Summary: Cache

- **Cache basics**
  - Memory-processor speed gap motivates “cache”
  - A subset of memory contents that are frequently referenced is stored in cache in “blocks”
  - Why does it work?
    - If size ratio is 5%, hit ratio is 5% in random order of references
    - In fact, it is about 95% due to “locality”

- **Three cache policies**
  - Where to put a memory block in cache: “Placement policy”
    - Direct-mapped: only one position
    - N-way set associative: N positions
    - Fully associative: any position
  - Which block to kill: “Replacement policy”
    - Direct: not necessary
    - N-way/Fully associate: “LRU (least recently used)” block among the set
  - “Write policy”: Write back or write through

- **Performance improvement**
  - Miss ratio: Fully associative, 2-level cache, Big cache size, Big block size
  - Hit time: Direct, Small cache size
  - Miss penalty: Small block size
Summary: Cache

- Memory address decomposition

<table>
<thead>
<tr>
<th>Cache</th>
<th>Tag</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

Direct-mapped

<table>
<thead>
<tr>
<th>tag</th>
<th>block index</th>
<th>block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

N-way set associative

<table>
<thead>
<tr>
<th>tag</th>
<th>set number</th>
<th>block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Fully associative

(index & match)

N-way set associative

(Fully associative (need to match with all tags and many comparators!))
Summary: Cache

- 2 Questions

- **Q1:** How about a mapping table from the memory side (one entry per memory block)?

<table>
<thead>
<tr>
<th>Tag</th>
<th>Valid</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Which memory block is this mapped from?**

<table>
<thead>
<tr>
<th>Mapping table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

- **Q2:** How about this address decomposition?

<table>
<thead>
<tr>
<th>block index</th>
<th>tag</th>
<th>block offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Which cache block is this mapped to?**

Contents

- Virtual memory
- Page table and TLB
- Interaction of virtual memory with cache
Virtual Memory: Motivation

- Historically, motivations for VM are
  - Allow efficient and safe sharing of memory among multiple programs
  - Remove the programming burdens of a small, limited amount of main memory
    - Programmers make overlays and load/unload manually
    - Not that important now
  - A number of programs are running concurrently
    - The total memory required to run all the programs exceeds the main memory available
    - But since only a small fraction of this memory is actively being used at any point of time, we can “cache” them in main memory and put others in disk
    - I.e., use disk to simulate more memory!
  - Multiple users on same computer
    - Want object code to have fixed addresses.
    - How to relocate multiple users’ programs into same memory?
    - How to protect one user program from the other?
    - Can automatically relocate code at run-time – separate address space per program!

Most common solution is virtual memory.

Can address more virtual memory than physical memory.
- So, think of disk as the “usual” place to store VM.
- Then, physical memory is mainly a cache for the VM on the disk.

So, VM very similar to caches.
- Different motivation.
- Different evolution → different terminology.

VM page = cache block.
VM page fault = cache miss.
Virtual Memory: Motivation

Virtual Memory $\approx$ Cache

Willing to do a lot to minimize access to the VERY SLOW disk.

Disk access: $\approx 100K$-$1M$ cycles!

Choose caching strategies which minimize disk access, & thus misses:

- Fully associative
- Write-back
  - Reduce disk access on writes by grouping them.
- Approximate LRU
  - Exact LRU too expensive.
- Fairly large pages (typically 4-64KB)
  - Amortize high access time. Not too large to lose good spatial locality.
Virtual Memory: Mapping (= Block Placement)

• Main memory can act as a cache for the secondary storage (disk)

  This is based on mapping table or “page table”

• Why fully associate?
  – Huge miss penalty (100k-1M cycles): the data is not in memory, retrieve it from disk; it is called Page faults
  – Pages should be fairly large (e.g., 4KB)
  – LRU is worth the price
  – Handle the faults in software instead of hardware
  – Writeback

Page Tables for Address Translation (VA to PA)

Fully associative: can be placed anywhere!
Page Tables for Address Translation (VA to PA)

![Diagram of page table with annotations]

1M entries x 32 bits = 4MB !!!
(page table size)

A Performance Problem

Every VM access uses multiple memory accesses:
1 for page table (or more if multi-level).
1 for data.

