Memory Hieararchy and Caches

Ian Hartwig
Section F
Feb. 18th, 2013
Thank Grant for today’s slides
Outline

- Schedule
- Memory organization
- Caching
  - Different types of locality
  - Cache organization
- Cachelab
  - Part (a) Building Cache Simulator
  - Part (b) Efficient Matrix Transpose
  - Bro do you even C
Before we Begin...

- **Encouraging Female Reverse Engineers**
  - Contest to reverse engineer malicious software
    - And document it with utmost understanding

- **Prize**
  - Ticket to Symposium on Security for Asia Network (Syscan)

- **Details**
  - [http://addxorrol.blogspot.de/2013/01/encouraging-female-reverse-engineers.html](http://addxorrol.blogspot.de/2013/01/encouraging-female-reverse-engineers.html)
  - Only women can submit (sorry guys)
  - One of the judges went to CMU!
  - Deadline: 24th of March 2013, 23:59 GMT+1
Class Schedule

- **Buflab**
  - Due Tomorrow, midnight

- **Cachelab**
  - Out tomorrow!
  - Due Thursday, February 28
  - 10 days
Outline

- Schedule
- Structures
- Memory organization
- Caching
  - Different types of locality
  - Cache organization
- Cachelab
  - Part (a) Building Cache Simulator
  - Part (b) Efficient Matrix Transpose
  - Bro do you even C
Satisfying Alignment with Structures

- **Within structure:**
  - Must satisfy each element’s alignment requirement

- **Overall structure placement**
  - Each structure has alignment requirement $K$
    - $K = \text{Largest alignment of any element}$
    - Initial address & structure length must be multiples of $K$

- **Example (under Windows or x86-64):**
  - $K = 8$, due to `double` element

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```
Different Alignment Conventions

- **x86-64 or IA32 Windows:**
  - $K = 8$, due to `double` element

```c
struct S1 {
    char c;
    int i[2];
    double v;
} *p;
```

- **IA32 Linux**
  - $K = 4$; `double` treated like a 4-byte data type
x86-64 Example Addresses

address range ~$2^{47}$

|$rsp$| 0x00007fffffff8d1f8
|p3| 0x00002aaabaadd010
|p1| 0x00002aaaaaadc010
|p4| 0x0000000011501120
|p2| 0x0000000011501010
&$p2$| 0x0000000010500a60
&$beyond$| 0x0000000000500a44
$big\_array$| 0x0000000010500a80
$huge\_array$| 0x0000000000500a50
$main()$| 0x0000000000400510
$useless()$| 0x0000000000400500
$final\ malloc()$| 0x00000000386ae6a170

$\text{malloc()}$ is dynamically linked
address determined at runtime
Memory Hierarchy

- Registers
- SRAM
- DRAM
- Local Secondary storage
- Remote Secondary storage

We will discuss this interaction
SRAM vs DRAM tradeoff

- **SRAM (cache)**
  - Faster (L1 cache: 1 CPU cycle)
  - Smaller (Kilobytes (L1) or Megabytes (L2))
  - More expensive and “energy-hungry”

- **DRAM (main memory)**
  - Relatively slower (hundreds of CPU cycles)
  - Larger (Gigabytes)
  - Cheaper
Caching

- **Temporal locality**
  - A memory location accessed is likely to be accessed again multiple times in the future
  - After accessing address X in memory, save the bytes in cache for future access

- **Spatial locality**
  - If a location is accessed, then nearby locations are likely to be accessed in the future.
  - After accessing address X, save the block of memory around X in cache for future access
Memory Address

- 64-bit on shark machines

- Block offset: \( b \) bits
- Set index: \( s \) bits
Cache

- A cache is a set of $2^s$ cache sets

- A cache set is a set of $E$ cache lines
  - $E$ is called associativity
  - If $E=1$, it is called “direct-mapped”

- Each cache line stores a block
  - Each block has $2^b$ bytes
General Cache Concepts

Cache

Smaller, faster, more expensive memory caches a subset of the blocks

Data is copied in block-sized transfer units

Larger, slower, cheaper memory viewed as partitioned into “blocks”

Memory
General Cache Concepts: Miss

Data in block b is needed

Block b is not in cache: Miss!

