University of Duisburg-Essen

Networked Embedded Systems Group

Institute for Computer Science and
Business Information Systems (ICB)
Schützenbahn 70
D-45127 Essen, Germany

Systems Programming
Assignment – 2 – CacheLab: Description

Dr. Matteo Zella

Agenda

- Cache Lab Overview
  - Part (a): Building Cache Simulator
  - Part (b): Efficient Matrix Transpose

Caches

- Memory Address
  - Block offset: b bits
  - Set index: s bits
  - Tag bits = address size – b – s
- A cache is a set of $S = 2^s$ 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 $B = 2^b$ bytes
- Total capacity = S * B * E

NES UNIVERSITY OF DUISBURG ESSEN

Cache Lab

- Part (a): Building a cache simulator
- Part (b): Optimizing matrix transpose

Part (a): Cache Simulator

- A cache simulator is NOT a cache!

  ◦ Memory contents 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 s, b, E, given at run time.

- Use LRU – Least Recently Used replacement policy

  ◦ Evict the least recently used block from the cache to make room for the next block.

  ◦ Queues? Time Stamps?

Part (a): 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 (only if you are not using a queue)

Part (a): 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

  - #include <unistd.h>

Part (a): 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

- For more information,
  - look at man 3 getopt
  - http://www.gnu.org/software/libc/manual/html_node/Getopt.html

Part (a): getopt Example

```
int main(int argc, char** argv){
    int opt,x,y;
    /* looping over arguments */
    while(-1 != (opt = getopt(argc, argv, "x:y:"))){
        /* determine which argument it's processing */
        switch(opt) {
            case 'x':
                x = atoi(optarg);
                break;
            case 'y':
                y = atoi(optarg);
                break;
            default:
                printf("wrong argument\n");
                break;
        }
    }
}
```

- Suppose the program executable was called "foo"; Then we would call "./foo -x 1 –y 3" to pass the value 1 to variable x and 3 to y.

NES
UNIVERSITY OF
DUISBURG
ESSEN

Part (a): fscanf

- The fscanf() function is just like scanf() except it can specify a stream to read from (scanf always reads from stdin)
    - parameters:
        - A stream pointer
        - format string with information on how to parse the file
        - the rest are pointers to variables to store the parsed data
    - You typically want to use this function in a loop. It returns -1 when it hits EOF or if the data doesn't match the format string
- For more information,
    - man fscanf
    - http://crasseux.com/books/ctutorial/fscanf.html
- fscanf will be useful in reading lines from the trace files.
    - L 10,1
    - M 20,1

NES UNIVERSITY OF DUISBURG ESSEN

Part (a): fscanf Example

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

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

char identifier;
unsigned address;
int size;
// Reading lines like " M 20,1" or "L 19,3"

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

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

Part (a): Malloc/free

- Use malloc 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

Part (b) Efficient Matrix Transpose

- Matrix Transpose (A -> B)

Matrix A

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

Matrix B

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

- How do we optimize this operation using the cache?

Part (b): Efficient Matrix Transpose

- Suppose Block size is 8 bytes

Matrix A

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

Matrix B

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16
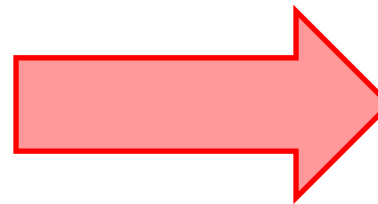
- Access A[0][0] cache miss
- Access B[0][0] cache miss
- Access A[0][1] cache hit
- Access B[1][0] cache miss

Should we handle 3 & 4 next or 5 & 6 ?

