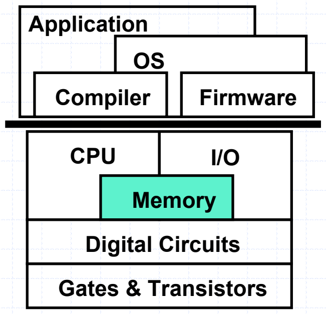
CIS 501
Introduction to Computer Architecture

Unit 4: Memory Hierarchy II: Main Memory

This Unit: Main Memory



- Memory hierarchy review
- Virtual memory
  - Address translation and page tables
  - Virtual memory's impact on caches
  - Page-based protection
- Organizing a memory system
  - Bandwidth matching
  - Error correction

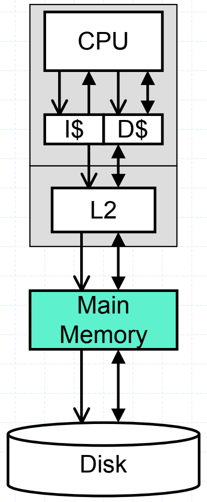
Readings

- P+H
  - Chapter 5.8-5.18
    - Skip Intel Pentium example in 5.11
    - Skim 5.14, 5.15, 5.18

Memory Hierarchy Review

- Storage: registers, **memory**, disk
  - Memory is the fundamental element
- Memory component performance
  - **$t_{avg} = t_{hit} + \%_{miss} * t_{miss}$**
  - Can't get both low t_{hit} and $\%_{miss}$ in a single structure
- Memory hierarchy
  - Upper components: small, fast, expensive
  - Lower components: big, slow, cheap
  - t_{avg} of hierarchy is close to t_{hit} of upper (fastest) component
    - 10/90 rule: 90% of stuff found in fastest component
  - **Temporal/spatial locality**: automatic up-down data movement

Concrete Memory Hierarchy



- 1st/2nd levels: caches (I$, D$, L2)
  - Made of SRAM
  - Last unit
- 3rd level: **main memory**
  - Made of DRAM
  - Managed in software
  - This unit
- 4th level: disk (swap space)
  - Made of magnetic iron oxide discs
  - Manage in software
  - Next unit

Memory Organization

- Paged "virtual" memory
  - Programs what a conceptually view of a memory of unlimited size
  - Use disk as a *backing store* when physical memory is exhausted
  - Memory acts like a cache, managed (mostly) by software
- How is the "memory as a cache" organized?
  - Block size? Pages that are typically 4KB or larger
  - Associativity? Fully associative
  - Replacement policy? In software
  - Write-back vs. write-through? Write-back
  - Write-allocate vs. write-non-allocate? Write allocate

Low $\%_{miss}$ At All Costs

- For a memory component: t_{hit} vs. $\%_{miss}$ tradeoff
- Upper components (I$, D$) emphasize low t_{hit}
  - Frequent access → minimal t_{hit} important
  - t_{miss} is not bad → minimal $\%_{miss}$ less important
  - Low capacity/associativity/block-size, write-back or write-thru
- Moving down (L2) emphasis turns to $\%_{miss}$
  - Infrequent access → minimal t_{hit} less important
  - t_{miss} is bad → minimal $\%_{miss}$ important
  - High capacity/associativity/block size, write-back
- For memory, emphasis entirely on $\%_{miss}$
  - t_{miss} is disk access time (measured in ms, not ns)

Memory Organization Parameters

Parameter	**I$/D$**	**L2**	**Main Memory**
t_{hit}	1-2ns	5-15ns	**100ns**
t_{miss}	**5-15ns**	**100ns**	**10ms (10M ns)**
Capacity	8–64KB	256KB–8MB	**256MB–4GB**
Block size	16–32B	32–256B	**8–64KB pages**
Associativity	1–4	4–16	**Full**
Replacement Policy	LRU/NMRU	LRU/NMRU	**working set**
Write-through?	Either	No	**No**
Write-allocate?	Either	Yes	**Yes**
Write buffer?	Yes	Yes	**No**
Victim buffer?	Yes	No	**No**
Prefetching?	Either	Yes	**Either**

