Cache Simulator

CS 370 Project 2 – Spring 2022
Due: May 2, 2022 at 11:00pm

1 Introduction

Read this entire document.

This is the second and final project for CS 370. The purpose of the project is to give you a more realistic processor modeling task. Once complete, your project will model the detailed behavior of a wide variety of caches. You will then use your completed simulator to find the best cache configuration for a given cache capacity and benchmark program.

Your program will need to read a sequence of memory operations, including the read/write status and address for each access. Your program will also take a cache configuration as an input. Your program will maintain the valid bits, dirty bits, and tags for the given cache configuration, and determine whether each memory access will hit or miss, and will also determine if and when write-backs/write-throughs to the next cache level occur. Since we do not need to know the cache payload in order to determine hit rates and writes to the next level, the cache payload does not need to be modeled.

This project is to be done individually. Any evidence of copying will be reported to the Student Life Office as an academic integrity violation. Be cautious when helping friends with the assignment – discussing your general approach for the various aspects of this assignment is okay, but sharing code is not.

The due date for this project is **Monday, May 2nd, 2022 at 11:00pm**, and must be submitted to Canvas. Late projects cannot be accepted. Your program must run correctly on the `compute.cs.uwlax.edu` departmental server. No other development environments will be supported. It is in your best interest to start this project as early as possible. All of the material that you need to know to complete this project has already been given in lectures.

When you are finished writing your program, you are not yet done with the project. You will need to use your program to find the best cache configuration for a given cache capacity and set of accesses. See Section 3 for details.

2 Project Specification

In this project, you must write a C program that reads a sequence of dynamic memory accesses and reports the number of hits and misses (separately for reads and writes), the hit rate, the number of write-backs, and the number of write-throughs for those memory accesses. The cache parameters (for instance number of sets and block size) will also be an input to your program, so we can vary the parameters to see how the hits/misses, hit rate and write-backs can be improved or worsened for a given memory access pattern.

Since we only are concerned with the hits/misses and number of write-backs/write-throughs, we do not need to maintain the data payload of the cache being modeled. Thus, we do not need to know what values are read from memory for loads, nor do we need to know what values will be written to memory for stores. The cache that you model only needs to maintain valid bits, dirty bits (if a write-back cache), and tags.

Similar to Program 1, there are no performance requirements nor any expectations on clean, well-documented source code. However, you do need to process addresses as binary values (`unsigned ints`) – you **must not** parse addresses as strings. Your program has to function as expected, and must be robust to all possible inputs as defined below. It is up to you to make sure your program is correct. If your program has bugs or does not report the correct statistics, clean and well-documented code may help your grade if I can determine where your program went wrong. But it is not required for full credit.

Similar to Program 1, there is no need to print the status of your program to the screen while it is executing. Only the final output is graded.

The input to your program will be two items: 1) a text file containing the cache parameters, and 2) a second text file containing the memory accesses. These will be discussed fully in the next sections. The output is a text file that reports the number of read hits, number of write hits, the number of read misses, the number of write misses, the hit rate, the number of write-backs, and the number of write-throughs. The format of the output file is also described in its own section below.

2.1 Cache Parameter Input Format

The first input file will be used for giving your program the parameters and policies of the cache that your program should model. This project requires that you model a direct-mapped cache, a 2-way set-associative cache, and a 4-way set-associative cache. The possible range of cache parameters and policies are only those that pertain to these two possible high-level cache configurations.

The input file will be formatted in ASCII plain text. Each line of the file will have one of the parameters that define the dimensions of the cache, as well as the cache policies. Table 1 lists all of the cache parameters and policies that will appear in this input file, their meanings, and the range possible values for each parameter.

Table 1: Cache parameters and policies that your program must support.

Line in Input File	Parameter/Policy	Range of Possible Values	Description
1	Direct-mapped or set-associative	1, 2, or 4	This number indicates the associativity of the cache and will be 1 if the cache should be a direct-mapped cache, 2 or 4 if the cache should be 2-way or 4-way set-associative, respectively.
2	Number of bits in offset	2 to 9	The number of bits of each address that will be used for the offset. This determines the block size of the cache, by raising 2 to the number of offset bits.
3	Number of bits in index	1 to 14	The number of bits of each address that will be used for the index. This determines how many sets the cache will have by raising 2 to the number of index bits.
4	Allocation policy	Write-Allocate (wa) or Write No-Allocate (wna)	Whether the cache should allocate (and possibly replace) blocks when a cache write miss occurs. Write-Allocate means that a block should be brought into the cache (setting valid bits, tag, etc.) when a write miss occurs. Write No-Allocate means that a write miss will not cause a block to be allocated (nor a replacement).
5	Write policy	Write-Through (wt) or Write-Back (wb)	When a write occurs, should the write automatically occur to the next level of the cache hierarchy, or should a write to the next level only occur when a block is replaced. This will determine whether or not you need to model dirty bits. Write-Back means that writes are only done to the current cache until the cache block is replaced.

