PyMTL3-mem: A Blocking Cache Generator

Xiaoyu Yan, Eric Tang

School of Electrical and Computer Engineering, Cornell University, Ithaca, NY

revision: 2020-05-30-03-28

Contents

1. Introduction

To mitigate rising non-recurring engineering costs, complex system-on-chip (SoC) design is increasingly using RTL design generators rather than specific RTL instances. To this end, we developed PyMTL3-mem, a 3-stage pipelined write-back, write-allocate blocking cache generator parameterizable by size, cache line size, data width, and associativity. The cache is an L1 cache for processors and will have great modularity to fit into any future projects and tape-outs within BRG. It is one of the many IPs available in PyMTL3, a Python-based hardware description language. For spring 2020, We used the cache generator in the CIFER chip tape-in to generate 4 KB, 2 way set-associative L1 instruction and data caches for a Manycore Tile. The cache allows for invalidating and flush transactions to support the CIFER specific software-centric cache coherence (SC3). The cache is fully translatable to Verilog and has been synthesized using a 1.25 GHz timing constraint.

2. Methodology

2.A Interfaces

The cache has two interfaces: `MemMinion` and `MemMaster`. `MemMinion` allows for the cache to behave as a minion handling requests from a master, usually a processor or an upper-level cache. The cache is a master in `MemMaster` where it sends requests to a minion module, which is usually a lower level cache or main memory. Currently, the cache is designed as an L1 cache but multiple of these caches could be connected for designs requiring a larger cache hierarchy. This interface provides modularity to the cache allowing it to fit nicely into many different RTL projects.

2.B The Pipeline

The pipelined cache generator has 3 stages. The M0 stage handles incoming requests from the master and responses from the minion. The cache deallocates from the miss status hit register (MSHR) in the case of a minion response and sets up the inputs to the tag array SRAM during this stage. The `M0` stage has an FSM to handle special transactions that will go down the pipeline multiple times such as cache initialization, flush, and invalidate. For each of these transactions, multiple lines of the tag array must be updated. However, since it takes one cycle to write a single line and this must occur during the M0 stage, these transactions must spend multiple cycles in the M0 stage. The FSM in the M0 stage is what allows for this to happen.

The M1 stage performs the tag check and processes the control bits such as valid and dirty bits with a data read from the tag array. The cache allocates to the `MSHR` if there is a miss and sets the inputs to the data array SRAM during this stage as well. The M2 stage sends responses to the master and requests to the minion if we need a refill.

We store the valid and dirty control bits in the tag array to save some area. Because of this, certain transactions will require a stall to update the tag array. This provides us with good returns for saving area because the CIFER cache uses per word dirty bits, requiring 4-bits for a 16-byte line. For a 2-way associative cache, we had to store the replacement bits in a register to implement true LRU. For multiway associative cache, we use multiple tag arrays since we need to read from all of them at the same time and check all the tags but only one large data array since by the M1 stage, we will have known which way to read or write to. Because we stored the control bits within the tag array SRAM, we must reset all values in the tag array during power on. For a 4 KB cache, this requires a one time cost of 256 cycles, which is insignificant in comparison to run time.