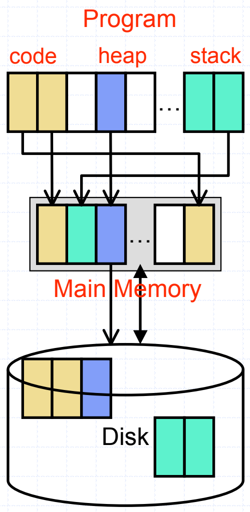
Software Managed Memory

- Isn't full associativity difficult to implement?
  - Yes … in hardware
  - Implement fully associative memory in software
- Let's take a step back…

Virtual Memory

- Idea of treating memory like a cache…
  - Contents are a dynamic subset of program's address space
  - Dynamic content management transparent to program
- Actually predates "caches" (by a little)
- Original motivation: **compatibility**
  - IBM System 370: a family of computers with one software suite
  + Same program could run on machines with different memory sizes
    - Caching mechanism made it appear as if memory was 2^N bytes
    - Regardless of how much there actually was
  – Prior, programmers explicitly accounted for memory size
- **Virtual memory**
  - Virtual: "in effect, but not in actuality" (i.e., appears to be, but isn't)

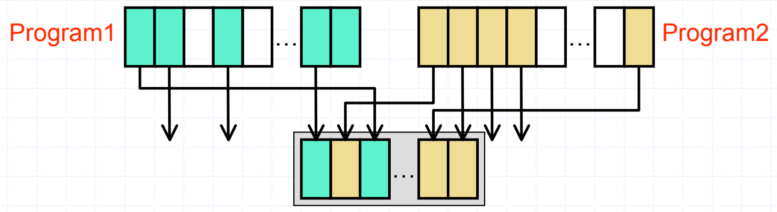
Virtual Memory



- Programs use **virtual addresses (VA)**
  - $0…2^N–1$
  - VA size also referred to as machine size
  - E.g., Pentium4 is 32-bit, Alpha is 64-bit
- Memory uses **physical addresses (PA)**
  - $0…2^M–1$ (typically $M<N$, especially if $N=64$)
  - 2^M is most physical memory machine supports
- VA→PA at **page** granularity (VP→PP)
  - By "system"
  - Mapping need not preserve contiguity
  - VP need not be mapped to any PP
  - Unmapped VPs live on disk (swap)

Uses of Virtual Memory

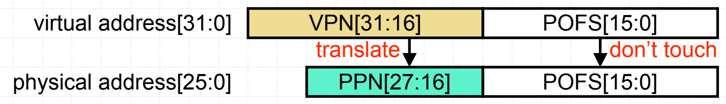
- Virtual memory is quite a useful feature
  - Automatic, transparent memory management just one use
  - "Functionality problems are solved by adding levels of indirection"
- Example: **program isolation and multiprogramming**
  - Each process thinks it has 2^N bytes of address space
  - Each thinks its stack starts at address 0xFFFFFFFF
  - System maps VPs from different processes to different PPs
    + Prevents processes from reading/writing each other's memory



More Uses of Virtual Memory

- **Isolation and Protection**
  - Piggy-back mechanism to implement page-level protection
  - Map virtual page to physical page
    - … and to Read/Write/Execute protection bits in page table
  - In multi-user systems
    - Prevent user from accessing another's memory
    - Only the operating system can see all system memory
  - Attempt to illegal access, to execute data, to write read-only data?
    - Exception → OS terminates program
  - More later
- Inter-process communication
  - Map virtual pages in different processes to same physical page
  - Share files via the UNIX mmap() call

Address Translation



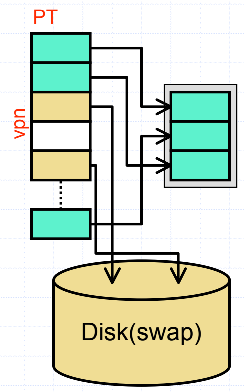
- VA→PA mapping called **address translation**
  - Split VA into **virtual page number (VPN)** and page offset (POFS)
  - Translate VPN into **physical page number (PPN)**
  - POFS is not translated
  - VA→PA = [VPN, POFS] → [PPN, POFS]
- Example above
  - 64KB pages → 16-bit POFS
  - 32-bit machine → 32-bit VA → 16-bit VPN
  - Maximum 256MB memory → 28-bit PA → 12-bit PPN

