LAB 3 REPORT: BLOCKING CACHE

Luke Puchalla [lcp77]
Eldor Bekpulatov [eb654]
Shivangi Gambhir [sg2439]

Luke Puchalla (RTL Design & Verification):

- Assisted with the development of both baseline and alternative designs
- Wrote various tests for both implementations
- Debugged and solved problems through testing the alternative design
- Wrote introduction and test strategy sections

Eldor Bekpulatov (RTL Architect, Design & Verification):

- Assisted with the development of both baseline and alternative designs
- Wrote various tests for both implementations
- Debugged and solved problems through testing the alternative design
- Wrote additional evaluation tests and most random tests
- Wrote the evaluation section

Shivangi Gambhir (RTL Design & Verification):

- Implemented the framework and functionality for both baseline and alternative design
- Wrote read/write hit/miss tests for clean paths in the baseline design and read/write hit/miss tests for clean and dirty paths in the alternative design
- Wrote tests for the eviction and refill paths in alternative design
- Wrote the baseline and alternative design section in the report

INTRODUCTION

The purpose of this lab is to understand cache principles and to design and evaluate two cache systems. We've learned about different types of caches in class (direct-mapped and set-associative) and how spatial and temporal locality are related to caching. Well-functioning caches are necessary to improve processor performance; the cache system we created will be used in the final lab when implementing a fully-functional single-core and multi-core processor. We were tasked with implementing a direct-mapped cache for the baseline design, and a two-way set-associative cache for the alternative design. The alternative design required some slight modifications to the baseline design in both the control module and the datapath/hardware module. We have thoroughly tested and successfully implemented both designs.

Upon evaluation of the two caches, we noticed that the performance of each cache is largely dependent on the memory access pattern that exists in the program. Both the direct-mapped and set-associative caches take advantage of spatial locality in the same way (they each load a line of four consecutive words). The advantage the set-associative cache has over the direct-mapped cache is that it performs better with temporal locality. In the evaluation case that expresses temporal locality, the set-associative cache has a miss rate of 20%, while the direct-mapped cache had a 100% miss rate. In the first two loop evaluations, however, where temporal locality is not expressed,the two caches perform relatively similarly. In fact, in one case, the direct-mapped actually outperforms the set-associative cache slightly. However, we conclude that, since there is little additional hardware cost, only slight modifications to the baseline, and nearly negligible increase in cycle time, the set-associative cache is often the preferred cache design.

BASELINE DESIGN

The baseline design for this lab is a direct mapped cache with 16 Bytes cache lines using write back write allocate policy. The cache is designed for a total capacity of 256 bytes with 16 cache lines.

In a direct mapped cache structure, as seen in **Figure 2.1**, , the cache is designed having multiple sets with a single cache line in every set. The set is determined by the index bit derived from the cache request address.The new data replaces the memory block in the cache if it was previously occupied. The longer cache lines help us harness spatial locality, resulting in an increased hit rate compared to shorter cache lines.

The basic approach in designing the baseline is by disintegrating the datapath, control unit and parent module into three different files. The datapath and control unit communicate with each other using control and status signals and are connected together in the parent module.

The datapath for the baseline design is shown in **Figure 2.2**. The tag bits derived from cachereq address are compared with the tag bits associated with the set from the tag array. If the tag matches, then there is a cache hit and the data memory is accessed using associated set bits. Else there is a cache miss and request has to be sent to the memory to refill the cache line. We started off with implementing the datapath for the baseline design using the existing modules from the available library.

We followed an incremental approach in implementing the finite state machine shown in **Figure 2.3** i.e. starting with the init data access and hit paths followed by more intricate refill and evict miss paths. In addition to having the FSM logic in the control unit, we also put in our valid and dirty arrays. The valid array is written with '1' only when the data is valid. Therefore, this is initialised with '0'. The dirty bit is written to '1' whenever there is a write because the cache is updated but not the memory which makes the cache line dirty.

The most fundamental state in our design is the Tag Check State where we take the decision whether there's a cache hit or miss. The hit signal is set only when there is a tag match and the data is valid.The read/write hit paths are fairly straightforward controlled by the read/write enable of the data array which is set to 1 if there's a read data access and write data access respectively. We also make sure to write the dirty array when there is a write operation by enabling the dirty array in write data access state.

In the case of read/write miss, we decide our path based on the cache lines' dirty/clean status. As per the write back write allocate policy shown in **Figure 2.4** , a cache line should be evicted and the memory should be updated before refilling the cache line. The refill state is fragmented into three states i.e. refill request, refill wait and refill update. In the refill request state, the cache simply requests the memory for data and in the refill wait stage, we notify the memory that the cache is ready for a refill. This is done using the memreq and memresp signals. In the refill update stage, we actually write the data array received from the memory and resetting the valid bit equal to 1.

