Project - Cache Organization and Performance Evaluation

In this assignment, you will become familiar with how caches work and how to evaluate their performance. To achieve these goals, you will first build a cache simulator and validate its correctness. Then you will use your cache simulator to study many different cache organizations and management policies as discussed in lecture and in Chapter 5 of Hennessy & Patterson.

Section 1 will walk you through how to build the cache simulator, and section 2 will specify the performance evaluation studies you will undertake using your simulator.

1 Cache Simulator

In the first part of this assignment, you will build a cache simulator. The type of simulator you will build is known as a *trace-driven* simulator because it takes as input a trace of events, in this case memory references. The trace, which we will provide for you, was acquired on another machine. Once acquired, it can be used to drive simulation studies. In this assignment, the memory reference events specified in the trace(s) we will give you will be used by your simulator to drive the movement of data into and out of the cache, thus simulating its behavior. Trace-driven simulators are very effective for studying caches.

Your cache simulator will be configurable based on arguments given at the command line, and must support the following functionality:

Total cache size
Block size
Unified vs. split I- and D-caches
Associativity
Write back vs. write through
Write allocate vs. write no allocate

In addition to implementing the functionality listed above, your simulator must also collect and report several statistics that will be used to verify the correctness of the simulator, and that will be used for performance evaluation later in the assignment. In particular, your simulator must track:

Number of instruction references
Number of data references
Number of instruction misses
Number of data misses
Number of words fetched from memory
Number of words copied back to memory

1.1 Files

For your project, there are five program files in total, as indicated in table below which lists the file names and a short description of their contents.

File Name	Description
Makefile	Builds the simulator.
main.c	Top-level routines for driving the simulator.
main.h	Header file for main.c.
cache.c	Cache model.
cache.h	Header file for cache.c.

The "Makefile" is a UNIX make file. Try typing **make** in the local directory where you've copied the files. This will build the simulator from the program files that have been provided, and will produce an executable called "sim." Of course, the executable doesn't do very much since the files we have given you are only a template for the simulator. However, you can use this make file to build your simulator as you add functionality. Be sure to update the make file if you have additional source files other than the four program files we've given you.

The four program files, `main.c, main.h, cache.c`, and `cache.h`, contain a template for the simulator written in C. These files contain many useful routines that will save you time (since you don't have to write them yourself).

`main.c` contains the top-level driver for the simulator. It has a front-end routine called `parse_args()` that parses command line arguments to allow configuring the cache model with all the different parameters specified earlier. To see a list of valid command line arguments, try typing **sim -h** (after compiling the template files). Note that your simulator code should interpret the four size parameters, block size, unified cache size, instruction cache size, and data cache size, in units of bytes. `main.c` also contains a top-level "simulator loop," called `play_trace()`, and a routine that parses lines from the trace file, called `read_trace_element()`. For each trace element read, `play_trace()` calls the cache model, via the routine `perform_access()`, to simulate a single memory reference to the cache. While you are free to modify `main.c`, you should be able to complete the assignment without making *any* modifications to this file.

`cache.c` contains the cache model itself. There are three routines in this file which you should be able to use without modification. `set_cache_param()` is the cache model interface to the argument parsing routine in `main.c`. It intercepts all the parameter requests and sets the proper parameter values which have been declared as static globals in `cache.c`. `delete` and `insert` are deletion and insertion routines for a doubly linked list data structure, which we will explain below. `dump_settings()` prints the cache configuration based on the configured parameters, and `print_stats()` prints the statistics that you will gather. In addition to these five routines, there are three template functions which you will have to write. `init_cache()` is called once to build and initialize the cache model data structures. `perform_access()` is called once for each iteration of the simulator loop to simulate a single memory reference to the cache. And `flush()` is called at the very end of the simulation to purge the cache of its contents. Note that the simulation is not finished until all dirty cache lines (if there are any) are flushed out of the cache, and all statistics are updated as a result of such flushes.

`main.h` is self-explanatory. `cache.h` contains several constants for initializing and changing the cache configuration, and contains the data structures used to implement the cache model (we will explain these in the next section). Finally, `cache.h` also contains a macro for computing the base-2 logarithm, called `LOG2`, which should become useful as you build the cache model.

In addition to the five program files, there are also three trace files that you will use to drive your simulator. Their names are "spice.trace," "cc.trace," and "tex.trace." These files are the result of tracing the memory reference behavior of the spice circuit simulator, a C compiler, and the TeX text formatting program, respectively. They represent roughly 1 million memory references each.

