**Lab 4 Report**
**Elise Song (eys29)**
**Katherine Zhou (kz273)**

## Introduction

In this lab, we combined all the components from our previous labs to make a single core and multicore system. The baseline design is a single core system that includes our fully bypassed five-stage processor with our variable-cycle multiplier, a data cache, and an instruction cache. The alternative design is a multicore system with four cores that uses a ring network along with the processor, multiplier, and data and instruction caches. The data cache has four banks so that we do not run into cache coherence problems, and each core has its own private instruction cache. In addition to hardware, this lab also requires us to use software to write a sorting algorithm benchmark for both the baseline and alternative design. This lab is the culmination of everything that we have learned in this semester, and aims to allow us to build a simple system using things that we already know. It allows us to see how everything we have learned fits together in practice. Our design shows the principle of modularity, for example the router is built from router units and switch units, and the network is built with routers.

## Alternative Design

The baseline design involves building a single core system along with a single core sorting microbenchmark. The single core system is made by connecting our processor from lab 2, and two instances of our cache from lab 3 together in a system. Once completed, the system is able to fetch instructions from the instruction cache, execute those instructions on the processor, and store data in the data cache. The is fully functional and we are able to test our sorting algorithm on it.

Compared to the base design, the alternative design is more complex in that it includes building a ring network to connect the cores together. Each core has a router, and each router consists of three route units and three switch units. A diagram of the router network and each router is shown below.

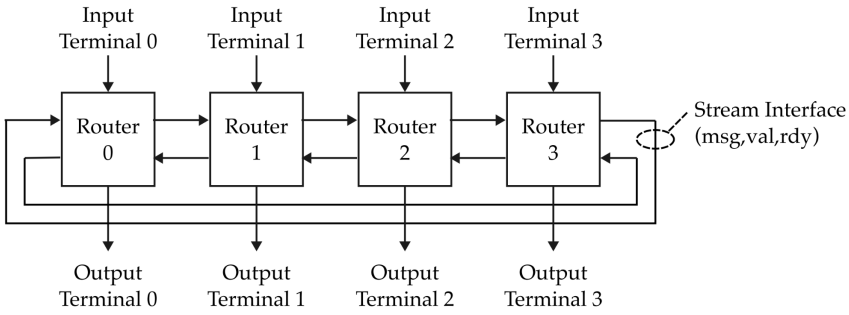


*Figure 1: Routing network diagram*

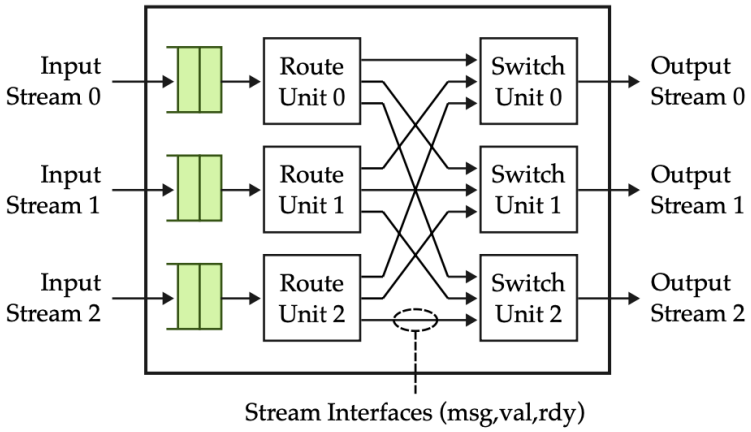


*Figure 2: Router unit diagram*

The router network allows for parallel processing, that is if one core is busy doing work, the new work can be given to a different core to execute. The route unit makes use of a routing algorithm which determines the direction of information flow between routers based on the router which the data is currently at. The route units follow a routing algorithm that chooses the shortest path to the destination router. If the destination is within two routers to the right of the current router, the algorithm sends the message to the right. If the destination is one to the left, the message is sent to the left. If the destination is the current router, the message is outputted. We decided to write the routing algorithm like this because it is able to optimize the path that is taken between routers. For example, if the data is at router 3 and has its destination at router 0, it will go directly right instead of through routers 2 and 1 like it would in fixed counterclockwise data flow.

