CS246 – Homework #4

Matheus S. Farias

School of Engineering and Applied Sciences, Harvard University

Spring, 2023

Abstract

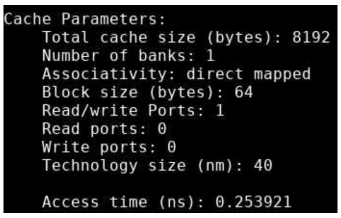
This document presents my solutions for homework 4 of CS246 taught in Spring 2023.

Contents

§1 Inclusion Property

(a) The inclusion property refers to the idea of sharing the same data across different levels of the memory hierarchy. For instance, let's consider we have an architecture with main memory, L2 cache, and L1 cache where the L2 cache is inclusive and the L1 cache is exclusive. According to the memory hierarchy, the main memory is the farthest one from the processor, and the L1 cache is the closest one. As the L2 cache is inclusive, we know that all the data stored in the main memory is indeed stored in the L2 cache. However, as the L1 cache is exclusive, we know that there is no data in the L1 that is present in the L2 cache.

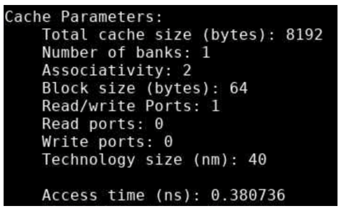
(b) When we assume inclusion property, we want the lower and the higher level memory to have the same line size. The larger the difference between the line sizes of lower and higher memory, the bigger the probability of breaking the inclusion property. For instance, if we have a lower memory with twice the line size of the



```
Cache Parameters:
    Total cache size (bytes): 8192
    Number of banks: 1
    Associativity: direct mapped
    Block size (bytes): 64
    Read/write Ports: 1
    Read ports: 0
    Write ports: 0
    Technology size (nm): 40

    Access time (ns): 0.253921
```

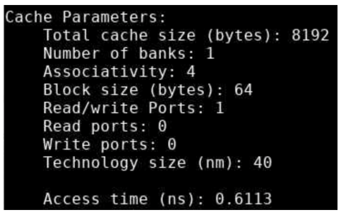
(a) Associativity 1



```
Cache Parameters:
    Total cache size (bytes): 8192
    Number of banks: 1
    Associativity: 2
    Block size (bytes): 64
    Read/write Ports: 1
    Read ports: 0
    Write ports: 0
    Technology size (nm): 40

    Access time (ns): 0.380736
```

(b) Associativity 2



```
Cache Parameters:
    Total cache size (bytes): 8192
    Number of banks: 1
    Associativity: 4
    Block size (bytes): 64
    Read/write Ports: 1
    Read ports: 0
    Write ports: 0
    Technology size (nm): 40

    Access time (ns): 0.6113
```

(c) Associativity 4

Figure 1. Values of access time obtained from `cacti`

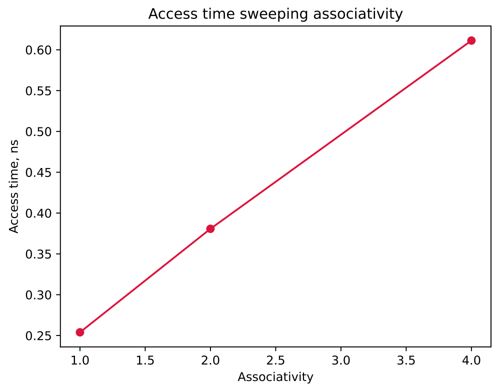


Figure 2. Access time sweeping associativity.

higher memory (64 vs 32 bits) and the lower has an offset of one whereas the higher has an offset of zero, we can map half the cache line of 64 bits into a line of 32 bits. Let's say we map line A of 64 bits into two lines in the 32 bits. One line receives A[0] and the second receives A[1]. When we evict a line, say A[1] from the 32 bits cache, we don't get rid of A[0], whereas we can only get rid of A[0] and A[1] from the 64 bits simultaneously. Thus, we still have the information of A[0] in the 32-bit cache, considering further replacement of A[1] with new data say B[1]. By definition, since the 32 bits cache has content of A and the 64 bits cache doesn't, we break the inclusion property.

§2 CACTI

The results can be shown in Figure 1. The requested plot is shown in Figure 2.

