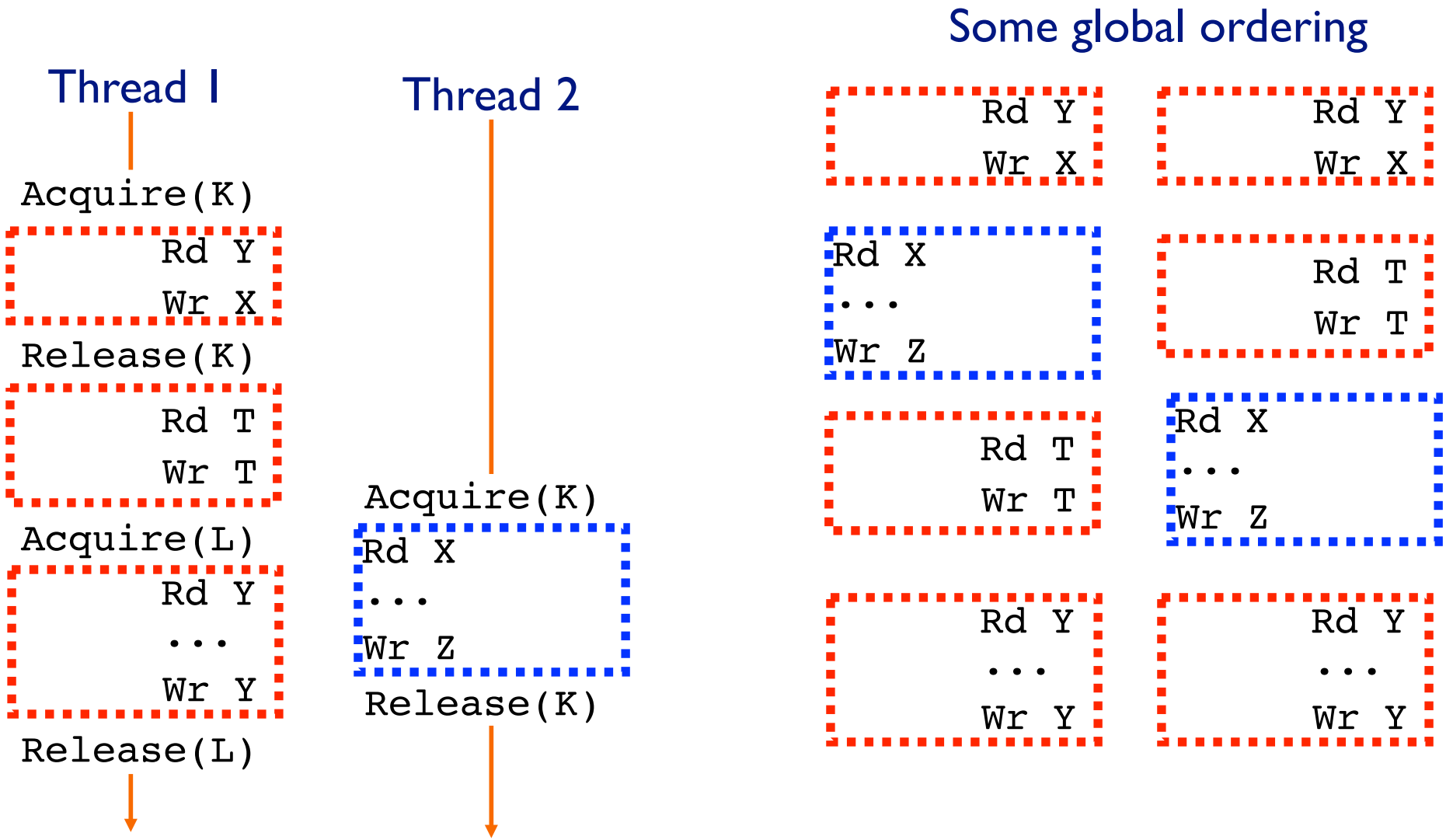
CSEP 548: Computer Systems Architecture

Transactional Memory

Luis Ceze, Spring 2017

(based on slides lifted from friends at UPenn, UIUC, UW, MIT, CMU)

