Cache-Assignment-Examples

April 6, 2025

1 Cache Assignment

This assignment contains three parts, all revolving around array accesses and the resulting miss rate for a chosen set asociativity. You will be simulating part of the functionality of an L1 cache. To perform a correct write operation requires logic to handle whether a block is valid or not, and write back logic. Therefore, all accesses that we examine in this assignment are for read requests only.

You will examine two iterators accessing an array the size of your `L2` or `L3` cache and the miss rate for each method. Remember that each data address is representing a byte of data only. The first iterator is row major order, and the second is column major order. An pseudo code example is provided for each using a character of 64x64:

1.0.1 Row Major Accesses

```
int i = 0; j = 0;
for (i = 0; i < 64; i++) {
    for (j = 0; j < 64; j++) {
        arr[i][j];  // Simple array access (read operation)
    }
}
```

1.0.2 Column Major Accesses

```
int i = 0; j = 0;
for (j = 0; j < 64; j++) {
    for (i = 0; i < 64; i++) {
        arr[i][j];  // Simple array access (read operation)
    }
}
```

1.1 Assignment Problems

While one may be able to reasonably be able to calculate the read miss rate for the row and column major accesses by hand, this is still a programming assignment.

1. (5 points) Research your CPU to find the size of the cache and report on the size at each level. Describe three commonly used file types, and what their expected file sizes are. Do these examples fit into the largest size of cache? What happens if it doesn't?

2. (5 points) Choose an arbitrary address (within the address size of your machine) to use as the starting address for your arrays. Describe your choice of cache layout (Direct, SA, FA).
3. (40 points) Perform row major and column access reads of your cache using arrays of 4096 bytes each, for each of the following `C` style data types. Provide a miss rate report for each of these data types in a table.
    1. `char` (1 byte): Use an array of 64x64
    2. `short` (2 bytes): Use a 2-D array size of your choice, matching the total bytes of your `L2` or `L3` cache.
    3. `int` (4 bytes): Use an array of 32x32
    4. `long` (8 bytes): Use a 2-D array size of your choice, matching the total bytes of your `L2` or `L3` cache.

Data Type	Miss Rate
char	
short	
int	
long	

4. (40 points) Now, perform random accesses for each of these data types (1, 2, 4, 8) with a range of equalling half the size of your `L2` or `L3` cache offset from your original address. In effect, you should expect a range that equals the size of your `L2` or `L3` cache. Report the miss rate for this address sequence. You may like to find a statistically meaningful miss rate by running this calculation at least 30 times.
5. (15 points) Provide your code, and a brief writeup of your findings.

To start, choose a set associativity for your L1 cache. To match the array addressing of the `RISC-V` instruction set, you will be using a total size of 4096 bytes. You will not need to store the data cache, but you will need to store the tags and a valid bit. Use the MSB of the tag to store this valid bit. Some rules for your cache:

1. On every read, all blocks will be brought up to date.
2. If there is a mismatch in the tags for the requested address, this is a miss.
3. If there is a match in the tags for the requested address, this is a hit.
4. Each read request must be checked against all tags for the set.

1.2 Cache Assignment Examples

[1]:
```
import numpy as np
```

[2]:
```
one_kilobyte = 2**10
2**12 / one_kilobyte
```

[2]:
```
4.0
```

You will need to store the tag, and a valid bit for each read transaction. For example, the following creates the data for a two-way set associative cache (you don't need a data cache, it is just for demonstration):

[3]:
```
cache_data = np.empty((2048, 2), dtype=np.int8)
cache_data
```

[3]:
```
array([[ 0, 32],
       [ 0,  0],
       [ 0,  0],
       …,
       [ 0,  0],
       [ 0,  0],
       [ 0,  0]], dtype=int8)
```

[4]:
```
cache_data.nbytes / one_kilobyte
```

[4]:
```
4.0
```

In order to store the tags, you will need to have an array just as large but now with a data size large enough to store the whole address (minus the index bits).

