Lab 5: A Non-Blocking Instruction Cache

4541.763 Laboratory 5
Assigned: November 3, 2009
Due: November 17, 2009

1 Introduction

In Lab 4, you were given a multi-cycle SMIPSv2 implementation, which you then pipelined to improve instruction throughput. In this lab, you will be asked to further improve the performance of your microarchitecture by replacing the existing blocking instruction cache for a non-blocking one.

To see why this might be a good idea, suppose that the instruction cache can return the memory responses in a different order from that in which the requests were made. This is actually what most modern processors do in regards to data memory: they tag the memory requests, and as long as the tags are returned with their corresponding memory responses, the processor can use some relatively standard techniques to execute the instructions out of order, requiring only that they be committed in a consistent manner. This technique increases parallelism by hiding latency.

To further illustrate this point, consider the simple example where memory request `a` is made before memory request `b`, and there is no data dependency between the two fetches. If `a` generates a cache miss (requiring an expensive memory request) and `b` hits in the cache, the absence of data dependency allows the processor to do useful work with `b` while waiting for `a`, effectively hiding some of the latency of `a`'s cache miss.

Of course, there is a substantial hardware burden to keep track of all the instructions in flight, (as well as their dependencies on each other), and the simple pipeline we created in Lab 4 may not be deep enough to exploit out-of-order caches. While implementing the extra microarchitecture is beyond the scope of a lab exercise, we can get part of the way there, and optimize the instruction fetch by creating a non-blocking instruction cache which still returns the responses in order. The serial channel we use to communicate between our caches and the backing store is fully duplex, as well as being pipelined. This means that if we allow multiple outstanding memory requests, we can hide a lot of latency, and improve the processor performance substantially.

This lab is the first time we are asking you to undertake a modification which takes direct advantage of a physical property of the hardware we are using to run the design. Consequently, you will be able to debug only certain aspects of your new microarchitecture in simulation; the true test of performance will need to take place on the FPGA.

2 Blocking vs. Non-Blocking

To begin with, let's examine the difference between a blocking and non-blocking cache. Look at the rules named "access", "init", and "refillResp" in the file `InstCacheBlocking.bsv`. The guards for the three rules are mutually exclusive Boolean expressions involving the `stage` register. Looking more closely at the `access` rule, you can see that when a miss occurs, a read request is sent to main memory, and the stage is set to `RefillResp`. No forward progress is made until the response returns from main memory, which is why we say that the cache **blocks** on a miss.

We have made your life somewhat easier by concentrating on the read-only instruction cache, not the more complicated read-write data cache. This simplifies matters substantially since we don't have to worry about dirty cache lines. In the data cache, when a cache line is evicted, it must be written back to main memory if it has been locally modified. This adds another ordering constraint which we can ignore: If a cache line is evicted in the instruction cache, all we have to do is overwrite it. In the non-blocking cache, we need to handle a miss by issuing the request to main memory, and then proceed to do more work while the memory response is generated. This work can fall into one of three categories:

1. Accept additional cache requests: This will result either in a cache hit, in which case we read the value and store it in a special structure, or a miss, requiring us to issue a memory request.
2. Process main memory responses: We need the response logic to be sufficiently decoupled from the other tasks so that we can respond immediately to responses from memory.
3. Return instructions to the processor: the memory responses must be returned in the same order in which they were received.

3 Achieving Non-Blocking Behavior

This section is designed to guide you through the refinement of the instruction cache. Before you begin, you will need to create a `lab5` model directory and a `lab5` module directory in your SVN repository. In your `lab5` module directory, duplicate the instruction cache module provided in the `mit-6.375 lab5` module directory (`instruction_cache_wide.awb`, and all the necessary files). We are using a modified cache with a longer cache line for this lab, so make sure you get the right one, and not the one you used in lab 4. Next duplicate the Processor module (`pipeline_processor.awb`, and all necessary files) which you created in lab 4. Rename the cache module `instruction_cache_nb.awb`, and the processor module `pipeline_processor_nb.awb`. While the majority of the changes you will need to make are with the instruction cache itself, we may require modifications to the processor which only make sense in the context of a non-blocking instruction cache and therefore want to leave the original module unchanged. As always, edit the module descriptions the AWB files of your new modules to make them readily identifiable in the apm editor.

You will surely have noticed that some additional benchmarks have appeared in `Benchmarks` ⇒ `mit-6.375` ⇒ `processor_test` ⇒ `mips_nonblocking.cfx`. As was explained in the previous lab, in order to concentrate on the processor pipeline performance, we ran each benchmark kernel twice: first to prime the caches and then to collect statistics. Since all of the benchmarks we are using fit entirely into the caches, this approach would not reflect any optimizations of instruction cache misses (the whole point of this lab). This where the `*_noprealloc` tests come in. They do not prime the caches before gathering statistics, and should therefore show substantially improved performance when run with a non-blocking instruction cache. The original benchmarks are useful to verify that we suffer little or no degradation in the common case of a cache hit.

