Lab 5: Multicore and Cache Coherence (BONUS)

Assigned: Sat., 28/12; Due: **Fri., 31/1** (11:59 PM)

Instructor: Onur Mutlu
TAs: Mohammed Alser, Rahul Bera, Geraldo Francisco De Oliveira Junior, Can Firtina, Juan Gomez Luna, Jawad Haj-Yahya, Hasan Hassan, Konstantinos Kanellopoulos, Jeremie Kim, Nika Mansouri Ghiasi, Lois Orosa Nogueira, Jisung Park, Minesh Hamenbhai Patel, Abdullah Giray Yaglikci

1. Objective

In this lab, you will extend your simulator from Lab 2 to model a *multicore* system with a *cache-coherent memory hierarchy.* We will describe how the cache coherence protocol works, and how the functional interface to the simulator will change in order to support multiple threads.

2. Multicore: Functional Changes

Before we model the microarchitectural details of a multicore system, we need to extend the *functional interface* of the system so that the user's program can make use of multiple threads. In this lab, we make the following changes to the functional (ISA-level) interface:

- The system has four CPUs (labeled CPU 0 through CPU 3). All CPUs have private architectural state (register files and program counters), but share memory. Thus, if CPU 0 stores a value to address `0x10000000`, and CPU 1 subsequently loads from address `0x10000000`, CPU 1 should receive the value that CPU 0 stored.
- All CPUs are independent except for the shared memory hierarchy (L2 cache, L2 MSHRs, and DRAM subsystem). You should replicate the pipeline, branch predictor, and L1 caches for each CPU.
- Initially, only CPU 0 is running. Thus, if a single-threaded program is run, it should behave exactly the same as if it were run on a previous lab's MIPS implementation.
- To start a thread on another CPU, we define a new *system call* to spawn the thread. When the SYSCALL instruction is executed with \$v0 = 1, 2, or 3, then a new thread is spawned on CPU 1, 2, or 3. To spawn the thread, the program counter of the respective CPU is set to the PC of the instruction following the SYSCALL (PC + 4), and the \$v1 register on the respective CPU is set to 1. The \$v1 register on the CPU that invoked the SYSCALL is set to 0. Hence, from the program's point of view, execution continues after the system call on both the initiating CPU and the newly-started CPU, with the \$v1 register distinguishing which thread is the new thread and which is the original thread.[1] When a new thread is spawned, execution begins in the subsequent cycle. (We will only test cases where the spawn-thread SYSCALL is invoked to spawn a thread onto a CPU that is not currently executing and has no instructions in its pipeline.)
- To help us test programs more easily, we add a new *console output* system call. When SYSCALL is invoked with \$v0 = 11, then the simulator takes the value in \$v1 as a SYSCALL argument, and prints a line such as "OUT (CPU 2): 1234fffe" on standard output.
- The program termination system call (with \$v0 = 10) now halts only the current CPU. Simulation continues until all CPUs are halted.

[1] This is a simplified version of the fork() Unix system call.

3. Simulator Shell Changes

For this lab, you will need to modify the shell. Unlike in previous labs, you are now allowed to modify `shell.c` and `shell.h`. In order to facilitate grading, we instead specify the exact output that your simulator must produce (please ensure that your turned-in simulator does not emit any debug output; any additional output will be construed as an incorrect result).

Your simulator should not print any output on standard output except for the output that we specify below. (This means that you should remove the welcome text and the prompt from the shell.)

We require your simulator to support only the following commands:

- `go`: execute until all CPUs are halted. Output should occur only when console-out system calls are executed.
- `run N`: run for N cycles. As above, only console output should be printed. If execution completes before N cycles, the simulator should halt at that point instead.
- `i X Y`: set register X on CPU 0 to value Y.
- `rdump`: Dump register state for all four CPUs. Your simulator should print a line "CPU n:" followed by lines for "PC: 0x01abcdef", "R0: 0x00000000" through "R31: 0xdeadbeef", "HI: 0x00000000", and "LO: 0x00000000" (with appropriate values substituted). After all four CPU register dumps, there should be a line "Cycles: n". There should be no blank lines.