Mechanics of Address Translation

- How are addresses translated?
  - In software (now) but with hardware acceleration (a little later)
- Each process allocated a **page table (PT)**
  - **Managed by the operating system**
  - Maps VPs to PPs or to disk (swap) addresses
    - VP entries empty if page never referenced
  - Translation is table lookup

```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty;
} PTE;
struct PTE pt[NUM_VIRTUAL_PAGES];

int translate(int vpn) {
  if (pt[vpn].is_valid)
     return pt[vpn].ppn;
}
```



Page Table Size

- How big is a page table on the following machine?
  - 4B page table entries (PTEs)
  - 32-bit machine
  - 4KB pages

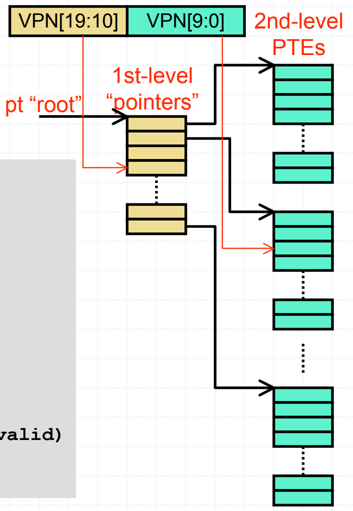
  - 32-bit machine → 32-bit VA → 4GB virtual memory
  - 4GB virtual memory / 4KB page size → 1M VPs
  - 1M VPs * 4B PTE → 4MB
- How big would the page table be with 64KB pages?
- How big would it be for a 64-bit machine?
- Page tables can get big
  - There are ways of making them smaller

Multi-Level Page Table

- One way: **multi-level page tables**
  - Tree of page tables
  - Lowest-level tables hold PTEs
  - Upper-level tables hold pointers to lower-level tables
  - Different parts of VPN used to index different levels
- Example: two-level page table for machine on last slide
  - Compute number of pages needed for lowest-level (PTEs)
    - 4KB pages / 4B PTEs → 1K PTEs/page
    - 1M PTEs / (1K PTEs/page) → 1K pages
  - Compute number of pages needed for upper-level (pointers)
    - 1K lowest-level pages → 1K pointers
    - 1K pointers * 32-bit VA → 4KB → 1 upper level page

Multi-Level Page Table

- 20-bit VPN
  - Upper 10 bits index 1st-level table
  - Lower 10 bits index 2nd-level table

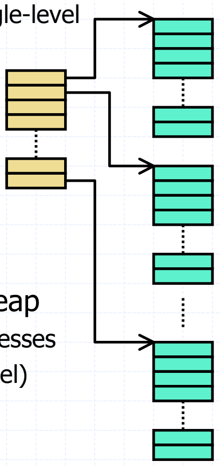


```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty;
} PTE;
struct {
   struct PTE ptes[1024];
} L2PT;
struct L2PT *pt[1024];

int translate(int vpn) {
  struct L2PT *l2pt = pt[vpn>>10];
  if (l2pt && l2pt->ptes[vpn&1023].is_valid)
     return l2pt->ptes[vpn&1023].ppn;
}
```

Multi-Level Page Table (PT)

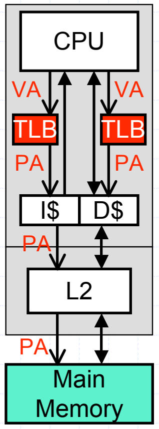
- Have we saved any space?
  - Isn't total size of 2nd level tables same as single-level table (i.e., 4MB)?
  - Yes, but…
- Large virtual address regions unused
  - Corresponding 2nd-level tables need not exist
  - Corresponding 1st-level pointers are null
- Example: 2MB code, 64KB stack, 16MB heap
  - Each 2nd-level table maps 4MB of virtual addresses
  - 1 for code, 1 for stack, 4 for heap, (+1 1st-level)
  - 7 total pages = 28KB (much less than 4MB)



Address Translation Mechanics

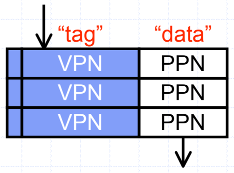
- The six questions
  - What? address translation
  - Why? compatibility, multi-programming, protection
  - How? page table
  - **Who performs it?**
  - **When do you translate?**
  - **Where does page table reside?**
- Conceptual view:
  - Translate virtual address before every cache access
  - Walk the page table for every load/store/instruction-fetch
  - Disallow program from modifying its own page table entries
- Actual approach:
  - Cache translations in a "translation cache" to avoid repeated lookup

