Optimized Processor Project Report ECSE 425 - Group 7

Malkolm Alburquenque
260562740
Rohit Gupta
260564446

Joel Lat
260580913
Will Nguyen
260638465

***Abstract*—The objective of this project is to optimize a pipelined processor through cache implementation. The pipelined processor integrates two different ways for cache optimization - a split cache optimization and a unified cache optimization. These improvements remove the constant memory access delay of the processor by placing an intermediate memory between the processor and memory to access data in a shorter time. When integrating these cache optimizations, we performed different testbenches to showcase the benefits of each optimization compared to the unoptimized pipelined processor. Most of the results from the testbenches of the cache optimizations provide a significant decrease in runtime compared to the unoptimized pipelined processor. This paper examines the selection of the aforementioned caches for optimization, the content of the testbenches used to test these optimizations, and the results obtained from each implementation as well as the reasoning behind those results.**

I. Design Problem

The capability of reducing the amount of processor time required to execute a program has always been an important factor in creating faster processors. However, this has proven to be a slow and tedious process, since the processor needs to execute one instruction at a time for a single CPU. Pipelining all the instructions allows the processor to execute many more instructions of a program rather than waiting for an instruction to begin executing; this results in the improvement of the throughput of the processor. Even though the pipelined processor increases the throughput, there are still some issues that can occur such as the presence of various hazards and overheads that can potentially interfere with the pipeline. One of the main overheads for the processor pipeline is using two different memories to store the variety of instructions native to a program as well as the program data from loads and stores. Since the processor fetches instructions and data through memory, the access delay becomes a relevant overhead that slows down the pipeline. This overhead is consistent throughout the execution of the program and becomes a hindrance for different instructions such as branches and loops. For the design problem, we want to reduce this latency such that it will increase the speed in which the CPU will execute programs by effectively lowering the overhead. If there are loops or branches that run through the same instructions multiple times, it can be wasteful to keep looking to memory for the storing, loading and fetching of instructions.

II. Design Solution

A. Cache Implementation

To reduce the latency from the fetching of instructions and the use of the store and load instructions, the implementation of a cache optimization seems like the ideal choice. This will provide the CPU with the common instructions and data fetches in a manner that makes them accessible through the cache adjacent to the memory. By applying a cache that is closer to the CPU, the fetch time for instructions and data are reduced since memory access is not always required. *Figures 1 and 2* refer to two different methods for caching the instructions and data.

1) Unified Cache: The first is through the use of a unified cache which both instruction and data memory read and write into and acquire the data as needed. If the content in the cache is incorrect, then the cache will fetch it from memory and then write its memory value. In both contexts we use direct mapping for the placement of data blocks in the caches for specific tags. To be specific, direct mapping ensures that cache accessing depends on the respective index values of the addresses requested from the processor.

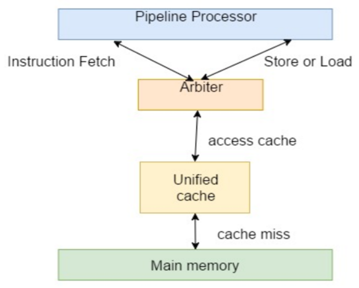

Fig. 1: Unified Cache

2) Split Cache: The other cache optimization implemented was the use of a split cache to separate instructions and data into their own proper caches for more efficient use. This allows for the smooth functionality of the cache and data as they do not interfere with each other when the appropriate data is being fetched. An arbiter component is now being used between the individual caches and the memory.This is in contrast to the unified cache where the arbiter was connected between the individual stages in the pipeline and the unified cache.

In both contexts we use direct mapping for the placement of data blocks in the caches for specific tags. Specifically, direct mapping ensures that cache accessing depends on the respective index values of the addresses requested from the processor.