Note that commands `go`, `run` and `i` work as before. Only `rdump` must be modified to show register state for all CPUs.

When standard input reaches EOF, the simulator should quit without printing any additional output.

4. Cache Coherence at a High Level

When multiple CPUs share the same memory hierarchy, but have their own L1 data and instruction caches, incorrect execution could occur if two or more copies of a given cache block exist, in two CPUs' caches, and one of these blocks is modified.

Consider the case where CPU 0 and CPU 1 both have a copy of Block X in their L1 data caches. Initially, the block was read from an L2 cache that both CPUs share, so both copies of the block contain the same values. If CPU 0 or CPU 1 performs a load instruction to read a value in Block X, correct execution occurs. However, if CPU 0 performs a store instruction that modifies a value in Block X, and CPU 1 subsequently performs a load instruction from Block X, the load instruction must see the new value. Hence, the new value must somehow be propagated to the copy of Block X in CPU 1. This is called the *cache coherence problem.* A set of *rules* that governs how multiple caches interact in order to solve this problem is called a *cache coherence protocol.*

A cache system that supports coherence can be built in multiple ways. For this lab, we will be focusing on one of the most common schemes used today, which is an *invalidation-based cache coherence protocol.* An invalidation-based protocol solves the cache coherence problem by ensuring that as soon as a CPU requests to write to a cache block, that CPU must *invalidate* (remove) the copy of the block in any other CPU's cache that contains the block. The requesting CPU then has the only copy of the cache block, and can make modifications to its contents. Later, when any other CPU attempts to read the block, it will experience a cache miss, and must obtain the new data from the CPU that modified the data. (The exact way in which this interaction happens will become clear below.)

An invalidation-based cache coherence protocol ensures that all CPUs see correct values in their caches by enforcing the following invariant (by invalidating cache blocks when necessary):

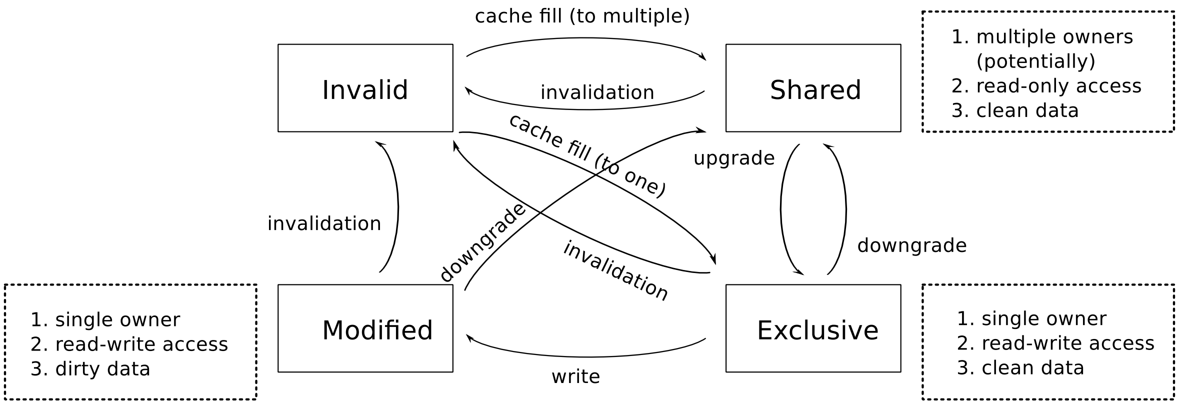
Invalidation-based Protocol Invariant. *Any cache block X is either present in up to* ***one*** *cache in a* ***writable*** *state, or is present in* ***more than one*** *cache in a* ***read-only*** *state.*

In this lab, we will implement a simple version of the *MESI* cache coherence protocol (also called the Illinois protocol [1]). The MESI protocol is an invalidation-based protocol that is named after the four states that a cache block in an L1 cache can have: **M**odified, **E**xclusive, **S**hared, or **I**nvalid. These states are defined as follows:

- **Modified**: When a cache block is in the Modified state, it is dirty with respect to the shared levels of the memory hierarchy. It is also the only copy of the given data among the caches that implement MESI, and so the CPU that owns the cache with the Modified block can make further changes at will.
- **Exclusive**: When a cache block is in the Exclusive state, it is clean with respect to the shared levels of the memory hierarchy, but it is still the only copy of the data among the caches that implement MESI. Hence, if the owning CPU wishes to write to the block, it can change the block's state to Modified without consulting any other CPUs.
- **Shared**: When a cache block is in the Shared state, it is clean with respect to the shared levels of the memory hierarchy. In addition, it is not guaranteed to be the only copy of the cache block among the caches that implement MESI. Hence, the block is *read-only.* If a CPU wishes to read a block in the Shared state, it may do so; however, if it wishes to write, then the block must be transitioned to the Exclusive state (as we will describe below).
- **Invalid**: Finally, the Invalid state represents a cache block that is not present in the cache.

The **Shared** state is sometimes called a multiple-owner state, because multiple caches can contain the same cache block in this state. By the same reasoning, the **Exclusive** and **Modified** states are sometimes called single-owner states, because no other cache can contain the block when a single cache contains the block in one of these states.

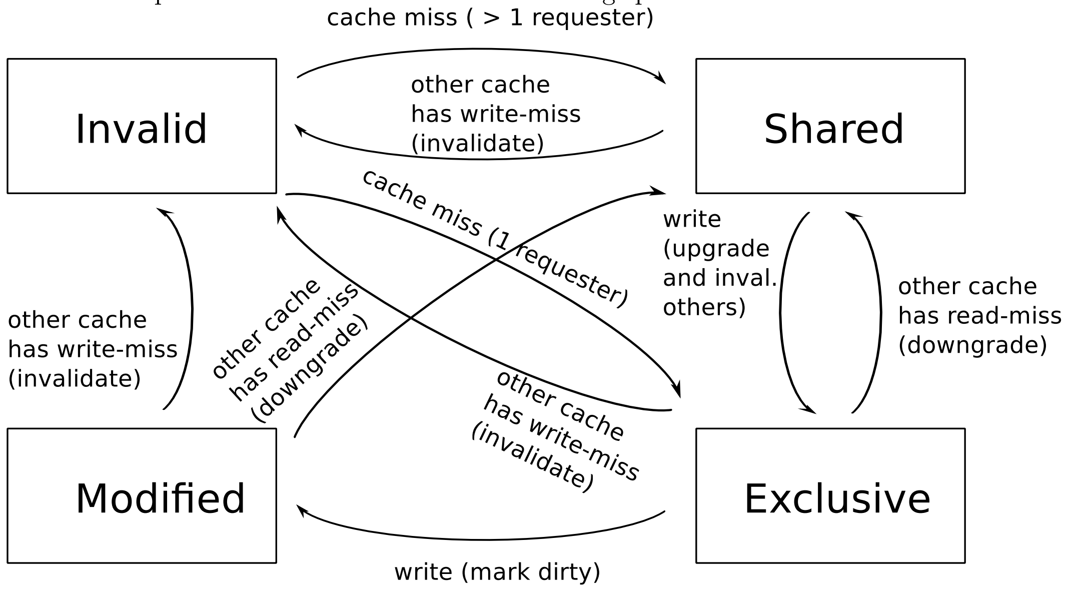
A block transitions between the four states Modified, Exclusive, Shared, and Invalid as shown in the state diagram below. Note that we will define the exact procedures that govern state transitions in the timing specifications below; this state diagram gives a high-level view of how a block transitions between the four states and what these transitions are called. Note that this diagram depicts the states for one block *in one cache*; other caches may hold the block in a different state (e.g., cache block X may be **M**odified in CPU 0's cache and **I**nvalid in CPU 1's cache).



A few types of transitions are common and have specific names:

- A transition from **Shared** to a *single-owner state* (**Exclusive** or **Modified**) is called an *upgrade*, because the transition grants the ability to the owner (the cache which contains the respective block) to *write* to the block.