This input file will always be called `parameters.txt` and will be in your program directory before running your program. This means that it is safe to hardcode the file name within your program. Your program should not overwrite this file.

Each parameter listed in Table 1 will appear on its own line in the `parameters.txt` file, and will not include any other item on the line other than the parameter/policy. The choice between direct-mapped and n-way set associative will always be either 1 (for direct-mapped), 2 or 4 (for 2-way or 4-way set associative, respectively). You do not need to handle associativity beyond 4-way. The number of bits in the offset, and the number of bits in the index will be listed in decimal. The allocation policy and write policy will be abbreviated, and will appear in lower case in the file.

For example, if the Write Allocate policy should be modeled, then the line in the `parameters.txt` file will be "wa". When modeling an n-way set-associative configuration, you should **always use a FIFO replacement policy**, starting with way 0 for each set. The other policies will appear as listed in Table 1 without the parenthesis. The input file will always have five lines, with each line given the meaning specified in Table 1.

Listing 1 shows an example of the `parameters.txt` input file. In this example, the cache is direct-mapped, there are 5 bits used for the offset, there are 8 bits used for the index, the allocation policy is to allocate blocks on write misses, and the write policy is write-back (so writes to the next level in the cache only occur when the block is replaced). This means that for this example, the block size is 32 bytes, there are 256 sets, you need to allocate (and possibly replace blocks) on writes, and you must properly maintain dirty bits in order to track the number of write-backs.

```
1  1
2  5
3  8
4  wa
5  wb
```

Listing 1: Example `parameters.txt` input file.

2.2 Memory Accesses Input Format

This program requires two input files. The first, discussed in the previous section, will give the size and shape of the cache that needs to be modeled. The second, discussed here, will give the sequence of reads and writes to your cache. This second input file will always be named `accesses.txt`, and will also be put in your program directory before running your program. So, similar to the first input file, you can hardcode the file name `accesses.txt` into your program and your program should not overwrite this file.

The `accesses.txt` input file will be a plain ASCII text file, with one <type, address>tuple per line. The *type* refers to whether the access is a read or a write. If the access is a read, then *type* will be a single character "r" (without the quotes). If the access is a write, then *type* will be a single character "w" (also without the quotes). The *address* is the memory location that is being accessed, and will be in hexadecimal without any prefix or postfix to denote hex (i.e. there will be no "0x" before the address). The *address* could be any 32-bit value.

There is no fixed upper limit on the number of memory accesses in the `accesses.txt` input file, with the exception that it is guaranteed that the address pattern will be chosen such that it never causes any single statistic to overflow a 32-bit integer. This means that it is safe for you to use 32-bit integers for your statistic variables. There will always be at least two memory accesses in the `accesses.txt` input file.

Listing 2 shows a small example `accesses.txt` input file, exactly as it would appear when used as an input to your program. The file will always be in ASCII text (i.e. one byte characters) with UNIX/Linux line breaks. Notice that the file has no header row, and has a single space between the type of access (first column) and the address (second column). A copy of the example input files can be obtained from Canvas in a single zipped file with the file name `project2-example.zip`. This will be the only example input file that will be posted. It is your responsibility to come up with additional test cases to stress-test your program.

```
1  r 004aaaa8
2  r 004aaab0
3  r 004a9cc0
4  r 004aa9a0
5  w 7fff9e60
6  w 7fff9e58
7  r 004aad08
8  w 7fff9e48
9  r 004aea70
10 w 7fff9e50
11 w 7fff9e68
12 r 004aad10
13 w 7fff9e38
14 r 004aad08
15 w 004aea70
```

Listing 2: Example `accesses.txt` input file.