Sequential Consistency for DRF Example



Implementing a Lock

- Shared counter/sum update example
  - Use a mutex variable for mutual exclusion
  - Only one processor can own the mutex
    - Many processors may call lock(), but only one will succeed (others block)
    - The winner updates the shared sum, then calls unlock() to release the mutex
    - Now one of the others gets it, etc.
  - But how do we implement a mutex?
    - As a shared variable (1 – owned, 0 –free)
- *How would you implement it?*

```
1. while (lock_var != 0);
2. lock_var = 1;
```

Locking

- Releasing a mutex is easy
  - Just set it to 0
- Acquiring a mutex is not so easy
  - Easy to spin waiting for it to become 0
  - But when it does, others will see it, too
  - What invariant do we need?

```
1. while (lock_var != 0);
2. lock_var = 1;
```

Thread 1	Thread 2
Line1: lock_var == 0	
… descheduled …	Line 1: lock_var == 0
	Line 2: Sets lock_var = 1 (Thinks it has the lock.)
Line 2: Sets lock_var = 1 (Thinks it has the lock)	… descheduled …

Locking

- Releasing a mutex is easy
  - Just set it to 0
- Acquiring a mutex is not so easy
  - Easy to spin waiting for it to become 0
  - But when it does, others will see it, too
  - Need a way to ***atomically*** see that the mutex is 0 ***and*** set it to 1
  - *How?*

Atomic Read-Update Instructions

- Atomic exchange instruction
  - E.g., EXCH R1,78(R2) will swap content of register R1 and mem location at address 78+R2
  - To acquire a mutex, 1 in R1 and EXCH
    - Then look at R1 and see whether mutex acquired
    - If R1 is 1, mutex was owned by somebody else and we will need to try again later
    - If R1 is 0, mutex was free and we set it to 1, which means we have acquired the mutex
- Other atomic read-and-update instructions
  - E.g., Test-and-Set

src = 1

```
xchg lock_var, src
```

If src == 0, you got the lock.

Implementing Locks

- A simple swap (or test-and-set) works
  - But causes a lot of invalidations
    - Every write sends an invalidation
    - Most writes redundant (swap 1 with 1)
- More efficient: test-and-swap (or test-and-test-and-set ☺)
  - Read, do swap only if 0
    - Read of 0 does not guarantee success (not atomic)
    - But if 1 we have little chance of success
  - Write only when good chance we will succeed
- *Would either scale? What can we do?*

Large-Scale Systems: Locks

- Contention even with test-and-test-and-set
  - Every write goes to many, many spinning procs
  - Making everybody test less often reduces contention for high-contention locks but hurts for low-contention locks
  - Solution: exponential back-off
    - If we have waited for a long time, lock is probably high-contention
    - Every time we check and fail, double the time between checks
      - Fast low-contention locks (checks frequent at first)
      - Scalable high-contention locks (checks infrequent in long waits)
  - Special hardware support
- Queuing locks

Queue Locks

- Test-and-test-and-set locks can still perform poorly
  - If lock is contended for by many processors
  - Lock release by one processor, creates "free-for-all" by others
  - Interconnect gets swamped with `swap` requests
- **Software queue lock**
  - Each waiting processor spins on a different location (a queue)
  - When lock is released by one processor...
    - Only the next processors sees its location go "unlocked"
    - Others continue spinning locally, unaware lock was released
  - Effectively, passes lock from one processor to the next, in order
  + Greatly reduced network traffic (no mad rush for the lock)
  + Fairness (lock acquired in FIFO order)
  - Higher overhead in case of no contention (more instructions)
  - Poor performance if one thread is descheduled by O.S.

What Are the Problems With Locks?

- Mapping between data->locks
  - Deadlocks
  - Races
  - Composability?
- Mmm, DB?
  - Optimistic concurrency

What If you Had Multi-Word LL-SC?

