Cache Lab Implementation and Blocking

Recitation 4: Apr 10th, 2019

Minsu Kang

Outline

- **Schedule**
- **Cache lab**
  - Part (a) Building Cache Simulator
  - Part (b) Efficient Matrix Transpose
- **Appendix: Memory organization and Caching**
  - Different types of locality
  - Cache organization
  - Blocking

Class Schedule

- **Cache Lab**
  - Due date: Sunday, Apr 21th

- **Exam Soon ! TT TT**

Cache Lab

- Part (a) Building a cache simulator

- Part (b) Optimizing matrix transpose

Part (a) : Cache simulator

- **Get parameters from command-line**
- **Read a trace file in directory ‘traces’**
- **Implement cache mechanism based on traces files and csim-ref file**
- **You can compare results from csim-ref with those from your code**

=> Please refer cachelab.pdf file in cachelab directory that you’ve downloaded from gitlab

Part (a) : Cache simulator

- **A cache simulator is NOT a cache, just simulator!!**
  - 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.

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

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**
  - If free, you may get runtime error instead of running result from what you implemented.

- **If you don’t know about malloc(or calloc) and free, please refer c review PPT or search them on google**

Other things you consider

- **All warnings are treated as errors!**
  - If your program raises warning sign, then compilation will be stopped!
- **Overflow may happen, when you don't declare enough-sized variable for storing 'address'!**
  - Seriously think about possible range of 'address'.
- **Read addresses in trace files using hexadecimal conversion specifier!**
  - If not, you may get in serious trouble.

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

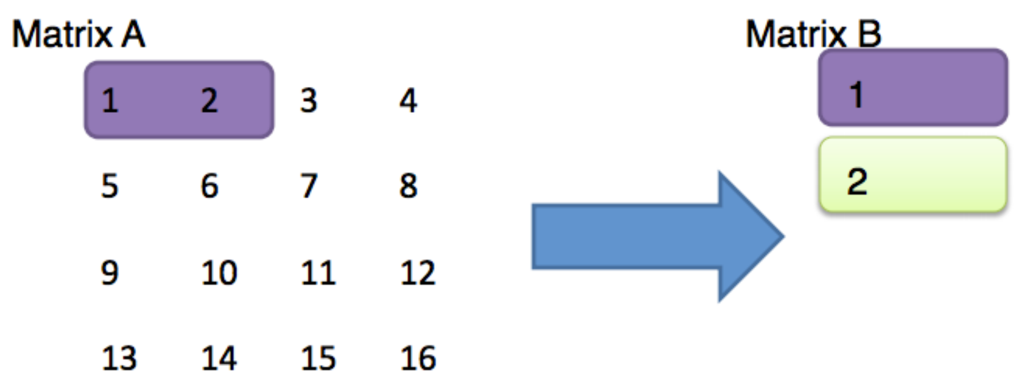
Matrix B

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 ?



- 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.
- You may use blocking to optimize your code.

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

Part (b)

- **Things you'll need to know:**
  - Warnings are errors
  - Header files
  - Eviction policies in the cache

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**

Warnings are Errors: about GCC

- **Used to compile C/C++ projects**
  - List the files that will be compiled to form an executable
  - Specify options via flags

- **Important Flags:**
  - -g: produce debug information (important; used by GDB/valgrind)
  - -Werror: treat all warnings as errors (this is our default)
  - -Wall/-Wextra: enable all construction warnings
  - -pedantic: indicate all mandatory diagnostics listed in C-standard
  - -O0/-O1/-O2: optimization levels
  - -o <filename>: name output binary file ‘filename’

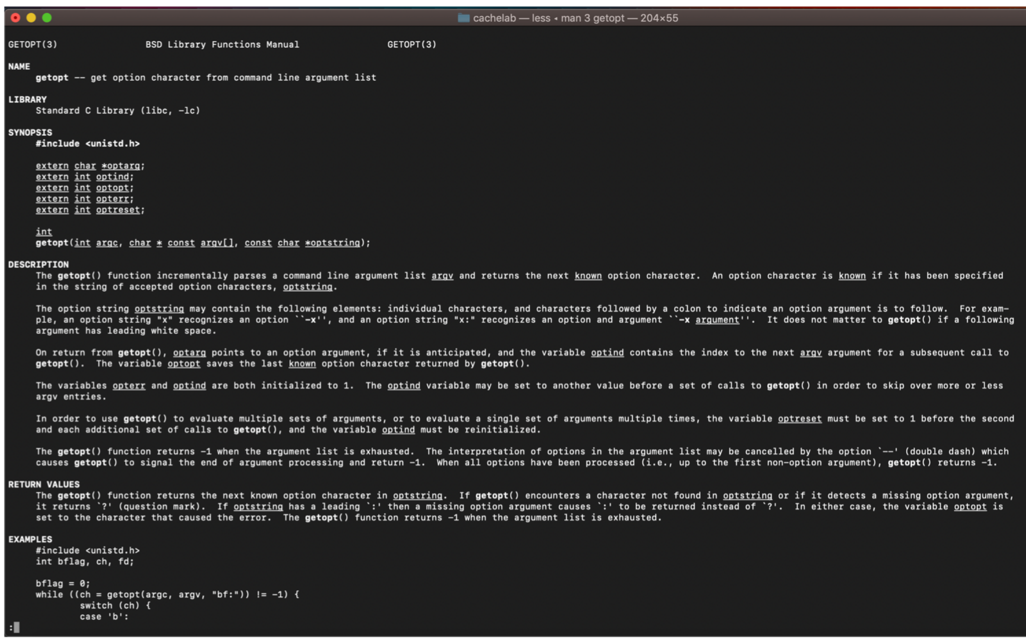
- **Example:**
  - gcc -g -Werror -Wall -Wextra -pedantic foo.c bar.c -o baz

