ENEE446: Project 2 - Snoopy Cache Coherence

Spring 2021

Deadline: May 12, 2021, at 11:59pm

At the core of this project is the 4-state MESI snoopy cache coherence protocol, a popular extension to the MSI protocol. Page 388 in the textbook summarizes the main differences between MSI and MESI. Please make sure that you have fully understood the MESI protocol **prior** to starting the project.

In this project, you will first build a uniprocessor cache model. Then you will extend this model to multi-core by adding state bits to cache blocks and performing state transitions in response to memory requests.

1 Cache simulator

In the first part of the project, you will build a *trace-driven* simulator. It takes as input a trace of events (i.e., memory references). The trace, which we will provide you, was acquired on another machine. Once acquired, it can be used to drive simulation. In this project, the memory reference events specified in the trace will be used by your simulator to drive the movement of data in and out of the cache to simulate its behavior. While *trace-driven* simulators are not as accurate as *event-driven* simulators, they are effective for studying caches.

The traces might come from parallel programs rather than sequential programs. Therefore, in addition to memory access information, the traces will also specify the *core* that performed each memory access. The cache model will be configurable based on the arguments given at the command line. The configurable functionality includes:

- Number of caches (cores)
- Total cache size
- Block size

Your simulator must be able to handle 1-8 cores. We assume caches are always *unified* (memory references belonging to data and instructions go to the same cache) and *fully associative*. Besides, they always employ a *write back write-hit policy* and a *write allocate write-miss policy.*

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

- Number of references
- Number of misses
- Number of words fetched into the cache
- Number of words copied back to the memory
- Number of bus transactions (broadcasts)

"Number of references" indicates the number of memory references that a cache is requested. "Number of misses" refers the number of cache misses among memory references. "Number of bus transactions" tracks the number of times a cache broadcasts across the snoopy bus to all the other caches. You should track all of these statistics on a **per core** basis. You will report the first two statistics, "number of references" and "number of misses", per core. For the last three statistics, you will report the total numbers across all of the cores.

2 Files

Please download the project file from "Project2" folder on ELMS. There are five program files in total, as indicated in the table below. The file names and a short description of their contents are listed.

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*.

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

The four program files, *main.c*, *main.h*, *cache.c*, and *cache.h*, contain a template for the simulator written in C. Besides, these files contain many useful routines for you to use.

main.c contains the top-level driver for the simulator. It has a routine called *parse_args()* which 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, please type **sim -h**.) Note that your simulator should interpret the two size parameters, *block size* and *cache size*, in units of **bytes**.

main.c also contains a top-level "simulator loop" called *play_trace()*, and a routine called *read_trace_element()* which parses lines of the trace file. This routine also reads in the core ("pid") performing the memory access. For each read of trace element, *play_trace()* calls the cache model with the routine *perform_access()* to simulate a single memory reference to the cache. While you are free to modify *main.c*, you are able to complete the project **without** making any modifications to this file.

cache.c contains the cache model. There are five routines in this file which you can use without modification. *set_cache_param()* is the cache model interface to the argument parsing routine in *main.c*. It takes 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 will be explained later. *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 in *cache.c* 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. *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 the data structures used to implement the cache model. *cache.h* also contains a macro called *LOG2* for computing the base-2 logarithm, which should be useful as you build the cache model.

In addition to the five program files, there are four trace files that you will use to drive your simulator. Their names are *spice-1c.trace, fft-2c.trace, fft-4c.trace,* and *fft-8c.trace*. These files are the results of tracing the memory reference behaviors of the FFT parallel benchmarks and spice serial benchmark. Their names reflect the benchmark names as well as the numbers of cores that were traced.

These trace files are in ASCII format, so they are human-readable. (Try typing **more fft-2c.trace**.) Each line in the trace file represents a single memory reference. It contains three numbers: The first number specifies the core performing the memory reference. The cores take turns in a round-robin fashion to perform individual memory references. The second number has encodings (i.e., 0 and 1) corresponding to load and store, respectively. The last number represents the byte address of the memory reference. This number is in hexadecimal format, and specifies a 32-bit byte address in the range of 0-0xffffffff.

3 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.

3.1 Incremental approach

We recommend you follow this approach to build your cache model:

1. Build a *unified*, *fixed block size*, *fully associative* uniprocessor cache with a *write back write-hit policy* and a *write allocate write-miss policy*.
    - For a uniprocessor cache, MESI and MSI protocols are identical. For a general write back cache (i.e., single core) with *write allocate write miss policy*, there are only three states for a cache block: *invalid*, "*clean*" (*exclusive/shared*) and "*dirty*" (*modified*).
2. Add *variable block size* functionality.
3. Extend it to multi-core caches with MESI cache coherence policy.

3.2 Cache structures

In *cache.h*, you will find the data structures *cache* and *cache_line* which implement most of the cache model.

```
typedef struct cache_ {
    int id;                 /* core ID */
    int size;               /* cache size in words */
    Pcache_line LRU_head;   /* head of LRU list for the cache */
    Pcache_line LRU_tail;   /* head of LRU list for the cache */
    int cache_contents;     /* number of valid entries in cache */
} cache, *Pcache;
```

Yous should initialize the cache size in *init_cache()*. The "id" field identifies which core the cache belongs to. The remaining three fields, *LRU_head*, *LRU_tail*, and *cache_contents*, are the main structures that implement the cache.