- Plus the ability to execute stores speculatively
- => Transactional Memory
  - Speculative execution + monitor CC trafic

Transactional Memory: The Big Idea

- Big idea I: **no locks, just shared data**
  - Look ma, no locks
- Big idea II: **optimistic (speculative) concurrency**
  - Execute critical section speculatively, abort on conflicts
  - "Better to beg for forgiveness than to ask for permission"

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

Transactional Memory: Read/Write Sets

- **Read set**: set of shared addresses critical section reads
  - Example: **`accts[37].bal`**, **`accts[241].bal`**
- **Write set**: set of shared addresses critical section writes
  - Example: **`accts[37].bal`**, **`accts[241].bal`**

```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

Transactional Memory: Begin

- **begin_transaction**
  - Take a local register checkpoint
  - Begin locally tracking read set (remember addresses you read)
    - See if anyone else is trying to write it
  - Locally buffer all of your writes (invisible to other processors)
  - \+ **Local actions only: no lock acquire**

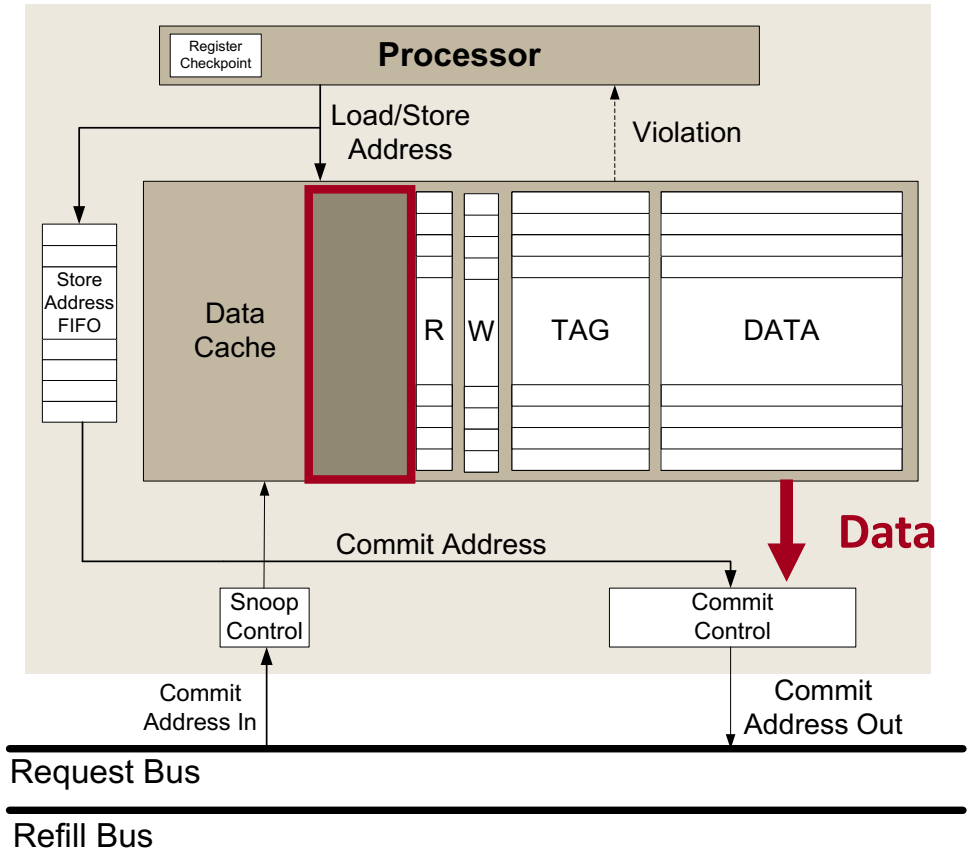
```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

Transactional Memory: End

- **`end_transaction`**
  - Check read set: is all data you read still valid (i.e., no writes to any)
  - Yes? Commit transactions: commit writes
  - No? Abort transaction: restore checkpoint