The outputs of the route unit are then passed into the switch unit, where it is then output to the correct output stream of the router. In addition to the routing algorithm, there is also logic in the switch units. The switch units follow a fixed priority arbitration algorithm. The input stream message from port 1 is prioritized first in the case that a message from two routers away is coming from the right. Then, port 2 is prioritized since the message is also already in the network. Port 0 is prioritized last since that message just arrived.

Similar to the baseline design, there is also a software aspect to the alternative design. The sorting algorithm uses a combination of the quicksort and mergesort algorithms. Our baseline sort used quicksort, so the alternative design divides the given array into four blocks. Each of the four threads sort a block using quicksort. Then the merge step from the mergesort algorithm merges the first two blocks and merges the last two blocks. Lastly, the two halves are merged together.

**Testing Strategy**

There was an extensive amount of testing done in this lab. Our overall testing strategy made use of white and black box testing, directed testing, unit testing, integration testing, random testing, and software testing. Hardware 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. Software testing was done natively using a C compiler before cross compiling and building into assembly code.

We used white box testing to test our router and switch units. White box testing is used because the outputs depend on our specific algorithm implementation. For example a clockwise routing algorithm would give different outputs than the algorithm we implemented. Additionally, black box testing was used to test the overall systems, such as the routing network, the single core system, and the multicore system. In the systems, the output is not affected by the internal wiring. To test the single core and multi core systems, we used the processor tests from lab 2 along with other tests for the data and instruction caches.

Directed testing was used to ensure that all of the processor instructions were able to execute on our single core and multi core system. We imported the tests that were used in lab 2, and made sure to include some of the interesting mixed instruction tests so that we could ensure that these execute correctly on both the single core and multi core systems.

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 inputs to ensure that our system can handle these inputs, 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. We expect these all to pass, since they are the same tests as in lab 2. If they failed, then we went back to check for connection errors, or small network bugs.

In addition to the random testing done to ensure correct processor performance, there was also delay testing done. The importance of delay testing is to ensure that our implementations follow the latency insensitive interface that we use. 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. We use a source and/or sink delay of value 100 in addition to all of our directed tests.

We also utilized unit testing to ensure that our individual units worked properly before we connected them together. Below is a table detailing where unit testing was done.

*Table 1: Unit testing performed*

| **Component** | **Test description and justification** |
|---|---|
| Router Unit | These test cases test if the input goes to the correct output stream based on our algorithm. It was tested using multiple route ID's to ensure that the behavior is correct at each router given the desired |

| | |
|---|---|
| | output location. For example, if we are at router ID 0 and the destination is 0, it should directly output. If we are at router ID 1 and the destination is 0, then the information should move to the left to get to router 0. We also added sink and source delays when testing the route unit. 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 tests. We had either a sink delay of 100, a source delay of 100, or both. |
| Switch Unit | The switch unit was tested in a similar fashion to the router unit. Given a destination, we wanted to see that our switch unit output the correct data given our switch unit algorithm. It was also tested using multiple route ID's to ensure that the behavior is correct at each router. Similar to the router unit tests, we also used sink and source delay tests. 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 had either a sink delay of 100, a source delay of 100, or both. |
| Router | Because the route network is built up of routers, we found it necessary to unit test the router as well. These tests ensure that the router units are all connected properly in the router and that the correct output stream is getting sent given the router ID and the destination router. Again, this was tested using different source/destination router combinations. Source and/or sink delays were also included for the same reasons as stated above. |

When we put all the pieces together, we needed to perform integration tests to ensure that all the parts worked together correctly. This comes in the form of the CacheNet, MultiCoreDataCache, MemNet, and MulticoreSys tests. Many of these tests were provided for us. We moved over our lab 2 processor tests to the MulticoreSys test file as described in the directed testing section.

Software testing was done on both the single core and multi core sorting algorithms. This ensures that the sorting algorithm is not bugged before we test it using our system. Our software testing protocol mainly consists of random testing to achieve coverage.

Below is a table detailing some of the tests that were done.