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



```
cachelab — less ◂ man 3 getopt — 204×55

GETOPT(3)                BSD Library Functions Manual                GETOPT(3)

NAME
     getopt -- get option character from command line argument list

LIBRARY
     Standard C Library (libc, -lc)

SYNOPSIS
     #include <unistd.h>

     extern char *optarg;
     extern int optind;
     extern int optopt;
     extern int opterr;
     extern int optreset;

     int
     getopt(int argc, char * const argv[], const char *optstring);

DESCRIPTION
     The getopt() function incrementally parses a command line argument list argv and returns the next known option character.  An option character is known if it has been specified
     in the string of accepted option characters, optstring.

     The option string optstring may contain the following elements: individual characters, and characters followed by a colon to indicate an option argument is to follow.  For exam-
     ple, an option string "x" recognizes an option ``-x'', and an option string "x:" recognizes an option and argument ``-x argument''.  It does not matter to getopt() if a following
     argument has leading white space.

     On return from getopt(), optarg points to an option argument, if it is anticipated, and the variable optind contains the index to the next argv argument for a subsequent call to
     getopt().  The variable optopt saves the last known option character returned by getopt().

     The variables opterr and optind are both initialized to 1.  The optind variable may be set to another value before a set of calls to getopt() in order to skip over more or less
     argv entries.

     In order to use getopt() to evaluate multiple sets of arguments, or to evaluate a single set of arguments multiple times, the variable optreset must be set to 1 before the second
     and each additional set of calls to getopt(), and the variable optind must be reinitialized.

     The getopt() function returns -1 when the argument list is exhausted.  The interpretation of options in the argument list may be cancelled by the option `--' (double dash) which
     causes getopt() to signal the end of argument processing and return -1.  When all options have been processed (i.e., up to the first non-option argument), getopt() returns -1.

RETURN VALUES
     The getopt() function returns the next known option character in optstring.  If getopt() encounters a character not found in optstring or if it detects a missing option argument,
     it returns `?' (question mark).  If optstring has a leading `:' then a missing option argument causes `:' to be returned instead of `?'.  In either case, the variable optopt is
     set to the character that caused the error.  The getopt() function returns -1 when the argument list is exhausted.

EXAMPLES
     #include <unistd.h>
     int bflag, ch, fd;

     bflag = 0;
     while ((ch = getopt(argc, argv, "bf:")) != -1) {
             switch (ch) {
             case 'b':
:
```

Eviction policies of Cache

- **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!

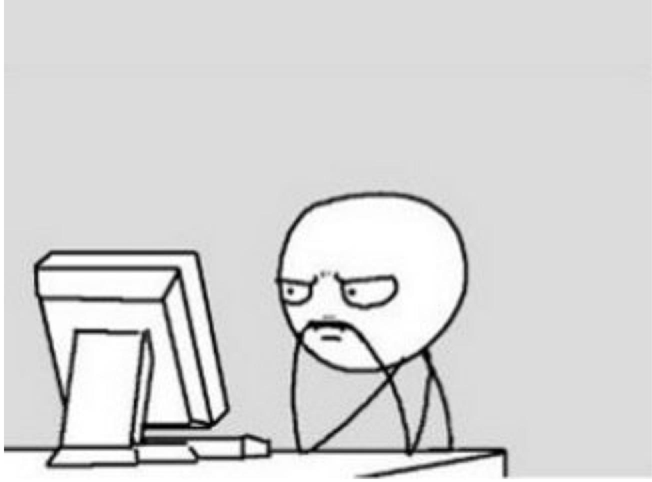
Style

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

- **Start forming good habits now!**

- **Please read cachelab.pdf carefully!**
  - Lots of restriction for implementing your lab exists!

