ECE 585
Elliot Lamb
Haydeny Galante
Jacob Louie
Alexander Maso

Team 6: Final Project Report

Set out to simulate a 16 MB last level cache that is capable of working in a system with three other processors. Simulated cache works in a shared memory configuration and employs a write allocate policy. The system maintains inclusivity and implements the MESI protocol as well as a Pseudo Least Recently Used replacement algorithm to ensure coherence. The cache is 8-way set associative, has a total capacity of 16 MB and uses 64 byte lines. The higher level cache in our sim also has 64 byte lines but is 4-way associative and employs a write once policy instead. The simulation receives input from a text file that specifies the command and address to be processed. The program records and returns the number of reads, writes, hits, and misses in the last level cache.

Assumptions

- Variables in function definitions are of type unsigned int, assume 32 bit machine.
- The cache does not need to be clock accurate.
- Assuming instructions provided in sequential order, no need for arbitration.
- Assuming normal behavior from other processors/caches interacting with LLC

Design Choices

- We chose to simulate our cache in C and support a debug mode (-d): verbose output
- Data Structure: We modeled our cache using an array of structs which consisted of a struct for each way containing unsigned int for tag and uint8_t for the state. We chose a uint8_t data type to limit the amount of memory consumed by the program during the execution of large test cases.
- The cache tag array is an array of sets indexed directly from the address index bits. Each set is a structure of n ways and a PLRU value. Each way consists of a tag value and a MESI status value. For this cache design, the 32-bit address breaks down into 32K sets (15 index bits) and an 11-bit tag, and there are 8 ways per set.
- MESI flags and PLRU values are stored as C 8-bit types (char or uint8_t) while the tag values are stored as ints (32 bits), but could be 16-bit values if the goal is to minimize space. Since most modern C compilers pad to int anyway, there's little point in economizing that.
- Macros are used to mask bits and separate Byte Select, Index and Tag
- calloc() is used to initialize all sets to 0. We used a separate lookup function to search for a given tag and set number. Additionally we created an eviction handler called DoEviction(). We used two functions for a write request from L1 data cache, to clear the cache and to print the contents and a single function to handle both read requests from L1 data cache and the L1 instruction cache.
- LLCs snoops and simulated report of snoop results from other caches handled by SnoopOp().

ECE 585
Elliot Lamb
Haydeny Galante
Jacob Louie
Alexander Maso

- The PLRU function is modeled as an array with indices numbered 0 through 7. The PLRU is determined by iterating through the array by two indexes per increment and adding one to the value depending on the position in the array.

Team Approach

The team met in person several times to plan and discuss the implementation. One Github repository was used to organize the files. Members were given individual coding tasks but every team member generated tests and independently verified the functionality on their own. We found and corrected minor host OS difficulties by this approach.

Deliverables

Included in the submission are the source files:

- defs.h - data structures, function declarations and macros
- llcmain.c - file parsing routines, main loop, bus operation reporting
- cacheops.c - L1 cache interactions, PLRU implementation, bus snooping function

A Makefile is provided to build the executable (“llc” or “llc.exe” depending on the platform).

A program called GenTest was written to aid in creating test cases. A folder called “tests” contains trace files used to test the cache implementation.

Testing

- See Test Plan