*Table 2: Tests performed*

| **Component** | **Type of test(s)** | **Test description and justification** |
|---|---|---|
| SingleCoreSystem | Directed, random, delay | The Single core system was tested using our processor tests from lab 2. These tests included directed tests for specific instructions, random value tests with and without random delays, and some interesting mixed instruction tests to ensure correct behavior in some tricky cases. |
| SingleCore Sorting Algorithm | Software testing, random | Software testing was done on the sorting algorithm to ensure that it would correctly sort a given array. We found that it was easy to have small bugs in the code that caused it to fail, so debugging was done using print statements to catch where the program was breaking. This allowed us to efficiently catch the spots that were causing us errors and look closely at them to fix the bugs. We included random testing to ensure coverage so we could be confident that we are able to sort any given array. |
| Router Network | White box, black box, directed tests, delay tests | The router network was tested using white box and black box testing. White box testing was done on the individual components of the router, such as the route unit and the switch unit, because the output would depend on the specific implementation of those units. The whole network was tested using black box testing, since the output should not change based on our implementation. We used directed tests to test the network implementation, where we give it data, a source, and a destination to see that the data will be correctly output at the given |

| | | |
|---|---|---|
| | | destination. Additionally, delay tests are used to ensure the correctness of the latency insensitive interface that we implemented. We used a sink and/or source delay of value 100. |
| Cache Network, Memory Network, MultiCore Data Cache | Integration, delay, | These components were important to test for integration testing. There are times where the parts work individually, but when they are put together they do not function properly. This shows the importance of integration testing. We did this to make sure that there were no bugs that we possibly did not catch in the previous labs that leak through to our multicore system. |
| MultiCore System | Integration, delay, directed, random | The overall multicore system also makes use of integration testing, since it is putting together a four core processor, data caches, and instruction caches. We use the directed, random, and delay tests from lab 2 to ensure that our system is working correctly. These tests included directed tests for specific instructions, random value tests with and without random delays, and some interesting mixed instruction tests to ensure correct behavior in some tricky cases. The random tests allow us to achieve coverage in the range of values that we test. The delay test ensures that our implementation is following the latency insensitive interface. |
| MultiCore Sorting Algorithm | Software testing, random | Software testing was done on the sorting algorithm to ensure that it would correctly sort a given array. We found that it was easy to have small bugs or misunderstanding of the way the code should work, so debugging was done using print statements to catch where the program was breaking. We included random testing to ensure coverage so we could be confident that we are able to sort any given array. |

**Evaluation**

Quantitative analysis:

Single core analysis:

We tested our single core system on five microbenchmarks: bsearch, cmult, mfilt, vvadd, and sort. These are the same benchmarks as we have previously tested, with the addition of our own sorting benchmark, which uses quicksort to sort an array of numbers. We ran these benchmarks using the single threaded system on one core and on four cores. Below, we see the comparison for the number of cycles and the CPI on each of the benchmarks.