What’s our general method for improving memory access performance?

Caches. Let’s add a cache for the page table!
Translation Look-aside Buffer (TLB)

TLB = Hardware cache for page table access.

Only one TLB, not one per process.
Either caches for all processes or only for current process.
Will see issues...

Only cache page table entries which map to physical memory.
Speed up the common case.
Disk access is VERY SLOW – an additional page table access is negligible.

Low miss rate because of locality & large page size.
Usually write-back to minimize memory access.

Making Address Translation Fast

- A cache for address translations: translation lookaside buffer

TLB size = 32–4096 entries
Tag size = 20-bit (V.PN) if fully associative
Example: DECStation 3100 (MIPS R2000)

- **Virtual Memory**
  - Page size is 256 bytes ($2^{8}$)
  - Virtual memory is 2048 bytes ($2^{11}$, 0~0x7ff) -> $2^{3} = 8$ pages
  - (Physical) Memory has 1024 bytes ($2^{10}$, 0~0x3ff) -> $2^{2} = 4$ pages

- **TLB**
  - TLB has 2 entries

- **Cache**
  - Block size is 16 bytes of data ($2^{4}$, 0~0xf)
  - Memory has 1024 bytes ($2^{10}$, 0~0x3ff) -> $2^{6} = 64$ blocks
  - Cache has 256 bytes ($2^{8}$, 0~0xff) -> $2^{4} = 16$ blocks
  - “Direct mapped“
Virtual Memory

256 bytes

Memory Address
000
100
200
300
400
500
600
700

Page number (VPN)
0
1
2
3
4
5
6
7

Page number (PPN)
0
1
2
3
4
5
6
7

Address decomposition?

Page Table

Page number
0
1
2
3
4
5
6
7

VPN
Valid Dirty
0
0
0
0
1
1
0
0

Direct-mapped cache

Memory access [376]

Cache

16 bytes

Memory Address
00
10
20
30
40
50
60
70
80
90
A0
B0
C0
D0
E0
F0

Direct-mapped cache

Tag
Valid
0
0
0
0
1
1
0
0
0
0
1
1
0
0
0
0

Address decomposition?

16 bytes

Block Index
0
1
2
3
4
5
6
7
8
9
A
B
C
D
E
F

Memory
Where to put Page Table?

Memory Address
000 100 200 300 400 500 600 700

Page number (VPN)
0 1 2 3 4 5 6 7

256 bytes

Expensive memory access to access PT => TLB

Memory Address
000 100 200 300 400 500 600 700

Page number (VPN)
0 1 2 3 4 5 6 7

256 bytes

Memory

Virtual Memory

256 bytes

200 300

_Xxx, 210, Xxx, 010,
Xxx, Xxx, Xxx, 111

Valid Dirty

PTR

Expensive memory access to access PT => TLB

Memory

Virtual Memory

256 bytes

200 300

_Xxx, 210, Xxx, 010,
Xxx, Xxx, Xxx, 111

Valid Dirty

TLB
(Translation Lookaside Buffer)
Now, the questions...

- Virtual address (VA) [776]
  - TLB hit, translated to [176]
  - Cache hit (block index=7, tag=1)

- VA [133]
  - TLB hit, translated [233]
  - Cache miss (block index=3, tag=2)

- VA [309]
  - TLB miss, PT (page table) lookup, translated to [009]
  - Cache hit (block index=0, tag=0)

- VA [452]
  - TLB miss, PT lookup, Page fault

---

**Cache & VM Interaction**

Virtual address = VPN + Page offset

- VPN in TLB
  - TLB miss
    - Access memory to get PPN from PT
      - Page fault
        - Access disk to get PPN
          - Update PT
          - Update TLB
    - TLB hit
      - Obtain PPN from TLB
      - Physical address = PPN + Page offset
        - Tag + Index + Block offset
      - Block in cache
        - Cache hit
          - Obtain the block
            - Obtain the desired data
          - Update cache
      - Cache miss
        - Access memory to get the block
          - TLB Page Cache

