Cache Wrap-up, System Control Flow
CSE 410 Winter 2017

Instructor: Justin Hsia
Teaching Assistants: Kathryn Chan, Kevin Bi, Ryan Wong, Waylon Huang, Xinyu Sui
Administrivia

- Homework 4 due Tuesday (2/28)
- Lab 4 released today
  - Cache runtimes and parameter puzzles
Anatomy of a Cache Question

- Cache questions come in a few flavors:
  1) TIO Breakdown
  2) For fixed cache parameters, analyze the performance of the given code/sequence
  3) For fixed cache parameters, find best/worst case scenarios
  4) For given code/sequence, how does changing your cache parameters affect performance?
  5) Average Memory Access Time (AMAT)
The Cache

- What are the important cache parameters?
  - Must figure these out from problem description
  - Address size, cache size, block size, associativity, replacement policy
  - Solve for TIO breakdown, # of sets, management bits

- What starts in the cache?
  - Not always specified (best/worst case)
Code: Arrays

- Elements stored contiguously in memory
  - Ideal for spatial locality – if used properly
  - Different arrays not necessarily next to each other

- Remember to account for data size!
  - char is 1 B, int/float is 4 B, long/double is 8 B

- Pay attention to access pattern
  - Touch all elements (e.g. shift, sum)
  - Touch some elements (e.g. histogram, stride)
  - How many times do we touch each element?
Code: Linked Lists/Structs

- Nodes stored separately in memory
  - Addresses of nodes may be very different
  - Method of linking and ordering of nodes are important
- Remember to account for size/ordering of struct elements
- Pay attention to access pattern
  - Generally must start from “head”
  - How many struct elements are touched?
Access Patterns

- How many hits within a single block once it is loaded into cache?
- Will block still be in cache when you revisit its elements?
- Are there special/edge cases to consider?
  - Usually edge of block boundary or edge of cache size boundary
Cache Example Problem (1/3)

- 1 MiB address space, 125 cycles to go to memory. Fill in the following table:

<table>
<thead>
<tr>
<th></th>
<th>L1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache Size</td>
<td>16 KiB</td>
</tr>
<tr>
<td>Block Size</td>
<td>16 B</td>
</tr>
<tr>
<td>Associativity</td>
<td>4-way</td>
</tr>
<tr>
<td>Hit Time</td>
<td>3 cycles</td>
</tr>
<tr>
<td>Miss Rate</td>
<td>20%</td>
</tr>
<tr>
<td>Write Policy</td>
<td>Write-through</td>
</tr>
<tr>
<td>Replacement Policy</td>
<td>LRU</td>
</tr>
<tr>
<td>Tag</td>
<td></td>
</tr>
<tr>
<td>Index</td>
<td></td>
</tr>
<tr>
<td>Offset</td>
<td></td>
</tr>
<tr>
<td>AMAT</td>
<td></td>
</tr>
</tbody>
</table>
Cache Example Problem (2/3)

- Assuming the cache starts **cold** (all blocks invalid), calculate the hit rate for the following loop:
  - $m = 20$ bits, $C = 16$ KiB, $K = 16$ B, $E = 4$

```c
#define AR_SIZE 8192
int int_ar[AR_SIZE]; // &int_ar=0x80000
for(int i = 0; i < AR_SIZE/2; i++) {
    int_ar[i] *= int_ar[ i + AR_SIZE/2 ];
}
```
Cache Example Problem (3/3)

- For each of the proposed changes below, write **U** for “increase”, **N** for “no change”, or **D** for “decrease” to indicate the effect on the hit rate for the loop:
  - Direct-mapped cache: ____
  - Increase cache size: ____
  - Double AR_SIZE: ____
Roadmap

C:

```c
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```java
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg = c.getMPG();
```

Assembly language:

```assembly
get_mpg:
    pushq  %rbp
    movq   %rsp, %rbp
    ...
    popq   %rbp
    ret
```

Machine code:

```
011101000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Operating Systems
Leading Up to Processes

- System Control Flow
  - Control flow
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
Control Flow

- **So far:** we’ve seen how the flow of control changes as a single program executes

- **Reality:** multiple programs running concurrently
  - How does control flow across the many components of the system?
  - In particular: More programs running than CPUs

- **Exceptional control flow** is basic mechanism used for:
  - Transferring control between processes and OS
  - Handling I/O and virtual memory within the OS
  - Implementing multi-process apps like shells and web servers
  - Implementing concurrency
Control Flow

- Processors do only one thing:
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time
  - This sequence is the CPU’s control flow (or flow of control)

![Physical control flow diagram](image-url)
Altering the Control Flow

- Up to now: two ways to change control flow:
  - Jumps (conditional and unconditional)
  - Call and return
  - Both react to changes in program state

- Processor also needs to react to changes in system state:
  - Unix/Linux user hits “Ctrl-C” at the keyboard
  - User clicks on a different application’s window on the screen
  - Data arrives from a disk or a network adapter
  - Instruction divides by zero
  - System timer expires

- Can jumps and procedure calls achieve this?
  - No – the system needs mechanisms for “exceptional” control flow!
Exceptional Control Flow

- Exists at all levels of a computer system

- Low level mechanisms
  - **Exceptions**
    - Change in processor’s control flow in response to a system event (i.e., change in system state, user-generated interrupt)
    - Implemented using a combination of hardware and OS software

- Higher level mechanisms
  - **Process context switch**
    - Implemented by OS software and hardware timer
  - **Signals**
    - Implemented by OS software
    - Covered briefly later
Exceptions

- An *exception* is transfer of control to the operating system (OS) kernel in response to some *event* (i.e. change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples: division by 0, page fault, I/O request completes, Ctrl-C

- How does the system know where to jump to in the OS?

