Optimal Hardware Implementation of Multi-Level Cache Based on Software Simulation

John Gierlach, Robert Bach, Harrison Lipton, Bradley Racey, Michael Cruz, Delroy Jordan, and Taylor Laird

Department of Electrical and Computer Engineering, University of Central Florida, Orlando, Florida, 32816-2450

***Abstract* — This project will attempt to analyze the optimal miss rates of different multi-level cache configurations and design a hardware implementation of said configurations to see how they impact power, timing, and utilization on a benchmark FPGA. There will be two main branches of the project which include a high-level cache simulator that will determine the optimal miss rates made in Java and Python and a hardware implementation that will be made using Verilog, SystemVerilog, and Vivado to see the hardware specifications. The main takeaway of this project is to determine the hardware costs and drawbacks of high-performance multi-level cache designs.**

***Index Terms* — HDL, RTL, SystemVerilog, Vivado, Java, Multi-Level Cache, etc.**

I. INTRODUCTION

Metrics regarding the performance of a specific cache architecture tend to include measurements such as the number of read misses, write misses, miss ratio, number of writebacks, and other measurements that can be directly correlated with the overall performance of the cache from a traditional perspective. This analysis, while useful, does not consider other metrics impacted by the design choices made when designing a new cache architecture. Some other metrics that could be used to classify a cache architecture’s performance include hardware utilization and power utilization.

In order to build a cache balanced around all of these performance metrics, a more complex analysis of cache architecture must be completed. This analysis should include a balance of the classical cache performance metrics as well as the metrics posed here like hardware utilization. A cache is a widely used component of the memory hierarchy designed with the goal to reduce the idle time of the processor it has been added to. In order to pair a given processor with a cache architecture, all of these metrics should be considered.

By considering all of these metrics when designing a cache architecture for a processor, the architecture can be more well-suited for the application for which it was designed. This paper will cover an analysis of the highest-performing simulated caches based on the research completed. With the highest-performing simulated caches determined, this information should be used to determine the impact of the cache configuration (replacement policy, block size, associativity, inclusion policy) on the proposed cache metrics. This experimental data regarding the correlation between the two types of cache metrics will be used to determine the tradeoffs and benefits of using certain types of cache configurations over other types.

II. SOFTWARE SIMULATION

*A) Simulator Design*

The cache simulator used for determining the highest-performing cache architecture was written in Java. This simulator accepts a series of command line arguments that are used to specify the cache architecture desired for simulating. These metrics include block size, L1 size, L1 associativity, L2 size, L2 associativity, replacement policy, inclusion policy, and the desired trace file for simulating [1]. The available replacement policies for simulation include the FIFO, LRU, LIFO, and MRU policies. The available inclusion policies for simulation include inclusive and non-inclusive policies [2].

Using Java, the simulator was designed with modularity in mind, allowing for each cache level to be defined as a separate object with its own methods for managing the current contents of the cache. This modularity also allows for the development of additional replacement policies by inserting a call to the method for the replacement policy. Modularity is essential to successful simulator design as it should be easy to add and remove components without affecting the other components of the simulator. This design principle has been maintained throughout the Java simulator for the cache and provides the ability to easily add additional inclusion or replacement policies as needed.

III. SOFTWARE SIMULATION RESULTS

When testing varying configurations with our software simulator, we opted for choosing values that had a high probability of yielding the best-ratioed results. As such, with our published testing, we stuck with constants throughout each round. Cache and block size are also determined to be powers of two while associativity was determined through previous base cases to determine the constant used. Note that the baseline consideration for these files varied slightly though there are constants among them being: L1 Cache Size of 1024 KB, L2 Cache Size of 8196 KB, and a block size of 16 bytes.

*A) Changing Replacement Policy*

The first experiment performed on the cache was comparing the possible miss rates between replacement policies of FIFO and LRU as optimal is impractical when working with the hardware side, and working with a non-inclusive policy. Our constants include 2-way L1 associativity, 4-way L2 associativity, 1 KB L1 cache size, 8 KB L1 cache size, and a 64-byte block size.

