CS 6354: Processor Networks

5 October 2016

To read more...

This day's papers:

Scott, "Synchronization and Communication in the T3E Multiprocessor"

C.mmp—A multi-mini-processor

Supplementary readings:

Hennessy and Patterson, section 5.1–2

Homework 1 Post-Mortem

Almost all students had trouble with:

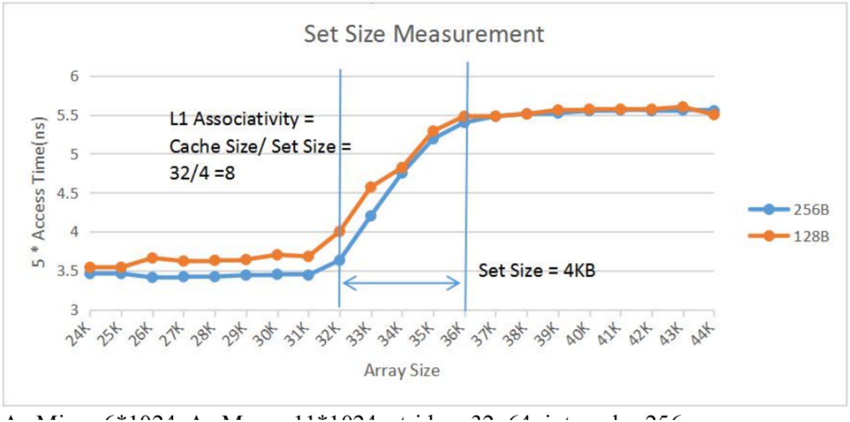
- associativity (TLB or cache)
- TLB size
- instruction cache size

Many not-great results on:

- latency and throughput
- block size

HW1: Cache assoc.

From Yizhe Zhang's submission:



HW1: Cache assoc.

Example: 2-way assoc. 4-entry cache

pattern: 0/1/2/3/0/1/2/3/...(0/4 misses)

	addresses
set 0 (addr mod 2 == 0)	0/2
set 1 (addr mod 2 == 1)	1/3

pattern: 0/1/2/3/4/0/1/2/3/4/...(3/5 misses)

	addresses		
set 0 (addr mod 2 == 0)	0/2 (after 2)	2/4 (after 4)	4/0 (after 0)
set 1 (addr mod 2 == 1)	1/3		

pattern: 0/1/2/3/4/5/0/1/2/3/4/5/...(6/6 misses)

	addresses		
set 0 (addr mod 2 == 0)	0/2 (after 2)	2/4 (after 4)	4/0 (after 0)
set 1 (addr mod 2 == 1)	1/3 (after 3)	3/5 (after 5)	5/1 (after 1)

HW1: Cache assoc. nits

Problem: virtual != physical addresses

Solution 1: Hope addresses are contiguous (often true shortly after boot)

Solution 2: Special large page allocation functions

HW1: TLB associativity

Things which seem like they should work (and full credit):

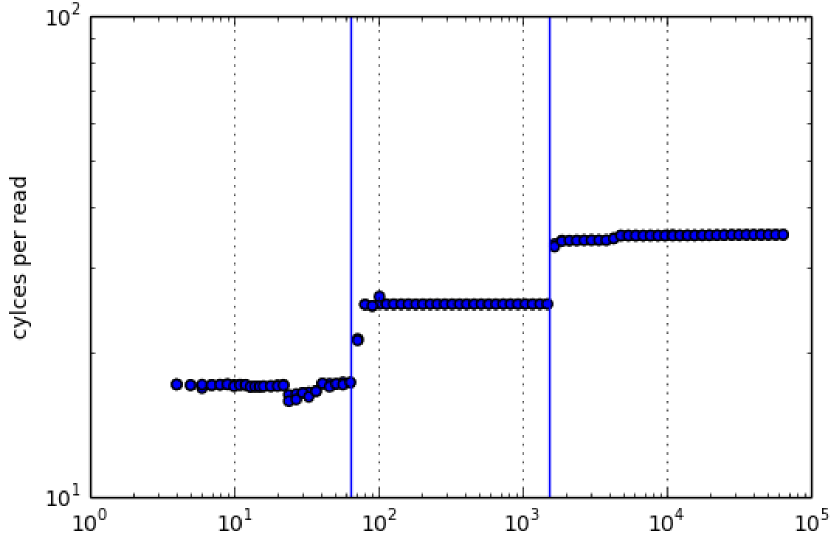
Strategy 1 — same as for cache, but stride by page size