- A transition from a *single-owner* state (**Exclusive** or **Modified**) to **Shared** is called a *downgrade*, because the transition takes away the owner's right to modify the data (in exchange for the ability to share copies across multiple caches).
- A transition from any of **Modified**, **Exclusive** or **Shared** to **Invalid** is called an *invalidation*. Such a transition allows another cache to enter a single-owner state so that the associated CPU can modify the data.
  An overview of the scenarios in which these transitions occur is shown below. Note that these transitions are *from the point of view of a single cache* unless "other cache" is specified. We will describe each specific case in more detail in the timing specifications below.


cache miss (> 1 requester)
Invalid
other cache has write-miss (invalidate)
Shared
cache miss (1 requester)
write (upgrade and inval. others)
other cache has read-miss (downgrade)
other cache has write-miss (invalidate)
other cache has read-miss (downgrade)
other cache has write-miss (invalidate)
Modified
Exclusive
write (mark dirty)

5. Timing Specifications: Cache-Coherent Memory Hierarchy

Your job is to implement the following behavior in a cycle-accurate manner, starting with your Lab 2 simulator. All specifications from Lab 1 and Lab 2 still apply to this lab, unless the specifications in this lab override the earlier labs' specifications.

- CPUs are simulated in ascending order (CPU 0, CPU 1, CPU 2, CPU 3), one cycle at a time. Hence, cycle 0 of CPU 0 is simulated, then cycle 0 of CPU 1, etc. When two or more CPUs perform actions in the same cycle that must be arbitrated, this ordering breaks the tie unless otherwise specified. After all CPUs are stepped by one cycle, then one cycle of the memory system is simulated.
- Note also that in this lab, multiple events can happen in one cache in one cycle (with a logical ordering prescribed by the rules above). When modeling LRU replacement, be sure that you can handle this case properly.
- Branch recoveries no longer cancel instruction cache miss stalls, for simplicity. (As we will see, cache coherence is complex enough without rolling back the side effects of canceled requests!)
- In order to ensure correctness with respect to system calls, a SYSCALL instruction now *serializes* the pipeline. When a SYSCALL instruction is in the decode stage, it must stall in the decode stage if any instructions are in subsequent stages (execute, memory, writeback). When a SYSCALL is in the execute, memory or writeback stage, any instruction in the decode stage must stall until the SYSCALL leaves the pipeline. The SYSCALL's action takes effect when the SYSCALL instruction reaches the writeback stage.
  To understand why we introduce this pipeline serialization, consider how correctness with re-

spect to program order is maintained in a pipelined CPU. For most resources that can create dependencies between instructions in the pipeline, the CPU works correctly either because the resource is handled in exactly one pipeline stage (e.g., store-to-load dependencies are handled correctly because stores and loads are executed in program order in the memory stage), or because the CPU implements full bypassing (e.g., for register flow dependencies). However, system calls can introduce non-obvious dependencies because they implement "special" actions in the system; for example, when a thread terminates, all instructions following the system call in the pipeline must be annulled. In previous labs, you may have introduced a specific fix for this particular case. But because the system now has a wider array of system calls, the CPU ensures correctness more simply by enforcing the constraint that a SYSCALL executes by itself in the pipeline.

