CS61C Final Study Guide - Matthew Tang [Updated 12/8/20 3:00pm]

1 Caches

– **Temporal locality**: Use it recently, want to use it again soon
- ∘ Keep most recently accessed data closer

– **Spatial locality**: Use a piece of memory, want to use neighboring pieces soon
- ∘ Move blocks of contiguous words closer to processor

1.1 TIO

– **Tag**: Check if have correct block
- ∘ Num bits = mem addr bits $-I-O$
- ∘ Distinguish diff blocks that use same index

– **Index**: Select block
- ∘ Num bits = $\log_2$(# of indicies)

– **Offset**: Byte offset within block
- ∘ Num bits = $\log_2$(size of block)

– **Valid bit**: 1 for valid entries

– Check in order of Index, Valid, Tag, Offset

– Cache size = $I * O$

– Address bit width = $\log_2$(memory size) = $T + I + O$

1.2 Policies

– **Write-through**: Update cache and memory at same time

– **Write-back**: Update word in cache block, allowing memory word to be stale
- ∘ Add dirty bit to block
- ∘ Update memory when block is removed (and dirty bit = 1)

– **Write-allocate**: Bring block into cache after write miss

1.3 Cache Types

– **Direct-mapped cache**: (1 way set assoc.)
- ∘ **Replacement**: Index specifies position to replace
- ∘ Each memory addr associated with 1 block in cache
- ∘ Look in single location in cache for data

– **Fully Associative Caches**: (M way set assoc.)
- ∘ **Replacement**: Can replace any position
- ∘ Tag and offset same as before, no index now
- ∘ **Benefits**: No conflict misses since data can go anywhere
- ∘ **Drawbacks**: Hardware is difficult to make

– **Set Associative Cache**:
- ∘ **Replacement**: Can replace any within the set
- ∘ Tag and offset same as before, index points to correct set
- ∘ Each set contains multiple blocks
- ∘ Once found correct set, compare with all blocks in set
  - ∗ Compare tag with all tags in set
- ∘ **Benefits**: Avoids conflict misses using N comparators

1.4 Cache Misses

1) **Compulsory**:
- ∘ First time you access a certain block
- ∘ **Solution**: bigger block size

2) **Conflict**:
- ∘ Wouldn't have occurred with fully associative cache (LRU)
- ∘ **Solution**: increase associativity or replacement policy

3) **Capacity**:
- ∘ Independent of associativity of cache
- ∘ If not compulsory or conflict miss
- ∘ Occurs you still miss with fully associative cache (LRU)
- ∘ **Solution**: Increase cache capacity

– **Replacement policies**:
- ∘ **Least Recently Used (LRU)**:
  - ∗ Better temporal locality, harder to keep track of
- ∘ **First In First Out (FIFO)**: Just track initial order
- ∘ **Random**: Good for data with low temporal locality

1.5 AMAT

– **Average Memory Access Time (AMAT)**:
- ∘ AMAT = Hit Time + Miss Rate $*$ Miss Penalty

– **Global miss rate**: misses / total # accesses for cache system

– **Local miss rate**: misses / total # accesses for cache level

– Recursive expression for multi-level cache

2 Virtual Memory

– **Virtual address**: Can be diff # bits than physical

Virtual Page Number (VPN)	offset

– **Physical address**: Can be diff # bits than physical

Physical Page Number (PPN)	offset

– **Page Table**: Stores mapping of VPN to PPN
- ∘ Also includes valid bit, dirty bit, permission bits
- ∘ Page Table entry: a row of the PT
- ∘ Page Table Base Register (PTBR) points to PT

Valid	Dirty	Permission Bits	PPN
– PT Entry (VPN: 0) –			
– PT Entry (VPN: 1) –			

– Hierarchical PT has many page numbers for L1, L2 caches

– **Protection Fault**: PT Entry permission bits prohibit op

– **Page fault**: PT Entry has valdi bit set to false (not in mem)
- ∘ Transfer page to memory (evict a page if necessary)
- ∘ Add mapping to PT and TLB

