VIRTUALIZING MEMORY: FASTER WITH TLBS

Questions answered in this lecture:

Review paging...
How can page translations be made faster?
What is the basic idea of a TLB (Translation Lookaside Buffer)?
What types of workloads perform well with TLBs?
How do TLBs interact with context-switches?

ANNOUNCEMENTS

• P1: Due tomorrow at 6pm
  • Create README file in your p1 directory: describe what you did a little bit (especially if you ran into problems and did not implement something). The most important bit, at the top, however, should be the authorship of the project.

• Late handin directory for unusual circumstances + communicate

• Project 2: Available by Monday; will announce
  • Due three weeks from tomorrow
  • Can work with project partner on PART 2 in your discussion section (unofficial)
  • Two parts:
    • Linux: Shell -- fork() and exec(), job control
    • Xv6: Scheduler – simplistic MLFQ with graph
    • Two discussion videos again; watch early and often!
  • Communicate with your project partner!
  • Form on course web page if you would like project partner assigned

• Exam 1: No conflicts, no alternate exam time

• Reading for today: Chapter 19
**REVIEW: PAGING**

Assume 4 KB pages

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x0000</td>
<td>load 0x0800</td>
</tr>
<tr>
<td>load 0x1444</td>
<td>load 0x0808</td>
</tr>
<tr>
<td>load 0x2444</td>
<td>load 0x0244</td>
</tr>
<tr>
<td>load 0x1444</td>
<td>load 0x0008</td>
</tr>
<tr>
<td>load 0x5444</td>
<td>load 0x0544</td>
</tr>
</tbody>
</table>

What do we need to know?
- Location of page table in memory (ptbr)
- Size of each page table entry (assume 8 bytes)

**REVIEW: PAGING PROS AND CONS**

**Advantages**
- No external fragmentation
  - don't need to find contiguous RAM
- All free pages are equivalent
  - Easy to manage, allocate, and free pages

**Disadvantages**
- Page tables are too big
  - Must have one entry for every page of address space
- Accessing page tables is too slow [today's focus]
  - Doubles number of memory references per instruction
TRANSLATION STEPS

H/W: for each mem reference:

1. extract **VPN** (virt page num) from **VA** (virt addr) *(cheap)*
2. calculate addr of **PTE** (page table entry) *(cheap)*
3. read **PTE** from memory *(expensive)*
4. extract **PFN** (page frame num) *(cheap)*
5. build **PA** (phys addr) *(cheap)*
6. read contents of **PA** from memory into register *(expensive)*

Which expensive step will we avoid in today's lecture?

3) Don't always have to read PTE from memory!

EXAMPLE:
ARRAY ITERATOR

```c
int sum = 0;
for (i=0; i<N; i++){
    sum += a[i];
}
```

What virtual addresses? Assume these physical addresses

- load 0x3000
- load 0x3004
- load 0x3008
- load 0x300C
- load 0x1000
- load 0x7000
- load 0x1004
- load 0x7004
- load 0x1008
- load 0x7008
- load 0x100C
- load 0x700C
- ...

Assume 'a' starts at 0x3000
Ignore instruction fetches

4KB pages

Observation:
Repeatedly access same PTE because program repeatedly accesses same virtual page

Aside: What can you infer?
- ptbr: 0x1000; PTE 4 bytes each
- VPN 3 -> PPN 7
**STRATEGY: CACHE PAGE TRANSLATIONS**

CPU

Translation Cache

RAM

PT

Some popular entries

<table>
<thead>
<tr>
<th>CPU</th>
<th>RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Translation Cache</td>
<td>PT</td>
</tr>
</tbody>
</table>

memory interconnect

**TLB**: Translation Lookaside Buffer (yes, a poor name!)

**TLB ORGANIZATION**

<table>
<thead>
<tr>
<th>TLB Entry</th>
<th>Tag (virtual page number)</th>
<th>Physical page number (page table entry)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Various ways to organize a 16-entry TLB (artificially small)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

Index

= 0

1

2

3

4

5

6

7

Set

= Two-way set associative

= Four-way set associative

= Fully associative

**Lookup**

- Calculate set (tag % num_sets)
- Search for tag within resulting set
**TLB EXAMPLE**

Various ways to organize a 16-entry TLB (artificially small)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x38</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two-way set associative

30 % 8 = ?