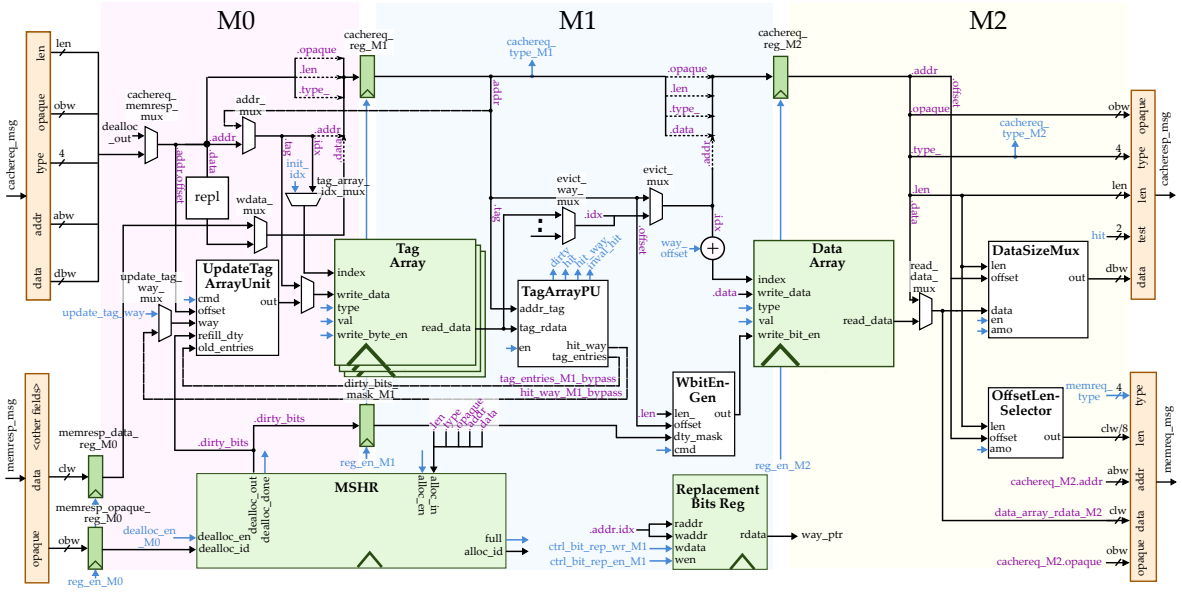

Figure 1: PyMTL3-mem Datapath - 3-stage pipeline blocking cache generator datapath.

The cache has an FSM in the M0 stage. The states are `INIT`, `READY`, `REPLAY`, `INV`, `FLUSH`, and `FLUSH_WAIT`. The `INIT` state is for SRAM initialization during power on. We use an FSM and a counter to write zeros to all entries in the tag array and data array. We load the counter with the total number of cache lines and it will count down each cycle. There's a modulo operator for different higher associative cache so that we can reset the index back for the next set. The `READY` state handles normal cache transactions that are not replays. The `REPLAY` stage handles replays from when the `memMaster` response returns with the refill or AMO. `INV`, `FLUSH` and `FLUSH_WAIT` states are for CIFER specific transactions involving iterating through every single cache line.

2.C Read and Write

Reading and writing are the most basic operations of the cache. For reads, the cache reads from the tag array in M0 and checks for a hit in M1 using the `Tag Array Processing Unit`. This module performs a tag check, reads the dirty bits, and marks which way the hit occurs. It also serves as a gateway between the output of the tag array and the rest of the cache. If we read from the tag array, then we will propagate the output of the tag array, otherwise, we ignore the tag array values. If we have a hit, then we read from the data array with the index at the correct way offset and send the minion response in M2. If we have a miss to a clean cache line, then we must perform a refill request. The cache allocates an entry in the `MSHR`, which prompts the cache to stall, and sends a memory request in the M2 stage `MemMaster` interface. When we get the response, we will first refill the cache and then replay the same transaction down the cache line. This requires two cycles but isn't significant in comparison to the memory access latency.

2.D Atomic Memory Operations

Atomic Memory Operations (AMO) are handled in the main memory or L2 cache. This cache's job is to pass the AMO using its `MemMaster` interface to a lower level cache, which will perform the AMO and return values. There are some specific constraints; if we have cached data with the same line as the target of the AMO, this data will be stale and we need to invalidate them. If the cached

data is dirty, then we need to write back. All AMO behave the same in this cache since we just forward the operation to the lower level cache. To build the request for `MemMaster` properly, we need to use the `OffsetLenSelector` as shown in figure 1 to choose the number of bytes to access. For AMO, this will be 4 for 4-byte access. For any other transactions, we will default the byte access to 0 meaning we want line access.

