In this assignment, you will implement a writeback cache from scratch and include it to simulate real programs in `gem5`. This will allow you to evaluate your cache under different workloads and with different configurations. There will be a few iterations of your implementation, where each step is designed to build on the previous step. You will only submit the **code** associated with the final iteration to Gradescope, but you will also submit a **written component** that you will answer as you go.

`gem5` is a complicated tool. Using it or extending it straight out of the box can be tricky, so it will be useful to examine the gem5 cheat sheet and/or to attend/revist Lab 4 in order to familiarize yourself with the tool. Feel free to come by office hours for further clarifications, but note we may refer you to the available resources if they believe the answer can be found there!

Submission

You are asked to submit the programming component and the write up to Gradescope. The programming component has an autograder with test cases to help facilitate your development process, but not all graded test cases are given. You are also expected to test your code independently!

To submit your code, create a subdirectory called "submission" in the base stencil directory with a copy of your `micro-cache.{hh, cc}` files and your `full-cache.{hh, cc}` files. Zip this subdirectory and submit it to the Gradescope.

Each section of the assignment has a written component, which you should include in a single "Written Response" document. The purpose of these exercises is to encourage you to engage with the nuance of the programming task, so respond in as much detail as you feel is necessary. As a guiding principle, a strong response can be made in about one paragraph.

Testing

For each part of this assignment, you can test your cache against each of the sample test files in the `configs/example/hw2/` directory! Check the output directory (`m5out` by default) for your `MicroCache` and `FullCache` stats and validate that they make sense. If your simulation crashes or if your simulations hangs (i.e., runs for more than 5 minutes without finishing) be sure to check out the Common Errors section of the `gem5` cheat sheet. Some (but not all) of the autograder test cases on Gradescope have been made public for you.

Collaboration Policy

Collaboration is highly encouraged on this assignment! All submitted code must be your own. Any AI use must conform to the course AI policy on the syllabus.

Getting Started

Installing the Source

To ensure that we are all using the same version of the source, download the source code from the course Github page.

Building

The configuration scripts to run this assignment assume that you are using a build of `gem5` that uses X86 as the simulated ISA so that we can simulate real programs (and so that you can write

your own programs to simulate)! This means that we need to rebuild gem5 for this ISA. You can build the source by running `scons build/X86/gem5.debug -j4 --linker=gold`. Note, in this case we specify `X86` as the ISA field.

1 Protocol Description

Before starting to code, read through this assignment handout and make sure you understand the general role of the cache you're being asked to implement. To guide your work, sketch a finite state machine controller for the cache protocol that deals with CPU requests (i.e., you don't have to describe the coherence protocols). The state machine is driven by the `handleRequest` and `handleResponse` callbacks (i.e. your transitions should happen based on the information from these functions). You do not have to turn this in, but you may be asked for it to check your conceptual understanding when you ask questions. Your state machine should be complete enough that the pseudocode for the assignment should follow. It doesn't need to be wholly exhaustive, but you should enter the state machine at a starting state, and consider the following questions:

- What sort of signals does the cache need to keep track of?
- Is the request a read? Write?
- Does the request hit in the cache given the current state?
- What should be sent to memory and when? What should happen on the response?

Why a FSM? A cache is a complicated mechanism that may take several steps for a specific action. A finite state machine allows you to express the order in which these steps happen. For example, caches might wait for a response from memory before being able to do some next step, and this sort of behavior is straightforward to express in a "wait state" of an FSM.

2 One-Block Cache

To understand the degree to which the smallest representative units of locality could impact performance, early simulation-based studies in computer architecture studied the impact of using a cache composed of a single 64-byte cache block. While somewhat academic, this study had significant impact on how we understand cache hierarchy construction today. However, this study was performed well in advance of the more capable architecture simulators used today, like `gem5`.

Update 10/7: Some of the functions were incorrectly named and have been updated.

Your Task

You will implement the one-block cache. Doing so will require translation of your high level FSM from the previous section into simulator source. When done correctly, the simulator will boot and run the workload to its completion.

In general, the one-block cache is implemented through the logic of the `handleRequest` and `handleResponse` functions. `handleRequest` is called when the component on the processor-side of the memory hierarchy from the one-block cache makes a request to fetch or set some data. You may assume that CPU requests will not cross blocks (i.e, you will not need to fetch multiple blocks to service a single request). If the cache cannot satisfy the request given its current state, it prompts the memory-side component for a request that carries the requisite information to process the processor-side request. Be sure to read through the `src/mem/cache/micro-cache.hh`, in particular the port functions, to get a sense of how this flow works.

