Lab 3 Report
Elise Song (eys29)
Katherine Zhou (kz273)

Introduction

For this lab, we implemented a write-back write-allocate cache design using a finite state machine. In the baseline design, we implemented a direct mapped cache. In the alternative design, we changed this to a two-way set associative cache to potentially allow for more cache hits. This involved fully understanding how the cache works, as we needed to add the proper hardware into the datapath and their corresponding control signals into the control unit for each state. In lecture, we learned about different cache organizations, such as direct mapped, set associative, and fully associative, different write miss policies, such as write through and write back, and replacement policies such as least recently used (LRU) and first in first out (FIFO). The lecture content was given to us at a more abstract level, and so the lab allows us to cement our understanding of caches by putting it in practice. This lab also required a high level of organization and consistency in order to keep track of the many control signals that we needed for all twelve FSM states. Thus, the encapsulation design principle was especially important; the Control Unit and Datapath only shared necessary communication. We intentionally used an incremental approach of implementation and testing to ensure that the control logic for each transaction was correct.

Alternative Design

The cache implemented in the base design of this lab is a direct mapped write-back write-allocate cache with a 256 byte capacity and 16 byte cache lines. The cache is implemented using a twelve state finite state machine, broken up into five general categories: IDLE, TAG_CHECK, DATA_ACCESS, REFILL, and EVICT. In the IDLE state the cache is waiting for a request. Once a request is received, the cache moves into the TAG_CHECK state, where it will check to see if the tag of the request is the same as the tag in the cache at the index specified in the request address. From there, depending on if there is a tag hit, or if there is a miss, in which case we look at a dirty bit register to see if the cache line is clean or dirty, the cache will move to DATA_ACCESS, REFILL, or EVICT respectively. One notable difference between the cache we implemented and the cache we learned about in class is the addition of a WRITE_INIT transaction, which works to prevent compulsory misses on reads and writes. The datapath diagram of our baseline cache design is shown below in Figure 1.