Copy the pipelined processor models from your `lab4` directory to your `lab5` models directory (one running in simulation and the other on the XUP), and reconfigure them to use the new processor and instruction cache modules you just created. Verify that these models compile and execute correctly. Record the performance of both these new models for all the benchmarks (both old and new). It is important to track the performance changes associated with each modification as you will be asked to discuss them in the lab writeup.

3.1 Adding a Completion Buffer

The first thing that should be obvious when considering multiple in-flight requests, is that they will not necessarily complete in the order in which they were issued. Consider the case where a cache miss is followed by a cache hit. Although we can immediately resolve the cache hit, doing so before resolving the cache miss would be an error, since memory responses must be returned in order. To assure the correct ordering we will use a completion buffer.

A completion buffer provides a place for results which are handled too early to stay. To use a completion buffer, we must first `reserve` a space in the buffer and get a token corresponding to that space. Once we have calculated the value we want to output (in your case this will most likely be having a memory response), we can `complete` the result, giving both the token we got when we called `reserve` and the actual result. Remember you must use a fresh token for each message that goes into the completion buffer. Finally, when the *oldest* slot in the has been completed, we are allowed to call `drain` to extract the result out of the completion buffer.

The Bluespec `CompletionBuffer` library module has overly restrictive scheduling properties. It does not permit `reserve` and `complete` to be invoked in the same cycle, something which is important for obtaining full throughput. Unfortunately, the Bluespec `CompletionBuffer` library does not provide access to it's internal token representation, so we have to define our own interface if want to use our own implementation. Our interface will closely mimic the Bluespec library design. Our completion buffer is represented using the following interface:

```
interface CBUFF#(numeric type n, type element_type);
    interface Get#(CBUFFToken#(n))                              reserve;
    interface Put#(Tuple2#(CBUFFToken#(n), element_type)) complete;
    interface Get#(element_type)                                  drain;
endinterface
```

Notice that the interface is parametrized by the numeric type `n` which represents how large many concurrent elements can the buffer at a time (and consequently the size of the token size). You can add the Completion Buffer library containing the `CBUFF` interface and the `mkCBUFF` module by including `asim/provides/processor_library.bsh`.

Once the data is returned from main memory, place the value in the completion buffer using the `complete` method. The method `drain` is used to remove completed elements (in the order of their reservation) and is guarded not by the presence of data, but by the completion status of the oldest reserved slot. Though the schedule does not prohibit such behavior, be aware that if you attempt to reserve a token and complete it in the same cycle, data will be dropped. Obviously, we want to be able to accept memory cache requests (each request requires a new token from `reserve`) and complete older requests (invoking `complete`) in parallel, we just can't reserve a token and complete it in the same cycle (something you might be inclined to do on a cache hit). Enforcing this latency requirement comes naturally on a cache miss, but in the case of a hit you will need to add mechanism to handle this latency requirement. Think about decoupling the completion of cache hits using a small pipeline, making sure that your fifo has the correct scheduling properties (hint hint).

With the addition of the completion buffer, we need one more piece of state to make a working design: an additional FIFO to track the outstanding main memory requests. When the memory returns, we must remember the user tag, the address which we requested, as well as the completion buffer token. Be careful to size this FIFO correctly; the number of in-flight memory requests depends both on the length of this FIFO as well as the capacity of the completion buffer. The number of outstanding memory requests we choose to allow is an important decision which depends primarily on the accuracy of our branch predictor. We will play with this number later, but for the time being, choose an arbitrary power of two (the scoreboard FIFO will complain if its length is not a power of two!). Remember that there is a relationship between the size of the `pcQ` in the processor module,

and the number of in-flight requests in the instruction cache. What do you think happens when the `pcQ` is smaller than the depth of the completion-buffer and in-flight FIFO?

Once you have instantiated the completion buffer and in-flight queue, you will need to change the behavior of some of the rules in your design. Instead of returning data directly back to the user, your design must now set the data in the completion buffer, along with the correct reservation token (both for cache hits, as well as cache misses). Additionally, you will have to add a rule which dequeues items from the completion buffer and places them in the `instRespQ`.

With the addition of the completion buffer and the in-flight FIFO to track outstanding memory requests, you should now have a functionally correct design, which should pass all the unit tests and benchmarks, both on the FPGA and in simulation. Verify that this is the case, and fix any bugs before proceeding.

3.2 Avoiding Redundant Memory Requests

If we get back-to-back instruction requests at sequential addresses (very common in the instruction caches, since sequential instructions are located in adjacent memory), there is a good chance that they will hit the same cache line. Of course we would like to avoid generating redundant memory requests, but unless we further augment the design, this is exactly what will happen. This problem sounds a lot like the problem of hazard detection in Lab 4, which we solved by using a searchable FIFO between the execute and writeback stage.