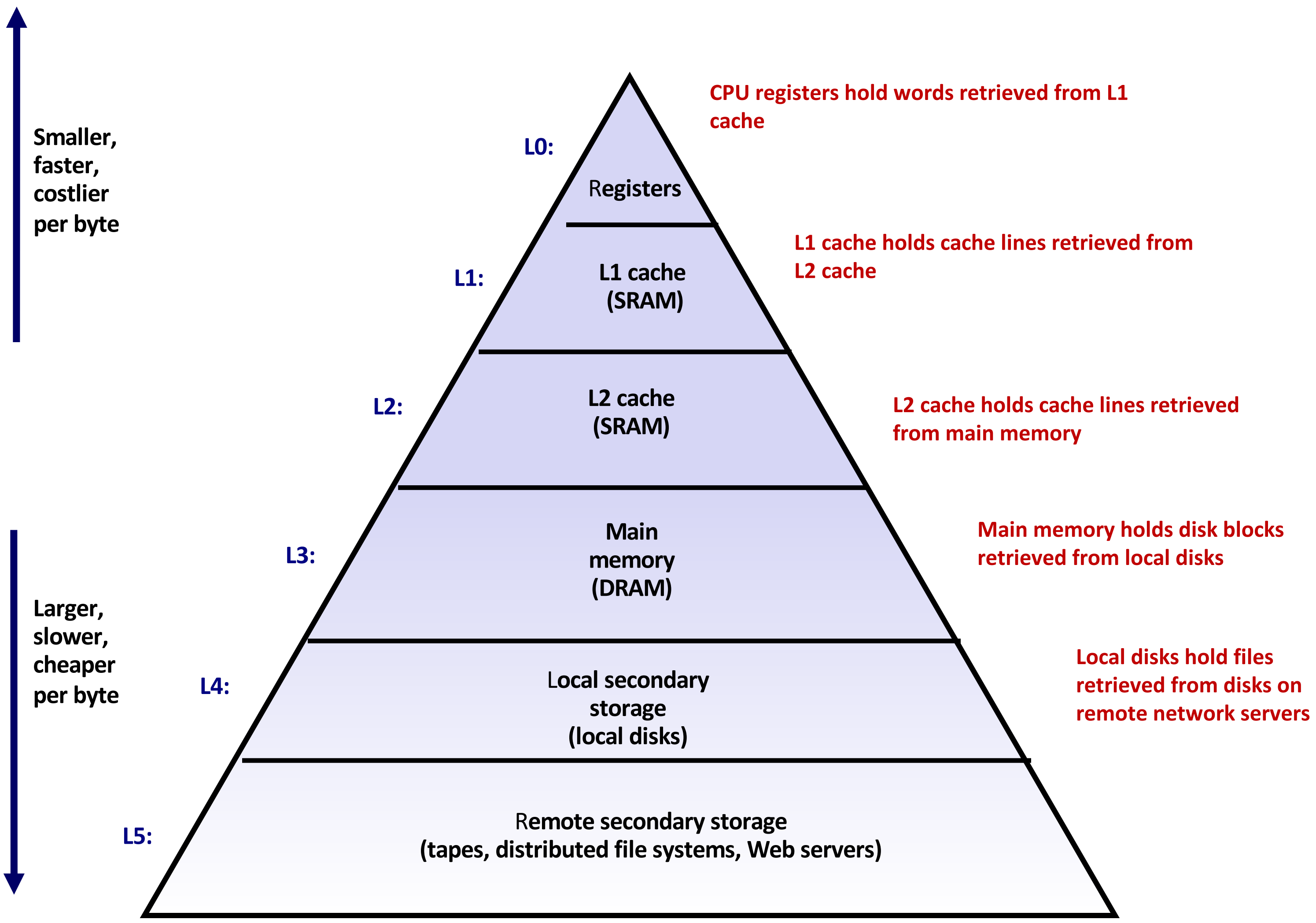
Questions?



Appendix

- **Memory organization and Caching**
  - Different types of locality
  - Cache organization
  - Blocking

Memory Hierarchy



Memory Hierarchy

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

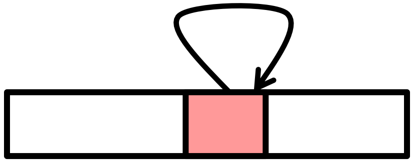
We will discuss this interaction

SRAM vs DRAM tradeoff

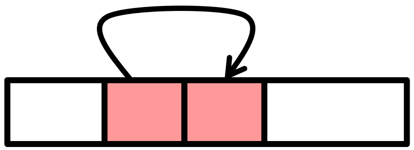
- **SRAM (cache)**
  - Faster (L1 cache: 1 CPU cycle)
  - Smaller (Kilobytes (L1) or Megabytes (L2))
  - More expensive and “energy-hungry”
- **DRAM (main memory)**
  - Relatively slower (hundreds of CPU cycles)
  - Larger (Gigabytes)
  - Cheaper