From the plot, I observed that as we increase the associativity with powers-of-two, the access time increases linearly, starting from 0.253921 ns to 0.6113 ns. This plot was expected according to Jouppi's paper. He claims that even though direct-mapped caches have more conflict misses due to their lack of associativity, their performance is still better than set-associative caches when the access time costs for hits are considered. Notice that the direct-mapped case is the only configuration where the critical path is merely the time required to access RAM. Naturally, as we increase the associativity, this critical path becomes winding due to additional circuitry to implement associativity and we see the behavior of Figure 2.

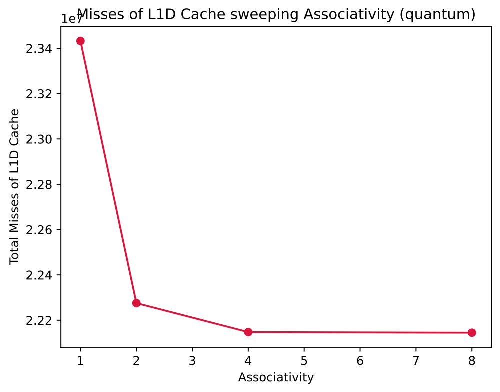
§3 Stream Buffers

(a) Stream buffer is a strategy to prefetch cache lines starting at a cache miss address. It reduces the ratio of cache misses, acting on capacity, compulsory, and some instruction cache conflict misses. Instead of placing the prefetched data in the cache, we now place it in stream buffers. The paper mentions that stream buffers are only effective for unit-stride reference streams that at most skip every other or every third word. This way, we can leverage the principle of spatial locality. In the case of a reference stream that skips to every third word, we will keep storing words in the buffer instead of in the cache, however, we skip whenever we prefetch a word whose index is a multiple of three. For instance, if we are dealing with data stored as a string and we have the words "advanced computer architecture is a great class". If we skip every third word, our buffer will store "advanced computer is a class"

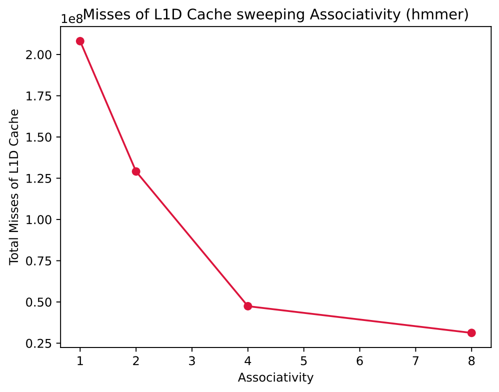
(b) I thought of two ways to solve this problem. A trivial way would be to increase the size of the buffer and then we don't rely on the prediction associated with the principle of spatial locality, but we increase the probability of having the next data just by having more data. A second way and more interesting is to design a method of predicting the next data based on access patterns of the reference stream. We can think of storing access patterns in a pattern table similar to the last homework and apply a machine learning algorithm to try to predict the next data.

§4 Victim Cache Implementation

(a) First, we navigate to `/sys/devices/system/cpu/cpu0/cache/index0`. By investigating files `type` and `level`, we make sure that the folder `index0` gives us information about the L1D cache. Then, the file `ways_of_associativity` shows that our L1D cache works with associativity 8. Now we know that among all the data we have, the fairest comparison between `likwid` and `pin` is when the associativity is 8. Figure 3 shows the plot of results from using only `pin` with `quantum` and `hammer` workloads and Figure 4 provides also `likwid`. As shown in Figure 4, `likwid` got results below `pin`. From what we discussed in slack, there are no counters for L1 Stores (hits or misses) in `likwid`, but this is not the only



(a) Quantum



(b) Hmmer

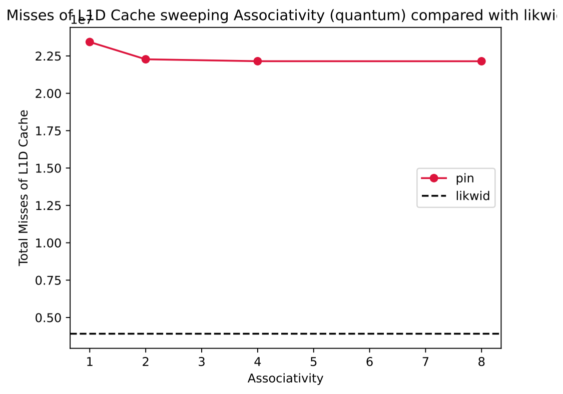
Figure 3. Total number of misses in the L1D cache using pin.

