Nicholas Matsumoto

932-709-078

CS 472

# Final Project



## Cache Simulator Overview

### Cache Simulator Control Flow

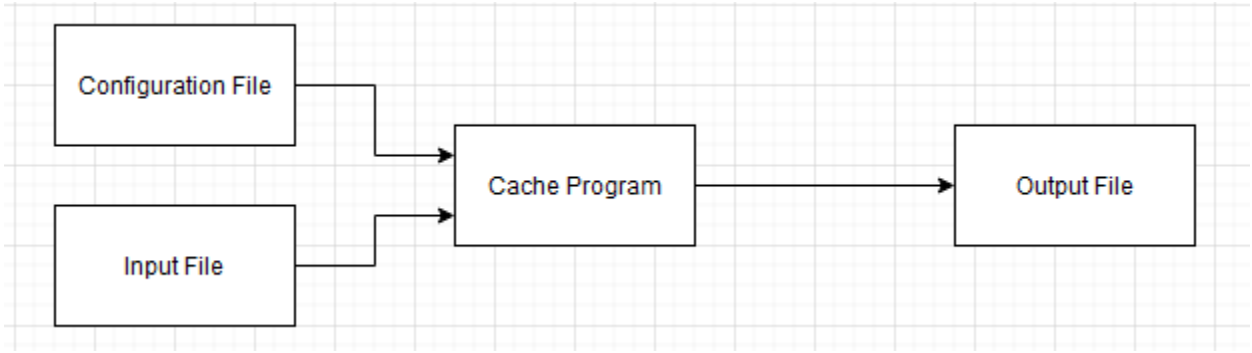


Figure 1: Cache Simulator High Level Overview

The cache simulator program takes two arguments, a cache configuration file and an input file. The cache simulator then outputs an output file that holds the global number of cycles, hits, misses, evictions, and a detailed description of every cache change.

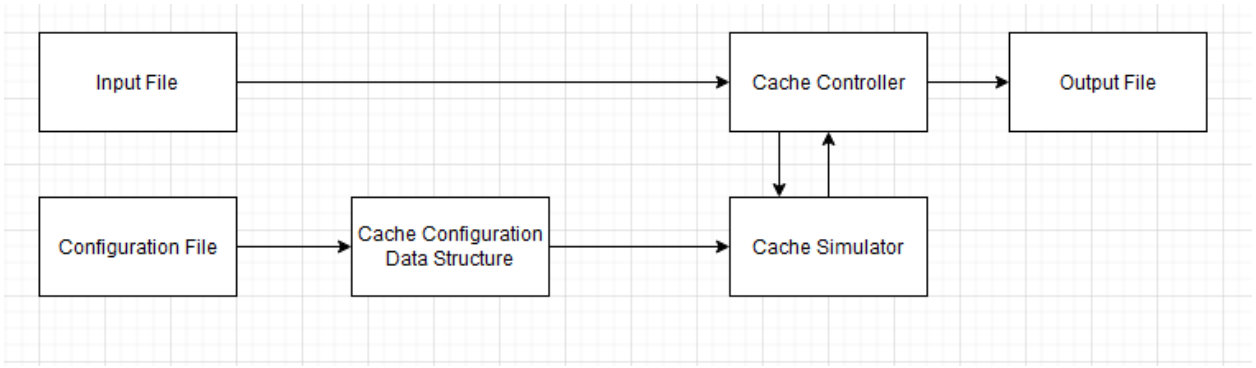


Figure 2: Cache Simulator Program Block Diagram

The cache simulator first reads the configuration file into a cache configuration data structure. Then the cache controller initializes an array of address info structures and clears their valid bits to represent that the cache blocks have meaningless data values. Next, the input file is read line by line and the cache controller manipulates the cache array according to the configuration data structure. Lastly, results of the cache accesses are printed to an output file.