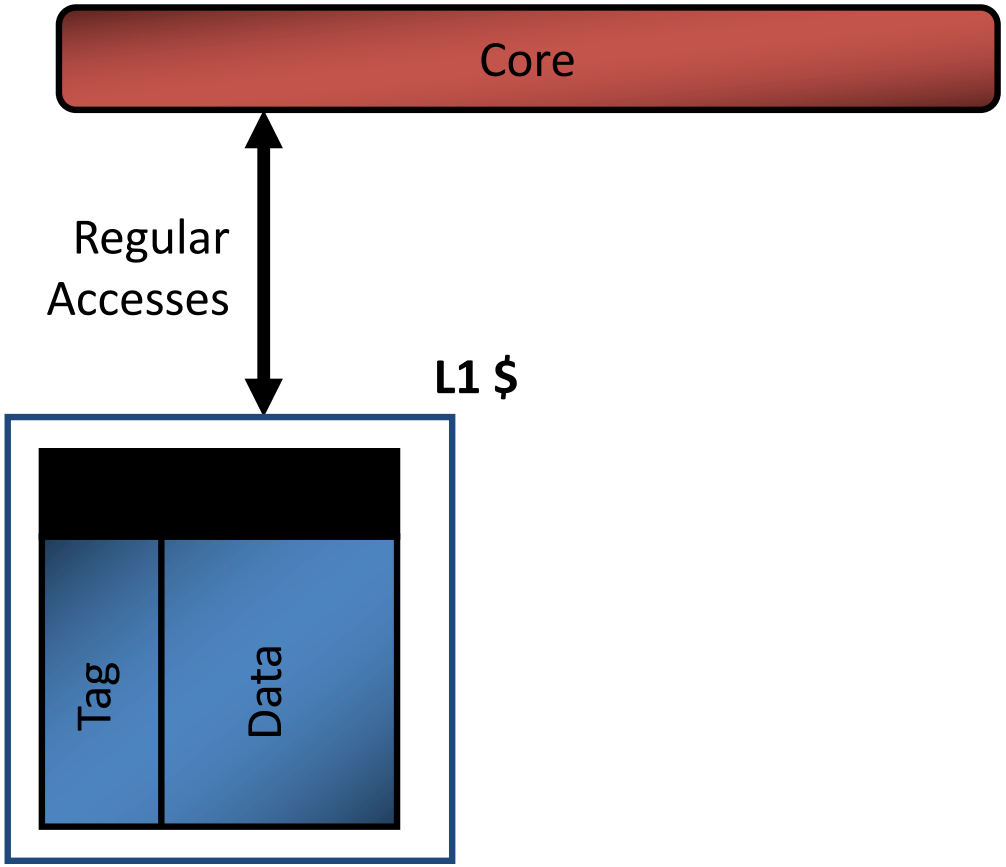
```
struct acct_t { int bal; };
shared struct acct_t  accts[MAX_ACCT];
int id_from,id_to,amt;

begin_transaction();
if (accts[id_from].bal >= amt) {
   accts[id_from].bal -= amt;
   accts[id_to].bal += amt; }
end_transaction();
```

Transactional Memory Hardware Support (HTM)



