15-213 Recitation 7: Caches and Blocking

22 Feb 2016
Ralf Brown and the 15-213 staff
Agenda

- Reminders
- Revisiting caching
- getopt() and fscanf()
- Blocking to reduce cache misses
Reminders

- Cache Lab is due **Thursday**!
- tshlab will be released **Thursday**.
- Exam1 is just a week away!
  - Start doing practice problems.
  - Come to the review session.

Image credit: flickr.com / kellt825@yahoo.com
CC-BY-SA 2.0
Reminders: Cache Lab

- Two parts
  - write a cache simulator – hopefully you've started this part by now
  - optimize some code to minimize cache misses – we'll talk about this today
- Programming style will be graded starting now
  - worth about a letter grade on this assignment
  - a summary slide is included as an appendix to this recitation, but be sure to carefully read the style guide
- Details are in the writeup!
Memory Hierarchy

- Local secondary storage (local disks)
  - megabytes, 2-5 ns random access
- Main memory (DRAM)
  - gigabytes, 20-100 ns random access
- L3 cache (SRAM)
  - 100s of KB, ~1 ns random access
- L2 cache
  - 10s of KB, <1 ns random access
- L1 cache
  - 100s of bytes, 0.2 ns access
- Reg
  - Each level caches the one below

- Remote secondary storage (e.g. cloud storage)
  - terabytes, 15,000,000+ ns random access
  - exabytes, >100,000,000 ns access
## Cache Miss Types

- First access is a *cold miss*
- Remember this example? We had to evict a line despite having unused cache lines
  - This causes a *conflict miss* when we later want to access the evicted line
- If we repeatedly read 1024 consecutive bytes, the first byte will have been evicted by the time we read it again
  - This is called a *capacity miss*

<table>
<thead>
<tr>
<th>V LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x15213</td>
<td>I J K L</td>
</tr>
</tbody>
</table>

Set0

<table>
<thead>
<tr>
<th>V LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>--</td>
</tr>
</tbody>
</table>

Set1

<table>
<thead>
<tr>
<th>V LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0x15212</td>
<td>E F G H</td>
</tr>
</tbody>
</table>

Set2

<table>
<thead>
<tr>
<th>V LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>--</td>
</tr>
</tbody>
</table>

Set3

<table>
<thead>
<tr>
<th>V LRU</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>--</td>
</tr>
</tbody>
</table>

```
 0x15213
 0x15212
 0x15213
```
Cache Replacement Policies

- When we need to evict a cache line, which one do we choose?
  - Least Recently Used – replace the data which has gone unused longest
  - Least Frequently Used
  - First-In, First Out – replace the oldest data
  - Random
  - (no choice for direct-mapped)
- Policy is implemented in hardware for speed
Cache Lab: Cache Simulator Hints

- You are simply counting hits, misses, and evictions
- Use LRU (Least Recently Used) replacement policy
- Structs are a great way to bundle up the different parts of a cache line (valid bit, tag, LRU counter, etc.)
- A cache is just a 2D array of cache lines
  - one dimension represents associativity E, the other the number of sets S:
    ```c
    struct cache_line cache[S][E];
    ```
- Your simulator needs to handle different values of S, E, and b (block size) given at run time
getopt() is a standard way to extract items from the command line
- usually called in a loop until it returns -1 (no more inputs)
- returns the flag it has just parsed (“a”, “n”, etc.)
  - use a switch statement to handle each flag separately
  - if the flag has an associated argument (“myarg”), its `string` value is stored in the global variable `optarg`
- See `man 3 getopt` for more information

Your program must `#include <unistd.h>` to use getopt
getopt Example

#include <stdio.h> /* for printf */
#include <unistd.h> /* for getopt */

int main(int argc, char **argv) {
    int opt, aflag=0, nflag=0;
    float xflag=0.0;
    /* loop over arguments */
    while (-1 != (opt = getopt(argc, argv, "an:y:"))) {
        /* determine which argument was found */
        switch (opt) {
            case 'a':  aflag=1;                    break;
            case 'n':  nflag=atoi(optarg);         break;
            case 'x':  xflag=atof(optarg);         break;
            default:   printf("unknown argument"); break;
        }
    }
    return 0;
}
getopt Example

```c
#include <stdio.h> /* for printf */
#include <unistd.h> /* for getopt */

int main(int argc, char **argv) {
    int opt, aflag=0, nflag=0;
    float xflag=0.0;
    /* loop over arguments */
    while (-1 != (opt = getopt(argc, argv, "an:y:"))) {
        /* determine which argument was found */
        switch (opt) {
            case 'a':  aflag=1;  break;
            case 'n':  nflag=atoi(optarg);  break;
            case 'x':  xflag=atof(optarg);  break;
            default:     printf("unknown argument");  break;
        }
    }
    return 0;
}
```

repeat until all flags have been handled

valid flag letters. an appended colon means the flag takes an argument to be put in optarg

commandline data (arg count and args)

handle the individual flags

always handle the unexpected

convert string to integer or float
Cache Lab: Parsing Input with fscanf

- fscanf() is exactly like scanf() except that you specify the stream to use (i.e. an open file) instead of always reading from standard input
- its parameters are
  1. a stream pointer of type FILE*, e.g. from fopen()
  2. a format string specifying how to parse the input
  3-n. a **pointer** to each of the variables that will store the parsed data
- fscanf() returns -1 if the data does not match the format string or there is no more input
- Use it to parse the trace files
fscanf() Example