Block b is fetched from memory

Block b is stored in cache
• Placement policy: determines where b goes
• Replacement policy: determines which block gets evicted (victim)
General Caching Concepts: Types of Cache Misses

- **Cold (compulsory) miss**
  - The first access to a block has to be a miss
  - Most cold misses occur at the beginning, because the cache is empty

- **Conflict miss**
  - Most caches limit blocks at level k+1 to a small subset (sometimes a singleton) of the block positions at level k
    - E.g., Block i at level k+1 must be placed in block (i mod 4) at level k
  - Conflict misses occur when the level k cache is large enough, but multiple data objects all map to the same level k block
    - E.g., Referencing blocks 0, 8, 0, 8, 0, 8, ... would miss every time

- **Capacity miss**
  - Occurs when the set of active cache blocks (working set) is larger than the cache
Cacheland

- Part (a) Building a cache simulator

- Part (b) Optimizing matrix transpose

- Bro do you even C (helpful C stuff)
Part (a) Cache simulator

- A cache simulator is NOT a cache!
  - Memory contents NOT stored
  - Block offsets are NOT used
  - Simply counts hits, misses, and evictions

- Your cache simulator need to work for different s, b, E, given at run time.
- Use LRU replacement policy
Cache simulator: Hints

- A cache is just 2D array of *cache lines*:
  - struct cache_line cache[S][E];
  - $S = 2^s$, is the number of sets
  - $E$ is associativity

- Each *cache_line* has:
  - Valid bit
  - Tag
  - LRU counter
Part (b) Efficient Matrix Transpose

Matrix Transpose (A -> B)

Matrix A

Matrix B

How do we optimize this operation using the cache?
Part (b)

- **Cache:**
  - You get 1 kilobytes of cache
  - Directly mapped (E=1)
  - Block size is 32 bytes (b=5)
  - There are 32 sets (s=5)

- **Test Matrices:**
  - 32 by 32, 64 by 64, 61 by 67
Part (b)

- Things you’ll need to know:
  - Warnings are errors
  - Header files
  - Useful functions
Warnings are Errors

- Strict compilation flags

- Reasons:
  - Avoid potential errors that are hard to debug
  - Learn good habits from the beginning

- Add “-Werror” to your compilation flags
Missing Header Files

- Remember to include files that we will be using functions from
- If function declaration is missing
  - Find corresponding header files
  - Use: man <function-name>

- Live example
  - man 3 getopt
getopt

- getopt() automates parsing elements on the unix command line If function declaration is missing
  - Typically called in a loop to retrieve arguments
  - Its return value is stored in a local variable
  - When getopt() returns -1, there are no more options

- To use getopt, your program must include the header file unistd.h
getopt

- A switch statement is used on the local variable holding the return value from getopt()
  - Each command line input case can be taken care of separately
  - “optarg” is an important variable – it will point to the value of the option argument
- Think about how to handle invalid inputs
Example

```c
int main(int argc, char** argv){
    int opt, x;
    while(-1 != (opt = getopt(argc,argv,"x:")) ) ){
        //looping over arguments

        switch(opt) { //determine which argument it’s processing
            case 'x':
                x = atoi(optarg);
                break;
            default:
                printf("wrong argument\n");
        }
    }
}
```

- Suppose the program executable was called “foo”. Then we would call “./foo -x 1” to pass the value 1 to variable x.
The `fscanf()` function is just like `scanf()` except it can specify a stream to read from (`scanf` always reads from stdin)

- parameters:
  - file pointer,
  - format string with information on how to read file,
  - the rest are pointers to variables to storing data from file

- Typically want to use this function in a loop until it hits the end of file

- `fscanf` will be useful in reading from the trace files
Example

FILE * pFile; //pointer to FILE object

pFile = fopen("myfile.txt","r"); //open file for reading

int x, y;
char c;
//read two ints and a char from file

while(fscanf(pFile, "%d %d %c", &x, &y, &c) > 0){
    // Do stuff
}

fclose(pFile); //remember to close file when done
Malloc/free

- Use malloc this to allocate memory on the heap
- Always free what you malloc, otherwise may get memory leak
  - Some_pointer_you_malloced = malloc(sizeof(int));
  - Free(some_pointer_you_malloced);
- Don’t free memory you didn’t allocate
Tutorials

- getopt:

- fscanf:
  - http://crasseux.com/books/ctutorial/fscanf.html

- Google is your friend
Style

- Read the style guideline
  - But I already read it!
  - Good, read it again.

- Pay special attention to failure and error checking
  - Functions don’t always work
  - What happens when a syscall fails??

- Start forming good habits now!
Questions?