Strategy 1 — stride = TLB reach, how many fit

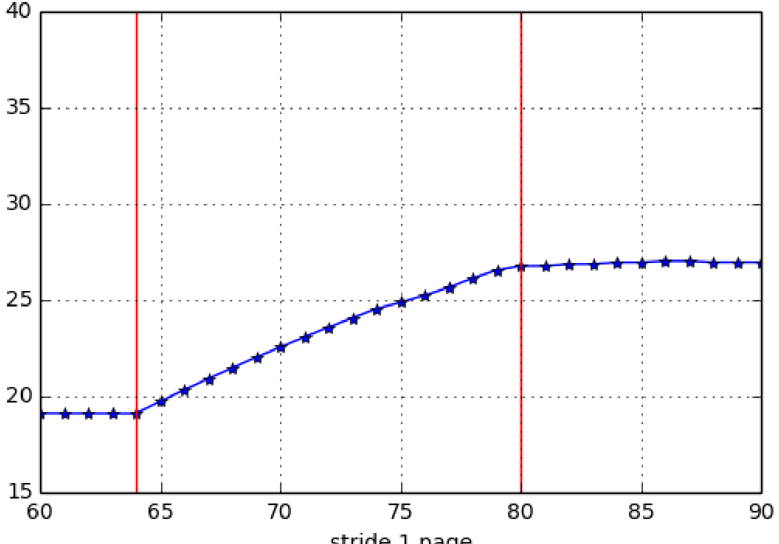
Strategy 2 — stride = TLB reach / guessed associativity

- idea: will get all-misses with # pages = associativity

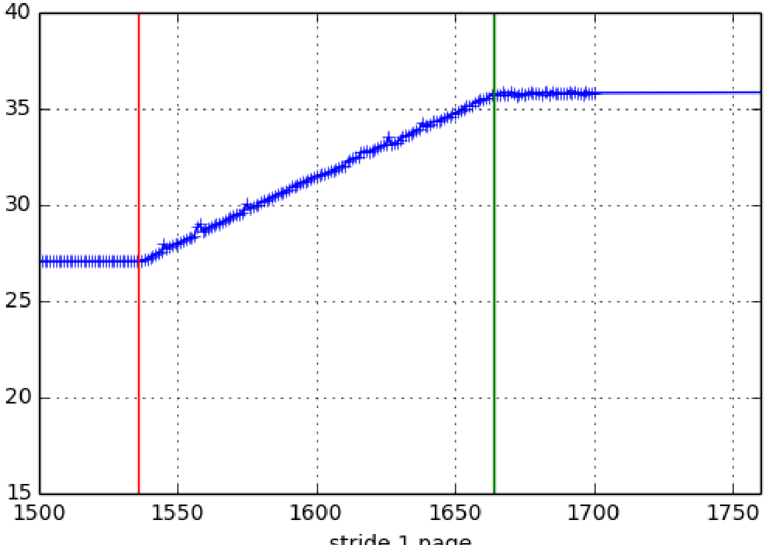
My TLB size results



My TLB associativity results (L1)



My TLB associativity results (L2)



TLB benchmark: controlling cache behavior

page table:

virtual page number	physical page number
1000	13248
1001	13248
1002	13248
1003	13248
1004	13248
1005	13248
…	…

TLB benchmark: preventing overlapping loads

multiple parallel page table lookups

don't want that for measuring miss time

```
index = index + stride + array[value];
```

force dependency

also an issue for many other benchmarks

Instruction cache benchmarking

Approx two students successful or mostly successful

Obstacle one: variable length programs?

Obstacle two: aggressive prefetching

Variable length programs

Solution 1: Write program to generate source code

- many functions of different lengths
- plus timing code
- multimegabyte source files

Solution 2: Figure out binary for 'jmp to address'

- allocate memory
- copy machine code to region
- add return instruction
- call as function (cast to function pointer)

Avoiding instruction prefetching

Lots of jumps (unconditional branches)!

Basically requires writing assembly/machine code

Might measure branch prediction tables!