<table>
<thead>
<tr>
<th>TLB</th>
<th>Page</th>
<th>Cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hit</td>
<td>Hit</td>
<td>Hit</td>
</tr>
<tr>
<td>Hit</td>
<td>Hit</td>
<td>Miss</td>
</tr>
<tr>
<td>Hit</td>
<td>Miss</td>
<td>Hit</td>
</tr>
<tr>
<td>Miss</td>
<td>Hit</td>
<td>Hit</td>
</tr>
<tr>
<td>Miss</td>
<td>Hit</td>
<td>Miss</td>
</tr>
<tr>
<td>Miss</td>
<td>Miss</td>
<td>Hit</td>
</tr>
<tr>
<td>Miss</td>
<td>Miss</td>
<td>Miss</td>
</tr>
</tbody>
</table>
Cache & VM Interaction: Details of a Single Access

Access TLB.
Hit? →
   TLB gives physical address of data.
   No need to look at page table.

   Access physical memory cache.  (Assume one level cache for brevity.)
   Hit? →
      Cache gives data value.
      **Best case.  No memory or disk access at all!**

   Miss? →
      Access memory for data value.
      Update cache.

...TLB Miss? →

   Access page table in physical memory.
      (Assume single page table for brevity.)
      (Page table entries could be in cache.  This possibility omitted for brevity.)

   No page fault? →
      Page table gives physical memory address for data value.
      Update TLB.
      Access physical memory cache.
      Hit? →
         Cache gives data value.
      Miss? →
         Access memory for data value.
         Update cache.
Cache & VM Interaction:
Details of a Single Access

...  
TLB Miss? →  
...  
Page fault? →  
Page table gives disk location for data value.  
Access disk for data value.  
(Disks usually have caches too! This possibility omitted for brevity.)  
Update memory with this page.  
Update page table & TLB.  
**Worst case. Both memory & disk access!**

Memory Performance

- Most memory accesses require a sequence of two cache accesses
  - TLB cache access
  - Cache access
- How can it be faster?
  - Overlapping TLB & Cache access
Cache & VM Interaction (revisited)

Virtual address = VPN + Page offset

Physical address = PPN + Page offset = Tag + Index + Block offset

Access disk to get PPN
Update PT
Update TLB

Access memory
To get the block
Update cache

Problems With Overlapped TLB Access

Overlapped access only works as long as the address bits used to index into the cache do not change as the result of VA translation

Overlapped access only works as long as the address bits used to index into the cache do not change as the result of VA translation

20
VPN
Page Off.

12

PPN
Page Off.

Tag
Block Index
Block Offset

These bits are changed by VA translation, but is needed for cache lookup

VPN
Page Off.

PPN
Page Off.

Tag
Block Index
Block Offset

Bits for block index doesn’t change during address translation

Can happen in parallel

(PPN from TLB, which is in fact tag, will be compared with cache tag.)
Questions

- Q1: How to expedite cache access?
  - Virtual addressed cache
- Q2: Page table per process? - Yes
- Q2: TLB per process? – No
  - Which means that we need to flush TLB upon a context switch
  - How to avoid that? : ASID (Address space identifier), ASN (AS number), PID (Process ID)
- Q3: Large PT size
  - 4GB VM (2^32) with 4KB page (2^12) gives 2^20 PTEs (page table entries)
  - 4 bytes per PTE means 4MB per PT
  - 100 processes means 400MB for 100 PTs
- Q4: Page size
- Q5: Page fault (page replacement)
- Q6: Unit of data transfer

Q1: Virtual Cache

Which is better?

VM allows processes to share physical memory locations. Could have multiple cache entries for single memory location.