This example input file shown in Listing 2 is not sufficient for testing your program. It only includes a small number of accesses, and only tests one size and one set of policies. You should write your own test input files that stress every possible set of policies, and many different cache capacities for both direct-mapped and n-way set-associative configurations. You will want to be sure to come up with test input files that exercise each statistic that is required for this project. The expected output file for this example will also be included in the `project2-example.zip` file on Canvas.

2.3 Output Format

The output should be an ASCII text file with the exact format as shown in Listing 3. The statistics gathered by your program should replace each of the angle-bracketed items in the output. The output file generated by your program should always have the exact name `statistics.txt`. When running your program, I will always be sure that there is not already an existing file with the same file name in your program directory before running your program.

Your output file should exactly match the format in Listing 3: no extra spaces, no extra blank lines, etc. All of the statistics should be reported, even if they are never used by your program. In that case, the statistic should be 0. For example, your program might determine that every memory access misses – in that case the number of read hits and number of write hits should both be 0. More importantly, if the cache that you are modeling has a write-through policy for writes, then the number of write-backs is not needed but your program should still report 0 for the number of write-backs. If the cache that you are modeling has a write-back policy, it is still possible to have write-throughs – see the next paragraph. The number of read hits, the number of write hits, the number of read misses, the number of write misses, the number of write-backs, and the number of write-throughs should always be reported as integer whole numbers. The hit rate (hit ratio), is the number of hits (reads and writes) divided by the total number of accesses (all hits plus all misses). This rate will always be a number between 0.0 and 1.0. Your program should write this number to the output file with at least six places after the decimal point. Having more than six places after the decimal point is allowable. This statistic can be rounded at the last place if you want.

Important clarification: The statistic that counts the number of write-throughs should be incremented whenever a write value is immediately written to the next level in the cache hierarchy. When modeling a write-through cache, this means that *every* write (hit or miss) will increment the write-through statistic. When modeling a write-back cache, it is possible to still increment the write-through statistic: if the allocation policy is Write No-Allocate and a write miss occurs. In a real system, a block would not be allocated, but the write value must still be saved, so it will be written-through to the next level of the hierarchy. Thus, in your program, you would increment the number of write-throughs to reflect this situation.

```
1 rhits: <num-read-hits>
2 whits: <num-write-hits>
3 rmisses: <num-read-misses>
4 wmisses: <num-write-misses>
5 hrate: <hit-rate>
6 wb: <num-writebacks>
7 wt: <num-writethroughs>
```

Listing 3: Output file format.

```
1 rhits: 3
2 whits: 4
3 rmisses: 5
4 wmisses: 3
5 hrate: 0.466667
6 wb: 0
7 wt: 0
```

Listing 4: Partial example output file that matches the example input files from Listing 1 and Listing 2.

Listing 4 shows the output that is expected when using the example input files shown in Listing 1 and Listing 2. When writing these statistics to the output file, you should not include the angle-brackets. This example output file is also included in the `project2-example.zip` file that can be downloaded from Canvas. Notice for the hit rate that the calulation will be seven hits divided by 15 total accesses. The result of that calculation is $0.4\overline{6}$ which cannot be expressed perfectly. Listing 4 shows 0.466667, which is one possible legal output. Other legal outputs for the hit rate

are 0.466666, 0.46666666, 0.46666667, etc. I simply require that you report at least six positions after the decimal, and if you round, you should only round the very last digit. You can search the internet on how to round and how to print a specific number of positions, but the default for `printf` using `"%f"` is already 6 positions after the decimal.

3 Cache Access Analysis

In addition to the program itself, you are required to use your program to determine the best cache configuration for a given 1) total cache payload capacity, and 2) sequence of memory accesses. You will be assigned a unique `accesses.txt` file and target cache size (i.e. you will have your own `accesses.txt` file compared to every other student in the class). Roughly one week before the deadline, I will provide to you an `accesses.txt` and a total cache capacity that you should use. It will be your responsibility to try all of the cache parameters that have the exact total payload capacity (i.e., not including valid bits, tags, or dirty bits) that you are assigned. For example, suppose you are assigned a total capacity of 1024 bytes (1k bytes), possible cache configurations with 1k bytes of capacity are:

- Direct-mapped cache with 4 byte blocks with 256 sets
- Direct-mapped cache with 8 byte blocks with 128 sets
- Direct-mapped cache with 16 byte blocks with 64 sets
- Direct-mapped cache with 32 byte blocks with 32 sets
- ... and so on, including the 2-way and 4-way set-associative configurations

Furthermore, for each combination of number of sets and block sizes, you need to try two different cache policies:

- Write-Allocate with Write-Back
- Write No-Allocate with Write-Back

Your program *does* need to properly model the Write-Through write policy for the regular grading test cases. But for this part of the project you do not need to run experiments for the Write-Through policies.

