LE-3
Due Date: Feb 25th 2024
Arrays vs Linked List

Description:
The lab exercise LE3 is an extension of the Report section of PA1. In PA1, you were tasked to record the timings to push a progressively large number of elements across the three implementations: StackArrayDouble, StackArrayLinear, and StackLinkedList. You would have discovered that the Linked List implementation of the Stack is actually much more inefficient compared to the Array-Based implementation of the Stack. This might have come as a surprise considering that the time complexity for the push operation is O(1) for the Linked List compared to Arrays which are O(1) on average (amortized time complexity) and O(n) in the worst case (during a resize).

In the Lab Exercise, we will explore how the underlying memory storage and retrieval mechanism impacts the actual timing for pushing elements into the array and the linked list.

Caches
Retrieving data from the main memory each time is expensive and inefficient. To avoid this inefficiency, our computers have multiple levels of smaller cache memories which are closer to the processors and can help access data quicker.

- On typical processors, we have three levels of caches
    - **L1 Cache**
        - Has the smallest storage capacity but is located closest to the processor
        - Provides fastest retrieval times
    - **L2 Cache**
        - Has larger memory capacity than L1 Cache; faster retrieval time than L3 Cache but slower than L1
    - **L3 Cache**
        - Has largest memory capacity and is located furthest away from the processor; slower retrieval times than L1 and L2 Caches

What data does the cache store?

- **Temporal Locality:**
    - This principle states that if a particular memory location is accessed, **it is likely that the same location will be accessed again in the near future.** Caches exploit this by keeping recently accessed data in the hopes that it will

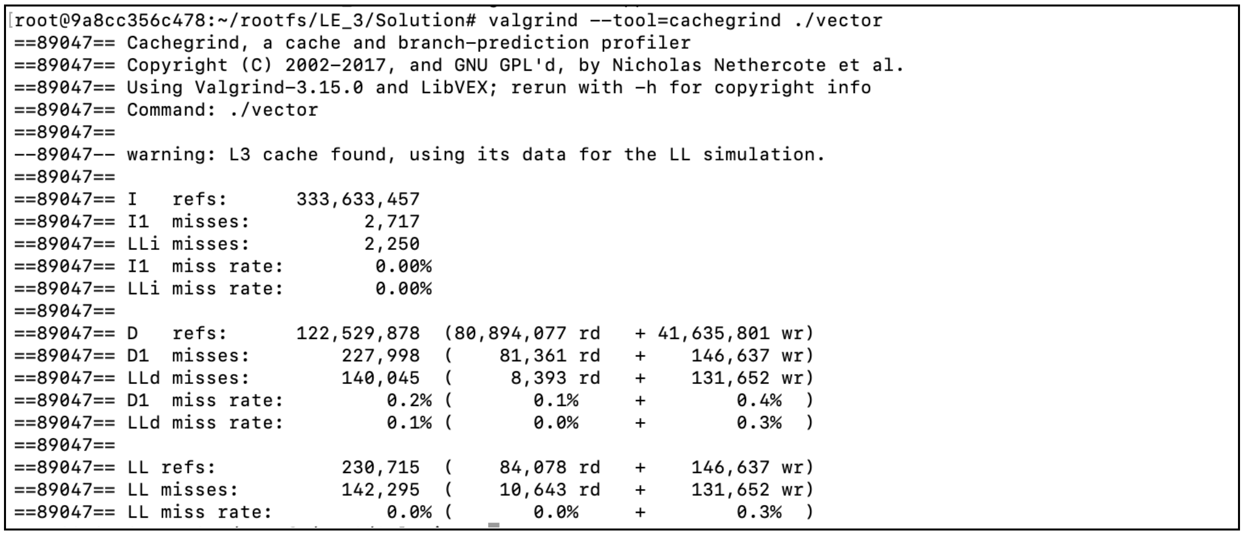
be used again soon. This allows for faster retrieval times of memory which are repeatedly used by the program.

