CS 61C: Great Ideas in Computer Architecture

More Memory:
-set associative caches + dynamic memory

Instructor:
Randy H. Katz

http://inst.eecs.Berkeley.edu/~cs61c/fa13

You Are Here!

- Parallel Requests
  Assigned to computer
e.g., Search "Katz"
- Parallel Threads
  Assigned to core
e.g., Lookup, Ads
- Parallel Instructions
  >1 instruction @ one time
e.g., 5 pipelined instructions
- Parallel Data
  >1 data item @ one time
e.g., Add of 4 pairs of words
- Hardware descriptions
  All gates @ one time
- Programming Languages
 Agenda

• Cache Memory Review
• Set-Associative Caches
• Dynamic Memory

Review: Memory Hierarchy

• Take advantage of the principle of locality to present the user with as much memory as is available in the cheapest technology at the speed offered by the fastest technology
Review: Cache Performance and Average Memory Access Time (AMAT)

- CPU time = IC \times CPI \times CC
- AMAT = \text{Time for a hit} + \text{Miss rate \times Miss penalty}

\[\text{Memory-stall cycles} = \text{Read-stall cycles} + \text{Write-stall cycles}\]
\[
\text{Read-stall cycles} = \frac{\text{reads/program}}{\text{write miss rate}} \times \text{read miss penalty}
\]
\[
\text{Write-stall cycles} = \left(\frac{\text{writes/program}}{\text{write miss rate}} \times \text{write miss penalty}\right) + \text{write buffer stalls}
\]

Improving Cache Performance

- Reduce the time to hit in the cache
  - E.g., Smaller cache, direct-mapped cache, special tricks for handling writes

- \textbf{Reduce the miss rate}
  - E.g., Bigger cache, larger blocks
  - \textit{More flexible placement (increase associativity)}

- Reduce the miss penalty
  - E.g., Smaller blocks or critical word first in large blocks, special tricks for handling writes, faster/higher bandwidth memories
  - Use multiple cache levels
Sources of Cache Misses: The 3Cs

- **Compulsory** (cold start or process migration, 1st reference):
  - First access to block impossible to avoid; small effect for long running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)

- **Capacity:**
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)

- **Conflict (collision):**
  - Multiple memory locations mapped to the same cache location
  - Solution 1: increase cache size
  - Solution 2: increase associativity (may increase access time)

Reducing Cache Misses

- Allow more flexible block placement
- **Direct mapped $\$:** memory block maps to exactly one cache block
- **Fully associative $\$:** allow a memory block to be mapped to any cache block
- Compromise: divide $\$ into sets, each of which consists of n “ways” (**n-way set associative**) to place memory block
  - Memory block maps to unique set determined by index field and is placed in any of the n-ways of that set
  - Calculation: (block address) modulo (# sets in the cache)
Alternative Block Placement Schemes

- **DM placement:** mem block 12 in 8 block cache: only one cache block where mem block 12 can be found—(12 modulo 8) = 4
- **SA placement:** four sets x 2-ways (8 cache blocks), memory block 12 in set (12 mod 4) = 0; either element of the set
- **FA placement:** mem block 12 can appear in any cache blocks

---

Example: 4-Word Direct-Mapped $\$Worst-Case Reference String

- Consider the main memory word reference string

  Start with an empty cache - all blocks initially marked as not valid

  0 4 0 4 0 4 0 4

  0 4 0 4 0 4 0 4

  0 4 0 4 0 4 0 4
**Example: 4-Word Direct-Mapped $ \rightarrow $ Worst-Case Reference String**

- Consider the main memory word reference string

  Start with an empty cache - all blocks initially marked as not valid

  $0 4 0 4 0 4 0 4$

  - 8 requests, 8 misses
  - Ping pong effect due to conflict misses - two memory locations that map into the same cache block

**Example: 2-Way Set Associative $ \rightarrow $ (4 words = 2 sets x 2 ways per set)**

- Compare **all** the cache tags in the set to the **high order 3 memory address bits** to tell if the memory block is in the cache

- Q: How do we find it?