Locality

- **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

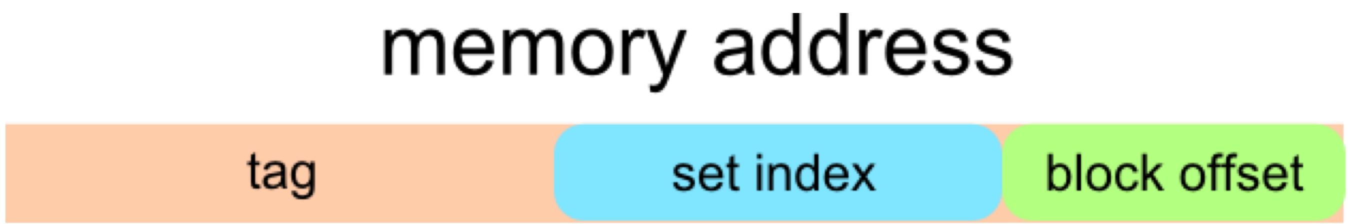


- **Spatial locality**
  - Items with nearby addresses tend to be referenced close together in time
  - After accessing address X, save the block of memory around X in cache for future access



Memory Address

- **64-bit on shark machines**



- **Block offset: b bits**
- **Set index: s bits**
- **Tag Bits: (Address Size – b – s)**

=> More details of parameters: refer textbook p.652~ 653.

Cache

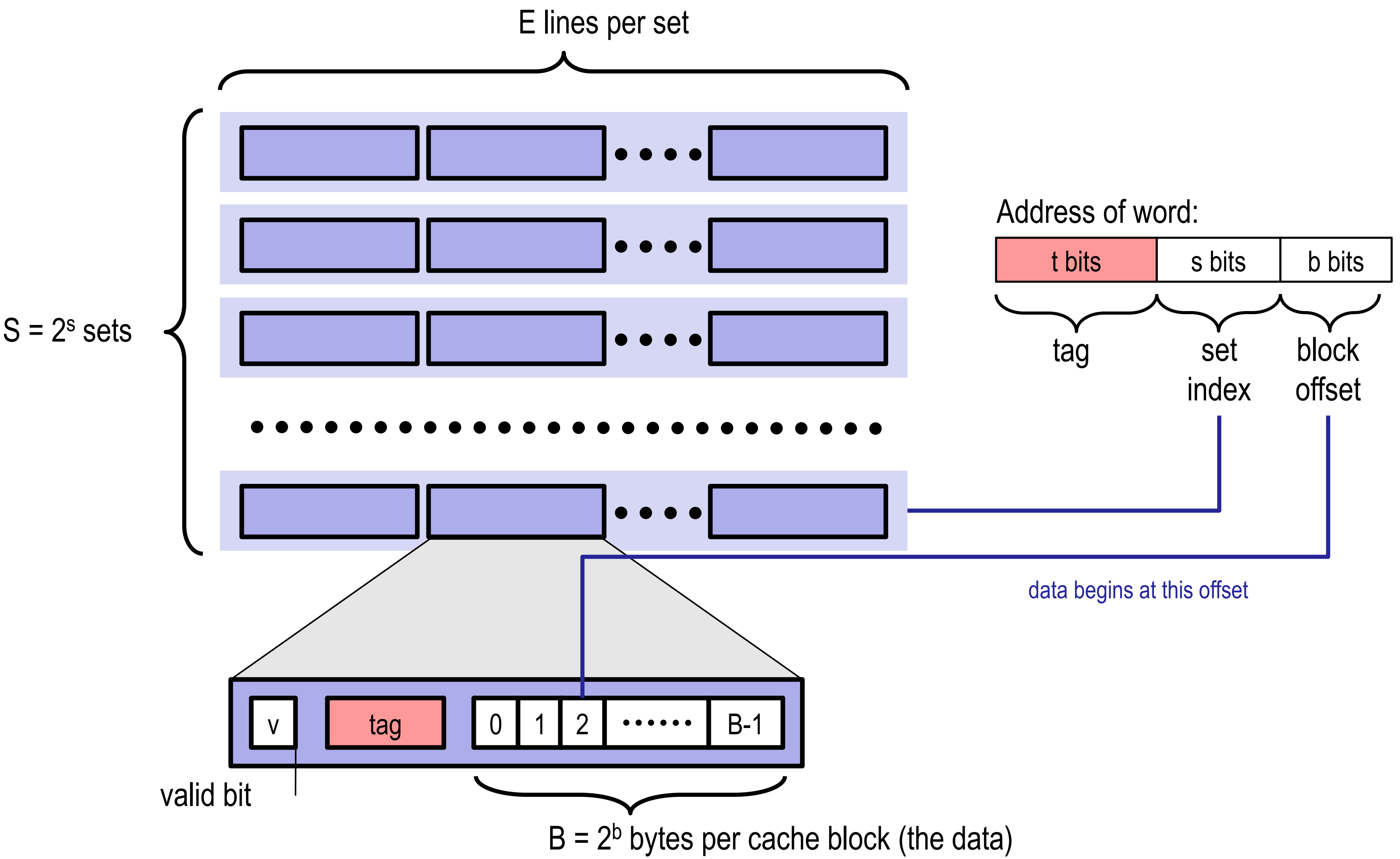
- **A cache is a set of S = 2^s *cache sets***