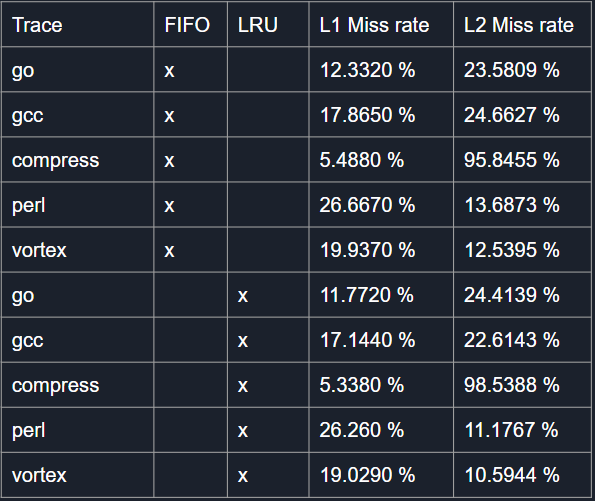


Figure 1: Replacement Policy Experiment Results

The results pictured in Figure 1, in comparison to the baseline configuration for the matching traces, we found that 64 bytes allowed for a well-balanced ratio with our configuration to provide miss rates lower than those of the benchmark configuration. In terms of the LRU replacement policy, compared to the base results for gcc of 16.0020% miss rate in L1 and 37.13191% in L2, results of the experiment yielded a 0.07134x decrease in performance for our L1 cache though the L2 cache resulted in a massive 1.642x increase.

Results from the go trace baseline were a 13.169% miss rate for L1 and a 74.7437% miss rate for L2, while our results achieved a 1.119x increase in performance in L1 and a whopping 3.06x increase for the L2 cache. Compress trace gave us little way for L2 improvement at a 1% increase through the L1 cache miss rate dropped from a 21.163% miss to a 5.338% miss rate. Almost 4x increase in performance. The Perl trace baseline resulted in the L1 cache at 22.356% miss having a 0.1746x decrease in performance and the L2 cache being a 15.1682% miss rate resulting in a 0.4317x. Lastly, vortex produced a baseline of 12.301% L1 and 22.5835% L2 cache miss rate, while the new configuration gave 19.029% L1 and 10.5944% L2 resulting in .547x decrease in performance but L2 increased performance of 2.13x.

On average, baseline performance for the L1 miss rate is 17.2266% and 46.9286% L2 while our averages for our custom inputs resulted in L1 having a 15.9086% miss and L2 being 31.3487%. This means, for the LRU policy, we had a net increase in performance of 0.0813x for L1 and 0.497x for L2.

Results for FIFO had similar trends with go resulting in an increase of 0.0813x L1 and 2.14x L2. Gcc resulted in a decreased 0.0828x L1 and L2 at 0.58x increase. Compress gave a 2.856x increase in L1 and a 0.0358x increase in L2. Perl produced a .124x increase on L1 cache and L2 at .2297x on L2. Finally, Vortex produced a 0.504x decrease in L1 and a 0.909x increase in L2. The average miss rate in L1 and L1 for FIFO resulted in an increase of 0.069x in L1 and a 0.486x increase in L2.

The results for our replacement policies showed that while both policies showed improvement from baseline cases, LRU proved more effective though slightly on average.

*B) Changing Cache Size*

The second experiment performed on the cache simulator to gather results about the miss ratio of the simulated cache focuses on the effect of changing the size of the L1 and L2 caches while holding all other simulator parameters constant. The simulation was completed across all trace files for two different configurations of cache size: 1KB/4KB and 4KB/16KB where the size of L1 is the first capacity and the size of L2 is the second capacity. This simulation also uses the non-inclusive policy, a block size of 64 bytes, 2-way L1 associativity, and 4-way L2 associativity. Figure 2 below contains the results for this series of simulations.

