CPRE 308 Lab Report
Final Project Part 2 - Cache

Written By:
Quinn Murphy
Ben Williams

Section D
Friday at 11:00 AM

Introduction

A well-designed cache can mean all the difference in the world to a computer operating on a small amount of physical memory. Regardless of how fast a processor is, if all of your time is spent sending data back and forth between registers, physical memory, and non-volatile storage, most of your clock cycles will be spent idling. This week's lab was focused on writing an efficient caching system that minimized the number of missed memory requests so that more time could be spent actually doing calculations.

Design

isu_mmu_page_fetch

We kept this function the pretty much the same. All we did was fill in cases 1 and 2 for the LRU and Clock algorithms as detailed below. We referenced the FIFO example quite a bit to see how pages were moved down and up through caches.

Least Recently Used

We implemented a basic O(n) find min search through all of L1 memory, sorting by access time, and set the variable "replace_index" to the lowest value's index. We then called isu_page_move() specifying this index of the L1 memory (moving it to a lower level of cache.) Finally, we placed our new page in the place of the moved page at replace_index.

Clock

We implemented our clock algorithm around the hand member of ISU_MMU_STRUCT. This hand points towards the page we are currently considering replacing and is preserved in between calls to isu_mmu_page_rep_clock(). Our algorithm begins by checking to see if the page we are requesting is already in cache. If so, it updates the page's access time and sets the reference bit to 1. If not, we need to find a place to store this new page. We begin by checking where the hand currently is (if it is at the end of the list, we reset to the beginning), and if the current location of the hand is at a page where the reference bit is 0, we replace that page and stop searching. If the hand is pointing where a reference bit is 1, we set it to 0 and continue around the clock by incrementing the hand until doing the same thing until we find a 0.

Results

Least Recently Used

We saw that the sequential access gave us 0% (0/1000 accesses were hits.) This was encouraging to know that the base case was correct. We found that the random access hit rate was 23.8% (238/1000), which was 2 better than FIFO. For spatial access we got a 83.9% hit rate, which was a significant improvement over the 81.1% of FIFO. This makes sense because in spatial locality, it is more likely that recently used memory addresses will be checked in the near future. Less of these will be the least recently used and therefore will stay in memory for longer.

Clock
The “sanity check” memory test of sequential access yielded the proper result of 1000 misses (hit rate of 0.0%). For random memory access, our clock algorithm had 762 misses (23.8% hit rate). Compared to the FIFO hit rate of 23.6%, our algorithm performed just about the same, which is a good thing to expect for completely random access. For spatial access, our algorithm only had 165 misses (83.5% hit rate). Again comparing to the FIFO hit rate of 81.1%, we see quite the improvement (as well as over LRU).

Conclusion

From the given algorithms (FIFO, LRU, Clock), we found that clock was the Clock algorithm was the most efficient of those examined, though Least Recently Used wasn’t too bad either. What is clear, is that without a cache any computer that needs a lot of memory would never reach its full speed potential. Furthermore, we can speed up processes by exploiting spatial locality in our cache, so as operating system writers, we should be allocating memory units intelligently to increase the probability that this will happen.