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>0x38</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Four-way set associative

30 % 4 = ?

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x38</td>
<td>0xbe</td>
<td>0x11</td>
<td>0x21</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Direct mapped

30 % 16 = ?

**Lookup**

- Calculate set (tag % num_sets)
- Search for tag within resulting set

---

**TLB: REPLACE ENTRY**

Various ways to organize a 16-entry TLB (artificially small)

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x38</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Two-way set associative

30 % 8 = ?

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x38</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td>0x38</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Four-way set associative

30 % 4 = ?

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0x38</td>
<td>0xbe</td>
<td>0x11</td>
<td>0x21</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Direct mapped

30 % 16 = ?

**Lookup**

- Calculate set (tag % num_sets)
- Search for tag within resulting set
TLB ASSOCIATIVITY TRADE-OFFS

Higher associativity
  + Better utilization, fewer collisions
    – Slower
    – More hardware

Lower associativity
  + Fast
  + Simple, less hardware
    – Greater chance of collisions

TLBs usually fully associative

ARRAY ITERATOR (W/ TLB)

```c
int sum = 0;
for (i = 0; i < 2048; i++) {
    sum += a[i];
}
```

Assume following virtual address stream:
load 0x1000
load 0x1004
load 0x1008
load 0x100C
...

What will TLB behavior look like?
**TLB ACCSESSES: SEQUENTIAL EXAMPLE**

**CPU's TLB**

<table>
<thead>
<tr>
<th>Valid</th>
<th>VPN</th>
<th>PPN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**PTBR**

<table>
<thead>
<tr>
<th>P1 P1</th>
<th>P1 P1</th>
</tr>
</thead>
</table>

**PT 0 KB**

- P1

**PT 4 KB**

- P1

**PT 8 KB**

- P1

**PT 12 KB**

- P2

**PT 16 KB**

- P2

**PT 20 KB**

- P1

**PT 24 KB**

- P2

**PT 28 KB**

- P2

**Virt**

- load 0x1000
- load 0x1004
- load 0x1008
- load 0x100c
- load 0x2000
- load 0x2004

**Miss**

- load 0x0000
- load 0x0004
- load 0x0008
- load 0x5000
- load 0x5004
- load 0x5008
- load 0x500c

**Phys**

- load 0x0000
- load 0x0004
- load 0x0008
- load 0x5000
- load 0x5004
- load 0x5008
- load 0x500c

**PERFORMANCE OF TLB?**

```c
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

Calculate miss rate of TLB for data (ignore code + sum)

# TLB misses / # TLB lookups

**# TLB lookups?**

- number of accesses to array a[]
- = 2048

**# TLB misses?**

- number of unique pages accessed
- = 2048 / (elements of a[] per 4K page)
- = 2K / (4KB / sizeof(int)) = 2K / 1K = 2

**Miss rate?**

- 2/2048 = 0.1%

**Hit rate? (1 – miss rate)**

- 99.9%

**Would hit rate get better or worse with smaller pages?**

Worse
TLB PERFORMANCE

How can system improve TLB performance (hit rate) given fixed number of TLB entries?

Increase page size
  Fewer unique page translations needed to access same amount of memory

TLB Reach:
  Number of TLB entries * Page Size

BREAK

- What did you do this summer?
- What was the best summer job you’ve ever had?
**TLB PERFORMANCE WITH WORKLOADS**

Sequential array accesses almost always hit in TLB
- Very fast!

What access pattern will be slow?
- Highly random, with no repeat accesses

**WORKLOAD ACCESS PATTERNS**