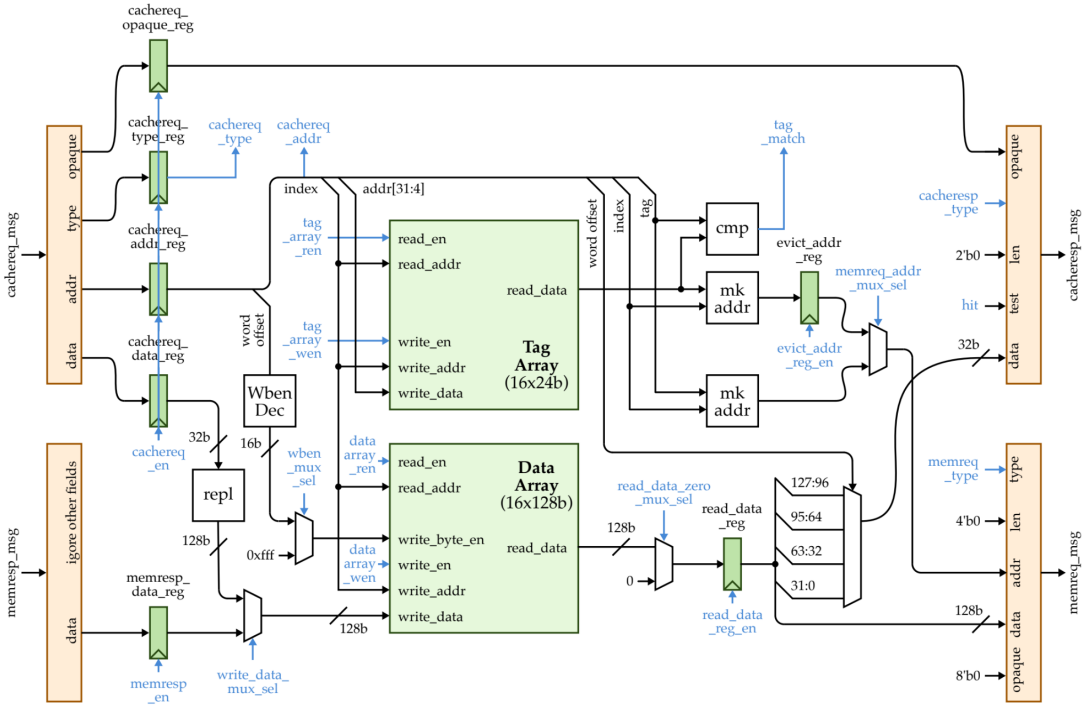


Figure 1. Baseline design datapath diagram

For the alternative design, we implemented a two-way set-associative, write-back, write-allocate cache with the same 256 byte capacity and 16 byte cache lines as the base design. The data array and tag array in the base datapath was replaced with two data arrays and two tag arrays of size 8 for each way. In Figure 3, this is represented by stacking the arrays. Additionally, the valid-bits register file was replaced with two register files of size 8 in the control unit for each array. The used-bits register size remained 16, however, the index was offset by 8 depending on which way. To maintain the capacity, the arrays have half the number of sets as before, and the address map changes accordingly:

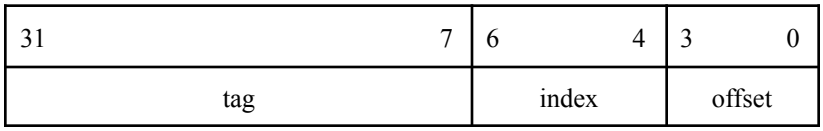


Figure 2.A. Two-way set associative cache address with no banking

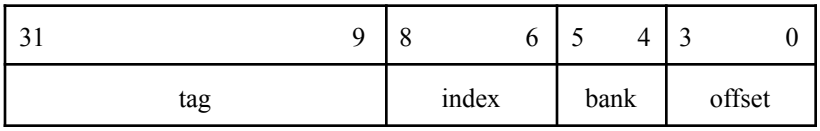


Figure 2.B. 2-way set associative cache address with banking

The index changes from 4 bits to 3 bits for 8 sets in each array. The leftover bit was added to the tag which was previously 24 bits. So, the tag array stores 25 bit sets.

To support an additional tag array and data array, the array read and write enable signals from the control unit are doubled. The tag array read enable signals, tag_array_ren0 and tag_array_ren1, are always enabled together to allow for parallel read. For parallel tag comparison, an additional compare module was added to the datapath, and two tag_match status signals are sent from the datapath to the control unit to compute a hit.

In order to implement a two-way set-associative cache, the control unit has to keep track of which way was least recently used (LRU). In the TAG_CHECK state, LRU is determined by the following logic:

```
assign lru_in = (!tag_match0 && !tag_match1) ? !lru : tag_match0 ? 1 : 0;
```

If there are no tag matches, LRU is toggled, otherwise the opposite of whichever way has a tag match is the least recently used. The LRU signal is used to select which tag and data from the two arrays during refill and eviction. The tag write enables and data read and write enables also depend on the LRU bit. The bits are kept in a register file that holds eight bits, one for each line of the cache.

Our design implements the design patterns of modularity through the separation of the datapath and the control unit modules. In the datapath and control module, everything is broken down further into smaller modules for each hardware component or always blocks for control logic. In the top level module, we declare the datapath and control modules. This demonstrates a hierarchical design. Additionally, our cache demonstrates the encapsulation design principle, since the control and datapath only need to share control and status signals.