Part (b): Blocking

- Blocking: divide matrix into sub-matrices

- Size of sub-matrix depends on cache block size, cache size, input matrix size

- Try different sub-matrix sizes

Example: Matrix Multiplication

```c
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++)
        for (j = 0; j < n; j++)
            for (k = 0; k < n; k++)
                c[i*n + j] += a[i*n + k] * b[k*n + j];
}
```
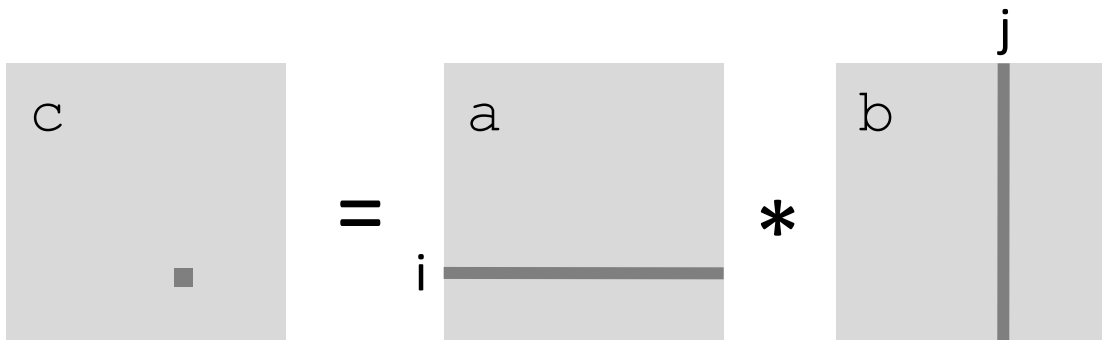
Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- First iteration:
  - n/8 + n = 9n/8 misses
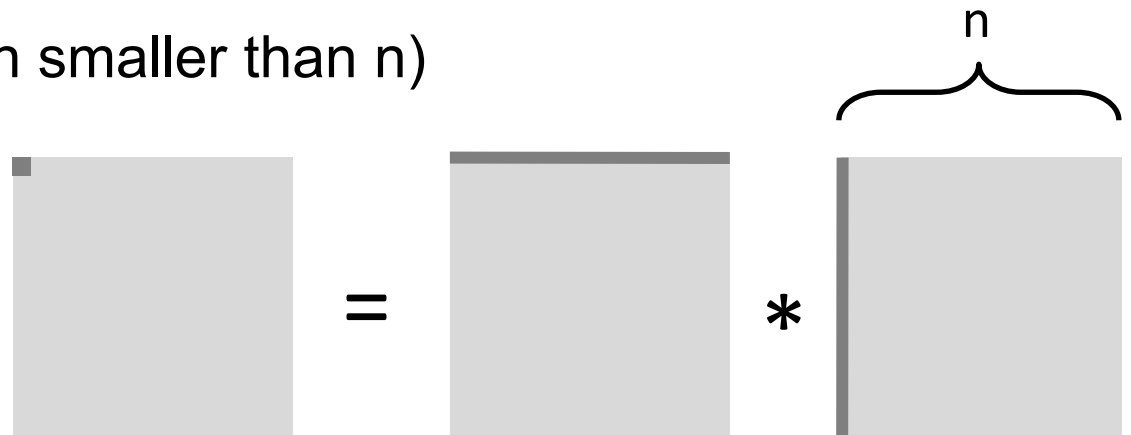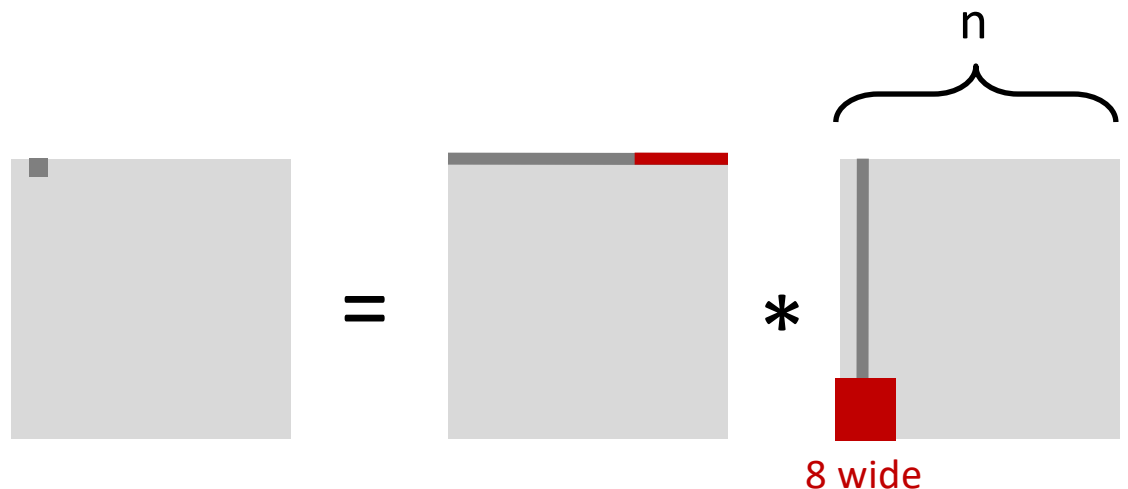
  - Afterwards in cache: (schematic)

Cache Miss Analysis

- Assume:
  - Matrix elements are doubles
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)

