Kairi Kozuma

GTID: 903050898

ECE 3056

04/22/2016

Assignment 5 Design and Verification Report

**I. Design Summary**

**A. Data Structures**

typedef struct cache\_line\_t{

boolean valid;

boolean dirty;

cache\_word\_t tag;

cache\_word\_t \*word;

struct cache\_line\_t \*next\_lru;

} cache\_line\_t;

// Cache set

typedef struct cache\_set\_t{

int stack\_count;

cache\_line\_t \*cache\_line\_array;

cache\_line\_t \*stack\_mru; // Most recently used line

cache\_line\_t \*stack\_lru; // Least recently used line

} cache\_set\_t;

// Cache

typedef struct cache\_t{

cache\_set\_t \*cache\_set\_array;

int ways;

int sets;

int cache\_size;

int block\_size;

int words\_per\_block;

int set\_bits;

int byte\_offset\_bits;

int tag\_bits;

} cache\_t ;

**Figure 1.** Data structures used to implement the caches.

Three structs were used to represent different cache components. The cache\_line\_t struct represents a single cache line, with a dirty byte, a valid byte, a tag word, and a pointer to the beginning word of the block. The cache\_set\_t struct contains the count of the number of valid lines in the set, a pointer to the beginning of the cache\_line\_t array for the set, and pointers to maintain the LRU stack. The cache struct maintained all configuration information, as well as a pointer to the beginning of the dynamically allocated cach\_set\_array.

**B. Address Parsing Code**

cache\_g->sets = cachesize / (blocksize \* ways);

cache\_g->words\_per\_block = cache\_g->block\_size / (cache\_word\_t) sizeof(cache\_word\_t);

cache\_g->set\_bits = (int) log2(cache\_g->sets);

cache\_g->byte\_offset\_bits = (int) 2 + log2(cache\_g->words\_per\_block);

cache\_g->tag\_bits = (int) 32 - cache\_g->byte\_offset\_bits - cache\_g-> set\_bits;

...

// Extract tag bits

cache\_word\_t tag = (cache\_word\_t) physical\_addr >> (32 - cache\_g-> tag\_bits);

// Extract set index

cache\_word\_t set\_index = (cache\_word\_t) (physical\_addr & (0xFFFFFFFF >> cache\_g->tag\_bits)) >> cache\_g->byte\_offset\_bits;

**Figure 2.** Code to extract tag bits and set index from physical address.

Extracting the tag bits requires shifting the physical address by (32 – number of tag bits). The number of tag bits is obtained previously in the cache initialization, by subtracting from 32 the number of byte offset bits and the number of bits for the set index. The set index is obtained by using a mask to obtain the lower bits of the physical address not contained in the tag, then shifting the result by the number of bytes per line.

**C. Cache LRU Stack Implementation**

// If no lines in set, add line at start of array

if (set\_p->stack\_count == 0) {

line\_p = set\_p->cache\_line\_array;

set\_p->stack\_mru = line\_p;

set\_p->stack\_lru = line\_p;

set\_p->stack\_count++;

...

// Else, search linked list for tag

} else {

cache\_line\_t \*line\_temp\_p = set\_p->stack\_mru;

while (line\_temp\_p->next\_lru != 0 && line\_temp\_p->next\_lru->tag != tag) {

line\_temp\_p = line\_temp\_p->next\_lru;

}

...

}

**Figure 3.** Implementation of LRU stack using pointers to cache lines.

The LRU stack is implemented as a linked list, with each valid cache line in the set pointing to the next least recently used line. The MRU (Most Recently Used) and LRU (Least Recently Used) pointers are used to point to the head and tail of the list, respectively. The linked list must be traversed to determine if a cache line with a given tag is present. Removing and inserting to the head requires updating the MRU and LRU pointers, as well as the pointers in between the removal point.

**II. Results and Analysis**

**A. Optimal Line Size Results**

**Table 1**

Analysis of Cache Performance for Optimal Line Size

|  |  |  |  |
| --- | --- | --- | --- |
| Trace | trace.bubble | trace.merge | Overall |
| Cache Size | 131072 | 131072 | 131072 |
| Associativity | 64 | 64 | 64 |
| Block Size | 512 | 512 | 512 |
| Accesses | 6,322,343 | 7,678,430 | 14,000,773 |
| Hits | 6,318,756 | 7,673,810 | 13,992,566 |
| Misses | 3,587 | 4,620 | 8,207 |
| ReadMiss | 3,082 | 3,876 | 6,958 |
| ReadCount | 5,666,010 | 6,544,752 | 12,210,762 |
| WriteMiss | 505 | 744 | 1,249 |
| WriteCount | 656,333 | 1,133,678 | 1,790,011 |
| Writebacks | 995 | 1,466 | 2,461 |
| MissRate | 0.05674% | 0.06017% | 0.05862% |
| ReadMissRate | 0.05439% | 0.05922% | 0.05698% |
| WriteMissRate | 0.07694% | 0.06563% | 0.06978% |
| WriteBackTrafficBytes | 2,037,760 | 3,002,368 | 5,040,128 |
| TotalMemoryFetch | 7,346,176 | 9,461,760 | 16,807,936 |
| TotalMemoryReference | 25,289,372 | 30,713,720 | 56,003,092 |
| VolumeSaved | 17,943,196 | 21,251,960 | 39,195,156 |

**B. Plot of Varying Line Size with Associativity of 4**

**Figure 1.** Miss rate vs. line size chart for both trace.merge and trace.bubble, with the following parameters: associativity of 4, cache size of 128Kbytes, and line size ranging from 32 bytes to 2Kbytes.

Based on the graph, the miss rate was minimized for both traces with a line size of 512 bytes.