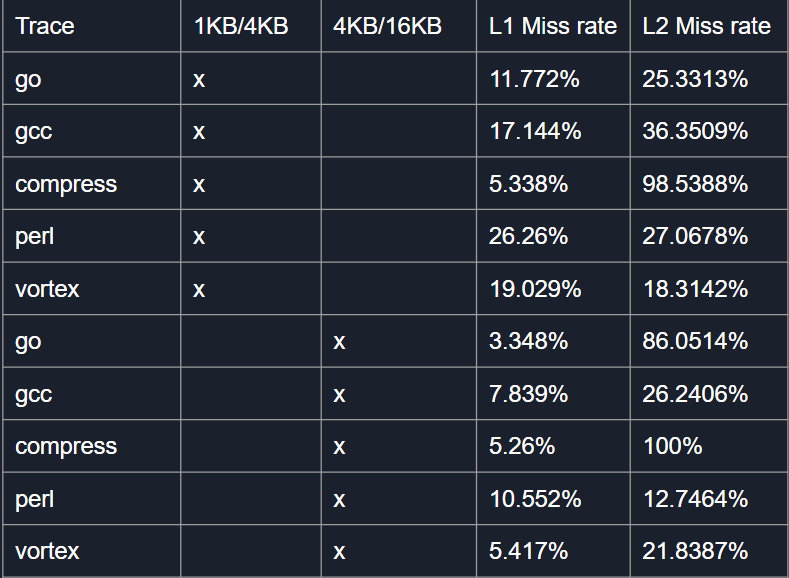


Figure 2: Cache Size Experiment Results

When examining the above results for how the miss ratio for both levels of cache changes as the size of the cache levels both change, a clear correlation between the first and second set of simulations can be observed with respect to the L1 cache miss ratio. Between the simulation using a 1KB/4KB design and the simulation using a 4KB/16KB design, the L1 cache consistently has a lower miss ratio for all trace files tested. For the go\_trace, increasing the size of the L1 cache improved the miss rate by a factor of 3.52x according to the results above. Continuing for the remaining trace files, an improvement factor of 2.19x, 1.01x, 2.49x, and 3.51x for the gcc\_trace, compress\_trace, perl\_trace, and vortex\_trace respectively. While the magnitude of the improvement for each trace file varies greatly, each L1 cache consistently performs better, with respect to miss ratio, when provided with additional cache storage. This intuitively makes sense as providing the L1 cache with additional capacity reduces the chance of a conflict when placing a block in the cache, thereby reducing the chance that the block needed has been evicted since it was last accessed.

In the case of L2, the results are not as clear as some L2 caches had an improvement in performance while others had a reduction in performance as a result of the increase in cache capacity for both L1 and L2. Looking at each trace file, moving from a 4KB L2 cache to a 16KB L2 cache resulted in performance improvements of -3.39x, 1.39x, -1.01x, 2.12x, and -1.19x for the go\_trace, gcc\_trace, compress\_trace, perl\_trace, and vortex\_trace respectively. These results are interesting as increasing the size of the L2 cache does not always result in a decrease in the miss ratio for every trace file. This implies that there must be some other factor that is not being considered here causing the L2 cache miss ratio to increase for certain patterns of memory accesses. One possible explanation for this increase in the miss ratio in the L2 cache is the nature of the trace files used in the test. The pattern of memory accesses in each of the trace files may have resulted in additional conflicts that did not occur with the smaller cache as the set index changes for each address traced.

*C) Changing Associativity*

The third experiment involved testing varying levels of associativity. At this stage, our constants were set to implement: the LRU replacement policy, 1 KB L1 cache, 8 KB L2 cache, a block size of 64 bytes, and using the non-inclusive policy. With this iteration, we chose to test each trace with 4-way and 8-way associativity for L1 and L2 cache respectively. The second instance of this test was set to using 2-way and 16-way associativity.

When referring to the changes in the L1 cache it is important to note that the associativity of the L1 cache is higher in the first test than in the second. This would explain why for almost all of the trace files the L1 cache had a lower miss rate in the first test as reducing the associativity would increase the miss rate of the cache. For the go, gcc, perl, and vortex trace file the second test had a miss rate increase of 2.105x, 1.082x, 1.07x, and 1.22x respectively. This may not seem like a massive increase, but when implementing a functional cache system many of these small changes could add up resulting in vastly different cache performances. The interesting thing to note in this test is that for only the compress trace the reduced associativity actually resulted in a decrease in the miss rate. The miss rate with 4-way associativity was 5.64% while the miss rate with 2-way associativity was 5.338%. While this is only a difference of .302% it is strange but is likely due to the lower amount of blocks that need to be accessed resulting in a lower miss rate.

For the L2 cache, the results were mostly the same. In the second test, the associativity of the L2 cache was higher and for all of the trace files except for the compressed trace, the miss rate results were better when the associativity was higher. The results for the go, gcc, perl, and vortex trace showed that with higher associativity the results were 2.45x, 1.2x, 1.1x, and 1.29x respectively. Once again for the compress trace file tests the results showed that the miss rates were lower with the lower associativity. Once again this is likely due to the configuration of that specific trace file working better with the lower amount of accessed blocks.



Figure 3: Associativity Experiment Results

IV. HARDWARE IMPLEMENTATION

*A) HDL and RTL Description*

To implement our cache design in hardware, we utilized a register transfer level (RTL) based language known as SystemVerilog. SystemVerilog was created in 2002 by the company Accellera with the intent to standardize the hardware description language (HDL) Verilog and to implement system verification functionality to the language. In 2005, SystemVerilog adopted the IEEE 1800 standard and is now one of the most widely used languages for RTL design. While SystemVerilog boasts multiple features that Verilog does not have such as interfaces and classes, the feature that prompted us to use this language rather than Verilog was the inclusion of additional data types such as floats. The existence of a floating point data type in the language made it possible to calculate the hit ratios of the hardware simulations with much higher accuracy.