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

pFile = fopen("trace.txt","r"); /* open trace file for reading */
/* verify that pFile is non-NULL! */

char access_type;
unsigned long address;
int size;

/* line format is " S 2f,1" or " L 7d0,3" */
/* so we need to read a character, a hex number, and a decimal number */
/* put those in the format string along with the fixed formatting */
while (fscanf(pFile," %c %lx,%d", &access_type, &address, &size) > 0) {
    /* do stuff */
}

fclose(pFile); /* always close file when done */
Cache Lab: malloc/free

- You will need to allocate memory that can persist across functions
- Use malloc() to get some memory from the heap
- Use free() to de-allocate that memory:
  ```c
  int *malloced_pointer = malloc(sizeof(int));
  ...
  do something with malloced_pointer ...
  free(malloced_pointer);
  ```
- Never free memory you didn't allocate: this can cause strange behavior or crashes with no obvious cause later in the program
- Always free what you malloc: if you forget, you get a memory leak
Cache Lab: Helpful Information

- **getopt**
  - man 3 getopt

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

- Keep memory accesses bunched together
  - in both time and space (address)
  - the *working set* at any time should be smaller than the cache
- Avoid access patterns that cause conflict misses
  - memory *strides* in powers of two that cause all accesses to use only a few (or just one!) cache set
Temporal Locality

Q: Which of the functions on the right has fewer cache misses when $n$ is large?

```c
void fn1(int *a, int *b, int n){
    for (int i=0; i<n; i++)
        b[i] *= a[i];
    for (int i=0; i<n; i++)
        a[i] += b[i];
}

void fn2(int *a, int *b, int n){
    for (int i=0; i<n; i++) {
        b[i] *= a[i] ;
        a[i] += b[i];
    }
}
```
Temporal Locality

Q: Which of the functions on the right has fewer cache misses when $n$ is large?

A: $f_{n2}$, because the repeated element accesses happen together.

In $f_{n1}$, each array element will have been evicted by the time the second loop accesses it again (a capacity miss because the working set is larger than the cache).

```c
void fn1(int *a, int *b, int n){
    for (int i=0; i<n; i++)
        b[i] *= a[i];
    for (int i=0; i<n; i++)
        a[i] += b[i];
}

void fn2(int *a, int *b, int n){
    for (int i=0; i<n; i++) {
        b[i] *= a[i] ;
        a[i] += b[i];
    }
}
```
Spatial Locality

Q: Which of the functions on the right has fewer cache misses when \( n \) is large?

```c
void fn3(int *a, int n) {
    for (int i=1; i<n; i++) {
        a[i] *= 2;
        a[i-1] *= 3;
    }
}

void fn4(int *a, int n) {
    for (int i=1; i<=n; i++) {
        a[n-i] *= 2; /*reverse order*/
        a[i-1] *= 3;
    }
}
```
Spatial Locality

Q: Which of the functions on the right has fewer cache misses when \( n \) is large?

