The goal of this lab is to gear-up to the cache assignment by building into the gem5 back-end. In doing so, you will build a dummy "do-nothing cache" in the cache assignment scaffolding that forwards traffic from the processor side to the memory side and sends memory responses back to the processor. This lab assumes that you are working from the cache assignment stencil to ensure that versions are consistent between remote resources from gem5 and the expected run targets.

Begin the lab by building the repository by running `scons build/X86/gem5.debug -j<nproc>`. Note this step will take a while (+/- an hour is not uncommon, but verify that the output is printing the compiled files as the compilation process executes). If your build fails due to being killed, it is likely an "out of memory" error. You can address this by running with "`--linker=gold`" or other linkers described here.

# 1 File Overview

## 1.1 Configurations

This scaffolding has two main directories for you to be aware of: ① the configuration files that you will use to run and test your code, and ② the back-end source where you will define your caches.

To find the configuration files, look in the directory `configs/example/hw2/`. In this directory, you will find five files that declare the backend source files used for the simulation: notably `{hello-world, memtest}_{full, micro}cache.py` and `traffic-generator_test.py`. In the `hello-world` and `memtest` configurations, the simulator is configured to use a cache hierarchy that uses the pre-built gem5 cache objects for the instruction and data caches, and uses your custom cache as an L2 cache to run the programs defined in the `hw2-tests` directory (the source is visible to you in `src/hw2-tests`). In particular, note that there is a line in the `hello-world_microcache.py` configuration file that defines the cache hierarchy to be:

```
cache_hierarchy = PrivateL1PrivateL2MicroCacheHierarchy(
    l1d_size="32KiB",
    l1i_size="16KiB",
)
```

The hierarchy is configured without explicitly defining the `l2_size` because the `MicroCache` does not vary by size. On the other hand, the `hello-world_fullcache.py` configuration specifies the `l2_size` and `l2_assoc` explicitly as implementing these features will be part of your task!

**Traffic Generator.** The `traffic-generator` does not run a program, but instead creates custom dummy traffic to the memory system as if the processor were running a program with a well defined behavior. Look through this file to see that this makes sense.

## 1.2 Back-End Source

In the back-end, you are also provided with scaffolding files in the `src/mem/cache/hw2` directory. Notably, there exists a header and source file for the `MicroCache` and `FullCache` classes in the `{micro,full}-cache.{hh,cc}`. In this lab, we will be adding to the `MicroCache` class such that it forwards traffic to memory rather than explicitly maintaining state. In the Cache Assignment, you are asked to write logic that builds on top of these techniques in both the `MicroCache` and `FullCache` to implement the desired functionality.

The majority of the scaffolding provided to you is in the header files of these classes. The header files define several fields, including the processor- and memory-side port classes, instances

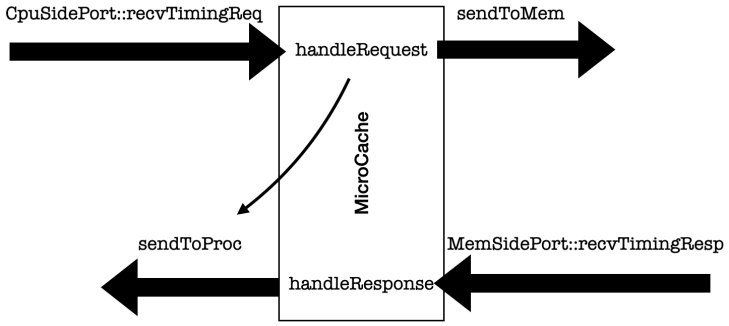


Figure 1: Designed flow control of cache assignment architecture. It's up to you to determine what triggers each arrow to execute and what should happen in the `handleRequest` and `handleResponse` functions!

of these ports that are accessible via the `{cpu,mem}_port` attributes of the class, along with several helper functions:

- `sendToProc(PacketPtr)` is a wrapper function that takes a packet as input and makes the appropriate call to the class port such that the data is appropriately transmitted to the processor;
- `sendToMem(PacketPtr)` is a wrapper function that takes a packet as input and makes the appropriate call to the class port such that the data is appropriately transmitted to memory;
- `makePacket(Addr, MemCmd` is a function that will help you create new packets to potentially forward to the processor- or memory-side components.

Beyond these functions, which you may find helpful, the class defines a `CacheBlock` struct of which the `MicroCache` has one instance of (`blk`). Your job in the Cache Assignment is to implement the logic in the `handleRequest` and `handleResponse` functions that implement the desired cache behaviors to manipulate this field upon receiving a request from the processor or a response from memory.

