Nathan Becker

Edward Sayers

[Email address]

ECE 485 Final Project   
 Fall 2014

CPU Cache Simulator

Starting from the conception of the architecture for this project, we stringently followed several design patterns intended to aid in its implementation and debugging. First, we decided to follow object-oriented principles to encapsulate and isolate functionality, enabling better debugging and maximizing code re-use. Every class has a specific task, and can be tested individually. By following this design pattern, it also makes the code compact, as there are no duplicated functions or repeated code. Because of the encapsulation, along with heavily utilizing enumerations, and comments, the code is easy to read and understand. Debugging and testing the code took minimal time, as the points of failure were minimal.

When implementing the project, we went in with several assumptions. One primary assumption we made was that the cache was snarfing data from other CPUs and their write operations to the bus. When another core writes data on the bus, our simulator checks its cache, and if there is a match that is marked invalid, our core marks it as shared. Another assumption we made was that our cache mixes the instructions and data, unlike the separated data and instruction caches of the L1 cache. Finally, we assumed that the L1 cache notifies the L2 cache and our cache simultaneously.

As far as the interfaces between the caches, we kept it simple, implementing the procedures designated in the project assignment description, and printing the results as long as the SILENT switch isn’t turned on. Whenever a bus operation is necessary, the appropriate function is called and the bus operation is printed to the console. When getting snoop operations, the program divides the byte select by 3 and uses the modulus to determine what result to return – if 0, then NOHIT, if 1, then HIT, if 2, then HITM. When pushing snoop operations, the address and the snoop operation is printed to the console. As far as the message to the L2 cache, we post a READ message when we return the address when the L1 requests a READ, and whenever a line is modified or invalidated with snooping, we post a WRITE/INVALIDATE/RWIM message with the address, so the lower level cache can update its cache accordingly.

Our program is structured in a nested hierarchy of objects. Starting at the top, the main program is responsible for reading in a file that contains a list of test cases to run. Each test case is run, line by line, and the proper trace operation is executed. If it relates to printing summary information or clearing the cache, it is handled by the main program function; otherwise, it is passed to the cache controller object.

The cache controller object is responsible for executing cache commands and for pushing/getting snoop results and bus operations. When a trace op is passed to the controller, it executes the proper command on the embedded cache object.

Moving to the cache object, its task is to manage all aspects relating to retrieving and placing lines in the cache, along with recording the statistics of the cache – the hits/misses/reads/writes. Any cache operations are directed to the appropriate function, and the associated statistics are incremented.

Next down the list is the cache set. The cache object maintains an array of cache set pointers, and when a cache line is placed or retrieved, it finds the cache index in the memory address and looks at that index in the cache set array. If the cache set pointer is null, it creates a new cache set object at the associated index. Inside the cache set object are functions responsible for placing and retrieving a line, along with determining whether a line needs to be evicted from the cache. Of particular interest in the cache set object is the implementation of the pseudo Lease Recently Used (LRU) algorithm. Our implementation flattens the tree into an array of bits, alternating between a bit on the lowest level of the tree, and a bit on a higher level of the tree. This lets us utilize a recursive function designed to find the midpoint of the LRU array, decide which half of the array to search, find the midpoint of that sub-array, and repeat the process until the sub-array size is one. At that point, the line of interest is the index of the array if the bit is 0, or the index+1 if the bit is 1. There are two functions relating to the LRU array, one restricted to finding a particular line index, and the other one designed to figure out which line to evict. This method is highly efficient and can find the line with minimal operations. Better yet, it is incredibly easy to test, as we only need to test the function with an associativity of four, and if it passes all permutations, it will scale to far greater associativity without needing additional testing.

Finally, there is the cache line. This is a simple object, and only contains the tag and the state. The cache set object contains an array of pointers of cache line objects determined by the associativity. They are null until initialized. When a line is placed, it first looks to see if there are any null pointers, or if there are any invalid lines, and if so, it places the line in the first available slot. If the line is full, it evicts a line via the pseudo-LRU and places the line accordingly, notifying the cache set object to write the evicted line to memory.

Along with the nested hierarchy of objects, there is a small static class designed to find the byte select, index, and tag of the address. These functions are simple, and utilize bit-shifting operators to maximize performance.

According to our test cases, detailed in the next section of our report, and running the cc1.din file, the simulator performs as designed with excellent performance and reliability. The following pages detail our test plan and source code.