We then implemented the eviction path which is taken on a miss and when the cache line is dirty.The evict state is again divided into three states i.e. evict prepare, evict request and evict wait. In the evict prepare state, we enable the eviction register and access the tag

and data array to identify what to evict. This is followed by sending a request to the memory in the evict request state. In the evict wait state, the cache notifies the memory that it ready again and resets the dirty bit to 0 since the cache line is no longer dirty.
In direct mapping i.e. our baseline design , each memory block is mapped to exactly one block in the cache. It is fairly simple technique and the mapping scheme is easy to implement. The FSM is written in such a way such that every path is very well defined, it helped us gain an understanding of the hits and miss paths in depth.
However, the disadvantage lies in the simplicity itself. Since there is a one to one mapping between the memory block and cache block, we can't take advantage of data temporality since it will cause a lot of evictions rising because of conflict misses leading to an increased miss rate.

ALTERNATIVE DESIGN

The alternative design for this lab is a two way set associative cache , as shown in **Figure 3.1,** with the same capacity as that of the direct mapped cache designed earlier. The Set associative cache can be imagined as a n by m matrix. The cache is divided into 'n' sets and each set contains 'm' cache lines. A memory block is first mapped onto a set and then placed into any cache line of the set.As we know in the baseline design, each block of memory maps to a single location in the cache. Therefore, if two different blocks map to the same location in cache and they are continually referenced,the two blocks will be continuously swapped in and out i.e. there will be more evictions increasing the number of misses and thus the miss penalty cycles which results in an increase in the Average Memory Access Latency (AMAL). With a two way set associative mapping, if two different blocks map to the same location and they are continually referenced, we need to evict every time as we now have one block of memory mapping to two locations in the cache. We need to evict when more than two blocks map to the same location in the cache. This reduces the miss rate and penalty and hence the AMAL.
We took the baseline design as a point of reference to start with the alternative design. The baseline datapath was modified as shown in Figure 3.1 . To implement the two way logic , the tag array is duplicated and the number of blocks is reduced from 16(16 cache lines) to 8 per way or tag array (8*2=16 cache lines). Each tag array has its own enable signals. Another comparator is added to compare the data from either of the way and the cachereq_addr which gives us two "tag_match" status signals. Another mux is added just before the mkaddr block which helps us decide which way should the data be selected from. This mux is controlled by the "way" control signal.
The Finite State Machine for the alternative remains the same as that of the baseline. We need to add additional logic to select the way to be accessed. We also need to incorporate a replacement policy in our alternative design. The need of replacement policy comes to decide which way has to be replaced in case both the ways are full. We use the LRU policy i.e. Least Recently Used which makes use of temporal locality. A LRU Cache organizes items in order of use, allowing you to identify an item that hasn't been used for the longest amount of time. LRU caches support fast cache accesses and fast cache updates. To implement this, we use an additional register file "use_array" to track the LRU bit in the cache.
We also need an additional valid array and dirty array corresponding to tag array 0 and 1. The implementation of the way logic depends on whether we get a miss/hit. Incase of a hit, the way should be simply be dependent on the tag_match and valid bit. If we get a miss, the way should be updated opposite to that of the LRU bit which means if the least recently used bit is 0, way 1 should get evicted as there is a possibility that in the near future way 0 can be accessed again as we need to make use of temporal locality. The hit logic now makes use of both the tag match signals and valid bits.
We used a similar incremental approach for our alternative design as well i.e. implementing the hit paths followed by the miss clean and dirty paths. The hit paths are implemented in a similar fashion to the baseline design making use of the way logic described above. For the miss paths, where the cache line is clean we move on to the refill request, wait and update states. We now need to update the logic in refill update stage to assign the way bit which depends on the valid array outputs. We need to make sure that way 0 is the preferred way and way 1 is only accessed if way 0 is full. The valid bits should also be updated here. The eviction paths need to closely track the LRU status to select the way to be evicted which depends on the logic described above. In the evict update state, the dirty bits should be updated depending on the way evicted.As soon as a way is evicted, the cache line is no longer dirty.
We implement the same long line cache consisting of 4 words in a single cache line to make use of spatial locality. Another thing to mention is, the word is selected using the 16 bit Word Byte Enable signal in the data array. It lets us choose the words to be written to at a single time.
Our alternative design becomes a good way to implement cache associativity as not much hardware is added to the datapath and control unit makes use of an additional use logic to implement the FSM. This aims to reduce the miss rate by reducing the number of evictions and making one more way available for the same mapped location.

TESTING STRATEGY

We were provided with a functional-level (FL) testing framework that could verify the validity of the tests we wrote for our actual implementations of the cache. Since the FL model is essentially memory, we couldn't test whether certain accesses should have been hits or misses; but the FL model is useful for ensuring that the tests we write expect the correct data outputs.

