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

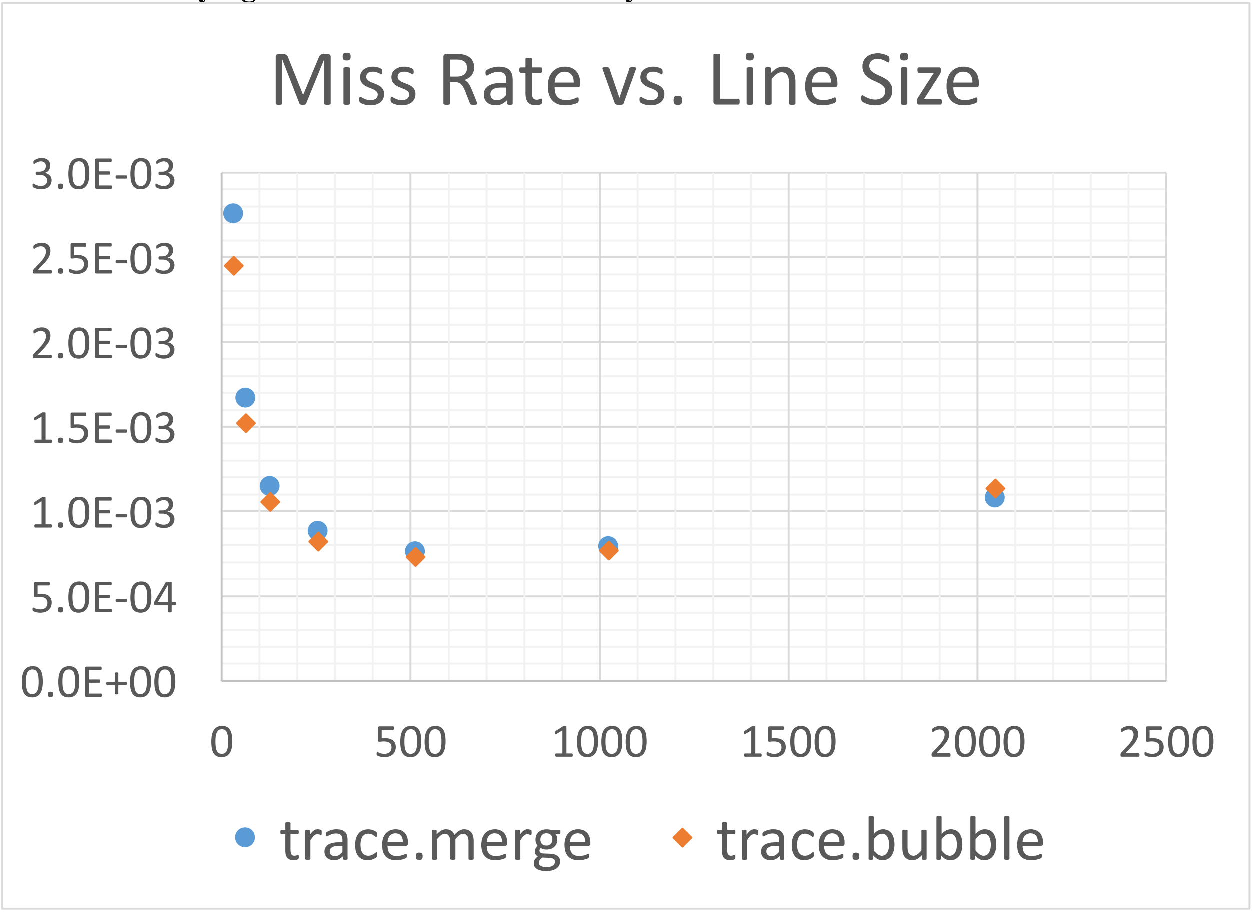

Figure 1. Miss rate vs. line size chart for both trace.merge and trace.bubble, with the following parameters: associativity of 4, cache size of 128Kbytes, and line size ranging from 32 bytes to 2Kbytes.

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