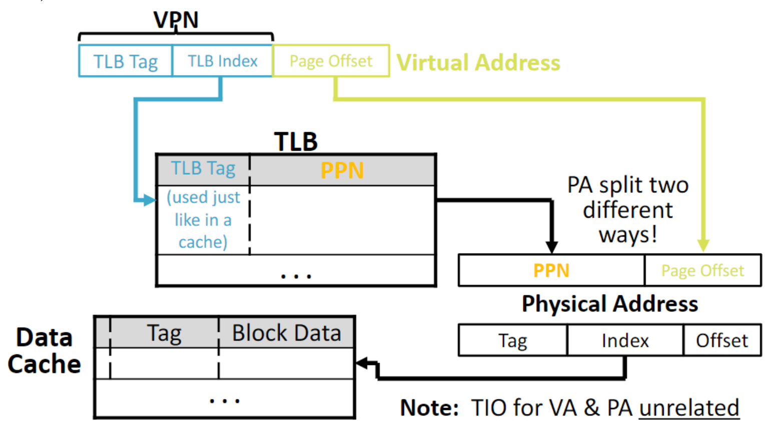
2.1 TLB Address Lookup

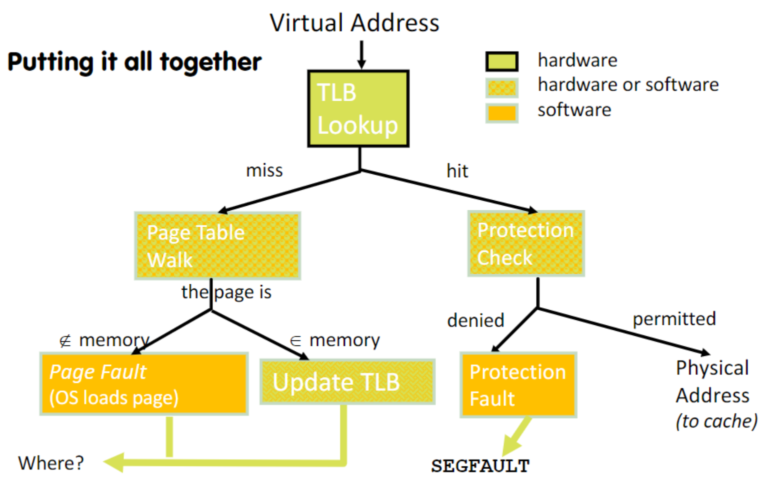
– **Translation Lookaside Buffer (TLB)**: Cache for PT
- ∘ Each block is a single PT entry
- ∘ **TLB Miss**: If entry is not in TLB
- ∘ TLB is a subset of the PT
- ∘ TLB Reach: Size of largest virtual address space that can fit in TLB

TLB Valid	TAG (VPN)	Page Table Entry		
		Page Dirty	Perm Bits	PPN
– TLB Entry –				
– TLB Entry –				

– **Procedure**:

1) Get VPN from Virtual Address
   - ∘ Translate VPN to PPN using TLB
2) If TLB does not have VPN, check if PT contains
3) If PT does not have VPN, this is a page fault
   - ∘ Fetch physical page from disk to memory
4) Update PT Entry, load entry into TLB
5) Use Physical Page and offset to access memory





3 IO

– **Polling**: Forces hardware to wait on ready bit
1) Processor reads from control register in loop until ready
2) Processor does IO, resets control register bit to 0
 ∘ **Pros**: Low latency, low overhead when data is available
  ∗ Good for when you need reply to make progress
 ∘ **Cons**: Can't do anything else (or sleep) while polling
– **Interrupts**: Hardware fires exception when ready
 ∘ CPU changes PC to execute interrupt handler
 ∘ Low data rate (mouse, keyboard): use interrupts
 ∘ High data rate (network, disk): start with interrupts, switch to DMA
 ∘ **Pros**: Can do work while waiting, wait for many at once
  ∗ Good for devices that take long to respond
 ∘ **Cons**: Nondeterministic when interrupt occurs, higher latency, worse throughput