The trace files are in ASCII format, so they are in human-readable form. Each line in the trace file represents a single memory reference and contains two numbers: a reference type, which is a number between 0–2, and a memory address. All other text following these two numbers should be ignored by your simulator. The reference number specifies what type of memory reference is being performed with the following encoding:

0	Data load reference
1	Data store reference
2	Instruction load reference

The number following the reference type is the byte address of the memory reference itself. This number is in hexadecimal format, and specifies a 32-bit byte address in the range `0-0xffffffff`, inclusive.

1.2 Building the Cache Model

There are many ways to construct the cache model. You will be graded only on the correctness of the model, so you are completely free to implement the cache model in any fashion you choose. In this section, we give some hints for an implementation that uses the data structures given in the template code.

1.2.1 Incremental Approach

The most important hint is a general software engineering rule of thumb: **build the simulator by incrementally adding functionality**. The biggest mistake you can make is to try to implement the cache functions all at once. Instead, build the very simplest cache model possible, and test it thoroughly before proceeding. Then, add a small piece of functionality, and then test that thoroughly before proceeding. And so on until you've finished the assignment. We recommend the following incremental approach:

1. Build a unified, fixed block size, direct-mapped cache with a write-back write policy and a write allocate allocation policy.

2. Add variable block size functionality.

3. Add variable associativity functionality.

4. Add split organization functionality.

5. Add write through write policy functionality.

6. Add write no-allocate allocation policy functionality.

You can test your cache model at each stage by comparing the results you get from your simulator with the validation numbers which we will provide.

1.2.2 Cache Structures

In `cache.h`, you will find the data structure `cache` which implements most of the cache model:

```
typedef struct cache_ {
  int size;                     /* cache size in words */
  int associativity;            /* cache associativity */
  int n_sets;                   /* number of cache sets */
  unsigned index_mask;          /* mask to find cache index */
  int index_mask_offset;        /* number of zero bits in mask */
  Pcache_line *LRU_head;        /* head of LRU list for each set */
  Pcache_line *LRU_tail;        /* tail of LRU list for each set */
  int *set_contents;            /* number of valid entries in set */
} cache, *Pcache;
```

The first six fields of this structure are cache configuration constants which you should initialize in `init_cache()`. Consult the lectures and text to see how these constants are computed. The remaining three fields, `LRU_head`, `LRU_tail`, and `set_contents`, are the main structures that implement the cache. Let us first consider how to implement the simplest case, a direct-mapped cache. In this case, you only need the `LRU_head` field.

Once you have computed the number of sets in the cache, `n_sets`, you should allocate an array of cache line pointers:

```
my_cache.LRU_head =
   (Pcache_line *)malloc(sizeof(Pcache_line)*my_cache.n_sets);
```

The `LRU_head` array is the data structure that keeps track of all the cache lines in the cache: each element in this array is a pointer to a cache line that occupies that set, or a NULL pointer if there is no valid cache line in the set (initially, all elements in the array should be set to NULL). The cache line itself is kept in the `cache_line` data structure, also declared in `cache.h`:

```
typedef struct cache_line_ {
  unsigned tag;
  int dirty;

  struct cache_line_ *LRU_next;
  struct cache_line_ *LRU_prev;
} cache_line, *Pcache_line;
```

This structure is very simple. The "tag" field should be set to the tag portion of the address cached in the cache line, and the "dirty" field should be set each time the cache line is written. The "dirty" field should be consulted when a cache line is replaced. If the field is set, the cache line must be written back to memory. While you won't be simulating this (since you won't be simulating main memory), this will affect your cache statistics. The remaining two fields are not needed for a direct-mapped cache, and will be discussed later. Notice that in this simulator, you do not need to keep track of any data. We are only simulating the memory reference patterns–we do not care about the data associated with those references.

One final hint: if you have computed the `index_mask` and `index_mask_offset` fields properly, then you should be able to compute the index into the cache for an address, `addr`, in the following way:

```
index = (addr & my_cache.index_mask) >> my_cache.index_mask_offset
```

Then, check the cache line at this index, `my_cache.LRU_head[index]`. If the cache line has a tag that matches the address' tag, you have a cache hit. Otherwise, you have a cache miss[1].