## Cache Data Structures

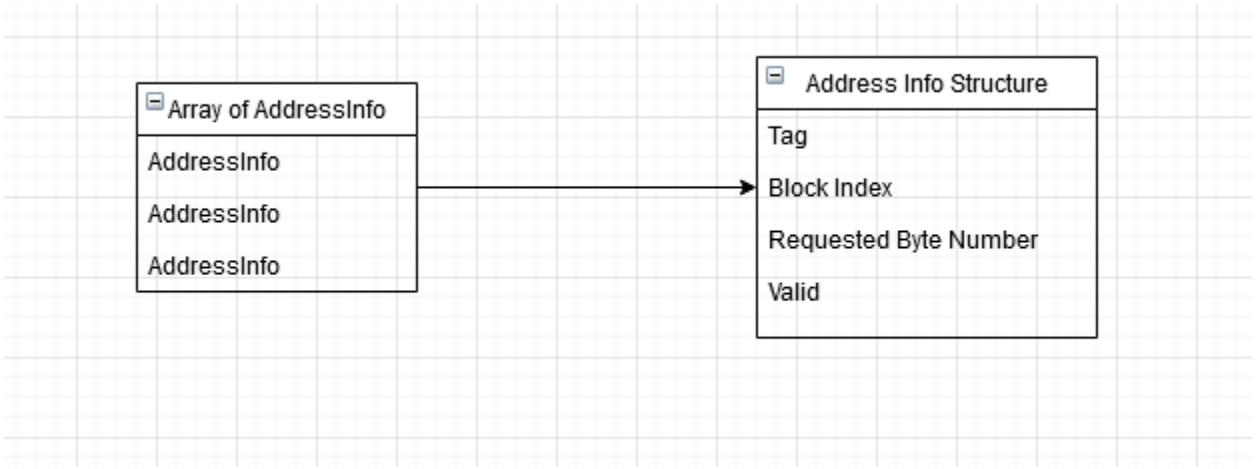


Figure 3: Cache Data Structure

The cache simulator used an array of cache address structures. A cache address structure contains: 1) A tag, 2) The index of the block where the data is to be stored, 3) A valid bit to determine if a cache entry has data in it, and 4) A byte number that is used for determining whether a request is split across multiple cache blocks.

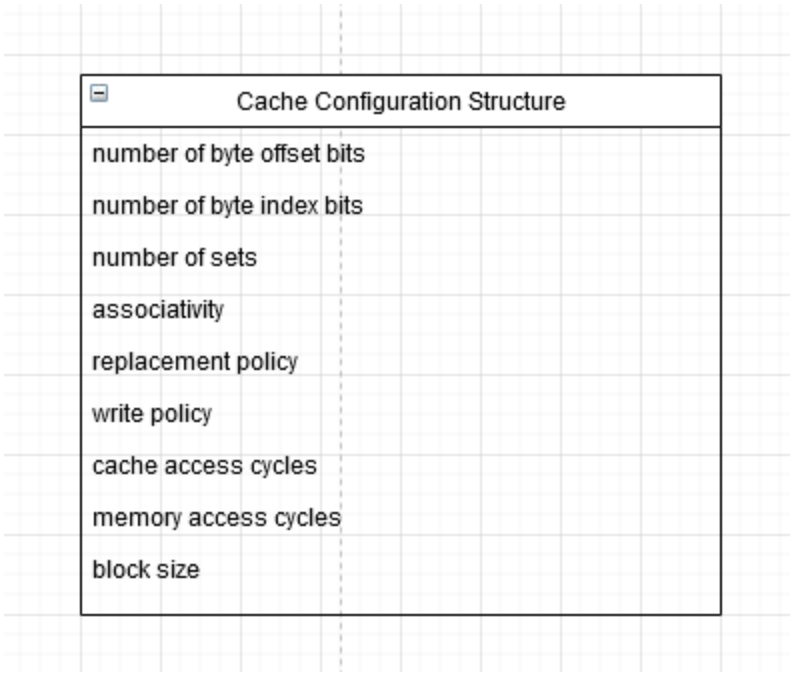


Figure 4: Cache Configuration Data Structure

The cache data configuration structure has fields for the number of byte offset bits, number of index bits, number of sets, associativity, replacement policy, write policy, cache access cycles, memory access cycles, and block size.