HW1: Choose two most popular

prefetching stride — see when increasing stride matches random pattern of same size

multicore/thread throughput — run MT code

large pages — straightforward if you can allocate large pages

HW1: optimization troubles

```c
int array[1024 * 1024 * 128];
int foo(void) {
    for (int i = 0; i < 1024 * 1024 * 128; ++i) {
        array[i] = 1;
    }
}
```

unoptimized loop: `gcc -S foo.c`

```
.L3:
 movl     -4(%rbp), %eax         // load 'i'
 cltq
 movl     $1, array(,%rax,4)     // 4-byte store 'array[i]'
 addl     $1, -4(%rbp)           // load+add+store 'i'
 movl     -4(%rbp), %eax         // load 'i'
 cmpl     $134217727, %eax
 jbe      .L3
```

HW1: optimization troubles

```
int array[1024 * 1024 * 128];
int foo(void) {
    for (int i = 0; i < 1024 * 1024 * 128; ++i) {
        array[i] = 1;
    }
}
```

optimized loop: `gcc -S -Ofast -march=native foo.c`

```
.L4:
 addl    $1, %eax         // 'i' in register
 vmovdqa %ymm0, (%rdx)    // 32—byte store
 addq    $32, %rdx
 cmpl    %ecx, %eax
 jb      .L4
```

HW1: optimization troubles

```
int array[1024 * 1024 * 128];
int foo(void) {
    for (int i = 0; i < 1024 * 1024 * 128; ++i) {
        array[i] = 1;
    }
}
```

HW1: Misc issues

allowing overlap (no dependency/pointer chasing)
- hard to see cache latencies
- wrong for measuring latency
- but right thing for throughput

not trying to control physical addresses
- easy technique — large pages
- sometimes serious OS limitation

controlling measurement error
- is it a fluke? how can I tell?

Homework 2

checkpoint due Saturday Oct 15

using gem5, a processor simulator

analyzing statistics from 4 benchmark programs

you will need: a 64-bit Linux environment (VM okay, no GUI okay)

... or to build gem5 yourself

multithreading

before: multiple streams of execution within a processor

- shared almost everything (but extras)
- shared memory

now: on multiple processors

- duplicated everything except...
- shared memory (sometimes)

a philosophical question

$$\begin{tikzcd}\text{multiprocessor machine} & \text{dividing line?} \arrow[l] \arrow[r] & \text{network of machines}\end{tikzcd}$$