- **A *cache set* is a set of E *cache lines***
  - E is called associativity
  - If E=1, it is called "direct-mapped"

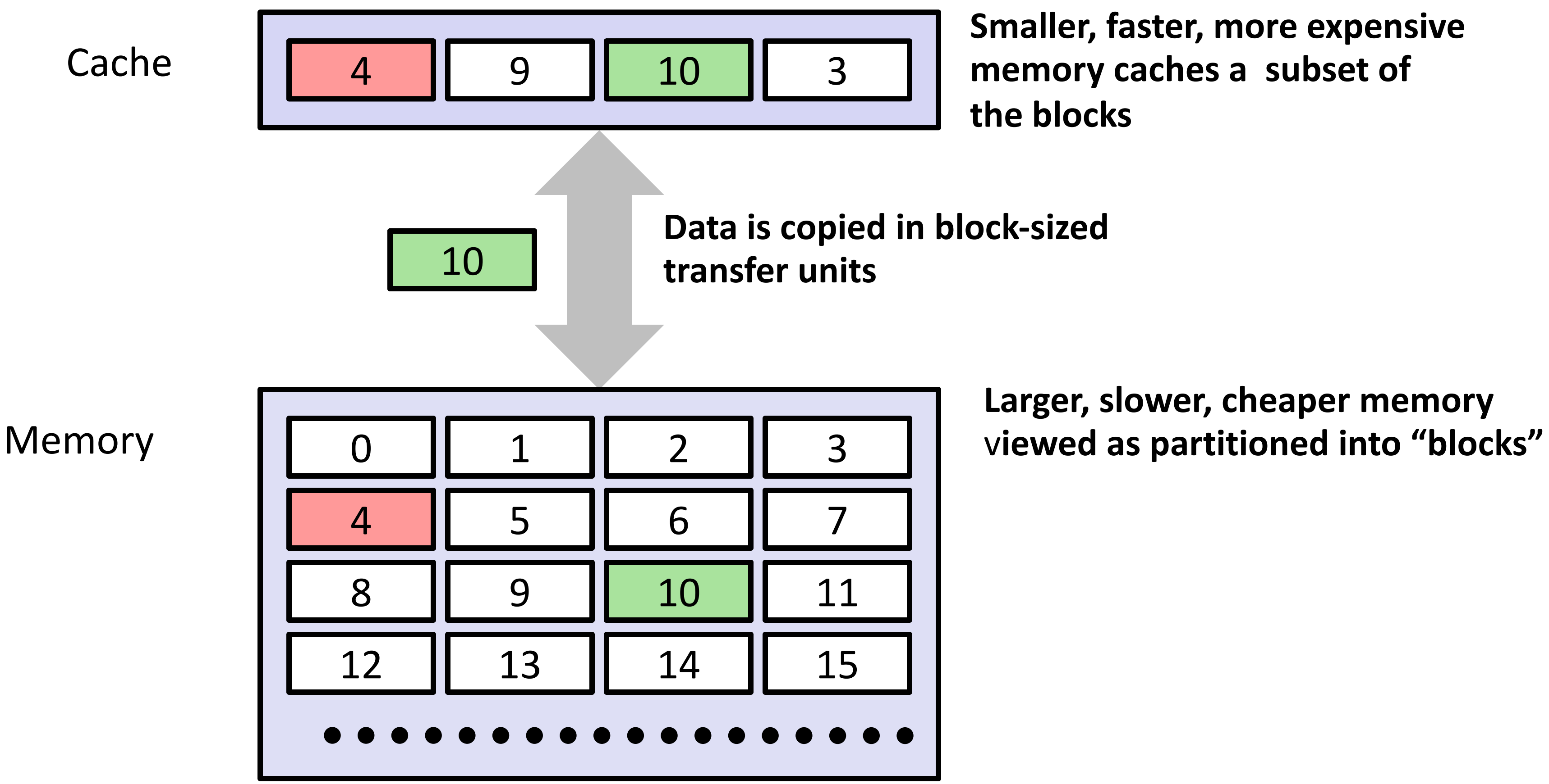
- **Each *cache line* stores a block**
  - Each block has B = 2^b bytes