## Cache Simulator Important Functions

### getAddressInfo Function

The getAddressInfo function returns an AddressInfo structure that holds data regarding the tag, index, and byte number. The function converts the integer address into a binary value which is then split into the tag, index, and byte number. The byte number represents which byte an address points to within the given offset.

### cacheAccess Function

The cacheAccess function defines the operations of a cache according to a set of instructions. The function loops through a set of addresses dependent on the number of bytes requested by the user. After fetching the address information the function then copies the appropriate cache set into a new array. The temporary cache is then compared with the desired tag with hits, misses, and evictions being marked. The temp cache is then copied back into memory and the number of cycles is then calculated from the number of hits, misses, and evictions with different formulas being used for contiguous and noncontiguous memory access. Contiguous memory access in this context is defined as whether the pointer to RAM is left to point to address N by an operation; a following operation that accesses RAM at N + 1 will be considered contiguous.

# Cache Simulator Challenges

The primary challenges encountered while working on this project was determining the correct amount of cycles. This challenge is mainly due to the amount of edge cases that exist such as a byte range request that spans multiple blocks, determining the cycles for contiguous RAM access, and calculating the clock cycles for a modification operation.

# Cache Simulator Revamp

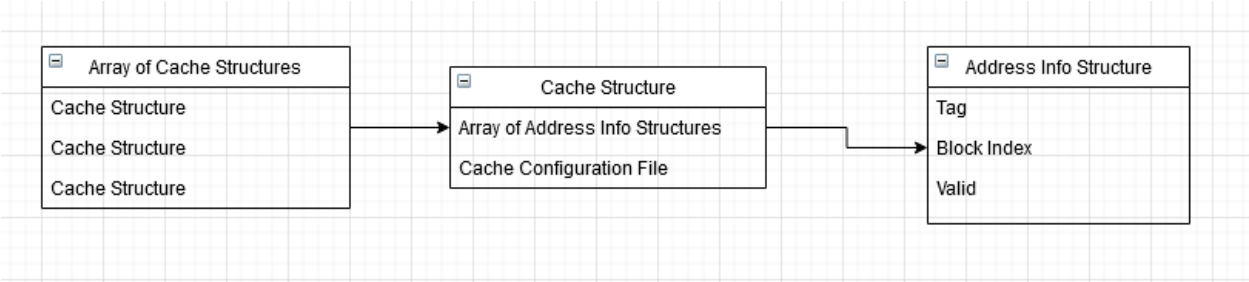


Figure 4: Improved Cache Structure Implementation

A reimplementation of the cache simulator would focus mainly on creating a more maintainable data structure for the cache. An improvement to the array of structures would be a cache structure that holds an array of address info structures and a configuration object. This object oriented architecture would allow for an easier implementation of a multitiered cache structure. Additionally, cycles rather than being calculated all at once after a block is finished processing would be calculated as soon as a hit, miss, and eviction were detected. This fine grained approach would make handling edge cases easier in comparison to calculating cycles at the end of a cache access. Lastly, in order to improve program performance the input file would be read all at once to eliminate the file I/O bottleneck.

# Clock Cycles Overview

```
if(isWrite == false && localResponse.contiguousAccess < 1){
    localResponse.cycles +=
        localResponse.hits * this->ci.cacheAccessCycles +
        localResponse.misses * this->ci.cacheAccessCycles +
        localResponse.misses * (this->ci.memoryAccessCycles + extraMemoryBlocks);
}
if(isWrite == true && localResponse.contiguousAccess < 1 && localResponse.hitAccess < 1){
    localResponse.cycles +=
        localResponse.hits * this->ci.cacheAccessCycles +
        localResponse.hits * (this->ci.memoryAccessCycles + extraMemoryBlocks) +
        2 * (localResponse.misses * this->ci.cacheAccessCycles +
        localResponse.misses * (this->ci.memoryAccessCycles + extraMemoryBlocks));
}
```