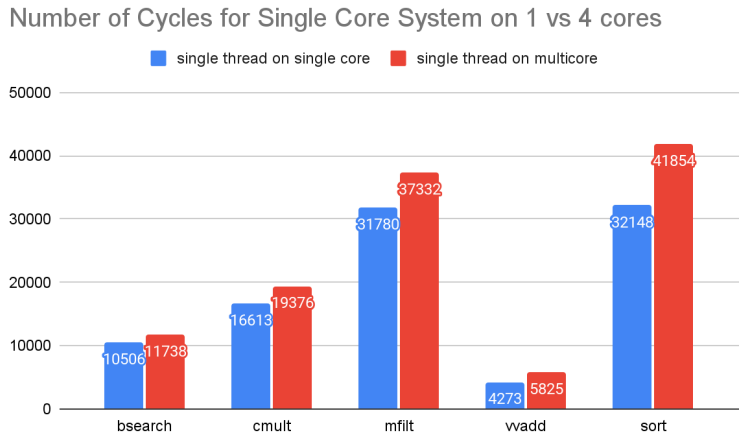


*Figure 3: Number of cycles comparison for single thread on 1 vs 4 cores*

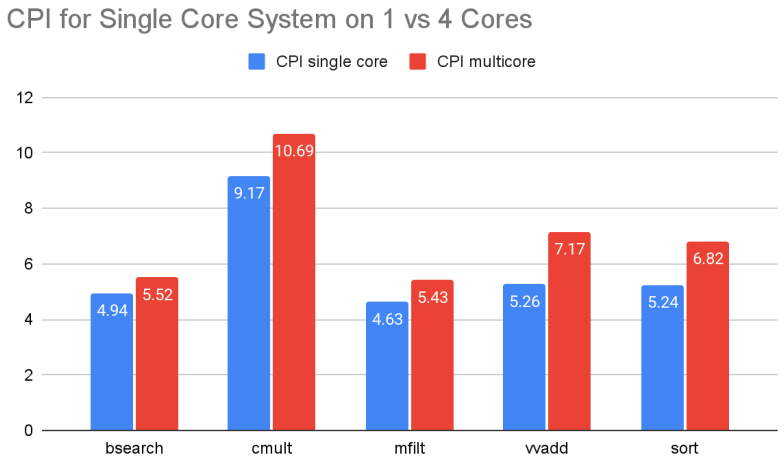


*Figure 3: CPI comparison for the single thread on 1 vs 4 cores*

As we can see the results are not the same when we run the single thread system on one core versus on four cores. This makes sense because there will be some hardware overhead when the single thread is run on multiple cores due to the addition of the router network, the cache network, the memory network, etc. We can clearly see the hardware overhead in the CPI numbers; running the single core benchmark on the multicore system requires more cycles for every benchmark tested. We see a 11.7, 16.6, 17.3, 36.3, and 30.2 percent increase in the CPI for the bsearch, cmult, mfilt, vvadd, and sort benchmarks respectively.

Multicore analysis:

We also tested our multithreaded system using five multithreaded benchmarks, bsearch, cmult, mfilt, vvadd, and sort. These benchmarks perform the same tasks as the ones in the single threaded system, but they are now written for a multithreaded system. We ran these benchmarks with our multithreaded system on four cores and on one core. Below, we see the comparison for the number of cycles and the CPI on each of the benchmarks.