Translation Lookaside Buffer



- Functionality problem? add indirection
- Performance problem? add cache
- Address translation too slow?
  - Cache translations in **translation lookaside buffer (TLB)**
    - Small cache: 16–64 entries
    - Often fully associative

  + Exploits temporal locality in page table (PT)
  - What if an entry isn't found in the TLB?
    - Invoke TLB miss handler



TLB Misses and Miss Handling

- **TLB miss:** requested PTE not in TLB, search page table
  - **Software routine**, e.g., Alpha
    - Special instructions for accessing TLB directly
    - Latency: one or two memory accesses + OS call
  - **Hardware finite state machine (FSM),** e.g., x86
    - Store page table root in hardware register
    - Page table root and table pointers are physical addresses

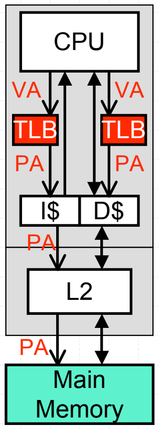
    + Latency: saves cost of OS call
  - In both cases, reads use the the standard cache hierarchy

    + Allows caches to help speed up search of the page table
- **Nested TLB miss**: miss handler itself misses in the TLB
  - Solution #1: Allow recursive TLB misses (very tricky)
  - Solution #2: Lock TLB entries for page table into TLB
  - Solution #3: Avoid problem using physical address in page table

Page Faults

- **Page fault**: PTE not in TLB or page table
  - Page is simply not in memory
  - Starts out as a TLB miss, detected by OS handler/hardware FSM
- **OS routine**
  - Choose a physical page to replace
    - **"Working set"**: more refined software version of LRU
      - Tries to see which pages are actively being used
      - Balances needs of all current running applications
    - If dirty, write to disk
  - Read missing page from disk
    - Takes so long (~10ms), OS schedules another task
  - Treat like a normal TLB miss from here

Physical Caches

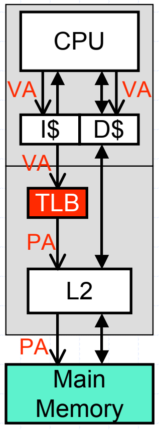


- Memory hierarchy so far: **physical caches**
  - Indexed and tagged by PAs
  - Translate to PA to VA at the outset

  + Cached inter-process communication works
    - Single copy indexed by PA

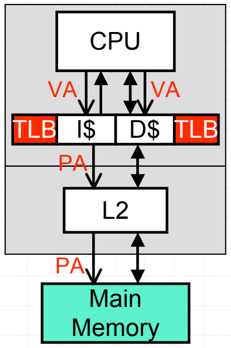
  – Slow: adds at least one cycle to t_{hit}

Virtual Caches



- Alternative: **virtual caches**
  - Indexed and tagged by VAs
  - Translate to PAs only to access L2
  - + Fast: avoids translation latency in common case
  - – Problem: VAs from ***different processes*** are distinct physical locations (with different values)
- What to do on process switches?
  - Flush caches? Slow
  - Add process IDs to cache tags
- Does inter-process communication work?
  - **Aliasing**: multiple VAs map to same PA
    - Can't allow same PA in the cache twice
    - Also a problem for DMA I/O
  - Can be handled, but very complicated