- **Spatial Locality:**
    - This principle states that if a particular memory location is accessed, nearby locations are also likely to be accessed soon. Caches take advantage of this by **fetching blocks of contiguous memory**, rather than just a single data point.
        - For example, if memory location at address 5 is used, not just address 5 but memory locations near it such as address 4, address 3, address 6, address 7 etc. are brought to the cache.

In programs, loop structures tend to exhibit generous leverage of temporal and spatial locality. When the processor needs to access data, it first checks if the data is in the L1 cache. If the data is not found in the L1 cache, it checks the L2 cache, and so on. If the data is not found in any of the caches, a request is made to the main memory which uses the above principles to bring the particular memory location **(and the surrounding memory regions)** to the caches for quick access in the future. **This event is called a Cache Miss.**
If the data is found in caches, it is called a **Cache Hit.** Cache Hits are naturally much more efficient than cache misses as we do not have to access the main memory.

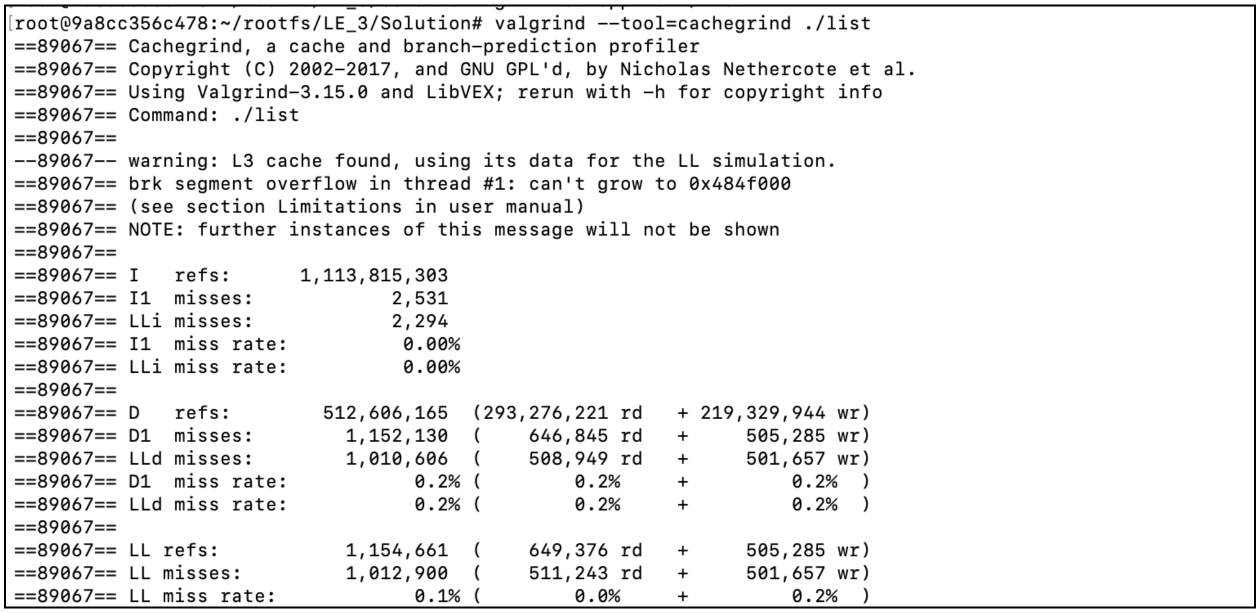
We can use the tool cachegrind (which is a part of valgrind) in order to simulate the L3 Cache Misses of any program. The number of "LLd" misses indicate the number of L3 Cache misses during the execution of a program.