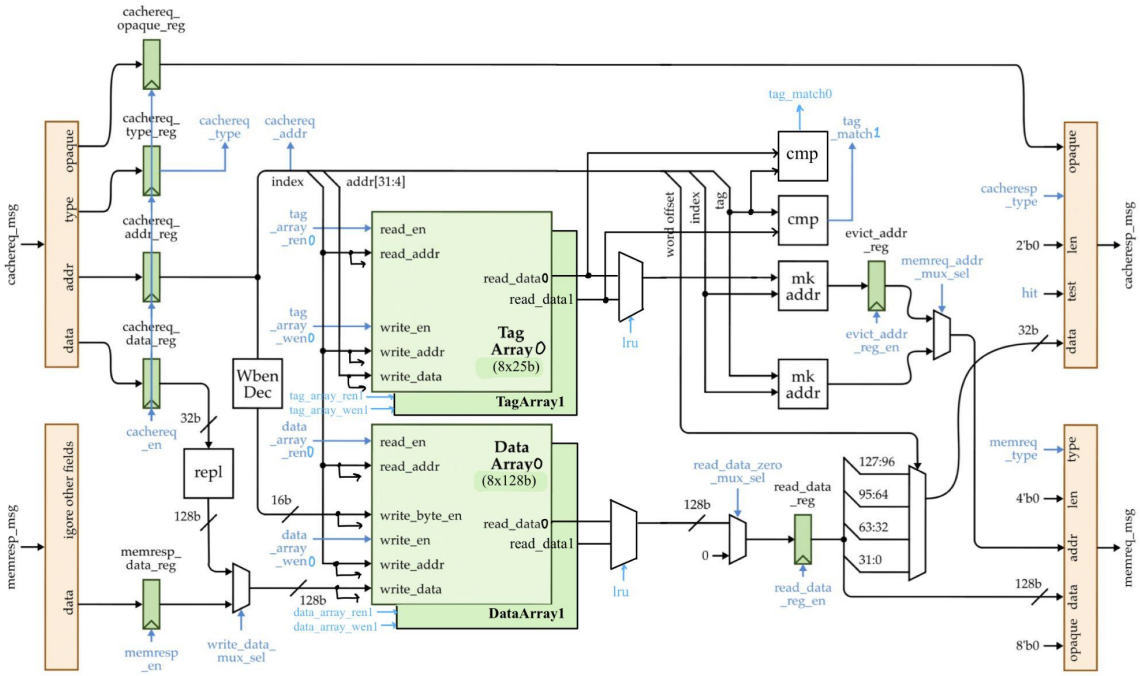


Figure 3. Alternative design datapath

Testing

Our overall testing strategy made use of black box testing, directed testing, random testing, and delay testing for every instruction. We also used unit tests for our decoder unit to ensure that there were no bugs there that could carry down into our processor implementation. Testing was done using pytest in Python. We made sure to run our tests on the functional level model first to guarantee the correctness of the test cases.

We made use of both black box and white box testing in order to ensure the overall functionality of our processor. Majority of our test cases do not take into account the specifics of the implementation of the cache itself, which is black box testing. This helps us to make sure that our tests are not passing solely because of some detail of the cache implementation, and that the overall correctness is maintained for all cache transactions..

We also made sure to use directed testing to test the specific behavior of the direct mapped and associative caches, since the hit pattern could potentially be different. We zoom in on specific instruction sequences, in this case the memory simulation instruction pattern, to make sure that they are executing properly. We also have a few more basic tests for the associative cache to test the LRU replacement policy to make sure that the correct entry is evicted.

Although directed tests are great at ensuring correct behavior of edge cases, it does not generate enough tests to achieve sufficient coverage. Therefore we made use of random tests to get a large number of tests with random addresses as well as random data to ensure that our cache is operating correctly, to potentially catch cases that we did not think of, and to add coverage to our tests. For these tests, it was important to fix the random seed so that the tests were replicable for debugging purposes.

All of the tests mentioned above were also done with the source and sink to ensure that the input or output is not getting “lost” during the delay period. Because all of our stream interfaces are latency insensitive, we need to make sure that the val/rdy protocol used is not affected by the amount of delay that we have. That is, the result should not depend on the number of cycles/the delay.