For each cache belonging to a core, you should allocate a cache line pointer.

```
my_cache.LRU_head =
    (Pcache_line )malloc(sizeof(cache_line));
```

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 (since the caches are fully associative, there is only one 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 is kept in the *cache_line* data structure, declared in *cache.h*.

```
    typedef struct cache_line_ {
        unsigned tag;
        int state;
        struct cache_line_ *LRU_next;
        struct cache_line_ *LRU_prev;
    } cache_line, *Pcache_line;
```

The "tag" field should be set to the tag portion of the address cached in the cache line. The "state" field encodes the current state for the associated cache line. In the MESI protocol, there are 4 cache line states: *modified*, *exclusive*, *shared*, and *invalid*. If a cache line is in the state *modified*, the cache line must be written back to memory. While you will not simulate this (since you will not simulate main memory), this will affect your cache statistics. We will simulate only the memory reference patterns - we do not care about the data associated with those references.

When you check for memory references in a cache line, if the cache line has a tag that matches the address' tag, you have a *cache hit*. Otherwise, you have a *cache miss*. If the pointer at the head is *NULL*, you also have a *cache miss*. The *cache.h* file declares the *cache_stat* structure which tracks statistics.

In this project, you must be able to handle up to 8 cores. Since there is one unified cache per core, we have statically declared an array of 8 cache structures in *cache.c*. We have also statically declared an array of 8 cache statistics structures as well. In the *init_cache()* function, you should initialize as many of these as needed (specified by the "num_core" variable in *cache.c*).

After cache initialization, you will enter the main "simulator loop" in *play_trace()*, which will read each trace element and call *perform_access()*. Within *perform_access()*, your simulator should access the cache specified by the "pid" parameter to check if there is a *hit* or *miss* on the address ("addr") within the local cache. This requires traversing the LRU list at the cache, and searching for a tag match. On a *hit*, it requires updating the LRU list to reflect the new ordering of cache blocks within the set. On a *miss*, it requires creating a new cache block, inserting it into the LRU list (at the LRU head), and evicting a block (from the LRU tail) if necessary.

3.3 Filling the cache and LRU replacement

Since the caches are *fully associative*, each cache has multiple cache lines in it. 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. However, your simulator should never allow the number of cache lines in each linked list to exceed the degree of associativity. To enforce this, use the *cache_contents* integer as a counter to count the number of cache lines in each cache, 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 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*. *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 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).

3.4 Cache State Transitions

In a uniprocessor cache, the address check is the only check that determines whether a memory reference *hits* or *misses* in the cache. However, for a multi-core cache, the state must also be compatible with the access. In particular, a load can proceed if the cache state is "*shared*", "*exclusive*", or "*modified*". But a store can only proceed if the cache state is "*modified*". Even if the address check is a *hit*, the cache access may still be a miss if the cache state is not compatible.

The *perform_access()* function should perform the cache state check, too. If the state is not compatible with the reference, you must perform a state transition operation for a MESI protocol (see pg. 388 in the textbook for more details on the protocol). You must implement all of these state transitions such that your simulator supports the MESI protocol correctly. Note that some state transitions require modifying the state of cache blocks in other caches (i.e., those which are not accessed by the current memory reference). For example, a write to a block in "shared" state requires changing the state of all remote sharers from the "*shared*" state to the "*invalid*" state. This requires **searching all other caches for the referenced cache block, and invalidating any which are found**.

4 Cache Statistics

As described in Section 1, your simulator must report (1) number of references for each core; (2) number of misses of each core; (3) number of words fetched into caches; (4) number of words copied back to memory; (5) number of bus transactions.

Please note the following when recording statistics for each cache:

- When tracking the number of misses, you must check both the address and the state. Specifically, a read/write hit happens only on an address match with a non-invalid state.
- The number of words fetched into cache is the aggregation of fetches from memory and fetches from a remote cache, not just the fetches from memory.
- A bus transaction happens when a cache makes a broadcast to the remote caches. You should consider two situations in general: (1) On a local *write hit*, broadcasting an invalidate message is required if the cache block is *shared*. The shared copies in the remote caches are then *invalidated*. (2) On a cache *miss*, regardless read or write, the local cache will broadcast a message to request for the desired cache block and might update the states of the remote caches. You should consider each of these requests as a broadcast, even if the remote caches may have nothing to react with.

5 Validation

Compare the output of your cache simulator to the validation statistics provided in *validate.txt* file. Each section in the text file should be read as follows: the first line is a concatenation of the trace file name (which includes the number of cores), cache size, and block size. The next 5 lines are as follows:

- Number of memory accesses per core (in the order: *cache0*, *cache1* and so on).
- Number of misses per core.
- Total number of words fetched into all caches.
- Total number of broadcasts.
- Total number of words copied back to memory.

6 Submission details

Please submit the following things in a zip file to ELMS.

1. Your modified cache.c and cache.h (and main.c & main.h, if you have modified them)
2. The statistics file of your simulator (please follow the same format as *validate.txt*). Your grade will be determined based on how close your file is to *validate.txt*.
3. A report in PDF that briefly illustrates what you have done