Hybrid: Parallel TLB/Cache Access

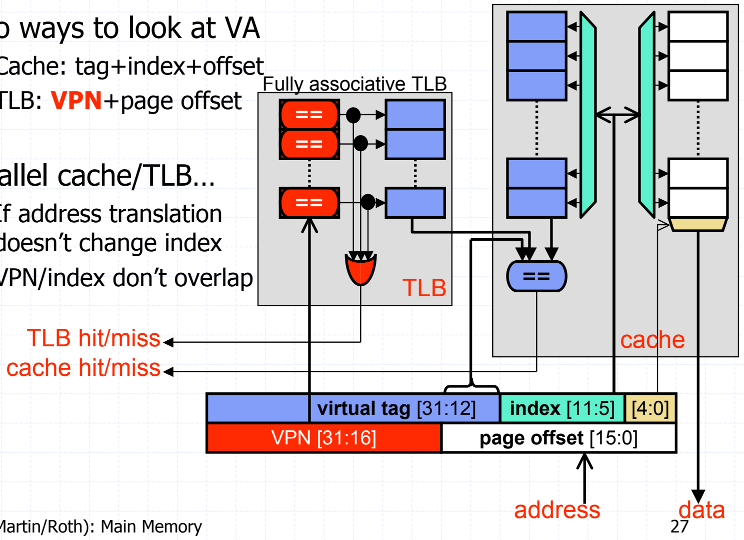


Compromise: **access TLB in parallel**

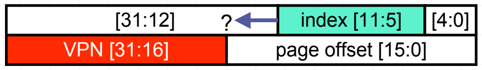
- *In small caches, index of VA and PA the same*
- Use the VA to generate PA index
- Tagged by PAs
- Cache access and address translation in parallel
- + No context-switching/aliasing problems
- + Fast: no additional t_{hit} cycles
- Common organization in processors today
- Note: TLB itself can't use this trick
  - Must have virtual tags
  - Adds process IDs to tags

Parallel Cache/TLB Access

- Two ways to look at VA
  - Cache: tag+index+offset
  - TLB: **VPN**+page offset
- Parallel cache/TLB…
  - If address translation doesn't change index
  - VPN/index don't overlap



Cache Size And Page Size



- Relationship between page size and L1 cache size
  - Forced by non-overlap between VPN and IDX portions of VA
    - Which is required for TLB access
  - **Rule: (cache size) / (associativity) ≤ page size**
  - Result: associativity increases allowable cache sizes
  - Systems are moving towards bigger (64KB) pages
    - To use parallel translation with bigger caches
    - To amortize disk latency
  - Example: Pentium 4, 4KB pages, 8KB, 2-way SA L1 data cache
- If cache is too big, same issues as virtually-indexed caches
- No relationship between page size and L2 cache size

TLB Organization

- **Like caches**: TLBs also have ABCs
  - Capacity
  - Associativity (At least 4-way associative, fully-associative common)
  - What does it mean for a TLB to have a block size of two?
    - Two consecutive VPs share a single tag
- **Like caches**: there can be L2 TLBs
  - Why? Think about this…
- **Rule of thumb**: TLB should "cover" L2 contents
  - In other words: (#PTEs in TLB) * page size ≥ L2 size
  - Why? Think about relative miss latency in each…

Virtual Memory

- Virtual memory ubiquitous today
  - Certainly in general-purpose (in a computer) processors
  - But even many embedded (in non-computer) processors support it
- Several forms of virtual memory
  - **Paging** (aka flat memory): equal sized translation blocks
    - Most systems do this
  - **Segmentation**: variable sized (overlapping?) translation blocks
    - x86 used this rather than 32-bits to break 16-bit (64KB) limit
    - Makes life hell
  - **Paged segments**: don't ask
- How does virtual memory work when system starts up?

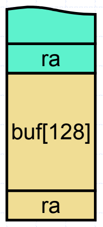
Memory Protection and Isolation

- Most important role of virtual memory today
- Virtual memory protects applications from one another
  - OS uses indirection to isolate applications
  - One buggy program should not corrupt the OS or other programs
  - + Comes "for free" with translation
  - – However, the protection is limited

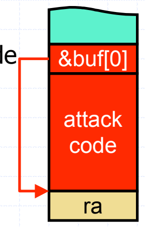
– What about protection from…
  - Viruses and worms?
    - Stack smashing
  - Malicious/buggy services?
    - Other applications with which you want to communicate

Stack Smashing via Buffer Overflow

```
int i = 0;
char buf[128];
while ((buf[i++] = getc()) != '\n') ;
return;
```



- Stack smashing via buffer overflow
  - Oldest trick in the virus book
  - Exploits stack frame layout and…
  - Sloppy code: **length-unchecked copy to stack buffer**
  - "Attack string": `code` (128B) + `&buf[0]` (4B)
  - Caller return address replaced with pointer to attack code
    - Caller return…
    - …executes attack code at caller's privilege level
  - Vulnerable programs: gzip-1.2.4, sendmail-8.7.5