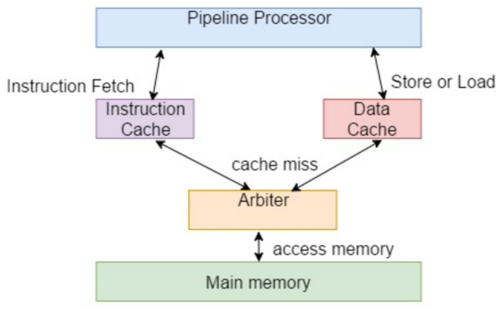

Fig. 2: Split Cache

III. Components

For the cache optimization, there are a few ways of integrating caches for the pipelined processor that result in the minimization of memory accesses which, leads to lower execution time.

One way is to implement two caches, one being for instructions and the other one for storing and loading data. By splitting up into two caches, each cache has to access the main memory that contains both the instructions and data when loading a program and storing the data when the data cache misses. If a cache miss occurs while reading or writing from the cache, an arbiter component is used to determine if the cache is allowed to request a read or write to the main memory. Since it is possible to have both caches miss, these caches cant access the main memory at the same time. The arbiter determines the cache that requested main memory access to give it permission to access, while the other cache waits for the accessing to complete. The arbiter then gives memory access permission to the cache that was waiting for the memory.

The other approach to cache optimization is to use a unified cache. By having a shared cache that is the equivalent size of the split caches elements combined, the instructions and the data can be contained in the same cache. In this unified cache, when examining the program instructions, its possible for the unified cache to contain more instructions available for retrieval if there are many that are continuously used. Meanwhile, the unified cache also contains very minimized amounts of loads or stores. For this implementation, the processor requests an instruction, load or store through the arbiter before checking the cache. The arbiter then needs to determine if the cache is available for use in order to request instructions or data.

IV. Design Approach

For the particular parameters, we wanted to make sure that there was no ambiguity when comparing the two possible optimizations to the unoptimized processor. To make the caches for both optimizations similar, the split cache has two sets of 16 slots for the instruction and data caches while the unified cache encompasses 32 slots. The address bits for the split cache include a 26-bit tag, 4-bit index and 2-bit offset while the unified cache uses a 25-bit tag, 5-bit index and also a 2-bit offset. The cache and arbiter components for both optimizations use the same components. It takes 3 clock cycles to process a cache and 20 clock cycles to access memory. When the cache misses it then has to refer to the main memory in which data retrieval is a total of 23 clock cycles in the optimized processor compared to a constant 20 clock cycle retrieval time for the unoptimized processor. For the split cache, if both instruction and store/load miss, then the arbiter needs to wait 43 clock cycles to handle both misses.

To determine if the optimizations are correct, the unoptimized processor, split cache processor and unified cache processor are put through the same test benches. Each test bench is used to determine which processor would perform the best. The unoptimized processor is used as a reference unit such that the optimizations need to be able to perform better than it. Each optimization has unique benefits over the current unoptimized pipelined processor. One of these is a program speedup through the use of numerous branches or loops, since these allow for the reuse of the same instructions to perform certain tasks. These instructions would be contained in a cache which now uses 3 clock cycles to provide the data when the cache hits instead of the constant 20 clock cycle memory access time for the unoptimized cache.

To evaluate the processors, we needed to add stall instructions in the testbenches since the data hazard implementation is not fully functional. The stall instructions would ensure that the processor performs the proper tasks. Also, since the cache has compulsory misses, we needed to make our test long enough to show that the optimization gives a faster time because the unoptimized cache will perform better when the caches are not yet filled. The first testbench we created was used to determine if the optimized caches would have better execution times than the unoptimized processor when executing a program that loops multiple times making the instructions reusable. This testbench consist of a program that executes *'addi'* operations into certain registers and then stores and loads those respective values into different registers. This program loops through sixfold. Through this implementation, the unoptimized processor would take longer to execute because of the memory access delay. Moreover, there would be enough cache hits to ensure that the optimized caches run faster.

To compare the split and unified caches, we created a testbench to determine if calling a number of common instructions greater than the size of either I/D cache in the split cache would affect the performance of the CPU such that the unified cache would run faster. We expected this outcome because the instruction cache in the split cache would miss after having more than 16 instructions in the cache, causing it to increase its execution time. We do multiple *addi* executions to the registers to fill up the cache until we exceed 16. We then integrate loops to make the split cache miss for extra instructions. This testbench loops over 15 times to test if multiple misses per iteration for the split cache will show a difference in performance between the two optimizations.