More typical. Also allows page tables to be cached.
Virtual Cache

TLB access is required only on cache miss!!
(most memory access requires one cache access)

**synonym problem:** two different virtual addresses map to same physical address => two different cache entries holding data for the same physical address!

nightmare for update: must update all cache entries with same physical address or memory becomes inconsistent

determining this requires significant hardware, essentially an associative lookup on the physical address tags to see if you have multiple hits. (usually disallowed by fiat)

or

software enforced alias boundary: same lsb of VA & PA > cache size

Possible but simple solution:
clear cache when context switching

---

Virtual Cache

<table>
<thead>
<tr>
<th>Process 1’s virtual address space</th>
<th>Physical Memory</th>
<th>Process 2’s virtual address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared_var 100</td>
<td>70</td>
<td>Shared_var 200</td>
</tr>
</tbody>
</table>

No problem with physical cache
100 -> 70 -> access
200 -> 70 -> access

Problem with virtual cache
100 -> access
200 -> access

Cache will contain two different objects for the same data object !!!
Virtual Memory & Context Switch

Why context switching?

Upon an I/O operation, CPU cannot proceed further until I/O completes (stalls)
Allow CPU to execute other processes while waiting
When multiple users are supported
   CPU allocates its time to multiple users in a time-shared fashion

Context switch → Save current process’ state. Restore next process’ state.

What are the state – PC, Registers, Memory (Page Table), and TLB

Saving & restoring registers:
   Use memory to store all registers, including pointer to page table.
   Very expensive!

Memory (Page Table):
   Just save/restore PTR rather than save/restore the entire PT

TLB caches for only one process’ page table? → Clear TLB. Increases compulsory misses.
TLB caches for all processes’ page tables? → Process id (or its page table pointer) must be provided to TLB.
How to deal with more than one Page Tables?

- Two PTs reside in memory
- When context switching to process 2
  - PT must be replaced
  - Just change PTR value
  - I.e., $\text{PTR} = \text{PTR}_2$

- How about TLB?
  - Only one TLB for all processes
  - Is the first entry for the 1st process or the second process?
  - Need to flush TLB when context switching

Q2+: How to avoid flushing TLB upon a context switch?

- How to avoid flushing TLB upon a context switch?
  - ASID (Address space identifier)
  - ASN (AS number)
  - PID (Process ID)
MIPS R3000

Pipeline

<table>
<thead>
<tr>
<th>Inst Fetch</th>
<th>Dcd / Reg</th>
<th>ALU / E.A.</th>
<th>Memory</th>
<th>Write Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>TLB</td>
<td>I-Cache</td>
<td>RF</td>
<td>Operation</td>
<td>WB</td>
</tr>
<tr>
<td>E.A.</td>
<td>TLB</td>
<td>D-Cache</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TLB

64 entry, on-chip, fully associative, software TLB fault handler

Virtual Address Space

<table>
<thead>
<tr>
<th>ASID</th>
<th>V. Page Number</th>
<th>Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>20</td>
<td>12</td>
</tr>
</tbody>
</table>

0xx User segment (caching based on PT/TLB entry)
100 Kernel physical space, cached
101 Kernel physical space, uncached
11x Kernel virtual space

Allows context switching among
64 user processes without TLB flush

Alpha 21164

Page size =
VM space =
PM space =
No. entries in TLB =
TLB access method =
Q3: Page Table Overhead

- 4MB page table size per each process => 400MB for 100 processes
- Bound register
  - That limits the size of the page table for a given process
  - If VPN becomes larger than the bound register, entries must be added to the PT
- Hashing function with inverted page table => next slide
- Multiple levels of page tables => next slide
- Allow page tables to be paged (PT also in virtual address space)

Large Address Spaces

Two-level Page Tables

32-bit address:

<table>
<thead>
<tr>
<th>P1 index</th>
<th>P2 index</th>
<th>page offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

1K PTEs

4 bytes

4KB

2 GB virtual address space

4 MB of PTE2
  - paged, holes

4 KB of PTE1

What about a 48-64 bit address space?
Inverted Page Tables

IBM System 38 (AS400) implements 64-bit addresses.
48 bits translated
start of object contains a 12-bit tag

=> TLBs or virtually addressed caches are critical

Q4: Optimal Page Size

° Minimize wasted storage
  — small page minimizes internal fragmentation
  — small page increase size of page table
° Minimize transfer time
  — large pages (multiple disk sectors) amortize access cost
  — sometimes transfer unnecessary info
  — sometimes prefetch useful data
  — sometimes discards useless data early
General trend toward larger pages because
  — big cheap RAM
  — increasing mem / disk performance gap
  — larger address spaces
Q5: Page Fault: What happens when you miss?

