Cache Lab
Implementation and Blocking

Marjorie Carlson
Section A
October 7th, 2013
Welcome to the World of Pointers!
Class Schedule

- **Cache Lab**
  - Due Thursday.
  - Start now (if you haven’t already).

- **The Midterm Starts in <10 Days!**
  - Wed Oct 16th – Sat Oct 19
  - Start now (if you haven’t already).

  - No, really. Start now.
Outline

- Memory organization
- Caching
  - Different types of locality
  - Cache organization
- Cachelab
  - Part (a) Building Cache Simulator
  - Part (b) Efficient Matrix Transpose
Memory Hierarchy

- **L0:** Registers
  - CPU registers hold words retrieved from L1 cache

- **L1:** L1 cache (SRAM)
  - L1 cache holds cache lines retrieved from L2 cache

- **L2:** L2 cache (SRAM)
  - L2 cache holds cache lines retrieved from main memory
  - Main memory holds disk blocks retrieved from local disks

- **L3:** Local secondary storage (local disks)
  - Local disks hold files retrieved from disks on remote network servers

- **L4:** Main memory (DRAM)

- **L5:** Remote secondary storage (tapes, distributed file systems, Web servers)

- Smaller, faster, costlier per byte
- Larger, slower, cheaper per byte
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
Locality

- The key concept that makes caching work:
  - If you use a piece of data, you’ll probably use it and/or nearby data again soon. So it’s worth taking the time to move that whole chunk of data to SRAM, so subsequent access to that block will be fast.

- Temporal locality
  - Recently referenced items are likely to be referenced again in the near future
  - After accessing address X in memory, save the bytes in cache for future access

- Spatial locality
  - Items with nearby addresses tend to be referenced close together in time
  - After accessing address X, save the block of memory around X in cache for future access
General Cache Concepts

Cache

Data is copied in block-sized transfer units

Memory

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

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

memory address

- Block offset: \( b \) bits
  - Size of block \( B = 2^b \)
- Set index: \( s \) bits
  - Number of sets \( S = 2^s \)
- Tag Bits: \( t \) bits = \{address size\} – \( b \) – \( s \)
  - (On shark machines, address size = 64 bits.)

- Key point: if the data at a given address is in the cache, it has to be in the \( block\ offset^\text{th} \) byte of the \( set\ index^\text{th} \) set – but it can be in any line in that set.
Cache Terminology

E lines per set

S = \(2^s\) sets

B = \(2^b\) bytes per cache block

Total cache size = \(S \times E \times B\)

Address of word:
- t bits
- s bits
- b bits
  - tag
  - set index
  - block offset
Cache Terminology

E lines per set

(S = 2^s) sets

B = 2^b bytes per cache block

B = 2^b bytes per cache block (the data)

Total cache size = S * E * B

Address of word:
- t bits
- s bits
- b bits

Tag
Set index
Block offset
Cache Terminology

- **E lines per set**
- **S = 2^s** sets
- **B = 2^b** bytes per cache block (the data)

Total cache size = \( S \times E \times B \)

Address of word:

- **t bits**
- **s bits**
- **b bits**
  - **tag**
  - **set index**
  - **block offset**

Data begins at this offset.
General Cache Concepts: Hit

Request: 14

Data in block x is needed

Block x is in cache and is valid: Hit!

Memory isn’t touched (yay!)
General Cache Concepts: Miss

Data in block y is needed

Block y is not in cache: Miss!

Block y is fetched from memory

Block y is stored in cache
• Placement policy: determines where b goes
General Cache Concepts: Miss & Evict

Data in block \( z \) is needed

Block \( z \) is not in cache:
Miss!

Block \( z \) is fetched from memory

Block \( z \) is stored in cache:
Evict!

- Placement policy: determines where \( b \) goes
- Replacement policy: determines which block gets evicted (victim)
General Caching Concepts: Types of Misses

- **Cold (compulsory) miss**
  - The first access to a block has to be a miss

- **Conflict miss**
  - Conflict misses occur when the cache is large enough, but multiple data objects all map to the same 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
General Cache Concepts: Conflict Misses

Data in block z is needed

Block z is not in cache: 
**Miss!**

Block z is fetched from memory

Block z is stored in cache: 
**Evict!**

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)
General Cache Concepts: Conflict Misses

Data in block z is needed

Block z is not in cache: Miss!