Finally, we needed to add some unit testing into our test strategy because of the added WbenDecoder unit that was added to our datapath to ensure that no errors trickled down in our implementation. The test was taken from the sec09 discussion github, and simply tests that the correct input pushes the correct output.

A table detailing our test cases is shown below.

Table 1. Summary of test cases

Test case category	**Direct Mapped or Associative**	**Description and justification**
Directed	Both	Because the direct mapped and associative cache are sometimes expected to behave differently, we used tests to ensure that this was true. For the direct mapped cache, we write to address 0x1000 and then to 0x2000, both of these miss. We then try to read from address 0x1000, which misses. In the associative cache, we do the same transactions, the difference being that the read from address 0x1000 should hit instead.
Directed	Direct Mapped	We wanted to test with high spatial and high temporal locality with the base_sim1, base_sim2, base_sim3 tests. The base_sim1 test does 5 transactions to array a: - read a[i] - write a[i+1] - i++ The base_sim2 test performs the same 5 transactions to two arrays, and base_sim3 on three arrays. - read a[i] - write a[i+1] - read b[i] - write b[i+1] - … In a direct mapped cache, base_sim1 follows a rm, wh, rh, wh, rh, wh, rh, wm, rh pattern (r=read, w=write, h=hit, m=miss). In base_sim2 and In base_sim3, all the reads miss and all the writes hit.
Directed	Direct Mapped	We wanted to stress the cache by accessing every line in the cache with base_stress and base_stress_cont
Directed	Associative	We wanted to test that the LRU was behaving properly to evict the correct way when both are full. We first write data to addresses 0x1000 and 0z2000. These have the same indexes but different tags, so they will go into way 0 and way 1 respectively. We then write address 0x3000, which still has the same index as the previous two entries. Therefore, the least recently accessed data should be evicted, in this case 0x1000. We then read from addresses 0x3000 and 0x2000, which should both hit. To ensure that eviction happened correctly, we then read from address 0x1000, which misses, but the response should contain the same data that was written to it from the first instruction.
Directed	Associative	The alt_sim1 and alt_sim2 tests use the same programs as the base_sim tests. However, alt_sim2 has different hit/miss behavior in an associative cache instead of a direct mapped cache. In an associative cache, alt_sim2 has similar behavior to the pattern described in base_sim1 since there are two ways.
Directed	Associative	We wanted to stress the cache by accessing every line in the cache with alt_stress and alt_stress_cont
Random	Both	We tested a sequence of 100 read transactions with random address accesses, as well as a write followed directly by a read with the same data and address.

Delay	Both	Delay tests serve to ensure that our implementation maintains the latency insensitive protocol. A source and/or sink delay is added to all of the above tests. We asl
Unit	WbenDecoder	Ensures the functionality of the decoder unit that was added to our datapath so that no errors are caused from it.