The aim of this part of the assignment is to familiarize you with the `gem5` environment, and evaluate the impact of having a single block cache in terms of performance with different benchmarks.

Programming. You are asked to complete the `handleRequest` and `handleResponse` functions in the `src/mem/cache/micro-cache.cc` file. In the `handleRequest` function, you will implement a hit routine, a miss routine, and the logic to determine whether the current request hits or misses. On the other hand `handleResponse` should perform the post-memory access routine. That is, the pkt in the function arguments is the response of a memory request (that you will create), and not the CPU request. Remember, your response may need to replace some existing data in the cache, so be sure to handle this case appropriately.

`handleRequest` should increment the `stats.hits` or `stats.misses` on each access in order for your assignment to be graded!

Submission Guidelines. To grade this section, the autograder checks for the `micro-cache.{hh, cc}` files in your submission. Be sure they are implemented appropriately and included in your submission!

Hints and suggestions. In the `handleRequest` function, your implementation should ensure that only one memory request can occur at a time by setting the "`blocked`" flag. That is, you do not need to enforce memory consistency or implement any synchronization in your cache. However, you may need to ensure that the `blocked` flag is appropriately set to false if the protocol demands it.

Overall, `gem5` uses a lot of objects that use a lot of functions – many of which are not relevant to this assignment. Full listings of packet functions can be found in the `src/mem/packet.hh` header, or in the formal documentation. For your convenience, here are some functions that you may find useful:

- `Addr Packet::getAddr()` (e.g., `pkt->getAddr()`): returns the address of the current packet. Note, `Addr` is an expansion of `uint64_t`
- `unsigned Packet::getSize()` (e.g., `pkt->getSize()`): returns the size of the current packet
- `bool Packet::isWrite()` (e.g., `pkt->isWrite()`): returns true when the current packet has data to write to the relevant block. (also, `pkt->hasData()` should return true here)
- `bool Packet::isRead()` (e.g., `pkt->isRead()`): returns true when the packet should read data from the relevant block
- `T* Packet::getConstPtr()*` (e.g., `pkt->getConstPtr<uint8_t *>()`): returns a pointer of type `T` to the data stored in the packet
- `void Packet::writeData(uint8_t *p)` (e.g., `pkt->writeData(p)`): sets the data stored in the packet to the pointer `p`
- `void Packet::setDataFromBlock(const uint8_t *p, int blk_size)` (e.g., `pkt->setDataFromBlock(p, BLOCK_SIZE)`): sets the data stored at pointer `p` to the packet
- `bool Packet::needsResponse()` (e.g., `pkt->needsResponse()`): returns true when the current packet should be returned to the CPU with the relevant fields updated (data should be set to the packet if it is a read, etc.)
- `void Packet::makeResponse()` (e.g., `pkt->makeResponse()`): updates the packet's request state to the appropriate response

Furthermore, there are a few functions and fields to help you in `src/mem/cache/micro-cache.hh`:

- `struct Block`: the starter scaffolding has provided a block structure for you. Your solution should create the appropriate number of block instances, and maintain them throughout the execution
- `void makePacket(Addr addr, MemCmd::Cmd cmd)`: given the appropriate input will create a Packet for the provided address with the provided memory command
- `void sendToMem(PacketPtr to_mem)`: takes the packet set as the current `to_mem` packet, and calls the appropriate port function to send to memory
- `void sendToProc(PacketPtr to_cpu)`: takes the packet set as the current `pkt` packet, and calls the appropriate port function to send back to the processor

Written Response

In a section clearly labeled "One-Block Cache", respond to the following prompts.

1. Compare the methodology of the one-block cache implementation to the study that it was based on. What is a limitation of the prior study that your implementation does not suffer from?
2. What is the memory consistency model for the one-block cache? How is the memory consistency model enforced?
3. What assumptions does your cache make about requests coming from the processor side? Is this assumption safer if your cache is an L1 or L2 cache? Why?

3 Fully Associative Cache

In the previous section, you implemented a scheme to keep make recently accessed data accessible to the processor. By doing so, mapping data to the appropriate cache block was trivial: if data is to be cached then it goes in the single block. If the cache has more capacity than a single block, however, a decision must be made about in which cache block the data should be stored and, if no block is available, how to replace that block.

There are multiple ways to map memory addresses to blocks. In this section you will implement a "fully associative" cache. This means that any memory address can go in any of the blocks. In other words, there is only one set for the entirety of the cache.