## 2 Designing a Do-Nothing Cache

Suppose we had a cache that always missed. What would this look like? Ultimately, this is the goal of this section of the lab. In particular, we want to build a cache that receives requests from the processor and, because the "do-nothing cache" does not maintain state, forwards these requests to the next lowest level of the memory hierarchy to handle.

As you go through this lab, start thinking about how the logic changes if the caches *does* maintain state, and the ways that your implementation will change as this cache is extended from a do-nothing cache to a full `MicroCache`.

From Fig. 1, we can see that information comes to the cache from the processor via the `handleRequest` function. Therefore, if we want to implement a "do-nothing" cache, we need to implement the behavior of having some information flow from `handleRequest` to `sendToMem`.

Let's start extending our scaffolding to implement this behavior! In particular, if we want to take the packet received as input and have the next level of the memory hierarchy handle its state, we need to call `sendToMem` with this packet as input. Where does it make sense to implement this behavior? Well, it seems unlikely that this behavior should be characterized as a hit, so let's add it to the miss side of the branch!

```
void
MicroCache::handleRequest(PacketPtr pkt)
{
    // TODO: complete this function!
    bool hit = false;

    if (hit) {
        stats.hits++;

    } else {
        stats.misses++;
        sendToMem(pkt); // add this call here!
    }
}
```

We can recompile our simulator to verify that our changes compile and are functional. Do so by rerunning your build command (i.e., `scons build/X86/gem5.debug -j<nproc> --linker=gold`). After the build completes, we can test the "do-nothing" implementation of the `MicroCache` against the `hello-world_microcache.py` configuration by running `./build/X86/gem5.debug configs/example/hw2/hello-world_microcache.py`. Does the program work? How can you tell? (see next page for answer)

Unfortunately, our code didn't execute to completion. We can tell because we did not get a clean exit in the output (i.e., `Exiting @ tick <n> because exiting with last active thread context.`). This means we'll have to debug what went wrong!

## 2.1 Debugging gem5

Because our code didn't execute correctly, it is worth running it through the debugger to examine its intermediate state and deduce what went wrong. We can do so by launching `gdb`:

```
gdb --args ./build/X86/gem5.debug configs/example/hw2/hello-world_microcache.py
```

Our program doesn't crash, so it's worth trying to deduce how many times through our new code the debugger gets to execute. A trick to do this is to set a breakpoint at the beginning of the execution, and ignore the breakpoint $n$ times for some large number $n$. Then, we can rerun the program from the beginning and ignore the breakpoint $n-1$ times to get to the last instance where this function executed.

```
(gdb) b micro-cache.cc:64
Breakpoint 1 at 0x29062ec: file src/mem/cache/hw2/micro-cache.cc, line 65.
(gdb) ignore 1 100000
Will ignore next 100000 crossings of breakpoint 1.
(gdb) r
Starting program: /home/stum2025/cache-assignment-stencil/build/X86/gem5.debug configs/
    example/hw2/hello-world_microcache.py
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
gem5 Simulator System.  https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 24.1.0.3
gem5 executing on itbdcv-lnx04p.campus.pomona.edu, pid 53803
command line: /home/stum2025/cache-assignment-stencil/build/X86/gem5.debug configs/
    example/hw2/hello-world_microcache.py

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
src/mem/dram_interface.cc:690: warn: DRAM device capacity (8192 Mbytes) does not match
    the address range assigned (32 Mbytes)
src/base/statistics.hh:279: warn: One of the stats is a legacy stat. Legacy stat is a
    stat that does not belong to any statistics::Group. Legacy stat is deprecated.
src/mem/coherent_xbar.cc:140: warn: CoherentXBar board.cache_hierarchy.membus has no
    snooping ports attached!
board.remote_gdb: Listening for connections on port 7000
src/sim/simulate.cc:199: info: Entering event queue @ 0.  Starting simulation...
^C
Program received signal SIGINT, Interrupt.
info bstd::_Vector_base<double, std::allocator<double> >::_Vector_base (
    this=0x7fffffffb760, __n=8, __a=...)
    at /opt/rh/gcc-toolset-11/root/usr/include/c++/11/bits/stl_vector.h:305
305           { _M_create_storage(__n); }
Missing separate debuginfos, use: dnf debuginfo-install bzip2-libs-1.0.6-28.el8_10.
    x86_64 xz-libs-5.2.4-4.el8_6.x86_64
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000029062ec in gem5::memory::MicroCache::
    handleRequest(gem5::Packet*) at src/mem/cache/hw2/micro-cache.cc:65
        breakpoint already hit 1 time
        ignore next 99999 hits
```