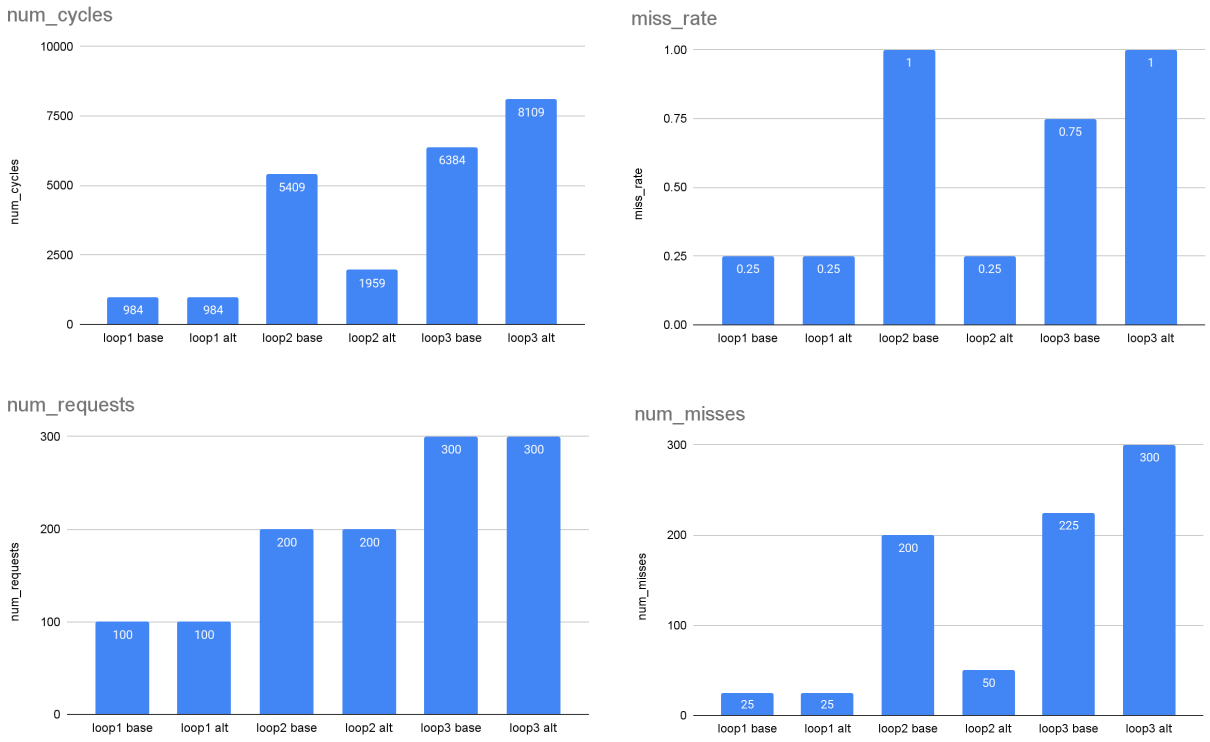
Evaluation

Quantitative Analysis:

We tested our Base and Alternative cache designs on two benchmarks: loopX and temploopX. The loop benchmark was given to us and does one read to X number of arrays consecutively. Figure 4 illustrates the differences between number of cycles, miss rate, number of requests, number of misses and average memory access latency (AMAL) for loop1, loop2, and loop3.

For the loop1 benchmark, Base and Alt perform the same which is as expected since only one way would be filled in the associative cache. For loop2, Alt performs much better than Base, reducing the number of cycles by 63.8%. The miss rate and number of misses both decrease by 75%. AMAL decreases by 63.8% which means the average number of cycles decreases by 63.8% since the number of memory accesses is constant over Base vs Alt loop2. This matches the decrease in total number of cycles.

However, for the loop3 benchmark, Base performs better than Alt since the third array address 0x3080 is in a different set and would not get evicted in a direct mapped cache but does in an associative cache. Thus Alt would have more cycles since it goes through the eviction path. Additionally, the address 0x3080 would get a hit in the direct mapped cache, resulting in a lower miss rate. So, the associative cache does not perform better than a direct mapped cache in all cases. But in ideal scenarios, the associative cache decreases the number of cycles and misses by more than 50%, as seen in the loop2 benchmark.