There are several ways to choose the block to evict; in this part of the assignment you will implement the least recently used (LRU) eviction policy. That is, when scanning for a place to place the new data, you should track which block is the one that has been used the least recently before following the writeback procedure.

Your Task

In this section, you will add some capacity to the cache from the previous section so that it can store more than 64 bytes at a time. Your implementation should use the `p->size` parameter in the constructor of the `FullCache`, and allocate the appropriate amount of space for each of the blocks. You may assume that all sizes will be provided in kilobytes with some value followed by `kB` without a space (e.g., `4kB`).

The key change from your single block cache is that your cache will now have multiple blocks. This will result in two key changes – for one, the `handleRequest` function will now have to search across several blocks in order to determine whether or not to follow the hit or miss procedure; and the `handleResponse` function now needs to choose which block to fill the new data. In this case, we would like to: ① fill an empty block if one exists, or ② choose a block in the cache to evict in order to make space for this new data to fill.

You will choose an eviction target by implementing the LRU eviction policy. Once a target to evict is selected, you will write this data back to lower levels of the cache hierarchy as done in Part 1.

Programming. Your task is to extend your implementation of the `handleRequest` and `handleResponse` functions from Part 1 in the `src/mem/cache/micro-cache.cc` file. Your implementation must be general to arbitrarily input sizes, which are passed to the `FullCache` constructor. The parameter `p->size` is provided as a member of the `FullCacheParams` as part of the starter scaffolding. Note, the parameters are declared in the `FullCache` Python class declaration in the `src/mem/cache/hw2/CacheAssignment.py` file.

As in Part 1, `handleRequest` should increment the `stats.hits` or `stats.misses` on each access in order for your assignment to be graded!

Submission Guidelines. To grade this section, the autograder checks for the `full-cache.{hh, cc}` files in your submission. They do not need to be fully functional for all components, but they should at least implement the fully associative functionality to pass the cases for this section. Be sure they are implemented appropriately and included in your submission!

Hints and suggestions. This part of the assignment asks you to build up a replacement policy in your emulated cache. To do this, you should modify what is stored with each cache block in the `micro-cache.hh` file, and update this timestamp with the current cycle time each time the block is accessed. You may find the simulator built-in `curTick()` function helpful to be able to do so.

You are also asked to construct a cache from a size. To determine the number of blocks to construct from that size, you may find the C++ `std::string` library helpful to perform any necessary string parsing. You may also find it helpful to modify the fields constructed by the constructor.

Update 10/7: Note the differences between the stencil in the `full-cache.hh` versus the `micro-cache.hh` in the `CpuSidePort::recvTimingReq` functions. In particular, the `MicroCache` has scaffolded the blocking protocol for you. In the `FullCache`, handling the coordination of parallelism is your task! For this section, you can implement a simple scheme in which a single packet is handled at a time by the `FullCache`, but later sections will ask you to extend this protocol to allow for more parallelism.

Written Response

In a section clearly labeled "Fully Associative Cache", respond to the following prompts.

1. Consider your implemented mechanism to find a cache block to fill a memory response in your implementation. It likely used some iteration across the blocks in the set to find the fill target. This implies an $O(n)$ search procedure. Given this, comment on the relationship is between simulated execution time and the runtime of the simulator.

2. Least recently used is far from the only possible replacement policy. In fact, there are several other replacement policies, including: least frequently used, time-aware least-recently used, most-recently used, etc (to read more, see here). From this list, choose one alternative replacement policy to LRU. Suppose you have 4 block fully associative cache. Come up with a memory trace (a sequence of address and operation types) for which the LRU replacement policy is sub-optimal compared to your selected alternative replacement policies. Then provide a different memory trace for which the LRU policy is optimal compared to that same replacement policy.

3. Besides having an increased complexity over the single-block cache, our fully-associative cache also has an increased capacity. If we wanted to do an apples-to-apples comparison of our single-block cache and our fully-associative cache, we could increase the block size of our single-block cache to the total capacity of the fully associative cache. While the hardware would certainly be less complex, what disadvantages would this giant-block cache have over a fully-associative cache of the same capacity? Consider how this cache would fit in to the entire hierarchy and how it would respond to different patterns of memory access.

4 Set Associative Cache

Suppose you have a large cache with many blocks (i.e., 1MB). Adding hardware to try to match the tag of every block in the cache quickly becomes complicated and unreasonable. What can be done? One possibility is to reduce the possible locations to which an address can be mapped (like a hash map). That is, there are a small number of well defined *set* of cache blocks to which an address may be mapped.