Figure 5: C++ noncontiguous memory clock cycle calculation implementation

Clock cycles are tracked by counting the number of hits, misses, and evictions and at the end of a cache access calculating the number of clock cycles. Two formulas are used to determine the clock cycles which are dependent on whether an access is a write or a read. This formula is dependent on the access being noncontiguous. The noncontiguous read access clock cycle formula is:

$$Clock\ Cycles\ =\ Hits\ *\ cache\ access\ cycles\ +\ Misses\ *\ cache\ access\ cycles$$

$$+\ Misses\ *\ (memory\ access\ cycles\ +\ extra\ memory\ blocks)$$

The noncontiguous memory write access clock cycle formula is:

$$Clock\ Cycles\ =\ Hits\ *\ cache\ access\ cycles\ +\ Hits\ *\ (memory\ access\ cycles\ +\ extra\ memory\ blocks)$$

$$+\ 2\ (Misses\ *\ (cache\ access\ cycles)$$
$$+\ Misses\ *\ (Memory\ Access\ Cycles\ +\ extra\ memory\ blocks))$$

```
if(isWrite == true){
    sequentialResponse.cycles =
        sequentialResponse.hitAccess * (this->ci.cacheAccessCycles + extraMemoryBlocks + 1)
        + sequentialResponse.contiguousAccess
        * (this->ci.cacheAccessCycles * 2 + extraMemoryBlocks*2 + 1 + this->ci.memoryAccessCycles)
        + (sequentialResponse.hits - sequentialResponse.hitAccess) * this->ci.cacheAccessCycles +
        (sequentialResponse.hits - sequentialResponse.hitAccess) * (this->ci.memoryAccessCycles + extraMemoryBlocks) +
        2 * ((sequentialResponse.misses-sequentialResponse.contiguousAccess) * this->ci.cacheAccessCycles +
        (sequentialResponse.misses-sequentialResponse.contiguousAccess) * (this->ci.memoryAccessCycles + extraMemoryBlocks));
}
if(isWrite == false){
    sequentialResponse.cycles =
        (sequentialResponse.contiguousAccess)
        * (this->ci.cacheAccessCycles + extraMemoryBlocks + 1)
        + sequentialResponse.hits * this->ci.cacheAccessCycles;
}
```

Figure 6: C++ contiguous memory clock cycle calculation implementation

Contiguous memory clock cycles are tracked using different methods. If the operation is a load, then the program tracks the number of sequential accesses and uses the cache access time, total number of blocks, and hits to determine the number of cycles. If the operation is a write, then the program tracks the number of contiguous memory hits, contiguous memory misses, hits, misses, and total number of blocks to calculate the number of cycles. Hits and Misses are the combined total number of hits and misses for an operation

The contiguous memory read access clock cycle formula is:

$$Clock\ Cycles\ =\ Hits\ *\ cache\ access\ cycles\ +\ SequentialMemoryMisses\ *\ (cache\ access\ cycles$$

$$+\ 1\ +\ extra\ memory\ blocks)$$

The contiguous memory write access clock cycle formula is:

$$Clock\ Cycles = ContiguousHits * (cache\ access\ cycles + 1 + extra\ memory\ blocks) + ContiguousMisses * (2 * cache\ access\ cycles + memory\ access\ cycles + 2 * extra\ memory\ blocks + + (Hits - Contiguous\ Hits) * cache\ access\ cycles + (Hits - Contiguous\ Hits) * (memory\ access\ cycles + extra\ memory\ blocks) + 2\ (Misses * (cache\ access\ cycles) + (Misses - ContigousMisses) * (Memory\ Access\ Cycles + extra\ memory\ blocks))$$

```
int extraMemoryBlocks = (ceil((float)( (float)this->ci.blockSize) / 8.) - 1) ;
extraMemoryBlocks = (extraMemoryBlocks > 0) ? extraMemoryBlocks : 0;
```

Figure 7: C++ extra memory block implementation

The variable extra memory blocks is the number of blocks of eight bytes that need to be read or written to RAM past the first access. For example, if 16 bytes needed to be written to RAM then the extra memory block variable would equal one.