HTM

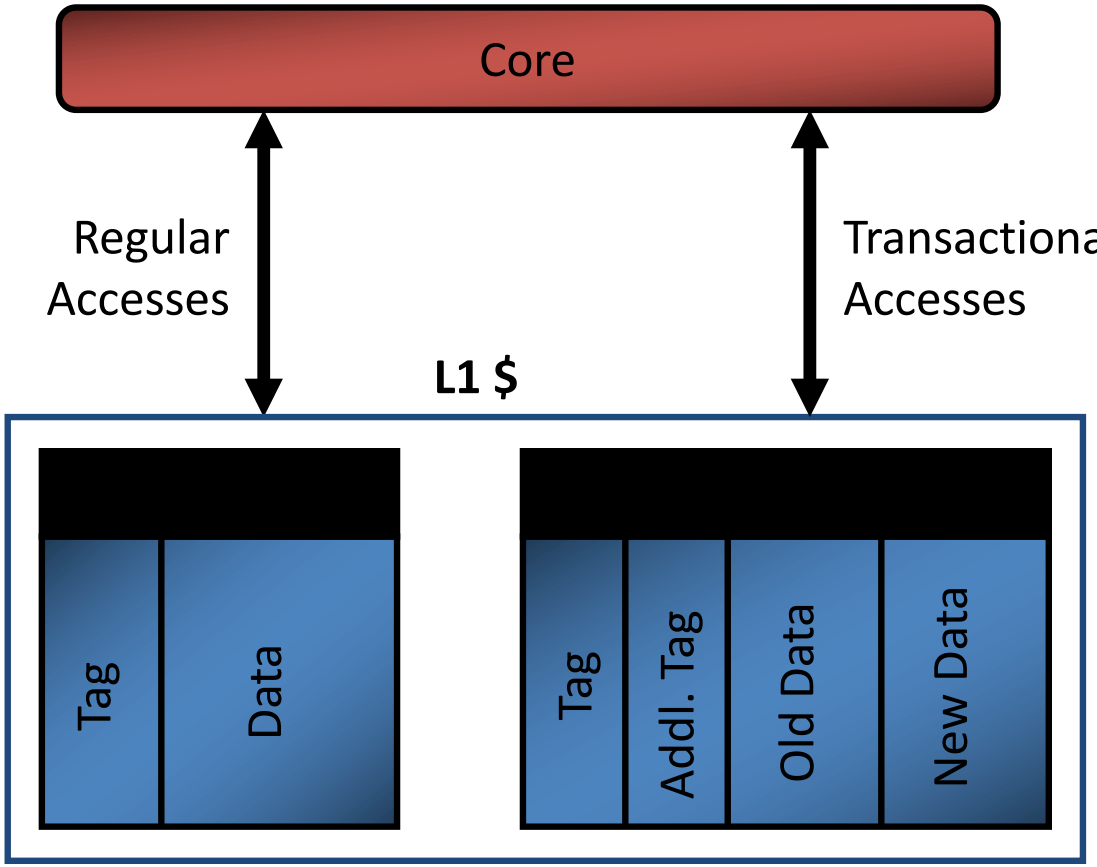
- Most hardware already exists
- Only small modification to cache needed



Kumar et al. (Intel)

HTM

- Most hardware already exists
- Only small modification to cache needed



Kumar et al. (Intel)

HTM Example

P1 Cache

P2 Cache

Tag	data	Trans?	State	Tag	data	Trans?	state

Bus Messages:

```
atomic {
  read A
  write B =1
}
```

```
atomic {
  read B


  Write A = 2
}
```

HTM Example

Tag	data	Trans?	State	Tag	data	Trans?	state
				B	0	Y	S

Bus Messages: **2** read B

```
atomic {
  read A
  write B =1
}
```

```
atomic {
  read B
```

```
  Write A = 2
}
```

HTM Example

Tag	data	Trans?	State	Tag	data	Trans?	state
A	0	Y	S				
				B	0	Y	S

Bus Messages: **1** read A

```
atomic {
  read A
  write B =1
}
```

```
atomic {
  read B


  Write A = 2
}
```

HTM Example

Tag	data	Trans?	State	Tag	data	Trans?	state
A	0	Y	S				
B	1	Y	M	B	0	Y	S

Bus Messages: NONE

```
atomic {
  read A
  write B =1
}
```

```
atomic {
  read B

  Write A = 2
}
```

Conflict, visibility on commit

Tag	data	Trans?	State	Tag	data	Trans?	state
A	0	N	S				
B	1	N	M	B	0	Y	S

Bus Messages: **1** B modified

```
atomic {
  read A
  write B =1
}
```

```
atomic {
  read B


ABORT


  Write A = 2
}
```

HTM –Strong isolation

Thread 1	Thread 2
`atomic {`	
`r1 = x;`	`x = 1;`
`r2 = x;`	
`}`	

Can `r1!=r2`?

(a) Non-repeatable reads

Initially `x==0`

Thread 1	Thread 2
`atomic {`	
`r = x;`	`x = 10;`
`x = r + 1;`	
`}`	

Can `x==1`?

(b) Lost updates

Initially `x` is even

Thread 1	Thread 2
`atomic {`	
`x++;`	`r = x;`
`x++;`	
`}`	

Can `r` be odd?

