Chris Fretz
Santosh Nagarakatte
Computer Architecture
12/14/13

Readme

My cache-sim binary is composed of 6 different linked C files: sim.h, sim.c, Cache.h, Cache.c, SmrtArr.h, and SmrtArr.c. Aptly named, the cache files house all structs and functions necessary to represent, and implement, a generic cache, the SmrtArr files house the structs and functions necessary to implement a dynamically allocated array which keeps track of its current maximum size and the number of elements housed within it, and the sim files include all functions necessary to actually carry out the simulation. I'll now go into each of the files in depth.

Cache:

The header, and subsequent implementation file, for this contain 3 different structs, named Cache, Set, and Line. Keeping with the straightforward naming scheme, these respectively represent an entire cache, a set within a cache, and a line within a set. The files also contain 6 functions for the creation, and destruction, of Cache, Set, and Line, structs. A Cache struct keeps track of its type (L1, L2, L3, or a fourth type used to represent the fully associative cache used in the secondary simulation used to calculate the miss types), its total size in bytes, its association level, its block size, the number of sets contained within, and the actual sets. A Set struct keeps track of the number of lines it contains, and the actual lines. A Line struct keeps track of how many insertions have taken place since it was inserted (used for FIFO), how many accesses have taken place since it was last accessed (used for LRU), a valid bit, and a tag.

SmrtArr:

The header, and subsequent implementation file, for this contain one struct named SmrtArr, and 3 functions used for the creation, insertion into, and destruction of, a SmrtArr struct. The SmrtArr struct itself keeps track of its current maximum size, the amount of items that it actually currently holds, and its contents (an unsigned long long int array. I regret to say that I didn't make the SmrtArr struct generic, but I didn't feel like dealing with void *'s).

Chris Fretz
Santosh Nagarakatte
Computer Architecture
12/14/13

sim:

The sim files consist purely of functions, and handle all of the real work of the project. The flow of the file is as follows:

- main takes in command line arguments, checks if a valid number of them have been passed in, and then calls validateParameters on the arguments.
- Although probably entirely unnecessary, validateParameters can accept the parameters in arbitrary order, and is responsible for all verification as far as the validity of the parameters is concerned. It reads all numerical information into a global array called intArgs, and uses two global strings to store the rep lament algorithm and the file name. After validation, validateParameters returns to main.
- main attempts to open the specified file, and upon success passes the file to the getLines function.
- Instead of using fscanf to read the lines from the files in as strings, and then being forced to convert them to integers, getLines reads each individual character in from the file, and, assuming it's a legitimate hex digit, converts it directly into its corresponding integer value, and adds it to a tally for the current line. Upon encountering a newline character, it assumes that the current line is completed, and inserts the completed integer into a SmrtArr struct. This continues until the function reaches an EOF character, upon which it returns the resultant SmrtArr to main.
- Using the user supplied parameters, main now creates 3 cache structs, and passes the lines SmrtArr to the insertionLoop function.
- The insertionLoop function loops over the contents of the SmrtArr and calls first the bitHash function (passing it the current address), and then checkAndUpdateCache.
- The bitHash function takes care of the grunt work of the bit shifting necessary to find where to insert the current address into the various caches. It returns its results inside of an unsigned long long int array (had to be, as the tags can be quite large).
- The checkAndUpdateCache function takes care of the majority of the work for the project. It first checks to see if the requested address exists within the specified cache, and, if it does not, it either inserts at the first available location, or chooses which existing data to evict based on the chosen algorithm. Under either circumstance it calls the updateLines function to update the insertion and usage counters on all of the lines contained in the specified set. It also takes care of the incrementation of the global miss counters for each set, and the cold miss counters for each set. This function returns true if the requested address exists in the cache, and returns false if it does not.
- Back in the insertionLoop function, if checkAndUpdateCache returns true, it increments the global cache hit counter for whatever cache the address

was found in, or calls checkAndUpdateCache on the next cache if it returned false. Upon reaching the end of the SmrtArr, it returns to main.

- In the case that the L1 cache was a set associative cache, main now creates a new, fully associative version of the L1 cache, and runs a secondary simulation on it for the purpose of calculating the conflict and capacity misses on the original cache. This secondary simulation uses the functions secondaryInsertionLoop and secondaryBitHash because the implementation details are slightly changed from the original simulation, and I couldn't think of any elegant way to unify the simulations.
- Finally, main prints the results of the simulation.