- Second iteration:
  - Again:
    n/8 + n = 9n/8 misses
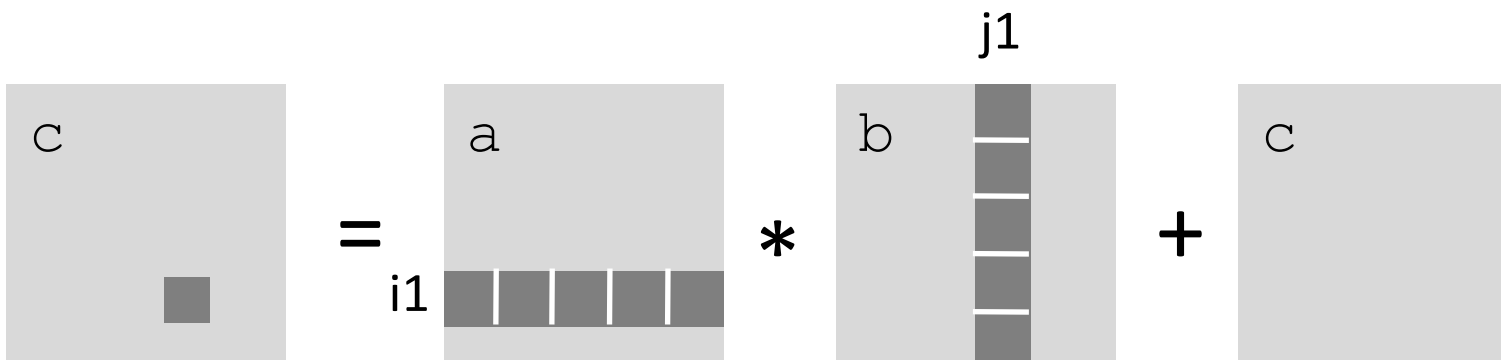
- Total misses:
  - $9n/8 * n^2 = (9/8) * n^3$

n

= *

8 wide

Blocked Matrix Multiplication

```c
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)
                /* 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; k++)
                            c[i1*n+j1] += a[i1*n + k1]*b[k1*n + j1];
}
```
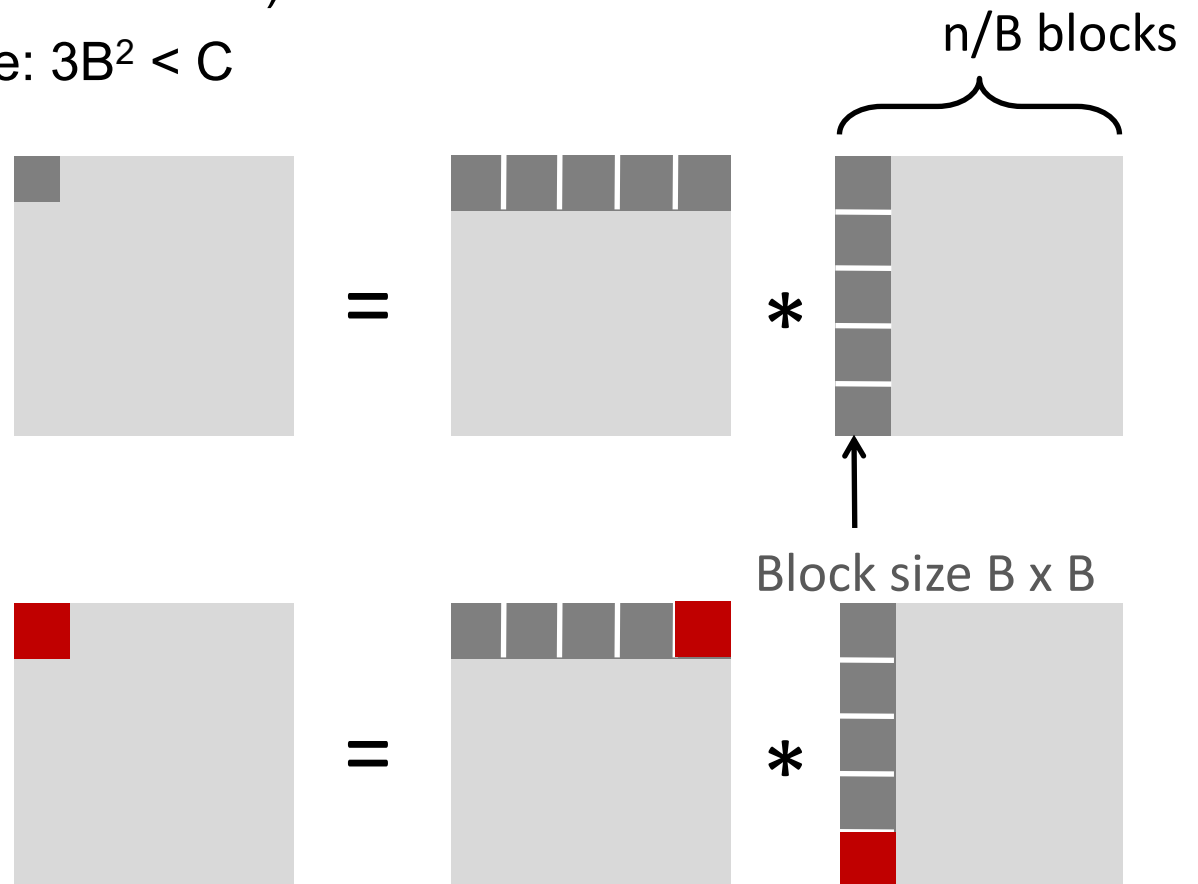
j1

c = a * b + c

i1

Block size B x B

NES UNIVERSITY OF DUISBURG ESSEN

Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

- First (block) iteration:
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$ (omitting matrix c)