  Use next 1 low order memory address bit to determine which cache set (i.e., modulo the number of sets in the cache)
Example: 4 Word 2-Way SA $  
Same Reference String

• Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

\[
\begin{array}{cccc}
0 & 4 & 0 & 4 \\
\end{array}
\]

Example: 4-Word 2-Way SA $  
Same Reference String

• Consider the main memory word reference string

Start with an empty cache - all blocks initially marked as not valid

\[
\begin{array}{cccc}
0 & 4 & 0 & 4 \\
\end{array}
\]

• 8 requests, 2 misses
• Solves the ping pong effect in a direct mapped cache due to conflict misses since now two memory locations that map into the same cache set can co-exist!
Example: Eight-Block Cache with Different Organizations

Total size of $s$ in blocks is equal to \textit{number of sets} \times \textit{associativity}. For fixed $s$ size, increasing associativity decreases number of sets while increasing number of elements per set. With eight blocks, an 8-way set-associative $s$ is same as a fully associative $s$. 

Four-Way Set-Associative Cache

\begin{itemize}
\item $2^8 = 256$ sets each with four ways (each with one block)
\end{itemize}

Index  V  Tag  Data  V  Tag  Data  V  Tag  Data  V  Tag  Data
\hline
0  &   &   &   & 0  &   &   &   & 0  &   &   &   \\
1  &   &   &   & 1  &   &   &   & 1  &   &   &   \\
2  &   &   &   & 2  &   &   &   & 2  &   &   &   \\
\hline
253 &   &   &   & 253 &   &   &   & 253 &   &   &   \\
254 &   &   &   & 254 &   &   &   & 254 &   &   &   \\
255 &   &   &   & 255 &   &   &   & 255 &   &   &   \\
\hline
31 &   &   &   & 31 &   &   &   & 31 &   &   &   \\
30 &   &   &   & 30 &   &   &   & 30 &   &   &   \\
15 &   &   &   & 15 &   &   &   & 15 &   &   &   \\
14 &   &   &   & 14 &   &   &   & 14 &   &   &   \\
13 &   &   &   & 13 &   &   &   & 13 &   &   &   \\
12 &   &   &   & 12 &   &   &   & 12 &   &   &   \\
11 &   &   &   & 11 &   &   &   & 11 &   &   &   \\
10 &   &   &   & 10 &   &   &   & 10 &   &   &   \\
9  &   &   &   & 9  &   &   &   & 9  &   &   &   \\
8  &   &   &   & 8  &   &   &   & 8  &   &   &   \\
7  &   &   &   & 7  &   &   &   & 7  &   &   &   \\
6  &   &   &   & 6  &   &   &   & 6  &   &   &   \\
5  &   &   &   & 5  &   &   &   & 5  &   &   &   \\
4  &   &   &   & 4  &   &   &   & 4  &   &   &   \\
3  &   &   &   & 3  &   &   &   & 3  &   &   &   \\
2  &   &   &   & 2  &   &   &   & 2  &   &   &   \\
1  &   &   &   & 1  &   &   &   & 1  &   &   &   \\
0  &   &   &   & 0  &   &   &   & 0  &   &   &   \\
\hline
\end{itemize}
Flashcard Quiz: For fixed capacity and fixed block size, how does increasing associativity effect AMAT?

* Increases hit time, decreases miss rate
* Decreases hit time, decreases miss rate
* Increases hit time, increases miss rate
* Decreases hit time, increases miss rate

Range of Set-Associative Caches

- For a fixed-size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit

| Tag | Index | Block offset | Byte offset |
Range of Set-Associative Caches

- For a fixed-size cache, each increase by a factor of two in associativity doubles the number of blocks per set (i.e., the number or ways) and halves the number of sets – decreases the size of the index by 1 bit and increases the size of the tag by 1 bit.

<table>
<thead>
<tr>
<th>Used for tag compare</th>
<th>Selects the set</th>
<th>Selects the word in the block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>Index</td>
<td>Block offset</td>
</tr>
</tbody>
</table>

- Decreasing associativity
  - Direct mapped (only one way)
  - Smaller tags, only a single comparator

- Increasing associativity
  - Fully associative (only one set)
  - Tag is all the bits except block and byte offset

Costs of Set-Associative Caches

- When miss occurs, which way’s block selected for replacement?
  - Least Recently Used (LRU): one that has been unused the longest
    - Must track when each way’s block was used relative to other blocks in the set
    - For 2-way SA $S$, one bit per set $\rightarrow$ set to 1 when a block is referenced; reset the other way’s bit (i.e., “last used”)
  - N-way set-associative cache costs
    - N comparators (delay and area)
    - MUX delay (set selection) before data is available
    - Data available after set selection (and Hit/Miss decision).
    - DM $S$: block is available before the Hit/Miss decision
      - In Set-Associative, not possible to just assume a hit and continue and recover later if it was a miss
Cache Block Replacement Policies