3.0.1 DMA

– **Direct Memory Access (DMA)**: Allows IO devices to directly read/write main memory
– **Incoming Data**:
1) Receive interrupt from device
2) CPU takes interrupt, initiates transfer
 ∘ Instructs DMA to place data at certain address
3) DMA handles transfer as CPU does other things
4) DMA interrupts CPU once finished
– **Outgoing Data**:
1) CPU initiates transfer, confirms device is ready
2) CPU begins transfer
 ∘ Instructs DMA that data is at certain address
3) DMA handles transfer as CPU does other things
4) DMA interrupts CPU once finished

3.1 OS

– When to transition into supervisor mode:
 ∘ Interrupts: Something external to the running program
  ∗ Async to curent program (ex. key press, disk IO)
 ∘ Exception: Caused by an event during execution of an instruction by the running program
  ∗ Sync, on instruction that causes exception (ex. mem error, buss error, illegal instruction)
 ∘ Trap: action of servicing interrupt or exception by hardware jump to interrupt/trap handler code
– Ecall: Trigger exception to higher privilege
– Ebreak: Trigger exception within current privilege

4 Data Parallelism

– **Instruction level parallelism**: Pipelining
– **Data level parallelism**: Single instruction operation on multiple streams of data
– **Request level parallelism**: Many requests/sec reading a database (not read/write sync)
– Flynn's Taxonomy:
 ∘ SISD: Single Instruction Single Data (traditional programs)
 ∘ SIMD: Single Instruction, Multiple Data (SSE intrinsics)
 ∘ MISD: Multiple Instruction, Single Data
 ∘ MIMD: Multiple Instruction, Multiple Data (map reduce)

4.1 SIMD

– Example methods:

```
_mm_loadu_si128((__m128i*) (vals + offset));
_mm_storeu_si128((__m128i*) p, _sum_vector);
```

– Tail Case:

```
for(int i = N / 4 * 4; i < N; i++) {
  // Tail Case
}}
```

– Unroll: 4 iterations per time

5 Thread-Level Parallelism

– Often more logical CPUs than physical CPUs
– OpenMP uses a fork-join model
– **Amdahl's Law**: True Speedup $= \frac{1}{S+\frac{1-S}{P}}$
 ∘ S is portion that is serial
 ∘ P is the speedup factor
 ∘ $\lim_{P\to\infty} \frac{1}{S+\frac{1-S}{P}} = \frac{1}{S}$

5.1 OpenMP Directives

– Import: `#include <omp.h>`
– Parallel: `#pragma omp parallel`
 ∘ Each thread runs a cop of the code in block
– For: `#pragma omp parallel for`
 ∘ Each thread runs different iterations of the for loop
– Thread functions:
 ∘ `int omp_get_thread_num()`: thread num executing the code
 ∘ `int omp_get_num_threads()`: total num of threads
– Critical: `#pragma omp critical`
 ∘ Only 1 thread can run critical part at a time
– Reduction: `#pragma omp parallel for reduction(op:var)`
 ∘ Reduces thread results into var using operation

5.2 Synchronization

– Use locks to control access to shared resources
– **Critical section**: Area that must be serialized
– **Solution**: Atomic read/write in single instruction
 ∘ **Atomic Memory Operation (AMO)**: Atomic swap of register and memory
  ∗ **Load-reserve, store-conditional**: Uninterrupted execution across multiple inst
  ∗ **Amo.swap**: Uninterrupted memory ops in 1 instruction
– **OpenMP locks**:
 ∘ Declare lock: `omp_lock_t lock`
 ∘ `omp_init_lock(&lock)`, `omp_set_lock(&lock)`
 ∘ `omp_unset_lock(&lock)`, `omp_destroy_lock(&lock)`

– Use `critical` directive to automate lock setting

```
Test-and-set

Start:  addi         t0 x0 1 # Locked = 1
        amoswap.w.aq t1 t0 (a0)
        bne          t1 x0 Start
# If the lock is not free, retry

        ... # Critical section

        amoswap.w.rl x0 x0 (a0) # Release lock
```

```
Compare-and-swap

# a0 holds address of memory location
# a1 holds expected value
# a2 holds desired value
# a0 holds return value, 0 if successful, !0
    otherwise
cas:
    lr.w t0, (a0) # Load original value.
    bne t0, a1, fail # Doesnt match, so fail.
    sc.w a0, a2, (a0) # Try to update.
    jr ra # Return.
fail:
    li a0, 1 # Set return to failure.
    jr ra # Return.
```

