Evaluate different cache replacement policies for their effectiveness.

The work requires modifying an open source cache simulator such as Dinero (http://pages.cs.wisc.edu/~markhill/DineroIV/) or Moola (https://github.com/fernandomosquera/Moola---Multicore-Cache-Simulator)

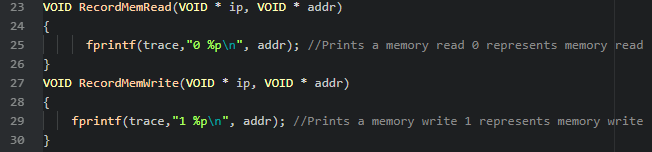
Learn how to use the cache simulator -- what input trace format is used, and how to configure the caches.

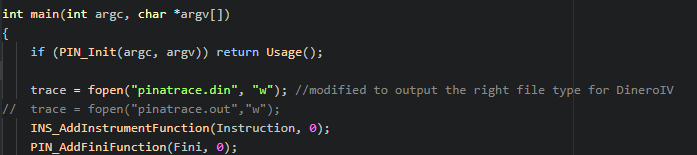
Next, identify different cache replacement policies, implement them and evaluate with a set of benchmarks. Identify which replacement policy achieves the lowest cache miss rate and its complexity in implementing it.

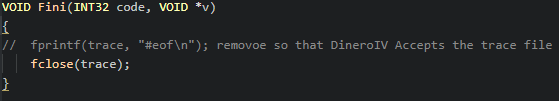
* Generating the trace files - Chris
* Specifically what types of files were created and why - Chris
* Different cache policies we selected and why - Tim
* Talk about results - Juan (be sure to make charts comparing the different rates
  + Chart for each policy (LRU, MRU ect for each with the total miss rate, read and write miss rate -- probably do a bar chart with 15 bars total (5 policies - 3 miss rates)
  + Also do a chart comparing the total miss rates across all 3 different files and the different miss rates (5 policies and 3 total miss rates)
* Talk about difficulty implementing - Tim
* Intro describing the sim, pin, project

Generating the Trace Files

The group used the intel pin tool to generate the trace files using the pinatrace.cpp. Generation of the trace files were generated using sample programs that the group implemented for the project. The total demand fetches for each trace file was 500,000 with a maximum variance of 4% between the files. We modified the source code of the pinatrace.cpp to output the right format for the DineroIV simulator, where the memory writes are represented by 1 and memory reads are represented by a 0. The file type that DineroIV accepts is a .din file so that was modified as well. The tool will by default place a #eof on the end of any file and that must be commented out for DineroIV to accept the trace file.







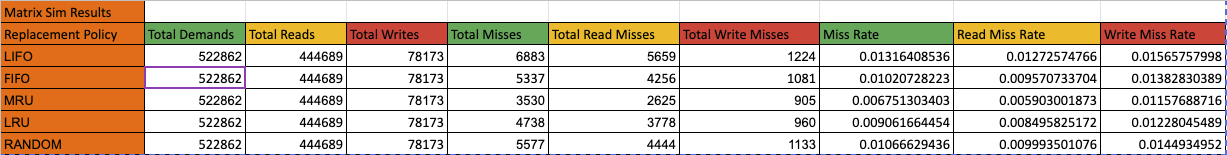
The Group implemented three test programs to simulate for the project. The test programs consisted of a Sorting algorithm using 190 array elements to sort. Also we used a File I/O test that would read a file in chunks 542 bytes from a 17 MB text file and write to a destination file. Finally we made a matrix multiplication test file with matrix sizes of 31x41 that would multiply doubles. We originally had large trace files but we scaled back the size of the traces to mainly draw out the contrast between cache replacement policies, and also reduce simulation and generation times if we had made changes to the simulator or test programs. Another factor for scaling each of the test files was to try and match the total demand fetches for each program, there was a small variance between the programs but not over four percent.