Our overall testing strategy consists of many *directed* and *random* test cases. We did not write unit tests because our cache designs involve finite state machines, and it would be difficult to test individual units inside the cache. Rather, we wrote and executed directed tests for different possible paths through the FSM of each cache as we incrementally added functionality (states) to the state machine. These tests are rather simple and short, since they were only designed to test a certain path or two through the state machine. Once we finished adding states, we would directly test more complicated instruction sequences that take multiple paths throughout program execution. Finally, we added tests that pseudo-randomly assign inputs or impose source/sink delays.

Additionally, we used a Design-For-Test approach to aid in testing certain features of the cache. Specifically, we added a special "INIT" transaction which writes a word into the cache and accordingly updates the tag, valid, and data arrays. This is specifically helpful in testing the relatively simple read/write hit paths without having implemented the more complex miss paths and state logic.

Through various directed tests written both incrementally and after finishing each design, we were able to write transaction sequences that would (collectively, as well as individually) use all possible paths in the FSM. We wrote generic tests that worked for both the baseline and alternative designs, as well as tests only for a specific cache. These "specific" tests were necessary because, obviously depending on the transaction sequence, direct-mapped caches will miss when set-associative caches hit, and vice versa. An easy example of this is a sequence of four reads alternating between two different addresses that use the same index in the cache line. In the direct-mapped cache, obviously all four reads will miss. In the two-way set-associative cache, the first two reads would miss, and the second two reads would hit. On this principle, we wrote many tests that verified that the baseline design was indeed a direct-mapped cache, and that the alternative design was indeed a two-way set-associative cache.

We also directly tested correct cache function with banking, though these tests were quick to develop as we often only needed to change the number of banks to 4 in the test file.

Finally, with directed tests, we tested each design's ability to handle different types of misses: compulsory, conflict, and capacity. We specifically test conflict misses for each design, because the set-associative cache will "take twice as long" to have conflicts. Additionally, through testing capacity misses, we were able to stress-test the entire cache.

For the "random" categories, we randomized many inputs and aspects of transaction sequences. Refer to **Figure 4.1** for a comprehensive list of random (as well as directed) tests/categories. Testing sequences with random source and sink delays is necessary to ensure that the wait and idle stages are functioning/transitioning properly. Also notice that several random test cases are developed incrementally. Particularly, first we tested simple address patterns with only a single access type with random data. From there, we randomized the access types, and after that, the address pattern. These randomized tests help increase the odds of finding possible bugs by testing many more transaction combinations than we had explicitly written in directed tests, though we are also confident that our directed tests cover a wide range of cases as well. Additionally, these random tests can demonstrate whether spatial or temporal locality are being utilized.

The testing process went hand-in-hand with the design process; we often used our tests to debug our design. We actually used an evaluation test in our FL test file to debug a small bug that went unnoticed for awhile, and even passed the evaluation completely! We wrote a test to essentially mimic the 2-D loop access for the set-associative, and eventually found and fixed the bug. Through extensive testing and using gtkwave and the line trace output, we were able to locate and correct many bugs. The smaller the bugs we found, the more confident in our designs we became after correcting them. Upon successful completion of our thorough testbed, we are confident in the correctness of our designs.

EVALUATION

Once we had a reasonable confidence in the implementation of both designs, in order to verify the functionality our designs, we ran the provided three loop functions against our baseline and alternative models. **Figure 5.1** shows the performance stats of our cache designs at a glance. The numbers in this figure played a significant role in evaluating the correctness and performance of our models. To fully explore our performance gains of our alternative design, let's explore each one of the provided benchmarks and more.

loop_1d: This function simply populated the memory with integers ranging from 0 to 99 in sequential order and address, then iterated over each one time. With 4 words loaded, each miss was followed by three hits, and this pattern would repeat for all numbers. Baseline offered 25% miss rate, fully taking advantage of spatial locality. Alternative design offered a similar behavior. This test did not highlight any of the performance benefits of alternative design, because its structure was designed to take advantage of locality.

loop_2d: This benchmark was an addition to loop_1d, where it iterates through each number 4 more times. Our alternative design still resulted in a 25% miss rate because of the LRU replacement policy, while baseline miss rate reduced to 19%. This function is a step towards exploiting a temporal locality, but because of the size of the original array, the baseline design was able to reach a lower miss rate than alternative design. Since cache had capacity to hold 64 numbers, there was a partial overlap when loading integers from 64 to 99. This meant that only some of the cache lines were refilled. However, if we were to run the same test on 64, 128, 256, ... (multiple of cache capacity) integers, baseline design would also result in a 25% miss rate.