You should exhaustively test the possibilities and find the cache configuration that has the *maximum* hit rate. If there are two caches that have the exact same hit rate (unlikely, but does occasionally happen), then the tie breaker is the cache configuration that causes the *fewest* write-backs plus write-throughs. It will be guaranteed that there is exactly one correct cache configuration for your given sequence of accesses and cache capacity. When you submit your program, you should not only include your program **source code**, but also the `parameters.txt` file that defines the best cache configuration for your cache capacity and access stream.

This part of the project is worth 30% of the overall Project 2 grade. There will be a deduction if your parameters differs from the expected best cache configuration. The deduction will be based on the difference between the hit rate for your best cache configuration compared to the expected best cache configuration.

Other than the `parameters.txt` file for the best cache configuration, you do not need to submit any other input files or output files. They will not be used for grading, and if you do include any input or output files, they will be replaced when testing anyway.

4 Hints

It is highly recommended to model the cache as an array of `structs` that have valid, dirty, and tag member variables. Each entry of the array represents one *set* of the cache. Since you need to be able to model direct-mapped cache and n-way set-associative caches, you will want to have the ability to save multiple valid bits (up to 4), multiple tags (again, up to 4), and multiple dirty bits per set. If you are modeling a direct-mapped cache, then you would only use one valid/dirty/tag combination of the *set*, when modeling a 2-way associative cache then you would only use two of the four. You can then index into the array of sets with the index bits from each address, and then iterate through the ways within that set. The indexing strategy is similar to the suggested method for keeping track of register statistics

in Program 1 – you can index directly into a set without iterating by using bitwise operators to extract the index bits from the addresses, but you will have a second dimension (the ways) that you will need to iterate through.

Remember to always assume FIFO replacement when modeling the 2-way and 4-way set-associative caches. Thus, you will also want to have a member variable of your *set* `struct` that keeps track of the block that should be replaced. This needs to be maintained on a per-set basis (not on a per-block basis).

To extract the bits from addresses that correspond to the index you will want to use the same bitwise operations suggested in Program 1 to extract bits. If you have a variable that has 1's (in binary) in the positions that correspond to the index (called a bit mask), then once you bitwise-AND an address with that variable, only the index bits remain... you can then right shift by the number of offset bits, to get the index bits into the lowest bit positions. In Program 1, you knew which bit positions needed 1's before running your program (since instruction fields were always in the same bit positions). But in this program, the index and tag bits may be different for each run of your program, since those parameters have a range of possibilities (listed in Table 1). So you will need to compute the bit mask at run time. There are many ways to do this, it is up to you to figure it out. I can give suggestions if you are stuck. **Under no circumstance** should you use the `pow()` function in the `math.h` library. `pow()` returns a floating-point value, and it's always a bad idea to use floating-point to integer typecasting when you could have used all integer operations to begin with.

5 Grading

You should submit your project **source code**, and your **parameters.txt file** with the best configuration for your accesses to Canvas. The project deadline is **Monday, May 2nd, 2022 at 11:00pm**. Late projects can only be accepted with a university-approved excused absence.

The program should compile without errors and run without infinite loops, segmentation faults, or any other crashes. The output format should exactly match the specification, if not, a deduction of 20% will apply. Your program should always read from input files with hard-coded names `parameters.txt` and `accesses.txt`, and the output should always be written to a file with hard-coded name `statistics.txt`. Any other file names will incur a 20% penalty. The values reported by your program must match the expected output for given the input files for full credit.

I will run your program for three different cache configurations and memory access streams. Each of these runs will be weighted equally, and are worth 25% each. If any values are erroneous, points will be deducted based on the scale of the difference between the expected value and the value returned by your program. Obviously, the hit rate does not need to exactly match for a fully correct solution since the result could be rounded, etc.

6 Language, Running Your Program

Your program must be written in C. While grading, I will use `compute.cs.uwlax.edu`. It is your responsibility to ensure your program runs correctly on that machine.

Example input files and their matching expected output file will be posted to Canvas.