The difference between an RTL and other types of programming languages such as Java or Python is that RTLs are used to define hardware structures. When RTL code is synthesized, it uses look-up tables based on microchip architectures supported by the HDL’s integrated development environment (IDE) to lay out flip-flops and logic gates to create a user-specified design.

Vivado 2022.1 was used as the IDE to create and execute our RTL code. Vivado was created by Xilinx to support their field programmable gate arrays (FPGA) architectures and can run code written in Verilog, SystemVerilog, or SystemC. The main benefits of using Vivado over another RTL IDE are the information that can be represented through waveform simulations and the ability to generate estimations on hardware power usage and physical space on the chip the design will occupy.

*B) Multi-Level Cache Architecture*

The hardware architecture was first constructed to include two forms of inclusion policies, replacement policies, and write policies. There are four modules used in this design: cache\_top, cache\_tb, cache\_params, and cache\_engine. There will be a series of inputs and outputs to check the total number of hits, misses, reads, writes, and contents in each level of the cache. There will be input signals that can dictate which policies will be active within the cache and which address and operation will be accessed or inserted. Lastly, there is a parameter module that can allow one to test the cache size, block size, and associativity of each cache level. There will be a brief description below of how all of the modules work and what they contain. Once the first draft of the collective multi-level cache architecture was designed, it will be pruned for specific policies to measure the timing, power, and utilization of three different cache configurations on a benchmark FPGA **(**XC7A200T-1FF1156C**)**.

*C) Cache\_top Design*

The cache\_top module takes in the following signals: clock, reset, write policy, replacement policy, inclusion policy, 48-bit input address, 8-bit input operation, L1 statistics, and L2 statistics. Each cache level can hold 32-bit address values that another unit could access or call from. The cache\_top module design serves as an upper-level module that can allow for future access to the specific cache lines of each level being used. Its job is to instantiate the cache\_engine module, which will be named cache\_block Once the cache\_engine has completed its computation, it will send back the Ll & L2 statistics alongside the current contents of each cache. Figure 4 shows the high-level schematic of how cache\_top is connected to cache\_engine.

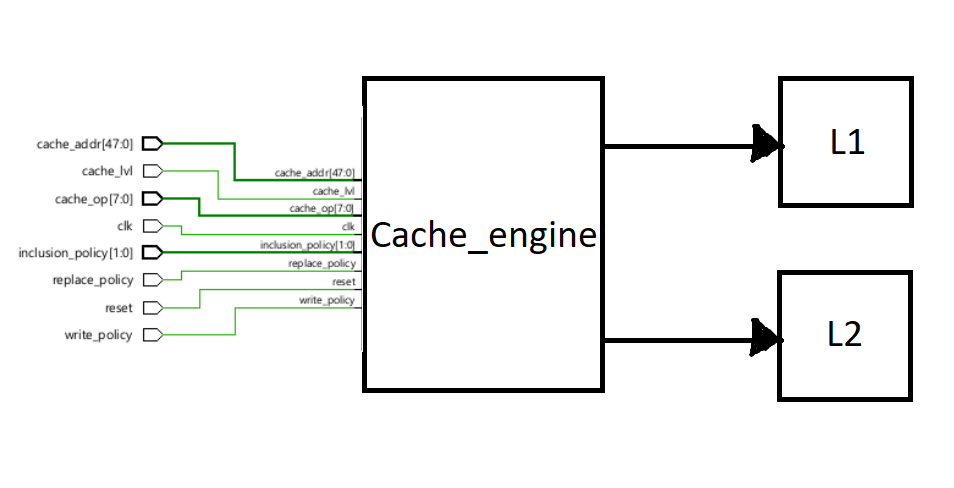


Figure 4: Cache\_top Schematic

*D) Cache\_engine Design*

The cache\_engine module is used to verify and test the functionality of a variety of different cache configurations. This includes inclusion policies, replacement policies, write policies, and cache metrics. The inclusion policies tested were non-inclusive and inclusive, which were decided by a 1-bit wide input signal (inclusion\_policy). If inclusion\_policy was set to 0, then the inclusive policy was run; non-inclusive otherwise. The replacement policies tested were LRU and FIFO, and the write policies tested were write-through and write-back. Similarly to how inlcusion\_policy was implemented, the signal replace\_policy and write\_policy dictate which type of policies will be used. For replace\_policy, the LRU is used if the signal is 1; FIFO otherwise. For the write\_policy, write-through is used if the signal is 0; write-back otherwise. To construct the multi-level cache functionality, a finite state machine (FSM) was used. There are seven states within the FSM: IDLE, READ, SEARCH, SHIFTFULL, SHIFTEMPTY, CACHEHIT, and DONE. Figure 5 shows the FSM in a diagram.