(c) Dirty reads

HTM – False Sharing

Tag	data	Trans?	State	Tag	data	Trans?	state
				C/D	0/0	Y	S

Bus Messages: Read C/D

```
atomic {
  read A
  write D = 1
}
```

```
atomic {
  read C
```

```
  Write B = 2
}
```

HTM – False Sharing

Tag	data	Trans?	State	Tag	data	Trans?	state
				C/D	0/0	Y	S
A/B	0/0	Y	S				

Bus Messages: Read A/B

```
atomic {
  read A
  write D = 1
}
```

```
atomic {
  read C

  Write B = 2
}
```

HTM – False Sharing

Tag	data	Trans?	State	Tag	data	Trans?	state
C/D	0/1	Y	M	C/D	0/0	Y	S
A/B	0/0	Y	S				

Bus Messages: Write C/D

```
atomic {
  read A
  write D = 1
}
```

```
atomic {
  read C

  Write B = 2
}
```

UH OH

HTM – Limited Size

Tag	data	Trans?	State	Tag	data	Trans?	state
A	0	Y	M				

Bus Messages: Read A

```
atomic {
  read A
  read B
  read C
  read D
}
  Write C/
```

HTM – Limited Size

Tag	data	Trans?	State	Tag	data	Trans?	state
A	0	Y	M				
B	0	Y	M				

Bus Messages: Read B

```
atomic {
  read A
  read B
  read C
  read D
}
```

HTM – Limited Size

Tag	data	Trans?	State	Tag	data	Trans?	state
A	0	Y	M				
B	0	Y	M				
C	0	Y	M				

Bus Messages: Read C

```
atomic {
  read A
  read B
  read C
  read D
}
```

HTM – Limited Size

Tag	data	Trans?	State	Tag	data	Trans?	state
A	0	Y	M				
B	0	Y	M				
C	0	Y	M				

Bus Messages: …

```
atomic {
  read A
  read B
  read C
  read D
}
```

UH OH

Can we just ignore locks and go ahead?

Speculative Lock Elision

Processor 0

```
acquire(accts[37].lock); // don't actually set lock to 1
// begin tracking read/write sets
// CRITICAL_SECTION
// check read set
//  no conflicts? Commit, don't actually set lock to 0
//  conflicts? Abort, retry by acquiring lock
release(accts[37].lock);
```

- Alternatively, keep the locks, but…
- … speculatively transactify lock-based programs in hardware
  - **Speculative Lock Elision (SLE)** [Rajwar+, MICRO'01]
    - Captures most of the advantages of transactional memory…
  - + No need to rewrite programs
  - + Can always fall back on lock-based execution (overflow, I/O, etc.)

