Virtual memory & memory hierarchy

Hung-Wei Tseng
Recap: What happens when we access data

- Processor sends load request to L1-$
  - if read hit — return data
  - if write hit — set dirty and update in the block
- if miss
  - Select a victim block
    - If the target “set” is not full — select an empty/invalidated block as the victim block
    - If the target “set is full — select a victim block using some policy
      - LRU is preferred — to exploit temporal locality!
        - If the victim block is “dirty” & “valid”
          - Write back the block to lower-level memory hierarchy
        - Fetch the requesting block from lower-level memory hierarchy and place in the victim block
        - If write-back or fetching causes any miss, repeat the same process
Recap: causes of $ misses

• Compulsory miss
  • Cold start miss. First-time access to a block

• Capacity miss
  • The working set size of an application is bigger than cache size

• Conflict miss
  • Required data block replaced by block(s) mapping to the same set
  • Similar collision in hash — if the conflict miss doesn’t go away even though you made the cache fully-associative — it’s a capacity miss
Recap: optimizations

• Software
  • Data layout — capacity miss, conflict miss, compulsory miss
  • Blocking — capacity miss, conflict miss
  • Loop fission — conflict miss — when $ has limited way associativity
  • Loop fusion — capacity miss — when $ has enough way associativity
  • Loop interchange — conflict/capacity miss

• Hardware
  • Prefetch — compulsory miss
Let’s dig into this code

```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    // Create processes
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    // Generate rand see
    srand((int)time(NULL)+(int)getpid());
    a = rand();
    fprintf(stderr, "\nProcess %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), a, &a);
    sleep(10);
    fprintf(stderr, "\nProcess %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    return 0;
}
```
Consider the following code ...

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>
double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i< number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    fprintf(stderr, "Process %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    sleep(10);
    fprintf(stderr, "Process %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    return 0;
}
```

- Consider the case when we run multiple instances of the given program at the same time on modern machines, which pair of statements is correct?
  1. The printed “address of a” is the same for every running instances
  2. The printed “address of a” is different for each instance
  3. All running instances will print the same value of a
  4. Some instances will print the same value of a
  5. Each instance will print a different value of a

A. (1) & (3)
B. (1) & (4)
C. (1) & (5)
D. (2) & (3)
E. (2) & (4)

If you still don’t know why — you need to take CS202
If we can only use physical memory ...

- If there is no abstraction between the processor and memory, the processor/cache needs to directly use main memory’s byte address to read/write data. How many of the following would be happening?
  1. The program’s memory footprint, including instructions/data, cannot exceed the capacity of the installed DRAM
  2. There is no guarantee the compiled program can execute on another machine if both machine have the same processor but different memory capacities
  3. Two programs cannot run simultaneously if they use the same memory addresses
  4. One program can maliciously access data from other concurrently executing programs

A. 0
B. 1
C. 2
D. 3
E. 4
If we expose memory directly to the processor (I)

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Program</th>
<th>Data</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0f00bb27</td>
<td>00c2e800</td>
<td>00c2f800</td>
<td>00c2e800</td>
</tr>
<tr>
<td>509cbd23</td>
<td>00000008</td>
<td>00000008</td>
<td>00000008</td>
</tr>
<tr>
<td>00005d24</td>
<td>00c2f000</td>
<td>00c2f000</td>
<td>00c2e800</td>
</tr>
<tr>
<td>0000bd24</td>
<td>00000008</td>
<td>00000008</td>
<td>00000008</td>
</tr>
<tr>
<td>2ca422a0</td>
<td>00c2f800</td>
<td></td>
<td></td>
</tr>
<tr>
<td>130020e4</td>
<td>00000008</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00003d24</td>
<td>00c30000</td>
<td>00c30000</td>
<td>00c2e800</td>
</tr>
<tr>
<td>2ca4e2b3</td>
<td>00000008</td>
<td>00000008</td>
<td>00000008</td>
</tr>
</tbody>
</table>

What if my program needs more memory?
If we expose memory directly to the processor (II)

What if my program runs on a machine with a different memory size?

<table>
<thead>
<tr>
<th>Program</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0f00bb27</td>
<td>00c2e800</td>
</tr>
<tr>
<td>509cbd23</td>
<td>00000008</td>
</tr>
<tr>
<td>00005d24</td>
<td>00c2f000</td>
</tr>
<tr>
<td>0000bd24</td>
<td>00000008</td>
</tr>
<tr>
<td>2ca422a0</td>
<td>00c2f800</td>
</tr>
<tr>
<td>130020e4</td>
<td>00000008</td>
</tr>
<tr>
<td>00003d24</td>
<td>00c30000</td>
</tr>
<tr>
<td>2ca4e2b3</td>
<td>00000008</td>
</tr>
</tbody>
</table>

Memory

| 0f00bb27 | 00c2e800 |
| 509cbd23 | 00000008 |
| 00005d24 | 00c2f000 |
| 0000bd24 | 00000008 |
| 2ca422a0 | 00c2f800 |
| 130020e4 | 00000008 |
| 00003d24 | 00c30000 |
| 2ca4e2b3 | 00000008 |
If we expose memory directly to the processor (III)

What if both programs need to use memory?
If we can only use physical memory ...

• If there is no abstraction between the processor and memory, the processor/cache needs to directly using main memory’s byte address to read/write data. How many of the following would be happening?
  ① The program’s memory footprint, including instructions/data, cannot exceed the capacity of the installed DRAM
  ② There is no guarantee the compiled program can execute on another machine if both machine have the same processor but different memory capacities
  ③ Two programs cannot run simultaneously if they use the same memory addresses
  ④ One program can maliciously access data from other concurrently executing programs

A. 0
B. 1
C. 2
D. 3
E. 4
Virtual memory

• An abstraction of memory space available for programs/software/programmer
• Programs execute using virtual memory address
• The operating system and hardware work together to handle the mapping between virtual memory addresses and real/physical memory addresses
• Virtual memory organizes memory locations into “pages”
Demand paging
The virtual memory abstraction

Virtual Memory Space

Processor Core

Registers

Main Memory (DRAM)

Page

Page #1

load 0x0009

0x0000
0x0100
0x0200
0x0300
0x0400
0x0500
0x0600
0x0700
0x0800
0xFFF
0x1FFF
0x2FFF
0x3FFF
0x4FFF
0x5FFF
0x6FFF
0x7FFF

Page #1
```c
#define _GNU_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <sched.h>
#include <sys/syscall.h>
#include <time.h>