- Not talking about TLB miss
  - TLB is HWs attempt to make page table lookup fast (on average)
- Page fault means that page is not resident in memory
- Hardware must detect situation
- Hardware cannot remedy the situation
- Therefore, hardware must trap to the operating system so that it can remedy the situation
  - pick a page to discard (possibly writing it to disk)
  - load the page in from disk
  - update the page table
  - resume to program so HW will retry and succeed!
- What is in the page fault handler?
- What can HW do to help it do a good job?

=> “Context switching” during disk read

Page Replacement: “Not” Recently Used (1-bit LRU, Clock)

Associated with each page is a reference flag such that
ref flag = 1 if the page has been referenced in recent past
= 0 otherwise

-- if replacement is necessary, choose any page frame such that its reference bit is 0. This is a page that has not been referenced in the recent past

\[
\begin{array}{c|c|c}
\text{Dirty/Used} & \text{Page Table Entry} \\
\hline
1 & 0 & \text{Page Table Entry} \\
1 & 0 & \\
1 & 0 & \\
0 & \\
0 & \\
0 & \\
\end{array}
\]

page fault handler:
- last replaced pointer (lrp)
  - if replacement is to take place, advance lrp to next entry (mod table size) until one with a 0 bit is found; this is the target for replacement; As a side effect, all examined PTE's have their reference bits set to zero.

Or search for the a page that is both not recently referenced AND not dirty.

Architecture part: support dirty and used bits in the page table

=> may need to update PTE on any instruction fetch, load, store

How does TLB affect this design problem? Software TLB miss?
Q6: Unit of data transfer

- CPU – Cache : word
- Cache – Memory : block
- Memory – Disk : page

Multilevel On-Chip Caches

Intel Nehalem 4-core processor

Per core: 32KB L1 I-cache, 32KB L1 D-cache, 512KB L2 cache
2-Level TLB Organization

<table>
<thead>
<tr>
<th></th>
<th>Intel Nehalem</th>
<th>AMD Opteron X4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Virtual addr</td>
<td>48 bits</td>
<td>48 bits</td>
</tr>
<tr>
<td>Physical addr</td>
<td>44 bits</td>
<td>48 bits</td>
</tr>
<tr>
<td>Page size</td>
<td>4KB, 2/4MB</td>
<td>4KB, 2/4MB</td>
</tr>
<tr>
<td>L1 TLB (per core)</td>
<td>L1 I-TLB: 128 entries for small pages, 7 per thread (2×) for large pages L1 D-TLB: 64 entries for small pages, 32 for large pages Both 4-way, LRU replacement</td>
<td>L1 I-TLB: 48 entries L1 D-TLB: 48 entries Both fully associative, LRU replacement</td>
</tr>
<tr>
<td>L2 TLB (per core)</td>
<td>Single L2 TLB: 512 entries 4-way, LRU replacement</td>
<td>L2 I-TLB: 512 entries L2 D-TLB: 512 entries Both 4-way, round-robin LRU</td>
</tr>
<tr>
<td>TLB misses</td>
<td>Handled in hardware</td>
<td>Handled in hardware</td>
</tr>
</tbody>
</table>

Summary: Why virtual memory?

- **Generality**
  - ability to run programs larger than size of physical memory
- **Storage management**
  - allocation/deallocation of variable sized blocks is costly and leads to (external) fragmentation
- **Protection**
  - regions of the address space can be R/O, Ex, . . .
- **Flexibility**
  - portions of a program can be placed anywhere, without relocation
- **Storage efficiency**
  - retain only most important portions of the program in memory
- **Concurrent I/O**
  - execute other processes while loading/dumping page
- **Expandability**
  - can leave room in virtual address space for objects to grow.
- **Performance**

Observe: impact of multiprogramming, impact of higher level languages