```
User Code          OS Kernel Code

event  current_instr  exception  exception processing by
        next_instr  exception handler, then:

• return to current_instr,
• return to next_instr, OR
• abort
```
Exception Table

- A jump table for exceptions (also called *Interrupt Vector Table*)
  - Each type of event has a unique exception number $k$
  - $k = \text{index into exception table (a.k.a interrupt vector)}$
  - Handler $k$ is called each time exception $k$ occurs

![Diagram of Exception Table]

- Exception Table
  - Exception numbers
  - Code for exception handler 0
  - Code for exception handler 1
  - Code for exception handler 2
  - Code for exception handler n-1
## Exception Table (Excerpt)

<table>
<thead>
<tr>
<th>Exception Number</th>
<th>Description</th>
<th>Exception Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Divide error</td>
<td>Fault</td>
</tr>
<tr>
<td>13</td>
<td>General protection fault</td>
<td>Fault</td>
</tr>
<tr>
<td>14</td>
<td>Page fault</td>
<td>Fault</td>
</tr>
<tr>
<td>18</td>
<td>Machine check</td>
<td>Abort</td>
</tr>
<tr>
<td>32-255</td>
<td>OS-defined</td>
<td>Interrupt or trap</td>
</tr>
</tbody>
</table>
Leading Up to Processes

- System Control Flow
  - Control flow
  - Exceptional control flow
  - Asynchronous exceptions (interrupts)
  - Synchronous exceptions (traps & faults)
Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor’s interrupt pin(s) (wire into CPU)
  - After interrupt handler runs, the handler returns to “next” instruction

- Examples:
  - I/O interrupts
    - Hitting Ctrl-C on the keyboard
    - Clicking a mouse button or tapping a touchscreen
    - Arrival of a packet from a network
    - Arrival of data from a disk
  - Timer interrupt
    - Every few ms, an external timer chip triggers an interrupt
    - Used by the OS kernel to take back control from user programs
Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - **Traps**
    - **Intentional**: transfer control to OS to perform some function
    - **Examples**: system calls, breakpoint traps, special instructions
    - Returns control to “next” instruction
  - **Faults**
    - **Unintentional** but possibly recoverable
    - **Examples**: page faults, segment protection faults, integer divide-by-zero exceptions
    - Either re-executes faulting (“current”) instruction or aborts
  - **Aborts**
    - **Unintentional** and unrecoverable
    - **Examples**: parity error, machine check (hardware failure detected)
    - Aborts current program
System Calls

- Each system call has a unique ID number
- Examples for Linux on x86-64:

<table>
<thead>
<tr>
<th>Number</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>read</td>
<td>Read file</td>
</tr>
<tr>
<td>1</td>
<td>write</td>
<td>Write file</td>
</tr>
<tr>
<td>2</td>
<td>open</td>
<td>Open file</td>
</tr>
<tr>
<td>3</td>
<td>close</td>
<td>Close file</td>
</tr>
<tr>
<td>4</td>
<td>stat</td>
<td>Get info about file</td>
</tr>
<tr>
<td>57</td>
<td>fork</td>
<td>Create process</td>
</tr>
<tr>
<td>59</td>
<td>execve</td>
<td>Execute a program</td>
</tr>
<tr>
<td>60</td>
<td>_exit</td>
<td>Terminate process</td>
</tr>
<tr>
<td>62</td>
<td>kill</td>
<td>Send signal to process</td>
</tr>
</tbody>
</table>
Traps Example: Opening File

- **User calls** `open(filename, options)`
- **Calls** `__open` function, which invokes system call instruction `syscall`

```assembly
00000000000e5d70 <__open>:
  ...
e5d79:  b8 02 00 00 00  mov    $0x2,%eax  # open is syscall 2
e5d7e:  0f 05                syscall  # return value in %rax
e5d80:  48 3d 01 f0 ff ff    cmp    $0xffffffffffffffff001,%rax
  ...
e5dfa:  c3                   retq
```

- `%rax` contains syscall number
- Other arguments in `%rdi, %rsi, %rdx, %r10, %r8, %r9`
- Return value in `%rax`
- Negative value is an error corresponding to negative `errno`
Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user’s memory is currently on disk

Page fault handler must load page into physical memory
- Returns to faulting instruction: `mov` is executed again!
  - Successful on second try
Fault Example: Invalid Memory Reference

```c
int a[1000];
int main()
{
    a[5000] = 13;
}
```

```assembly
80483b7: c7 05 60 e3 04 08 0d movl $0xd,0x804e360
```

- Page fault handler detects invalid address
- Sends SIGSEGV signal to user process
- User process exits with “segmentation fault”
Summary

Exceptions

- Events that require non-standard control flow
- Generated externally (interrupts) or internally (traps and faults)
- After an exception is handled, one of three things may happen:
  - Re-execute the current instruction
  - Resume execution with the next instruction
  - Abort the process that caused the exception