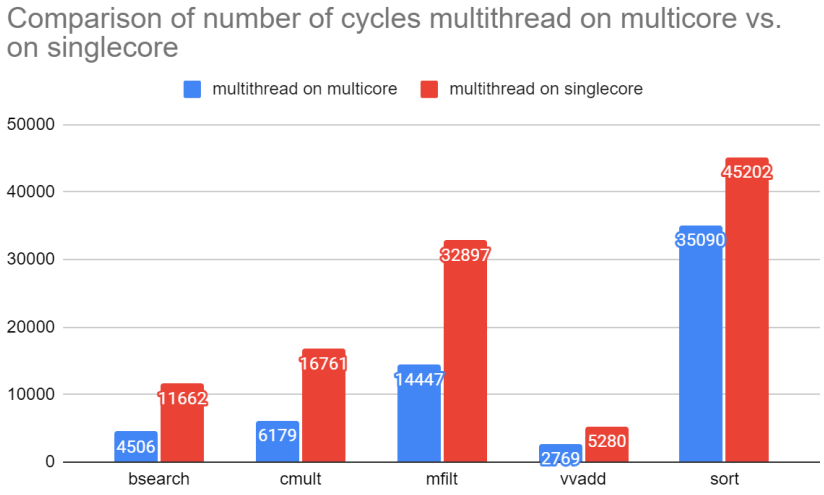


*Figure 5: Number of cycles comparison for multi thread on 1 vs 4 cores*

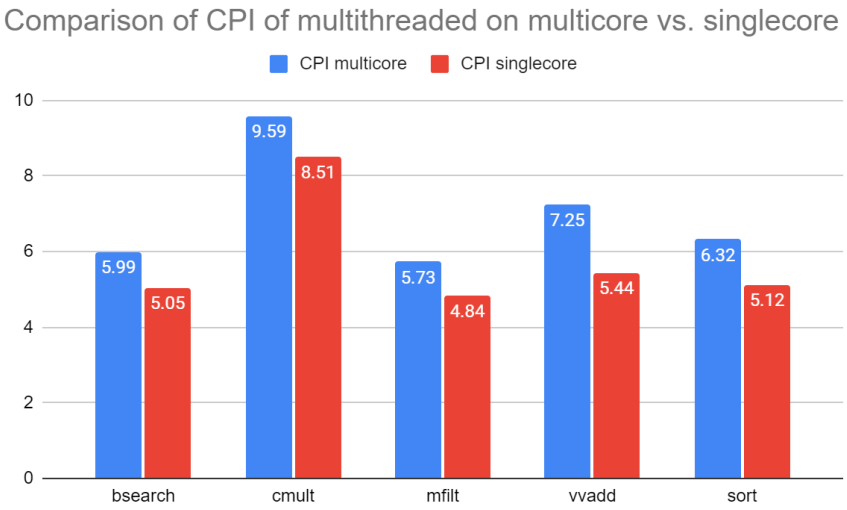


*Figure 6: CPI comparison for the multi thread on 1 vs 4 cores*

By running the multi thread system on one core and comparing its performance to when it is run on four cores, we are able to see the software overhead of writing a multithreaded program. This is shown in the total number of cycles needed to execute the benchmarks. We can see that there is a 158.8, 171.3, 127.7, 90.7, and 28.8 percent increase in the number of cycles needed to execute the bsearch, cmult, mfilt, vvadd, and sort benchmarks respectively.

Comparative analysis:

Finally, we were able to compare the speed up in each of the five benchmarks relative to the baseline design, which was the single thread on a single core. Speedup is defined as the number of cycles the baseline design takes divided by the number of cycles the alternative design takes. In this case, we have four different alternative experiments that are all being compared to the same baseline. A graph of the speedup and CPI comparisons is shown below.

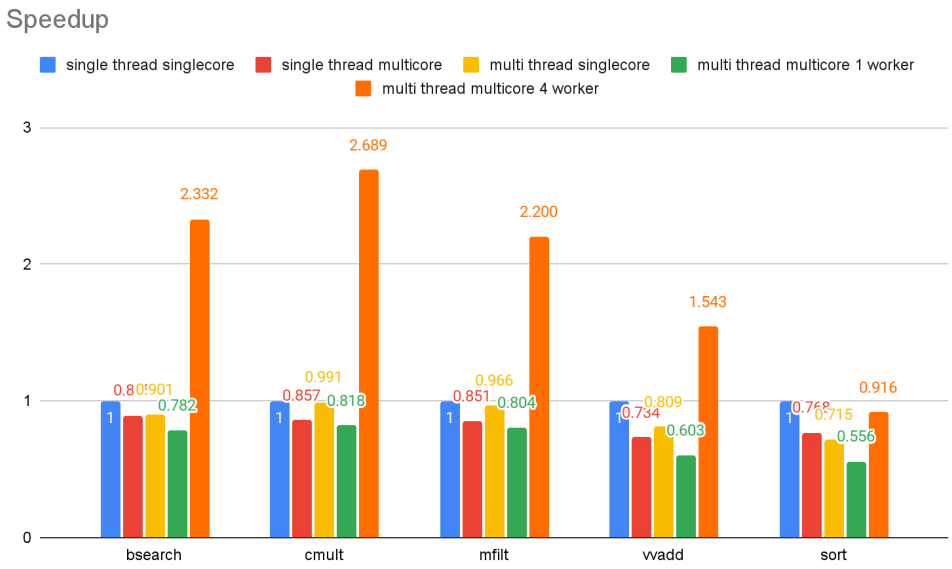


*Figure 7: Comparison of speedup of number of cycles normalized to single thread single core*