[5]:
```
cache_tags = np.empty(cache_data.shape, dtype=np.int32)
cache_tags
```

[5]:
```
array([[8192,    0],
       [ 632,    0],
       [   0,    0],
       …,
       [   0,    0],
       [   0,    0],
       [   0,    0]], dtype=int32)
```

[6]:
```
cache_tags.nbytes / one_kilobyte
```

[6]:
```
16.0
```

Generate 4k of random data (addresses):

[7]:
```
random_values = np.random.random((2048, 2))
```

[8]:
```
random_int64 = random_values.astype(np.int64)
random_int64
```

[8]:
```
array([[0, 0],
       [0, 0],
       [0, 0],
       …,
       [0, 0],
       [0, 0],
       [0, 0]])
```

You are required to use address sizes from your cpu architecture and cache sizes. On a linux operating system, the command `lscpu` is available to provide this information.

[9]:
```
!lscpu
```

```
Architecture:             x86_64
  CPU op-mode(s):         32-bit, 64-bit
  Address sizes:          39 bits physical, 48 bits virtual
  Byte Order:             Little Endian
CPU(s):                   32
  On-line CPU(s) list:    0-31
Vendor ID:                GenuineIntel
  Model name:             13th Gen Intel(R) Core(TM) i9-13900HX
    CPU family:           6
    Model:                183
    Thread(s) per core:   2
    Core(s) per socket:   24
    Socket(s):            1
    Stepping:             1
    CPU max MHz:          5400.0000
    CPU min MHz:          800.0000
    BogoMIPS:             4838.40
    Flags:                fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge m
                          ca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 s
                          s ht tm pbe syscall nx pdpe1gb rdtscp lm constant_tsc
                          art arch_perfmon pebs bts rep_good nopl xtopology nons
                          top_tsc cpuid aperfmperf tsc_known_freq pni pclmulqdq
                          dtes64 monitor ds_cpl vmx est tm2 ssse3 sdbg fma cx16
                          xtpr pdcm sse4_1 sse4_2 x2apic movbe popcnt tsc_deadli
                          ne_timer aes xsave avx f16c rdrand lahf_lm abm 3dnowpr
                          efetch cpuid_fault epb ssbd ibrs ibpb stibp ibrs_enhan
                          ced tpr_shadow flexpriority ept vpid ept_ad fsgsbase t
                          sc_adjust bmi1 avx2 smep bmi2 erms invpcid rdseed adx
                          smap clflushopt clwb intel_pt sha_ni xsaveopt xsavec x
                          getbv1 xsaves split_lock_detect user_shstk avx_vnni dt
                          herm ida arat pln pts hwp hwp_notify hwp_act_window hw
                          p_epp hwp_pkg_req hfi vnmi umip pku ospke waitpkg gfni
                           vaes vpclmulqdq rdpid movdiri movdir64b fsrm md_clear
                           serialize arch_lbr ibt flush_l1d arch_capabilities
Virtualization features:
  Virtualization:         VT-x
Caches (sum of all):
  L1d:                    896 KiB (24 instances)
  L1i:                    1.3 MiB (24 instances)
  L2:                     32 MiB (12 instances)
  L3:                     36 MiB (1 instance)
NUMA:
  NUMA node(s):           1
```

```
  NUMA node0 CPU(s):      0-31
Vulnerabilities:
  Gather data sampling:   Not affected
  Itlb multihit:          Not affected
  L1tf:                   Not affected
  Mds:                    Not affected
  Meltdown:               Not affected
  Mmio stale data:        Not affected
  Reg file data sampling: Mitigation; Clear Register File
  Retbleed:               Not affected
  Spec rstack overflow:   Not affected
  Spec store bypass:      Mitigation; Speculative Store Bypass disabled via prct
                          l
  Spectre v1:             Mitigation; usercopy/swapgs barriers and __user pointe
                          r sanitization
  Spectre v2:             Mitigation; Enhanced / Automatic IBRS; IBPB conditiona
                          l; RSB filling; PBRSB-eIBRS SW sequence; BHI BHI_DIS_S
  Srbds:                  Not affected
  Tsx async abort:        Not affected
```

Some pointer arithmetic examples in C:

```c
#include <stdio.h>

int log_line(int line) {
  printf("Output for line: %d\n", line);
  return 0;
}

int main() {
  int * iptr = (int *) 0x12341230;
  int * iptr1 = iptr + 1;
  log_line(__LINE__);
  printf("iptr %p\n", iptr1);

  char * cptr = (char *) 0x12341230;
  char * cptr1 = cptr + 1;
  printf("cptr %p\n", cptr1);

  char * vptr = (int *) 0x12341230;
  void * vptr1 = vptr + 1;
  printf("vptr %p\n", vptr1);
  return 0;
}
```