5.3 Cache Coherency

– Cache-incoherence: Each hardware thread cache does not have up to date memory
– Read-modify-write: Interleaving of R M W stages
– Cache states: MOESI
 1) **Modified**: Up to date data, changed/dirty
  ∘ No other cache has a copy, ok to write
  ∘ Memory out of date (need to write back)
 2) **Owner**: Up to date data
  ∘ Other caches can have copy, but must be shared
  ∘ Variation of Shared, supplies data instead of going to memory on read miss
  ∘ Can become Shared after writing modifications to memory
  ∘ Variation of Modified, avoids writing to memory on miss
  ∘ Can become Modified if all shared are invalidated
 3) **Exclusive**: Up to date data, unchanged
  ∘ No other cache has a copy, ok to write
  ∘ Memory up to date (don't need to write back)
 4) **Shared**: Up to date data
  ∘ Other caches can have a copy
 5) **Invalid**: Not in cache
– Blocks move to Invalid if another processor writes to it
– Blocks move to Shared if another processor reads it
– **False sharing**: Block goes between 2 caches even though variables are disjoint
 ∘ **Coherence/communication miss**: Disjoint variables, but have to invalidate block as one updates
– Permitted cache states:

	M	O	E	S	I
M					Y
O				Y	Y
E					Y
S		Y			Y
I	Y	Y	Y	Y	Y

– **MOESI FSM**:
 ∘ Start at invalid
 ∘ Transition based on the processor reading/writing
 ∘ **Read/Write Hits**: The processor that is caching the block
 ∘ **Probe Read/Write Hit**: A different processor attempts to access a block that is cached by another processor.

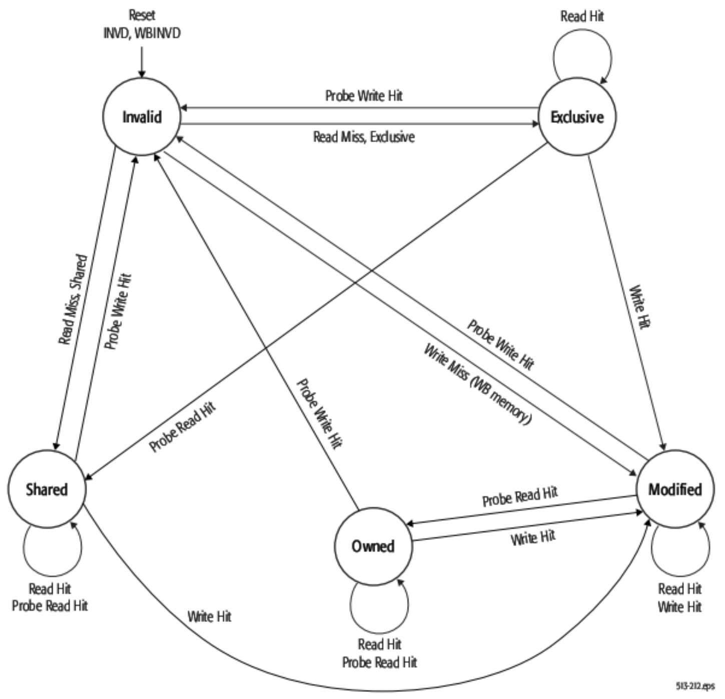
6 Distributed Computing

6.1 MapReduce

– **Map**: processes input key/value pair
 ∘ Slices data into shards/splits, distributed to workers
 ∘ Produces set of intermediate pairs
  ∗ Use `emit(key k, value v)` to output K-V pair (k, v)
– Group by key, send to reduce
– **Reduce**: Combines intermediate values for a particular key, produces an output value
– All Map must finish before reduce, but can start reading values