loop_3d: This benchmark was the first test where we notice the performance benefits of alternative design, because this function allowed for leveraging of temporal locality. Test was very simple, a stride function, where every integer requested from memory always fell in the beginning of the cache line. Also, we were reading form two sections of the memory separated by a large chunks. This allowed for the data from the first sector of the memory remain in the cache, while loading the data from the other sector. In the next iterations, the read request from the first sector of memory always resulted in a hit. Thus remarkably reducing the miss rate to 20%, while baseline design resulted a compulsory miss on every request, making the miss rate 100%.

loop_fifty: This benchmark is much like the loop_3d model. The only difference is that it leverages some spatial locality along with temporal locality. As before, integer requested from memory fell in the beginning of the cache line, but before moving on, it also requests another number from the same cache line, thus reducing the miss rate by half in baseline. This has the exact same effect on the alternative design, meaning the miss rate was simply halved, since now we have additional hit for every miss.

loop_hundo: This benchmark tests the scenarios where we get no visible improvements between the designs while also operating at the worst miss rate of 100%. This is another scenario of stride, where each stride would refill the same cache line, meaning no temporal locality to exploit. Every cache request compulsory misses in baseline. Although it has two sets of data in each line, alternative design also falls victim to this pattern. Since we never request previous values, it will always miss.

With outlined performance benefits in the figures below, the question shifts to at what costs are we gaining such improvements. To analyze this we must look at a few factors:

Area: The extra hardware required for the alternative design is a multiplexer and an additional comparator and a bunch of wires for control signals. Tag array is split into two sectors. Without additional hardware costs, except few wires and thorough logic signals from control table, we can conclude that area did not increase by much relative to the baseline design. With the potential decrease in AMAL, this cost is negligible.

Energy: The alternative design required minor updates in the datapath and a bypassing logic in control unit. The addition energy cost of operating the alternative design is minute relative to the performance benefits. In terms of man hours, the alternative design required an additional 9 hours after the thorough development and testing of the baseline design. This means that thorough implementation and testing that which optimizes the baseline design is the only overhead to gain performance.

Cycle time: In terms of cycle time, we must notice that there is an addition of a multiplexer in series to the critical path. Although, alternative design reduces the miss rate by a considerable margin, it performs equally badly in its worst cases as the baseline design. This is shown in loop_hundo benchmark pattern. Only in scenarios where the miss rates match between the designs, longer cycle time will cause worse performance for the alternative design.

Overall, our alternate design proved to be more effective than the baseline design in a variety of memory access patterns. We ran the above patterns against each design in order to collect more detailed statistical analysis. The results are shown in the **Table 5.1**. It is important to note that miss rate is highly correlated with the AMAL, since miss penalty weighs heavily on the average. **Figure 5.2** shows the respective miss rate for each benchmark pattern. Needless to say, **Figure 5.3** displaying AMAL for each design highly correlates to **Figure 5.2**.

