Cache Simulation

Stefano Lupo - 14334933 - SS Computer Engineering - 16/12/17

The repository contains the code for simulating the operations of an instruction cache and data cache. A trace of over 2 million physical addresses collected from an Intel i486 microprocessor while gcc was compiling a program is used as the list of addresses requested from the cache by the CPU.

Setup

The setup contaions two caches - one for instructions and one for data. The instruction cache is a 16KB direct mapped cache *(L=16, K=1, N=1024)* and the data cache is a an 8-way 32KB cache *(L=16, K=8, N=256)*. Both caches have L=16 bytes meaning four separate 32bit words can be stored in each cache line.

The instruction cache is direct mapped *(K=1)* meaning it has only one tag in each set. Thus the `tag` extracted from each physical address is compared with exactly one tag in the cache.

The data cache is a set associative cache containing *N=256* sets with *K=8* directories in each. The set number is first extracted from the physical address and used to select the relevant set. Each of the k tags in that set are then compared in parallel against the tag extracted from the physical address. If a match is found, the data can then be returned.

The traces contain a `burstCount` field which represents the number of adjacent (same set and directory) addresses the CPU would like to retrieve from the cache. As discussed, there are four 32 bit words in each cache line *(L=16)* and all four of these words may be retrieved simultaneously from the cache in one operation. When reading data from memory address *A*, the 3 memory addresses adjacent to **A** (modulo 16 bytes alligned on a 16 byte boundary) may also be read simultaneously into the other three words in the same cache line. This takes advantage of *locality of reference* - addresses nearby to an address that has just been read are likely to be required in the future. Thus caching these ahead of time provides a significant performance improvement.

When the cache inevitably fills up, a protocol must be put in place in order to replace entries in the cache. The protocol used was **Least Recently Used** meaning the tags that had not been accessed in the longest amount of time were evicted first.

Implementation

The implementation consists of two main classes - an `Analyser` class and a `Cache` class. The analyser class is responsible for examining the meta data associated with each trace and routing the request to the appropriate cache (eg if the request was an instruction fetch, it goes to the instruction cache etc). There is also a third `Runner` class which is just responsible for reading in / displaying (in dec / binary / hex) some of the data to help understand the make up of each of the traces.

Analyser Class

This class is instantiated by providing it with one or two Cache objects to be used as the caches for the simulation and an array of traces. Once instantiated an `analyseAssignment()` method may be called which does the following:

- Iterates over each trace (skipping every second 32bit word which is not required)
- Extracting the `cycleType` in order to determine if this trace should be routed through the caches
  - This is done by *ANDING* the 32bit value with the `CYCLE_MASK` and (unsigned) shifting it right by `CYCLE_SHIFT` places.
  - This extracts the three most significant bits which encode the cycle type.

  ```
  int cycleType = (word & 0xE0000000) >>> 29;
  ```

- Extracting the `burstCount` in order to inform the cache of how many adjacent values the CPU would like to retrieve.
  - This is done by *ANDING* the 32bit word with the `BURST_MASK` and shifting it right by `BURST_SHIFT`
  - This extracts bits 27 and 28 which encode the `burstCount` .

  ```
  int burstCount = (word & 0x18000000) >>> 27;
  ```

- Extracting the physical `address` in order to pass it to the `cache` objects.
  - This is done by first *ANDING* the 32bit word with the `ADDRESS_MASK` and then shifting it left twice in order to fill the missing two address bits (which are ommited from the trace file)
  - This extracts bits 24 to 2 from the trace and pads it to the required 25bit physical address.

  ```
  int address = word & 0x007FFFFF;
  address <<= 2;
  ```

- Ensures these values are valid and passes the `address` and `burstCount` to the appropriate cache depending on `cycleType` .

Cache Class

This class is responsible for implementing the actual behaviour of a cache. It can be instantiated with values for `l` , `k` and `n` , and an optional `addressSize` (which defaults to 25). This does the following:

- Computes `setBits` and `offsetBits` using log2 - the number of bits required for encoding the set (indexes N) and offset (indexes L).
- Computes the masks and shifts for the set, offset and tag, for the given `l` , `k` , `k` and `addressSize` configuration by the following:

```
int offsetMask = (1 << offsetBits) - 1;
int offsetShift = 0;
int setMask = (1 << setBits) - 1;
int setShift = offsetBits;
int tagMask = (1 << (addressSize - offsetBits - setBits)) -1;
int tagShift = offsetBits + setBits;
```

The Cache class contains a Hash Map (to represent a *set*) which maps from a `setNumber` --> `HashMap` . This second Hash Map then maps from `tag` --> `TagData` object. This `TagData` object contains a `timestamp` (which is just an integer) in order to implement LRU and could contain further objects for representing the *actual data* in the cache (not needed for our purposes).

This allows us to lookup our cache first by set number, returning a hash map which represents a `set` . This `set` hash map can then be looked up using the `tag` integer contained in the physical address, allowing us to simulate extracting data from the cache.

The Cache class also has the `feedAddress(int physicalAddress, int burstCount)` method which does the following:

- Extracts the `setNumber` , `offset` and `tagNumber` from the phsyical address using the above masks and shifts.
- Extracts the required `set` (by `setNumber`) from the `sets` Hash Map and extracts the `tagData` from that set (by `tagNumber` and checks:
    - if `tagData` is not null, this is a cache hit: `hits += (burstCount + 1)`
    - else it was a miss: `hits += burstCount` , `misses++`
        - if the k directories are not all filled - insert the new tag in the empty directory
        - else remove LRU tag (the tag with the lowest `timestamp` integer) in that set and replace it with the new tag.

Simulation Results

The simulation produced the following results.

```
Data Reads: 41.870308%
Data Writes: 14.540052%
Instruction Reads: 40.848686%
Skips: 2.7409554%
Cycle Types (encoded): [8836, 38901, 100, 9645, 856659, 0, 878084, 304927]
Analysed 2039670 traces, skipped 57482

Instruction Cache
Total accesses: 3426555
Misses: 128052
Hits: 3298503
Hit Rate: 96.262955%

Data Cache
Total accesses: 1188855
Misses: 31778
Hits: 1157077
Hit Rate: 97.327%

File Reading Time: 70ms
Cache Simulation Time: 214ms
Total Elapsed Time: 284ms
```