• Random Replacement
  – Hardware randomly selects a cache item and throw it out

• Least Recently Used
  – Hardware keeps track of access history
  – Replace the entry that has not been used for the longest time
  – For 2-way set-associative cache, need one bit for LRU replacement

• Example of a Simple “Pseudo” LRU Implementation
  – Assume 64 Fully Associative entries
  – Hardware replacement pointer points to one cache entry
  – Whenever access is made to the entry the pointer points to:
    • Move the pointer to the next entry
  – Otherwise: do not move the pointer

Benefits of Set-Associative Caches

• Choice of DM $ or SA $ depends on the cost of a miss versus the cost of implementation

• Largest gains are in going from direct mapped to 2-way (20%+ reduction in miss rate)
How to Calculate 3C’s using Cache Simulator

1. **Compulsory**: set cache size to infinity and fully associative, and count number of misses

2. **Capacity**: Change cache size from infinity, usually in powers of 2, and count misses for each reduction in size
   - 16 MB, 8 MB, 4 MB, … 128 KB, 64 KB, 16 KB

3. **Conflict**: Change from fully associative to n-way set associative while counting misses
   - Fully associative, 16-way, 8-way, 4-way, 2-way, 1-way

---

**3Cs Revisted**

- Three sources of misses (SPEC2000 integer and floating-point benchmarks)
  - Compulsory misses 0.006%; not visible
  - Capacity misses, function of cache size
  - Conflict portion depends on associativity and cache size
Improving Cache Performance

• Reduce the time to hit in the cache
  – E.g., Smaller cache, direct-mapped cache, special tricks for handling writes
• Reduce the miss rate
  – E.g., Bigger cache, larger blocks
  – More flexible placement (increase associativity)
• Reduce the miss penalty
  – E.g., Smaller blocks or critical word first in large blocks, special tricks for handling for writes, faster/higher bandwidth memories
  – Use multiple cache levels

Reduce AMAT

• Use multiple levels of cache
• As technology advances, more room on IC die for larger L1$ or for additional levels of cache (e.g., L2$ and L3$)
• Normally the higher cache levels are unified, holding both instructions and data
**Design Considerations**

- Different design considerations for L1$ and L2$
  - L1$ focuses on **fast access**: minimize hit time to achieve shorter clock cycle, e.g., smaller $
  - L2$, L3$ focus on **low miss rate**: reduce penalty of long main memory access times: e.g., Larger $ with larger block sizes/higher levels of associativity
- Miss penalty of L1$ is significantly reduced by presence of L2$, so can be smaller/faster even with higher miss rate
- For the L2$, fast hit time is less important than low miss rate
  - L2$ hit time determines L1$’s miss penalty
  - L2$ local miss rate $>>$ than the global miss rate

**Flashcard Quiz:** In a machine with two levels of cache, what effect does increasing L1 capacity have on L2*?

- **Decreases L2 capacity misses**
- **Increases L2 local miss rate**
- **Decreases L2 compulsory misses**
- **Decreases L2 global miss rate**
Cache Design Space

- Several interacting dimensions
  - Cache size
  - Block size
  - Associativity
  - Replacement policy
  - Write-through vs. write-back
  - Write allocation
- Optimal choice is a compromise
  - Depends on access characteristics
    - Workload
    - Use (I-cache, D-cache)
  - Depends on technology / cost
- Simplicity often wins

Recap: C Memory Management

- Program’s address space contains 4 regions:
  - stack: local variables, grows downward
  - heap: space requested for pointers via malloc(); resizes dynamically, grows upward
  - static data: variables declared outside main, does not grow or shrink
  - code: loaded when program starts, does not change

OS prevents accesses between stack and heap (via virtual memory)
Recap: Where are Variables Allocated?