reason why `likwid` should have fewer misses. In the cluster, we have a much more complex architecture with three cache levels possibly with advanced prefetching methods such as stream buffers. Our cache simulator is very simple, and at this point, we do not implement any method on top of what the cache naturally does. In `quantum` workload, there is a huge gap between the performance of `pin` versus `likwid` whereas in `hmmer` this gap is tight. One reason is that `quantum` is a harder workload, and then prefetching methods stand out in performance benefits, whereas `hmmer` seems to be an easier workload and just spatial locality can be enough to achieve to have good accuracy.

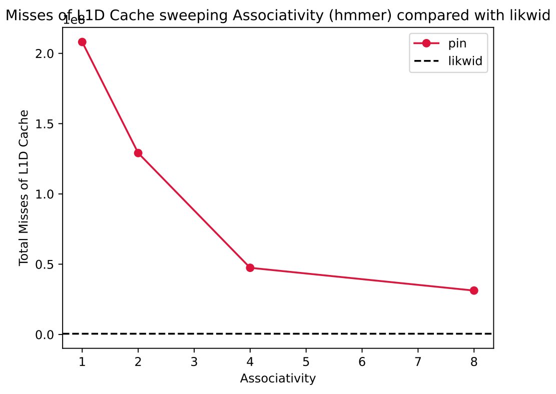
(b) The implementation is provided in `code_1_7/hw4.cpp`. The parameterization was done in line 591, the block size is `bsize` and the number of entries is the multiple of `bsize` that goes in `csize`.

(c) See the Figure 5. Both cases show that as we increase the number of entries, we have fewer misses, which is expected. In the workload `hmmer`, the slope was more aggressive and we had more benefits with the victim cache. On the other hand, `quantum` had a significant decrease for a small number of entries, but then it remains constant.

(d) For this problem, I restricted my analysis to compare against only an 8-entry victim cache for both workloads. With direct mapping, we have 22132802 and 28936293 misses for `quantum` and `hmmer` respectively. With associativity 8 we have 22134951 and 18698648 for `quantum` and `hmmer`. These values represent a negligible improvement for `quantum`, but an improvement of 35.4% for `hmmer`. That is, for some workloads, the benefit can be negligible, but for others we can have improvements.

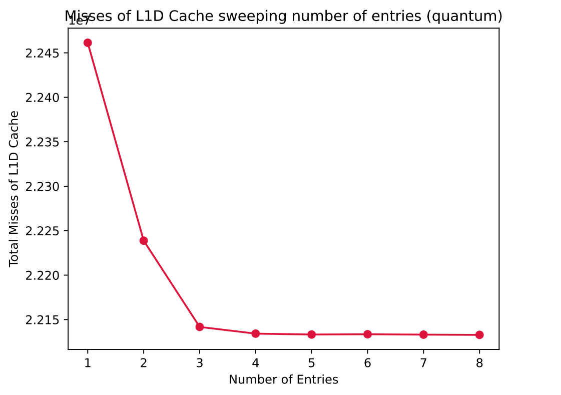


(a) Quantum

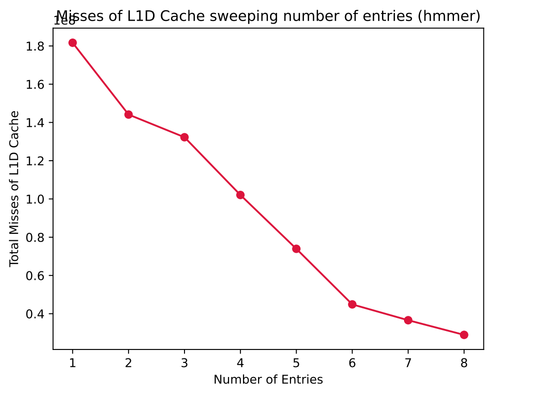


(b) Hmmer

Figure 4. Total number of misses in the L1D cache using pin versus likwid.



(a) Quantum



(b) Hmmer

Figure 5. Total number of misses in the L1D cache using victim cache sweeping the number of entries.