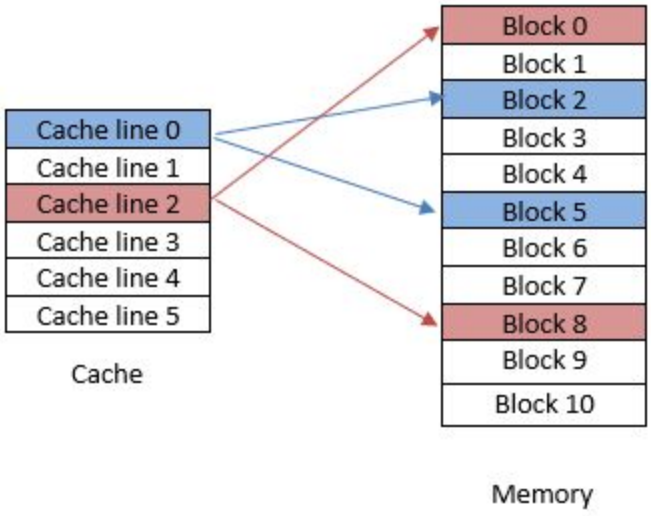

Figure 2.1

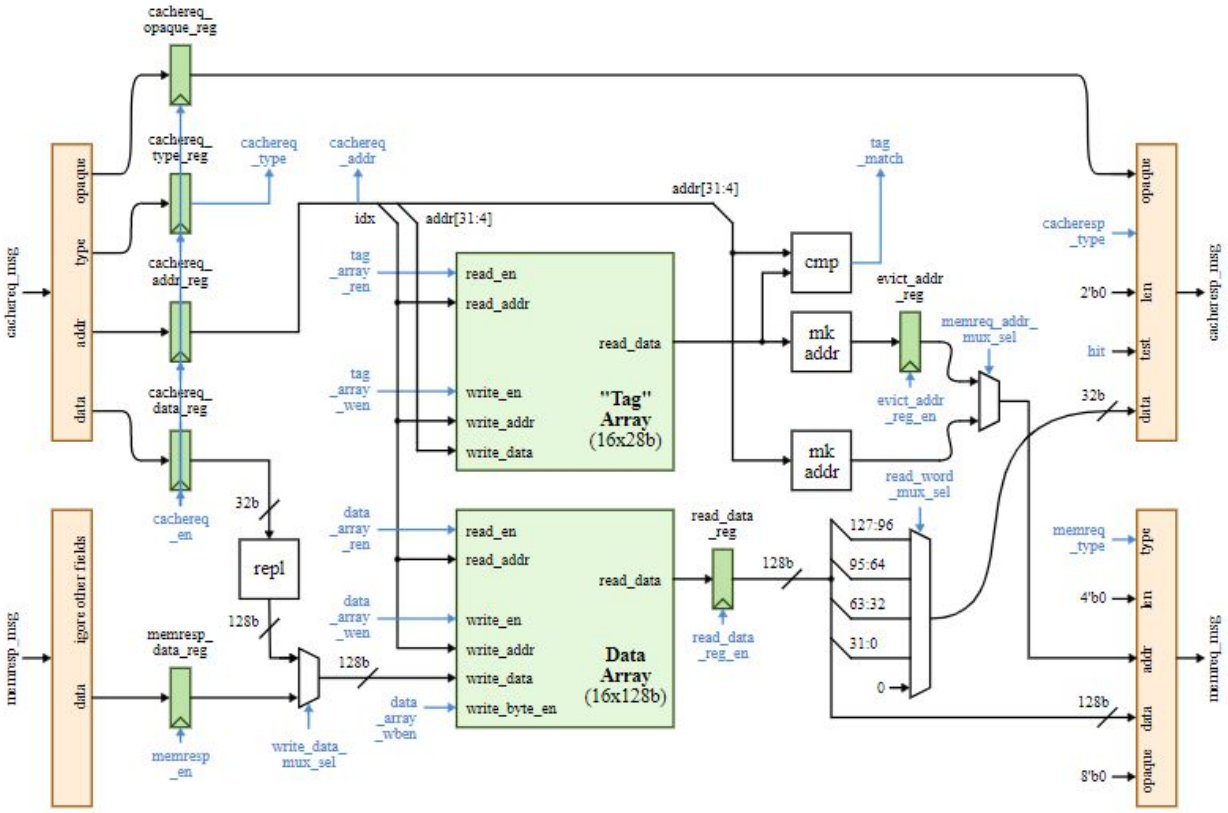

Figure 2.2

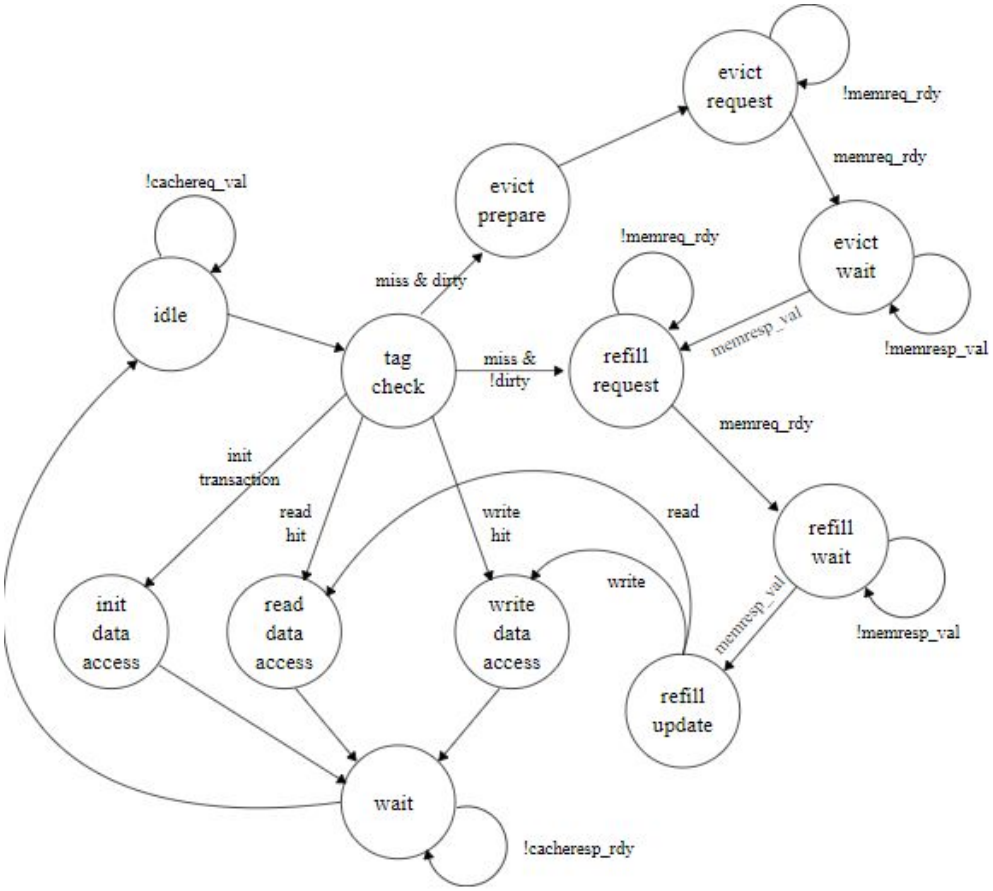

Figure 2.3

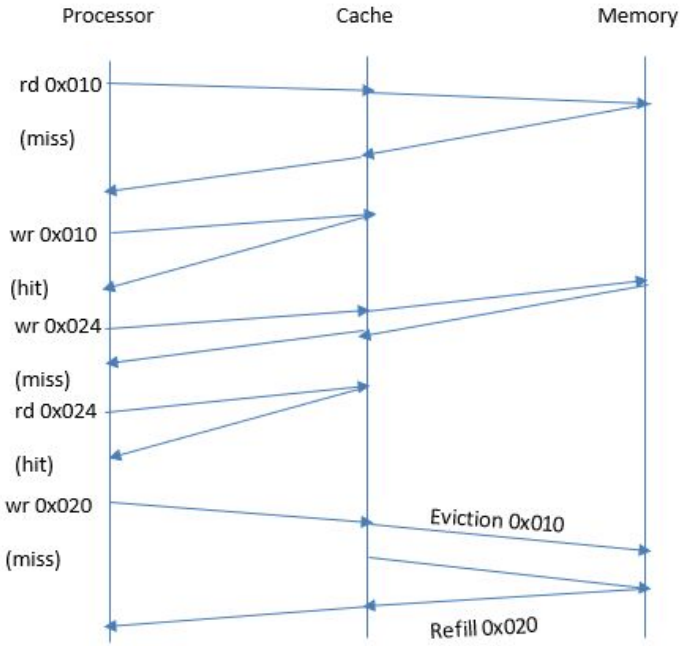

Figure 2.4

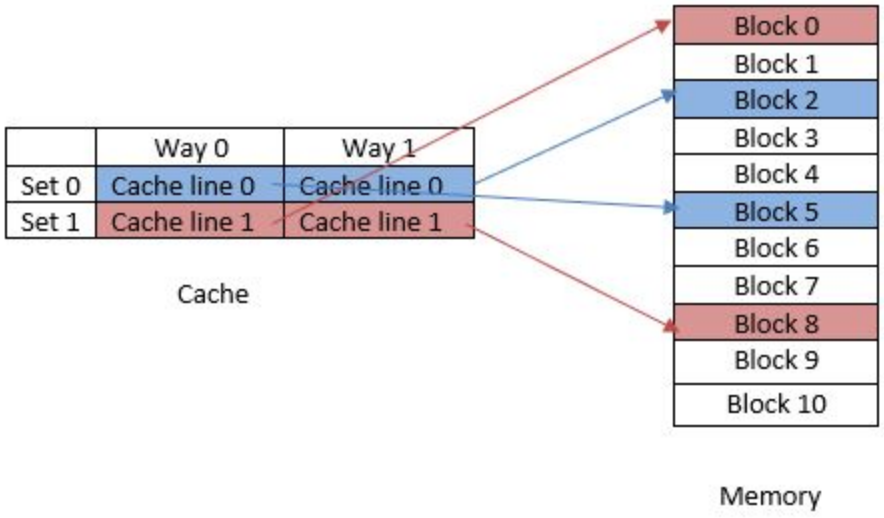

Figure 3.1

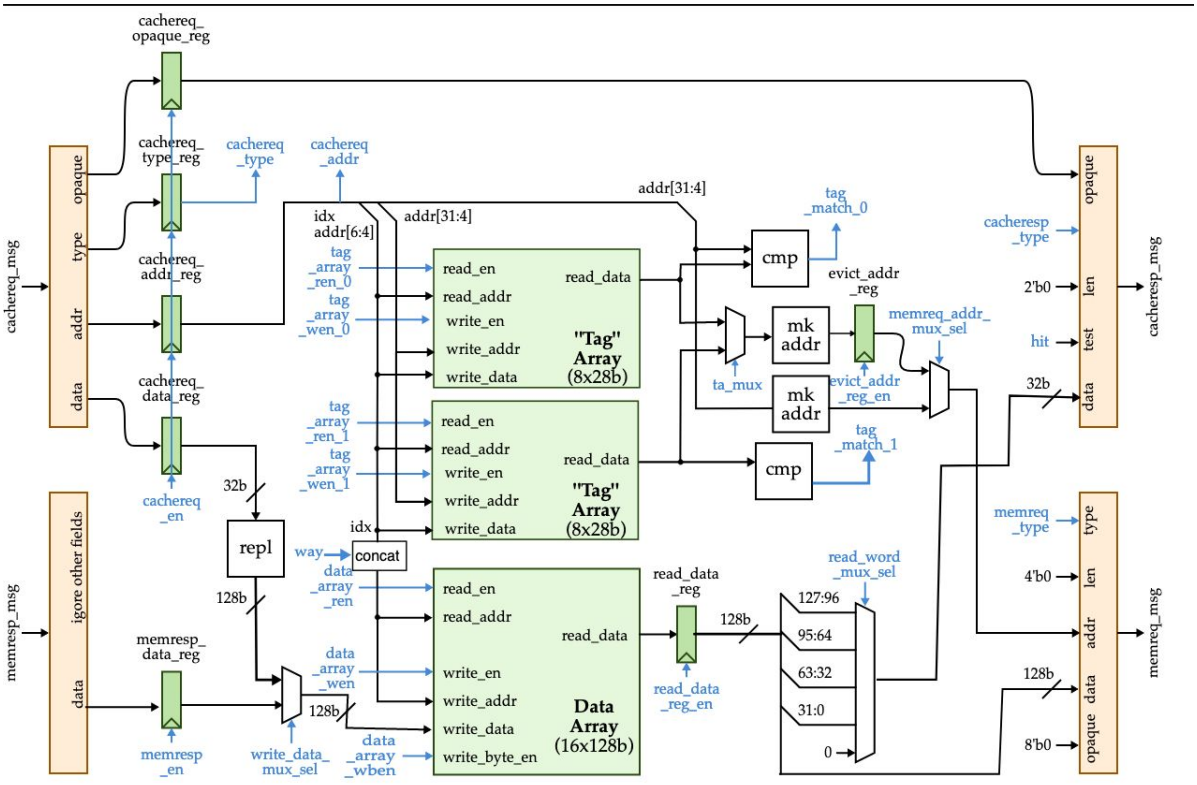

Figure 3.2

Section	Tests
Directed Tests (Baseline and Alternative)	Read hit path (clean/dirty) Write hit path (clean/dirty) Read miss & refill path (no eviction, and eviction) Write miss & refill path (no eviction, and eviction) Capacity miss test and tests that stress entire cache Conflict Misses 4-bank cache tests LRU replacement policy test other various corner cases