We also tried to make a testbench that would showcase a scenario where the split cache would function better than the unified cache. The goal of this testbench is to fill the data cache with addresses that will be reused while not having too many addresses to fill the instruction cache. Looping over this code would cause hits for both data and instruction calls however the unified cache would not have enough room for all the data locations and the instructions causing it to miss more often.

In our last testbench we decided to use a real-life implementation in which we determine the number of prime numbers that exist below the number 50. The processor loops through each number and determines if the number is prime and increments the counter if it qualifies. When the iteration is equal to 50, the program exits and returns the number of prime numbers (the counter value). This test is used to verify if the optimization would work for a program that can be potentially useful and not a synthesized test.

V. Evaluation

The equal cache optimization testbench was used to determine if the optimizations would provide a faster execution time for the programs in comparison to the unoptimized processor. From the results obtained from this testbench, the unoptimized processor takes 1440.5 ns to complete program execution, which is slower than both optimizations. The unoptimized processor would always fetch from the memory which took 20 clock cycles each time. After the compulsory cache misses for the first iteration of the loop, both optimized caches started performing faster than the unoptimized one since the caches started hitting and providing instructions within 3 clock cycles compared to the original 20. Using the results from the testbench, compared to the unoptimized processor, the unified cache runs 1.88 times faster and the split cache runs 1.33 times faster. These results were expected since program iterates more than once and starts using the caches. A breakdown of the execution time for each processor can be seen in *Figure 3*.

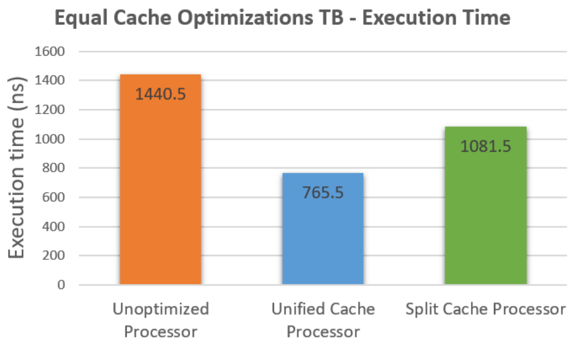

Fig. 3: Equal Cache Optimizations Testbench

The unified cache optimization testbench performed just as expected. When competing against the split cache optimization, it performs best when there are more than 16 instructions or loads/stores which are called into the split cache. This testbench aims to fill the maximum number of slots with instructions. The split cache starts to miss even after its compulsory misses because of the limited capacity of each of its caches. The split caches are unable to contain all the data from the program and start to miss when they have to deal with more than 16 instructions. However, the unified cache can accommodate more capacity to store the instructions by lowering the amount of space for the data and stores. This allows the unified cache to have a higher cache hit rate for instructions but lower for stores/loads. Since the testbench uses many instructions but a lower amount of stores/loads, the split cache performs worse/slower than the unified cache optimization. In *Figure 4*, we can see that the unified cache perform 2.26 times better than the split cache and the unoptimized performs significantly worse, once again due to the constant memory access delay.

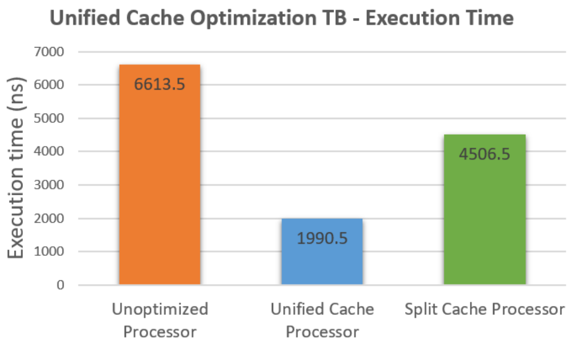

Fig. 4: Unified Cache Optimization Testbench

Due to the difficulties in implementation that caused the data hazards functionality to not be completed, it was difficult to find an implementation where the split cache outperformed the unified cache. This is because for every ”normal” instruction we had 3 stall instructions that would fill the cache to work around the problem. This meant that there are always more instruction calls to the cache than data calls making it harder to create a scenario where the split cache is best/fastest. As a result our split cache only ran 1% faster then our unified cache. We believe that if we had been able to properly implement the data hazards functionality and had allowed 75% of the instructions in the instruction memory to be stalls, the speedup difference would have been much more significant.

