CS5600 Final project – Lu Yan

Introduction

Our VMM simulator mainly focused on comparing TLB hit rates with both fixed 4kb page size and variable page size. In this extended project, **4 different cache replacement policies will be implemented and evaluated.** They are:

1. **FIFO**

   The entry that is first inserted in will be the first one to evict.

   Advantages: simple to implement

   Disadvantages: does not use history; may have worst-case performance under some cases, for example sequential access

2. **Random**

   Pick a random entry to evict each time.

   Advantages: simple to implement; avoid worst-case performance for corner cases that FIFO or LRU may encounter

   Disadvantages: does not use history and consider locality

3. **Least frequently used**

   When cache is full, evict the least frequently referenced entry.

   Considering its problems (listed below), LFU is often used in combination with other algorithms.

   Advantages: make use of history and memory access locality to increase cache hit rate

   Disadvantages: cause overhead by scanning the frequency per memory reference; may easily evict the latest coming entry but that entry can be referenced again;

4. **Least recently used**

   When cache is full, evict the least recently referenced entry.

   Considering its overhead, an approximation of LRU is usually used to achieve similar performance at a lower hardware cost.

   Advantages: make use of history and memory access locality to increase cache hit rate

   Disadvantages: cause overhead by updating the data structure per memory reference, or scanning through the data structure for each eviction; may have worst-case performance under some cases, for example sequential access

In the first part, a simple but more flexible cache (basically a vector) with these 4 policies is implemented to test limited numbers of test cases, in order to specify the optimal cases and some corner cases, etc. (This cache is independent from VMM simulator).

In the second part, these 4 policies are implemented into the VMM simulator, then the test generator can be used to test TLB hit rate with large amount of memory access.

Part 1

Using a simple implementation to test customized cases.

Please see github repo for code.

https://github.com/luyan72999/Computer-System/tree/main/vmm_policies_final_project/cache_implementation

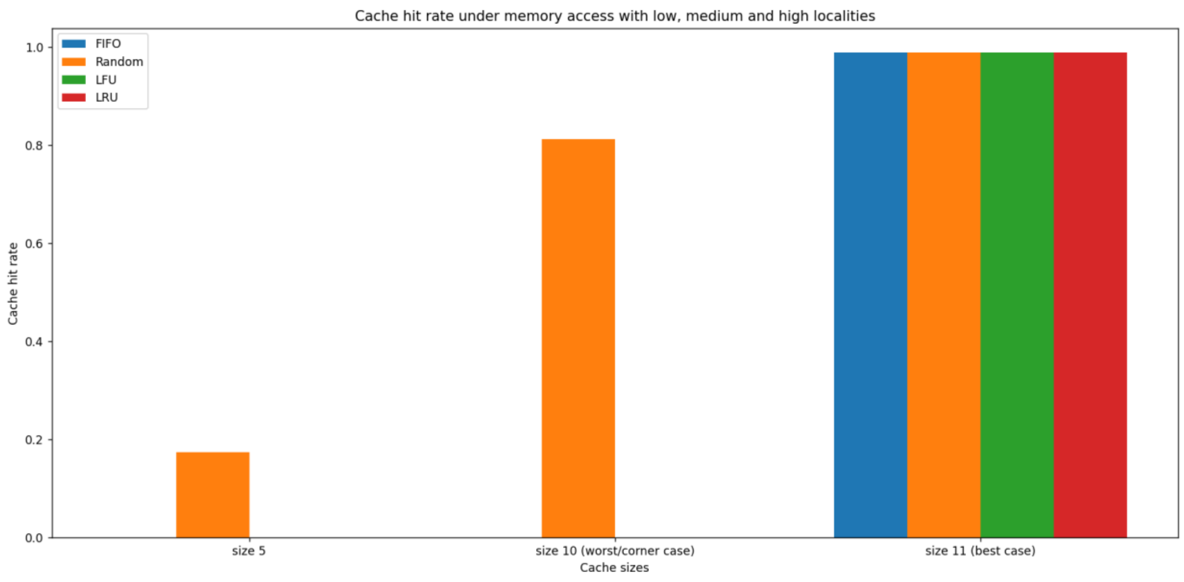
(1)Tests with worst/corner and best cases

The following graph is drawn under cache size 5,10 and 11, to test the following repeated sequential workload.

```
1,2,3,4,5,6,7,8,9,10,11,1,2,3..10,11,1,2,3...10,11,1,2,3..
```

Therefore, when cache size is too small (5) or in the corner case (10), worst cases are generated for fifo, lfu and lru with hit rate 0.

When cache size is 11 or big enough, best cases can be generated for all 4 policies with almost 100% hit rate.