Random Tests	Random source/sink and memory delays (using above test cases) Simple address patterns, single request type, with random data Simple address patterns, with random request types and data Random address patterns, request types, and data Unit stride with random data Stride with random data Unit stride (high spatial locality) mixed with shared (high temporal locality)

Figure 4.1

```
Results from running simulator

   Impl  Pattern    Miss Rate
   ------------------------
 - base  loop-1d         0.25
 - base  loop-2d         0.25
 - base  loop-3d         1.00
 - base  loop-fifty      0.50
 - base  loop-hundo      1.00
 - alt   loop-1d         0.25
 - alt   loop-2d         0.25
 - alt   loop-3d         0.20
 - alt   loop-fifty      0.10
 - alt   loop-hundo      1.00
```

Figure 5.1

--impl	--pattern	num_cycles	num_requests	num_misses	miss_rate	amal
	loop-1d	976	100	25	0.25	9.76
	loop-2d	4232	500	97	0.194	8.464
base	loop-3d	2161	80	80	1	27.0125
	loop-fifty	1241	80	40	0.5	15.5125
	loop-hundo	2701	100	100	1	27.01
	loop-1d	976	100	25	0.25	9.76
	loop-2d	4876	500	125	0.25	9.752
alternate	loop-3d	689	80	16	0.2	8.6125
	loop-fifty	505	80	8	0.1	6.3125
	loop-hundo	2701	100	100	1	27.01