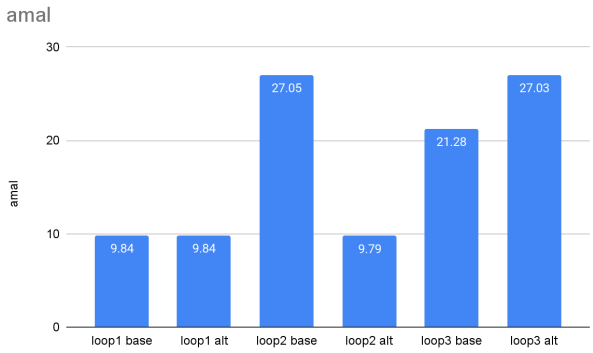


Figure 4. loopX benchmark

The temploopX benchmark exploits temporal and spatial locality. The C code for the benchmark is as follows:

```
temploop1:
# a array allocated at 0x1000
for ( i = 0; i < 100; i++ )
       a[i+1] = a[1] + 4

# temploop2:
# a array allocated at 0x1000
# b array allocated at 0x2000
for ( i = 0; i < 100; i++ )
       a[i+1] = a[i] + 4
       b[i+1] = b[i] + 8

# temploop3:
# a array allocated at 0x1000
# b array allocated at 0x2000
# c array allocated at 0x3000
for ( i = 0; i < 100; i++ )
       a[i+1] = a[i] + 4
       b[i+1] = b[i] + 8
       c[i+1] = c[i] + 12
```

The assembly code would read a[i] then write to a[i+1], increment i and loop 100 times. In the temploop1 benchmark, Base and Alt are comparable as expected since only Way 0 is filled in an associative cache. In temploop2, the number of cycles and AMAL decreased by 65.3%. Number of misses and miss rate decreased by 74%. For temploop3, Base and Alt are comparable again since we used address 0x3000. In both direct mapped and associative caches, the b array read would evict the a array write, and the c array read would evict the b array write. So as expected, the performance is the same. The hit and miss table for the temploopX benchmark is shown in Table 2.

Table 2. temploopX hit and miss table

Benchmark	Base	Alt
temploop1:		
rd 0x1000	m	same h/m
wr 0x1004	h	pattern
rd 0x1004	h	as base
wr 0x1008	h	
rd 0x1008	h	
wr 0x100c	h	
rd 0x100c	h	
wr 0x1010	m	
rd 0x1010	h	
wr 0x1014	h	
...	...	

Benchmark	Base	Alt
temploop2:		
rd 0x1000	m	m
wr 0x1004	h	h
rd 0x2000	m	m
wr 0x2004	h	h
rd 0x1004	m	h
wr 0x1008	h	h
rd 0x2004	m	h
wr 0x2008	h	h
rd 0x1008	m	h
wr 0x100c	h	h
rd 0x2008	m	h
wr 0x200c	h	h
rd 0x100c	m	h
wr 0x1010	m	m
rd 0x200c	m	h
wr 0x2010	m	m
rd 0x1010	m	h
wr 0x1014	m	h
rd 0x2010	m	h
wr 0x2014	h	h
...	...	...

Benchmark	Base	Alt
temploop3:		
rd 0x1000	m	same h/m
wr 0x1004	h	pattern
rd 0x2000	m	as base
wr 0x2004	h	
rd 0x3000	m	
wr 0x3004	h	
rd 0x1004	m	
wr 0x1008	h	
rd 0x2004	m	
wr 0x2008	h	
rd 0x3004	m	
wr 0x3008	h	
rd 0x1008	m	
wr 0x100c	h	
rd 0x2008	m	
wr 0x200c	h	
rd 0x3008	m	
wr 0x300c	h	
rd 0x100c	m	
wr 0x1010	m	
rd 0x200c	m	
wr 0x2010	m	
rd 0x300c	m	
wr 0x3010	m	
rd 0x1010	m	
wr 0x1014	h	
rd 0x2010	m	
wr 0x2014	h	
rd 0x3010	m	
wr 0x3014	h	
...	...	