C.mmp worries

efficient networks

memory access conflicts

OS software complexity

user software complexity

C.mmp worries

efficient networks

memory access conflicts

OS software complexity

user software complexity

topologies for processor networks

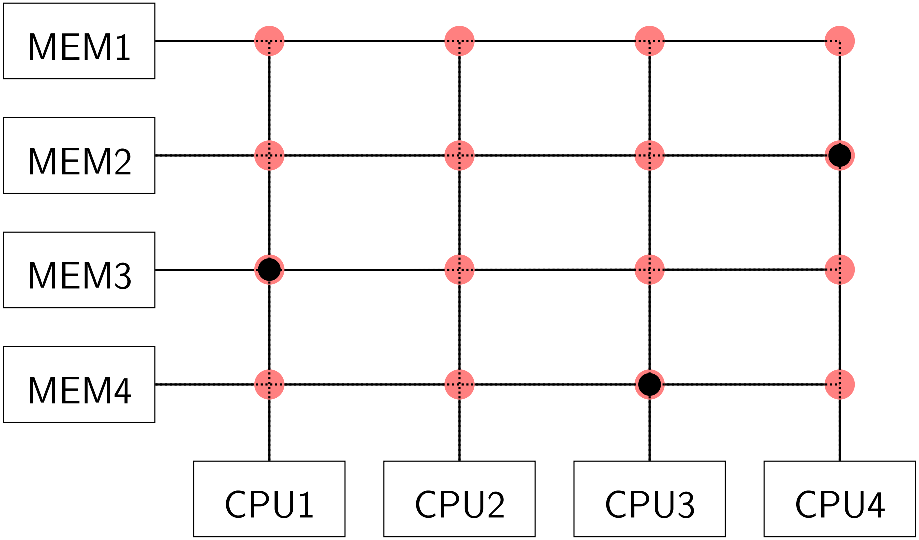
crossbar

shared bus

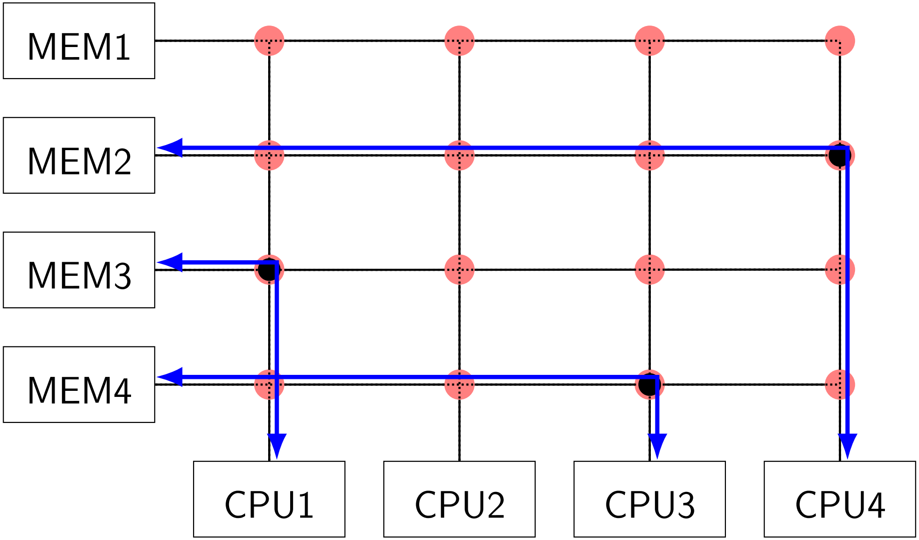
mesh/hypertorus

fat tree/Clos network

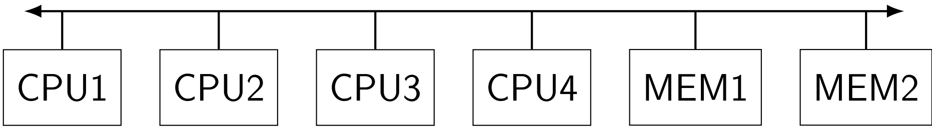
crossbar (approx. C.mmp switch)



crossbar (approx. C.mmp switch)