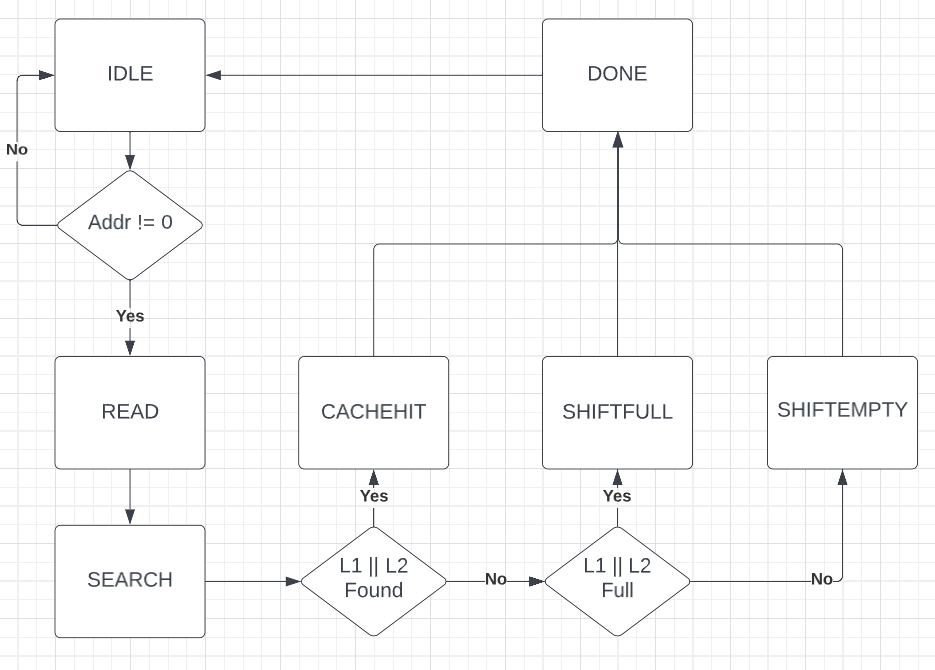


Figure 5: Multi-level Cache FSM

*E) FSM: IDLE State*

The IDLE state was used to keep the cache\_engine from performing any computation so long as the current input address from cache\_top contained no significant input (0 valued address). While in IDLE, if there is a new address that contains content (not 0), then the FSM will move into the READ state.

*F) FSM: READ State*

The READ state, the tag, and the set number for L1 & L2 will be stored in a register to be used later on within the FSM. While in the READ state, if the write\_policy is set to write through and the current operation is a write, then the number of writes to L1 and L2 will increment. Once all the operations of the READ state have been accomplished, the FSM will then go into the SEARCH state.

*G) FSM: SEARCH State*

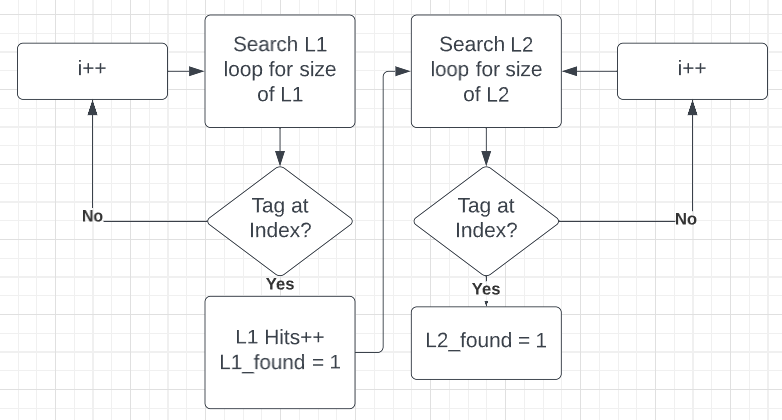
The SEARCH will search through the contents of the current set number of each cache to determine if the current tag of a given cache exists within the set of interests. If the tag exists in either level of the cache, a 1-bit flag (L1\_found or L2\_found) will be asserted to note that a tag has been found in either L1 or L2 respectively. In addition, if the replace\_policy is signaled to LRU and ifL1\_found or L2\_found are flagged, then the current index at which the tag was found in the cache will be stored in an integer(L1\_lru\_index or L2\_lru\_index) for the corresponding cache the hit was marked in. If there is a hit in L1, then the hit count for L1 will increment, but for L2 it will only increment if it isn’t found in L2 since that address will be passed to the CPU. While in SEARCH, if either found flag is asserted, then the FSM will jump to the CACHEHIT state; however, if no found flag is asserted, the FSM will jump to SHIFTEMPTY or SHIFTFULL depending if the last index in the current set in either cache is filled. Figure 3 shows a diagram of the logic within the SEARCH state in FIFO. Figure 7 shows the same diagram but for LRU.   
  