- **Total Capacity = S*B*E**

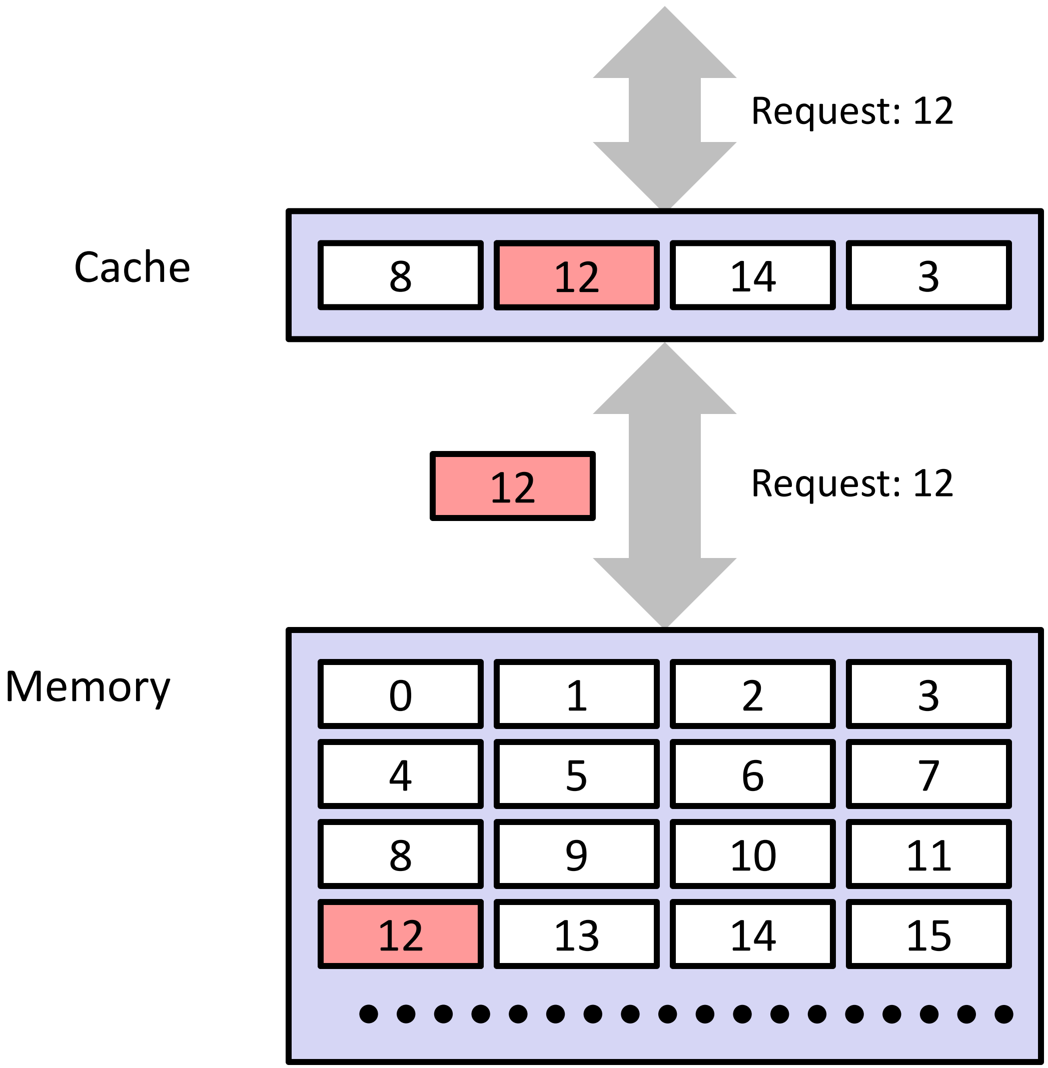
Visual Cache Terminology



General Cache Concepts



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

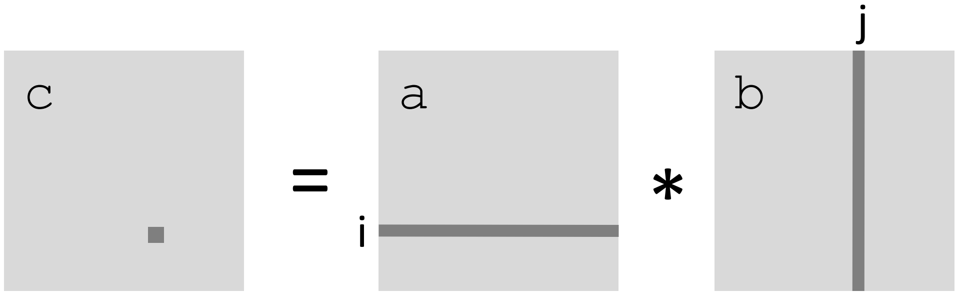
- Placement policy: determines where b goes
- Replacement policy: determines which block gets evicted (victim)

General Caching Concepts: Types of Cache Misses

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

Example: Matrix Multiplication

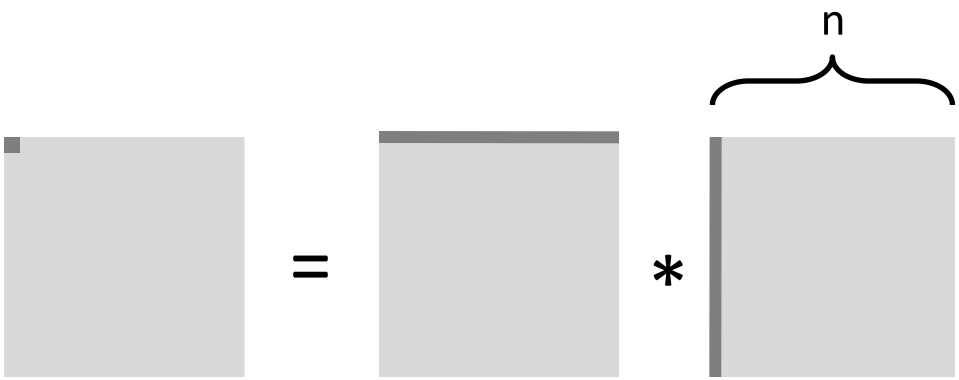
```
c = (double *) calloc(sizeof(double), n*n);

/* 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];
}
```