1.2.3 Adding Associativity and LRU Replacement

Once you have built a direct-mapped cache, you can extend it to handle set-associative caches by allowing multiple cache lines to reside in each set. The `cache` and `cache_line` data structures we have provided are designed to handle this by implementing each set as a doubly linked list of cache line data structures. Therefore, if your simulator needs to add a cache line to a set that already contains a cache line, simply insert the cache line into the linked list. Your simulator, however, should never allow the number of cache lines in each linked list to exceed the degree of associativity configured in the cache. To enforce this, allocate an array of integers to the `set_contents` field in the `cache` data structure, one integer per set. Use these integers as counters to count the number of cache lines in each set, and make sure that none of the counters ever exceeds the associativity of the cache.

If you need to insert a cache line into a set that is already at full capacity, then it is necessary to evict one of the cache lines. In the case of a direct-mapped cache, the eviction is easy since there is at most one cache line in every set. When a cache has higher associativity, it becomes necessary to choose a cache line for replacement. In this assignment, you will implement an *LRU replacement*

[1] Of course, if the pointer at the index is NULL, you also have a cache miss.

policy. One way to implement LRU is to keep the linked list in each set sorted by the order in which the cache lines were referenced. This can be done by removing a cache line from the linked list each time you reference it, and inserting it back into the linked list at the head. In this case, you should always evict the cache line at the tail of the list.

To build set-associative caches and to implement the LRU replacement policy described above, you should use the two routines, `delete` and `insert`, provided in the cache.c module. `delete` removes an item from a linked list, and `insert` inserts an item at the head of a linked list. Both routines assume a doubly linked list (which our data structures provide) and take three parameters: a head pointer (passed by reference), a tail pointer (passed by reference), and a pointer to the cache line to be inserted or deleted (passed by value).

One final hint: if you implement a set associative cache whose associativity can be configured, then you have also implemented a fully associative cache. A fully associative cache is simply an N-way set-associative cache with 1 set, and in which N is the total number of cache lines in the cache.

1.3 Validation

In Section 2, you will use your cache simulator to analyze the characteristics and performance of various cache configurations on the traces that we have provided for you. Before doing this, it is important to validate the *accuracy* of your cache simulator.

Compare the output of your cache simulator to the validation statistics provided in Table 1. The table shows the statistics that were obtained on a working cache simulator given various cache configurations on the spice workload trace (the file, “spice.trace” in the class directory). The statistics emitted by your simulator should match identically to the statistics reported in Table 1. Notice that in Table 1, the values in the columns labeled “CS” and “BS” are given in bytes, while the values in the columns labeled “DF” and “CB” are given in words.[2]

2 Performance Evaluation

You should now have a cache simulator that can be configured (for total size, block size, unified versus split, associativity, write-through versus write-back, and write-allocate versus write-no-allocate). Furthermore, this simulator should be verified against the sample cache statistics that we have given you. (If you don’t have such a cache simulator, do not attempt this portion of the assignment until you do). Now you will use your cache simulator to perform studies on the three sample traces (spice, gcc, and TeX) that we have provided.

[2] Remember, we are assuming a 32-bit architecture, so a word contains 4 bytes.

						Instructions		Data		Total	
CS	I- vs D-	BS	Ass	Write	Alloc	Misses	Repl	Misses	Repl	DF	CB
8 K	Split	16	1	WB	WA	24681	24173	8283	7818	131856	12024
16 K	Split	16	1	WB	WA	11514	10560	5839	5051	69412	8008
32 K	Split	16	1	WB	WA	5922	4321	1567	520	29956	3628
64 K	Split	16	1	WB	WA	2619	484	1290	103	15636	3324
8 K	Unified	16	1	WB	WA	36136	35787	21261	21098	229588	37844
8 K	Unified	32	1	WB	WA	26673	26502	19561	19476	369872	69104
8 K	Unified	64	1	WB	WA	23104	23029	20377	20324	695696	136112
32 K	Split	128	1	WB	WA	1964	1726	459	280	77536	7296
8 K	Split	64	2	WB	WA	6590	6462	3160	3032	156000	18880
8 K	Split	64	4	WB	WA	6025	5897	875	747	110400	7296
8 K	Split	64	8	WB	WA	6435	6307	803	675	115808	6656
8 K	Split	64	16	WB	WA	6536	6408	799	671	117360	6624
8 K	Split	64	128	WB	WA	6523	6395	790	662	117008	6576
1 K	Split	64	2	WB	WA	44962	44946	24767	24751	1115664	149200
1 K	Split	64	8	WB	WA	45885	45869	22808	22792	1099088	112480
1 K	Split	64	16	WB	WA	45969	45953	20667	20651	1066176	90416
8 K	Split	16	1	WT	WA	24681	24173	8283	7818	131856	66538
8 K	Split	32	1	WT	WA	15868	15612	7504	7264	186976	66538
8 K	Split	64	2	WT	WA	6590	6462	3160	3032	156000	66538
8 K	Split	16	1	WB	WNA	24681	24173	14904	6688	127304	14643
8 K	Split	32	1	WB	WNA	15868	15612	15098	6421	180200	22033
8 K	Split	64	2	WB	WNA	6590	6462	8638	2726	151104	13624