Figure 6: SEARCH | FIFO State

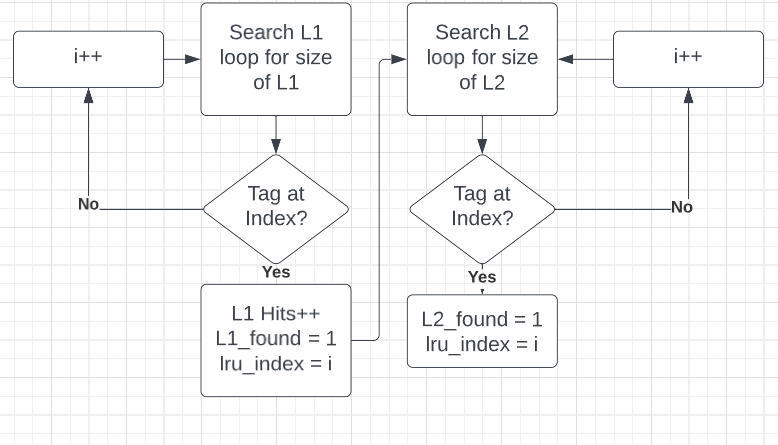


Figure 7: SEARCH | LRU State

*H) FSM: CACHEHIT State*

The CACHEHIT state will be processed with the logic needed to compute LRU hit cases, FIFO hit cases, and how inclusion policies can share the data between the two caches. If our current replace\_policy is set to the LRU, then there will be a shifting operation to put the tag that was hit to the front of the current set. If the current replace\_policy is FIFO, then the cache\_engine will go on to the inclusion policy rules. While in CACHEHIT, if L1 is a miss and L2 is a hit, then the inclusion\_policy will need to be taken into account when moving/inserting addresses. In this case, L2 hits will be incremented since it wasn’t found in L2 and L1 will receive a miss. The inclusion policies under a hit act are the same between non-inclusive and inclusive; the only case where they differ is on an L2 eviction which will cause back-invalidation for inclusive policy (removal of tag in L2 and L1). Once this process is completed both found flags will be reset to 0 and the FSM will then jump to the DONE state. Figure 8 shows the CACHEHIT state for FIFO. Figure 9 shows the CACHEHIT state for LRU.