2.E Invalidate and Operations

The invalidate transaction iterates through each line in the cache and invalidates it. Therefore, we only need to write the control bits in the tag array. Just like cache initialization, we use the same counter to set the index to the tag array and write zero to the valid bit. To help organize the writing of the control bits and the tag in the tag array, we used a `UpdateTagArrayUnit` to build the tag array write data depending on the transaction and state. We use this unit for the initialization state to write clear all the data and for refills to set the control bits. To properly implement SC3, when we have a miss to an invalid but dirty cache line, we do not perform a write-back because that is a flush. Instead, we request a refill like a clean miss and then use the inverted dirty bits as the write mask for the data array during refill. To achieve this, we store the dirty bits in the `MSHR` and use the `WbitEnGen` to generate the write enable mask for the data array. The enable is bitwise so the `WbitEnGen` converts the 4-bit dirty control to a 128-bit write enable mask where each bit in the dirty control is one 32-bit word. We also added a `invalid_hit` status bit to signal that we have a hit to a dirty invalid line and not just a regular invalid line. The cache relies on the invariant that an invalid and dirty line will only be the result of an invalidate transaction because we initialize the tag array. One area for improvement is that instead of always refill, we can look at the dirty bit and offset and check if the word accessed is dirty. If it is dirty, then we simply respond with the word without requiring an expensive refill request.

The flush transaction goes through every line and write back dirty lines and leave clean lines. This transaction moves the FSM to `FLUSH`. The FSM is useful because the flush will spawn multiple transactions that will go down the pipeline whose goal is to look at each line in the cache. Because we stored the dirty bits in the tag array, we need two cycles max for each line. One cycle to read from the tag array and if dirty, clear that dirty bit. This requires two accesses to the tag array, which makes the two-cycle latency mandatory. We want to check if the line is dirty because we only write back dirty lines in order to efficiently use the memory bandwidth. First, we read a line from the tag array to check the first bit. If the line is dirty, the cache will then read the data from the data array and send the write request to the main memory. Finally, the cache will transition to the state `FLUSH_WAIT` and block until it receives a write acknowledgment from the main memory. For read and write transactions, the cache ignores these because it uses the refill acknowledgment instead but for flush it allows us to go down the pipeline again to clear the dirty bit in the tag array. After the acknowledgment and dirty line update, we repeat this process for the next line.

3. Testing

3.A Single Cache Testing

As shown in figure 2, single cache testing contains the test source, which drives the design under test (DUT) with test transactions. The sink collects the output from the cache and checks for correctness. The CIFER processor only sets the cache response ready signal only when it sends a cache request. However, since we have separate models for the source and sink we cannot easily replicate this dependency by simply connecting the source and sink to the DUT. For this reason, we created a `ProcModel` where we implement this rule to unify the cache request enable signal in the