(2)Tests with different localities

Low locality pattern (repeated 100 times)

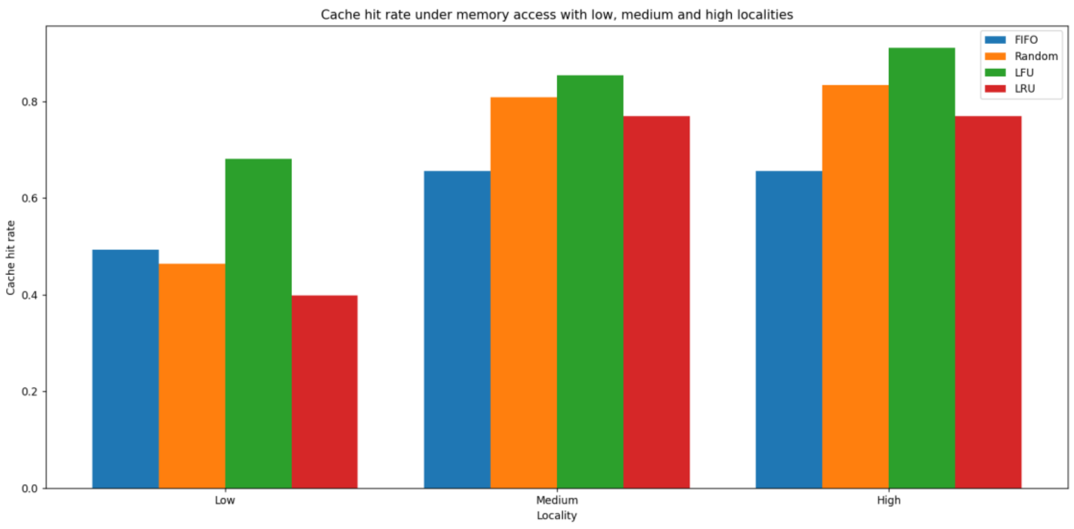
```
{1,2,3,4,5,6,7,8,9,10,16,20,11,6,1,2,18,17,3,9,10,2,3,20,19,15,3,5,10,1
1,7,2,4,5,8}
```

Medium locality pattern (repeated 100 times)

```
{1,2,3,4,5,6,7,8,9,10,16,2,6,1,3,9,16,10,2,9,3,2,1,8,6,2,3,5,10,11,1,2,
4,5,11}
```

High locality pattern (repeated 100 times)

```
{1,2,3,4,5,6,7,8,9,10,16,2,16,7,9,9,7,10,2,9,3,2,3,2,9,2,3,5,10,11,1,2,
4,5,11}
```



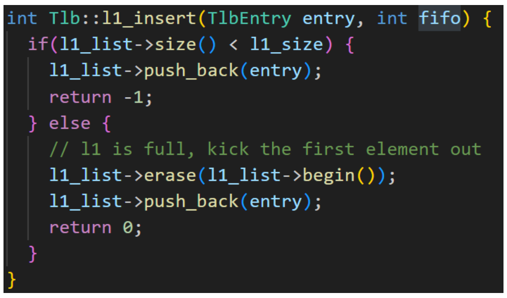
For the 3 test cases above, firstly, least frequently used policy outperforms all the other ones. Secondly, Random and LRU performs better than FIFO. More test cases need to be designed.

Part 2

Policy implementation in VMM simulator

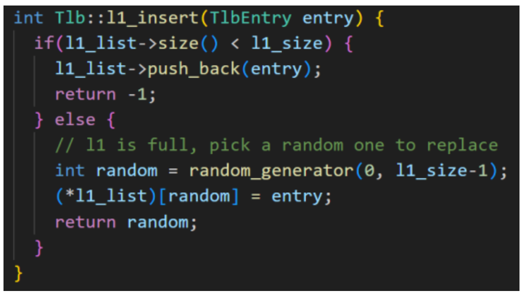
The following pictures show the 4 policies implemented in the 1st level TLB for single process. For full code, please see github repo https://github.com/luyan72999/Computer-System/tree/main/vmm_policies_final_project

FIFO (first in first out)