  - Afterwards in cache (schematic)

n/B blocks

$=$ $*$
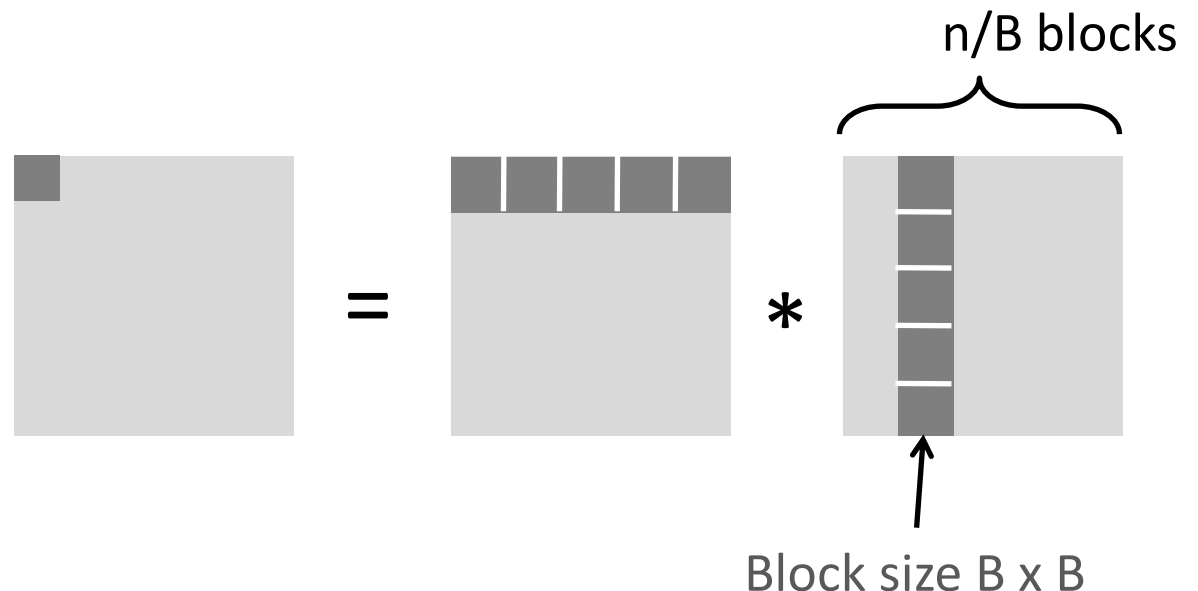
Block size B x B

$=$ $*$

Cache Miss Analysis

- Assume:
  - Cache block = 8 doubles
  - Cache size C << n (much smaller than n)
  - Three blocks ■ fit into cache: $3B^2 < C$

- Second (block) iteration:
  - Same as first iteration
  - $2n/B * B^2/8 = nB/4$

- Total misses:
  - $nB/4 * (n/B)^2 = n^3/(4B)$

n/B blocks

= *

Block size B x B

Part (b): Blocking Summary

- No blocking: $(9/8) * n^3$
- Blocking: $1/(4B) * n^3$

- Suggest largest possible block size B, but limit $3B^2 < C$!

- Reason for dramatic difference:
  - Matrix multiplication has inherent temporal locality:
    - Input data: $3n^2$, computation $2n^3$
    - Every array elements used $O(n)$ times!
  - But program must be written properly

- For a detailed discussion of blocking:
  - http://csapp.cs.cmu.edu/public/waside.html

Part (b): Specs

- 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

NES UNIVERSITY OF DUISBURG ESSEN

Part (b): Eviction Policies of Caches

- The first row of Matrix A evicts the first row of Matrix B
    - Caches are memory aligned
    - Matrix A and B are stored in memory at addresses such that both the first elements align to the same place in cache!
    - Diagonal elements evict each other

- Matrices are stored in memory in a row major order
    - If the entire matrix can't fit in the cache, then after the cache is full with all the elements it can load, the next elements will evict the existing elements of the cache
    - Example: 4x4 Matrix of integers and a 32-byte cache
        - The third row will evict the first row!

NES UNIVERSITY OF DUISBURG ESSEN