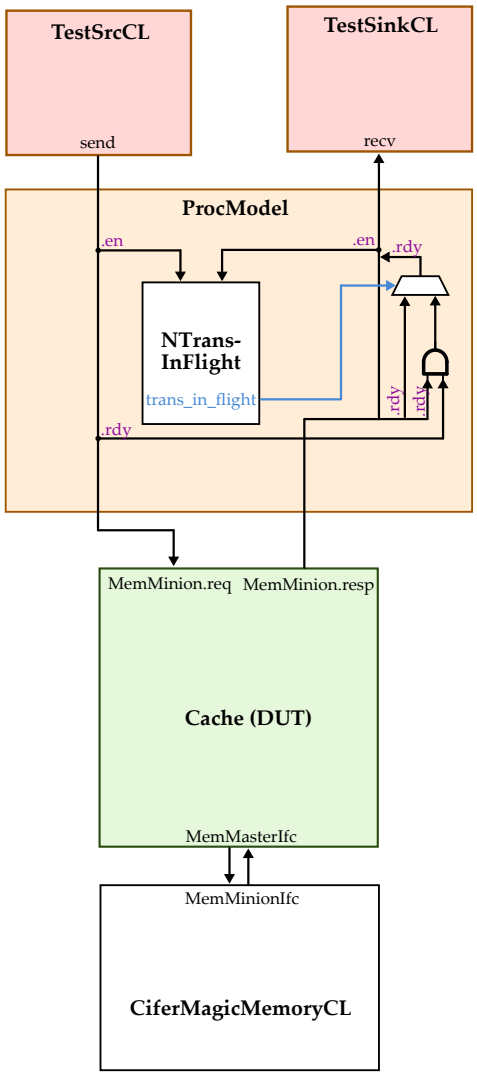

Figure 2: Single Cache Testbench - The single cache testbench is modified from past designs by adding a ProcModel. This module is essential to allow for testing source and sink dependencies.

source and the cache response ready signal in the sink. The cache also connects to the magic memory that acts as a lower level cache for refills and AMO transactions.

We use a combination of direct test cases and random test cases for verification. The direct test case suite tests the correctness of specific transactions as well as sequences of transactions. This past semester new direct tests for AMO, flush and invalidate transactions were added. Each of these tests is also done for various cache configurations. Each test is run with different data bit widths, cache line bit widths, number of cache lines, and associativity (1 or 2). We also test obscure cache configurations such as having the data bit width equal to the cache line bit width. Any bugs found by using Hypothesis tests were added to this suite of direct test cases as well.

The random tests use both complete random and property-based testing with Hypothesis. The completely random tests help find logic bugs by using a fixed cache configuration and driving the cache with hundreds of transactions to stress test the cache. One focus of this test is to keep the cache

sizes small so that we can test as many unique combinations of transactions as possible. This strategy relies on brute force to find bugs and was able to help catch some obscure bugs. Hypothesis tests both logic and parameter but due to a large number of possible cache combinations, we have to run the tests for a much longer time before catching the bug in comparison to random testing. Additionally, random testing allows us to have more control over the distribution of transaction types. We want most of our transactions to be read and write transactions since they help fill up the cache and can lead to more interesting cases for AMO, invalidate, and flush operations. Hypothesis does not offer the same flexibility and can lead to more difficulty in finding edge cases.

