Rohit Mahankali

Dr. Menendez

Computer Architecture

1 May 2018

PA5 Report

To begin with, I used a 2D array of "cacheLine" structs to represent a cache for this simulator. Based on the associativity arguments (direct, assoc, assoc:n) provided on the command line, I was able to calculate the number of sets, or number of rows for the 2D array, and also the number of lines per set, or columns for the 2D array. The inspiration for using the 2D array as the representative data structure was the slide deck provided to us on caches. Therein, a cache was defined to be an array of sets, each of which is composed of cache lines, each of which may contain a cache block. Based on this definition, I knew I was aiming for an array of sets, which is equivalent to a 2D array of cache lines, being as a set is an array of cache lines. As for the necessary algorithms to simulate memory accesses, I began by considering the scope of the simulation. I knew that the output was simply the number of cache hits, cache misses, memory reads, and memory writes for a non-prefetching cache and a prefetching cache. To produce these statistics via a simulation, I reasoned that the valid bit and cache block components of a cache line were extraneous. This is because the valid bit becomes relevant when communicating updates between multiple caches, and no actual data was being retrieved from memory, so simply knowing the block offsets for bit calculations would be sufficient. As a result, only the tag bits for a given cache line were pertinent, as these were the bits that would determine whether or not a given address was found to be in the cache or not. So really, I could have simulated this cache with a 2D array of ints, where each int would represent the base 10

expression of the tag bits, but I wrapped these tag bits in a cache line struct to make the whole process easier to abstract and understand. So ultimately, I just needed to calculate the set bits and tag bits for any address in the trace file (this was done via bit shifting and bit masking), and then I could use these values as search parameters to determine whether a given address was in the cache itself. This was done via a for loop of all the cache lines in a desired set, checking to see if any of the cache line tags matched with the desired tag value. Finally, given this data structure and algorithm, I simply my knowledge of caches to determine when to increment each output statistic for the non-prefetching and prefetching cache.

Answers to Questions

1. In general, the prefetcher achieved its purpose of utilizing spatial locality, as the majority of the test cases had more cache hits and fewer cache misses for the prefetching cache than the non-prefetching cache. The better performance of the prefetching was more pronounced in scenarios, where the cache had fewer sets, i.e. where the cache was closer to fully associative. This effect was consistent across almost all of testing except for a few cases. In these cases, the trace file had relatively fewer addresses, and the cache had many sets, i.e. close to or exactly direct-mapped caches. Due to the frequent evictions of blocks in these caches, prefetching couldn't generate a significantly greater number of cache hits, as loading an adjacent cache block would force the previous one out; in fact, some of these cases showed that the prefetching cache produced fewer cache hits than the non-prefetching one.
2. Yes it would be possible to make this adaptation. To do this, I would have to create an additional layer of logic in my main function. In other words, after checking to see if a given address is in the cache (presumably the L1 cache in this case), the program would

have to subsequently check if that address is in the L2 cache as well. If the address is in either one of these caches, the “cache_hits” statistic would need to be updated. By contrast, this statistic could be broked down further into an “L1_cache_hits” and “L2_cache_hits,” so the reported statistics would be more specific as to which cache contained the desired address/block. Only after checking these two caches, would the program read data from main memory and increment the “cache_misses” statistic. For memory writes, the program would function vary similarly, but with the addition of this extra logic layer of checking to see whether a given address/block is contained in either the L1 or L2 cache.

3. To make this modification, I would simply remove the statements that increment the “memory_writes” statistics in my code after making changes to blocks in the cache. This would then stop resembling a write-through cache, as was the goal in this an assignment, and start resembling something like a write-back cache. In this case, data at certain addresses within memory are only updated at certain intervals, even if the corresponding addresses have updated data in the cache. The justification for this particular change could be to amp up system performance at the expense of data recoverability.