- If declared outside a procedure, allocated in “static” storage
- If declared inside procedure, allocated on the “stack” and freed when procedure returns
  - main() is treated like a procedure

```
int myGlobal;
main() {
    int myTemp;
}
```

Recap: The Stack

- Stack frame includes:
  - Return “instruction” address
  - Parameters
  - Space for other local variables
- Stack frames contiguous blocks of memory; stack pointer indicates top of stack frame
- When procedure ends, stack frame is tossed off the stack; frees memory for future stack frames
Recap: The Stack

- Last In, First Out (LIFO) data structure

```
main ()
{
    a(0);
    void a (int m)
    {
        b(1);
        void b (int n)
        {
            c(2);
            void c (int o)
            {
                d(3);
                void d (int p)
                {
                }
            }
        }
    }
}
```

Observations

- Code, Static storage are easy: they never grow or shrink
- Stack space is relatively easy: stack frames are created and destroyed in last-in, first-out (LIFO) order
- Managing the heap is tricky: memory can be allocated / deallocated at any time
Managing the Heap

- C supports five functions for heap management: `malloc()`, `calloc()`, `free()`, `cfree()`, `realloc()`
- `malloc(n)`:
  - Allocate a block of uninitialized memory
  - **NOTE**: Subsequent calls need not yield blocks in continuous sequence
  - `n` is an integer, indicating size of allocated memory block in bytes
  - `sizeof` determines size of given type in bytes, produces more portable code
  - Returns a pointer to that memory location; NULL return indicates no more memory
  - Think of `ptr` as a handle that also describes the allocated block of memory;
    Additional control information stored in the heap around the allocated block!

- **Example**:
  ```c
  int *ip;
  ip = malloc(sizeof(int));

  struct treeNode *tp;
  tp = malloc(sizeof(struct treeNode));
  ```

- **free(p)**:
  - Releases memory allocated by `malloc()`
  - `p` is pointer containing the address originally returned by `malloc()`
  ```c
  int *ip;
  ip = malloc(sizeof(int));

  ... ...

  free(ip); /* Can you free(ip) after ip++ ? */

  struct treeNode *tp;
  tp = malloc(sizeof(struct treeNode));

  ... ...

  free(tp);
  ```
  - When insufficient free memory, `malloc()` returns NULL pointer; **Check for it!**
  ```c
  if ((ip = malloc(sizeof(int))) == NULL){
    printf("\nMemory is FULL\n");
    exit(1);
  }
  ```
  - When you free memory, you must be sure that you pass the **original address**
    returned from `malloc()` to `free()`. Otherwise, system exception (or worse)!
Common Memory Problems

- Using uninitialized values
- Using memory that you don’t own
  - Deallocated stack or heap variable
  - Out-of-bounds reference to stack or heap array
  - Using NULL or garbage data as a pointer
- Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
- Memory leaks (you allocated something you forgot to later free)

Memory Debugging Tools

- Runtime analysis tools for finding memory errors
  - Dynamic analysis tool: collects information on memory management while program runs
  - Contrast with static analysis tool like lint, which analyzes source code without compiling or executing it
  - No tool is guaranteed to find ALL memory bugs – this is a very challenging programming language research problem
  - Runs 10X slower

http://valgrind.org
Using Memory You Don’t Own

• What is wrong with this code?

```c
int *ipr, *ipw;
void ReadMem() {
    int i, j;
    *ipr = malloc(4 * sizeof(int));
    i = *(ipr - 1000); j = *(ipr + 1000);
    free(ipr);
}

void WriteMem() {
    *ipw = malloc(5 * sizeof(int));
    *(ipw - 1000) = 0; *(ipw + 1000) = 0;
    free(ipw);
}
```

How are Malloc/Free implemented?

• Underlying operating system allows malloc library to ask for large blocks of memory to use in heap (e.g., using Unix sbrk() call)

• C Malloc library creates data structure inside unused portions to track free space
Simple Slow Malloc Implementation

- Initial Empty Heap space from Operating System
- Free Space
- Malloc library creates linked list of empty blocks (one block initially)
- Object 1
- First allocation chews up space from start of free space
- After many mallocs and frees, have potentially long linked list of odd-sized blocks
  Frees link block back onto linked list – might merge with neighboring free space

Faster malloc implementations

- Keep separate pools of blocks for different sized objects
- “Buddy allocators” always round up to power-of-2 sized chunks to simplify finding correct size and merging neighboring blocks:
Power-of-2 "Buddy Allocator"

Malloc Implementations

- All provide the same library interface, but can have radically different implementations
- Uses headers at start of allocated blocks and space in unallocated memory to hold malloc’s internal data structures
- Rely on programmer remembering to free with same pointer returned by alloc
- Rely on programmer not messing with internal data structures accidentally!
Faulty Heap Management