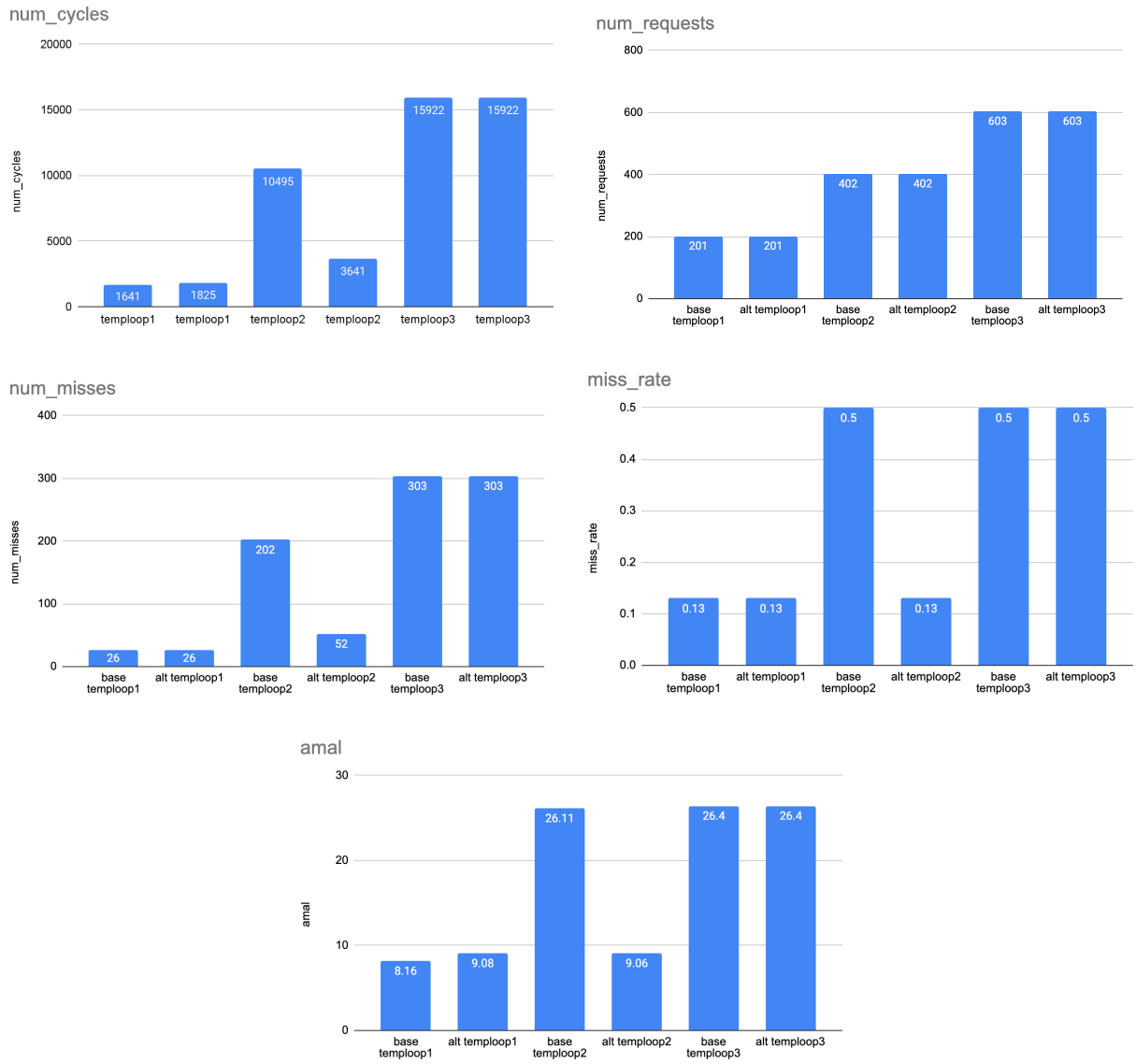


Figure 5. temploopX benchmark

Qualitative Analysis:

The addition of the array output muxes and the necessary duplication of the tag comparator does not necessarily cause the critical path to increase. In lecture, we determined that the critical path of the FSM cache was through the memory request and response path, however depending on the actual latency of the components the critical path may be through the data arrays. If this is the case, then the critical path is extended by one mux, slightly increasing the cycle time of our cache, which decreases the clock frequency. However, since muxes are basic pieces of hardware, the new critical path is not a drastic change, and the cycle time and clock frequency will be minorly impacted.