3.B Multi-cache Testing

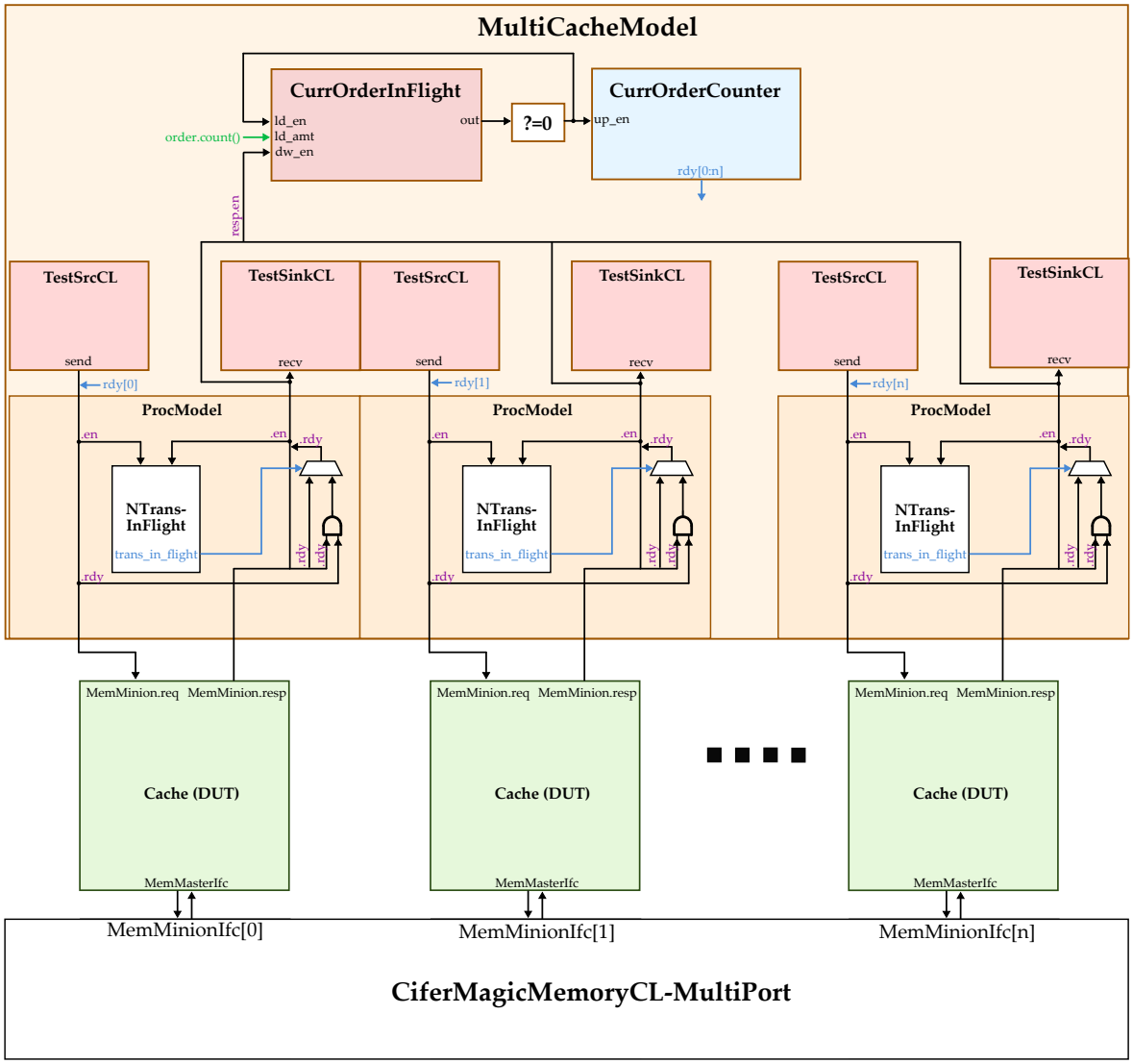

Figure 3: Multi-cache Testbench - The multicache testbench contains elements from the single cache test bench in figure 2 while adding modules to allow testing parallel and sequential execution.

For multi-cache testing, we designed a modular test bench reusing components from single cache testing. As shown in figure 2, the multi-cache test bench contains multiple single cache test benches. The main challenge for testing multiple caches is that we have the possibility of transactions executing in parallel. For example, transactions to two different caches can execute simultaneously or sequentially and the user should be able to decide this outcome. To this end, we added an extra layer to the test bench called the multicache model. We added two extra fields to each transaction: cache number and order. Cache number indicates which cache this transaction is for and the order is the priority of the transaction. The order of the transaction will determine when it will execute or sent the cache. All transactions with the same order can execute simultaneously while transactions with higher orders must wait for transactions with lower orders to finish executing. Currently, the multi-cache testing only supports direct test cases where the user comes up with a sequence of transactions and then test it. Random testing has many additional challenges such as the test generator needing to generate a valid sequence of transactions that ensure coherence. This requires implementing an additional coherence reference model for transactions.

As shown in figure 3, the multi-cache testbench introduces two new modules `CurrOrderInFlight`, which tracks the total number of transactions of this order still in flight, and `CurrOrderCounter`, which keep track of the current order in the testbench. Before each order of transactions executes, we load `CurrOrderInFlight` with the count of all transactions with the order. We can tell if the transaction terminates by looking at the cache's `MemMinion.resp.en` signal, which the cache pulls high when it finishes executing a transaction. This is used to keep track of the number of transactions that terminated per cycle and we subtract that from the amount in `CurrOrderInFlight`. Once we reach zero, we increment the current order stored in `CurrOrderCounter` and load the number of transactions with the next order. The `CurrOrderCounter` controls the `MemMinion.req.rdy` signal along with the cache. It will only set the ready signal to high if the order of the incoming transaction matches the order in `CurrOrderCounter`.