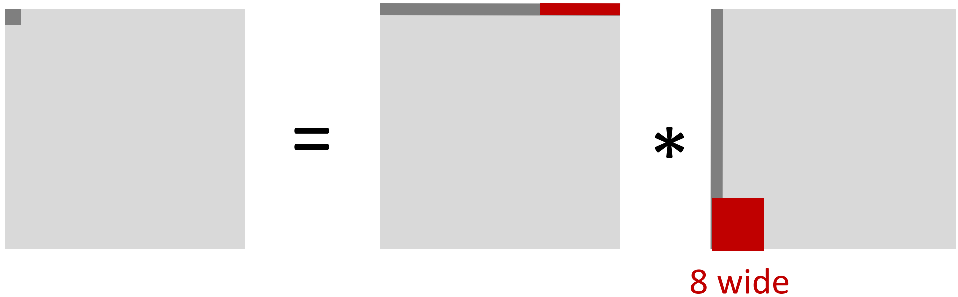
Cache Miss Analysis

- **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



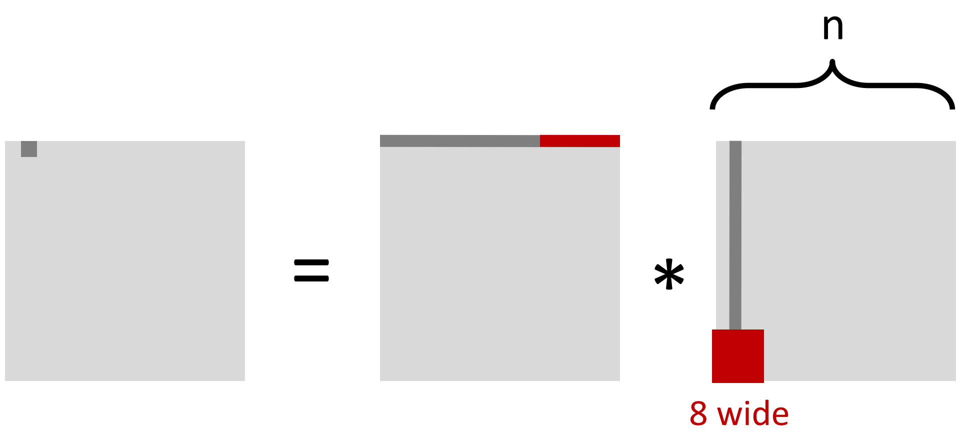
  - Afterwards in cache: (schematic)



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:**
  - $9n/8 * n^2 = (9/8) * n^3$

Blocking

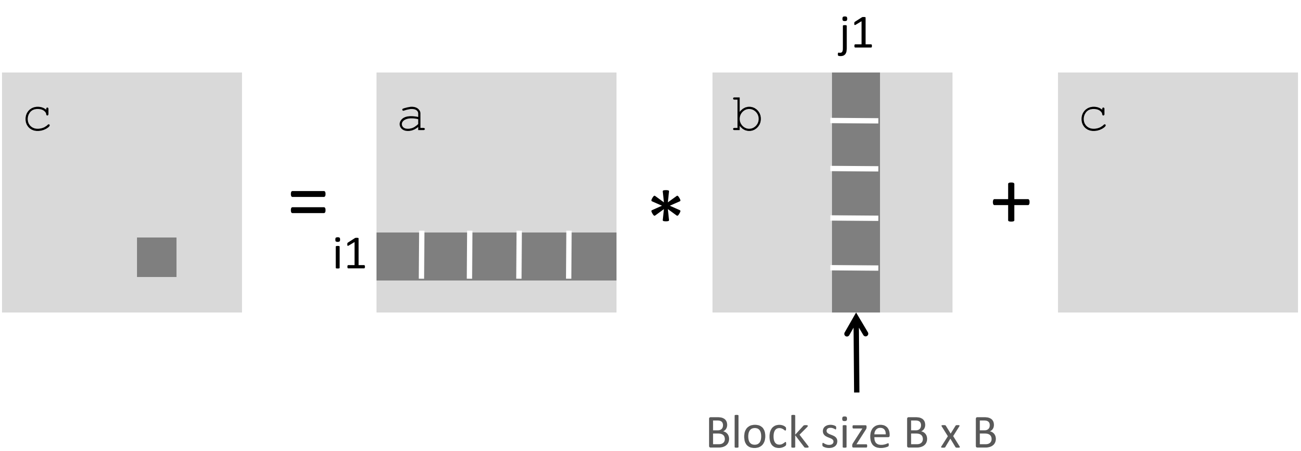
- Blocking: divide matrix into sub-matrices.