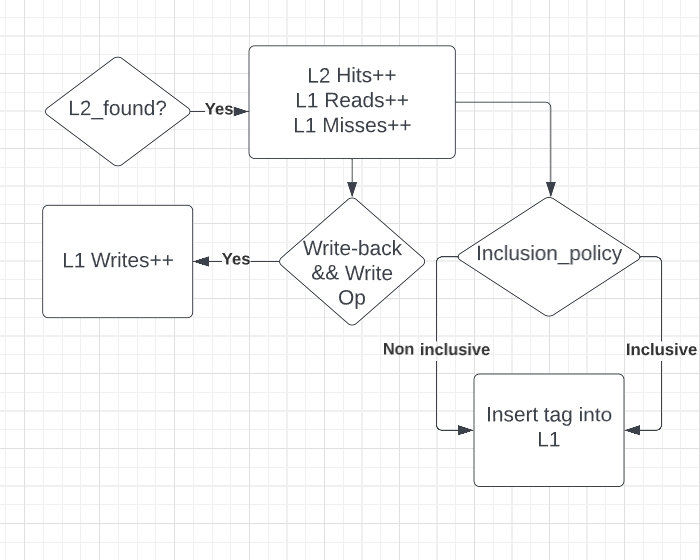


Figure 8: CACHEHIT | FIFO State

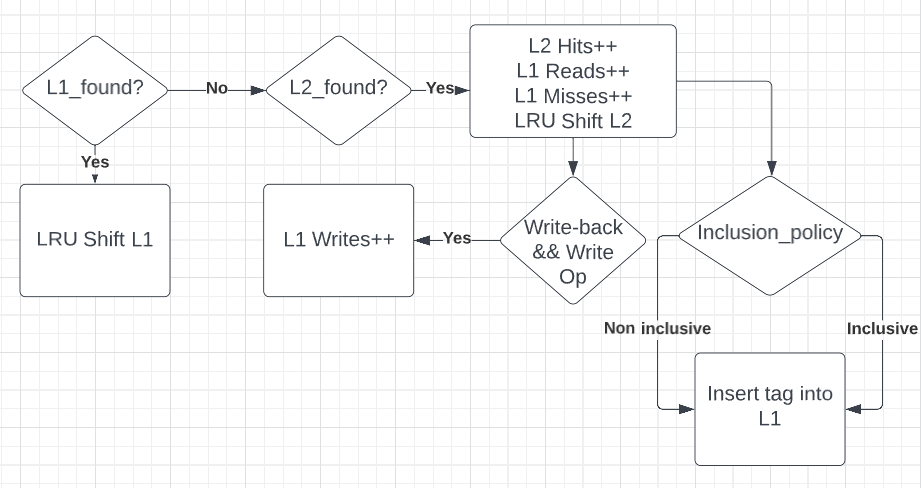


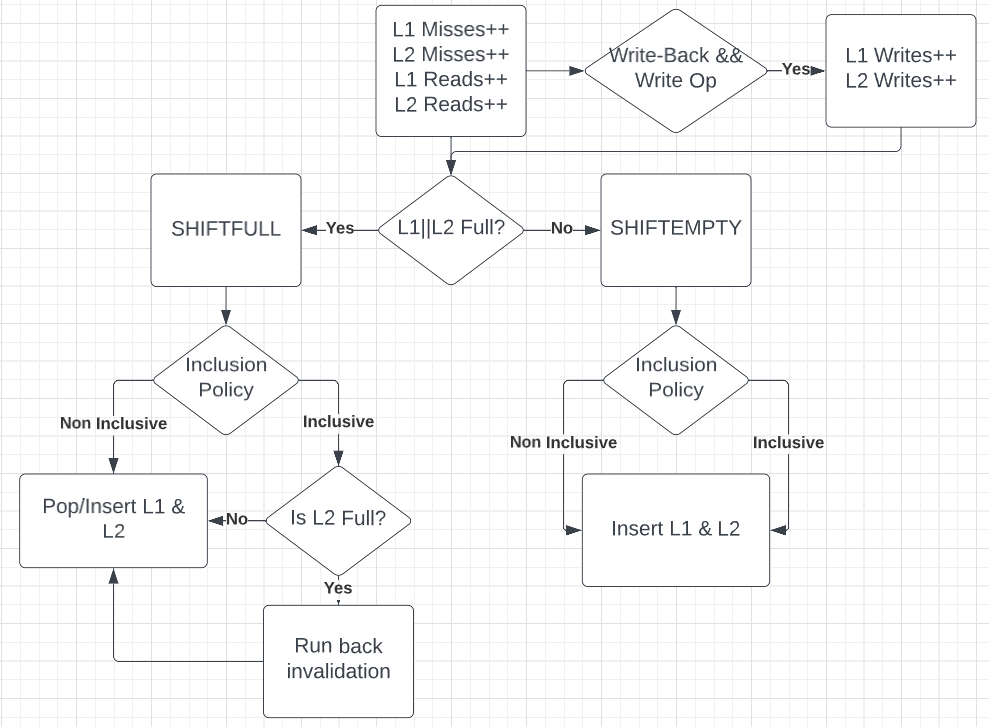
Figure 9: CACHEHIT | LRU State

*I) FSM: SHIFTEMPTY State*

The SHIFTEMPTY state happens when there isn’t a found flag asserted for either cache level and neither current cache set is full. Within this state, each cache level will increment a miss, and if the current write policy is write-back & the current operation is a write, then the writes will also increment. Regardless of which inclusion policy is chosen, the operation is the same, shift the old tag further in the set and insert the new tag at the front of the set. Once this operation is complete, the FSM will jump to the DONE state.

J) *FSM: SHIFTFULL State*

The SHIFTFULL state happens when there isn’t a found flag asserted for either cache level but one of the cache levels has a full set. Similarly to the SHIFTEMPTY state, the misses and writes will increment due to the write\_policy/operations and found flags not being asserted. If the cache is full under a miss condition, the last tag address within the current set will be popped out and the new tag will be added to the set while shifting the other tag addresses down. If the inclusive\_policy is set to non-inclusive, then the two levels of the cache act independently of each other when addresses are evicted, so there is no need to consider back-invalidation. In contrast, if the inclusion\_policy is set to inclusive, then it will need to be logically created to deal with back invalidation. In this case, if L2 is full and is looking to evict an address, then that address is stored in a temporary 32-bit register back\_inval. Then, back\_inval is compared to every tag within L1’s current set to see if it exists, if it does, then that tag within L1 is also evicted [4]. This will cover the case of back-invalidation, and once the operation is complete the FSM will jump to the DONE state. Figure 10 shows the SHFITFULL and SHIFTEMPTY states.