shared bus



tagged messages — everyone gets everything, filters

arbitrartion mechanism — who communicates

contention if multiple communicators

hypermesh/torus

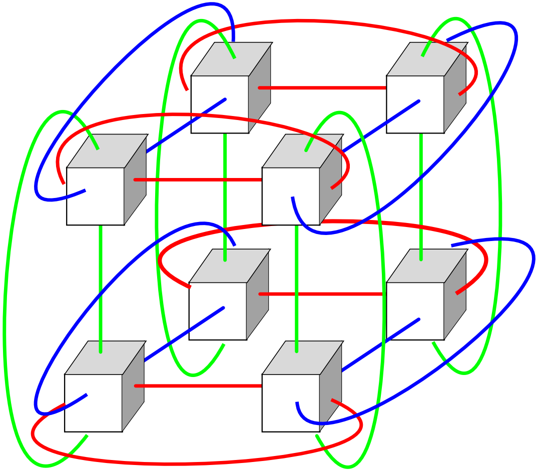


Image: Wikimedia Commons user おむこさん志望

hypermesh/torus communication

some nodes are closer than others

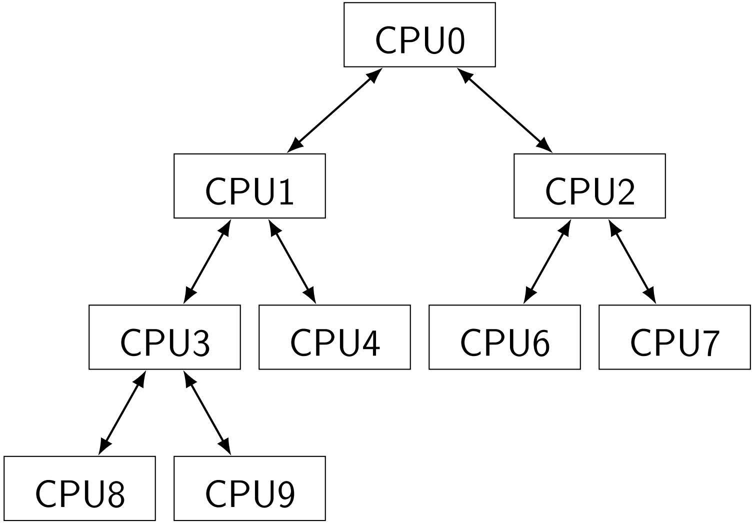
take advantage of physical locality

multiple hops — need routers

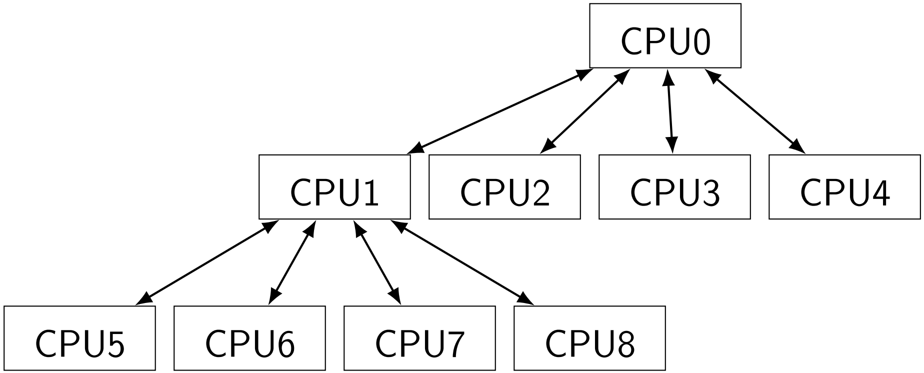
simple algorithm:

- get to right x coordinate
- then y coordinate
- then z coordinate

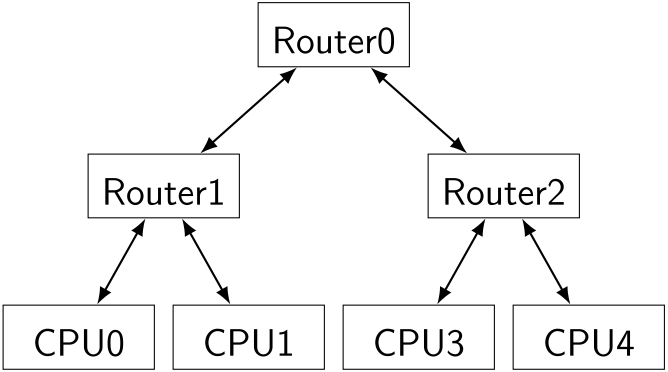
trees



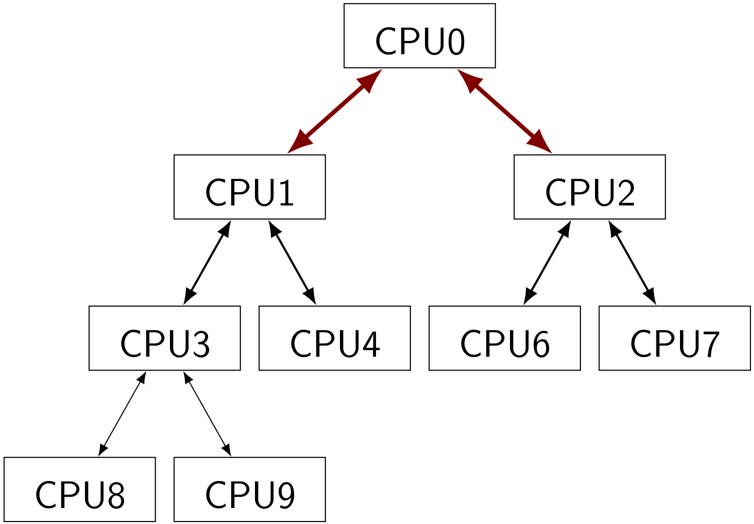
trees (thicker)



trees (alternative)



fat trees



fat trees

don't need really thick/fast wires

take bundle of switches

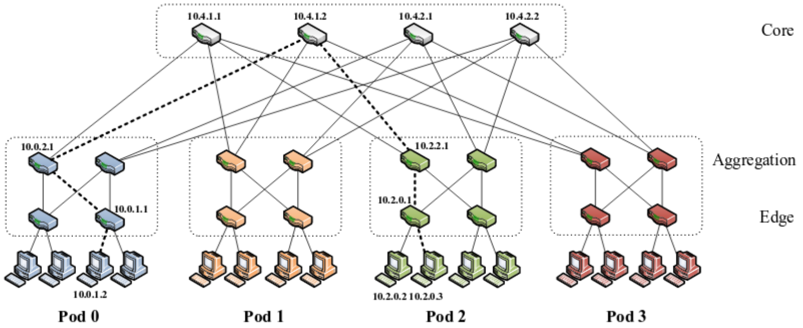


Figure from Al-Fares et al, “A Scalable, Commodity Data Center Network Architecture”, SIGCOMM'08 31