A: \texttt{fn3}, because successive array accesses are adjacent to each other.

```c
void fn3(int *a, int n) {
    for (int i=1; i<n; i++) {
        a[i] *= 2;
        a[i-1] *= 3;
    }
}

void fn4(int *a, int n) {
    for (int i=1; i<=n; i++) {
        a[n-i] *= 2; /*reverse order*/
        a[i-1] *= 3;
    }
}
Cache Lab: Efficient Matrix Transpose

- Transposing a matrix essentially swaps its two dimensions:

<table>
<thead>
<tr>
<th></th>
<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>
<td></td>
</tr>
<tr>
<td>9</td>
<td>10</td>
<td>11</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td></td>
</tr>
</tbody>
</table>

- How do we minimize the number of cache misses while performing this operation?

- a[i][j] becomes a[j][i]
Efficient Matrix Transpose

If a cache line holds 2 doubles, copying 1 and 2 causes
- A[0][0] miss
- B[0][0] miss
- A[0][1] hit
- B[1][0] miss

Will it be better to copy 3&4 or 5&6 next?
Efficient Matrix Transpose

A:

1  2  3  4  
5  6  7  8  
9 10 11 12  
13 14 15 16

B:

1  -  -  -  
2  -  -  -  
-  -  -  -  
-  -  -  -  

- Copying 3&4:
  - A[0][2] miss
  - B[2][0] miss
  - A[0][3] hit
  - B[3][0] miss

- Copying 5&6:
  - A[1][0] miss
  - B[0][1] hit
  - A[1][1] hit
  - B[1][1] hit
-blocking example: matrix multiplication

- Multiply an MxN matrix by a NxK matrix to yield an MxK matrix by taking the “dot product” of corresponding rows and columns:

\[
\begin{array}{c}
7\times 6 \\
7\times 8 \\
8\times 6 \\
\end{array}
\]

- The dot product is the sum of the products of corresponding elements:
  - for \((k=0; k<n; k++)\) \(c[i][j] += a[i][k] \times b[k][j];\)
Blocking Example: Matrix Multiplication

- Assume a really tiny cache with four lines of 16 bytes (2 doubles)
- After computing c[0][0], we will have accessed the dark gray cells, and the red-bordered cells will be in the cache:
- In computing c[0][1], every single access in b, and half the accesses in a, will be capacity misses! (why?) Once done, the cache contents will be as shown.
Blocked Matrix Multiplication

- Let's re-order the computation to build up the $c[i][j]$ values a piece at a time
  - this works because they are sums of products
- Notice how each step of the calculation stays within the cache's capacity
Blocked Matrix Multiplication

Multiply a block at a time, accumulating the results of the BxB dot products.

We'll call the blocking size $B$ and work on BxB blocks.

Then move on to the next result block.
When B=1, this becomes exactly the same as standard matrix multiplication.
Standard Matrix Multiplication: Code

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

void matrix_mult(double *a, double *b, double *c, int n) {
    int i, j, k;
    /* process each row */
    for (i=0; i<n; i++)
        /* process each column */
        for (j=0; j<n; j++)
            /* run the dot-product of the current row/column */
            for (k=0; k<n; k++)
                c[i*n+j] += a[i*n+k] * b[k*n+j];
}
Blocked Matrix Multiplication: Code

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

void matrix_mult(double *a, double *b, double *c, int n) {
    int i, j, k;
    /* process each row of blocks */
    for (i=0; i<n; i+=B)
        /* process each column of blocks */
        for (j=0; j<n; j+=B)
            /* run the dot-product of the current row/column of blocks */
            for (k=0; k<n; k+=B)
                /* perform BxB mini matrix multiplications */
                for (int i1=i; i1<i+B; i1++)
                    for (int j1=j; j1<j+B; j1++)
                        for (int k1=k; k1<k+B; k1++)
                            c[i1*n+j1] += a[i1*n+k1] * b[k1*n+j1];
}
Cache Miss Analysis: Standard vs Blocked Matrix Mult

**Standard:**
- computing element $c[i][j]$ costs
  - $n/8$ misses for $a[i][k]$
  - $n$ misses for $b[k][j]$
  - $= 9n/8$ misses per element
- there are $n^2$ elements in $c$
- total misses: $9n/8 \times n^2 = (9/8) \times n^3$

**Blocked:**
- for block size $B \times B$ such that $3B^2$ lines fit in cache
  - $B^2/8$ misses per block
  - $2n/B$ input blocks touched per result block
  - $n^2/B^2$ result blocks
- total misses: $B^2/8 \times 2n/B \times n^2/B^2$
  - $= 1/(4B) \times n^3$
- example: 4096 cache lines, $B=32$
  - $1/128 \times n^3$ misses

* Assuming $nxn$ matrices, 8 doubles per cache line, and cache size much smaller than $n$ lines
Cache Lab: Blocking for Efficient Matrix Transposition

- Divide matrix into sub-matrices that best fit into cache
- The optimum size depends on the cache parameters and matrix size
  - try different sub-matrix sizes
- CacheLab's cache specs:
  - 1 KB direct-mapped (E=1)
  - 32 sets (s=5) with block size 32 bytes (b=5)
- Your test matrices:
  - 32 by 32
  - 64 by 64
  - 61 by 67

It's OK to special-case each of these sizes
If You Get Stuck

- Please read the writeup. Please read the writeup. Please read the writeup. Please read the writeup!
- CS:APP Chapter 6
- View lecture notes and course FAQ at http://www.cs.cmu.edu/~213
- Office hours Sunday through Thursday 5:00-9:00pm in WeH 5207
- Post a private question on Piazza
- man malloc, man gdb, gdb's help command
If I had a penny for every time someone asked a question answered in the writeup....
Appendix: Programming Style

- Properly document your code
  - header comments, overall operation of large blocks, any tricky bits
- Write robust code – check error and failure conditions
- Write modular code
  - use interfaces for data structures, e.g. create/insert/remove/free functions for a linked list
  - no magic numbers – use #define
- Formatting
  - 80 characters per line
  - consistent braces and whitespace
- No memory or file descriptor leaks