  
Figure 10: SHFITFULL & SHFITEMPTY states

K) *FSM: DONE State*

The DONE state is a simple state that will jump to the IDLE position marking that all current cache operations with all policies considered have been completed. Finally, all of the data collected will be pushed out of cache\_engine to cache\_top to be used.

*L) Cache\_tb Design*

The cache\_tb is a test bench file that enables testing for specific inputs to ensure the cache is functioning properly. In this case, there are five trace files that will be tested: compress, go, gcc, perl, and vortex. All of these trace files will include 100,000 addresses and operations that will be inserted into the cache\_top module to verify all is functioning as intended with the cache. A waveform GUI interface will be generated to determine the state of registers and cache levels during each insertion of an address and operation.

*M) Cache\_params Design*

The cache\_params module allows for simple declarations of L1 & L2 cache parameters to be used through the cache\_top, cache\_engine, and cache\_tb modules. There will be five inputs needed: BLOCKSIZE, L1\_CACHESIZE, L1\_ASSOC, L2\_CACHESIZE, and L2\_ASSOC. Once these values have been established, the number of sets will be calculated for both cache levels: L1\_NUMSETS and L2\_NUMSETS. This will be a Verilog header file that allows the other modules to access these parameters. This concludes the overall multi-level cache architecture.

V. HARDWARE IMPLEMENTATION RESULTS

*A) Input Dataset*

For the hardware implementation, our team created five dataset files that could be imported. Each dataset file contained 100,000 addresses and a specifier for if the address was to be processed as a read or write operation. The operation specifier was at the start of a new line and stored as a hexadecimal ASCII value representing “R” or “W”. The address itself followed the operation specifier on the same line; it was stored as a 48-bit hex address.

Similar to the software simulator, one of the objectives of the hardware was to calculate the miss rates for the L1 and L2 caches while varying the replacement policy, cache size, and cache associativity. In addition to this, information on hardware power usage, space-on-chip utilization, and thermals was also collected.

The hardware has implemented LRU and FIFO replacement policies. These policies were tested by holding the L1 associativity at 2-way, the L2 associativity at 4-way, the L1 cache size at 1KB, the L2 cache size at 8KB, the inclusion policy as non-inclusive, and the block size at 64B. The next parameter to be tested is the cache size; the hardware has the ability to define the sizes of both the L1 and L2 caches. In total, two cache size configurations were tested for the hardware, the first was 1KB for L1, and 4KB for L2 while the second configuration was 4KB for L1 and 16KB for L2. During the cache size tests, L1 associativity was held at 2-way while L2 was held at 4-way. The block size was 64B, the inclusion policy was non-inclusive, and the replacement policy was LRU. Finally, the cache associativity was tested. Similarly to the cache size, the associativity could be defined for L1 and L2 separately; we also performed two tests with different values for this parameter. The first test was with L1 being 4-way and L2 at 8-way, the second test was with L1 at 2-way and L2 at 16-way. For these tests, the L1 cache size was 1KB, the L2 cache size was 8KB, the replacement policy was LRU, the inclusion policy was non-inclusive, and the block size was 64B.

*B) Hardware Test: Replacement Policy*

*C) Hardware Test: Cache Size*

*D) Hardware Test: Associativity*

VI. CONCLUSION

Based on the data collected, it can be inferred that the increase in cache size, associativity, and change of policies can have an effect on hardware utilization and power consumption. The software simulation was able to give a theoretical best-case scenario for miss rates between the caches, but it can be seen that increasing the cache size can be great for performance, but costly for power and utilization. If the replacement policy is set to LRU then one could expect better hit rates between both caches, but compared to FIFO, the number of LUTs used seems to grow drastically, so if the area is a concern, limit the amount of LRU functions used within the hardware. Lastly, increasing the associativity can give decent results, but after increasing it too much, one will see diminishing returns on the miss rate and see an increase in power/thermals. Overall, all of these multi-level cache implementations can be resourceful, but when designing a cache, it's important to keep in mind these trade-offs of performance, power, and utilization.

REFERENCES

[1] A. Barnwal, “LRU Cache Implementation,” *GeeksforGeeks*, 20-Mar-2023. [Online]. Available: https://www.geeksforgeeks.org/lru-cache-implementation/. [Accessed: 24-Apr-2023].

[2] Solihin, Yan (2016). *Fundamentals of Parallel Multicore Architecture*. Chapman and Hall/CRC. pp. 146–150. ISBN 9781482211184