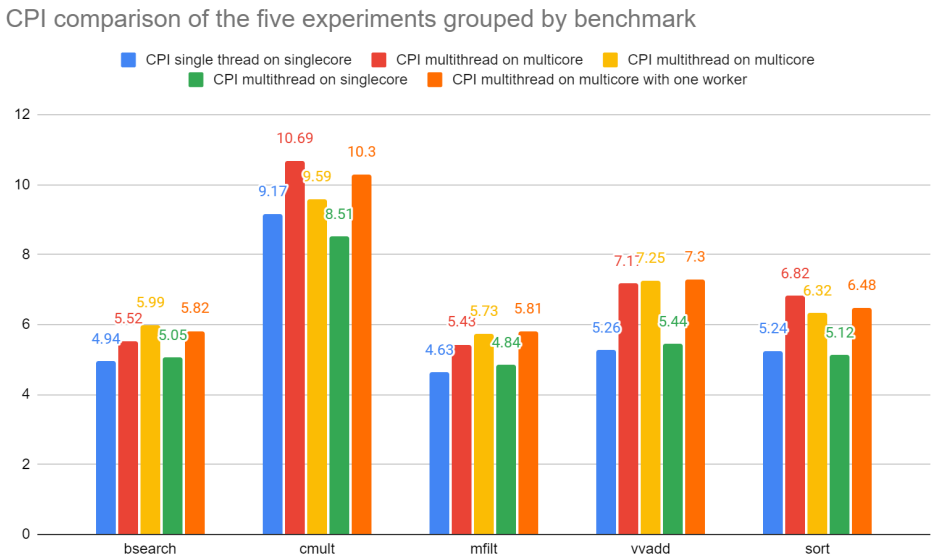


*Figure 8: Comparison of CPI of the five experiments*

Generally, we would expect to see the largest speedup in the multicore with all four cores working. As we can see from the simulations, this does usually appear to be the case. However, this is not true with the sort benchmark, and the multicore with four cores actually performs worse than the single thread on a single core. This could be for a multitude of reasons. As shown before, there is a large hardware overhead when using the multicore system due to the addition of the router network, cache network, memory network, etc. If the decrease in the number of cycles is not able to offset the increase in CPI, then we will not see a speedup, which is the case in the sort benchmark.

Qualitative analysis:

When compared to the baseline single core design, the multicore system should have a longer critical path due to the addition of the routing network and the time it would take for data to pass through the network. Thus, the multicore system has a longer cycle time when compared to the single core system. However, as we can see in the evaluations, the multicore system executes multiple instructions at once, so there are a fewer number of instructions executed for a given program. Work done on the other cores is done in parallel, so they would be executed at the same time. Therefore, the time it takes for a program to execute is dependent on

Additionally, the multicore system uses more hardware than the single core system does. The single core system consists of a processor, an instruction cache, and a data cache. In the multicore system, this is multiplied by four because there are four cores, and on top of that there is a cache net system, a memory net system, and a data cache system. Therefore, there is a large increase in area used, which also creates a lot of hardware overhead. To estimate the amount of energy increase in the multicore system, we can calculate energy consumption of the entire instruction sequence by doing

```
energy / instruction sequence = energy / cycle * cycles / instruction * instruction /
                                instruction sequence.
```

We can see from the evaluation that the multithread program on a multicore system has a CPI which is higher than the corresponding single thread program on a single core system. From the simulation, we also saw that there was on average about 30% fewer cycles in the multi thread program on a multicore system compared to the single thread on a single core system. Additionally, each cycle on the multicore system will consume more energy than on the single core system because of the additional hardware that is added. Thus, we can not conclude definitively that one system will use less energy for a given task, since it will depend on the specific program and how much of a decrease in instruction count there is when the program is converted to a multithreaded program.

Overall, there does not seem to be a clear “better” system when comparing the single core and multicore. The multicore system has a large amount of hardware and software overhead which harm the execution time and can not always be mitigated by other factors such as a decrease in the number of instructions needed to execute. The

multicore system takes up more area and can consume more energy than the single core system. However, we can see that there is, in general, if a user is running multithreaded programs on the multicore system, then they will see a large speedup in execution time in cycles. However, if they are running a single threaded program on a multicore system, the hardware overhead causes the multicore system to perform worse than the single core system. From this, we can conclude that the multicore system would be better in cases where the user is able to optimize their code to execute on multiple threads, but otherwise the single core system is the better option.

**Conclusion**

The multicore system adds a significant amount of hardware overhead due to the inclusion of multiple network systems. This means that the critical path is increased, so the cycle time increases. However, we felt that even though the multicore system adds area and energy to our design, the potential that it has for speedup makes it, in some cases, better than the single core system. Being able to have four programs run in parallel is sometimes able to mitigate the effects of the increased cycle time. Through the simulations, we were able to observe both the hardware overhead and the software overhead of the multicore system. We found that on average, there was a 22.4 percent increase in CPI when comparing the single thread on a single core system to single thread on a multicore system, and a 115.5 percent increase in the total number of cycles for execution in the multithreaded on multicore when compared to the multithreaded on the single core across the five benchmarks. When the user is able to optimize for a multicore system, the multicore system is able to provide a large amount of speedup. However, if this is not the case then the multicore system will perform worse when compared to the single core system.

**Work Distribution**

The majority of code was written during partner coding sessions so that we could discuss what we were implementing and catch each others' mistakes. We worked together to come up with a routing algorithm, which Elise then implemented. Katherine focused more on the hardware side of this lab, while Elise was able to handle the software side. We equally split up the Testing Strategy section based on what tests we wrote, and the Evaluation section by quantitative and quantitative analysis.