minimum bisection bandwidth

half of the CPUs communicate with other half

what's the worst case:

crossbar: same as best case

fat tree, folded Clos: same as best case
    or can be built with less (cheaper)

tree: $1/N$ of best (everything through root)

shared bus: $1/N$ of best (take turns)

hypertorus: in between

other network considerations

	crossbar	fat tree (full BW)	hypertorus
bandwidth	N	N	$\sqrt[d]{N}$
max hops	1	$2k$	$d\sqrt[d]{N}$
$\#$ switches	1	k	N
switch capacity	N	$2k$	$2d$
short cables	no	no	yes

non-asymptotic factors omitted

N: number of CPUs

k: switch capacity

d: number of dimensions in hypertorus

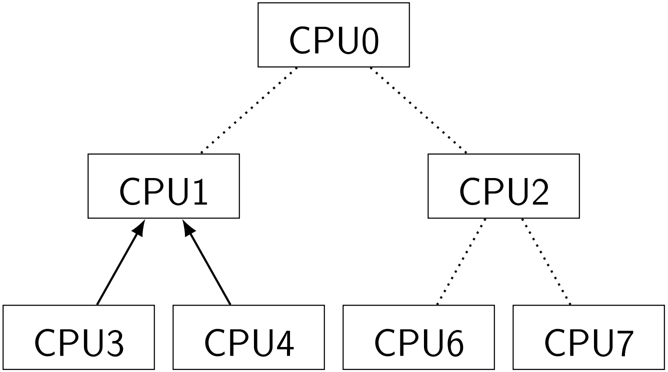
metereological simulation

```
compute_weather_at(int x, int y) {
    for step in 1,...,MAX_STEP {
        weather[step][x][y] = computeWeather(
            weather[step−1][x−1][y  ],
            weather[step−1][x  ][y−1],
            weather[step−1][x  ][y  ],
            weather[step−1][x+1][y  ],
            weather[step−1][x  ][y+1]
        );
        BARRIER();
    }
}
```

barriers

wait for everyone to be done

two messages on each edge of tree



C.mmp worries

efficient networks

memory access conflicts

OS software complexity

user software complexity

memory access conflicts

assumption: memory distributed randomly

may need to wait in line for memory bank

makes extra processors less effective

T3E's solution

local memories

explicit access to remote memories

programmer/compiler's job to decide

tools to help:

- centrifuge — regular distribution across CPUs
- virtual processor numbers + mapping table

hiding memory latencies

C.mmp — don't; CPUs were too slow

T3E local memory — caches

T3E remote memory — many parallel accesses

need 100s to hide multi-microsecond latencies

programming models

T3E — explicit accesses to remote memory

programmer must find parallelism in accesses

C.mmp — maybe OS chooses memories?

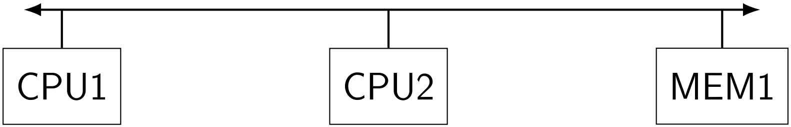
caching

C.mmp — read-only data only

T3E — local only

remote accesses check cache next to memory

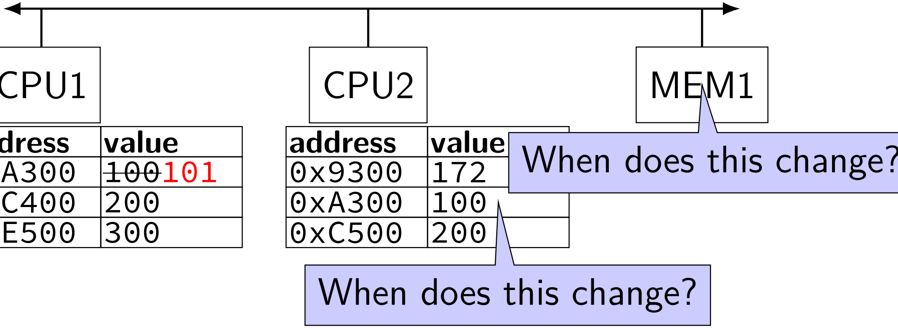
caching shared memories



address	value
`0xA300`	`100`
`0xC400`	`200`
`0xE500`	`300`

address	value
`0x9300`	`172`
`0xA300`	`100`
`0xC500`	`200`

caching shared memories



address	**value**
0xA300	~~100~~101
0xC400	200
0xE500	300

address	**value**
0x9300	172
0xA300	100
0xC500	200

CPU1 writes 101 to 0xA300?