`SFIFO.bsv` contains a module called `mkSizedSearchableFIFO`, which is perfectly suited for our needs. You should use your experience from last lab to design the appropriate search function, and upgrade the in-flight FIFO to a searchable one. Once you have detected a redundant memory request, you need to ensure that you don't respond to the `InstReq` in question before the memory request has been filled; essentially you need to establish a dependence between the two requests. **Hint:** It is possible to do this without adding any additional state to your design. Think instead about enriching the data-structure stored in your in-flight FIFO.

Once again you should take time to test your design both in simulation and on the FPGA. Running it on the FPGA is good, since the simulated serial channel is not at all realistic and won't expose some potential concurrency bugs you might have in your design. Don't be discouraged if your non-blocking design performs worse than your Lab 4 submission, as there is still some fine-tuning required to optimize the common case.

3.3 Killing Mis-Predictions

Your current design will probably run quite a bit slower then the design with the blocking cache. This is partly due to the fact that even in the case of a mis-prediction, we are still waiting for the memory to return before killing the instruction in the execute stage of the processor. A useful optimization here would be to add logic to detect when the processor's epoch has been updated, and take the following steps in the instruction cache:

1. Complete all outstanding instruction request from the previous epoch with bogus data (they will be discarded anyways), allowing the completion buffer to drain making room for instruction requests from the new epoch.
2. Figure out how many of these bogus instruction requests corresponded to outstanding main memory requests, and record that number.

3. Receive and discard the memory requests we no longer care about.

To make this modification, we will need to add two registers to our design; one register to hold the current epoch, and one to record the number of main memory responses to discard. In addition to this new state, we must modify the instruction cache interface (by adding an `epoch_put`), as well as the Processor interface (by adding an `epoch_get`) to get the epoch from the Processor core. Examine how the `InstCache`'s `statsEn` is set, and copy that mechanism exactly for your new epoch register. This will require you to create a local copy of the processor core `copy_core.awb`, and `Core.bsv` to your `lab5` modules directory, and reconfigure your two models to use this modified core), since we need to instantiate an additional `mkConnection` between the processor pipeline and the cache. Since we have already made local copies of the Instruction Cache and processor, we can modify their interfaces freely.

Once again you should verify that your design runs correctly both in simulation and on the benchmark. At this point, you should see improved performance of the `*_noprealloc` benchmarks run on the FPGA, though your performance on the original benchmarks will almost assuredly still be quite poor. It is important to understand why you don't see any performance benefits when running your non-blocking cache in simulation: This optimizations we are making take advantage of the fully-duplexed pipelined property of our physical serial channel. We have no idea how the simulated physical channel is implemented, but chances are it is neither pipelined nor fully duplexed.

3.4 Performance Tuning in Common Case

This is the part where we try to recover the lost ground on the original benchmarks. While it is good to optimize cache misses, we know from our architecture course that the common case is really the most important and in the case of processors, cache hits generally occur far more often than misses (that is what makes a cache effective). This is perhaps the most difficult part of the lab, so if you are able to get within 5% of the your performance on the original benchmarks with your Lab 4 solution, we'll consider that good enough. Let's concentrate on the original benchmarks for this section.

One of the liabilities of a deeper pipeline is that mis-predictions now take more cycles to be corrected. In the original design, the execute stage was never more than on instruction ahead of the instruction fetch, therefore in the event of a mis-predict, we only needed to kill one instruction. Much of the problem lies in the fact that the branch predictor we built in Lab 4 lacks sophistication, and without adding a lot of logic to it, it will remain an impediment. You should to think about lowering the limit on the number of in-flight memory requests as a way of mitigating the branch-predictors mediocrity. This way, we avoid compounded mis-predictions, where the branch predictor more or less falls off a cliff.

The next thing to think about either resizing, replacing, or removing all-together the plethora of queues in the instruction cache. Remember that the original choice for `reqQ` and `respQ` was influenced by the need to avoid a combinational path through the instruction cache. Now that we have the completion buffer, is it possible to remove these entirely? Asking questions like these are the key to regaining the lost ground on the original benchmarks.

4 Critical Thinking Questions

Provide the answers to these questions in the `answers.txt` file in your `lab5` directory.

IPC Changes

list the IPCs of your processor models on all the benchmarks after each successive modification you were asked to make. What (if anything) can you conclude about the individual contribution of each microarchitectural change?

Benchmark Analysis

Some of the noprealloc benchmarks shows a significant improvement when using the non-blocking instruction cache while others were less dramatic. What characteristics of an executable make it more amenable to the optimizations we made to the instruction cache in this lab?

Regaining Lost Performance

If your performance (IPC) on *any* of the original benchmarks is worse than your solution to Lab 4, please explain what may be causing your reduced performance. If not, you may just write that there was no performance degradation.