Policies Used

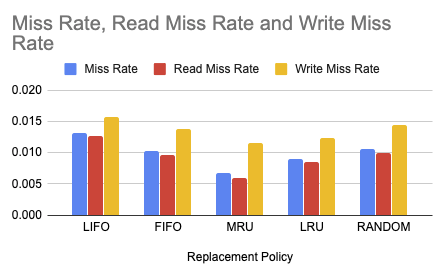
To select the different cache policies we first opted to include the 3 supported replacement policies in DineroIV; LRU, FIFO, and random replacement. Random was included to provide a good baseline performance for the code. Any policy performing worse than random would be assumed to be inefficient. LRU and FIFO on the other hand were included to give an idea of a decent cache replacement policies performance. In LRU and FIFO we expect values that are older and unused to be removed first therefore code that is often using the same few values over and over would benefit from this system. Then we decided to implement two of our own policies, MRU and LIFO. These two methods replace the freshest values and should perform very well with code that does not require data recursively. Using MRU, LRU, FIFO, LIFO, and Random we hoped to gain a full picture on our code's performance in different cache systems.

Results

Matrix Multiplication

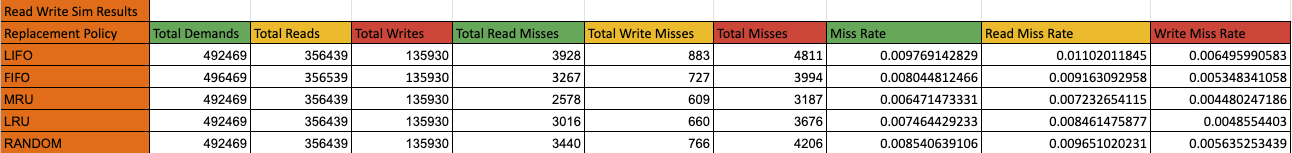


(Overall Results of the Matrix Simulation)

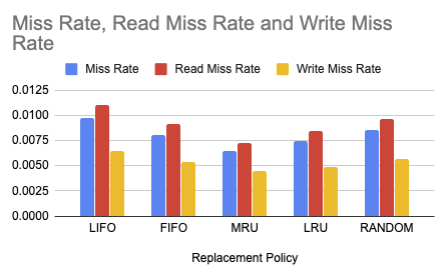


(Miss Rate, Read Miss Rate and Write Miss Rate of the Matrix Simulation)

Read and Write

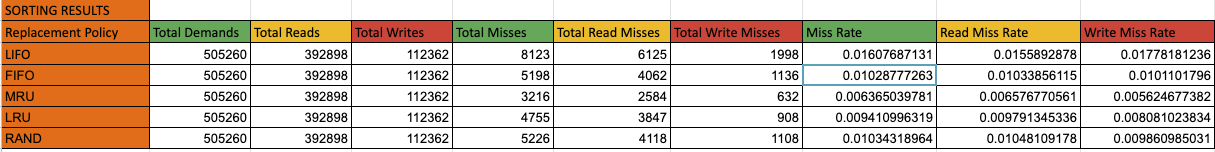


(Overall Results of the Read and Write Simulation)

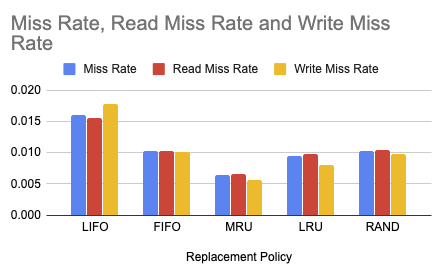


(Miss Rate, Read Miss Rate and Write Miss Rate of the Read and Write Simulation)

Sorting

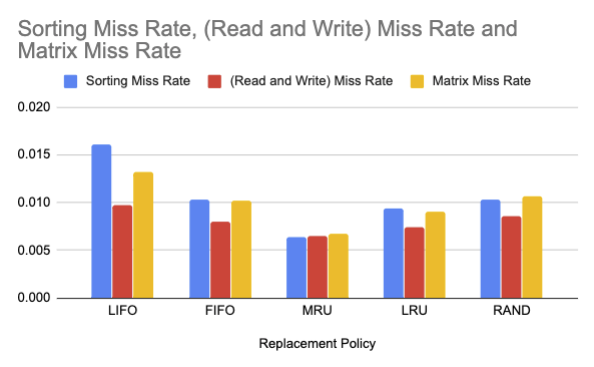


(Overall Results of the SortingSimulation)

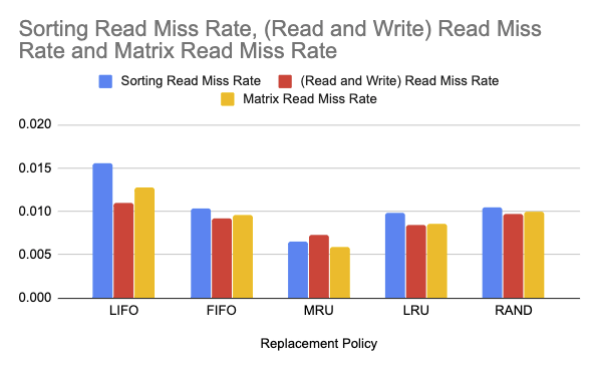


(Miss Rate, Read Miss Rate and Write Miss Rate of the Sorting Simulation)