Page-Level Protection

```
struct {
   union { int ppn, disk_block; }
   int is_valid, is_dirty, permissions;
} PTE;
```

- **Page-level protection**
  - Piggy-backs on translation infrastructure
  - Each PTE associated with permission bits: **R**ead, **W**rite, e**X**ecute
    - **Read/execute (RX)**: for code
    - **Read (R)**: read-only data
    - **Read/write (RW)**: read-write data
  - TLB access traps on illegal operations (e.g., write to **RX** page)
  - To defeat stack-smashing? Set stack permissions to **RW**
    - Will trap if you try to execute `&buf[0]`
  - + **X** bits recently added to x86 for this specific purpose
  - – Unfortunately, hackers have many other tricks

Safe and Efficient Services

- Scenario: module (application) A wants service B provides
  - A doesn't "trust" B and vice versa (e.g., B is kernel)
  - How is service provided?
- Option I: conventional call in same address space
  - + Can easily pass data back and forth (pass pointers)
  - – Untrusted module can corrupt your data
- Option II: trap or cross address space call
  - – Copy data across address spaces: slow, hard if data uses pointers
  - + Data is not vulnerable
- Page-level protection helps somewhat, but…
  - Page-level protection can be too coarse grained
  - If modules share address space, both can change protections

Research: Mondriaan Memory Protection

- **Mondriaan Memory Protection (MMP)**
  - Research project from MIT [Witchel+, ASPLOS'00]
  - Separates translation from protection
- MMP translation: as usual
  - One translation structure (page table) per address space
  - Hardware acceleration: TLB caches translations
- MMP protection
  - Protection domains: orthogonal to address spaces
  - Services run in same address space but different protection domains
  - One protection structure (protection table) per protection domain
    - **Protection bits represented at word-level**
    - Two bits per 32-bit word, only 6.25% overhead
  - Hardware acceleration: PLB caches protection bits

Brief History of DRAM

- DRAM (memory): a major force behind computer industry
  - Modern DRAM came with introduction of IC (1970)
  - Preceded by magnetic "core" memory (1950s)
    - More closely resembles today's disks than memory
  - And by mercury delay lines before that (ENIAC)
    - Re-circulating vibrations in mercury tubes

"the one single development that put computers on their feet was the invention of a reliable form of memory, namely the core memory… It's cost was reasonable, it was reliable, and because it was reliable it could in due course be made large"

Maurice Wilkes

Memoirs of a Computer Programmer, 1985

DRAM Technology

- DRAM optimized for density (bits per chip)
  - Capacitor & one transistor
    - In contrast, SRAM has six transistors
  - Capacitor stores charge (for 1) or no charge (for 0)
  - Transistor controls access to the charge
  - Analogy of balloon + valve
- Destructive read
  - Sensing if the capacitor is charged or not destroy value
  - Solution: every read is immediately followed by a write
- Refresh
  - Charge leaks away
  - Occasionally read then write back the values
  - Not needed for SRAM
- High latency (50ns to 150ns)

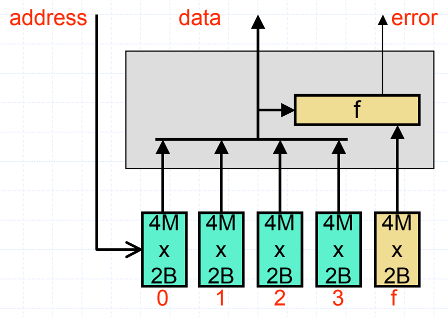
DRAM Bandwidth

- Use multiple DRAM chips to increase bandwidth
  - Recall, access are the same size as second-level cache
  - Example, 16 2-byte wide chips for 32B access
- DRAM density increasing faster than demand
  - Result: number of memory chips per system decreasing
- Need to increase the **bandwidth per chip**
  - Especially important in game consoles
  - SDRAM ➔ DDR ➔ DDR2
  - Rambus - high-bandwidth memory
    - Used by several game consoles