In this case, our $n = 1$! So let's ignore our breakpoint $n-1$ times to get to the last time this function executes during the program execution.

```
(gdb) info b
Num     Type           Disp Enb Address            What
1       breakpoint     keep y   0x00000000029062ec in gem5::memory::MicroCache::
    handleRequest(gem5::Packet*) at src/mem/cache/hw2/micro-cache.cc:65
        breakpoint already hit 1 time
        ignore next 99999 hits
(gdb) ignore 1 0
Will stop next time breakpoint 1 is reached.
(gdb) run
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/stum2025/cache-assignment-stencil/build/X86/gem5.debug configs/
    example/hw2/hello-world_microcache.py
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib64/libthread_db.so.1".
gem5 Simulator System.  https://www.gem5.org
gem5 is copyrighted software; use the --copyright option for details.

gem5 version 24.1.0.3
gem5 executing on itbdcv-lnx04p.campus.pomona.edu, pid 55804
command line: /home/stum2025/cache-assignment-stencil/build/X86/gem5.debug configs/
    example/hw2/hello-world_microcache.py

Global frequency set at 1000000000000 ticks per second
warn: No dot file generated. Please install pydot to generate the dot file and pdf.
src/mem/dram_interface.cc:690: warn: DRAM device capacity (8192 Mbytes) does not match
    the address range assigned (32 Mbytes)
src/base/statistics.hh:279: warn: One of the stats is a legacy stat. Legacy stat is a
    stat that does not belong to any statistics::Group. Legacy stat is deprecated.
src/mem/coherent_xbar.cc:140: warn: CoherentXBar board.cache_hierarchy.membus has no
    snooping ports attached!
board.remote_gdb: Listening for connections on port 7000
src/sim/simulate.cc:199: info: Entering event queue @ 0.  Starting simulation...

Breakpoint 1, gem5::memory::MicroCache::handleRequest (this=0x494dff0, pkt=
    0x4e21d60) at src/mem/cache/hw2/micro-cache.cc:65
65          bool hit = false;
```

Now that we've stopped the execution, we can examine all sorts of state, move to the next lines of execution, step into function calls, etc. Some things that might be helpful are to look at the state of the current packet, any queues or buffers in which you are storing state while waiting for a request or response to come back, looking at the packet's data, etc. For instance, in this current execution, I show what happens if you examine some of the packet state before performing subsequent lines of code and stepping into the scaffolding.

```
Breakpoint 1, gem5::memory::MicroCache::handleRequest (this=0x494dff0,
    pkt=0x4e21d60) at src/mem/cache/hw2/micro-cache.cc:65
65          bool hit = false;
(gdb) p pkt->getAddr()
$4 = 5888
(gdb) p pkt->print()
$5 = "ReadSharedReq [1700:173f] IF"
(gdb) p pkt->data
$6 = (gem5::PacketDataPtr) 0x4e21d10 "!N"
(gdb) n
67          if (hit) {
(gdb)
71              stats.misses++;
(gdb) p hit
$7 = false
(gdb) n
```

```
72              sendToMem(pkt);
(gdb) s
gem5::memory::MicroCache::sendToMem (this=0x494dff0, pkt=0x4e21d60)
    at src/mem/cache/hw2/micro-cache.hh:281
281             if (!pkt->needsResponse()) {
(gdb) n
285             mem_port.sendPacket(pkt);
(gdb) n
286         };
(gdb) c
Continuing.
```

Some of this may be useful, and some may not. Such is the plight of debugging!

You will notice that the first part of the Cache Assignment asks you to build a state machine to guide your development. This is because maintaining a mental model of your larger goal can help clarify lower level tasks to execute. System design is often about the transformation of complex, abstract concepts into simple expressions of code! Given this, do you see anything in the abstract system design that is unimplemented or incorrectly deployed in our design? (see next page for answer)

As you may have noticed, our "do-nothing" cache correctly sends requests to memory from the processor side, but we do not send the responses from the memory back to the processor! Thus, the last thing to do is to implement the `handleResponse` function to forward these data back to the processor. Do so by adding the following line to your `micro-cache.cc` file:

```
void
MicroCache::handleResponse(PacketPtr pkt)
{
    // TODO: complete this function!
    sendToProc(pkt); // add this line!
}
```

Recompile your code (`scons build/X86/gem5.debug`) and rerun your code. Does it work? If so, congratulations! 🥳 you've completed the lab. If not, happy debugging 🙃 come by office hours if you are still stuck!