Table 1: Sample statistics from the spice workload for validation. Column "CS" indicates cache size in bytes. Column "I- vs D-" indicates a split or unified cache. Column "BS" indicates block size in bytes. Column "Ass" indicates associativity. Column "Write" indicates the write policy, either write-back ("WB") or write-through ("WT"). Column "Alloc" indicates the allocation policy, either write-allocate ("WA") or write-no-allocate ("WNA"). The last six columns present the validation statistics. They are the instruction misses, the instruction replacements, the data misses, the data replacements, the total demand fetches in words, and the total copies back in words, respectively.

2.1 Working Set Characterization

In this performance evaluation exercise, you will characterize the working set size of the three sample traces given.

Using your cache simulator, plot the hit rate of the cache as a function of cache size. Start with a cache size of 4 bytes, and increase the size (each time by a factor of 2) until the hit rate remains insensitive to cache size. Use a split cache organization so that you can separately characterize the behavior of instruction and data references (i.e., you will have two plots per sample trace–one for instructions, and one for data). Factor out the effects of conflict misses by *always* using a fully associative cache. Also, always set the block size to 4 bytes, use a write-back cache, and use the write-allocate policy.

Answer the following questions:

1. Explain what this experiment is doing, and how it works. Also, explain the significance of the features in the hit-rate vs. cache size plots.
2. What is the total instruction working set size and data working set size for each of the three sample traces?

2.2 Impact of Block Size

Set your cache simulator for a split I- D-cache organization, each of size 8 K-bytes. Set the associativity to 2, use a write-back cache, and use a write-allocate policy. Plot the hit rate of the cache as a function of block size. Vary the block size between 4 bytes and 4 K-bytes, in powers of 2. Do this for the three traces, and make a separate plot for instruction and data references.

Answer the following questions:

1. Explain why the hit rate vs. block size plot has the shape that it does. In particular, explain the relevance of *spatial locality* to the shape of this curve.
2. What is the optimal block size (consider instruction and data references separately) for each trace?
3. Is the optimal block size for instruction and data references different? What does this tell you about the nature of instruction references versus data references?

2.3 Impact of Associativity

Set your cache simulator for a split I- D-cache organization, each of size 8 K-bytes. Set the block size to 128 bytes, use a write-back cache, and use a write-allocate policy. Plot the hit rate of the cache as a function of associativity. Vary the associativity between 1 and 64, in powers of 2. Do this for the three traces, and make a separate plot for instruction and data references.

Answer the following questions:

1. Explain why the hit rate vs. associativity plot has the shape that it does.
2. Is there a difference between the plots for instruction and data references? What does this tell you about the difference in impact of associativity on instruction versus data references?

2.4 Memory Bandwidth

Using your cache simulator, compare the total memory traffic generated by a write-through versus write-back cache. Use a split I- D-cache organization. Simulate a couple of reasonable cache sizes (such as 8 K-bytes and 16 K-bytes), reasonable block sizes (such as 64 and 128 bytes), and reasonable associativities (such as 2 and 4). For now, use a write-no-allocate policy. Run 4 or 5 different simulations total.

Answer the following questions:

1. Which cache has the smaller memory traffic, the write-through cache or the write-back cache? Why?
2. Is there any scenario under which your answer to the question above would flip? Explain.

Now use the same parameters, but fix the policy to write-back. Perform the same experiment, but this time compare the write-allocate and write-no-allocate policies.

Answer the following questions:

1. Which cache has the smaller memory traffic, the write-allocate or the write-no-allocate cache? Why?
2. Is there any scenario under which your answer to the question above would flip? Explain.