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

- Try different sub-matrix sizes.

Blocked Matrix Multiplication

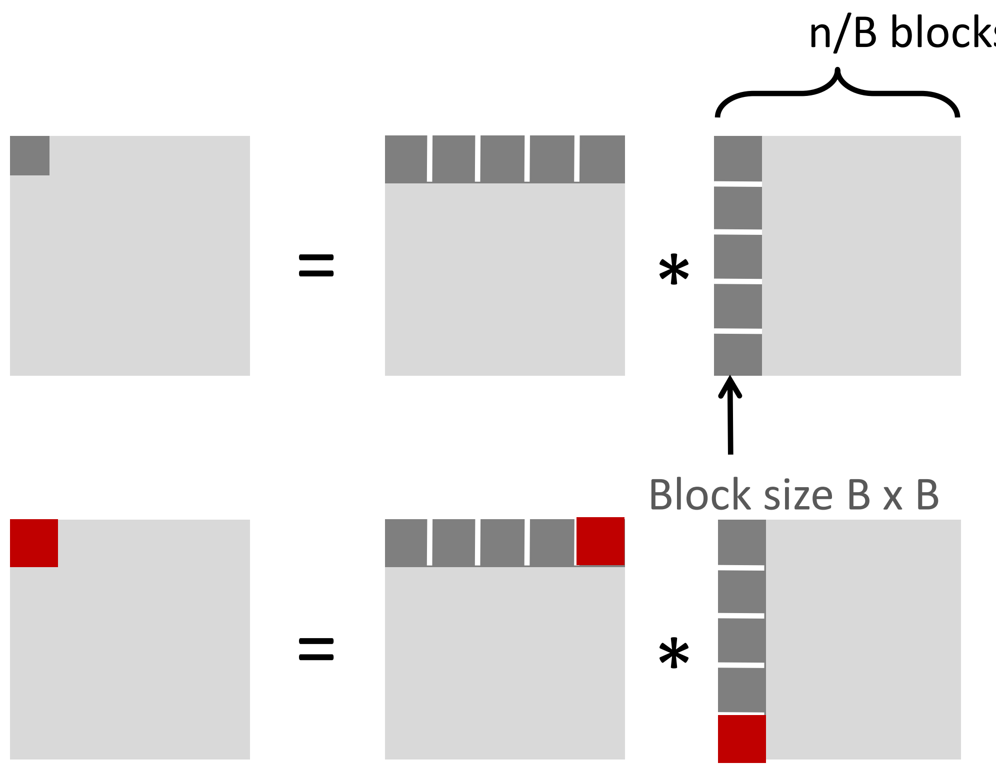
```
c = (double *) calloc(sizeof(double), n*n);

/* 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];
}
```



Cache Miss Analysis

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

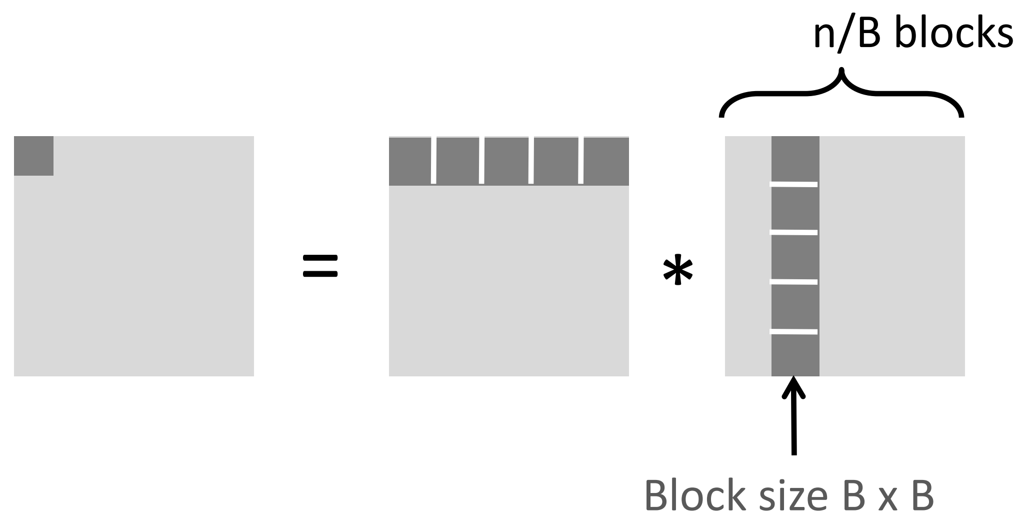
- **First (block) iteration:**
  - $B^2/8$ misses for each block
  - $2n/B * B^2/8 = nB/4$
    (omitting matrix c)
  - Afterwards in cache
    (schematic)



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

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 has to be written properly

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