Block z is fetched from memory

Block z is stored in cache: Evict!

- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)
General Cache Concepts: Conflict Misses

Data in block z is needed

Block z is not in cache: Miss!

Block z is fetched from memory

Block z is stored in cache: Evict!

• Placement policy: determines where b goes
• Replacement policy: determines which block gets evicted (victim)
Sets vs. Lines

- **Why arrange cache in sets?**
  - If a block can be stored *anywhere*, then you have to search for it *everywhere*.

- **Why arrange cache in lines?**
  - If a block can only be stored *in one place*, it’ll be evicted a lot.

“**The rule of thumb is that doubling the associativity, from direct mapped to two-way, or from two-way to four-way, has about the same effect on hit rate as doubling the cache size.**” –*Wikipedia, CPU Cache*
Sets vs. Lines

- An 8-byte cache with 2-byte blocks could be arranged as:
  - one set of four lines ("fully associative"): 
  - four sets of one line ("direct-mapped"): 
  - two sets of two lines (2-way associative):
Sets vs. Lines

<table>
<thead>
<tr>
<th>Data</th>
<th>‘a’</th>
<th>‘b’</th>
<th>‘c’</th>
<th>‘d’</th>
<th>‘e’</th>
<th>‘f’</th>
<th>‘g’</th>
<th>‘h’</th>
<th>‘i’</th>
<th>‘j’</th>
<th>‘k’</th>
<th>‘l’</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
</tr>
</tbody>
</table>

- For each possible configuration of an 8-byte cache with 2-bytes blocks:
  - How many hits/misses/evictions will there be for the following sequence of operations?
  - What will be in the cache at the end?

1. L 0101
2. L 0100
3. L 0000
4. L 0010
5. L 1000
6. L 0000
7. L 0101
8. L 1011
Outline

- Memory organization

- Caching
  - Different types of locality
  - Cache organization

- Cachelab
  - Part (a) Building Cache Simulator
  - Part (b) Efficient Matrix Transpose
Part (a) Cache simulator

- A cache simulator is NOT a cache!
  - Memory contents are not stored.
  - Block offsets are not used – the b bits in your address don’t matter.
  - Simply count hits, misses, and evictions.

- Your cache simulator needs to work for different values of s, b, and E — given at run time.

- Use LRU – a Least Recently Used replacement policy
  - Evict the least recently used block from the cache to make room for the next block.
  - Queues? Time stamps? Counter?
Part (a) Hints

■ Structs are a great way to represent your cache line. Each cache line has:
  ▪ A valid bit.
  ▪ A tag.
  ▪ Some sort of LRU counter (if you are not using a queue).

■ A cache is just 2D array of cache lines:
  ▪ struct cache_line cache[S][E];
  ▪ Number of sets: $S = 2^s$
  ▪ Number of lines per set: $E$
  ▪ You know $S$ and $E$ at run time, but not at compile time. What does that mean you’ll have to do when you declare your cache?
Part (a) malloc/free

- Use malloc to allocate memory on the heap.

- Always free what you malloc, otherwise you will leak memory!
  
  ```c
  my_pointer = malloc(sizeof(int));
  ... use that pointer for a while ...
  free(my_pointer);
  ```

- Common mistake: freeing your array of pointers, but forgetting to free the objects those pointers point to.

- Valgrind is your friend!
Part (a) getopt

- getopt() automates parsing elements on the Unix command line.
  - It’s typically called in a loop to deal with each flag in turn. (It returns -1 when it’s out of inputs.)
  - Its return value is the flag it’s currently parsing (“x”, “y”, “r”). You can then use a switch statement on the local variable you stored that value to.
  - If a flag has an associated argument, getopt also gives you optarg, a pointer to that argument (“1”, “3”). Remember this argument is a string, not an integer.
  - Think about how to handle invalid inputs.