- The L2 cache has the same parameters as in Lab 2. However, it now has a dirty bit per block.
- Both the instruction and data L1 caches have the same parameters as in previous labs, except they are now coherent, implementing the MESI cache coherence protocol described in this handout. To support this, each cache block has a 2-bit MESI status. This status can indicate that a block is **M**odified, **E**xclusive, **S**hared, or **I**nvalid (as described earlier in this handout). A cache block that is present in Modified or Exclusive state results in a hit on access (load or store). A store to a block in the Exclusive state upgrades that block to the Modified state. A cache block that is present in the Shared state results in a hit for any load, but results in an upgrade miss for any store. Any other access is a miss.
- Each L1 cache now has one MSHR which tracks the outstanding request (if any) at that cache. The MSHR has fields to indicate requested block, whether the request is a read (load) or a write (store), and at what cycle the data will be ready.
- On a cache miss (which may occur due to a data upgrade when a write occurs to a block in the **S**hared state, as described previously, or may occur simply when the requested data is not present in the cache), the following procedure is executed by the cache controller in the cycle of the miss. Note that for simplicity, all of this happens instantaneously at the time of the miss (unlike in Lab 2). If the simulator modeled delays between each step, then the coherence protocol would become much more complicated because of these transition states.
  - **Step 1 (Write Exclusion).** If any write to this block is pending at any other L1 cache (as defined by MSHR status), *or* if this new access is a write and a read to this block is pending at any other L1 cache, then this access stalls, and the cache controller restarts from Step 1 in the next cycle. (This rule would not exist in a real cache hierarchy, but we include it here to eliminate many corner cases and simplify your job.)
  - **Step 2 (Check Active MSHRs).** The cache controller scans all valid L2 MSHRs to determine whether another miss is already active to the same block. If this is the case, then this miss simply stalls (the already-active miss will eventually return data and satisfy this cache miss as well). Otherwise, the cache controller continues to Step 3.
  - **Step 3 (Ensure MSHR is Available).** At this point, the request is an L1 miss to a new block for which no other request is pending. First, if no L2 MSHRs are available, this access stalls. The cache controller restarts from Step 1 in the next cycle. L2 MSHRs are only necessary for L2 misses, so we do not allocate the MSHR at this point. However, stalling here eliminates the need to back out of half-updated state later (e.g., after determining that this access is an L2 miss in Step 5) if the cache controller discovers that there is no free L2 MSHR.
  - **Step 4 (Probe Other L1 Caches).** All other L1 caches are probed (assume that every cache has enough ports for an arbitrary number of these probes to occur in a single cycle). If the data is present in any other L1 cache, then the cache controller proceeds to either Step 4a (if the request being serviced is a read) or Step 4b (if the request being serviced is

a write).

  * **Step 4a (Found in Other L1, Read).** If the cache controller finds the requested cache block in any other cache, and if this request is a read, then the following occurs. The cache controller downgrades the state of the block in all other caches in which it is present to the **S**hared state, and updates the LRU order in that cache. This data will be inserted into the L1 cache of the requesting CPU in a **S**hared state. If the downgraded block was previously **M**odified in the L1 cache in which it was found, then a writeback to L2 is performed as follows. If the block is present in the L2 cache, then the block's dirty bit is set and the LRU order is updated (since the block is accessed by the writeback). Otherwise, no writeback occurs. (In a real system, the written-back address would cause an L2 miss, which might cause another eviction in L2; we have omitted this for simplicity.) The cache controller proceeds to Step 4c.
  * **Step 4b (Found in Other L1, Write).** If the cache controller finds the requested cache block in any other cache, and if this request is a write, then the following occurs. The cache controller invalidates the data in any other L1 caches in which it was found. The controller then inserts the data into the L1 cache of the requesting CPU. If the block was in a **M**odified state in the L1 cache in which it was found, then it is inserted in the **M**odified state (the dirty data is thus transferred without a writeback to L2). Otherwise, the block is inserted into the **E**xclusive state, but immediately set to **M**odified by the write. The cache controller proceeds to Step 4c.
  * **Step 4c (L1 Writeback).** When the requested cache block is inserted into the L1 cache of the requesting CPU in either Step 4a or Step 4b, if another cache block is evicted and the evicted cache block's state is **M**odified, then the writeback from L1 to L2 is handled as described in Step 4a above.

  If the cache miss is satisfied in Step 4, then the miss takes 5 cycles to complete.