## Direct Mapped and 2-Way Set Associative Cache Comparison

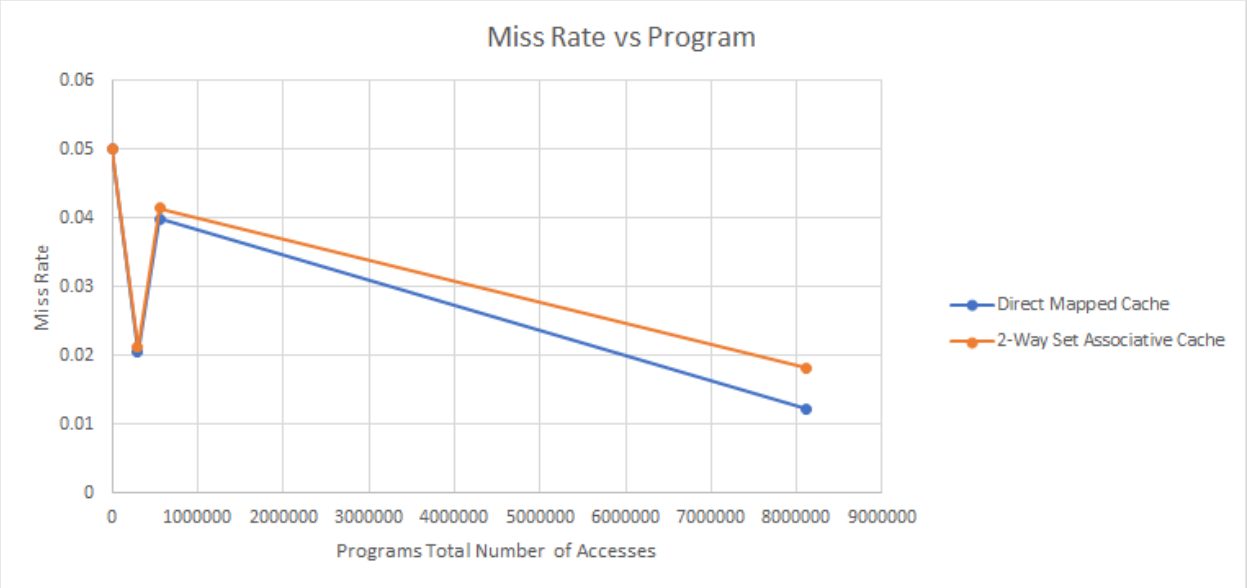


Figure 7: Miss Rate vs Programs Total Number of Accesses

| Direct Mapped Cache | Hits | Misses | Total Accesses | Miss Rate Percentage |
|---|---|---|---|---|
| Simpletracefile | 19 | 1 | 20 | 5% |
| GCC | 290043 | 6077 | 296120 | 2.052208564% |
| NET | 532665 | 22167 | 554832 | 3.995263431% |
| OPENSS | 8009469 | 99750 | 8109219 | 1.230081467% |

Table 1: Direct Mapped Cache Results Per Program

| 2-Way Set Associative Cache | Hits | Misses | Total Accesses | Miss Rate Percentage |
|---|---|---|---|---|
| Simpletracefile | 19 | 1 | 20 | 5% |
| GCC | 289836 | 6284 | 296120 | 2.122112657% |
| NET | 531805 | 23027 | 554832 | 4.150265306% |
| OPENSS | 7961348 | 147871 | 8109219 | 1.823492497% |

Table 2: 2-Way Set Associative Cache Results Per Program

| Program | Miss Rate Percentage Difference |
|---|---|
| Simpletracefile | 0% |
| GCC | 0.0699% |
| NET | 0.155% |
| OPENSS | 0.59341% |

Table 3: Miss Rate Percentage Difference between a Direct Mapped Cache and a 2-Way Set Associative Cache

## Conclusion

The miss rate between a direct mapped cache and a 2-way set associative cache is very similar. This correlation can be seen from the graph as the plotted miss rates are all very similar to one another. As Table 3 shows the difference between the different cache miss rates was at most 0.155% which is fairly negligible. Furthermore, since a variety of program sizes was used it is likely that the data is representative of most programs. Therefore, we can conclude that a direct mapped cache of size N has about the same miss rate as a 2-way set associative cache of size N/2