Table 5.1

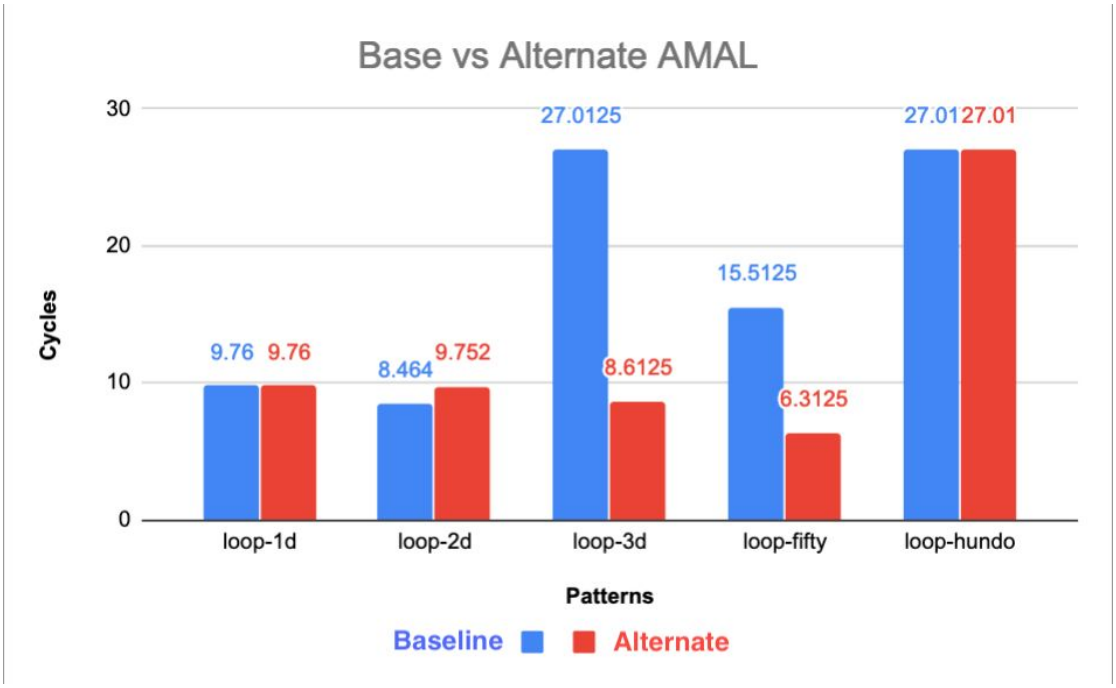

Figure 5.2

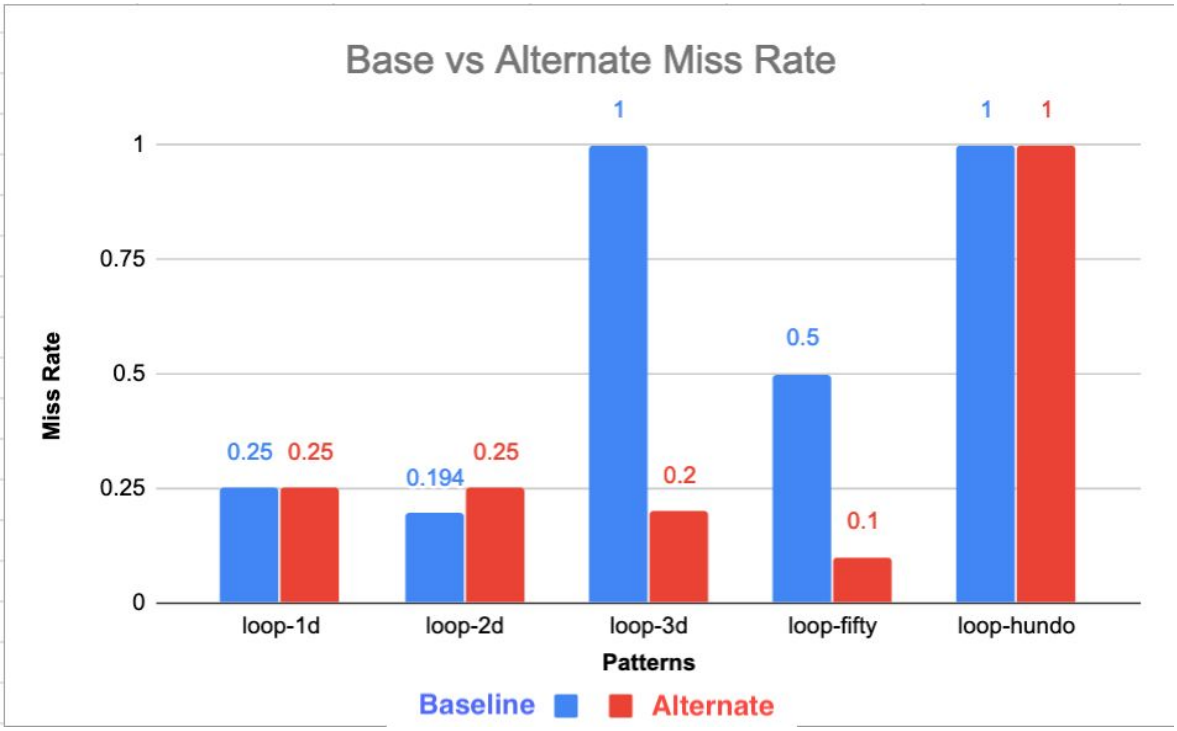
Base vs Alternate Miss Rate
Miss Rate
1
0.75
0.5
0.25
0
0.25
0.25
0.194
0.25
1
0.2
0.5
0.1
1
1
loop-1d
loop-2d
loop-3d
loop-fifty
loop-hundo
Patterns
Baseline
Alternate

Figure 5.3