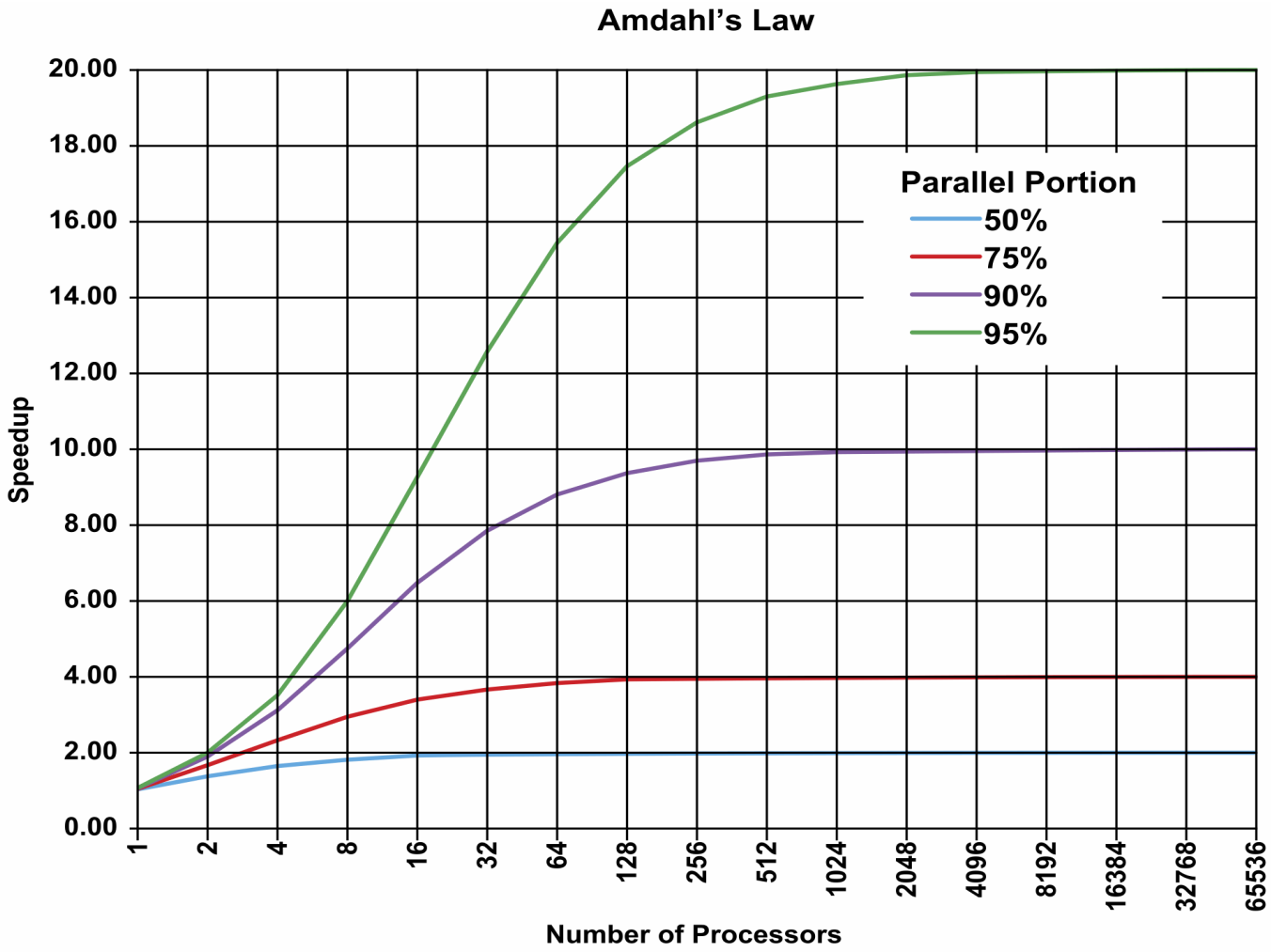
Amdahl's Law

- Restatement of the law of diminishing returns
  - Total speedup limited by non-accelerated piece
  - Analogy: drive to work & park car, walk to building

- Consider a task with a "parallel" and "serial" portion
  - What is the speedup with N cores?
  - Speedup(n, p, s) = (s+p) / (s + (p/n))
    - p is "parallel percentage", s is "serial percentage"
  - What about infinite cores?
    - Speedup(p, s) = (s+p) / s = 1 / s

- Example: can optimize 50% of program A
  - Even a "magic" optimization that makes this 50% disappear…
  - …only yields a 2X speedup

Amdahl's Law Graph



Source: Wikipedia

Discussion

- Does cache coherence scale?
  - Does message passing scale?

- What would you do with 1000 (or 1M) cores?
  - Speculation for parallelism
  - Fusing cores to form a larger one that better exploits ILP
  - Continuous system improvement? (codegen, monitoring watchdogs, etc)

Memory Consistency in GPUs/CPU+GPU

- Main issue
  - fences involve getting ACKs from whole system
  - GPUs are "scoped" to avoid excessing control communication
- What makes sense in this context?
  - Often explicit kernels operating on large data sets/regions
  - Explicit data partitioning/communication
  - Sync barriers are frequent
- Goal of a memory model in this context:
  - Provide reasonable semantics for system software
  - Enable optimizations
  - Avoid excessive hardware complexity
- Recent proposal from AMD: HRF, up next week