4. CIFER

We used the cache as L1 instruction and data caches for the CIFER Manycore Tile in a 14nm standard cell library and 1.25 GHz timing. Both instruction and data caches are 4KB, 2-way associative caches with 16-byte cache lines. The instruction cache has a 128-bit data port to allow for the cache line read used by the processor's prefetcher. It has an area of 9389 μm^2. The data cache has a 32-bit data port meaning that the largest access it can support is 4 bytes and gets us an area of 9132 μm^2. The SRAMs take up nearly 83% of the total area with the data array taking up 57% and tag arrays taking up 25%. This is great because we want most of the area to be useful storage. Figure 4 shows the distribution of the non-SRAM area in the cache. The pipeline registers take up about one-third of the remaining area. Unfortunately, this area is fixed since it sends the transactions from one stage to the next. The M2 stage stall engine takes up a significant amount of area since it has to store the 128-bit wide output from the data array. We observe that the replacement bits registers, which stores the LRU bits take up only 2% of the area because it has one bit per line.

5. Future Work

The next steps for this cache going forward are to allow for more capability regarding the associativity parameter. Currently, the associativity is only allowed to be 1 or 2, but support can be added so that any number of ways may be allowed. This will require adjusting the tag array, muxes, and comparator in the datapath along with adjusting the replacement policy module.

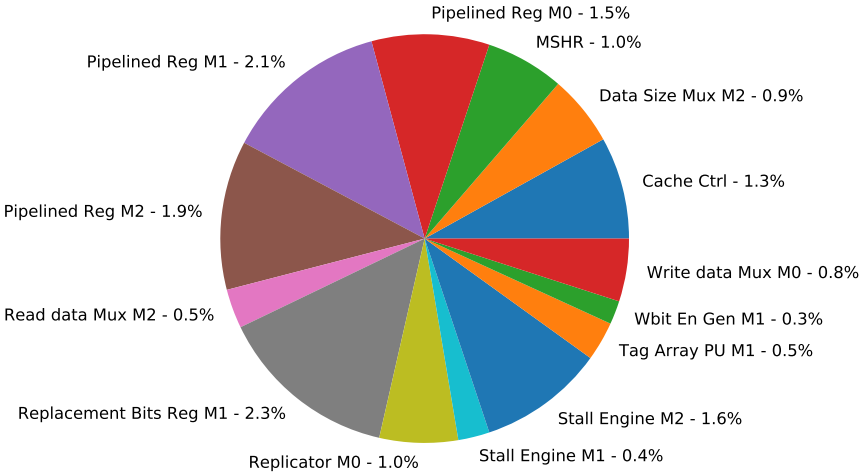

Figure 4: Area of Non-Sram Components in the Cache - Area of modules shown in figure 1. The percentage is for the entire cache area including the SRAMs.

Following this, a much more major change that can be made is to make this cache non-blocking. Given that this cache is fairly similar to the one developed in ECE 5745 in Spring 2019, by using that cache as a model, this change may not be too difficult. This will require allowing for more than one MSHR entry along with adding some method of keeping track of secondary misses. (In the past this was done using a linked list structure but this had a large area overhead.) Following this, the control logic for the cache must be adjusted to first allow for multiple transactions to be in flight and secondly to handle hazards that may occur now that there are multiple transactions in flight. Again, by referencing the report and cache from ECE 5745, most of these cases have already been identified. In this case, it may be easier to first add registered inputs so that it is easier to handle these cases. As always when adding new features, more testing will be required as each incremental improvement is made.