simple shared caching policies

don't do it — T3E policy

if read-only — C.mmp policy

tell all caches about every write

- "free" if write-through policy and shared bus

all caches know about every write?

doesn't scale

worse than write-through with one CPU!

wait in line behind other processors to send write

(or overpay for network)

next week: better strategies

don't want one message for every write

will require extra bookkeeping

next Monday — for shared bus

next Wednesday — for non-shared-bus

C.mmp synchronization

C.mmp: locking — exclusive use of OS resource

concern about granularity:

too much overhead from locking?

too much overhead from waiting for lock?

T3E synchronization

atomic operations — easy, executed at one location

read-modify-write commands to remote memory

compare-and-swap-if-equal

read-and-add

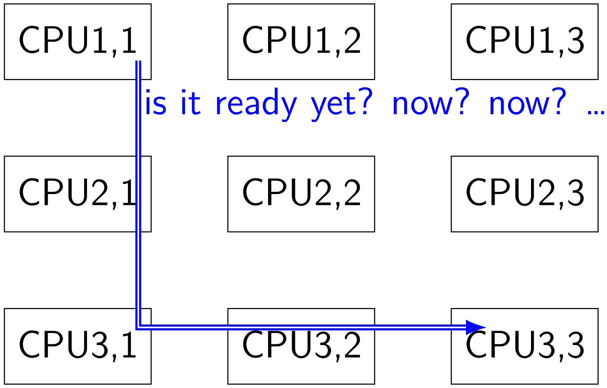
compare-and-swap

```
compare-and-swap(address, expect-old-value, new-va
    atomically {
        if (expect-old-value == memory[address])
            memory[address] = new-value
        }
    }
}
```

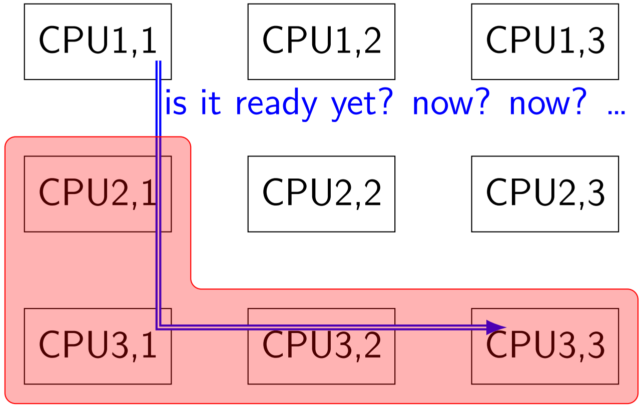
locks with compare-and-swap

```
// compare-and-swap(address,
//                  expect-old-value,
//                  new-value) == 1 if changed
while (!compare-and-swap(&lock, 0, 1)) {}
    // keep trying until value is 0

// use shared resource here

lock = 0; // release lock
```

memory traffic and the lock



memory traffic and the lock



performance lost here because of CPU1,1!

lock-free queue

```
struct queue { int value; struct queue *next; };
struct queue *head;
void addToQueue(int value) {
    struct queue *newEntry = allocateNew(value);
    do {
        newEntry->next = head; // read head
        // other thread might change head here!
        // replace head if not changed yet
    } while (compare-and-swap(&head,
                 newEntry->next, newEntry));
}
```

lock-free/wait-free data structures

use atomic operations 'directly' (no lock)

very tricky to reason about

wait-free: program makes progress even if one thread stops

T3E alternative: message queues

atomic “add to remote queue” operation

- only keep retrying if remote queue was full

check own queue — repeatedly read local memory

no extra traffic

shared memory synchronization

usually ‘just’ atomic ‘read-modify-write’ operations

- compare-and-swap
- read-and-add

later: using these to implement locks

later: transactional memory: an alternative to read/modify/write

papers for next time

Goodman, “Using cache memory to reduce processor-memory traffic”

seminal paper on cache coherency

Archibald and Baer, “Cache coherence protocols: evaluation using a multiprocessor simulation model

several expansions on Goodman’s work

Snooping coherency trick

Listen to other cache's requests to memory

Respond with data if requested

even though request was for memory, not you

Allows write-back policy