Overall Results



(Miss Rate results of the three simulations)



(Read Miss Rate results of the three simulations)



(Write Miss Rate results of the three simulations)

During the replacement policies we were able to identify which replacement policies fit best for each of the algorithms. The best replacement policy for the sorting algorithm is MRU (Most Recently Used). This is the perfect replacement policy for sorting because the algorithm needs to keep the most recently used numbers ready to be compared and used again. The best replacement policy for the Read and Write algorithm is MRU (Most Recently Used) as well. Read and Write constantly reads from a file and writes to a file so it must have the most recently used in the cache. The best replacement policy for the matrix algorithm is MRU (Most Recently Used) as well. Overall, we saw a major improvement by using the MRU Replacement Policy across the three algorithms that we tested. We also saw the worst algorithm for all three algorithms. LIFO (Last In First Out) is the worst algorithm for all three tests.

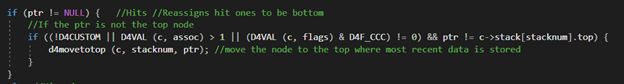
Implementation and Difficulties

The implementation of the different cache policies into the DineroIV simulator was a fairly painless process. By integrating our policy on top of the old policies we were able to make use of all the other simulator logic and the compilation flags. From there we just needed to understand how best to implement the given policy onto the simulator using the built in functions and data members. DineroIV uses a doubly linked list to maintain its cache blocks so all of the operations we added were making modifications to what linked list node was operated on.

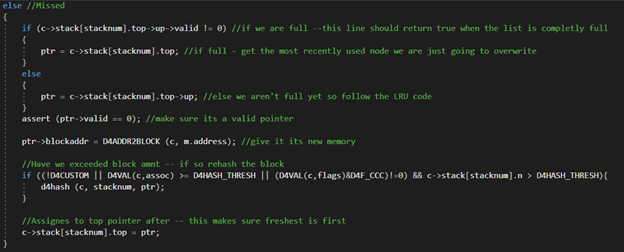
Before creating our own custom replacement policies, it was important to understand how DineroIV implemented its own policies. First, we located the replacement policies definition in “ref.c” and began to understand the source code. The first thing we noticed was that the blocks were organized in a doubly linked list with each node having an up and a down node. The tops up node pointed to the very bottom and bottoms down node pointed to the top. Next, we found that all three methods used the same technique to replace cache blocks.

Each function was given a pointer to a cache block. If the pointer was valid, we assumed that we hit a valid node and didn't need to replace it. The only operation needed to be performed was to move the hit node to the top of the list in LRU case so that the oldest data would naturally move to the bottom. If we missed (got a null for the pointer value) then we would be passed in a valid memory address to save. From there we grabbed the bottom node in the list by referencing the tops->up node and overwrote that data with the new valid memory data. Finally, the data would be assigned as the top of the list and the entire block would be rehashed.

After understanding the already implemented code we began to integrate our own MRU and LIFO replacement policies using the same technique. For MRU the first step was to check if we hit a valid block in the list and if the block was not already the “top” node. If the value was not the top node then the node was sent to the top so it could be identified as the most recent node.



If the information was not found in cache however we then needed to implement our replacement policy. For MRU and LIFO, you first have to ensure that you fill up the full list. That means until the list is full the MRU and LIFO operate the same way the LRU and LIFO do where the empty bottom node is overwritten with the new data and set as the top. Then upon all nodes in the list being full you can simply overwrite the most recent data found at the top of the list.



For LIFO, the code process is the same however you do not have the initial hit check in the code as nodes are never readjusted. The last node added is always the first to go even if another node is accessed after the addition. This actually leads to some interesting performance decreases as detailed above.

The primary difficulty in implementing these two policies was understanding what exactly was happening in the original source code. Once found the policies were very straightforward to implement using basic linked list operations. Additionally, there was significant difficulty with utilizing the software in a virtual machine especially with larger trace files. We would see performance problems and even some inconsistencies in our simulations occasionally.