Exercises for Architectures of Supercomputers

6th Exercise, 11./12.12.2019

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
TECHNISCHE FAKULTÄT

Exercise 1: Cache Simulator

The goal of this exercise is to write a cache simulator and use it to investigate the behavior of the L1 cache of modern Intel architectures

Cache parameters of the CPUs used in *emmy* nodes:

- L1 cache: 32 kB, 64 sets, 8-way associative,
  64-B cache lines (blocks), replacement strategy: LRU
- L2 cache: 256 kB, 512 sets, 8-way associative,
  64-B cache lines (blocks), replacement strategy: LRU
- L3 cache: 10x2.5 MB, 10x2048 sets, 20-way associative,
  64-B cache lines (blocks), replacement strategy: pseudo-LRU

- Verify this information with the help of `likwid-topology`.[1]
  Replacement strategies are documented in the „Intel 64 and IA-32 Architectures Optimization Reference Manual.“

[1] Make sure to first load likwid from the module system using `module load`

Exercise 1: Cache Simulator

- Your cache simulator should receive the following parameters over the command line:
  - Cache size [in bytes]
  - Number of sets
  - Associativity
  - Cache line (block) size [in bytes]
  - Size of an array that is sequentially traversed in the simulator

- The cache should implement an **LRU** (Least Recently Used) replacement strategy
  - LRU will select that cache line for replacement which that hasn't been used the longest (or the one least recently used)

- Internal working of caches (see blackboard)

Exercise 1: Cache Simulator (Hints)

Strategy in `main()`

- Sanity check of parameters
- Initialize data structures
- Access the array sequentially twice
  - During the first traversal, the cache will be filled with data
  - The second traversal serves the purpose of determining the cache hit rate
    - In the second traversal, each cache line (CL) should only be accessed once: use a 64-byte stride (gap between accesses)

```
// measure cache hits during second array traversal
for (address=0; address<array_size; address += 64) {
    hit = cache_access(address)
    cache_accesses++;
    if (hit)
        cache_hits++;
}
```

Exercise 1: Cache Simulator (Hints)

Suggestions for helper functions `cache.c`

- `cache_init` Called once from the `main` funktion. Allocates memory for housekeeping data and initializes them (e.g., setting the invalid bit for all entries)
- `cache_access` Encapsulates accesses to the caches
  - Determine if CL is in the cache (cache hit)
    - Extract cache set *s* and tag t from address *a*
    - Compare the tags of all blocks in set *s* with the tag *t* of the address to be accessed
  - In case of cache hit
    - Update housekeeping data (e.g., LRU information)
  - In case of cache miss
    - Select CL to be replaced in set *s* according to housekeeping data (LRU information)
    - (Load cache line into cache block)
    - Adjust tag *t* and LRU data

Exercise 1: Cache Simulator (Hints)

Helper function useful for debugging:

```
void convert_address_to_bits(uint64_t address)
{
  int i;
  printf("converting address %llu to bits: |", address);
  for (i=0; i<64; ++i) {
    if (((63-i)==cl_bits-1) || ((63-i)==cl_bits+set_bits-1))
      printf("|");
    printf("%d", (int)(address >> (63 - i)) & 1);
  }
  printf("|\n");
}
```

Exercise 1: Cache Simulator

- Hint: You can implement your cache simulator in any programming language you want

- Hint: When using time stamps to implement LRU, make sure the resolution of your clock is sufficient (system time in ms is not sufficient for a cores that clock in the GHz area)

- With the help of our simulator, determine the cache hit rates for arrays of different sizes
  - Range: 20kB – 40kB
  - At least 21 linearly distributed samples

- Plot the cache hit rate in a graph
  - *x*-axis (linear): array size [kB]
  - *y*-axis (linear): cache hit rate [%]