DRAM Reliability

- One last thing about DRAM technology… **errors**
  - DRAM fails at a higher rate than SRAM (CPU logic)
    - Very few electrons stored per bit
    - Bit flips from energetic α-particle strikes
    - Many more bits
  - Modern DRAM systems: built-in error detection/correction
    - Today all servers; desktop and laptops soon
- **Key idea: checksum-style redundancy**
  - Main DRAM chips store data, additional chips store f(data)
    - $|f(data)| < |data|$
  - On read: re-compute f(data), compare with stored f(data)
    - Different ? Error…
  - Option I (**detect**): kill program
  - Option II (**correct**): enough information to fix error? fix and go on

DRAM Error Detection and Correction



- Performed by memory controller (not the DRAM chip)
- Error detection/correction schemes distinguished by…
  - How many (simultaneous) errors they can detect
  - How many (simultaneous) errors they can correct

Error Detection: Parity

- **Parity**: simplest scheme
  - $f(data_{N-1\ldots0}) = XOR(data_{N-1}, \ldots, data_1, data_0)$
  - + Single-error detect: detects a single bit flip (common case)
    - Will miss two simultaneous bit flips…
    - But what are the odds of that happening?
  - – Zero-error correct: no way to tell which bit flipped

Error Correction: Hamming Codes

- **Hamming Code**
  - H(A,B) = number of 1's in A^B (number of bits that differ)
    - Called "Hamming distance"
  - Use D data bits + C check bits construct a set of "codewords"
    - Check bits are parities on different subsets of data bits
  - $\forall$codewords A,B H(A,B) $\geq \alpha$
    - No combination of $\alpha-1$ transforms one codeword into another
    - For simple parity: $\alpha = 2$
  - Errors of δ bits (or fewer) can be detected if $\alpha = \delta + 1$
  - Errors of β bits or fewer can be corrected if $\alpha = 2\beta + 1$
  - Errors of δ bits can be detected and errors of β bits can be corrected if $\alpha = \beta + \delta + 1$

SEC Hamming Code

- **SEC**: single-error correct
  - $C = \log_2 D + 1$
  - + Relative overhead decreases as D grows
- Example: D = 4 → C = 3
  - $d_1\ d_2\ d_3\ d_4\ \mathbf{c_1\ c_2\ c_3} \rightarrow \mathbf{c_1\ c_2}\ d_1\ \mathbf{c_3}\ d_2\ d_3\ d_4$
  - $c_1 = d_1 \wedge d_2 \wedge d_4$, $c_2 = d_1 \wedge d_3 \wedge d_4$, $c_3 = d_2 \wedge d_3 \wedge d_4$
  - Syndrome: $c_i \wedge c'_i = 0$? no error : points to flipped-bit
- Working example
  - Original data = 0110 → $c_1 = 1$, $c_2 = 1$, $c_3 = 0$
  - Flip d_2 = 0010 → $c'_1 = 0$, $c'_2 = 1$, $c'_3 = 1$
    - Syndrome = 101 (binary 5) → 5th bit? D_2
  - Flip c_2 → $c'_1 = 1$, $c'_2 = 0$, $c'_3 = 0$
    - Syndrome = 010 (binary 2) → 2nd bit? c_2

SECDED Hamming Code

- **SECDED**: single error correct, double error detect
  - $C = \log_2 D + 2$
  - Additional parity bit to detect additional error
- Example: D = 4 → C = 4
  - $d_1\ d_2\ d_3\ d_4\ \mathbf{c_1\ c_2\ c_3} \rightarrow \mathbf{c_1\ c_2}\ d_1\ \mathbf{c_3}\ d_2\ d_3\ d_4\ \mathbf{c_4}$
  - $c_4 = c_1 \wedge c_2 \wedge d_1 \wedge c_3 \wedge d_2 \wedge d_3 \wedge d_4$
  - Syndrome == 0 and $c'_4 == c_4$ → no error
  - Syndrome != 0 and $c'_4 != c_4$ → 1-bit error
  - Syndrome != 0 and $c'_4 == c_4$ → 2-bit error
  - Syndrome == 0 and $c'_4 != c_4$ → c_4 error
- Many machines today use 64-bit SECDED code
  - C = 8 (one additional byte, 12% overhead)
  - ChipKill - correct any aligned 4-bit error
    - If an entire DRAM chips dies, the system still works!

Summary

- Virtual memory
  - Page tables and address translation
  - Virtual, physical, and hybrid caches
  - TLBs
- DRAM technology
- Error detection and correction
  - Parity and Hamming codes