• What is wrong with this code?

```c
int *pi;
void foo() {
    pi = malloc(8*sizeof(int));
    ...
    free(pi);
}

void main() {
    pi = malloc(4*sizeof(int));
    foo();
    ...
}
```

Faulty Heap Management

• What is wrong with this code?

```c
int *plk = NULL;
void genPLK() {
    plk = malloc(2 * sizeof(int));
    ...
    plk++;
}
```
Faulty Heap Management

• What is wrong with this code?

```c
void FreeMemX() {
    int fnh = 0;
    free(&fnh);
}

void FreeMemY() {
    int *fum = malloc(4 * sizeof(int));
    free(fum+1);
    free(fum);
    free(fum);
}
```

Using Memory You Haven’t Allocated

• What is wrong with this code?

```c
void StringManipulate() {
    const char *name = "Safety Critical";
    char *str = malloc(10);
    strncpy(str, name, 10);
    str[10] = '\0';
    printf("%s\n", str);
}
```
Using Memory You Don’t Own

• What’s wrong with this code?

```c
char *append(const char* s1, const char *s2) {
    const int MAXSIZE = 128;
    char result[MAXSIZE];
    int i=0, j=0;
    for (j=0; i<MAXSIZE-1 && j<strlen(s1); i++,j++) {
        result[i] = s1[j];
    }
    for (j=0; i<MAXSIZE-1 && j<strlen(s2); i++,j++) {
        result[i] = s2[j];
    }
    result[++i] = '\0';
    return result;
}
```

Using Memory You Don’t Own

• What is wrong with this code?

```c
typedef struct node {
    struct node* next;
    int val;
} Node;

int findLastNodeValue(Node* head) {
    while (head->next != NULL) {
        head = head->next;
    }
    return head->val;
}
```
Managing the Heap

• calloc(n, size):
  - Allocate n elements of same data type; n can be an integer variable, use calloc() to allocate a dynamically size array
  - n is the # of array elements to be allocated
  - size is the number of bytes of each element
  - calloc() guarantees that the memory contents are initialized to zero

  E.g.: allocate an array of 10 elements
       
       int *ip;
       ip = calloc(10, sizeof(int));
       *(ip+1) refers to the 2nd element, like ip[1]
       *(ip+i) refers to the i+1th element, like ip[i]

       Beware of referencing beyond the allocated block: e.g., *(ip+10)
  - calloc() returns NULL if no further memory is available

• cfrry(p) // Legacy function – same as free
  - cfrry() releases the memory allocated by calloc(); E.g.: cfrry(ip);

• realloc(p, size):
  - Resize a previously allocated block at p to a new size
  - If p is NULL, then realloc behaves like malloc
  - If size is 0, then realloc behaves like free, deallocating the block from the heap
  - Returns new address of the memory block; NOTE: it is likely to have moved!

  E.g.: allocate an array of 10 elements, expand to 20 elements later
       
       int *ip;
       ip = malloc(10*sizeof(int));
       /* always check for ip == NULL */
       ... ... 
       ip = realloc(ip,20*sizeof(int));
       /* always check for ip == NULL */
       /* contents of first 10 elements retained */
       ... ... 
       realloc(ip,0); /* identical to free(ip) */
Using Memory You Don’t Own

• What is wrong with this code?

```c
int* init_array(int *ptr, int new_size) {
    ptr = realloc(ptr, new_size*sizeof(int));
    memset(ptr, 0, new_size*sizeof(int));
    return ptr;
}

int* fill_fibonacci(int *fib, int size) {
    int i;
    init_array(fib, size);
    /* fib[0] = 0; */ fib[1] = 1;
    for (i=2; i<size; i++)
        fib[i] = fib[i-1] + fib[i-2];
    return fib;
}
```

And, In Conclusion ...

• Name of the Game: Reduce Cache Misses
  – One way to do it: set-associativity
  – N-way Cache of $2^{N+M}$ blocks: $2^N$ ways x $2^M$ sets
  – Multi-level caches: Optimize 1st level to be fast! 2nd and 3rd to minimize memory access penalty

• Dynamic Memory
  – Program Storage + Static storage + Stack Storage
  – The Heap (dynamic storage): `malloc()` and `free()`

• Common Dynamic Memory Problems
  – Using uninitialized values
  – Accessing memory beyond your allocated region
  – Improper use of free by changing pointer handle returned by malloc
  – Memory leaks: mismatched malloc/free pairs