double a;

int main(int argc, char *argv[])
{
    int i, number_of_total_processes=4;
    number_of_total_processes = atoi(argv[1]);
    for(i = 0; i < number_of_total_processes-1 && fork(); i++);
    srand((int)time(NULL)+(int)getpid());
    fprintf(stderr, "Process %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    sleep(10);
    fprintf(stderr, "Process %d is using CPU: %d. Value of a is %lf and address of a is %p\n", getpid(), cpu, a, &a);
    return 0;
}
```
Address translation

- Processor receives virtual addresses from the running code, main memory uses physical memory addresses.
- Virtual address space is organized into “pages”.
- The system references the page table to translate addresses.
  - Each process has its own page table.
  - The page table content is maintained by OS.
Demand paging

• Treating physical main memory as a “cache” of virtual memory
• The block size is the “page size”
• The page table is the “tag array”
• It’s a “fully-associate” cache — a virtual page can go anywhere in the physical main memory
Do we really need a large table?

Your program probably never uses this huge area!

If you still don’t know why — you need to take CS202
Do we really need a large table?
# Address translation in x86-64

<table>
<thead>
<tr>
<th>63:48 (16)</th>
<th>47:39 (9 bits)</th>
<th>38:30 (9 bits)</th>
<th>29:21 (9 bits)</th>
<th>20:12 (9 bits)</th>
<th>11:0 (12 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SignExt</td>
<td>L4 index</td>
<td>L3 index</td>
<td>L2 index</td>
<td>L1 index</td>
<td>page offset</td>
</tr>
</tbody>
</table>

- **X86 Processor CR3 Reg.**
  - 512 entries
  - 512 entries

- **physical page #**
- **page offset**
May have 10 memory accesses for a “MOV” instruction! — 5 for instruction fetch and 5 for data access
Avoiding the address translation overhead
• TLB — a small SRAM stores frequently used page table entries
• Good — A lot faster than having everything going to the DRAM
• Bad — Still on the critical path
TLB + Virtual cache

• L1 $ accepts virtual address — you don’t need to translate
• Good — you can access both TLB and L1-$ at the same time and physical address is only needed if L1-$ misses
• Bad — it doesn’t work in practice
  • Many applications have the same virtual address but should be pointing different physical addresses
  • An application can have “aliasing virtual addresses” pointing to the same physical address

You really need “physical address” to judge if that’s what you want
Virtually indexed, physically tagged cache

- Can we find physical address directly in the virtual address — Not everything — but the page offset isn’t changing!
- Can we indexing the cache using the “partial physical address”?
  — Yes — Just make set index + block set to be exactly the page offset
Virtually indexed, physically tagged cache

set
4
block

virtual page #

index

offset

memory address:

0x0 8 2

memory address:

0b0000100000100100

V  virtual page #  physical page #

<table>
<thead>
<tr>
<th>V</th>
<th>virtual page #</th>
<th>physical page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x29</td>
<td>0x45</td>
</tr>
<tr>
<td>1</td>
<td>0xDE</td>
<td>0x68</td>
</tr>
<tr>
<td>1</td>
<td>0x10</td>
<td>0xA1</td>
</tr>
<tr>
<td>0</td>
<td>0x8A</td>
<td>0x98</td>
</tr>
</tbody>
</table>

V  D  tag  data

<table>
<thead>
<tr>
<th>V</th>
<th>D</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0x00</td>
<td>AABBCDDEEGGFFHH</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x10</td>
<td>IIJJKKLLMMNNOOPP</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0xA1</td>
<td>QQRRSSSTTUUVVVwXX</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0x10</td>
<td>YYZZAABBCDDEEFF</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x31</td>
<td>AABBCDDEEGGFFHH</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0x45</td>
<td>IIJJKKLLMMNNOOPP</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0x41</td>
<td>QQRRSSSTTUUVVVwXX</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0x68</td>
<td>YYZZAABBCDDEEFF</td>
</tr>
</tbody>
</table>

hit?  

0xA 1

=?