- **Step 5 (Probe L2 Cache).** If the miss has not been satisfied yet, the cache controller probes the L2 cache. If the data is present in the L2 cache, then the controller updates the LRU order on the L2 cache, and inserts the new block into the L1 cache of the requesting CPU. The block is inserted with **M**odified state if this access is due to a write (store), and **E**xclusive state if this access is due to a read (load). If a block is evicted due to this L1 insertion, then its writeback is handled as in Step 4a above.
  If the cache miss is satisfied in Step 5, then the miss takes 15 cycles to complete.
- **Step 6 (Go to Memory).** The controller has now determined that the request is an L2 miss and must load data from main memory. At this point, an L2 MSHR is allocated, and the memory request handling proceeds as in Lab 2 (i.e., the memory request is placed in the memory request queue after five cycles of transit delay).

• Whenever a memory request completes and the data is ready at the L2 controller (5 cycles after the request is complete, as specified in Lab 4), the following cache fill sequence occurs.
  - **Step 1 (L2 Insertion).** The cache block associated with the request's address is inserted into the L2 cache. No writeback handling is performed for any data evicted from the L2 cache by this insertion.
  - **Step 2 (L1 Insertion).** The cache controller scans all L1 MSHRs. For any L1 MSHR which has a pending request for the cache block returning from memory, the new block is inserted into the associated L1 cache. The new block will have **M**odified state if this pending request is a write (store), **E**xclusive state if this pending request is a read (load) and is the only pending request for the address of the cache block being inserted, and **S**hared state otherwise. If a block is evicted from this L1 by the new block's insertion, then that evicted block's writeback is handled as described in Step 4a above.
  - **Step 3 (Free MSHR).** The cache controller frees the L2 MSHR associated with this

request.

– **Step 4 (Complete L1 Request).** Each L1 cache into which the cache block was inserted will unstall 5 cycles after the insertion is performed. Thus, execution can continue with the value from the cache miss 10 cycles after the memory request completes. When the L1 cache unstalls, the L1 MSHR is released.

6. Getting Started

There is no lab handout directory for this lab. Instead, you should begin with your Lab 2 simulator (ensuring that it is correct), and extend it to model the system described above.

You may modify any file in the simulator source (and/or add new files). Since we have specified the external program interface for this lab, there are no restrictions on how you may modify any source file.

7. Submission

Use the corresponding assignment in Moodle (https://moodle-app2.let.ethz.ch/). You should submit all the files needed to compile and simulate your code in a single tarball (with the name lab5_*YourSurname*_*YourName*.tar.gz). Please include comments to explain what you have done in the simulator code.

8. Grading

As in previous labs, we will grade this lab objectively based on a set of tests which are not released beforehand. Your simulator must be functionally correct as if it were run on a single thread. Furthermore, your simulator should model timing accurately with respect to the cycle-by-cycle specifications in this lab document. We will, at our discretion, give some partial credit for results that are "close," with a scale to be determined. However, you should aim to match cycle counts exactly.

9. Reference Timing Results

We are specifying several reference timing results here. Note that we may grade your submission using many other tests, for which your timing should also match. These timing results are given only in order to allow you to debug your implementation and verify that you are correctly modeling timing.

This set of tests is not meant to be exhaustive. You should write other tests that exercise the various aspects of the system, and examine cycle-by-cycle behavior, to make sure that you are modeling everything as specified in this lab document.

We have provided an updated set of tests, which includes all of the tests listed below, in the Materials link (at https://safari.ethz.ch/architecture/fall2019/doku.php?id=labs).

Benchmark Name	Cycles	Expected Output
thread_tests/parmatmult	8546075	`OUT (CPU 0): 7f0c0000`
thread_tests/test1	833	`OUT (CPU 0): 00000000` `OUT (CPU 1): 00000001` `OUT (CPU 2): 00000002` `OUT (CPU 3): 00000003`
thread_tests/test2	1463	`OUT (CPU 0): 00000001` `OUT (CPU 1): 00000001` `OUT (CPU 2): 00000001` `OUT (CPU 3): 00000001`
long_tests/primes	2448549	(nothing)
long_tests/fibonacci	5243441	(nothing)
cache_tests/test1	1751621	(nothing)

References

[1] M. Papamarcos and J. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *ISCA-11*, 1984.