In addition to potentially increasing the cycle time, the data array mux also adds area to our baseline design because it adds hardware. This does not necessarily mean that there is a corresponding increase in energy consumption. We can calculate energy consumption of the entire instruction sequence by doing energy/instruction sequence = energy/cycle * cycles/instruction * instruction/instruction sequence. We know that the energy per cycle increases and the instructions per instruction sequence stays the same. Then, the total energy consumption of the

instruction sequence depends on the cycles per instruction. This value depends on the path of the actual instruction, for example a miss on a dirty cache line would take longer than both a hit and a miss on a clean cache line since it needs to go through memory, which usually has a high latency compared to other components. Thus, we can not definitively say whether or not the energy consumption increases or decreases. However, based on the results of the simulations, we can infer that for memory access patterns that require reading consecutively from two separate arrays, which demonstrates high spatial locality, the two-way set associative cache uses less energy than the direct mapped cache because of the large decrease in the number of misses. Similarly, when reading consecutively from three separate arrays, the two-way set associative cache actually performs worse in terms of the number of misses compared to the direct mapped cache. This leads to the two-way set associative cache consuming more energy than the direct mapped cache for the same instruction sequence.

Overall, there is not a clearly "better" cache design, but that the drastic improvement that the alternative design gives over the base design when sequentially iterating through two separate arrays makes the alternative design slightly better. The increase in performance in this case outweighs the decrease in performance that is shown in the loop3 simulation. From the simulations, we can conclude that the alternative design gives us the lowest miss rate when going through two loops that have high spatial and/or temporal locality and have different addresses with the same index. If the two loop addresses were to have different indices completely, then the second way would not be utilized and the performance would not see a boost when compared to the baseline design. When iterating over just one loop, the baseline and alternative design have the same miss rate, which gives the alternative design worse overall performance. We can predict this to be true for similar patterns as well, that a memory pattern involving looping over two arrays that have the same index but different addresses will give a large performance boost to the alternative design due to the decreased number of conflict misses. However, other memory access patterns, including accessing only one array of data and three or more arrays of data will generally see worse performance with the set associative cache. This could on a small scale be solved by increasing the associativity of the cache, say from two way to four way, which would allow us to have four entries with the same index but different addresses in the cache at once. The tradeoff to this would be that in order to have the same number of cache lines, we would have to double the size of the cache, which is non ideal.

Conclusion

The set associative cache required us to split the tag and data array into two separate arrays, one for each way. As a consequence, we needed to duplicate other pieces of hardware, such as our tag comparator, and add additional muxes to choose which read output of the arrays to use. Additionally, we needed to implement a least recently used policy for eviction from the cache. The critical path of our cache is not necessarily affected by the additional hardware, because in general the memory interface has the longest latency. Although area is added in the alternative design, it also allows for a much lower miss rate in some cases, which for those access patterns would decrease the amount of energy consumed. Through the memory simulations, we found that for the loop2 benchmark, the alternative design reduces the number of cycles by 63.8%. The miss rate and the number of misses both decrease by 75%. The AMAL decreases by 63.8%, which means that the average number of cycles decreases by that amount, since the number of memory accesses is constant over the baseline and alternative design. This matches the decrease in the total number of cycles. Similarly, for the temploop2 benchmark, the number of cycles and AMAL decreased by 65.3%, and the miss rate decreased by 74%. However, for other memory access patterns the alternative design potentially performs worse, an example being if the second way is not being utilized, or even if it is using the same number of cycles, the area increase in the alternative design would lead to an increase in energy consumption as well.

Work Distribution

Elise implemented the datapath modules for Base and Alt design and Katherine implemented the control modules for Base and Alt design. We worked together to debug our code using waveforms. For the lab report, Katherine wrote the Introduction and Conclusion sections, and Elise wrote the Alt Design section. Katherine and Elise both worked on the Evaluation and Testing sections.