In a set-associative cache, each address can be mapped to any of the possible blocks in a set. If there are n blocks per set, these caches are sometimes called n-way set associative caches. This entails partitioning the blocks in the cache into sets. A consequence of this partitioning is that the tasks of searching for an address and choosing an eviction target are now reduced to scanning n blocks.

If a 2-way set associative cache has 64 blocks (i.e., a 4kB cache), then there are 32 possible sets that a block can be mapped to. If we are going to search for an address in the cache, we will not look in each set for that block. Instead, addresses are assigned to cache sets based on the tag bits of the address (i.e., the highest order bits).

Your Task

In this section, you will modify your fully associative cache implementation of the `FullCache` to now implement an n-way set associative cache as specified by the input parameter `p->assoc` in the class constructor. This will require a modification to the cache lookup procedure and cache replacement behavior, and your implementation must do so in order to achieve correct behavior.

Programming. Your task is to extend your implementation of the `handleRequest` and `handleResponse` functions from Parts 1 and 2 in the `src/mem/cache/full-cache.cc` file. Your implementation must be general to arbitrary input associativity, which are passed to the `FullCache` constructor. The parameter `p->assoc` is provided as a member of the `FullCacheParams` as part of the starter scaffolding.

As in Parts 1 and 2, `handleRequest` should increment the `stats.hits` or `stats.misses` on each access in order for your assignment to be graded!

Submission Guidelines. To grade this section, the autograder checks for the `full-cache.{hh, cc}` files in your submission. They do not need to be fully functional for all components, but they should at least implement the set associative functionality to pass the cases for this section. Be sure they are implemented appropriately and included in your submission!

Hints and suggestions. Change log here as issues come up!

Written Response

In a section clearly labeled "Set Associative Cache", respond to the following prompts.

1. Think about the latencies of your simulated caches. Does it make sense to have equivalent latencies for different associativities? Why or why not?

2. What is the implication of having an associativity of 1 (called a direct-mapped cache)? If your application uses 1MB of data, and you have a 1MB 1-way set associative cache, would you expect the hit rate to be higher or lower than with an 8-way set associative cache? Would your performance be better? Justify your hypothesis.

5 Cache Parallelism

With instruction- and thread-level parallelism, we may have multiple memory operations in parallel. In this section, you'll consider the implications of multiple requests for the cache in parallel – both for different addresses/sets and for the same set!

Notice how in your implementation we are still only allowing a single memory access into the cache at a time. The starter scaffolding has been implemented in this way to aid the simplicity of coordinating multiple memory operations, which allows for your simulated hardware to maintain clean memory consistency. To highlight why this consistency is important, suppose "hand-break" to handle multiple requests in the same cycle was removed. All of a sudden, the cache logic would need to handle a case where there are multiple memory responses that needed to choose an eviction target. In your current implementation, it is possible that both responses choose the same block (implying that the second response would overwrite the first without keeping both values in the cache at the same time), it is possible that they choose different targets but don't conform to the LRU policy, or some other undefined behavior could occur!

At the same time, your current implementation only handles a single memory operation at a time, which is highly inefficient. You can improve the efficiency of your implementation by adding *parallelism* to the cache logic. In such an approach, concurrent memory accesses may be safely handled to increase the parallelism of the memory system.

Your Task

In this section, you will modify your set associative cache implementation of the `FullCache` to now implement cross-set parallelism. This will require a modification to the blocking procedure when we try to access certain blocks, and your implementation must do so to achieve correct behavior.

Programming and Submission. This section of the development is not evaluated by the autograder. Instead, I will be looking at your implementation to see that the fine-grained blocking mechanism or non-blocking mechanism is implemented appropriately. Be sure that your submission to Gradescope implements this functionality for full credit on this section and/or has comments in the appropriate places explaining what would need to change for partial credit on this section!

Written Response

In a section clearly labeled "Cache Parallelism", respond to the following prompt.

1. What you have described in the previous part is a lightweight miss status holding register (MSHR) which is a hardware primitive used in real caches (blocking and non-blocking) in order to maintain the global memory consistency model handling parallel misses for similar and dissimilar addresses in memory. What is the relationship between maximum achievable parallelism in a cache (i.e., the bandwidth of a cache) and the number of MSHR registers that the cache can support? It may help to justify your answer with an example.