./point -x 1 -y 3 -r
Part (a) getopt Example

```c
int main(int argc, char** argv){
    int opt, x, y;
    int r = 0;
    while(-1 != (opt = getopt(argc, argv, "x:y:r"))){
        switch(opt) {
            case 'x':
                x = atoi(optarg);
                break;
            case 'y':
                y = atoi(optarg);
                break;
            case 'r':
                r = 1;
                break;
            default:
                printf("Invalid argument.\n");
                break;
        }
    }
}
```
Part (a) fscanf

- fscanf will be useful in reading lines from the trace files.
  - L 10, 4
  - M 20, 8
- fscanf() is just like scanf() except it can specify a stream to read from (i.e., the file you just opened).
- Its parameters are:
  1. a stream pointer (e.g. your file descriptor).
  2. a format string with information on how to parse the file
  3. n. the appropriate number of pointers to the variables in which you want to store the data from your file.
- You typically want to use it in a loop; it returns -1 if it hits EOF (or if the data doesn’t match the format string).
Part (a) fscanf Example

```c
FILE *pFile;               //pointer to FILE object

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

char operation;
unsigned address;
int size;

// read a series of lines like " M 20,1" or "L 19,3"

while(fscanf(pFile, " %c %x,%d", &operation, &address, &size)>0){
    // do stuff ...
}

fclose(pFile);           //remember to close file
```
Part (a) Header files!

- If you use a library function, always remember to \#include the relevant library!

- Use `man <function-name>` to figure out what header you need.
  - `man 3 getopt`
  - If you’re not using a shark machine, you’ll need `<getopt.h>` as well as `<unistd.h>`. (So why not use a shark machine?)

- If you get a warning about a missing or implicit function declaration, you probably forgot to include a header file.
Part (a) Relevant tutorials

- **getopt:**

- **fscanf:**
  - [http://crasseux.com/books/ctutorial/fscanf.html](http://crasseux.com/books/ctutorial/fscanf.html)

- Google is your friend!
Part (b) Efficient Matrix Transpose

Matrix Transpose (A -> B)

Matrix A

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
</tr>
</tbody>
</table>

Matrix B

<table>
<thead>
<tr>
<th>1</th>
<th>5</th>
<th>9</th>
<th>13</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
<td>11</td>
<td>15</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>12</td>
<td>16</td>
</tr>
</tbody>
</table>

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

Suppose block size is 8 bytes. Each int is 4 bytes.

Access A[0][0]: cache miss

Access B[0][0]: cache miss

Access A[0][1]: cache hit

Access B[1][0]: cache miss

Should we handle 3 & 4 next or 5 & 6?
Part (b) Blocked Matrix Multiplication

```c
double *c = (double *) calloc(sizeof(double), n*n);

/* Multiply n x n matrices a and b */
void mmm(double *a, double *b, double *c, int n) {
    int i, j, k;
    for (i = 0; i < n; i += B)
        for (j = 0; j < n; j += B)
            for (k = 0; k < n; k += B)
                for (i1 = i; i1 < i+B; i1++)
                    for (j1 = j; j1 < j+B; j1++)
                        for (k1 = k; k1 < k+B; k1++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```

“Sometimes it is faster to do more faster work than less slower work.”
-Greg Kesden

/* B x B mini matrix multiplications */
for (i1 = i; i1 < i+B; i++)
    for (j1 = j; j1 < j+B; j++)
        for (k1 = k; k1 < k+B; k1++)
            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];

Block size B x B

Part (b) Blocking

- Blocking: dividing your matrix into sub-matrices.

- The ideal size of each sub-matrix depends on your cache block size, cache size, and input matrix size.

- Try different sub-matrix sizes and see what happens!

Part (b) Specs

- **Cache:**
  - You get 1 KB of cache
  - It’s 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
  - Your solution need not work on other size matrices.
General Advice: Warnings are Errors!

- **Strict compilation flags:**
  - `-Wall` “enables all the warnings about constructions that some users consider questionable, and that are easy to avoid.”
  - `-Werror` treats warnings as errors.

- **Why?**
  - Avoid potential errors that are hard to debug.
  - Learn good habits from the beginning.

```bash
# # Student makefile for Cache Lab
#
CC = gcc
CFLAGS = -g -Wall -Werror -std=c99
...
```
General Advice: Style!!!

- The rest of the labs in this course will be hand-graded for style as well as auto-graded for correctness.

- 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 system call fails?

- Start forming good habits now!