```c
int sum = 0;
for (i=0; i<2048; i++) {
    sum += a[i];
}
```

**Workload A**

```c
int sum = 0;
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
srand(1234);
for (i=0; i<1000; i++) {
    sum += a[rand() % N];
}
```

**Workload B**

- Spatial Locality
- Temporal Locality
- Repeated Random Accesses
**WORKLOAD LOCALITY**

**Spatial Locality**: future access will be to nearby addresses

**Temporal Locality**: future access will be repeats to the same data accessed in recent time

What TLB characteristics are best for each type?

Spatial:
- Access same page next; need same vpn->ppn translation
- Same TLB entry re-used (just 1 TLB entry could be fine!)

Temporal:
- Access same address near in future
- Same TLB entry re-used in near future
- How near in future? How many TLB entries are there?

**TLB REPLACEMENT POLICIES**

**LRU**: evict Least-Recently Used TLB slot when needed
(More on LRU later in policies next week)

**Random**: Evict randomly choosen entry

Which is better?

![TLB Replacement Policies Diagram]
**LRU TROUBLES**

Workload repeatedly accesses same offset across 5 pages (strided access), but only 4 TLB entries

What will TLB contents be over time?
How will TLB perform?

---

**TLB REPLACEMENT POLICIES**

**LRU**: evict Least-Recently Used TLB slot when needed
(More on LRU later in policies next week)

**Random**: Evict randomly choosen entry

Sometimes random is better than a “smart” policy!
CONTEXT SWITCHES

What happens if a process uses cached TLB entries from another process?

Solutions?

1. Flush TLB on each context switch
   - Costly; lose all recently cached translations, more misses

2. Track which entries are for which process
   - Address Space Identifier
   - Tag each TLB entry with an 8-bit ASID
     - how many ASIDs do we get?

TLB EXAMPLE WITH ASID

<table>
<thead>
<tr>
<th>Virtual</th>
<th>Physical</th>
</tr>
</thead>
<tbody>
<tr>
<td>load 0x1444</td>
<td>ASID: 12</td>
</tr>
<tr>
<td>load 0x1444</td>
<td>ASID: 11</td>
</tr>
<tr>
<td>load 0x5444</td>
<td>ASID: 12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>TLB:</th>
<th>Valid</th>
<th>Virt</th>
<th>Phys</th>
<th>ASID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>1</td>
<td>9</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>11</td>
</tr>
</tbody>
</table>
TLB PERFORMANCE

With ASIDs, do context switches hurt TLB performance? (increase miss rate)
• Even with ASID, other processes “pollute” TLB
  • Discard process A’s TLB entries for process B’s entries

Context switches are expensive for memory performance!

Architectures can have multiple TLBs
• 1 TLB for data, 1 TLB for instructions
• 1 TLB for regular pages, 1 TLB for “super pages”

HW AND OS ROLES

Who Handles TLB MISS? H/W or OS?

OS: CPU traps into OS upon TLB miss
  • “Software-managed TLB”
  • OS interprets pagetables as it chooses with special instructions
  • Modifying TLB entries is privileged
    - otherwise what could process do?

H/W: CPU must know where pagetables are
  • CR3 register on x86
  • Pagetable structure fixed and agreed upon between HW and OS
  • HW “walks” the pagetable and fills TLB

Need same protection bits in TLB as pagetable
  - rwx
SUMMARY

• Pages are great, but accessing page tables for every memory access is slow

• Cache recent page translations → TLB
  • Hardware performs TLB lookup on every memory access

• TLB performance depends strongly on workload
  • Sequential workloads perform well
  • Workloads with temporal locality can perform well
  • Increase TLB reach by increasing page size

• In different systems, hardware or OS handles TLB misses

• TLBs increase cost of context switches
  • Flush TLB on every context switch
  • Add ASID to every TLB entry