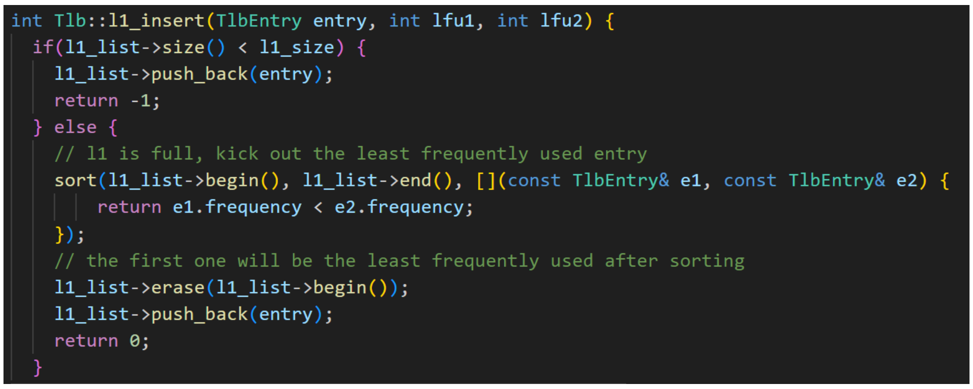
```
int Tlb::l1_insert(TlbEntry entry, int fifo) {
  if(l1_list->size() < l1_size) {
    l1_list->push_back(entry);
    return -1;
  } else {
    // l1 is full, kick the first element out
    l1_list->erase(l1_list->begin());
    l1_list->push_back(entry);
    return 0;
  }
}
```

Random



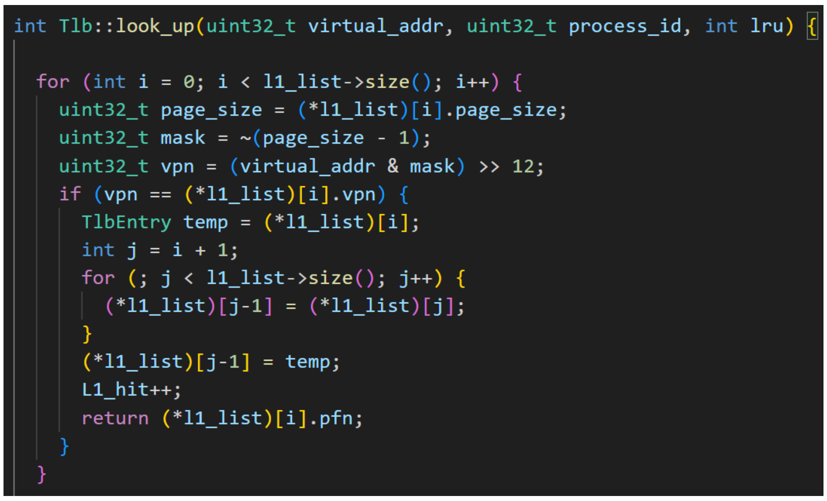
```
int Tlb::l1_insert(TlbEntry entry) {
  if(l1_list->size() < l1_size) {
    l1_list->push_back(entry);
    return -1;
  } else {
    // l1 is full, pick a random one to replace
    int random = random_generator(0, l1_size-1);
    (*l1_list)[random] = entry;
    return random;
  }
}
```

LFU (use a frequency attribute of TLB entry to keep track of the frequency of that entry)



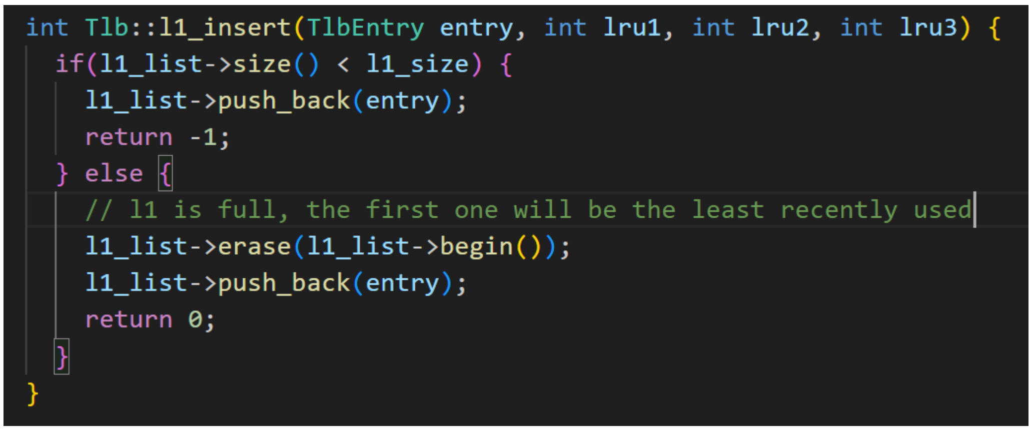
```
int Tlb::l1_insert(TlbEntry entry, int lfu1, int lfu2) {
  if(l1_list->size() < l1_size) {
    l1_list->push_back(entry);
    return -1;
  } else {
    // l1 is full, kick out the least frequently used entry
    sort(l1_list->begin(), l1_list->end(), [](const TlbEntry& e1, const TlbEntry& e2) {
        return e1.frequency < e2.frequency;
    });
    // the first one will be the least frequently used after sorting
    l1_list->erase(l1_list->begin());
    l1_list->push_back(entry);
    return 0;
  }
```