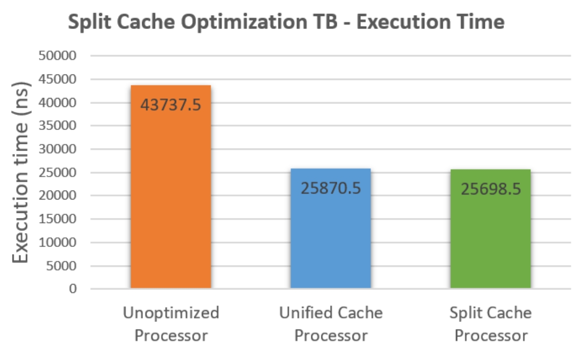

Fig. 5: Split Cache Optimization Testbench

For the final 'Prime Numbers' testbench, the results differed from what we expected. The split cache performed the worst for the prime number counting test while the unified cache performed the best. We were expecting the unoptimized cache to always perform the worst after multiple loops but in this context, it seemingly performed better than the split cache optimization. The unified cache ran 1.74 times faster than the unoptimized processor. However, the split cache executed the program 0.93 times slower than the speed of the unoptimized processor. The unified cache performed as expected since there were many jumps and reused instructions and yet the split cache was expected to exhibit a similar performance. This may, once again be caused by the pipelined processor data hazard functionality not fully being implemented. Resultingly, the program needed the insertion of stalls to run correctly. Since there are numerous stall instructions in this testbench, the instructions would fill up its respective cache rapidly due to the limited capacity of the cache. Once again this would result in continuous misses when not using the same 16 instructions that have the same addresses. When a cache misses, it takes up to 23 clock cycles and when examining the context, it makes it possible for the split cache to perform much worse than the unified cache with regard to the number of cache misses. This does not happen to the unified cache since it can modify the capacity of instruction calls up to 32 slots; however, this test does not use that many. Again, it is worth nothing that the unified cache optimization would not miss after the compulsory cache misses. By providing a larger capacity for the split caches and by removing the data hazard issue, this execution error would not appear for this test and the results would be significantly different and further favorable towards the split cache optimization.

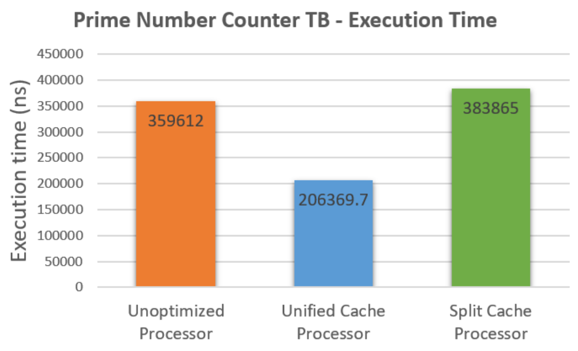

Fig. 6: Prime Number Counter Testbench

VI. Conclusion and Recommendation

If we had the chance to redesign the project, we would likely spend more time clearing any issues with the pipelined processor that we implemented before the optimization. This would have made it so that we wouldnt have needed to worry about the processor not functioning correctly and would have allowed us to focus more on the errors obtained from the optimization negatively affecting the pipelined processor. We would also have liked to redesign the processor to make it less dependable on each stage since every stage of our pipelined processor acts as a separate entity that contains all the necessary blocks for that stage. In our current context we did not take into account the possibility for cache optimization. We would replace the memory of both the fetch and memory stages with either the arbiter for the unified cache or the caches for the split caches optimization. We also would make sure that all the signals were correctly tested to get the proper results by attaching the new components.