6.2 Spark

– Much faster than MapReduce
– Resilient Distributed Dataset (RDD): Abstraction of a DB
– Transforms (RDD $\rightarrow$ RDD):
 ∘ `map(f)` : Calls f on each source element
 ∘ `flatmap(f)` : Each input item can be mapped to ≥ 0 output items
 ∘ `reduceByKey(f)` : Aggregate by key using $f: (V, V) \rightarrow V$
– Actions (RDD $\rightarrow$ RDD):
 ∘ `reduce(f)` : Aggregate elements regardless of key
– Example: ∘ `coinData = sc.parallelize(coinPairs)`
 ∘ `out = coinData.map(lambda (k1, k2):  ((k1, k2), 1))`
  ∗ `.reduceByKey(lambda v1, v2:  v1 + v2)`
  ∗ Computes # coins of each type of each person
– Lazy evaluation model, evaluates after `.collect()`



7 Dependability

– Power Usage Effectiveness (PUE): Total building power / IT equipment power
  - 1.0 is perfect

– Rack contains 40-80 servers, Array contains 16-32 racks

7.0.1 Dependability Measures

– **Mean Time To Failure (MTTF)**: Reliability
– **Mean Time To Repair (MTTR)**: Service interruption
– **Mean Time Between Failures (MTBF)**: MTBF = MTTF + MTTR
– **Availability**: MTTF / (MTTF + MTTR)
  - Number of 9s availability per year: 1 nine: 90%, 2 nine: 99%

7.0.2 Reliability Measures

– **Annualized Failure Rate (AFR)**: Average number of failures per year
  - AFR = 24 * 365 / MTTF

– **Failures in Time (FIT)**: Num of failures expected in 1 billion device-hours of operation
  - MTBF = 1 billion / FIT

7.1 RAID

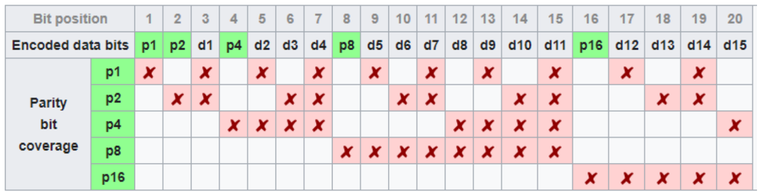
– Always know which disk fails
– RAID0 provides maximum usable disk space
– RAID1 is most expensive but highest availability
  - RAID1 writes are faster than RAID5 since no parity

	Config	Pros	Cons
0	Split data across multiple disks	No overhead, fast read/write	Reliability
1	Mirrored disks: extra copy of data	Fast read/write, fast recovery	High overhead = expensive
2	Bit-level striping, 1 disk per parity group	Smaller overhead	Redundant check disks
3	Byte-level striping, single parity disk	Smallest overhead to check parity	Need to read all disks to detect errors
4	Block-level striping, single parity disk	Higher throughput for small reads	Slow small writes (single check disk)
5	Block-level striping, parity distributed	Higher throughput for small writes	Time to repair disk is long

7.2 ECC

– Error Correction Codes (ECC): Protect soft errors
– **Hamming distance**: Diff in # of bits for valid codewords
– **Parity bit**: Forces stored words to have even parity
  - Placed at binary positions of powers of 2
  - XOR all 1 means error
  - Cannot detect an even number of errors

– **Detection**: bit pattern fails codeword check
– **Correction**: map to nearest valid code word
– Hamming ECC uses 1 index left to right (MSB)



Bit position		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Encoded data bits		p1	p2	d1	p4	d2	d3	d4	p8	d5	d6	d7	d8	d9	d10	d11	p16	d12	d13	d14	d15
Parity bit coverage	p1	✗		✗		✗		✗		✗		✗		✗		✗		✗		✗	
	p2		✗	✗			✗	✗			✗	✗			✗	✗			✗	✗	
	p4				✗	✗	✗	✗					✗	✗	✗	✗					✗
	p8								✗	✗	✗	✗	✗	✗	✗	✗					
	p16																✗	✗	✗	✗	✗