LRU (every time a page is referenced, change its position to the back; when inserting new TLB entries, insert it also to the back)



```
int Tlb::look_up(uint32_t virtual_addr, uint32_t process_id, int lru) {

  for (int i = 0; i < l1_list->size(); i++) {
    uint32_t page_size = (*l1_list)[i].page_size;
    uint32_t mask = ~(page_size - 1);
    uint32_t vpn = (virtual_addr & mask) >> 12;
    if (vpn == (*l1_list)[i].vpn) {
      TlbEntry temp = (*l1_list)[i];
      int j = i + 1;
      for (; j < l1_list->size(); j++) {
        (*l1_list)[j-1] = (*l1_list)[j];
      }
      (*l1_list)[j-1] = temp;
      L1_hit++;
      return (*l1_list)[i].pfn;
    }
  }
```



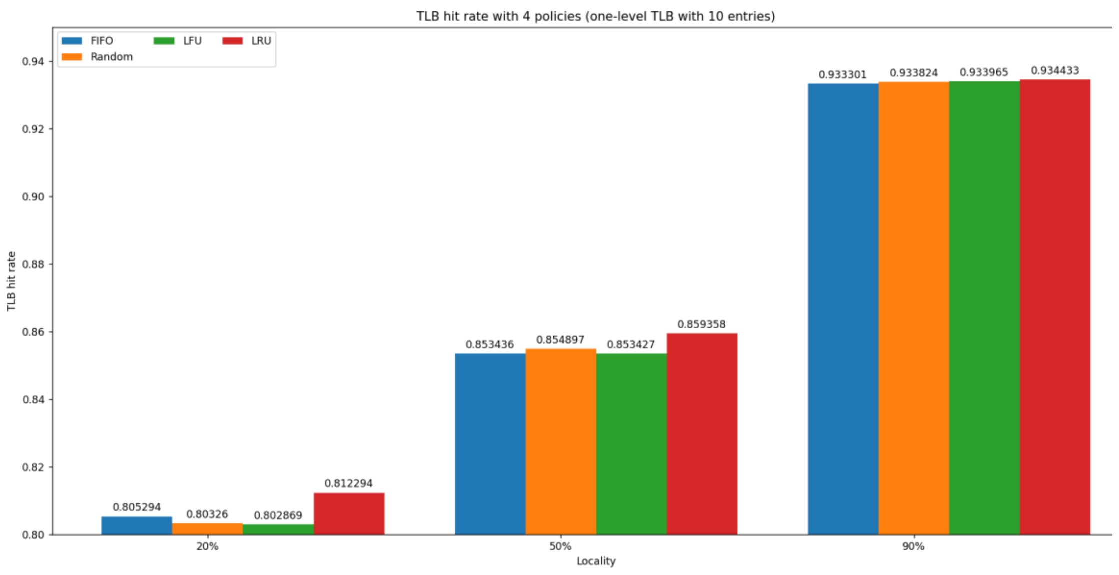
```
int Tlb::l1_insert(TlbEntry entry, int lru1, int lru2, int lru3) {
  if(l1_list->size() < l1_size) {
    l1_list->push_back(entry);
    return -1;
  } else {
    // l1 is full, the first one will be the least recently used
    l1_list->erase(l1_list->begin());
    l1_list->push_back(entry);
    return 0;
  }
}
```

Test cases and Results

The test cases are classified by **3 different localities: 20%, 50% and 90%**. With the test generator, the term locality means the probability that the next address to access is near its previous one.
The aim is to **compare the performance of 4 policies** (algorithms) under low to high locality access patterns.

Under each locality class, 10 test files are generated each containing 9195 access, then the mean of these 10 files is extracted as the final TLB hit rate shown in the graph below. The test cases are run under the condition for one-level TLB and variable page sizes.

More information can be found under the folder named “test_cases”.



When locality is relatively low (20% and 50%), LRU performs better than other algorithms. When locality is very high (90%), all 4 algorithms perform similarly with LRU slightly better than others.

This experiment helps me validate the ideas in textbook ostep that cache replacement policies may only slightly impact overall performance (hit rate). Also, the complexity and additional time and space overheads of implementing complex policies must also be considered as a trade-off to know whether a replacement policy can really improve the overall performance of cache.

Note: The TLB hit rate calculation method chosen counts the retried TLB hit after a TLB miss also as a hit. Due to this calculation method, the overall hit rates are high even when localities are low.