Below is the output of running Cachegrind on vector.cpp:



```
root@9a8cc356c478:~/rootfs/LE_3/Solution# valgrind --tool=cachegrind ./vector
==89047== Cachegrind, a cache and branch-prediction profiler
==89047== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==89047== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==89047== Command: ./vector
==89047==
--89047-- warning: L3 cache found, using its data for the LL simulation.
==89047==
==89047== I   refs:      333,633,457
==89047== I1  misses:          2,717
==89047== LLi misses:          2,250
==89047== I1  miss rate:        0.00%
==89047== LLi miss rate:        0.00%
==89047==
==89047== D   refs:      122,529,878  (80,894,077 rd   + 41,635,801 wr)
==89047== D1  misses:        227,998  (    81,361 rd   +    146,637 wr)
==89047== LLd misses:        140,045  (     8,393 rd   +    131,652 wr)
==89047== D1  miss rate:         0.2% (       0.1%     +        0.4%  )
==89047== LLd miss rate:         0.1% (       0.0%     +        0.3%  )
==89047==
==89047== LL refs:           230,715  (    84,078 rd   +    146,637 wr)
==89047== LL misses:         142,295  (    10,643 rd   +    131,652 wr)
==89047== LL miss rate:          0.0% (       0.0%     +        0.3%  )
```

Below is the output of running Cachegrind on list.cpp:



```
root@9a8cc356c478:~/rootfs/LE_3/Solution# valgrind --tool=cachegrind ./list
==89067== Cachegrind, a cache and branch-prediction profiler
==89067== Copyright (C) 2002-2017, and GNU GPL'd, by Nicholas Nethercote et al.
==89067== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==89067== Command: ./list
==89067==
--89067-- warning: L3 cache found, using its data for the LL simulation.
==89067== brk segment overflow in thread #1: can't grow to 0x484f000
==89067== (see section Limitations in user manual)
==89067== NOTE: further instances of this message will not be shown
==89067==
==89067== I   refs:      1,113,815,303
==89067== I1  misses:            2,531
==89067== LLi misses:            2,294
==89067== I1  miss rate:          0.00%
==89067== LLi miss rate:          0.00%
==89067==
==89067== D   refs:        512,606,165  (293,276,221 rd   + 219,329,944 wr)
==89067== D1  misses:        1,152,130  (    646,845 rd   +     505,285 wr)
==89067== LLd misses:        1,010,606  (    508,949 rd   +     501,657 wr)
==89067== D1  miss rate:           0.2% (        0.2%     +         0.2%  )
==89067== LLd miss rate:           0.2% (        0.2%     +         0.2%  )
==89067==
==89067== LL refs:           1,154,661  (    649,376 rd   +     505,285 wr)
==89067== LL misses:         1,012,900  (    511,243 rd   +     501,657 wr)
==89067== LL miss rate:            0.1% (        0.0%     +         0.2%  )
```

Consider the relative number of L3 Cache misses ("LLd misses") in each data structure and how spatial locality can explain this difference in cache misses between the two data structures. Furthermore, consider how cache misses play a role in efficiency.

There are two expectations associated with LE3:

1. In the **starter.zip** folder, you have been provided with vector.cpp and list.cpp (which records the time to push a large number of elements for 7 data points - starting from 10^0 and ending at 10^6 elements)

    a. Compile these cpp files using the makefile provided
    b. Run their respective executables (./vector and ./list)
    c. This will generate two csv files: vector.csv and list.csv which contains the times (in microseconds) needed to push elements of the above mentioned sizes
    d. **Create a single graph (number of elements vs time) comparing the vector timings to the linked list timings**
2. Explain the reasoning behind the difference in timings recorded
    a. First, state the theoretical time complexity of the push operation for both the data structures
    b. Next, **use the principle of spatial locality** to explain why one data structure seems to be significantly more inefficient than the other despite having similar time complexities.
        i. **You must use the above outputs of cachegrind on both the data structures to support and back up your answer.**

Deliverables and Submissions:

You must submit a PDF Report on Canvas containing:

- The graph comparing the timings
- The theoretical time complexities of each data structure's push operation
- The reasoning behind observed time discrepancies using the principle of spatial locality and the output of cachegrind.